Simulation and Validation of Temporal Neural Networks

MSc Thesis

Gio Lin

Simulation and Validation of Temporal Neural Networks

by

Gio Lin

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 22, 2024 at 14:00.

Student number:	4885090	
Project duration:	January 26, 2024 – August 22, 2024	
Thesis committee:	Prof. dr. ir. G. N. (Georgi) Gaydadjiev,	TU Delft, supervisor
	Dr. ir. C. J. M. (Chris) Verhoeven,	TU Delft

Cover: Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

It was a difficult task for me to finish this thesis that I could not have done entirely by myself. Therefore, I would like to use this space to thank the people that have helped me emotionally and technically.

First, I would like to thank my supervisor Prof. dr. ir. Georgi Gaydadjiev from Delft University of Technology in The Netherlands. I am grateful for the weekly meetings and discussions I had with him, for introducing me to this interesting topic to write my thesis about, for their guidance during the project and for all the useful comments with respect to writing this thesis.

Second, I would like to thank Nils Voß, for taking a look at the codebase of my thesis and helping with figuring out problems and bugs within the codebase, as well as improving codestyle.

Third, I would like to thank the friends I have made at Delft University of Technology. You have made the academic process a lot more fun and I would not have been here without you.

Next, I would like to thank Mathijs Beemsterboer, for supporting me and giving me life advice throughout the years since 2016. Ever since meeting you in 2016, a lot in my life has changed and I truly believe you have played a big part in that. Every time that we had a set up a meetup to drink coffee together, those conversations we had, I will never forget them. I look forward to future our future meetups, thank you Mathijs.

Last but certainly not least, I would like to thank the people closest to me for the support and motivation they have given me throughout the process of working on and writing this thesis. I love you all, thank you very much for being there for me.

Gio Lin
Delft, April 2026

Abstract

This thesis explores the simulation and validation of Temporal Neural Networks (TNNs), a form of Artificial Neural Network (ANN) which is relatively underdeveloped, presenting opportunities for further exploration and innovation. TNNs are an interesting research topic, because they attempt to mimic the way biological neural networks process information, relying on temporal pulses or spikes rather than continuous activation. Past works have developed simulators for Temporal Neural Network (TNN) systems, but these simulators often face significant limitations due to the environments in which they are implemented. The primary challenge lies in their inefficiency, which makes conducting large-scale tests difficult. This gives the research question of this thesis is: Can a simulator be developed in an environment that would enable large-scale experimentation and testing of TNN systems? This thesis presents a compiled code event driven simulator for TNN systems, with results for a pre-trained network matching past work, as well as preliminary results for reinforcement learning using TNN systems.

Contents

Preface i

Abstract ii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 State-of-the-Art: Temporal Neural Networks . . . 1
- 1.3 Research Question and Thesis Contributions . . . 2
- 1.4 Thesis Organization . . . 2

2 Introduction to Artificial Neural Networks 3
- 2.1 Artificial Neural Networks . . . 3
- 2.2 Composition and Components of Artificial Neural Networks . . . 3
- 2.3 Network Training and Learning Methods . . . 5
  - 2.3.1 Network Training: Initialization . . . 5
  - 2.3.2 Network Training: Optimization . . . 6
  - 2.3.3 Network Training: Regularization . . . 7
  - 2.3.4 Supervised and Unsupervised Learning . . . 8
  - 2.3.5 Online and Offline Learning . . . 8
  - 2.3.6 Reinforcement Learning . . . 8
- 2.4 Use Cases and Applications . . . 9
- 2.5 Fundamental and Prominent Artificial Neural Network Types . . . 10
  - 2.5.1 Feedforward Neural Network/Multilayer Perceptron . . . 10
  - 2.5.2 Recurrent Neural Network . . . 10
  - 2.5.3 Convolutional Neural Network . . . 11
  - 2.5.4 Autoencoder . . . 12
  - 2.5.5 Boltzmann Machine . . . 13
  - 2.5.6 Transformer . . . 13
- 2.6 Less Prominent Types of Artificial Neural Networks . . . 14

3 Introduction to Spiking Neural Networks 16
- 3.1 Spiking Neural Networks . . . 16
- 3.2 Composition, Components and Information Processing of Spiking Neural Networks . . . 16
- 3.3 Types of Spiking Neural Network . . . 16

4 Temporal Neural Networks 18
- 4.1 Temporal Neural Networks . . . 18
- 4.2 Composition, Components and Information processing of Temporal Neural Networks . . . 18
- 4.3 Network Training and Learning Methods . . . 19
- 4.4 Background and Related Work . . . 20
- 4.5 Connecting Temporal Neural Networks with Biology . . . 22
- 4.6 Validation of Temporal Neural Networks . . . 23

5 Key Subjects and Methodology 24
- 5.1 Online Reinforcement Learning using Temporal Neural Networks . . . 24
- 5.2 Cart-Pole Problem . . . 30
- 5.3 Implementation of Simulator and Experiment Environment . . . 32

6 Design and Implementation 34
- 6.1 Setup of System and Environment . . . 34
- 6.2 Implementation of Temporal Neural Networks . . . 34
  - 6.2.1 Spikes . . . 34

6.2.2 Structure of Neurons 35
6.2.3 Structure of Network 35
6.2.4 Network Configuration 35
6.2.5 Encoding and Decoding 36
6.2.6 Spike Timing Dependent Plasticity and Weight Updating 36
6.3 Implementation of Simulator 37
6.3.1 Environment Around Neural Network 37
6.3.2 Pre-Simulation Setup of Environment and Network 37
6.3.3 Simulation Running 37

7 Experimental Setup, Results and Discussion **39**
7.1 Setup of Environment: Cart-Pole Problem 39
7.2 Experimental Setup: Verification of Cart-Pole Equations 39
7.3 Experimental Setup: Simulations 40
7.4 Results: Cart-Pole Equations 41
7.5 Results: Simulations 42
7.6 Results: Experiments with Spike Timing Dependent Plasticity 43
7.7 Discussion: Additional Improvements 45

8 Conclusion **47**
8.1 Summary 47
8.2 Future Work 47

References **48**

A Tables of Results: Cart-Pole Equations **51**

1 Introduction

This chapter introduces the topic addressed in this thesis, the motivation behind the research, the current state-of-the-art and the main contribution. Section 1.1 presents the motivation and relevance of Artificial Neural Networks, discusses what use cases there are for variants of Artificial Neural Networks, and highlights why the variant Temporal Neural Network is an engaging research topic. Section 1.2 presents the current state-of-the-art in Temporal Neural Network research and highlights the need for further research in this area. Section 1.3 lays out the research questions of this thesis as well as the scientific contributions of it. Section 1.4 finally presents the outline of this thesis.

1.1. Motivation

Artificial Neural Networks (ANNs) have become a cornerstone of modern technology, gaining significant attention across multiple domains. Their versatility is evident in a wide range of applications, including advanced text processing systems like GPT and Gemini, recommendation algorithms on platforms such as YouTube, TikTok, and Netflix, and AI-powered enhancements in graphics processing, such as NVIDIA's DLSS and AMD's FSR. These examples highlight the diverse and powerful impact of ANNs on everyday technologies, each variant tailored to its specific use case.

Despite the remarkable advancements in ANNs, certain variants remain relatively underdeveloped, presenting opportunities for further exploration and innovation. Among these, Spiking Neural Networks (SNNs) and, more specifically, Temporal Neural Networks (TNNs), stand out as promising candidates for future research. These networks are intriguing because they attempt to mimic the way biological neural networks process information, relying on temporal pulses or spikes rather than continuous activation.

Past works [34], [35], have developed simulators for Temporal Neural Network (TNN) systems, but these simulators often face significant limitations due to the environments in which they are implemented. The primary challenge lies in their inefficiency, which makes conducting large-scale tests difficult. This raises the research question: Can a more efficient simulator be developed in a different environment, one that would enable large-scale experimentation and testing of TNN systems? Addressing this question could lead to significant advancements and progress in the ability to evaluate and utilize TNNs effectively.

1.2. State-of-the-Art: Temporal Neural Networks

Temporal Neural Networks (TNNs) are type of Artificial Neural Network (ANN), specifically it is a type of Spiking Neural Network (SNN). For SNNs, there are two variants that use different method of encoding information. These are spike rate encoding and time based encoding. TNNs are SNNs that use time based encoding. The concept of TNNs was first introduced by Wolfgang Maass [22], later [24] and [6] have made refined implementations. [37], [34] and [35] have made simulators in MATLAB for TNN systems, however these suffer from the technical limitations of MATLAB, which makes large-scale experiments difficult.

1.3. Research Question and Thesis Contributions

The research question of this thesis is: Can a simulator be developed in an environment that would enable large-scale experimentation and testing of TNN systems? This makes the objective of this thesis to build a simulator and to verify that TNN systems can be simulated with it. The main contributions of this thesis are:

- A compiled code event driven simulator for TNN systems. This simultor is written in C++, which is typically considered to be more efficient than MATLAB;
- Matching results to [35] with increased time discretization on a pre-trained TNN system;
- Preliminary results for reinforcement learning using TNN systems.

1.4. Thesis Organization

This thesis is structured as follows, Chapter 2 provides background on the topic of Artificial Neural Networks. Chapter 3 gives background on the Artificial Neural Network type Spiking Neural Networks. Chapter 4 introduces the topic of Temporal Neural Networks and explains all components of Temporal Neural Network systems. Chapter 5 lays out the key topics of this thesis as well as methodology of this thesis. Chapter 6 explains the implementation of the simulator. Chapter 7 discusses the setup of experiments, it presents the results of each experiment and provides a discussion. Finally, Chapter 8 contains the conclusions and key takeaways of this thesis.

2

Introduction to Artificial Neural Networks

This chapter gives an introduction to the subject Artificial Neural Networks, it discusses the composition and components of an Artificial Neural Network, it explains the configuration and initialization of an Artificial Neural Network, it lays out their use cases and it presents some prominently used types of Artificial Neural Network as well as less prominent, more niche types. Section 2.1 introduces the basic principles of Artificial Neural Networks, outlining their structure and core functionalities. Section 2.2 explores the key components and composition of Artificial Neural Networks, including neurons and network structure. Section 2.3 discusses the methodologies for training and learning in Artificial Neural Networks, covering essential techniques and paradigms. Section 2.4 presents various real-world applications and use cases where Artificial Neural Networks have demonstrated significant impact. Section 2.5 provides an overview of the fundamental and prominent Artificial Neural Network types, highlighting their unique characteristics and applications. Section 2.6 discusses less prominent Artificial Neural Network types, laying out their shortcomings and why they are less prominent.

