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

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.

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

List of Figures

List of Algorithms

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

P_{CC} 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
$\mathbf{I}$	Real part of the received symbol
$\mathbf{Q}$	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	ith sample of random variable X
$\overline{X}$	Mean of random variable X
$\overline{m}_i$	ith partition's mean of X
L	Number of a random variable's samples
M_n^i	nth 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
$\sigma(.)$	Sigmoid function
T	Time steps in LSTM layer
$\mathbf{W}_\alpha$	Shared time-distributed NN weight matrix
ζ_α	Shared time-distributed NN bias matrix
α_t	tth attention weight
y_t	tth 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
$\mathbf{Z(.)}$	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].

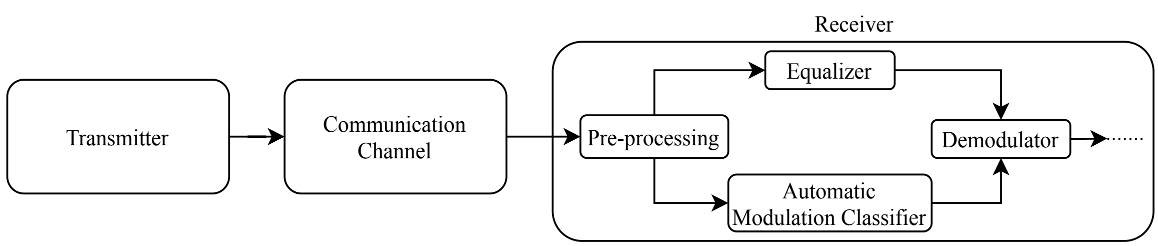


Figure 1.1: *Overview of an AMC operation in a communication system.*

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.

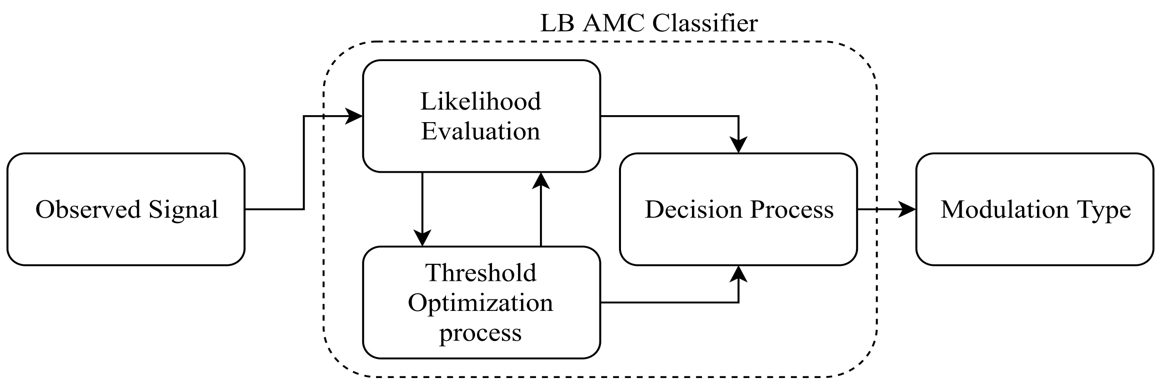


Figure 1.2: *LB approach operation structure.*

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 deterministics. 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.

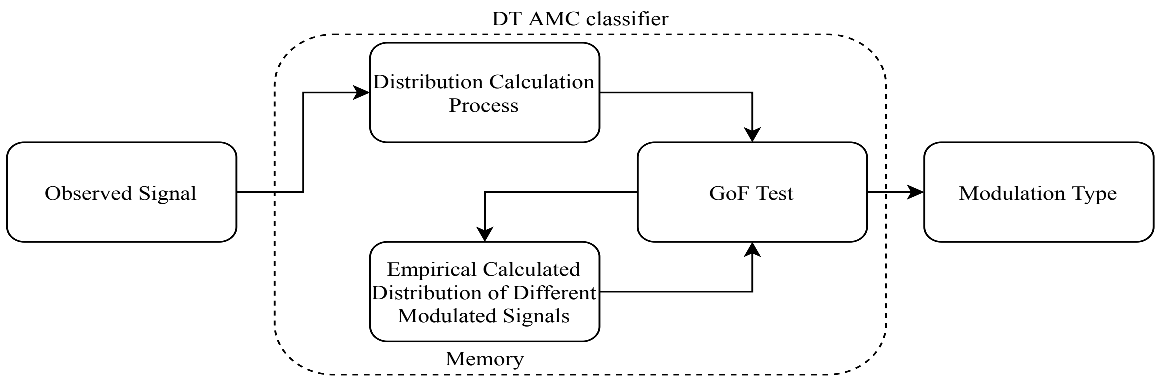


Figure 1.3: *DT approach operation structure.*

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.

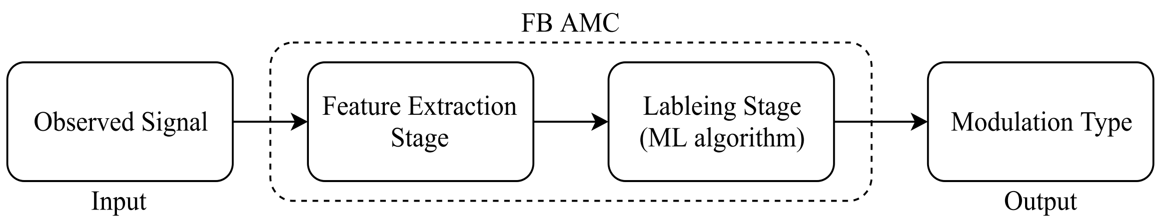


Figure 1.4: *FB AMC overview structure.*

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.

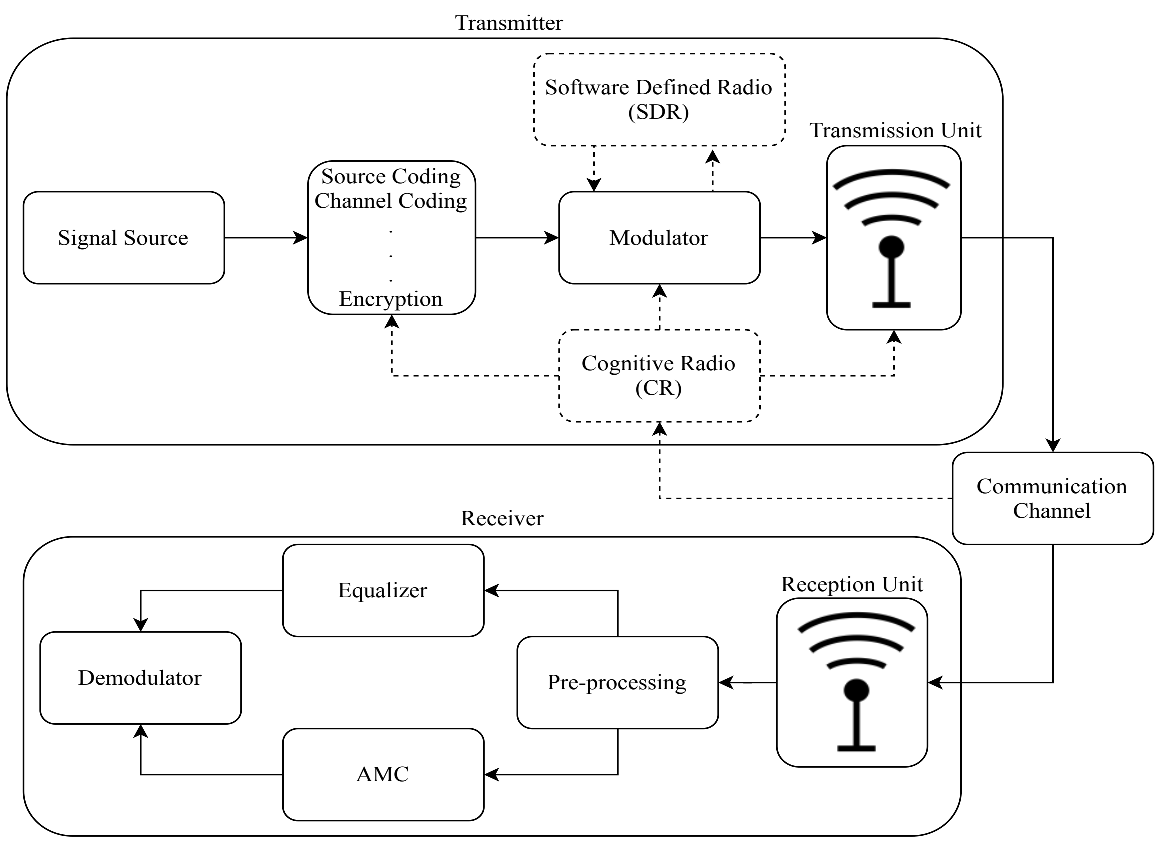


Figure 1.5: *Required components in transmitter to necessitate inclusion of AMC in the receiver.*

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 appraoch. 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 deigning 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.

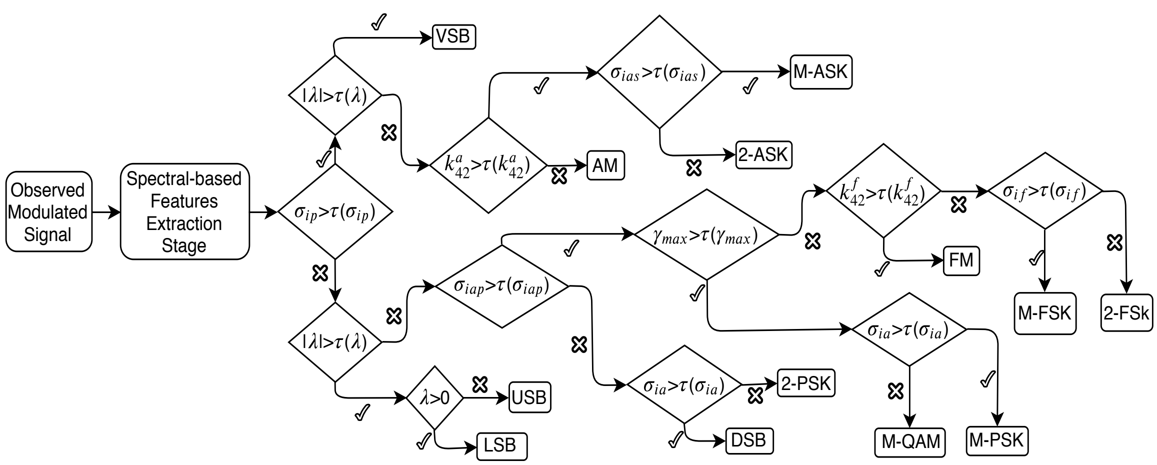


Figure 2.1: *Spectral-based features tree classification procedure.*

- **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 form 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 *Wannes* 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 user 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-Nyquit 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-Nyquit 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.

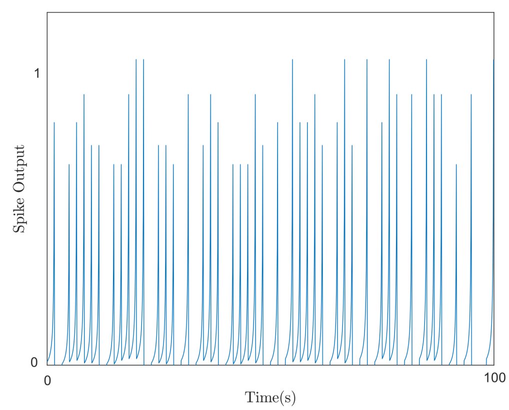
4.1.2.2 Computational Complexity

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.

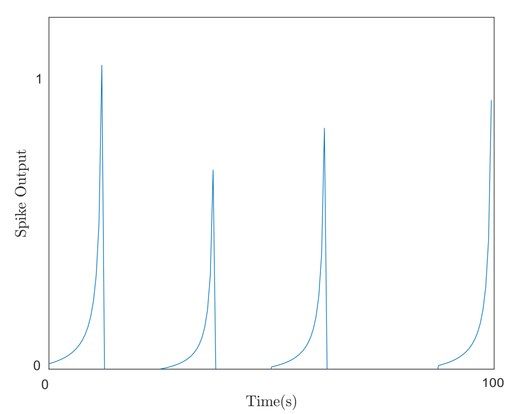
1. All neurons forming a layer do not participate in classification operation. This is a direct consequence of SNN's property where not all neurons fire at each propagation cycle, but rather fire only when a membrane potential – an intrinsic quality of the neuron related to its membrane electrical charge – 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 7^{th}neuron in third layer.



(b) Output spikes of 12^{th}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.

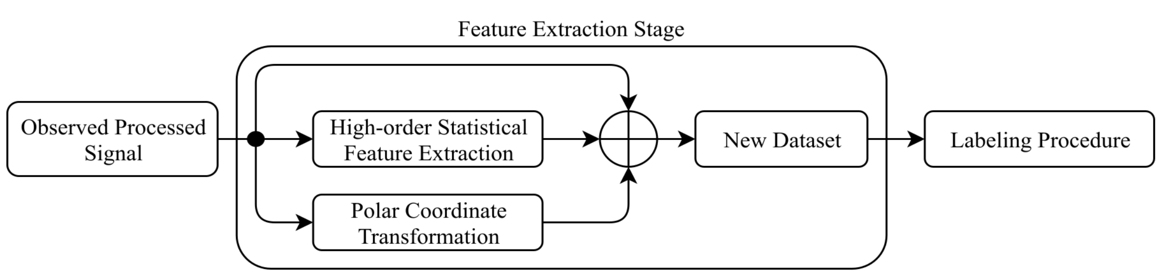


Figure 4.2: *Architecture of the new feature extraction stage.*

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 $\overline{m} = 1$, 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:

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

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

$$M_n = \frac{1}{L}\sum_{i=1}^{L}(x_i - \overline{m}), \tag{4.2}$$

where

$$\overline{m} = \frac{1}{L}\sum_{i=1}^{L} x_i \tag{4.3}$$

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 (4.6) can provide cumulants of nth 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 $(x_i - \overline{m})^n$ to explicit powers of x_i and $\overline{m}$ in binomial theorem, Equation (4.2) can thus be rewritten as:

$$M_n = \sum_{k=0}^{n} \binom{n}{k} (\frac{1}{L} \sum_{i=1}^{L} x_i^{n-k})(-\overline{m})^k \tag{4.4}$$

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, $\overline{m}_{\mathcal{A}}$ and $\overline{m}_{\mathcal{B}}$ 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 $\overline{m}_{\mathcal{B}} - \overline{m}_{\mathcal{A}}$, where each term stands for the mean of their corresponding signal's portion.

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

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:

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

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).

$$\begin{aligned} \mathcal{C}_4(X) = \frac{L^2}{L^3 - 6L^2 + 13L - 12} [(L+1)\overline{X^4} - 4(L+1)\overline{X^3}\,\overline{X} \\ - 3(L-1)\overline{X^2}\,\overline{X^2} + 12L(\overline{X^2}\,\overline{X}\,\overline{X}) - 6L\overline{X}^4] \end{aligned} \tag{4.7}$$

The derivation process is presented below.

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

$$\mathcal{C}_n(X_1, X_2, \cdots, X_n) = \frac{\partial^n}{\partial k_1 \cdots \partial k_n} K_X(k) \tag{4.8}$$

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

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

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

$$
\begin{aligned}
\mathcal{C}_4(X_1, X_2, X_3, X_4) = {} & E[X_1X_2X_3X_4] - E[X_1X_2X_3]E[X_4] - E[X_1X_2X_4]E[X_4] - E[X_1X_3X_4]E[X_2] \\
& - E[X_2X_3X_4]E[X_1] - E[X_1X_2]E[X_3X_4] - E[X_1X_3]E[X_2X_4] - E[X_1X_4]E[X_2X_3] \\
& + 2\{E[X_1X_2]E[X_3]E[X_4] + E[X_1X_3]E[X_2]E[X_4] + E[X_1X_4]E[X_2]E[X_3] \\
& + E[X_2X_3]E[X_1]E[X_4] + E[X_2X_4]E[X_1]E[X_3] + E[X_3X_4]E[X_1]E[X_2]\} \\
& - 6E[X_1]E[X_2]E[X_3]E[X_4] \qquad (4.10)
\end{aligned}
$$

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 $E[\overline{X_1\ X_2\ X_4}\ \overline{X_3}]$, $E[\overline{X_1\ X_3\ X_4}\ \overline{X_2}]$, $E[\overline{X_2\ X_3\ X_4}\ \overline{X_1}]$ and $E[\overline{X_1\ X_2}\ \overline{X_3\ X_4}]$ can also be calculated through equation (4.11).

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

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

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

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

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

that results in:

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

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

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

which can be explicitly stated as:

$$
\begin{aligned}
&L^4 E[\overline{X_1}\,\overline{X_2}\,\overline{X_3}\,\overline{X_4}] = \\
&\quad L(L-1)(L-2)(L-3)E[X_1]E[X_2]E[X_3]E[X_4] \\
&\quad +L(L-1)(L-2)\{E[X_1X_2]E[X_3]E[X_4] + 5\ o.p.\} \\
&\quad +L(L-1)\{E[X_1X_2X_3]E[X_4] + 3\ o.p.\} \\
&\quad +L(L-1)\{E[X_1X_2]E[X_3X_4] + 2\ o.p.\} + LE[X_1X_2X_3X_4]
\end{aligned} \tag{4.16}
$$

where 'o.p.' means other permutations of the variables in e.g. $E[\overline{X_1X_2}]\ E[\overline{X_3}]\ E[\overline{X_4}]$ that give rise to (non-identical) terms like $E[\overline{X_1X_3}]\ E[\overline{X_2}]\ E[\overline{X_4}]$. Then, the $\mathcal{C}_4(X_1, X_2, X_3, X_4)$ can be derived for equation (4.10) as:

$$
\begin{aligned}
\mathcal{C}_4(X_1, X_2, X_3, X_4) = &\frac{L^2}{L^3 - 6L^2 + 13L - 12} \times \\
&\{(L+1)\overline{X_1\, X_2\, X_3\, X_4} - (L+1)(\overline{X_1\, X_2\, X_3}\,\overline{X_4} + 3\ o.p.) - \\
&(L-1)(\overline{X_1\, X_2}\,\overline{X_3\, X_4} + 2\ o.p.) + 2L(\overline{X_1\, X_2}\,\overline{X_3}\,\overline{X_4} + 5\ o.p.) \\
&\qquad - 6L\overline{X_1}\,\overline{X_2}\,\overline{X_3}\,\overline{X_4}\ \}
\end{aligned} \tag{4.17}
$$

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-θ values as $\mathbf{r} = \sqrt{\mathbf{I}^2 + \mathbf{Q}^2}$ and θ = arctan(**Q**/**I**) where **I** and **Q** indicate the real and imaginary parts of the received complex symbols, and **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.

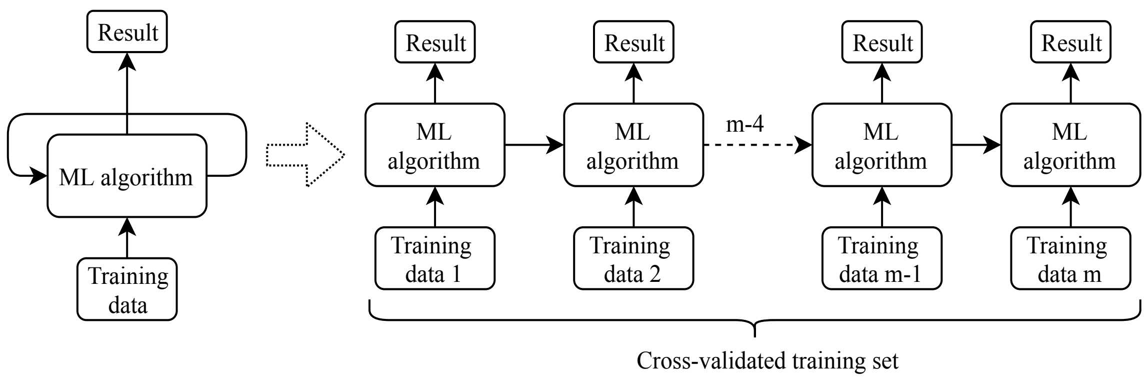


Figure 4.3: *RNN training process over m portion of cross-validated training set.*

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.

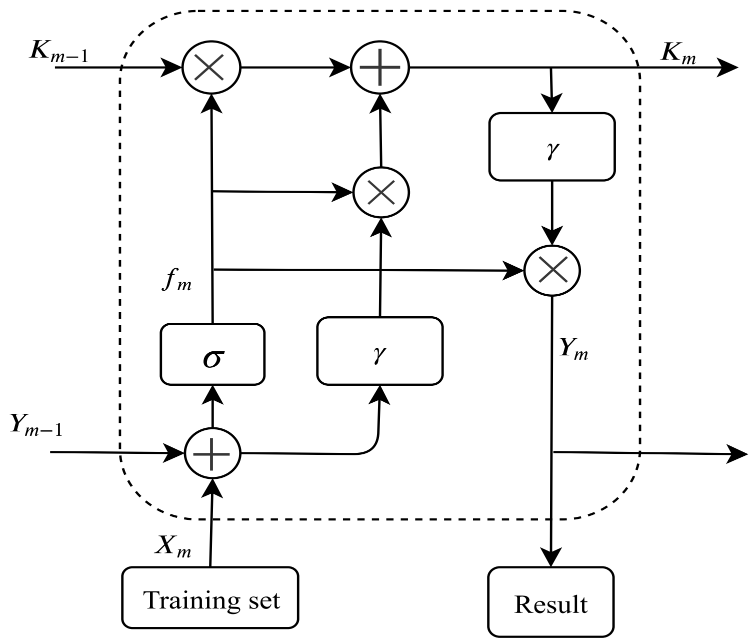


Figure 4.4: *LSTM module in each RNN architecture training iteration.*

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:

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

$$Y_m = f_m \cdot \gamma(K_m) \tag{4.19}$$

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 $\mathbf{W}_\alpha$ 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).

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

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:

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

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.

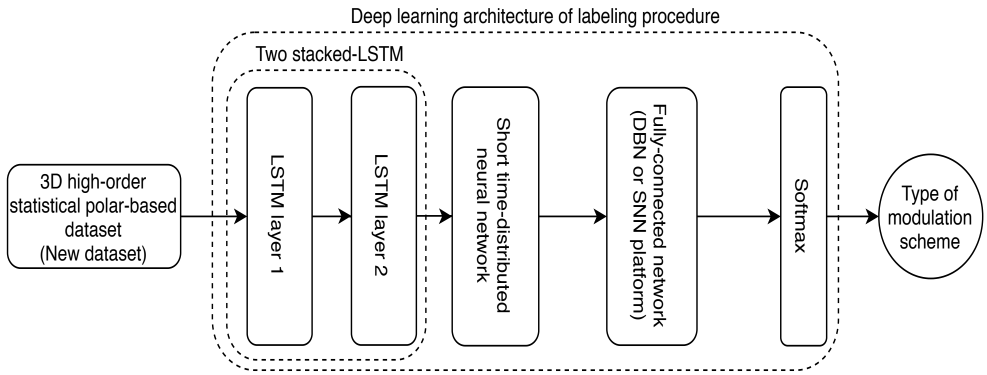


Figure 4.5: *Deep Learning architecture of the labeling stage.*

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:

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

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

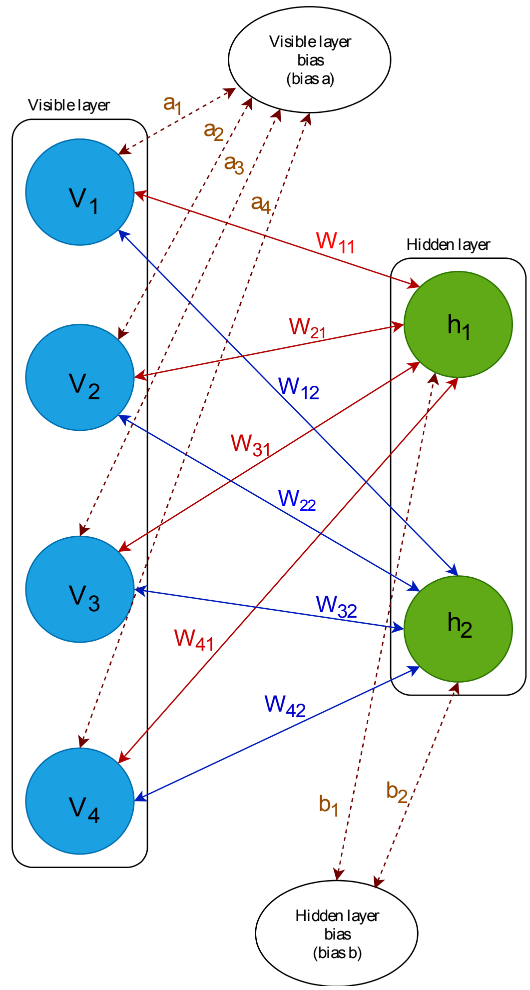


Figure 4.6: *An example RBM structure with 4 visible and 2 hidden units in their corresponding layers in which the effect of biases on visible and hidden layers units can be observed.*

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]:

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

Where $\mathbf{Z}$ is called partition function.

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

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.

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

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

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:

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

And analogously,

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

where $\sigma(.)$ 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]:

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

where $\mathbf{h} = \{h_1, h_2, \ldots, h_l\}$ 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. nth 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:

$$\mathcal{M}_t = \beta_1 \mathcal{M}_{t-1} + (1-\beta_1) g_t \tag{4.31}$$

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

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 V are estimates of first and second moments, the following property should be held for all iterations:

$$E[M_t] = E[g_t] \tag{4.33}$$

$$E[\mathcal{V}_t] = E[g_t^2] \tag{4.34}$$

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.

$$M_0 = 0 \tag{4.35}$$

$$M_1 = \beta_1\, M_0 + (1-\beta_1) \tag{4.36}$$

$$M_2 = \beta_1\, M_1 + (1 -_1) g_2 \tag{4.37}$$

$$M_3 = \beta_1\, M_2 + (1-\beta_1) g_3 \tag{4.38}$$

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:

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

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

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

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

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

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:

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

$$\hat{\mathcal{V}}_t = \frac{\mathcal{V}_t}{1-\beta_2^t} \tag{4.44}$$

$$\tag{4.45}$$

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:

$$W_t = W_{t-1} - \eta \frac{\hat{M}_t}{\sqrt{\hat{\mathcal{V}}_t} + \varepsilon} \tag{4.46}$$

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
  $t \leftarrow 0$
  $M_{t=0} \leftarrow 0$
  $\mathcal{V}_{t=0} \leftarrow 0$
  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)
    $M_t \leftarrow \beta_1 M_{t-1} + (1-\beta_1)g_t$
    $\mathcal{V}_t \leftarrow \beta_2 \mathcal{V}_{t-1} + (1-\beta_2)g_t^2$
    $\hat{M}_t \leftarrow \frac{M_t}{(1-\beta_1^t)}$
    $\hat{\mathcal{V}}_t \leftarrow \frac{\mathbf{V}_t}{(1-\beta_2^t)}$
    $W_t \leftarrow W_{t-1} - \eta \frac{\hat{M}_t}{\sqrt{\hat{\mathcal{V}}_t}+\varepsilon}$
  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 rst 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 re 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:

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

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:

$$\Delta w_i = \lambda \rho_i y \tag{4.48}$$

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:

$$y = g(\sum_i w_i \rho_i - b) \tag{4.49}$$

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:

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

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

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 $F : [0,1]^n \to [0,1]^k$ 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 $CD_n : \{0,1\}^{2n} \to \{0,1\}$

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

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:

$$\frac{d}{dt}w_{ji}(t) = a_0 + a_1 S_i(t) + a_2 S_j(t) + a_3 S_i(t)\overline{S_j}(t) + a_4 \overline{S_i}(t) S_j(t) \tag{4.53}$$

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 ai 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.

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

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.

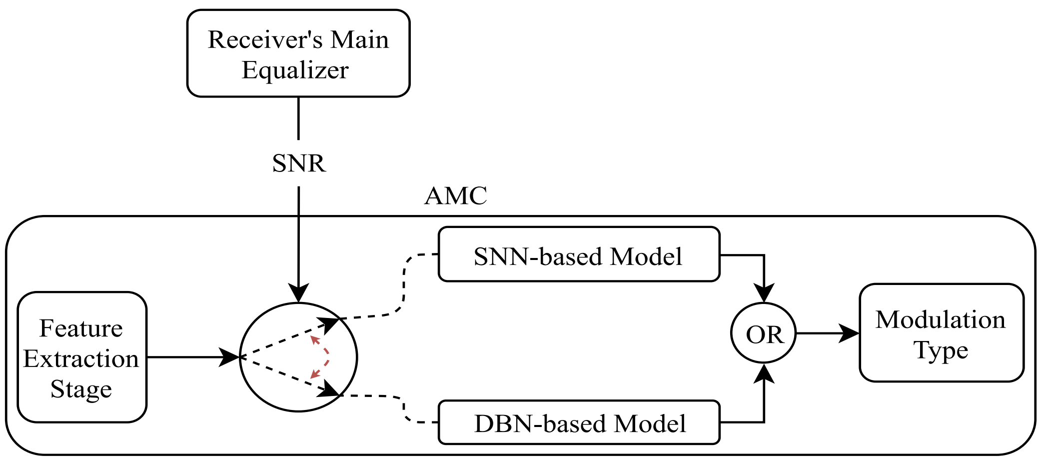


Figure 4.7: *The proposed novel framework's principal working.*

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.

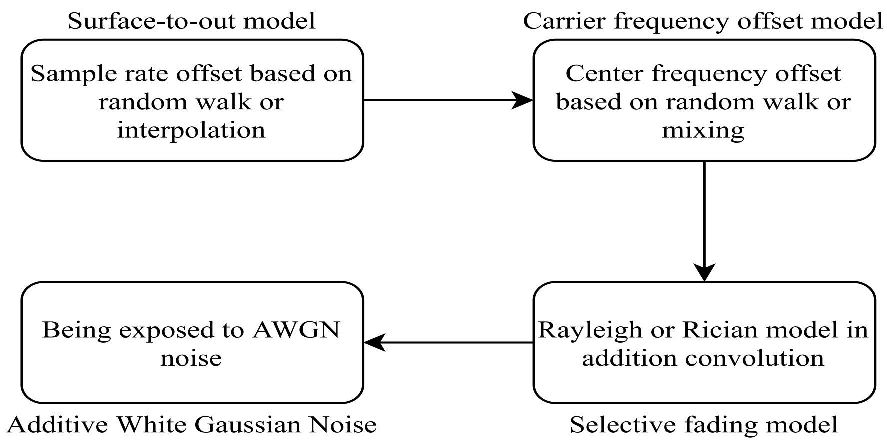
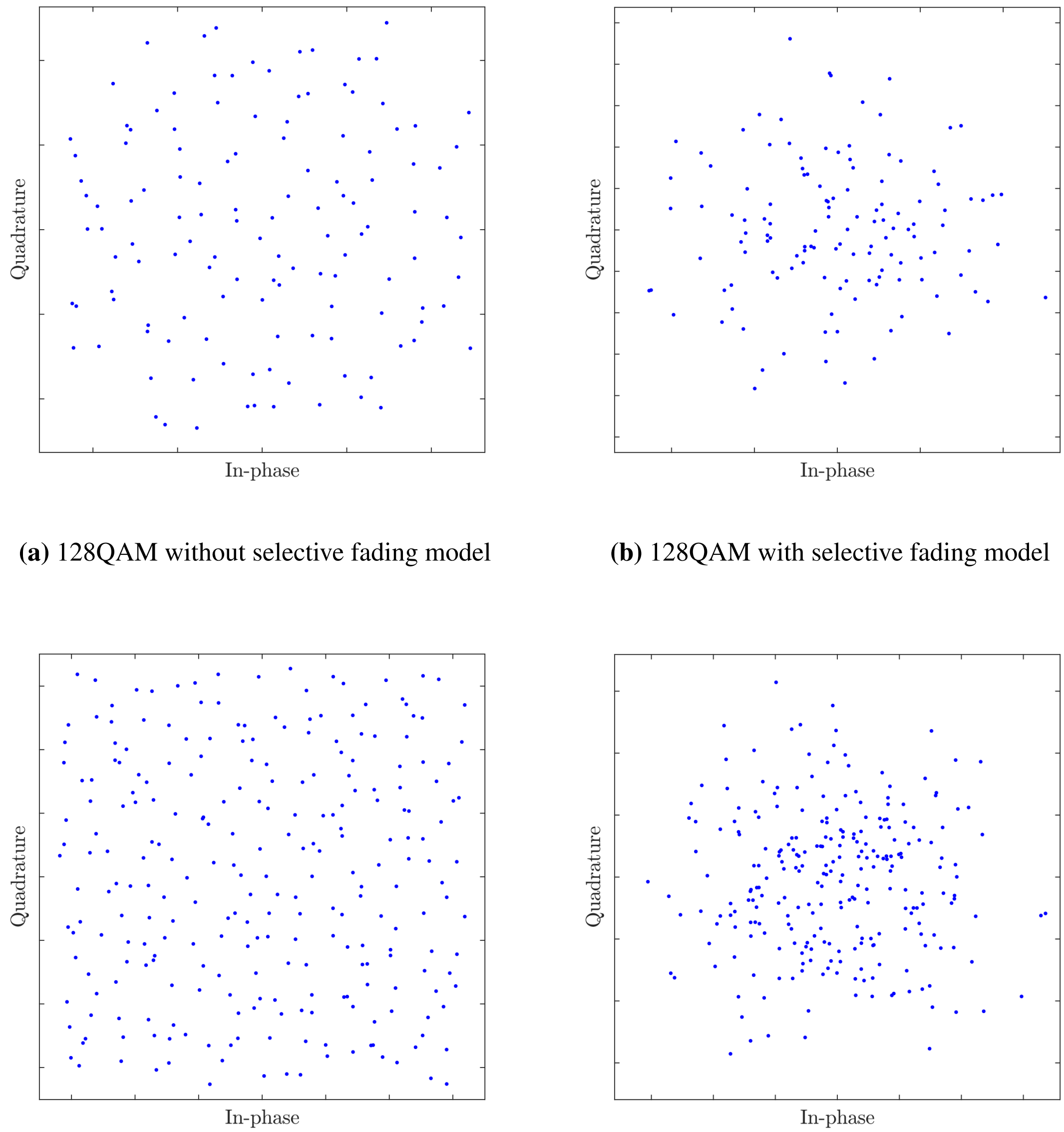


Figure 5.1: *Process of applying channel effects to the transmitted signal*

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

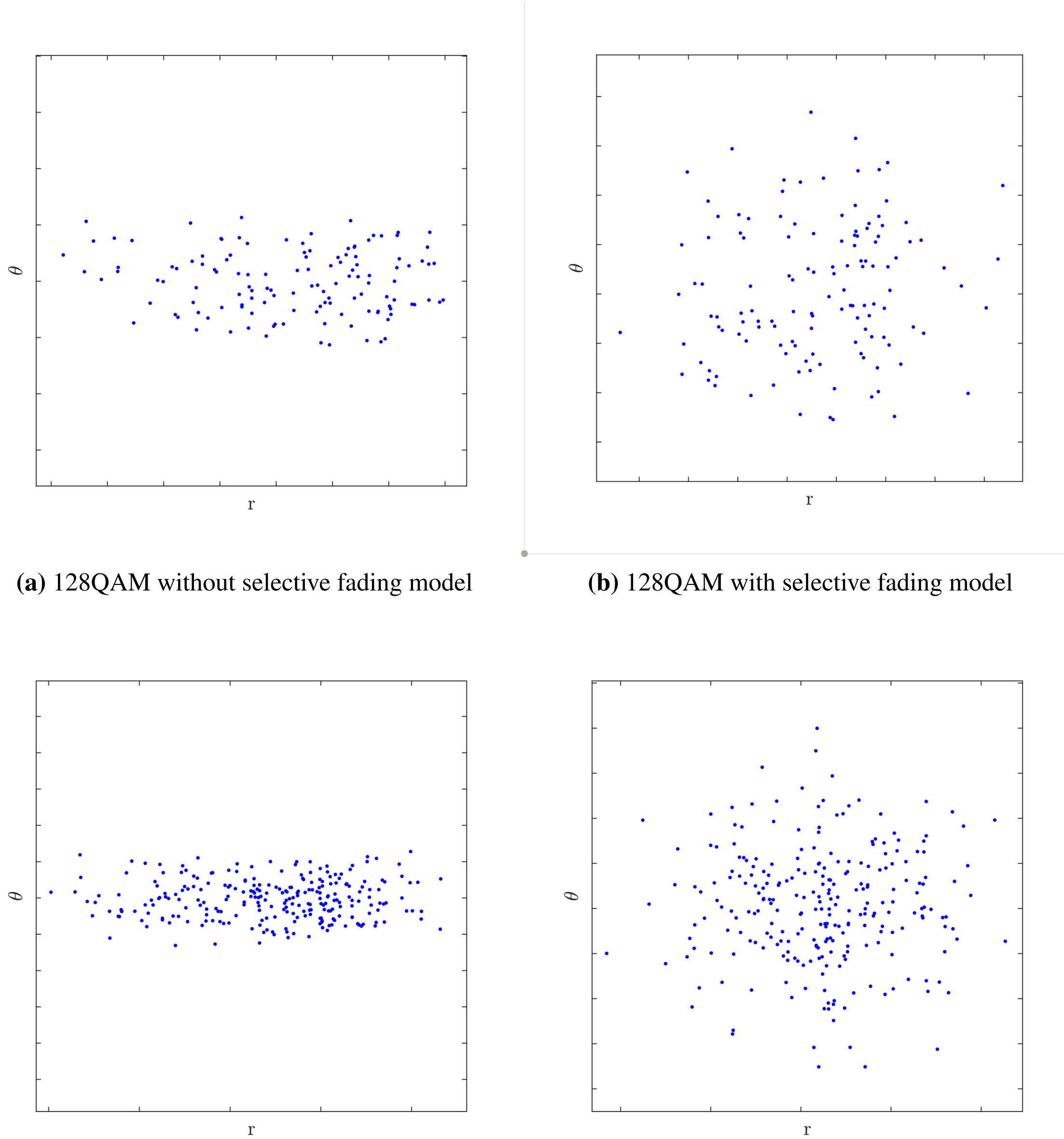


(a) 128QAM without selective fading model **(b)** 128QAM with selective fading model

(c) 256QAM without selective fading model **(d)** 256QAM with selective fading model

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.



