

University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

Theses, Dissertations, and Student Research
from Electrical & Computer Engineering

Electrical & Computer Engineering, Department
of

5-2020

Deep Learning and Polar Transformation to Achieve a Novel Adaptive Automatic Modulation Classification Framework

Pejman Ghasemzadeh

University of Nebraska - Lincoln, pejman.ghasemzadeh@huskers.unl.edu

Follow this and additional works at: <https://digitalcommons.unl.edu/elecengtheses>



Part of the [Computer Engineering Commons](#), and the [Other Electrical and Computer Engineering Commons](#)

Ghasemzadeh, Pejman, "Deep Learning and Polar Transformation to Achieve a Novel Adaptive Automatic Modulation Classification Framework" (2020). *Theses, Dissertations, and Student Research from Electrical & Computer Engineering*. 114.

<https://digitalcommons.unl.edu/elecengtheses/114>

This Article is brought to you for free and open access by the Electrical & Computer Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Theses, Dissertations, and Student Research from Electrical & Computer Engineering by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

DEEP LEARNING AND POLAR TRANSFORMATION TO ACHIEVE A NOVEL
ADAPTIVE AUTOMATIC MODULATION CLASSIFICATION FRAMEWORK

by

Pejman Ghasemzadeh

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Telecommunications Engineering

Under the Supervision of Professor Hamid R. Sharif-Kashani

Lincoln, Nebraska

May, 2020

DEEP LEARNING AND POLAR TRANSFORMATION TO ACHIEVE A NOVEL
ADAPTIVE AUTOMATIC MODULATION CLASSIFICATION FRAMEWORK

Pejman Ghasemzadeh, M.S.

University of Nebraska, 2020

Advisor: Hamid R. Sharif-Kashani

Automatic modulation classification (AMC) is an approach that can be leveraged to identify an observed signal's most likely employed modulation scheme without any *a priori* knowledge of the intercepted signal. Of the three primary approaches proposed in literature, which are likelihood-based, distribution test-based, and feature-based (FB), the latter is considered to be the most promising approach for real-world implementations due to its favorable computational complexity and classification accuracy. FB AMC is comprised of two stages: feature extraction and labeling. In this thesis, we enhance the FB approach in both stages. In the feature extraction stage, we propose a new architecture in which it first removes the bias issue for the estimator of fourth-order cumulants, then extracts polar-transformed information of the received *IQ* waveform's samples, and finally forms a unique dataset to be used in the labeling stage. The labeling stage utilizes a deep learning architecture. Furthermore, we propose a new approach to increasing the classification accuracy in low signal-to-noise ratio conditions by employing a deep belief network platform in addition to the spiking neural network platform to overcome computational complexity concerns associated with deep learning architecture. In the process of evaluating the contributions, we first study each individual FB AMC classifier to derive the respective upper and lower performance bounds. We then propose an adaptive framework that is built upon and developed around these findings. This framework aims to efficiently classify the received signal's modulation scheme by intelligently switching between these different FB classifiers to achieve an optimal balance between classification accuracy and computational complexity for any observed channel conditions derived from the main receiver's equalizer. This

framework also provides flexibility in deploying FB AMC classifiers in various environments. We conduct a performance analysis using this framework in which we employ the standard RadioML dataset to achieve a realistic evaluation. Numerical results indicate a notably higher classification accuracy by 16.02% on average when the deep belief network is employed, whereas the spiking neural network requires significantly less computational complexity by 34.31% to label the modulation scheme compared to the other platforms. Moreover, the analysis of employing framework exhibits higher efficiency versus employing an individual FB AMC classifier.

@ Copyright 2020, Pejman Ghasemzadeh

To my mother, Jinus.

Acknowledgments

First and foremost, I would like to wholeheartedly thank my precious mother and dear father for their unwavering support and belief in my dreams, although no amount of gratitude can ever be enough for them. I would not have accomplished any of my successes in life without their constant support, inspiration and encouragement.

I would specially like to thank my advisor, Prof. Hamid Sharif, for his invaluable and indispensable experience, guidance, support and inspiration during the course of my research work. I am also grateful to Dr. Hempel for his insights and directions in my graduate research.

Contents

List of Tables	xi
List of Figures	xi
List of Algorithms	xiii
List of Acronyms	xv
List of Symbols	xix
1 Introduction	1
1.1 AMC Applications	2
1.1.1 Civilian AMC Applications	2
1.1.2 Military AMC Applications	3
1.2 AMC Approaches	4
1.2.1 Likelihood-based AMC	4
1.2.2 Distribution Test-based AMC	7
1.2.3 Feature-based AMC	9
1.3 AMC Implementation	11
1.4 Summary of AMC Approaches	13
1.5 Thesis Organization	14
2 Problem Statement	15
2.1 Feature Extraction Stage	16

2.2	Classification Stage	21
2.2.1	Classification Accuracy	21
2.2.2	Computational complexity	22
3	Literature Review	24
4	Proposed Solution	31
4.1	Solutions Structure	31
4.1.1	Feature Extraction Stage	31
4.1.2	Labeling Stage	32
4.1.2.1	Classification Accuracy	32
4.1.2.2	Computational Complexity	33
4.1.3	The Proposed Novel Framework Structure	35
4.2	New Feature Extraction Stage Architecture	35
4.2.1	High-Order Statistical Feature Extraction Component	36
4.2.1.1	One-Pass Algorithm	38
4.2.1.2	Two-Pass Algorithm	38
4.2.2	Polar Coordinate Transformation	42
4.3	Proposed Deep Learning Structure for Labeling Stage	42
4.4	Deep Belief Network as Fully-Connected Network in Deep Learning Structure	46
4.4.1	Adaptive Moment Estimation	51
4.5	Spiking Neural Network as a Fully-Connected Network in a Deep Learning Structure	55
4.5.1	Threshold Unit Networks	55
4.5.2	Continuous Neural Networks	57
4.5.3	Spiking Neural Networks	58
4.5.4	Unsupervised Learning-Hebbian Learning	61
4.6	Proposed Novel Framework Structure	62

5	Results, Analysis and Discussion	65
5.1	RadioML2018.01A Dataset	65
5.2	DBN-based FB AMC Classifier Analysis	70
5.2.1	DBN Architecture and Employment	70
5.2.2	AMC Results and Discussion	72
5.2.2.1	Lower-bound Discussion	73
5.2.2.2	Upper-bound Discussion	74
5.2.2.3	Number of Training Samples Discussion	75
5.2.3	Computational Complexity Analysis	76
5.2.4	Model Conclusion	77
5.3	SNN-based FB AMC Classifier Analysis	77
5.3.1	Results and Discussion	80
5.3.1.1	Lower-bound performance	80
5.3.1.2	Upper-bound Performance	81
5.3.2	Computational Complexity Analysis	82
5.3.3	Model Conclusion	82
5.4	Proposed Novel Framework Analysis	83
6	Conclusion and Future Work	84
	Bibliography	86
A	Deep Learning Models Participating in Comparing Results	90
A.1	Deep CNN-based Model	90
A.2	RNN-based Model	91
B	Spiking Neural Network-Based Platform Utilized in This Research	93
B.1	Initialization and Refactored Conversion Module	93
B.2	Environment and its Initialization	98

B.3	Network	104
B.4	Pipeline	163
B.5	Encoding	177
B.6	Conversion	185
B.7	Model	198
B.8	Learning	210
B.9	Evaluation	232
B.10	Analysis	237

List of Tables

4.1	Values to be collected	63
5.1	Input data dimensions.	66
5.2	<i>3D high-order statistical polar-based</i> dataset dimensions.	70
5.3	Optimized hyperparameters and configurations of DBN.	70
5.4	Computational complexity of the proposed DBN-based model.	76
5.5	SNN hyperparameters' architecture.	78
5.6	Proposed SNN-based model computational complexity measurement.	82
5.7	Tradeoff-driven model selection for classification at each SNR.	83
A.1	CNN-based model structure dimensions.	90

List of Figures

1.1	Overview of an AMC operation in a communication system.	1
1.2	LB approach operation structure.	5
1.3	DT approach operation structure.	8
1.4	FB AMC overview structure.	11
1.5	Required components in transmitter to necessitate inclusion of AMC in the receiver.	12
2.1	Spectral-based features tree classification procedure.	17
4.1	Output spikes of neurons in third and fifth for the proposed SNN-based model.	34
4.2	Architecture of the new feature extraction stage.	35
4.3	RNN training process over m portion of cross-validated training set.	43
4.4	LSTM module in each RNN architecture training iteration.	44
4.5	Deep Learning architecture of the labeling stage.	46
4.6	An example RBM structure with 4 visible and 2 hidden units in their corresponding layers in which the effect of biases on visible and hidden layers units can be observed.	47
4.7	The proposed novel framework's principal working.	64
5.1	Process of applying channel effects to the transmitted signal	67
5.2	The destructive effect of the selective fading model over the constellation of 128QAM and 256QAM from RadiomL dataset at SNR = 5 dB.	68

5.3	Polar-based constellation of 128QAM and 256QAM from RadioML dataset at SNR = 5 dB with and without destructive effect of the selective fading model.	69
5.4	Validation accuracy of DBN-based model in training stage.	71
5.5	Training loss of DBN-based model in training stage.	72
5.6	Proposed DBN-based lower-bound performance compared to RNN and CNN	73
5.7	Proposed DBN-based upper-bound performance compared to RNN and CNN	74
5.8	Number of training sample impact on classification accuracy.	75
5.9	Proposed SNN-based model performance of number of training epochs versus validation accuracy.	79
5.10	Proposed SNN-based model performance of number of training epochs versus training loss.	79
5.11	Proposed SNN-based model lower-bound performance compared to proposed DBN-, RNN- and CNN-based models.	80
5.12	Proposed SNN-based model upper-bound performance compared to proposed DBN-, RNN- and CNN-based models.	81
A.1	The deep CNN-based classifier's structure.	91
A.2	The deep RNN-based classifier's structure.	92

List of Algorithms

- 1 Adam algorithm to optimize DBN hyperparameters. 54

List of Acronyms

AMC	Automatic modulation classification
IMD	Intelligent modem design
DSA	Dynamic spectrum access
SC	Spectrum congestion
EW	Electronic warfare
ES	Electronic support
EA	Electronic attack
EP	Electronic protect
LB	Likelihood-based
PDF	Probability density function
ML	Maximum likelihood
ALRT	Average likelihood ratio test
GLRT	Generalized likelihood ratio test
SNR	Signal-to-noise ratio
HLRT	Hybrid likelihood ratio test

DLRT	Discrete likelihood ratio test
MDLF	Minimum distance likelihood function
NPLF	Non-parametric likelihood function
DT	Distribution test-based
GoF	Goodness of fit
KS	Kolmogorov–Smirnov
CDF	Cumulative distribution function
OKS	One-sample Kolmogorov–Smirnov
TKS	Two-sample Kolmogorov–Smirnov
CVM	Cramer–Von Mises
AD	Anderson–Darling
FB	Feature-based
ML	Machine learning
SS	Signal spectral-based
PSD	Power spectral density
WT	Wavelet transform-based
HoS	High-order statistics-based
CA	Cyclostationary analysis-based
KNN	K-Nearest Neighbour
SVM	Support vector machine

DL	Dictionary learning
ANN/NN	Artificial neural network
DNN	Deep neural network
CNN	Convolutional neural network
RNN	Recurrent neural network
ResNN	Residual neural network
CR	Cognitive radio
SDR	Software defined radio
CSI	Channel state information
QoS	Quality of service
QoE	Quality of experience
<i>P_{CC}</i>	Probability of correct classification
M-ASK	M-amplitude shift keying
M-PSK	M-phase shift keying
M-FSK	M-frequency shift keying
M-QAM	M-quadrature amplitude modulation
CWT	Continuous wavelet transform
AWGN	Additive white Gaussian noise
LSTM	Long short-term memory
DBN	Deep belief network

RBM	Restricted boltzmann machine
SNN	Spiking neural network
FCN	Fully-connected network
ADAM	Adaptive moment estimation algorithm
SGD	Stochastic gradient descent
EPSP	Excitatory postsynaptic potential
IPSP	Inhibitory postsynaptic potential
MLP	Multilayer perceptrons
ReLU	rectified linear units
ODE	Ordinary differential equation
IF	Integrate-and-fire
LIF	Leaky-integrateand-fire
STDP	Spike-timing-dependent-plasticity
GP	Genetic programming

List of Symbols

γ_{max}	Normalized and centred maximum instantaneous amplitude value of the intercepted signal's spectral power density
σ_{iap}	Non-linear component absolute value's standard deviation of the instantaneous phase
σ_{ip}	Non-linear component direct value's standard deviation of the instantaneous phase
λ	Evaluation of the spectrum symmetry around the carrier frequency
σ_{ias}	Normalized and centered of absolute value of instantaneous amplitude of signal's symbols' standard deviation
σ_{if}	Normalized and centered of absolute value of instantaneous frequency's standard deviation
σ_{ia}	Normalized and centered instantaneous amplitude's standard deviation
K_{42}^a	Normalized and centered instantaneous amplitude's Kurtosis
K_{42}^f	Normalized and centered instantaneous frequency's Kurtosis
I	Real part of the received symbol
Q	Imaginary part of the received symbol

\mathbf{r}	Radius of the polar transformed symbol
θ	Angle of the polar transformed symbol
X	Random variable
$E\{X\}$	Expected value of random variable X
x_i	i^{th} sample of random variable X
\bar{X}	Mean of random variable X
\bar{m}_i	i^{th} partition's mean of X
L	Number of a random variable's samples
M_n^i	n^{th} order of central moment of i^{th} partition of X
$\Delta_{\mathcal{B}, \mathcal{A}}$	Mean difference of portions \mathcal{A} and \mathcal{B} of X
l_i	Length of i^{th} partition of X
$\sigma(\cdot)$	Sigmoid function
T	Time steps in LSTM layer
\mathbf{W}_α	Shared time-distributed NN weight matrix
ζ_α	Shared time-distributed NN bias matrix
α_t	t^{th} attention weight
y_t	t^{th} output of LSTM layers
τ	Timing unit
V_i	i^{th} neuron/node in visible layer
h_j	j^{th} neuron/node in hidden layer

\mathcal{E}	Energy function
\mathbf{V}	Vectors of units in visible layers
\mathbf{h}	Vectors of units in hidden layers
$\mathbf{Z}(\cdot)$	Partition function
a_i	Visible layers' biases
b_i	Hidden layers' biases
\mathcal{V}	Moving average
g	Gradient on current mini-batch
β_i	New introduced hyperparameter to ADAM algorithm
η	Step size
ρ_i	Binary inputs to neurons
λ	Predefined learning rate
C	Membrane capacitance
R	Membrane resistance
$I(t)$	Total input current to the neuron at time t
$v(t)$	Membrane potential
$I_{ext}(t)$	External current
S_i	Spike train
\overline{S}_i	Low-pass filtered versions of spike train

CHAPTER 1

Introduction

Automatic modulation classification (AMC) refers to a signal processing mechanism through which the intercepted signal's modulation scheme can be classified with minimal information on the signal's configurations. This process is exclusively operated at the receiver side of a communication, as illustrated in Fig 1.1 [1].

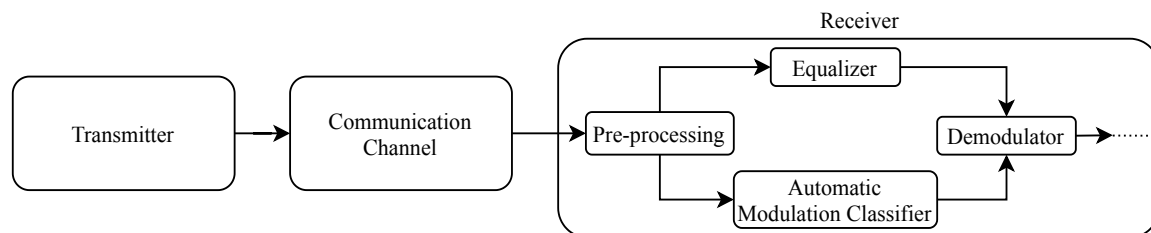


Figure 1.1: Overview of an AMC operation in a communication system.

The modulation scheme's information will then be used in the demodulator for further processing. The term “*automatic*” is used to oppose the initial implementations of fixed modulation classification procedures, where signals are modulated by electronic processors capable of operating one fixed modulation scheme. AMC essentially gained importance when link adjustment methods that use adaptive modulation and coding were introduced. These methods created an adaptive selection of modulation schemes in which a pool of multiple modulation schemes are employed by the system [2]. In this manner, the communication system further enabled an optimization process through which the transmission reliability

and data rate are investigated to lead to the adaptive selection of the modulation scheme according to communication channel conditions. While the transmitter has the freedom to choose the most reliable modulation scheme, the receiver must know the modulation scheme in order to demodulate the received signal so that the transmission can be successful. An easy way to inform the receiver about changes in the modulation scheme at the transmitter is to include the modulation scheme's information in each transmitted signal frame [3]. However, this solution affects the spectrum efficiency due to the extra information that is included in each signal frame. In the current era of wireless communication, where the wireless spectrum is extremely limited and valuable, this solution is not considered to be efficiently sufficient. For this reason, AMC is an attractive solution to the problem of notifying the receiver of the transmitted signal's modulation scheme.

Additionally, AMC also has other applications, which are briefly discussed below.

1.1 AMC Applications

AMC applications can be generally categorized into two groups: civilian applications and military applications.

1.1.1 Civilian AMC Applications

In this category, AMC mainly targets applications for intelligent modem designs (IMDs), spectrum sensing, safety monitoring in open or working areas, interference cancellation, dynamic spectrum access (DSA), link-adjustment to data rate and channel capacity, and signal protection [4]. In general, AMC's primary contribution to these applications can be summarized into three aspects. Below, we highlight these three aspects and describe them in more detail.

- **Signal Cancellation:**

With the near ubiquitous presence of wireless devices, we face a significant problem

of spectrum congestion (SC). At any given moment, a receiver faces the challenge of observing multiple radio signals, and has to filter out all but the intended transmitter's signal. The intended transmitter's signal may not be of favorable strength or quality, which results in the receiver cancelling out competing signals. One option to determine unfavorable signals is to deploy AMC at the receiver to find the signal's modulation scheme. After the modulation scheme is determined, the receiver then can filter out any signals that do not match the modulation schemes of the targeted transmitter [5].

- **Spectrum Surveillance:**

By deploying AMC at the receiver and then conducting a sweep of all supported frequencies, the receiver can then easily conduct a survey of mapping modulation schemes used at each particular frequency. This knowledge provides the primary tool to either eavesdrop or jam a signal in the area.

- **Removing Overhead in the receiver:**

In many communication systems, the transmitter changes the modulation scheme during the connection. This can be caused by any number of reasons, such as adjusting transmission parameters with channel rates. When the transmitter changes the modulation scheme, it typically notifies the receiver by sending information to the receiver. This overhead can be removed by deploying AMC in the receiver.

1.1.2 Military AMC Applications

AMC can assist with three tasks in electronic warfare (EW). These tasks are electronic support (ES), electronic attack (EA), and electronic protect (EP) [6]. The main duty in ES is to collect communication information on the battlefield, especially concerning hostile units. Frequency bands used by hostile units are examples of essential information. These frequency bands can be determined by utilizing AMC in friendly units to monitor modulation schemes and their corresponding frequency bands which are being employed

on the battlefield. This assists friendly units in differentiating among known (friendly) and unknown (hostile) modulation schemes and their frequency bands. As a result, friendly units can obtain two pieces of information from battlefield communications: frequency bands and modulation schemes used by hostile units. In EA, after capturing hostile communication information, jamming these hostile transmissions is a relatively easy task that can be accomplished by transmitting a signal with a higher power in the same frequency band to override the hostile transmission. In EP, if friendly communications are cut off by the same EA mechanism done by hostile units, the friendly frequency bands can be changed to those that are free and safe. The primary means by which these tasks can be accomplished is to gain information on the hostile communications' modulation schemes. Moreover, all the information gathered through this process can be leveraged to eavesdrop on hostile communications.

We will next briefly introduce proposed AMC approaches in literature.

1.2 AMC Approaches

AMC can fundamentally be organized into three primary widely discussed approaches in the literature: likelihood-based, distribution test-based, and feature-based. In the following subsections, we not only introduce these approaches, but also discuss their working principles.

1.2.1 Likelihood-based AMC

In the likelihood-based (LB) approach, it is believed that the probability density function (PDF) of the intercepted signal conditioned over an observed embedded modulated waveform consists of all required information for the modulation classification process. The LB classification process is generally accomplished in three following steps at the receiver:

1. Establishing a likelihood evaluation process in addition to an optimizing process for

threshold determination.

2. Performing likelihood evaluation between the calculated pool of modulation schemes' PDF and the observed signal's PDF for each observed received frame while updating the threshold through a predefined optimization process.
3. Determining which likelihood evaluation reaches the optimized threshold to make the final decision.

This operation can be seen in Fig 1.2.

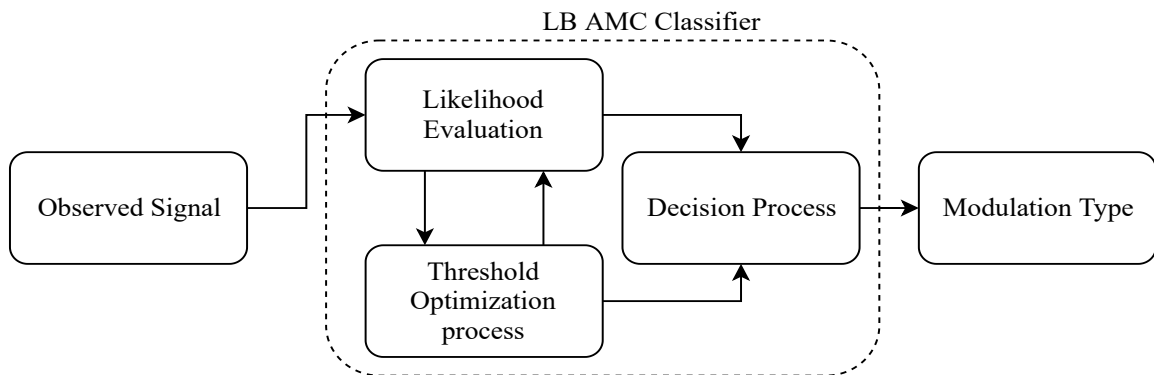


Figure 1.2: LB approach operation structure.

It is evident that all three steps are highly expensive computational processes for the receiver to simultaneously execute along with other processes at receiver such as equalizing. Furthermore, this procedure is also dependent on the channel's parameters since likelihood evaluation is conditioned over them. There is no way to exclude the effects of channel parameters since likelihood evaluation cannot handle any missing parameter. This also creates more computational complexity and requires the receiver to have some knowledge of the channel's parameters as conditional variables of PDF. Having perfect knowledge of channel parameters does not occur in real-world scenarios. Therefore, there is a theoretical classifier that uses a known channel's parameters, in addition to executing all aforementioned steps in a precise manner, called *Maximum likelihood (ML) classifier*. This classifier reaches the highest classification accuracy in comparison with all other classifiers. LB AMC was

then developed to make this approach more practical in real-world scenarios where there is no available information on channel parameters at the receiver, which resulted in proposing three other classifiers in this approach [7]. These classifiers attempt to estimate the channel's parameters along with executing AMC likelihood evaluation. They also ease the above three likelihood evaluation steps by calculating an approximation of likelihood evaluation, and performing a comparison scenario rather than optimizing the decision threshold to determine the intercepted signal's modulation scheme. Hence, the channel parameters' calculation mechanism, in addition to the degree of likelihood evaluation approximation and the comparison process, determines the classification accuracy and computational complexity of these three classifiers. The first of these classifiers is called *average likelihood ratio test (ALRT) classifier*. This classifier handles the channel's parameters that are unknown to it as random variables with certain PDFs that very well fit with the mathematical definition of the properties of the channel's parameters. Essentially, the most likely value of the channel parameters is computed by considering the integration of all possible values in the likelihood evaluation. The classifier consequently calculates the likelihood evaluation for each observed signal's PDF conditioned on channel parameters, while forming and updating a likelihood ratio test as part of the decision making process. This procedure, which forms the ALRT classifier, produces the highest computational complexity of all other non-theoretical classifiers. It attempts to precisely obtain the channel parameters in addition to the typical procedure of LB AMC, which is to calculate the likelihood evaluation for each observed waveform, and makes decisions based on an updated ratio. Despite this disadvantage, the classifier reaches the highest classification accuracy of all other non-theoretical classifiers. Consequently, ALRT can be called the optimal AMC classifier [8]. In order to ease computational complexity of this classifier, another alternative classifier was proposed called *generalized likelihood ratio test (GLRT) classifier*. This classifier handles a channel's parameters as unknown deterministic. Hence, in order to estimate the values of the channel's parameters, likelihood evaluation can be done over a specific

range of values, not over all possible values such as in ALRT. This approximation reduces the computational complexity of this classifier in addition to its classification accuracy, especially in lower signal-to-noise ratio (SNR) conditions where it is extremely difficult to determine the aforementioned range for nested modulation schemes. To solve this problem, the *hybrid likelihood ratio test (HLRT) classifier* was proposed, which is fundamentally built upon the signal carrier phase. Then this classifier acquires the values of the channel's parameters by performing discrete likelihood evaluation. In other words, this classifier finds the most likely values within a few number of candidates. This mechanism of approximation notably reduces computational complexity while slightly overcoming the lower SNR issue with GLRT. On the other hand, it makes the average classification accuracy of HLRT to be lower than both the ALRT and GLRT classifiers. In the sequence of reducing this approach's computational complexity, we can point to a few other classifiers such as *discrete likelihood ratio test (DLRT) and look-up table classifier*, *minimum distance likelihood function (MDLF) classifier* and *non-parametric likelihood function (NPLF) classifier*. These classifiers reduce the LB approach's computational complexity by applying various channel parameters' estimation methods as well as approximation of likelihood evaluation. These classifiers still have high enough computational complexity, which hinders their real-world implementation due to creating delays in further processes at the receiver. It can be concluded that this approach's classifiers obtain the highest classification accuracy on average, and have the highest computational complexity of all other approaches' classifiers due to the mathematical perspective of this approach. Therefore, another mathematical perspective is needed to overcome the computational complexity and cost issues.

1.2.2 Distribution Test-based AMC

The distribution test-based (DT) approach is built upon a mathematical definition called *goodness of fit (GoF)*, which in this domain represents the difference of two signals' distributions [9]. Thus, in order to build a classifier upon this definition, the calculated distribution

of the intercepted signal of adequate length, compared with the empirical one of different modulated signals, should be used in the GoF test. The modulation scheme that has the closest empirical distribution to the calculated one from the intercepted signal will be selected as the signal's modulation scheme. This process can be seen in Fig 1.3.

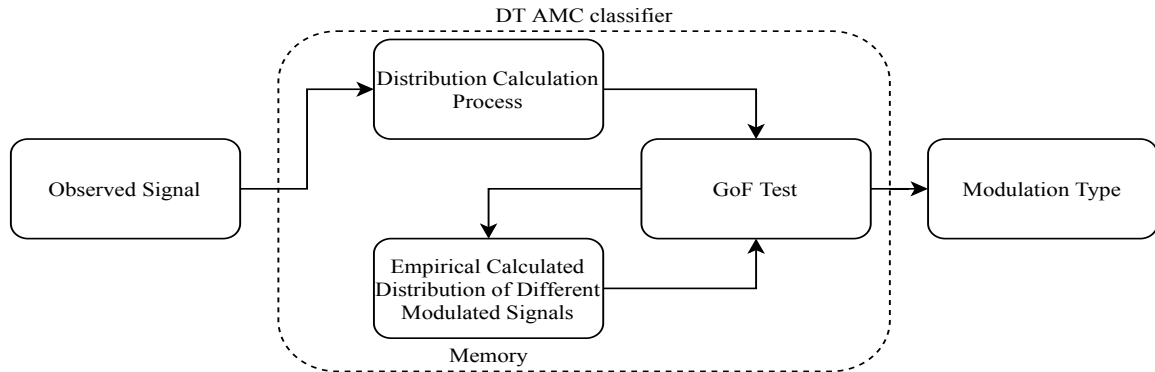


Figure 1.3: DT approach operation structure.

There are many proposed different distribution tests for GoF, but only a few are appropriate for our modulation classification purposes. The first one is the *Kolmogorov–Smirnov (KS) test*. In this test, the cumulative distribution function (CDF) is used for distribution calculations. This test was originally selected for AMC because of its notable lower computational complexity compared to the LB approach. Comparing these two approaches in this test creates two classifiers: the *One-sample KS (OKS) test classifier* and the *Two-sample KS (TKS) test classifier* [10]. These two classifiers have different implementable applications. TKS should be used when the channel is in a harsh condition, which results in the need for precisely reconstructing the CDF of an actual transmitted signal's CDF. In other words, TKS performs higher in lower SNR scenarios, and also has a higher computational complexity than OKS. Two alternatives to the KS test, which have their own comparison mechanisms between empirical CDF and a reconstructed CDF, are called the *Cramer–Von Mises (CVM) test* and the *Anderson–Darling (AD) test*. The difference between these two tests lies in their sensitivity to the changes in the tail of the signal's distributions. The AD test shows less sensitivity than CVM to sudden changes in the tail of the distribution. This makes AD

less computationally complex than both KS and CVM. Moreover, this also results in lower performance than both KS and CVM classifiers on average [11]. There are other tests that have attempted to do various models of approximation to ease the computational complexity issue to make this approach operational in real-time. In summary, the DT approach performs less accurately than the LB approach on average, especially in lower SNRs, while having less computational complexity as well. But this approach's performance highly depends on the surroundings where transceivers are deployed, because that determines the SNR condition. Additionally, the computational complexity of the DT approach can vary based on the selected test for the mechanism of GoF. Therefore, this approach cannot provide a global solution for AMC.

1.2.3 Feature-based AMC

The feature-based (FB) approach gained importance when machine learning (ML) algorithms became popular in classification applications. Machine learning algorithms are important in real-world applications when there are no patterns of changes in the data structures. This perspective can well connect with the disorderly changes caused by channel parameters over transmitted signals [12]. Thus, machine learning algorithms can assist with assessing and analyzing these negative effects and eventually lead to modulation classification. This approach involves two stages. In the first stage, an instantaneous signal's feature is extracted and then used in the second stage, which is also called labeling stage. The features that are utilized in the first stage relate mostly to a signal's characteristics. They can be categorized as follows:

- Signal spectral-based (SS) features, which also include:
 - The normalized and centered maximum instantaneous amplitude values of the intercepted signal's power spectral density (PSD) (γ_{max})
 - The non-linear component absolute value's standard deviation of the instanta-

- neous phase (σ_{iap})
 - The non-linear component direct value's standard deviation of the instantaneous phase (σ_{ip})
 - Evaluation of the spectrum symmetry around the carrier frequency (λ)
 - The normalized and centered absolute values of the instantaneous amplitude of the standard deviations of a signal's symbols. (σ_{ias})
 - The normalized and centered absolute values of instantaneous frequency's standard deviations (σ_{if})
 - The normalized and centered instantaneous amplitude's standard deviations (σ_{ia})
 - The normalized and centered instantaneous amplitude's Kurtosis (K_{42}^a)
 - The normalized and centered instantaneous frequency's Kurtosis (K_{42}^f)
- Wavelet transform-based (WT) features
- High-order statistics-based (HoS) features, which include:
 - Moment-based features
 - Cumulant-based features
- Cyclostationary analysis-based (CA) features

In the second stage, machine learning algorithms execute the labeling procedure [13].

Machine learning algorithms that have been utilized in literature are:

- K-nearest neighbor (KNN)
- Genetic programming (GP)
- Support vector machine (SVM)
- Dictionary learning (DL)

- Artificial neural network (ANN/NN), which also includes:
 - Deep neural network (DNN)
 - Convolutional neural network (CNN)
 - Recurrent neural network (RNN)
 - Residual neural network (ResNN)

An overview of this approach can be seen in Fig 1.4.

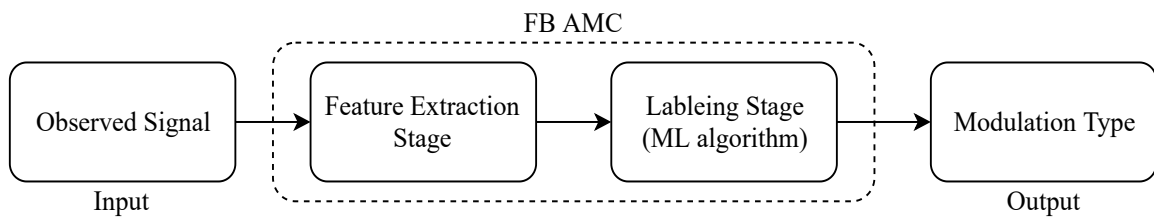


Figure 1.4: FB AMC overview structure.

As it can easily be observed, a combination of various feature extraction techniques and labeling procedures can create different AMC classification mechanisms with different levels of computational complexity and classification accuracy. One important factor in the computational complexity and cost of ML algorithms is the environment where the AMC classifier is intended to be deployed. The impact of this parameter and other elements will be further investigated in detail in Chapter 3 and 4. It can be concluded that this approach can be targeted for more real-world applications of AMC while its computational complexity and classification accuracy are related to the environment, feature extraction techniques and the labeling stage (ML algorithm) that form the FB AMC classifier; which are mostly under control of the AMC classifier designer.

1.3 AMC Implementation

The implementation of AMC was widely regarded as an impossible task before the introduction of cognitive radio (CR) and software-defined radio (SDR) technologies. Historically,

transceiver designs were very limited in the number of modulation schemes they could implement for transmission and reception for a variety of reasons, including hardware component and size limits, computation limitations, energy considerations, band limits, and more. Even today, with all the developments in modern technologies, it is still not possible to have a transmitter modulate a signal with all available types of modulation schemes. It only became feasible with the advent of CR and SDR [14]. CR transmitters are capable of sensing their environments and changing transmission parameters based on the obtained results from the environment. One of these parameters could be the employed signal's modulation scheme. SDR technology, on the other hand, enables transceivers to avoid being locked into fixed functionality sets. On the other hand, it makes the transmitter capable of employing virtually any type of modulation schemes simply through software upgrades. This entire process can be seen in Fig 1.5.

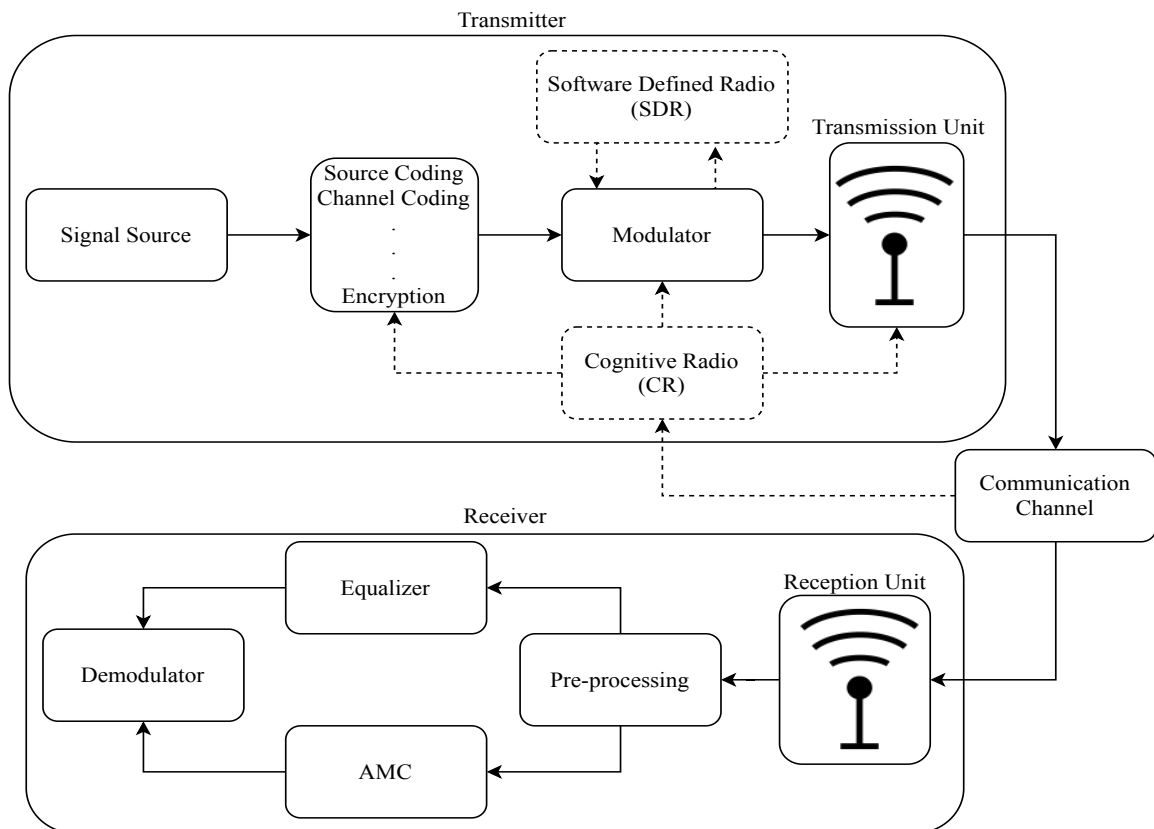


Figure 1.5: Required components in transmitter to necessitate inclusion of AMC in the receiver.

But CR and SDR only help in providing the necessary functionality for AMC. It does not mandate inclusion of AMC, as AMC inherently suffers from high computational complexity. This problem stems from the need for multi-dimensional computations to extract features and estimate a channel's unknown parameters, such as the signal and carrier frequency offset, signal and carrier phase offset, channel state information (CSI), and so on. And yet, the expectation is that AMC needs to operate in real-time with minimal latency. Once the signal is received and sampled, it is expected to immediately go through demodulating and decoding processes. Hence, AMC information is also needed virtually immediately in order to control the demodulation process. Latency stemming from computational complexity can cause delays in this process. Therefore, the continued presence of the excessive computational complexity prevents AMC from being employed in current transceiver systems [15]. However, the reduction in computational complexity through the use of low complex algorithms results in a significant degradation in classification accuracy. As a result, there is a trade-off between computational complexity and classification accuracy. Depending on which application AMC is being deployed, this trade-off needs to be resolved either in favor of lower latency or higher classification accuracy. It should be noted that for most military applications that consider AMC, real-time low-latency operation is vital. On the other hand, for civilian applications, a somewhat higher latency may still be acceptable, as long as it does not negatively impact either quality of service (QoS) or Quality of Experience (QoE). In order to further investigate AMC implementation, related works will be discussed and analyzed in this thesis.

1.4 Summary of AMC Approaches

After providing an introduction and background of different AMC approaches and reviewing various aspects of AMC, we can conclude that the theoretical aspects of AMC have reached a point that it can classify the intercepted signal's modulation scheme with the highest

potential probability of correct classification (P_{CC}) at a given SNR condition with the LB AMC approach. Even though the DT approach reduces the computational complexity of the LB approach, its performance is notably degraded in lower SNR conditions. The proportional degradation in classification accuracy over reduction of computational complexity is not considered efficient compared to what the FB approach can offer. On the other hand, in feature-based AMC, there is still research that needs to be done to improve its classification accuracy performance as well as decreasing its corresponding computational complexity to make it an efficient and flexible real-world implementable approach.

1.5 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents the problem statement of this thesis specially concerning FB AMC approach. Chapter 3 provides a literature review of prominent works in this domain. Chapter 4 presents the proposed solutions to the problem statement, which their corresponding findings finally form the novel framework. Chapter 5 presents the numerical results, analysis and discussion of the proposed solutions and the novel framework. And finally, Chapter 6 concludes the thesis.

CHAPTER 2

Problem Statement

In this thesis, we deeply focus on addressing the problem of increasing the classification accuracy for high-order modulation schemes in harsh channel conditions (low SNR values). Moreover, the computational complexity of the AMC architecture is also taken into consideration in order to make the AMC classifier operate simultaneously alongside other components in the receiver. Reducing computational complexity should be accomplished while not sacrificing classification accuracy as much as computational complexity is decreased. We focus on featuring the AMC classifier with flexibility, where it can freely switch between various architectures to obtain an efficient balance between computational complexity and classification accuracy in different environments.

To accomplish the above objectives, we select feature-based automatic modulation classification as the potential real-world implementable approach with advancements in computational power of next communication systems, especially the receiver. The proposed next generation of communication systems that involve machine learning algorithms for different tasks make this approach even more promising. In addition to these advantages, the feature-based approach also provides the following properties when designing the AMC classifier:

1. A combination of various methods in both stages of the FB AMC classifier that can create myriad methods with different levels of computational complexity and

classification accuracy can help the designer easily deal with particular applications, which require certain computational complexity and classification accuracy.

2. The designer has control over the FB AMC classifier's computational complexity and classification accuracy by designing a complex or non-complex architecture.
3. The FB AMC approach provides flexibility for various applications in different environments since computational complexity and classification accuracy of this approach can be designed by application's requirements and environment's condition.

As mentioned in the previous chapter, this approach consists of two stages. Each stage suffers from a number of problems that can eventually affect the performance of the FB AMC classifier. We investigate each stage's issues below.

2.1 Feature Extraction Stage

There are several methods that can be utilized to extract the intercepted signal's features. All of these methods have their own advantages and disadvantages, although they represent some information about a signal's modulation scheme. Our goal is to investigate which one is capable of providing more correlative in-depth information to finally help address our problem statement of increasing classification accuracy of high-order modulation schemes with low SNR values.

- **Signal Spectral-based Features:** Spectral-based features provide frequency-domain metrics on the intercepted signal. These metrics can target various aspects of the frequency domain of the intercepted signal. These aspects can include functions that use the zeroth and first power of the intercepted signal's samples, i.e., spectral power, absolute or direct instantaneous phase, absolute or normalized or centered instantaneous amplitude, normalized or centered instantaneous carrier frequency. The functions that use the second order of the intercepted signal's samples can include

features such as the normalized or centered amplitude's and frequency's Kurtosis. These features provide generic information about changes in density of the modulation parameters such as amplitude and phase density.

- **Advantages:** These features can be used to generally group the intercepted signal's modulation scheme into one of the M-amplitude shift keying (M-ASK), M-phase shift keying (M-PSK), M-frequency shift keying (M-FSK), or M-quadrature amplitude modulation (M-QAM) groups without specifying the order of modulation scheme (M). These features are also resilient against destructive environmental effects over transmitted signals. When this feature extraction technique is used, a simple tree classification can be sufficient for executing the labeling procedure. Hence, if no detailed information of the intercepted signal's modulation scheme is of interest, then this technique is considered to possess efficient computational complexity as well. The simplicity of this method, in addition to its labeling stage, can be seen in Fig 2.1.

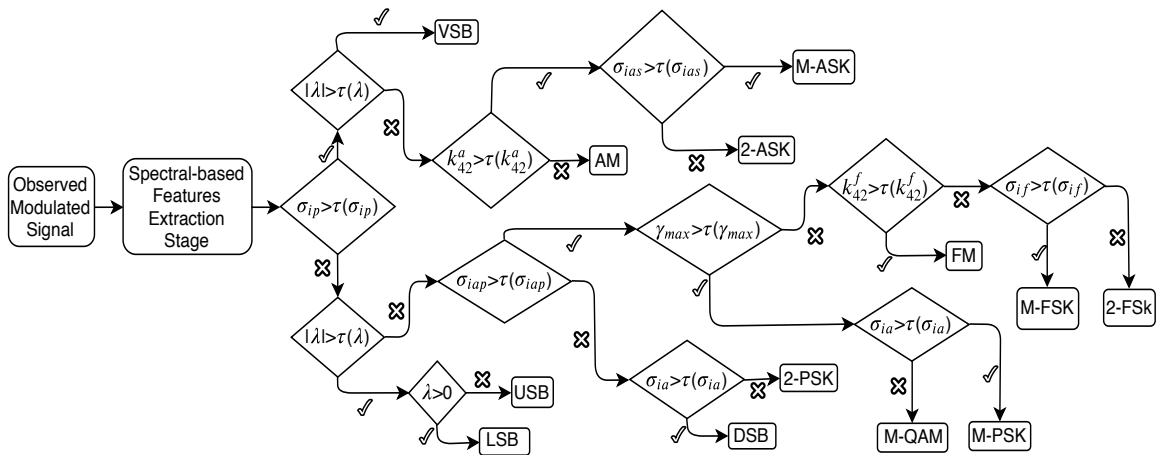


Figure 2.1: Spectral-based features tree classification procedure.

- **Disadvantages:** These features do not provide any in-depth information about determining the order of the intercepted signal's modulation scheme. Hence, they cannot be used to extract features from the intercepted signal when the environment is characterized with a high SNR value since in such environments,

high-order modulation schemes are deployed to transmit and receive more data. This problem is intensified nowadays when the communication protocols tend to increase the bandwidth of transmitted and received data.

- **Wavelet Transform-based Features:** These features form series that represent a square-integrable (real- or complex-valued) function by a certain orthonormal series generated by a wavelet. In other words, functions that use the third power of the intercepted signal's samples can form such features. In literature, three mother wavelet functions, namely Morlet, Haar and Shannon, are involved in calculating the continuous wavelet transform (CWT) to extract the intercepted signal's features.
 - **Advantages:** Each mother wavelet function is capable of providing a different level of correlation between received signal's symbols. This provides more in-depth information regarding symbols' patterns that can finally enable the labeling stage to also classify low-order modulation schemes in addition to the aforementioned general groups.
 - **Disadvantages:** Deploying the mother wavelet function, which provides more correlative information about received signal's symbols, can increase the computational complexity. This is because it requires computationally expensive procedures to extract the features and longer waveforms to enable feature extraction stage to form these relations. On the other hand, the classification accuracy of higher-order modulation schemes does not satisfactorily increase compared to the increase in computational complexity.
- **Cyclostationary Analysis-based Features:** These features represent statistical properties of the received signal that vary cyclically with time. In literature, there are two ways to treat these features: probabilistic approaches and deterministic approaches. A deterministic approach views the measurements of the intercepted signal as a single time series, from which a probability distribution for a sample associated with the

time series can be defined as the fraction of time. This entire process occurs over the samples' lifetime of the time series. In both approaches, the process or time series is considered to be cyclostationary if and only if its associated probability distributions vary periodically with time.

- **Advantages:** As this method measures the intercepted signal samples over time, it can provide precise information about amplitude and phase changes happening in signal over time if the sampling rate adheres to the Nyquist rate. Hence, enough in-depth information can be obtained from this method to enable the FB AMC classifier to also classify higher modulation schemes as well as lower-order ones.
- **Disadvantages:** This method's principal working element is time, and it also needs multiple waveforms to extract enough in-depth features of the intercepted signal. Hence, this method cannot operate in real time alongside other components in the receiver because the extracted features are also used in the labeling stage for training and classifying. Moreover, literature shows that as environmental conditions become harsh, this method significantly loses its precision in extracting features.
- **Higher-order Statistics-based Features:** Higher-order statistics-based features, which are also known as statistics moments, refer to utilizing functions with a third or higher power of the intercepted signal's sample. Whereas, conventional methods utilize low powers such as constant, linear and quadratic terms in their calculations. These calculations involve with zeroth, first and second powers, respectively. HoS features are used in estimation of shape parameters, which indicate the changing behavior of a sample.
 - **Advantages:** Estimating the properties of shape parameters can not only reveal the disorderly changes in a signal's symbols, but also provide enough in-depth

information on the correlation between the received signal's symbols. This can easily enable the labeling stage to achieve higher classification accuracy on higher-order modulation schemes. Moreover, these features follow a simple recursive mathematical procedure that does not impose any expensive computational complexity on the entire operation of the FB AMC classifier. Additionally, these features have a special property. That is, if the received signal is degraded by noise, which follows the additive white Gaussian noise (AWGN) distribution, calculating these features with any order greater than 2 will automatically result in cancelling out the effects of the AWGN noise in output. Hence, no matter how harsh the environment becomes, this method can compensate for AWGN degradation. This is one the reasons why they are widely used for AMC purposes.

- **Disadvantage)** The estimator function of a signal's moments is proven to be biased. In other words, the extracted features by a signal's moment estimator is not accurate. That's because the true value of the features being estimated is different than the estimator's expected value. Further, this provides an inaccurate or mirage correlation among received symbols. As the order of modulation scheme increases, this inaccuracy intensifies due to a decrease in spatial distance among symbols in the signal's constellation. We also should take into consideration the environmental effect on the received signal's constellation. It is clear that as the environmental conditions become harsh, the received signal's constellation's shape becomes even more disorderly, which makes the AMC task more difficult.

Among feature extraction methods, HoS features exhibit higher effectiveness in providing more in-depth correlative information about the received signal's constellation. Hence, HoS features are considered effective enough for implementation in the feature extraction stage. Although, HoS features hold many advantages, they provide inaccurate quantitative features in the labelling stage as the environmental conditions become more harsh, or the order of

the modulation scheme increases. To improve the performance of the FB AMC classifier in lower SNR conditions when high-order modulation schemes are classified, we should address the bias issue of the HoS method's estimator function in the feature extraction stage. We next explore the classification stage problem statement.

2.2 Classification Stage

As mentioned in the introduction, machine learning algorithms execute classification procedures in the labeling stage of FB AMC classifiers. Each machine learning algorithm has various factors measuring its performance. Two of these factors that are critical in the AMC domain are classification accuracy and computational complexity. These two factors affect a FB AMC classifier's performance based on the provided features of the received signal. We next investigate the problem statement for each of these factors.

2.2.1 Classification Accuracy

Classification accuracy measures the precision of a machine learning algorithm in the classification of an intercepted signal's modulation scheme. In other words, to specifically explain this factor in the AMC domain, it corresponds to a fraction whose numerator is the number of waveforms with correctly classified modulation schemes, and its denominator represents all analyzed waveforms.

The environment in which the FB AMC classifier is deployed is one of the major elements that can affect the FB AMC classifier's classification accuracy. As environmental conditions degrade in terms of scattering elements, e.g., densely developed urban environments representing a multipath propagation case, it is more difficult for an AMC classifier to classify the intercepted signal's modulation scheme with a high degree of accuracy. This mostly happens to nested modulation schemes with higher orders such as 128-QAM and 256-QAM. Therefore, there should be a mechanism that can resolve this issue in lower SNR

values. This finds its application in situations where classification accuracy is of crucial importance, such as military applications.

2.2.2 Computational complexity

A machine learning algorithm's computational complexity refers to the required time for executing training, validation, and classification steps. Among these steps, training requires the longest time, since it needs to find hidden patterns in data changes. A classification step, on the other hand, can be done in a short amount of time. There are two methods to measure computational complexity of a machine learning algorithm.

1. Theoretically, where it involves the process of Big O notation to describe the limiting behavior of a machine learning algorithm when its running time tends towards a particular value or infinity in the defined space of an experiment.
2. Experimentally, where the specifications of a platform in which the machine learning algorithm runs are given, and then the required time to do training, validation, and classification are normalized based on the platform's timing unit.

These methods are of importance in the AMC domain because we need to simulate the theoretical performance of the FB AMC classifier while considering its real-world implementation. Implementing a real-time FB AMC classifier in receivers has been historically considered to be an impossible task, since it inherently suffers from high computational complexity. This problem stems from the need for multi-dimensional computations to estimate unknown pattern changes in the intercepted signal to finally extract information of sufficient fidelity regarding the signal's modulation scheme. However, FB AMC must operate in real time while imposing minimal latency on other receiver components. Hence, FB AMC information is needed virtually immediately in order to control the demodulation process. Latency stemming from computational complexity can cause delays in this process.

Therefore, the continued presence of this excessive computational complexity prevents AMC from being implemented in most current receivers.

In a very concise comparison between the computational complexity of these steps and the feature extraction stage, we can easily observe that the computational complexity of the labeling stage is much higher than the feature extraction stage. Hence, we mainly focus on reducing the labelling stage's computational complexity to provide for the possibility that the FB AMC classifier can operate real time. However, when employing low computational complexity algorithms to achieve a reduction in computational complexity, the achievable results show a significant degradation in classification accuracy. As a result, there is a trade-off between computational complexity and classification accuracy. Depending on which application FB AMC is being utilized for, this trade-off needs to be resolved either in favor of lower latency or higher classification accuracy. For most military applications that consider AMC deployment, real-time operation is vital. On the other hand, for civilian applications, a somewhat higher latency may still be acceptable as long as it does not negatively impact quality of service and experience.

CHAPTER 3

Literature Review

In this chapter, we briefly investigate some of the prominent works in the FB AMC domain to see how they have attempted to address the aforementioned problems.

Yu and *Miao* in [16] proposed a deep learning-based method combining two CNNs with different structures trained on different datasets with their samples composed of in-phase and quadrature component signals, otherwise known as in-phase and quadrature samples, to distinguish modulation modes. They also adopted a dropout instead of a pooling operation. Their entire system design is based on constellation diagrams for 16QAM and 64QAM modulation schemes. Combining two machine learning platforms with different learning structures creates a powerful FB AMC classifier capable of learning more hidden patterns of the received signal's constellation shape when, in particular, the focus is on classifying M-QAM modulation schemes. This easily increases the classification accuracy for all environmental conditions. Even though this work has increased classification accuracy for constellation-based modulation schemes, their FB AMC classifier is not capable of classifying M-FSK and M-PSK modulation groups as well as M-QAM. Moreover, combining two NN platforms notably increases computational complexity. This makes their FB AMC classifier difficult to implement in the real world.

Fan and *Peng* in [17] proposed a CNN-based deep learning structure where the labeling stage is trained in two steps. After the first training step is done, the learned characteristics

are transferred to the second step of training. Through this two-step training structure, not only are features from the long symbol-rate observation sequence extracted, but the environmental condition is also estimated. Their classifier can also dimensionally accommodate varying inputs. Two-step training can increase the robustness of the labeling stage in the presence of carrier phase offset under most environmental conditions. This can also provide independence for the FB AMC classifier from receiver's main equalizer in providing the SNR value. Moreover, their deep learning structure is flexible over the input data's dimension, which makes the FB AMC classifier capable of operating on different waveform lengths. Two-step training significantly increase the computational complexity of the labeling stage, which makes the FB AMC classifier incapable of operating the AMC task real-time. Additionally, having a CNN-based platform does not create a robust FB AMC classifier for the M-FSK and M-PSK groups of modulation schemes.

Chieh-Fang and *Ching-chun* in [18] proposed a CNN-based deep learning platform in which a mechanism to estimate channel state information is created. In such a mechanism, the FB AMC classifier is enabled to compensate for the destructive channel's effect on the received signal. CNN-based platform dimensions were increased to a two-dimensional platform, which is capable of being trained not only over $I - Q$, but also over $r - \theta$ samples. It is obvious that equalizing the channel's effect over signal, and reconstructing the estimated transmitted signal, will significantly increase classification accuracy. Additionally, providing extra information ($r - \theta$ samples) to the labeling stage will lead to a more accurate training process of the CNN-based platform, which can eventually increase the classification accuracy. Having the mechanism to estimate channel state information is inherently a computationally expensive task since it requires multiple iterations to obtain channel parameters. Hence, adding such mechanism to a deep learning platform will result in a significant increase in computational complexity of a proposed FB AMC classifier. Moreover, providing another dataset to the deep learning structure in order to increase classification accuracy will result in increasing the computational complexity of the training stage.

Zhe and *Yong* in [19] proposed a novel pre-processing stage that eliminates the effect of a multipath channel coefficient over the intercepted signal. They utilized an estimation mechanism to achieve channel state information parameters. They furthermore utilized a logarithmic functional fitting method to classify received modulated signals. Their proposed FB AMC classifier increases the classification accuracy of low-order modulation schemes under harsh environmental conditions. Utilizing the logarithmic functional fitting method can decrease the computational complexity of the labeling stage due to its simple classification mechanism. The authors attempted to compensate for the increase in computational complexity of their proposed FB AMC classifier due to utilizing the channel state information estimation method by selecting a simple labeling mechanism. Adding another stage such as pre-processing to a typical FB AMC classifier's components will increase its total computational complexity, which hinders its practical implementation. Moreover, estimating channel state information when receiving a high-order modulated signal is a very computationally expensive task, since it requires several iterations of the modulated signal's waveforms. Deploying a simple mechanism in the labeling stage can decrease the accuracy of detecting hidden patterns in data, which can finally result in lower classification accuracy for high-order modulation schemes.

Shengliang and *Hanyu* in [20] mainly focused on decreasing the labeling stage's computational complexity. To accomplish this, they employed two CNN platforms built by Google called AlexNet and GoogLeNet, which are capable of parallel computation over various partitioned data. It is obvious that parallel computation, especially in the training stage, will significantly decrease the required time to execute each step of the labeling stage. This idea can be leveraged for real-world implementation where faster classification of an intercepted signal's modulation scheme is of importance. Partitioning the data, received samples of an intercepted signal, can result in losing the correlation between partitioned data considering the fact that each partitioned data will separately be processed by the ML platforms. This can finally lead to a less accurate training process of the ML algorithm

that can negatively impact classification accuracy, especially with high-order modulation schemes. Additionally, since AlexNet and GoogLeNet are CNN-based, they are not capable of exhibiting high performance when M-FSK and M-PSK modulation schemes groups participate in classification.

Sudhan and Rahul in [21] proposed a new architecture of a feature extraction stage that combines two feature extraction methods, namely elementary cumulants and cyclic cumulants. This method can easily detect if the intercepted signal's modulation scheme is within a real, circular or rectangular class (group of the modulation scheme). Moreover, they used cyclic cumulants that describe positions of non-zero cyclic frequencies to classify the order of the modulation scheme. Utilizing two feature extraction methods in the feature extraction stage can not only provide more in-depth information to the labeling stage, leading to a more accurate training process, but also can enable the FB AMC classifier to blindly make a final decision without knowing the channel state information. Utilizing non-zero cyclic frequency features can be beneficial in increasing the classification accuracy of M-FSK modulation schemes. Having two feature extraction methods increases the computational complexity of the FB AMC classifier. Although more in-depth information is provided to the labeling stage by extracting two different features from the intercepted signal, their proposed classifier can robustly classify the general group of modulation schemes in addition to low-order ones. In order for high-order modulation schemes to be robustly classified, more correlative information that cannot be provided by elementary cumulants and cyclic cumulants is needed.

Sreeraj and Wannas in [22] attempted to enhance the training process of the labeling stage by adding a long short-term memory (LSTM) layer to conventional deep learning structures. In the training process, LSTM layers are characterized by executing several iterations over the data to lead the LSTM layer to memorize the features of the intercepted signal. This property will become useful later to balance the links weights of NNs. Memorizing extracted features of intercepted signal and helping to balance the links weights of NN

will result in increasing classification accuracy regardless of the group of an intercepted signal's modulation scheme. Utilizing the LSTM layer also provides more robustness to the classification process in various environments. All these advantages together can make this FB AMC classifier more suitable for those applications where classification accuracy is of crucial importance. LSTM layers are considered computationally expensive due to several iterations performed to memorize features. Therefore, the FB AMC classifier in which LSTM layers are utilized is not recommended for selection for real-time practical applications.