2.1. Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models that are inspired by the structure and function of biological neural networks in the human brain. The human brain consists out of a large network of interconnected neurons. Each neuron operates as a cell executing one simple task: to accumulate input values, process them and then to produce an output. However, when these neurons form a network, a neural network, they are able to accomplish intricate tasks such as speech and image recognition [48]. Like a biological neural network the ANN consists out of nodes interconnected with each other as a network, similar to neurons. Neurons have a system that accumulates inputs from their neighbours and when this accumulation crosses a certain threshold, the neuron emits an output for its neighbours. Through this system of input accumulation, threshold crossing and output generation the brain performs computation. An ANN tries to emulate this process artificially.

2.2. Composition and Components of Artificial Neural Networks

To discuss the composition and components of an ANN, first a definition of each component is required. In [48], a definition is presented that encompasses each basic component. Zou et al. [48] name three critical components that each neural network has, these are: *node character*, *network topology*, and *learning rules*. Given the complexity of *node character*, *network topology*, and *learning rules*, it is necessary to break down these components into more fundamental concepts for a clearer understanding. *Node character* refers to the *nodes* or *neurons* of the network itself, what they do and how they function. *Network topology* refers to the *structure of the network*, how neurons are oriented in the network and how connections between neurons are established. Lastly, *learning rules* refers to *learning and training* of an ANN, to train an ANN to achieve the desired functionality and the process by which it learns

during training. This learning process is crucial because it enables the ANN to improve its performance and accuracy by adjusting its internal parameters according to the learning rules applied to the network. This section discusses the neurons and the network structure of an ANN, Section 2.3 discusses the training and learning.

In ANNs a neuron or node is the most basic computational unit. A neuron receives information as input, it transforms this input and it produces an output which is passed onto other neurons deeper in the network. The concept of a neuron in an ANN is inspired by biological neurons in the brain where a neuron receives signals via dendrites, it processes them in the cell body and then transmits an output signal through its axon to its neighboring neurons. Neurons in an ANN are a simplified abstraction of the biological model, nevertheless they capture the idea of signal processing and transmission, which is key to both biological and artificial neural networks. The transformation of an input typically is done by an activation function. An activation function introduces non-linearity in the network, which is required since most tasks have non-linear behaviour. In a neural network, neurons are responsible for identifying features in data, the activation function helps with identifying more complex features. Some commonly used activation functions are Sigmoid (sig, Equation 2.1), hyperbolic tangent ($tanh$, Equation 2.2) and Rectified Linear Unit ($ReLU$, Equation 2.3).

$$sig(t) = \frac{1}{1 + e^{-t}} \tag{2.1}$$

$$tanh(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}} \tag{2.2}$$

$$ReLU(t) = max(0, t) \tag{2.3}$$

In an ANN neurons are organized into layers, this is crucial for the networks ability to function. A layer consists out of a group of neurons that process input data simultaneously. Data is passed from one layer to the next, with each layer performing specific transformations on the data. This layered structure enables the network to learn patterns and representations in data, with each subsequent layer extracting more complex features. The significance of layers lies in their ability to build hierarchical representations of data. For example, in image recognition tasks, the initial layers may learn to detect simple features like edges, while deeper layers learn to recognize complex structures like shapes and objects. This progressive refinement of features is what allows neural networks to excel in tasks involving large amounts of unstructured data. The amount of neurons per layer is called the width of a network and the amount of layers in a network is called the depth of a network.

Neurons within a neural network are interconnected from layer to layer. Typically, neurons in a layer are only connected to neurons in layers adjacent to their layer. Connections between these neurons are weighted, these connections, often referred to as synaptic weights, determine the strength and influence of one neuron's output on the next neuron's input. The value of these weights is determined by the network's learning process. During the network's training phase, it adjusts the value of the synaptic weights based on the error between the output of the network and the target. This adjustment is guided by learning algorithms. A commonly used weight update rule is backpropagation, which iteratively fine-tunes the weights to minimize the error, allowing the network to learn from data and improve its predictions over time.

Neural networks typically consist of three main types of layers: input layers, hidden layers, and output layers. The input layer is where the raw data enters the network. Each neuron in this layer represents a feature of the input data, passing it to the next layer for further processing. Hidden layers are where most of the computational work happens. These layers perform transformations on the input data, enabling the network to learn patterns and representations. The term "hidden" refers to the fact that these layers are not directly observable from the input or output; they are internal to the network's structure. Finally, the output layer is responsible for transforming the processed data into a form that is interpretable and meaningful to humans. This layer reduces the high-dimensional representations learned by the network into outputs that align with the specific task at hand. Going back to the example of image recognition tasks, this layer outputs the label that classifies the image or the network outputs

a numerical value attached to the image. Some types of ANN have specialized layers, for instance Convolutional Neural Networks (CNNs) specialize in processing media, such as image, video or audio. For this task they have specialized layers, namely convolution layers and pooling layers to reduce the dimensionality of the data.

2.3. Network Training and Learning Methods

As mentioned before, before a network can be utilized for its intended task, it needs to be trained to achieve its intended functionality. Training is the process of feeding an ANN data, letting it produce outputs, comparing these to target values, calculating the error between the network's output and the target values and adjusting the weights according to learning algorithms based on this error. Training has multiple stages, first comes the initialization stage where a network is setup, training and test data is chosen and a weight update rule is chosen. Then comes the optimization stage, where training data is fed to the network, a hypothesis and loss function are established to calculate error between network output and target and the error of the loss function is minimized. Lastly, the regularization stage, where the network is checked for overtraining.

Some of the major skills of a neural network are its ability to recoginize features and patterns in data, and its ability to create new representation of data. These major skills are all based on (deep) learning. These abilities are inherent to a neural network, unlike for example machine learning algorithms that require an algorithm to extract features and patterns from data. For learning there are multiple methods: supervised or unsupervised learning, online or offline learning, and a special variant reinforcement learning. Subsection 2.3.1, Subsection 2.3.2 and Subsection 2.3.3 will explain the different training phases for ANNs, then Subsection 2.3.4, Subsection 2.3.5 and Subsection 2.3.6 will discuss different learning methods.

2.3.1. Network Training: Initialization

Before initializing an ANN, several decisions must be made to lay the foundation for the network's architecture, the learning process, and how data will be handled. The first decision is width and depth of the network. These parameters are typically chosen given the use case of the neural network. There is no established rule on what are the ideal width and depth of a network are, however more complex tasks usually require larger, wider and deeper, networks. Next, a decision needs to be made on what type of layers to use in the network. For example, if a task requires processing of media, then it is commonly best practice to use a convolutional neural network with layers specialized for these tasks. Then, an activation function needs to be chosen. Common choices, as mentioned before, include ReLU for its simplicity and efficiency, or sig and tanh for their ability to map inputs to a bounded range. Afterwards, the selection of a learning method, supervised or unsupervised learning, online or offline learning, or reinforcement learning. Following, the user needs to decide how data should be processed for the network to work with. How will the data be encoded so it can be input into the network, what kind of output does the network need to produce, which data will be used for training and which will be used for testing, how will the data be divided for multiple training cycles (usually called epochs). Lastly, a loss function, a learning rate and metric to calculate the average error on need to be selected.

After these parameters have been selected, the network can be initialized. The first step is to initialize all the weights in the network to a default value. This value can be the same for all weights or a method/technique can be used to select a default value for each weight. A commonly used technique is Xavier Initialization (or Glorot Initialization) [13]. For this technique there are two methods of picking weights, for the first, weights are randomly drawn from a uniform distribution within a range from $[-x, x]$ for $x = \sqrt{\frac{6}{n_i+n_{i+1}}}$, where n is the number of inputs for neuron i. For the second method, weights are randomly selected form a normal distribution with a mean of zero and a standard deviation $\sigma = \sqrt{\frac{2}{n_i+n_{i+1}}}$. The next step is to initialize biases in the network, an addition of a (usually small) value to the input before transformation. Typically there is no bias (bias set to zero), however some networks that for example use ReLU add a small positive bias to avoid dead neurons (neurons with all weights zero) at the start. A bias can also be added to better fit the network model to the data, due to the bias shifting the input by an arbitrary amount before it is transformed. Now, the network can be compiled and instantiated using the selected parameters, the initialized weights and biases, and it is ready for

training and optimization.

2.3.2. Network Training: Optimization

For the optimization phase, the goal is to train the network to the point where it is able to achieve its intended functionality. This is done by adjusting the weights and biases according to the calculated loss using the loss function. Before the loss can be calculated, a hypothesis function is required. This is a mathematical function that models the network, it takes a given input from the data, the data gets processed through this function and the output of the function is what the network should produce as output given the same input. The loss function takes the output of the hypothesis function and the output of the network and calculates the loss for all training samples in an epoch. Some commonly used methods for calculating the loss are Cross-Entropy Loss (Equation 2.4) and Mean Squared Error (MSE, Equation 2.5). Here m is the number of training samples in an epoch, $\hat{y}$ is the output of hypothesis function and y is the output of the network.

$$Loss = -\frac{1}{m}\sum_{i=0}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{2.4}$$

$$Loss = \frac{1}{m}\sum_{i=0}^{m}(\hat{y}_i - y_i)^2 \tag{2.5}$$

To calculate how much each weight and bias needs to be adjusted, the Gradient Descent (GD) optimization method is used. For each weight and bias, the gradient of the loss function with respect to the weights and biases is calculated. The gradient consists of partial derivatives of the loss function with respect to each parameter in the network. For each parameter the gradient is calculated as in Equation 2.6. Here $\frac{\partial L}{\partial \hat{y}}$ represents the derivative of the loss with respect to the output of the hypothesis function, and $\frac{\partial \hat{y}}{\partial \theta}$ represents the derivative of the network's predicted output with respect to the weight or bias (θ). After computing the gradients, the weights and biases are updated by subtracting the product of the gradient and the learning rate from the current parameter values. The learning rate is a chosen network parameter that determines the rate of learning of the network. A small learning rate leads slow convergence because the steps towards the minimum on the gradient are very small. A large learning rate can cause the algorithm to overshoot the minimum, leading to divergence or oscillations around the minimum.