(a) 128QAM without selective fading model **(b)** 128QAM with selective fading model

(c) 256QAM without selective fading model **(d)** 256QAM with selective fading model

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

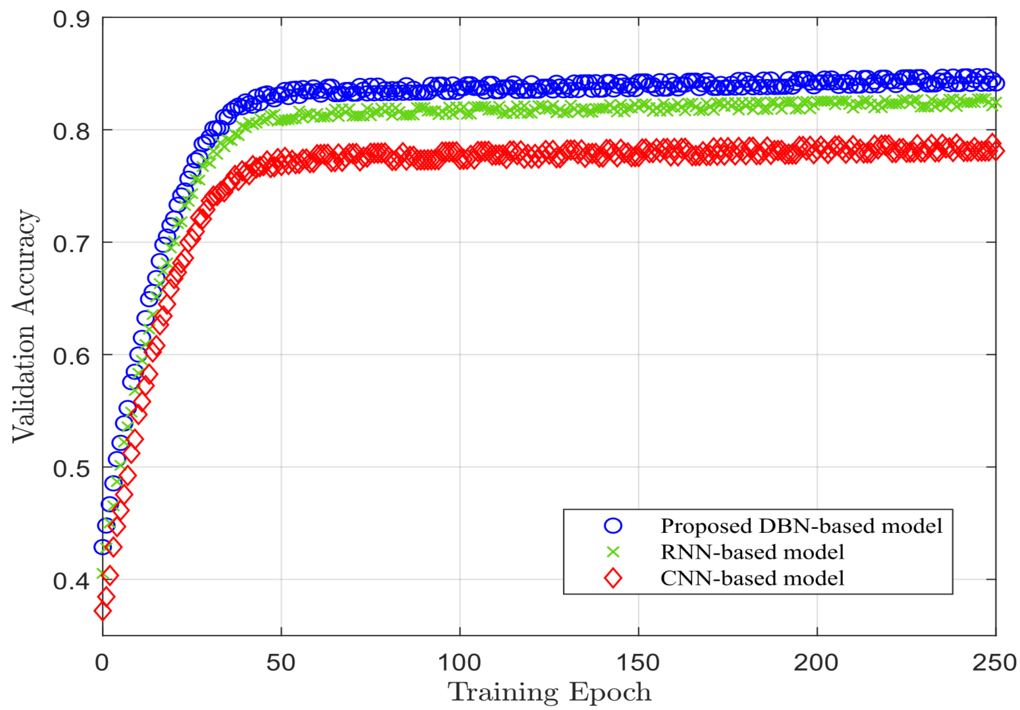


Figure 5.4: *Validation accuracy of DBN-based model in training stage.*

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.

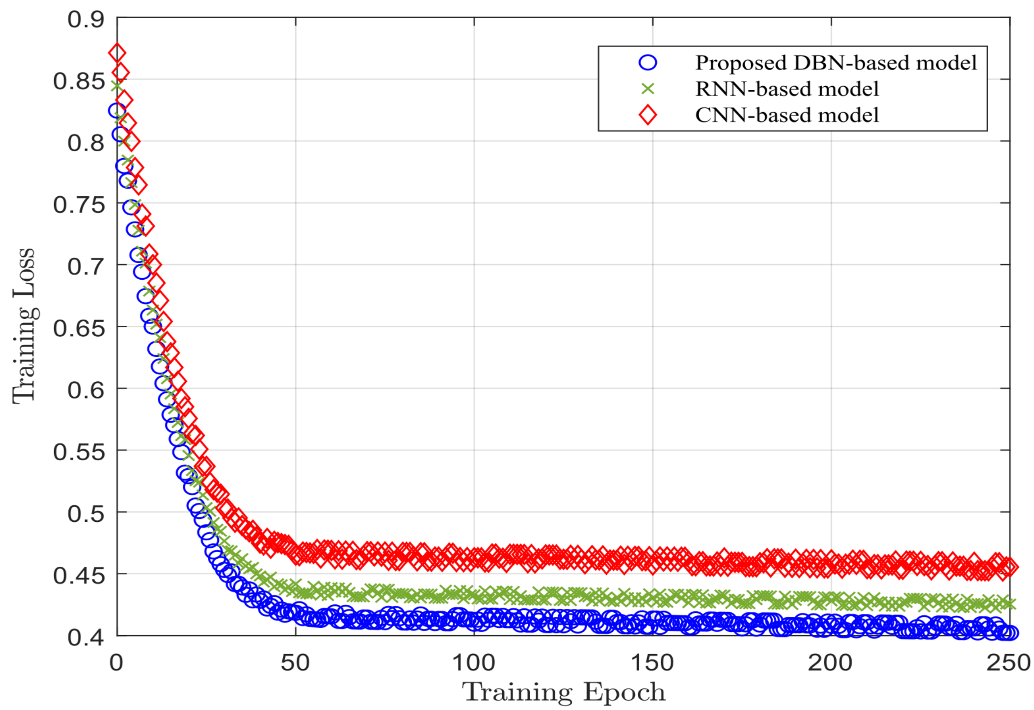


Figure 5.5: *Training loss of DBN-based model in training stage.*

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.

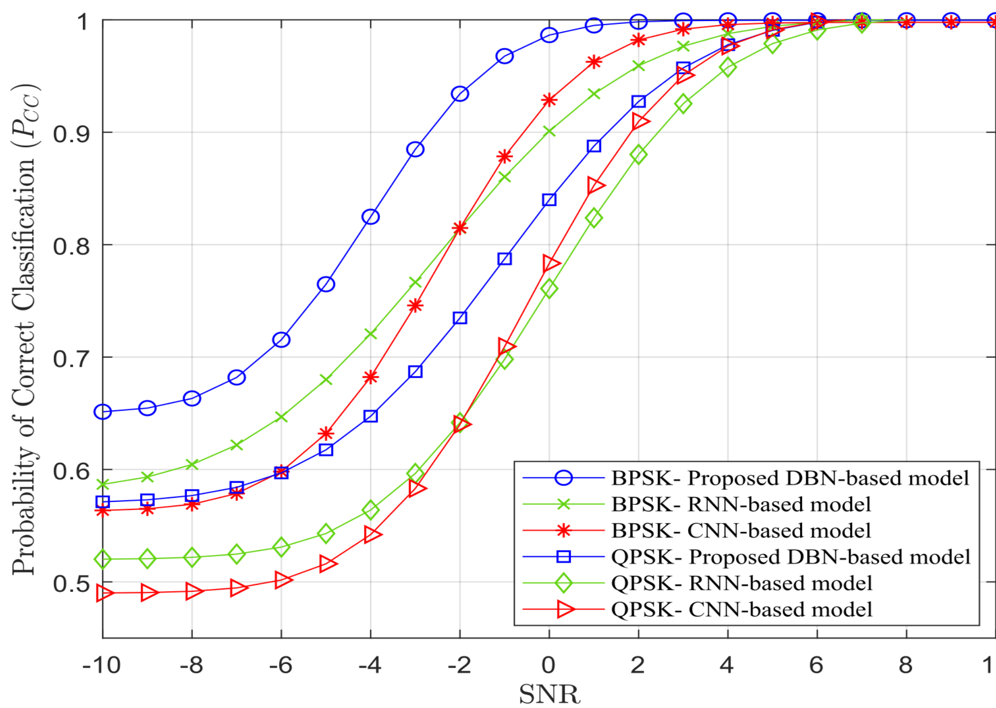


Figure 5.6: *Proposed DBN-based lower-bound performance compared to RNN and CNN*

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.

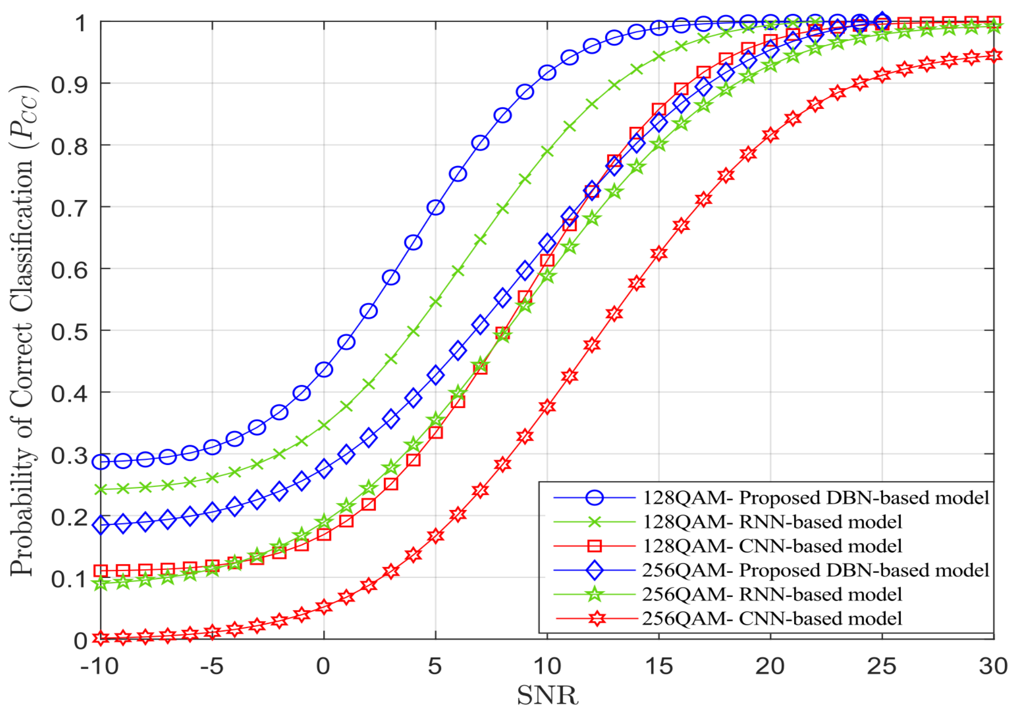


Figure 5.7: *Proposed DBN-based upper-bound performance compared to RNN and CNN*

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

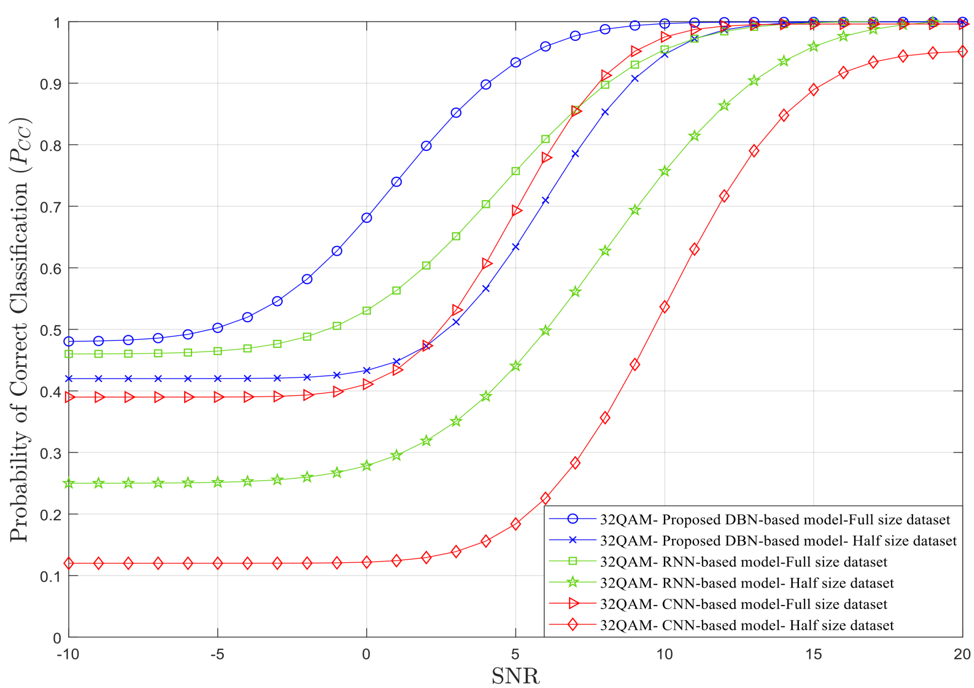


Figure 5.8: *Number of training sample impact on classification accuracy.*

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	0.0022τ	0.0021τ	0.0020τ
Each epoch	22.4716τ	21.826τ	20.6637τ
Total classification	5056.11τ	4910.86τ	4649.35τ

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].

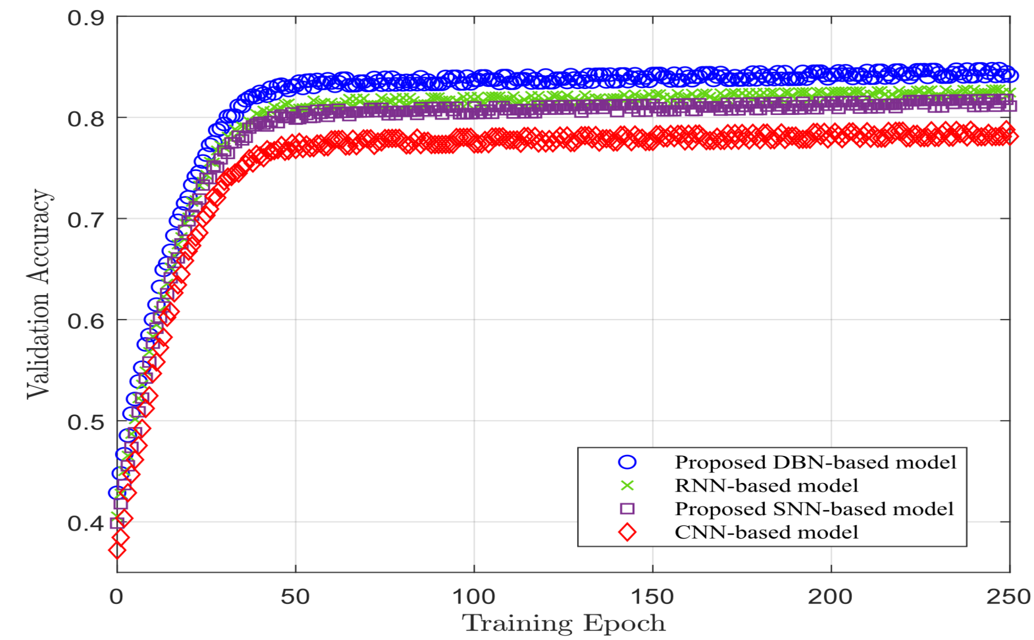


Figure 5.9: *Proposed SNN-based model performance of number of training epochs versus validation accuracy.*

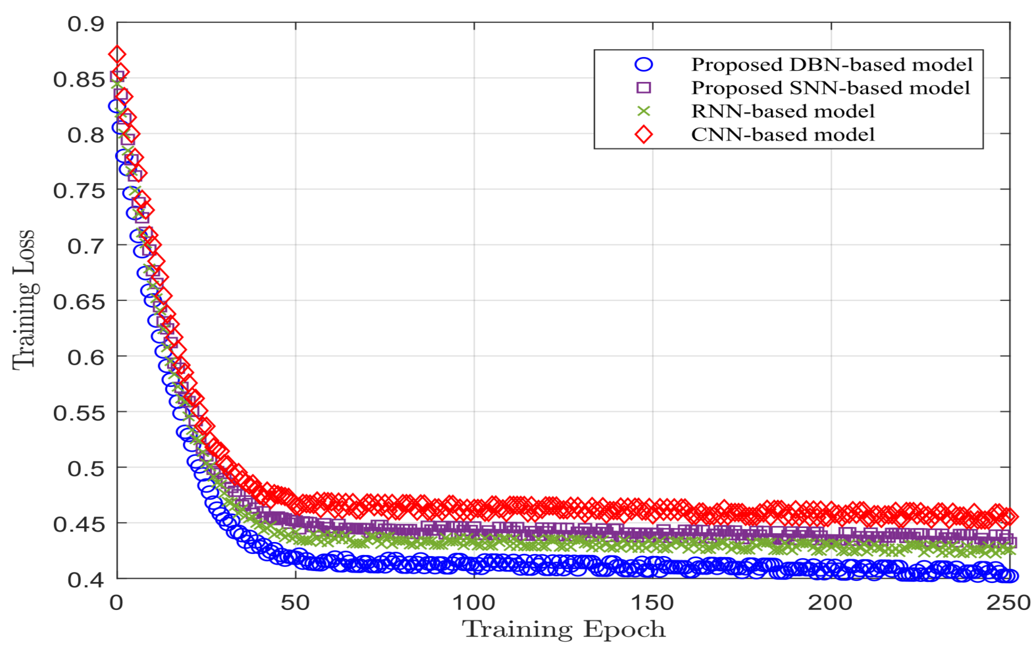


Figure 5.10: *Proposed SNN-based model performance of number of training epochs versus training loss.*

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.

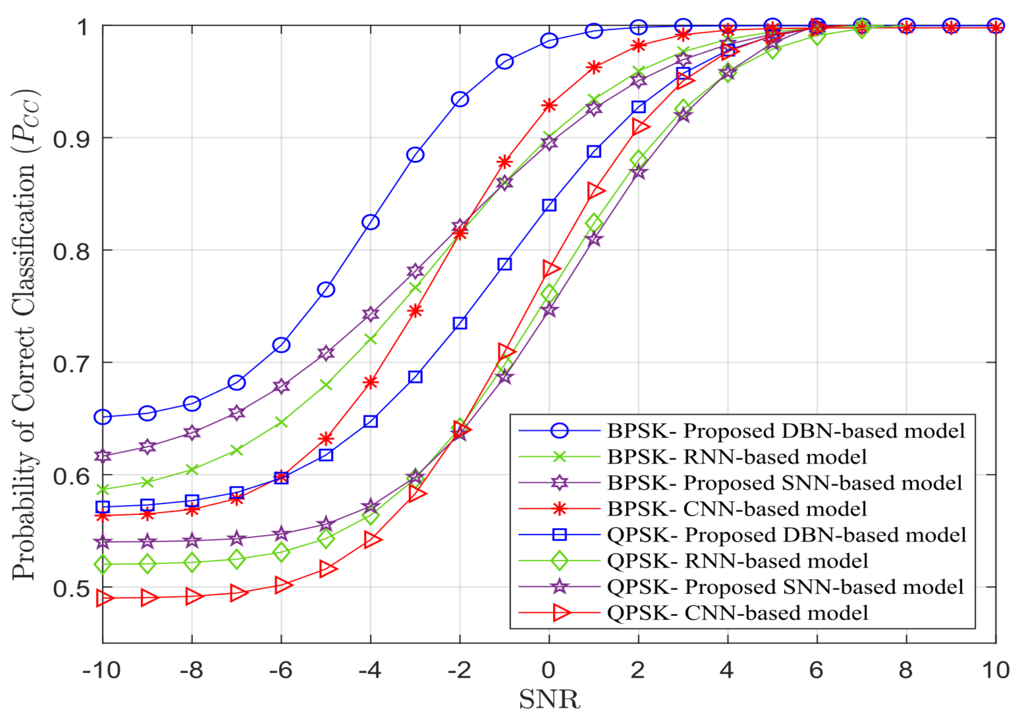


Figure 5.11: *Proposed SNN-based model lower-bound performance compared to proposed DBN-, RNN- and CNN-based models.*

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%.

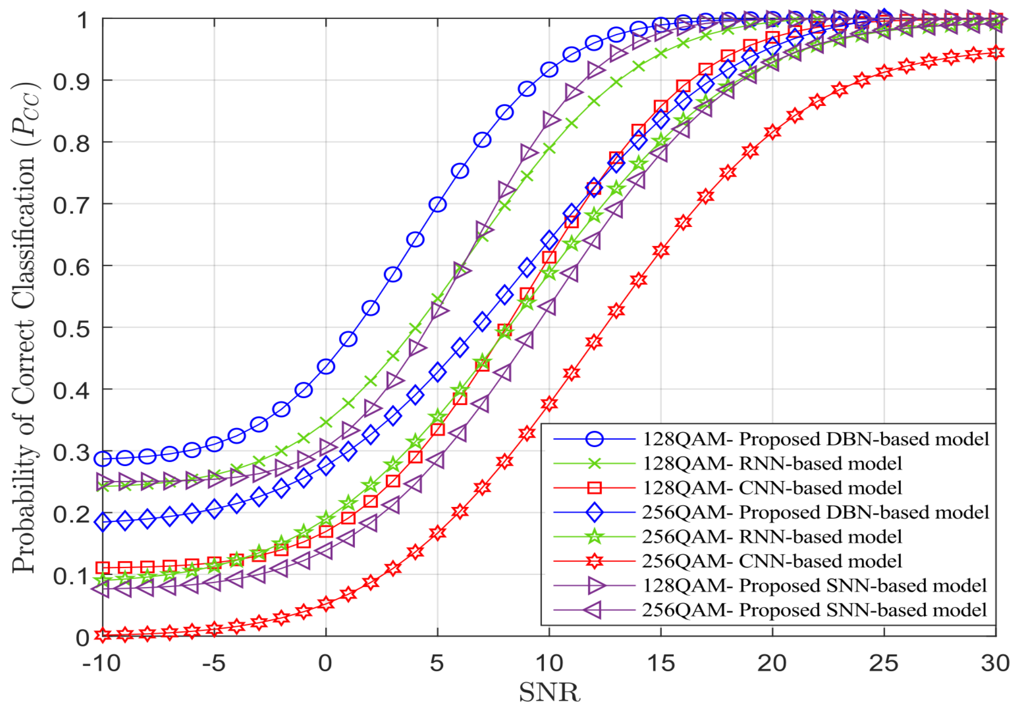


Figure 5.12: *Proposed SNN-based model upper-bound performance compared to proposed DBN-, RNN- and CNN-based models.*

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	0.00158τ
Each epoch	$15.84\ \tau$
Total classification	$3565.32\ \tau$

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

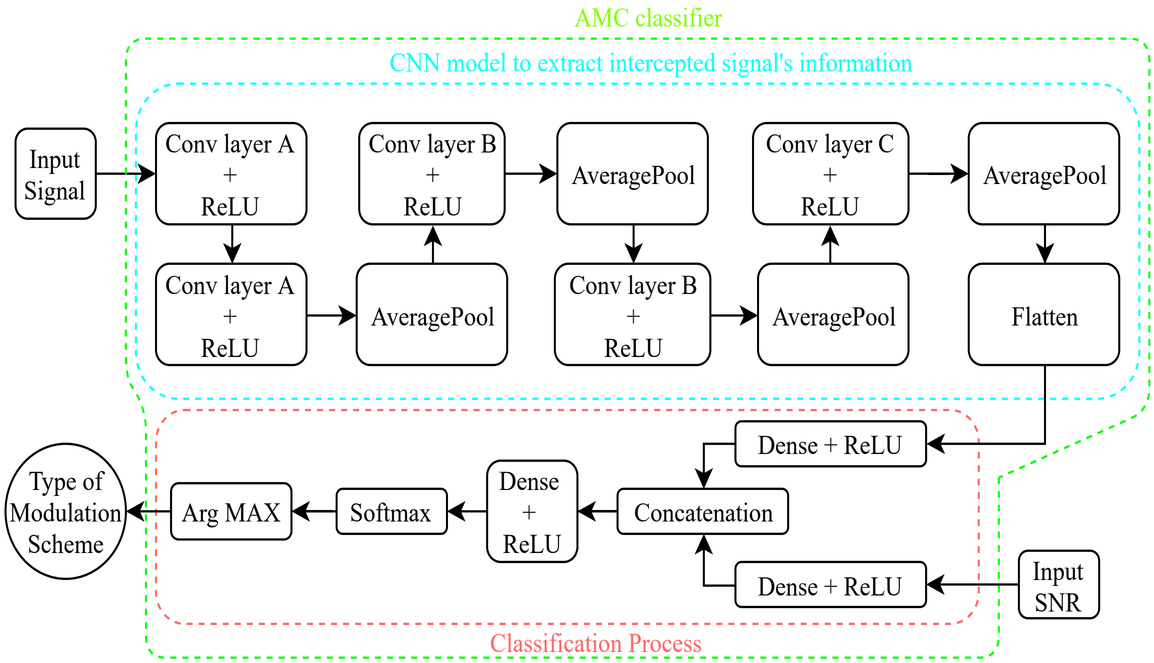


Figure A.1: *The deep CNN-based classifier's structure.*

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

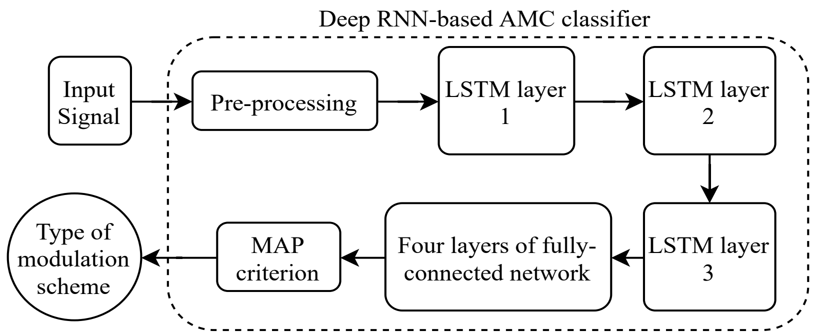


Figure A.2: *The deep RNN-based classifier's structure.*

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 written[1] by 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

```
from pathlib import Path

from . import (
    utils,
    network,
    models,
    analysis,
    preprocessing,
    datasets,
    encoding,
    pipeline,
    learning,
    evaluation,
    environment,
    conversion,
)

ROOT_DIR = Path(__file__).parents[0].parents[0]
```

Listing B.1: *Initialization*

```
import math
import torch
import numpy as np
```

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

```python

from torch import Tensor
import torch.nn.functional as F
from numpy import ndarray
from typing import Tuple, Union
from torch.nn.modules.utils import _pair


def im2col_indices(
    x: Tensor,
    kernel_height: int,
    kernel_width: int,
    padding: Tuple[int, int] = (0, 0),
    stride: Tuple[int, int] = (1, 1),
) -> Tensor:
    # language=rst
    """
    im2col is a special case of unfold which is implemented inside of Pytorch.

    :param x: Input image tensor to be reshaped to column-wise format.
    :param kernel_height: Height of the convolutional kernel in pixels.
    :param kernel_width: Width of the convolutional kernel in pixels.
    :param padding: Amount of zero padding on the input image.
    :param stride: Amount to stride over image by per convolution.
    :return: Input tensor reshaped to column-wise format.
    """
    return F.unfold(x, (kernel_height, kernel_width), padding=padding, stride=stride)


def col2im_indices(
    cols: Tensor,
    x_shape: Tuple[int, int, int, int],
    kernel_height: int,
    kernel_width: int,
    padding: Tuple[int, int] = (0, 0),
    stride: Tuple[int, int] = (1, 1),
) -> Tensor:
    # language=rst
    """
    col2im is a special case of fold which is implemented inside of Pytorch.

    :param cols: Image tensor in column-wise format.
    :param x_shape: Shape of original image tensor.
    :param kernel_height: Height of the convolutional kernel in pixels.
    :param kernel_width: Width of the convolutional kernel in pixels.
    :param padding: Amount of zero padding on the input image.
```

```
    :param stride: Amount to stride over image by per convolution.
    :return: Image tensor in original image shape.
    """
    return F.fold(
        cols, x_shape, (kernel_height, kernel_width), padding=
padding, stride=stride
    )


def get_square_weights(
    weights: Tensor, n_sqrt: int, side: Union[int, Tuple[int, int]]
) -> Tensor:
    # language=rst
    """
    Return a grid of a number of filters ``sqrt ** 2`` with side
lengths ``side``.

    :param weights: Two-dimensional tensor of weights for two-
dimensional data.
    :param n_sqrt: Square root of no. of filters.
    :param side: Side length(s) of filter.
    :return: Reshaped weights to square matrix of filters.
    """
    if isinstance(side, int):
        side = (side, side)

    square_weights = torch.zeros(side[0] * n_sqrt, side[1] * n_sqrt)
    for i in range(n_sqrt):
        for j in range(n_sqrt):
            n = i * n_sqrt + j

            if not n < weights.size(1):
                break

            x = i * side[0]
            y = (j % n_sqrt) * side[1]
            filter_ = weights[:, n].contiguous().view(*side)
            square_weights[x : x + side[0], y : y + side[1]] =
filter_

    return square_weights


def get_square_assignments(assignments: Tensor, n_sqrt: int) ->
Tensor:
    # language=rst
    """
    Return a grid of assignments.

    :param assignments: Vector of integers corresponding to class
labels.
    :param n_sqrt: Square root of no. of assignments.
    :return: Reshaped square matrix of assignments.
    """
```

```
    square_assignments = torch.mul(torch.ones(n_sqrt, n_sqrt), -1.0)
    for i in range(n_sqrt):
        for j in range(n_sqrt):
            n = i * n_sqrt + j

            if not n < assignments.size(0):
                break

            square_assignments[
                i : (i + 1), (j % n_sqrt) : ((j % n_sqrt) + 1)
            ] = assignments[n]

    return square_assignments


def reshape_locally_connected_weights(
    w: Tensor,
    n_filters: int,
    kernel_size: Union[int, Tuple[int, int]],
    conv_size: Union[int, Tuple[int, int]],
    locations: Tensor,
    input_sqrt: Union[int, Tuple[int, int]],
) -> Tensor:
    # language=rst
    """
    Get the weights from a locally connected layer and reshape them to be two-dimensional and square.

    :param w: Weights from a locally connected layer.
    :param n_filters: No. of neuron filters.
    :param kernel_size: Side length(s) of convolutional kernel.
    :param conv_size: Side length(s) of convolution population.
    :param locations: Binary mask indicating receptive fields of convolution population neurons.
    :param input_sqrt: Sides length(s) of input neurons.
    :return: Locally connected weights reshaped as a collection of spatially ordered square grids.
    """
    kernel_size = _pair(kernel_size)
    conv_size = _pair(conv_size)
    input_sqrt = _pair(input_sqrt)

    k1, k2 = kernel_size
    c1, c2 = conv_size
    i1, i2 = input_sqrt
    c1sqrt, c2sqrt = int(math.ceil(math.sqrt(c1))), int(math.ceil(math.sqrt(c2)))
    fs = int(math.ceil(math.sqrt(n_filters)))

    w_ = torch.zeros((n_filters * k1, k2 * c1 * c2))

    for n1 in range(c1):
        for n2 in range(c2):
            for feature in range(n_filters):
```

```
                n = n1 * c2 + n2
                filter_ = w[
                    locations[:, n],
                    feature * (c1 * c2) + (n // c2sqrt) * c2sqrt + (n % c2sqrt),
                ].view(k1, k2)
                w_[feature * k1 : (feature + 1) * k1, n * k2 : (n + 1) * k2] = filter_

    if c1 == 1 and c2 == 1:
        square = torch.zeros((i1 * fs, i2 * fs))

        for n in range(n_filters):
            square[
                (n // fs) * i1 : ((n // fs) + 1) * i2,
                (n % fs) * i2 : ((n % fs) + 1) * i2,
            ] = w_[n * i1 : (n + 1) * i2]

        return square
    else:
        square = torch.zeros((k1 * fs * c1, k2 * fs * c2))

        for n1 in range(c1):
            for n2 in range(c2):
                for f1 in range(fs):
                    for f2 in range(fs):
                        if f1 * fs + f2 < n_filters:
                            square[
                                k1 * (n1 * fs + f1) : k1 * (n1 * fs + f1 + 1),
                                k2 * (n2 * fs + f2) : k2 * (n2 * fs + f2 + 1),
                            ] = w_[
                                (f1 * fs + f2) * k1 : (f1 * fs + f2 + 1) * k1,
                                (n1 * c2 + n2) * k2 : (n1 * c2 + n2 + 1) * k2,
                            ]

        return square


def reshape_conv2d_weights(weights: torch.Tensor) -> torch.Tensor:
    # language=rst
    """
    Flattens a connection weight matrix of a Conv2dConnection

    :param weights: Weight matrix of Conv2dConnection object.
    :param wmin: Minimum allowed weight value.
    :param wmax: Maximum allowed weight value.
    """
    sqrt1 = int(np.ceil(np.sqrt(weights.size(0))))
    sqrt2 = int(np.ceil(np.sqrt(weights.size(1))))
    height, width = weights.size(2), weights.size(3)
```

```
    reshaped = torch.zeros(
        sqrt1 * sqrt2 * weights.size(2), sqrt1 * sqrt2 * weights.
size(3)
    )

    for i in range(sqrt1):
        for j in range(sqrt1):
            for k in range(sqrt2):
                for l in range(sqrt2):
                    if i * sqrt1 + j < weights.size(0) and k * sqrt2
 + l < weights.size(
                        1
                    ):
                        fltr = weights[i * sqrt1 + j, k * sqrt2 + l
].view(height, width)
                        reshaped[
                            i * height
                            + k * height * sqrt1 : (i + 1) * height
                            + k * height * sqrt1,
                            (j % sqrt1) * width
                            + (l % sqrt2) * width * sqrt1 : ((j %
sqrt1) + 1) * width
                            + (l % sqrt2) * width * sqrt1,
                        ] = fltr

    return reshaped
```

Listing B.2: *Refactored conversion module*

B.2 Environment and its Initialization

```
from .environment import Environment, GymEnvironment
```

Listing B.3: *Initialization*

```
from abc import ABC, abstractmethod
from typing import Tuple, Dict, Any

import gym
import numpy as np
import torch

from ..datasets.preprocess import subsample, gray_scale,
    binary_image, crop
from ..encoding import Encoder, NullEncoder


class Environment(ABC):
    # language=rst
    """
    Abstract environment class.
    """

    @abstractmethod
```

```
    def step(self, a: int) -> Tuple[Any, ...]:
        # language=rst
        """
        Abstract method head for ``step()``.

        :param a: Integer action to take in environment.
        """
        pass

    @abstractmethod
    def reset(self) -> None:
        # language=rst
        """
        Abstract method header for ``reset()``.
        """
        pass

    @abstractmethod
    def render(self) -> None:
        # language=rst
        """
        Abstract method header for ``render()``.
        """
        pass

    @abstractmethod
    def close(self) -> None:
        # language=rst
        """
        Abstract method header for ``close()``.
        """
        pass

    @abstractmethod
    def preprocess(self) -> None:
        # language=rst
        """
        Abstract method header for ``preprocess()``.
        """
        pass


class GymEnvironment(Environment):
    # language=rst
    """
    A wrapper around the OpenAI ``gym`` environments.
    """

    def __init__(self, name: str, encoder: Encoder = NullEncoder(),
   **kwargs) -> None:
        # language=rst
        """
        Initializes the environment wrapper. This class makes the
        assumption that the OpenAI ``gym`` environment will provide
```

```
an image
        of format HxW or CxHxW as an observation (we will add the C
        dimension to HxW tensors) or a 1D observation in which case no
        dimensions will be added.

        :param name: The name of an OpenAI ``gym`` environment.
        :param encoder: Function to encode observations into spike trains.

        Keyword arguments:

        :param float max_prob: Maximum spiking probability.
        :param bool clip_rewards: Whether or not to use ``np.sign`` of rewards.

        :param int history: Number of observations to keep track of.
        :param int delta: Step size to save observations in history.
        :param bool add_channel_dim: Allows for the adding of the channel dimension in
            2D inputs.
        """
        self.name = name
        self.env = gym.make(name)
        self.action_space = self.env.action_space

        self.encoder = encoder

        # Keyword arguments.
        self.max_prob = kwargs.get("max_prob", 1.0)
        self.clip_rewards = kwargs.get("clip_rewards", True)

        self.history_length = kwargs.get("history_length", None)
        self.delta = kwargs.get("delta", 1)
        self.add_channel_dim = kwargs.get("add_channel_dim", True)

        if self.history_length is not None and self.delta is not None:
            self.history = {
                i: torch.Tensor()
                for i in range(1, self.history_length * self.delta + 1, self.delta)
            }
        else:
            self.history = {}

        self.episode_step_count = 0
        self.history_index = 1

        self.obs = None
        self.reward = None

        assert (
            0.0 < self.max_prob <= 1.0
```

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

    def step(self, a: int) -> Tuple[torch.Tensor, float, bool, Dict[Any, Any]]:
        # language=rst
        """
        Wrapper around the OpenAI ``gym`` environment ``step()`` function.