Yahia and Octavia in [23] conducted an experiment to classify phase shift-keying modulated signals based on the graph representation of the Fourier transform of the second and fourth powers of these signals. This experiment shows the capability of classifying low-order of phase shift-keying signals with high accuracy. For high-order of phase shift-keying signals, the experiment is not as successful as for low-order signal classification. Graph representation of the Fourier transform of the second and fourth powers of the intercepted signal does not provide in-depth enough information for high-order modulation schemes to be robustly classified.

Muhammad and Zhechen in [24] proposed a FB AMC classifier where this classifier combines genetic programming and K-nearest neighbor in labeling stage. In this work, K-nearest neighbor has been used to evaluate the fitness of GP individuals during the training step. Additionally, in the testing step, K-nearest neighbor has been used for deducing the classification performance of the best individual produced by GP. The classification step has been divided into two phases for improving the classification accuracy. In feature extraction stage, cumulants have been used as input feature for GP. They tested their classifier's performance with four modulation schemes: BPSK, QPSK, 16QAM and 64QAM. Utilizing two machine learning algorithms in labeling stage can increase the classification accuracy. This increase in classification accuracy is built upon the fact that K-nearest neighbor oversees the performance of GP to deduce the classification performance of the best individual

produced by GP. This feedback operation helps classifier to more deeply find hidden patterns in symbol changes. On the other hand, this entire procedure is highly computationally expensive because the feedback operation in this classifier is built on top of K-nearest neighbor algorithm, which itself is based on several nested loops. Hence, although the classification accuracy is increased, computational complexity of this classifier is considered to be much higher than other ones on this domain.

Lei and Hong in [25] proposed a classifier where it uses a distributed AMC scheme based on compressive sensing by taking advantage of the sparse property of cyclic feature mapping. Thus, they introduced a novel method based on compressive sensing principle for capturing the prominent peaks of the feature mapping. This method is capable to acceptably perform AMC task at sub-Nyquist rate of sampling. Additionally, they proposed a novel neural network fusion strategy for better cooperation with compressive sensing principle. Using a classifier that can operate at sub-Nyquist rate can be extremely helpful for situations where there is no knowledge of transmitted signal such as in battlefields. Moreover, since a compressive sensing method is used, the training step of neural network in labeling stage is considered to be accomplished in shorter time, which implies the decrease in computational complexity. This work has shown that their classifier strongly performs with low-order modulation schemes. On the other hand, using a compressive sensing method in addition to operating at sub-Nyquist sampling rate can decrease the probability of correct classification for higher-order modulation schemes.

Octavia and Ali in [26] proposed a classifier where it employs higher-order cyclic cumulants to discriminate linear or low-order digital modulation schemes under various channel conditions. In order to more deeply investigate the performance of this classifier, they not only test its performance in single-antenna mode, but they also consider a multiple-antenna case to assess the effect of spatial diversity. Additionally, they derived analytical closed-form expressions for the cyclic cumulant polyspectra of linearly digitally modulated signals affected by fading, carrier frequency and timing offsets, and additive Gaussian noise.

Their proposed classifier significantly increase the classification accuracy for low-order modulation schemes especially in multiple-antenna scenario due to taking the advantage of spatial diversity to eliminate the fading effect over received signal. On the other hand, their classifier is also capable to address the problem of increasing the classification accuracy for high-order modulation schemes in low SNR conditions while its computational complexity is notably increases. This increase is due to the fact that higher-order cyclic cumulants requires several waveforms to be able to establish the relationships between received signal's moment and cumulants. It also should be noted that the increase in computational complexity is much more than the increase in average classification accuracy for higher-order modulation schemes.

After reviewing the literature, we can conclude:

1. No study, to the best of our knowledge, has considered the impact of the estimator's bias in the feature extraction stage when a signal's moment is to be extracted.
2. Literature contributions that aim to increase classification accuracy have thus far not achieved acceptable performance at lower SNR values for high-order modulation schemes.
3. Computational complexity of deep learning architecture in the labeling stage has not been effectively reduced up to a point of operating real-time.
4. Thus, to date, no efficient and adaptive framework has been presented in literature to provide flexibility in controlling computational complexity and classification accuracy.

The problems statement and our findings from reviewing the scientific literature on the topic of AMC have led us to conduct research that aims to address these challenges. Our contributions and the resulting novel framework structure are presented in the next chapter.

CHAPTER 4

Proposed Solution

This thesis aims to address the aforementioned prominent problem statements in both stages of the FB AMC classifier. Providing an overview of contributions in below can strongly help the reader understand our procedure to address these problems.

4.1 Solutions Structure

The following subsections will provide an overview of proposed solutions to solve the stated problems of both forming stages of the FB AMC classifier.

4.1.1 Feature Extraction Stage

Feature extraction stage procedures in literature have not been promising in providing in-depth enough correlative information of received signal to labeling stage, specially for high-order modulation schemes. Therefore, we propose a new architecture for the feature extraction stage. This architecture is comprised of two components. The first component extracts the fourth-order cumulants from the received signal's $I - Q$ symbols. For this process, we also address the problem of biased estimators for fourth-order cumulants for two different cases: when the received signal's symbols are 1) real values, and 2) complex values. The second component extracts polar coordinates $r - \theta$ from the received $I - Q$ symbols

to provide more in-depth information to the labeling stage of a signal's constellation. This solution will be explored in detail in this chapter in section 4.2.

4.1.2 Labeling Stage

Contrary to other efforts in the literature that aim to modify conventional machine learning algorithms or deep learning structures that were proposed for AMC, we instead introduce the idea of using two entirely different machine learning algorithms in a deep learning structure for AMC. These algorithms will specifically address the stated problems traditionally associated with this stage that can be generally categorized as classification accuracy and computational complexity.

4.1.2.1 Classification Accuracy

We introduce a deep belief network (DBN) platform to be utilized in AMC for the first time, to the best of our knowledge. There are two motivations to employ DBN in AMC that can eventually improve the performance of the FB AMC classifier under low SNR conditions.

1. In any ANN platform, there is a problem called vanishing gradient that spreads throughout the network and imbalances the link weights as the training cycles are executed. This results in an inaccurate training process, and furthermore increases the misclassification error. The proposed solution is to use a gradient-based learning method combined with back-propagation. This solution involves a Restricted Boltzmann Machine (RBM) which automatically finds hidden patterns in the data by reconstructing the input. This property enables DBN to not only be trained like conventional NNs, but to tag the important portion of data with higher probability. A DBN is created by stacking RBM layers. Hence, vanishing gradients are removed in DBNs. This leads to a more accurate training process of DBN.
2. A DBN benefits from reconstructing the input in a back-propagation loop. This

finally results in a high capability of learning how to probabilistically reconstruct the input. This allows DBNs to recognize the influential portion of the input with high probability. Then by focusing the training process onto this portion of the input it can be more accurately trained.

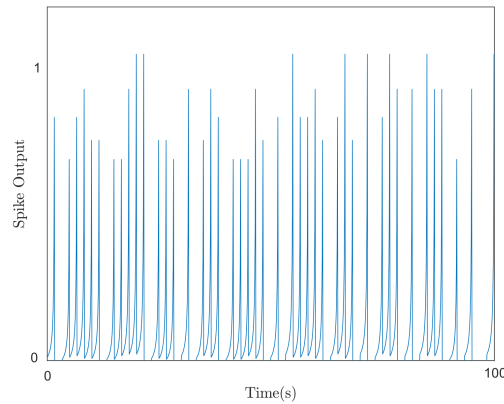
These two properties allow a DBN to be capable of finding deeper hidden patterns in the input data while removing the vanishing gradient problem. Hence, this platform exhibits superior capabilities in classifying high-order modulation schemes in lower SNR conditions. This solution will be explored in detail in this chapter in section 4.4.

4.1.2.2 Computational Complexity

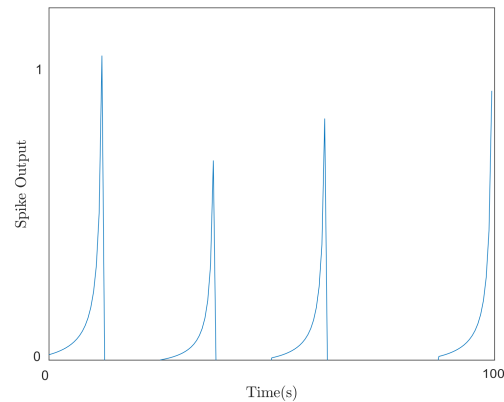
We also introduce the use of a spiking neural network (SNN) platform for AMC. Spiking neural networks are a close mathematical approximation of natural neurons' operations. In their operations, not only the neuron's state are applied, but time is also incorporated into the synaptic property of the neurons. This leads to the main motivation to introduce the use of this type of neural network platform in AMC: Neurons in SNNs fire only when they have reached a specific value by accumulating the spikes' values from neurons in former layer. This results in two important SNNs' characteristics.

1. All neurons forming a layer do not participate in classification operation. This is a direct consequence of SNN's property where not all neurons fire at each propagation cycle, but rather fire only when a membrane potential – an intrinsic quality of the neuron related to its membrane electrical charge – reaches a specific value. This eliminates notable required computations in each layer. Therefore, in layer-wise comparison with other platforms introduced in AMC, SNNs are characterized by lower computational cost.
2. The aforementioned process of neurons not participating in classification and eliminating computations of each layer intensifies as the data moves forward in a network's

layers. In other words, let's assume that the fifth layer in an SNN requires significantly less computations than the third layer in the same network due to the notably lower produced number of spikes in the fifth layer. The computational cost of an SNN corresponds to the neurons' involvement in producing spikes. Therefore, not only is the computational cost decreased layer-wise, but the entire network also requires notably less computational cost. This can be better understood with the intuition provided in Fig 4.1 for the proposed SNN-based model that will be explored in detail in chapter 4.



(a) Output spikes of 7th neuron in third layer.



(b) Output spikes of 12th neuron in fifth layer.

Figure 4.1: Output spikes of neurons in third and fifth for the proposed SNN-based model.

Overall, this platform, by its inherent design, requires less computational cost in all steps

of training, validation and classification. As a result, this FB AMC classifier will have a high likelihood of achieving real-time operation for most AMC applications in classification step. This solution will be explored in detail in this chapter in section 4.5.

4.1.3 The Proposed Novel Framework Structure

We propose an adaptive framework that efficiently switches between the two aforementioned labeling platforms based on each platform's specific characteristics, i.e., computational complexity and classification accuracy. In other words, this framework attempts to automatically adapt between classification accuracy and computational complexity for any derived SNR from the main receiver's equalizer. In this way, this framework can be flexible in implementation in different environments. We describe the principal functionality of this novel framework and investigate it in detail in section 4.6.

4.2 New Feature Extraction Stage Architecture

In this section, we present our design of a new architecture for this stage, which takes in the received signal and extracts from it a stream of feature descriptors that are then used in the labeling stage. This new architecture is comprised of two components that operate in parallel. The output of these components will be appended to the original received signal to create an augmented signal data stream. This architecture can be seen in Fig 4.2.

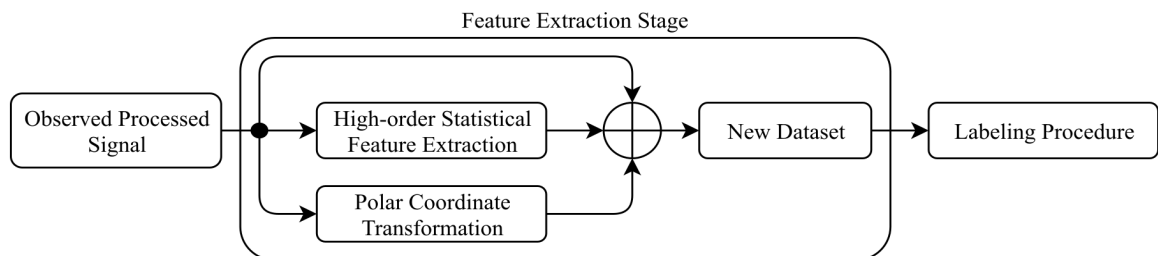


Figure 4.2: Architecture of the new feature extraction stage.

The resulting *3D high-order statistical polar-based* dataset is built upon the idea that

the more in-depth correlative information that can be provided to the labeling stage, the more precisely the training and validating steps of labeling procedure will be executed. Consequently, the system will be able to achieve a higher classification accuracy. Adding the polar coordinate transform component does not increase the computational complexity of the feature extraction stage since 1) it does not require any recursive computation compared to the high-order statistical feature extraction component, and 2) the polar coordinate transform component operates in parallel with the high-order statistical feature extraction component. It should be mentioned that due to the high-order statistical feature extraction's recursive nature, its computational complexity is longer than the polar coordinate transform. These components will be explored in more detail in the next subsections.

4.2.1 High-Order Statistical Feature Extraction Component

High-order statistics refers to applying functions with third-order or higher powers over sample data. Cumulants are one of these functions used in the literature for AMC. The definition of cumulants is simply the formal relation between the coefficients in the Taylor expansion of function M_n with $\bar{m} = 1$, and the coefficients in the Taylor expansion of $\log M_n$. They have beneficial properties such as their symmetric and additive operation over input arguments and their homogeneous behavior over partitions. The most important property of cumulants is that if the input arguments follow a Gaussian distribution, then their cumulants of any order higher than two equate to zero. By taking advantage of this property, the negative effect of noise that is typically observed with Gaussian distributions will be automatically eliminated from the labeling stage. Moreover, these useful properties enable cumulants to estimate shape parameters, which indicate the changing behavior of sample data. This helps significantly in producing enough in-depth correlative information about the high-order modulation schemes. However, the estimator for any order of cumulants greater than two has been proven to be biased [27], which produces inaccurate values that can, in turn, result in the labeling stage producing misclassified results, especially at lower

SNRs for high-order modulation schemes such as 128QAM and 256QAM. Therefore, in our framework, we address the problem of removing the bias of fourth-order cumulants for the scenarios when the received signal's symbols are 1) real values for low-order modulation schemes, and 2) complex values for high-order modulation schemes. For a random variable of X , M_n is defined as:

$$M_n \triangleq E\{(X - E[X])^n\} \quad (4.1)$$

For finite and equiprobable samples $x_i \in X$, M_n can be written as:

$$M_n = \frac{1}{L} \sum_{i=1}^L (x_i - \bar{m})^n, \quad (4.2)$$

where

$$\bar{m} = \frac{1}{L} \sum_{i=1}^L x_i \quad (4.3)$$

If we do not consider the estimator's bias issue, referring to the difference between the estimator's expected value and the true value of the parameter being estimated, then Equation (4.2) can provide cumulants of n^{th} order. However, the clear advantage to addressing the bias issue is to provide accurate correlative quantitative features to the labeling stage. The more accurate these quantitative features are, the more accurately the training, validation and classification steps of the labeling stage are executed. Hence, we can accomplish it using two algorithms called *one-pass* and *two-pass* for the case where the received symbols are real-valued.

4.2.1.1 One-Pass Algorithm

Through using the binomial theorem and expanding the term of $(x_i - \bar{m})^n$ to explicit powers of x_i and \bar{m} in binomial theorem, Equation (4.2) can thus be rewritten as:

$$M_n = \sum_{k=0}^n \binom{n}{k} \left(\frac{1}{L} \sum_{i=1}^L x_i^{n-k} \right) (-\bar{m})^k \quad (4.4)$$

This algorithm thus attempts to remove any estimation bias by considering the probability of whether estimation bias has happened.

4.2.1.2 Two-Pass Algorithm

As an alternative to the one-pass algorithm's approach, the *two-pass* algorithm attempts to solve this issue statistically, by using the following statistical procedure. It first divides the received signal symbols into two partitions, \mathcal{A} and \mathcal{B} ; where $l_{\mathcal{A}}$ and $l_{\mathcal{B}}$ are respectively the length of partitions \mathcal{A} and \mathcal{B} . Moreover, $\bar{m}_{\mathcal{A}}$ and $\bar{m}_{\mathcal{B}}$ represent the mean of each partition. After that, Equation (4.2) can be rewritten as shown in Equation (4.4) for any order equal or greater than 2. In Equation (4.4), $M_n^{\mathcal{A}}$ and $M_n^{\mathcal{B}}$ are moments of order n over each \mathcal{A} and \mathcal{B} portions. $\Delta_{\mathcal{B}\mathcal{A}}$ also equates with $\bar{m}_{\mathcal{B}} - \bar{m}_{\mathcal{A}}$, where each term stands for the mean of their corresponding signal's portion.

$$M_n = M_n^{\mathcal{A}} + M_n^{\mathcal{B}} + l_{\mathcal{A}} \left(\frac{-l_{\mathcal{B}} \Delta_{\mathcal{B}\mathcal{A}}}{L} \right)^n + l_{\mathcal{B}} \left(\frac{l_{\mathcal{A}} \Delta_{\mathcal{B}\mathcal{A}}}{L} \right)^n + \sum_{k=1}^{n-2} \binom{n}{k} \Delta_{\mathcal{B}\mathcal{A}}^k \left[M_{n-k}^{\mathcal{A}} \left(\frac{-l_{\mathcal{B}}}{L} \right)^k + M_{n-k}^{\mathcal{B}} \left(\frac{l_{\mathcal{A}}}{L} \right)^k \right] \quad (4.5)$$

The *Two-Pass* algorithm removes the estimator's bias more accurately compared to the *One-Pass* algorithm. On the other hand, this algorithm also requires more computational resources. After addressing the discrete moment estimator's bias issue, the cumulants of the

received signal can be recursively calculated from the discrete moment of the signal as:

$$C_n = M_n - \sum_{i=1}^{n-1} \frac{(n-1)!}{(i-1)!(n-i)!} C_i M_{n-i} \quad (4.6)$$

If the received signal has complex-valued symbols, then joint cumulants of the symbols need to be calculated. This applies to modulated signals with high-order modulation schemes. We derive the fourth-order cumulants over the intercepted signal (X) in Equation (4.7).

$$C_4(X) = \frac{L^2}{L^3 - 6L^2 + 13L - 12} [(L+1)\overline{X^4} - 4(L+1)\overline{X^3} \overline{X} - 3(L-1)\overline{X^2} \overline{X^2} + 12L(\overline{X^2} \overline{X} \overline{X}) - 6L\overline{X^4}] \quad (4.7)$$

The derivation process is presented below.

The generic format of an estimator for n^{th} -order cumulants of a random events vector $X = \{X_1, X_2, \dots, X_n\}$ is defined as:

$$C_n(X_1, X_2, \dots, X_n) = \frac{\partial^n}{\partial k_1 \dots \partial k_n} K_X(k) \quad (4.8)$$

with the following generating function at $k = 0$.

$$K_X(k) = \ln\{E[\exp(k \cdot X)]\} \quad (4.9)$$

This leads to the biased fourth-order multivariate cumulants estimator in terms of products

of higher order moments.

$$\begin{aligned}
C_4(X_1, X_2, X_3, X_4) &= E[X_1 X_2 X_3 X_4] - E[X_1 X_2 X_3]E[X_4] - E[X_1 X_2 X_4]E[X_3] - E[X_1 X_3 X_4]E[X_2] \\
&\quad - E[X_2 X_3 X_4]E[X_1] - E[X_1 X_2]E[X_3 X_4] - E[X_1 X_3]E[X_2 X_4] - E[X_1 X_4]E[X_2 X_3] \\
&\quad + 2\{E[X_1 X_2]E[X_3]E[X_4] + E[X_1 X_3]E[X_2]E[X_4] + E[X_1 X_4]E[X_2]E[X_3] \\
&\quad + E[X_2 X_3]E[X_1]E[X_4] + E[X_2 X_4]E[X_1]E[X_3] + E[X_3 X_4]E[X_1]E[X_2]\} \\
&\quad - 6E[X_1]E[X_2]E[X_3]E[X_4]
\end{aligned} \tag{4.10}$$

The procedure for obtaining the derivation of the unbiased fourth-order multivariate cumulants is as follows. We can easily observe that $E[\overline{X_1 X_2 X_3 X_4}] = E[X_1 X_2 X_3 X_4]$. Hence, different multiplicity structures of $\{X_1, X_2, X_3, X_4\}$ can be calculated based on the expressions below. Expressions $E[\overline{X_1 X_2 X_4 X_3}]$, $E[\overline{X_1 X_3 X_4 X_2}]$, $E[\overline{X_2 X_3 X_4 X_1}]$ and $E[\overline{X_1 X_2 X_3 X_4}]$ can also be calculated through equation (4.11).

$$E[\overline{X_1 X_2 X_3 X_4}] = \frac{1}{L^2} \sum_{i,j}^L E[X_{1_i} X_{2_i} X_{3_j} X_{4_j}] \tag{4.11}$$

where i, j are realizations of multiplicities. This leads to:

$$L^2 E[\overline{X_1 X_2 X_3 X_4}] = \{L(L-1)E[X_1 X_2 X_3]E[X_4] + LE[X_1 X_2 X_3 X_4]\} \tag{4.12}$$

Expressions $E[\overline{X_1 X_3 X_2 X_4}]$, $E[\overline{X_1 X_4 X_2 X_3}]$, $E[\overline{X_2 X_3 X_1 X_4}]$, $E[\overline{X_2 X_4 X_1 X_3}]$ and $E[\overline{X_3 X_4 X_1 X_2}]$ can also be calculated based on:

$$E[\overline{X_1 X_2 X_3 X_4}] = \frac{1}{L^3} \sum_{i,j,k}^L E[X_{1_i} X_{2_i} X_{3_j}, X_{4_k}] \tag{4.13}$$

that results in:

$$\begin{aligned}
L^3 E[\overline{X_1 X_2 X_3 X_4}] = & \\
& L(L-1)(L-2)E[X_1 X_2]E[X_3]E[X_4] \\
& + L(L-1)\{E[X_1 X_2 X_3]E[X_4] + E[X_1 X_2 X_4]E[X_3]\} \\
& + L(L-1)E[X_1 X_2]E[X_3 X_4] + LE[X_1 X_2 X_3 X_4] \tag{4.14}
\end{aligned}$$

Eventually, the expression $E[\overline{X_1 X_2 X_3 X_4}]$ can be calculated as:

$$E[\overline{X_1 X_2 X_3 X_4}] = \frac{1}{L^4} \sum_{i,j,k,l}^L E[X_{1_i} X_{2_j}, X_{3_k} X_{4_l}] \tag{4.15}$$

which can be explicitly stated as:

$$\begin{aligned}
L^4 E[\overline{X_1 X_2 X_3 X_4}] = & \\
& L(L-1)(L-2)(L-3)E[X_1]E[X_2]E[X_3]E[X_4] \\
& + L(L-1)(L-2)\{E[X_1 X_2]E[X_3]E[X_4] + 5 \text{ o.p.}\} \\
& + L(L-1)\{E[X_1 X_2 X_3]E[X_4] + 3 \text{ o.p.}\} \\
& + L(L-1)\{E[X_1 X_2]E[X_3 X_4] + 2 \text{ o.p.}\} + LE[X_1 X_2 X_3 X_4] \tag{4.16}
\end{aligned}$$

where ‘o.p.’ means other permutations of the variables in e.g. $E[\overline{X_1 X_2}] E[\overline{X_3}] E[\overline{X_4}]$ that give rise to (non-identical) terms like $E[\overline{X_1 X_3}] E[\overline{X_2}] E[\overline{X_4}]$. Then, the $C_4(X_1, X_2, X_3, X_4)$ can be derived for equation (4.10) as:

$$\begin{aligned}
C_4(X_1, X_2, X_3, X_4) = & \frac{L^2}{L^3 - 6L^2 + 13L - 12} \times \\
& \{(L+1)\overline{X_1 X_2 X_3 X_4} - (L+1)(\overline{X_1 X_2 X_3 X_4} + 3 \text{ o.p.}) - \\
& (L-1)(\overline{X_1 X_2 X_3 X_4} + 2 \text{ o.p.}) + 2L(\overline{X_1 X_2 X_3 X_4} + 5 \text{ o.p.}) \\
& - 6L\overline{X_1 X_2 X_3 X_4}\} \tag{4.17}
\end{aligned}$$

For the special case if $X_1 = X_2 = X_3 = X_4$, we obtain the equation (4.7).

4.2.2 Polar Coordinate Transformation

Mapping $I - Q$ values of the received signal's symbols in $I - Q$ plane to polar coordinates can be easily conducted through establishing the relationship between $I - Q$ and $r - \theta$ values as $\mathbf{r} = \sqrt{\mathbf{I}^2 + \mathbf{Q}^2}$ and $\theta = \arctan(\mathbf{Q}/\mathbf{I})$ where \mathbf{I} and \mathbf{Q} indicate the real and imaginary parts of the received complex symbols, and \mathbf{r} and θ represent the radius and angle of the polar transformed coordinates. As can be seen from the simple mathematical process of polar coordinate transformation, this component does not increase the computational complexity of the feature extraction stage since 1) it does not require any recursive computation compared to the high-order statistical feature extraction component, and 2) the polar coordinate transform component operates in parallel with high-order statistical feature extraction component. It should be mentioned that due to the high-order statistical feature extraction's recursive nature, its computational complexity is longer than the polar coordinate transform. On the other hand, this transformation provides more in-depth information on the symbols' placements within the constellation for the subsequent labeling stage.

4.3 Proposed Deep Learning Structure for Labeling Stage

NNs contain hidden layers consisting of some number of neurons. Each single neuron in each middle hidden layer is connected to all neurons in the previous and subsequent hidden layer through links, each with associated weights, which determine the overall value for a neuron's output. There are different functions, also known as activation functions, to calculate the link weights. The selected numbers for hidden layers and neurons, as well as the activation function computing the links' weights, influence the accuracy and computational cost of the FB AMC classifier. If we select more than 2 hidden layers, then the neural network is called a deep neural network. DNN classifiers generally are capable of faster classification

operations and more accurate learning of non-linear patterns than SVM. Although DNN performs on average higher than SVM, its performance is not appreciably higher than that of SVM, especially in lower SNR values. Thus, a recurrent neural network was proposed to be used in AMC to overcome this issue. We will next investigate a specific NN architecture, RNN-LSTM, coupled with DNN that has been recently proposed in AMC applications. A recurrent neural network is an architecture that aims to address the issue of ML algorithms' learning process having to restart from scratch. In practice, this task is done by creating loops over the ML algorithm's learning process to be frequently trained over different portions (m number) of a cross-validated training set, as shown in Fig 4.3.

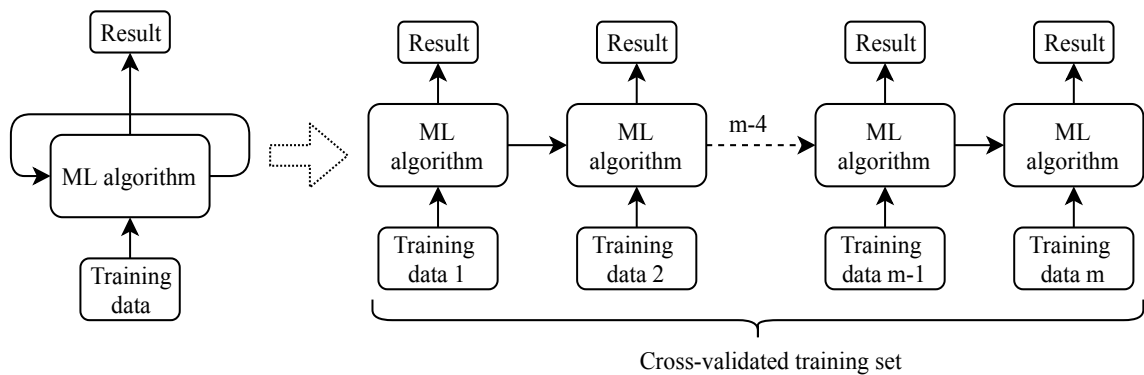


Figure 4.3: RNN training process over m portion of cross-validated training set.

This procedure trains the ML algorithm m times which, for an ML algorithm, results in learning and discerning the relationship in a much more accurate manner between various parameters' values in a dataset by not being trained only once as in conventional techniques. In other words, RNN architecture is capable of connecting and relating previously learned information to the present learning process. Although it is correct that RNN architecture builds a stronger learning procedure, it might not be necessary for the ML algorithm to learn previous information and correlate it with the new one. In order to address this issue in the RNN architecture, long-short term memory as an architecture derived from RNN was proposed. LSTM is explicitly designed to avoid the long-term dependency problem. In all RNN architectures, there exists the form of a chain of repeating modules of a neural

network. In standard RNNs, this repeating module will have a very simple structure, such as a single \tanh layer. LSTMs also follow the same chain-like structure, but the repeating module has a different procedure. This module follows the procedure in the block shown in Fig 4.4 where it is replaced with ML algorithm blocks in Fig 4.3.

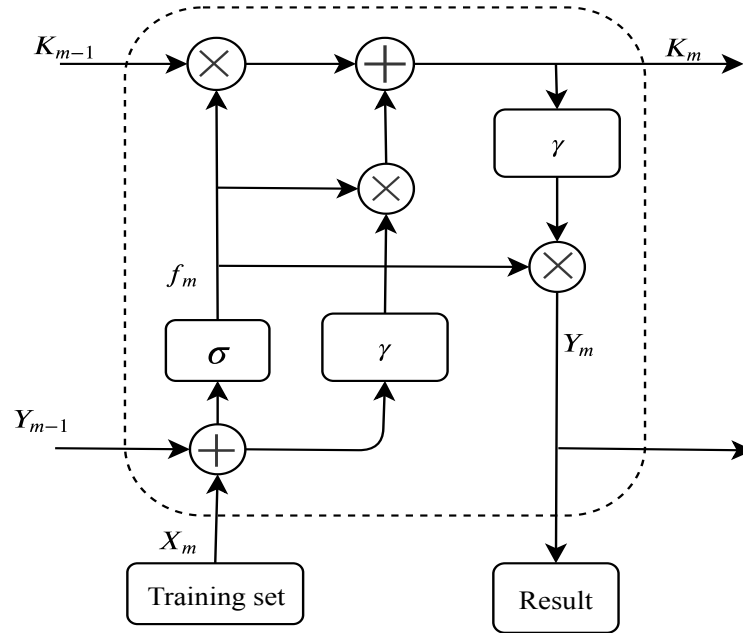


Figure 4.4: LSTM module in each RNN architecture training iteration.

In this module, the σ block, also known as sigmoid, is responsible for producing a binary value indicating what information from a previous training iteration will be neglected and vice versa. The γ block, on the other hand, creates a vector of new candidate values that will be added to the previous training iteration's results if the output of sigmoid block is 1. The outputs of LSTM module are as follows:

$$K_m = \{K_{m-1} \cdot f_m\} + \{f_m \cdot \gamma(Y_{m-1} + X_m)\} \quad (4.18)$$

$$Y_m = f_m \cdot \gamma(K_m) \quad (4.19)$$

where $f_m = \sigma(Y_{m-1} + X_m)$. The performance of an FB AMC classifier using LSTM layers is expected to be much higher than a conventional DNN in terms of final classification

accuracy (P_{CC}), since it first follows RNN architecture and then selects more correlated information to be transferred to the next iteration of training. To summarize, utilizing these layers can benefit the FB AMC classifier by 1) extracting temporal information of the received signal, 2) distinguishing more accurately among different signal samples, and 3) requiring less optimization of hyperparameters due to its property of weight sharing across time steps. Hence, we use two stacked-LSTM layers in our deep learning architecture. The output of these LSTM layers follows the *temporal attention mechanism* in order to: 1) save parts of the derived information, and 2) avoid overfitting. This mechanism also benefits the deep learning architecture by adaptively deriving the final output of an LSTM layer using the outputs of all time steps. This mechanism works as follows. The output of the LSTM layer $\{\mathbf{y}_t\}_{t=0}^{T-1}$ through processing of a shared time-distributed neural network layer that is characterized with weight \mathbf{W}_α and bias ζ_α matrices results in the calculation of attention weights $\alpha = \{\alpha_t\}_{t=0}^{T-1}$ based on a softmax activation function as in (4.20).

$$\alpha_t = \frac{\sigma(\mathbf{y}_t \cdot \mathbf{W}_\alpha + \zeta_\alpha)}{\sum_{t=0}^{T-1} \sigma(\mathbf{y}_t \cdot \mathbf{W}_\alpha + \zeta_\alpha)} \quad (4.20)$$

It should be noted that $\sum_{t=0}^{T-1} \alpha_t = 1$ while $\alpha_t \geq 0$. Then the final output is calculated as:

$$\mathbf{y} = \sum_{t=0}^{T-1} \alpha_t \mathbf{y}_t \quad (4.21)$$

This output is then provided to a fully-connected network (FCN), which we will introduce in the following sections. At the end, a softmax activation function is also employed to provide the final result. The Deep Learning architecture for our labeling stage can be seen in Fig 4.5.

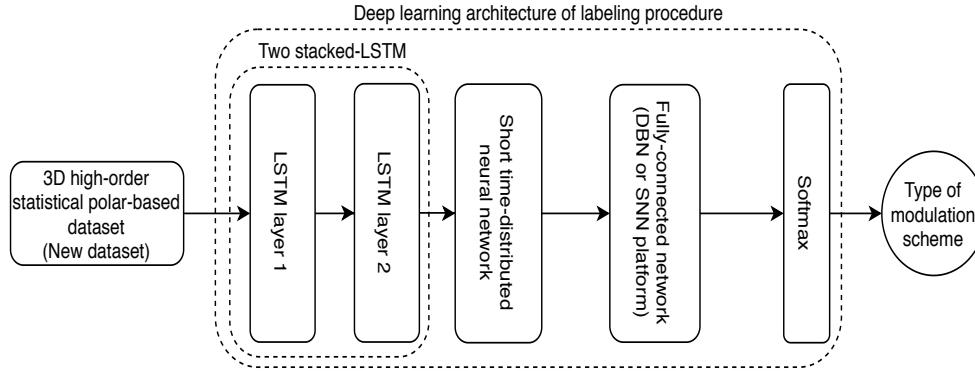


Figure 4.5: Deep Learning architecture of the labeling stage.

4.4 Deep Belief Network as Fully-Connected Network in Deep Learning Structure

The main idea behind proposing the use of DBN for AMC applications is to address the vanishing gradient problem in training the artificial neural network with gradient-based learning methods and back-propagation that have already been used for AMC. The solution to this problem is comprised of two parts [28]. The first involves a restricted boltzmann machine (RBM). This is a method that can automatically find patterns in our data by reconstructing the input. An RBM is a shallow two-layer network shown in Fig 4.6; the first layer is known as the visible layer and the second is called the hidden layer.

Each node in the visible layer (V_i) is connected to every node in the hidden layer (h_j). An RBM is considered restricted because no two nodes in the same layer share a connection [29]. An RBM is the mathematical equivalent of a two-way translator, where its energy function can be defined as the following based on parameters in Fig 4.6 in one layer of an RBM network:

$$\mathcal{E}(\mathbf{V}, \mathbf{h}) = -\sum_i a_i V_i - \sum_j b_j h_j - \sum_{i,j} V_i h_j w_{ij} \quad (4.22)$$

where \mathbf{V} and \mathbf{h} are respectively the vectors of units in the visible and hidden layers. In the forward pass, an RBM takes the inputs and translates them into a set of numbers that

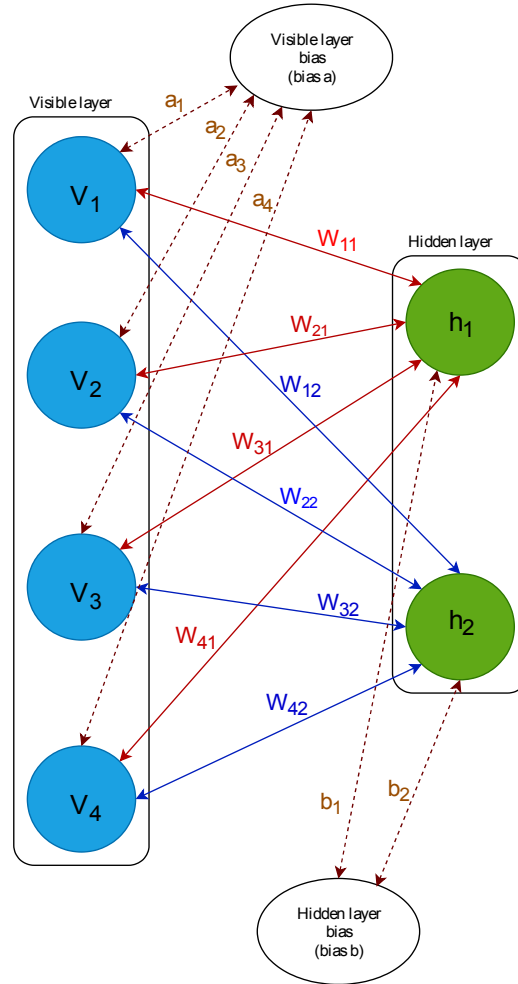


Figure 4.6: An example RBM structure with 4 visible and 2 hidden units in their corresponding layers in which the effect of biases on visible and hidden layers units can be observed.

encode the inputs. In the backward pass, it takes this set of numbers and translates them back to form the re-constructed inputs [29]. A well-trained network will be able to perform the backwards translation with a high degree of accuracy. In both steps, the weights and biases have a very important role. In each certain state, an RBM assigns probabilities rather than discrete values to each link. This makes RBMs probabilistic. Accordingly, the joint distribution of each certain state is defined as [29]:

$$p(\mathbf{V}, \mathbf{h}) = \frac{1}{\mathbf{Z}} \exp(-E(\mathbf{V}, \mathbf{h})) \quad (4.23)$$

Where \mathbf{Z} is called partition function.

$$\mathbf{Z} = \sum_{\mathbf{V}, \mathbf{h}} \exp(-E(\mathbf{V}, \mathbf{h})) \quad (4.24)$$

It should be noted that it is a difficult process to calculate the above joint probability due to the large number of possible combinations of \mathbf{V} and \mathbf{h} in the partition function \mathbf{Z} . On the other hand, conditional probabilities of \mathbf{V} given \mathbf{h} , and \mathbf{h} given \mathbf{V} are much easier to calculate.

$$p(\mathbf{V}|\mathbf{h}) = \prod_i p(v_i|\mathbf{h}) \quad (4.25)$$

$$p(\mathbf{h}|\mathbf{V}) = \prod_j p(h_j|\mathbf{V}) \quad (4.26)$$

Breaking down the above conditional probabilities while considering that each unit can only exist in a binary state of 0 or 1 will lead us to:

$$p(v_i = 1|\mathbf{h}) = \sigma(a_i + \sum_j W_{ij}h_j) \quad (4.27)$$

And analogously,

$$p(h_j = 1|\mathbf{V}) = \sigma(b_j + \sum_i W_{ij}V_i) \quad (4.28)$$

where $\sigma(\cdot)$ is the sigmoid function with $\sigma(x) = \frac{1}{1+\exp(-x)}$ as its definition. The entire procedure above helps the RBM in deciding which input features are the most important when detecting patterns. Through several forward and backward passes, an RBM is trained to reconstruct the input data. Three steps are repeated iteratively through the training process [29]:

1. With a forward pass, every input is combined with an individual weight and one overall bias, and the result is passed to the hidden layer which may or may not activate.
2. Next, in a backward pass, each activation is combined with an individual weight and an overall bias, and the result is passed to the visible layer for reconstruction.
3. At the visible layer, the reconstruction is compared against the original input to determine the quality of the result.

Steps 1 through 3 are repeated with varying weights and biases until the input and the re-construction are as close as possible [29]. In other words, the update matrix for new weights is: $\mathbf{W}_{new} = \mathbf{W}_{old} + \Delta\mathbf{W}$, where $\Delta\mathbf{W} = \mathbf{V}_0 \otimes p(\mathbf{h}_0|\mathbf{V}_0) - \sum_{ij} \mathbf{V}_i \otimes p(\mathbf{h}_j|\mathbf{V}_i)$. An interesting aspect of an RBM is that the data does not need to be labelled. This turns out to be very important for real-world data sets such as over-the-air received signals. An RBM automatically sorts through the data, and by properly adjusting the weights and biases an RBM is able to extract the important features and reconstruct the input[7]. An important note is that an RBM is actually making decisions about which input features are important and how they should be combined to form patterns[30]. In other words, an RBM is part of a family of feature extraction neural networks that are all designed to recognize inherent patterns in data. These networks are also called *auto-encoders*, because in a way they have to encode their own structure[8]. For the second part of the solution, we obtain a powerful new model that finally solves the problem by combining RBMs together and introducing a carefully chosen training method. A Deep Belief Network (DBN) can be viewed as a stack of RBMs, where the hidden layer of one RBM is the visible layer of the one above it within the DBN. Therefore, the joint distribution of DBN of l layers is as follows [30]:

$$p(\{h_1, h_2, \dots, h_l\}) = \left(\prod_{k=0}^{l-2} p(h_k|h_{k+1}) \right) p(h_{l-1}, h_l) \quad (4.29)$$

where $\mathbf{h} = \{h_1, h_2, \dots, h_l\}$ is the vector of all layers in the DBN. A DBN is trained as follows:

- The first RBM is trained to re-construct its input as accurately as possible.
- The hidden layer of the first RBM is treated as the visible layer for the second and the second RBM is trained using the outputs from the first RBM.
- This process is repeated until every layer in the network is trained.

An important note about DBNs is that each RBM layer learns the entire input. In other kinds of models, such as convolutional networks, early layers detect simple patterns and later layers recombine them [30]. A DBN, on the other hand, works globally by fine-tuning the entire input in succession as the model slowly improves. The reason that a DBN works so well is that a stack of RBMs will outperform a single unit [29]. After this initial training, the RBMs create a model that can detect inherent patterns in the data [30]. But we still don't know exactly what the patterns are called [29]. To finish training, we need to introduce labels to the patterns and fine-tune the network with supervised learning [29]. To do this, we need a very small set of labeled samples so that the features and patterns can be associated with a name. The weights and biases are altered slightly, resulting in a small change in the network's perception of the patterns, and often a small increase in the total accuracy [29]. Fortunately, the set of labelled data can be small relative to the original data set, which as we've discussed is extremely helpful in real-world applications [30]. As mentioned before, a DBN only needs a small labelled dataset, which is important for real-world applications. The training process can also be completed in a reasonable amount of time through the use of GPUs. And best of all, the resulting network will be very accurate compared to a shallow network.

Learning parameters, also known as hyperparameters, play a very important role in the accuracy of the training process. Hence, setting optimized hyperparameters can finally result in an increase in classification accuracy. Optimizing the hyperparameters when DBN is deployed as FCN in a deep learning structure can be done by two methods.

1. Manually, by several trials to evaluate the training process's performance. This method

is not only inefficient, but it also exhibits lower effectiveness in enhancing the training process.

2. Optimizing algorithms specifically designed for such a purpose, like gradient-based optimization.

We select the adaptive moment estimation (Adam) algorithm to optimize the hyperparameters of the learning process in a deep learning structure when DBN is employed.

4.4.1 Adaptive Moment Estimation

Adaptive moment estimation algorithm can be looked at as a combination of RMSprop and stochastic gradient descent (SGD) with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using the moving average of the gradient instead of the gradient itself like SGD with momentum. Adam is an adaptive learning rate method, which means it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network. n^{th} moment of a random variable is defined as the expected value of that variable to the power of n . This relation can be seen below.

$$M_n = E[X^n] \tag{4.30}$$

Note that the gradient of the cost function of a neural network can be considered a random variable, since it is usually evaluated on some small random batch of data. The first moment is mean, and the second moment is uncentered variance. In other words, there is no need to subtract the mean during variance calculation. To estimate the moments, Adam utilizes

exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

$$M_t = \beta_1 M_{t-1} + (1 - \beta_1) g_t \quad (4.31)$$

$$\mathcal{V}_t = \beta_2 \mathcal{V}_{t-1} + (1 - \beta_2) g_t^2 \quad (4.32)$$

where M and \mathcal{V} are moving averages, g is the gradient on the current mini-batch, and β_i — is new introduced hyper-parameters of the algorithm. They have very good default values of 0.9 and 0.999, respectively. Almost no designer ever changes these values. The vectors of moving averages are initialized with zeros at the first iteration. These values correlate with the moment, defined as in (4.30). Since M and \mathcal{V} are estimates of first and second moments, the following property should be held for all iterations:

$$E[M_t] = E[g_t] \quad (4.33)$$

$$E[\mathcal{V}_t] = E[g_t^2] \quad (4.34)$$

Expected values of the estimators should equal the parameter we are trying to estimate. Fortunately, the parameter in our case is also the expected value. If these properties held true, that would mean that we have unbiased estimators. Now, we will see that these do not hold true for our moving averages. Because the process is an initialized averages with zeros, the estimators are biased towards zero. We can prove that for M . It should be noted that the proof for \mathcal{V} would be analogous to prove that we need the formula for M to the very first gradient. By feeding some values of M , we can see the direction the pattern follows.

$$M_0 = 0 \quad (4.35)$$

$$M_1 = \beta_1 M_0 + (1 - \beta_1) g_1 \quad (4.36)$$

$$M_2 = \beta_1 M_1 + (1 - \beta_1) g_2 \quad (4.37)$$

$$M_3 = \beta_1 M_2 + (1 - \beta_1) g_3 \quad (4.38)$$

As can be seen, the further this process expands the value of M , the less the first values of the gradients contribute to the overall value, as they get multiplied by smaller and smaller β_i . Capturing this pattern, we can rewrite the formula for our moving average as:

$$M_t = (1 - \beta_1) \sum_{i=0}^t \beta_1^{t-i} g_i \quad (4.39)$$

In order to remove the discrepancy of two expected values of M , we can relate it to the true first moment:

$$E[M_t] = E[(1 - \beta_1) \sum_{i=0}^t \beta_1^{t-i} g_i] \quad (4.40)$$

$$E[M_t] = E[g_i](1 - \beta_1) \sum_{i=0}^t \beta_1^{t-i} + K \quad (4.41)$$

$$E[M_t] = E[g_i](1 - \beta_1^t) + K \quad (4.42)$$

In the first row, the new formula is used for the moving average to expand M . Next, by approximating g_i with g_t , we can take it out of the sum, since it does not now depend on i . Because the approximation is taking place, the error K emerges in the formula. In the last line, we just use the formula for the sum of a finite geometric series. There are two things we should note from that equation.

1. We have a biased estimator. This is not just true for Adam only; the same holds for algorithms, using moving averages (SGD with momentum, RMSprop, etc.).
2. It won't have much effect because the value β to the power of t is quickly going towards zero.

Now we need to correct the estimator, so that the expected value is the accurate one. This step is usually referred to as bias correction. The final formulas for our estimator will be as

follows:

$$\hat{M}_t = \frac{M_t}{1 - \beta_1^t} \quad (4.43)$$

$$\hat{V}_t = \frac{V_t}{1 - \beta_2^t} \quad (4.44)$$

$$(4.45)$$

The only step left to do in this algorithm is to use those moving averages to scale the learning rate individually for each parameter. The way it is done in Adam is by performing a weight update as follows:

$$W_t = W_{t-1} - \eta \frac{\hat{M}_t}{\sqrt{\hat{V}_t + \epsilon}} \quad (4.46)$$

where W is model weights, and η is the step size.

Algorithm 1 summarizes aforementioned description of Adam algorithm.

Algorithm 1 Adam algorithm to optimize DBN hyperparameters.

Require: η

Require: $f(W)$ (Stochastic objective function)

Require: initializing W_0

$t \leftarrow 0$

$M_{t=0} \leftarrow 0$

$V_{t=0} \leftarrow 0$

while W_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \Delta_W f_t(W_{t-1})$ (Obtaining gradients with respect to objective at timestep t)

$M_t \leftarrow \beta_1 M_{t-1} + (1 - \beta_1) g_t$

$V_t \leftarrow \beta_2 V_{t-1} + (1 - \beta_2) g_t^2$

$\hat{M}_t \leftarrow \frac{M_t}{(1 - \beta_1^t)}$

$\hat{V}_t \leftarrow \frac{V_t}{(1 - \beta_2^t)}$

$W_t \leftarrow W_{t-1} - \eta \frac{\hat{M}_t}{\sqrt{\hat{V}_t + \epsilon}}$

end while

return W_t

In general, we can list the three main Adam properties below.

1. The actual step size taken by the Adam in each iteration is approximately bounded by the step size hyperparameter. This property adds intuitive understanding to our previous unintuitive learning rate hyperparameter.
2. The step size of the Adam update rule is invariant to the magnitude of the gradient, which helps a lot when going through areas with tiny gradients (such as saddle points or ravines). In these areas, SGD struggles to quickly navigate through them.
3. Adam was designed to combine the advantages of Adagrad, which works well with sparse gradients, and RMSprop, which works well in online settings. Having both of these enables us to use Adam for a broader range of tasks. Adam can also be looked at as the combination of RMSprop and SGD with momentum.

Chapter 4 presents the optimized hyperparameters by Adam when DBN is used as FCN.

4.5 Spiking Neural Network as a Fully-Connected Network in a Deep Learning Structure

In order to fully understand how SNNs work, we first need to go through two other basic model networks.

4.5.1 Threshold Unit Networks

The first generation of neural network models is based on McCulloch-Pitts neurons or logical threshold units. Such neurons from the first generation are Boolean, that is, they can output either a zero or a one. This is of course motivated by the observation that biological neurons either fire or not, i.e., a spike is transmitted along the axon if and only if the membrane potential surpasses a certain threshold value (provided the cell is not in the absolutely refractory phase shortly after a spike, when no new action potential can

be initiated). Binary encoding of neural activity therefore seems very plausible at rst. So neurons of the first generation receive binary inputs (ρ_i) from other neurons which, depending on the corresponding synaptic weights (w_i), can cause either a excitatory or inhibitory postsynaptic potential (EPSP and IPSP respectively). All stimuli are added up, and if their sum is large enough (corresponding to sufficient depolarization of the membrane at the axon hillhock), a spike is transmitted to the next neurons, i.e., the output is 1. The dynamics of such a threshold unit can thus be summarized as:

$$y = \begin{cases} 1 & , \text{if } h = \sum_i \rho_i w_i > u \\ 0 & , \text{Otherwise} \end{cases} \quad (4.47)$$

While the threshold u was taken to be zero in the first models, non-zero thresholds (biases) were introduced in later models for more flexibility. Models of the first generation include, among others, multilayer perceptrons (MLPs). Due to the binary nature of threshold unit neurons resembling spiking activity, the appropriate learning algorithm for such networks is Hebbian learning. In the case of threshold unit networks, this translates to a weight update rule of the form:

$$\Delta w_i = \lambda \rho_i y \quad (4.48)$$

where λ is a predefined learning rate. In other words, when input i 's firing is involved in a neuron's firing, the connection is strengthened. Neural networks of the first generation have been shown to be universal for digital, i.e., logical, computation. The proof is relatively simple: it can be shown that by choosing appropriate weights, a McCulloch-Pitts neuron can compute the logical AND, OR and NOT operators; since any Boolean function can be expressed as a composition of these three logical operators, any such function can be computed by an MLP with a single hidden layer. However, the restriction to binary outputs is also a considerable limitation in an analog world. Moreover, the first generation models

do not include a notion of time, but instead assume synchronous updates of units.

4.5.2 Continuous Neural Networks

The second generation of neural networks arise as a very natural extension of the previous generation by allowing real numbers as inputs and outputs, thereby facilitating analog computation. This is achieved by simply replacing the thresholding after summing up the weighted stimuli by a continuous activation function g (note that the step function is discontinuous at the threshold value). The dynamics of a single unit from the second generation can then (except in very special cases) be summarized as:

$$y = g\left(\sum_i w_i \rho_i - b\right) \quad (4.49)$$

where b is the previously mentioned bias term. The standard choice for g used to be a sigmoid function, e.g., the logistic or hyperbolic tangent function; however, other simpler functions such as rectified linear units (ReLUs) have become very popular recently in the context of deep learning. Almost all modern NNs such as sigmoid feed-forward, radial basis function, or recurrent neural networks, e.g., the popular and powerful LSTMs. While the binary outputs of McCulloch-Pitts neurons have a very intuitive interpretation as spikes, it is not straightforward to motivate the real valued outputs of artificial neurons of the second generation. Yet, a biological motivation exists also in this case. Instead of coding spikes, these units encode the firing rates of neurons, i.e., frequencies of spikes averaged over some time window. This is biologically justified by observations that neurons often react to stimulus not with a single spike, but instead fire bursts of many spikes within a very short time, and they can fire at a range of intermediate frequencies between their maximal and minimal firing rates. Such an information coding scheme is known as (firing-) rate coding, and it is one of the most prominent coding schemes in neuroscience. However, its applicability is very likely limited. In contrast, Hebbian learning is no longer directly

applicable to the second generation networks since units are computing rates instead of spikes. One of the key advantages of continuous output activation functions is that such models are receptive for the large class of gradient-based optimization techniques from mathematics. The standard learning algorithm for second generation networks is therefore gradient-descent, usually in combination with the backpropagation algorithm, which allows us to propagate error signals (in the case of supervised learning) backwards through the topological ordering of the network. This enables the training of networks with many layers of artificial neurons, so-called deep networks, and is one of the key ingredients of modern deep learning approaches. Hence, neural networks from the second generation introduce an implicit notion of time into the model by computing with firing rates instead of spikes, which makes them biologically more plausible than their predecessors. This was achieved by adapting the previous model to incorporate some new findings from neuroscience. Moreover, networks of the second generation are also universal for digital computation (applying thresholding to the real valued output), and can in fact compute certain Boolean functions with fewer units than threshold units from the first generation. In this sense, second generation networks are computationally more powerful than first generation networks. Furthermore, in a result known as the universal approximation theorem, it has been proven that a network from the second generation with a single hidden layer can approximate any continuous function arbitrarily well.

4.5.3 Spiking Neural Networks

The major difference between SNNs and the other neural networks described above is that SNNs model time explicitly. This is based on the central paradigm of spiking networks in that it is the exact timing of individual spikes, rather than their firing rate averaged over some time window, which carries information in biological brains. SNNs are thus dynamic systems which are usually formulated as systems of ordinary differential equations (ODEs). The central object of SNN models is the membrane potential $v(t)$ of a neuron,

which represents its internal state at time t . Since the membrane potential determines the spiking activity, it is necessary to model its time evolution in order to work with exact timing of spikes. Many different models for spiking neurons exist, and a common framework for different models is still largely lacking - an issue that will be discussed in more detail later. These common characteristics are shared by almost all such models:

1. Neurons receive multiple continuous-time inputs from their synapses with other neurons in form of spikes, which are usually modelled as Dirac delta functions.
2. Such synaptic stimuli can be either excitatory, i.e., increasing the membrane potential and thereby the probability of firing, or inhibitory, i.e., decreasing the membrane potential.
3. They produce a single output spike whenever their membrane potential reaches a certain threshold value.

Probably the first and simplest SNN models are the Integrate-and-fire (IF) model and a slightly improved version thereof called the Leaky-integrate-and-fire (LIF) model, whose dynamics are given by:

$$C \frac{dv}{dt}(t) = I(t) - \frac{v(t)}{R} \quad (4.50)$$

$$C \frac{dv}{dt}(t) = I_{ext}(t) + \sum_j w_j I_j(t) - \frac{v(t)}{R} \quad (4.51)$$

where C is the membrane capacitance, R the membrane resistance, and $I(t)$ the total input current to the neuron at time t , which can be decomposed as external current and a sum over all currents transmitted from neurons j with synaptic weights w_j . The last term ($v(t)$) was introduced in the LIF model to account for an exponential decay of the membrane potential to its resting state in lack of new stimuli not accounted for in the earlier IF model. The external current, $I_{ext}(t)$, in the LIF ODE is only relevant for the subset of input neurons in the case of artificial SNNs, and can be ignored for the remaining network dynamics. The

resulting equation looks somewhat similar to the inner term of the activation function for second generation models, also containing a summation over inputs multiplied by their synaptic weights. The key difference here is that inputs are spikes in time rather than rates. The exponential decay term might be loosely interpreted as a biased term over time: if no new spikes arrive timely enough to cause an action potential, the membrane returns to its resting state.

Whereas it is relatively easy to understand why SNNs are more biologically plausible than standard ANNs, the issue regarding their computational power is less clear. However, various theoretical results about the computational power of artificial SNNs have been published. In these works, it was shown that any function which can be computed by a network of the first two generations can also be computed by an artificial SNN. In particular, this includes the universal approximation property for SNNs, stating that any continuous function $F : [0, 1]^n \rightarrow [0, 1]^k$ can be approximated arbitrarily closely (with regard to uniform convergence) by one hidden layer network of spiking neurons with simple piece-wise linear response- (shape of the EPSPs and IPSPs for non-linear responses) and threshold functions (regulating firing threshold and absolutely refractory period). Moreover, it was demonstrated that certain functions can in fact be computed by SNNs with much fewer units than those required by networks of the first and second generation. For example, the Boolean function $CD_n : \{0, 1\}^{2n} \rightarrow \{0, 1\}$

$$CD_n(x_1, \dots, x_n, y_1, \dots, y_n) = \begin{cases} 1 & , \text{ if } \rho_i = y_i \text{ for some } i \\ 0 & , \text{ Otherwise} \end{cases} \quad (4.52)$$

can be computed by a single spiking neuron with appropriately chosen weights and delays, whereas a threshold gate network from the first generation computing CD_n has at least $\frac{n}{\log(n+1)}$ units, and a continuous neural network from the second generation computing CD_n has at least $n^{1/4}$ units asymptotically. Note that CD_n is not just any arbitrary function, but has a biological interpretation as pattern matching or coincidence detection from two sources

x_i and y_i , and is therefore likely to be computed by biological brains as well.

In summary, in contrast to other NNs, SNNs include time explicitly in the model. The output and input of individual neurons in SNNs (except for the case of input units which receive external currents) are completely determined by the firing times of other neurons j . However, computing the output firing times requires us to explicitly model the membrane potential as an ODE. The structure of SNNs is a directed graph, identical to other NNs.

Training SNNs has been historically a difficult task. SNNs have been originally destined to be unsupervisedly trained.

4.5.4 Unsupervised Learning-Hebbian Learning

Similar to the first generation of neural networks, modelling the firing of neurons explicitly in SNNs has the advantage that Hebbian learning is applicable. However, Hebbian's rule does not explicitly mention the timing of spikes, but instead just refers to neurons firing together. While this issue does not arise for threshold gates, which are assumed to be synchronized and therefore always either fire together or do not fire together. For lack of better understanding, the units' activities in Hebbian's rule have traditionally been interpreted as firing rates and used in this context for continuous neural networks. A study in which the timing of presynaptic stimulus and postsynaptic action potential was systematically varied finally found a convincing correlation between the timing of pre- and postsynaptic spikes and the synaptic changes affected thereby. Generally speaking, presynaptic spikes occurring shortly before postsynaptic activity resulted in strengthening of the synaptic connection (potentiation), whereas presynaptic spiked arriving shortly after the postsynaptic spike weakened the connection (depression). This process has been termed Spike-timing-dependent-plasticity (STDP) and can be seen as an extension of Hebbian learning by exact spike timing. Interestingly, a similar but opposite process to STDP has also been observed in some synapses, i.e., presynaptic spiked following postsynaptic ones resulted in potentiation and vice versa for depression, and was termed anti-STDP. A rule for applying STDP-like

unsupervised learning in SNN models derived from these experimental findings can be formulated as:

$$\frac{d}{dt}w_{ji}(t) = a_0 + a_1S_i(t) + a_2S_j(t) + a_3S_i(t)\overline{S}_j(t) + a_4\overline{S}_i(t)S_j(t) \quad (4.53)$$

where S_i and S_j are the spike trains. \overline{S}_i and \overline{S}_j are low pass filtered versions thereof, i.e. with exponential decay instead of a single pulse, and a_i are constants governing the synaptic changes. Hence, different choices of the hyperparameters (considering w_{ji} as the main parameters) a_i can account for, e.g., STDP or anti-STDP.