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta} \tag{2.6}$$

GD can be done in multiple ways, the three most used variants are:

- Batch Gradient Descent, which uses the entire training data to compute the gradient and update the weights and biases. This method is computationally expensive for larger data sets, however it provides more stable and consistent convergence;
- Stochastic Gradient Descent, which updates the parameters using the gradient computed from single training instances. It is faster but is more prone to errors, due to noise between single training instances;
- Mini-Batch Gradient Descent, which is a compromise between the two previously mentioned methods. Here parameters are updated using the gradient from small batches of training data. This method is optimal for balancing speed and convergence stability,

The process of updating the weights and biases is repeated over many epochs until the loss function converges to a minimum. Convergence means that subsequent updates result in very small changes in the loss function, indicating that the optimal or near-optimal parameters have been found. This method is easily applied for single layer networks. However, for networks with multiple layers, the errors need to be propagated further back toward the input layer of the network. Here a second technique called backpropagation comes in.

Backpropagation

For backpropagation, there are two extra stages before the weights and biases of the network are

updated, these are the forward pass and the backward pass (backpropagation). During the forward pass, the input data is passed through the network, layer by layer, to compute the final output and the loss. This involves determining the output of each neuron in the network after applying the weighted sum of inputs and the activation function and calculating the loss based on this predicted output and the target value. For the backward pass, the chain rule from calculus is used to combine gradients between layers. First, the error at the output layer is calculated by calculating the gradient with respect to the network's output. Then, the gradient with respect to the weights and biases in the output layer is calculated. This error is then propagated backwards, for each hidden layer the error with respect to its output is calculated, and the chain rule is then used to combine the error from the current layer and the next layer. After combining errors, the gradient for weight and biases in that layer is calculated and the weights and biases are updated.

2.3.3. Network Training: Regularization

The regularization phase aims to assess whether the network has been overtrained, this issue is commonly referred to as overfitting. A network experiences overfitting when it is only able to achieve its intended functionality on the training data, but not on the test data or actual data. Some commonly used methods to prevent overfitting are L1/L2 Norm, Early Stopping and Dropout.

For the L1/L2 Norm, a penalty is added to the loss function. For the L1 Norm, also named L1 Regularization or Lasso Regularization, a penalty equal to the sum of the absolute values of the network's weights. How this is applied is explained in Equation 2.7, where L is the loss and the L1 norm is on the right, where λ controls the strength of the regularization, a higher value increases the penalty, and W_i represents the weights of the network. L1 Regularization introduces sparsity into the network, where large number of its weights are equal to zero. This results in a simpler model with fewer active parameters. It has the effect of pushing many of the weights towards zero. This is because the L1 penalty increases linearly with the magnitude of the weights. As a result, during optimization, the model tends to drive some weights to exactly zero to minimize the total regularization penalty, especially when λ is large. This results in automatic feature selection, where features associated with zero weights are excluded from the model, improving model simplicity and potentially enhancing generalization.

$$L_{regularized} = L_{original} + \lambda \sum_i |W_i| \tag{2.7}$$

For the L2 norm, also named L2 Regularization or Ridge Regularization, a pentalty equal to the sum squared values of the network's weights. How this is applied is explained in Equation 2.8, where L is the loss and the L2 norm is on the right, where λ controls the strength of the regularization and W_i represents the weights of the network. L2 regularization deters large weights by penalizing them more heavily. By having smaller weights, the complexity of the network model is reduced. When both L1 and L2 regularization are applied in loss calculation it is called Elastic Net Regularization.

$$L_{regularized} = L_{original} + \lambda \sum_i W_i^2 \tag{2.8}$$

For Early Stopping, the network is monitored between training epochs using the test set. It's performance is measured and compared to the previous epoch, if the performance no longer improves or it deteriorates, training is stopped. This helps prevent overfitting by stopping training before the model starts the memorize the actual data instead of learning patterns and features in the data. [45] has found that accuracy of the network was higher when Early Stopping was applied, compared to no usage of Early Stopping.

For Dropout, during training of the network, a random subset of neurons is "dropped out", this entails setting all their weights to zero. The amount of neurons to drop out is determined by a ratio of the total number of neurons. This prevents the network from relying too much on certain neurons in the network. [39] as found that accuracy of the network was higher when Dropout was applied, compared to no use of Dropout.

2.3.4. Supervised and Unsupervised Learning

Learning of a neural network can be done in a supervised or unsupervised manner. For supervised learning, all data is labelled. Networks trained in a supervised manner can be used, for instance, for classification or regression. The output of the network is then a label in the form of a class or in the form of a number or value. An example of classification is a network trained to recognize pictures of cats or dogs and to output the label 'cat' or 'dog'. An example of regression is a network trained which take as input a picture of a house and to output a potential value of the house. During network optimization, the hypothesis function maps the inputs to the correct classification labels or numerical values. Loss minimization is then achieved by reducing the number of incorrect labels.

For unsupervised learning, data is entirely unlabelled. Neural networks trained using unsupervised methods must analyze the input data to identify patterns and features based on its inherent structure and characteristics. Common examples of tasks in unsupervised learning are data clustering, dimensionality reduction of data or data quality enhancement. The optimization stage is a core part of unsupervised learning. A crucial aspect of unsupervised learning is the optimization stage, where the network's hypothesis function aims to approximate the reconstruction of the input data. The loss function, in this context, measures the discrepancy between the reconstructed data and the original input, guiding the network in structuring the data effectively. An iterative process is applied where, in each training epoch, parameters are adjusted to better reconstruct the input data. In the next epoch, loss is calculated using the improved reconstructed input data, and the network's performance can be improved even further.

An example of unsupervised learning is k-means clustering, where k denotes the number of clusters into which the data is to be divided. The data is clustered around k centroids. The assignment of data points to centroids is determined by the hypothesis function. For the first training epoch, the centroids are randomly given random coordinates. After the first epoch, the centroids are recalculated as the mean of all data points in each cluster. This is done by averaging the coordinates of all points assigned to each cluster. For k-means clustering, loss is the sum of squared distances from each data point to its assigned centroid. By adjusting the centroids, the loss is minimized, as the centroids better fit the data and each data point is closer to its assigned centroid, resulting in an improved reconstruction of the input data.

2.3.5. Online and Offline Learning

Learning of a neural network can be done in an online or offline manner. Offline learning has been mentioned before, where training data is divided in batches, training is done in epochs and a test set is used to validate the functionality of the model. Online learning takes place as data becomes available. Online learning is used in dynamic environments, where learning systems receive data as a continuous flow, here the network needs to be able to adapt to rapidly changing conditions. Online learning is additionally applicable to environments where there the quantity of data is too large to divide it into batches, moreover such a large data quantity would result an amount of training epochs that is practically infeasible. Therefore, online learning is employed in scenarios with enormous data quantities, where the model continuously updates and learns from data in a sequential manner.

2.3.6. Reinforcement Learning

Learning can additionally be done in a unique manner different from supervised or unsupervised learning. This method of learning is based on trial and error and is called reinforcement learning. The network is given an input, processes this and produces an output, based on this output a reward signal is generated by the environment the network is operating in. This reward signal tells the network whether it has produced a correct or incorrect output. Based on this reward, the network adjusts its network parameters according to a set of pre determined rules. This reward signal can be produced automatically by the environment based on a set of rules or it can be manually generated by the users of the network.

An example of reinforcement learning in neural networks is a neural network used as a classical control system. Take for example a neural network that is implemented to control a faucet, which waters the soil of a plant. If the soil reaches a satisfactory moisture level, the faucet needs to be closed and if the soil becomes too dry, the faucet needs to be opened. The input of the network is the moisture

level of the soil (s) and the output of the network are the actions 'open faucet', 'close faucet' or 'do nothing'. The environment (the soil) has two thresholds, one where the soil is too dry ($s < \theta_d$) and one where the soil is too moist ($s > \theta_m$). The goal of the network is to learn what action to do given the circumstances and to learn where these thresholds are without knowing them exactly. The network would receive a reward signal based on the conditions of the soil and what it outputs, an overview of this can be seen in Table 2.1. For a positive reward ($+R$), the network parameters are positively adjusted and for a negative reward ($-R$), the network parameters are negatively adjusted. As this happens in an automated fashion, the network quickly learns when to have the faucet open and when to have the faucet closed, resulting in highly efficient training. Reinforcement learning also has its downsides, as tasks become more complex, specifying when and how to assign positive or negative rewards can become intricate. This is because the agent must learn not only the immediate consequences of its actions but also how they affect long-term goals. This can potentially introduce scalability issues for highly complex tasks, with difficult to define specifications for reward signals.

Condition	Action	Reward (R)
$s < \theta_d$	Open faucet	$+R$
$s < \theta_d$	Close faucet	$-R$
$s < \theta_d$	Do nothing	$-R$
$s > \theta_m$	Open faucet	$-R$
$s > \theta_m$	Close faucet	$+R$
$s > \theta_m$	Do nothing	$-R$
$\theta_d \leq s \leq \theta_m$	Open faucet	$-R$
$\theta_d \leq s \leq \theta_m$	Close faucet	$-R$
$\theta_d \leq s \leq \theta_m$	Do nothing	$+R$

Table 2.1: Reward signal conditions for example of neural network using reinforcement learning implemented as control system for plant waterer

2.4. Use Cases and Applications

Having explored the fundamental concepts and learning mechanisms of ANNs, it is crucial to understand the wide range of practical applications where these networks have been successfully implemented. ANNs have become a crucial part of modern computational intelligence due to their ability to execute complex tasks and make predictions based on the data they are trained on. The following section provides an overview of some prominent use cases and applications across various domains.

One of the most well-known applications of ANNs is in the field of processing media, extracting features from images, video and audio. Convolutional Neural Networks (CNNs), a specialized type of ANN, have demonstrated to be especially effective at these tasks. For example, CNNs have been used to identify anomalies in x-ray images to identify breast cancer [47].

Another well-known application of ANNs is Natural Language Processing (NLP). NLP is especially well executed by Recurrurent Neural Networks (RNNs) and Transformers, due to their excellence in processing sequential data. Large Language Models (LLMs) represent some of the most prominent applications of artificial neural networks, widely adopted at time of writing due to their ability to generate human-like text, understand context, and perform complex language tasks with high accuracy. Examples of such models include OpenAI's ChatGPT, Google's Gemini, and Microsoft's Copilot. Additionally, these networks are able to take a natural language prompt as input and generate media (images, video) based on this prompt.

The various applications of ANNs discussed above underscores their versatility and applicability. However, the effectiveness of these applications is heavily dependent on the choice of neural network architecture. As seen in the given examples, each of them use a specialized network variant, designed to achieve optimal performance for that task. The following section explores the fundamental and most prominent types of ANN, laying out the characteristics and features that make them a fundamental or prominent type of ANN.