        :param a: Action to take in the environment.
        :return: Observation, reward, done flag, and information dictionary.
        """
        # Call gym's environment step function.
        self.obs, self.reward, self.done, info = self.env.step(a)

        if self.clip_rewards:
            self.reward = np.sign(self.reward)

        self.preprocess()

        # Add the raw observation from the gym environment into the info
        # for debugging and display.
        info["gym_obs"] = self.obs

        # Store frame of history and encode the inputs.
        if len(self.history) > 0:
            self.update_history()
            self.update_index()
            # Add the delta observation into the info for debugging and display.
            info["delta_obs"] = self.obs

        # The new standard for images is BxTxCxHxW.
        # The gym environment doesn't follow exactly the same protocol.
        #
        # 1D observations will be left as is before the encoder and will become BxTxL.
        # 2D observations are assumed to be mono images will become BxTx1xHxW
        # 3D observations will become BxTxCxHxW
        if self.obs.dim() == 2 and self.add_channel_dim:
            # We want CxHxW, it is currently HxW.
            self.obs = self.obs.unsqueeze(0)

        # The encoder will add time - now Tx...
        if self.encoder is not None:
            self.obs = self.encoder(self.obs)

        # Add the batch - now BxTx...
        self.obs = self.obs.unsqueeze(0)

```

```
        self.episode_step_count += 1

        # Return converted observations and other information.
        return self.obs, self.reward, self.done, info

    def reset(self) -> torch.Tensor:
        # language=rst
        """
        Wrapper around the OpenAI ``gym`` environment ``reset()`` function.

        :return: Observation from the environment.
        """
        # Call gym's environment reset function.
        self.obs = self.env.reset()
        self.preprocess()

        self.history = {i: torch.Tensor() for i in self.history}

        self.episode_step_count = 0

        return self.obs

    def render(self) -> None:
        # language=rst
        """
        Wrapper around the OpenAI ``gym`` environment ``render()`` function.
        """
        self.env.render()

    def close(self) -> None:
        # language=rst
        """
        Wrapper around the OpenAI ``gym`` environment ``close()`` function.
        """
        self.env.close()

    def preprocess(self) -> None:
        # language=rst
        """
        Pre-processing step for an observation from a ``gym`` environment.
        """
        if self.name == "SpaceInvaders-v0":
            self.obs = subsample(gray_scale(self.obs), 84, 110)
            self.obs = self.obs[26:104, :]
            self.obs = binary_image(self.obs)
        elif self.name == "BreakoutDeterministic-v4":
            self.obs = subsample(gray_scale(crop(self.obs, 34, 194, 0, 160)), 80, 80)
            self.obs = binary_image(self.obs)
        else:  # Default pre-processing step.
```

```
            pass

        self.obs = torch.from_numpy(self.obs).float()

    def update_history(self) -> None:
        # language=rst
        """
        Updates the observations inside history by performing subtraction from most
        recent observation and the sum of previous observations. If there are not enough
        observations to take a difference from, simply store the observation without any
        differencing.
        """
        # Recording initial observations.
        if self.episode_step_count < len(self.history) * self.delta:
            # Store observation based on delta value.
            if self.episode_step_count % self.delta == 0:
                self.history[self.history_index] = self.obs
        else:
            # Take difference between stored frames and current frame.
            temp = torch.clamp(self.obs - sum(self.history.values()), 0, 1)

            # Store observation based on delta value.
            if self.episode_step_count % self.delta == 0:
                self.history[self.history_index] = self.obs

            assert (
                len(self.history) == self.history_length
            ), "History size is out of bounds"
            self.obs = temp

    def update_index(self) -> None:
        # language=rst
        """
        Updates the index to keep track of history. For example: ``history = 4``,
        ``delta = 3`` will produce ``self.history = {1, 4, 7, 10}`` and
        ``self.history_index`` will be updated according to ``self.delta`` and will wrap
        around the history dictionary.
        """
        if self.episode_step_count % self.delta == 0:
            if self.history_index != max(self.history.keys()):
                self.history_index += self.delta
            else:
                # Wrap around the history.
                self.history_index = (self.history_index % max(self.
```

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

Listing B.4: Environment

B.3 Network

```python
from .network import Network, load
from . import nodes, topology, monitors
```

Listing B.5: Initialization

```python
import tempfile
from typing import Dict, Optional, Type, Iterable

import torch

from .monitors import AbstractMonitor
from .nodes import Nodes
from .topology import AbstractConnection
from ..learning.reward import AbstractReward


def load(file_name: str, map_location: str = "cpu", learning: bool = None) -> "Network":
    # language=rst
    """
    Loads serialized network object from disk.

    :param file_name: Path to serialized network object on disk.
    :param map_location: One of ``"cpu"`` or ``"cuda"``. Defaults to ``"cpu"``.
    :param learning: Whether to load with learning enabled. Default loads value from
        disk.
    """
    network = torch.load(open(file_name, "rb"), map_location=map_location)
    if learning is not None and "learning" in vars(network):
        network.learning = learning

    return network


class Network(torch.nn.Module):
    # language=rst
    """
    Central object of the ``bindsnet`` package. Responsible for the simulation and
    interaction of nodes and connections.

    **Example:**

    .. code-block:: python

```

```
        import torch
        import matplotlib.pyplot as plt

        from bindsnet         import encoding
        from bindsnet.network import Network, nodes, topology, monitors

        network = Network(dt=1.0)  # Instantiates network.

        X = nodes.Input(100)  # Input layer.
        Y = nodes.LIFNodes(100)  # Layer of LIF neurons.
        C = topology.Connection(source=X, target=Y, w=torch.rand(X.n, Y.n))  # Connection from X to Y.

        # Spike monitor objects.
        M1 = monitors.Monitor(obj=X, state_vars=['s'])
        M2 = monitors.Monitor(obj=Y, state_vars=['s'])

        # Add everything to the network object.
        network.add_layer(layer=X, name='X')
        network.add_layer(layer=Y, name='Y')
        network.add_connection(connection=C, source='X', target='Y')
        network.add_monitor(monitor=M1, name='X')
        network.add_monitor(monitor=M2, name='Y')

        # Create Poisson-distributed spike train inputs.
        data = 15 * torch.rand(100)  # Generate random Poisson rates for 100 input neurons.
        train = encoding.poisson(datum=data, time=5000)  # Encode input as 5000ms Poisson spike trains.

        # Simulate network on generated spike trains.
        inputs = {'X' : train}  # Create inputs mapping.
        network.run(inputs=inputs, time=5000)  # Run network simulation.

        # Plot spikes of input and output layers.
        spikes = {'X' : M1.get('s'), 'Y' : M2.get('s')}

        fig, axes = plt.subplots(2, 1, figsize=(12, 7))
        for i, layer in enumerate(spikes):
            axes[i].matshow(spikes[layer], cmap='binary')
            axes[i].set_title('%s spikes' % layer)
            axes[i].set_xlabel('Time'); axes[i].set_ylabel('Index of neuron')
            axes[i].set_xticks(()); axes[i].set_yticks(())
            axes[i].set_aspect('auto')

        plt.tight_layout(); plt.show()
    """

    def __init__(
        self,
        dt: float = 1.0,
```

```
        batch_size: int = 1,
        learning: bool = True,
        reward_fn: Optional[Type[AbstractReward]] = None,
    ) -> None:
        # language=rst
        """
        Initializes network object.

        :param dt: Simulation timestep.
        :param batch_size: Mini-batch size.
        :param learning: Whether to allow connection updates. True by default.
        :param reward_fn: Optional class allowing for modification of reward in case of
            reward-modulated learning.
        """
        super().__init__()

        self.dt = dt
        self.batch_size = batch_size

        self.layers = {}
        self.connections = {}
        self.monitors = {}

        self.train(learning)

        if reward_fn is not None:
            self.reward_fn = reward_fn()
        else:
            self.reward_fn = None

    def add_layer(self, layer: Nodes, name: str) -> None:
        # language=rst
        """
        Adds a layer of nodes to the network.

        :param layer: A subclass of the ``Nodes`` object.
        :param name: Logical name of layer.
        """
        self.layers[name] = layer
        self.add_module(name, layer)

        layer.train(self.learning)
        layer.compute_decays(self.dt)
        layer.set_batch_size(self.batch_size)

    def add_connection(
        self, connection: AbstractConnection, source: str, target: str
    ) -> None:
        # language=rst
        """
        Adds a connection between layers of nodes to the network.
```

```python

        :param connection: An instance of class ``Connection``.
        :param source: Logical name of the connection's source layer.
        :param target: Logical name of the connection's target layer.
        """
        self.connections[(source, target)] = connection
        self.add_module(source + "_to_" + target, connection)

        connection.dt = self.dt
        connection.train(self.learning)

    def add_monitor(self, monitor: AbstractMonitor, name: str) -> None:
        # language=rst
        """
        Adds a monitor on a network object to the network.

        :param monitor: An instance of class ``Monitor``.
        :param name: Logical name of monitor object.
        """
        self.monitors[name] = monitor
        monitor.network = self
        monitor.dt = self.dt

    def save(self, file_name: str) -> None:
        # language=rst
        """
        Serializes the network object to disk.

        :param file_name: Path to store serialized network object on disk.

        **Example:**

        .. code-block:: python

            import torch
            import matplotlib.pyplot as plt

            from pathlib          import Path
            from bindsnet.network import *
            from bindsnet.network import topology

            # Build simple network.
            network = Network(dt=1.0)

            X = nodes.Input(100)  # Input layer.
            Y = nodes.LIFNodes(100)  # Layer of LIF neurons.
            C = topology.Connection(source=X, target=Y, w=torch.rand(X.n, Y.n))  # Connection from X to Y.

            # Add everything to the network object.
```

```
            network.add_layer(layer=X, name='X')
            network.add_layer(layer=Y, name='Y')
            network.add_connection(connection=C, source='X', target='Y')

            # Save the network to disk.
            network.save(str(Path.home()) + '/network.pt')
        """
        torch.save(self, open(file_name, "wb"))

    def clone(self) -> "Network":
        # language=rst
        """
        Returns a cloned network object.

        :return: A copy of this network.
        """
        virtual_file = tempfile.SpooledTemporaryFile()
        torch.save(self, virtual_file)
        virtual_file.seek(0)
        return torch.load(virtual_file)

    def _get_inputs(self, layers: Iterable = None) -> Dict[str, torch.Tensor]:
        # language=rst
        """
        Fetches outputs from network layers to use as input to downstream layers.

        :param layers: Layers to update inputs for. Defaults to all network layers.
        :return: Inputs to all layers for the current iteration.
        """
        inputs = {}

        if layers is None:
            layers = self.layers

        # Loop over network connections.
        for c in self.connections:
            if c[1] in layers:
                # Fetch source and target populations.
                source = self.connections[c].source
                target = self.connections[c].target

                if not c[1] in inputs:
                    inputs[c[1]] = torch.zeros(
                        self.batch_size, *target.shape, device=target.s.device
                    )

                # Add to input: source's spikes multiplied by connection weights.
                inputs[c[1]] += self.connections[c].compute(source.s
```

```
        )

        return inputs

    def run(
        self, inputs: Dict[str, torch.Tensor], time: int, one_step=False, **kwargs
    ) -> None:
        # language=rst
        """
        Simulate network for given inputs and time.

        :param inputs: Dictionary of ``Tensor``s of shape ``[time, *input_shape]`` or
                       ``[time, batch_size, *input_shape]``.
        :param time: Simulation time.
        :param one_step: Whether to run the network in "feed-forward" mode, where inputs
            propagate all the way through the network in a single simulation time step.
            Layers are updated in the order they are added to the network.

        Keyword arguments:

        :param Dict[str, torch.Tensor] clamp: Mapping of layer names to boolean masks if
            neurons should be clamped to spiking. The ``Tensor``s have shape
            ``[n_neurons]`` or ``[time, n_neurons]``.
        :param Dict[str, torch.Tensor] unclamp: Mapping of layer names to boolean masks
            if neurons should be clamped to not spiking. The ``Tensor``s should have
            shape ``[n_neurons]`` or ``[time, n_neurons]``.
        :param Dict[str, torch.Tensor] injects_v: Mapping of layer names to boolean
            masks if neurons should be added voltage. The ``Tensor``s should have shape
            ``[n_neurons]`` or ``[time, n_neurons]``.
        :param Union[float, torch.Tensor] reward: Scalar value used in reward-modulated
            learning.
        :param Dict[Tuple[str], torch.Tensor] masks: Mapping of connection names to
            boolean masks determining which weights to clamp to zero.

        **Example:**

        .. code-block:: python

            import torch
            import matplotlib.pyplot as plt
```

```

            from bindsnet.network import Network
            from bindsnet.network.nodes import Input
            from bindsnet.network.monitors import Monitor

            # Build simple network.
            network = Network()
            network.add_layer(Input(500), name='I')
            network.add_monitor(Monitor(network.layers['I'], state_vars=['s']), 'I')

            # Generate spikes by running Bernoulli trials on Uniform(0, 0.5) samples.
            spikes = torch.bernoulli(0.5 * torch.rand(500, 500))

            # Run network simulation.
            network.run(inputs={'I' : spikes}, time=500)

            # Look at input spiking activity.
            spikes = network.monitors['I'].get('s')
            plt.matshow(spikes, cmap='binary')
            plt.xticks(()); plt.yticks(());
            plt.xlabel('Time'); plt.ylabel('Neuron index')
            plt.title('Input spiking')
            plt.show()
        """
        # Parse keyword arguments.
        clamps = kwargs.get("clamp", {})
        unclamps = kwargs.get("unclamp", {})
        masks = kwargs.get("masks", {})
        injects_v = kwargs.get("injects_v", {})

        # Compute reward.
        if self.reward_fn is not None:
            kwargs["reward"] = self.reward_fn.compute(**kwargs)

        # Dynamic setting of batch size.
        if inputs != {}:
            for key in inputs:
                # goal shape is [time, batch, n_0, ...]
                if len(inputs[key].size()) == 1:
                    # current shape is [n_0, ...]
                    # unsqueeze twice to make [1, 1, n_0, ...]
                    inputs[key] = inputs[key].unsqueeze(0).unsqueeze(0)
                elif len(inputs[key].size()) == 2:
                    # current shape is [time, n_0, ...]
                    # unsqueeze dim 1 so that we have
                    # [time, 1, n_0, ...]
                    inputs[key] = inputs[key].unsqueeze(1)

            for key in inputs:
                # batch dimension is 1, grab this and use for batch size
```

```
                if inputs[key].size(1) != self.batch_size:
                    self.batch_size = inputs[key].size(1)

                    for l in self.layers:
                        self.layers[l].set_batch_size(self.batch_size)

                    for m in self.monitors:
                        self.monitors[m].reset_state_variables()

                break

        # Effective number of timesteps.
        timesteps = int(time / self.dt)

        # Simulate network activity for `time` timesteps.
        for t in range(timesteps):
            # Get input to all layers (synchronous mode).
            current_inputs = {}
            if not one_step:
                current_inputs.update(self._get_inputs())

            for l in self.layers:
                # Update each layer of nodes.
                if l in inputs:
                    if l in current_inputs:
                        current_inputs[l] += inputs[l][t]
                    else:
                        current_inputs[l] = inputs[l][t]

                if one_step:
                    # Get input to this layer (one-step mode).
                    current_inputs.update(self._get_inputs(layers=[l]))

                self.layers[l].forward(x=current_inputs[l])

                # Clamp neurons to spike.
                clamp = clamps.get(l, None)
                if clamp is not None:
                    if clamp.ndimension() == 1:
                        self.layers[l].s[:, clamp] = 1
                    else:
                        self.layers[l].s[:, clamp[t]] = 1

                # Clamp neurons not to spike.
                unclamp = unclamps.get(l, None)
                if unclamp is not None:
                    if unclamp.ndimension() == 1:
                        self.layers[l].s[unclamp] = 0
                    else:
                        self.layers[l].s[unclamp[t]] = 0

                # Inject voltage to neurons.
```

```
                inject_v = injects_v.get(l, None)
                if inject_v is not None:
                    if inject_v.ndimension() == 1:
                        self.layers[l].v += inject_v
                    else:
                        self.layers[l].v += inject_v[t]

            # Run synapse updates.
            for c in self.connections:
                self.connections[c].update(
                    mask=masks.get(c, None), learning=self.learning, **kwargs
                )

            # Get input to all layers.
            current_inputs.update(self._get_inputs())

            # Record state variables of interest.
            for m in self.monitors:
                self.monitors[m].record()

        # Re-normalize connections.
        for c in self.connections:
            self.connections[c].normalize()

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Reset state variables of objects in network.
        """
        for layer in self.layers:
            self.layers[layer].reset_state_variables()

        for connection in self.connections:
            self.connections[connection].reset_state_variables()

        for monitor in self.monitors:
            self.monitors[monitor].reset_state_variables()

    def train(self, mode: bool = True) -> "torch.nn.Module":
        # language=rst
        """
        Sets the node in training mode.

        :param mode: Turn training on or off.

        :return: ``self`` as specified in ``torch.nn.Module``.
        """
        self.learning = mode
        return super().train(mode)
```

Listing B.6: *Network*

```
from abc import ABC, abstractmethod
```

```python
from functools import reduce
from operator import mul
from typing import Iterable, Optional, Union

import torch


class Nodes(torch.nn.Module):
    # language=rst
    """
    Abstract base class for groups of neurons.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        learning: bool = True,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Abstract base class constructor.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record decaying spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param learning: Whether to be in learning or testing.
        """
        super().__init__()

        assert (
            n is not None or shape is not None
        ), "Must provide either no. of neurons or shape of layer"

        if n is None:
            self.n = reduce(mul, shape)  # No. of neurons product of shape.
        else:
            self.n = n  # No. of neurons provided.

        if shape is None:
            self.shape = [self.n]  # Shape is equal to the size of the layer.
```

```
        else:
            self.shape = shape  # Shape is passed in as an argument.

        assert self.n == reduce(
            mul, self.shape
        ), "No. of neurons and shape do not match"

        self.traces = traces  # Whether to record synaptic traces.
        self.traces_additive = (
            traces_additive
        )  # Whether to record spike traces additively.
        self.register_buffer("s", torch.ByteTensor())  # Spike occurrences.

        self.sum_input = sum_input  # Whether to sum all inputs.

        if self.traces:
            self.register_buffer("x", torch.Tensor())  # Firing traces.
            self.register_buffer(
                "tc_trace", torch.tensor(tc_trace)
            )  # Time constant of spike trace decay.
            if self.traces_additive:
                self.register_buffer(
                    "trace_scale", torch.tensor(trace_scale)
                )  # Scaling factor for spike trace.
            self.register_buffer(
                "trace_decay", torch.empty_like(self.tc_trace)
            )  # Set in compute_decays.

        if self.sum_input:
            self.register_buffer("summed", torch.FloatTensor())  # Summed inputs.

        self.dt = None
        self.batch_size = None
        self.trace_decay = None
        self.learning = learning

    @abstractmethod
    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Abstract base class method for a single simulation step.

        :param x: Inputs to the layer.
        """
        if self.traces:
            # Decay and set spike traces.
            self.x *= self.trace_decay

            if self.traces_additive:
                self.x += self.trace_scale * self.s.float()
            else:
```

```
                self.x.masked_fill_(self.s != 0, 1)

        if self.sum_input:
            # Add current input to running sum.
            self.summed += x.float()

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Abstract base class method for resetting state variables.
        """
        self.s.zero_()

        if self.traces:
            self.x.zero_()  # Spike traces.

        if self.sum_input:
            self.summed.zero_()  # Summed inputs.

    def compute_decays(self, dt) -> None:
        # language=rst
        """
        Abstract base class method for setting decays.
        """
        self.dt = dt
        if self.traces:
            self.trace_decay = torch.exp(
                -self.dt / self.tc_trace
            )  # Spike trace decay (per timestep).

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
        self.batch_size = batch_size
        self.s = torch.zeros(batch_size, *self.shape, device=self.s.device)

        if self.traces:
            self.x = torch.zeros(batch_size, *self.shape, device=self.x.device)

        if self.sum_input:
            self.summed = torch.zeros(
                batch_size, *self.shape, device=self.summed.device
            )

    def train(self, mode: bool = True) -> "Nodes":
        # language=rst
        """
```

```
        Sets the layer in training mode.

        :param bool mode: Turn training on or off
        :return: self as specified in `torch.nn.Module`
        """
        self.learning = mode
        return super().train(mode)


class AbstractInput(ABC):
    # language=rst
    """
    Abstract base class for groups of input neurons.
    """


class Input(Nodes, AbstractInput):
    # language=rst
    """
    Layer of nodes with user-specified spiking behavior.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of input neurons.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record decaying spike traces.
        :param traces_additive: Whether to record spike traces
    additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
```

```
        )

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        On each simulation step, set the spikes of the population equal to the inputs.

        :param x: Inputs to the layer.
        """
        # Set spike occurrences to input values.
        self.s = x

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()


class McCullochPitts(Nodes):
    # language=rst
    """
    Layer of `McCulloch-Pitts neurons
    <http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html>`_.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        thresh: Union[float, torch.Tensor] = 1.0,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a McCulloch-Pitts layer of neurons.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
```

```
        :param sum_input: Whether to sum all inputs.
        :param thresh: Spike threshold voltage.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer(
            "thresh", torch.tensor(thresh, dtype=torch.float)
        )  # Spike threshold voltage.
        self.register_buffer("v", torch.FloatTensor())  # Neuron voltages.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        self.v = x  # Voltages are equal to the inputs.
        self.s = self.v >= self.thresh  # Check for spiking neurons.

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
        super().set_batch_size(batch_size=batch_size)
        self.v = torch.zeros(batch_size, *self.shape, device=self.v.device)


class IFNodes(Nodes):
    # language=rst
    """
```

```
    Layer of `integrate-and-fire (IF) neurons <http://
neuronaldynamics.epfl.ch/online/Ch1.S3.html>`_.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        thresh: Union[float, torch.Tensor] = -52.0,
        reset: Union[float, torch.Tensor] = -65.0,
        refrac: Union[int, torch.Tensor] = 5,
        lbound: float = None,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of IF neurons.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces
additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param thresh: Spike threshold voltage.
        :param reset: Post-spike reset voltage.
        :param refrac: Refractory (non-firing) period of the neuron.
        :param lbound: Lower bound of the voltage.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer(
            "reset", torch.tensor(reset, dtype=torch.float)
        )  # Post-spike reset voltage.
        self.register_buffer(
            "thresh", torch.tensor(thresh, dtype=torch.float)
        )  # Spike threshold voltage.
        self.register_buffer(
            "refrac", torch.tensor(refrac)
```

```
        )  # Post-spike refractory period.
        self.register_buffer("v", torch.FloatTensor())  # Neuron voltages.
        self.register_buffer(
            "refrac_count", torch.FloatTensor()
        )  # Refractory period counters.

        self.lbound = lbound  # Lower bound of voltage.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        # Integrate input voltages.
        self.v += (self.refrac_count == 0).float() * x

        # Decrement refractory counters.
        self.refrac_count = (self.refrac_count > 0).float() * (
            self.refrac_count - self.dt
        )

        # Check for spiking neurons.
        self.s = self.v >= self.thresh

        # Refractoriness and voltage reset.
        self.refrac_count.masked_fill_(self.s, self.refrac)
        self.v.masked_fill_(self.s, self.reset)

        # Voltage clipping to lower bound.
        if self.lbound is not None:
            self.v.masked_fill_(self.v < self.lbound, self.lbound)

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()
        self.v.fill_(self.reset)  # Neuron voltages.
        self.refrac_count.zero_()  # Refractory period counters.

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
```

```
        super().set_batch_size(batch_size=batch_size)
        self.v = self.reset * torch.ones(batch_size, *self.shape, device=self.v.device)
        self.refrac_count = torch.zeros_like(self.v, device=self.refrac_count.device)


class LIFNodes(Nodes):
    # language=rst
    """
    Layer of `leaky integrate-and-fire (LIF) neurons
    <http://icwww.epfl.ch/~gerstner/SPNM/node26.html#SECTION02311000000000000000>`_.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        thresh: Union[float, torch.Tensor] = -52.0,
        rest: Union[float, torch.Tensor] = -65.0,
        reset: Union[float, torch.Tensor] = -65.0,
        refrac: Union[int, torch.Tensor] = 5,
        tc_decay: Union[float, torch.Tensor] = 100.0,
        lbound: float = None,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of LIF neurons.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param thresh: Spike threshold voltage.
        :param rest: Resting membrane voltage.
        :param reset: Post-spike reset voltage.
        :param refrac: Refractory (non-firing) period of the neuron.
        :param tc_decay: Time constant of neuron voltage decay.
        :param lbound: Lower bound of the voltage.
        """
        super().__init__(
            n=n,
            shape=shape,
```

```
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer(
            "rest", torch.tensor(rest, dtype=torch.float)
        )  # Rest voltage.
        self.register_buffer(
            "reset", torch.tensor(reset, dtype=torch.float)
        )  # Post-spike reset voltage.
        self.register_buffer(
            "thresh", torch.tensor(thresh, dtype=torch.float)
        )  # Spike threshold voltage.
        self.register_buffer(
            "refrac", torch.tensor(refrac)
        )  # Post-spike refractory period.
        self.register_buffer(
            "tc_decay", torch.tensor(tc_decay)
        )  # Time constant of neuron voltage decay.
        self.register_buffer(
            "decay", torch.zeros(*self.shape)
        )  # Set in compute_decays.
        self.register_buffer("v", torch.FloatTensor())  # Neuron
    voltages.
        self.register_buffer(
            "refrac_count", torch.FloatTensor()
        )  # Refractory period counters.

        self.lbound = lbound  # Lower bound of voltage.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        # Decay voltages.
        self.v = self.decay * (self.v - self.rest) + self.rest

        # Integrate inputs.
        self.v += (self.refrac_count == 0).float() * x

        # Decrement refractory counters.
        self.refrac_count = (self.refrac_count > 0).float() * (
            self.refrac_count - self.dt
        )

        # Check for spiking neurons.
        self.s = self.v >= self.thresh

```

```
        # Refractoriness and voltage reset.
        self.refrac_count.masked_fill_(self.s, self.refrac)
        self.v.masked_fill_(self.s, self.reset)

        # Voltage clipping to lower bound.
        if self.lbound is not None:
            self.v.masked_fill_(self.v < self.lbound, self.lbound)

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()
        self.v.fill_(self.rest)  # Neuron voltages.
        self.refrac_count.zero_()  # Refractory period counters.

    def compute_decays(self, dt) -> None:
        # language=rst
        """
        Sets the relevant decays.
        """
        super().compute_decays(dt=dt)
        self.decay = torch.exp(
            -self.dt / self.tc_decay
        )  # Neuron voltage decay (per timestep).

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
        super().set_batch_size(batch_size=batch_size)
        self.v = self.rest * torch.ones(batch_size, *self.shape, device=self.v.device)
        self.refrac_count = torch.zeros_like(self.v, device=self.refrac_count.device)


class CurrentLIFNodes(Nodes):
    # language=rst
    """
    Layer of `current-based leaky integrate-and-fire (LIF) neurons
    <http://icwww.epfl.ch/~gerstner/SPNM/node26.html#SECTION02313000000000000000>`_.
    Total synaptic input current is modeled as a decaying memory of input spikes multiplied by synaptic strengths.
    """

```

```
    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        thresh: Union[float, torch.Tensor] = -52.0,
        rest: Union[float, torch.Tensor] = -65.0,
        reset: Union[float, torch.Tensor] = -65.0,
        refrac: Union[int, torch.Tensor] = 5,
        tc_decay: Union[float, torch.Tensor] = 100.0,
        tc_i_decay: Union[float, torch.Tensor] = 2.0,
        lbound: float = None,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of synaptic input current-based LIF neurons.
        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param thresh: Spike threshold voltage.
        :param rest: Resting membrane voltage.
        :param reset: Post-spike reset voltage.
        :param refrac: Refractory (non-firing) period of the neuron.
        :param tc_decay: Time constant of neuron voltage decay.
        :param tc_i_decay: Time constant of synaptic input current decay.
        :param lbound: Lower bound of the voltage.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer("rest", torch.tensor(rest))  # Rest voltage.
        self.register_buffer("reset", torch.tensor(reset))  # Post-spike reset voltage.
        self.register_buffer("thresh", torch.tensor(thresh))  #
```

```
Spike threshold voltage.
        self.register_buffer(
            "refrac", torch.tensor(refrac)
        )  # Post-spike refractory period.
        self.register_buffer(
            "tc_decay", torch.tensor(tc_decay)
        )  # Time constant of neuron voltage decay.
        self.register_buffer(
            "decay", torch.empty_like(self.tc_decay)
        )  # Set in compute_decays.
        self.register_buffer(
            "tc_i_decay", torch.tensor(tc_i_decay)
        )  # Time constant of synaptic input current decay.
        self.register_buffer(
            "i_decay", torch.empty_like(self.tc_i_decay)
        )  # Set in compute_decays.

        self.register_buffer("v", torch.FloatTensor())  # Neuron voltages.
        self.register_buffer("i", torch.FloatTensor())  # Synaptic input currents.
        self.register_buffer(
            "refrac_count", torch.FloatTensor()
        )  # Refractory period counters.

        self.lbound = lbound  # Lower bound of voltage.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        # Decay voltages and current.
        self.v = self.decay * (self.v - self.rest) + self.rest
        self.i *= self.i_decay

        # Decrement refractory counters.
        self.refrac_count = (self.refrac_count > 0).float() * (
            self.refrac_count - self.dt
        )

        # Integrate inputs.
        self.i += x
        self.v += (self.refrac_count == 0).float() * self.i

        # Check for spiking neurons.
        self.s = self.v >= self.thresh

        # Refractoriness and voltage reset.
        self.refrac_count.masked_fill_(self.s, self.refrac)
        self.v.masked_fill_(self.s, self.reset)
```

```
        # Voltage clipping to lower bound.
        if self.lbound is not None:
            self.v.masked_fill_(self.v < self.lbound, self.lbound)

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()
        self.v.fill_(self.rest)  # Neuron voltages.
        self.i.zero_()  # Synaptic input currents.
        self.refrac_count.zero_()  # Refractory period counters.

    def compute_decays(self, dt) -> None:
        # language=rst
        """
        Sets the relevant decays.
        """
        super().compute_decays(dt=dt)
        self.decay = torch.exp(
            -self.dt / self.tc_decay
        )  # Neuron voltage decay (per timestep).
        self.i_decay = torch.exp(
            -self.dt / self.tc_i_decay
        )  # Synaptic input current decay (per timestep).

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
        super().set_batch_size(batch_size=batch_size)
        self.v = self.rest * torch.ones(batch_size, *self.shape, device=self.v.device)
        self.i = torch.zeros_like(self.v, device=self.i.device)
        self.refrac_count = torch.zeros_like(self.v, device=self.refrac_count.device)


class AdaptiveLIFNodes(Nodes):
    # language=rst
    """
    Layer of leaky integrate-and-fire (LIF) neurons with adaptive thresholds. A neuron's voltage threshold is increased
    by some constant each time it spikes; otherwise, it is decaying back to its default value.
    """

```

```
    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        rest: Union[float, torch.Tensor] = -65.0,
        reset: Union[float, torch.Tensor] = -65.0,
        thresh: Union[float, torch.Tensor] = -52.0,
        refrac: Union[int, torch.Tensor] = 5,
        tc_decay: Union[float, torch.Tensor] = 100.0,
        theta_plus: Union[float, torch.Tensor] = 0.05,
        tc_theta_decay: Union[float, torch.Tensor] = 1e7,
        lbound: float = None,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of LIF neurons with adaptive firing thresholds.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param rest: Resting membrane voltage.
        :param reset: Post-spike reset voltage.
        :param thresh: Spike threshold voltage.
        :param refrac: Refractory (non-firing) period of the neuron.
        :param tc_decay: Time constant of neuron voltage decay.
        :param theta_plus: Voltage increase of threshold after spiking.
        :param tc_theta_decay: Time constant of adaptive threshold decay.
        :param lbound: Lower bound of the voltage.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer("rest", torch.tensor(rest))  # Rest
```

```
voltage.
        self.register_buffer("reset", torch.tensor(reset))  # Post-spike reset voltage.
        self.register_buffer("thresh", torch.tensor(thresh))  # Spike threshold voltage.
        self.register_buffer(
            "refrac", torch.tensor(refrac)
        )  # Post-spike refractory period.
        self.register_buffer(
            "tc_decay", torch.tensor(tc_decay)
        )  # Time constant of neuron voltage decay.
        self.register_buffer(
            "decay", torch.empty_like(self.tc_decay)
        )  # Set in compute_decays.
        self.register_buffer(
            "theta_plus", torch.tensor(theta_plus)
        )  # Constant threshold increase on spike.
        self.register_buffer(
            "tc_theta_decay", torch.tensor(tc_theta_decay)
        )  # Time constant of adaptive threshold decay.
        self.register_buffer(
            "theta_decay", torch.empty_like(self.tc_theta_decay)
        )  # Set in compute_decays.

        self.register_buffer("v", torch.FloatTensor())  # Neuron voltages.
        self.register_buffer("theta", torch.zeros(*self.shape))  # Adaptive thresholds.
        self.register_buffer(
            "refrac_count", torch.FloatTensor()
        )  # Refractory period counters.
        self.lbound = lbound  # Lower bound of voltage.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        # Decay voltages and adaptive thresholds.
        self.v = self.decay * (self.v - self.rest) + self.rest
        if self.learning:
            self.theta *= self.theta_decay

        # Integrate inputs.
        self.v += (self.refrac_count == 0).float() * x

        # Decrement refractory counters.
        self.refrac_count = (self.refrac_count > 0).float() * (
            self.refrac_count - self.dt
        )

        # Check for spiking neurons.
```

```
        self.s = self.v >= self.thresh + self.theta

        # Refractoriness, voltage reset, and adaptive thresholds.
        self.refrac_count.masked_fill_(self.s, self.refrac)
        self.v.masked_fill_(self.s, self.reset)
        if self.learning:
            self.theta += self.theta_plus * self.s.float().sum(0)

        # voltage clipping to lowerbound
        if self.lbound is not None:
            self.v.masked_fill_(self.v < self.lbound, self.lbound)

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()
        self.v.fill_(self.rest)  # Neuron voltages.
        self.refrac_count.zero_()  # Refractory period counters.

    def compute_decays(self, dt) -> None:
        # language=rst
        """
        Sets the relevant decays.
        """
        super().compute_decays(dt=dt)
        self.decay = torch.exp(
            -self.dt / self.tc_decay
        )  # Neuron voltage decay (per timestep).
        self.theta_decay = torch.exp(
            -self.dt / self.tc_theta_decay
        )  # Adaptive threshold decay (per timestep).