The obtained hyperparameters associated with our deep learning structure are presented in Chapter 4.

4.6 Proposed Novel Framework Structure

Our literature review illustrates the fact that there is no framework available for AMC that provides flexibility and efficiency in terms of classification accuracy and computational complexity. Therefore, we address this shortcoming by proposing our adaptive framework that follows the deep learning architecture proposed in this research. This framework is built upon the fact that the AMC component in the receiver has to be trained for both ML platforms (DBN- and SNN-based models) before deployment. After the training processes are completed, the average classification accuracy for each platform for all modulation schemes at a particular SNR is recorded together with the number of timing units that are required for classification. In this manner, the FB AMC classifier, before deployment, has knowledge over required computational complexity and the likelihood of maximum classification accuracy of both ML platforms. Table 4.1 illustrates the required information to be recorded alongside with their notations, which will be further used in forming the decision ratio.

In this framework, the main equalizer of the receiver leverages the SNR value of

Table 4.1: Values to be collected

Values	Notation
Average DBN-based model's classification accuracy of all modulation schemes at each SNR	DBN-ACA-SNR
Average SNN-based model's classification accuracy of all modulation schemes at each SNR	SNN-ACA-SNR
Timing units required to perform DBN classification	DBN-t
Timing units required to perform SNN classification	SNN-t

the intercepted signal, and provides it to the AMC component, which then calculates the following ratio to determine the quantified trade-off of classification accuracy versus computational complexity. The obtained value represents the preferability of using the DBN-based model versus the SNN-based model at that SNR.

$$\frac{\left(\frac{DBN-ACA-SNR}{SNN-ACA-SNR}\right)}{\left(\frac{DBN-t}{SNN-t}\right)} \leq 1 \quad (4.54)$$

If this ratio is less than 1, favorability indicates the use of the SNN-based model to be employed for classification. If the ratio is above 1, then DBN is the better choice. This is due to the fact that the SNN-based model is capable of performing the classification with significantly less required number of timing units while maintaining classification accuracy comparable to what the DBN-based model offers. Therefore, this ratio creates an efficient trade-off between classification accuracy and computational complexity between the two models. This provides the flexibility to the AMC classifier to adapt to changing conditions, instead of being compelled to follow a fixed design. An overview of this framework's principal operation can be seen in Fig 4.7.

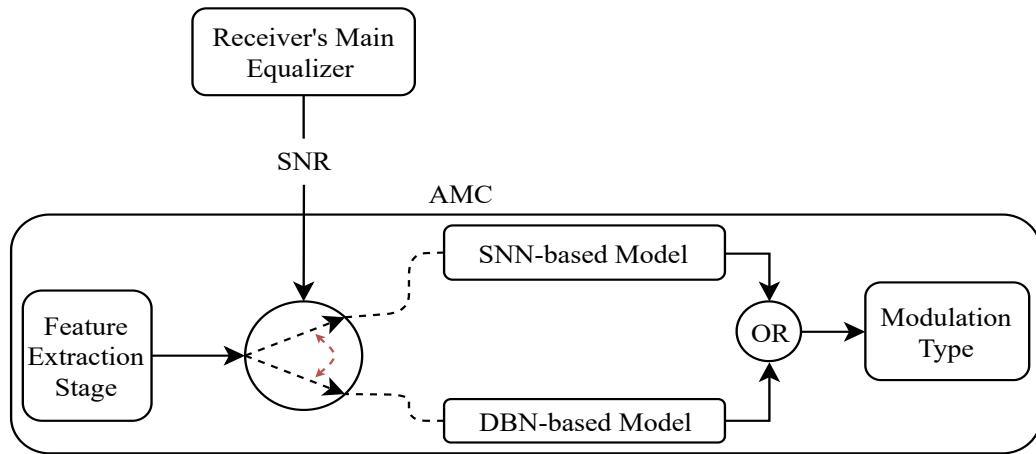


Figure 4.7: The proposed novel framework's principal working.

CHAPTER 5

Results, Analysis and Discussion

In this chapter, we will present the numerical results when each individual FB AMC classifier, DBN- and SNN-based, is implemented for classification. Afterwards, the novel framework will be analyzed.

In order to conduct the evaluation of our proposed platforms and our overall framework, we first obtain the lower and upper performance bounds for each introduced platform by selecting two of the lowest-order (BPSK and QPSK) and highest-order (128QAM and 256QAM) digital modulation schemes from the RadioML dataset [31]. Further, we compare their performance with other works from the scientific literature, specifically [32], [33], which represent two recent efforts that modified the structures of CNN and RNN in order to achieve higher classification accuracy. Details of this evaluation can be found in Appendix A.

We first present the RadioML dataset that is used as the signal reference for experiments of this thesis.

5.1 RadioML2018.01A Dataset

The RadioML dataset is widely used as a reference modulation dataset, and has been generated and recorded over the air by utilizing GNU Radio. The RadioML2018.01A dataset,

specifically, contains two categories of modulated signals with a total of 24 modulation schemes:

- **Normal group:** OOK, 4ASK, BPSK, QPSK, 8PSK, 16QAM, AM-SSB-SC, AM-DSB-SC, FM, GMSK, OQPSK.
- **Difficult group:** 8ASK, 16PSK, 32PSK, 16APSK, 32APSK, 64APSK, 128APSK, 32QAM, 64QAM, 128QAM, 256QAM, AM-SSB-WC, AM-DSB-WC.

The dataset's SNR covers a range of -20 dB to +30 dB in increments of 2dB, thus totalling 26 SNR values. There are 4096 signal waveforms for each modulation scheme at a particular SNR. Therefore, in this dataset, there are 106,496 signal waveforms for each modulation scheme for the entire SNR range. Each signal waveform includes 1024 separate complex IQ samples (2×1024). This creates a dataset of 2,555,904 vectors of modulated signal waveforms, with each vector having 1024 IQ samples. In our experiment, we split the dataset into 60% for training and validation, and 40% for testing processes. Table 5.1 investigates the dimensions of the input data for ML algorithms.

Table 5.1: Input data dimensions.

Data Type	Dimensions
RadioML Dataset	$2 \times 1024 \times 2,555,904$
Training Dataset	$2 \times 1024 \times 1,277,952$
Validation Dataset	$2 \times 1024 \times 255,590$
Testing Dataset	$2 \times 1024 \times 1,022,361$

In order to apply different wireless channels, or environmental effects, all modulated signals were exposed to real-world effects, such as additive white Gaussian noise, multipath fading, frequency offset and phase offset, in order to represent dynamic channel effects. The order of applying real-world effects has been selected to begin from most destructive one while continuing towards less destructive one. It also should be noted that in a given environment, the transmitted signal is not exposed to all of these real-world affects. In other

words, the RadioML dataset is created upon the assumption of having the most destructive environmental effects applied to signal. This process can be seen in Fig 5.1.

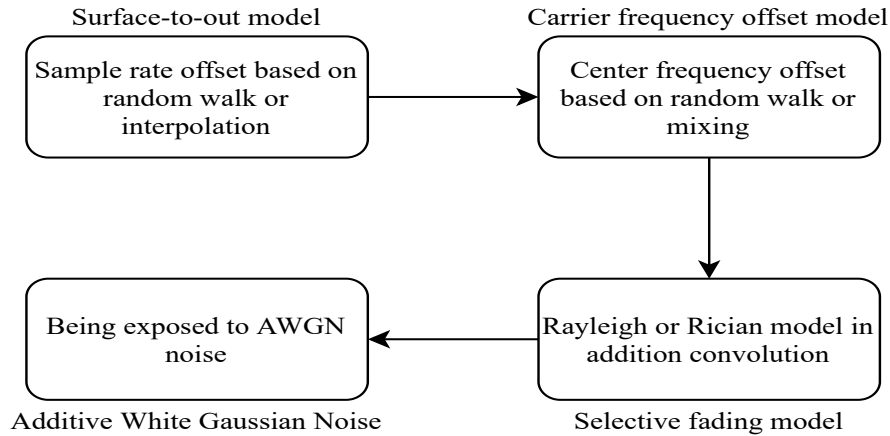


Figure 5.1: Process of applying channel effects to the transmitted signal

The order of applying aforementioned destructive environmental effects has been designed based on likely real-world scenario where the first effect has the highest likeliness of being occurred. But, as can be seen in Fig 5.1, AWGN effect is designed to be the last destructive effect although this effect is considered as first destructive ones in any environment. This is because authors in [31] have attempted to create a worst-case scenario. In such a manner, AWGN effect has to be applied to signal in last step.

From among the effects that the signal is exposed to, the selective fading model has the most destructive impact on higher-order modulation schemes, primarily due to the phase shift offset resulting from this model. Fig 5.2 shows the constellation of a received signal from this dataset for 128QAM and 256QAM at an SNR of 5 dB, both with and without the selective fading model applied. Therefore, in order to achieve higher classification accuracy, information on the properties impacted by the selective fading model, such as phase shift offset, should be extracted during the feature extraction stage, and provided to the labeling stage. This can be achieved through the use of polar coordinate transforms.

Furthermore, in order to provide intuitive understanding of how polar coordinates can provide another domain of information for labeling stage, we plot the output of this

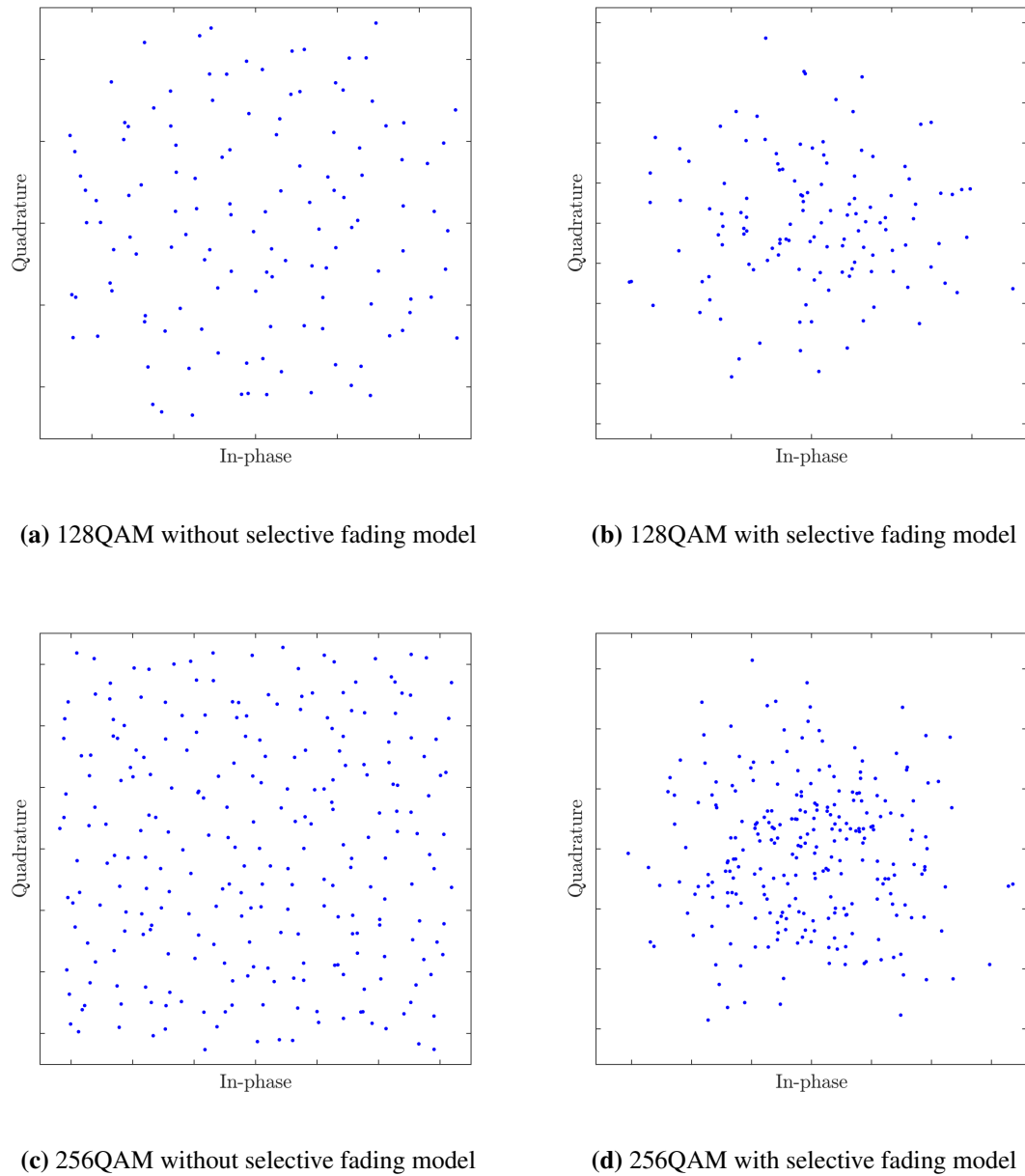


Figure 5.2: The destructive effect of the selective fading model over the constellation of 128QAM and 256QAM from RadioML dataset at SNR = 5 dB.

component in feature extraction stage. Moreover, we also investigate the effect of selective fading model over polar coordinates. Fig 5.3 shows the polar-based constellation of a received signal from this dataset for 128QAM and 256QAM at an SNR of 5 dB, both with and without the selective fading model applied.

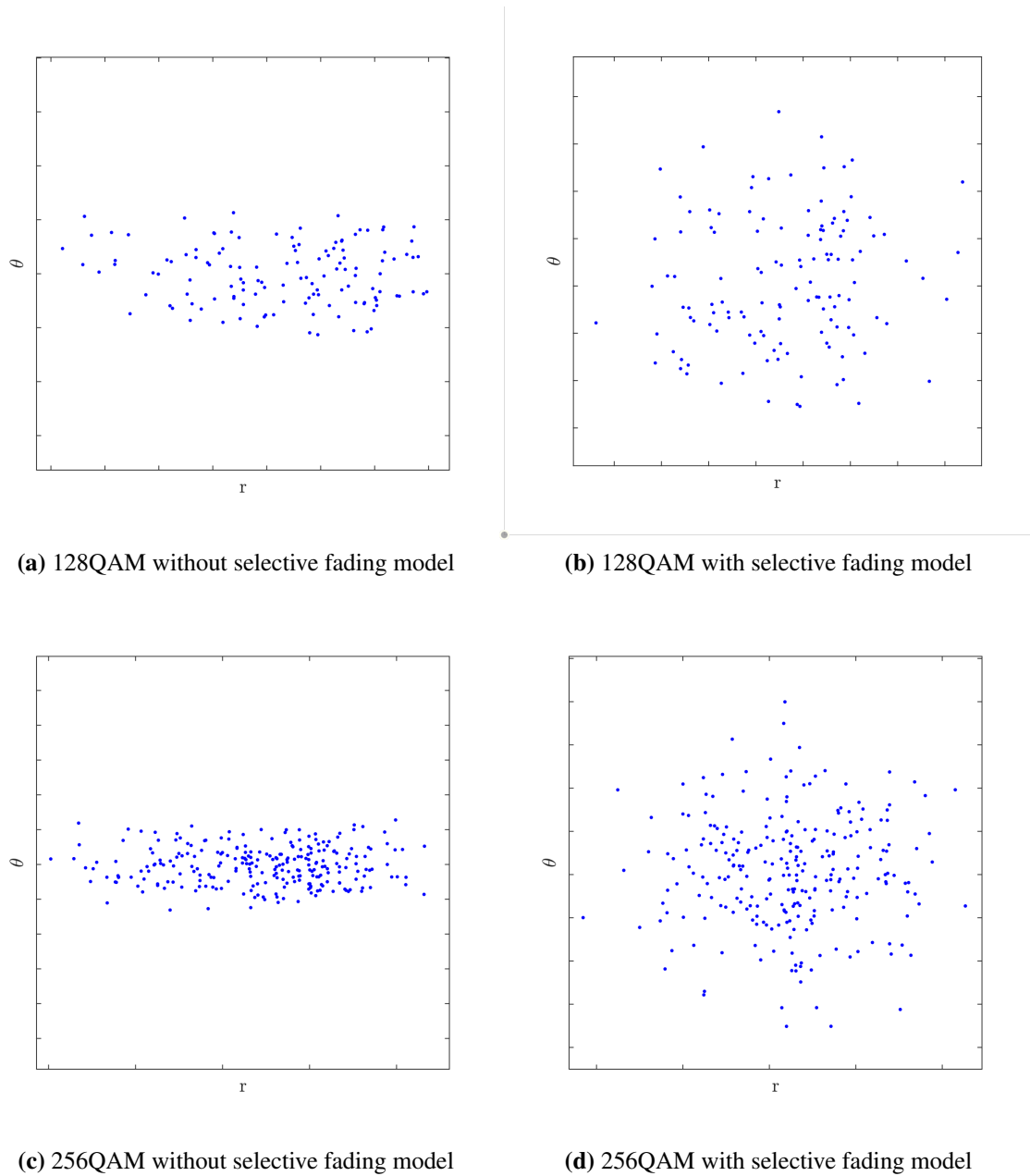


Figure 5.3: Polar-based constellation of 128QAM and 256QAM from RadioML dataset at SNR = 5 dB with and without destructive effect of the selective fading model.

After the RadioML dataset has gone through the new architecture of feature extraction stage, the operation results in *3D high-order statistical polar-based* dataset, for which Table 5.2 shows the corresponding dimensions.

Table 5.2: 3D high-order statistical polar-based dataset dimensions.

Data Type	Dimensions
New Dataset	$4 \times 1025 \times 2,555,904$
Training Dataset	$4 \times 1025 \times 1,277,952$
Validation Dataset	$4 \times 1025 \times 255,590$
Testing Dataset	$4 \times 1025 \times 1,022,361$

5.2 DBN-based FB AMC Classifier Analysis

In order to start analyzing the performance of this classifier, we first need to build its architecture including setting its hyperparameters.

5.2.1 DBN Architecture and Employment

Designing the architecture of DBN and deployment involves setting its hyperparameters and defining the environment in which it will be simulated.

The input and output layers of our DBN will, respectively, include 24 and 4 units to represent the 24 modulation schemes in the RadioML dataset, and 4 modulation schemes in this research’s evaluation. In order to achieve an optimized DBN model, we have selected the adaptive moment estimation (Adam) algorithm [34], [35]. This resulted in optimized hyperparameters and other configurations of DBN presented in Table 5.3.

Table 5.3: Optimized hyperparameters and configurations of DBN.

Hyperparameters	Values
Number of hidden layers	6
Number of units in each hidden layer	18
Activation function of the first 5 layers	ReLU
Activation function of the last layer	Softmax
Batch size	256
Number of maximum training epoch	250
Learning rate	0.0005
Number of times where contrastive divergence is run (k)	12

From an RBM architecture perspective, this implies that there are 3 visible and 3 shallow layers, each including 18 units. Our training, validating and testing environments are

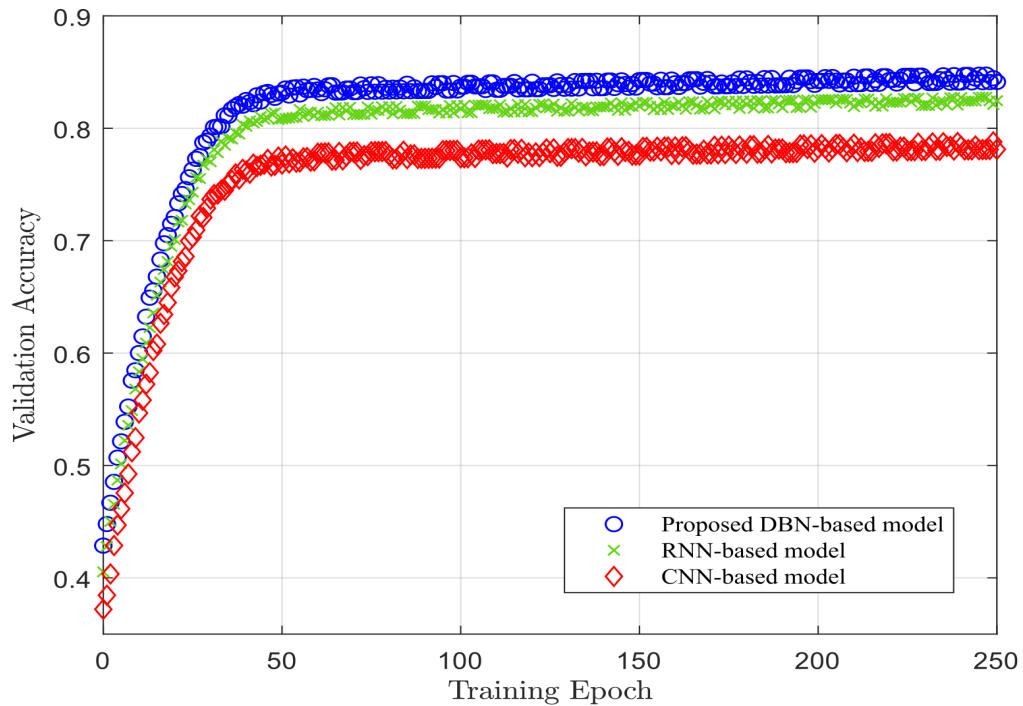


Figure 5.4: Validation accuracy of DBN-based model in training stage.

implemented using the deep learning library Keras running on top of TensorFlow executed in our university's supercomputing infrastructure, HCC Crane JupyterHub [36]. Fig 5.4 and Fig 5.5 respectively show the training performance of the presented DBN-based model in terms of the number of training epochs versus validation accuracy and training loss in addition to a comparison with deep RNN- and CNN-based models. To have a fair representation of validation accuracy and training loss, parameters such as batch size and learning rate are set to be the same for all three models.

The proposed DBN-based model shows higher rising and declining slopes in both evaluations. This points to the fact that DBN-based models are capable of a more accurate training process compared to the other two models. Next, we will present the AMC numerical results and discuss them.

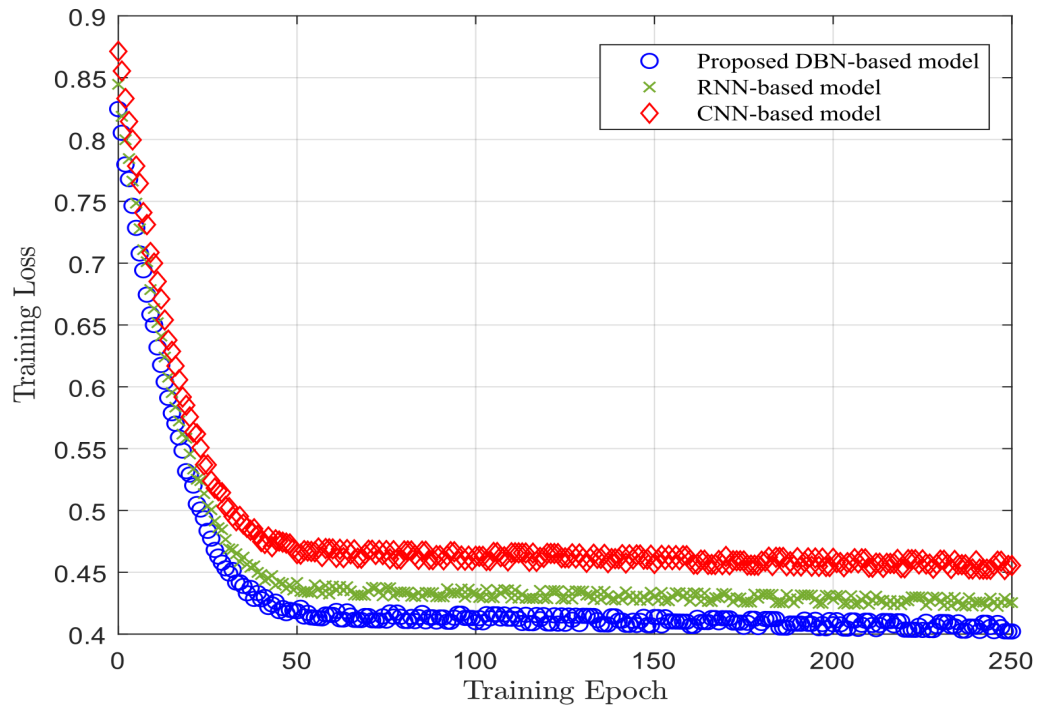


Figure 5.5: Training loss of DBN-based model in training stage.

5.2.2 AMC Results and Discussion

We herein present our results from evaluating the AMC classification performance using our framework, and obtain the lower and upper bounds of the proposed DBN-based model as well as a comparison with RNN- and CNN-based models. Due to the performance plateau of classifiers below -10 dB, we selected the range of SNR to be between -10 and +30 dB for the upper-bound performance analysis. We similarly selected the range of SNR between -10 and +10dB for the lower-bound performance evaluation. Fig 5.6 and Fig 5.7 in next subsections show the aforementioned performances for lower and upper bounds, respectively. Final results are achieved by interpolating between each two consecutive available SNR's probability of correct classification (P_{CC}). The results show the proposed DBN-based model performance compared to RNN- and CNN-based models to obtain lower- and upper-bounds of performance when {BPSK, QPSK} and {128QAM, 256QAM} are classified, respectively.

We will separately discuss the numerical results for the lower- and upper-bounds performance.

5.2.2.1 Lower-bound Discussion

As can be observed from Fig ??, where modulations {BPSK and QPSK} are classified, the proposed DBN-based model outperforms the other two models over the entire range of SNR. This higher performance is notably observable in lower SNRs.

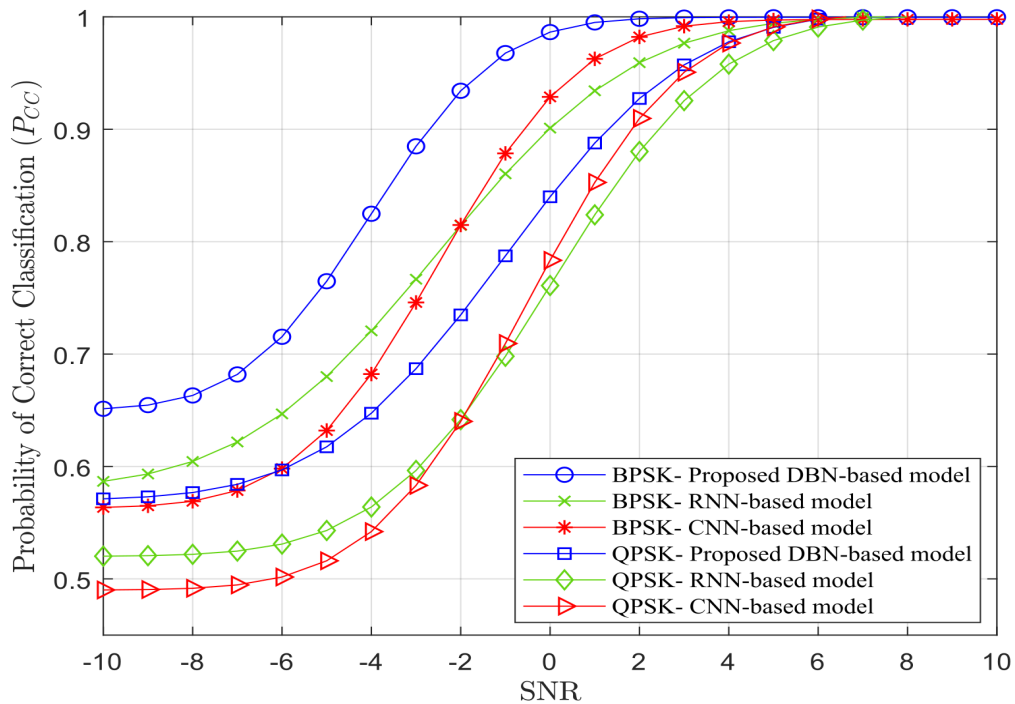


Figure 5.6: Proposed DBN-based lower-bound performance compared to RNN and CNN

The performance of the proposed DBN-based model is on average 21.8% and 16.2% higher than the average performance of the other two models when classifying BPSK and QPSK, respectively. It also should be noted that there is a performance advantage of CNN for lower SNRs over that of RNN. However, the RNN-based model performs better than the CNN-based model in SNRs below -3 dB. That is due to CNN operating based on the constellation shape of the intercepted signal's modulation scheme. And a signal's

constellation shape becomes less apparent as the SNR value decreases.

5.2.2.2 Upper-bound Discussion

Similarly to the discussion of the lower-bound performance, we can observe in Fig 5.7 that the DBN-based model shows a higher capability for correct classification over the entire range of SNR, especially in lower SNR cases. The proposed DBN-based model performs 14.7% and 11.4% on average better than RNN and CNN when 128QAM and 256QAM are classified, respectively.

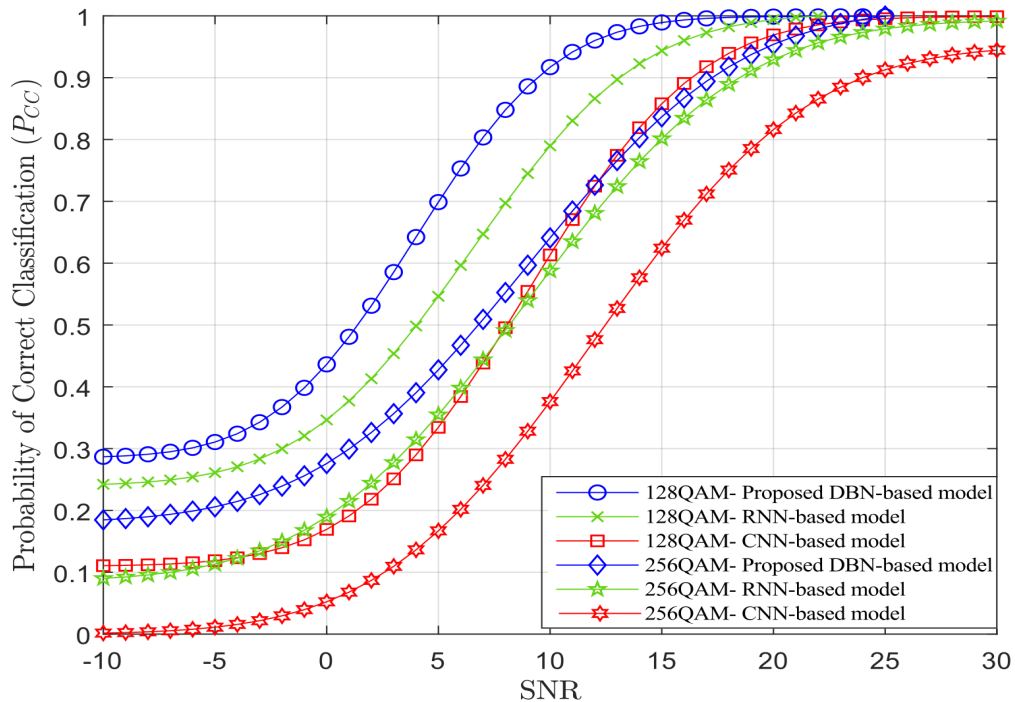


Figure 5.7: Proposed DBN-based upper-bound performance compared to RNN and CNN

As can be seen, while classifying 256QAM, the performance of the proposed DBN- and RNN-based models tends to merge at higher SNRs since the training process of RNN-based model becomes more accurate as SNR increases. This even can be seen in classifying QPSK for the lower-bound performance. Since the CNN-based model depends on the constellation shape to extract the signal's information, its performance generally is degraded as the order

of modulation scheme increases. This degradation in performance is easily noticeable from comparing lower- and upper-bound performances.

Another aspect of DBN performance is that the set of labelled data can be small relative to the original dataset, which is extremely helpful in real-world applications where low latency is of importance [37].

5.2.2.3 Number of Training Samples Discussion

One of the important factors to consider in the AMC domain is how to train the classifier as quickly as possible, and subsequently test it in real-time. One way to accomplish this task is to reduce the sample size while not sacrificing the classification accuracy. DBNs are capable of achieving this goal [38]. Therefore, we herein investigate this property and compare it to the other two models when classifying 32QAM modulation scheme in two scenarios where 1) the full size of and 2) half of the *3D high-order polar-based* dataset is to be used for training, validation and testing with the same aforementioned proportionality. As

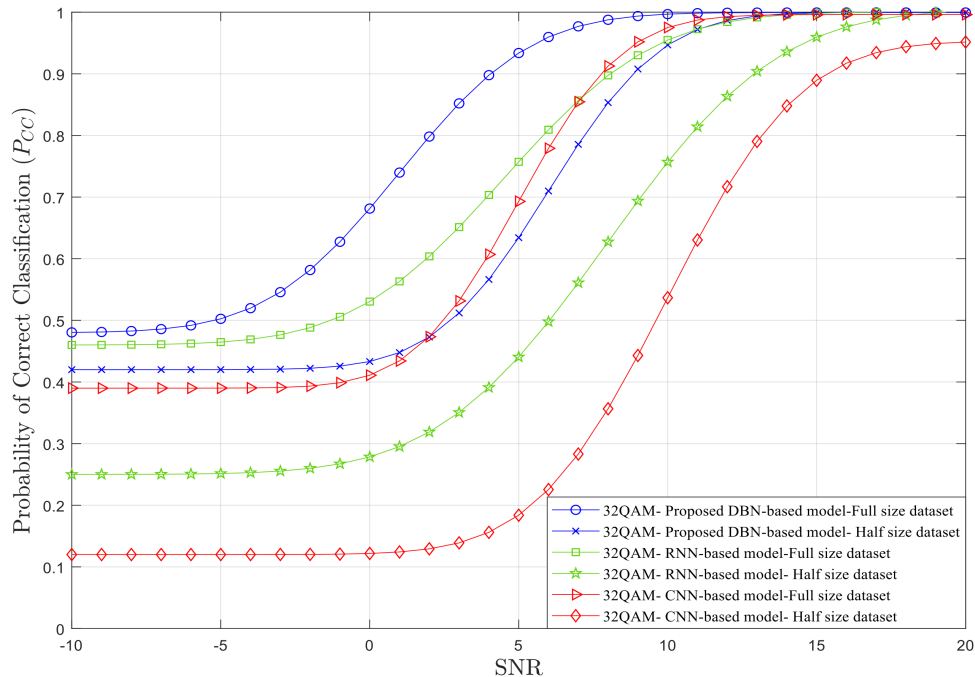


Figure 5.8: Number of training sample impact on classification accuracy.

can be seen from Fig 5.8, when the size of the input sample is reduced to half, the average performance of the proposed DBN-based model drops much less than that of the other two models. At lower SNRs, the proposed DBN-based model's performance trained with half the size of the dataset is even higher than the CNN-based model when trained with the full dataset. Additionally, over higher SNRs, the proposed DBN-based model trained on half of the dataset achieves maximum classification accuracy at approximately the same SNR than the RNN-based model after utilizing the full dataset training size. The main reason behind this key benefit is DBN's capability of probabilistic reconstruction of the input samples, as explained earlier. Additionally, DBN platforms are more resilient than other platforms against the overfitting problem [39].

5.2.3 Computational Complexity Analysis

Computational complexity of any AMC classifier is of high importance in order to measure its capability of operating real-time [40]. Hence, time has been traditionally a metric to measure an AMC classifier's computational complexity. Thus, to analyze computational complexity of the proposed DBN-based model, we take into consideration the time required to perform the classification on this model, and compare it with the other two models in the simulation environment mentioned before. But, in order to exclude the effect of computing power of simulation platform, we define a parameter τ representing the timing unit in this evaluation. Note that τ is scalable based on the simulation platform's computing power. Table 5.4 shows details of this analysis.

Table 5.4: Computational complexity of the proposed DBN-based model.

	Proposed DBN-based model	RNN-based model	CNN-based model
Each batch	0.0022 τ	0.0021 τ	0.0020 τ
Each epoch	22.4716 τ	21.826 τ	20.6637 τ
Total classification	5056.11 τ	4910.86 τ	4649.35 τ

As can be observed from results, the computational complexity of the proposed DBN-

based model is slightly higher than the RNN-based model, by 2.9%, although the average classification accuracy of the proposed DBN-based model is notably higher than the other two models, especially at low SNRs by on average 16.02%. Therefore, we can state in general that the achievable significant increase in classification accuracy easily offsets the slight increase in computational complexity.

5.2.4 Model Conclusion

The proposed DBN-based model increases the average classification accuracy for lower-bound evaluation by 19%, and 13.05% for higher-bound evaluation. This results in a total average increase of 16.02% over the four modulation schemes chosen for this evaluation. This significant increase can be seen over all ranges of SNR, but especially so for lower SNR cases. As a result, the slight increase in computational complexity by 2.9% compared to RNN-based model is acceptable and often negligible. But the proportionality of this increase in classification accuracy compared to that of the increase in computational complexity (5.52) shows the efficiency of this proposed model to be employed for lower SNR cases.

5.3 SNN-based FB AMC Classifier Analysis

Similarly to DBN-based classifier's analysis, we first build the architecture of SNN-based model and then proceed to analyze it.

In order to have a fair evaluation of a DBN-based model, we follow the architecture of the DBN-based model with minimal modifications specifically needed to accommodate SNN, including its training process.

The input and output layers will respectively include 24 and 4 neurons based on the same reasoning as in the previous section. The training algorithm for SNN is selected to be Hebbian learning, which strongly involves the spike-timing-dependent plasticity (STDP) rule in unsupervised learning [41]. Hence, we need to not only set typical hyperparameters

of a neural network, but we are also required to predetermine other parameters regarding Hebbian learning [42]. We set SNN’s typical hyperparameters as shown for the proposed DBN-based model, to once again ensure a fair comparison. Table 5.5 indicates these hyperparameters.

Table 5.5: SNN hyperparameters’ architecture.

Hyperparameters	Values
Number of hidden layers	6
Number of units in each hidden layer	18
Activation function of the last layer	Softmax
Batch size	256
Number of maximum training epoch	250
Learning rate	0.0005
Discharge	0.1
Long term potentiation (LTP)	1.5
inhLTP	1.5
Long term depression (LTD)	0.1

Here, inhLTP represents the fact that if a pre-synaptic neuron is inhibitory, the weight always increases if the pair of neurons fire within a certain time-window, irrespective of the order of firing (inhLTP) [41]–[43]. Building an SNN model as FCN in the proposed deep learning architecture is shown in Fig 5.9 and Fig 5.10 in terms of training validation and loss process measurements, respectively.

The training process measurements mainly include the performance of the number of training epochs versus validation accuracy and training loss. Both Fig 5.9 and Fig 5.10 also include the other three models’ performances for comparison purposes. As can be seen from Fig 5.9 and Fig 5.10, SNN’s training performance is notably similar to that of the RNN-based model, even overlapping in some epochs, but it is always lower than the proposed DBN-based model and always higher than the CNN-based model. This implies that although SNN benefits from eliminating neurons having no apparent impact on classification from computations [44], classification accuracy suffers as a result of that same property [45]. The simulation environment is the same as explained in the previous section [36], [46].

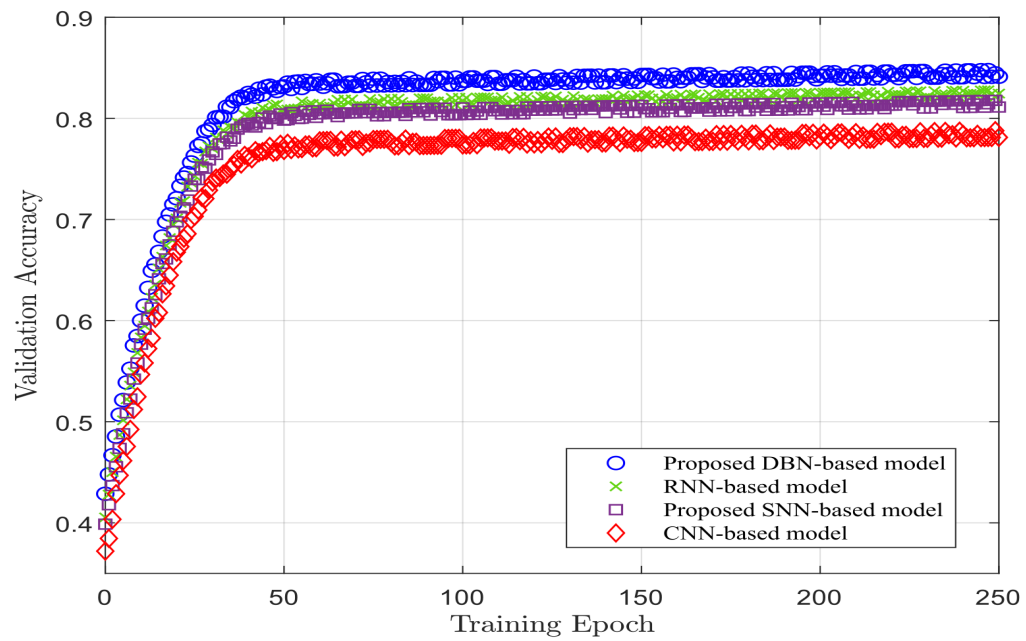


Figure 5.9: Proposed SNN-based model performance of number of training epochs versus validation accuracy.

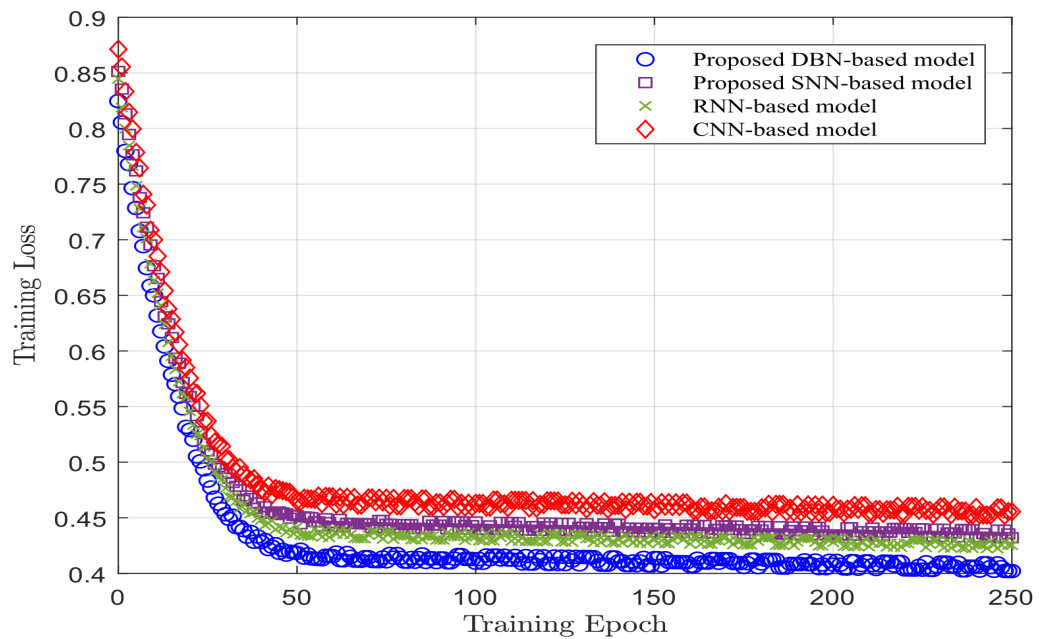


Figure 5.10: Proposed SNN-based model performance of number of training epochs versus training loss.

5.3.1 Results and Discussion

Implementing SNN as FCN in the proposed deep learning architecture results in Fig 5.11 and Fig 5.12, in order to obtain the lower- and upper-bounds performance of the proposed SNN-based model where {BPSK and QPSK} and {128QAM and 256QAM} are respectively classified.

Similar to the proposed DBN-based model discussion, we separately investigate the upper and lower performance bounds for the proposed SNN-based model.

5.3.1.1 Lower-bound performance

Observing Fig 5.11 shows that the proposed SNN-based model significantly outperforms the CNN-based model by an average of 17.1% and 9.8% when classifying 128QAM and 256QAM, respectively.

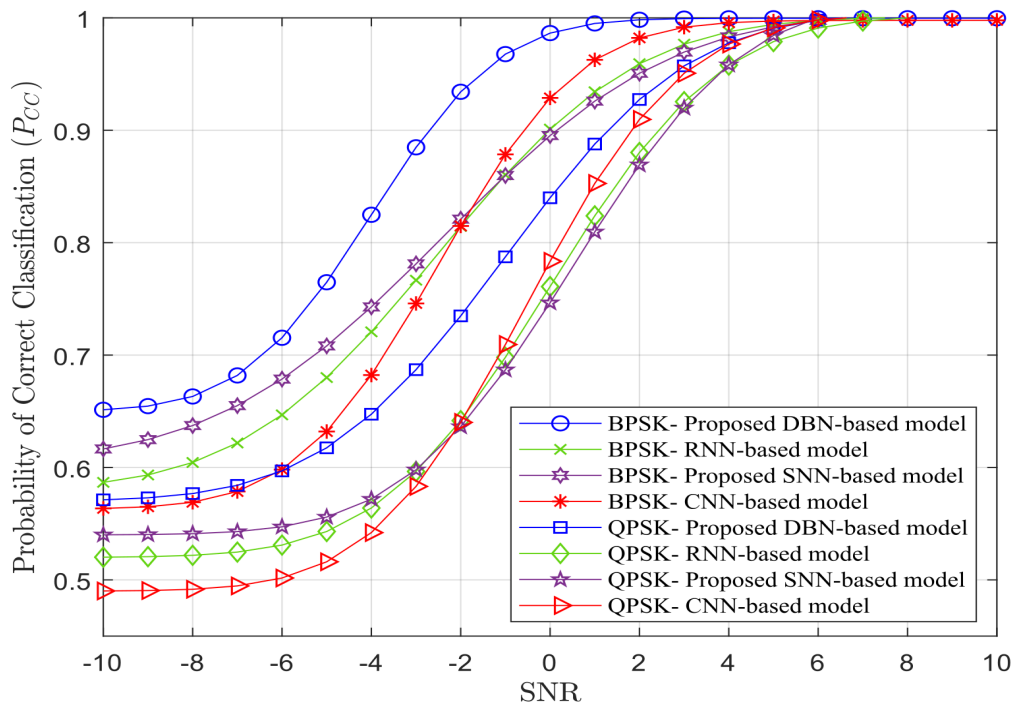


Figure 5.11: Proposed SNN-based model lower-bound performance compared to proposed DBN-, RNN- and CNN-based models.

On the other hand, the proposed SNN-based model also exhibits modestly higher

performance compared to the RNN-based model at higher SNRs. Overall, it can be stated that the proposed SNN-based model performs on average 14.3% and 7.8% higher than CNN- and RNN-based models when classifying 128QAM and 256QAM, respectively.

5.3.1.2 Upper-bound Performance

As can be seen from Fig 5.12, the proposed SNN-based model outperforms the RNN- and CNN-based models at SNRs lower than -2 dB by 9.2%.

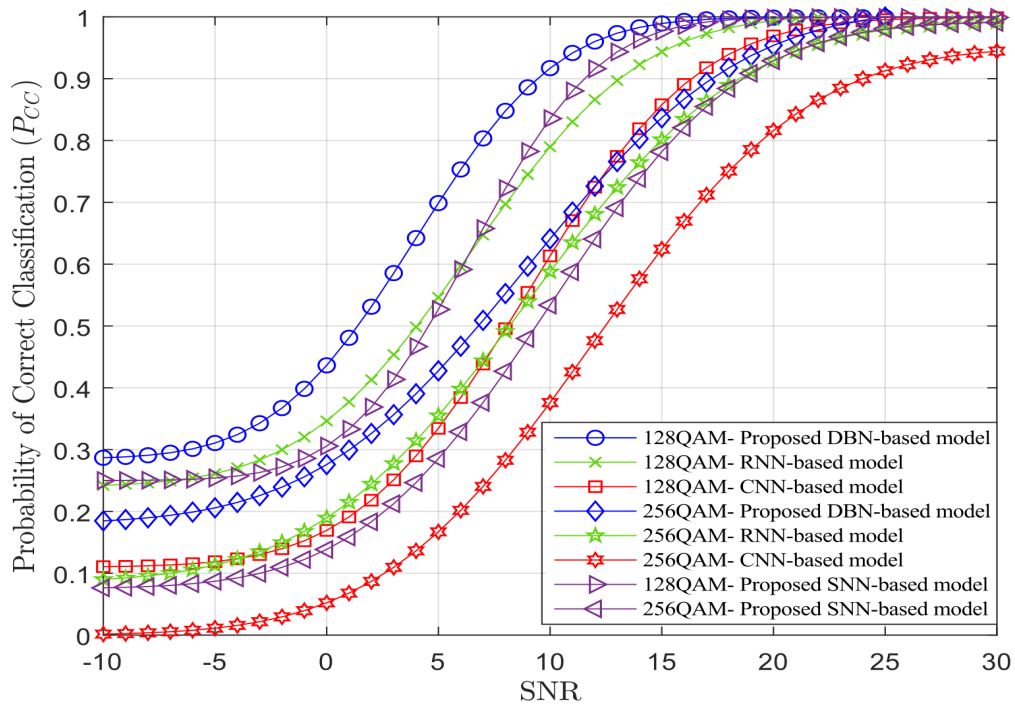


Figure 5.12: Proposed SNN-based model upper-bound performance compared to proposed DBN-, RNN- and CNN-based models.

This is a consequence of the neurons' behavior of not firing when they have no constructive effect on the classification [47]. This can be seen in lower SNR cases, where involving misled neurons can degrade the classification process. On the other hand, the training process of the proposed DBN-based model is significantly more accurate and outperforms the proposed SNN-based model over the entire SNR range. Overall, the proposed SNN-based model performs on average 5.7% and 2.9% higher than CNN- and RNN-based models over

the entire SNR range. However, the CNN-based model has a higher performance at SNRs above -2 dB due to the higher orderly constellation shape of the received signal.

5.3.2 Computational Complexity Analysis

Similar to the analysis provided in the previous section, we now investigate the computational complexity of the proposed SNN-based model, based on analyzing the timing unit required to achieve classification. For this model, Table 5.6 indicates the obtained computational complexity measurements.

Table 5.6: Proposed SNN-based model computational complexity measurement.

Proposed SNN-based model's architecture parameters	Elapsed timing units
Each batch	0.00158 τ
Each epoch	15.84 τ
Total classification	3565.32 τ

This analysis clearly shows a significant decrease in the required number of timing units to train, validate and classify for the SNN-based model. The number of timing units required by the proposed SNN-model is 1084τ lower than that of the CNN-based model, which has the lowest computational complexity among the proposed DBN- and RNN-based models. This implies that the proposed SNN-based model is 23.31% faster than CNN-based model. Additionally, the proposed SNN-based model performs the AMC operation 41.81% and 34.31% faster when compared to the proposed DBN- and RNN-based models.

5.3.3 Model Conclusion

Although the main purpose behind the proposal to use SNN-based models was to reduce computational complexity, its classification accuracy performance in obtaining both lower and upper bounds indicates that it is very competitive compared to the average classification accuracy of the other models investigated during our research. The proposed SNN-based

model reduces computational complexity by an average of 33.14% compared to the three other models, while it outperforms CNN- and RNN-based models by on average 4.3% and 11% in obtaining lower and upper bounds, respectively.

5.4 Proposed Novel Framework Analysis

Applying the presented ratio for framework (4.54) provides us with the answer to which ML approach to use for different channel conditions, as shown in Table 5.7. SNN is clearly the

Table 5.7: Tradeoff-driven model selection for classification at each SNR.

SNR dB	Model	SNR dB	Model
-20	DBN	6	SNN
-18	DBN	8	SNN
-16	DBN	10	SNN
-14	DBN	12	SNN
-12	DBN	14	SNN
-10	DBN	16	SNN
-8	DBN	18	SNN
-6	DBN	20	SNN
-4	DBN	22	SNN
-2	SNN	24	SNN
0	SNN	26	SNN
2	SNN	28	SNN
4	SNN	30	SNN

preferred choice over a significant portion of the evaluated SNR range. This would indicate that SNN is also the overall better choice for non-adaptive AMC implementation. However, having an FB-AMC classifier that can adapt according to Table 5.7 reduces computational complexity by 39.2% compared to the case where the proposed DBN-based model is utilized for classification over all SNR ranges. Additionally, such a classifier only sacrifices on average 5.8% in classification accuracy for SNRs lower than -2 dB.

CHAPTER 6

Conclusion and Future Work

In this thesis, in the area of automatic modulation classification, we achieved significant performance improvements in both stages of the feature-based AMC approach. We contributed to the feature extraction stage by designing a new architecture that enables the retrieval of unbiased fourth-order cumulants, and leverages polar coordinates of the intercepted signal for improved accuracy. This resulted in a new *3D high-order statistical polar-based* dataset to be used in the labeling stage. Here, through a deep learning architecture design, we introduced the platforms of deep belief network and spiking neural network in order to increase classification accuracy, specifically at lower SNR values, and decrease computational complexity of the classifier, respectively. In our evaluation, we obtained the lower- and upper-bounds performance of the proposed classifiers while classifying the lowest- (BPSK, QPSK) and highest-order modulation schemes (128QAM, 256QAM) utilizing the signals from the RadioML dataset. We not only provided numerical results of the proposed classifiers, but we also compared them with two recent modified classifiers based on convolutional- and recurrent-based neural networks. The comparison showed a significant increase in classification accuracy by an average of 16.02% when a deep belief network was employed. Specifically, for lower SNR values, we obtained both lower-bound performance improvements of 19% and upper-bound improvements of 13.05%. Additionally, we showed that the spiking neural network's performance remained

competitive compared to recurrent-based neural networks while performing classification on average 34.31% faster. We showed that the proposed spiking neural network achieved notable higher classification accuracy compared to the convolutional-based neural network while also performing AMC 23.31% faster. Finally, we developed an efficient and flexible framework based on the proposed platforms' characteristics, computational complexity and classification accuracy. It can adapt between these ML approaches based on the currently observed SNR obtained from the receiver's equalizer component. This framework achieved a 36.2% higher efficiency in terms of required computational cost while sacrificing only 5.8% in classification accuracy for SNRs lower than -2 dB.

Our future research direction involves both DBN and SNN models. For DBN platform, it is necessary to decrease the computational complexity of the DBN-based model without sacrificing notable classification accuracy. This requires modification in inherent design of DBN-based model to be specifically adapted for AMC applications. In this manner, we will be able to decrease the computational complexity of DBN-based model while classification accuracy of such platform does not decrease as much as computational complexity. The very optimistic scenario occurs when we decrease the DBN-based model's computational complexity without sacrificing any of classification accuracy. We can have such a same direction for SNN-based model.

For SNN platform, the goal should be to increase the classification accuracy without increasing the computational complexity. To do this task again, we need to apply some modification in inherent design of SNN platform to specially adjust its operation for AMC applications. In this direction, the very optimistic path is to increase the classification accuracy while computational complexity is decreased.

Bibliography

- [1] P. Ghasemzadeh, S. Banerjee, M. Hempel, and H. Sharif, "Performance evaluation of feature-based automatic modulation classification", in *2018 12th International Conference on Signal Processing and Communication Systems (ICSPCS)*, IEEE, 2018, pp. 1–5.
- [2] P. Ghasemzadeh, S. Banerjee, and M. Hempel, "Accuracy analysis of feature-based automatic modulation classification with blind modulation detection", in *2019 International Conference on Computing, Networking and Communications (ICNC)*, IEEE, 2019, pp. 1000–1004.
- [3] P. Ghasemzadeh, S. Banerjee, M. Hempel, M. Alahmad, and H. Sharif, "Analysis of distribution test-based and feature-based approaches toward automatic modulation classification", in *2019 IEEE 30th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, IEEE, 2019, pp. 1–6.
- [4] S. Banerjee, M. Hempel, P. Ghasemzadeh, Y. Qian, and H. Sharif, "A novel approach to social-behavioral d2d trust associations using self-propelled voronoi", in *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, IEEE, 2019, pp. 1–5.
- [5] S. Banerjee, J. Santos, M. Hempel, P. Ghasemzadeh, and H. Sharif, "A novel method of near-miss event detection with software defined radar in improving railyard safety", *Safety*, vol. 5, no. 3, p. 55, 2019.
- [6] S. Banerjee, M. Hempel, N. Albakay, P. Ghasemzadeh, and H. Sharif, "A framework for high-speed passenger train wireless network radio evaluations", in *2019 Joint Rail Conference*, American Society of Mechanical Engineers Digital Collection, 2019.
- [7] S. Banerjee, M. Hempel, P. Ghasemzadeh, N. Albakay, and H. Sharif, "High speed train wireless communication: Handover performance analysis for different radio access technologies", in *2019 Joint Rail Conference*, American Society of Mechanical Engineers Digital Collection, 2019.
- [8] S. Banerjee, M. Hempel, P. Ghasemzadeh, and H. Sharif, "A novel biomimicry-based analysis of d2d user association retention for achieving maximal throughput", in *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, IEEE, 2019, pp. 2036–2042.
- [9] S. Banerjee, P. Ghasemzadeh, M. Hempel, and H. Sharif, "Topography relaxation in determining unsafe state intersections for uncertain cps", *IEEE Sensors Letters*, vol. 4, no. 4, pp. 1–4, 2020.