2.5. Fundamental and Prominent Artificial Neural Network Types

In light of the diverse applications of ANNs, it is fundamental to understand the specialized neural network architectures that enable their success. This section lays out a selection of the most fundamental or prominent types of ANN. Variants of these networks are discussed in their respective subsections. ANN architectures not included are discussed in Section 2.6. If an architecture is omitted from both Section 2.5 and Section 2.6 it is due to the architecture being combinations of the fundamental/prominent ANN types or because they belong to computational paradigms not exclusive to ANNs, such as Echo State Networks and reservoir computing.

2.5.1. Feedforward Neural Network/Multilayer Perceptron

The first of the two fundamental types of is the Feedforward Neural Network (FFNN) or Multilayer Perceptron (MLP). The perceptron is a concept that models an artificial neuron. It was first introduced by [26], later it was first implemented by [33]. Essentially, a perceptron receives multiple input signals, each having an associated weight. These weighted inputs are summed up, and a bias is added to produce a weighted sum. This weighted sum is then passed through an activation function, which determines the output of the perceptron. Multiple perceptrons can be combined to form a Multilayer Perceptron (MLP) by connecting individual perceptrons in a layered architecture, where the output of one layer serves as the input to the next layer. The network can capture and learn complex patters by passing data through multiple layers of activation functions that introduce nonlinearity into the data, which can be useful to find nonlinear relations in the data. These neural networks are the most basic form of FFNN, which is why they are often called that.

An MLP typically has one input layer, one or multiple hidden layer(s) and an output layer. Some commonly used activation functions in MLPs are the ones mentioned in Section 2.2. MLPs are typically trained in a supervised manner using backpropagation. The MLP is one of the most basic forms of neural network and is therefore often used for the simpler tasks mentioned in Section 2.3. Data classification and regression, pattern recognition or anomaly detection.

A variant of the MLP is the Radial Basis Function Network (RBFN), where there is only one singular hidden layer and the activation function is a radial basis function. This concept was first introduced by [30]. RBFNs are effective at function approximation, to approximate an unknown function based on input and output data, or interpolation, to estimate the values of a function at unknown points within a given range based on known inputs.

2.5.2. Recurrent Neural Network

The Recurrent Neural Network (RNN) is one of the most basic forms of feedback neural network. The RNN was first introduced by [10], in this paper a form of neural network is presented that has memory, through the use of recurrent (feedback) connections. How the RNN differs from the MLP is that each node in the hidden layer has an extra connection to itself, this forms the recurrent connection. Through these feedback connections the sum of the inputs not only depends on the inputs from the previous layer, but also the output of itself that loops back to its own input through the recurrent connection. Therefore, the output of the network is influenced by the order of input of the data.

RNNs utilize the same activation functions as MLPs. RNNs are highly effective at processing sequential data, due to their memory capabilities. The training and learning of RNNs is therefore also done in a sequential manner, which can be done both supervised or unsupervised, based on the preferences of the user of the RNN. RNNs are often used for natural language processing, time series prediction, such as predicting weather, or speech recognition, due to sequential data typically being encountered in these contexts.

The problem with basic RNNs is that there is no differentiation between what to “memorize” and what not to “memorize”, which makes it unable to capture long term dependencies in the sequential data. To combat this issue, two variants of RNNs are discussed with mechanisms to selectively remember and forget information in the memory nodes of the network. These are the Long Short-Term Memory (LSTM) and the Gated Recurrent Unit (GRU).

Long Short-Term Memory

The Long Short-Term Memory (LSTM) is a variation on the RNN. First introduced by [15], this variation has mechanisms to selectively remember and forget information in the memory nodes of the network. This enables LSTMs to better capture long term dependencies in data. The network achieves this by keeping track of a Cell State and hidden state of each node. The Cell State represents the internal memory of the node and the hidden state represents the information or data that flows through the node. Additionally, it introduces multiple stages in which the data is processed. These stages are called "Gates" of which there are three: the Forget Gate, the Input Gate and the Output Gate. First, the hidden state (information/data) from the previous time step ($t-1$) and the current time step (t) are passed through the Forget Gate, here a mathematical function determines what information from $t-1$ and t should be forgotten. The Cell State is then updated based on the output of the Forget Gate. Then, the hidden state from $t-1$ and t goes through the Input Gate, which determines what values in the Cell State should be updated or added in the memory of the cell. The output of the Input Gate is then multiplied with a vector of candidate values to update the Cell State. These candidate values are produced by another mathematical function based on the hidden state from $t-1$ and t. Lastly, the hidden state from $t-1$ and t are passed through the Output Gate to determine what information should be output and what information should be passed on to the next time step ($t+1$). The output from the Output Gate is then multiplied with the Cell State to create the definitive output of the node. This information is then output to the next node and to $t+1$ of itself. The LSTM is trained in the same manner as an RNN and it is used for the same applications.

Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is another variation on the RNN. First introduced by [9], this variation functions similarly to the LSTM. Like the LSTM, it uses a hidden state and "Gates" to process information. The hidden state in the GRU is a combination of the Cell state and the hidden state of the LSTM. The GRU uses two gates instead of three: the Update Gate and the Output Gate, here the Update Gate is a merged version of the LSTM's Input Gate and Forget Gate. First, it takes the hidden state from $t-1$ and new information input from t, these are processed through the Update Gate to update the hidden state. Then, the hidden state from $t-1$ and new information input from t are passed through the Output Gate to determine the output of the node. The generated output is used to update the hidden state of the node through a mathematical function and then the hidden state is output. Like the RNN and LSTM it has the same use cases and it is also trained in the same manner.

2.5.3. Convolutional Neural Network

The Convolutional Neural Network (CNN), first introduced by [20], is a type of neural network that uniquely uses a combination convolution and pooling in its layers. Due to its usage of convolution and pooling it excels at processing media such as images, audio, and video for classification and regression.

First convolution is done, where a filter, typically called a kernel, is applied to the data. The filter is a small matrix of weights that slides over the input data, computing the dot product at each step. Through this process, local features and patterns can be detected. For example for a 500 by 500 image, take an area of 50 by 50 pixels. Start in the top left of the image and for each step slide the 50 by 50 window to the right by one pixel. After reaching the border on the right, move one pixel down and start on the left again. The amount the window is moved by, named the "stride", is determined by whether the user of the CNN wants a more detailed feature extraction, such as a stride of one pixel, or less detailed feature extraction, such as a stride of two (usually not more), in exchange for higher dimensionality reduction in the next layer and faster or more efficient processing.

After convolution a non-linear activation function is applied to introduce non-linearity into the data. The most commonly used activation function is the Rectified Linear Unit (ReLU). It replaces all negative pixel values in the feature map with zero and leaves positive values unchanged.

After the non-linear activation function comes the pooling step. In this step a down sampling operation is done that reduces the dimensionality of each feature map but retains key information. The most common pooling operation is max pooling, where a small window (e.g. 2x2) slides over the feature map, and only the maximum value in each window is preserved. This helps in retaining the most relevant information while discarding less important details and reducing the computational load. The three steps of convolution, activation, and pooling are done in what are called the convolutional layers of the

network. These three steps are repeated until the CNN user is content with the reduced dimensionality of the data.

After multiple cycles of convolution, activation and pooling comes the flattening step. After multiple cycles of the three steps mentioned before, the data remains multidimensional. To further process it in a CNN, the data must be flattened to one dimensional vector. This one dimensional vector can then be used as input for the fully connected layers which come after the convolutional layers.

The final step in CNNs is either classification or regression depending on its use case. This is done by the fully connected layers, where each connection between nodes has a weight. This weight is determined by the training of the CNN based on the relationships between features in the input data and the final output. Again, an activation is applied to the output of each node. The final layer is then the output layer which has nodes corresponding to the number of classes in a classification task or a single node for regression tasks.

Training of the network is typically done using backpropagation. Data is fed through the network and the network does its classification or regression task. Then, the output of the network is compared to the actual labels or classes for the classification task, or the output value compared to the actual value for the regression task. The loss is determined based on this comparison, which is then used for Gradient Descent optimization.

As mentioned before, CNNs are useful for classification and regression tasks. For example, for classification, an image of an animal can be fed to the CNN and the output will say what animal is on the image. For regression, a picture of a house can be fed into the CNN and the CNN will give an approximate price of that house. For example, the CNN introduced by [20] excels at recognition of handwritten characters and Google's GoogLeNet [41] performs well in image recognition, feature classification and object detection.

2.5.4. Autoencoder

The Autoencoder (AE) is a type of Feedforward Neural Network, that is designed to compress and decompress information automatically. This type of Neural Network was first introduced by [8]. Data is fed into the input layer of the AE, in the hidden layers up to the middle hidden layer, the input is compressed. From the middle hidden layer to the final hidden layer the data is decompressed and then output, this makes AEs symmetrical. Compression is achieved by reducing the width of each layer up to and including the middle hidden layer. Decompression is achieved by increasing the width of each layer from the middle hidden layer to the output layer. AEs are usually trained using backpropagation, where the error is calculated based on what was input and how the new compressed, then decompressed output differs from the original input. AEs are mostly used for data compression and data dimensionality reduction. The middle layer that contains the compressed data could be read out to store the compressed data, or to analyse the lower dimensional representation to find patterns that are not visually present in the original.

Variations of AEs are the Variational Autoencoder (VAE), the Denoising Autoencoder (DAE) and the Sparse Autoencoder (SAE). VAEs, first introduced by [18], is a type of Autoencoder that encodes and decodes based on a probabilistic distribution of the encoding space. This makes the goal of the VAE to encode input data into a lower dimensional space with the added feature of modeling a probability distribution in that space. DAEs, first introduced by [44], are an Autoencoder variation designed to handle noisy input data. VAEs are trained using noisy input data, the error is then computed between the network's output and the noiseless input data. Through this method of training it looks at larger or broader features in the data that are not as sensitive to noise unlike smaller, less broad features. SAEs, first introduced in [32], are a type of Autoencoder that in a certain sense do the opposite of a regular Autoencoder. Instead of encoding the data into "less" space, SAEs aim to encode data in "more" space. The hidden layers in SAEs have more nodes, however during training a sparsity constraint is imposed, where only a small fraction of neurons in the hidden layer are active at any given time. By enforcing sparsity, these networks encourage the network to learn a smaller set of active neurons (features), which often correspond to important, high-level patterns in the data.