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
        super().set_batch_size(batch_size=batch_size)
        self.v = self.rest * torch.ones(batch_size, *self.shape, device=self.v.device)
        self.refrac_count = torch.zeros_like(self.v, device=self.refrac_count.device)


class DiehlAndCookNodes(Nodes):
    # language=rst
    """
```

```
    Layer of leaky integrate-and-fire (LIF) neurons with adaptive thresholds (modified for Diehl & Cook 2015
    replication).
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        thresh: Union[float, torch.Tensor] = -52.0,
        rest: Union[float, torch.Tensor] = -65.0,
        reset: Union[float, torch.Tensor] = -65.0,
        refrac: Union[int, torch.Tensor] = 5,
        tc_decay: Union[float, torch.Tensor] = 100.0,
        theta_plus: Union[float, torch.Tensor] = 0.05,
        tc_theta_decay: Union[float, torch.Tensor] = 1e7,
        lbound: float = None,
        one_spike: bool = True,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of Diehl & Cook 2015 neurons.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param thresh: Spike threshold voltage.
        :param rest: Resting membrane voltage.
        :param reset: Post-spike reset voltage.
        :param refrac: Refractory (non-firing) period of the neuron.
        :param tc_decay: Time constant of neuron voltage decay.
        :param theta_plus: Voltage increase of threshold after spiking.
        :param tc_theta_decay: Time constant of adaptive threshold decay.
        :param lbound: Lower bound of the voltage.
        :param one_spike: Whether to allow only one spike per timestep.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
```

```
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer("rest", torch.tensor(rest))  # Rest voltage.
        self.register_buffer("reset", torch.tensor(reset))  # Post-spike reset voltage.
        self.register_buffer("thresh", torch.tensor(thresh))  # Spike threshold voltage.
        self.register_buffer(
            "refrac", torch.tensor(refrac)
        )  # Post-spike refractory period.
        self.register_buffer(
            "tc_decay", torch.tensor(tc_decay)
        )  # Time constant of neuron voltage decay.
        self.register_buffer(
            "decay", torch.empty_like(self.tc_decay)
        )  # Set in compute_decays.
        self.register_buffer(
            "theta_plus", torch.tensor(theta_plus)
        )  # Constant threshold increase on spike.
        self.register_buffer(
            "tc_theta_decay", torch.tensor(tc_theta_decay)
        )  # Time constant of adaptive threshold decay.
        self.register_buffer(
            "theta_decay", torch.empty_like(self.tc_theta_decay)
        )  # Set in compute_decays.
        self.register_buffer("v", torch.FloatTensor())  # Neuron voltages.
        self.register_buffer("theta", torch.zeros(*self.shape))  # Adaptive thresholds.
        self.register_buffer(
            "refrac_count", torch.FloatTensor()
        )  # Refractory period counters.

        self.lbound = lbound  # Lower bound of voltage.
        self.one_spike = one_spike  # One spike per timestep.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        # Decay voltages and adaptive thresholds.
        self.v = self.decay * (self.v - self.rest) + self.rest
        if self.learning:
            self.theta *= self.theta_decay

        # Integrate inputs.
```

```
        self.v += (self.refrac_count == 0).float() * x

        # Decrement refractory counters.
        self.refrac_count = (self.refrac_count > 0).float() * (
            self.refrac_count - self.dt
        )

        # Check for spiking neurons.
        self.s = self.v >= self.thresh + self.theta

        # Refractoriness, voltage reset, and adaptive thresholds.
        self.refrac_count.masked_fill_(self.s, self.refrac)
        self.v.masked_fill_(self.s, self.reset)
        if self.learning:
            self.theta += self.theta_plus * self.s.float().sum(0)

        # Choose only a single neuron to spike.
        if self.one_spike:
            if self.s.any():
                _any = self.s.view(self.batch_size, -1).any(1)
                ind = torch.multinomial(
                    self.s.float().view(self.batch_size, -1)[_any], 1
                )
                _any = _any.nonzero()
                self.s.zero_()
                self.s.view(self.batch_size, -1)[_any, ind] = 1

        # Voltage clipping to lower bound.
        if self.lbound is not None:
            self.v.masked_fill_(self.v < self.lbound, self.lbound)

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()
        self.v.fill_(self.rest)  # Neuron voltages.
        self.refrac_count.zero_()  # Refractory period counters.

    def compute_decays(self, dt) -> None:
        # language=rst
        """
        Sets the relevant decays.
        """
        super().compute_decays(dt=dt)
        self.decay = torch.exp(
            -self.dt / self.tc_decay
        )  # Neuron voltage decay (per timestep).
        self.theta_decay = torch.exp(
            -self.dt / self.tc_theta_decay
```

```
        )  # Adaptive threshold decay (per timestep).

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
        super().set_batch_size(batch_size=batch_size)
        self.v = self.rest * torch.ones(batch_size, *self.shape, device=self.v.device)
        self.refrac_count = torch.zeros_like(self.v, device=self.refrac_count.device)


class IzhikevichNodes(Nodes):
    # language=rst
    """
    Layer of Izhikevich neurons.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        excitatory: float = 1,
        thresh: Union[float, torch.Tensor] = 45.0,
        rest: Union[float, torch.Tensor] = -65.0,
        lbound: float = None,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of Izhikevich neurons.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param excitatory: Percent of excitatory (vs. inhibitory) neurons in the layer; in range ``[0, 1]``.
        :param thresh: Spike threshold voltage.
        :param rest: Resting membrane voltage.
```

```
        :param lbound: Lower bound of the voltage.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer("rest", torch.tensor(rest))  # Rest
    voltage.
        self.register_buffer("thresh", torch.tensor(thresh))  #
    Spike threshold voltage.
        self.lbound = lbound

        self.register_buffer("r", None)
        self.register_buffer("a", None)
        self.register_buffer("b", None)
        self.register_buffer("c", None)
        self.register_buffer("d", None)
        self.register_buffer("S", None)
        self.register_buffer("excitatory", None)

        if excitatory > 1:
            excitatory = 1
        elif excitatory < 0:
            excitatory = 0

        if excitatory == 1:
            self.r = torch.rand(n)
            self.a = 0.02 * torch.ones(n)
            self.b = 0.2 * torch.ones(n)
            self.c = -65.0 + 15 * (self.r ** 2)
            self.d = 8 - 6 * (self.r ** 2)
            self.S = 0.5 * torch.rand(n, n)
            self.excitatory = torch.ones(n).byte()

        elif excitatory == 0:
            self.r = torch.rand(n)
            self.a = 0.02 + 0.08 * self.r
            self.b = 0.25 - 0.05 * self.r
            self.c = -65.0 * torch.ones(n)
            self.d = 2 * torch.ones(n)
            self.S = -torch.rand(n, n)

            self.excitatory = torch.zeros(n).byte()

        else:
            self.excitatory = torch.zeros(n).byte()

            ex = int(n * excitatory)
```

```
            inh = n - ex

            # init
            self.r = torch.zeros(n)
            self.a = torch.zeros(n)
            self.b = torch.zeros(n)
            self.c = torch.zeros(n)
            self.d = torch.zeros(n)
            self.S = torch.zeros(n, n)

            # excitatory
            self.r[:ex] = torch.rand(ex)
            self.a[:ex] = 0.02 * torch.ones(ex)
            self.b[:ex] = 0.2 * torch.ones(ex)
            self.c[:ex] = -65.0 + 15 * self.r[:ex] ** 2
            self.d[:ex] = 8 - 6 * self.r[:ex] ** 2
            self.S[:, :ex] = 0.5 * torch.rand(n, ex)
            self.excitatory[:ex] = 1

            # inhibitory
            self.r[ex:] = torch.rand(inh)
            self.a[ex:] = 0.02 + 0.08 * self.r[ex:]
            self.b[ex:] = 0.25 - 0.05 * self.r[ex:]
            self.c[ex:] = -65.0 * torch.ones(inh)
            self.d[ex:] = 2 * torch.ones(inh)
            self.S[:, ex:] = -torch.rand(n, inh)
            self.excitatory[ex:] = 0

        self.register_buffer("v", self.rest * torch.ones(n))  # Neuron voltages.
        self.register_buffer("u", self.b * self.v)  # Neuron recovery.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        # Check for spiking neurons.
        self.s = self.v >= self.thresh

        # Voltage and recovery reset.
        self.v = torch.where(self.s, self.c, self.v)
        self.u = torch.where(self.s, self.u + self.d, self.u)

        # Add inter-columnar input.
        if self.s.any():
            x += torch.cat(
                [self.S[:, self.s[i]].sum(dim=1)[None] for i in range(self.s.shape[0])],
                dim=0,
            )
```

```
        # Apply v and u updates.
        self.v += self.dt * 0.5 * (0.04 * self.v ** 2 + 5 * self.v + 140 - self.u + x)
        self.v += self.dt * 0.5 * (0.04 * self.v ** 2 + 5 * self.v + 140 - self.u + x)
        self.u += self.dt * self.a * (self.b * self.v - self.u)

        # Voltage clipping to lower bound.
        if self.lbound is not None:
            self.v.masked_fill_(self.v < self.lbound, self.lbound)

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()
        self.v.fill_(self.rest)  # Neuron voltages.
        self.u = self.b * self.v  # Neuron recovery.

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
        super().set_batch_size(batch_size=batch_size)
        self.v = self.rest * torch.ones(batch_size, *self.shape, device=self.v.device)
        self.u = self.b * self.v


class SRM0Nodes(Nodes):
    # language=rst
    """
    Layer of simplified spike response model (SRM0) neurons with stochastic threshold (escape noise). Adapted from
    `(Vasilaki et al., 2009) <https://intranet.physio.unibe.ch/Publikationen/Dokumente/Vasilaki2009PloSComputBio_1.pdf>`_.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
```

```
        sum_input: bool = False,
        thresh: Union[float, torch.Tensor] = -50.0,
        rest: Union[float, torch.Tensor] = -70.0,
        reset: Union[float, torch.Tensor] = -70.0,
        refrac: Union[int, torch.Tensor] = 5,
        tc_decay: Union[float, torch.Tensor] = 10.0,
        lbound: float = None,
        eps_0: Union[float, torch.Tensor] = 1.0,
        rho_0: Union[float, torch.Tensor] = 1.0,
        d_thresh: Union[float, torch.Tensor] = 5.0,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of SRM0 neurons.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param thresh: Spike threshold voltage.
        :param rest: Resting membrane voltage.
        :param reset: Post-spike reset voltage.
        :param refrac: Refractory (non-firing) period of the neuron.
        :param tc_decay: Time constant of neuron voltage decay.
        :param lbound: Lower bound of the voltage.
        :param eps_0: Scaling factor for pre-synaptic spike contributions.
        :param rho_0: Stochastic intensity at threshold.
        :param d_thresh: Width of the threshold region.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer("rest", torch.tensor(rest))  # Rest voltage.
        self.register_buffer("reset", torch.tensor(reset))  # Post-spike reset voltage.
        self.register_buffer("thresh", torch.tensor(thresh))  # Spike threshold voltage.
        self.register_buffer(
            "refrac", torch.tensor(refrac)
        )  # Post-spike refractory period.
```

```
        self.register_buffer(
            "tc_decay", torch.tensor(tc_decay)
        )  # Time constant of neuron voltage decay.
        self.register_buffer("decay", torch.tensor(tc_decay))  # Set in compute_decays.
        self.register_buffer(
            "eps_0", torch.tensor(eps_0)
        )  # Scaling factor for pre-synaptic spike contributions.
        self.register_buffer(
            "rho_0", torch.tensor(rho_0)
        )  # Stochastic intensity at threshold.
        self.register_buffer(
            "d_thresh", torch.tensor(d_thresh)
        )  # Width of the threshold region.
        self.register_buffer("v", torch.FloatTensor())  # Neuron voltages.
        self.register_buffer(
            "refrac_count", torch.FloatTensor()
        )  # Refractory period counters.

        self.lbound = lbound  # Lower bound of voltage.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        # Decay voltages.
        self.v = self.decay * (self.v - self.rest) + self.rest

        # Integrate inputs.
        self.v += (self.refrac_count == 0).float() * self.eps_0 * x

        # Compute (instantaneous) probabilities of spiking, clamp between 0 and 1 using exponentials.
        # Also known as 'escape noise', this simulates nearby neurons.
        self.rho = self.rho_0 * torch.exp((self.v - self.thresh) / self.d_thresh)
        self.s_prob = 1.0 - torch.exp(-self.rho * self.dt)

        # Decrement refractory counters.
        self.refrac_count = (self.refrac_count > 0).float() * (
            self.refrac_count - self.dt
        )

        # Check for spiking neurons (spike when probability > some random number).
        self.s = torch.rand_like(self.s_prob) < self.s_prob

        # Refractoriness and voltage reset.
        self.refrac_count.masked_fill_(self.s, self.refrac)
```

```
        self.v.masked_fill_(self.s, self.reset)

        # Voltage clipping to lower bound.
        if self.lbound is not None:
            self.v.masked_fill_(self.v < self.lbound, self.lbound)

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()
        self.v.fill_(self.rest)  # Neuron voltages.
        self.refrac_count.zero_()  # Refractory period counters.

    def compute_decays(self, dt) -> None:
        # language=rst
        """
        Sets the relevant decays.
        """
        super().compute_decays(dt=dt)
        self.decay = torch.exp(
            -self.dt / self.tc_decay
        )  # Neuron voltage decay (per timestep).

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a
    network.

        :param batch_size: Mini-batch size.
        """
        super().set_batch_size(batch_size=batch_size)
        self.v = self.rest * torch.ones(batch_size, *self.shape,
    device=self.v.device)
        self.refrac_count = torch.zeros_like(self.v, device=self.
    refrac_count.device)
```

Listing B.7: *Nodes*

```
from abc import ABC, abstractmethod
from typing import Union, Tuple, Optional, Sequence

import numpy as np
import torch
from torch.nn import Module, Parameter
import torch.nn.functional as F
from torch.nn.modules.utils import _pair

from .nodes import Nodes

```

```python

class AbstractConnection(ABC, Module):
    # language=rst
    """
    Abstract base method for connections between ``Nodes``.
    """

    def __init__(
        self,
        source: Nodes,
        target: Nodes,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Constructor for abstract base class for connection objects.

        :param source: A layer of nodes from which the connection originates.
        :param target: A layer of nodes to which the connection connects.
        :param nu: Learning rate for both pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:

        :param LearningRule update_rule: Modifies connection parameters according to
            some rule.
        :param float wmin: The minimum value on the connection weights.
        :param float wmax: The maximum value on the connection weights.
        :param float norm: Total weight per target neuron normalization.
        """
        super().__init__()

        assert isinstance(source, Nodes), "Source is not a Nodes object"
        assert isinstance(target, Nodes), "Target is not a Nodes object"

        self.source = source
        self.target = target

```

```
        self.nu = nu
        self.weight_decay = weight_decay
        self.reduction = reduction

        from ..learning import NoOp

        self.update_rule = kwargs.get("update_rule", NoOp)
        self.wmin = kwargs.get("wmin", -np.inf)
        self.wmax = kwargs.get("wmax", np.inf)
        self.norm = kwargs.get("norm", None)
        self.decay = kwargs.get("decay", None)

        if self.update_rule is None:
            self.update_rule = NoOp

        self.update_rule = self.update_rule(
            connection=self,
            nu=nu,
            reduction=reduction,
            weight_decay=weight_decay,
            **kwargs
        )

    @abstractmethod
    def compute(self, s: torch.Tensor) -> None:
        # language=rst
        """
        Compute pre-activations of downstream neurons given spikes
 of upstream neurons.

        :param s: Incoming spikes.
        """
        pass

    @abstractmethod
    def update(self, **kwargs) -> None:
        # language=rst
        """
        Compute connection's update rule.

        Keyword arguments:

        :param bool learning: Whether to allow connection updates.
        :param ByteTensor mask: Boolean mask determining which
 weights to clamp to zero.
        """
        learning = kwargs.get("learning", True)

        if learning:
            self.update_rule.update(**kwargs)

        mask = kwargs.get("mask", None)
        if mask is not None:
            self.w.masked_fill_(mask, 0)
```

```python

    @abstractmethod
    def reset_state_variables(self) -> None:
        # language=rst
        """
        Contains resetting logic for the connection.
        """
        pass


class Connection(AbstractConnection):
    # language=rst
    """
    Specifies synapses between one or two populations of neurons.
    """

    def __init__(
        self,
        source: Nodes,
        target: Nodes,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Instantiates a :code:`Connection` object.

        :param source: A layer of nodes from which the connection originates.
        :param target: A layer of nodes to which the connection connects.
        :param nu: Learning rate for both pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:

        :param LearningRule update_rule: Modifies connection parameters according to
            some rule.
        :param torch.Tensor w: Strengths of synapses.
        :param torch.Tensor b: Target population bias.
        :param float wmin: Minimum allowed value on the connection weights.
        :param float wmax: Maximum allowed value on the connection weights.
        :param float norm: Total weight per target neuron normalization constant.
```

```python
        """
        super().__init__(source, target, nu, reduction, weight_decay, **kwargs)

        w = kwargs.get("w", None)
        if w is None:
            if self.wmin == -np.inf or self.wmax == np.inf:
                w = torch.clamp(torch.rand(source.n, target.n), self.wmin, self.wmax)
            else:
                w = self.wmin + torch.rand(source.n, target.n) * (self.wmax - self.wmin)
        else:
            if self.wmin != -np.inf or self.wmax != np.inf:
                w = torch.clamp(w, self.wmin, self.wmax)

        self.w = Parameter(w, requires_grad=False)
        self.b = Parameter(kwargs.get("b", torch.zeros(target.n)), requires_grad=False)

    def compute(self, s: torch.Tensor) -> torch.Tensor:
        # language=rst
        """
        Compute pre-activations given spikes using connection weights.

        :param s: Incoming spikes.
        :return: Incoming spikes multiplied by synaptic weights (with or without
                 decaying spike activation).
        """
        # Compute multiplication of spike activations by weights and add bias.
        post = s.float().view(s.size(0), -1) @ self.w + self.b
        return post.view(s.size(0), *self.target.shape)

    def update(self, **kwargs) -> None:
        # language=rst
        """
        Compute connection's update rule.
        """
        super().update(**kwargs)

    def normalize(self) -> None:
        # language=rst
        """
        Normalize weights so each target neuron has sum of connection weights equal to
        ``self.norm``.
        """
        if self.norm is not None:
            w_abs_sum = self.w.abs().sum(0).unsqueeze(0)
            w_abs_sum[w_abs_sum == 0] = 1.0
            self.w *= self.norm / w_abs_sum
```

```python

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Contains resetting logic for the connection.
        """
        super().reset_state_variables()


class Conv2dConnection(AbstractConnection):
    # language=rst
    """
    Specifies convolutional synapses between one or two populations of neurons.
    """

    def __init__(
        self,
        source: Nodes,
        target: Nodes,
        kernel_size: Union[int, Tuple[int, int]],
        stride: Union[int, Tuple[int, int]] = 1,
        padding: Union[int, Tuple[int, int]] = 0,
        dilation: Union[int, Tuple[int, int]] = 1,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Instantiates a ``Conv2dConnection`` object.

        :param source: A layer of nodes from which the connection originates.
        :param target: A layer of nodes to which the connection connects.
        :param kernel_size: Horizontal and vertical size of convolutional kernels.
        :param stride: Horizontal and vertical stride for convolution.
        :param padding: Horizontal and vertical padding for convolution.
        :param dilation: Horizontal and vertical dilation for convolution.
        :param nu: Learning rate for both pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:
```

```python

        :param LearningRule update_rule: Modifies connection parameters according to
            some rule.
        :param torch.Tensor w: Strengths of synapses.
        :param torch.Tensor b: Target population bias.
        :param float wmin: Minimum allowed value on the connection weights.
        :param float wmax: Maximum allowed value on the connection weights.
        :param float norm: Total weight per target neuron normalization constant.
        """
        super().__init__(source, target, nu, reduction, weight_decay, **kwargs)

        self.kernel_size = _pair(kernel_size)
        self.stride = _pair(stride)
        self.padding = _pair(padding)
        self.dilation = _pair(dilation)

        self.in_channels, input_height, input_width = (
            source.shape[0],
            source.shape[1],
            source.shape[2],
        )
        self.out_channels, output_height, output_width = (
            target.shape[0],
            target.shape[1],
            target.shape[2],
        )

        width = (
            input_height - self.kernel_size[0] + 2 * self.padding[0]
        ) / self.stride[0] + 1
        height = (
            input_width - self.kernel_size[1] + 2 * self.padding[1]
        ) / self.stride[1] + 1
        shape = (self.in_channels, self.out_channels, int(width), int(height))

        error = (
            "Target dimensionality must be (out_channels, ?,"
            "(input_height - filter_height + 2 * padding_height) / stride_height + 1,"
            "(input_width - filter_width + 2 * padding_width) / stride_width + 1"
        )

        assert (
            target.shape[0] == shape[1]
            and target.shape[1] == shape[2]
            and target.shape[2] == shape[3]
        ), error
```

```python

        w = kwargs.get("w", None)
        if w is None:
            if self.wmin == -np.inf or self.wmax == np.inf:
                w = torch.clamp(
                    torch.rand(self.out_channels, self.in_channels, *self.kernel_size),
                    self.wmin,
                    self.wmax,
                )
            else:
                w = (self.wmax - self.wmin) * torch.rand(
                    self.out_channels, self.in_channels, *self.kernel_size
                )
                w += self.wmin
        else:
            if self.wmin != -np.inf or self.wmax != np.inf:
                w = torch.clamp(w, self.wmin, self.wmax)

        self.w = Parameter(w, requires_grad=False)
        self.b = Parameter(
            kwargs.get("b", torch.zeros(self.out_channels)), requires_grad=False
        )

    def compute(self, s: torch.Tensor) -> torch.Tensor:
        # language=rst
        """
        Compute convolutional pre-activations given spikes using layer weights.

        :param s: Incoming spikes.
        :return: Incoming spikes multiplied by synaptic weights (with or without
            decaying spike activation).
        """
        return F.conv2d(
            s.float(),
            self.w,
            self.b,
            stride=self.stride,
            padding=self.padding,
            dilation=self.dilation,
        )

    def update(self, **kwargs) -> None:
        # language=rst
        """
        Compute connection's update rule.
        """
        super().update(**kwargs)

    def normalize(self) -> None:
```

```
        # language=rst
        """
        Normalize weights along the first axis according to total weight per target
        neuron.
        """
        if self.norm is not None:
            # get a view and modify in place
            w = self.w.view(
                self.w.size(0) * self.w.size(1), self.w.size(2) * self.w.size(3)
            )

            for fltr in range(w.size(0)):
                w[fltr] *= self.norm / w[fltr].sum(0)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Contains resetting logic for the connection.
        """
        super().reset_state_variables()


class MaxPool2dConnection(AbstractConnection):
    # language=rst
    """
    Specifies max-pooling synapses between one or two populations of neurons by keeping
    online estimates of maximally firing neurons.
    """

    def __init__(
        self,
        source: Nodes,
        target: Nodes,
        kernel_size: Union[int, Tuple[int, int]],
        stride: Union[int, Tuple[int, int]] = 1,
        padding: Union[int, Tuple[int, int]] = 0,
        dilation: Union[int, Tuple[int, int]] = 1,
        **kwargs
    ) -> None:
        # language=rst
        """
        Instantiates a ``MaxPool2dConnection`` object.

        :param source: A layer of nodes from which the connection originates.
        :param target: A layer of nodes to which the connection connects.
        :param kernel_size: Horizontal and vertical size of convolutional kernels.
        :param stride: Horizontal and vertical stride for convolution.
```

```
        :param padding: Horizontal and vertical padding for convolution.
        :param dilation: Horizontal and vertical dilation for convolution.

        Keyword arguments:

        :param decay: Decay rate of online estimates of average firing activity.
        """
        super().__init__(source, target, None, None, 0.0, **kwargs)

        self.kernel_size = _pair(kernel_size)
        self.stride = _pair(stride)
        self.padding = _pair(padding)
        self.dilation = _pair(dilation)

        self.register_buffer("firing_rates", torch.ones(source.shape))

    def compute(self, s: torch.Tensor) -> torch.Tensor:
        # language=rst
        """
        Compute max-pool pre-activations given spikes using online firing rate
        estimates.

        :param s: Incoming spikes.
        :return: Incoming spikes multiplied by synaptic weights (with or without
            decaying spike activation).
        """
        self.firing_rates -= self.decay * self.firing_rates
        self.firing_rates += s.float()

        _, indices = F.max_pool2d(
            self.firing_rates,
            kernel_size=self.kernel_size,
            stride=self.stride,
            padding=self.padding,
            dilation=self.dilation,
            return_indices=True,
        )

        return s.take(indices).float()

    def update(self, **kwargs) -> None:
        # language=rst
        """
        Compute connection's update rule.
        """
        super().update(**kwargs)

    def normalize(self) -> None:
```

```
        # language=rst
        """
        No weights -> no normalization.
        """
        pass

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Contains resetting logic for the connection.
        """
        super().reset_state_variables()

        self.firing_rates = torch.zeros(self.source.shape)

class LocalConnection(AbstractConnection):
    # language=rst
    """
    Specifies a locally connected connection between one or two populations of neurons.
    """

    def __init__(
        self,
        source: Nodes,
        target: Nodes,
        kernel_size: Union[int, Tuple[int, int]],
        stride: Union[int, Tuple[int, int]],
        n_filters: int,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Instantiates a ``LocalConnection`` object. Source population should be
        two-dimensional.

        Neurons in the post-synaptic population are ordered by receptive field; that is,
        if there are ``n_conv`` neurons in each post-synaptic patch, then the first
        ``n_conv`` neurons in the post-synaptic population correspond to the first
        receptive field, the second ``n_conv`` to the second receptive field, and so on.

        :param source: A layer of nodes from which the connection originates.
        :param target: A layer of nodes to which the connection connects.
        :param kernel_size: Horizontal and vertical size of
```

```
convolutional kernels.
        :param stride: Horizontal and vertical stride for convolution.
        :param n_filters: Number of locally connected filters per pre-synaptic region.
        :param nu: Learning rate for both pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:

        :param LearningRule update_rule: Modifies connection parameters according to
            some rule.
        :param torch.Tensor w: Strengths of synapses.
        :param torch.Tensor b: Target population bias.
        :param float wmin: Minimum allowed value on the connection weights.
        :param float wmax: Maximum allowed value on the connection weights.
        :param float norm: Total weight per target neuron normalization constant.
        :param Tuple[int, int] input_shape: Shape of input population if it's not
            ``[sqrt, sqrt]``.
        """
        super().__init__(source, target, nu, reduction, weight_decay, **kwargs)

        kernel_size = _pair(kernel_size)
        stride = _pair(stride)

        self.kernel_size = kernel_size
        self.stride = stride
        self.n_filters = n_filters

        shape = kwargs.get("input_shape", None)
        if shape is None:
            sqrt = int(np.sqrt(source.n))
            shape = _pair(sqrt)

        if kernel_size == shape:
            conv_size = [1, 1]
        else:
            conv_size = (
                int((shape[0] - kernel_size[0]) / stride[0]) + 1,
                int((shape[1] - kernel_size[1]) / stride[1]) + 1,
            )

        self.conv_size = conv_size
```

```python

        conv_prod = int(np.prod(conv_size))
        kernel_prod = int(np.prod(kernel_size))

        assert (
            target.n == n_filters * conv_prod
        ), "Target layer size must be n_filters * (kernel_size ** 2)."

        locations = torch.zeros(
            kernel_size[0], kernel_size[1], conv_size[0], conv_size[1]
        ).long()
        for c1 in range(conv_size[0]):
            for c2 in range(conv_size[1]):
                for k1 in range(kernel_size[0]):
                    for k2 in range(kernel_size[1]):
                        location = (
                            c1 * stride[0] * shape[1]
                            + c2 * stride[1]
                            + k1 * shape[0]
                            + k2
                        )
                        locations[k1, k2, c1, c2] = location

        self.register_buffer("locations", locations.view(kernel_prod, conv_prod))
        w = kwargs.get("w", None)

        if w is None:
            w = torch.zeros(source.n, target.n)
            for f in range(n_filters):
                for c in range(conv_prod):
                    for k in range(kernel_prod):
                        if self.wmin == -np.inf or self.wmax == np.inf:
                            w[self.locations[k, c], f * conv_prod + c] = np.clip(
                                np.random.rand(), self.wmin, self.wmax
                            )
                        else:
                            w[
                                self.locations[k, c], f * conv_prod + c
                            ] = self.wmin + np.random.rand() * (self.wmax - self.wmin)
        else:
            if self.wmin != -np.inf or self.wmax != np.inf:
                w = torch.clamp(w, self.wmin, self.wmax)

        self.w = Parameter(w, requires_grad=False)

        self.register_buffer("mask", self.w == 0)
```

```python

        self.b = Parameter(kwargs.get("b", torch.zeros(target.n)), requires_grad=False)

        if self.norm is not None:
            self.norm *= kernel_prod

    def compute(self, s: torch.Tensor) -> torch.Tensor:
        # language=rst
        """
        Compute pre-activations given spikes using layer weights.

        :param s: Incoming spikes.
        :return: Incoming spikes multiplied by synaptic weights (with or without
            decaying spike activation).
        """
        # Compute multiplication of pre-activations by connection weights.
        if self.w.shape[0] == self.source.n and self.w.shape[1] == self.target.n:
            return s.float().view(s.size(0), -1) @ self.w + self.b
        else:
            a_post = (
                s.float().view(s.size(0), -1)
                @ self.w.view(self.source.n, self.target.n)
                + self.b
            )
            return a_post.view(*self.target.shape)

    def update(self, **kwargs) -> None:
        # language=rst
        """
        Compute connection's update rule.

        Keyword arguments:

        :param ByteTensor mask: Boolean mask determining which weights to clamp to zero.
        """
        if kwargs["mask"] is None:
            kwargs["mask"] = self.mask

        super().update(**kwargs)

    def normalize(self) -> None:
        # language=rst
        """
        Normalize weights so each target neuron has sum of connection weights equal to
        ``self.norm``.
        """
        if self.norm is not None:
            w = self.w.view(self.source.n, self.target.n)
```

```
            w *= self.norm / self.w.sum(0).view(1, -1)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Contains resetting logic for the connection.
        """
        super().reset_state_variables()


class MeanFieldConnection(AbstractConnection):
    # language=rst
    """
    A connection between one or two populations of neurons which computes a summary of
    the pre-synaptic population to use as weighted input to the post-synaptic
    population.
    """

    def __init__(
        self,
        source: Nodes,
        target: Nodes,
        nu: Optional[Union[float, Sequence[float]]] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Instantiates a :code:`MeanFieldConnection` object.
        :param source: A layer of nodes from which the connection originates.
        :param target: A layer of nodes to which the connection connects.
        :param nu: Learning rate for both pre- and post-synaptic events.
        :param weight_decay: Constant multiple to decay weights by on each iteration.
        Keyword arguments:
        :param LearningRule update_rule: Modifies connection parameters according to
            some rule.
        :param torch.Tensor w: Strengths of synapses.
        :param float wmin: Minimum allowed value on the connection weights.
        :param float wmax: Maximum allowed value on the connection weights.
        :param float norm: Total weight per target neuron normalization constant.
        """
        super().__init__(source, target, nu, weight_decay, **kwargs)

        w = kwargs.get("w", None)
```

```python
        if w is None:
            if self.wmin == -np.inf or self.wmax == np.inf:
                w = torch.clamp((torch.randn(1)[0] + 1) / 10, self.wmin, self.wmax)
            else:
                w = self.wmin + ((torch.randn(1)[0] + 1) / 10) * (self.wmax - self.wmin)
        else:
            if self.wmin != -np.inf or self.wmax != np.inf:
                w = torch.clamp(w, self.wmin, self.wmax)

        self.w = Parameter(w, requires_grad=False)

    def compute(self, s: torch.Tensor) -> torch.Tensor:
        # language=rst
        """
        Compute pre-activations given spikes using layer weights.
        :param s: Incoming spikes.
        :return: Incoming spikes multiplied by synaptic weights (with or without
            decaying spike activation).
        """
        # Compute multiplication of mean-field pre-activation by connection weights.
        return s.float().mean() * self.w

    def update(self, **kwargs) -> None:
        # language=rst
        """
        Compute connection's update rule.
        """
        super().update(**kwargs)

    def normalize(self) -> None:
        # language=rst
        """
        Normalize weights so each target neuron has sum of connection weights equal to
        ``self.norm``.
        """
        if self.norm is not None:
            self.w = self.w.view(1, self.target.n)
            self.w *= self.norm / self.w.sum()
            self.w = self.w.view(1, *self.target.shape)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Contains resetting logic for the connection.
        """
        super().reset_state_variables()


class SparseConnection(AbstractConnection):
```

```
    # language=rst
    """
    Specifies sparse synapses between one or two populations of neurons.
    """

    def __init__(
        self,
        source: Nodes,
        target: Nodes,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = None,
        **kwargs
    ) -> None:
        # language=rst
        """
        Instantiates a :code:`Connection` object with sparse weights.

        :param source: A layer of nodes from which the connection originates.
        :param target: A layer of nodes to which the connection connects.
        :param nu: Learning rate for both pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:

        :param torch.Tensor w: Strengths of synapses.
        :param float sparsity: Fraction of sparse connections to use.
        :param LearningRule update_rule: Modifies connection parameters according to
            some rule.
        :param float wmin: Minimum allowed value on the connection weights.
        :param float wmax: Maximum allowed value on the connection weights.
        :param float norm: Total weight per target neuron normalization constant.
        """
        super().__init__(source, target, nu, reduction, weight_decay, **kwargs)

        w = kwargs.get("w", None)
        self.sparsity = kwargs.get("sparsity", None)

        assert (
```

```
                w is not None
                and self.sparsity is None
                or w is None
                and self.sparsity is not None
            ), 'Only one of "weights" or "sparsity" must be specified'

        if w is None and self.sparsity is not None:
            i = torch.bernoulli(
                1 - self.sparsity * torch.ones(*source.shape, *target.shape)
            )
            if self.wmin == -np.inf or self.wmax == np.inf:
                v = torch.clamp(
                    torch.rand(*source.shape, *target.shape)[i.byte()],
                    self.wmin,
                    self.wmax,
                )
            else:
                v = self.wmin + torch.rand(*source.shape, *target.shape)[i.byte()] * (
                    self.wmax - self.wmin
                )
            w = torch.sparse.FloatTensor(i.nonzero().t(), v)
        elif w is not None and self.sparsity is None:
            assert w.is_sparse, "Weight matrix is not sparse (see torch.sparse module)"
            if self.wmin != -np.inf or self.wmax != np.inf:
                w = torch.clamp(w, self.wmin, self.wmax)

        self.w = Parameter(w, requires_grad=False)

    def compute(self, s: torch.Tensor) -> torch.Tensor:
        # language=rst
        """
        Compute convolutional pre-activations given spikes using layer weights.

        :param s: Incoming spikes.
        :return: Incoming spikes multiplied by synaptic weights (with or without
            decaying spike activation).
        """
        return torch.mm(self.w, s.unsqueeze(-1).float()).squeeze(-1)

    def update(self, **kwargs) -> None:
        # language=rst
        """
        Compute connection's update rule.
        """
        pass

    def normalize(self) -> None:
        # language=rst
```

```
        """
        Normalize weights along the first axis according to total
    weight per target
        neuron.
        """
        pass

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Contains resetting logic for the connection.
        """
        super().reset_state_variables()
```

Listing B.8: *Topology*

```
import os
import torch
import numpy as np

from abc import ABC
from typing import Union, Optional, Iterable, Dict

from .nodes import Nodes
from .topology import AbstractConnection


class AbstractMonitor(ABC):
    # language=rst
    """
    Abstract base class for state variable monitors.
    """


class Monitor(AbstractMonitor):
    # language=rst
    """
    Records state variables of interest.
    """

    def __init__(
        self,
        obj: Union[Nodes, AbstractConnection],
        state_vars: Iterable[str],
        time: Optional[int] = None,
        batch_size: int = 1,
    ):
        # language=rst
        """
        Constructs a ``Monitor`` object.

        :param obj: An object to record state variables from during
    network simulation.
        :param state_vars: Iterable of strings indicating names of
```

```
    state variables to
            record.
        :param time: If not ``None``, pre-allocate memory for state variable recording.
        """
        super().__init__()

        self.obj = obj
        self.state_vars = state_vars
        self.time = time
        self.batch_size = batch_size

        # Deal with time later, the same underlying list is used
        self.recording = {v: [] for v in self.state_vars}

    def get(self, var: str) -> torch.Tensor:
        # language=rst
        """
        Return recording to user.

        :param var: State variable recording to return.
        :return: Tensor of shape ``[time, n_1, ..., n_k]``, where ``[n_1, ..., n_k]`` is
            the shape of the recorded state variable.
        """
        return torch.cat(self.recording[var], 0)

    def record(self) -> None:
        # language=rst
        """
        Appends the current value of the recorded state variables to the recording.
        """
        for v in self.state_vars:
            data = getattr(self.obj, v).unsqueeze(0)
            self.recording[v].append(data.detach().clone())

        # remove the oldest element (first in the list)
        if self.time is not None:
            for v in self.state_vars:
                if len(self.recording[v]) > self.time:
                    self.recording[v].pop(0)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets recordings to empty ``torch.Tensor``s.
        """
        self.recording = {v: [] for v in self.state_vars}


class NetworkMonitor(AbstractMonitor):
    # language=rst
    """
```

```
    Record state variables of all layers and connections.
    """

    def __init__(
        self,
        network: "Network",
        layers: Optional[Iterable[str]] = None,
        connections: Optional[Iterable[str]] = None,
        state_vars: Optional[Iterable[str]] = None,
        time: Optional[int] = None,
    ):
        # language=rst
        """
        Constructs a ``NetworkMonitor`` object.