- [10] P. Ghasemzadeh, S. Banerjee, M. Hempel, A. Harms, and H. Sharif, "Detecting dark cars using a novel multi-antenna aei tag reader design for increased read distance and reliability", in *2020 Joint Rail Conference*, American Society of Mechanical Engineers Digital Collection, 2020.
- [11] P. Ghasemzadeh, S. Banerjee, M. Hempel, H. Sharif, and T. Omar, "Evaluation of machine learning-driven automatic modulation classifiers under various signal models", in *2020 Joint Rail Conference*, American Society of Mechanical Engineers Digital Collection, 2020.
- [12] P. Ghasemzadeh, S. Banerjee, M. Hempel, and H. Sharif, "A new framework for automatic modulation classification using deep belief networks", in *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, IEEE, 2020.
- [13] P. Ghasemzadeh, S. Banerjee, M. Hempel, A. Harms, and H. Sharif, "Detecting dark cars in railroad operations using multi-antenna beamforming for long-distance discovery and identification of aei tags", in *2020 16th International Wireless Communications & Mobile Computing Conference (IWCMC)*, IEEE, 2020.
- [14] P. Ghasemzadeh, S. Banerjee, M. Hempel, and H. Sharif, "A novel deep learning and polar transformation framework for an adaptive automatic modulation classification", *IEEE Transactions on Vehicular Technology*, 2020.
- [15] S. Banerjee, M. Hempel, P. Ghasemzadeh, H. Sharif, and T. Omar, "Wireless communication for high-speed passenger rail services: A study on the design and evaluation of a unified architecture", in *2020 Joint Rail Conference*, American Society of Mechanical Engineers Digital Collection, 2020.
- [16] Y. Wang, M. Liu, J. Yang, and G. Gui, "Data-driven deep learning for automatic modulation recognition in cognitive radios", *IEEE Transactions on Vehicular Technology*, vol. 68, no. 4, pp. 4074–4077, 2019.
- [17] F. Meng, P. Chen, L. Wu, and X. Wang, "Automatic modulation classification: A deep learning enabled approach", *IEEE Transactions on Vehicular Technology*, vol. 67, no. 11, pp. 10 760–10 772, 2018.
- [18] C.-F. Teng, C.-C. Liao, C.-H. Chen, and A.-Y. A. Wu, "Polar feature based deep architectures for automatic modulation classification considering channel fading", in *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, IEEE, 2018, pp. 554–558.
- [19] Z. Xing and Y. Gao, "A modulation classification algorithm for multipath signals based on cepstrum", *IEEE Transactions on Instrumentation and Measurement*, 2019.
- [20] S. Peng, H. Jiang, H. Wang, H. Alwageed, Y. Zhou, M. M. Sebdani, and Y.-D. Yao, "Modulation classification based on signal constellation diagrams and deep learning", *IEEE transactions on neural networks and learning systems*, vol. 30, no. 3, pp. 718–727, 2018.

- [21] S. Majhi, R. Gupta, W. Xiang, and S. Glisic, “Hierarchical hypothesis and feature-based blind modulation classification for linearly modulated signals”, *IEEE Transactions on Vehicular Technology*, vol. 66, no. 12, pp. 11 057–11 069, 2017.
- [22] S. Rajendran, W. Meert, D. Giustiniano, V. Lenders, and S. Pollin, “Deep learning models for wireless signal classification with distributed low-cost spectrum sensors”, *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 3, pp. 433–445, 2018.
- [23] Y. A. Eldemerdash, O. A. Dobre, O. Üreten, and T. Yensen, “A robust modulation classification method for psk signals using random graphs”, *IEEE Transactions on Instrumentation and Measurement*, vol. 68, no. 2, pp. 642–644, 2018.
- [24] M. W. Aslam, Z. Zhu, and A. K. Nandi, “Automatic modulation classification using combination of genetic programming and knn”, *IEEE Transactions on wireless communications*, vol. 11, no. 8, pp. 2742–2750, 2012.
- [25] L. Zhou and H. Man, “Distributed automatic modulation classification based on cyclic feature via compressive sensing”, in *MILCOM 2013-2013 IEEE Military Communications Conference*, IEEE, 2013, pp. 40–45.
- [26] O. A. Dobre, A. Abdi, Y. Bar-Ness, and W. Su, “Cyclostationarity-based modulation classification of linear digital modulations in flat fading channels”, *Wireless Personal Communications*, vol. 54, no. 4, pp. 699–717, 2010.
- [27] S. Huang, Y. Yao, Z. Wei, Z. Feng, and P. Zhang, “Automatic modulation classification of overlapped sources using multiple cumulants”, *IEEE Transactions on Vehicular Technology*, vol. 66, no. 7, pp. 6089–6101, 2016.
- [28] L. Fei, G. Xiaoguang, and W. Kaifang, “Training restricted boltzmann machine using gradient fixing based algorithm”, *Chinese Journal of Electronics*, vol. 27, no. 4, pp. 694–703, 2018.
- [29] H. Yi, S. Shiyu, D. Xiusheng, and C. Zhigang, “A study on deep neural networks framework”, in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, IEEE, 2016, pp. 1519–1522.
- [30] Y. Hua, J. Guo, and H. Zhao, “Deep belief networks and deep learning”, in *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things*, IEEE, 2015, pp. 1–4.
- [31] T. J. O’Shea, T. Roy, and T. C. Clancy, “Over-the-air deep learning based radio signal classification”, *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168–179, 2018.
- [32] F. Meng, P. Chen, L. Wu, and X. Wang, “Automatic modulation classification: A deep learning enabled approach”, *IEEE Transactions on Vehicular Technology*, vol. 67, no. 11, pp. 10 760–10 772, 2018.
- [33] S. Hu, Y. Pei, P. P. Liang, and Y.-C. Liang, “Deep neural network for robust modulation classification under uncertain noise conditions”, *IEEE Transactions on Vehicular Technology*, 2019.

- [34] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization”, *arXiv preprint arXiv:1412.6980*, 2014.
- [35] A. S. Prabowo, A. Sihabuddin, and S. Azhari, “Adaptive moment estimation on deep belief network for rupiah currency forecasting”, *IJCCS (Indonesian Journal of Computing and Cybernetics Systems)*, vol. 13, no. 1, pp. 31–42, 2019.
- [36] *Holand computing center at university of nebraska-lincoln*, <https://hcc.unl.edu/>.
- [37] S. N. Tran and A. S. d. Garcez, “Deep logic networks: Inserting and extracting knowledge from deep belief networks”, *IEEE transactions on neural networks and learning systems*, vol. 29, no. 2, pp. 246–258, 2016.
- [38] M. Ma, X. Xu, J. Wu, and M. Guo, “Design and analyze the structure based on deep belief network for gesture recognition”, in *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, IEEE, 2018, pp. 40–44.
- [39] Y. Hua, J. Guo, and H. Zhao, “Deep belief networks and deep learning”, in *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things*, IEEE, 2015, pp. 1–4.
- [40] C.-F. Teng, C.-Y. Chou, C.-H. Chen, and A.-Y. Wu, “Accumulated feature based deep learning with channel compensation mechanism for efficient automatic modulation classification under time varying channels”, *arXiv preprint arXiv:2001.01395*, 2020.
- [41] K. Kozdon and P. Bentley, “The evolution of training parameters for spiking neural networks with hebbian learning”, in *Artificial Life Conference Proceedings*, MIT Press, 2018, pp. 276–283.
- [42] D. Neil, M. Pfeiffer, and S.-C. Liu, “Learning to be efficient: Algorithms for training low-latency, low-compute deep spiking neural networks”, in *Proceedings of the 31st annual ACM symposium on applied computing*, 2016, pp. 293–298.
- [43] M. M. S. Mustari, “Configuring spiking neural network training algorithms”, PhD thesis, Memorial University of Newfoundland, 2017.
- [44] W. Maass, “On the computational complexity of networks of spiking neurons”, in *Advances in neural information processing systems*, 1995, pp. 183–190.
- [45] M. Pfeiffer and T. Pfeil, “Deep learning with spiking neurons: Opportunities and challenges”, *Frontiers in neuroscience*, vol. 12, p. 774, 2018.
- [46] H. Hazan, D. J. Saunders, H. Khan, D. Patel, D. T. Sanghavi, H. T. Siegelmann, and R. Kozma, “Bindsnet: A machine learning-oriented spiking neural networks library in python”, *Frontiers in neuroinformatics*, vol. 12, p. 89, 2018.
- [47] J. M. Brader, W. Senn, and S. Fusi, “Learning real-world stimuli in a neural network with spike-driven synaptic dynamics”, *Neural computation*, vol. 19, no. 11, pp. 2881–2912, 2007.
- [48] *Bindsnet 0.2.7 python package to simulate spiking neural network*, <https://pypi.org/project/bindsnet/> or <https://github.com/BINDS-LAB-UMASS/bindsnet/tree/master/bindsnet>.

APPENDIX A

Deep Learning Models Participating in Comparing Results

We herein investigate two of the most studied ANN platforms, CNN and RNN, in the AMC domain. We built a comparison mechanism by modeling two of the most recently proposed deep CNN- and RNN-based models that significantly improve the performance of the FB-AMC classifier.

A.1 Deep CNN-based Model

The authors of [32] proposed an end-to-end trainable deep CNN-based model that is capable of automatically learning a signal’s features without going through feature extraction. They also proposed a two-step training approach that includes pre-training and fine-tuning steps for the proposed CNN-based model. Their model also divides the input data into unit sizes for parallel computation. Then, a maximum *a posteriori* probability (MAP) criterion classifies the modulation schemes. Eliminating the feature extraction stage and parallel computation simplified their system design, but they had to remove some decision processes to keep their system’s computational complexity low. Their system model also requires SNR to be known for the classifier to work. We implemented their system in order to compare this model with our proposed classifiers. The general view of their network architecture can be seen in Fig A.1.

The structure dimensions of this network are described in Table A.1 where ‘n.a.’ represents not-applied.

Table A.1: CNN-based model structure dimensions.

Layers	Kernel Number	Kernel Size	Stride	Window Size
Convolutional A	12	3	1	n.a.
Convolutional B	24	3	1	n.a.
Convolutional C	32	3	1	n.a.
AveragePool	n.a.	n.a.	1	2

The output vectors of the last AveragePool layer are compacted into a vector by the flattened layer. Then, the dense layer encodes this vector into a 256-dimensional vector that

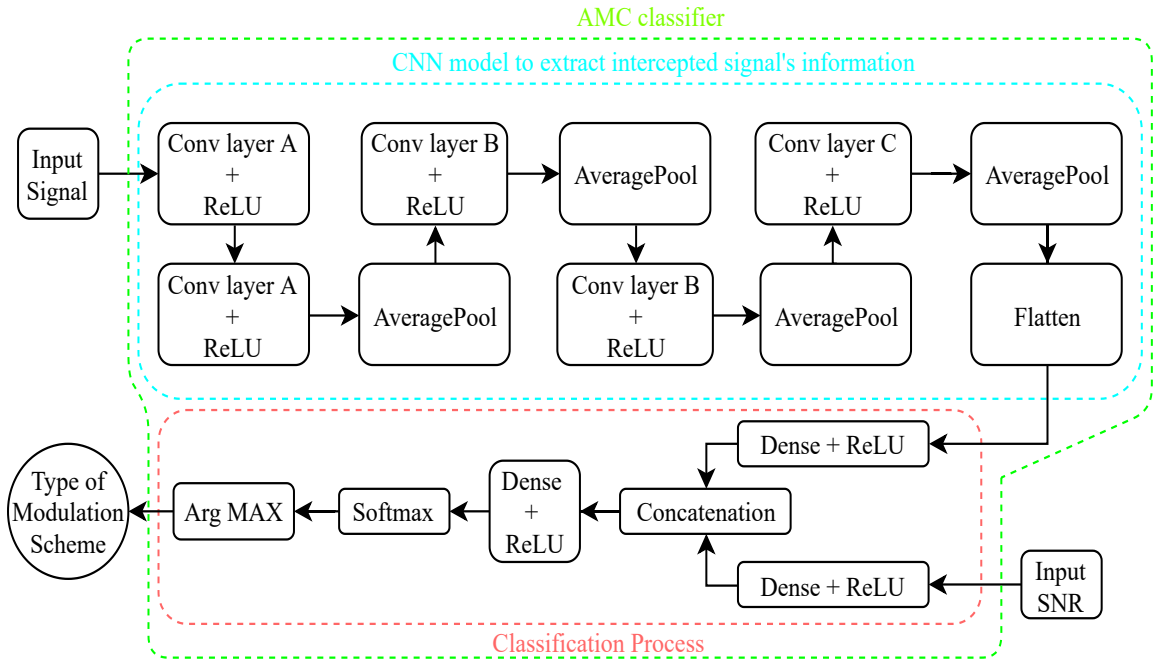


Figure A.1: The deep CNN-based classifier's structure.

carries high-level extracted information from the intercepted signal. Pre-training executes the following steps: 1) generating and sampling training data, 2) randomly setting system model parameters, and 3) training the model and storing the updated parameters with minimal validation loss. Fine-tuning steps are: 1) producing training data, 2) randomly initializing the top layer of system model after its replacement with updated parameters from pre-training, and finally 3) training the model and storing the updated parameters with minimal validation loss. More details of this model can be found in [32]. In order to fit this model with our experiment, we do not execute the first steps of pre-training and fine-tuning. Moreover, note that the input data dimension in the RadioML dataset is 2×1024 for the received symbols. On the other hand, their input data dimension is 2×1000 .

A.2 RNN-based Model

We also implemented the deep RNN-based system model proposed in [33]. In this model, a pre-processing stage is first used to re-order the structure of the received samples based on their phases. Then, a deep RNN-based model is built upon a long short-term memory (LSTM) network that can properly learn long-term dependencies of the received samples. Finally, a MAP criterion is used for modulation classification. The deep RNN-based model can be seen in Fig A.2.

This model consists of three stacked-LSTM layers that are later connected to a four-layer fully-connected network. In each layer of the fully-connected network, the number of units is set to 11 because of using of RadioML2016.10a, which contains a total of 11 modulated signals. Hence, in order to fit this model into our experiment, we needed to increase the

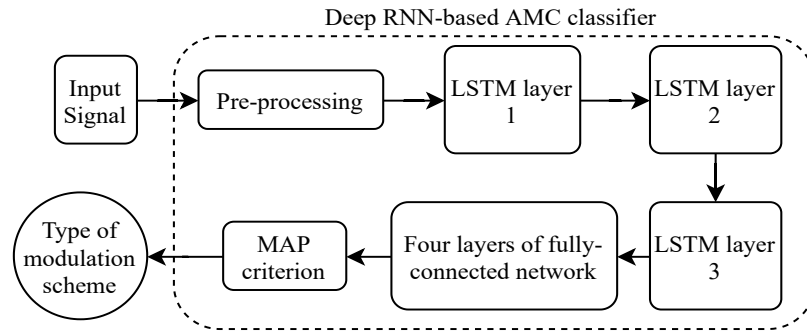


Figure A.2: The deep RNN-based classifier's structure.

number of units in the input layer to 24 and decrease the output layer's units to 4. All other steps remained the same as shown in [33], which also contains more details for the interested reader.

APPENDIX B

Spiking Neural Network-Based Platform Utilized in This Research

We herein provide the code written¹by author to create the environment and simulate the spiking neural network in Python language based on PyTorch simulator, especially adapted to AMC applications. We group the codes that relate to one another in following sections in order to operate.

B.1 Initialization and Refactored Conversion Module

```

1 from pathlib import Path
2
3 from . import (
4     utils,
5     network,
6     models,
7     analysis,
8     preprocessing,
9     datasets,
10    encoding,
11    pipeline,
12    learning,
13    evaluation,
14    environment,
15    conversion,
16 )
17
18 ROOT_DIR = Path(__file__).parents[0].parents[0]
```

Listing B.1: Initialization

```

1 import math
2 import torch
3 import numpy as np
```

¹The structure and flow of following SNN code were inspired and followed based on bindsnet 0.2.7 package [48].

```

4
5 from torch import Tensor
6 import torch.nn.functional as F
7 from numpy import ndarray
8 from typing import Tuple, Union
9 from torch.nn.modules.utils import _pair
10
11
12 def im2col_indices(
13     x: Tensor,
14     kernel_height: int,
15     kernel_width: int,
16     padding: Tuple[int, int] = (0, 0),
17     stride: Tuple[int, int] = (1, 1),
18 ) -> Tensor:
19     # language=rst
20     """
21     im2col is a special case of unfold which is implemented inside
22     of Pytorch.
23
24     :param x: Input image tensor to be reshaped to column-wise
25     format.
26     :param kernel_height: Height of the convolutional kernel in
27     pixels.
28     :param kernel_width: Width of the convolutional kernel in pixels
29     .
30     :param padding: Amount of zero padding on the input image.
31     :param stride: Amount to stride over image by per convolution.
32     :return: Input tensor reshaped to column-wise format.
33     """
34     return F.unfold(x, (kernel_height, kernel_width), padding=
35     padding, stride=stride)
36
37 def col2im_indices(
38     cols: Tensor,
39     x_shape: Tuple[int, int, int, int],
40     kernel_height: int,
41     kernel_width: int,
42     padding: Tuple[int, int] = (0, 0),
43     stride: Tuple[int, int] = (1, 1),
44 ) -> Tensor:
45     # language=rst
46     """
47     col2im is a special case of fold which is implemented inside of
48     Pytorch.
49
50     :param cols: Image tensor in column-wise format.
51     :param x_shape: Shape of original image tensor.
52     :param kernel_height: Height of the convolutional kernel in
53     pixels.
54     :param kernel_width: Width of the convolutional kernel in pixels
55     .
56     :param padding: Amount of zero padding on the input image.

```

```

50 :param stride: Amount to stride over image by per convolution.
51 :return: Image tensor in original image shape.
52 """
53 return F.fold(
54     cols, x_shape, (kernel_height, kernel_width), padding=
padding, stride=stride
55 )
56
57
58 def get_square_weights(
59     weights: Tensor, n_sqrt: int, side: Union[int, Tuple[int, int]]
60 ) -> Tensor:
61     # language=rst
62     """
63     Return a grid of a number of filters ‘‘sqrt ** 2’’ with side
lengths ‘‘side‘‘.
64
65     :param weights: Two-dimensional tensor of weights for two-
dimensional data.
66     :param n_sqrt: Square root of no. of filters.
67     :param side: Side length(s) of filter.
68     :return: Reshaped weights to square matrix of filters.
69     """
70     if isinstance(side, int):
71         side = (side, side)
72
73     square_weights = torch.zeros(side[0] * n_sqrt, side[1] * n_sqrt)
74     for i in range(n_sqrt):
75         for j in range(n_sqrt):
76             n = i * n_sqrt + j
77
78             if not n < weights.size(1):
79                 break
80
81             x = i * side[0]
82             y = (j % n_sqrt) * side[1]
83             filter_ = weights[:, n].contiguous().view(*side)
84             square_weights[x : x + side[0], y : y + side[1]] =
filter_
85
86     return square_weights
87
88
89 def get_square_assignments(assignments: Tensor, n_sqrt: int) ->
Tensor:
90     # language=rst
91     """
92     Return a grid of assignments.
93
94     :param assignments: Vector of integers corresponding to class
labels.
95     :param n_sqrt: Square root of no. of assignments.
96     :return: Reshaped square matrix of assignments.
97     """

```

```

98     square_assignments = torch.mul(torch.ones(n_sqrt, n_sqrt), -1.0)
99     for i in range(n_sqrt):
100         for j in range(n_sqrt):
101             n = i * n_sqrt + j
102
103             if not n < assignments.size(0):
104                 break
105
106             square_assignments[
107                 i : (i + 1), (j % n_sqrt) : ((j % n_sqrt) + 1)
108             ] = assignments[n]
109
110     return square_assignments
111
112
113 def reshape_locally_connected_weights(
114     w: Tensor,
115     n_filters: int,
116     kernel_size: Union[int, Tuple[int, int]],
117     conv_size: Union[int, Tuple[int, int]],
118     locations: Tensor,
119     input_sqrt: Union[int, Tuple[int, int]],
120 ) -> Tensor:
121     # language=rst
122     """
123     Get the weights from a locally connected layer and reshape them
124     to be two-dimensional and square.
125
126     :param w: Weights from a locally connected layer.
127     :param n_filters: No. of neuron filters.
128     :param kernel_size: Side length(s) of convolutional kernel.
129     :param conv_size: Side length(s) of convolution population.
130     :param locations: Binary mask indicating receptive fields of
131     convolution population neurons.
132     :param input_sqrt: Sides length(s) of input neurons.
133     :return: Locally connected weights reshaped as a collection of
134     spatially ordered square grids.
135     """
136     kernel_size = _pair(kernel_size)
137     conv_size = _pair(conv_size)
138     input_sqrt = _pair(input_sqrt)
139
140     k1, k2 = kernel_size
141     c1, c2 = conv_size
142     i1, i2 = input_sqrt
143     c1sqrt, c2sqrt = int(math.ceil(math.sqrt(c1))), int(math.ceil(
144     math.sqrt(c2)))
145     fs = int(math.ceil(math.sqrt(n_filters)))
146
147     w_ = torch.zeros((n_filters * k1, k2 * c1 * c2))
148
149     for n1 in range(c1):
150         for n2 in range(c2):
151             for feature in range(n_filters):

```

```

148         n = n1 * c2 + n2
149         filter_ = w[
150             locations[:, n],
151             feature * (c1 * c2) + (n // c2sqrt) * c2sqrt + (
n % c2sqrt),
152         ].view(k1, k2)
153         w_[feature * k1 : (feature + 1) * k1, n * k2 : (n +
1) * k2] = filter_
154
155     if c1 == 1 and c2 == 1:
156         square = torch.zeros((i1 * fs, i2 * fs))
157
158     for n in range(n_filters):
159         square[
160             (n // fs) * i1 : ((n // fs) + 1) * i2,
161             (n % fs) * i2 : ((n % fs) + 1) * i2,
162         ] = w_[n * i1 : (n + 1) * i2]
163
164     return square
165 else:
166     square = torch.zeros((k1 * fs * c1, k2 * fs * c2))
167
168     for n1 in range(c1):
169         for n2 in range(c2):
170             for f1 in range(fs):
171                 for f2 in range(fs):
172                     if f1 * fs + f2 < n_filters:
173                         square[
174 + f1 + 1),
175                             k2 * (n2 * fs + f2) : k2 * (n2 * fs
+ f2 + 1),
176                             ] = w_[
177 + 1) * k1,
178                             (n1 * c2 + n2) * k2 : (n1 * c2 + n2
+ 1) * k2,
179                             ]
180
181     return square
182
183
184 def reshape_conv2d_weights(weights: torch.Tensor) -> torch.Tensor:
185     # language=rst
186     """
187     Flattens a connection weight matrix of a Conv2dConnection
188
189     :param weights: Weight matrix of Conv2dConnection object.
190     :param wmin: Minimum allowed weight value.
191     :param wmax: Maximum allowed weight value.
192     """
193     sqrt1 = int(np.ceil(np.sqrt(weights.size(0))))
194     sqrt2 = int(np.ceil(np.sqrt(weights.size(1))))
195     height, width = weights.size(2), weights.size(3)

```

```

196     reshaped = torch.zeros(
197         sqrt1 * sqrt2 * weights.size(2), sqrt1 * sqrt2 * weights.
198         size(3)
199     )
200     for i in range(sqrt1):
201         for j in range(sqrt1):
202             for k in range(sqrt2):
203                 for l in range(sqrt2):
204                     if i * sqrt1 + j < weights.size(0) and k * sqrt2
205                     + l < weights.size(
206                         1
207                     ):
208                         fltr = weights[i * sqrt1 + j, k * sqrt2 + l
209                         ].view(height, width)
210                         reshaped[
211                             i * height
212                             + k * height * sqrt1 : (i + 1) * height
213                             + k * height * sqrt1,
214                             (j % sqrt1) * width
215                             + (l % sqrt2) * width * sqrt1 : ((j %
216                             sqrt1) + 1) * width
217                             + (l % sqrt2) * width * sqrt1,
218                         ] = fltr
219     return reshaped

```

Listing B.2: Refactored conversion module

B.2 Environment and its Initialization

```

1 from .environment import Environment, GymEnvironment

```

Listing B.3: Initialization

```

1 from abc import ABC, abstractmethod
2 from typing import Tuple, Dict, Any
3
4 import gym
5 import numpy as np
6 import torch
7
8 from ..datasets.preprocess import subsample, gray_scale,
9     binary_image, crop
10 from ..encoding import Encoder, NullEncoder
11
12 class Environment(ABC):
13     # language=rst
14     """
15     Abstract environment class.
16     """
17
18     @abstractmethod

```



```

19     def step(self, a: int) -> Tuple[Any, ...]:
20         # language=rst
21         """
22         Abstract method head for ‘‘step()‘‘.
23
24         :param a: Integer action to take in environment.
25         """
26         pass
27
28     @abstractmethod
29     def reset(self) -> None:
30         # language=rst
31         """
32         Abstract method header for ‘‘reset()‘‘.
33         """
34         pass
35
36     @abstractmethod
37     def render(self) -> None:
38         # language=rst
39         """
40         Abstract method header for ‘‘render()‘‘.
41         """
42         pass
43
44     @abstractmethod
45     def close(self) -> None:
46         # language=rst
47         """
48         Abstract method header for ‘‘close()‘‘.
49         """
50         pass
51
52     @abstractmethod
53     def preprocess(self) -> None:
54         # language=rst
55         """
56         Abstract method header for ‘‘preprocess()‘‘.
57         """
58         pass
59
60
61 class GymEnvironment(Environment):
62     # language=rst
63     """
64     A wrapper around the OpenAI ‘‘gym‘‘ environments.
65     """
66
67     def __init__(self, name: str, encoder: Encoder = NullEncoder(),
68     **kwargs) -> None:
69         # language=rst
70         """
71         Initializes the environment wrapper. This class makes the
72         assumption that the OpenAI ‘‘gym‘‘ environment will provide

```

```

an image
72     of format HxW or CxHxW as an observation (we will add the C
73     dimension to HxW tensors) or a 1D observation in which case
no
74     dimensions will be added.
75
76     :param name: The name of an OpenAI ‘‘gym’’ environment.
77     :param encoder: Function to encode observations into spike
trains.
78
79     Keyword arguments:
80
81     :param float max_prob: Maximum spiking probability.
82     :param bool clip_rewards: Whether or not to use ‘‘np.sign’’
of rewards.
83
84     :param int history: Number of observations to keep track of.
85     :param int delta: Step size to save observations in history.
86     :param bool add_channel_dim: Allows for the adding of the
channel dimension in
87         2D inputs.
88     """
89     self.name = name
90     self.env = gym.make(name)
91     self.action_space = self.env.action_space
92
93     self.encoder = encoder
94
95     # Keyword arguments.
96     self.max_prob = kwargs.get("max_prob", 1.0)
97     self.clip_rewards = kwargs.get("clip_rewards", True)
98
99     self.history_length = kwargs.get("history_length", None)
100    self.delta = kwargs.get("delta", 1)
101    self.add_channel_dim = kwargs.get("add_channel_dim", True)
102
103    if self.history_length is not None and self.delta is not
None:
104        self.history = {
105            i: torch.Tensor()
106            for i in range(1, self.history_length * self.delta +
1, self.delta)
107        }
108    else:
109        self.history = {}
110
111    self.episode_step_count = 0
112    self.history_index = 1
113
114    self.obs = None
115    self.reward = None
116
117    assert (
118        0.0 < self.max_prob <= 1.0

```

```

119         ), "Maximum spiking probability must be in (0, 1]."
```

```

120
121     def step(self, a: int) -> Tuple[torch.Tensor, float, bool, Dict[
122         Any, Any]]:
123         # language=rst
124         """
125         Wrapper around the OpenAI ‘gym‘ environment ‘step()‘
126         function.
127
128         :param a: Action to take in the environment.
129         :return: Observation, reward, done flag, and information
130         dictionary.
131         """
132         # Call gym’s environment step function.
133         self.obs, self.reward, self.done, info = self.env.step(a)
134
135         if self.clip_rewards:
136             self.reward = np.sign(self.reward)
137
138         self.preprocess()
139
140         # Add the raw observation from the gym environment into the
141         info
142         # for debugging and display.
143         info["gym_obs"] = self.obs
144
145         # Store frame of history and encode the inputs.
146         if len(self.history) > 0:
147             self.update_history()
148             self.update_index()
149             # Add the delta observation into the info for debugging
150             and display.
151             info["delta_obs"] = self.obs
152
153         # The new standard for images is BxCxHxW.
154         # The gym environment doesn’t follow exactly the same
155         protocol.
156         #
157         # 1D observations will be left as is before the encoder and
158         will become BxCxL.
159         # 2D observations are assumed to be mono images will become
160         BxCx1xHxW
161         # 3D observations will become BxCxHxW
162         if self.obs.dim() == 2 and self.add_channel_dim:
163             # We want CxHxW, it is currently HxW.
164             self.obs = self.obs.unsqueeze(0)
165
166         # The encoder will add time - now Tx...
167         if self.encoder is not None:
168             self.obs = self.encoder(self.obs)
169
170         # Add the batch - now BxCx...
171         self.obs = self.obs.unsqueeze(0)

```

```

165         self.episode_step_count += 1
166
167         # Return converted observations and other information.
168         return self.obs, self.reward, self.done, info
169
170     def reset(self) -> torch.Tensor:
171         # language=rst
172         """
173         Wrapper around the OpenAI ‘‘gym’’ environment ‘‘reset()’’
174         function.
175
176         :return: Observation from the environment.
177         """
178         # Call gym’s environment reset function.
179         self.obs = self.env.reset()
180         self.preprocess()
181
182         self.history = {i: torch.Tensor() for i in self.history}
183
184         self.episode_step_count = 0
185
186         return self.obs
187
188     def render(self) -> None:
189         # language=rst
190         """
191         Wrapper around the OpenAI ‘‘gym’’ environment ‘‘render()’’
192         function.
193
194         """
195         self.env.render()
196
197     def close(self) -> None:
198         # language=rst
199         """
200         Wrapper around the OpenAI ‘‘gym’’ environment ‘‘close()’’
201         function.
202
203         """
204         self.env.close()
205
206     def preprocess(self) -> None:
207         # language=rst
208         """
209         Pre-processing step for an observation from a ‘‘gym’’
210         environment.
211         """
212         if self.name == "SpaceInvaders-v0":
213             self.obs = subsample(gray_scale(self.obs), 84, 110)
214             self.obs = self.obs[26:104, :]
215             self.obs = binary_image(self.obs)
216         elif self.name == "BreakoutDeterministic-v4":
217             self.obs = subsample(gray_scale(crop(self.obs, 34, 194,
218             0, 160)), 80, 80)
219             self.obs = binary_image(self.obs)
220         else: # Default pre-processing step.

```

```

214         pass
215
216         self.obs = torch.from_numpy(self.obs).float()
217
218     def update_history(self) -> None:
219         # language=rst
220         """
221         Updates the observations inside history by performing
222         subtraction from most
223         recent observation and the sum of previous observations. If
224         there are not enough
225         observations to take a difference from, simply store the
226         observation without any
227         differencing.
228         """
229         # Recording initial observations.
230         if self.episode_step_count < len(self.history) * self.delta:
231             # Store observation based on delta value.
232             if self.episode_step_count % self.delta == 0:
233                 self.history[self.history_index] = self.obs
234             else:
235                 # Take difference between stored frames and current
236                 frame.
237                 temp = torch.clamp(self.obs - sum(self.history.values())
238                 , 0, 1)
239
240                 # Store observation based on delta value.
241                 if self.episode_step_count % self.delta == 0:
242                     self.history[self.history_index] = self.obs
243
244                 assert (
245                     len(self.history) == self.history_length
246                 ), "History size is out of bounds"
247                 self.obs = temp
248
249     def update_index(self) -> None:
250         # language=rst
251         """
252         Updates the index to keep track of history. For example: ‘‘
253         history = 4‘‘,
254         ‘‘delta = 3‘‘ will produce ‘‘self.history = {1, 4, 7, 10}‘‘
255         and
256         ‘‘self.history_index‘‘ will be updated according to ‘‘self.
257         delta‘‘ and will wrap
258         around the history dictionary.
259         """
260         if self.episode_step_count % self.delta == 0:
261             if self.history_index != max(self.history.keys()):
262                 self.history_index += self.delta
263             else:
264                 # Wrap around the history.
265                 self.history_index = (self.history_index % max(self.

```

```
history.keys())) + 1
```

Listing B.4: Environment

B.3 Network

```
1 from .network import Network, load
2 from . import nodes, topology, monitors
```

Listing B.5: Initialization

```
1 import tempfile
2 from typing import Dict, Optional, Type, Iterable
3
4 import torch
5
6 from .monitors import AbstractMonitor
7 from .nodes import Nodes
8 from .topology import AbstractConnection
9 from ..learning.reward import AbstractReward
10
11
12 def load(file_name: str, map_location: str = "cpu", learning: bool =
13         None) -> "Network":
14     # language=rst
15     """
16     Loads serialized network object from disk.
17
18     :param file_name: Path to serialized network object on disk.
19     :param map_location: One of ``"cpu"`` or ``"cuda"``. Defaults to
20     ``"cpu"``.
21     :param learning: Whether to load with learning enabled. Default
22     loads value from
23     disk.
24     """
25     network = torch.load(open(file_name, "rb"), map_location=
26     map_location)
27     if learning is not None and "learning" in vars(network):
28         network.learning = learning
29
30     return network
31
32 class Network(torch.nn.Module):
33     # language=rst
34     """
35     Central object of the ``bindsnet`` package. Responsible for the
36     simulation and
37     interaction of nodes and connections.
38
39     **Example:**
40
41     .. code-block:: python
```

```

39     import torch
40     import matplotlib.pyplot as plt
41
42     from bindsnet          import encoding
43     from bindsnet.network import Network, nodes, topology,
monitors
44
45     network = Network(dt=1.0) # Instantiates network.
46
47     X = nodes.Input(100) # Input layer.
48     Y = nodes.LIFNodes(100) # Layer of LIF neurons.
49     C = topology.Connection(source=X, target=Y, w=torch.rand(X.n
, Y.n)) # Connection from X to Y.
50
51     # Spike monitor objects.
52     M1 = monitors.Monitor(obj=X, state_vars=['s'])
53     M2 = monitors.Monitor(obj=Y, state_vars=['s'])
54
55     # Add everything to the network object.
56     network.add_layer(layer=X, name='X')
57     network.add_layer(layer=Y, name='Y')
58     network.add_connection(connection=C, source='X', target='Y')
59     network.add_monitor(monitor=M1, name='X')
60     network.add_monitor(monitor=M2, name='Y')
61
62     # Create Poisson-distributed spike train inputs.
63     data = 15 * torch.rand(100) # Generate random Poisson rates
for 100 input neurons.
64     train = encoding.poisson(datum=data, time=5000) # Encode
input as 5000ms Poisson spike trains.
65
66     # Simulate network on generated spike trains.
67     inputs = {'X' : train} # Create inputs mapping.
68     network.run(inputs=inputs, time=5000) # Run network
simulation.
69
70     # Plot spikes of input and output layers.
71     spikes = {'X' : M1.get('s'), 'Y' : M2.get('s')}
72
73     fig, axes = plt.subplots(2, 1, figsize=(12, 7))
74     for i, layer in enumerate(spikes):
75         axes[i].matshow(spikes[layer], cmap='binary')
76         axes[i].set_title('%s spikes' % layer)
77         axes[i].set_xlabel('Time'); axes[i].set_ylabel('Index of
neuron')
78         axes[i].set_xticks(()); axes[i].set_yticks(())
79         axes[i].set_aspect('auto')
80
81     plt.tight_layout(); plt.show()
82     """
83
84     def __init__(
85         self,
86         dt: float = 1.0,

```

```

87     batch_size: int = 1,
88     learning: bool = True,
89     reward_fn: Optional[Type[AbstractReward]] = None,
90 ) -> None:
91     # language=rst
92     """
93     Initializes network object.
94
95     :param dt: Simulation timestep.
96     :param batch_size: Mini-batch size.
97     :param learning: Whether to allow connection updates. True
98 by default.
99     :param reward_fn: Optional class allowing for modification
100 of reward in case of
101         reward-modulated learning.
102     """
103     super().__init__()
104
105     self.dt = dt
106     self.batch_size = batch_size
107
108     self.layers = {}
109     self.connections = {}
110     self.monitors = {}
111
112     self.train(learning)
113
114     if reward_fn is not None:
115         self.reward_fn = reward_fn()
116     else:
117         self.reward_fn = None
118
119 def add_layer(self, layer: Nodes, name: str) -> None:
120     # language=rst
121     """
122     Adds a layer of nodes to the network.
123
124     :param layer: A subclass of the ‘‘Nodes’’ object.
125     :param name: Logical name of layer.
126     """
127     self.layers[name] = layer
128     self.add_module(name, layer)
129
130     layer.train(self.learning)
131     layer.compute_decays(self.dt)
132     layer.set_batch_size(self.batch_size)
133
134 def add_connection(
135     self, connection: AbstractConnection, source: str, target:
136     str
137 ) -> None:
138     # language=rst
139     """
140     Adds a connection between layers of nodes to the network.

```



```

138
139     :param connection: An instance of class ‘‘Connection‘‘.
140     :param source: Logical name of the connection’s source layer
141     .
142     :param target: Logical name of the connection’s target layer
143     .
144     """
145     self.connections[(source, target)] = connection
146     self.add_module(source + "_to_" + target, connection)
147
148     connection.dt = self.dt
149     connection.train(self.learning)
150
151     def add_monitor(self, monitor: AbstractMonitor, name: str) ->
152     None:
153         # language=rst
154         """
155         Adds a monitor on a network object to the network.
156
157         :param monitor: An instance of class ‘‘Monitor‘‘.
158         :param name: Logical name of monitor object.
159         """
160         self.monitors[name] = monitor
161         monitor.network = self
162         monitor.dt = self.dt
163
164     def save(self, file_name: str) -> None:
165         # language=rst
166         """
167         Serializes the network object to disk.
168
169         :param file_name: Path to store serialized network object on
170         disk.
171
172         **Example:**
173
174         .. code-block:: python
175
176             import torch
177             import matplotlib.pyplot as plt
178
179             from pathlib          import Path
180             from bindsnet.network import *
181             from bindsnet.network import topology
182
183             # Build simple network.
184             network = Network(dt=1.0)
185
186             X = nodes.Input(100) # Input layer.
187             Y = nodes.LIFNodes(100) # Layer of LIF neurons.
188             C = topology.Connection(source=X, target=Y, w=torch.rand
189             (X.n, Y.n)) # Connection from X to Y.
190
191             # Add everything to the network object.

```

```

187         network.add_layer(layer=X, name='X')
188         network.add_layer(layer=Y, name='Y')
189         network.add_connection(connection=C, source='X', target
='Y')
190
191         # Save the network to disk.
192         network.save(str(Path.home()) + '/network.pt')
193     """
194     torch.save(self, open(file_name, "wb"))
195
196     def clone(self) -> "Network":
197         # language=rst
198         """
199         Returns a cloned network object.
200
201         :return: A copy of this network.
202         """
203         virtual_file = tempfile.SpooledTemporaryFile()
204         torch.save(self, virtual_file)
205         virtual_file.seek(0)
206         return torch.load(virtual_file)
207
208     def _get_inputs(self, layers: Iterable = None) -> Dict[str,
torch.Tensor]:
209         # language=rst
210         """
211         Fetches outputs from network layers to use as input to
downstream layers.
212
213         :param layers: Layers to update inputs for. Defaults to all
network layers.
214         :return: Inputs to all layers for the current iteration.
215         """
216         inputs = {}
217
218         if layers is None:
219             layers = self.layers
220
221         # Loop over network connections.
222         for c in self.connections:
223             if c[1] in layers:
224                 # Fetch source and target populations.
225                 source = self.connections[c].source
226                 target = self.connections[c].target
227
228                 if not c[1] in inputs:
229                     inputs[c[1]] = torch.zeros(
230                         self.batch_size, *target.shape, device=
target.s.device
231                     )
232
233                 # Add to input: source's spikes multiplied by
connection weights.
234                 inputs[c[1]] += self.connections[c].compute(source.s

```

```

)
235
236     return inputs
237
238     def run(
239         self, inputs: Dict[str, torch.Tensor], time: int, one_step=
False, **kwargs
240     ) -> None:
241         # language=rst
242         """
243         Simulate network for given inputs and time.
244
245         :param inputs: Dictionary of ‘‘Tensor’’s of shape ‘‘[time, *
input_shape]’’ or
246             ‘‘[time, batch_size, *input_shape]’’.
247         :param time: Simulation time.
248         :param one_step: Whether to run the network in "feed-forward
" mode, where inputs
249             propagate all the way through the network in a single
simulation time step.
250             Layers are updated in the order they are added to the
network.
251
252         Keyword arguments:
253
254         :param Dict[str, torch.Tensor] clamp: Mapping of layer names
to boolean masks if
255             neurons should be clamped to spiking. The ‘‘Tensor’’s
have shape
256             ‘‘[n_neurons]’’ or ‘‘[time, n_neurons]’’.
257         :param Dict[str, torch.Tensor] unclamp: Mapping of layer
names to boolean masks
258             if neurons should be clamped to not spiking. The ‘‘
Tensor’’s should have
259             shape ‘‘[n_neurons]’’ or ‘‘[time, n_neurons]’’.
260         :param Dict[str, torch.Tensor] injects_v: Mapping of layer
names to boolean
261             masks if neurons should be added voltage. The ‘‘Tensor’’
s should have shape
262             ‘‘[n_neurons]’’ or ‘‘[time, n_neurons]’’.
263         :param Union[float, torch.Tensor] reward: Scalar value used
in reward-modulated
264             learning.
265         :param Dict[Tuple[str], torch.Tensor] masks: Mapping of
connection names to
266             boolean masks determining which weights to clamp to zero
.
267
268         **Example:**
269
270         .. code-block:: python
271
272             import torch
273             import matplotlib.pyplot as plt

```

```

274
275     from bindsnet.network import Network
276     from bindsnet.network.nodes import Input
277     from bindsnet.network.monitors import Monitor
278
279     # Build simple network.
280     network = Network()
281     network.add_layer(Input(500), name='I')
282     network.add_monitor(Monitor(network.layers['I'],
state_vars=['s']), 'I')
283
284     # Generate spikes by running Bernoulli trials on Uniform
(0, 0.5) samples.
285     spikes = torch.bernoulli(0.5 * torch.rand(500, 500))
286
287     # Run network simulation.
288     network.run(inputs={'I' : spikes}, time=500)
289
290     # Look at input spiking activity.
291     spikes = network.monitors['I'].get('s')
292     plt.matshow(spikes, cmap='binary')
293     plt.xticks(()); plt.yticks(());
294     plt.xlabel('Time'); plt.ylabel('Neuron index')
295     plt.title('Input spiking')
296     plt.show()
297     """
298     # Parse keyword arguments.
299     clamps = kwargs.get("clamp", {})
300     unclamps = kwargs.get("unclamp", {})
301     masks = kwargs.get("masks", {})
302     injects_v = kwargs.get("injects_v", {})
303
304     # Compute reward.
305     if self.reward_fn is not None:
306         kwargs["reward"] = self.reward_fn.compute(**kwargs)
307
308     # Dynamic setting of batch size.
309     if inputs != {}:
310         for key in inputs:
311             # goal shape is [time, batch, n_0, ...]
312             if len(inputs[key].size()) == 1:
313                 # current shape is [n_0, ...]
314                 # unsqueeze twice to make [1, 1, n_0, ...]
315                 inputs[key] = inputs[key].unsqueeze(0).unsqueeze
(0)
316
317             elif len(inputs[key].size()) == 2:
318                 # current shape is [time, n_0, ...]
319                 # unsqueeze dim 1 so that we have
320                 # [time, 1, n_0, ...]
321                 inputs[key] = inputs[key].unsqueeze(1)
322
323         for key in inputs:
324             # batch dimension is 1, grab this and use for batch
size

```

```

324         if inputs[key].size(1) != self.batch_size:
325             self.batch_size = inputs[key].size(1)
326
327         for l in self.layers:
328             self.layers[l].set_batch_size(self.
batch_size)
329
330         for m in self.monitors:
331             self.monitors[m].reset_state_variables()
332
333         break
334
335     # Effective number of timesteps.
336     timesteps = int(time / self.dt)
337
338     # Simulate network activity for 'time' timesteps.
339     for t in range(timesteps):
340         # Get input to all layers (synchronous mode).
341         current_inputs = {}
342         if not one_step:
343             current_inputs.update(self._get_inputs())
344
345         for l in self.layers:
346             # Update each layer of nodes.
347             if l in inputs:
348                 if l in current_inputs:
349                     current_inputs[l] += inputs[l][t]
350                 else:
351                     current_inputs[l] = inputs[l][t]
352
353             if one_step:
354                 # Get input to this layer (one-step mode).
355                 current_inputs.update(self._get_inputs(layers=[l
]))
356
357         self.layers[l].forward(x=current_inputs[l])
358
359         # Clamp neurons to spike.
360         clamp = clamps.get(l, None)
361         if clamp is not None:
362             if clamp.ndimimension() == 1:
363                 self.layers[l].s[:, clamp] = 1
364             else:
365                 self.layers[l].s[:, clamp[t]] = 1
366
367         # Clamp neurons not to spike.
368         unclamp = unclamps.get(l, None)
369         if unclamp is not None:
370             if unclamp.ndimimension() == 1:
371                 self.layers[l].s[unclamp] = 0
372             else:
373                 self.layers[l].s[unclamp[t]] = 0
374
375         # Inject voltage to neurons.

```

```

376         inject_v = injects_v.get(l, None)
377         if inject_v is not None:
378             if inject_v.ndimimension() == 1:
379                 self.layers[l].v += inject_v
380             else:
381                 self.layers[l].v += inject_v[t]
382
383         # Run synapse updates.
384         for c in self.connections:
385             self.connections[c].update(
386                 mask=masks.get(c, None), learning=self.learning,
**kwargs
387             )
388
389         # Get input to all layers.
390         current_inputs.update(self._get_inputs())
391
392         # Record state variables of interest.
393         for m in self.monitors:
394             self.monitors[m].record()
395
396         # Re-normalize connections.
397         for c in self.connections:
398             self.connections[c].normalize()
399
400     def reset_state_variables(self) -> None:
401         # language=rst
402         """
403         Reset state variables of objects in network.
404         """
405         for layer in self.layers:
406             self.layers[layer].reset_state_variables()
407
408         for connection in self.connections:
409             self.connections[connection].reset_state_variables()
410
411         for monitor in self.monitors:
412             self.monitors[monitor].reset_state_variables()
413
414     def train(self, mode: bool = True) -> "torch.nn.Module":
415         # language=rst
416         """
417         Sets the node in training mode.
418
419         :param mode: Turn training on or off.
420
421         :return: ‘‘self’’ as specified in ‘‘torch.nn.Module’’.
422         """
423         self.learning = mode
424         return super().train(mode)

```

Listing B.6: Network

```

1 from abc import ABC, abstractmethod

```

```

2 from functools import reduce
3 from operator import mul
4 from typing import Iterable, Optional, Union
5
6 import torch
7
8
9 class Nodes(torch.nn.Module):
10     # language=rst
11     """
12     Abstract base class for groups of neurons.
13     """
14
15     def __init__(
16         self,
17         n: Optional[int] = None,
18         shape: Optional[Iterable[int]] = None,
19         traces: bool = False,
20         traces_additive: bool = False,
21         tc_trace: Union[float, torch.Tensor] = 20.0,
22         trace_scale: Union[float, torch.Tensor] = 1.0,
23         sum_input: bool = False,
24         learning: bool = True,
25         **kwargs,
26     ) -> None:
27         # language=rst
28         """
29         Abstract base class constructor.
30
31         :param n: The number of neurons in the layer.
32         :param shape: The dimensionality of the layer.
33         :param traces: Whether to record decaying spike traces.
34         :param traces_additive: Whether to record spike traces
35         additively.
36         :param tc_trace: Time constant of spike trace decay.
37         :param trace_scale: Scaling factor for spike trace.
38         :param sum_input: Whether to sum all inputs.
39         :param learning: Whether to be in learning or testing.
40         """
41         super().__init__()
42
43         assert (
44             n is not None or shape is not None
45         ), "Must provide either no. of neurons or shape of layer"
46
47         if n is None:
48             self.n = reduce(mul, shape) # No. of neurons product of
49             shape.
50         else:
51             self.n = n # No. of neurons provided.
52
53         if shape is None:
54             self.shape = [self.n] # Shape is equal to the size of
55             the layer.

```

```

53     else:
54         self.shape = shape # Shape is passed in as an argument.
55
56     assert self.n == reduce(
57         mul, self.shape
58     ), "No. of neurons and shape do not match"
59
60     self.traces = traces # Whether to record synaptic traces.
61     self.traces_additive = (
62         traces_additive
63     ) # Whether to record spike traces additively.
64     self.register_buffer("s", torch.ByteTensor()) # Spike
occurrences.
65
66     self.sum_input = sum_input # Whether to sum all inputs.
67
68     if self.traces:
69         self.register_buffer("x", torch.Tensor()) # Firing
traces.
70         self.register_buffer(
71             "tc_trace", torch.tensor(tc_trace)
72         ) # Time constant of spike trace decay.
73         if self.traces_additive:
74             self.register_buffer(
75                 "trace_scale", torch.tensor(trace_scale)
76             ) # Scaling factor for spike trace.
77         self.register_buffer(
78             "trace_decay", torch.empty_like(self.tc_trace)
79         ) # Set in compute_decays.
80
81     if self.sum_input:
82         self.register_buffer("summed", torch.FloatTensor()) #
Summed inputs.
83
84     self.dt = None
85     self.batch_size = None
86     self.trace_decay = None
87     self.learning = learning
88
89     @abstractmethod
90     def forward(self, x: torch.Tensor) -> None:
91         # language=rst
92         """
93         Abstract base class method for a single simulation step.
94
95         :param x: Inputs to the layer.
96         """
97         if self.traces:
98             # Decay and set spike traces.
99             self.x *= self.trace_decay
100
101         if self.traces_additive:
102             self.x += self.trace_scale * self.s.float()
103         else:

```



```

104         self.x.masked_fill_(self.s != 0, 1)
105
106     if self.sum_input:
107         # Add current input to running sum.
108         self.summed += x.float()
109
110     def reset_state_variables(self) -> None:
111         # language=rst
112         """
113         Abstract base class method for resetting state variables.
114         """
115         self.s.zero_()
116
117     if self.traces:
118         self.x.zero_() # Spike traces.
119
120     if self.sum_input:
121         self.summed.zero_() # Summed inputs.
122
123     def compute_decays(self, dt) -> None:
124         # language=rst
125         """
126         Abstract base class method for setting decays.
127         """
128         self.dt = dt
129     if self.traces:
130         self.trace_decay = torch.exp(
131             -self.dt / self.tc_trace
132         ) # Spike trace decay (per timestep).
133
134     def set_batch_size(self, batch_size) -> None:
135         # language=rst
136         """
137         Sets mini-batch size. Called when layer is added to a
138         network.
139
140         :param batch_size: Mini-batch size.
141         """
142         self.batch_size = batch_size
143         self.s = torch.zeros(batch_size, *self.shape, device=self.s.
144                               device)
145
146     if self.traces:
147         self.x = torch.zeros(batch_size, *self.shape, device=
148                               self.x.device)
149
150     if self.sum_input:
151         self.summed = torch.zeros(
152             batch_size, *self.shape, device=self.summed.device
153         )
154
155     def train(self, mode: bool = True) -> "Nodes":
156         # language=rst
157         """

```

```

155         Sets the layer in training mode.
156
157         :param bool mode: Turn training on or off
158         :return: self as specified in 'torch.nn.Module'
159         """
160         self.learning = mode
161         return super().train(mode)
162
163
164 class AbstractInput(ABC):
165     # language=rst
166     """
167     Abstract base class for groups of input neurons.
168     """
169
170
171 class Input(Nodes, AbstractInput):
172     # language=rst
173     """
174     Layer of nodes with user-specified spiking behavior.
175     """
176
177     def __init__(
178         self,
179         n: Optional[int] = None,
180         shape: Optional[Iterable[int]] = None,
181         traces: bool = False,
182         traces_additive: bool = False,
183         tc_trace: Union[float, torch.Tensor] = 20.0,
184         trace_scale: Union[float, torch.Tensor] = 1.0,
185         sum_input: bool = False,
186         **kwargs,
187     ) -> None:
188         # language=rst
189         """
190         Instantiates a layer of input neurons.
191
192         :param n: The number of neurons in the layer.
193         :param shape: The dimensionality of the layer.
194         :param traces: Whether to record decaying spike traces.
195         :param traces_additive: Whether to record spike traces
196         additively.
197         :param tc_trace: Time constant of spike trace decay.
198         :param trace_scale: Scaling factor for spike trace.
199         :param sum_input: Whether to sum all inputs.
200         """
201         super().__init__(
202             n=n,
203             shape=shape,
204             traces=traces,
205             traces_additive=traces_additive,
206             tc_trace=tc_trace,
207             trace_scale=trace_scale,
208             sum_input=sum_input,

```

```

208     )
209
210     def forward(self, x: torch.Tensor) -> None:
211         # language=rst
212         """
213         On each simulation step, set the spikes of the population
214         equal to the inputs.
215
216         :param x: Inputs to the layer.
217         """
218         # Set spike occurrences to input values.
219         self.s = x
220
221         super().forward(x)
222
223     def reset_state_variables(self) -> None:
224         # language=rst
225         """
226         Resets relevant state variables.
227         """
228         super().reset_state_variables()
229
230 class McCullochPitts(Nodes):
231     # language=rst
232     """
233     Layer of 'McCulloch-Pitts neurons
234     <http://wwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html>'_.
235     """
236
237     def __init__(
238         self,
239         n: Optional[int] = None,
240         shape: Optional[Iterable[int]] = None,
241         traces: bool = False,
242         traces_additive: bool = False,
243         tc_trace: Union[float, torch.Tensor] = 20.0,
244         trace_scale: Union[float, torch.Tensor] = 1.0,
245         sum_input: bool = False,
246         thresh: Union[float, torch.Tensor] = 1.0,
247         **kwargs,
248     ) -> None:
249         # language=rst
250         """
251         Instantiates a McCulloch-Pitts layer of neurons.
252
253         :param n: The number of neurons in the layer.
254         :param shape: The dimensionality of the layer.
255         :param traces: Whether to record spike traces.
256         :param traces_additive: Whether to record spike traces
257         additively.
258         :param tc_trace: Time constant of spike trace decay.
259         :param trace_scale: Scaling factor for spike trace.