2.5.5. Boltzmann Machine

Before Boltzmann Machines can be explained, another network type needs to be discussed. This network, named the Hopfield Network (HN), forms the basis of Boltzmann Machines. HNs, first introduced by [16], are single layer Feedback Neural Networks that are fully connected. Each node serves as input before training, during training they are then hidden and output is read from each node after. HNs are designed to store and retrieve patterns or states. For training, the patterns or states it should "remember" are given and then weights can be adjusted. After training, a partial state or pattern can be input and the network will converge to a "remembered" state.

Boltzmann Machines (BMs), first introduced by [14], are single layer Feedback Neural Networks that are fully connected. How it differs from HNs, is that a set of nodes serve as input nodes and the rest are hidden. At the end of a full network update, the input nodes become output nodes. Each node in the network is binary, which means it can only take the value 0 or 1. BMs are stochastic, which means they have some elements of randomness to them. The weights of the connections in BMs are updated using stochastic gradient descent or a variant. Stochasticity is introduced through the random sampling of states during learning. The activation of the neurons is controlled by a global value that determines the stochasticity. If this value is higher the network will explore more different states and it will add more randomness. If this value is lower the network will stabilize into more stable and deterministic patterns. BMs are often used for unsupervised feature learning, allowing the network to automatically discover relevant features in the input data. BMs are also used for dimensionality reduction, capturing essential patterns in high-dimensional data. Additionally, they are used as recommender systems to give personalized content to users of an application, such as social media content algorithms.

A variant more popular than the original is the Restricted Boltzmann Machine (RBM). RBMs, first introduced by [38], are a variant on the Boltzmann Machine in which the self-connections between input nodes and the self-connections between hidden nodes are removed. Except for the removal of self-connections input layer and hidden layer remain fully connected. This restriction simplifies the learning process and makes training more efficient. Due to the absence of connections within the same layer, the computation of conditional probabilities that determine the activation of nodes is simplified during the learning process.

A new type of network can be made by "stacking" RBMs and/or VAEs, these are called Deep Belief Networks (DBNs), first introduced by [4] (only using RBMs). Each building block of the stack is a singular RBM or VAE, where each building block only has to learn to encode the previous network. [4] dubs it greedy training, where "greedy" indicates making locally optimal choices to reach a satisfactory, although possibly not optimal, answer. Due to their structuring, DBNs are effective at learning hierarchical, abstract features from raw data in an unsupervised manner. They can automatically discover patterns in data without needing labeled examples, making them valuable for tasks where labeled data is scarce or expensive to obtain.

2.5.6. Transformer

Transformers also known as Attention Networks, first introduced by [43], are a type of neural network that use attention mechanisms to mitigate information decay in the network. Analogous to how recurrent neural networks address decay through memory. However, instead of relying on memory, transformers utilize attention to effectively preserve information across the input sequence.

In the attention mechanism, each word or token in the input sequence is associated with three vectors: a query vector, a key vector, and a value vector. These vectors are derived from the representation of the input in the vector space itself and serve as the basis for calculating attention scores. The attention mechanism computes the attention scores between a query vector and key vectors of all other elements in the input sequence. This computation typically involves a dot product operation followed by normalization using a softmax function ($(\sigma(\vec{z})_i = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}}$, where i is the input, j is the output, for $i = 1, ..., K$). The result is a set of attention weights representing the importance of each token in relation to the input element that is being processed, called the query token.

The attention weights obtained are used to compute a weighted sum of the value vectors. The weighted sum operation combines information from all input elements in the sequence into a single vector representation, called the context vector. This context vector captures the most relevant information from

the input sequence with respect to the query token. By giving more weight to elements deemed more relevant based on the attention scores, the weighted sum ensures that the context vector contains a representation of the most important information for the query token. The context vector is used to enhance the initial representation of the query token, adding context based on the full input sequence. This enhanced version of the query token is then used in the subsequent steps.

To capture different aspects of the input sequence effectively, many transformer architectures employ multi-head attention. This involves performing attention computations in parallel multiple times, each with different sets of learned query, key, and value projections. The resulting attention heads capture diverse aspects of the input, allowing the model to attend to different parts of the sequence in parallel.

A transformer can be divided into two components. The first is the previously explained (often multi-head) attention component. The second component is a Feedforward Neural Network, that functions as explained in Subsection 2.5.1. The attention mechanism allows transformers to capture long-range dependencies and improves the model's ability to capture intricate patterns in the data. Transformers are trained in a supervised manner, where weights and parameters in the network are adjusted using backpropagation. As mentioned in Section 2.4, transformers perform well in text and language processing tasks.

2.6. Less Prominent Types of Artificial Neural Networks

Next to the prominent ANN types, there are also less prominent more niche ANN architectures. These include Radial Basis Function Networks (RBFNs), mentioned in Subsection 2.5.1, Self Organizing Maps, Hopfield Networks, mentioned in Subsection 2.5.5, and Spiking Neural Networks. These networks are less prominent for different reasons.

The before mentioned RBFNs are less prominent due to their limited use cases and scalability uses. RBFNs are often overshadowed by more versatile architectures like MLPs and CNNs, which can handle a broader range of tasks. RBFNs, due to their small depth, do not scale well to large datasets or complex tasks, limiting their applicability in modern, large-scale machine learning problems.

Self Organizing Maps (SOMs), first introduced by [19], is a type of neural network capable of unsupervised data classification. This network is trained in an unsupervised manner, where data is input and the network examines its neurons to identify the one that most closely corresponds to the input. If a neuron is identified as the best match for a particular input, its position or weights are adjusted to become more similar to the input. Neurons in a SOM are often organized in a topological map, therefore when a neuron's position or weight is adjusted, its neighbours' position or weight is adjusted as well. This is done to preserve the spatial relationships in the input space, encouraging similar inputs to map to neighboring regions. The extent to which neighbors are moved depends on the distance of the neighbors to the closest matching nodes. Nodes that are closer to the best-matching neuron experience a stronger influence or adjustment compared to neurons that are farther away. SOMs are less prominent due to their limited use cases and difficulty of interpretation. SOMs are primarily used for visualization and clustering, but other methods like t-distributed Stochastic Neighbor Embedding (t-SNE), Uniform Manifold Approximation and Projection (UMAP), or k-means clustering often outperform them in these tasks. While SOMs provide a visual representation of data, interpreting the map in practical applications can be challenging, leading to limited adoption.

Hopfield Networks (HNs), as mentioned in Subsection 2.5.5, are less prominent due to capacity limits and convergence issues. Hopfield networks, due to their limited depth and width, have a limited capacity for storing patterns, making them impractical for large-scale applications. They may not always converge to a correct pattern, particularly in cases of noisy or incomplete input, limiting their reliability.

Spiking Neural Networks (SNNs), are a type of ANN inspired by the biological neural networks found in the brain. Unlike traditional ANNs, which primarily use continuous-valued activations and operate in discrete time steps, SNNs communicate through discrete, asynchronous events called spikes, which model the firing of neurons in biological systems. The concept was first introduced by [27] in 1989, which introduced VLSI (Very Large-Scale Integration) circuits inspired by biological neural systems. The concept was later refined by [17], where a simplified model of spiking neurons was introduced and its computational capabilities were demonstrated. SNNs are less prominent due to their complexity and

due to the limited amount of tools and frameworks. SNNs typically require more complex computational models and algorithms compared to traditional ANNs, making them difficult to implement and train. Furthermore, there is a lack of mature software frameworks and tools for SNNs, making it harder for researchers and practitioners to experiment with and deploy these networks. This brings SNNs into a stalemate situation, due to the lack of frameworks and tools hardly an research is done and because hardly any research is done no new tools or frameworks are built. Add to this the complexity factor of doing implementing SNNs, this makes SNNs a difficult topic to research.

However, since SNNs model biological neural networks, they can be highly energy-efficient. They operate using sparse, event-driven computations rather than continuous signal processing. This characteristic makes them particularly well-suited for low-power devices and applications where energy efficiency is critical, such as in embedded systems. Furthermore, SNNs are inherently good at processing temporal data and sequences, which makes them ideal for tasks involving time-dependent patterns, such as speech recognition, event-based sensory processing, and real-time data analysis. Their ability to capture and process time dynamics more naturally than traditional neural networks could lead to breakthroughs in these areas. Moreover, SNNs can leverage biologically inspired learning mechanisms like Spike Timing Dependent Plasticity, where the timing of spikes influences synaptic strength adjustments. This type of learning could enable more efficient and adaptive learning processes, potentially reducing the amount of data and time required to train networks compared to traditional ANNs.

The aforementioned reasons represent just a few examples of why research in the field of Spiking Neural Networks is both compelling and valuable. While SNNs face challenges in implementation and training, their potential for energy efficiency, temporal processing and biological plausibility makes them a promising area of research. As technology advances and new methods for training and deploying SNNs are developed, they could become more prominent in areas where traditional ANNs are less efficient or effective.

3

Introduction to Spiking Neural Networks

This chapter gives an introduction to the subject of Spiking Neural Networks, it discusses the composition and components that are unique to Spiking Neural Networks and presents the different types of Spiking Neural Network. Section 3.1 introduces the basic principles of Spiking Neural Networks and their core functionalities. Section 3.2 explores the key components and composition of Spiking Neural Networks. Section 3.3 presents the two main types of Spiking Neural Network and explains the significance of each type.

3.1. Spiking Neural Networks

To repeat Chapter 2, Section 2.6, Spiking Neural Networks (SNNs) are a type of ANN inspired by the biological neural networks found in the brain. Unlike traditional ANNs, which primarily use continuous-valued activations and operate in discrete time steps, SNNs communicate through discrete, asynchronous events called spikes, which model the firing of neurons in biological systems. The concept was first introduced by [27] in 1989, which introduced VLSI (Very Large-Scale Integration) circuits inspired by biological neural systems. The concept was later refined by [17], where a simplified model of spiking neurons was introduced and its computational capabilities were demonstrated.

3.2. Composition, Components and Information Processing of Spiking Neural Networks

In SNNs, information is encoded in the form of spikes or action potentials. Each neuron accumulates incoming spikes from its connected neurons. When the accumulated input surpasses a certain threshold, the neuron generates an output spike, which can then propagate to other neurons in the network. This threshold-crossing mechanism allows SNNs to process information in a highly parallel and event-driven manner, mimicking the behavior of biological neurons. SNNs are trained using traditional methods, such as backpropagation, or methods unique to them such as Spike Timing Dependent Plasticity.

3.3. Types of Spiking Neural Network

SNNs can be divided into two types, rate-based SNNs and time-based SNNs. In rate-based SNNs, the encoding of information is determined by the rate at which spikes are received by a neuron within a fixed time window. This rate, or the number of spikes, serves as the primary mechanism for representing values, similar to how traditional Artificial Neural Networks (ANNs) process numerical inputs within a defined range. This method of encoding is commonly referred to as rate coding. The key distinction between rate-based SNNs and traditional ANNs lies in the former's reliance on spikes, where the frequency of spikes within a set period encodes the relevant information.