        :param network: Network to record state variables from.
        :param layers: Layers to record state variables from.
        :param connections: Connections to record state variables from.
        :param state_vars: List of strings indicating names of state variables to
            record.
        :param time: If not ``None``, pre-allocate memory for state variable recording.
        """
        super().__init__()

        self.network = network
        self.layers = layers if layers is not None else list(self.network.layers.keys())
        self.connections = (
            connections
            if connections is not None
            else list(self.network.connections.keys())
        )
        self.state_vars = state_vars if state_vars is not None else ("v", "s", "w")
        self.time = time

        if self.time is not None:
            self.i = 0

        # Initialize empty recording.
        self.recording = {k: {} for k in self.layers + self.connections}

        # If no simulation time is specified, specify 0-dimensional recordings.
        if self.time is None:
            for v in self.state_vars:
                for l in self.layers:
                    if hasattr(self.network.layers[l], v):
                        self.recording[l][v] = torch.Tensor()

```

```
                for c in self.connections:
                    if hasattr(self.network.connections[c], v):
                        self.recording[c][v] = torch.Tensor()

        # If simulation time is specified, pre-allocate recordings in memory for speed.
        else:
            for v in self.state_vars:
                for l in self.layers:
                    if hasattr(self.network.layers[l], v):
                        self.recording[l][v] = torch.zeros(
                            self.time, *getattr(self.network.layers[l], v).size()
                        )

                for c in self.connections:
                    if hasattr(self.network.connections[c], v):
                        self.recording[c][v] = torch.zeros(
                            self.time, *getattr(self.network.connections[c], v).size()
                        )

    def get(self) -> Dict[str, Dict[str, Union[Nodes, AbstractConnection]]]:
        # language=rst
        """
        Return entire recording to user.

        :return: Dictionary of dictionary of all layers' and connections' recorded
            state variables.
        """
        return self.recording

    def record(self) -> None:
        # language=rst
        """
        Appends the current value of the recorded state variables to the recording.
        """
        if self.time is None:
            for v in self.state_vars:
                for l in self.layers:
                    if hasattr(self.network.layers[l], v):
                        data = getattr(self.network.layers[l], v).unsqueeze(0).float()
                        self.recording[l][v] = torch.cat(
                            (self.recording[l][v], data), 0
                        )

                for c in self.connections:
                    if hasattr(self.network.connections[c], v):
                        data = getattr(self.network.connections[c], v).unsqueeze(0)
```

```
                        self.recording[c][v] = torch.cat(
                            (self.recording[c][v], data), 0
                        )

        else:
            for v in self.state_vars:
                for l in self.layers:
                    if hasattr(self.network.layers[l], v):
                        data = getattr(self.network.layers[l], v).float().unsqueeze(0)
                        self.recording[l][v] = torch.cat(
                            (self.recording[l][v][1:].type(data.type()), data), 0
                        )

                for c in self.connections:
                    if hasattr(self.network.connections[c], v):
                        data = getattr(self.network.connections[c], v).unsqueeze(0)
                        self.recording[c][v] = torch.cat(
                            (self.recording[c][v][1:].type(data.type()), data), 0
                        )

            self.i += 1

    def save(self, path: str, fmt: str = "npz") -> None:
        # language=rst
        """
        Write the recording dictionary out to file.

        :param path: The directory to which to write the monitor's recording.
        :param fmt: Type of file to write to disk. One of ``"pickle"`` or ``"npz"``.
        """
        if not os.path.exists(os.path.dirname(path)):
            os.makedirs(os.path.dirname(path))

        if fmt == "npz":
            # Build a list of arrays to write to disk.
            arrays = {}
            for o in self.recording:
                if type(o) == tuple:
                    arrays.update(
                        {
                            "_".join(["-".join(o), v]): self.recording[o][v]
                            for v in self.recording[o]
                        }
                    )
                elif type(o) == str:
                    arrays.update(
                        {
```

```
                            "_".join([o, v]): self.recording[o][v]
                            for v in self.recording[o]
                        }
                    )

            np.savez_compressed(path, **arrays)

        elif fmt == "pickle":
            with open(path, "wb") as f:
                torch.save(self.recording, f)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets recordings to empty ``torch.Tensors``.
        """
        # Reset to empty recordings
        self.recording = {k: {} for k in self.layers + self.connections}

        if self.time is not None:
            self.i = 0

        # If no simulation time is specified, specify 0-dimensional recordings.
        if self.time is None:
            for v in self.state_vars:
                for l in self.layers:
                    if hasattr(self.network.layers[l], v):
                        self.recording[l][v] = torch.Tensor()

                for c in self.connections:
                    if hasattr(self.network.connections[c], v):
                        self.recording[c][v] = torch.Tensor()

        # If simulation time is specified, pre-allocate recordings in memory for speed.
        else:
            for v in self.state_vars:
                for l in self.layers:
                    if hasattr(self.network.layers[l], v):
                        self.recording[l][v] = torch.zeros(
                            self.time, *getattr(self.network.layers[l], v).size()
                        )

                for c in self.connections:
                    if hasattr(self.network.connections[c], v):
                        self.recording[c][v] = torch.zeros(
                            self.time, *getattr(self.network.layers[c], v).size()
                        )
```

Listing B.9: *Monitors*

B.4 Pipeline

```python
from .environment_pipeline import EnvironmentPipeline
from .base_pipeline import BasePipeline
from .dataloader_pipeline import DataLoaderPipeline, TorchVisionDatasetPipeline
from . import action
```

Listing B.10: *Initialization*

```python
import itertools
from typing import Callable, Optional, Tuple, Dict

import torch

from .base_pipeline import BasePipeline
from ..analysis.pipeline_analysis import MatplotlibAnalyzer
from ..environment import Environment
from ..network import Network
from ..network.nodes import AbstractInput
from ..network.monitors import Monitor


class EnvironmentPipeline(BasePipeline):
    # language=rst
    """
    Abstracts the interaction between ``Network``, ``Environment``, and environment
    feedback action.
    """

    def __init__(
        self,
        network: Network,
        environment: Environment,
        action_function: Optional[Callable] = None,
        **kwargs,
    ):
        # language=rst
        """
        Initializes the pipeline.

        :param network: Arbitrary network object.
        :param environment: Arbitrary environment.
        :param action_function: Function to convert network outputs into environment
            inputs.

        Keyword arguments:

        :param int num_episodes: Number of episodes to train for. Defaults to 100.
        :param str output: String name of the layer from which to take output.
```

```
        :param int render_interval: Interval to render the environment.
        :param int reward_delay: How many iterations to delay delivery of reward.
        :param int time: Time for which to run the network. Defaults to the network's
            timestep.
        """
        super().__init__(network, **kwargs)

        self.episode = 0

        self.env = environment
        self.action_function = action_function

        self.accumulated_reward = 0.0
        self.reward_list = []

        # Setting kwargs.
        self.num_episodes = kwargs.get("num_episodes", 100)
        self.output = kwargs.get("output", None)
        self.render_interval = kwargs.get("render_interval", None)
        self.reward_delay = kwargs.get("reward_delay", None)
        self.time = kwargs.get("time", int(network.dt))

        if self.reward_delay is not None:
            assert self.reward_delay > 0
            self.rewards = torch.zeros(self.reward_delay)

        # Set up for multiple layers of input layers.
        self.inputs = [
            name
            for name, layer in network.layers.items()
            if isinstance(layer, AbstractInput)
        ]

        self.action = None

        self.voltage_record = None
        self.threshold_value = None
        self.reward_plot = None

        self.first = True
        self.analyzer = MatplotlibAnalyzer(**self.plot_config)

        if self.output is not None:
            self.network.add_monitor(
                Monitor(self.network.layers[self.output], ["s"]), self.output
            )

            self.spike_record = {
                self.output: torch.zeros((self.time, self.env.action_space.n))
```

```
            }

    def init_fn(self) -> None:
        pass

    def train(self, **kwargs) -> None:
        # language=rst
        """
        Trains for the specified number of episodes. Each episode can be of arbitrary
        length.
        """
        while self.episode < self.num_episodes:
            self.reset_state_variables()

            for _ in itertools.count():
                obs, reward, done, info = self.env_step()

                self.step((obs, reward, done, info), **kwargs)

                if done:
                    break

            print(
                f"Episode: {self.episode} - "
                f"accumulated reward: {self.accumulated_reward:.2f}"
            )
            self.episode += 1

    def env_step(self) -> Tuple[torch.Tensor, float, bool, Dict]:
        # language=rst
        """
        Single step of the environment which includes rendering, getting and performing
        the action, and accumulating/delaying rewards.

        :return: An OpenAI ``gym`` compatible tuple with modified reward and info.
        """
        # Render game.
        if (
            self.render_interval is not None
            and self.step_count % self.render_interval == 0
        ):
            self.env.render()

        # Choose action based on output neuron spiking.
        if self.action_function is not None:
            self.action = self.action_function(self, output=self.output)

        # Run a step of the environment.
        obs, reward, done, info = self.env.step(self.action)

```

```
        # Set reward in case of delay.
        if self.reward_delay is not None:
            self.rewards = torch.tensor([reward, *self.rewards[1:]]).float()
            reward = self.rewards[-1]

        # Accumulate reward.
        self.accumulated_reward += reward

        info["accumulated_reward"] = self.accumulated_reward

        return obs, reward, done, info

    def step_(
        self, gym_batch: Tuple[torch.Tensor, float, bool, Dict], **kwargs
    ) -> None:
        # language=rst
        """
        Run a single iteration of the network and update it and the reward list when
        done.

        :param gym_batch: An OpenAI ``gym`` compatible tuple.
        """
        obs, reward, done, info = gym_batch

        # Place the observations into the inputs.
        obs_shape = [1] * len(obs.shape[1:])
        inputs = {k: obs.repeat(self.time, *obs_shape) for k in self.inputs}

        # Run the network on the spike train-encoded inputs.
        self.network.run(inputs=inputs, time=self.time, reward=reward, **kwargs)

        if self.output is not None:
            self.spike_record[self.output] = (
                self.network.monitors[self.output].get("s").float()
            )

        if done:
            if self.network.reward_fn is not None:
                self.network.reward_fn.update(
                    accumulated_reward=self.accumulated_reward,
                    steps=self.step_count,
                    **kwargs,
                )
            self.reward_list.append(self.accumulated_reward)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Reset the pipeline.
```

```
        """
        self.env.reset()
        self.network.reset_state_variables()
        self.accumulated_reward = 0.0
        self.step_count = 0

    def plots(self, gym_batch: Tuple[torch.Tensor, float, bool, Dict], *args) -> None:
        # language=rst
        """
        Plot the encoded input, layer spikes, and layer voltages.

        :param gym_batch: An OpenAI ``gym`` compatible tuple.
        """
        obs, reward, done, info = gym_batch

        for key, item in self.plot_config.items():
            if key == "obs_step" and item is not None:
                if self.step_count % item == 0:
                    self.analyzer.plot_obs(obs[0, ...].sum(0))
            elif key == "data_step" and item is not None:
                if self.step_count % item == 0:
                    self.analyzer.plot_spikes(self.get_spike_data())
                    self.analyzer.plot_voltages(*self.get_voltage_data())
            elif key == "reward_eps" and item is not None:
                if self.episode % item == 0 and done:
                    self.analyzer.plot_reward(self.reward_list)

        self.analyzer.finalize_step()
```

Listing B.11: *Environmental pipeline*

```
import time
from typing import Tuple, Dict, Any

import torch
from torch._six import container_abcs, string_classes

from ..network import Network
from ..network.monitors import Monitor


def recursive_to(item, device):
    # language=rst
    """
    Recursively transfers everything contained in item to the target
    device.

    :param item: An individual tensor or container of tensors.
    :param device: ``torch.device`` pointing to ``"cuda"`` or ``"cpu"``.

    :return: A version of the item that has been sent to a device.
```

```python
    """

    if isinstance(item, torch.Tensor):
        return item.to(device)
    elif isinstance(item, (string_classes, int, float, bool)):
        return item
    elif isinstance(item, container_abcs.Mapping):
        return {key: recursive_to(item[key], device) for key in item }
    elif isinstance(item, tuple) and hasattr(item, "_fields"):
        return type(item)(*(recursive_to(i, device) for i in item))
    elif isinstance(item, container_abcs.Sequence):
        return [recursive_to(i, device) for i in item]
    else:
        raise NotImplementedError(f"Target type {type(item)} not supported.")


class BasePipeline:
    # language=rst
    """
    A generic pipeline that handles high level functionality.
    """

    def __init__(self, network: Network, **kwargs) -> None:
        # language=rst
        """
        Initializes the pipeline.

        :param network: Arbitrary network object, will be managed by the
            ``BasePipeline`` class.

        Keyword arguments:

        :param int save_interval: How often to save the network to disk.
        :param str save_dir: Directory to save network object to.
        :param Dict[str, Any] plot_config: Dict containing the plot configuration.
            Includes length, type (``"color"`` or ``"line"``), and interval per plot
            type.
        :param int print_interval: Interval to print text output.
        :param bool allow_gpu: Allows automatic transfer to the GPU.
        """
        self.network = network

        # Network saving handles caching of intermediate results.
        self.save_dir = kwargs.get("save_dir", "network.pt")
        self.save_interval = kwargs.get("save_interval", None)

        # Handles plotting of all layer spikes and voltages.
        # This constructs monitors at every level.
```

```python
        self.plot_config = kwargs.get(
            "plot_config", {"data_step": None, "data_length": 10}
        )

        if self.plot_config["data_step"] is not None:
            for l in self.network.layers:
                self.network.add_monitor(
                    Monitor(
                        self.network.layers[l], "s", self.plot_config["data_length"]
                    ),
                    name=f"{l}_spikes",
                )
                if hasattr(self.network.layers[l], "v"):
                    self.network.add_monitor(
                        Monitor(
                            self.network.layers[l], "v", self.plot_config["data_length"]
                        ),
                        name=f"{l}_voltages",
                    )

        self.print_interval = kwargs.get("print_interval", None)
        self.test_interval = kwargs.get("test_interval", None)
        self.step_count = 0
        self.init_fn()
        self.clock = time.time()
        self.allow_gpu = kwargs.get("allow_gpu", True)

        if torch.cuda.is_available() and self.allow_gpu:
            self.device = torch.device("cuda")
        else:
            self.device = torch.device("cpu")

        self.network.to(self.device)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Reset the pipeline.
        """
        self.network.reset_state_variables()
        self.step_count = 0

    def step(self, batch: Any, **kwargs) -> Any:
        # language=rst
        """
        Single step of any pipeline at a high level.

        :param batch: A batch of inputs to be handed to the ``step_()`` function.
                      Standard in subclasses of ``BasePipeline``.
        :return: The output from the subclass's ``step_()`` method, which could be
```

```
            anything. Passed to plotting to accommodate this.
        """
        self.step_count += 1

        batch = recursive_to(batch, self.device)
        step_out = self.step_(batch, **kwargs)

        if (
            self.print_interval is not None
            and self.step_count % self.print_interval == 0
        ):
            print(
                f"Iteration: {self.step_count} (Time: {time.time() - self.clock:.4f})"
            )
            self.clock = time.time()

        self.plots(batch, step_out)

        if self.save_interval is not None and self.step_count % self.save_interval == 0:
            self.network.save(self.save_dir)

        if self.test_interval is not None and self.step_count % self.test_interval == 0:
            self.test()

        return step_out

    def get_spike_data(self) -> Dict[str, torch.Tensor]:
        # language=rst
        """
        Get the spike data from all layers in the pipeline's network.

        :return: A dictionary containing all spike monitors from the network.
        """
        return {
            l: self.network.monitors[f"{l}_spikes"].get("s")
            for l in self.network.layers
        }

    def get_voltage_data(
        self
    ) -> Tuple[Dict[str, torch.Tensor], Dict[str, torch.Tensor]]:
        # language=rst
        """
        Get the voltage data and threshold value from all applicable layers in the
        pipeline's network.

        :return: Two dictionaries containing the voltage data and threshold values from
```

```
            the network.
        """
        voltage_record = {}
        threshold_value = {}
        for l in self.network.layers:
            if hasattr(self.network.layers[l], "v"):
                voltage_record[l] = self.network.monitors[f"{l}
_voltages"].get("v")
            if hasattr(self.network.layers[l], "thresh"):
                threshold_value[l] = self.network.layers[l].thresh

        return voltage_record, threshold_value

    def step_(self, batch: Any, **kwargs) -> Any:
        # language=rst
        """
        Perform a pass of the network given the input batch.

        :param batch: The current batch. This could be anything as
long as the subclass
            agrees upon the format in some way.
        :return: Any output that is need for recording purposes.
        """
        raise NotImplementedError("You need to provide a step_
method.")

    def train(self) -> None:
        # language=rst
        """
        A fully self-contained training loop.
        """
        raise NotImplementedError("You need to provide a train
method.")

    def test(self) -> None:
        # language=rst
        """
        A fully self contained test function.
        """
        raise NotImplementedError("You need to provide a test method
.")

    def init_fn(self) -> None:
        # language=rst
        """
        Placeholder function for subclass-specific actions that need
 to
        happen during the construction of the ``BasePipeline``.
        """
        raise NotImplementedError("You need to provide an init_fn
method.")

    def plots(self, batch: Any, step_out: Any) -> None:
        # language=rst
```

```
        """
        Create any plots and logs for a step given the input batch
    and step output.

        :param batch: The current batch. This could be anything as
    long as the subclass
            agrees upon the format in some way.
        :param step_out: The output from the ``step_()`` method.
        """
        raise NotImplementedError("You need to provide a plots
    method.")
```

Listing B.12: *Base pipeline*

```
from typing import Optional, Dict

import torch
from torch.utils.data import Dataset
from tqdm import tqdm

from ..network import Network
from .base_pipeline import BasePipeline
from ..analysis.pipeline_analysis import PipelineAnalyzer
from ..datasets import DataLoader


class DataLoaderPipeline(BasePipeline):
    # language=rst
    """
    A generic ``DataLoader`` pipeline that leverages the ``torch.
utils.data`` setup.
    This still needs to be subclassed for specific implementations
for functions given
    the dataset that will be used. An example can be seen in
    ``TorchVisionDatasetPipeline``.
    """

    def __init__(
        self,
        network: Network,
        train_ds: Dataset,
        test_ds: Optional[Dataset] = None,
        **kwargs
    ) -> None:
        # language=rst
        """
        Initializes the pipeline.

        :param network: Arbitrary ``network`` object.
        :param train_ds: Arbitrary ``torch.utils.data.Dataset``
object.
        :param test_ds: Arbitrary ``torch.utils.data.Dataset``
object.
        """
```

```
        super().__init__(network, **kwargs)

        self.train_ds = train_ds
        self.test_ds = test_ds

        self.num_epochs = kwargs.get("num_epochs", 10)
        self.batch_size = kwargs.get("batch_size", 1)
        self.num_workers = kwargs.get("num_workers", 0)
        self.pin_memory = kwargs.get("pin_memory", True)
        self.shuffle = kwargs.get("shuffle", True)

    def train(self) -> None:
        # language=rst
        """
        Training loop that runs for the set number of epochs and
    creates a new
        ``DataLoader`` at each epoch.
        """
        for epoch in range(self.num_epochs):
            train_dataloader = DataLoader(
                self.train_ds,
                batch_size=self.batch_size,
                num_workers=self.num_workers,
                pin_memory=self.pin_memory,
                shuffle=self.shuffle,
            )

            for step, batch in enumerate(
                tqdm(
                    train_dataloader,
                    desc="Epoch %d/%d" % (epoch + 1, self.num_epochs
    ),
                    total=len(self.train_ds) // self.batch_size,
                )
            ):
                self.step(batch)

    def test(self) -> None:
        raise NotImplementedError("You need to provide a test
    function.")


class TorchVisionDatasetPipeline(DataLoaderPipeline):
    # language=rst
    """
    An example implementation of ``DataLoaderPipeline`` that runs
    all of the datasets
    inside of ``bindsnet.datasets`` that inherit from an instance of
    a
    ``torchvision.datasets``. These are documented in ``bindsnet/
    datasets/README.md``.
    This specific class just runs an unsupervised network.
    """

```

```python
    def __init__(
        self,
        network: Network,
        train_ds: Dataset,
        pipeline_analyzer: Optional[PipelineAnalyzer] = None,
        **kwargs
    ) -> None:
        # language=rst
        """
        Initializes the pipeline.

        :param network: Arbitrary ``network`` object.
        :param train_ds: A ``torchvision.datasets`` wrapper dataset from
            ``bindsnet.datasets``.

        Keyword arguments:

        :param str input_layer: Layer of the network that receives input.
        """
        super().__init__(network, train_ds, None, **kwargs)

        self.input_layer = kwargs.get("input_layer", "X")
        self.pipeline_analyzer = pipeline_analyzer

    def step_(self, batch: Dict[str, torch.Tensor], **kwargs) -> None:
        # language=rst
        """
        Perform a pass of the network given the input batch. Unsupervised training
        (implying everything is stored inside of the ``network`` object, therefore
        returns ``None``.

        :param batch: A dictionary of the current batch. Includes image, label and
            encoded versions.
        """
        self.network.reset_state_variables()
        inputs = {self.input_layer: batch["encoded_image"]}
        self.network.run(inputs, time=batch["encoded_image"].shape[0])

    def init_fn(self) -> None:
        pass

    def plots(self, batch: Dict[str, torch.Tensor], *args) -> None:
        # language=rst
        """
        Create any plots and logs for a step given the input batch.

        :param batch: A dictionary of the current batch. Includes
```

```
image, label and
            encoded versions.
        """
        if self.pipeline_analyzer is not None:
            self.pipeline_analyzer.plot_obs(
                batch["encoded_image"][0, ...].sum(0), step=self.step_count
            )

            self.pipeline_analyzer.plot_spikes(
                self.get_spike_data(), step=self.step_count
            )

            vr, tv = self.get_voltage_data()
            self.pipeline_analyzer.plot_voltages(vr, tv, step=self.step_count)

            self.pipeline_analyzer.finalize_step()

    def test_step(self):
        pass
```

Listing B.13: *Data-loader pipeline*

```
import torch
import numpy as np

from . import EnvironmentPipeline


def select_multinomial(pipeline: EnvironmentPipeline, **kwargs) -> int:
    # language=rst
    """
    Selects an action probabilistically based on spiking activity from a network layer.

    :param pipeline: EnvironmentPipeline with environment that has an integer action
        space.
    :return: Action sampled from multinomial over activity of similarly-sized output
        layer.

    Keyword arguments:

    :param str output: Name of output layer whose activity to base action selection on.
    """
    try:
        output = kwargs["output"]
    except KeyError:
        raise KeyError('select_multinomial() requires an "output" layer argument.')
```

```

    output = pipeline.network.layers[output]
    action_space = pipeline.env.action_space

    assert (
        output.n % action_space.n == 0
    ), f"Output layer size of {output.n} is not divisible by action space size of {action_space.n}."

    pop_size = int(output.n / action_space.n)
    spikes = output.s
    _sum = spikes.sum().float()

    # Choose action based on population's spiking.
    if _sum == 0:
        action = np.random.choice(pipeline.env.action_space.n)
    else:
        pop_spikes = torch.tensor(
            [
                spikes[(i * pop_size) : (i * pop_size) + pop_size].sum()
                for i in range(action_space.n)
            ]
        )
        action = torch.multinomial((pop_spikes.float() / _sum).view(-1), 1)[0].item()

    return action


def select_softmax(pipeline: EnvironmentPipeline, **kwargs) -> int:
    # language=rst
    """
    Selects an action using softmax function based on spiking from a network layer.

    :param pipeline: EnvironmentPipeline with environment that has an integer action
        space and :code:`spike_record` set.
    :return: Action sampled from softmax over activity of similarly-sized output layer.

    Keyword arguments:

    :param str output: Name of output layer whose activity to base action selection on.
    """
    try:
        output = kwargs["output"]
    except KeyError:
        raise KeyError('select_softmax() requires an "output" layer argument.')

    assert (
```

```
        pipeline.network.layers[output].n == pipeline.env.action_space.n
    ), "Output layer size is not equal to the size of the action space."

    assert hasattr(
        pipeline, "spike_record"
    ), "EnvironmentPipeline is missing the attribute: spike_record."

    spikes = torch.sum(pipeline.spike_record[output], dim=0)
    probabilities = torch.softmax(spikes, dim=0)
    return torch.multinomial(probabilities, num_samples=1).item()


def select_random(pipeline: EnvironmentPipeline, **kwargs) -> int:
    # language=rst
    """
    Selects an action randomly from the action space.

    :param pipeline: EnvironmentPipeline with environment that has an integer action
        space.
    :return: Action randomly sampled over size of pipeline's action space.
    """
    # Choose action randomly from the action space.
    return np.random.choice(pipeline.env.action_space.n)
```

Listing B.14: *Action*

B.5 Encoding

```
from .encodings import single, repeat, bernoulli, poisson, rank_order
from .loaders import bernoulli_loader, poisson_loader, rank_order_loader
from .encoders import (
    Encoder,
    NullEncoder,
    SingleEncoder,
    RepeatEncoder,
    BernoulliEncoder,
    PoissonEncoder,
    RankOrderEncoder,
)
```

Listing B.15: *Initialization*

```
from typing import Optional, Union, Iterable, Iterator

import torch

from .encodings import bernoulli, poisson, rank_order

```

```python

def bernoulli_loader(
    data: Union[torch.Tensor, Iterable[torch.Tensor]],
    time: Optional[int] = None,
    dt: float = 1.0,
    **kwargs
) -> Iterator[torch.Tensor]:
    # language=rst
    """
    Lazily invokes ``bindsnet.encoding.bernoulli`` to iteratively encode a sequence of
    data.

    :param data: Tensor of shape ``[n_samples, n_1, ..., n_k]``.
    :param time: Length of Bernoulli spike train per input variable.
    :param dt: Simulation time step.
    :return: Tensors of shape ``[time, n_1, ..., n_k]`` of Bernoulli-distributed spikes.

    Keyword arguments:

    :param float max_prob: Maximum probability of spike per Bernoulli trial.
    """
    # Setting kwargs.
    max_prob = kwargs.get("dt", 1.0)

    for i in range(len(data)):
        # Encode datum as Bernoulli spike trains.
        yield bernoulli(datum=data[i], time=time, dt=dt, max_prob=max_prob)


def poisson_loader(
    data: Union[torch.Tensor, Iterable[torch.Tensor]],
    time: int,
    dt: float = 1.0,
    **kwargs
) -> Iterator[torch.Tensor]:
    # language=rst
    """
    Lazily invokes ``bindsnet.encoding.poisson`` to iteratively encode a sequence of
    data.

    :param data: Tensor of shape ``[n_samples, n_1, ..., n_k]``.
    :param time: Length of Poisson spike train per input variable.
    :param dt: Simulation time step.
    :return: Tensors of shape ``[time, n_1, ..., n_k]`` of Poisson-distributed spikes.
    """
    for i in range(len(data)):
        # Encode datum as Poisson spike trains.
        yield poisson(datum=data[i], time=time, dt=dt)
```

```


def rank_order_loader(
    data: Union[torch.Tensor, Iterable[torch.Tensor]],
    time: int,
    dt: float = 1.0,
    **kwargs
) -> Iterator[torch.Tensor]:
    # language=rst
    """
    Lazily invokes ``bindsnet.encoding.rank_order`` to iteratively
   encode a sequence of
    data.

    :param data: Tensor of shape ``[n_samples, n_1, ..., n_k]``.
    :param time: Length of rank order-encoded spike train per input
   variable.
    :param dt: Simulation time step.
    :return: Tensors of shape ``[time, n_1, ..., n_k]`` of rank
   order-encoded spikes.
    """
    for i in range(len(data)):
        # Encode datum as rank order-encoded spike trains.
        yield rank_order(datum=data[i], time=time, dt=dt)
```

Listing B.16: *Loaders*

```
from . import encodings


class Encoder:
    # language=rst
    """
    Base class for spike encodings transforms.

    Calls ``self.enc`` from the subclass and passes whatever
   arguments were provided.
    ``self.enc`` must be callable with ``torch.Tensor``, ``*args``,
   ``**kwargs``
    """

    def __init__(self, *args, **kwargs) -> None:
        self.enc_args = args
        self.enc_kwargs = kwargs

    def __call__(self, img):
        return self.enc(img, *self.enc_args, **self.enc_kwargs)


class NullEncoder(Encoder):
    # language=rst
    """
    Pass through of the datum that was input.

```

```
    .. note::
        This is not a real spike encoder. Be careful with the usage of this class.
    """

    def __init__(self):
        super().__init__()

    def __call__(self, img):
        return img


class SingleEncoder(Encoder):
    def __init__(self, time: int, dt: float = 1.0, sparsity: float = 0.5, **kwargs):
        # language=rst
        """
        Creates a callable SingleEncoder which encodes as defined in
        ``bindsnet.encoding.single``

        :param time: Length of single spike train per input variable.
        :param dt: Simulation time step.
        :param sparsity: Sparsity of the input representation. 0 for no spikes and 1 for
            all spikes.
        """
        super().__init__(time, dt=dt, sparsity=sparsity, **kwargs)

        self.enc = encodings.single


class RepeatEncoder(Encoder):
    def __init__(self, time: int, dt: float = 1.0, **kwargs):
        # language=rst
        """
        Creates a callable ``RepeatEncoder`` which encodes as defined in
        ``bindsnet.encoding.repeat``

        :param time: Length of repeat spike train per input variable.
        :param dt: Simulation time step.
        """
        super().__init__(time, dt=dt, **kwargs)

        self.enc = encodings.repeat


class BernoulliEncoder(Encoder):
    def __init__(self, time: int, dt: float = 1.0, **kwargs):
        # language=rst
        """
        Creates a callable ``BernoulliEncoder`` which encodes as
```

```
    defined in
        :code:`bindsnet.encoding.bernoulli`

        :param time: Length of Bernoulli spike train per input variable.
        :param dt: Simulation time step.

        Keyword arguments:

        :param float max_prob: Maximum probability of spike per time step.
        """
        super().__init__(time, dt=dt, **kwargs)

        self.enc = encodings.bernoulli


class PoissonEncoder(Encoder):
    def __init__(self, time: int, dt: float = 1.0, **kwargs):
        # language=rst
        """
        Creates a callable PoissonEncoder which encodes as defined in
        ``bindsnet.encoding.poisson`

        :param time: Length of Poisson spike train per input variable.
        :param dt: Simulation time step.
        """
        super().__init__(time, dt=dt, **kwargs)

        self.enc = encodings.poisson


class RankOrderEncoder(Encoder):
    def __init__(self, time: int, dt: float = 1.0, **kwargs):
        # language=rst
        """
        Creates a callable RankOrderEncoder which encodes as defined in
        :code:`bindsnet.encoding.rank_order`

        :param time: Length of RankOrder spike train per input variable.
        :param dt: Simulation time step.
        """
        super().__init__(time, dt=dt, **kwargs)

        self.enc = encodings.rank_order
```

Listing B.17: *Encoders*

```
from typing import Optional

```

```
import torch
import numpy as np


def single(
    datum: torch.Tensor, time: int, dt: float = 1.0, sparsity: float = 0.5, **kwargs
) -> torch.Tensor:
    # language=rst
    """
    Generates timing based single-spike encoding. Spike occurs earlier if the
    intensity of the input feature is higher. Features whose value is lower than
    threshold is remain silent.

    :param datum: Tensor of shape ``[n_1, ..., n_k]``.
    :param time: Length of the input and output.
    :param dt: Simulation time step.
    :param sparsity: Sparsity of the input representation. 0 for no spikes and 1 for all
        spikes.
    :return: Tensor of shape ``[time, n_1, ..., n_k]``.
    """
    time = int(time / dt)
    shape = list(datum.shape)
    datum = np.copy(datum)
    quantile = np.quantile(datum, 1 - sparsity)
    s = np.zeros([time, *shape])
    s[0] = np.where(datum > quantile, np.ones(shape), np.zeros(shape))
    return torch.Tensor(s).byte()


def repeat(datum: torch.Tensor, time: int, dt: float = 1.0, **kwargs) -> torch.Tensor:
    # language=rst
    """
    :param datum: Repeats a tensor along a new dimension in the 0th position for
        ``int(time / dt)`` timesteps.
    :param time: Tensor of shape ``[n_1, ..., n_k]``.
    :param dt: Simulation time step.
    :return: Tensor of shape ``[time, n_1, ..., n_k]`` of repeated data along the 0-th
        dimension.
    """
    time = int(time / dt)
    return datum.repeat([time, *([1] * len(datum.shape))])


def bernoulli(
    datum: torch.Tensor, time: Optional[int] = None, dt: float = 1.0, **kwargs
```

```
) -> torch.Tensor:
    # language=rst
    """
    Generates Bernoulli-distributed spike trains based on input
    intensity. Inputs must
    be non-negative. Spikes correspond to successful Bernoulli
    trials, with success
    probability equal to (normalized in [0, 1]) input value.

    :param datum: Tensor of shape ``[n_1, ..., n_k]``.
    :param time: Length of Bernoulli spike train per input variable.
    :param dt: Simulation time step.
    :return: Tensor of shape ``[time, n_1, ..., n_k]`` of Bernoulli-
    distributed spikes.

    Keyword arguments:

    :param float max_prob: Maximum probability of spike per
    Bernoulli trial.
    """
    # Setting kwargs.
    max_prob = kwargs.get("max_prob", 1.0)

    assert 0 <= max_prob <= 1, "Maximum firing probability must be
    in range [0, 1]"
    assert (datum >= 0).all(), "Inputs must be non-negative"

    shape, size = datum.shape, datum.numel()
    datum = datum.flatten()

    if time is not None:
        time = int(time / dt)

    # Normalize inputs and rescale (spike probability proportional
    to input intensity).
    if datum.max() > 1.0:
        datum /= datum.max()

    # Make spike data from Bernoulli sampling.
    if time is None:
        spikes = torch.bernoulli(max_prob * datum)
        spikes = spikes.view(*shape)
    else:
        spikes = torch.bernoulli(max_prob * datum.repeat([time, 1]))
        spikes = spikes.view(time, *shape)

    return spikes.byte()


def poisson(datum: torch.Tensor, time: int, dt: float = 1.0, **
    kwargs) -> torch.Tensor:
    # language=rst
    """
    Generates Poisson-distributed spike trains based on input
```

```
    intensity. Inputs must be
    non-negative, and give the firing rate in Hz. Inter-spike intervals (ISIs) for
    non-negative data incremented by one to avoid zero intervals while maintaining ISI
    distributions.