```

```

259     :param sum_input: Whether to sum all inputs.
260     :param thresh: Spike threshold voltage.
261     """
262     super().__init__(
263         n=n,
264         shape=shape,
265         traces=traces,
266         traces_additive=traces_additive,
267         tc_trace=tc_trace,
268         trace_scale=trace_scale,
269         sum_input=sum_input,
270     )
271
272     self.register_buffer(
273         "thresh", torch.tensor(thresh, dtype=torch.float)
274     ) # Spike threshold voltage.
275     self.register_buffer("v", torch.FloatTensor()) # Neuron
276     voltages.
277
278     def forward(self, x: torch.Tensor) -> None:
279         # language=rst
280         """
281         Runs a single simulation step.
282
283         :param x: Inputs to the layer.
284         """
285         self.v = x # Voltages are equal to the inputs.
286         self.s = self.v >= self.thresh # Check for spiking neurons.
287
288         super().forward(x)
289
290     def reset_state_variables(self) -> None:
291         # language=rst
292         """
293         Resets relevant state variables.
294         """
295         super().reset_state_variables()
296
297     def set_batch_size(self, batch_size) -> None:
298         # language=rst
299         """
300         Sets mini-batch size. Called when layer is added to a
301         network.
302
303         :param batch_size: Mini-batch size.
304         """
305         super().set_batch_size(batch_size=batch_size)
306         self.v = torch.zeros(batch_size, *self.shape, device=self.v.
307         device)
308
309 class IFNodes(Nodes):
310     # language=rst
311     """

```

```

310 Layer of 'integrate-and-fire (IF) neurons <http://
neurondynamics.epfl.ch/online/Ch1.S3.html>'.
311 """
312
313 def __init__(
314     self,
315     n: Optional[int] = None,
316     shape: Optional[Iterable[int]] = None,
317     traces: bool = False,
318     traces_additive: bool = False,
319     tc_trace: Union[float, torch.Tensor] = 20.0,
320     trace_scale: Union[float, torch.Tensor] = 1.0,
321     sum_input: bool = False,
322     thresh: Union[float, torch.Tensor] = -52.0,
323     reset: Union[float, torch.Tensor] = -65.0,
324     refrac: Union[int, torch.Tensor] = 5,
325     lbound: float = None,
326     **kwargs,
327 ) -> None:
328     # language=rst
329     """
330     Instantiates a layer of IF neurons.
331
332     :param n: The number of neurons in the layer.
333     :param shape: The dimensionality of the layer.
334     :param traces: Whether to record spike traces.
335     :param traces_additive: Whether to record spike traces
additively.
336     :param tc_trace: Time constant of spike trace decay.
337     :param trace_scale: Scaling factor for spike trace.
338     :param sum_input: Whether to sum all inputs.
339     :param thresh: Spike threshold voltage.
340     :param reset: Post-spike reset voltage.
341     :param refrac: Refractory (non-firing) period of the neuron.
342     :param lbound: Lower bound of the voltage.
343     """
344     super().__init__(
345         n=n,
346         shape=shape,
347         traces=traces,
348         traces_additive=traces_additive,
349         tc_trace=tc_trace,
350         trace_scale=trace_scale,
351         sum_input=sum_input,
352     )
353
354     self.register_buffer(
355         "reset", torch.tensor(reset, dtype=torch.float)
356     ) # Post-spike reset voltage.
357     self.register_buffer(
358         "thresh", torch.tensor(thresh, dtype=torch.float)
359     ) # Spike threshold voltage.
360     self.register_buffer(
361         "refrac", torch.tensor(refrac)

```

```

362         ) # Post-spike refractory period.
363         self.register_buffer("v", torch.FloatTensor()) # Neuron
voltages.
364         self.register_buffer(
365             "refrac_count", torch.FloatTensor()
366         ) # Refractory period counters.
367
368         self.lbound = lbound # Lower bound of voltage.
369
370     def forward(self, x: torch.Tensor) -> None:
371         # language=rst
372         """
373         Runs a single simulation step.
374
375         :param x: Inputs to the layer.
376         """
377         # Integrate input voltages.
378         self.v += (self.refrac_count == 0).float() * x
379
380         # Decrement refractory counters.
381         self.refrac_count = (self.refrac_count > 0).float() * (
382             self.refrac_count - self.dt
383         )
384
385         # Check for spiking neurons.
386         self.s = self.v >= self.thresh
387
388         # Refractoriness and voltage reset.
389         self.refrac_count.masked_fill_(self.s, self.refrac)
390         self.v.masked_fill_(self.s, self.reset)
391
392         # Voltage clipping to lower bound.
393         if self.lbound is not None:
394             self.v.masked_fill_(self.v < self.lbound, self.lbound)
395
396         super().forward(x)
397
398     def reset_state_variables(self) -> None:
399         # language=rst
400         """
401         Resets relevant state variables.
402         """
403         super().reset_state_variables()
404         self.v.fill_(self.reset) # Neuron voltages.
405         self.refrac_count.zero_() # Refractory period counters.
406
407     def set_batch_size(self, batch_size) -> None:
408         # language=rst
409         """
410         Sets mini-batch size. Called when layer is added to a
network.
411
412         :param batch_size: Mini-batch size.
413         """

```

```

414         super().set_batch_size(batch_size=batch_size)
415         self.v = self.reset * torch.ones(batch_size, *self.shape,
device=self.v.device)
416         self.refrac_count = torch.zeros_like(self.v, device=self.
refrac_count.device)
417
418
419 class LIFNodes(Nodes):
420     # language=rst
421     """
422     Layer of 'leaky integrate-and-fire (LIF) neurons
423     <http://icwww.epfl.ch/~gerstner/SPNM/node26.html#
SECTION02311000000000000000>'_.
424     """
425
426     def __init__(
427         self,
428         n: Optional[int] = None,
429         shape: Optional[Iterable[int]] = None,
430         traces: bool = False,
431         traces_additive: bool = False,
432         tc_trace: Union[float, torch.Tensor] = 20.0,
433         trace_scale: Union[float, torch.Tensor] = 1.0,
434         sum_input: bool = False,
435         thresh: Union[float, torch.Tensor] = -52.0,
436         rest: Union[float, torch.Tensor] = -65.0,
437         reset: Union[float, torch.Tensor] = -65.0,
438         refrac: Union[int, torch.Tensor] = 5,
439         tc_decay: Union[float, torch.Tensor] = 100.0,
440         lbound: float = None,
441         **kwargs,
442     ) -> None:
443         # language=rst
444         """
445         Instantiates a layer of LIF neurons.
446
447         :param n: The number of neurons in the layer.
448         :param shape: The dimensionality of the layer.
449         :param traces: Whether to record spike traces.
450         :param traces_additive: Whether to record spike traces
additively.
451         :param tc_trace: Time constant of spike trace decay.
452         :param trace_scale: Scaling factor for spike trace.
453         :param sum_input: Whether to sum all inputs.
454         :param thresh: Spike threshold voltage.
455         :param rest: Resting membrane voltage.
456         :param reset: Post-spike reset voltage.
457         :param refrac: Refractory (non-firing) period of the neuron.
458         :param tc_decay: Time constant of neuron voltage decay.
459         :param lbound: Lower bound of the voltage.
460         """
461         super().__init__(
462             n=n,
463             shape=shape,

```

```

464         traces=traces,
465         traces_additive=traces_additive,
466         tc_trace=tc_trace,
467         trace_scale=trace_scale,
468         sum_input=sum_input,
469     )
470
471     self.register_buffer(
472         "rest", torch.tensor(rest, dtype=torch.float)
473     ) # Rest voltage.
474     self.register_buffer(
475         "reset", torch.tensor(reset, dtype=torch.float)
476     ) # Post-spike reset voltage.
477     self.register_buffer(
478         "thresh", torch.tensor(thresh, dtype=torch.float)
479     ) # Spike threshold voltage.
480     self.register_buffer(
481         "refrac", torch.tensor(refrac)
482     ) # Post-spike refractory period.
483     self.register_buffer(
484         "tc_decay", torch.tensor(tc_decay)
485     ) # Time constant of neuron voltage decay.
486     self.register_buffer(
487         "decay", torch.zeros(*self.shape)
488     ) # Set in compute_decays.
489     self.register_buffer("v", torch.FloatTensor()) # Neuron
voltage.
490     self.register_buffer(
491         "refrac_count", torch.FloatTensor()
492     ) # Refractory period counters.
493
494     self.lbound = lbound # Lower bound of voltage.
495
496     def forward(self, x: torch.Tensor) -> None:
497         # language=rst
498         """
499         Runs a single simulation step.
500
501         :param x: Inputs to the layer.
502         """
503         # Decay voltages.
504         self.v = self.decay * (self.v - self.rest) + self.rest
505
506         # Integrate inputs.
507         self.v += (self.refrac_count == 0).float() * x
508
509         # Decrement refractory counters.
510         self.refrac_count = (self.refrac_count > 0).float() * (
511             self.refrac_count - self.dt
512         )
513
514         # Check for spiking neurons.
515         self.s = self.v >= self.thresh
516

```



```

517     # Refractoriness and voltage reset.
518     self.refrac_count.masked_fill_(self.s, self.refrac)
519     self.v.masked_fill_(self.s, self.reset)
520
521     # Voltage clipping to lower bound.
522     if self.lbound is not None:
523         self.v.masked_fill_(self.v < self.lbound, self.lbound)
524
525     super().forward(x)
526
527     def reset_state_variables(self) -> None:
528         # language=rst
529         """
530         Resets relevant state variables.
531         """
532         super().reset_state_variables()
533         self.v.fill_(self.rest) # Neuron voltages.
534         self.refrac_count.zero_() # Refractory period counters.
535
536     def compute_decays(self, dt) -> None:
537         # language=rst
538         """
539         Sets the relevant decays.
540         """
541         super().compute_decays(dt=dt)
542         self.decay = torch.exp(
543             -self.dt / self.tc_decay
544         ) # Neuron voltage decay (per timestep).
545
546     def set_batch_size(self, batch_size) -> None:
547         # language=rst
548         """
549         Sets mini-batch size. Called when layer is added to a
550         network.
551
552         :param batch_size: Mini-batch size.
553         """
554         super().set_batch_size(batch_size=batch_size)
555         self.v = self.rest * torch.ones(batch_size, *self.shape,
556                                         device=self.v.device)
557         self.refrac_count = torch.zeros_like(self.v, device=self.
558                                             refrac_count.device)
559
560     class CurrentLIFNodes(Nodes):
561         # language=rst
562         """
563         Layer of 'current-based leaky integrate-and-fire (LIF) neurons
564         <http://icwww.epfl.ch/~gerstner/SPNM/node26.html>#
565         SECTION02313000000000000000>'_.
566         Total synaptic input current is modeled as a decaying memory of
567         input spikes multiplied by synaptic strengths.
568         """

```

```

566     def __init__(
567         self,
568         n: Optional[int] = None,
569         shape: Optional[Iterable[int]] = None,
570         traces: bool = False,
571         traces_additive: bool = False,
572         tc_trace: Union[float, torch.Tensor] = 20.0,
573         trace_scale: Union[float, torch.Tensor] = 1.0,
574         sum_input: bool = False,
575         thresh: Union[float, torch.Tensor] = -52.0,
576         rest: Union[float, torch.Tensor] = -65.0,
577         reset: Union[float, torch.Tensor] = -65.0,
578         refrac: Union[int, torch.Tensor] = 5,
579         tc_decay: Union[float, torch.Tensor] = 100.0,
580         tc_i_decay: Union[float, torch.Tensor] = 2.0,
581         lbound: float = None,
582         **kwargs,
583     ) -> None:
584         # language=rst
585         """
586         Instantiates a layer of synaptic input current-based LIF
587         neurons.
588         :param n: The number of neurons in the layer.
589         :param shape: The dimensionality of the layer.
590         :param traces: Whether to record spike traces.
591         :param traces_additive: Whether to record spike traces
592         additively.
593         :param tc_trace: Time constant of spike trace decay.
594         :param trace_scale: Scaling factor for spike trace.
595         :param sum_input: Whether to sum all inputs.
596         :param thresh: Spike threshold voltage.
597         :param rest: Resting membrane voltage.
598         :param reset: Post-spike reset voltage.
599         :param refrac: Refractory (non-firing) period of the neuron.
600         :param tc_decay: Time constant of neuron voltage decay.
601         :param tc_i_decay: Time constant of synaptic input current
602         decay.
603         :param lbound: Lower bound of the voltage.
604         """
605         super().__init__(
606             n=n,
607             shape=shape,
608             traces=traces,
609             traces_additive=traces_additive,
610             tc_trace=tc_trace,
611             trace_scale=trace_scale,
612             sum_input=sum_input,
613         )
614         self.register_buffer("rest", torch.tensor(rest)) # Rest
615         self.register_buffer("reset", torch.tensor(reset)) # Post-
616         self.register_buffer("thresh", torch.tensor(thresh)) #

```

```

Spike threshold voltage.
615     self.register_buffer(
616         "refrac", torch.tensor(refrac)
617     ) # Post-spike refractory period.
618     self.register_buffer(
619         "tc_decay", torch.tensor(tc_decay)
620     ) # Time constant of neuron voltage decay.
621     self.register_buffer(
622         "decay", torch.empty_like(self.tc_decay)
623     ) # Set in compute_decays.
624     self.register_buffer(
625         "tc_i_decay", torch.tensor(tc_i_decay)
626     ) # Time constant of synaptic input current decay.
627     self.register_buffer(
628         "i_decay", torch.empty_like(self.tc_i_decay)
629     ) # Set in compute_decays.
630
631     self.register_buffer("v", torch.FloatTensor()) # Neuron
voltage.
632     self.register_buffer("i", torch.FloatTensor()) # Synaptic
input currents.
633     self.register_buffer(
634         "refrac_count", torch.FloatTensor()
635     ) # Refractory period counters.
636
637     self.lbound = lbound # Lower bound of voltage.
638
639     def forward(self, x: torch.Tensor) -> None:
640         # language=rst
641         """
642         Runs a single simulation step.
643
644         :param x: Inputs to the layer.
645         """
646         # Decay voltages and current.
647         self.v = self.decay * (self.v - self.rest) + self.rest
648         self.i *= self.i_decay
649
650         # Decrement refractory counters.
651         self.refrac_count = (self.refrac_count > 0).float() * (
652             self.refrac_count - self.dt
653         )
654
655         # Integrate inputs.
656         self.i += x
657         self.v += (self.refrac_count == 0).float() * self.i
658
659         # Check for spiking neurons.
660         self.s = self.v >= self.thresh
661
662         # Refractoriness and voltage reset.
663         self.refrac_count.masked_fill_(self.s, self.refrac)
664         self.v.masked_fill_(self.s, self.reset)
665

```

```

666     # Voltage clipping to lower bound.
667     if self.lbound is not None:
668         self.v.masked_fill_(self.v < self.lbound, self.lbound)
669
670     super().forward(x)
671
672     def reset_state_variables(self) -> None:
673         # language=rst
674         """
675         Resets relevant state variables.
676         """
677         super().reset_state_variables()
678         self.v.fill_(self.rest) # Neuron voltages.
679         self.i.zero_() # Synaptic input currents.
680         self.refrac_count.zero_() # Refractory period counters.
681
682     def compute_decays(self, dt) -> None:
683         # language=rst
684         """
685         Sets the relevant decays.
686         """
687         super().compute_decays(dt=dt)
688         self.decay = torch.exp(
689             -self.dt / self.tc_decay
690         ) # Neuron voltage decay (per timestep).
691         self.i_decay = torch.exp(
692             -self.dt / self.tc_i_decay
693         ) # Synaptic input current decay (per timestep).
694
695     def set_batch_size(self, batch_size) -> None:
696         # language=rst
697         """
698         Sets mini-batch size. Called when layer is added to a
699         network.
700
701         :param batch_size: Mini-batch size.
702         """
703         super().set_batch_size(batch_size=batch_size)
704         self.v = self.rest * torch.ones(batch_size, *self.shape,
705             device=self.v.device)
706         self.i = torch.zeros_like(self.v, device=self.i.device)
707         self.refrac_count = torch.zeros_like(self.v, device=self.
708             refrac_count.device)
709
710     class AdaptiveLIFNodes(Nodes):
711         # language=rst
712         """
713         Layer of leaky integrate-and-fire (LIF) neurons with adaptive
714         thresholds. A neuron's voltage threshold is increased
715         by some constant each time it spikes; otherwise, it is decaying
716         back to its default value.
717         """

```

```

715     def __init__(
716         self,
717         n: Optional[int] = None,
718         shape: Optional[Iterable[int]] = None,
719         traces: bool = False,
720         traces_additive: bool = False,
721         tc_trace: Union[float, torch.Tensor] = 20.0,
722         trace_scale: Union[float, torch.Tensor] = 1.0,
723         sum_input: bool = False,
724         rest: Union[float, torch.Tensor] = -65.0,
725         reset: Union[float, torch.Tensor] = -65.0,
726         thresh: Union[float, torch.Tensor] = -52.0,
727         refrac: Union[int, torch.Tensor] = 5,
728         tc_decay: Union[float, torch.Tensor] = 100.0,
729         theta_plus: Union[float, torch.Tensor] = 0.05,
730         tc_theta_decay: Union[float, torch.Tensor] = 1e7,
731         lbound: float = None,
732         **kwargs,
733     ) -> None:
734         # language=rst
735         """
736         Instantiates a layer of LIF neurons with adaptive firing
737         thresholds.
738
739         :param n: The number of neurons in the layer.
740         :param shape: The dimensionality of the layer.
741         :param traces: Whether to record spike traces.
742         :param traces_additive: Whether to record spike traces
743         additively.
744         :param tc_trace: Time constant of spike trace decay.
745         :param trace_scale: Scaling factor for spike trace.
746         :param sum_input: Whether to sum all inputs.
747         :param rest: Resting membrane voltage.
748         :param reset: Post-spike reset voltage.
749         :param thresh: Spike threshold voltage.
750         :param refrac: Refractory (non-firing) period of the neuron.
751         :param tc_decay: Time constant of neuron voltage decay.
752         :param theta_plus: Voltage increase of threshold after
753         spiking.
754         :param tc_theta_decay: Time constant of adaptive threshold
755         decay.
756         :param lbound: Lower bound of the voltage.
757         """
758         super().__init__(
759             n=n,
760             shape=shape,
761             traces=traces,
762             traces_additive=traces_additive,
763             tc_trace=tc_trace,
764             trace_scale=trace_scale,
765             sum_input=sum_input,
766         )
767
768         self.register_buffer("rest", torch.tensor(rest)) # Rest

```

```

voltage.
765     self.register_buffer("reset", torch.tensor(reset)) # Post-
      spike reset voltage.
766     self.register_buffer("thresh", torch.tensor(thresh)) #
      Spike threshold voltage.
767     self.register_buffer(
768         "refrac", torch.tensor(refrac)
769     ) # Post-spike refractory period.
770     self.register_buffer(
771         "tc_decay", torch.tensor(tc_decay)
772     ) # Time constant of neuron voltage decay.
773     self.register_buffer(
774         "decay", torch.empty_like(self.tc_decay)
775     ) # Set in compute_decays.
776     self.register_buffer(
777         "theta_plus", torch.tensor(theta_plus)
778     ) # Constant threshold increase on spike.
779     self.register_buffer(
780         "tc_theta_decay", torch.tensor(tc_theta_decay)
781     ) # Time constant of adaptive threshold decay.
782     self.register_buffer(
783         "theta_decay", torch.empty_like(self.tc_theta_decay)
784     ) # Set in compute_decays.
785
786     self.register_buffer("v", torch.FloatTensor()) # Neuron
      voltages.
787     self.register_buffer("theta", torch.zeros(*self.shape)) #
      Adaptive thresholds.
788     self.register_buffer(
789         "refrac_count", torch.FloatTensor()
790     ) # Refractory period counters.
791     self.lbound = lbound # Lower bound of voltage.
792
793     def forward(self, x: torch.Tensor) -> None:
794         # language=rst
795         """
796         Runs a single simulation step.
797
798         :param x: Inputs to the layer.
799         """
800         # Decay voltages and adaptive thresholds.
801         self.v = self.decay * (self.v - self.rest) + self.rest
802         if self.learning:
803             self.theta *= self.theta_decay
804
805         # Integrate inputs.
806         self.v += (self.refrac_count == 0).float() * x
807
808         # Decrement refractory counters.
809         self.refrac_count = (self.refrac_count > 0).float() * (
810             self.refrac_count - self.dt
811         )
812
813         # Check for spiking neurons.

```

```

814     self.s = self.v >= self.thresh + self.theta
815
816     # Refractoriness, voltage reset, and adaptive thresholds.
817     self.refrac_count.masked_fill_(self.s, self.refrac)
818     self.v.masked_fill_(self.s, self.reset)
819     if self.learning:
820         self.theta += self.theta_plus * self.s.float().sum(0)
821
822     # voltage clipping to lowerbound
823     if self.lbound is not None:
824         self.v.masked_fill_(self.v < self.lbound, self.lbound)
825
826     super().forward(x)
827
828     def reset_state_variables(self) -> None:
829         # language=rst
830         """
831         Resets relevant state variables.
832         """
833         super().reset_state_variables()
834         self.v.fill_(self.rest) # Neuron voltages.
835         self.refrac_count.zero_() # Refractory period counters.
836
837     def compute_decays(self, dt) -> None:
838         # language=rst
839         """
840         Sets the relevant decays.
841         """
842         super().compute_decays(dt=dt)
843         self.decay = torch.exp(
844             -self.dt / self.tc_decay
845         ) # Neuron voltage decay (per timestep).
846         self.theta_decay = torch.exp(
847             -self.dt / self.tc_theta_decay
848         ) # Adaptive threshold decay (per timestep).
849
850     def set_batch_size(self, batch_size) -> None:
851         # language=rst
852         """
853         Sets mini-batch size. Called when layer is added to a
854         network.
855
856         :param batch_size: Mini-batch size.
857         """
858         super().set_batch_size(batch_size=batch_size)
859         self.v = self.rest * torch.ones(batch_size, *self.shape,
860             device=self.v.device)
861         self.refrac_count = torch.zeros_like(self.v, device=self.
862             refrac_count.device)
863
864     class DiehlAndCookNodes(Nodes):
865         # language=rst
866         """

```

```

865     Layer of leaky integrate-and-fire (LIF) neurons with adaptive
866     thresholds (modified for Diehl & Cook 2015
867     replication).
868     """
869
870     def __init__(
871         self,
872         n: Optional[int] = None,
873         shape: Optional[Iterable[int]] = None,
874         traces: bool = False,
875         traces_additive: bool = False,
876         tc_trace: Union[float, torch.Tensor] = 20.0,
877         trace_scale: Union[float, torch.Tensor] = 1.0,
878         sum_input: bool = False,
879         thresh: Union[float, torch.Tensor] = -52.0,
880         rest: Union[float, torch.Tensor] = -65.0,
881         reset: Union[float, torch.Tensor] = -65.0,
882         refrac: Union[int, torch.Tensor] = 5,
883         tc_decay: Union[float, torch.Tensor] = 100.0,
884         theta_plus: Union[float, torch.Tensor] = 0.05,
885         tc_theta_decay: Union[float, torch.Tensor] = 1e7,
886         lbound: float = None,
887         one_spike: bool = True,
888         **kwargs,
889     ) -> None:
890         # language=rst
891         """
892         Instantiates a layer of Diehl & Cook 2015 neurons.
893
894         :param n: The number of neurons in the layer.
895         :param shape: The dimensionality of the layer.
896         :param traces: Whether to record spike traces.
897         :param traces_additive: Whether to record spike traces
898         additively.
899         :param tc_trace: Time constant of spike trace decay.
900         :param trace_scale: Scaling factor for spike trace.
901         :param sum_input: Whether to sum all inputs.
902         :param thresh: Spike threshold voltage.
903         :param rest: Resting membrane voltage.
904         :param reset: Post-spike reset voltage.
905         :param refrac: Refractory (non-firing) period of the neuron.
906         :param tc_decay: Time constant of neuron voltage decay.
907         :param theta_plus: Voltage increase of threshold after
908         spiking.
909         :param tc_theta_decay: Time constant of adaptive threshold
910         decay.
911         :param lbound: Lower bound of the voltage.
912         :param one_spike: Whether to allow only one spike per
913         timestep.
914         """
915         super().__init__(
916             n=n,
917             shape=shape,
918             traces=traces,

```



```

914         traces_additive=traces_additive,
915         tc_trace=tc_trace,
916         trace_scale=trace_scale,
917         sum_input=sum_input,
918     )
919
920     self.register_buffer("rest", torch.tensor(rest)) # Rest
voltage.
921     self.register_buffer("reset", torch.tensor(reset)) # Post-
spike reset voltage.
922     self.register_buffer("thresh", torch.tensor(thresh)) #
Spike threshold voltage.
923     self.register_buffer(
924         "refrac", torch.tensor(refrac)
925     ) # Post-spike refractory period.
926     self.register_buffer(
927         "tc_decay", torch.tensor(tc_decay)
928     ) # Time constant of neuron voltage decay.
929     self.register_buffer(
930         "decay", torch.empty_like(self.tc_decay)
931     ) # Set in compute_decays.
932     self.register_buffer(
933         "theta_plus", torch.tensor(theta_plus)
934     ) # Constant threshold increase on spike.
935     self.register_buffer(
936         "tc_theta_decay", torch.tensor(tc_theta_decay)
937     ) # Time constant of adaptive threshold decay.
938     self.register_buffer(
939         "theta_decay", torch.empty_like(self.tc_theta_decay)
940     ) # Set in compute_decays.
941     self.register_buffer("v", torch.FloatTensor()) # Neuron
voltage.
942     self.register_buffer("theta", torch.zeros(*self.shape)) #
Adaptive thresholds.
943     self.register_buffer(
944         "refrac_count", torch.FloatTensor()
945     ) # Refractory period counters.
946
947     self.lbound = lbound # Lower bound of voltage.
948     self.one_spike = one_spike # One spike per timestep.
949
950     def forward(self, x: torch.Tensor) -> None:
951         # language=rst
952         """
953         Runs a single simulation step.
954
955         :param x: Inputs to the layer.
956         """
957         # Decay voltages and adaptive thresholds.
958         self.v = self.decay * (self.v - self.rest) + self.rest
959         if self.learning:
960             self.theta *= self.theta_decay
961
962         # Integrate inputs.

```

```

963     self.v += (self.refrac_count == 0).float() * x
964
965     # Decrement refractory counters.
966     self.refrac_count = (self.refrac_count > 0).float() * (
967         self.refrac_count - self.dt
968     )
969
970     # Check for spiking neurons.
971     self.s = self.v >= self.thresh + self.theta
972
973     # Refractoriness, voltage reset, and adaptive thresholds.
974     self.refrac_count.masked_fill_(self.s, self.refrac)
975     self.v.masked_fill_(self.s, self.reset)
976     if self.learning:
977         self.theta += self.theta_plus * self.s.float().sum(0)
978
979     # Choose only a single neuron to spike.
980     if self.one_spike:
981         if self.s.any():
982             _any = self.s.view(self.batch_size, -1).any(1)
983             ind = torch.multinomial(
984                 self.s.float().view(self.batch_size, -1)[_any],
1
985                 )
986             _any = _any.nonzero()
987             self.s.zero_()
988             self.s.view(self.batch_size, -1)[_any, ind] = 1
989
990     # Voltage clipping to lower bound.
991     if self.lbound is not None:
992         self.v.masked_fill_(self.v < self.lbound, self.lbound)
993
994     super().forward(x)
995
996     def reset_state_variables(self) -> None:
997         # language=rst
998         """
999         Resets relevant state variables.
1000         """
1001         super().reset_state_variables()
1002         self.v.fill_(self.reset) # Neuron voltages.
1003         self.refrac_count.zero_() # Refractory period counters.
1004
1005     def compute_decays(self, dt) -> None:
1006         # language=rst
1007         """
1008         Sets the relevant decays.
1009         """
1010         super().compute_decays(dt=dt)
1011         self.decay = torch.exp(
1012             -self.dt / self.tc_decay
1013         ) # Neuron voltage decay (per timestep).
1014         self.theta_decay = torch.exp(
1015             -self.dt / self.tc_theta_decay

```

```

1016         ) # Adaptive threshold decay (per timestep).
1017
1018     def set_batch_size(self, batch_size) -> None:
1019         # language=rst
1020         """
1021         Sets mini-batch size. Called when layer is added to a
1022         network.
1023
1024         :param batch_size: Mini-batch size.
1025         """
1026         super().set_batch_size(batch_size=batch_size)
1027         self.v = self.rest * torch.ones(batch_size, *self.shape,
1028         device=self.v.device)
1029         self.refrac_count = torch.zeros_like(self.v, device=self.
1030         refrac_count.device)
1031
1032 class IzhikevichNodes(Nodes):
1033     # language=rst
1034     """
1035     Layer of Izhikevich neurons.
1036     """
1037
1038     def __init__(
1039     self,
1040     n: Optional[int] = None,
1041     shape: Optional[Iterable[int]] = None,
1042     traces: bool = False,
1043     traces_additive: bool = False,
1044     tc_trace: Union[float, torch.Tensor] = 20.0,
1045     trace_scale: Union[float, torch.Tensor] = 1.0,
1046     sum_input: bool = False,
1047     excitatory: float = 1,
1048     thresh: Union[float, torch.Tensor] = 45.0,
1049     rest: Union[float, torch.Tensor] = -65.0,
1050     lbound: float = None,
1051     **kwargs,
1052 ) -> None:
1053     # language=rst
1054     """
1055     Instantiates a layer of Izhikevich neurons.
1056
1057     :param n: The number of neurons in the layer.
1058     :param shape: The dimensionality of the layer.
1059     :param traces: Whether to record spike traces.
1060     :param traces_additive: Whether to record spike traces
1061     additively.
1062     :param tc_trace: Time constant of spike trace decay.
1063     :param trace_scale: Scaling factor for spike trace.
1064     :param sum_input: Whether to sum all inputs.
1065     :param excitatory: Percent of excitatory (vs. inhibitory)
1066     neurons in the layer; in range "[0, 1]".
1067     :param thresh: Spike threshold voltage.
1068     :param rest: Resting membrane voltage.

```

```

1065 :param lbound: Lower bound of the voltage.
1066 """
1067 super().__init__(
1068     n=n,
1069     shape=shape,
1070     traces=traces,
1071     traces_additive=traces_additive,
1072     tc_trace=tc_trace,
1073     trace_scale=trace_scale,
1074     sum_input=sum_input,
1075 )
1076
1077 self.register_buffer("rest", torch.tensor(rest)) # Rest
1078 voltage.
1079 self.register_buffer("thresh", torch.tensor(thresh)) #
1080 Spike threshold voltage.
1081 self.lbound = lbound
1082
1083 self.register_buffer("r", None)
1084 self.register_buffer("a", None)
1085 self.register_buffer("b", None)
1086 self.register_buffer("c", None)
1087 self.register_buffer("d", None)
1088 self.register_buffer("S", None)
1089 self.register_buffer("excitatory", None)
1090
1091 if excitatory > 1:
1092     excitatory = 1
1093 elif excitatory < 0:
1094     excitatory = 0
1095
1096 if excitatory == 1:
1097     self.r = torch.rand(n)
1098     self.a = 0.02 * torch.ones(n)
1099     self.b = 0.2 * torch.ones(n)
1100     self.c = -65.0 + 15 * (self.r ** 2)
1101     self.d = 8 - 6 * (self.r ** 2)
1102     self.S = 0.5 * torch.rand(n, n)
1103     self.excitatory = torch.ones(n).byte()
1104
1105 elif excitatory == 0:
1106     self.r = torch.rand(n)
1107     self.a = 0.02 + 0.08 * self.r
1108     self.b = 0.25 - 0.05 * self.r
1109     self.c = -65.0 * torch.ones(n)
1110     self.d = 2 * torch.ones(n)
1111     self.S = -torch.rand(n, n)
1112
1113     self.excitatory = torch.zeros(n).byte()
1114
1115 else:
1116     self.excitatory = torch.zeros(n).byte()
1117
1118     ex = int(n * excitatory)

```

```

1117         inh = n - ex
1118
1119         # init
1120         self.r = torch.zeros(n)
1121         self.a = torch.zeros(n)
1122         self.b = torch.zeros(n)
1123         self.c = torch.zeros(n)
1124         self.d = torch.zeros(n)
1125         self.S = torch.zeros(n, n)
1126
1127         # excitatory
1128         self.r[:ex] = torch.rand(ex)
1129         self.a[:ex] = 0.02 * torch.ones(ex)
1130         self.b[:ex] = 0.2 * torch.ones(ex)
1131         self.c[:ex] = -65.0 + 15 * self.r[:ex] ** 2
1132         self.d[:ex] = 8 - 6 * self.r[:ex] ** 2
1133         self.S[:, :ex] = 0.5 * torch.rand(n, ex)
1134         self.excitatory[:ex] = 1
1135
1136         # inhibitory
1137         self.r[ex:] = torch.rand(inh)
1138         self.a[ex:] = 0.02 + 0.08 * self.r[ex:]
1139         self.b[ex:] = 0.25 - 0.05 * self.r[ex:]
1140         self.c[ex:] = -65.0 * torch.ones(inh)
1141         self.d[ex:] = 2 * torch.ones(inh)
1142         self.S[:, ex:] = -torch.rand(n, inh)
1143         self.excitatory[ex:] = 0
1144
1145         self.register_buffer("v", self.rest * torch.ones(n)) #
Neuron voltages.
1146         self.register_buffer("u", self.b * self.v) # Neuron
recovery.
1147
1148     def forward(self, x: torch.Tensor) -> None:
1149         # language=rst
1150         """
1151         Runs a single simulation step.
1152
1153         :param x: Inputs to the layer.
1154         """
1155         # Check for spiking neurons.
1156         self.s = self.v >= self.thresh
1157
1158         # Voltage and recovery reset.
1159         self.v = torch.where(self.s, self.c, self.v)
1160         self.u = torch.where(self.s, self.u + self.d, self.u)
1161
1162         # Add inter-columnar input.
1163         if self.s.any():
1164             x += torch.cat(
1165                 [self.S[:, self.s[i]].sum(dim=1)[None] for i in
range(self.s.shape[0])],
1166                 dim=0,
1167             )

```

```

1168
1169     # Apply v and u updates.
1170     self.v += self.dt * 0.5 * (0.04 * self.v ** 2 + 5 * self.v +
140 - self.u + x)
1171     self.v += self.dt * 0.5 * (0.04 * self.v ** 2 + 5 * self.v +
140 - self.u + x)
1172     self.u += self.dt * self.a * (self.b * self.v - self.u)
1173
1174     # Voltage clipping to lower bound.
1175     if self.lbound is not None:
1176         self.v.masked_fill_(self.v < self.lbound, self.lbound)
1177
1178     super().forward(x)
1179
1180     def reset_state_variables(self) -> None:
1181         # language=rst
1182         """
1183         Resets relevant state variables.
1184         """
1185         super().reset_state_variables()
1186         self.v.fill_(self.rest) # Neuron voltages.
1187         self.u = self.b * self.v # Neuron recovery.
1188
1189     def set_batch_size(self, batch_size) -> None:
1190         # language=rst
1191         """
1192         Sets mini-batch size. Called when layer is added to a
1193         network.
1194
1195         :param batch_size: Mini-batch size.
1196         """
1197         super().set_batch_size(batch_size=batch_size)
1198         self.v = self.rest * torch.ones(batch_size, *self.shape,
1199         device=self.v.device)
1200         self.u = self.b * self.v
1201
1202     class SRMONodes(Nodes):
1203         # language=rst
1204         """
1205         Layer of simplified spike response model (SRM0) neurons with
1206         stochastic threshold (escape noise). Adapted from
1207         '(Vasilaki et al., 2009) <https://intranet.physio.unibe.ch/
1208         Publikationen/Dokumente/Vasilaki2009PloSComputBio\_1.pdf>'.
1209         """
1210         def __init__(
1211             self,
1212             n: Optional[int] = None,
1213             shape: Optional[Iterable[int]] = None,
1214             traces: bool = False,
1215             traces_additive: bool = False,
1216             tc_trace: Union[float, torch.Tensor] = 20.0,
1217             trace_scale: Union[float, torch.Tensor] = 1.0,

```

```

1216     sum_input: bool = False,
1217     thresh: Union[float, torch.Tensor] = -50.0,
1218     rest: Union[float, torch.Tensor] = -70.0,
1219     reset: Union[float, torch.Tensor] = -70.0,
1220     refrac: Union[int, torch.Tensor] = 5,
1221     tc_decay: Union[float, torch.Tensor] = 10.0,
1222     lbound: float = None,
1223     eps_0: Union[float, torch.Tensor] = 1.0,
1224     rho_0: Union[float, torch.Tensor] = 1.0,
1225     d_thresh: Union[float, torch.Tensor] = 5.0,
1226     **kwargs,
1227 ) -> None:
1228     # language=rst
1229     """
1230     Instantiates a layer of SRMO neurons.
1231
1232     :param n: The number of neurons in the layer.
1233     :param shape: The dimensionality of the layer.
1234     :param traces: Whether to record spike traces.
1235     :param traces_additive: Whether to record spike traces
1236     additively.
1237     :param tc_trace: Time constant of spike trace decay.
1238     :param trace_scale: Scaling factor for spike trace.
1239     :param sum_input: Whether to sum all inputs.
1240     :param thresh: Spike threshold voltage.
1241     :param rest: Resting membrane voltage.
1242     :param reset: Post-spike reset voltage.
1243     :param refrac: Refractory (non-firing) period of the neuron.
1244     :param tc_decay: Time constant of neuron voltage decay.
1245     :param lbound: Lower bound of the voltage.
1246     :param eps_0: Scaling factor for pre-synaptic spike
1247     contributions.
1248     :param rho_0: Stochastic intensity at threshold.
1249     :param d_thresh: Width of the threshold region.
1250     """
1251     super().__init__(
1252         n=n,
1253         shape=shape,
1254         traces=traces,
1255         traces_additive=traces_additive,
1256         tc_trace=tc_trace,
1257         trace_scale=trace_scale,
1258         sum_input=sum_input,
1259     )
1260
1261     self.register_buffer("rest", torch.tensor(rest)) # Rest
1262     self.register_buffer("reset", torch.tensor(reset)) # Post-
1263     self.register_buffer("thresh", torch.tensor(thresh)) #
1264     self.register_buffer(
1265         "refrac", torch.tensor(refrac)) #
1266     self.register_buffer(
1267         "refrac", torch.tensor(refrac)) # Post-spike refractory period.

```

```

1265     self.register_buffer(
1266         "tc_decay", torch.tensor(tc_decay)
1267     ) # Time constant of neuron voltage decay.
1268     self.register_buffer("decay", torch.tensor(tc_decay)) # Set
in compute_decays.
1269     self.register_buffer(
1270         "eps_0", torch.tensor(eps_0)
1271     ) # Scaling factor for pre-synaptic spike contributions.
1272     self.register_buffer(
1273         "rho_0", torch.tensor(rho_0)
1274     ) # Stochastic intensity at threshold.
1275     self.register_buffer(
1276         "d_thresh", torch.tensor(d_thresh)
1277     ) # Width of the threshold region.
1278     self.register_buffer("v", torch.FloatTensor()) # Neuron
voltages.
1279     self.register_buffer(
1280         "refrac_count", torch.FloatTensor()
1281     ) # Refractory period counters.
1282
1283     self.lbound = lbound # Lower bound of voltage.
1284
1285     def forward(self, x: torch.Tensor) -> None:
1286         # language=rst
1287         """
1288         Runs a single simulation step.
1289
1290         :param x: Inputs to the layer.
1291         """
1292         # Decay voltages.
1293         self.v = self.decay * (self.v - self.rest) + self.rest
1294
1295         # Integrate inputs.
1296         self.v += (self.refrac_count == 0).float() * self.eps_0 * x
1297
1298         # Compute (instantaneous) probabilities of spiking, clamp
between 0 and 1 using exponentials.
1299         # Also known as 'escape noise', this simulates nearby
neurons.
1300         self.rho = self.rho_0 * torch.exp((self.v - self.thresh) /
self.d_thresh)
1301         self.s_prob = 1.0 - torch.exp(-self.rho * self.dt)
1302
1303         # Decrement refractory counters.
1304         self.refrac_count = (self.refrac_count > 0).float() * (
1305             self.refrac_count - self.dt
1306         )
1307
1308         # Check for spiking neurons (spike when probability > some
random number).
1309         self.s = torch.rand_like(self.s_prob) < self.s_prob
1310
1311         # Refractoriness and voltage reset.
1312         self.refrac_count.masked_fill_(self.s, self.refrac)

```



```

1313     self.v.masked_fill_(self.s, self.reset)
1314
1315     # Voltage clipping to lower bound.
1316     if self.lbound is not None:
1317         self.v.masked_fill_(self.v < self.lbound, self.lbound)
1318
1319     super().forward(x)
1320
1321     def reset_state_variables(self) -> None:
1322         # language=rst
1323         """
1324         Resets relevant state variables.
1325         """
1326         super().reset_state_variables()
1327         self.v.fill_(self.rest) # Neuron voltages.
1328         self.refrac_count.zero_() # Refractory period counters.
1329
1330     def compute_decays(self, dt) -> None:
1331         # language=rst
1332         """
1333         Sets the relevant decays.
1334         """
1335         super().compute_decays(dt=dt)
1336         self.decay = torch.exp(
1337             -self.dt / self.tc_decay
1338         ) # Neuron voltage decay (per timestep).
1339
1340     def set_batch_size(self, batch_size) -> None:
1341         # language=rst
1342         """
1343         Sets mini-batch size. Called when layer is added to a
1344         network.
1345
1346         :param batch_size: Mini-batch size.
1347         """
1347         super().set_batch_size(batch_size=batch_size)
1348         self.v = self.rest * torch.ones(batch_size, *self.shape,
1349             device=self.v.device)
1349         self.refrac_count = torch.zeros_like(self.v, device=self.
1350             refrac_count.device)

```

Listing B.7: Nodes

```

1 from abc import ABC, abstractmethod
2 from typing import Union, Tuple, Optional, Sequence
3
4 import numpy as np
5 import torch
6 from torch.nn import Module, Parameter
7 import torch.nn.functional as F
8 from torch.nn.modules.utils import _pair
9
10 from .nodes import Nodes
11

```

```

12
13 class AbstractConnection(ABC, Module):
14     # language=rst
15     """
16     Abstract base method for connections between ‘‘Nodes‘‘.
17     """
18
19     def __init__(
20         self,
21         source: Nodes,
22         target: Nodes,
23         nu: Optional[Union[float, Sequence[float]]] = None,
24         reduction: Optional[Callable] = None,
25         weight_decay: float = 0.0,
26         **kwargs
27     ) -> None:
28         # language=rst
29         """
30         Constructor for abstract base class for connection objects.
31
32         :param source: A layer of nodes from which the connection
33         originates.
34         :param target: A layer of nodes to which the connection
35         connects.
36         :param nu: Learning rate for both pre- and post-synaptic
37         events.
38         :param reduction: Method for reducing parameter updates
39         along the minibatch
40         dimension.
41         :param weight_decay: Constant multiple to decay weights by
42         on each iteration.
43
44         Keyword arguments:
45
46         :param LearningRule update_rule: Modifies connection
47         parameters according to
48         some rule.
49         :param float wmin: The minimum value on the connection
50         weights.
51         :param float wmax: The maximum value on the connection
52         weights.
53         :param float norm: Total weight per target neuron
54         normalization.
55         """
56         super().__init__()
57
58         assert isinstance(source, Nodes), "Source is not a Nodes
59         object"
60         assert isinstance(target, Nodes), "Target is not a Nodes
61         object"
62
63         self.source = source
64         self.target = target

```

```

55     self.nu = nu
56     self.weight_decay = weight_decay
57     self.reduction = reduction
58
59     from ..learning import NoOp
60
61     self.update_rule = kwargs.get("update_rule", NoOp)
62     self.wmin = kwargs.get("wmin", -np.inf)
63     self.wmax = kwargs.get("wmax", np.inf)
64     self.norm = kwargs.get("norm", None)
65     self.decay = kwargs.get("decay", None)
66
67     if self.update_rule is None:
68         self.update_rule = NoOp
69
70     self.update_rule = self.update_rule(
71         connection=self,
72         nu=nu,
73         reduction=reduction,
74         weight_decay=weight_decay,
75         **kwargs
76     )
77
78     @abstractmethod
79     def compute(self, s: torch.Tensor) -> None:
80         # language=rst
81         """
82         Compute pre-activations of downstream neurons given spikes
83         of upstream neurons.
84
85         :param s: Incoming spikes.
86         """
87         pass
88
89     @abstractmethod
90     def update(self, **kwargs) -> None:
91         # language=rst
92         """
93         Compute connection's update rule.
94
95         Keyword arguments:
96
97         :param bool learning: Whether to allow connection updates.
98         :param ByteTensor mask: Boolean mask determining which
99         weights to clamp to zero.
100         """
101         learning = kwargs.get("learning", True)
102
103         if learning:
104             self.update_rule.update(**kwargs)
105
106         mask = kwargs.get("mask", None)
107         if mask is not None:
108             self.w.masked_fill_(mask, 0)

```

```

107
108     @abstractmethod
109     def reset_state_variables(self) -> None:
110         # language=rst
111         """
112         Contains resetting logic for the connection.
113         """
114         pass
115
116
117 class Connection(AbstractConnection):
118     # language=rst
119     """
120     Specifies synapses between one or two populations of neurons.
121     """
122
123     def __init__(
124         self,
125         source: Nodes,
126         target: Nodes,
127         nu: Optional[Union[float, Sequence[float]]] = None,
128         reduction: Optional[Callable] = None,
129         weight_decay: float = 0.0,
130         **kwargs
131     ) -> None:
132         # language=rst
133         """
134         Instantiates a :code:`Connection` object.
135
136         :param source: A layer of nodes from which the connection
137         originates.
138         :param target: A layer of nodes to which the connection
139         connects.
140         :param nu: Learning rate for both pre- and post-synaptic
141         events.
142         :param reduction: Method for reducing parameter updates
143         along the minibatch
144         dimension.
145         :param weight_decay: Constant multiple to decay weights by
146         on each iteration.
147
148         Keyword arguments:
149
150         :param LearningRule update_rule: Modifies connection
151         parameters according to
152         some rule.
153         :param torch.Tensor w: Strengths of synapses.
154         :param torch.Tensor b: Target population bias.
155         :param float wmin: Minimum allowed value on the connection
156         weights.
157         :param float wmax: Maximum allowed value on the connection
158         weights.
159         :param float norm: Total weight per target neuron
160         normalization constant.

```

```

152     """
153     super().__init__(source, target, nu, reduction, weight_decay
154     , **kwargs)
155
156     w = kwargs.get("w", None)
157     if w is None:
158         if self.wmin == -np.inf or self.wmax == np.inf:
159             w = torch.clamp(torch.rand(source.n, target.n), self
160             .wmin, self.wmax)
161         else:
162             w = self.wmin + torch.rand(source.n, target.n) * (
163             self.wmax - self.wmin)
164         else:
165             if self.wmin != -np.inf or self.wmax != np.inf:
166                 w = torch.clamp(w, self.wmin, self.wmax)
167
168     self.w = Parameter(w, requires_grad=False)
169     self.b = Parameter(kwargs.get("b", torch.zeros(target.n)),
170     requires_grad=False)
171
172     def compute(self, s: torch.Tensor) -> torch.Tensor:
173         # language=rst
174         """
175         Compute pre-activations given spikes using connection
176         weights.
177
178         :param s: Incoming spikes.
179         :return: Incoming spikes multiplied by synaptic weights (
180         with or without
181         decaying spike activation).
182         """
183         # Compute multiplication of spike activations by weights and
184         add bias.
185         post = s.float().view(s.size(0), -1) @ self.w + self.b
186         return post.view(s.size(0), *self.target.shape)
187
188     def update(self, **kwargs) -> None:
189         # language=rst
190         """
191         Compute connection's update rule.
192         """
193         super().update(**kwargs)
194
195     def normalize(self) -> None:
196         # language=rst
197         """
198         Normalize weights so each target neuron has sum of
199         connection weights equal to
200         'self.norm'.
201         """
202         if self.norm is not None:
203             w_abs_sum = self.w.abs().sum(0).unsqueeze(0)
204             w_abs_sum[w_abs_sum == 0] = 1.0
205             self.w *= self.norm / w_abs_sum

```

```

198
199     def reset_state_variables(self) -> None:
200         # language=rst
201         """
202         Contains resetting logic for the connection.
203         """
204         super().reset_state_variables()
205
206
207 class Conv2dConnection(AbstractConnection):
208     # language=rst
209     """
210     Specifies convolutional synapses between one or two populations
211     of neurons.
212     """
213     def __init__(
214         self,
215         source: Nodes,
216         target: Nodes,
217         kernel_size: Union[int, Tuple[int, int]],
218         stride: Union[int, Tuple[int, int]] = 1,
219         padding: Union[int, Tuple[int, int]] = 0,
220         dilation: Union[int, Tuple[int, int]] = 1,
221         nu: Optional[Union[float, Sequence[float]]] = None,
222         reduction: Optional[callable] = None,
223         weight_decay: float = 0.0,
224         **kwargs
225     ) -> None:
226         # language=rst
227         """
228         Instantiates a ‘‘Conv2dConnection’’ object.
229
230         :param source: A layer of nodes from which the connection
231         originates.
232         :param target: A layer of nodes to which the connection
233         connects.
234         :param kernel_size: Horizontal and vertical size of
235         convolutional kernels.
236         :param stride: Horizontal and vertical stride for
237         convolution.
238         :param padding: Horizontal and vertical padding for
239         convolution.
240         :param dilation: Horizontal and vertical dilation for
241         convolution.
242         :param nu: Learning rate for both pre- and post-synaptic
243         events.
244         :param reduction: Method for reducing parameter updates
245         along the minibatch
246         dimension.
247         :param weight_decay: Constant multiple to decay weights by
248         on each iteration.
249
250         Keyword arguments:

```

```

242
243     :param LearningRule update_rule: Modifies connection
parameters according to
244         some rule.
245     :param torch.Tensor w: Strengths of synapses.
246     :param torch.Tensor b: Target population bias.
247     :param float wmin: Minimum allowed value on the connection
weights.
248     :param float wmax: Maximum allowed value on the connection
weights.
249     :param float norm: Total weight per target neuron
normalization constant.
250     """
251     super().__init__(source, target, nu, reduction, weight_decay
, **kwargs)
252
253     self.kernel_size = _pair(kernel_size)
254     self.stride = _pair(stride)
255     self.padding = _pair(padding)
256     self.dilation = _pair(dilation)
257
258     self.in_channels, input_height, input_width = (
259         source.shape[0],
260         source.shape[1],
261         source.shape[2],
262     )
263     self.out_channels, output_height, output_width = (
264         target.shape[0],
265         target.shape[1],
266         target.shape[2],
267     )
268
269     width = (
270         input_width - self.kernel_size[1] + 2 * self.padding[1]
271     ) / self.stride[1] + 1
272     height = (
273         input_height - self.kernel_size[0] + 2 * self.padding[0]
274     ) / self.stride[0] + 1
275     shape = (self.in_channels, self.out_channels, int(width),
int(height))
276
277     error = (
278         "Target dimensionality must be (out_channels, ?, "
279         "(input_height - filter_height + 2 * padding_height) /
stride_height + 1,"
280         "(input_width - filter_width + 2 * padding_width) /
stride_width + 1"
281     )
282
283     assert (
284         target.shape[0] == shape[1]
285         and target.shape[1] == shape[2]
286         and target.shape[2] == shape[3]
287     ), error

```

```

288
289     w = kwargs.get("w", None)
290     if w is None:
291         if self.wmin == -np.inf or self.wmax == np.inf:
292             w = torch.clamp(
293                 torch.rand(self.out_channels, self.in_channels,
294 *self.kernel_size),
295                 self.wmin,
296                 self.wmax,
297             )
298         else:
299             w = (self.wmax - self.wmin) * torch.rand(
300                 self.out_channels, self.in_channels, *self.
301 kernel_size
302             )
303             w += self.wmin
304         else:
305             if self.wmin != -np.inf or self.wmax != np.inf:
306                 w = torch.clamp(w, self.wmin, self.wmax)
307
308     self.w = Parameter(w, requires_grad=False)
309     self.b = Parameter(
310         kwargs.get("b", torch.zeros(self.out_channels)),
311         requires_grad=False
312     )
313
314     def compute(self, s: torch.Tensor) -> torch.Tensor:
315         # language=rst
316         """
317         Compute convolutional pre-activations given spikes using
318         layer weights.
319
320         :param s: Incoming spikes.
321         :return: Incoming spikes multiplied by synaptic weights (
322         with or without
323         decaying spike activation).
324         """
325         return F.conv2d(
326             s.float(),
327             self.w,
328             self.b,
329             stride=self.stride,
330             padding=self.padding,
331             dilation=self.dilation,
332         )
333
334     def update(self, **kwargs) -> None:
335         # language=rst
336         """
337         Compute connection's update rule.
338         """
339         super().update(**kwargs)
340
341     def normalize(self) -> None:

```



```

337     # language=rst
338     """
339     Normalize weights along the first axis according to total
weight per target
340     neuron.
341     """
342     if self.norm is not None:
343         # get a view and modify in place
344         w = self.w.view(
345             self.w.size(0) * self.w.size(1), self.w.size(2) *
self.w.size(3)
346         )
347
348         for fltr in range(w.size(0)):
349             w[fltr] *= self.norm / w[fltr].sum(0)
350
351     def reset_state_variables(self) -> None:
352         # language=rst
353         """
354         Contains resetting logic for the connection.
355         """
356         super().reset_state_variables()
357
358
359 class MaxPool2dConnection(AbstractConnection):
360     # language=rst
361     """
362     Specifies max-pooling synapses between one or two populations of
neurons by keeping
363     online estimates of maximally firing neurons.
364     """
365
366     def __init__(
367         self,
368         source: Nodes,
369         target: Nodes,
370         kernel_size: Union[int, Tuple[int, int]],
371         stride: Union[int, Tuple[int, int]] = 1,
372         padding: Union[int, Tuple[int, int]] = 0,
373         dilation: Union[int, Tuple[int, int]] = 1,
374         **kwargs
375     ) -> None:
376         # language=rst
377         """
378         Instantiates a ‘‘MaxPool2dConnection’’ object.
379
380         :param source: A layer of nodes from which the connection
originates.
381         :param target: A layer of nodes to which the connection
connects.
382         :param kernel_size: Horizontal and vertical size of
convolutional kernels.
383         :param stride: Horizontal and vertical stride for
convolution.

```

```

384         :param padding: Horizontal and vertical padding for
convolution.
385         :param dilation: Horizontal and vertical dilation for
convolution.
386
387         Keyword arguments:
388
389         :param decay: Decay rate of online estimates of average
firing activity.
390         """
391         super().__init__(source, target, None, None, 0.0, **kwargs)
392
393         self.kernel_size = _pair(kernel_size)
394         self.stride = _pair(stride)
395         self.padding = _pair(padding)
396         self.dilation = _pair(dilation)
397
398         self.register_buffer("firing_rates", torch.ones(source.shape
))
399
400     def compute(self, s: torch.Tensor) -> torch.Tensor:
401         # language=rst
402         """
403         Compute max-pool pre-activations given spikes using online
firing rate
404         estimates.
405
406         :param s: Incoming spikes.
407         :return: Incoming spikes multiplied by synaptic weights (
with or without
408             decaying spike activation).
409         """
410         self.firing_rates -= self.decay * self.firing_rates
411         self.firing_rates += s.float()
412
413         _, indices = F.max_pool2d(
414             self.firing_rates,
415             kernel_size=self.kernel_size,
416             stride=self.stride,
417             padding=self.padding,
418             dilation=self.dilation,
419             return_indices=True,
420         )
421
422         return s.take(indices).float()
423
424     def update(self, **kwargs) -> None:
425         # language=rst
426         """
427         Compute connection's update rule.
428         """
429         super().update(**kwargs)
430
431     def normalize(self) -> None:

```

```

432     # language=rst
433     """
434     No weights -> no normalization.
435     """
436     pass
437
438     def reset_state_variables(self) -> None:
439         # language=rst
440         """
441         Contains resetting logic for the connection.
442         """
443         super().reset_state_variables()
444
445         self.firing_rates = torch.zeros(self.source.shape)
446
447 class LocalConnection(AbstractConnection):
448     # language=rst
449     """
450     Specifies a locally connected connection between one or two
451     populations of neurons.
452     """
453     def __init__(
454         self,
455         source: Nodes,
456         target: Nodes,
457         kernel_size: Union[int, Tuple[int, int]],
458         stride: Union[int, Tuple[int, int]],
459         n_filters: int,
460         nu: Optional[Union[float, Sequence[float]]] = None,
461         reduction: Optional[callable] = None,
462         weight_decay: float = 0.0,
463         **kwargs
464     ) -> None:
465         # language=rst
466         """
467         Instantiates a ‘‘LocalConnection’’ object. Source population
468         should be
469         two-dimensional.
470
471         Neurons in the post-synaptic population are ordered by
472         receptive field; that is,
473         if there are ‘‘n_conv’’ neurons in each post-synaptic patch,
474         then the first
475         ‘‘n_conv’’ neurons in the post-synaptic population
476         correspond to the first
477         receptive field, the second ‘‘n_conv’’ to the second
478         receptive field, and so on.
479
480         :param source: A layer of nodes from which the connection
481         originates.
482         :param target: A layer of nodes to which the connection
483         connects.
484         :param kernel_size: Horizontal and vertical size of

```

```

convolutional kernels.
478     :param stride: Horizontal and vertical stride for
convolution.
479     :param n_filters: Number of locally connected filters per
pre-synaptic region.
480     :param nu: Learning rate for both pre- and post-synaptic
events.
481     :param reduction: Method for reducing parameter updates
along the minibatch
482         dimension.
483     :param weight_decay: Constant multiple to decay weights by
on each iteration.
484
485     Keyword arguments:
486
487     :param LearningRule update_rule: Modifies connection
parameters according to
488         some rule.
489     :param torch.Tensor w: Strengths of synapses.
490     :param torch.Tensor b: Target population bias.
491     :param float wmin: Minimum allowed value on the connection
weights.
492     :param float wmax: Maximum allowed value on the connection
weights.
493     :param float norm: Total weight per target neuron
normalization constant.
494     :param Tuple[int, int] input_shape: Shape of input
population if it's not
495         "[sqrt, sqrt]".
496     """
497     super().__init__(source, target, nu, reduction, weight_decay
, **kwargs)
498
499     kernel_size = _pair(kernel_size)
500     stride = _pair(stride)
501
502     self.kernel_size = kernel_size
503     self.stride = stride
504     self.n_filters = n_filters
505
506     shape = kwargs.get("input_shape", None)
507     if shape is None:
508         sqrt = int(np.sqrt(source.n))
509         shape = _pair(sqrt)
510
511     if kernel_size == shape:
512         conv_size = [1, 1]
513     else:
514         conv_size = (
515             int((shape[0] - kernel_size[0]) / stride[0]) + 1,
516             int((shape[1] - kernel_size[1]) / stride[1]) + 1,
517         )
518
519     self.conv_size = conv_size

```

```

520
521     conv_prod = int(np.prod(conv_size))
522     kernel_prod = int(np.prod(kernel_size))
523
524     assert (
525         target.n == n_filters * conv_prod
526     ), "Target layer size must be n_filters * (kernel_size ** 2)
527     ."
528
529     locations = torch.zeros(
530         kernel_size[0], kernel_size[1], conv_size[0], conv_size
531         [1]
532     ).long()
533     for c1 in range(conv_size[0]):
534         for c2 in range(conv_size[1]):
535             for k1 in range(kernel_size[0]):
536                 for k2 in range(kernel_size[1]):
537                     location = (
538                         c1 * stride[0] * shape[1]
539                         + c2 * stride[1]
540                         + k1 * shape[0]
541                         + k2
542                     )
543                     locations[k1, k2, c1, c2] = location
544
545     self.register_buffer("locations", locations.view(kernel_prod
546     , conv_prod))
547     w = kwargs.get("w", None)
548
549     if w is None:
550         w = torch.zeros(source.n, target.n)
551         for f in range(n_filters):
552             for c in range(conv_prod):
553                 for k in range(kernel_prod):
554                     if self.wmin == -np.inf or self.wmax == np.
555     inf:
556                         w[self.locations[k, c], f * conv_prod +
557     c] = np.clip(
558                             w[self.locations[k, c], f * conv_prod +
559     c],
560                             np.random.rand(), self.wmin, self.
561     wmax
562                             )
563                     else:
564                         w[
565     self.locations[k, c], f * conv_prod
566     + c
567                             ] = self.wmin + np.random.rand() * (self
568     .wmax - self.wmin)
569                     else:
570                         if self.wmin != -np.inf or self.wmax != np.inf:
571                             w = torch.clamp(w, self.wmin, self.wmax)
572
573     self.w = Parameter(w, requires_grad=False)
574
575     self.register_buffer("mask", self.w == 0)

```

```

566
567     self.b = Parameter(kwargs.get("b", torch.zeros(target.n)),
requires_grad=False)
568
569     if self.norm is not None:
570         self.norm *= kernel_prod
571
572     def compute(self, s: torch.Tensor) -> torch.Tensor:
573         # language=rst
574         """
575         Compute pre-activations given spikes using layer weights.
576
577         :param s: Incoming spikes.
578         :return: Incoming spikes multiplied by synaptic weights (
with or without
579             decaying spike activation).
580         """
581         # Compute multiplication of pre-activations by connection
weights.
582         if self.w.shape[0] == self.source.n and self.w.shape[1] ==
self.target.n:
583             return s.float().view(s.size(0), -1) @ self.w + self.b
584         else:
585             a_post = (
586                 s.float().view(s.size(0), -1)
587                 @ self.w.view(self.source.n, self.target.n)
588                 + self.b
589             )
590             return a_post.view(*self.target.shape)
591
592     def update(self, **kwargs) -> None:
593         # language=rst
594         """
595         Compute connection's update rule.
596
597         Keyword arguments:
598
599         :param ByteTensor mask: Boolean mask determining which
weights to clamp to zero.
600         """
601         if kwargs["mask"] is None:
602             kwargs["mask"] = self.mask
603
604         super().update(**kwargs)
605
606     def normalize(self) -> None:
607         # language=rst
608         """
609         Normalize weights so each target neuron has sum of
connection weights equal to
610         ‘self.norm‘.
611         """
612         if self.norm is not None:
613             w = self.w.view(self.source.n, self.target.n)

```

```

614         w *= self.norm / self.w.sum(0).view(1, -1)
615
616     def reset_state_variables(self) -> None:
617         # language=rst
618         """
619         Contains resetting logic for the connection.
620         """
621         super().reset_state_variables()
622
623
624 class MeanFieldConnection(AbstractConnection):
625     # language=rst
626     """
627     A connection between one or two populations of neurons which
628     computes a summary of
629     the pre-synaptic population to use as weighted input to the post
630     -synaptic
631     population.
632     """
633
634     def __init__(
635         self,
636         source: Nodes,
637         target: Nodes,
638         nu: Optional[Union[float, Sequence[float]]] = None,
639         weight_decay: float = 0.0,
640         **kwargs
641     ) -> None:
642         # language=rst
643         """
644         Instantiates a :code:`MeanFieldConnection` object.
645         :param source: A layer of nodes from which the connection
646         originates.
647         :param target: A layer of nodes to which the connection
648         connects.
649         :param nu: Learning rate for both pre- and post-synaptic
650         events.
651         :param weight_decay: Constant multiple to decay weights by
652         on each iteration.
653         Keyword arguments:
654         :param LearningRule update_rule: Modifies connection
655         parameters according to
656         some rule.
657         :param torch.Tensor w: Strengths of synapses.
658         :param float wmin: Minimum allowed value on the connection
659         weights.
660         :param float wmax: Maximum allowed value on the connection
661         weights.
662         :param float norm: Total weight per target neuron
663         normalization constant.
664         """
665         super().__init__(source, target, nu, weight_decay, **kwargs)
666
667         w = kwargs.get("w", None)