In contrast, time-based SNNs, often referred to as Temporal Neural Networks (TNNs), encode information based on the timing of individual spikes relative to a specific time cycle or a reference event, such as the initiation of spike timing. This method, known as temporal coding, is notable for its efficiency and distinctiveness compared to rate-based approaches. While rate-based SNNs focus on spike frequency, temporal coding leverages the precise timing of spikes to convey information, potentially offering greater energy efficiency. This efficiency arises from the reduced number of spikes required to transmit information in time-based SNNs, making them a compelling area of research.

The majority of contemporary neural network research is centered on rate-coding networks, which dominate practical applications due to their close alignment with traditional ANN methodologies. However, time-based SNNs offer possibilities, particularly in terms of energy efficiency and their hypothesized alignment with biological neural systems. Some researchers [37] propose that biological neural networks may utilize temporal coding rather than rate coding, a topic explored further in Section 4.5.

4

Temporal Neural Networks

This chapter gives a brief overview on the subject of Temporal Neural Networks, it discusses the composition and components that are unique to Temporal Neural Networks, it lays out the training and learning methods for Temporal Neural Networks, it presents work already done on the subject, it explains the connection between Temporal Neural Networks and biological neural networks and it discusses ways of validating and verifying Temporal Neural Network concepts introduced by work previously done on the subject. Section 4.1 introduces the basic principles of Temporal Neural Networks. Section 4.2 discusses the composition of Temporal Neural Networks, explains the model used for neurons in the network and lays out each core component of Temporal Neural Networks. Section 4.3 presents the training and learning methods utilized by Temporal Neural Networks. Section 4.4 discusses work already done on the subject of Temporal Neural Networks. Section 4.6 explains how the concept of Temporal Neural Networks can be validated in this thesis.

4.1. Temporal Neural Networks

Time-based Spiking Neural Networks, also commonly known as Temporal Neural Networks (TNNs), are a type of ANN inspired by the biological neural networks found in the brain, which communicate through discrete, asynchronous events called spikes, which model the firing of neurons in biological systems. TNNs specifically encode information based on the timing of individual spikes relative to a specific time cycle or a reference event, such as the initiation of spike timing. This form of information encoding is dubbed temporal coding and is notably more efficient compared to rate-based approaches, due to the need for less spikes to encode information. The concept of Spiking Neural Networks with temporal coding was first introduced by [22]. The term Temporal Neural Network is coined by James E. Smith in [37]. Smith is one of the leading researchers in the field of Temporal Neural Networks having written a book [37] and multiple papers [34], [35], [36] surrounding the topic.

4.2. Composition, Components and Information processing of Temporal Neural Networks

TNN neurons are modeled after the Leaky Integrate and Fire (LIF) model [1] or the Spike Responde Model (SRM) [12]. LIF and SRM are extensions of Louis Lapicque's Integrate and Fire (I&F) Model [1]. The I&F model is one of the simplest models to describe neuronal activity. It abstracts a neuron to its core functional components, focusing on how the membrane potential accumulates over time due to incoming synaptic inputs (spikes). The model integrates incoming signals until a threshold is reached, at which point the neuron "fires" (outputs) a spike (action potential) and then resets its membrane potential. The I&F model captures essential aspects of real neurons, particularly the threshold mechanism for action potential generation. The LIF model is an extension from Lapicque himself, to model how incoming spikes accumulate membrane potential. The SRM is an extension of the I&F model that incorporates a more detailed description of the neuron's response to spikes, including the aftereffects of each spike on the membrane potential. It uses a response function that characterizes

how each incoming spike affects the membrane potential over time, capturing both the immediate and delayed effects of input spikes. The SRM takes into account the history of the neuron's activity, making it more biologically plausible. Real neurons have memory-like properties where past spikes influence the likelihood of future spikes. The combination of the response function and the weight of the synapse determines how much membrane potential should be accumulated. If the sum of potentials over all synaptic inputs crosses the threshold, an output spike is generated.

There is one fundamental response functions that models how an incoming spike and the weight of the synaptic input affect the membrane potential. This response function comes from the LIF model. A synaptic input accumulates potential up to a certain level, after it reaches this level, membrane potential decays over time, until a new synaptic input accumulates more potential. The amount of membrane potential accumulated by a spike is decided by the weight of synapse on which the spike is input. An example of this process is shown in Figure 4.1.

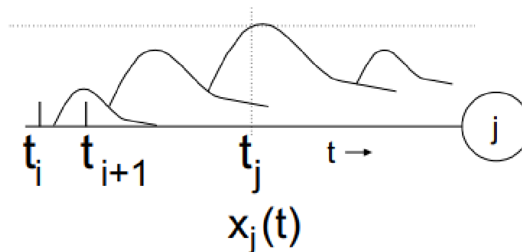


Figure 4.1: Example of Leaky Integrate and Fire

Like traditional ANNs, TNNs are structured in layers, for TNNs these are also called columns. However, TNN layers typically have extra components, which perform inhibition on the spikes before they are input to a layer or output by the layer, analogous to inhibition in biological neurons. Usually, input spikes need to be filtered and not all output spikes produced by a layer should be propagated to the next. For input spikes, only relevant information should be input into a layer, therefore inhibition takes place to filter out unwanted synaptic inputs. For output spikes, often times, synaptic inputs accumulate potential over multiple neurons and multiple neurons will fire output spikes. However, only the spikes that carry important information should be propagated to the next layer, therefore inhibition takes place to only propagate the spikes that carry fundamental information. One of the most simple and often used forms in TNNs is k-Winner Takes All (k-WTA) inhibition [37]. Here only the first k spikes fired by the layer are allowed to go to the next column, these are the "winners". Spikes fired after the first k spikes are held back by the inhibition component.

4.3. Network Training and Learning Methods

Training of TNNs can be done using classical methods like backpropagation or a method unique to them called Spike Timing Dependent Plasticity (STDP), which comes from the biological phenomenon with the same name. For instance, in [7] backpropagation on TNNs is performed by calculating the error between desired firing times of spikes and actual firing times. Spike Timing Dependent Plasticity is a phenomenon first suggested in [40], then researched and discovered by Levy and Steward [21]. Further research by Markram et al. [23] definitively demonstrated the occurrence of the phenomenon and research by Bi and Poo [5] further refined the exact details of the STDP process. In biology, the STDP process slightly strengthens inputs that on average help accumulate membrane potential for output spikes. Additionally, the opposite happens, where inputs are slightly weakened that have incoming spikes after an output spike has been produced. This process gives it the name "Spike Timing Dependent Plasticity", inputs that might be the cause of an output spike firing are made more to likely to contribute in the future, whereas inputs that are not the cause of firing an output spike are made less likely to contribute. This process is done in TNNs by adjusting the weights of the input synapses according to these rules.

What makes STDP more interesting than classical methods, is that often classical methods are offline

and supervised, while STDP is an online, unsupervised and potentially also a form of reinforcement learning. This makes it so that STDP is more time and energy efficient. STDP requires less time from the TNNs user to train, since the user does not need to cycle through epochs of training. Moreover, STDP is more efficient as it doesn't require the energy needed to run the network and create spikes for the training phase, since it learns as it processes more and more data.

Examples of STDP used in TNNs can be find in the works of Smith [34], [35]. Here Smith uses two variants of STDP applied to different layers in his neural networks. The first network layer in both works use a fully online, unsupervised learning method to update the synaptic weights. Weights are updated according the biological rules of STDP, if a synaptic input helps accumulate potential for an output spike and an output spike is fired, the weight is increased. If an output spike is fired and the synaptic input comes after the output is fired, the synapse's weight is decreased. If there is no synaptic input, but an output is fired, the synapse's weight is decreased. Finally, if there is a synaptic input, but no output is fired, the weight is also increased, but by a different, usually smaller, amount compared to input that accumulates for an output. An overview of this STDP weight updating strategy is given in Table 4.1. Here there are four major combinations of input (x_i) and output (y_j), where spikes are present ($\neq \infty$) and not present ($= \infty$). When spikes are present on both the input and output, there are two subcases. One for when the synaptic input is before the firing of the output, one where the synaptic input is after the firing of the output. The weight is denoted by w_{ij} and μ is the amount of adjustment of the weight. The final case denotes no input or output spike, therefore the weight remains unadjusted.

Input conditions		Weight Update
$x_i \neq \infty$	$x_i \leq y_j$	$\Delta w_{ij} = +\mu_c$
$y_j \neq \infty$	$x_i > y_j$	$\Delta w_{ij} = -\mu_b$
$x_i \neq \infty$ $y_j = \infty$		$\Delta w_{ij} = +\mu_s$
$x_i = \infty$ $y_j \neq \infty$		$\Delta w_{ij} = -\mu_b$
$x_i = \infty$ $y_j = \infty$		$\Delta w_{ij} = 0$

Table 4.1: Overview of STDP weight updating rules in the first layer of Smith's TNN Implementations

The second network layer in Smith's [35] uses reinforcement learning. Before the weights can be updated, a reward signal is broadcast to the layer. If the reward signal is positive, the network's synaptic weights are adjusted according to reward parameters. If the reward signal is negative, synaptic weights are adjusted according to punishment parameters. Again the weights are updated according to the same rules of before the output spike or after the output spike, however an extra condition is introduced. The time between the input and output spike, determines how much the weight is positively or negatively updated. A more comprehensive overview is given in Section 5.1.

4.4. Background and Related Work

As mentioned in Section 4.1, the concept of SNNs with temporal coding was first introduced by Wolfgang Maass in [22]. Here, Maass introduces the "third generation" neurons, building upon the first two generations. The first generation being the perceptron [26], the basis of Multilayer Perceptrons and other ANN types. The second generation being the sigmoid neuron, that use a sigmoid function as activation function to smoothen the output of the neuron, instead of the perceptron's step function. For the third generation, Maass introduces temporal coding, where the timing of spikes is crucial. Unlike previous models that focus on output values, SNNs consider the exact timing of neuron spikes for information processing. Maass argues that spiking neurons are more biologically realistic, modeling the timing and dynamics of actual neuron firing more accurately. Maass' concept captures the idea that neurons in the brain encode and process information not just through rates of firing (as in second-generation models) but additionally through the precise timing of individual spikes. Maass also argues that networks of spiking neurons are computationally more powerful than first and second-generation networks, especially in terms of the number of neurons required to perform certain computations. Maass bases his neuron model on the I&F neuron model, where a neuron fires when its membrane potential, influenced by incoming spikes, reaches a certain threshold. The membrane potential is modeled as the sum of excitatory and inhibitory postsynaptic potentials, which is determined by the timing and weights of incoming spikes. Maass' work is one of the foundational works introducing spiking neural networks with

temporal coding. By emphasizing the role of timing in neural computations, it has opened new avenues for understanding and modeling brain-like computations.