    :param datum: Tensor of shape ``[n_1, ..., n_k]``.
    :param time: Length of Poisson spike train per input variable.
    :param dt: Simulation time step.
    :return: Tensor of shape ``[time, n_1, ..., n_k]`` of Poisson-distributed spikes.
    """
    assert (datum >= 0).all(), "Inputs must be non-negative"

    # Get shape and size of data.
    shape, size = datum.shape, datum.numel()
    datum = datum.flatten()
    time = int(time / dt)

    # Compute firing rates in seconds as function of data intensity,
    # accounting for simulation time step.
    rate = torch.zeros(size)
    rate[datum != 0] = 1 / datum[datum != 0] * (1000 / dt)

    # Create Poisson distribution and sample inter-spike intervals
    # (incrementing by 1 to avoid zero intervals).
    dist = torch.distributions.Poisson(rate=rate)
    intervals = dist.sample(sample_shape=torch.Size([time + 1]))
    intervals[:, datum != 0] += (intervals[:, datum != 0] == 0).float()

    # Calculate spike times by cumulatively summing over time dimension.
    times = torch.cumsum(intervals, dim=0).long()
    times[times >= time + 1] = 0

    # Create tensor of spikes.
    spikes = torch.zeros(time + 1, size).byte()
    spikes[times, torch.arange(size)] = 1
    spikes = spikes[1:]

    return spikes.view(time, *shape)


def rank_order(
    datum: torch.Tensor, time: int, dt: float = 1.0, **kwargs
) -> torch.Tensor:
    # language=rst
    """
    Encodes data via a rank order coding-like representation. One spike per neuron,
    temporally ordered by decreasing intensity. Inputs must be non-negative.
```

```

    :param datum: Tensor of shape ``[n_samples, n_1, ..., n_k]``.
    :param time: Length of rank order-encoded spike train per input
    variable.
    :param dt: Simulation time step.
    :return: Tensor of shape ``[time, n_1, ..., n_k]`` of rank order
    -encoded spikes.
    """
    assert (datum >= 0).all(), "Inputs must be non-negative"

    shape, size = datum.shape, datum.numel()
    datum = datum.flatten()
    time = int(time / dt)

    # Create spike times in order of decreasing intensity.
    datum /= datum.max()
    times = torch.zeros(size)
    times[datum != 0] = 1 / datum[datum != 0]
    times *= time / times.max()  # Extended through simulation time.
    times = torch.ceil(times).long()

    # Create spike times tensor.
    spikes = torch.zeros(time, size).byte()
    for i in range(size):
        if 0 < times[i] < time:
            spikes[times[i] - 1, i] = 1

    return spikes.reshape(time, *shape)
```

Listing B.18: *Encodings*

B.6 Conversion

```
from .conversion import (
    Permute,
    FeatureExtractor,
    SubtractiveResetIFNodes,
    PassThroughNodes,
    PermuteConnection,
    ConstantPad2dConnection,
    data_based_normalization,
    ann_to_snn,
)
```

Listing B.19: *Initialization*

```
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F

from torch.nn.modules.utils import _pair

from copy import deepcopy
```

```python
from typing import Union, Sequence, Optional, Tuple, Dict, Iterable

import bindsnet.network.nodes as nodes
import bindsnet.network.topology as topology

from bindsnet.network import Network


class Permute(nn.Module):
    # language=rst
    """
    PyTorch module for the explicit permutation of a tensor's
   dimensions in a parent
    module's ``forward`` pass (as opposed to ``torch.permute``).
    """

    def __init__(self, dims):
        # language=rst
        """
        Constructor for ``Permute`` module.

        :param dims: Ordering of dimensions for permutation.
        """
        super(Permute, self).__init__()

        self.dims = dims

    def forward(self, x):
        # language=rst
        """
        Forward pass of permutation module.

        :param x: Input tensor to permute.
        :return: Permuted input tensor.
        """
        return x.permute(*self.dims).contiguous()


class FeatureExtractor(nn.Module):
    # language=rst
    """
    Special-purpose PyTorch module for the extraction of child
   module's activations.
    """

    def __init__(self, submodule):
        # language=rst
        """
        Constructor for ``FeatureExtractor`` module.

        :param submodule: The module who's children modules are to
   be extracted.
        """
        super(FeatureExtractor, self).__init__()
```

```python

        self.submodule = submodule

    def forward(self, x: torch.Tensor) -> Dict[nn.Module, torch.Tensor]:
        # language=rst
        """
        Forward pass of the feature extractor.

        :param x: Input data for the ``submodule''.
        :return: A dictionary mapping
        """
        activations = {"input": x}
        for name, module in self.submodule._modules.items():
            if isinstance(module, nn.Linear):
                x = x.view(-1, module.in_features)

            x = module(x)
            activations[name] = x

        return activations


class SubtractiveResetIFNodes(nodes.Nodes):
    # language=rst
    """
    Layer of `integrate-and-fire (IF) neurons
    <http://neuronaldynamics.epfl.ch/online/Ch1.S3.html>` using reset by subtraction.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Iterable[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
        thresh: Union[float, torch.Tensor] = -52.0,
        reset: Union[float, torch.Tensor] = -65.0,
        refrac: Union[int, torch.Tensor] = 5,
        lbound: float = None,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of IF neurons with the subtractive reset mechanism from
        `this paper
        <https://www.frontiersin.org/articles/10.3389/fnins.2017.00682/full>`_.

```

```
        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param traces_additive: Whether to record spike traces additively.
        :param tc_trace: Time constant of spike trace decay.
        :param trace_scale: Scaling factor for spike trace.
        :param sum_input: Whether to sum all inputs.
        :param thresh: Spike threshold voltage.
        :param reset: Post-spike reset voltage.
        :param refrac: Refractory (non-firing) period of the neuron.
        :param lbound: Lower bound of the voltage.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )

        self.register_buffer(
            "reset", torch.tensor(reset, dtype=torch.float)
        )  # Post-spike reset voltage.
        self.register_buffer(
            "thresh", torch.tensor(thresh, dtype=torch.float)
        )  # Spike threshold voltage.
        self.register_buffer(
            "refrac", torch.tensor(refrac)
        )  # Post-spike refractory period.
        self.register_buffer("v", torch.FloatTensor())  # Neuron voltages.
        self.register_buffer(
            "refrac_count", torch.FloatTensor()
        )  # Refractory period counters.

        self.lbound = lbound  # Lower bound of voltage.

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param x: Inputs to the layer.
        """
        # Integrate input voltages.
        self.v += (self.refrac_count == 0).float() * x

        # Decrement refractory counters.
        self.refrac_count = (self.refrac_count > 0).float() * (
            self.refrac_count - self.dt
        )
```

```

        # Check for spiking neurons.
        self.s = self.v >= self.thresh

        # Refractoriness and voltage reset.
        self.refrac_count.masked_fill_(self.s, self.refrac)
        self.v[self.s] = self.v[self.s] - self.thresh

        # Voltage clipping to lower bound.
        if self.lbound is not None:
            self.v.masked_fill_(self.v < self.lbound, self.lbound)

        super().forward(x)

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        super().reset_state_variables()
        self.v.fill_(self.reset)  # Neuron voltages.
        self.refrac_count.zero_()  # Refractory period counters.

    def set_batch_size(self, batch_size) -> None:
        # language=rst
        """
        Sets mini-batch size. Called when layer is added to a network.

        :param batch_size: Mini-batch size.
        """
        super().set_batch_size(batch_size=batch_size)
        self.v = self.reset * torch.ones(batch_size, *self.shape, device=self.v.device)
        self.refrac_count = torch.zeros_like(self.v, device=self.refrac_count.device)


class PassThroughNodes(nodes.Nodes):
    # language=rst
    """
    Layer of `integrate-and-fire (IF) neurons
    <http://neuronaldynamics.epfl.ch/online/Ch1.S3.html>`_ with using reset by
    subtraction.
    """

    def __init__(
        self,
        n: Optional[int] = None,
        shape: Optional[Sequence[int]] = None,
        traces: bool = False,
        traces_additive: bool = False,
        tc_trace: Union[float, torch.Tensor] = 20.0,
```

```
        trace_scale: Union[float, torch.Tensor] = 1.0,
        sum_input: bool = False,
    ) -> None:
        # language=rst
        """
        Instantiates a layer of IF neurons.

        :param n: The number of neurons in the layer.
        :param shape: The dimensionality of the layer.
        :param traces: Whether to record spike traces.
        :param trace_tc: Time constant of spike trace decay.
        :param sum_input: Whether to sum all inputs.
        """
        super().__init__(
            n=n,
            shape=shape,
            traces=traces,
            traces_additive=traces_additive,
            tc_trace=tc_trace,
            trace_scale=trace_scale,
            sum_input=sum_input,
        )
        self.register_buffer("v", torch.zeros(self.shape))

    def forward(self, x: torch.Tensor) -> None:
        # language=rst
        """
        Runs a single simulation step.

        :param inputs: Inputs to the layer.
        :param dt: Simulation time step.
        """
        self.s = x

    def reset_state_variables(self) -> None:
        # language=rst
        """
        Resets relevant state variables.
        """
        self.s.zero_()


class PermuteConnection(topology.AbstractConnection):
    # language=rst
    """
    Special-purpose connection for emulating the custom ``Permute``
   module in spiking
    neural networks.
    """

    def __init__(
        self,
        source: nodes.Nodes,
        target: nodes.Nodes,
```

```
        dims: Sequence,
        nu: Optional[Union[float, Sequence[float]]] = None,
        weight_decay: float = 0.0,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Constructor for ``PermuteConnection``.

        :param source: A layer of nodes from which the connection
originates.
        :param target: A layer of nodes to which the connection
connects.
        :param dims: Order of dimensions to permute.
        :param nu: Learning rate for both pre- and post-synaptic
events.
        :param weight_decay: Constant multiple to decay weights by
on each iteration.

        Keyword arguments:

        :param function update_rule: Modifies connection parameters
according to some
            rule.
        :param float wmin: The minimum value on the connection
weights.
        :param float wmax: The maximum value on the connection
weights.
        :param float norm: Total weight per target neuron
normalization.
        """
        super().__init__(source, target, nu, weight_decay, **kwargs)

        self.dims = dims

    def compute(self, s: torch.Tensor) -> torch.Tensor:
        # language=rst
        """
        Permute input.

        :param s: Input.
        :return: Permuted input.
        """
        return s.permute(self.dims).float()


class ConstantPad2dConnection(topology.AbstractConnection):
    # language=rst
    """
    Special-purpose connection for emulating the ``ConstantPad2d``
PyTorch module in
        spiking neural networks.
    """

```

```python
    def __init__(
        self,
        source: nodes.Nodes,
        target: nodes.Nodes,
        padding: Tuple,
        nu: Optional[Union[float, Sequence[float]]] = None,
        weight_decay: float = 0.0,
        **kwargs,
    ) -> None:
        # language=rst
        """
        Constructor for ``ConstantPad2dConnection``.

        :param source: A layer of nodes from which the connection originates.
        :param target: A layer of nodes to which the connection connects.
        :param padding: Padding of input tensors; passed to ``torch.nn.functional.pad``.
        :param nu: Learning rate for both pre- and post-synaptic events.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:

        :param function update_rule: Modifies connection parameters according to some
            rule.
        :param float wmin: The minimum value on the connection weights.
        :param float wmax: The maximum value on the connection weights.
        :param float norm: Total weight per target neuron normalization.
        """

        super().__init__(source, target, nu, weight_decay, **kwargs)

        self.padding = padding

    def compute(self, s: torch.Tensor):
        # language=rst
        """
        Pad input.

        :param s: Input.
        :return: Padding input.
        """
        return F.pad(s, self.padding).float()


def data_based_normalization(
    ann: Union[nn.Module, str], data: torch.Tensor, percentile:
```

```
    float = 99.9
):
    # language=rst
    """
    Use a dataset to rescale ANN weights and biases such that that the max ReLU
    activation is less than 1.

    :param ann: Artificial neural network implemented in PyTorch. Accepts either
        ``torch.nn.Module`` or path to network saved using ``torch.save()``.
    :param data: Data to use to perform data-based weight normalization of shape
        ``[n_examples, ...]``.
    :param percentile: Percentile (in ``[0, 100]``) of activations to scale by in
        data-based normalization scheme.
    :return: Artificial neural network with rescaled weights and biases according to
        activations on the dataset.
    """
    if isinstance(ann, str):
        ann = torch.load(ann)

    assert isinstance(ann, nn.Module)

    def set_requires_grad(module, value):
        for param in module.parameters():
            param.requires_grad = value

    set_requires_grad(ann, value=False)
    extractor = FeatureExtractor(ann)
    all_activations = extractor.forward(data)

    prev_module = None
    prev_factor = 1
    for name, module in ann._modules.items():
        if isinstance(module, nn.Sequential):

            extractor2 = FeatureExtractor(module)
            all_activations2 = extractor2.forward(data)
            for name2, module2 in module.named_children():
                activations = all_activations2[name2]

                if isinstance(module2, nn.ReLU):
                    if prev_module is not None:
                        scale_factor = np.percentile(activations.cpu(), percentile)

                        prev_module.weight *= prev_factor / scale_factor
                        prev_module.bias /= scale_factor

```

```
                        prev_factor = scale_factor

                elif isinstance(module2, nn.Linear) or isinstance(module2, nn.Conv2d):
                    prev_module = module2

        else:
            activations = all_activations[name]
            if isinstance(module, nn.ReLU):
                if prev_module is not None:
                    scale_factor = np.percentile(activations.cpu(), percentile)

                    prev_module.weight *= prev_factor / scale_factor
                    prev_module.bias /= scale_factor

                    prev_factor = scale_factor

            elif isinstance(module, nn.Linear) or isinstance(module, nn.Conv2d):
                prev_module = module

    return ann


def _ann_to_snn_helper(prev, current, node_type, last=False, **kwargs):
    # language=rst
    """
    Helper function for main ``ann_to_snn`` method.

    :param prev: Previous PyTorch module in artificial neural network.
    :param current: Current PyTorch module in artificial neural network.
    :param node_type: Type of ``bindsnet.network.nodes`` to use.
    :param last: Whether this connection and layer is the last to be converted.
    :return: Spiking neural network layer and connection corresponding to ``prev`` and
        ``current`` PyTorch modules.
    """
    if isinstance(current, nn.Linear):
        layer = node_type(
            n=current.out_features,
            reset=0,
            thresh=1,
            refrac=0,
            sum_input=last,
            **kwargs,
        )
        bias = current.bias if current.bias is not None else torch.zeros(layer.n)
        connection = topology.Connection(
```

```
            source=prev, target=layer, w=current.weight.t(), b=bias
        )

    elif isinstance(current, nn.Conv2d):
        input_height, input_width = prev.shape[2], prev.shape[3]
        out_channels, output_height, output_width = (
            current.out_channels,
            prev.shape[2],
            prev.shape[3],
        )

        width = (
            input_height - current.kernel_size[0] + 2 * current.padding[0]
        ) / current.stride[0] + 1
        height = (
            input_width - current.kernel_size[1] + 2 * current.padding[1]
        ) / current.stride[1] + 1
        shape = (1, out_channels, int(width), int(height))

        layer = node_type(
            shape=shape, reset=0, thresh=1, refrac=0, sum_input=last, **kwargs
        )
        bias = current.bias if current.bias is not None else torch.zeros(layer.shape[1])
        connection = topology.Conv2dConnection(
            source=prev,
            target=layer,
            kernel_size=current.kernel_size,
            stride=current.stride,
            padding=current.padding,
            dilation=current.dilation,
            w=current.weight,
            b=bias,
        )

    elif isinstance(current, nn.MaxPool2d):
        input_height, input_width = prev.shape[2], prev.shape[3]
        current.kernel_size = _pair(current.kernel_size)
        current.padding = _pair(current.padding)
        current.stride = _pair(current.stride)

        width = (
            input_height - current.kernel_size[0] + 2 * current.padding[0]
        ) / current.stride[0] + 1
        height = (
            input_width - current.kernel_size[1] + 2 * current.padding[1]
        ) / current.stride[1] + 1
        shape = (1, prev.shape[1], int(width), int(height))

```

```
        layer = PassThroughNodes(shape=shape)
        connection = topology.MaxPool2dConnection(
            source=prev,
            target=layer,
            kernel_size=current.kernel_size,
            stride=current.stride,
            padding=current.padding,
            dilation=current.dilation,
            decay=1,
        )

    elif isinstance(current, Permute):
        layer = PassThroughNodes(
            shape=[
                prev.shape[current.dims[0]],
                prev.shape[current.dims[1]],
                prev.shape[current.dims[2]],
                prev.shape[current.dims[3]],
            ]
        )

        connection = PermuteConnection(source=prev, target=layer, dims=current.dims)

    elif isinstance(current, nn.ConstantPad2d):
        layer = PassThroughNodes(
            shape=[
                prev.shape[0],
                prev.shape[1],
                current.padding[0] + current.padding[1] + prev.shape[2],
                current.padding[2] + current.padding[3] + prev.shape[3],
            ]
        )

        connection = ConstantPad2dConnection(
            source=prev, target=layer, padding=current.padding
        )

    else:
        return None, None

    return layer, connection


def ann_to_snn(
    ann: Union[nn.Module, str],
    input_shape: Sequence[int],
    data: Optional[torch.Tensor] = None,
    percentile: float = 99.9,
    node_type: Optional[nodes.Nodes] = SubtractiveResetIFNodes,
    **kwargs,
) -> Network:
```

```
    # language=rst
    """
    Converts an artificial neural network (ANN) written as a ``torch
.nn.Module`` into a
    near-equivalent spiking neural network.

    :param ann: Artificial neural network implemented in PyTorch.
Accepts either
        ``torch.nn.Module`` or path to network saved using ``torch.
save()``.
    :param input_shape: Shape of input data.
    :param data: Data to use to perform data-based weight
normalization of shape
        ``[n_examples, ...]``.
    :param percentile: Percentile (in ``[0, 100]``) of activations
to scale by in
        data-based normalization scheme.
    :param node_type: Class of ``Nodes`` to use in replacing ``torch
.nn.Linear`` layers
        in original ANN.
    :return: Spiking neural network implemented in PyTorch.
    """
    if isinstance(ann, str):
        ann = torch.load(ann)
    else:
        ann = deepcopy(ann)

    assert isinstance(ann, nn.Module)

    if data is None:
        import warnings

        warnings.warn("Data is None. Weights will not be scaled.",
RuntimeWarning)
    else:
        ann = data_based_normalization(
            ann=ann, data=data.detach(), percentile=percentile
        )

    snn = Network()

    input_layer = nodes.Input(shape=input_shape)
    snn.add_layer(input_layer, name="Input")

    children = []
    for c in ann.children():
        if isinstance(c, nn.Sequential):
            for c2 in list(c.children()):
                children.append(c2)
        else:
            children.append(c)

    i = 0
    prev = input_layer
```

```python
    while i < len(children) - 1:
        current, nxt = children[i : i + 2]
        layer, connection = _ann_to_snn_helper(prev, current, node_type, **kwargs)

        i += 1

        if layer is None or connection is None:
            continue

        snn.add_layer(layer, name=str(i))
        snn.add_connection(connection, source=str(i - 1), target=str(i))

        prev = layer

    current = children[-1]
    layer, connection = _ann_to_snn_helper(
        prev, current, node_type, last=True, **kwargs
    )

    i += 1

    if layer is not None or connection is not None:
        snn.add_layer(layer, name=str(i))
        snn.add_connection(connection, source=str(i - 1), target=str(i))

    return snn
```

Listing B.20: *Conversion*

B.7 Model

```python
from .models import (
    TwoLayerNetwork,
    DiehlAndCook2015,
    DiehlAndCook2015v2,
    IncreasingInhibitionNetwork,
    LocallyConnectedNetwork,
)
```

Listing B.21: *Initialization*

```python
from typing import Optional, Union, Tuple, List, Sequence, Iterable

import numpy as np
import torch
from scipy.spatial.distance import euclidean
from torch.nn.modules.utils import _pair
import torch.nn as nn
from torchvision import models

from ..learning import PostPre
```

```python
from ..network import Network
from ..network.nodes import Input, LIFNodes, DiehlAndCookNodes
from ..network.topology import Connection, LocalConnection


class TwoLayerNetwork(Network):
    # language=rst
    """
    Implements an ``Input`` instance connected to a ``LIFNodes`` instance with a
    fully-connected ``Connection``.
    """

    def __init__(
        self,
        n_inpt: int,
        n_neurons: int = 100,
        dt: float = 1.0,
        wmin: float = 0.0,
        wmax: float = 1.0,
        nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
        reduction: Optional[callable] = None,
        norm: float = 78.4,
    ) -> None:
        # language=rst
        """
        Constructor for class ``TwoLayerNetwork``.

        :param n_inpt: Number of input neurons. Matches the 1D size of the input data.
        :param n_neurons: Number of neurons in the ``LIFNodes`` population.
        :param dt: Simulation time step.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events,
            respectively.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param wmin: Minimum allowed weight on ``Input`` to ``LIFNodes`` synapses.
        :param wmax: Maximum allowed weight on ``Input`` to ``LIFNodes`` synapses.
        :param norm: ``Input`` to ``LIFNodes`` layer connection weights normalization
            constant.
        """
        super().__init__(dt=dt)

        self.n_inpt = n_inpt
        self.n_neurons = n_neurons
        self.dt = dt

        self.add_layer(Input(n=self.n_inpt, traces=True, tc_trace
```

```
    =20.0), name="X")
        self.add_layer(
            LIFNodes(
                n=self.n_neurons,
                traces=True,
                rest=-65.0,
                reset=-65.0,
                thresh=-52.0,
                refrac=5,
                tc_decay=100.0,
                tc_trace=20.0,
            ),
            name="Y",
        )

        w = 0.3 * torch.rand(self.n_inpt, self.n_neurons)
        self.add_connection(
            Connection(
                source=self.layers["X"],
                target=self.layers["Y"],
                w=w,
                update_rule=PostPre,
                nu=nu,
                reduction=reduction,
                wmin=wmin,
                wmax=wmax,
                norm=norm,
            ),
            source="X",
            target="Y",
        )


class DiehlAndCook2015(Network):
    # language=rst
    """
    Implements the spiking neural network architecture from `(Diehl
    & Cook 2015)
    <https://www.frontiersin.org/articles/10.3389/fncom.2015.00099/
    full>`_.
    """

    def __init__(
        self,
        n_inpt: int,
        n_neurons: int = 100,
        exc: float = 22.5,
        inh: float = 17.5,
        dt: float = 1.0,
        nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
        reduction: Optional[callable] = None,
        wmin: float = 0.0,
        wmax: float = 1.0,
        norm: float = 78.4,
```

```
        theta_plus: float = 0.05,
        tc_theta_decay: float = 1e7,
        inpt_shape: Optional[Iterable[int]] = None,
    ) -> None:
        # language=rst
        """
        Constructor for class ``DiehlAndCook2015``.

        :param n_inpt: Number of input neurons. Matches the 1D size of the input data.
        :param n_neurons: Number of excitatory, inhibitory neurons.
        :param exc: Strength of synapse weights from excitatory to inhibitory layer.
        :param inh: Strength of synapse weights from inhibitory to excitatory layer.
        :param dt: Simulation time step.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events,
            respectively.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param wmin: Minimum allowed weight on input to excitatory synapses.
        :param wmax: Maximum allowed weight on input to excitatory synapses.
        :param norm: Input to excitatory layer connection weights normalization
            constant.
        :param theta_plus: On-spike increment of ``DiehlAndCookNodes`` membrane
            threshold potential.
        :param tc_theta_decay: Time constant of ``DiehlAndCookNodes`` threshold
            potential decay.
        :param inpt_shape: The dimensionality of the input layer.
        """
        super().__init__(dt=dt)

        self.n_inpt = n_inpt
        self.inpt_shape = inpt_shape
        self.n_neurons = n_neurons
        self.exc = exc
        self.inh = inh
        self.dt = dt

        # Layers
        input_layer = Input(
            n=self.n_inpt, shape=self.inpt_shape, traces=True, tc_trace=20.0
        )
        exc_layer = DiehlAndCookNodes(
            n=self.n_neurons,
            traces=True,
```

```
            rest=-65.0,
            reset=-60.0,
            thresh=-52.0,
            refrac=5,
            tc_decay=100.0,
            tc_trace=20.0,
            theta_plus=theta_plus,
            tc_theta_decay=tc_theta_decay,
        )
        inh_layer = LIFNodes(
            n=self.n_neurons,
            traces=False,
            rest=-60.0,
            reset=-45.0,
            thresh=-40.0,
            tc_decay=10.0,
            refrac=2,
            tc_trace=20.0,
        )

        # Connections
        w = 0.3 * torch.rand(self.n_inpt, self.n_neurons)
        input_exc_conn = Connection(
            source=input_layer,
            target=exc_layer,
            w=w,
            update_rule=PostPre,
            nu=nu,
            reduction=reduction,
            wmin=wmin,
            wmax=wmax,
            norm=norm,
        )
        w = self.exc * torch.diag(torch.ones(self.n_neurons))
        exc_inh_conn = Connection(
            source=exc_layer, target=inh_layer, w=w, wmin=0, wmax=
    self.exc
        )
        w = -self.inh * (
            torch.ones(self.n_neurons, self.n_neurons)
            - torch.diag(torch.ones(self.n_neurons))
        )
        inh_exc_conn = Connection(
            source=inh_layer, target=exc_layer, w=w, wmin=-self.inh,
     wmax=0
        )

        # Add to network
        self.add_layer(input_layer, name="X")
        self.add_layer(exc_layer, name="Ae")
        self.add_layer(inh_layer, name="Ai")
        self.add_connection(input_exc_conn, source="X", target="Ae")
        self.add_connection(exc_inh_conn, source="Ae", target="Ai")
        self.add_connection(inh_exc_conn, source="Ai", target="Ae")
```

```python


class DiehlAndCook2015v2(Network):
    # language=rst
    """
    Slightly modifies the spiking neural network architecture from `(Diehl & Cook 2015)
    <https://www.frontiersin.org/articles/10.3389/fncom.2015.00099/full>`_ by removing
    the inhibitory layer and replacing it with a recurrent inhibitory connection in the
    output layer (what used to be the excitatory layer).
    """

    def __init__(
        self,
        n_inpt: int,
        n_neurons: int = 100,
        inh: float = 17.5,
        dt: float = 1.0,
        nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
        reduction: Optional[callable] = None,
        wmin: Optional[float] = 0.0,
        wmax: Optional[float] = 1.0,
        norm: float = 78.4,
        theta_plus: float = 0.05,
        tc_theta_decay: float = 1e7,
        inpt_shape: Optional[Iterable[int]] = None,
    ) -> None:
        # language=rst
        """
        Constructor for class ``DiehlAndCook2015v2``.

        :param n_inpt: Number of input neurons. Matches the 1D size of the input data.
        :param n_neurons: Number of excitatory, inhibitory neurons.
        :param inh: Strength of synapse weights from inhibitory to excitatory layer.
        :param dt: Simulation time step.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events,
            respectively.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param wmin: Minimum allowed weight on input to excitatory synapses.
        :param wmax: Maximum allowed weight on input to excitatory synapses.
        :param norm: Input to excitatory layer connection weights normalization
            constant.
        :param theta_plus: On-spike increment of ``DiehlAndCookNodes`` membrane
```

```
            threshold potential.
        :param tc_theta_decay: Time constant of ``DiehlAndCookNodes
    `` threshold
            potential decay.
        :param inpt_shape: The dimensionality of the input layer.
        """
        super().__init__(dt=dt)

        self.n_inpt = n_inpt
        self.inpt_shape = inpt_shape
        self.n_neurons = n_neurons
        self.inh = inh
        self.dt = dt

        input_layer = Input(
            n=self.n_inpt, shape=self.inpt_shape, traces=True,
    tc_trace=20.0
        )
        self.add_layer(input_layer, name="X")

        output_layer = DiehlAndCookNodes(
            n=self.n_neurons,
            traces=True,
            rest=-65.0,
            reset=-60.0,
            thresh=-52.0,
            refrac=5,
            tc_decay=100.0,
            tc_trace=20.0,
            theta_plus=theta_plus,
            tc_theta_decay=tc_theta_decay,
        )
        self.add_layer(output_layer, name="Y")

        w = 0.3 * torch.rand(self.n_inpt, self.n_neurons)
        input_connection = Connection(
            source=self.layers["X"],
            target=self.layers["Y"],
            w=w,
            update_rule=PostPre,
            nu=nu,
            reduction=reduction,
            wmin=wmin,
            wmax=wmax,
            norm=norm,
        )
        self.add_connection(input_connection, source="X", target="Y"
    )

        w = -self.inh * (
            torch.ones(self.n_neurons, self.n_neurons)
            - torch.diag(torch.ones(self.n_neurons))
        )
        recurrent_connection = Connection(
```

```
            source=self.layers["Y"],
            target=self.layers["Y"],
            w=w,
            wmin=-self.inh,
            wmax=0,
        )
        self.add_connection(recurrent_connection, source="Y", target
    ="Y")


class IncreasingInhibitionNetwork(Network):
    # language=rst
    """
    Implements the inhibitory layer structure of the spiking neural
    network architecture
    from `(Hazan et al. 2018) <https://arxiv.org/abs/1807.09374>`_
    """

    def __init__(
        self,
        n_input: int,
        n_neurons: int = 100,
        start_inhib: float = 1.0,
        max_inhib: float = 100.0,
        dt: float = 1.0,
        nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
        reduction: Optional[callable] = None,
        wmin: float = 0.0,
        wmax: float = 1.0,
        norm: float = 78.4,
        theta_plus: float = 0.05,
        tc_theta_decay: float = 1e7,
    ) -> None:
        # language=rst
        """
        Constructor for class ``IncreasingInhibitionNetwork``.

        :param n_inpt: Number of input neurons. Matches the 1D size
    of the input data.
        :param n_neurons: Number of excitatory, inhibitory neurons.
        :param inh: Strength of synapse weights from inhibitory to
    excitatory layer.
        :param dt: Simulation time step.
        :param nu: Single or pair of learning rates for pre- and
    post-synaptic events,
            respectively.
        :param reduction: Method for reducing parameter updates
    along the minibatch
            dimension.
        :param wmin: Minimum allowed weight on input to excitatory
    synapses.
        :param wmax: Maximum allowed weight on input to excitatory
    synapses.
        :param norm: Input to excitatory layer connection weights
```

```
   normalization
            constant.
        :param theta_plus: On-spike increment of ``DiehlAndCookNodes
   `` membrane
            threshold potential.
        :param tc_theta_decay: Time constant of ``DiehlAndCookNodes
   `` threshold
            potential decay.
        """
        super().__init__(dt=dt)

        self.n_input = n_input
        self.n_neurons = n_neurons
        self.n_sqrt = int(np.sqrt(n_neurons))
        self.start_inhib = start_inhib
        self.max_inhib = max_inhib
        self.dt = dt

        input_layer = Input(n=self.n_input, traces=True, tc_trace
   =20.0)
        self.add_layer(input_layer, name="X")

        output_layer = DiehlAndCookNodes(
            n=self.n_neurons,
            traces=True,
            rest=-65.0,
            reset=-60.0,
            thresh=-52.0,
            refrac=5,
            tc_decay=100.0,
            tc_trace=20.0,
            theta_plus=theta_plus,
            tc_theta_decay=tc_theta_decay,
        )
        self.add_layer(output_layer, name="Y")

        w = 0.3 * torch.rand(self.n_input, self.n_neurons)
        input_output_conn = Connection(
            source=self.layers["X"],
            target=self.layers["Y"],
            w=w,
            update_rule=PostPre,
            nu=nu,
            reduction=reduction,
            wmin=wmin,
            wmax=wmax,
            norm=norm,
        )
        self.add_connection(input_output_conn, source="X", target="Y
   ")

        w = torch.zeros(self.n_neurons, self.n_neurons)
        for i in range(self.n_neurons):
            for j in range(self.n_neurons):
```

```
                if i != j:
                    x1, y1 = i // self.n_sqrt, i % self.n_sqrt
                    x2, y2 = j // self.n_sqrt, j % self.n_sqrt

                    inhib = self.start_inhib * np.sqrt(euclidean([x1, y1], [x2, y2]))
                    w[i, j] = -min(self.max_inhib, inhib)

        recurrent_output_conn = Connection(
            source=self.layers["Y"],
            target=self.layers["Y"],
            w=w,
            wmin=-self.max_inhib,
            wmax=0,
        )
        self.add_connection(recurrent_output_conn, source="Y", target="Y")


class LocallyConnectedNetwork(Network):
    # language=rst
    """
    Defines a two-layer network in which the input layer is "locally connected" to the
    output layer, and the output layer is recurrently inhibited connected such that
    neurons with the same input receptive field inhibit each other.
    """

    def __init__(
        self,
        n_inpt: int,
        input_shape: List[int],
        kernel_size: Union[int, Tuple[int, int]],
        stride: Union[int, Tuple[int, int]],
        n_filters: int,
        inh: float = 25.0,
        dt: float = 1.0,
        nu: Optional[Union[float, Sequence[float]]] = (1e-4, 1e-2),
        reduction: Optional[callable] = None,
        theta_plus: float = 0.05,
        tc_theta_decay: float = 1e7,
        wmin: float = 0.0,
        wmax: float = 1.0,
        norm: Optional[float] = 0.2,
    ) -> None:
        # language=rst
        """
        Constructor for class ``LocallyConnectedNetwork``. Uses ``DiehlAndCookNodes`` to
        avoid multiple spikes per timestep in the output layer population.

        :param n_inpt: Number of input neurons. Matches the 1D size
```

```
of the input data.
        :param input_shape: Two-dimensional shape of input population.
        :param kernel_size: Size of input windows. Integer or two-tuple of integers.
        :param stride: Length of horizontal, vertical stride across input space. Integer
            or two-tuple of integers.
        :param n_filters: Number of locally connected filters per input region. Integer
            or two-tuple of integers.
        :param inh: Strength of synapse weights from output layer back onto itself.
        :param dt: Simulation time step.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events,
            respectively.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param wmin: Minimum allowed weight on ``Input`` to ``DiehlAndCookNodes``
            synapses.
        :param wmax: Maximum allowed weight on ``Input`` to ``DiehlAndCookNodes``
            synapses.
        :param theta_plus: On-spike increment of ``DiehlAndCookNodes`` membrane
            threshold potential.
        :param tc_theta_decay: Time constant of ``DiehlAndCookNodes`` threshold
            potential decay.
        :param norm: ``Input`` to ``DiehlAndCookNodes`` layer connection weights
            normalization constant.
        """
        super().__init__(dt=dt)

        kernel_size = _pair(kernel_size)
        stride = _pair(stride)

        self.n_inpt = n_inpt
        self.input_shape = input_shape
        self.kernel_size = kernel_size
        self.stride = stride
        self.n_filters = n_filters
        self.inh = inh
        self.dt = dt
        self.theta_plus = theta_plus
        self.tc_theta_decay = tc_theta_decay
        self.wmin = wmin
        self.wmax = wmax
        self.norm = norm

```

```
        if kernel_size == input_shape:
            conv_size = [1, 1]
        else:
            conv_size = (
                int((input_shape[0] - kernel_size[0]) / stride[0]) + 1,
                int((input_shape[1] - kernel_size[1]) / stride[1]) + 1,
            )

        input_layer = Input(n=self.n_inpt, traces=True, tc_trace=20.0)

        output_layer = DiehlAndCookNodes(
            n=self.n_filters * conv_size[0] * conv_size[1],
            traces=True,
            rest=-65.0,
            reset=-60.0,
            thresh=-52.0,
            refrac=5,
            tc_decay=100.0,
            tc_trace=20.0,
            theta_plus=theta_plus,
            tc_theta_decay=tc_theta_decay,
        )
        input_output_conn = LocalConnection(
            input_layer,
            output_layer,
            kernel_size=kernel_size,
            stride=stride,
            n_filters=n_filters,
            nu=nu,
            reduction=reduction,
            update_rule=PostPre,
            wmin=wmin,
            wmax=wmax,
            norm=norm,
            input_shape=input_shape,
        )

        w = torch.zeros(n_filters, *conv_size, n_filters, *conv_size)
        for fltr1 in range(n_filters):
            for fltr2 in range(n_filters):
                if fltr1 != fltr2:
                    for i in range(conv_size[0]):
                        for j in range(conv_size[1]):
                            w[fltr1, i, j, fltr2, i, j] = -inh

        w = w.view(
            n_filters * conv_size[0] * conv_size[1],
            n_filters * conv_size[0] * conv_size[1],
        )
        recurrent_conn = Connection(output_layer, output_layer, w=w)
```

```
        self.add_layer(input_layer, name="X")
        self.add_layer(output_layer, name="Y")
        self.add_connection(input_output_conn, source="X", target="Y
    ")
        self.add_connection(recurrent_conn, source="Y", target="Y")
```

Listing B.22: *Models*

B.8 Learning

```
from .learning import (
    LearningRule,
    NoOp,
    PostPre,
    WeightDependentPostPre,
    Hebbian,
    MSTDP,
    MSTDPET,
    Rmax,
)
```

Listing B.23: *Initialization*

```
from abc import ABC
from typing import Union, Optional, Sequence

import torch
import numpy as np

from ..network.nodes import SRM0Nodes
from ..network.topology import (
    AbstractConnection,
    Connection,
    Conv2dConnection,
    LocalConnection,
)
from ..utils import im2col_indices


class LearningRule(ABC):
    # language=rst
    """
    Abstract base class for learning rules.
    """

    def __init__(
        self,
        connection: AbstractConnection,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
```

```
        # language=rst
        """
        Abstract constructor for the ``LearningRule`` object.

        :param connection: An ``AbstractConnection`` object.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the batch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.
        """
        # Connection parameters.
        self.connection = connection
        self.source = connection.source
        self.target = connection.target

        self.wmin = connection.wmin
        self.wmax = connection.wmax

        # Learning rate(s).
        if nu is None:
            nu = [0.0, 0.0]
        elif isinstance(nu, float) or isinstance(nu, int):
            nu = [nu, nu]

        self.nu = nu

        # Parameter update reduction across minibatch dimension.
        if reduction is None:
            reduction = torch.mean

        self.reduction = reduction

        # Weight decay.
        self.weight_decay = weight_decay

    def update(self) -> None:
        # language=rst
        """
        Abstract method for a learning rule update.
        """
        # Implement weight decay.
        if self.weight_decay:
            self.connection.w -= self.weight_decay * self.connection.w

        # Bound weights.
        if (
            self.connection.wmin != -np.inf or self.connection.wmax != np.inf
        ) and not isinstance(self, NoOp):
            self.connection.w.clamp_(self.connection.wmin, self.
```

```
    connection.wmax)


class NoOp(LearningRule):
    # language=rst
    """
    Learning rule with no effect.
    """

    def __init__(
        self,
        connection: AbstractConnection,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Abstract constructor for the ``LearningRule`` object.