```

```

658     if w is None:
659         if self.wmin == -np.inf or self.wmax == np.inf:
660             w = torch.clamp((torch.randn(1)[0] + 1) / 10, self.
wmin, self.wmax)
661         else:
662             w = self.wmin + ((torch.randn(1)[0] + 1) / 10) * (
self.wmax - self.wmin)
663         else:
664             if self.wmin != -np.inf or self.wmax != np.inf:
665                 w = torch.clamp(w, self.wmin, self.wmax)
666
667         self.w = Parameter(w, requires_grad=False)
668
669     def compute(self, s: torch.Tensor) -> torch.Tensor:
670         # language=rst
671         """
672         Compute pre-activations given spikes using layer weights.
673         :param s: Incoming spikes.
674         :return: Incoming spikes multiplied by synaptic weights (
with or without
675             decaying spike activation).
676         """
677         # Compute multiplication of mean-field pre-activation by
connection weights.
678         return s.float().mean() * self.w
679
680     def update(self, **kwargs) -> None:
681         # language=rst
682         """
683         Compute connection's update rule.
684         """
685         super().update(**kwargs)
686
687     def normalize(self) -> None:
688         # language=rst
689         """
690         Normalize weights so each target neuron has sum of
connection weights equal to
691         'self.norm'.
692         """
693         if self.norm is not None:
694             self.w = self.w.view(1, self.target.n)
695             self.w *= self.norm / self.w.sum()
696             self.w = self.w.view(1, *self.target.shape)
697
698     def reset_state_variables(self) -> None:
699         # language=rst
700         """
701         Contains resetting logic for the connection.
702         """
703         super().reset_state_variables()
704
705
706 class SparseConnection(AbstractConnection):

```



```

707     # language=rst
708     """
709     Specifies sparse synapses between one or two populations of
710     neurons.
711     """
712     def __init__(
713         self,
714         source: Nodes,
715         target: Nodes,
716         nu: Optional[Union[float, Sequence[float]]] = None,
717         reduction: Optional[Callable] = None,
718         weight_decay: float = None,
719         **kwargs
720     ) -> None:
721         # language=rst
722         """
723         Instantiates a :code:`Connection` object with sparse weights
724         .
725         :param source: A layer of nodes from which the connection
726         originates.
727         :param target: A layer of nodes to which the connection
728         connects.
729         :param nu: Learning rate for both pre- and post-synaptic
730         events.
731         :param reduction: Method for reducing parameter updates
732         along the minibatch
733         dimension.
734         :param weight_decay: Constant multiple to decay weights by
735         on each iteration.
736
737         Keyword arguments:
738
739         :param torch.Tensor w: Strengths of synapses.
740         :param float sparsity: Fraction of sparse connections to use
741         .
742         :param LearningRule update_rule: Modifies connection
743         parameters according to
744         some rule.
745         :param float wmin: Minimum allowed value on the connection
746         weights.
747         :param float wmax: Maximum allowed value on the connection
748         weights.
749         :param float norm: Total weight per target neuron
750         normalization constant.
751         """
752         super().__init__(source, target, nu, reduction, weight_decay
753         , **kwargs)
754
755         w = kwargs.get("w", None)
756         self.sparsity = kwargs.get("sparsity", None)
757
758         assert (

```

```

748         w is not None
749         and self.sparsity is None
750         or w is None
751         and self.sparsity is not None
752     ), 'Only one of "weights" or "sparsity" must be specified'
753
754     if w is None and self.sparsity is not None:
755         i = torch.bernoulli(
756             1 - self.sparsity * torch.ones(*source.shape, *
target.shape)
757         )
758         if self.wmin == -np.inf or self.wmax == np.inf:
759             v = torch.clamp(
760                 torch.rand(*source.shape, *target.shape)[i.byte
761                 ],
762                 self.wmin,
763                 self.wmax,
764             )
765         else:
766             v = self.wmin + torch.rand(*source.shape, *target.
shape)[i.byte()] * (
767                 self.wmax - self.wmin
768             )
769         w = torch.sparse.FloatTensor(i.nonzero().t(), v)
770     elif w is not None and self.sparsity is None:
771         assert w.is_sparse, "Weight matrix is not sparse (see
torch.sparse module)"
772         if self.wmin != -np.inf or self.wmax != np.inf:
773             w = torch.clamp(w, self.wmin, self.wmax)
774
775     self.w = Parameter(w, requires_grad=False)
776
777     def compute(self, s: torch.Tensor) -> torch.Tensor:
778         # language=rst
779         """
780         Compute convolutional pre-activations given spikes using
781         layer weights.
782         :param s: Incoming spikes.
783         :return: Incoming spikes multiplied by synaptic weights (
with or without
784             decaying spike activation).
785         """
786         return torch.mm(self.w, s.unsqueeze(-1).float()).squeeze(-1)
787
788     def update(self, **kwargs) -> None:
789         # language=rst
790         """
791         Compute connection's update rule.
792         """
793         pass
794
795     def normalize(self) -> None:
796         # language=rst

```

```

796     """
797     Normalize weights along the first axis according to total
weight per target
798     neuron.
799     """
800     pass
801
802     def reset_state_variables(self) -> None:
803         # language=rst
804         """
805         Contains resetting logic for the connection.
806         """
807         super().reset_state_variables()

```

Listing B.8: Topology

```

1  import os
2  import torch
3  import numpy as np
4
5  from abc import ABC
6  from typing import Union, Optional, Iterable, Dict
7
8  from .nodes import Nodes
9  from .topology import AbstractConnection
10
11
12  class AbstractMonitor(ABC):
13     # language=rst
14     """
15     Abstract base class for state variable monitors.
16     """
17
18
19  class Monitor(AbstractMonitor):
20     # language=rst
21     """
22     Records state variables of interest.
23     """
24
25     def __init__(
26         self,
27         obj: Union[Nodes, AbstractConnection],
28         state_vars: Iterable[str],
29         time: Optional[int] = None,
30         batch_size: int = 1,
31     ):
32         # language=rst
33         """
34         Constructs a ‘‘Monitor’’ object.
35
36         :param obj: An object to record state variables from during
network simulation.
37         :param state_vars: Iterable of strings indicating names of

```

```

state variables to
    record.
38
    :param time: If not 'None', pre-allocate memory for state
39 variable recording.
    """
40
    super().__init__()
41
42
    self.obj = obj
43
    self.state_vars = state_vars
44
    self.time = time
45
    self.batch_size = batch_size
46
47
    # Deal with time later, the same underlying list is used
48 self.recording = {v: [] for v in self.state_vars}
49
50
51 def get(self, var: str) -> torch.Tensor:
52     # language=rst
53     """
54     Return recording to user.
55
56     :param var: State variable recording to return.
57     :return: Tensor of shape '[time, n_1, ..., n_k]', where
    '[n_1, ..., n_k]' is
58         the shape of the recorded state variable.
59     """
60     return torch.cat(self.recording[var], 0)
61
62 def record(self) -> None:
63     # language=rst
64     """
65     Appends the current value of the recorded state variables to
66     the recording.
67     """
68     for v in self.state_vars:
69         data = getattr(self.obj, v).unsqueeze(0)
70         self.recording[v].append(data.detach().clone())
71
72     # remove the oldest element (first in the list)
73     if self.time is not None:
74         for v in self.state_vars:
75             if len(self.recording[v]) > self.time:
76                 self.recording[v].pop(0)
77
78 def reset_state_variables(self) -> None:
79     # language=rst
80     """
81     Resets recordings to empty 'torch.Tensor's.
82     """
83     self.recording = {v: [] for v in self.state_vars}
84
85 class NetworkMonitor(AbstractMonitor):
86     # language=rst
87     """

```

```

88     Record state variables of all layers and connections.
89     """
90
91     def __init__(
92         self,
93         network: "Network",
94         layers: Optional[Iterable[str]] = None,
95         connections: Optional[Iterable[str]] = None,
96         state_vars: Optional[Iterable[str]] = None,
97         time: Optional[int] = None,
98     ):
99         # language=rst
100         """
101         Constructs a ‘‘NetworkMonitor‘‘ object.
102
103         :param network: Network to record state variables from.
104         :param layers: Layers to record state variables from.
105         :param connections: Connections to record state variables
106         from.
107         :param state_vars: List of strings indicating names of state
108         variables to
109             record.
110         :param time: If not ‘‘None‘‘, pre-allocate memory for state
111         variable recording.
112         """
113         super().__init__()
114
115         self.network = network
116         self.layers = layers if layers is not None else list(self.
117         network.layers.keys())
118         self.connections = (
119             connections
120             if connections is not None
121             else list(self.network.connections.keys())
122         )
123         self.state_vars = state_vars if state_vars is not None else
124         ("v", "s", "w")
125         self.time = time
126
127         if self.time is not None:
128             self.i = 0
129
130         # Initialize empty recording.
131         self.recording = {k: {} for k in self.layers + self.
132         connections}
133
134         # If no simulation time is specified, specify 0-dimensional
135         recordings.
136         if self.time is None:
137             for v in self.state_vars:
138                 for l in self.layers:
139                     if hasattr(self.network.layers[l], v):
140                         self.recording[l][v] = torch.Tensor()

```

```

135         for c in self.connections:
136             if hasattr(self.network.connections[c], v):
137                 self.recording[c][v] = torch.Tensor()
138
139         # If simulation time is specified, pre-allocate recordings
140         # in memory for speed.
141         else:
142             for v in self.state_vars:
143                 for l in self.layers:
144                     if hasattr(self.network.layers[l], v):
145                         self.recording[l][v] = torch.zeros(
146                             self.time, *getattr(self.network.layers[
147                                 l], v).size()
148                             )
149
150             for c in self.connections:
151                 if hasattr(self.network.connections[c], v):
152                     self.recording[c][v] = torch.zeros(
153                         self.time, *getattr(self.network.
154                             connections[c], v).size()
155                     )
156
157         def get(self) -> Dict[str, Dict[str, Union[Nodes,
158             AbstractConnection]]]:
159             # language=rst
160             """
161             Return entire recording to user.
162
163             :return: Dictionary of dictionary of all layers' and
164             connections' recorded
165             state variables.
166             """
167             return self.recording
168
169         def record(self) -> None:
170             # language=rst
171             """
172             Appends the current value of the recorded state variables to
173             the recording.
174             """
175             if self.time is None:
176                 for v in self.state_vars:
177                     for l in self.layers:
178                         if hasattr(self.network.layers[l], v):
179                             data = getattr(self.network.layers[l], v).
180                                 unsqueeze(0).float()
181                             self.recording[l][v] = torch.cat(
182                                 (self.recording[l][v], data), 0
183                             )
184
185                 for c in self.connections:
186                     if hasattr(self.network.connections[c], v):
187                         data = getattr(self.network.connections[c],
188                             v).unsqueeze(0)

```

```

181         self.recording[c][v] = torch.cat(
182             (self.recording[c][v], data), 0
183         )
184
185     else:
186         for v in self.state_vars:
187             for l in self.layers:
188                 if hasattr(self.network.layers[l], v):
189                     data = getattr(self.network.layers[l], v).
float().unsqueeze(0)
190                     self.recording[l][v] = torch.cat(
191                         (self.recording[l][v][1:].type(data.type
()), data), 0
192                     )
193
194                 for c in self.connections:
195                     if hasattr(self.network.connections[c], v):
196                         data = getattr(self.network.connections[c],
v).unsqueeze(0)
197                         self.recording[c][v] = torch.cat(
198                             (self.recording[c][v][1:].type(data.type
()), data), 0
199                         )
200
201                 self.i += 1
202
203     def save(self, path: str, fmt: str = "npz") -> None:
204         # language=rst
205         """
206         Write the recording dictionary out to file.
207
208         :param path: The directory to which to write the monitor's
recording.
209         :param fmt: Type of file to write to disk. One of '"pickle
"' or '"npz"'.
210         """
211         if not os.path.exists(os.path.dirname(path)):
212             os.makedirs(os.path.dirname(path))
213
214         if fmt == "npz":
215             # Build a list of arrays to write to disk.
216             arrays = {}
217             for o in self.recording:
218                 if type(o) == tuple:
219                     arrays.update(
220                         {
221                             "_".join(["-".join(o), v]): self.
recording[o][v]
222                             for v in self.recording[o]
223                         }
224                     )
225                 elif type(o) == str:
226                     arrays.update(
227                         {

```

```

228         "_".join([o, v]): self.recording[o][v]
229         for v in self.recording[o]
230     }
231 )
232
233     np.savez_compressed(path, **arrays)
234
235     elif fmt == "pickle":
236         with open(path, "wb") as f:
237             torch.save(self.recording, f)
238
239     def reset_state_variables(self) -> None:
240         # language=rst
241         """
242         Resets recordings to empty ‘‘torch.Tensors‘‘.
243         """
244         # Reset to empty recordings
245         self.recording = {k: {} for k in self.layers + self.
connections}
246
247         if self.time is not None:
248             self.i = 0
249
250         # If no simulation time is specified, specify 0-dimensional
recordings.
251         if self.time is None:
252             for v in self.state_vars:
253                 for l in self.layers:
254                     if hasattr(self.network.layers[l], v):
255                         self.recording[l][v] = torch.Tensor()
256
257                 for c in self.connections:
258                     if hasattr(self.network.connections[c], v):
259                         self.recording[c][v] = torch.Tensor()
260
261         # If simulation time is specified, pre-allocate recordings
in memory for speed.
262         else:
263             for v in self.state_vars:
264                 for l in self.layers:
265                     if hasattr(self.network.layers[l], v):
266                         self.recording[l][v] = torch.zeros(
267                             self.time, *getattr(self.network.layers[
268                             l], v).size()
269                             )
270
271                 for c in self.connections:
272                     if hasattr(self.network.connections[c], v):
273                         self.recording[c][v] = torch.zeros(
274                             self.time, *getattr(self.network.layers[

```

Listing B.9: Monitors

B.4 Pipeline

```

1 from .environment_pipeline import EnvironmentPipeline
2 from .base_pipeline import BasePipeline
3 from .dataloader_pipeline import DataLoaderPipeline,
  TorchVisionDatasetPipeline
4 from . import action

```

Listing B.10: Initialization

```

1 import itertools
2 from typing import Callable, Optional, Tuple, Dict
3
4 import torch
5
6 from .base_pipeline import BasePipeline
7 from ..analysis.pipeline_analysis import MatplotlibAnalyzer
8 from ..environment import Environment
9 from ..network import Network
10 from ..network.nodes import AbstractInput
11 from ..network.monitors import Monitor
12
13
14 class EnvironmentPipeline(BasePipeline):
15     # language=rst
16     """
17     Abstracts the interaction between ‘‘Network‘‘, ‘‘Environment‘‘,
18     and environment
19     feedback action.
20     """
21
22     def __init__(
23         self,
24         network: Network,
25         environment: Environment,
26         action_function: Optional[Callable] = None,
27         **kwargs,
28     ):
29         # language=rst
30         """
31         Initializes the pipeline.
32
33         :param network: Arbitrary network object.
34         :param environment: Arbitrary environment.
35         :param action_function: Function to convert network outputs
36         into environment
37         inputs.
38
39         Keyword arguments:
40
41         :param int num_episodes: Number of episodes to train for.
42         Defaults to 100.
43         :param str output: String name of the layer from which to
44         take output.

```

```

41     :param int render_interval: Interval to render the
environment.
42     :param int reward_delay: How many iterations to delay
delivery of reward.
43     :param int time: Time for which to run the network. Defaults
to the network's
44         timestep.
45     """
46     super().__init__(network, **kwargs)
47
48     self.episode = 0
49
50     self.env = environment
51     self.action_function = action_function
52
53     self.accumulated_reward = 0.0
54     self.reward_list = []
55
56     # Setting kwargs.
57     self.num_episodes = kwargs.get("num_episodes", 100)
58     self.output = kwargs.get("output", None)
59     self.render_interval = kwargs.get("render_interval", None)
60     self.reward_delay = kwargs.get("reward_delay", None)
61     self.time = kwargs.get("time", int(network.dt))
62
63     if self.reward_delay is not None:
64         assert self.reward_delay > 0
65         self.rewards = torch.zeros(self.reward_delay)
66
67     # Set up for multiple layers of input layers.
68     self.inputs = [
69         name
70         for name, layer in network.layers.items()
71         if isinstance(layer, AbstractInput)
72     ]
73
74     self.action = None
75
76     self.voltage_record = None
77     self.threshold_value = None
78     self.reward_plot = None
79
80     self.first = True
81     self.analyzer = MatplotlibAnalyzer(**self.plot_config)
82
83     if self.output is not None:
84         self.network.add_monitor(
85             Monitor(self.network.layers[self.output], ["s"]),
self.output
86         )
87
88         self.spike_record = {
89             self.output: torch.zeros((self.time, self.env.
action_space.n))

```

```

90         }
91
92     def init_fn(self) -> None:
93         pass
94
95     def train(self, **kwargs) -> None:
96         # language=rst
97         """
98         Trains for the specified number of episodes. Each episode
99         can be of arbitrary
100        length.
101        """
102        while self.episode < self.num_episodes:
103            self.reset_state_variables()
104
105            for _ in itertools.count():
106                obs, reward, done, info = self.env_step()
107
108                self.step((obs, reward, done, info), **kwargs)
109
110                if done:
111                    break
112
113                print(
114                    f"Episode: {self.episode} - "
115                    f"accumulated reward: {self.accumulated_reward:.2f}"
116                )
117                self.episode += 1
118
119     def env_step(self) -> Tuple[torch.Tensor, float, bool, Dict]:
120         # language=rst
121         """
122         Single step of the environment which includes rendering,
123         getting and performing
124         the action, and accumulating/delaying rewards.
125
126         :return: An OpenAI ‘‘gym’’ compatible tuple with modified
127         reward and info.
128         """
129         # Render game.
130         if (
131             self.render_interval is not None
132             and self.step_count % self.render_interval == 0
133         ):
134             self.env.render()
135
136         # Choose action based on output neuron spiking.
137         if self.action_function is not None:
138             self.action = self.action_function(self, output=self.
139             output)

```

```

140     # Set reward in case of delay.
141     if self.reward_delay is not None:
142         self.rewards = torch.tensor([reward, *self.rewards[1:]])
143     .float()
144         reward = self.rewards[-1]
145
146     # Accumulate reward.
147     self.accumulated_reward += reward
148
149     info["accumulated_reward"] = self.accumulated_reward
150
151     return obs, reward, done, info
152
153 def step_(
154     self, gym_batch: Tuple[torch.Tensor, float, bool, Dict], **
155     kwargs
156 ) -> None:
157     # language=rst
158     """
159     Run a single iteration of the network and update it and the
160     reward list when
161     done.
162
163     :param gym_batch: An OpenAI ‘‘gym’’ compatible tuple.
164     """
165     obs, reward, done, info = gym_batch
166
167     # Place the observations into the inputs.
168     obs_shape = [1] * len(obs.shape[1:])
169     inputs = {k: obs.repeat(self.time, *obs_shape) for k in self
170     .inputs}
171
172     # Run the network on the spike train-encoded inputs.
173     self.network.run(inputs=inputs, time=self.time, reward=
174     reward, **kwargs)
175
176     if self.output is not None:
177         self.spike_record[self.output] = (
178             self.network.monitors[self.output].get("s").float()
179         )
180
181     if done:
182         if self.network.reward_fn is not None:
183             self.network.reward_fn.update(
184                 accumulated_reward=self.accumulated_reward,
185                 steps=self.step_count,
186                 **kwargs,
187             )
188         self.reward_list.append(self.accumulated_reward)
189
190 def reset_state_variables(self) -> None:
191     # language=rst
192     """
193     Reset the pipeline.

```

```

189     """
190     self.env.reset()
191     self.network.reset_state_variables()
192     self.accumulated_reward = 0.0
193     self.step_count = 0
194
195     def plots(self, gym_batch: Tuple[torch.Tensor, float, bool, Dict
196 ], *args) -> None:
197         # language=rst
198         """
199         Plot the encoded input, layer spikes, and layer voltages.
200
201         :param gym_batch: An OpenAI ‘‘gym’’ compatible tuple.
202         """
203         obs, reward, done, info = gym_batch
204
205         for key, item in self.plot_config.items():
206             if key == "obs_step" and item is not None:
207                 if self.step_count % item == 0:
208                     self.analyzer.plot_obs(obs[0, ...].sum(0))
209             elif key == "data_step" and item is not None:
210                 if self.step_count % item == 0:
211                     self.analyzer.plot_spikes(self.get_spike_data())
212                     self.analyzer.plot_voltages(*self.
213 get_voltage_data())
214             elif key == "reward_eps" and item is not None:
215                 if self.episode % item == 0 and done:
216                     self.analyzer.plot_reward(self.reward_list)
217
218     self.analyzer.finalize_step()

```

Listing B.11: Environmental pipeline

```

1 import time
2 from typing import Tuple, Dict, Any
3
4 import torch
5 from torch._six import container_abcs, string_classes
6
7 from ..network import Network
8 from ..network.monitors import Monitor
9
10
11 def recursive_to(item, device):
12     # language=rst
13     """
14     Recursively transfers everything contained in item to the target
15     device.
16
17     :param item: An individual tensor or container of tensors.
18     :param device: ‘‘torch.device’’ pointing to ‘‘"cuda"’’ or ‘‘"cpu"
19     ’’.
20
21     :return: A version of the item that has been sent to a device.

```

```

21     """
22
23     if isinstance(item, torch.Tensor):
24         return item.to(device)
25     elif isinstance(item, (string_classes, int, float, bool)):
26         return item
27     elif isinstance(item, container_abcs.Mapping):
28         return {key: recursive_to(item[key], device) for key in item
29 }
30     elif isinstance(item, tuple) and hasattr(item, "_fields"):
31         return type(item)(*recursive_to(i, device) for i in item)
32     elif isinstance(item, container_abcs.Sequence):
33         return [recursive_to(i, device) for i in item]
34     else:
35         raise NotImplementedError(f"Target type {type(item)} not
36 supported.")
37
38 class BasePipeline:
39     # language=rst
40     """
41     A generic pipeline that handles high level functionality.
42     """
43
44     def __init__(self, network: Network, **kwargs) -> None:
45         # language=rst
46         """
47         Initializes the pipeline.
48
49         :param network: Arbitrary network object, will be managed by
50 the
51         ‘‘BasePipeline’’ class.
52
53         Keyword arguments:
54
55         :param int save_interval: How often to save the network to
56 disk.
57         :param str save_dir: Directory to save network object to.
58         :param Dict[str, Any] plot_config: Dict containing the plot
59 configuration.
60         Includes length, type (‘‘color’’ or ‘‘line’’), and
61 interval per plot
62         type.
63         :param int print_interval: Interval to print text output.
64         :param bool allow_gpu: Allows automatic transfer to the GPU.
65 """
66         self.network = network
67
68         # Network saving handles caching of intermediate results.
69         self.save_dir = kwargs.get("save_dir", "network.pt")
70         self.save_interval = kwargs.get("save_interval", None)
71
72         # Handles plotting of all layer spikes and voltages.
73         # This constructs monitors at every level.

```

```

69     self.plot_config = kwargs.get(
70         "plot_config", {"data_step": None, "data_length": 10}
71     )
72
73     if self.plot_config["data_step"] is not None:
74         for l in self.network.layers:
75             self.network.add_monitor(
76                 Monitor(
77                     self.network.layers[l], "s", self.
plot_config["data_length"]
78                 ),
79                 name=f"{l}_spikes",
80             )
81             if hasattr(self.network.layers[l], "v"):
82                 self.network.add_monitor(
83                     Monitor(
84                         self.network.layers[l], "v", self.
plot_config["data_length"]
85                     ),
86                     name=f"{l}_voltages",
87                 )
88
89     self.print_interval = kwargs.get("print_interval", None)
90     self.test_interval = kwargs.get("test_interval", None)
91     self.step_count = 0
92     self.init_fn()
93     self.clock = time.time()
94     self.allow_gpu = kwargs.get("allow_gpu", True)
95
96     if torch.cuda.is_available() and self.allow_gpu:
97         self.device = torch.device("cuda")
98     else:
99         self.device = torch.device("cpu")
100
101     self.network.to(self.device)
102
103     def reset_state_variables(self) -> None:
104         # language=rst
105         """
106         Reset the pipeline.
107         """
108         self.network.reset_state_variables()
109         self.step_count = 0
110
111     def step(self, batch: Any, **kwargs) -> Any:
112         # language=rst
113         """
114         Single step of any pipeline at a high level.
115
116         :param batch: A batch of inputs to be handed to the ‘‘step_
()’’ function.
117
118         Standard in subclasses of ‘‘BasePipeline‘‘.
119         :return: The output from the subclass’s ‘‘step_()’’ method,
which could be

```

```

119         anything. Passed to plotting to accommodate this.
120         """
121         self.step_count += 1
122
123         batch = recursive_to(batch, self.device)
124         step_out = self.step_(batch, **kwargs)
125
126         if (
127             self.print_interval is not None
128             and self.step_count % self.print_interval == 0
129         ):
130             print(
131                 f"Iteration: {self.step_count} (Time: {time.time() -
self.clock:.4f})"
132             )
133             self.clock = time.time()
134
135             self.plots(batch, step_out)
136
137             if self.save_interval is not None and self.step_count % self
.save_interval == 0:
138                 self.network.save(self.save_dir)
139
140             if self.test_interval is not None and self.step_count % self
.test_interval == 0:
141                 self.test()
142
143             return step_out
144
145         def get_spike_data(self) -> Dict[str, torch.Tensor]:
146             # language=rst
147             """
148             Get the spike data from all layers in the pipeline's network
149             .
150             :return: A dictionary containing all spike monitors from the
network.
151             """
152             return {
153                 l: self.network.monitors[f"{l}_spikes"].get("s")
154                 for l in self.network.layers
155             }
156
157         def get_voltage_data(
158             self
159         ) -> Tuple[Dict[str, torch.Tensor], Dict[str, torch.Tensor]]:
160             # language=rst
161             """
162             Get the voltage data and threshold value from all applicable
layers in the
163             pipeline's network.
164
165             :return: Two dictionaries containing the voltage data and
threshold values from

```



```

166         the network.
167         """
168         voltage_record = {}
169         threshold_value = {}
170         for l in self.network.layers:
171             if hasattr(self.network.layers[l], "v"):
172                 voltage_record[l] = self.network.monitors[f"{l}
173 _voltages"].get("v")
174             if hasattr(self.network.layers[l], "thresh"):
175                 threshold_value[l] = self.network.layers[l].thresh
176
177         return voltage_record, threshold_value
178
179     def step_(self, batch: Any, **kwargs) -> Any:
180         # language=rst
181         """
182         Perform a pass of the network given the input batch.
183
184         :param batch: The current batch. This could be anything as
185 long as the subclass
186         agrees upon the format in some way.
187         :return: Any output that is need for recording purposes.
188         """
189         raise NotImplementedError("You need to provide a step_
190 method.")
191
192     def train(self) -> None:
193         # language=rst
194         """
195         A fully self-contained training loop.
196         """
197         raise NotImplementedError("You need to provide a train
198 method.")
199
200     def test(self) -> None:
201         # language=rst
202         """
203         A fully self contained test function.
204         """
205         raise NotImplementedError("You need to provide a test method
206 .")
207
208     def init_fn(self) -> None:
209         # language=rst
210         """
211         Placeholder function for subclass-specific actions that need
212 to
213 happen during the construction of the ‘‘BasePipeline‘‘.
214         """
215         raise NotImplementedError("You need to provide an init_fn
216 method.")
217
218     def plots(self, batch: Any, step_out: Any) -> None:
219         # language=rst

```

```

213     """
214     Create any plots and logs for a step given the input batch
    and step output.
215
216     :param batch: The current batch. This could be anything as
    long as the subclass
217         agrees upon the format in some way.
218     :param step_out: The output from the ‘‘step_()’’ method.
219     """
220     raise NotImplementedError("You need to provide a plots
    method.")

```

Listing B.12: Base pipeline

```

1 from typing import Optional, Dict
2
3 import torch
4 from torch.utils.data import Dataset
5 from tqdm import tqdm
6
7 from ..network import Network
8 from .base_pipeline import BasePipeline
9 from ..analysis.pipeline_analysis import PipelineAnalyzer
10 from ..datasets import DataLoader
11
12
13 class DataLoaderPipeline(BasePipeline):
14     # language=rst
15     """
16     A generic ‘‘DataLoader’’ pipeline that leverages the ‘‘torch.
    utils.data’’ setup.
17     This still needs to be subclassed for specific implementations
    for functions given
18     the dataset that will be used. An example can be seen in
19     ‘‘TorchVisionDatasetPipeline’’.
20     """
21
22     def __init__(
23         self,
24         network: Network,
25         train_ds: Dataset,
26         test_ds: Optional[Dataset] = None,
27         **kwargs
28     ) -> None:
29         # language=rst
30         """
31         Initializes the pipeline.
32
33         :param network: Arbitrary ‘‘network’’ object.
34         :param train_ds: Arbitrary ‘‘torch.utils.data.Dataset’’
    object.
35         :param test_ds: Arbitrary ‘‘torch.utils.data.Dataset’’
    object.
36         """

```

```

37     super().__init__(network, **kwargs)
38
39     self.train_ds = train_ds
40     self.test_ds = test_ds
41
42     self.num_epochs = kwargs.get("num_epochs", 10)
43     self.batch_size = kwargs.get("batch_size", 1)
44     self.num_workers = kwargs.get("num_workers", 0)
45     self.pin_memory = kwargs.get("pin_memory", True)
46     self.shuffle = kwargs.get("shuffle", True)
47
48     def train(self) -> None:
49         # language=rst
50         """
51         Training loop that runs for the set number of epochs and
52         creates a new
53         ‘DataLoader‘ at each epoch.
54         """
55         for epoch in range(self.num_epochs):
56             train_dataloader = DataLoader(
57                 self.train_ds,
58                 batch_size=self.batch_size,
59                 num_workers=self.num_workers,
60                 pin_memory=self.pin_memory,
61                 shuffle=self.shuffle,
62             )
63
64             for step, batch in enumerate(
65                 tqdm(
66                     train_dataloader,
67                     desc="Epoch %d/%d" % (epoch + 1, self.num_epochs),
68                     total=len(self.train_ds) // self.batch_size,
69                 )
70             ):
71                 self.step(batch)
72
73     def test(self) -> None:
74         raise NotImplementedError("You need to provide a test
75         function.")
76
77 class TorchVisionDatasetPipeline(DataLoaderPipeline):
78     # language=rst
79     """
80     An example implementation of ‘DataLoaderPipeline‘ that runs
81     all of the datasets
82     inside of ‘bindsnet.datasets‘ that inherit from an instance of
83     a
84     ‘torchvision.datasets‘. These are documented in ‘bindsnet/
85     datasets/README.md‘.
86     This specific class just runs an unsupervised network.
87     """

```

```

85     def __init__(
86         self,
87         network: Network,
88         train_ds: Dataset,
89         pipeline_analyzer: Optional[PipelineAnalyzer] = None,
90         **kwargs
91     ) -> None:
92         # language=rst
93         """
94         Initializes the pipeline.
95
96         :param network: Arbitrary ‘‘network‘‘ object.
97         :param train_ds: A ‘‘torchvision.datasets‘‘ wrapper dataset
98         from
99             ‘‘bindsnet.datasets‘‘.
100
101         Keyword arguments:
102
103         :param str input_layer: Layer of the network that receives
104         input.
105         """
106         super().__init__(network, train_ds, None, **kwargs)
107         self.input_layer = kwargs.get("input_layer", "X")
108         self.pipeline_analyzer = pipeline_analyzer
109
110     def step_(self, batch: Dict[str, torch.Tensor], **kwargs) ->
111     None:
112         # language=rst
113         """
114         Perform a pass of the network given the input batch.
115         Unsupervised training
116         (implying everything is stored inside of the ‘‘network‘‘
117         object, therefore
118         returns ‘‘None‘‘.
119
120         :param batch: A dictionary of the current batch. Includes
121         image, label and
122         encoded versions.
123         """
124         self.network.reset_state_variables()
125         inputs = {self.input_layer: batch["encoded_image"]}
126         self.network.run(inputs, time=batch["encoded_image"].shape
127         [0])
128
129     def init_fn(self) -> None:
130         pass
131
132     def plots(self, batch: Dict[str, torch.Tensor], *args) -> None:
133         # language=rst
134         """
135         Create any plots and logs for a step given the input batch.
136
137         :param batch: A dictionary of the current batch. Includes

```

```

132 image, label and
        encoded versions.
133     """
134     if self.pipeline_analyzer is not None:
135         self.pipeline_analyzer.plot_obs(
136             batch["encoded_image"][0, ...].sum(0), step=self.
step_count
137         )
138
139         self.pipeline_analyzer.plot_spikes(
140             self.get_spike_data(), step=self.step_count
141         )
142
143         vr, tv = self.get_voltage_data()
144         self.pipeline_analyzer.plot_voltages(vr, tv, step=self.
step_count)
145
146         self.pipeline_analyzer.finalize_step()
147
148     def test_step(self):
149         pass

```

Listing B.13: Data-loader pipeline

```

1 import torch
2 import numpy as np
3
4 from . import EnvironmentPipeline
5
6
7 def select_multinomial(pipeline: EnvironmentPipeline, **kwargs) ->
int:
8     # language=rst
9     """
10     Selects an action probabilistically based on spiking activity
from a network layer.
11
12     :param pipeline: EnvironmentPipeline with environment that has
an integer action
13     space.
14     :return: Action sampled from multinomial over activity of
similarly-sized output
15     layer.
16
17     Keyword arguments:
18
19     :param str output: Name of output layer whose activity to base
action selection on.
20     """
21     try:
22         output = kwargs["output"]
23     except KeyError:
24         raise KeyError('select_multinomial() requires an "output"
layer argument.')

```

```

25
26     output = pipeline.network.layers[output]
27     action_space = pipeline.env.action_space
28
29     assert (
30         output.n % action_space.n == 0
31     ), f"Output layer size of {output.n} is not divisible by action
32     space size of {action_space.n}."
33
34     pop_size = int(output.n / action_space.n)
35     spikes = output.s
36     _sum = spikes.sum().float()
37
38     # Choose action based on population's spiking.
39     if _sum == 0:
40         action = np.random.choice(pipeline.env.action_space.n)
41     else:
42         pop_spikes = torch.tensor(
43             [
44                 spikes[(i * pop_size) : (i * pop_size) + pop_size].
45                 sum()
46                 for i in range(action_space.n)
47             ]
48         )
49         action = torch.multinomial((pop_spikes.float() / _sum).view(
50             (-1), 1)[0].item())
51
52     return action
53
54 def select_softmax(pipeline: EnvironmentPipeline, **kwargs) -> int:
55     # language=rst
56     """
57     Selects an action using softmax function based on spiking from a
58     network layer.
59
60     :param pipeline: EnvironmentPipeline with environment that has
61     an integer action
62     space and :code:'spike_record' set.
63     :return: Action sampled from softmax over activity of similarly-
64     sized output layer.
65
66     Keyword arguments:
67
68     :param str output: Name of output layer whose activity to base
69     action selection on.
70     """
71     try:
72         output = kwargs["output"]
73     except KeyError:
74         raise KeyError('select_softmax() requires an "output" layer
75         argument.')
```

```

71     pipeline.network.layers[output].n == pipeline.env.
action_space.n
72 ), "Output layer size is not equal to the size of the action
space."
73
74     assert hasattr(
75         pipeline, "spike_record"
76     ), "EnvironmentPipeline is missing the attribute: spike_record."
77
78     spikes = torch.sum(pipeline.spike_record[output], dim=0)
79     probabilities = torch.softmax(spikes, dim=0)
80     return torch.multinomial(probabilities, num_samples=1).item()
81
82
83 def select_random(pipeline: EnvironmentPipeline, **kwargs) -> int:
84     # language=rst
85     """
86     Selects an action randomly from the action space.
87
88     :param pipeline: EnvironmentPipeline with environment that has
an integer action
89         space.
90     :return: Action randomly sampled over size of pipeline's action
space.
91     """
92     # Choose action randomly from the action space.
93     return np.random.choice(pipeline.env.action_space.n)

```

Listing B.14: Action

B.5 Encoding

```

1 from .encodings import single, repeat, bernoulli, poisson,
rank_order
2 from .loaders import bernoulli_loader, poisson_loader,
rank_order_loader
3 from .encoders import (
4     Encoder,
5     NullEncoder,
6     SingleEncoder,
7     RepeatEncoder,
8     BernoulliEncoder,
9     PoissonEncoder,
10    RankOrderEncoder,
11 )

```

Listing B.15: Initialization

```

1 from typing import Optional, Union, Iterable, Iterator
2
3 import torch
4
5 from .encodings import bernoulli, poisson, rank_order
6

```

```

7
8 def bernoulli_loader(
9     data: Union[torch.Tensor, Iterable[torch.Tensor]],
10    time: Optional[int] = None,
11    dt: float = 1.0,
12    **kwargs
13 ) -> Iterator[torch.Tensor]:
14     # language=rst
15     """
16     Lazily invokes ‘bindsnet.encoding.bernoulli‘ to iteratively
17     encode a sequence of
18     data.
19
20     :param data: Tensor of shape ‘[n_samples, n_1, ..., n_k]‘.
21     :param time: Length of Bernoulli spike train per input variable.
22     :param dt: Simulation time step.
23     :return: Tensors of shape ‘[time, n_1, ..., n_k]‘ of Bernoulli
24     -distributed spikes.
25
26     Keyword arguments:
27
28     :param float max_prob: Maximum probability of spike per
29     Bernoulli trial.
30     """
31     # Setting kwargs.
32     max_prob = kwargs.get("dt", 1.0)
33
34     for i in range(len(data)):
35         # Encode datum as Bernoulli spike trains.
36         yield bernoulli(datum=data[i], time=time, dt=dt, max_prob=
37         max_prob)
38
39 def poisson_loader(
40     data: Union[torch.Tensor, Iterable[torch.Tensor]],
41     time: int,
42     dt: float = 1.0,
43     **kwargs
44 ) -> Iterator[torch.Tensor]:
45     # language=rst
46     """
47     Lazily invokes ‘bindsnet.encoding.poisson‘ to iteratively
48     encode a sequence of
49     data.
50
51     :param data: Tensor of shape ‘[n_samples, n_1, ..., n_k]‘.
52     :param time: Length of Poisson spike train per input variable.
53     :param dt: Simulation time step.
54     :return: Tensors of shape ‘[time, n_1, ..., n_k]‘ of Poisson-
55     distributed spikes.
56     """
57     for i in range(len(data)):
58         # Encode datum as Poisson spike trains.
59         yield poisson(datum=data[i], time=time, dt=dt)

```



```

55
56
57 def rank_order_loader(
58     data: Union[torch.Tensor, Iterable[torch.Tensor]],
59     time: int,
60     dt: float = 1.0,
61     **kwargs
62 ) -> Iterator[torch.Tensor]:
63     # language=rst
64     """
65     Lazily invokes ‘bindsnet.encoding.rank_order‘ to iteratively
66     encode a sequence of
67     data.
68
69     :param data: Tensor of shape ‘[n_samples, n_1, ..., n_k]‘.
70     :param time: Length of rank order-encoded spike train per input
71     variable.
72     :param dt: Simulation time step.
73     :return: Tensors of shape ‘[time, n_1, ..., n_k]‘ of rank
74     order-encoded spikes.
75     """
76     for i in range(len(data)):
77         # Encode datum as rank order-encoded spike trains.
78         yield rank_order(datum=data[i], time=time, dt=dt)

```

Listing B.16: Loaders

```

1 from . import encodings
2
3
4 class Encoder:
5     # language=rst
6     """
7     Base class for spike encodings transforms.
8
9     Calls ‘self.enc‘ from the subclass and passes whatever
10    arguments were provided.
11    ‘self.enc‘ must be callable with ‘torch.Tensor‘, ‘*args‘,
12    ‘**kwargs‘
13    """
14
15    def __init__(self, *args, **kwargs) -> None:
16        self.enc_args = args
17        self.enc_kwargs = kwargs
18
19    def __call__(self, img):
20        return self.enc(img, *self.enc_args, **self.enc_kwargs)
21
22 class NullEncoder(Encoder):
23     # language=rst
24     """
25     Pass through of the datum that was input.

```

```

26 .. note::
27     This is not a real spike encoder. Be careful with the usage
of this class.
28     """
29
30     def __init__(self):
31         super().__init__()
32
33     def __call__(self, img):
34         return img
35
36
37 class SingleEncoder(Encoder):
38     def __init__(self, time: int, dt: float = 1.0, sparsity: float =
0.5, **kwargs):
39         # language=rst
40         """
41         Creates a callable SingleEncoder which encodes as defined in
42         ‘‘bindsnet.encoding.single‘‘
43
44         :param time: Length of single spike train per input variable
45
46         :param dt: Simulation time step.
47         :param sparsity: Sparsity of the input representation. 0 for
no spikes and 1 for
48         all spikes.
49         """
50         super().__init__(time, dt=dt, sparsity=sparsity, **kwargs)
51
52         self.enc = encodings.single
53
54 class RepeatEncoder(Encoder):
55     def __init__(self, time: int, dt: float = 1.0, **kwargs):
56         # language=rst
57         """
58         Creates a callable ‘‘RepeatEncoder‘‘ which encodes as
59         defined in
60         ‘‘bindsnet.encoding.repeat‘‘
61
62         :param time: Length of repeat spike train per input variable
63
64         :param dt: Simulation time step.
65         """
66         super().__init__(time, dt=dt, **kwargs)
67
68         self.enc = encodings.repeat
69
70 class BernoulliEncoder(Encoder):
71     def __init__(self, time: int, dt: float = 1.0, **kwargs):
72         # language=rst
73         """
74         Creates a callable ‘‘BernoulliEncoder‘‘ which encodes as

```

```

defined in
74     :code: 'bindsnet.encoding.bernoulli'
75
76     :param time: Length of Bernoulli spike train per input
variable.
77     :param dt: Simulation time step.
78
79     Keyword arguments:
80
81     :param float max_prob: Maximum probability of spike per time
step.
82     """
83     super().__init__(time, dt=dt, **kwargs)
84
85     self.enc = encodings.bernoulli
86
87
88 class PoissonEncoder(Encoder):
89     def __init__(self, time: int, dt: float = 1.0, **kwargs):
90         # language=rst
91         """
92         Creates a callable PoissonEncoder which encodes as defined
in
93         'bindsnet.encoding.poisson'
94
95         :param time: Length of Poisson spike train per input
variable.
96         :param dt: Simulation time step.
97         """
98         super().__init__(time, dt=dt, **kwargs)
99
100        self.enc = encodings.poisson
101
102
103 class RankOrderEncoder(Encoder):
104     def __init__(self, time: int, dt: float = 1.0, **kwargs):
105         # language=rst
106         """
107         Creates a callable RankOrderEncoder which encodes as defined
in
108         :code: 'bindsnet.encoding.rank_order'
109
110         :param time: Length of RankOrder spike train per input
variable.
111         :param dt: Simulation time step.
112         """
113         super().__init__(time, dt=dt, **kwargs)
114
115        self.enc = encodings.rank_order

```

Listing B.17: Encoders

```

1 from typing import Optional
2

```

```

3 import torch
4 import numpy as np
5
6
7 def single(
8     datum: torch.Tensor, time: int, dt: float = 1.0, sparsity: float
9     = 0.5, **kwargs
10 ) -> torch.Tensor:
11     # language=rst
12     """
13     Generates timing based single-spike encoding. Spike occurs
14     earlier if the
15     intensity of the input feature is higher. Features whose value
16     is lower than
17     threshold is remain silent.
18
19     :param datum: Tensor of shape ‘[n_1, ..., n_k]‘.
20     :param time: Length of the input and output.
21     :param dt: Simulation time step.
22     :param sparsity: Sparsity of the input representation. 0 for no
23     spikes and 1 for all
24     spikes.
25     :return: Tensor of shape ‘[time, n_1, ..., n_k]‘.
26     """
27     time = int(time / dt)
28     shape = list(datum.shape)
29     datum = np.copy(datum)
30     quantile = np.quantile(datum, 1 - sparsity)
31     s = np.zeros([time, *shape])
32     s[0] = np.where(datum > quantile, np.ones(shape), np.zeros(shape))
33     return torch.Tensor(s).byte()
34
35
36 def repeat(datum: torch.Tensor, time: int, dt: float = 1.0, **kwargs
37 ) -> torch.Tensor:
38     # language=rst
39     """
40     :param datum: Repeats a tensor along a new dimension in the 0th
41     position for
42     ‘int(time / dt)‘ timesteps.
43     :param time: Tensor of shape ‘[n_1, ..., n_k]‘.
44     :param dt: Simulation time step.
45     :return: Tensor of shape ‘[time, n_1, ..., n_k]‘ of repeated
46     data along the 0-th
47     dimension.
48     """
49     time = int(time / dt)
50     return datum.repeat([time, *[1] * len(datum.shape)])
51
52
53 def bernoulli(
54     datum: torch.Tensor, time: Optional[int] = None, dt: float =
55     1.0, **kwargs

```

```

48 ) -> torch.Tensor:
49     # language=rst
50     """
51     Generates Bernoulli-distributed spike trains based on input
52     intensity. Inputs must
53     be non-negative. Spikes correspond to successful Bernoulli
54     trials, with success
55     probability equal to (normalized in [0, 1]) input value.
56
57     :param datum: Tensor of shape "[n_1, ..., n_k]".
58     :param time: Length of Bernoulli spike train per input variable.
59     :param dt: Simulation time step.
60     :return: Tensor of shape "[time, n_1, ..., n_k]" of Bernoulli-
61     distributed spikes.
62
63     Keyword arguments:
64
65     :param float max_prob: Maximum probability of spike per
66     Bernoulli trial.
67     """
68     # Setting kwargs.
69     max_prob = kwargs.get("max_prob", 1.0)
70
71     assert 0 <= max_prob <= 1, "Maximum firing probability must be
72     in range [0, 1]"
73     assert (datum >= 0).all(), "Inputs must be non-negative"
74
75     shape, size = datum.shape, datum.numel()
76     datum = datum.flatten()
77
78     if time is not None:
79         time = int(time / dt)
80
81     # Normalize inputs and rescale (spike probability proportional
82     to input intensity).
83     if datum.max() > 1.0:
84         datum /= datum.max()
85
86     # Make spike data from Bernoulli sampling.
87     if time is None:
88         spikes = torch.bernoulli(max_prob * datum)
89         spikes = spikes.view(*shape)
90     else:
91         spikes = torch.bernoulli(max_prob * datum.repeat([time, 1]))
92         spikes = spikes.view(time, *shape)
93
94     return spikes.byte()
95
96 def poisson(datum: torch.Tensor, time: int, dt: float = 1.0, **
97 kwargs) -> torch.Tensor:
98     # language=rst
99     """
100     Generates Poisson-distributed spike trains based on input

```

```

intensity. Inputs must be
95 non-negative, and give the firing rate in Hz. Inter-spike
intervals (ISIs) for
96 non-negative data incremented by one to avoid zero intervals
while maintaining ISI
97 distributions.
98
99 :param datum: Tensor of shape "[n_1, ..., n_k]".
100 :param time: Length of Poisson spike train per input variable.
101 :param dt: Simulation time step.
102 :return: Tensor of shape "[time, n_1, ..., n_k]" of Poisson-
distributed spikes.
103 """
104 assert (datum >= 0).all(), "Inputs must be non-negative"
105
106 # Get shape and size of data.
107 shape, size = datum.shape, datum.numel()
108 datum = datum.flatten()
109 time = int(time / dt)
110
111 # Compute firing rates in seconds as function of data intensity,
112 # accounting for simulation time step.
113 rate = torch.zeros(size)
114 rate[datum != 0] = 1 / datum[datum != 0] * (1000 / dt)
115
116 # Create Poisson distribution and sample inter-spike intervals
117 # (incrementing by 1 to avoid zero intervals).
118 dist = torch.distributions.Poisson(rate=rate)
119 intervals = dist.sample(sample_shape=torch.Size([time + 1]))
120 intervals[:, datum != 0] += (intervals[:, datum != 0] == 0).
float()
121
122 # Calculate spike times by cumulatively summing over time
dimension.
123 times = torch.cumsum(intervals, dim=0).long()
124 times[times >= time + 1] = 0
125
126 # Create tensor of spikes.
127 spikes = torch.zeros(time + 1, size).byte()
128 spikes[times, torch.arange(size)] = 1
129 spikes = spikes[1:]
130
131 return spikes.view(time, *shape)
132
133
134 def rank_order(
135     datum: torch.Tensor, time: int, dt: float = 1.0, **kwargs
136 ) -> torch.Tensor:
137     # language=rst
138     """
139     Encodes data via a rank order coding-like representation. One
spike per neuron,
140     temporally ordered by decreasing intensity. Inputs must be non-
negative.

```

```

141
142     :param datum: Tensor of shape '[n_samples, n_1, ..., n_k]'.
143     :param time: Length of rank order-encoded spike train per input
variable.
144     :param dt: Simulation time step.
145     :return: Tensor of shape '[time, n_1, ..., n_k]' of rank order
-encod
ed spikes.
146     """
147     assert (datum >= 0).all(), "Inputs must be non-negative"
148
149     shape, size = datum.shape, datum.numel()
150     datum = datum.flatten()
151     time = int(time / dt)
152
153     # Create spike times in order of decreasing intensity.
154     datum /= datum.max()
155     times = torch.zeros(size)
156     times[datum != 0] = 1 / datum[datum != 0]
157     times *= time / times.max() # Extended through simulation time.
158     times = torch.ceil(times).long()
159
160     # Create spike times tensor.
161     spikes = torch.zeros(time, size).byte()
162     for i in range(size):
163         if 0 < times[i] < time:
164             spikes[times[i] - 1, i] = 1
165
166     return spikes.reshape(time, *shape)

```

Listing B.18: Encodings

B.6 Conversion

```

1 from .conversion import (
2     Permute,
3     FeatureExtractor,
4     SubtractiveResetIFNodes,
5     PassThroughNodes,
6     PermuteConnection,
7     ConstantPad2dConnection,
8     data_based_normalization,
9     ann_to_snn,
10 )

```

Listing B.19: Initialization

```

1 import torch
2 import numpy as np
3 import torch.nn as nn
4 import torch.nn.functional as F
5
6 from torch.nn.modules.utils import _pair
7
8 from copy import deepcopy

```

```

9 from typing import Union, Sequence, Optional, Tuple, Dict, Iterable
10
11 import bindsnet.network.nodes as nodes
12 import bindsnet.network.topology as topology
13
14 from bindsnet.network import Network
15
16
17 class Permute(nn.Module):
18     # language=rst
19     """
20     PyTorch module for the explicit permutation of a tensor's
21     dimensions in a parent
22     module's 'forward' pass (as opposed to 'torch.permute').
23     """
24     def __init__(self, dims):
25         # language=rst
26         """
27         Constructor for 'Permute' module.
28
29         :param dims: Ordering of dimensions for permutation.
30         """
31         super(Permute, self).__init__()
32
33         self.dims = dims
34
35     def forward(self, x):
36         # language=rst
37         """
38         Forward pass of permutation module.
39
40         :param x: Input tensor to permute.
41         :return: Permuted input tensor.
42         """
43         return x.permute(*self.dims).contiguous()
44
45
46 class FeatureExtractor(nn.Module):
47     # language=rst
48     """
49     Special-purpose PyTorch module for the extraction of child
50     module's activations.
51     """
52     def __init__(self, submodule):
53         # language=rst
54         """
55         Constructor for 'FeatureExtractor' module.
56
57         :param submodule: The module who's children modules are to
58         be extracted.
59         """
60         super(FeatureExtractor, self).__init__()

```



```

60
61     self.submodule = submodule
62
63     def forward(self, x: torch.Tensor) -> Dict[nn.Module, torch.
Tensor]:
64         # language=rst
65         """
66         Forward pass of the feature extractor.
67
68         :param x: Input data for the ‘‘submodule’’.
69         :return: A dictionary mapping
70         """
71         activations = {"input": x}
72         for name, module in self.submodule._modules.items():
73             if isinstance(module, nn.Linear):
74                 x = x.view(-1, module.in_features)
75
76                 x = module(x)
77                 activations[name] = x
78
79         return activations
80
81
82 class SubtractiveResetIFNodes(nodes.Nodes):
83     # language=rst
84     """
85     Layer of ‘integrate-and-fire (IF) neurons
86     <http://neurondynamics.epfl.ch/online/Ch1.S3.html>’ using
reset by subtraction.
87     """
88
89     def __init__(
90         self,
91         n: Optional[int] = None,
92         shape: Optional[Iterable[int]] = None,
93         traces: bool = False,
94         traces_additive: bool = False,
95         tc_trace: Union[float, torch.Tensor] = 20.0,
96         trace_scale: Union[float, torch.Tensor] = 1.0,
97         sum_input: bool = False,
98         thresh: Union[float, torch.Tensor] = -52.0,
99         reset: Union[float, torch.Tensor] = -65.0,
100        refrac: Union[int, torch.Tensor] = 5,
101        lbound: float = None,
102        **kwargs,
103    ) -> None:
104        # language=rst
105        """
106        Instantiates a layer of IF neurons with the subtractive
reset mechanism from
107        ‘this paper
108        <https://www.frontiersin.org/articles/10.3389/fnins.2017.00682/full>’.
109

```

```

110     :param n: The number of neurons in the layer.
111     :param shape: The dimensionality of the layer.
112     :param traces: Whether to record spike traces.
113     :param traces_additive: Whether to record spike traces
additively.
114     :param tc_trace: Time constant of spike trace decay.
115     :param trace_scale: Scaling factor for spike trace.
116     :param sum_input: Whether to sum all inputs.
117     :param thresh: Spike threshold voltage.
118     :param reset: Post-spike reset voltage.
119     :param refrac: Refractory (non-firing) period of the neuron.
120     :param lbound: Lower bound of the voltage.
121     """
122     super().__init__(
123         n=n,
124         shape=shape,
125         traces=traces,
126         traces_additive=traces_additive,
127         tc_trace=tc_trace,
128         trace_scale=trace_scale,
129         sum_input=sum_input,
130     )
131
132     self.register_buffer(
133         "reset", torch.tensor(reset, dtype=torch.float)
134     ) # Post-spike reset voltage.
135     self.register_buffer(
136         "thresh", torch.tensor(thresh, dtype=torch.float)
137     ) # Spike threshold voltage.
138     self.register_buffer(
139         "refrac", torch.tensor(refrac)
140     ) # Post-spike refractory period.
141     self.register_buffer("v", torch.FloatTensor()) # Neuron
voltage.
142     self.register_buffer(
143         "refrac_count", torch.FloatTensor()
144     ) # Refractory period counters.
145
146     self.lbound = lbound # Lower bound of voltage.
147
148     def forward(self, x: torch.Tensor) -> None:
149         # language=rst
150         """
151         Runs a single simulation step.
152
153         :param x: Inputs to the layer.
154         """
155         # Integrate input voltages.
156         self.v += (self.refrac_count == 0).float() * x
157
158         # Decrement refractory counters.
159         self.refrac_count = (self.refrac_count > 0).float() * (
160             self.refrac_count - self.dt
161         )

```

```

162
163     # Check for spiking neurons.
164     self.s = self.v >= self.thresh
165
166     # Refractoriness and voltage reset.
167     self.refrac_count.masked_fill_(self.s, self.refrac)
168     self.v[self.s] = self.v[self.s] - self.thresh
169
170     # Voltage clipping to lower bound.
171     if self.lbound is not None:
172         self.v.masked_fill_(self.v < self.lbound, self.lbound)
173
174     super().forward(x)
175
176     def reset_state_variables(self) -> None:
177         # language=rst
178         """
179         Resets relevant state variables.
180         """
181         super().reset_state_variables()
182         self.v.fill_(self.reset) # Neuron voltages.
183         self.refrac_count.zero_() # Refractory period counters.
184
185     def set_batch_size(self, batch_size) -> None:
186         # language=rst
187         """
188         Sets mini-batch size. Called when layer is added to a
189         network.
190
191         :param batch_size: Mini-batch size.
192         """
193         super().set_batch_size(batch_size=batch_size)
194         self.v = self.reset * torch.ones(batch_size, *self.shape,
195         device=self.v.device)
196         self.refrac_count = torch.zeros_like(self.v, device=self.
197         refrac_count.device)
198
199 class PassThroughNodes(nodes.Nodes):
200     # language=rst
201     """
202     Layer of 'integrate-and-fire (IF) neurons
203     <http://neurondynamics.epfl.ch/online/Ch1.S3.html>' with
204     using reset by
205     subtraction.
206     """
207
208     def __init__(
209         self,
210         n: Optional[int] = None,
211         shape: Optional[Sequence[int]] = None,
212         traces: bool = False,
213         traces_additive: bool = False,
214         tc_trace: Union[float, torch.Tensor] = 20.0,

```

```

212     trace_scale: Union[float, torch.Tensor] = 1.0,
213     sum_input: bool = False,
214 ) -> None:
215     # language=rst
216     """
217     Instantiates a layer of IF neurons.
218
219     :param n: The number of neurons in the layer.
220     :param shape: The dimensionality of the layer.
221     :param traces: Whether to record spike traces.
222     :param trace_tc: Time constant of spike trace decay.
223     :param sum_input: Whether to sum all inputs.
224     """
225     super().__init__(
226         n=n,
227         shape=shape,
228         traces=traces,
229         traces_additive=traces_additive,
230         tc_trace=tc_trace,
231         trace_scale=trace_scale,
232         sum_input=sum_input,
233     )
234     self.register_buffer("v", torch.zeros(self.shape))
235
236     def forward(self, x: torch.Tensor) -> None:
237         # language=rst
238         """
239         Runs a single simulation step.
240
241         :param inputs: Inputs to the layer.
242         :param dt: Simulation time step.
243         """
244         self.s = x
245
246     def reset_state_variables(self) -> None:
247         # language=rst
248         """
249         Resets relevant state variables.
250         """
251         self.s.zero_()
252
253
254 class PermuteConnection(topology.AbstractConnection):
255     # language=rst
256     """
257     Special-purpose connection for emulating the custom ‘Permute’
258     module in spiking
259     neural networks.
260     """
261     def __init__(
262         self,
263         source: nodes.Nodes,
264         target: nodes.Nodes,

```

```

265     dims: Sequence,
266     nu: Optional[Union[float, Sequence[float]]] = None,
267     weight_decay: float = 0.0,
268     **kwargs,
269 ) -> None:
270     # language=rst
271     """
272     Constructor for ‘‘PermuteConnection‘‘.
273
274     :param source: A layer of nodes from which the connection
275     originates.
276     :param target: A layer of nodes to which the connection
277     connects.
278     :param dims: Order of dimensions to permute.
279     :param nu: Learning rate for both pre- and post-synaptic
280     events.
281     :param weight_decay: Constant multiple to decay weights by
282     on each iteration.
283
284     Keyword arguments:
285
286     :param function update_rule: Modifies connection parameters
287     according to some
288     rule.
289     :param float wmin: The minimum value on the connection
290     weights.
291     :param float wmax: The maximum value on the connection
292     weights.
293     :param float norm: Total weight per target neuron
294     normalization.
295     """
296     super().__init__(source, target, nu, weight_decay, **kwargs)
297
298     self.dims = dims
299
300     def compute(self, s: torch.Tensor) -> torch.Tensor:
301         # language=rst
302         """
303         Permute input.
304
305         :param s: Input.
306         :return: Permuted input.
307         """
308         return s.permute(self.dims).float()
309
310 class ConstantPad2dConnection(topology.AbstractConnection):
311     # language=rst
312     """
313     Special-purpose connection for emulating the ‘‘ConstantPad2d‘‘
314     PyTorch module in
315     spiking neural networks.
316     """