A key part to modeling brain-like computations as neural networks is STDP. Since, the process of strengthening and weakening neurons takes place in biological neurons, it must also be present in the model. Markram et al. [23] demonstrated that in biological neurons the precise timing of postsynaptic spikes relative to synaptic inputs can lead to either strengthening or weakening of synaptic connections. If an accurate model of brain-like computation is to be constructed, this process should be present in the model. [24] has shown that this is achievable. In [24], STDP is applied to neurons in the network to adjust synaptic weights based on the relative timing of spikes, enabling the network to learn and recognize consistent patterns within the input data. Their TNN is built using LIF neurons, the neurons are setup in four layers, with the two middle layers playing a key part in the STDP process. For their experiment, two data sets from the California Institute of Technology were used, one containing images of faces and the other containing images of motorbikes. The network learned to identify visual features from these images using the STDP rule applied at the two middle layers of the network. These layers perform max-pooling operations and visual feature recognition. The learned features were then used for classification tasks, such as distinguishing between faces and non-faces or motorbikes and non-motorbikes.

Another example of TNNs using STDP is in [6], where the learning mechanism is based on a simplified version of STDP. When a neuron spikes, synapses that were activated shortly before the spike are strengthened, while all other synapses are weakened. The potentiation occurs for synapses active within a specific time window. This approach allows neurons to become more sensitive to patterns of spikes that have a consistent temporal correlation. Importantly, this variant of STDP weakens synapses even if they were never activated, ensuring that the network adapts to meaningful temporal patterns rather than just noise. Their network is set up using LIF neurons, structured in two layers, STDP occurs in both layers. Their network is tested with both synthetic and real-world data. For synthetic data, they simulated inputs mimicking a ball moving across a 16x16 pixel grid in 8 possible directions. The input sequences were generated to imitate the output of a dynamic vision sensor. The goal was for neurons to learn parts of these trajectories. After learning, the neurons were found to selectively respond to specific parts of these motion sequences. The network was also tested with real dynamic vision sensor data recording traffic patterns, where the system successfully learned and detected car trajectories with high accuracy.

Smith, who has done much research into TNNs, is also in favor of using STDP for learning in TNNs. His works [37], [34] and [35] all use STDP for online unsupervised learning. In [37], a clustering experiment is set up using a TNN and the MNIST dataset. The network consists out of LIF neurons in a single layer, with inhibition components before and after. The inhibition component before the single layer of neurons, serves as a filter for the input and the inhibition component after the layer makes sure only the earliest output spikes are propagated. The goal is to group similar input patterns using a TNN using STDP for learning. The inputs, consisting of 28x28 pixel grayscale images, were divided into overlapping 5x5 receptive fields (RFs), resulting in 144 receptive fields per image. STDP adjusted synaptic weights based on the timing difference between pre- and post-synaptic spikes. Lateral inhibition was used for reducing the number of output clusters by suppressing all but the earliest spikes.

In [34] a classification experiment is set up using a TNN and the MNIST dataset. The system consists out of Ramp Integrate and Fire (RIF) neurons. These are based off Spike Response Model (SRM) neurons, however these neurons do not leak membrane potential. Instead, the response function is a ramp and the weight of the synaptic input determines how much membrane potential accumulates over time. The system consists out of two components, the actual TNN and a decoding network. The goal of the experiment is for the TNN to cluster the input, clustering the same handwritten numbers. The decode network then decodes the clusters into actual numbers, labelling each cluster. The synaptic weights in the TNN are updated using STDP, according to the rules in Table 4.1. The decode network uses a form of supervised learning to perform weight updating, using a reward signal which is similar to reinforcement learning.

Finally, in [35] a TNN is implemented as controller for a classical control experiment, the cart-pole problem. The system consists out of Ramp Integrate and Fire (RIF) neurons and again it consists out of two component, a clustering TNN (C-TNN) and a reinforcement learning TNN (R-TNN). As mentioned

in Section 4.3, both networks utilize STDP to update their synaptic weights, where the C-TNN updates according to the rules in Table 4.1 and the R-TNN uses a special form of STDP with a reward signal. The goal of the experiment is to balance a two-dimensional pole that is attached by a rotation point to a cart. The cart can be pushed left or right and by doing so the angle of the pole changes. This experiment is especially interesting, since it shows that TNNs can not only be used for visual tasks of clustering or classification, but for other computational tasks as well.

All these implementations of TNNs provide a relatively simple model that models the biological brain as a neural network. However, the question should then be asked, is it possible that a model for the biological brain can be this simple? In [37], argues that it is possible, by taking results from neuroscience that support the TNN model. In Chapter 4 of [37], parallels are drawn between the TNN model and neuroscientific research. The next section provides a summary of the key components of Chapter 4 from [37], as to provide support for the interest in TNN research.

4.5. Connecting Temporal Neural Networks with Biology

Chapter 4 of [37] presents multiple arguments supporting the TNN model as model of the biological brain. These arguments are communication via voltage spikes, columns and spike bundles, spike synchronization, first spikes carry information, feedforward processing and plasticity and training. With a counter argument in fault tolerance and temporal stability. Each of the following paragraphs discusses a single argument mentioned.

Communication via voltage spikes: In brains, the fundamental mode of communication is through voltage spikes, or action potentials. TNNs also utilize voltage spikes for communication, mirroring this biological mechanism. This aligns with the ”neuron doctrine” [46], which defines neurons as the basic functional units of the brain, receiving and transmitting information via dendrites and axons respectively.

Columns and spike bundles: Brains process information using columns of neurons that communicate through spike bundles. In TNNs, neurons are grouped into computational units called layers, layers can be seen as analogous to cortical columns. Layers operate collectively to process information, similar to the way columns in the neocortex do.

Spike synchronization: Neurons often synchronize their spikes to process information efficiently, which can occur both periodically and aperiodically. TNNs incorporate mechanisms for spike synchronization, reflecting the temporal coherence observed in biological systems. This synchronization is essential for coherent processing and is modeled to mimic biological oscillations like theta and gamma cycles. Analogous to the theta and gamma cycles in biological neural networks, TNN neurons are “reset”, where all membrane potentials are returned to zero, after producing an output or until a time limit is reached, this is the TNN’s form of a gamma cycle.

First spikes carry information: In brains, the initial spikes in a neural response are crucial for conveying information. TNNs emphasize the importance of the timing of the first spike in transmitting data, similar to the way early spikes are considered vital in biological neurons for encoding information. For instance, the I&F model only accumulates potential from the first spike on a synaptic input.

Feedforward processing: A significant amount of neural processing in the biological brain occurs in a feedforward manner, particularly for rapid sensory processing. TNNs are designed to perform processing in a feedforward fashion, facilitating quick and efficient information flow. This is inspired by the feedforward pathways seen in sensory processing in the neocortex. All works cited in previous sections of this chapter all utilize feedforward processing.

Plasticity and training: Neural plasticity is a cornerstone of learning and adaptation in the brain, involving changes in synaptic weights, axon diameters, and network connectivity. TNNs incorporate plasticity by allowing adjustments in neuron parameters, synaptic weights, and network configurations during training in an online fashion. This mirrors the plastic nature of biological neural networks that adapt based on experience and learning. A prime example of this is STDP, where synaptic weights are updated in an online unsupervised manner.

Fault tolerance and temporal stability: The brain exhibits robust fault tolerance and temporal stability, ensuring reliable function despite potential disruptions. While TNNs strive to replicate this robustness,

the TNN model simplifies certain aspects to focus on core computational paradigms. Fault tolerance and stability are considered, however the primary focus remains on the computational efficiency and accuracy of spike-timing-based processing.

4.6. Validation of Temporal Neural Networks

For this thesis, the goal is to validate the concept of TNNs by reproducing research done on the topic. Two options that are within the scope of this thesis are [34] and [35]. Both works have been published recently, they both have a clearly laid out experiment set up and have results that can be easily compared.

This thesis covers online reinforcement learning using TNNs from [35]. This work is more recent compared to the other and is a unique implementation of TNNs compared to the works discussed in Section 4.4. To validate the concept of online reinforcement learning using TNNs, a simulator is built to execute the experiments in. This simulator is built in such a way that all components are modular. If future researchers would like to validate more TNN concepts, then the simulator can be expanded upon and used for their research.

5 Key Subjects and Methodology

This chapter explains the key subjects of this thesis, it describes how online reinforcement learning is implemented using Temporal Neural Networks, it explains the cart-pole problem and it discusses the choices made to implement the simulator and the experiment environment. Section 5.1 explains the concept of online reinforcement learning using Temporal Neural networks. It describes and explains each component of the whole system. Section 5.2 lays out the cart-pole problem used by Smith in [35], what the classical control problem entails and explains how a Temporal Neural Network system can be used as a controller for the control problem. Section 5.3 discusses the goal of thesis, to create a simulator for the Temporal Neural Network system and environment around the network and it discusses design decisions for the simulator.

5.1. Online Reinforcement Learning using Temporal Neural Networks

In [35], online reinforcement learning is used to train system to act as a controller for the cart-pole problem. Here, online reinforcement learning is implemented using TNNs by combining STDP with a reward signal. Before this can be explained, the system using the TNNs needs to be explained. The core components that make up the system, the composition of the networks, how the system is used as controller and how spikes are encoded and decoded. Afterwards, the learning methods can be discussed. The following paragraphs of this section will first explain the core components, system composition, usage as controller and encoding and decoding. The last two paragraphs explain the learning methods used for both TNNs in the system.

The neurons of the system are Ramp Integrate and Fire (RIF) neurons. As mentioned in Section 4.4, these are based off Spike Response Model (SRM) neurons, with the caveat that neurons do not leak membrane potential. Instead, the response function is a ramp and the weight of the synaptic input determines how much membrane potential accumulates over time. An example of RIF neurons with synaptic inputs and membrane potential accumulation is depicted in Figure 5.1. Smith uses the term body potential for membrane potential, therefore for the rest of the report the same term as Smith is used. In Figure 5.1, there are three input spikes on inputs x_1, x_2 and x_4, x_3 has no input spike which is denoted by the ∞. A combination of spike inputs is called a spike volley. The number next to the spike represents the time of the spike. The first spike in the volley is always given time 0, the following spikes are then timed relative to the first. Smith uses the notation $[1, 0, \infty, 3]$ to denote this spike volley, with each index representing the number of the input (x_i).