        :param connection: An ``AbstractConnection`` object.
        :param nu: Single or pair of learning rates for pre- and
    post-synaptic events.
        :param reduction: Method for reducing parameter updates
    along the batch
            dimension.
        :param weight_decay: Constant multiple to decay weights by
    on each iteration.
        """
        super().__init__(
            connection=connection,
            nu=nu,
            reduction=reduction,
            weight_decay=weight_decay,
            **kwargs
        )

    def update(self, **kwargs) -> None:
        # language=rst
        """
        Abstract method for a learning rule update.
        """
        super().update()


class PostPre(LearningRule):
    # language=rst
    """
    Simple STDP rule involving both pre- and post-synaptic spiking
    activity. By default,
    pre-synaptic update is negative and the post-synaptic update is
    positive.
    """
```

```python

    def __init__(
        self,
        connection: AbstractConnection,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Constructor for ``PostPre`` learning rule.

        :param connection: An ``AbstractConnection`` object whose weights the
            ``PostPre`` learning rule will modify.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the batch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.
        """
        super().__init__(
            connection=connection,
            nu=nu,
            reduction=reduction,
            weight_decay=weight_decay,
            **kwargs
        )

        assert (
            self.source.traces and self.target.traces
        ), "Both pre- and post-synaptic nodes must record spike traces."

        if isinstance(connection, (Connection, LocalConnection)):
            self.update = self._connection_update
        elif isinstance(connection, Conv2dConnection):
            self.update = self._conv2d_connection_update
        else:
            raise NotImplementedError(
                "This learning rule is not supported for this Connection type."
            )

    def _connection_update(self, **kwargs) -> None:
        # language=rst
        """
        Post-pre learning rule for ``Connection`` subclass of ``AbstractConnection``
        class.
        """
```

```
        batch_size = self.source.batch_size

        source_s = self.source.s.view(batch_size, -1).unsqueeze(2).float()
        source_x = self.source.x.view(batch_size, -1).unsqueeze(2)
        target_s = self.target.s.view(batch_size, -1).unsqueeze(1).float()
        target_x = self.target.x.view(batch_size, -1).unsqueeze(1)

        # Pre-synaptic update.
        if self.nu[0]:
            update = self.reduction(torch.bmm(source_s, target_x), dim=0)
            self.connection.w -= self.nu[0] * update

        # Post-synaptic update.
        if self.nu[1]:
            update = self.reduction(torch.bmm(source_x, target_s), dim=0)
            self.connection.w += self.nu[1] * update

        super().update()

    def _conv2d_connection_update(self, **kwargs) -> None:
        # language=rst
        """
        Post-pre learning rule for ``Conv2dConnection`` subclass of
        ``AbstractConnection`` class.
        """
        # Get convolutional layer parameters.
        out_channels, _, kernel_height, kernel_width = self.connection.w.size()
        padding, stride = self.connection.padding, self.connection.stride
        batch_size = self.source.batch_size

        # Reshaping spike traces and spike occurrences.
        source_x = im2col_indices(
            self.source.x, kernel_height, kernel_width, padding=padding, stride=stride
        )
        target_x = self.target.x.view(batch_size, out_channels, -1)
        source_s = im2col_indices(
            self.source.s.float(),
            kernel_height,
            kernel_width,
            padding=padding,
            stride=stride,
        )
        target_s = self.target.s.view(batch_size, out_channels, -1).float()

        # Pre-synaptic update.
        if self.nu[0]:
```

```python
            pre = self.reduction(
                torch.bmm(target_x, source_s.permute((0, 2, 1))), dim=0
            )
            self.connection.w -= self.nu[0] * pre.view(self.connection.w.size())

        # Post-synaptic update.
        if self.nu[1]:
            post = self.reduction(
                torch.bmm(target_s, source_x.permute((0, 2, 1))), dim=0
            )
            self.connection.w += self.nu[1] * post.view(self.connection.w.size())

        super().update()


class WeightDependentPostPre(LearningRule):
    # language=rst
    """
    STDP rule involving both pre- and post-synaptic spiking activity. The post-synaptic
    update is positive and the pre- synaptic update is negative, and both are dependent
    on the magnitude of the synaptic weights.
    """

    def __init__(
        self,
        connection: AbstractConnection,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Constructor for ``WeightDependentPostPre`` learning rule.

        :param connection: An ``AbstractConnection`` object whose weights the
            ``WeightDependentPostPre`` learning rule will modify.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the batch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.
        """
        super().__init__(
            connection=connection,
```

```
            nu=nu,
            reduction=reduction,
            weight_decay=weight_decay,
            **kwargs
        )

        assert self.source.traces, "Pre-synaptic nodes must record spike traces."
        assert (
            connection.wmin != -np.inf and connection.wmax != np.inf
        ), "Connection must define finite wmin and wmax."

        self.wmin = connection.wmin
        self.wmax = connection.wmax

        if isinstance(connection, (Connection, LocalConnection)):
            self.update = self._connection_update
        elif isinstance(connection, Conv2dConnection):
            self.update = self._conv2d_connection_update
        else:
            raise NotImplementedError(
                "This learning rule is not supported for this Connection type."
            )

    def _connection_update(self, **kwargs) -> None:
        # language=rst
        """
        Post-pre learning rule for ``Connection`` subclass of ``AbstractConnection``
        class.
        """
        batch_size = self.source.batch_size

        source_s = self.source.s.view(batch_size, -1).unsqueeze(2).float()
        source_x = self.source.x.view(batch_size, -1).unsqueeze(2)
        target_s = self.target.s.view(batch_size, -1).unsqueeze(1).float()
        target_x = self.target.x.view(batch_size, -1).unsqueeze(1)

        update = 0

        # Pre-synaptic update.
        if self.nu[0]:
            outer_product = self.reduction(torch.bmm(source_s, target_x), dim=0)
            update -= self.nu[0] * outer_product * (self.connection.w - self.wmin)

        # Post-synaptic update.
        if self.nu[1]:
            outer_product = self.reduction(torch.bmm(source_x, target_s), dim=0)
```

```python
            update += self.nu[1] * outer_product * (self.wmax - self.connection.w)

        self.connection.w += update

        super().update()

    def _conv2d_connection_update(self, **kwargs) -> None:
        # language=rst
        """
        Post-pre learning rule for ``Conv2dConnection`` subclass of
        ``AbstractConnection`` class.
        """
        # Get convolutional layer parameters.
        (
            out_channels,
            in_channels,
            kernel_height,
            kernel_width,
        ) = self.connection.w.size()
        padding, stride = self.connection.padding, self.connection.stride
        batch_size = self.source.batch_size

        # Reshaping spike traces and spike occurrences.
        source_x = im2col_indices(
            self.source.x, kernel_height, kernel_width, padding=padding, stride=stride
        )
        target_x = self.target.x.view(batch_size, out_channels, -1)
        source_s = im2col_indices(
            self.source.s.float(),
            kernel_height,
            kernel_width,
            padding=padding,
            stride=stride,
        )
        target_s = self.target.s.view(batch_size, out_channels, -1).float()

        update = 0

        # Pre-synaptic update.
        if self.nu[0]:
            pre = self.reduction(
                torch.bmm(target_x, source_s.permute((0, 2, 1))), dim=0
            )
            update -= (
                self.nu[0]
                * pre.view(self.connection.w.size())
                * (self.connection.w - self.wmin)
            )

```

```python
        # Post-synaptic update.
        if self.nu[1]:
            post = self.reduction(
                torch.bmm(target_s, source_x.permute((0, 2, 1))), dim=0
            )
            update += (
                self.nu[1]
                * post.view(self.connection.w.size())
                * (self.wmax - self.connection.wmin)
            )

        self.connection.w += update

        super().update()


class Hebbian(LearningRule):
    # language=rst
    """
    Simple Hebbian learning rule. Pre- and post-synaptic updates are both positive.
    """

    def __init__(
        self,
        connection: AbstractConnection,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Constructor for ``Hebbian`` learning rule.

        :param connection: An ``AbstractConnection`` object whose weights the
            ``Hebbian`` learning rule will modify.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events.
        :param reduction: Method for reducing parameter updates along the batch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.
        """
        super().__init__(
            connection=connection,
            nu=nu,
            reduction=reduction,
            weight_decay=weight_decay,
            **kwargs
        )
```

```
        assert (
            self.source.traces and self.target.traces
        ), "Both pre- and post-synaptic nodes must record spike traces."

        if isinstance(connection, (Connection, LocalConnection)):
            self.update = self._connection_update
        elif isinstance(connection, Conv2dConnection):
            self.update = self._conv2d_connection_update
        else:
            raise NotImplementedError(
                "This learning rule is not supported for this Connection type."
            )

    def _connection_update(self, **kwargs) -> None:
        # language=rst
        """
        Hebbian learning rule for ``Connection`` subclass of ``AbstractConnection``
        class.
        """
        batch_size = self.source.batch_size

        source_s = self.source.s.view(batch_size, -1).unsqueeze(2).float()
        source_x = self.source.x.view(batch_size, -1).unsqueeze(2)
        target_s = self.target.s.view(batch_size, -1).unsqueeze(1).float()
        target_x = self.target.x.view(batch_size, -1).unsqueeze(1)

        # Pre-synaptic update.
        update = self.reduction(torch.bmm(source_s, target_x), dim=0)
        self.connection.w += self.nu[0] * update

        # Post-synaptic update.
        update = self.reduction(torch.bmm(source_x, target_s), dim=0)
        self.connection.w += self.nu[1] * update

        super().update()

    def _conv2d_connection_update(self, **kwargs) -> None:
        # language=rst
        """
        Hebbian learning rule for ``Conv2dConnection`` subclass of
        ``AbstractConnection`` class.
        """
        out_channels, _, kernel_height, kernel_width = self.connection.w.size()
        padding, stride = self.connection.padding, self.connection.stride
```

```
        batch_size = self.source.batch_size

        # Reshaping spike traces and spike occurrences.
        source_x = im2col_indices(
            self.source.x, kernel_height, kernel_width, padding=padding, stride=stride
        )
        target_x = self.target.x.view(batch_size, out_channels, -1)
        source_s = im2col_indices(
            self.source.s.float(),
            kernel_height,
            kernel_width,
            padding=padding,
            stride=stride,
        )
        target_s = self.target.s.view(batch_size, out_channels, -1).float()

        # Pre-synaptic update.
        pre = self.reduction(torch.bmm(target_x, source_s.permute((0, 2, 1))), dim=0)
        self.connection.w += self.nu[0] * pre.view(self.connection.w.size())

        # Post-synaptic update.
        post = self.reduction(torch.bmm(target_s, source_x.permute((0, 2, 1))), dim=0)
        self.connection.w += self.nu[1] * post.view(self.connection.w.size())

        super().update()


class MSTDP(LearningRule):
    # language=rst
    """
    Reward-modulated STDP. Adapted from `(Florian 2007)
    <https://florian.io/papers/2007_Florian_Modulated_STDP.pdf>`_.
    """

    def __init__(
        self,
        connection: AbstractConnection,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Constructor for ``MSTDP`` learning rule.

        :param connection: An ``AbstractConnection`` object whose weights the ``MSTDP``
```

```python
                learning rule will modify.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events,
                respectively.
        :param reduction: Method for reducing parameter updates along the minibatch
                dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:

        :param tc_plus: Time constant for pre-synaptic firing trace.
        :param tc_minus: Time constant for post-synaptic firing trace.
        """
        super().__init__(
            connection=connection,
            nu=nu,
            reduction=reduction,
            weight_decay=weight_decay,
            **kwargs
        )

        if isinstance(connection, (Connection, LocalConnection)):
            self.update = self._connection_update
        elif isinstance(connection, Conv2dConnection):
            self.update = self._conv2d_connection_update
        else:
            raise NotImplementedError(
                "This learning rule is not supported for this Connection type."
            )

        self.tc_plus = torch.tensor(kwargs.get("tc_plus", 20.0))
        self.tc_minus = torch.tensor(kwargs.get("tc_minus", 20.0))

    def _connection_update(self, **kwargs) -> None:
        # language=rst
        """
        MSTDP learning rule for ``Connection`` subclass of ``AbstractConnection`` class.

        Keyword arguments:

        :param Union[float, torch.Tensor] reward: Reward signal from reinforcement
                learning task.
        :param float a_plus: Learning rate (post-synaptic).
        :param float a_minus: Learning rate (pre-synaptic).
        """
        batch_size = self.source.batch_size

        # Initialize eligibility, P^+, and P^-.
```

```
        if not hasattr(self, "p_plus"):
            self.p_plus = torch.zeros(batch_size, *self.source.shape)
        if not hasattr(self, "p_minus"):
            self.p_minus = torch.zeros(batch_size, *self.target.shape)
        if not hasattr(self, "eligibility"):
            self.eligibility = torch.zeros(batch_size, *self.connection.w.shape)

        # Reshape pre- and post-synaptic spikes.
        source_s = self.source.s.view(batch_size, -1).float()
        target_s = self.target.s.view(batch_size, -1).float()

        # Parse keyword arguments.
        reward = kwargs["reward"]
        a_plus = torch.tensor(kwargs.get("a_plus", 1.0))
        a_minus = torch.tensor(kwargs.get("a_minus", -1.0))

        # Compute weight update based on the eligibility value of the past timestep.
        update = reward * self.eligibility
        self.connection.w += self.nu[0] * self.reduction(update, dim=0)

        # Update P^+ and P^- values.
        self.p_plus *= torch.exp(-self.connection.dt / self.tc_plus)
        self.p_plus += a_plus * source_s
        self.p_minus *= torch.exp(-self.connection.dt / self.tc_minus)
        self.p_minus += a_minus * target_s

        # Calculate point eligibility value.
        self.eligibility = torch.bmm(
            self.p_plus.unsqueeze(2), target_s.unsqueeze(1)
        ) + torch.bmm(source_s.unsqueeze(2), self.p_minus.unsqueeze(1))

        super().update()

    def _conv2d_connection_update(self, **kwargs) -> None:
        # language=rst
        """
        MSTDP learning rule for ``Conv2dConnection`` subclass of ``AbstractConnection``
        class.

        Keyword arguments:

        :param Union[float, torch.Tensor] reward: Reward signal from reinforcement
            learning task.
        :param float a_plus: Learning rate (post-synaptic).
        :param float a_minus: Learning rate (pre-synaptic).
```

```
        """
        batch_size = self.source.batch_size

        # Initialize eligibility.
        if not hasattr(self, "eligibility"):
            self.eligibility = torch.zeros(batch_size, *self.connection.w.shape)

        # Parse keyword arguments.
        reward = kwargs["reward"]
        a_plus = torch.tensor(kwargs.get("a_plus", 1.0))
        a_minus = torch.tensor(kwargs.get("a_minus", -1.0))

        batch_size = self.source.batch_size

        # Compute weight update based on the eligibility value of the past timestep.
        update = reward * self.eligibility
        self.connection.w += self.nu[0] * torch.sum(update, dim=0)

        out_channels, _, kernel_height, kernel_width = self.connection.w.size()
        padding, stride = self.connection.padding, self.connection.stride

        # Initialize P^+ and P^-.
        if not hasattr(self, "p_plus"):
            self.p_plus = torch.zeros(batch_size, *self.source.shape)
            self.p_plus = im2col_indices(
                self.p_plus, kernel_height, kernel_width, padding=padding, stride=stride
            )
        if not hasattr(self, "p_minus"):
            self.p_minus = torch.zeros(batch_size, *self.target.shape)
            self.p_minus = self.p_minus.view(batch_size, out_channels, -1).float()

        # Reshaping spike occurrences.
        source_s = im2col_indices(
            self.source.s.float(),
            kernel_height,
            kernel_width,
            padding=padding,
            stride=stride,
        )
        target_s = self.target.s.view(batch_size, out_channels, -1).float()

        # Update P^+ and P^- values.
        self.p_plus *= torch.exp(-self.connection.dt / self.tc_plus)
        self.p_plus += a_plus * source_s
        self.p_minus *= torch.exp(-self.connection.dt / self.
```

```
tc_minus)
        self.p_minus += a_minus * target_s

        # Calculate point eligibility value.
        self.eligibility = torch.bmm(
            target_s, self.p_plus.permute((0, 2, 1))
        ) + torch.bmm(self.p_minus, source_s.permute((0, 2, 1)))
        self.eligibility = self.eligibility.view(self.connection.w.size())

        super().update()


class MSTDPET(LearningRule):
    # language=rst
    """
    Reward-modulated STDP with eligibility trace. Adapted from
    `(Florian 2007) <https://florian.io/papers/2007_Florian_Modulated_STDP.pdf>`_.
    """

    def __init__(
        self,
        connection: AbstractConnection,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Constructor for ``MSTDPET`` learning rule.

        :param connection: An ``AbstractConnection`` object whose weights the
            ``MSTDPET`` learning rule will modify.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events,
            respectively.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:

        :param float tc_plus: Time constant for pre-synaptic firing trace.
        :param float tc_minus: Time constant for post-synaptic firing trace.
        :param float tc_e_trace: Time constant for the eligibility trace.
        """
```

```
        super().__init__(
            connection=connection,
            nu=nu,
            reduction=reduction,
            weight_decay=weight_decay,
            **kwargs
        )

        if isinstance(connection, (Connection, LocalConnection)):
            self.update = self._connection_update
        elif isinstance(connection, Conv2dConnection):
            self.update = self._conv2d_connection_update
        else:
            raise NotImplementedError(
                "This learning rule is not supported for this Connection type."
            )

        self.tc_plus = torch.tensor(kwargs.get("tc_plus", 20.0))
        self.tc_minus = torch.tensor(kwargs.get("tc_minus", 20.0))
        self.tc_e_trace = torch.tensor(kwargs.get("tc_e_trace", 25.0))

    def _connection_update(self, **kwargs) -> None:
        # language=rst
        """
        MSTDPET learning rule for ``Connection`` subclass of ``AbstractConnection``
        class.

        Keyword arguments:

        :param Union[float, torch.Tensor] reward: Reward signal from reinforcement
            learning task.
        :param float a_plus: Learning rate (post-synaptic).
        :param float a_minus: Learning rate (pre-synaptic).
        """
        # Initialize eligibility, eligibility trace, P^+, and P^-.
        if not hasattr(self, "p_plus"):
            self.p_plus = torch.zeros(self.source.n)
        if not hasattr(self, "p_minus"):
            self.p_minus = torch.zeros(self.target.n)
        if not hasattr(self, "eligibility"):
            self.eligibility = torch.zeros(*self.connection.w.shape)
        if not hasattr(self, "eligibility_trace"):
            self.eligibility_trace = torch.zeros(*self.connection.w.shape)

        # Reshape pre- and post-synaptic spikes.
        source_s = self.source.s.view(-1).float()
        target_s = self.target.s.view(-1).float()

        # Parse keyword arguments.
```

```
        reward = kwargs["reward"]
        a_plus = torch.tensor(kwargs.get("a_plus", 1.0))
        a_minus = torch.tensor(kwargs.get("a_minus", -1.0))

        # Calculate value of eligibility trace based on the value
        # of the point eligibility value of the past timestep.
        self.eligibility_trace *= torch.exp(-self.connection.dt / self.tc_e_trace)
        self.eligibility_trace += self.eligibility / self.tc_e_trace

        # Compute weight update.
        self.connection.w += (
            self.nu[0] * self.connection.dt * reward * self.eligibility_trace
        )

        # Update P^+ and P^- values.
        self.p_plus *= torch.exp(-self.connection.dt / self.tc_plus)
        self.p_plus += a_plus * source_s
        self.p_minus *= torch.exp(-self.connection.dt / self.tc_minus)
        self.p_minus += a_minus * target_s

        # Calculate point eligibility value.
        self.eligibility = torch.ger(self.p_plus, target_s) + torch.ger(
            source_s, self.p_minus
        )

        super().update()

    def _conv2d_connection_update(self, **kwargs) -> None:
        # language=rst
        """
        MSTDPET learning rule for ``Conv2dConnection`` subclass of
        ``AbstractConnection`` class.

        Keyword arguments:

        :param Union[float, torch.Tensor] reward: Reward signal from reinforcement
            learning task.
        :param float a_plus: Learning rate (post-synaptic).
        :param float a_minus: Learning rate (pre-synaptic).
        """
        batch_size = self.source.batch_size

        # Initialize eligibility and eligibility trace.
        if not hasattr(self, "eligibility"):
            self.eligibility = torch.zeros(batch_size, *self.connection.w.shape)
        if not hasattr(self, "eligibility_trace"):
            self.eligibility_trace = torch.zeros(batch_size, *self.connection.w.shape)
```

```

        # Parse keyword arguments.
        reward = kwargs["reward"]
        a_plus = torch.tensor(kwargs.get("a_plus", 1.0))
        a_minus = torch.tensor(kwargs.get("a_minus", -1.0))

        # Calculate value of eligibility trace based on the value
        # of the point eligibility value of the past timestep.
        self.eligibility_trace *= torch.exp(-self.connection.dt / self.tc_e_trace)

        # Compute weight update.
        update = reward * self.eligibility_trace
        self.connection.w += self.nu[0] * self.connection.dt * torch.sum(update, dim=0)

        out_channels, _, kernel_height, kernel_width = self.connection.w.size()
        padding, stride = self.connection.padding, self.connection.stride

        # Initialize P^+ and P^-.
        if not hasattr(self, "p_plus"):
            self.p_plus = torch.zeros(batch_size, *self.source.shape)
            self.p_plus = im2col_indices(
                self.p_plus, kernel_height, kernel_width, padding=padding, stride=stride
            )
        if not hasattr(self, "p_minus"):
            self.p_minus = torch.zeros(batch_size, *self.target.shape)
            self.p_minus = self.p_minus.view(batch_size, out_channels, -1).float()

        # Reshaping spike occurrences.
        source_s = im2col_indices(
            self.source.s.float(),
            kernel_height,
            kernel_width,
            padding=padding,
            stride=stride,
        )
        target_s = (
            self.target.s.permute(1, 2, 3, 0).view(batch_size, out_channels, -1).float()
        )

        # Update P^+ and P^- values.
        self.p_plus *= torch.exp(-self.connection.dt / self.tc_plus)
        self.p_plus += a_plus * source_s
        self.p_minus *= torch.exp(-self.connection.dt / self.tc_minus)
        self.p_minus += a_minus * target_s
```

```python

        # Calculate point eligibility value.
        self.eligibility = torch.bmm(
            target_s, self.p_plus.permute((0, 2, 1))
        ) + torch.bmm(self.p_minus, source_s.permute((0, 2, 1)))
        self.eligibility = self.eligibility.view(self.connection.w.size())

        super().update()


class Rmax(LearningRule):
    # language=rst
    """
    Reward-modulated learning rule derived from reward maximization principles. Adapted
    from `(Vasilaki et al., 2009)
    <https://intranet.physio.unibe.ch/Publikationen/Dokumente/Vasilaki2009PloSComputBio_1.pdf>`_.
    """

    def __init__(
        self,
        connection: AbstractConnection,
        nu: Optional[Union[float, Sequence[float]]] = None,
        reduction: Optional[callable] = None,
        weight_decay: float = 0.0,
        **kwargs
    ) -> None:
        # language=rst
        """
        Constructor for ``R-max`` learning rule.

        :param connection: An ``AbstractConnection`` object whose weights the ``R-max``
            learning rule will modify.
        :param nu: Single or pair of learning rates for pre- and post-synaptic events,
            respectively.
        :param reduction: Method for reducing parameter updates along the minibatch
            dimension.
        :param weight_decay: Constant multiple to decay weights by on each iteration.

        Keyword arguments:

        :param float tc_c: Time constant for balancing naive Hebbian and policy gradient
            learning.
        :param float tc_e_trace: Time constant for the eligibility trace.
        """
        super().__init__(
```

```
            connection=connection,
            nu=nu,
            reduction=reduction,
            weight_decay=weight_decay,
            **kwargs
        )

        # Trace is needed for computing epsilon.
        assert (
            self.source.traces and self.source.traces_additive
        ), "Pre-synaptic nodes must use additive spike traces."

        # Derivation of R-max depends on stochastic SRM neurons!
        assert isinstance(
            self.target, SRM0Nodes
        ), "R-max needs stochastically firing neurons, use SRM0Nodes."

        if isinstance(connection, (Connection, LocalConnection)):
            self.update = self._connection_update
        else:
            raise NotImplementedError(
                "This learning rule is not supported for this Connection type."
            )

        self.tc_c = torch.tensor(
            kwargs.get("tc_c", 5.0)
        )  # 0 for pure naive Hebbian, inf for pure policy gradient.
        self.tc_e_trace = torch.tensor(kwargs.get("tc_e_trace", 25.0))

    def _connection_update(self, **kwargs) -> None:
        # language=rst
        """
        R-max learning rule for ``Connection`` subclass of ``AbstractConnection`` class.

        Keyword arguments:

        :param Union[float, torch.Tensor] reward: Reward signal from reinforcement
            learning task.
        """
        # Initialize eligibility trace.
        if not hasattr(self, "eligibility_trace"):
            self.eligibility_trace = torch.zeros(*self.connection.w.shape)

        # Reshape variables.
        target_s = self.target.s.view(-1).float()
        target_s_prob = self.target.s_prob.view(-1)
        source_x = self.source.x.view(-1)

```

```
        # Parse keyword arguments.
        reward = kwargs["reward"]

        # New eligibility trace.
        self.eligibility_trace *= 1 - self.connection.dt / self.
    tc_e_trace
        self.eligibility_trace += (
            target_s
            - (target_s_prob / (1.0 + self.tc_c / self.connection.dt
     * target_s_prob))
        ) * source_x[:, None]

        # Compute weight update.
        self.connection.w += self.nu[0] * reward * self.
    eligibility_trace

        super().update()
```

Listing B.24: *Learning*

```
from abc import ABC, abstractmethod

import torch


class AbstractReward(ABC):
    # language=rst
    """
    Abstract base class for reward computation.
    """

    @abstractmethod
    def compute(self, **kwargs) -> None:
        # language=rst
        """
        Computes/modifies reward.
        """
        pass

    @abstractmethod
    def update(self, **kwargs) -> None:
        # language=rst
        """
        Updates internal variables needed to modify reward. Usually
    called once per
        episode.
        """
        pass


class MovingAvgRPE(AbstractReward):
    # language=rst
    """
    Computes reward prediction error (RPE) based on an exponential
```

```
    moving average (EMA)
    of past rewards.
    """

    def __init__(self, **kwargs) -> None:
        # language=rst
        """
        Constructor for EMA reward prediction error.
        """
        self.reward_predict = torch.tensor(0.0)  # Predicted reward (per step).
        self.reward_predict_episode = torch.tensor(0.0)  # Predicted reward per episode.
        self.rewards_predict_episode = (
            []
        )  # List of predicted rewards per episode (used for plotting).

    def compute(self, **kwargs) -> torch.Tensor:
        # language=rst
        """
        Computes the reward prediction error using EMA.

        Keyword arguments:

        :param Union[float, torch.Tensor] reward: Current reward.
        :return: Reward prediction error.
        """
        # Get keyword arguments.
        reward = kwargs["reward"]

        return reward - self.reward_predict

    def update(self, **kwargs) -> None:
        # language=rst
        """
        Updates the EMAs. Called once per episode.

        Keyword arguments:

        :param Union[float, torch.Tensor] accumulated_reward: Reward accumulated over
            one episode.
        :param int steps: Steps in that episode.
        :param float ema_window: Width of the averaging window.
        """
        # Get keyword arguments.
        accumulated_reward = kwargs["accumulated_reward"]
        steps = torch.tensor(kwargs["steps"]).float()
        ema_window = torch.tensor(kwargs.get("ema_window", 10.0))

        # Compute average reward per step.
        reward = accumulated_reward / steps

```

```
        # Update EMAs.
        self.reward_predict = (
            1 - 1 / ema_window
        ) * self.reward_predict + 1 / ema_window * reward
        self.reward_predict_episode = (
            1 - 1 / ema_window
        ) * self.reward_predict_episode + 1 / ema_window *
    accumulated_reward
        self.rewards_predict_episode.append(self.
    reward_predict_episode.item())
```

Listing B.25: *Reward*

B.9 Evaluation

```
from .evaluation import (
    assign_labels,
    logreg_fit,
    logreg_predict,
    all_activity,
    proportion_weighting,
    ngram,
    update_ngram_scores,
)
```

Listing B.26: *Initialization*

```
from itertools import product
from typing import Optional, Tuple, Dict

import torch
from sklearn.linear_model import LogisticRegression


def assign_labels(
    spikes: torch.Tensor,
    labels: torch.Tensor,
    n_labels: int,
    rates: Optional[torch.Tensor] = None,
    alpha: float = 1.0,
) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    # language=rst
    """
    Assign labels to the neurons based on highest average spiking
    activity.

    :param spikes: Binary tensor of shape ``(n_samples, time,
    n_neurons)`` of a single
        layer's spiking activity.
    :param labels: Vector of shape ``(n_samples,)`` with data labels
    corresponding to
        spiking activity.
    :param n_labels: The number of target labels in the data.
```

```
    :param rates: If passed, these represent spike rates from a
   previous
        ``assign_labels()`` call.
    :param alpha: Rate of decay of label assignments.
    :return: Tuple of class assignments, per-class spike proportions
   , and per-class
        firing rates.
    """
    n_neurons = spikes.size(2)

    if rates is None:
        rates = torch.zeros(n_neurons, n_labels)

    # Sum over time dimension (spike ordering doesn't matter).
    spikes = spikes.sum(1)

    for i in range(n_labels):
        # Count the number of samples with this label.
        n_labeled = torch.sum(labels == i).float()

        if n_labeled > 0:
            # Get indices of samples with this label.
            indices = torch.nonzero(labels == i).view(-1)

            # Compute average firing rates for this label.
            rates[:, i] = alpha * rates[:, i] + (
                torch.sum(spikes[indices], 0) / n_labeled
            )

    # Compute proportions of spike activity per class.
    proportions = rates / rates.sum(1, keepdim=True)
    proportions[proportions != proportions] = 0  # Set NaNs to 0

    # Neuron assignments are the labels they fire most for.
    assignments = torch.max(proportions, 1)[1]

    return assignments, proportions, rates


def logreg_fit(
    spikes: torch.Tensor, labels: torch.Tensor, logreg:
   LogisticRegression
) -> LogisticRegression:
    # language=rst
    """
    (Re)fit logistic regression model to spike data summed over time
   .

    :param spikes: Summed (over time) spikes of shape ``(n_examples,
    time, n_neurons)``.
    :param labels: Vector of shape ``(n_samples,)`` with data labels
    corresponding to
        spiking activity.
    :param logreg: Logistic regression model from previous fits.
```

```python
    :return: (Re)fitted logistic regression model.
    """
    # (Re)fit logistic regression model.
    logreg.fit(spikes, labels)
    return logreg


def logreg_predict(spikes: torch.Tensor, logreg: LogisticRegression) -> torch.Tensor:
    # language=rst
    """
    Predicts classes according to spike data summed over time.

    :param spikes: Summed (over time) spikes of shape ``(n_examples, time, n_neurons)``.
    :param logreg: Logistic regression model from previous fits.
    :return: Predictions per example.
    """
    # Make class label predictions.
    if not hasattr(logreg, "coef_") or logreg.coef_ is None:
        return -1 * torch.ones(spikes.size(0)).long()

    predictions = logreg.predict(spikes)
    return torch.Tensor(predictions).long()


def all_activity(
    spikes: torch.Tensor, assignments: torch.Tensor, n_labels: int
) -> torch.Tensor:
    # language=rst
    """
    Classify data with the label with highest average spiking activity over all neurons.

    :param spikes: Binary tensor of shape ``(n_samples, time, n_neurons)`` of a layer's
        spiking activity.
    :param assignments: A vector of shape ``(n_neurons,)`` of neuron label assignments.
    :param n_labels: The number of target labels in the data.
    :return: Predictions tensor of shape ``(n_samples,)`` resulting from the "all
        activity" classification scheme.
    """
    n_samples = spikes.size(0)

    # Sum over time dimension (spike ordering doesn't matter).
    spikes = spikes.sum(1)

    rates = torch.zeros(n_samples, n_labels)
    for i in range(n_labels):
        # Count the number of neurons with this label assignment.
        n_assigns = torch.sum(assignments == i).float()

```

```
        if n_assigns > 0:
            # Get indices of samples with this label.
            indices = torch.nonzero(assignments == i).view(-1)

            # Compute layer-wise firing rate for this label.
            rates[:, i] = torch.sum(spikes[:, indices], 1) / n_assigns

    # Predictions are arg-max of layer-wise firing rates.
    return torch.sort(rates, dim=1, descending=True)[1][:, 0]


def proportion_weighting(
    spikes: torch.Tensor,
    assignments: torch.Tensor,
    proportions: torch.Tensor,
    n_labels: int,
) -> torch.Tensor:
    # language=rst
    """
    Classify data with the label with highest average spiking activity over all neurons,
    weighted by class-wise proportion.

    :param spikes: Binary tensor of shape ``(n_samples, time, n_neurons)`` of a single
        layer's spiking activity.
    :param assignments: A vector of shape ``(n_neurons,)`` of neuron label assignments.
    :param proportions: A matrix of shape ``(n_neurons, n_labels)`` giving the per-class
        proportions of neuron spiking activity.
    :param n_labels: The number of target labels in the data.
    :return: Predictions tensor of shape ``(n_samples,)`` resulting from the "proportion
        weighting" classification scheme.
    """
    n_samples = spikes.size(0)

    # Sum over time dimension (spike ordering doesn't matter).
    spikes = spikes.sum(1)

    rates = torch.zeros(n_samples, n_labels)
    for i in range(n_labels):
        # Count the number of neurons with this label assignment.
        n_assigns = torch.sum(assignments == i).float()

        if n_assigns > 0:
            # Get indices of samples with this label.
            indices = torch.nonzero(assignments == i).view(-1)

            # Compute layer-wise firing rate for this label.
            rates[:, i] += (
                torch.sum((proportions[:, i] * spikes)[:, indices],
```

```
    1) / n_assigns
            )

    # Predictions are arg-max of layer-wise firing rates.
    predictions = torch.sort(rates, dim=1, descending=True)[1][:, 0]

    return predictions


def ngram(
    spikes: torch.Tensor,
    ngram_scores: Dict[Tuple[int, ...], torch.Tensor],
    n_labels: int,
    n: int,
) -> torch.Tensor:
    # language=rst
    """
    Predicts between ``n_labels`` using ``ngram_scores``.

    :param spikes: Spikes of shape ``(n_examples, time, n_neurons)
    ``.
    :param ngram_scores: Previously recorded scores to update.
    :param n_labels: The number of target labels in the data.
    :param n: The max size of n-gram to use.
    :return: Predictions per example.
    """
    predictions = []
    for activity in spikes:
        score = torch.zeros(n_labels)

        # Aggregate all of the firing neurons' indices
        fire_order = []
        for t in range(activity.size()[0]):
            ordering = torch.nonzero(activity[t].view(-1))
            if ordering.numel() > 0:
                fire_order += ordering[:, 0].tolist()

        # Consider all n-gram sequences.
        for j in range(len(fire_order) - n):
            if tuple(fire_order[j : j + n]) in ngram_scores:
                score += ngram_scores[tuple(fire_order[j : j + n])]

        predictions.append(torch.argmax(score))

    return torch.tensor(predictions).long()


def update_ngram_scores(
    spikes: torch.Tensor,
    labels: torch.Tensor,
    n_labels: int,
    n: int,
    ngram_scores: Dict[Tuple[int, ...], torch.Tensor],
) -> Dict[Tuple[int, ...], torch.Tensor]:
```

```
    # language=rst
    """
    Updates ngram scores by adding the count of each spike sequence of length n from the
    past ``n_examples``.