```

```

310     def __init__(
311         self,
312         source: nodes.Nodes,
313         target: nodes.Nodes,
314         padding: Tuple,
315         nu: Optional[Union[float, Sequence[float]]] = None,
316         weight_decay: float = 0.0,
317         **kwargs,
318     ) -> None:
319         # language=rst
320         """
321         Constructor for ‘‘ConstantPad2dConnection‘‘.
322
323         :param source: A layer of nodes from which the connection
324         originates.
325         :param target: A layer of nodes to which the connection
326         connects.
327         :param padding: Padding of input tensors; passed to ‘‘torch.
328         nn.functional.pad‘‘.
329         :param nu: Learning rate for both pre- and post-synaptic
330         events.
331         :param weight_decay: Constant multiple to decay weights by
332         on each iteration.
333
334         Keyword arguments:
335
336         :param function update_rule: Modifies connection parameters
337         according to some
338         rule.
339         :param float wmin: The minimum value on the connection
340         weights.
341         :param float wmax: The maximum value on the connection
342         weights.
343         :param float norm: Total weight per target neuron
344         normalization.
345         """
346
347         super().__init__(source, target, nu, weight_decay, **kwargs)
348
349         self.padding = padding
350
351     def compute(self, s: torch.Tensor):
352         # language=rst
353         """
354         Pad input.
355
356         :param s: Input.
357         :return: Padding input.
358         """
359         return F.pad(s, self.padding).float()
360
361     def data_based_normalization(
362         ann: Union[nn.Module, str], data: torch.Tensor, percentile:

```

```

float = 99.9
355 ):
356     # language=rst
357     """
358     Use a dataset to rescale ANN weights and biases such that that
359     the max ReLU
360     activation is less than 1.
361
362     :param ann: Artificial neural network implemented in PyTorch.
363     Accepts either
364     "torch.nn.Module" or path to network saved using "torch.
365     save()".
366     :param data: Data to use to perform data-based weight
367     normalization of shape
368     "[n_examples, ...]".
369     :param percentile: Percentile (in "[0, 100]") of activations
370     to scale by in
371     data-based normalization scheme.
372     :return: Artificial neural network with rescaled weights and
373     biases according to
374     activations on the dataset.
375     """
376     if isinstance(ann, str):
377         ann = torch.load(ann)
378
379     assert isinstance(ann, nn.Module)
380
381     def set_requires_grad(module, value):
382         for param in module.parameters():
383             param.requires_grad = value
384
385     set_requires_grad(ann, value=False)
386     extractor = FeatureExtractor(ann)
387     all_activations = extractor.forward(data)
388
389     prev_module = None
390     prev_factor = 1
391     for name, module in ann._modules.items():
392         if isinstance(module, nn.Sequential):
393             extractor2 = FeatureExtractor(module)
394             all_activations2 = extractor2.forward(data)
395             for name2, module2 in module.named_children():
396                 activations = all_activations2[name2]
397
398                 if isinstance(module2, nn.ReLU):
399                     if prev_module is not None:
400                         scale_factor = np.percentile(activations.cpu
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

400         prev_factor = scale_factor
401
402         elif isinstance(module2, nn.Linear) or isinstance(
403 module2, nn.Conv2d):
404             prev_module = module2
405
406     else:
407         activations = all_activations[name]
408         if isinstance(module, nn.ReLU):
409             if prev_module is not None:
410                 scale_factor = np.percentile(activations.cpu(),
411 percentile)
412
413             prev_module.weight *= prev_factor / scale_factor
414             prev_module.bias /= scale_factor
415
416             prev_factor = scale_factor
417
418         elif isinstance(module, nn.Linear) or isinstance(module,
419 nn.Conv2d):
420             prev_module = module
421
422     return ann
423
424 def _ann_to_snn_helper(prev, current, node_type, last=False, **
425 kwargs):
426     # language=rst
427     """
428     Helper function for main ‘‘ann_to_snn’’ method.
429
430     :param prev: Previous PyTorch module in artificial neural
431     network.
432     :param current: Current PyTorch module in artificial neural
433     network.
434     :param node_type: Type of ‘‘bindsnet.network.nodes’’ to use.
435     :param last: Whether this connection and layer is the last to be
436     converted.
437     :return: Spiking neural network layer and connection
438     corresponding to ‘‘prev’’ and
439     ‘‘current’’ PyTorch modules.
440     """
441     if isinstance(current, nn.Linear):
442         layer = node_type(
443             n=current.out_features,
444             reset=0,
445             thresh=1,
446             refrac=0,
447             sum_input=last,
448             **kwargs,
449         )
450         bias = current.bias if current.bias is not None else torch.
451         zeros(layer.n)
452         connection = topology.Connection(

```



```

445         source=prev, target=layer, w=current.weight.t(), b=bias
446     )
447
448     elif isinstance(current, nn.Conv2d):
449         input_height, input_width = prev.shape[2], prev.shape[3]
450         out_channels, output_height, output_width = (
451             current.out_channels,
452             prev.shape[2],
453             prev.shape[3],
454         )
455
456         width = (
457             input_height - current.kernel_size[0] + 2 * current.
padding[0]
458         ) / current.stride[0] + 1
459         height = (
460             input_width - current.kernel_size[1] + 2 * current.
padding[1]
461         ) / current.stride[1] + 1
462         shape = (1, out_channels, int(width), int(height))
463
464         layer = node_type(
465             shape=shape, reset=0, thresh=1, refrac=0, sum_input=last
, **kwargs
466         )
467         bias = current.bias if current.bias is not None else torch.
zeros(layer.shape[1])
468         connection = topology.Conv2dConnection(
469             source=prev,
470             target=layer,
471             kernel_size=current.kernel_size,
472             stride=current.stride,
473             padding=current.padding,
474             dilation=current.dilation,
475             w=current.weight,
476             b=bias,
477         )
478
479     elif isinstance(current, nn.MaxPool2d):
480         input_height, input_width = prev.shape[2], prev.shape[3]
481         current.kernel_size = _pair(current.kernel_size)
482         current.padding = _pair(current.padding)
483         current.stride = _pair(current.stride)
484
485         width = (
486             input_height - current.kernel_size[0] + 2 * current.
padding[0]
487         ) / current.stride[0] + 1
488         height = (
489             input_width - current.kernel_size[1] + 2 * current.
padding[1]
490         ) / current.stride[1] + 1
491         shape = (1, prev.shape[1], int(width), int(height))
492

```

```

493     layer = PassThroughNodes(shape=shape)
494     connection = topology.MaxPool2dConnection(
495         source=prev,
496         target=layer,
497         kernel_size=current.kernel_size,
498         stride=current.stride,
499         padding=current.padding,
500         dilation=current.dilation,
501         decay=1,
502     )
503
504     elif isinstance(current, Permute):
505         layer = PassThroughNodes(
506             shape=[
507                 prev.shape[current.dims[0]],
508                 prev.shape[current.dims[1]],
509                 prev.shape[current.dims[2]],
510                 prev.shape[current.dims[3]],
511             ]
512         )
513
514         connection = PermuteConnection(source=prev, target=layer,
515                                       dims=current.dims)
516
517     elif isinstance(current, nn.ConstantPad2d):
518         layer = PassThroughNodes(
519             shape=[
520                 prev.shape[0],
521                 prev.shape[1],
522                 current.padding[0] + current.padding[1] + prev.shape
523                 [2],
524                 current.padding[2] + current.padding[3] + prev.shape
525                 [3],
526             ]
527         )
528
529         connection = ConstantPad2dConnection(
530             source=prev, target=layer, padding=current.padding
531         )
532
533     else:
534         return None, None
535
536     return layer, connection
537
538 def ann_to_snn(
539     ann: Union[nn.Module, str],
540     input_shape: Sequence[int],
541     data: Optional[torch.Tensor] = None,
542     percentile: float = 99.9,
543     node_type: Optional[nodes.Nodes] = SubtractiveResetIFNodes,
544     **kwargs,
545 ) -> Network:

```

```

544 # language=rst
545 """
546 Converts an artificial neural network (ANN) written as a ‘torch
547 .nn.Module‘ into a
548 near-equivalent spiking neural network.
549
550 :param ann: Artificial neural network implemented in PyTorch.
551 Accepts either
552 ‘‘torch.nn.Module‘‘ or path to network saved using ‘‘torch.
553 save()‘‘.
554 :param input_shape: Shape of input data.
555 :param data: Data to use to perform data-based weight
556 normalization of shape
557 ‘‘[n_examples, ...]‘‘.
558 :param percentile: Percentile (in ‘‘[0, 100]‘‘) of activations
559 to scale by in
560 data-based normalization scheme.
561 :param node_type: Class of ‘‘Nodes‘‘ to use in replacing ‘‘torch
562 .nn.Linear‘‘ layers
563 in original ANN.
564 :return: Spiking neural network implemented in PyTorch.
565 """
566
567 if isinstance(ann, str):
568     ann = torch.load(ann)
569 else:
570     ann = deepcopy(ann)
571
572 assert isinstance(ann, nn.Module)
573
574 if data is None:
575     import warnings
576
577     warnings.warn("Data is None. Weights will not be scaled.",
578 RuntimeWarning)
579 else:
580     ann = data_based_normalization(
581         ann=ann, data=data.detach(), percentile=percentile
582     )
583
584 snn = Network()
585
586 input_layer = nodes.Input(shape=input_shape)
587 snn.add_layer(input_layer, name="Input")
588
589 children = []
590 for c in ann.children():
591     if isinstance(c, nn.Sequential):
592         for c2 in list(c.children()):
593             children.append(c2)
594     else:
595         children.append(c)
596
597 i = 0
598 prev = input_layer

```

```

591     while i < len(children) - 1:
592         current, nxt = children[i : i + 2]
593         layer, connection = _ann_to_snn_helper(prev, current,
node_type, **kwargs)
594
595         i += 1
596
597         if layer is None or connection is None:
598             continue
599
600         snn.add_layer(layer, name=str(i))
601         snn.add_connection(connection, source=str(i - 1), target=str
(i))
602
603         prev = layer
604
605         current = children[-1]
606         layer, connection = _ann_to_snn_helper(
607             prev, current, node_type, last=True, **kwargs
608         )
609
610         i += 1
611
612         if layer is not None or connection is not None:
613             snn.add_layer(layer, name=str(i))
614             snn.add_connection(connection, source=str(i - 1), target=str
(i))
615
616     return snn

```

Listing B.20: Conversion

B.7 Model

```

1 from .models import (
2     TwoLayerNetwork,
3     DiehlAndCook2015,
4     DiehlAndCook2015v2,
5     IncreasingInhibitionNetwork,
6     LocallyConnectedNetwork,
7 )

```

Listing B.21: Initialization

```

1 from typing import Optional, Union, Tuple, List, Sequence, Iterable
2
3 import numpy as np
4 import torch
5 from scipy.spatial.distance import euclidean
6 from torch.nn.modules.utils import _pair
7 import torch.nn as nn
8 from torchvision import models
9
10 from ..learning import PostPre

```

```

11 from ..network import Network
12 from ..network.nodes import Input, LIFNodes, DiehlAndCookNodes
13 from ..network.topology import Connection, LocalConnection
14
15
16 class TwoLayerNetwork(Network):
17     # language=rst
18     """
19     Implements an ‘‘Input’’ instance connected to a ‘‘LIFNodes’’
20     instance with a
21     fully-connected ‘‘Connection’’.
22     """
23
24     def __init__(
25         self,
26         n_inpt: int,
27         n_neurons: int = 100,
28         dt: float = 1.0,
29         wmin: float = 0.0,
30         wmax: float = 1.0,
31         nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
32         reduction: Optional[callable] = None,
33         norm: float = 78.4,
34     ) -> None:
35         # language=rst
36         """
37         Constructor for class ‘‘TwoLayerNetwork’’.
38
39         :param n_inpt: Number of input neurons. Matches the 1D size
40         of the input data.
41         :param n_neurons: Number of neurons in the ‘‘LIFNodes’’
42         population.
43         :param dt: Simulation time step.
44         :param nu: Single or pair of learning rates for pre- and
45         post-synaptic events,
46         respectively.
47         :param reduction: Method for reducing parameter updates
48         along the minibatch
49         dimension.
50         :param wmin: Minimum allowed weight on ‘‘Input’’ to ‘‘
51         LIFNodes’’ synapses.
52         :param wmax: Maximum allowed weight on ‘‘Input’’ to ‘‘
53         LIFNodes’’ synapses.
54         :param norm: ‘‘Input’’ to ‘‘LIFNodes’’ layer connection
55         weights normalization
56         constant.
57         """
58         super().__init__(dt=dt)
59
60         self.n_inpt = n_inpt
61         self.n_neurons = n_neurons
62         self.dt = dt
63
64         self.add_layer(Input(n=self.n_inpt, traces=True, tc_trace

```

```

=20.0), name="X")
57     self.add_layer(
58         LIFNodes(
59             n=self.n_neurons,
60             traces=True,
61             rest=-65.0,
62             reset=-65.0,
63             thresh=-52.0,
64             refrac=5,
65             tc_decay=100.0,
66             tc_trace=20.0,
67         ),
68         name="Y",
69     )
70
71     w = 0.3 * torch.rand(self.n_inpt, self.n_neurons)
72     self.add_connection(
73         Connection(
74             source=self.layers["X"],
75             target=self.layers["Y"],
76             w=w,
77             update_rule=PostPre,
78             nu=nu,
79             reduction=reduction,
80             wmin=wmin,
81             wmax=wmax,
82             norm=norm,
83         ),
84         source="X",
85         target="Y",
86     )
87
88
89 class DiehlAndCook2015(Network):
90     # language=rst
91     """
92     Implements the spiking neural network architecture from '(Diehl
93     & Cook 2015)
94     <https://www.frontiersin.org/articles/10.3389/fncom.2015.00099/
95     full>'.
96     """
97     def __init__(
98         self,
99         n_inpt: int,
100         n_neurons: int = 100,
101         exc: float = 22.5,
102         inh: float = 17.5,
103         dt: float = 1.0,
104         nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
105         reduction: Optional[callable] = None,
106         wmin: float = 0.0,
107         wmax: float = 1.0,
108         norm: float = 78.4,

```

```

108     theta_plus: float = 0.05,
109     tc_theta_decay: float = 1e7,
110     inpt_shape: Optional[Iterable[int]] = None,
111 ) -> None:
112     # language=rst
113     """
114     Constructor for class ‘‘DiehlAndCook2015‘‘.
115
116     :param n_inpt: Number of input neurons. Matches the 1D size
117     of the input data.
118     :param n_neurons: Number of excitatory, inhibitory neurons.
119     :param exc: Strength of synapse weights from excitatory to
120     inhibitory layer.
121     :param inh: Strength of synapse weights from inhibitory to
122     excitatory layer.
123     :param dt: Simulation time step.
124     :param nu: Single or pair of learning rates for pre- and
125     post-synaptic events,
126     respectively.
127     :param reduction: Method for reducing parameter updates
128     along the minibatch
129     dimension.
130     :param wmin: Minimum allowed weight on input to excitatory
131     synapses.
132     :param wmax: Maximum allowed weight on input to excitatory
133     synapses.
134     :param norm: Input to excitatory layer connection weights
135     normalization
136     constant.
137     :param theta_plus: On-spike increment of ‘‘DiehlAndCookNodes
138     ‘‘ membrane
139     threshold potential.
140     :param tc_theta_decay: Time constant of ‘‘DiehlAndCookNodes
141     ‘‘ threshold
142     potential decay.
143     :param inpt_shape: The dimensionality of the input layer.
144     """
145     super().__init__(dt=dt)
146
147     self.n_inpt = n_inpt
148     self.inpt_shape = inpt_shape
149     self.n_neurons = n_neurons
150     self.exc = exc
151     self.inh = inh
152     self.dt = dt
153
154     # Layers
155     input_layer = Input(
156         n=self.n_inpt, shape=self.inpt_shape, traces=True,
157         tc_trace=20.0
158     )
159     exc_layer = DiehlAndCookNodes(
160         n=self.n_neurons,
161         traces=True,

```

```

151         rest=-65.0,
152         reset=-60.0,
153         thresh=-52.0,
154         refrac=5,
155         tc_decay=100.0,
156         tc_trace=20.0,
157         theta_plus=theta_plus,
158         tc_theta_decay=tc_theta_decay,
159     )
160     inh_layer = LIFNodes(
161         n=self.n_neurons,
162         traces=False,
163         rest=-60.0,
164         reset=-45.0,
165         thresh=-40.0,
166         tc_decay=10.0,
167         refrac=2,
168         tc_trace=20.0,
169     )
170
171     # Connections
172     w = 0.3 * torch.rand(self.n_inpt, self.n_neurons)
173     input_exc_conn = Connection(
174         source=input_layer,
175         target=exc_layer,
176         w=w,
177         update_rule=PostPre,
178         nu=nu,
179         reduction=reduction,
180         wmin=wmin,
181         wmax=wmax,
182         norm=norm,
183     )
184     w = self.exc * torch.diag(torch.ones(self.n_neurons))
185     exc_inh_conn = Connection(
186         source=exc_layer, target=inh_layer, w=w, wmin=0, wmax=
self.exc
187     )
188     w = -self.inh * (
189         torch.ones(self.n_neurons, self.n_neurons)
190         - torch.diag(torch.ones(self.n_neurons))
191     )
192     inh_exc_conn = Connection(
193         source=inh_layer, target=exc_layer, w=w, wmin=-self.inh,
wmax=0
194     )
195
196     # Add to network
197     self.add_layer(input_layer, name="X")
198     self.add_layer(exc_layer, name="Ae")
199     self.add_layer(inh_layer, name="Ai")
200     self.add_connection(input_exc_conn, source="X", target="Ae")
201     self.add_connection(exc_inh_conn, source="Ae", target="Ai")
202     self.add_connection(inh_exc_conn, source="Ai", target="Ae")

```



```

203
204
205 class DiehlAndCook2015v2(Network):
206     # language=rst
207     """
208     Slightly modifies the spiking neural network architecture from
209     '(Diehl & Cook 2015)
210     <https://www.frontiersin.org/articles/10.3389/fncom.2015.00099/
211     full>'_ by removing
212     the inhibitory layer and replacing it with a recurrent
213     inhibitory connection in the
214     output layer (what used to be the excitatory layer).
215     """
216
217     def __init__(
218         self,
219         n_inpt: int,
220         n_neurons: int = 100,
221         inh: float = 17.5,
222         dt: float = 1.0,
223         nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
224         reduction: Optional[callable] = None,
225         wmin: Optional[float] = 0.0,
226         wmax: Optional[float] = 1.0,
227         norm: float = 78.4,
228         theta_plus: float = 0.05,
229         tc_theta_decay: float = 1e7,
230         inpt_shape: Optional[Iterable[int]] = None,
231     ) -> None:
232         # language=rst
233         """
234         Constructor for class 'DiehlAndCook2015v2'.
235
236         :param n_inpt: Number of input neurons. Matches the 1D size
237         of the input data.
238         :param n_neurons: Number of excitatory, inhibitory neurons.
239         :param inh: Strength of synapse weights from inhibitory to
240         excitatory layer.
241         :param dt: Simulation time step.
242         :param nu: Single or pair of learning rates for pre- and
243         post-synaptic events,
244         respectively.
245         :param reduction: Method for reducing parameter updates
246         along the minibatch
247         dimension.
248         :param wmin: Minimum allowed weight on input to excitatory
249         synapses.
250         :param wmax: Maximum allowed weight on input to excitatory
251         synapses.
252         :param norm: Input to excitatory layer connection weights
253         normalization
254         constant.
255         :param theta_plus: On-spike increment of 'DiehlAndCookNodes
256         ' membrane

```

```

246         threshold potential.
247         :param tc_theta_decay: Time constant of ‘‘DiehlAndCookNodes
‘‘ threshold
248             potential decay.
249         :param inpt_shape: The dimensionality of the input layer.
250         """
251         super().__init__(dt=dt)
252
253         self.n_inpt = n_inpt
254         self.inpt_shape = inpt_shape
255         self.n_neurons = n_neurons
256         self.inh = inh
257         self.dt = dt
258
259         input_layer = Input(
260             n=self.n_inpt, shape=self.inpt_shape, traces=True,
tc_trace=20.0
261         )
262         self.add_layer(input_layer, name="X")
263
264         output_layer = DiehlAndCookNodes(
265             n=self.n_neurons,
266             traces=True,
267             rest=-65.0,
268             reset=-60.0,
269             thresh=-52.0,
270             refrac=5,
271             tc_decay=100.0,
272             tc_trace=20.0,
273             theta_plus=theta_plus,
274             tc_theta_decay=tc_theta_decay,
275         )
276         self.add_layer(output_layer, name="Y")
277
278         w = 0.3 * torch.rand(self.n_inpt, self.n_neurons)
279         input_connection = Connection(
280             source=self.layers["X"],
281             target=self.layers["Y"],
282             w=w,
283             update_rule=PostPre,
284             nu=nu,
285             reduction=reduction,
286             wmin=wmin,
287             wmax=wmax,
288             norm=norm,
289         )
290         self.add_connection(input_connection, source="X", target="Y"
)
291
292         w = -self.inh * (
293             torch.ones(self.n_neurons, self.n_neurons)
294             - torch.diag(torch.ones(self.n_neurons)))
295         )
296         recurrent_connection = Connection(

```

```

297         source=self.layers["Y"],
298         target=self.layers["Y"],
299         w=w,
300         wmin=-self.inh,
301         wmax=0,
302     )
303     self.add_connection(recurrent_connection, source="Y", target
304                         ="Y")
305
306 class IncreasingInhibitionNetwork(Network):
307     # language=rst
308     """
309     Implements the inhibitory layer structure of the spiking neural
310     network architecture
311     from '(Hazan et al. 2018) <https://arxiv.org/abs/1807.09374>'_
312     """
313     def __init__(
314         self,
315         n_input: int,
316         n_neurons: int = 100,
317         start_inhib: float = 1.0,
318         max_inhib: float = 100.0,
319         dt: float = 1.0,
320         nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
321         reduction: Optional[callable] = None,
322         wmin: float = 0.0,
323         wmax: float = 1.0,
324         norm: float = 78.4,
325         theta_plus: float = 0.05,
326         tc_theta_decay: float = 1e7,
327     ) -> None:
328         # language=rst
329         """
330         Constructor for class 'IncreasingInhibitionNetwork'.
331
332         :param n_inpt: Number of input neurons. Matches the 1D size
333         of the input data.
334         :param n_neurons: Number of excitatory, inhibitory neurons.
335         :param inh: Strength of synapse weights from inhibitory to
336         excitatory layer.
337         :param dt: Simulation time step.
338         :param nu: Single or pair of learning rates for pre- and
339         post-synaptic events,
340         respectively.
341         :param reduction: Method for reducing parameter updates
342         along the minibatch
343         dimension.
344         :param wmin: Minimum allowed weight on input to excitatory
345         synapses.
346         :param wmax: Maximum allowed weight on input to excitatory
347         synapses.
348         :param norm: Input to excitatory layer connection weights

```

```

normalization
    constant.
343     :param theta_plus: On-spike increment of ‘‘DiehlAndCookNodes
344     ‘‘ membrane
345         threshold potential.
346     :param tc_theta_decay: Time constant of ‘‘DiehlAndCookNodes
347     ‘‘ threshold
348         potential decay.
349     """
350     super().__init__(dt=dt)
351
352     self.n_input = n_input
353     self.n_neurons = n_neurons
354     self.n_sqrt = int(np.sqrt(n_neurons))
355     self.start_inhib = start_inhib
356     self.max_inhib = max_inhib
357     self.dt = dt
358
359     input_layer = Input(n=self.n_input, traces=True, tc_trace
=20.0)
360     self.add_layer(input_layer, name="X")
361
362     output_layer = DiehlAndCookNodes(
363         n=self.n_neurons,
364         traces=True,
365         rest=-65.0,
366         reset=-60.0,
367         thresh=-52.0,
368         refrac=5,
369         tc_decay=100.0,
370         tc_trace=20.0,
371         theta_plus=theta_plus,
372         tc_theta_decay=tc_theta_decay,
373     )
374     self.add_layer(output_layer, name="Y")
375
376     w = 0.3 * torch.rand(self.n_input, self.n_neurons)
377     input_output_conn = Connection(
378         source=self.layers["X"],
379         target=self.layers["Y"],
380         w=w,
381         update_rule=PostPre,
382         nu=nu,
383         reduction=reduction,
384         wmin=wmin,
385         wmax=wmax,
386         norm=norm,
387     )
388     self.add_connection(input_output_conn, source="X", target="Y")
389
390     w = torch.zeros(self.n_neurons, self.n_neurons)
391     for i in range(self.n_neurons):
392         for j in range(self.n_neurons):

```

```

392         if i != j:
393             x1, y1 = i // self.n_sqrt, i % self.n_sqrt
394             x2, y2 = j // self.n_sqrt, j % self.n_sqrt
395
396             inhib = self.start_inhib * np.sqrt(euclidean([x1
, y1], [x2, y2]))
397             w[i, j] = -min(self.max_inhib, inhib)
398
399         recurrent_output_conn = Connection(
400             source=self.layers["Y"],
401             target=self.layers["Y"],
402             w=w,
403             wmin=-self.max_inhib,
404             wmax=0,
405         )
406         self.add_connection(recurrent_output_conn, source="Y",
target="Y")
407
408
409 class LocallyConnectedNetwork(Network):
410     # language=rst
411     """
412     Defines a two-layer network in which the input layer is "locally
connected" to the
413     output layer, and the output layer is recurrently inhibited
connected such that
414     neurons with the same input receptive field inhibit each other.
415     """
416
417     def __init__(
418         self,
419         n_inpt: int,
420         input_shape: List[int],
421         kernel_size: Union[int, Tuple[int, int]],
422         stride: Union[int, Tuple[int, int]],
423         n_filters: int,
424         inh: float = 25.0,
425         dt: float = 1.0,
426         nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
427         reduction: Optional[Callable] = None,
428         theta_plus: float = 0.05,
429         tc_theta_decay: float = 1e7,
430         wmin: float = 0.0,
431         wmax: float = 1.0,
432         norm: Optional[float] = 0.2,
433     ) -> None:
434         # language=rst
435         """
436         Constructor for class ‘‘LocallyConnectedNetwork‘‘. Uses ‘‘
DiehlAndCookNodes‘‘ to
437         avoid multiple spikes per timestep in the output layer
population.
438
439         :param n_inpt: Number of input neurons. Matches the 1D size

```

```

of the input data.
440     :param input_shape: Two-dimensional shape of input
population.
441     :param kernel_size: Size of input windows. Integer or two-
tuple of integers.
442     :param stride: Length of horizontal, vertical stride across
input space. Integer
443         or two-tuple of integers.
444     :param n_filters: Number of locally connected filters per
input region. Integer
445         or two-tuple of integers.
446     :param inh: Strength of synapse weights from output layer
back onto itself.
447     :param dt: Simulation time step.
448     :param nu: Single or pair of learning rates for pre- and
post-synaptic events,
449         respectively.
450     :param reduction: Method for reducing parameter updates
along the minibatch
451         dimension.
452     :param wmin: Minimum allowed weight on ‘‘Input‘‘ to ‘‘
DiehlAndCookNodes‘‘
453         synapses.
454     :param wmax: Maximum allowed weight on ‘‘Input‘‘ to ‘‘
DiehlAndCookNodes‘‘
455         synapses.
456     :param theta_plus: On-spike increment of ‘‘DiehlAndCookNodes
‘‘ membrane
457         threshold potential.
458     :param tc_theta_decay: Time constant of ‘‘DiehlAndCookNodes
‘‘ threshold
459         potential decay.
460     :param norm: ‘‘Input‘‘ to ‘‘DiehlAndCookNodes‘‘ layer
connection weights
461         normalization constant.
462     """
463     super().__init__(dt=dt)
464
465     kernel_size = _pair(kernel_size)
466     stride = _pair(stride)
467
468     self.n_inpt = n_inpt
469     self.input_shape = input_shape
470     self.kernel_size = kernel_size
471     self.stride = stride
472     self.n_filters = n_filters
473     self.inh = inh
474     self.dt = dt
475     self.theta_plus = theta_plus
476     self.tc_theta_decay = tc_theta_decay
477     self.wmin = wmin
478     self.wmax = wmax
479     self.norm = norm
480

```

```

481     if kernel_size == input_shape:
482         conv_size = [1, 1]
483     else:
484         conv_size = (
485             int((input_shape[0] - kernel_size[0]) / stride[0]) +
1,
486             int((input_shape[1] - kernel_size[1]) / stride[1]) +
1,
487         )
488
489     input_layer = Input(n=self.n_inpt, traces=True, tc_trace
=20.0)
490
491     output_layer = DiehlAndCookNodes(
492         n=self.n_filters * conv_size[0] * conv_size[1],
493         traces=True,
494         rest=-65.0,
495         reset=-60.0,
496         thresh=-52.0,
497         refrac=5,
498         tc_decay=100.0,
499         tc_trace=20.0,
500         theta_plus=theta_plus,
501         tc_theta_decay=tc_theta_decay,
502     )
503     input_output_conn = LocalConnection(
504         input_layer,
505         output_layer,
506         kernel_size=kernel_size,
507         stride=stride,
508         n_filters=n_filters,
509         nu=nu,
510         reduction=reduction,
511         update_rule=PostPre,
512         wmin=wmin,
513         wmax=wmax,
514         norm=norm,
515         input_shape=input_shape,
516     )
517
518     w = torch.zeros(n_filters, *conv_size, n_filters, *conv_size
)
519     for fltr1 in range(n_filters):
520         for fltr2 in range(n_filters):
521             if fltr1 != fltr2:
522                 for i in range(conv_size[0]):
523                     for j in range(conv_size[1]):
524                         w[fltr1, i, j, fltr2, i, j] = -inh
525
526     w = w.view(
527         n_filters * conv_size[0] * conv_size[1],
528         n_filters * conv_size[0] * conv_size[1],
529     )
530     recurrent_conn = Connection(output_layer, output_layer, w=w)

```

```

531
532     self.add_layer(input_layer, name="X")
533     self.add_layer(output_layer, name="Y")
534     self.add_connection(input_output_conn, source="X", target="Y
")
535     self.add_connection(recurrent_conn, source="Y", target="Y")

```

Listing B.22: Models

B.8 Learning

```

1 from .learning import (
2     LearningRule,
3     NoOp,
4     PostPre,
5     WeightDependentPostPre,
6     Hebbian,
7     MSTDP,
8     MSTDPET,
9     Rmax,
10 )

```

Listing B.23: Initialization

```

1 from abc import ABC
2 from typing import Union, Optional, Sequence
3
4 import torch
5 import numpy as np
6
7 from ..network.nodes import SRMONodes
8 from ..network.topology import (
9     AbstractConnection,
10    Connection,
11    Conv2dConnection,
12    LocalConnection,
13 )
14 from ..utils import im2col_indices
15
16
17 class LearningRule(ABC):
18     # language=rst
19     """
20     Abstract base class for learning rules.
21     """
22
23     def __init__(
24         self,
25         connection: AbstractConnection,
26         nu: Optional[Union[float, Sequence[float]]] = None,
27         reduction: Optional[callable] = None,
28         weight_decay: float = 0.0,
29         **kwargs
30     ) -> None:

```



```

31     # language=rst
32     """
33     Abstract constructor for the ‘‘LearningRule’’ object.
34
35     :param connection: An ‘‘AbstractConnection’’ object.
36     :param nu: Single or pair of learning rates for pre- and
37     post-synaptic events.
38     :param reduction: Method for reducing parameter updates
39     along the batch
40     dimension.
41     :param weight_decay: Constant multiple to decay weights by
42     on each iteration.
43     """
44     # Connection parameters.
45     self.connection = connection
46     self.source = connection.source
47     self.target = connection.target
48
49     self.wmin = connection.wmin
50     self.wmax = connection.wmax
51
52     # Learning rate(s).
53     if nu is None:
54         nu = [0.0, 0.0]
55     elif isinstance(nu, float) or isinstance(nu, int):
56         nu = [nu, nu]
57
58     self.nu = nu
59
60     # Parameter update reduction across minibatch dimension.
61     if reduction is None:
62         reduction = torch.mean
63
64     self.reduction = reduction
65
66     # Weight decay.
67     self.weight_decay = weight_decay
68
69     def update(self) -> None:
70         # language=rst
71         """
72         Abstract method for a learning rule update.
73         """
74         # Implement weight decay.
75         if self.weight_decay:
76             self.connection.w -= self.weight_decay * self.connection
77             .w
78
79         # Bound weights.
80         if (
81             self.connection.wmin != -np.inf or self.connection.wmax
82             != np.inf
83         ) and not isinstance(self, NoOp):
84             self.connection.w.clamp_(self.connection.wmin, self.

```

```

connection.wmax)
80
81
82 class NoOp(LearningRule):
83     # language=rst
84     """
85     Learning rule with no effect.
86     """
87
88     def __init__(
89         self,
90         connection: AbstractConnection,
91         nu: Optional[Union[float, Sequence[float]]] = None,
92         reduction: Optional[callable] = None,
93         weight_decay: float = 0.0,
94         **kwargs
95     ) -> None:
96         # language=rst
97         """
98         Abstract constructor for the ‘‘LearningRule‘‘ object.
99
100         :param connection: An ‘‘AbstractConnection‘‘ object.
101         :param nu: Single or pair of learning rates for pre- and
102         post-synaptic events.
103         :param reduction: Method for reducing parameter updates
104         along the batch
105         dimension.
106         :param weight_decay: Constant multiple to decay weights by
107         on each iteration.
108         """
109         super().__init__(
110             connection=connection,
111             nu=nu,
112             reduction=reduction,
113             weight_decay=weight_decay,
114             **kwargs
115         )
116
117     def update(self, **kwargs) -> None:
118         # language=rst
119         """
120         Abstract method for a learning rule update.
121         """
122         super().update()
123
124 class PostPre(LearningRule):
125     # language=rst
126     """
127     Simple STDP rule involving both pre- and post-synaptic spiking
128     activity. By default,
129     pre-synaptic update is negative and the post-synaptic update is
130     positive.
131     """

```

```

128
129     def __init__(
130         self,
131         connection: AbstractConnection,
132         nu: Optional[Union[float, Sequence[float]]] = None,
133         reduction: Optional[callable] = None,
134         weight_decay: float = 0.0,
135         **kwargs
136     ) -> None:
137         # language=rst
138         """
139         Constructor for ‘‘PostPre‘‘ learning rule.
140
141         :param connection: An ‘‘AbstractConnection‘‘ object whose
142         weights the
143         ‘‘PostPre‘‘ learning rule will modify.
144         :param nu: Single or pair of learning rates for pre- and
145         post-synaptic events.
146         :param reduction: Method for reducing parameter updates
147         along the batch
148         dimension.
149         :param weight_decay: Constant multiple to decay weights by
150         on each iteration.
151         """
152         super().__init__(
153             connection=connection,
154             nu=nu,
155             reduction=reduction,
156             weight_decay=weight_decay,
157             **kwargs
158         )
159
160         assert (
161             self.source.traces and self.target.traces
162         ), "Both pre- and post-synaptic nodes must record spike
163         traces."
164
165         if isinstance(connection, (Connection, LocalConnection)):
166             self.update = self._connection_update
167         elif isinstance(connection, Conv2dConnection):
168             self.update = self._conv2d_connection_update
169         else:
170             raise NotImplementedError(
171                 "This learning rule is not supported for this
172                 Connection type."
173             )
174
175     def _connection_update(self, **kwargs) -> None:
176         # language=rst
177         """
178         Post-pre learning rule for ‘‘Connection‘‘ subclass of ‘‘
179         AbstractConnection‘‘
180         class.
181         """

```

```

175     batch_size = self.source.batch_size
176
177     source_s = self.source.s.view(batch_size, -1).unsqueeze(2).
float()
178     source_x = self.source.x.view(batch_size, -1).unsqueeze(2)
179     target_s = self.target.s.view(batch_size, -1).unsqueeze(1).
float()
180     target_x = self.target.x.view(batch_size, -1).unsqueeze(1)
181
182     # Pre-synaptic update.
183     if self.nu[0]:
184         update = self.reduction(torch.bmm(source_s, target_x),
dim=0)
185         self.connection.w -= self.nu[0] * update
186
187     # Post-synaptic update.
188     if self.nu[1]:
189         update = self.reduction(torch.bmm(source_x, target_s),
dim=0)
190         self.connection.w += self.nu[1] * update
191
192     super().update()
193
194     def _conv2d_connection_update(self, **kwargs) -> None:
195         # language=rst
196         """
197         Post-pre learning rule for ‘Conv2dConnection‘ subclass of
198         ‘AbstractConnection‘ class.
199         """
200         # Get convolutional layer parameters.
201         out_channels, _, kernel_height, kernel_width = self.
connection.w.size()
202         padding, stride = self.connection.padding, self.connection.
stride
203         batch_size = self.source.batch_size
204
205         # Reshaping spike traces and spike occurrences.
206         source_x = im2col_indices(
207             self.source.x, kernel_height, kernel_width, padding=
padding, stride=stride
208         )
209         target_x = self.target.x.view(batch_size, out_channels, -1)
210         source_s = im2col_indices(
211             self.source.s.float(),
212             kernel_height,
213             kernel_width,
214             padding=padding,
215             stride=stride,
216         )
217         target_s = self.target.s.view(batch_size, out_channels, -1).
float()
218
219         # Pre-synaptic update.
220         if self.nu[0]:

```

```

221         pre = self.reduction(
222             torch.bmm(target_x, source_s.permute((0, 2, 1))),
dim=0
223         )
224         self.connection.w -= self.nu[0] * pre.view(self.
connection.w.size())
225
226         # Post-synaptic update.
227         if self.nu[1]:
228             post = self.reduction(
229                 torch.bmm(target_s, source_x.permute((0, 2, 1))),
dim=0
230             )
231             self.connection.w += self.nu[1] * post.view(self.
connection.w.size())
232
233             super().update()
234
235
236 class WeightDependentPostPre(LearningRule):
237     # language=rst
238     """
239     STDP rule involving both pre- and post-synaptic spiking activity
240     . The post-synaptic
241     update is positive and the pre- synaptic update is negative, and
242     both are dependent
243     on the magnitude of the synaptic weights.
244     """
245
246     def __init__(
247         self,
248         connection: AbstractConnection,
249         nu: Optional[Union[float, Sequence[float]]] = None,
250         reduction: Optional[Callable] = None,
251         weight_decay: float = 0.0,
252         **kwargs
253     ) -> None:
254         # language=rst
255         """
256         Constructor for ‘‘WeightDependentPostPre’’ learning rule.
257
258         :param connection: An ‘‘AbstractConnection’’ object whose
259         weights the
260         ‘‘WeightDependentPostPre’’ learning rule will modify.
261         :param nu: Single or pair of learning rates for pre- and
262         post-synaptic events.
263         :param reduction: Method for reducing parameter updates
264         along the batch
265         dimension.
266         :param weight_decay: Constant multiple to decay weights by
267         on each iteration.
268         """
269         super().__init__(
270             connection=connection,

```

```

265         nu=nu,
266         reduction=reduction,
267         weight_decay=weight_decay,
268         **kwargs
269     )
270
271     assert self.source.traces, "Pre-synaptic nodes must record
spike traces."
272     assert (
273         connection.wmin != -np.inf and connection.wmax != np.inf
274     ), "Connection must define finite wmin and wmax."
275
276     self.wmin = connection.wmin
277     self.wmax = connection.wmax
278
279     if isinstance(connection, (Connection, LocalConnection)):
280         self.update = self._connection_update
281     elif isinstance(connection, Conv2dConnection):
282         self.update = self._conv2d_connection_update
283     else:
284         raise NotImplementedError(
285             "This learning rule is not supported for this
Connection type."
286         )
287
288     def _connection_update(self, **kwargs) -> None:
289         # language=rst
290         """
291         Post-pre learning rule for ‘‘Connection’’ subclass of ‘‘
AbstractConnection’’
292         class.
293         """
294         batch_size = self.source.batch_size
295
296         source_s = self.source.s.view(batch_size, -1).unsqueeze(2).
float()
297         source_x = self.source.x.view(batch_size, -1).unsqueeze(2)
298         target_s = self.target.s.view(batch_size, -1).unsqueeze(1).
float()
299         target_x = self.target.x.view(batch_size, -1).unsqueeze(1)
300
301         update = 0
302
303         # Pre-synaptic update.
304         if self.nu[0]:
305             outer_product = self.reduction(torch.bmm(source_s,
target_x), dim=0)
306             update -= self.nu[0] * outer_product * (self.connection.
w - self.wmin)
307
308         # Post-synaptic update.
309         if self.nu[1]:
310             outer_product = self.reduction(torch.bmm(source_x,
target_s), dim=0)

```

```

311         update += self.nu[1] * outer_product * (self.wmax - self
    .connection.w)
312
313         self.connection.w += update
314
315         super().update()
316
317     def _conv2d_connection_update(self, **kwargs) -> None:
318         # language=rst
319         """
320         Post-pre learning rule for ‘‘Conv2dConnection‘‘ subclass of
321         ‘‘AbstractConnection‘‘ class.
322         """
323         # Get convolutional layer parameters.
324         (
325             out_channels,
326             in_channels,
327             kernel_height,
328             kernel_width,
329         ) = self.connection.w.size()
330         padding, stride = self.connection.padding, self.connection.
stride
331         batch_size = self.source.batch_size
332
333         # Reshaping spike traces and spike occurrences.
334         source_x = im2col_indices(
335             self.source.x, kernel_height, kernel_width, padding=
padding, stride=stride
336         )
337         target_x = self.target.x.view(batch_size, out_channels, -1)
338         source_s = im2col_indices(
339             self.source.s.float(),
340             kernel_height,
341             kernel_width,
342             padding=padding,
343             stride=stride,
344         )
345         target_s = self.target.s.view(batch_size, out_channels, -1).
float()
346
347         update = 0
348
349         # Pre-synaptic update.
350         if self.nu[0]:
351             pre = self.reduction(
352                 torch.bmm(target_x, source_s.permute((0, 2, 1))),
dim=0
353             )
354             update -= (
355                 self.nu[0]
356                 * pre.view(self.connection.w.size())
357                 * (self.connection.w - self.wmin)
358             )
359

```

```

360     # Post-synaptic update.
361     if self.nu[1]:
362         post = self.reduction(
363             torch.bmm(target_s, source_x.permute((0, 2, 1))),
364             dim=0
365         )
366         update += (
367             self.nu[1]
368             * post.view(self.connection.w.size())
369             * (self.wmax - self.connection.wmin)
370         )
371         self.connection.w += update
372
373     super().update()
374
375
376 class Hebbian(LearningRule):
377     # language=rst
378     """
379     Simple Hebbian learning rule. Pre- and post-synaptic updates are
380     both positive.
381     """
382     def __init__(
383         self,
384         connection: AbstractConnection,
385         nu: Optional[Union[float, Sequence[float]]] = None,
386         reduction: Optional[callable] = None,
387         weight_decay: float = 0.0,
388         **kwargs
389     ) -> None:
390         # language=rst
391         """
392         Constructor for ‘‘Hebbian‘‘ learning rule.
393
394         :param connection: An ‘‘AbstractConnection‘‘ object whose
395         weights the
396         ‘‘Hebbian‘‘ learning rule will modify.
397         :param nu: Single or pair of learning rates for pre- and
398         post-synaptic events.
399         :param reduction: Method for reducing parameter updates
400         along the batch
401         dimension.
402         :param weight_decay: Constant multiple to decay weights by
403         on each iteration.
404         """
405         super().__init__(
406             connection=connection,
407             nu=nu,
408             reduction=reduction,
409             weight_decay=weight_decay,
410             **kwargs
411         )

```



```

408
409     assert (
410         self.source.traces and self.target.traces
411     ), "Both pre- and post-synaptic nodes must record spike
traces."
412
413     if isinstance(connection, (Connection, LocalConnection)):
414         self.update = self._connection_update
415     elif isinstance(connection, Conv2dConnection):
416         self.update = self._conv2d_connection_update
417     else:
418         raise NotImplementedError(
419             "This learning rule is not supported for this
Connection type."
420         )
421
422     def _connection_update(self, **kwargs) -> None:
423         # language=rst
424         """
425         Hebbian learning rule for ‘‘Connection‘‘ subclass of ‘‘
AbstractConnection‘‘
426         class.
427         """
428         batch_size = self.source.batch_size
429
430         source_s = self.source.s.view(batch_size, -1).unsqueeze(2).
float()
431         source_x = self.source.x.view(batch_size, -1).unsqueeze(2)
432         target_s = self.target.s.view(batch_size, -1).unsqueeze(1).
float()
433         target_x = self.target.x.view(batch_size, -1).unsqueeze(1)
434
435         # Pre-synaptic update.
436         update = self.reduction(torch.bmm(source_s, target_x), dim
=0)
437         self.connection.w += self.nu[0] * update
438
439         # Post-synaptic update.
440         update = self.reduction(torch.bmm(source_x, target_s), dim
=0)
441         self.connection.w += self.nu[1] * update
442
443         super().update()
444
445     def _conv2d_connection_update(self, **kwargs) -> None:
446         # language=rst
447         """
448         Hebbian learning rule for ‘‘Conv2dConnection‘‘ subclass of
‘‘AbstractConnection‘‘ class.
449         """
450
451         out_channels, _, kernel_height, kernel_width = self.
connection.w.size()
452         padding, stride = self.connection.padding, self.connection.
stride

```

```

453     batch_size = self.source.batch_size
454
455     # Reshaping spike traces and spike occurrences.
456     source_x = im2col_indices(
457         self.source.x, kernel_height, kernel_width, padding=
padding, stride=stride
458     )
459     target_x = self.target.x.view(batch_size, out_channels, -1)
460     source_s = im2col_indices(
461         self.source.s.float(),
462         kernel_height,
463         kernel_width,
464         padding=padding,
465         stride=stride,
466     )
467     target_s = self.target.s.view(batch_size, out_channels, -1).
float()
468
469     # Pre-synaptic update.
470     pre = self.reduction(torch.bmm(target_x, source_s.permute
((0, 2, 1))), dim=0)
471     self.connection.w += self.nu[0] * pre.view(self.connection.w
.size())
472
473     # Post-synaptic update.
474     post = self.reduction(torch.bmm(target_s, source_x.permute
((0, 2, 1))), dim=0)
475     self.connection.w += self.nu[1] * post.view(self.connection.
w.size())
476
477     super().update()
478
479
480 class MSTDP(LearningRule):
481     # language=rst
482     """
483     Reward-modulated STDP. Adapted from '(Florian 2007)
484     <https://florian.io/papers/2007\_Florian\_Modulated\_STDP.pdf>'.
485     """
486
487     def __init__(
488         self,
489         connection: AbstractConnection,
490         nu: Optional[Union[float, Sequence[float]]] = None,
491         reduction: Optional[callable] = None,
492         weight_decay: float = 0.0,
493         **kwargs
494     ) -> None:
495         # language=rst
496         """
497         Constructor for 'MSTDP' learning rule.
498
499         :param connection: An 'AbstractConnection' object whose
weights the 'MSTDP'

```

```

500         learning rule will modify.
501         :param nu: Single or pair of learning rates for pre- and
post-synaptic events,
502             respectively.
503         :param reduction: Method for reducing parameter updates
along the minibatch
504             dimension.
505         :param weight_decay: Constant multiple to decay weights by
on each iteration.
506
507         Keyword arguments:
508
509         :param tc_plus: Time constant for pre-synaptic firing trace.
510         :param tc_minus: Time constant for post-synaptic firing
trace.
511         """
512         super().__init__(
513             connection=connection,
514             nu=nu,
515             reduction=reduction,
516             weight_decay=weight_decay,
517             **kwargs
518         )
519
520         if isinstance(connection, (Connection, LocalConnection)):
521             self.update = self._connection_update
522         elif isinstance(connection, Conv2dConnection):
523             self.update = self._conv2d_connection_update
524         else:
525             raise NotImplementedError(
526                 "This learning rule is not supported for this
Connection type."
527             )
528
529         self.tc_plus = torch.tensor(kwargs.get("tc_plus", 20.0))
530         self.tc_minus = torch.tensor(kwargs.get("tc_minus", 20.0))
531
532         def _connection_update(self, **kwargs) -> None:
533             # language=rst
534             """
535             MSTDP learning rule for ‘‘Connection’’ subclass of ‘‘
AbstractConnection’’ class.
536
537             Keyword arguments:
538
539             :param Union[float, torch.Tensor] reward: Reward signal from
reinforcement
540                 learning task.
541             :param float a_plus: Learning rate (post-synaptic).
542             :param float a_minus: Learning rate (pre-synaptic).
543             """
544             batch_size = self.source.batch_size
545
546             # Initialize eligibility, P+, and P-.

```

```

547         if not hasattr(self, "p_plus"):
548             self.p_plus = torch.zeros(batch_size, *self.source.shape
)
549         if not hasattr(self, "p_minus"):
550             self.p_minus = torch.zeros(batch_size, *self.target.
shape)
551         if not hasattr(self, "eligibility"):
552             self.eligibility = torch.zeros(batch_size, *self.
connection.w.shape)
553
554         # Reshape pre- and post-synaptic spikes.
555         source_s = self.source.s.view(batch_size, -1).float()
556         target_s = self.target.s.view(batch_size, -1).float()
557
558         # Parse keyword arguments.
559         reward = kwargs["reward"]
560         a_plus = torch.tensor(kwargs.get("a_plus", 1.0))
561         a_minus = torch.tensor(kwargs.get("a_minus", -1.0))
562
563         # Compute weight update based on the eligibility value of
the past timestep.
564         update = reward * self.eligibility
565         self.connection.w += self.nu[0] * self.reduction(update, dim
=0)
566
567         # Update P^+ and P^- values.
568         self.p_plus *= torch.exp(-self.connection.dt / self.tc_plus)
569         self.p_plus += a_plus * source_s
570         self.p_minus *= torch.exp(-self.connection.dt / self.
tc_minus)
571         self.p_minus += a_minus * target_s
572
573         # Calculate point eligibility value.
574         self.eligibility = torch.bmm(
575             self.p_plus.unsqueeze(2), target_s.unsqueeze(1)
576         ) + torch.bmm(source_s.unsqueeze(2), self.p_minus.unsqueeze
(1))
577
578         super().update()
579
580     def _conv2d_connection_update(self, **kwargs) -> None:
581         # language=rst
582         """
583         MSTDP learning rule for ‘‘Conv2dConnection’’ subclass of ‘‘
AbstractConnection’’
584         class.
585
586         Keyword arguments:
587
588         :param Union[float, torch.Tensor] reward: Reward signal from
reinforcement
589             learning task.
590         :param float a_plus: Learning rate (post-synaptic).
591         :param float a_minus: Learning rate (pre-synaptic).

```

```

592     """
593     batch_size = self.source.batch_size
594
595     # Initialize eligibility.
596     if not hasattr(self, "eligibility"):
597         self.eligibility = torch.zeros(batch_size, *self.
connection.w.shape)
598
599     # Parse keyword arguments.
600     reward = kwargs["reward"]
601     a_plus = torch.tensor(kwargs.get("a_plus", 1.0))
602     a_minus = torch.tensor(kwargs.get("a_minus", -1.0))
603
604     batch_size = self.source.batch_size
605
606     # Compute weight update based on the eligibility value of
the past timestep.
607     update = reward * self.eligibility
608     self.connection.w += self.nu[0] * torch.sum(update, dim=0)
609
610     out_channels, _, kernel_height, kernel_width = self.
connection.w.size()
611     padding, stride = self.connection.padding, self.connection.
stride
612
613     # Initialize P+ and P-.
614     if not hasattr(self, "p_plus"):
615         self.p_plus = torch.zeros(batch_size, *self.source.shape
)
616         self.p_plus = im2col_indices(
617             self.p_plus, kernel_height, kernel_width, padding=
padding, stride=stride
618         )
619     if not hasattr(self, "p_minus"):
620         self.p_minus = torch.zeros(batch_size, *self.target.
shape)
621         self.p_minus = self.p_minus.view(batch_size,
out_channels, -1).float()
622
623     # Reshaping spike occurrences.
624     source_s = im2col_indices(
625         self.source.s.float(),
626         kernel_height,
627         kernel_width,
628         padding=padding,
629         stride=stride,
630     )
631     target_s = self.target.s.view(batch_size, out_channels, -1).
float()
632
633     # Update P+ and P- values.
634     self.p_plus *= torch.exp(-self.connection.dt / self.tc_plus)
635     self.p_plus += a_plus * source_s
636     self.p_minus *= torch.exp(-self.connection.dt / self.

```

```

tc_minus)
637     self.p_minus += a_minus * target_s
638
639     # Calculate point eligibility value.
640     self.eligibility = torch.bmm(
641         target_s, self.p_plus.permute((0, 2, 1))
642     ) + torch.bmm(self.p_minus, source_s.permute((0, 2, 1)))
643     self.eligibility = self.eligibility.view(self.connection.w.
size())
644
645     super().update()
646
647
648 class MSTDPET(LearningRule):
649     # language=rst
650     """
651     Reward-modulated STDP with eligibility trace. Adapted from
652     '(Florian 2007) <https://florian.io/papers/2007
\_Florian\_Modulated\_STDP.pdf>'.
653     """
654
655     def __init__(
656         self,
657         connection: AbstractConnection,
658         nu: Optional[Union[float, Sequence[float]]] = None,
659         reduction: Optional[callable] = None,
660         weight_decay: float = 0.0,
661         **kwargs
662     ) -> None:
663         # language=rst
664         """
665         Constructor for 'MSTDPET' learning rule.
666
667         :param connection: An 'AbstractConnection' object whose
weights the
668             'MSTDPET' learning rule will modify.
669         :param nu: Single or pair of learning rates for pre- and
post-synaptic events,
670             respectively.
671         :param reduction: Method for reducing parameter updates
along the minibatch
672             dimension.
673         :param weight_decay: Constant multiple to decay weights by
on each iteration.
674
675         Keyword arguments:
676
677         :param float tc_plus: Time constant for pre-synaptic firing
trace.
678         :param float tc_minus: Time constant for post-synaptic
firing trace.
679         :param float tc_e_trace: Time constant for the eligibility
trace.
680         """

```

```

681     super().__init__(
682         connection=connection,
683         nu=nu,
684         reduction=reduction,
685         weight_decay=weight_decay,
686         **kwargs
687     )
688
689     if isinstance(connection, (Connection, LocalConnection)):
690         self.update = self._connection_update
691     elif isinstance(connection, Conv2dConnection):
692         self.update = self._conv2d_connection_update
693     else:
694         raise NotImplementedError(
695             "This learning rule is not supported for this
696             Connection type."
697         )
698
699     self.tc_plus = torch.tensor(kwargs.get("tc_plus", 20.0))
700     self.tc_minus = torch.tensor(kwargs.get("tc_minus", 20.0))
701     self.tc_e_trace = torch.tensor(kwargs.get("tc_e_trace",
702     25.0))
703
704     def _connection_update(self, **kwargs) -> None:
705         # language=rst
706         """
707         MSTDPET learning rule for ‘‘Connection’’ subclass of ‘‘
708         AbstractConnection’’
709         class.
710
711         Keyword arguments:
712
713         :param Union[float, torch.Tensor] reward: Reward signal from
714         reinforcement
715         learning task.
716         :param float a_plus: Learning rate (post-synaptic).
717         :param float a_minus: Learning rate (pre-synaptic).
718         """
719         # Initialize eligibility, eligibility trace, P+, and P-.
720         if not hasattr(self, "p_plus"):
721             self.p_plus = torch.zeros(self.source.n)
722         if not hasattr(self, "p_minus"):
723             self.p_minus = torch.zeros(self.target.n)
724         if not hasattr(self, "eligibility"):
725             self.eligibility = torch.zeros(*self.connection.w.shape)
726         if not hasattr(self, "eligibility_trace"):
727             self.eligibility_trace = torch.zeros(*self.connection.w.
728             shape)
729
730         # Reshape pre- and post-synaptic spikes.
731         source_s = self.source.s.view(-1).float()
732         target_s = self.target.s.view(-1).float()
733
734         # Parse keyword arguments.