    :param spikes: Spikes of shape ``(n_examples, time, n_neurons)``.
    :param labels: The ground truth labels of shape ``(n_examples)``.
    :param n_labels: The number of target labels in the data.
    :param n: The max size of n-gram to use.
    :param ngram_scores: Previously recorded scores to update.
    :return: Dictionary mapping n-grams to vectors of per-class spike counts.
    """
    for i, activity in enumerate(spikes):
        # Obtain firing order for spiking activity.
        fire_order = []

        # Aggregate all of the firing neurons' indices.
        for t in range(spikes.size(1)):
            # Gets the indices of the neurons which fired on this timestep.
            ordering = torch.nonzero(activity[t]).view(-1)
            if ordering.numel() > 0:  # If there was more than one spike...
                # Add the indices of spiked neurons to the fire ordering.
                ordering = ordering.tolist()
                fire_order.append(ordering)

        # Check every sequence of length n.
        for order in zip(*(fire_order[k:] for k in range(n))):
            for sequence in product(*order):
                if sequence not in ngram_scores:
                    ngram_scores[sequence] = torch.zeros(n_labels)

                ngram_scores[sequence][int(labels[i])] += 1

    return ngram_scores
```

Listing B.27: *Evaluation*

B.10 Analysis

```
from . import plotting, visualization, pipeline_analysis
```

Listing B.28: *Initialization*

```
from abc import ABC, abstractmethod
from typing import Dict, Optional

import matplotlib.pyplot as plt
```

```python
import numpy as np
import pandas as pd
import torch
from tensorboardX import SummaryWriter
from torchvision.utils import make_grid

from .plotting import plot_spikes, plot_voltages, plot_conv2d_weights
from ..utils import reshape_conv2d_weights


class PipelineAnalyzer(ABC):
    # language=rst
    """
    Responsible for pipeline analysis. Subclasses maintain state
    information related to plotting or logging.
    """

    @abstractmethod
    def finalize_step(self) -> None:
        # language=rst
        """
        Flush the output from the current step.
        """
        pass

    @abstractmethod
    def plot_obs(self, obs: torch.Tensor, tag: str = "obs", step: int = None) -> None:
        # language=rst
        """
        Pulls the observation from PyTorch and sets up for Matplotlib
        plotting.

        :param obs: A 2D array of floats depicting an input image.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        pass

    @abstractmethod
    def plot_reward(
        self,
        reward_list: list,
        reward_window: int = None,
        tag: str = "reward",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plot the accumulated reward for each episode.

        :param reward_list: The list of recent rewards to be plotted
```

```
.
        :param reward_window: The length of the window to compute a moving average over.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        pass

    @abstractmethod
    def plot_spikes(
        self,
        spike_record: Dict[str, torch.Tensor],
        tag: str = "spike",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plots all spike records inside of ``spike_record``. Keeps unique
        plots for all unique tags that are given.

        :param spike_record: Dictionary of spikes to be rasterized.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        pass

    @abstractmethod
    def plot_voltages(
        self,
        voltage_record: Dict[str, torch.Tensor],
        thresholds: Optional[Dict[str, torch.Tensor]] = None,
        tag: str = "voltage",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plots all voltage records and given thresholds. Keeps unique
        plots for all unique tags that are given.

        :param voltage_record: Dictionary of voltages for neurons inside of networks
                               organized by the layer they correspond to.
        :param thresholds: Optional dictionary of threshold values for neurons.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        pass

    @abstractmethod
    def plot_conv2d_weights(
        self, weights: torch.Tensor, tag: str = "conv2d", step: int
```

```
= None
    ) -> None:
        # language=rst
        """
        Plot a connection weight matrix of a ``Conv2dConnection``.

        :param weights: Weight matrix of ``Conv2dConnection`` object.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        pass


class MatplotlibAnalyzer(PipelineAnalyzer):
    # language=rst
    """
    Renders output using Matplotlib.

    Matplotlib requires objects to be kept around over the full lifetime
    of the plots; this is done through ``self.plots``. An interactive session
    is needed so that we can continue processing and just update the
    plots.
    """

    def __init__(self, **kwargs) -> None:
        # language=rst
        """
        Initializes the analyzer.

        Keyword arguments:

        :param str volts_type: Type of plotting for voltages (``"color"`` or ``"line"``).
        """
        self.volts_type = kwargs.get("volts_type", "color")
        plt.ion()
        self.plots = {}

    def plot_obs(self, obs: torch.Tensor, tag: str = "obs", step: int = None) -> None:
        # language=rst
        """
        Pulls the observation off of torch and sets up for Matplotlib
        plotting.

        :param obs: A 2D array of floats depicting an input image.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        obs = obs.detach().cpu().numpy()
```

```
        obs = np.transpose(obs, (1, 2, 0)).squeeze()

        if tag in self.plots:
            obs_ax, obs_im = self.plots[tag]
        else:
            obs_ax, obs_im = None, None

        if obs_im is None and obs_ax is None:
            fig, obs_ax = plt.subplots()
            obs_ax.set_title("Observation")
            obs_ax.set_xticks(())
            obs_ax.set_yticks(())
            obs_im = obs_ax.imshow(obs, cmap="gray")

            self.plots[tag] = obs_ax, obs_im
        else:
            obs_im.set_data(obs)

    def plot_reward(
        self,
        reward_list: list,
        reward_window: int = None,
        tag: str = "reward",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plot the accumulated reward for each episode.

        :param reward_list: The list of recent rewards to be plotted.
        :param reward_window: The length of the window to compute a moving average over.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        if tag in self.plots:
            reward_im, reward_ax, reward_plot = self.plots[tag]
        else:
            reward_im, reward_ax, reward_plot = None, None, None

        # Compute moving average.
        if reward_window is not None:
            # Ensure window size > 0 and < size of reward list.
            window = max(min(len(reward_list), reward_window), 0)

            # Fastest implementation of moving average.
            reward_list_ = (
                pd.Series(reward_list)
                .rolling(window=window, min_periods=1)
                .mean()
                .values
            )
        else:
```

```
            reward_list_ = reward_list[:]

        if reward_im is None and reward_ax is None:
            reward_im, reward_ax = plt.subplots()
            reward_ax.set_title("Accumulated reward")
            reward_ax.set_xlabel("Episode")
            reward_ax.set_ylabel("Reward")
            (reward_plot,) = reward_ax.plot(reward_list_)

            self.plots[tag] = reward_im, reward_ax, reward_plot
        else:
            reward_plot.set_data(range(len(reward_list_)), reward_list_)
            reward_ax.relim()
            reward_ax.autoscale_view()

    def plot_spikes(
        self,
        spike_record: Dict[str, torch.Tensor],
        tag: str = "spike",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plots all spike records inside of ``spike_record``. Keeps unique
        plots for all unique tags that are given.

        :param spike_record: Dictionary of spikes to be rasterized.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        if tag not in self.plots:
            self.plots[tag] = plot_spikes(spike_record)
        else:
            s_im, s_ax = self.plots[tag]
            self.plots[tag] = plot_spikes(spike_record, ims=s_im, axes=s_ax)

    def plot_voltages(
        self,
        voltage_record: Dict[str, torch.Tensor],
        thresholds: Optional[Dict[str, torch.Tensor]] = None,
        tag: str = "voltage",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plots all voltage records and given thresholds. Keeps unique
        plots for all unique tags that are given.

        :param voltage_record: Dictionary of voltages for neurons inside of networks
                               organized by the layer they
```

```
correspond to.
        :param thresholds: Optional dictionary of threshold values for neurons.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        if tag not in self.plots:
            self.plots[tag] = plot_voltages(
                voltage_record, plot_type=self.volts_type, thresholds=thresholds
            )
        else:
            v_im, v_ax = self.plots[tag]
            self.plots[tag] = plot_voltages(
                voltage_record,
                ims=v_im,
                axes=v_ax,
                plot_type=self.volts_type,
                thresholds=thresholds,
            )

    def plot_conv2d_weights(
        self, weights: torch.Tensor, tag: str = "conv2d", step: int = None
    ) -> None:
        # language=rst
        """
        Plot a connection weight matrix of a ``Conv2dConnection``.

        :param weights: Weight matrix of ``Conv2dConnection`` object.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        wmin = weights.min().item()
        wmax = weights.max().item()

        if tag not in self.plots:
            self.plots[tag] = plot_conv2d_weights(weights, wmin, wmax)
        else:
            im = self.plots[tag]
            plot_conv2d_weights(weights, wmin, wmax, im=im)

    def finalize_step(self) -> None:
        # language=rst
        """
        Flush the output from the current step
        """
        plt.draw()
        plt.pause(1e-8)
        plt.show()
```

```
class TensorboardAnalyzer(PipelineAnalyzer):
    def __init__(self, summary_directory: str = "./logs"):
        # language=rst
        """
        Initializes the analyzer.

        :param summary_directory: Directory to save log files.
        """
        self.writer = SummaryWriter(summary_directory)

    def finalize_step(self) -> None:
        # language=rst
        """
        No-op for ``TensorboardAnalyzer``.
        """
        pass

    def plot_obs(self, obs: torch.Tensor, tag: str = "obs", step:
    int = None) -> None:
        # language=rst
        """
        Pulls the observation off of torch and sets up for
    Matplotlib
        plotting.

        :param obs: A 2D array of floats depicting an input image.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        obs_grid = make_grid(obs.float(), nrow=4, normalize=True)
        self.writer.add_image(tag, obs_grid, step)

    def plot_reward(
        self,
        reward_list: list,
        reward_window: int = None,
        tag: str = "reward",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plot the accumulated reward for each episode.

        :param reward_list: The list of recent rewards to be plotted
    .
        :param reward_window: The length of the window to compute a
    moving average over.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        self.writer.add_scalar(tag, reward_list[-1], step)

    def plot_spikes(
        self,
```

```
        spike_record: Dict[str, torch.Tensor],
        tag: str = "spike",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plots all spike records inside of ``spike_record``. Keeps unique
        plots for all unique tags that are given.

        :param spike_record: Dictionary of spikes to be rasterized.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        for k, spikes in spike_record.items():
            # shuffle spikes into 1x1x#NueronsxT
            spikes = spikes.view(1, 1, -1, spikes.shape[-1]).float()
            spike_grid_img = make_grid(spikes, nrow=1, pad_value=0.5)

            self.writer.add_image(tag + "_" + str(k), spike_grid_img, step)

    def plot_voltages(
        self,
        voltage_record: Dict[str, torch.Tensor],
        thresholds: Optional[Dict[str, torch.Tensor]] = None,
        tag: str = "voltage",
        step: int = None,
    ) -> None:
        # language=rst
        """
        Plots all voltage records and given thresholds. Keeps unique
        plots for all unique tags that are given.

        :param voltage_record: Dictionary of voltages for neurons inside of networks
                               organized by the layer they correspond to.
        :param thresholds: Optional dictionary of threshold values for neurons.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        for k, v in voltage_record.items():
            # Shuffle voltages into 1x1x#neuronsxT
            v = v.view(1, 1, -1, v.shape[-1])
            voltage_grid_img = make_grid(v, nrow=1, pad_value=0)

            self.writer.add_image(tag + "_" + str(k), voltage_grid_img, step)

    def plot_conv2d_weights(
        self, weights: torch.Tensor, tag: str = "conv2d", step: int
```

```
    = None
    ) -> None:
        # language=rst
        """
        Plot a connection weight matrix of a ``Conv2dConnection``.

        :param weights: Weight matrix of ``Conv2dConnection`` object.
        :param tag: A unique tag to associate the data with.
        :param step: The step of the pipeline.
        """
        reshaped = reshape_conv2d_weights(weights).unsqueeze(0)

        reshaped -= reshaped.min()
        reshaped /= reshaped.max()

        self.writer.add_image(tag, reshaped, step)
```

Listing B.29: *Pipeline analysis*

```
import torch
import numpy as np
import matplotlib.pyplot as plt

from matplotlib.axes import Axes
from matplotlib.image import AxesImage
from torch.nn.modules.utils import _pair
from matplotlib.collections import PathCollection
from mpl_toolkits.axes_grid1 import make_axes_locatable
from typing import Tuple, List, Optional, Sized, Dict, Union

from ..utils import reshape_locally_connected_weights, reshape_conv2d_weights

plt.ion()


def plot_input(
    image: torch.Tensor,
    inpt: torch.Tensor,
    label: Optional[int] = None,
    axes: List[Axes] = None,
    ims: List[AxesImage] = None,
    figsize: Tuple[int, int] = (8, 4),
) -> Tuple[List[Axes], List[AxesImage]]:
    # language=rst
    """
    Plots a two-dimensional image and its corresponding spike-train representation.

    :param image: A 2D array of floats depicting an input image.
    :param inpt: A 2D array of floats depicting an image's spike-train encoding.
    :param label: Class label of the input data.
```

```
    :param axes: Used for re-drawing the input plots.
    :param ims: Used for re-drawing the input plots.
    :param figsize: Horizontal, vertical figure size in inches.
    :return: Tuple of ``(axes, ims)`` used for re-drawing the input plots.
    """
    local_image = image.detach().clone().cpu().numpy()
    local_inpy = inpt.detach().clone().cpu().numpy()

    if axes is None:
        fig, axes = plt.subplots(1, 2, figsize=figsize)
        ims = (
            axes[0].imshow(local_image, cmap="binary"),
            axes[1].imshow(local_inpy, cmap="binary"),
        )

        if label is None:
            axes[0].set_title("Current image")
        else:
            axes[0].set_title("Current image (label = %d)" % label)

        axes[1].set_title("Reconstruction")

        for ax in axes:
            ax.set_xticks(())
            ax.set_yticks(())

        fig.tight_layout()
    else:
        if label is not None:
            axes[0].set_title("Current image (label = %d)" % label)

        ims[0].set_data(local_image)
        ims[1].set_data(local_inpy)

    return axes, ims


def plot_spikes(
    spikes: Dict[str, torch.Tensor],
    time: Optional[Tuple[int, int]] = None,
    n_neurons: Optional[Dict[str, Tuple[int, int]]] = None,
    ims: Optional[List[PathCollection]] = None,
    axes: Optional[Union[Axes, List[Axes]]] = None,
    figsize: Tuple[float, float] = (8.0, 4.5),
) -> Tuple[List[AxesImage], List[Axes]]:
    # language=rst
    """
    Plot spikes for any group(s) of neurons.

    :param spikes: Mapping from layer names to spiking data. Spike data has shape
        ``[time, n_1, ..., n_k]``, where ``[n_1, ..., n_k]`` is the shape of the
```

```
        recorded layer.
    :param time: Plot spiking activity of neurons in the given time range. Default is
        entire simulation time.
    :param n_neurons: Plot spiking activity of neurons in the given range of neurons.
        Default is all neurons.
    :param ims: Used for re-drawing the plots.
    :param axes: Used for re-drawing the plots.
    :param figsize: Horizontal, vertical figure size in inches.
    :return: ``ims, axes``: Used for re-drawing the plots.
    """
    n_subplots = len(spikes.keys())
    if n_neurons is None:
        n_neurons = {}

    spikes = {k: v.view(v.size(0), -1) for (k, v) in spikes.items()}
    if time is None:
        # Set it for entire duration
        for key in spikes.keys():
            time = (0, spikes[key].shape[0])
            break

    # Use all neurons if no argument provided.
    for key, val in spikes.items():
        if key not in n_neurons.keys():
            n_neurons[key] = (0, val.shape[1])

    if ims is None:
        fig, axes = plt.subplots(n_subplots, 1, figsize=figsize)
        if n_subplots == 1:
            axes = [axes]

        ims = []
        for i, datum in enumerate(spikes.items()):
            spikes = (
                datum[1][
                    time[0] : time[1], n_neurons[datum[0]][0] : n_neurons[datum[0]][1]
                ]
                .detach()
                .clone()
                .cpu()
                .numpy()
            )
            ims.append(
                axes[i].scatter(
                    x=np.array(spikes.nonzero()).T[:, 0],
                    y=np.array(spikes.nonzero()).T[:, 1],
                    s=1,
                )
            )
            args = (
                datum[0],
```

```
                n_neurons[datum[0]][0],
                n_neurons[datum[0]][1],
                time[0],
                time[1],
            )
            axes[i].set_title(
                "%s spikes for neurons (%d - %d) from t = %d to %d " % args
            )
        for ax in axes:
            ax.set_aspect("auto")

        plt.setp(
            axes, xticks=[], yticks=[], xlabel="Simulation time", ylabel="Neuron index"
        )
        plt.tight_layout()
    else:
        for i, datum in enumerate(spikes.items()):
            spikes = (
                datum[1][
                    time[0] : time[1], n_neurons[datum[0]][0] : n_neurons[datum[0]][1]
                ]
                .detach()
                .clone()
                .cpu()
                .numpy()
            )
            ims[i].set_offsets(np.array(spikes.nonzero()).T)
            args = (
                datum[0],
                n_neurons[datum[0]][0],
                n_neurons[datum[0]][1],
                time[0],
                time[1],
            )
            axes[i].set_title(
                "%s spikes for neurons (%d - %d) from t = %d to %d " % args
            )

    plt.draw()

    return ims, axes


def plot_weights(
    weights: torch.Tensor,
    wmin: Optional[float] = 0,
    wmax: Optional[float] = 1,
    im: Optional[AxesImage] = None,
    figsize: Tuple[int, int] = (5, 5),
    cmap: str = "hot_r",
```

```
) -> AxesImage:
    # language=rst
    """
    Plot a connection weight matrix.

    :param weights: Weight matrix of ``Connection`` object.
    :param wmin: Minimum allowed weight value.
    :param wmax: Maximum allowed weight value.
    :param im: Used for re-drawing the weights plot.
    :param figsize: Horizontal, vertical figure size in inches.
    :param cmap: Matplotlib colormap.
    :return: ``AxesImage`` for re-drawing the weights plot.
    """
    local_weights = weights.detach().clone().cpu().numpy()
    if not im:
        fig, ax = plt.subplots(figsize=figsize)

        im = ax.imshow(local_weights, cmap=cmap, vmin=wmin, vmax=
    wmax)
        div = make_axes_locatable(ax)
        cax = div.append_axes("right", size="5%", pad=0.05)

        ax.set_xticks(())
        ax.set_yticks(())
        ax.set_aspect("auto")

        plt.colorbar(im, cax=cax)
        fig.tight_layout()
    else:
        im.set_data(local_weights)

    return im


def plot_conv2d_weights(
    weights: torch.Tensor,
    wmin: float = 0.0,
    wmax: float = 1.0,
    im: Optional[AxesImage] = None,
    figsize: Tuple[int, int] = (5, 5),
    cmap: str = "hot_r",
) -> AxesImage:
    # language=rst
    """
    Plot a connection weight matrix of a Conv2dConnection.

    :param weights: Weight matrix of Conv2dConnection object.
    :param wmin: Minimum allowed weight value.
    :param wmax: Maximum allowed weight value.
    :param im: Used for re-drawing the weights plot.
    :param figsize: Horizontal, vertical figure size in inches.
    :param cmap: Matplotlib colormap.
    :return: Used for re-drawing the weights plot.
    """
```

```
    sqrt1 = int(np.ceil(np.sqrt(weights.size(0))))
    sqrt2 = int(np.ceil(np.sqrt(weights.size(1))))
    height, width = weights.size(2), weights.size(3)
    reshaped = reshape_conv2d_weights(weights)

    if not im:
        fig, ax = plt.subplots(figsize=figsize)
        im = ax.imshow(reshaped, cmap=cmap, vmin=wmin, vmax=wmax)
        div = make_axes_locatable(ax)
        cax = div.append_axes("right", size="5%", pad=0.05)

        for i in range(height, sqrt1 * sqrt2 * height, height):
            ax.axhline(i - 0.5, color="g", linestyle="--")
            if i % sqrt1 == 0:
                ax.axhline(i - 0.5, color="g", linestyle="-")

        for i in range(width, sqrt1 * sqrt2 * width, width):
            ax.axvline(i - 0.5, color="g", linestyle="--")
            if i % sqrt1 == 0:
                ax.axvline(i - 0.5, color="g", linestyle="-")

        ax.set_xticks(())
        ax.set_yticks(())
        ax.set_aspect("auto")

        plt.colorbar(im, cax=cax)
        fig.tight_layout()
    else:
        im.set_data(reshaped)

    return im


def plot_locally_connected_weights(
    weights: torch.Tensor,
    n_filters: int,
    kernel_size: Union[int, Tuple[int, int]],
    conv_size: Union[int, Tuple[int, int]],
    locations: torch.Tensor,
    input_sqrt: Union[int, Tuple[int, int]],
    wmin: float = 0.0,
    wmax: float = 1.0,
    im: Optional[AxesImage] = None,
    lines: bool = True,
    figsize: Tuple[int, int] = (5, 5),
    cmap: str = "hot_r",
) -> AxesImage:
    # language=rst
    """
    Plot a connection weight matrix of a :code:`Connection` with `
   locally connected
    structure <http://yann.lecun.com/exdb/publis/pdf/gregor-nips-11.
   pdf>`_.
```

```

    :param weights: Weight matrix of Conv2dConnection object.
    :param n_filters: No. of convolution kernels in use.
    :param kernel_size: Side length(s) of 2D convolution kernels.
    :param conv_size: Side length(s) of 2D convolution population.
    :param locations: Indices of input receptive fields for convolution population
        neurons.
    :param input_sqrt: Side length(s) of 2D input data.
    :param wmin: Minimum allowed weight value.
    :param wmax: Maximum allowed weight value.
    :param im: Used for re-drawing the weights plot.
    :param lines: Whether or not to draw horizontal and vertical lines separating input
        regions.
    :param figsize: Horizontal, vertical figure size in inches.
    :param cmap: Matplotlib colormap.
    :return: Used for re-drawing the weights plot.
    """
    kernel_size = _pair(kernel_size)
    conv_size = _pair(conv_size)
    input_sqrt = _pair(input_sqrt)

    reshaped = reshape_locally_connected_weights(
        weights, n_filters, kernel_size, conv_size, locations, input_sqrt
    )
    n_sqrt = int(np.ceil(np.sqrt(n_filters)))

    if not im:
        fig, ax = plt.subplots(figsize=figsize)

        im = ax.imshow(reshaped.cpu(), cmap=cmap, vmin=wmin, vmax=wmax)
        div = make_axes_locatable(ax)
        cax = div.append_axes("right", size="5%", pad=0.05)

        if lines:
            for i in range(
                n_sqrt * kernel_size[0],
                n_sqrt * conv_size[0] * kernel_size[0],
                n_sqrt * kernel_size[0],
            ):
                ax.axhline(i - 0.5, color="g", linestyle="--")

            for i in range(
                n_sqrt * kernel_size[1],
                n_sqrt * conv_size[1] * kernel_size[1],
                n_sqrt * kernel_size[1],
            ):
                ax.axvline(i - 0.5, color="g", linestyle="--")

        ax.set_xticks(())
        ax.set_yticks(())
```

```
        ax.set_aspect("auto")

        plt.colorbar(im, cax=cax)
        fig.tight_layout()
    else:
        im.set_data(reshaped)

    return im


def plot_assignments(
    assignments: torch.Tensor,
    im: Optional[AxesImage] = None,
    figsize: Tuple[int, int] = (5, 5),
    classes: Optional[Sized] = None,
) -> AxesImage:
    # language=rst
    """
    Plot the two-dimensional neuron assignments.

    :param assignments: Vector of neuron label assignments.
    :param im: Used for re-drawing the assignments plot.
    :param figsize: Horizontal, vertical figure size in inches.
    :param classes: Iterable of labels for colorbar ticks corresponding to data labels.
    :return: Used for re-drawing the assigments plot.
    """
    locals_assignments = assignments.detach().clone().cpu().numpy()
    if not im:
        fig, ax = plt.subplots(figsize=figsize)
        ax.set_title("Categorical assignments")

        if classes is None:
            color = plt.get_cmap("RdBu", 11)
            im = ax.matshow(locals_assignments, cmap=color, vmin=-1.5, vmax=9.5)
        else:
            color = plt.get_cmap("RdBu", len(classes) + 1)
            im = ax.matshow(
                locals_assignments, cmap=color, vmin=-1.5, vmax=len(classes) - 0.5
            )

        div = make_axes_locatable(ax)
        cax = div.append_axes("right", size="5%", pad=0.05)

        if classes is None:
            cbar = plt.colorbar(im, cax=cax, ticks=list(range(-1, 11)))
            cbar.ax.set_yticklabels(["none"] + list(range(10)))
        else:
            cbar = plt.colorbar(im, cax=cax, ticks=np.arange(-1, len(classes)))
            cbar.ax.set_yticklabels(["none"] + list(classes))
```

```
            ax.set_xticks(())
            ax.set_yticks(())
            fig.tight_layout()
    else:
        im.set_data(locals_assignments)

    return im


def plot_performance(
    performances: Dict[str, List[float]],
    ax: Optional[Axes] = None,
    figsize: Tuple[int, int] = (7, 4),
) -> Axes:
    # language=rst
    """
    Plot training accuracy curves.

    :param performances: Lists of training accuracy estimates per
  voting scheme.
    :param ax: Used for re-drawing the performance plot.
    :param figsize: Horizontal, vertical figure size in inches.
    :return: Used for re-drawing the performance plot.
    """
    if not ax:
        _, ax = plt.subplots(figsize=figsize)
    else:
        ax.clear()

    for scheme in performances:
        ax.plot(
            range(len(performances[scheme])),
            [p for p in performances[scheme]],
            label=scheme,
        )

    ax.set_ylim([0, 100])
    ax.set_title("Estimated classification accuracy")
    ax.set_xlabel("No. of examples")
    ax.set_ylabel("Accuracy")
    ax.set_xticks(())
    ax.set_yticks(range(0, 110, 10))
    ax.legend()

    return ax


def plot_voltages(
    voltages: Dict[str, torch.Tensor],
    ims: Optional[List[AxesImage]] = None,
    axes: Optional[List[Axes]] = None,
    time: Tuple[int, int] = None,
    n_neurons: Optional[Dict[str, Tuple[int, int]]] = None,
```

```
    cmap: Optional[str] = "jet",
    plot_type: str = "color",
    thresholds: Dict[str, torch.Tensor] = None,
    figsize: Tuple[float, float] = (8.0, 4.5),
) -> Tuple[List[AxesImage], List[Axes]]:
    # language=rst
    """
    Plot voltages for any group(s) of neurons.

    :param voltages: Contains voltage data by neuron layers.
    :param ims: Used for re-drawing the plots.
    :param axes: Used for re-drawing the plots.
    :param time: Plot voltages of neurons in given time range. Default is entire
        simulation time.
    :param n_neurons: Plot voltages of neurons in given range of neurons. Default is all
        neurons.
    :param cmap: Matplotlib colormap to use.
    :param figsize: Horizontal, vertical figure size in inches.
    :param plot_type: The way how to draw graph. 'color' for pcolormesh, 'line' for
        curved lines.
    :param thresholds: Thresholds of the neurons in each layer.
    :return: ``ims, axes``: Used for re-drawing the plots.
    """
    n_subplots = len(voltages.keys())

    for key in voltages.keys():
        voltages[key] = voltages[key].view(-1, voltages[key].size(-1))

    if time is None:
        for key in voltages.keys():
            time = (0, voltages[key].size(-1))
            break

    if n_neurons is None:
        n_neurons = {}

    for key, val in voltages.items():
        if key not in n_neurons.keys():
            n_neurons[key] = (0, val.size(0))

    if not ims:
        fig, axes = plt.subplots(n_subplots, 1, figsize=figsize)
        ims = []
        if n_subplots == 1:  # Plotting only one image
            for v in voltages.items():
                if plot_type == "line":
                    ims.append(
                        axes.plot(
                            v[1]
                            .detach()
```

```
                            .clone()
                            .cpu()
                            .numpy()[
                                n_neurons[v[0]][0] : n_neurons[v[0]][1],
                                time[0] : time[1],
                            ]
                        )
                    )

                    if thresholds is not None and thresholds[v[0]].size() == torch.Size(
                        []
                    ):
                        ims.append(
                            axes.axhline(
                                y=thresholds[v[0]].item(), c="r", linestyle="--"
                            )
                        )
                else:
                    ims.append(
                        axes.pcolormesh(
                            v[1]
                            .cpu()
                            .numpy()[
                                n_neurons[v[0]][0] : n_neurons[v[0]][1],
                                time[0] : time[1],
                            ]
                            .T,
                            cmap=cmap,
                        )
                    )

                args = (v[0], n_neurons[v[0]][0], n_neurons[v[0]][1], time[0], time[1])
                plt.title("%s voltages for neurons (%d - %d) from t = %d to %d " % args)
                plt.xlabel("Time (ms)")

                if plot_type == "line":
                    plt.ylabel("Voltage")
                else:
                    plt.ylabel("Neuron index")

                axes.set_aspect("auto")

        else:  # Plot each layer at a time
            for i, v in enumerate(voltages.items()):
                if plot_type == "line":
                    ims.append(
                        axes[i].plot(
                            v[1]
```

```
                            .cpu()
                            .numpy()[
                                n_neurons[v[0]][0] : n_neurons[v[0]][1],
                                time[0] : time[1],
                            ]
                        )
                    )
                    if thresholds is not None and thresholds[v[0]].size() == torch.Size(
                        []
                    ):
                        ims.append(
                            axes[i].axhline(
                                y=thresholds[v[0]].item(), c="r", linestyle="--"
                            )
                        )
                else:
                    ims.append(
                        axes[i].matshow(
                            v[1]
                            .cpu()
                            .numpy()[
                                n_neurons[v[0]][0] : n_neurons[v[0]][1],
                                time[0] : time[1],
                            ]
                            .T,
                            cmap=cmap,
                        )
                    )
                args = (v[0], n_neurons[v[0]][0], n_neurons[v[0]][1], time[0], time[1])
                axes[i].set_title(
                    "%s voltages for neurons (%d - %d) from t = %d to %d " % args
                )

            for ax in axes:
                ax.set_aspect("auto")

        if plot_type == "color":
            plt.setp(axes, xlabel="Simulation time", ylabel="Neuron index")
        elif plot_type == "line":
            plt.setp(axes, xlabel="Simulation time", ylabel="Voltage")

        plt.tight_layout()

    else:
        # Plotting figure given
        if n_subplots == 1:  # Plotting only one image
```

```
            for v in voltages.items():
                axes.clear()
                if plot_type == "line":
                    axes.plot(
                        v[1]
                        .cpu()
                        .numpy()[
                            n_neurons[v[0]][0] : n_neurons[v[0]][1], time[0] : time[1]
                        ]
                    )
                    if thresholds is not None and thresholds[v[0]].size() == torch.Size(
                        []
                    ):
                        axes.axhline(y=thresholds[v[0]].item(), c="r", linestyle="--")
                else:
                    axes.matshow(
                        v[1]
                        .cpu()
                        .numpy()[
                            n_neurons[v[0]][0] : n_neurons[v[0]][1], time[0] : time[1]
                        ]
                        .T,
                        cmap=cmap,
                    )
                args = (v[0], n_neurons[v[0]][0], n_neurons[v[0]][1], time[0], time[1])
                axes.set_title(
                    "%s voltages for neurons (%d - %d) from t = %d to %d " % args
                )
                axes.set_aspect("auto")

        else:
            # Plot each layer at a time
            for i, v in enumerate(voltages.items()):
                axes[i].clear()
                if plot_type == "line":
                    axes[i].plot(
                        v[1]
                        .cpu()
                        .numpy()[
                            n_neurons[v[0]][0] : n_neurons[v[0]][1], time[0] : time[1]
                        ]
                    )
                    if thresholds is not None and thresholds[v[0]].size() == torch.Size(
                        []
                    ):
                        axes[i].axhline(
```

```
                            y=thresholds[v[0]].item(), c="r", linestyle="--"
                        )
                else:
                    axes[i].matshow(
                        v[1]
                        .cpu()
                        .numpy()[
                            n_neurons[v[0]][0] : n_neurons[v[0]][1], time[0] : time[1]
                        ]
                        .T,
                        cmap=cmap,
                    )
                args = (v[0], n_neurons[v[0]][0], n_neurons[v[0]][1], time[0], time[1])
                axes[i].set_title(
                    "%s voltages for neurons (%d - %d) from t = %d to %d " % args
                )

            for ax in axes:
                ax.set_aspect("auto")

        if plot_type == "color":
            plt.setp(axes, xlabel="Simulation time", ylabel="Neuron index")
        elif plot_type == "line":
            plt.setp(axes, xlabel="Simulation time", ylabel="Voltage")

        plt.tight_layout()

    return ims, axes
```

Listing B.30: *Plotting*

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

from typing import List, Tuple, Optional


def plot_weights_movie(ws: np.ndarray, sample_every: int = 1) -> None:
    # language=rst
    """
    Create and plot movie of weights.

    :param ws: Array of shape ``[n_examples, source, target, time]``.
    :param sample_every: Sub-sample using this parameter.
```

```
    """
    weights = []

    # Obtain samples from the weights for every example.
    for i in range(ws.shape[0]):
        sub_sampled_weight = ws[i, :, :, range(0, ws[i].shape[2],
sample_every)]
        weights.append(sub_sampled_weight)
    else:
        weights = np.concatenate(weights, axis=0)

    # Initialize plot.
    fig = plt.figure()
    im = plt.imshow(weights[0, :, :], cmap="hot_r", animated=True,
vmin=0, vmax=1)
    plt.axis("off")
    plt.colorbar(im)

    # Update function for the animation.
    def update(j):
        im.set_data(weights[j, :, :])
        return [im]

    # Initialize animation.
    global ani
    ani = 0
    ani = animation.FuncAnimation(
        fig, update, frames=weights.shape[-1], interval=1000, blit=
True
    )
    plt.show()


def plot_spike_trains_for_example(
    spikes: torch.Tensor,
    n_ex: Optional[int] = None,
    top_k: Optional[int] = None,
    indices: Optional[List[int]] = None,
) -> None:
    # language=rst
    """
    Plot spike trains for top-k neurons or for specific indices.

    :param spikes: Spikes for one simulation run of shape
        ``(n_examples, n_neurons, time)``.
    :param n_ex: Allows user to pick which example to plot spikes
for.
    :param top_k: Plot k neurons that spiked the most for n_ex
example.
    :param indices: Plot specific neurons' spiking activity instead
of top_k.
    """
    assert n_ex is not None and 0 <= n_ex < spikes.shape[0]

```

```python
    plt.figure()

    if top_k is None and indices is None:  # Plot all neurons' spiking activity
        spike_per_neuron = [np.argwhere(i == 1).flatten() for i in spikes[n_ex, :, :]]
        plt.title("Spiking activity for all %d neurons" % spikes.shape[1])

    elif top_k is None:  # Plot based on indices parameter
        assert indices is not None
        spike_per_neuron = [
            np.argwhere(i == 1).flatten() for i in spikes[n_ex, indices, :]
        ]

    elif indices is None:  # Plot based on top_k parameter
        assert top_k is not None
        # Obtain the top k neurons that fired the most
        top_k_loc = np.argsort(np.sum(spikes[n_ex, :, :], axis=1), axis=0)[::-1]
        spike_per_neuron = [
            np.argwhere(i == 1).flatten() for i in spikes[n_ex, top_k_loc[0:top_k], :]
        ]
        plt.title("Spiking activity for top %d neurons" % top_k)

    else:
        raise ValueError('One of "top_k" or "indices" or both must be None')

    plt.eventplot(spike_per_neuron, linelengths=[0.5] * len(spike_per_neuron))
    plt.xlabel("Simulation Time")
    plt.ylabel("Neuron index")
    plt.show()


def plot_voltage(
    voltage: torch.Tensor,
    n_ex: int = 0,
    n_neuron: int = 0,
    time: Optional[Tuple[int, int]] = None,
    threshold: float = None,
) -> None:
    # language=rst
    """
    Plot voltage for a single neuron on a specific example.

    :param voltage: Tensor or array of shape ``[n_examples, n_neurons, time]``.
    :param n_ex: Allows user to pick which example to plot voltage for.
    :param n_neuron: Neuron index for which to plot voltages for.
```

```
    :param time: Plot spiking activity of neurons between the given range of time.
    :param threshold: Neuron spiking threshold.
    """
    assert n_ex >= 0 and n_neuron >= 0
    assert n_ex < voltage.shape[0] and n_neuron < voltage.shape[1]

    if time is None:
        time = (0, voltage.shape[-1])
    else:
        assert time[0] < time[1]
        assert time[1] <= voltage.shape[-1]

    timer = np.arange(time[0], time[1])
    time_ticks = np.arange(time[0], time[1] + 1, 10)

    plt.figure()
    plt.plot(voltage[n_ex, n_neuron, timer])
    plt.xlabel("Simulation Time")
    plt.ylabel("Voltage")
    plt.title("Membrane voltage of neuron %d for example %d" % (n_neuron, n_ex + 1))
    locs, labels = plt.xticks()
    locs = range(int(locs[1]), int(locs[-1]), 10)
    plt.xticks(locs, time_ticks)

    # Draw threshold line only if given
    if threshold is not None:
        plt.axhline(threshold, linestyle="--", color="black", zorder=0)

    plt.show()
```

Listing B.31: *Visualization*