```

```

730     reward = kwargs["reward"]
731     a_plus = torch.tensor(kwargs.get("a_plus", 1.0))
732     a_minus = torch.tensor(kwargs.get("a_minus", -1.0))
733
734     # Calculate value of eligibility trace based on the value
735     # of the point eligibility value of the past timestep.
736     self.eligibility_trace *= torch.exp(-self.connection.dt /
self.tc_e_trace)
737     self.eligibility_trace += self.eligibility / self.tc_e_trace
738
739     # Compute weight update.
740     self.connection.w += (
741         self.nu[0] * self.connection.dt * reward * self.
eligibility_trace
742     )
743
744     # Update P^+ and P^- values.
745     self.p_plus *= torch.exp(-self.connection.dt / self.tc_plus)
746     self.p_plus += a_plus * source_s
747     self.p_minus *= torch.exp(-self.connection.dt / self.
tc_minus)
748     self.p_minus += a_minus * target_s
749
750     # Calculate point eligibility value.
751     self.eligibility = torch.ger(self.p_plus, target_s) + torch.
ger(
752         source_s, self.p_minus
753     )
754
755     super().update()
756
757     def _conv2d_connection_update(self, **kwargs) -> None:
758         # language=rst
759         """
760         MSTDPET learning rule for ‘‘Conv2dConnection‘‘ subclass of
761         ‘‘AbstractConnection‘‘ class.
762
763         Keyword arguments:
764
765         :param Union[float, torch.Tensor] reward: Reward signal from
reinforcement
766         learning task.
767         :param float a_plus: Learning rate (post-synaptic).
768         :param float a_minus: Learning rate (pre-synaptic).
769         """
770         batch_size = self.source.batch_size
771
772         # Initialize eligibility and eligibility trace.
773         if not hasattr(self, "eligibility"):
774             self.eligibility = torch.zeros(batch_size, *self.
connection.w.shape)
775         if not hasattr(self, "eligibility_trace"):
776             self.eligibility_trace = torch.zeros(batch_size, *self.
connection.w.shape)

```



```

777
778     # Parse keyword arguments.
779     reward = kwargs["reward"]
780     a_plus = torch.tensor(kwargs.get("a_plus", 1.0))
781     a_minus = torch.tensor(kwargs.get("a_minus", -1.0))
782
783     # Calculate value of eligibility trace based on the value
784     # of the point eligibility value of the past timestep.
785     self.eligibility_trace *= torch.exp(-self.connection.dt /
self.tc_e_trace)
786
787     # Compute weight update.
788     update = reward * self.eligibility_trace
789     self.connection.w += self.nu[0] * self.connection.dt * torch
.sum(update, dim=0)
790
791     out_channels, _, kernel_height, kernel_width = self.
connection.w.size()
792     padding, stride = self.connection.padding, self.connection.
stride
793
794     # Initialize P+ and P-.
795     if not hasattr(self, "p_plus"):
796         self.p_plus = torch.zeros(batch_size, *self.source.shape
)
797         self.p_plus = im2col_indices(
798             self.p_plus, kernel_height, kernel_width, padding=
padding, stride=stride
799         )
800     if not hasattr(self, "p_minus"):
801         self.p_minus = torch.zeros(batch_size, *self.target.
shape)
802         self.p_minus = self.p_minus.view(batch_size,
out_channels, -1).float()
803
804     # Reshaping spike occurrences.
805     source_s = im2col_indices(
806         self.source.s.float(),
807         kernel_height,
808         kernel_width,
809         padding=padding,
810         stride=stride,
811     )
812     target_s = (
813         self.target.s.permute(1, 2, 3, 0).view(batch_size,
out_channels, -1).float()
814     )
815
816     # Update P+ and P- values.
817     self.p_plus *= torch.exp(-self.connection.dt / self.tc_plus)
818     self.p_plus += a_plus * source_s
819     self.p_minus *= torch.exp(-self.connection.dt / self.
tc_minus)
820     self.p_minus += a_minus * target_s

```

```

821
822     # Calculate point eligibility value.
823     self.eligibility = torch.bmm(
824         target_s, self.p_plus.permute((0, 2, 1))
825     ) + torch.bmm(self.p_minus, source_s.permute((0, 2, 1)))
826     self.eligibility = self.eligibility.view(self.connection.w.
size())
827
828     super().update()
829
830
831 class Rmax(LearningRule):
832     # language=rst
833     """
834     Reward-modulated learning rule derived from reward maximization
835     principles. Adapted
836     from '(Vasilaki et al., 2009)
837     <https://intranet.physio.unibe.ch/Publikationen/Dokumente/
Vasilaki2009PloSComputBio\_1.pdf>'.
838     """
839     def __init__(
840         self,
841         connection: AbstractConnection,
842         nu: Optional[Union[float, Sequence[float]]] = None,
843         reduction: Optional[callable] = None,
844         weight_decay: float = 0.0,
845         **kwargs
846     ) -> None:
847         # language=rst
848         """
849         Constructor for 'R-max' learning rule.
850
851         :param connection: An 'AbstractConnection' object whose
852         weights the 'R-max'
853         learning rule will modify.
854         :param nu: Single or pair of learning rates for pre- and
855         post-synaptic events,
856         respectively.
857         :param reduction: Method for reducing parameter updates
858         along the minibatch
859         dimension.
860         :param weight_decay: Constant multiple to decay weights by
861         on each iteration.
862
863         Keyword arguments:
864
865         :param float tc_c: Time constant for balancing naive Hebbian
866         and policy gradient
867         learning.
868         :param float tc_e_trace: Time constant for the eligibility
869         trace.
870         """
871         super().__init__(

```

```

866         connection=connection,
867         nu=nu,
868         reduction=reduction,
869         weight_decay=weight_decay,
870         **kwargs
871     )
872
873     # Trace is needed for computing epsilon.
874     assert (
875         self.source.traces and self.source.traces_additive
876     ), "Pre-synaptic nodes must use additive spike traces."
877
878     # Derivation of R-max depends on stochastic SRM neurons!
879     assert isinstance(
880         self.target, SRMONodes
881     ), "R-max needs stochastically firing neurons, use SRMONodes
882     ."
883
884     if isinstance(connection, (Connection, LocalConnection)):
885         self.update = self._connection_update
886     else:
887         raise NotImplementedError(
888             "This learning rule is not supported for this
889             Connection type."
890         )
891
892     self.tc_c = torch.tensor(
893         kwargs.get("tc_c", 5.0)
894     ) # 0 for pure naive Hebbian, inf for pure policy gradient.
895     self.tc_e_trace = torch.tensor(kwargs.get("tc_e_trace",
896     25.0))
897
898     def _connection_update(self, **kwargs) -> None:
899         # language=rst
900         """
901         R-max learning rule for ‘‘Connection’’ subclass of ‘‘
902         AbstractConnection’’ class.
903
904         Keyword arguments:
905
906         :param Union[float, torch.Tensor] reward: Reward signal from
907         reinforcement
908         learning task.
909         """
910         # Initialize eligibility trace.
911         if not hasattr(self, "eligibility_trace"):
912             self.eligibility_trace = torch.zeros(*self.connection.w.
913             shape)
914
915         # Reshape variables.
916         target_s = self.target.s.view(-1).float()
917         target_s_prob = self.target.s_prob.view(-1)
918         source_x = self.source.x.view(-1)
919

```

```

914     # Parse keyword arguments.
915     reward = kwargs["reward"]
916
917     # New eligibility trace.
918     self.eligibility_trace *= 1 - self.connection.dt / self.
tc_e_trace
919     self.eligibility_trace += (
920         target_s
921         - (target_s_prob / (1.0 + self.tc_c / self.connection.dt
* target_s_prob))
922         ) * source_x[:, None]
923
924     # Compute weight update.
925     self.connection.w += self.nu[0] * reward * self.
eligibility_trace
926
927     super().update()

```

Listing B.24: Learning

```

1 from abc import ABC, abstractmethod
2
3 import torch
4
5
6 class AbstractReward(ABC):
7     # language=rst
8     """
9     Abstract base class for reward computation.
10    """
11
12    @abstractmethod
13    def compute(self, **kwargs) -> None:
14        # language=rst
15        """
16        Computes/modifies reward.
17        """
18        pass
19
20    @abstractmethod
21    def update(self, **kwargs) -> None:
22        # language=rst
23        """
24        Updates internal variables needed to modify reward. Usually
called once per
25        episode.
26        """
27        pass
28
29
30 class MovingAvgRPE(AbstractReward):
31     # language=rst
32     """
33     Computes reward prediction error (RPE) based on an exponential

```

```

moving average (EMA)
of past rewards.
"""
34
35
36
37 def __init__(self, **kwargs) -> None:
38     # language=rst
39     """
40     Constructor for EMA reward prediction error.
41     """
42     self.reward_predict = torch.tensor(0.0) # Predicted reward
(per step).
43     self.reward_predict_episode = torch.tensor(0.0) # Predicted
reward per episode.
44     self.rewards_predict_episode = (
45         []
46     ) # List of predicted rewards per episode (used for
plotting).
47
48 def compute(self, **kwargs) -> torch.Tensor:
49     # language=rst
50     """
51     Computes the reward prediction error using EMA.
52
53     Keyword arguments:
54
55     :param Union[float, torch.Tensor] reward: Current reward.
56     :return: Reward prediction error.
57     """
58     # Get keyword arguments.
59     reward = kwargs["reward"]
60
61     return reward - self.reward_predict
62
63 def update(self, **kwargs) -> None:
64     # language=rst
65     """
66     Updates the EMAs. Called once per episode.
67
68     Keyword arguments:
69
70     :param Union[float, torch.Tensor] accumulated_reward: Reward
accumulated over
71     one episode.
72     :param int steps: Steps in that episode.
73     :param float ema_window: Width of the averaging window.
74     """
75     # Get keyword arguments.
76     accumulated_reward = kwargs["accumulated_reward"]
77     steps = torch.tensor(kwargs["steps"]).float()
78     ema_window = torch.tensor(kwargs.get("ema_window", 10.0))
79
80     # Compute average reward per step.
81     reward = accumulated_reward / steps
82

```

```

83     # Update EMAs.
84     self.reward_predict = (
85         1 - 1 / ema_window
86     ) * self.reward_predict + 1 / ema_window * reward
87     self.reward_predict_episode = (
88         1 - 1 / ema_window
89     ) * self.reward_predict_episode + 1 / ema_window *
accumulated_reward
90     self.rewards_predict_episode.append(self.
reward_predict_episode.item())

```

Listing B.25: Reward

B.9 Evaluation

```

1 from .evaluation import (
2     assign_labels,
3     logreg_fit,
4     logreg_predict,
5     all_activity,
6     proportion_weighting,
7     ngram,
8     update_ngram_scores,
9 )

```

Listing B.26: Initialization

```

1 from itertools import product
2 from typing import Optional, Tuple, Dict
3
4 import torch
5 from sklearn.linear_model import LogisticRegression
6
7
8 def assign_labels(
9     spikes: torch.Tensor,
10    labels: torch.Tensor,
11    n_labels: int,
12    rates: Optional[torch.Tensor] = None,
13    alpha: float = 1.0,
14 ) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
15     # language=rst
16     """
17     Assign labels to the neurons based on highest average spiking
activity.
18
19     :param spikes: Binary tensor of shape ``(n_samples, time,
n_neurons)`` of a single
20         layer's spiking activity.
21     :param labels: Vector of shape ``(n_samples,)`` with data labels
corresponding to
22         spiking activity.
23     :param n_labels: The number of target labels in the data.

```

```

24     :param rates: If passed, these represent spike rates from a
25     previous
26         ‘‘assign_labels()’’ call.
27     :param alpha: Rate of decay of label assignments.
28     :return: Tuple of class assignments, per-class spike proportions
29     , and per-class
30     firing rates.
31     """
32     n_neurons = spikes.size(2)
33
34     if rates is None:
35         rates = torch.zeros(n_neurons, n_labels)
36
37     # Sum over time dimension (spike ordering doesn't matter).
38     spikes = spikes.sum(1)
39
40     for i in range(n_labels):
41         # Count the number of samples with this label.
42         n_labeled = torch.sum(labels == i).float()
43
44         if n_labeled > 0:
45             # Get indices of samples with this label.
46             indices = torch.nonzero(labels == i).view(-1)
47
48             # Compute average firing rates for this label.
49             rates[:, i] = alpha * rates[:, i] + (
50                 torch.sum(spikes[indices], 0) / n_labeled
51             )
52
53     # Compute proportions of spike activity per class.
54     proportions = rates / rates.sum(1, keepdim=True)
55     proportions[proportions != proportions] = 0 # Set NaNs to 0
56
57     # Neuron assignments are the labels they fire most for.
58     assignments = torch.max(proportions, 1)[1]
59
60     return assignments, proportions, rates
61
62 def logreg_fit(
63     spikes: torch.Tensor, labels: torch.Tensor, logreg:
64     LogisticRegression
65 ) -> LogisticRegression:
66     # language=rst
67     """
68     (Re)fit logistic regression model to spike data summed over time
69     .
70
71     :param spikes: Summed (over time) spikes of shape ‘‘(n_examples,
72     time, n_neurons)’’.
73     :param labels: Vector of shape ‘‘(n_samples,)’’ with data labels
74     corresponding to
75     spiking activity.
76     :param logreg: Logistic regression model from previous fits.

```

```

72     :return: (Re)fitted logistic regression model.
73     """
74     # (Re)fit logistic regression model.
75     logreg.fit(spikes, labels)
76     return logreg
77
78
79 def logreg_predict(spikes: torch.Tensor, logreg: LogisticRegression)
80     -> torch.Tensor:
81     # language=rst
82     """
83     Predicts classes according to spike data summed over time.
84
85     :param spikes: Summed (over time) spikes of shape '(n_examples,
86     time, n_neurons)'.
87     :param logreg: Logistic regression model from previous fits.
88     :return: Predictions per example.
89     """
90     # Make class label predictions.
91     if not hasattr(logreg, "coef_") or logreg.coef_ is None:
92         return -1 * torch.ones(spikes.size(0)).long()
93
94     predictions = logreg.predict(spikes)
95     return torch.Tensor(predictions).long()
96
97 def all_activity(
98     spikes: torch.Tensor, assignments: torch.Tensor, n_labels: int
99 ) -> torch.Tensor:
100     # language=rst
101     """
102     Classify data with the label with highest average spiking
103     activity over all neurons.
104
105     :param spikes: Binary tensor of shape '(n_samples, time,
106     n_neurons)' of a layer's
107     spiking activity.
108     :param assignments: A vector of shape '(n_neurons,)' of neuron
109     label assignments.
110     :param n_labels: The number of target labels in the data.
111     :return: Predictions tensor of shape '(n_samples,)' resulting
112     from the "all
113     activity" classification scheme.
114     """
115     n_samples = spikes.size(0)
116
117     # Sum over time dimension (spike ordering doesn't matter).
118     spikes = spikes.sum(1)
119
120     rates = torch.zeros(n_samples, n_labels)
121     for i in range(n_labels):
122         # Count the number of neurons with this label assignment.
123         n_assigns = torch.sum(assignments == i).float()

```



```

120         if n_assigns > 0:
121             # Get indices of samples with this label.
122             indices = torch.nonzero(assignments == i).view(-1)
123
124             # Compute layer-wise firing rate for this label.
125             rates[:, i] = torch.sum(spikes[:, indices], 1) /
n_assigns
126
127         # Predictions are arg-max of layer-wise firing rates.
128         return torch.sort(rates, dim=1, descending=True)[1][:, 0]
129
130
131 def proportion_weighting(
132     spikes: torch.Tensor,
133     assignments: torch.Tensor,
134     proportions: torch.Tensor,
135     n_labels: int,
136 ) -> torch.Tensor:
137     # language=rst
138     """
139     Classify data with the label with highest average spiking
140     activity over all neurons,
141     weighted by class-wise proportion.
142
143     :param spikes: Binary tensor of shape ``(n_samples, time,
n_neurons)`` of a single
144         layer's spiking activity.
145     :param assignments: A vector of shape ``(n_neurons,)`` of neuron
label assignments.
146     :param proportions: A matrix of shape ``(n_neurons, n_labels)``
giving the per-class
147         proportions of neuron spiking activity.
148     :param n_labels: The number of target labels in the data.
149     :return: Predictions tensor of shape ``(n_samples,)`` resulting
from the "proportion
150         weighting" classification scheme.
151     """
152     n_samples = spikes.size(0)
153
154     # Sum over time dimension (spike ordering doesn't matter).
155     spikes = spikes.sum(1)
156
157     rates = torch.zeros(n_samples, n_labels)
158     for i in range(n_labels):
159         # Count the number of neurons with this label assignment.
160         n_assigns = torch.sum(assignments == i).float()
161
162         if n_assigns > 0:
163             # Get indices of samples with this label.
164             indices = torch.nonzero(assignments == i).view(-1)
165
166             # Compute layer-wise firing rate for this label.
167             rates[:, i] += (
                torch.sum((proportions[:, i] * spikes)[:, indices],

```

```

168         1) / n_assigns
169         )
170
171         # Predictions are arg-max of layer-wise firing rates.
172         predictions = torch.sort(rates, dim=1, descending=True)[1][:, 0]
173
174         return predictions
175
176 def ngram(
177     spikes: torch.Tensor,
178     ngram_scores: Dict[Tuple[int, ...], torch.Tensor],
179     n_labels: int,
180     n: int,
181 ) -> torch.Tensor:
182     # language=rst
183     """
184     Predicts between ‘n_labels‘ using ‘ngram_scores‘.
185
186     :param spikes: Spikes of shape ‘(n_examples, time, n_neurons)‘.
187     :param ngram_scores: Previously recorded scores to update.
188     :param n_labels: The number of target labels in the data.
189     :param n: The max size of n-gram to use.
190     :return: Predictions per example.
191     """
192     predictions = []
193     for activity in spikes:
194         score = torch.zeros(n_labels)
195
196         # Aggregate all of the firing neurons' indices
197         fire_order = []
198         for t in range(activity.size()[0]):
199             ordering = torch.nonzero(activity[t].view(-1))
200             if ordering.numel() > 0:
201                 fire_order += ordering[:, 0].tolist()
202
203         # Consider all n-gram sequences.
204         for j in range(len(fire_order) - n):
205             if tuple(fire_order[j : j + n]) in ngram_scores:
206                 score += ngram_scores[tuple(fire_order[j : j + n])]
207
208         predictions.append(torch.argmax(score))
209
210     return torch.tensor(predictions).long()
211
212
213 def update_ngram_scores(
214     spikes: torch.Tensor,
215     labels: torch.Tensor,
216     n_labels: int,
217     n: int,
218     ngram_scores: Dict[Tuple[int, ...], torch.Tensor],
219 ) -> Dict[Tuple[int, ...], torch.Tensor]:

```

```

220     # language=rst
221     """
222     Updates ngram scores by adding the count of each spike sequence
223     of length n from the
224     past ‘‘n_examples‘‘.
225
226     :param spikes: Spikes of shape ‘‘(n_examples, time, n_neurons)
227     ‘‘.
228     :param labels: The ground truth labels of shape ‘‘(n_examples)
229     ‘‘.
230     :param n_labels: The number of target labels in the data.
231     :param n: The max size of n-gram to use.
232     :param ngram_scores: Previously recorded scores to update.
233     :return: Dictionary mapping n-grams to vectors of per-class
234     spike counts.
235     """
236
237     for i, activity in enumerate(spikes):
238         # Obtain firing order for spiking activity.
239         fire_order = []
240
241         # Aggregate all of the firing neurons' indices.
242         for t in range(spikes.size(1)):
243             # Gets the indices of the neurons which fired on this
244             timestep.
245             ordering = torch.nonzero(activity[t]).view(-1)
246             if ordering.numel() > 0: # If there was more than one
247             spike...
248                 # Add the indices of spiked neurons to the fire
249                 ordering.
250
251                 ordering = ordering.tolist()
252                 fire_order.append(ordering)
253
254         # Check every sequence of length n.
255         for order in zip(*(fire_order[k:] for k in range(n))):
256             for sequence in product(*order):
257                 if sequence not in ngram_scores:
258                     ngram_scores[sequence] = torch.zeros(n_labels)
259
260                 ngram_scores[sequence][int(labels[i])] += 1
261
262     return ngram_scores

```

Listing B.27: Evaluation

B.10 Analysis

```

1 from . import plotting, visualization, pipeline_analysis

```

Listing B.28: Initialization

```

1 from abc import ABC, abstractmethod
2 from typing import Dict, Optional
3
4 import matplotlib.pyplot as plt

```

```

5 import numpy as np
6 import pandas as pd
7 import torch
8 from tensorboardX import SummaryWriter
9 from torchvision.utils import make_grid
10
11 from .plotting import plot_spikes, plot_voltages,
    plot_conv2d_weights
12 from ..utils import reshape_conv2d_weights
13
14
15 class PipelineAnalyzer(ABC):
16     # language=rst
17     """
18     Responsible for pipeline analysis. Subclasses maintain state
19     information related to plotting or logging.
20     """
21
22     @abstractmethod
23     def finalize_step(self) -> None:
24         # language=rst
25         """
26         Flush the output from the current step.
27         """
28         pass
29
30     @abstractmethod
31     def plot_obs(self, obs: torch.Tensor, tag: str = "obs", step:
    int = None) -> None:
32         # language=rst
33         """
34         Pulls the observation from PyTorch and sets up for
    Matplotlib
35         plotting.
36
37         :param obs: A 2D array of floats depicting an input image.
38         :param tag: A unique tag to associate the data with.
39         :param step: The step of the pipeline.
40         """
41         pass
42
43     @abstractmethod
44     def plot_reward(
45         self,
46         reward_list: list,
47         reward_window: int = None,
48         tag: str = "reward",
49         step: int = None,
50     ) -> None:
51         # language=rst
52         """
53         Plot the accumulated reward for each episode.
54
55         :param reward_list: The list of recent rewards to be plotted

```

```

56     :param reward_window: The length of the window to compute a
moving average over.
57     :param tag: A unique tag to associate the data with.
58     :param step: The step of the pipeline.
59     """
60     pass
61
62     @abstractmethod
63     def plot_spikes(
64         self,
65         spike_record: Dict[str, torch.Tensor],
66         tag: str = "spike",
67         step: int = None,
68     ) -> None:
69         # language=rst
70         """
71         Plots all spike records inside of ‘‘spike_record‘‘. Keeps
unique
72         unique
73         plots for all unique tags that are given.
74
75         :param spike_record: Dictionary of spikes to be rasterized.
76         :param tag: A unique tag to associate the data with.
77         :param step: The step of the pipeline.
78         """
79         pass
80
81     @abstractmethod
82     def plot_voltages(
83         self,
84         voltage_record: Dict[str, torch.Tensor],
85         thresholds: Optional[Dict[str, torch.Tensor]] = None,
86         tag: str = "voltage",
87         step: int = None,
88     ) -> None:
89         # language=rst
90         """
91         Plots all voltage records and given thresholds. Keeps unique
92         unique
93         plots for all unique tags that are given.
94
95         :param voltage_record: Dictionary of voltages for neurons
inside of networks
96         inside of networks
97         organized by the layer they
98         correspond to.
99         :param thresholds: Optional dictionary of threshold values
for neurons.
100         :param tag: A unique tag to associate the data with.
101         :param step: The step of the pipeline.
102         """
103         pass
104
105     @abstractmethod
106     def plot_conv2d_weights(
107         self, weights: torch.Tensor, tag: str = "conv2d", step: int

```

```

= None
104 ) -> None:
105     # language=rst
106     """
107     Plot a connection weight matrix of a ‘‘Conv2dConnection‘‘.
108
109     :param weights: Weight matrix of ‘‘Conv2dConnection‘‘ object
110     .
111     :param tag: A unique tag to associate the data with.
112     :param step: The step of the pipeline.
113     """
114     pass
115
116 class MatplotlibAnalyzer(PipelineAnalyzer):
117     # language=rst
118     """
119     Renders output using Matplotlib.
120
121     Matplotlib requires objects to be kept around over the full
122     lifetime
123     of the plots; this is done through ‘‘self.plots‘‘. An
124     interactive session
125     is needed so that we can continue processing and just update the
126     plots.
127     """
128     def __init__(self, **kwargs) -> None:
129         # language=rst
130         """
131         Initializes the analyzer.
132
133         Keyword arguments:
134
135         :param str volts_type: Type of plotting for voltages (‘‘"
136         color"‘‘ or ‘‘"line"‘‘).
137         """
138         self.volts_type = kwargs.get("volts_type", "color")
139         plt.ion()
140         self.plots = {}
141
142     def plot_obs(self, obs: torch.Tensor, tag: str = "obs", step:
143     int = None) -> None:
144         # language=rst
145         """
146         Pulls the observation off of torch and sets up for
147         Matplotlib
148         plotting.
149
150         :param obs: A 2D array of floats depicting an input image.
151         :param tag: A unique tag to associate the data with.
152         :param step: The step of the pipeline.
153         """
154         obs = obs.detach().cpu().numpy()

```

```

151     obs = np.transpose(obs, (1, 2, 0)).squeeze()
152
153     if tag in self.plots:
154         obs_ax, obs_im = self.plots[tag]
155     else:
156         obs_ax, obs_im = None, None
157
158     if obs_im is None and obs_ax is None:
159         fig, obs_ax = plt.subplots()
160         obs_ax.set_title("Observation")
161         obs_ax.set_xticks(())
162         obs_ax.set_yticks(())
163         obs_im = obs_ax.imshow(obs, cmap="gray")
164
165         self.plots[tag] = obs_ax, obs_im
166     else:
167         obs_im.set_data(obs)
168
169     def plot_reward(
170         self,
171         reward_list: list,
172         reward_window: int = None,
173         tag: str = "reward",
174         step: int = None,
175     ) -> None:
176         # language=rst
177         """
178         Plot the accumulated reward for each episode.
179
180         :param reward_list: The list of recent rewards to be plotted
181         .
182         :param reward_window: The length of the window to compute a
183         moving average over.
184         :param tag: A unique tag to associate the data with.
185         :param step: The step of the pipeline.
186         """
187         if tag in self.plots:
188             reward_im, reward_ax, reward_plot = self.plots[tag]
189         else:
190             reward_im, reward_ax, reward_plot = None, None, None
191
192         # Compute moving average.
193         if reward_window is not None:
194             # Ensure window size > 0 and < size of reward list.
195             window = max(min(len(reward_list), reward_window), 0)
196
197             # Fastest implementation of moving average.
198             reward_list_ = (
199                 pd.Series(reward_list)
200                 .rolling(window=window, min_periods=1)
201                 .mean()
202                 .values
203             )
204         else:

```

```

203         reward_list_ = reward_list[:]
204
205         if reward_im is None and reward_ax is None:
206             reward_im, reward_ax = plt.subplots()
207             reward_ax.set_title("Accumulated reward")
208             reward_ax.set_xlabel("Episode")
209             reward_ax.set_ylabel("Reward")
210             (reward_plot,) = reward_ax.plot(reward_list_)
211
212             self.plots[tag] = reward_im, reward_ax, reward_plot
213         else:
214             reward_plot.set_data(range(len(reward_list_)),
215 reward_list_)
216             reward_ax.relim()
217             reward_ax.autoscale_view()
218
219     def plot_spikes(
220         self,
221         spike_record: Dict[str, torch.Tensor],
222         tag: str = "spike",
223         step: int = None,
224     ) -> None:
225         # language=rst
226         """
227         Plots all spike records inside of ‘‘spike_record‘‘. Keeps
228         unique
229         plots for all unique tags that are given.
230
231         :param spike_record: Dictionary of spikes to be rasterized.
232         :param tag: A unique tag to associate the data with.
233         :param step: The step of the pipeline.
234         """
235         if tag not in self.plots:
236             self.plots[tag] = plot_spikes(spike_record)
237         else:
238             s_im, s_ax = self.plots[tag]
239             self.plots[tag] = plot_spikes(spike_record, ims=s_im,
240 axes=s_ax)
241
242     def plot_voltages(
243         self,
244         voltage_record: Dict[str, torch.Tensor],
245         thresholds: Optional[Dict[str, torch.Tensor]] = None,
246         tag: str = "voltage",
247         step: int = None,
248     ) -> None:
249         # language=rst
250         """
251         Plots all voltage records and given thresholds. Keeps unique
252         plots for all unique tags that are given.
253
254         :param voltage_record: Dictionary of voltages for neurons
255         inside of networks
256
257         organized by the layer they

```



```

correspond to.
253     :param thresholds: Optional dictionary of threshold values
for neurons.
254     :param tag: A unique tag to associate the data with.
255     :param step: The step of the pipeline.
256     """
257     if tag not in self.plots:
258         self.plots[tag] = plot_voltages(
259             voltage_record, plot_type=self.volts_type,
thresholds=thresholds
260         )
261     else:
262         v_im, v_ax = self.plots[tag]
263         self.plots[tag] = plot_voltages(
264             voltage_record,
265             ims=v_im,
266             axes=v_ax,
267             plot_type=self.volts_type,
268             thresholds=thresholds,
269         )
270
271     def plot_conv2d_weights(
272         self, weights: torch.Tensor, tag: str = "conv2d", step: int
= None
273     ) -> None:
274         # language=rst
275         """
276         Plot a connection weight matrix of a ‘‘Conv2dConnection‘‘.
277
278         :param weights: Weight matrix of ‘‘Conv2dConnection‘‘ object
.
279
280         :param tag: A unique tag to associate the data with.
281         :param step: The step of the pipeline.
282         """
282         wmin = weights.min().item()
283         wmax = weights.max().item()
284
285         if tag not in self.plots:
286             self.plots[tag] = plot_conv2d_weights(weights, wmin,
wmax)
287         else:
288             im = self.plots[tag]
289             plot_conv2d_weights(weights, wmin, wmax, im=im)
290
291     def finalize_step(self) -> None:
292         # language=rst
293         """
294         Flush the output from the current step
295         """
296         plt.draw()
297         plt.pause(1e-8)
298         plt.show()
299
300

```

```

301 class TensorboardAnalyzer(PipelineAnalyzer):
302     def __init__(self, summary_directory: str = "./logs"):
303         # language=rst
304         """
305         Initializes the analyzer.
306
307         :param summary_directory: Directory to save log files.
308         """
309         self.writer = SummaryWriter(summary_directory)
310
311     def finalize_step(self) -> None:
312         # language=rst
313         """
314         No-op for 'TensorboardAnalyzer'.
315         """
316         pass
317
318     def plot_obs(self, obs: torch.Tensor, tag: str = "obs", step:
319 int = None) -> None:
320         # language=rst
321         """
322         Pulls the observation off of torch and sets up for
323 Matplotlib
324 plotting.
325
326         :param obs: A 2D array of floats depicting an input image.
327         :param tag: A unique tag to associate the data with.
328         :param step: The step of the pipeline.
329         """
330         obs_grid = make_grid(obs.float(), nrow=4, normalize=True)
331         self.writer.add_image(tag, obs_grid, step)
332
333     def plot_reward(
334 self,
335 reward_list: list,
336 reward_window: int = None,
337 tag: str = "reward",
338 step: int = None,
339 ) -> None:
340         # language=rst
341         """
342         Plot the accumulated reward for each episode.
343
344         :param reward_list: The list of recent rewards to be plotted
345 .
346         :param reward_window: The length of the window to compute a
347 moving average over.
348         :param tag: A unique tag to associate the data with.
349         :param step: The step of the pipeline.
350         """
351         self.writer.add_scalar(tag, reward_list[-1], step)
352
353     def plot_spikes(
354 self,

```

```

351     spike_record: Dict[str, torch.Tensor],
352     tag: str = "spike",
353     step: int = None,
354 ) -> None:
355     # language=rst
356     """
357     Plots all spike records inside of ‘‘spike_record‘‘. Keeps
unique
358     plots for all unique tags that are given.
359
360     :param spike_record: Dictionary of spikes to be rasterized.
361     :param tag: A unique tag to associate the data with.
362     :param step: The step of the pipeline.
363     """
364     for k, spikes in spike_record.items():
365         # shuffle spikes into 1x1x#NueronsxT
366         spikes = spikes.view(1, 1, -1, spikes.shape[-1]).float()
367         spike_grid_img = make_grid(spikes, nrow=1, pad_value
=0.5)
368
369         self.writer.add_image(tag + "_" + str(k), spike_grid_img
, step)
370
371     def plot_voltages(
372         self,
373         voltage_record: Dict[str, torch.Tensor],
374         thresholds: Optional[Dict[str, torch.Tensor]] = None,
375         tag: str = "voltage",
376         step: int = None,
377     ) -> None:
378         # language=rst
379         """
380         Plots all voltage records and given thresholds. Keeps unique
381         plots for all unique tags that are given.
382
383         :param voltage_record: Dictionary of voltages for neurons
inside of networks
384                                     organized by the layer they
correspond to.
385         :param thresholds: Optional dictionary of threshold values
for neurons.
386         :param tag: A unique tag to associate the data with.
387         :param step: The step of the pipeline.
388         """
389         for k, v in voltage_record.items():
390             # Shuffle voltages into 1x1x#neuronsxT
391             v = v.view(1, 1, -1, v.shape[-1])
392             voltage_grid_img = make_grid(v, nrow=1, pad_value=0)
393
394             self.writer.add_image(tag + "_" + str(k),
voltage_grid_img, step)
395
396     def plot_conv2d_weights(
397         self, weights: torch.Tensor, tag: str = "conv2d", step: int

```

```

= None
398 ) -> None:
399     # language=rst
400     """
401     Plot a connection weight matrix of a ‘‘Conv2dConnection‘‘.
402
403     :param weights: Weight matrix of ‘‘Conv2dConnection‘‘ object
404     .
405     :param tag: A unique tag to associate the data with.
406     :param step: The step of the pipeline.
407     """
408     reshaped = reshape_conv2d_weights(weights).unsqueeze(0)
409
410     reshaped -= reshaped.min()
411     reshaped /= reshaped.max()
412
413     self.writer.add_image(tag, reshaped, step)

```

Listing B.29: Pipeline analysis

```

1 import torch
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 from matplotlib.axes import Axes
6 from matplotlib.image import AxesImage
7 from torch.nn.modules.utils import _pair
8 from matplotlib.collections import PathCollection
9 from mpl_toolkits.axes_grid1 import make_axes_locatable
10 from typing import Tuple, List, Optional, Sized, Dict, Union
11
12 from ..utils import reshape_locally_connected_weights,
13     reshape_conv2d_weights
14
15 plt.ion()
16
17 def plot_input(
18     image: torch.Tensor,
19     inpt: torch.Tensor,
20     label: Optional[int] = None,
21     axes: List[Axes] = None,
22     ims: List[AxesImage] = None,
23     figsize: Tuple[int, int] = (8, 4),
24 ) -> Tuple[List[Axes], List[AxesImage]]:
25     # language=rst
26     """
27     Plots a two-dimensional image and its corresponding spike-train
28     representation.
29
30     :param image: A 2D array of floats depicting an input image.
31     :param inpt: A 2D array of floats depicting an image’s spike-
32     train encoding.
33     :param label: Class label of the input data.

```

```

32 :param axes: Used for re-drawing the input plots.
33 :param ims: Used for re-drawing the input plots.
34 :param figsize: Horizontal, vertical figure size in inches.
35 :return: Tuple of '(axes, ims)' used for re-drawing the input
plots.
36 """
37 local_image = image.detach().clone().cpu().numpy()
38 local_inpy = inpt.detach().clone().cpu().numpy()
39
40 if axes is None:
41     fig, axes = plt.subplots(1, 2, figsize=figsize)
42     ims = (
43         axes[0].imshow(local_image, cmap="binary"),
44         axes[1].imshow(local_inpy, cmap="binary"),
45     )
46
47     if label is None:
48         axes[0].set_title("Current image")
49     else:
50         axes[0].set_title("Current image (label = %d)" % label)
51
52     axes[1].set_title("Reconstruction")
53
54     for ax in axes:
55         ax.set_xticks(())
56         ax.set_yticks(())
57
58     fig.tight_layout()
59 else:
60     if label is not None:
61         axes[0].set_title("Current image (label = %d)" % label)
62
63     ims[0].set_data(local_image)
64     ims[1].set_data(local_inpy)
65
66     return axes, ims
67
68
69 def plot_spikes(
70     spikes: Dict[str, torch.Tensor],
71     time: Optional[Tuple[int, int]] = None,
72     n_neurons: Optional[Dict[str, Tuple[int, int]]] = None,
73     ims: Optional[List[PathCollection]] = None,
74     axes: Optional[Union[Axes, List[Axes]]] = None,
75     figsize: Tuple[float, float] = (8.0, 4.5),
76 ) -> Tuple[List[AxesImage], List[Axes]]:
77     # language=rst
78     """
79     Plot spikes for any group(s) of neurons.
80
81     :param spikes: Mapping from layer names to spiking data. Spike
data has shape
82         '[time, n_1, ..., n_k]', where '[n_1, ..., n_k]' is the
shape of the

```

```

83     recorded layer.
84     :param time: Plot spiking activity of neurons in the given time
85     range. Default is
86     entire simulation time.
87     :param n_neurons: Plot spiking activity of neurons in the given
88     range of neurons.
89     Default is all neurons.
90     :param ims: Used for re-drawing the plots.
91     :param axes: Used for re-drawing the plots.
92     :param figsize: Horizontal, vertical figure size in inches.
93     :return: 'ims, axes': Used for re-drawing the plots.
94     """
95     n_subplots = len(spikes.keys())
96     if n_neurons is None:
97         n_neurons = {}
98
99     spikes = {k: v.view(v.size(0), -1) for (k, v) in spikes.items()}
100     if time is None:
101         # Set it for entire duration
102         for key in spikes.keys():
103             time = (0, spikes[key].shape[0])
104             break
105
106     # Use all neurons if no argument provided.
107     for key, val in spikes.items():
108         if key not in n_neurons.keys():
109             n_neurons[key] = (0, val.shape[1])
110
111     if ims is None:
112         fig, axes = plt.subplots(n_subplots, 1, figsize=figsize)
113         if n_subplots == 1:
114             axes = [axes]
115
116     ims = []
117     for i, datum in enumerate(spikes.items()):
118         spikes = (
119             datum[1][
120                 time[0] : time[1], n_neurons[datum[0]][0] :
121                 n_neurons[datum[0]][1]
122             ]
123             .detach()
124             .clone()
125             .cpu()
126             .numpy()
127         )
128         ims.append(
129             axes[i].scatter(
130                 x=np.array(spikes.nonzero()).T[:, 0],
131                 y=np.array(spikes.nonzero()).T[:, 1],
132                 s=1,
133             )
134         )
135     args = (
136         datum[0],

```

```

134         n_neurons[datum[0]][0],
135         n_neurons[datum[0]][1],
136         time[0],
137         time[1],
138     )
139     axes[i].set_title(
140         "%s spikes for neurons (%d - %d) from t = %d to %d "
141     % args
142     )
143     for ax in axes:
144         ax.set_aspect("auto")
145
146     plt.setp(
147         axes, xticks=[], yticks=[], xlabel="Simulation time",
148         ylabel="Neuron index"
149     )
150     plt.tight_layout()
151     else:
152         for i, datum in enumerate(spikes.items()):
153             spikes = (
154                 datum[1][
155                     time[0] : time[1], n_neurons[datum[0]][0] :
156                     n_neurons[datum[0]][1]
157                 ]
158                 .detach()
159                 .clone()
160                 .cpu()
161                 .numpy()
162             )
163             ims[i].set_offsets(np.array(spikes.nonzero()).T)
164             args = (
165                 datum[0],
166                 n_neurons[datum[0]][0],
167                 n_neurons[datum[0]][1],
168                 time[0],
169                 time[1],
170             )
171             axes[i].set_title(
172                 "%s spikes for neurons (%d - %d) from t = %d to %d "
173             % args
174             )
175
176     plt.draw()
177
178     return ims, axes
179
180 def plot_weights(
181     weights: torch.Tensor,
182     wmin: Optional[float] = 0,
183     wmax: Optional[float] = 1,
184     im: Optional[AxesImage] = None,
185     figsize: Tuple[int, int] = (5, 5),
186     cmap: str = "hot_r",

```

```

184 ) -> AxesImage:
185     # language=rst
186     """
187     Plot a connection weight matrix.
188
189     :param weights: Weight matrix of ‘‘Connection‘‘ object.
190     :param wmin: Minimum allowed weight value.
191     :param wmax: Maximum allowed weight value.
192     :param im: Used for re-drawing the weights plot.
193     :param figsize: Horizontal, vertical figure size in inches.
194     :param cmap: Matplotlib colormap.
195     :return: ‘‘AxesImage‘‘ for re-drawing the weights plot.
196     """
197     local_weights = weights.detach().clone().cpu().numpy()
198     if not im:
199         fig, ax = plt.subplots(figsize=figsize)
200
201         im = ax.imshow(local_weights, cmap=cmap, vmin=wmin, vmax=
202 wmax)
203         div = make_axes_locatable(ax)
204         cax = div.append_axes("right", size="5%", pad=0.05)
205
206         ax.set_xticks(())
207         ax.set_yticks(())
208         ax.set_aspect("auto")
209
210         plt.colorbar(im, cax=cax)
211         fig.tight_layout()
212     else:
213         im.set_data(local_weights)
214
215     return im
216
217 def plot_conv2d_weights(
218     weights: torch.Tensor,
219     wmin: float = 0.0,
220     wmax: float = 1.0,
221     im: Optional[AxesImage] = None,
222     figsize: Tuple[int, int] = (5, 5),
223     cmap: str = "hot_r",
224 ) -> AxesImage:
225     # language=rst
226     """
227     Plot a connection weight matrix of a Conv2dConnection.
228
229     :param weights: Weight matrix of Conv2dConnection object.
230     :param wmin: Minimum allowed weight value.
231     :param wmax: Maximum allowed weight value.
232     :param im: Used for re-drawing the weights plot.
233     :param figsize: Horizontal, vertical figure size in inches.
234     :param cmap: Matplotlib colormap.
235     :return: Used for re-drawing the weights plot.
236     """

```



```

237
238 sqrt1 = int(np.ceil(np.sqrt(weights.size(0))))
239 sqrt2 = int(np.ceil(np.sqrt(weights.size(1))))
240 height, width = weights.size(2), weights.size(3)
241 reshaped = reshape_conv2d_weights(weights)
242
243 if not im:
244     fig, ax = plt.subplots(figsize=figsize)
245     im = ax.imshow(reshaped, cmap=cmap, vmin=wmin, vmax=wmax)
246     div = make_axes_locatable(ax)
247     cax = div.append_axes("right", size="5%", pad=0.05)
248
249     for i in range(height, sqrt1 * sqrt2 * height, height):
250         ax.axhline(i - 0.5, color="g", linestyle="--")
251         if i % sqrt1 == 0:
252             ax.axhline(i - 0.5, color="g", linestyle="-")
253
254     for i in range(width, sqrt1 * sqrt2 * width, width):
255         ax.axvline(i - 0.5, color="g", linestyle="--")
256         if i % sqrt1 == 0:
257             ax.axvline(i - 0.5, color="g", linestyle="-")
258
259     ax.set_xticks(())
260     ax.set_yticks(())
261     ax.set_aspect("auto")
262
263     plt.colorbar(im, cax=cax)
264     fig.tight_layout()
265 else:
266     im.set_data(reshaped)
267
268 return im
269
270
271 def plot_locally_connected_weights(
272     weights: torch.Tensor,
273     n_filters: int,
274     kernel_size: Union[int, Tuple[int, int]],
275     conv_size: Union[int, Tuple[int, int]],
276     locations: torch.Tensor,
277     input_sqrt: Union[int, Tuple[int, int]],
278     wmin: float = 0.0,
279     wmax: float = 1.0,
280     im: Optional[AxesImage] = None,
281     lines: bool = True,
282     figsize: Tuple[int, int] = (5, 5),
283     cmap: str = "hot_r",
284 ) -> AxesImage:
285     # language=rst
286     """
287     Plot a connection weight matrix of a :code:`Connection` with `
288     locally connected
289     structure <http://yann.lecun.com/exdb/publis/pdf/gregor-nips-11.
290     pdf>_ .

```

```

289
290 :param weights: Weight matrix of Conv2dConnection object.
291 :param n_filters: No. of convolution kernels in use.
292 :param kernel_size: Side length(s) of 2D convolution kernels.
293 :param conv_size: Side length(s) of 2D convolution population.
294 :param locations: Indices of input receptive fields for
convolution population
295     neurons.
296 :param input_sqrt: Side length(s) of 2D input data.
297 :param wmin: Minimum allowed weight value.
298 :param wmax: Maximum allowed weight value.
299 :param im: Used for re-drawing the weights plot.
300 :param lines: Whether or not to draw horizontal and vertical
lines separating input
301     regions.
302 :param figsize: Horizontal, vertical figure size in inches.
303 :param cmap: Matplotlib colormap.
304 :return: Used for re-drawing the weights plot.
305 """
306 kernel_size = _pair(kernel_size)
307 conv_size = _pair(conv_size)
308 input_sqrt = _pair(input_sqrt)
309
310 reshaped = reshape_locally_connected_weights(
311     weights, n_filters, kernel_size, conv_size, locations,
input_sqrt
312 )
313 n_sqrt = int(np.ceil(np.sqrt(n_filters)))
314
315 if not im:
316     fig, ax = plt.subplots(figsize=figsize)
317
318     im = ax.imshow(reshaped.cpu(), cmap=cmap, vmin=wmin, vmax=
wmax)
319     div = make_axes_locatable(ax)
320     cax = div.append_axes("right", size="5%", pad=0.05)
321
322     if lines:
323         for i in range(
324             n_sqrt * kernel_size[0],
325             n_sqrt * conv_size[0] * kernel_size[0],
326             n_sqrt * kernel_size[0],
327         ):
328             ax.axhline(i - 0.5, color="g", linestyle="--")
329
330         for i in range(
331             n_sqrt * kernel_size[1],
332             n_sqrt * conv_size[1] * kernel_size[1],
333             n_sqrt * kernel_size[1],
334         ):
335             ax.axvline(i - 0.5, color="g", linestyle="--")
336
337     ax.set_xticks(())
338     ax.set_yticks(())

```

```

339         ax.set_aspect("auto")
340
341         plt.colorbar(im, cax=cax)
342         fig.tight_layout()
343     else:
344         im.set_data(reshaped)
345
346     return im
347
348
349 def plot_assignments(
350     assignments: torch.Tensor,
351     im: Optional[AxesImage] = None,
352     figsize: Tuple[int, int] = (5, 5),
353     classes: Optional[Sized] = None,
354 ) -> AxesImage:
355     # language=rst
356     """
357     Plot the two-dimensional neuron assignments.
358
359     :param assignments: Vector of neuron label assignments.
360     :param im: Used for re-drawing the assignments plot.
361     :param figsize: Horizontal, vertical figure size in inches.
362     :param classes: Iterable of labels for colorbar ticks
363     corresponding to data labels.
364     :return: Used for re-drawing the assignments plot.
365     """
366     locals_assignments = assignments.detach().clone().cpu().numpy()
367     if not im:
368         fig, ax = plt.subplots(figsize=figsize)
369         ax.set_title("Categorical assignments")
370
371         if classes is None:
372             color = plt.get_cmap("RdBu", 11)
373             im = ax.matshow(locals_assignments, cmap=color, vmin
374                             =-1.5, vmax=9.5)
375         else:
376             color = plt.get_cmap("RdBu", len(classes) + 1)
377             im = ax.matshow(
378                 locals_assignments, cmap=color, vmin=-1.5, vmax=len(
379                     classes) - 0.5
380             )
381
382         div = make_axes_locatable(ax)
383         cax = div.append_axes("right", size="5%", pad=0.05)
384
385         if classes is None:
386             cbar = plt.colorbar(im, cax=cax, ticks=list(range(-1,
387                             11)))
388         else:
389             cbar = plt.colorbar(im, cax=cax, ticks=np.arange(-1, len(
390                 classes)))
391         cbar.ax.set_yticklabels(["none"] + list(classes))

```

```

388         ax.set_xticks(())
389         ax.set_yticks(())
390         fig.tight_layout()
391     else:
392         im.set_data(locals_assignments)
393
394     return im
395
396
397
398 def plot_performance(
399     performances: Dict[str, List[float]],
400     ax: Optional[Axes] = None,
401     figsize: Tuple[int, int] = (7, 4),
402 ) -> Axes:
403     # language=rst
404     """
405     Plot training accuracy curves.
406
407     :param performances: Lists of training accuracy estimates per
408     voting scheme.
409     :param ax: Used for re-drawing the performance plot.
410     :param figsize: Horizontal, vertical figure size in inches.
411     :return: Used for re-drawing the performance plot.
412     """
413     if not ax:
414         _, ax = plt.subplots(figsize=figsize)
415     else:
416         ax.clear()
417
418     for scheme in performances:
419         ax.plot(
420             range(len(performances[scheme])),
421             [p for p in performances[scheme]],
422             label=scheme,
423         )
424
425     ax.set_ylim([0, 100])
426     ax.set_title("Estimated classification accuracy")
427     ax.set_xlabel("No. of examples")
428     ax.set_ylabel("Accuracy")
429     ax.set_xticks(())
430     ax.set_yticks(range(0, 110, 10))
431     ax.legend()
432
433     return ax
434
435 def plot_voltages(
436     voltages: Dict[str, torch.Tensor],
437     ims: Optional[List[AxesImage]] = None,
438     axes: Optional[List[Axes]] = None,
439     time: Tuple[int, int] = None,
440     n_neurons: Optional[Dict[str, Tuple[int, int]]] = None,

```

```

441     cmap: Optional[str] = "jet",
442     plot_type: str = "color",
443     thresholds: Dict[str, torch.Tensor] = None,
444     figsize: Tuple[float, float] = (8.0, 4.5),
445 ) -> Tuple[List[AxesImage], List[Axes]]:
446     # language=rst
447     """
448     Plot voltages for any group(s) of neurons.
449
450     :param voltages: Contains voltage data by neuron layers.
451     :param ims: Used for re-drawing the plots.
452     :param axes: Used for re-drawing the plots.
453     :param time: Plot voltages of neurons in given time range.
454     Default is entire
455         simulation time.
456     :param n_neurons: Plot voltages of neurons in given range of
457     neurons. Default is all
458     neurons.
459     :param cmap: Matplotlib colormap to use.
460     :param figsize: Horizontal, vertical figure size in inches.
461     :param plot_type: The way how to draw graph. 'color' for
462     pcolormesh, 'line' for
463     curved lines.
464     :param thresholds: Thresholds of the neurons in each layer.
465     :return: 'ims, axes': Used for re-drawing the plots.
466     """
467     n_subplots = len(voltages.keys())
468
469     for key in voltages.keys():
470         voltages[key] = voltages[key].view(-1, voltages[key].size
471         (-1))
472
473     if time is None:
474         for key in voltages.keys():
475             time = (0, voltages[key].size(-1))
476             break
477
478     if n_neurons is None:
479         n_neurons = {}
480
481     for key, val in voltages.items():
482         if key not in n_neurons.keys():
483             n_neurons[key] = (0, val.size(0))
484
485     if not ims:
486         fig, axes = plt.subplots(n_subplots, 1, figsize=figsize)
487         ims = []
488         if n_subplots == 1: # Plotting only one image
489             for v in voltages.items():
490                 if plot_type == "line":
491                     ims.append(
492                         axes.plot(
493                             v[1]
494                             .detach()

```

```

491         .clone()
492         .cpu()
493         .numpy()[
494             n_neurons[v[0]][0] : n_neurons[v
[0]][1],
495             time[0] : time[1],
496         ]
497     )
498 )
499
500     if thresholds is not None and thresholds[v[0]].
size() == torch.Size(
501         []
502     ):
503         ims.append(
504             axes.axhline(
505                 y=thresholds[v[0]].item(), c="r",
linestyle="--"
506             )
507         )
508     else:
509         ims.append(
510             axes.pcolormesh(
511                 v[1]
512                 .cpu()
513                 .numpy()[
514             n_neurons[v[0]][0] : n_neurons[v
[0]][1],
515                 time[0] : time[1],
516             ]
517             .T,
518             cmap=cmap,
519         )
520     )
521
522     args = (v[0], n_neurons[v[0]][0], n_neurons[v
[0]][1], time[0], time[1])
523     plt.title("%s voltages for neurons (%d - %d) from t
= %d to %d " % args)
524     plt.xlabel("Time (ms)")
525
526     if plot_type == "line":
527         plt.ylabel("Voltage")
528     else:
529         plt.ylabel("Neuron index")
530
531     axes.set_aspect("auto")
532
533     else: # Plot each layer at a time
534         for i, v in enumerate(voltages.items()):
535             if plot_type == "line":
536                 ims.append(
537                     axes[i].plot(
538                         v[1]

```

```

539         .cpu()
540         .numpy()[
541             n_neurons[v[0]][0] : n_neurons[v
542 [0]][1],
543             time[0] : time[1],
544         ]
545     )
546     if thresholds is not None and thresholds[v[0]].
size() == torch.Size(
547         []
548     ):
549         ims.append(
550             axes[i].axhline(
551                 y=thresholds[v[0]].item(), c="r",
linestyle="--"
552             )
553         )
554     else:
555         ims.append(
556             axes[i].matshow(
557                 v[1]
558                 .cpu()
559                 .numpy()[
560 [0]][1],
561                 time[0] : time[1],
562                 ]
563                 .T,
564                 cmap=cmap,
565             )
566         )
567         args = (v[0], n_neurons[v[0]][0], n_neurons[v
568 [0]][1], time[0], time[1])
569         axes[i].set_title(
570             "%s voltages for neurons (%d - %d) from t = %d
to %d " % args
571         )
572         for ax in axes:
573             ax.set_aspect("auto")
574
575         if plot_type == "color":
576             plt.setp(axes, xlabel="Simulation time", ylabel="Neuron
index")
577         elif plot_type == "line":
578             plt.setp(axes, xlabel="Simulation time", ylabel="Voltage
")
579
580         plt.tight_layout()
581
582     else:
583         # Plotting figure given
584         if n_subplots == 1: # Plotting only one image

```

```

585         for v in voltages.items():
586             axes.clear()
587             if plot_type == "line":
588                 axes.plot(
589                     v[1]
590                     .cpu()
591                     .numpy()[
592                         n_neurons[v[0]][0] : n_neurons[v[0]][1],
593                     time[0] : time[1]
594                 ]
595             )
596             if thresholds is not None and thresholds[v[0]].
size() == torch.Size(
597                 []
598             ):
599                 axes.axhline(y=thresholds[v[0]].item(), c="r
", linestyle="--")
600             else:
601                 axes.matshow(
602                     v[1]
603                     .cpu()
604                     .numpy()[
605                         n_neurons[v[0]][0] : n_neurons[v[0]][1],
606                     time[0] : time[1]
607                 ]
608                 .T,
609                 cmap=cmap,
610             )
611             args = (v[0], n_neurons[v[0]][0], n_neurons[v
[0]][1], time[0], time[1])
612             axes.set_title(
613                 "%s voltages for neurons (%d - %d) from t = %d
to %d " % args
614             )
615             axes.set_aspect("auto")
616         else:
617             # Plot each layer at a time
618             for i, v in enumerate(voltages.items()):
619                 axes[i].clear()
620                 if plot_type == "line":
621                     axes[i].plot(
622                         v[1]
623                         .cpu()
624                         .numpy()[
625                             n_neurons[v[0]][0] : n_neurons[v[0]][1],
626                         time[0] : time[1]
627                     ]
628                 )
629                 if thresholds is not None and thresholds[v[0]].
size() == torch.Size(
630                     []
631                 ):
632                     axes[i].axhline(

```



```

631         y=thresholds[v[0]].item(), c="r",
linestyle="--"
632     )
633     else:
634         axes[i].matshow(
635             v[1]
636             .cpu()
637             .numpy()[
638                 n_neurons[v[0]][0] : n_neurons[v[0]][1],
time[0] : time[1]
639             ]
640             .T,
641             cmap=cmap,
642         )
643         args = (v[0], n_neurons[v[0]][0], n_neurons[v
[0]][1], time[0], time[1])
644         axes[i].set_title(
645             "%s voltages for neurons (%d - %d) from t = %d
to %d " % args
646         )
647
648         for ax in axes:
649             ax.set_aspect("auto")
650
651         if plot_type == "color":
652             plt.setp(axes, xlabel="Simulation time", ylabel="Neuron
index")
653         elif plot_type == "line":
654             plt.setp(axes, xlabel="Simulation time", ylabel="Voltage
")
655
656         plt.tight_layout()
657
658     return ims, axes

```

Listing B.30: Plotting

```

1 import torch
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib.animation as animation
5
6 from typing import List, Tuple, Optional
7
8
9 def plot_weights_movie(ws: np.ndarray, sample_every: int = 1) ->
None:
10     # language=rst
11     """
12     Create and plot movie of weights.
13
14     :param ws: Array of shape "[n_examples, source, target, time
]"'.
15     :param sample_every: Sub-sample using this parameter.

```

```

16 """
17 weights = []
18
19 # Obtain samples from the weights for every example.
20 for i in range(ws.shape[0]):
21     sub_sampled_weight = ws[i, :, :, range(0, ws[i].shape[2],
22 sample_every)]
23     weights.append(sub_sampled_weight)
24 else:
25     weights = np.concatenate(weights, axis=0)
26
27 # Initialize plot.
28 fig = plt.figure()
29 im = plt.imshow(weights[0, :, :], cmap="hot_r", animated=True,
30 vmin=0, vmax=1)
31 plt.axis("off")
32 plt.colorbar(im)
33
34 # Update function for the animation.
35 def update(j):
36     im.set_data(weights[j, :, :])
37     return [im]
38
39 # Initialize animation.
40 global ani
41 ani = 0
42 ani = animation.FuncAnimation(
43     fig, update, frames=weights.shape[-1], interval=1000, blit=
44 True
45 )
46 plt.show()
47
48 def plot_spike_trains_for_example(
49     spikes: torch.Tensor,
50     n_ex: Optional[int] = None,
51     top_k: Optional[int] = None,
52     indices: Optional[List[int]] = None,
53 ) -> None:
54     # language=rst
55     """
56     Plot spike trains for top-k neurons or for specific indices.
57
58     :param spikes: Spikes for one simulation run of shape
59         '(n_examples, n_neurons, time)'.
60     :param n_ex: Allows user to pick which example to plot spikes
61         for.
62     :param top_k: Plot k neurons that spiked the most for n_ex
63         example.
64     :param indices: Plot specific neurons' spiking activity instead
65         of top_k.
66     """
67     assert n_ex is not None and 0 <= n_ex < spikes.shape[0]

```

```

64 plt.figure()
65
66 if top_k is None and indices is None: # Plot all neurons'
67     spiking activity
68     spike_per_neuron = [np.argwhere(i == 1).flatten() for i in
69     spikes[n_ex, :, :]]
70     plt.title("Spiking activity for all %d neurons" % spikes.
71     shape[1])
72
73 elif top_k is None: # Plot based on indices parameter
74     assert indices is not None
75     spike_per_neuron = [
76     np.argwhere(i == 1).flatten() for i in spikes[n_ex,
77     indices, :]]
78
79 elif indices is None: # Plot based on top_k parameter
80     assert top_k is not None
81     # Obtain the top k neurons that fired the most
82     top_k_loc = np.argsort(np.sum(spikes[n_ex, :, :], axis=1),
83     axis=0)[::-1]
84     spike_per_neuron = [
85     np.argwhere(i == 1).flatten() for i in spikes[n_ex,
86     top_k_loc[0:top_k], :]]
87     plt.title("Spiking activity for top %d neurons" % top_k)
88
89 else:
90     raise ValueError('One of "top_k" or "indices" or both must
91     be None')
92
93 plt.eventplot(spike_per_neuron, linelengths=[0.5] * len(
94     spike_per_neuron))
95 plt.xlabel("Simulation Time")
96 plt.ylabel("Neuron index")
97 plt.show()
98
99
100 def plot_voltage(
101     voltage: torch.Tensor,
102     n_ex: int = 0,
103     n_neuron: int = 0,
104     time: Optional[Tuple[int, int]] = None,
105     threshold: float = None,
106 ) -> None:
107     # language=rst
108     """
109     Plot voltage for a single neuron on a specific example.
110
111     :param voltage: Tensor or array of shape '[n_examples,
112     n_neurons, time]'.
113     :param n_ex: Allows user to pick which example to plot voltage
114     for.
115     :param n_neuron: Neuron index for which to plot voltages for.

```

```

108     :param time: Plot spiking activity of neurons between the given
109     range of time.
110     :param threshold: Neuron spiking threshold.
111     """
112     assert n_ex >= 0 and n_neuron >= 0
113     assert n_ex < voltage.shape[0] and n_neuron < voltage.shape[1]
114
115     if time is None:
116         time = (0, voltage.shape[-1])
117     else:
118         assert time[0] < time[1]
119         assert time[1] <= voltage.shape[-1]
120
121     timer = np.arange(time[0], time[1])
122     time_ticks = np.arange(time[0], time[1] + 1, 10)
123
124     plt.figure()
125     plt.plot(voltage[n_ex, n_neuron, timer])
126     plt.xlabel("Simulation Time")
127     plt.ylabel("Voltage")
128     plt.title("Membrane voltage of neuron %d for example %d" % (
129         n_neuron, n_ex + 1))
130     locs, labels = plt.xticks()
131     locs = range(int(locs[1]), int(locs[-1]), 10)
132     plt.xticks(locs, time_ticks)
133
134     # Draw threshold line only if given
135     if threshold is not None:
136         plt.axhline(threshold, linestyle="--", color="black", zorder
137                     =0)
138
139     plt.show()

```

Listing B.31: Visualization