The input spikes arrive at the synapse, where the synaptic weight determines how much body potential should be accumulated according to RIF rules. The amount of potential for a given weight can be seen in Figure 5.2. The TNNs run on time cycles, with $t = 0$ being the time of the first spike. After the activation of the response function, each cycle accumulates potential, the upper limit of potential accumulation is determined by the weight of the synapse. The offset of one time unit for weights 4 and lower accounts for a rise in slower body potential for synapses with lower weights. According to [35], this threshold for faster or slower body potential rising is always at half of w_{max}. The potential

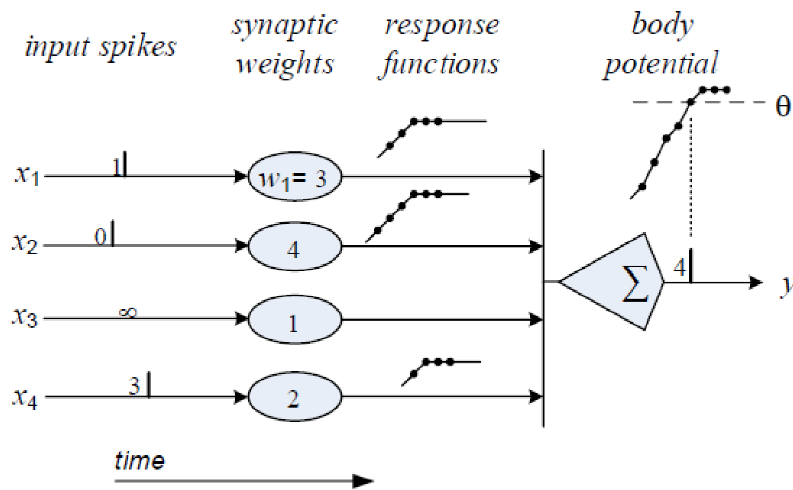


Figure 5.1: Example given by Smith in [35] for RIF neurons.

from each synapse is summed and if the sum of potentials is higher than the threshold potential (θ), an output spike is fired. In Figure 5.1, the output spike is fired at $t = 4$, since at $t = 4$ the sum of potentials crosses θ. Here is the step-by-step process:

1. In the example $\theta = 6$.
2. At $t = 0$ the first spike arrives at the synapse on x_2, due to all weights being equal to or lower than 4, no potential is accumulated at arrival of the spike.
3. At $t = 1$ the second spike arrives at the synapse on x_1. From the spike on x_2 potential is accumulated. The total body potential is equal to 1.
4. At $t = 2$ no spikes arrive. From the spikes on x_1 and x_2 potential is accumulated. The total body potential is equal to 3.
5. At $t = 3$ the third spike arrives at the synapse on x_4. From the spikes on x_1 and x_2 potential is accumulated. The total body potential is equal to 5.
6. At $t = 4$ no spikes arrive. From the spikes on x_1, x_2 and x_4 potential is accumulated. The total body potential is equal to 8, $8 > \theta$ thus an output spike is fired.
7. At $t = 5$ no spikes arrive. From the spike on x_4 potential is accumulated. The total body potential is equal to 9. Since an output spike has already been fired, this can no longer happen again until the network is reset.

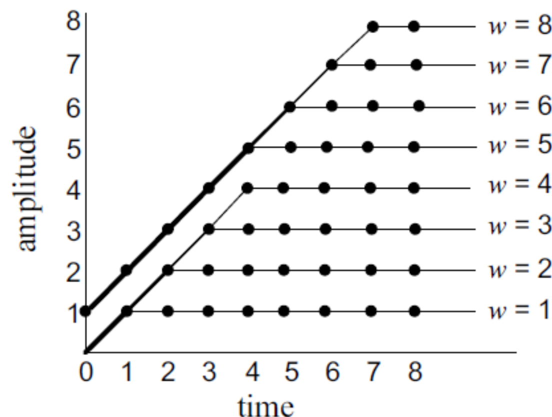


Figure 5.2: Ramp Integrate and Fire response function for weights 1 through 8 [35].

The first part of the system is the Clustering Temporal Neural Network (C-TNN). An example of a C-TNN is given in Figure 5.3. In the example in Figure 5.3, the maximum weight (w_{max}) is 4 and $\theta = 6$. Synapses with no weight are not depicted in the image. Due to $w_{max} = 4$, weights of 2 and lower are shifted down by 1 amplitude in the RIF response function. In Figure 5.3, it can be seen that each neuron has synapses with each input, each synapse having their own weight. A volley of spikes all on $t = 0$ is input on synaptic inputs x_2, x_3 and x_4. Three neurons can be seen accumulating body potential, with one of the neurons having one synapse with $w = 4$ more than the other two. For this neuron, its body potential at $t = 0$ is 3 and at $t = 1$ is 6, $6 > \theta$ therefore it fires an output spike at $t = 1$. For the other two neurons, the body potential at $t = 0$ is 2, at $t = 1$ is 4 and at $t = 2$ is 6, therefore these neurons fire at $t = 2$. After the neurons, a WTA block can be seen. This block is a 1-Winner Takes All (1-WTA) block, where the first spike is the “winner” and is propagated. Spikes later are held back. If two output spikes are fired at the same time, the winner is chosen by the higher body potential neuron. If the body potentials of the firing neurons are equal, the neuron with the lower index wins.

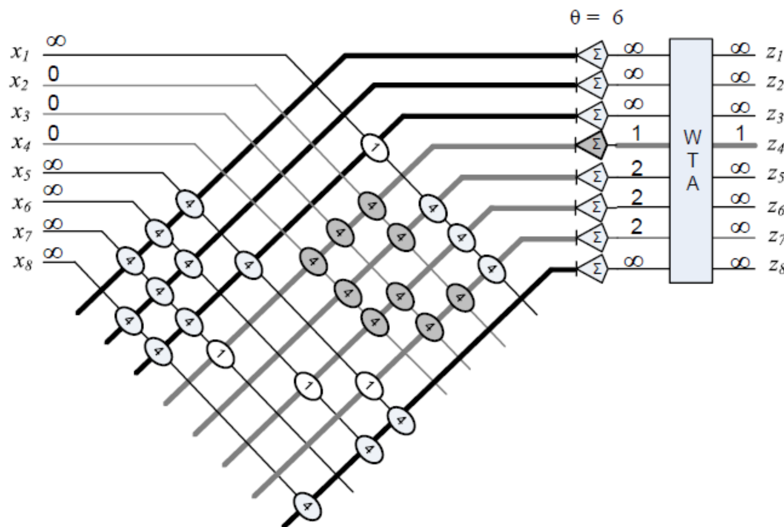


Figure 5.3: An example of a Clustering Temporal Neural Network from [35].

The second part of the system is the Reinforcement Learning Temporal Neural Network (R-TNN). An example of an R-TNN is given in Figure 5.4. In the example in Figure 5.4, $w_{max} = 3$ and $\theta = 2$. Due to $w_{max} = 3$, weights of 1 are shifted down by 1 amplitude in the RIF response function. In Figure 5.4, a single spike is input on synaptic input x_2. In the implementation of the R-TNN in [35], a spike volley can only have a singular spike. Therefore, there will be at most one active synapse per neuron, which makes accumulation of potential is not necessary. In Figure 5.4, synaptic input x_2 has its w_{max} synapse with the fourth neuron, therefore an output spike is fired by the fourth neuron. After the neurons, a WTA block can be seen, which is a 1-WTA block. The same rules apply as for the C-TNN, if two output spikes are fired at the same time, the winner is chosen by the higher body potential neuron. If the body potentials of the firing neurons are equal, the neuron with the lower index wins.

The system is used as a controller for the cart-pole problem. This is made possible by letting the system identify the state of the pole and the car. Then, based on state, the system produces an output which says to push the cart left or right. The direction the cart is pushed in should help balance the pole on the cart. By using STDP, the system should learn to make clusters for different states of the cart and pole in the C-TNN. Based on these clusters, the R-TNN learns which direction to push the cart in to balance the pole. A more detailed explanation of the cart-pole problem can be found in Section 5.2.

The system only processes spikes, therefore the state of the system has to be encoded as spikes for the system to process and the output spike of the system has to be decoded to a direction to push the cart in. To explain encoding, an example will be used where the angle of the pole is encoded as spikes. For example in the cart-pole problem, the pole angle ranges from -12 to $+12$ degrees. These pole angles can be divided into 12 equal intervals, with the end points $[-12, -10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10, 12]$. Then, for example, the angle 3.14 would map to interval $[2, 4]$. Note that angles such as 2.5, 3.96 and

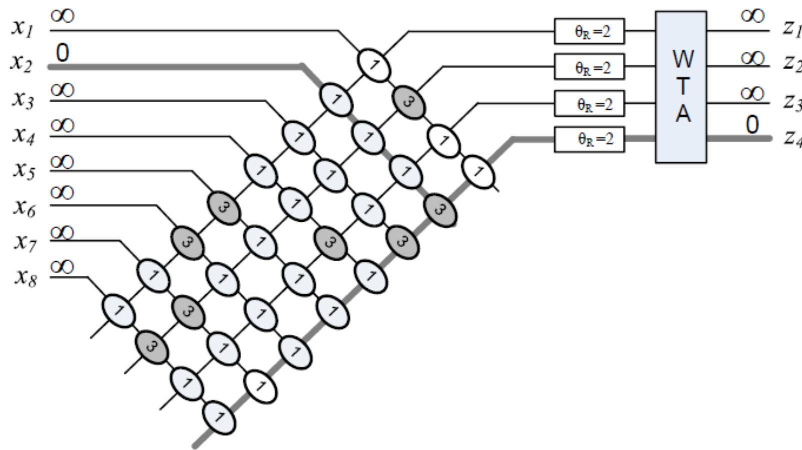


Figure 5.4: An example of a Reinforcement Learning Temporal Neural Network from [35].

2.65 would all map to the same interval of $[2, 4]$. This means some clustering is already done before anything is input to the network. Note that intervals do not have to be equal, an example of this is pole angles divided into 6 ranges defined by the end points $[-12, -6, -1, 0, 1, 6, 12]$. Intervals can be encoded for the system by creating synaptic inputs equal to the number of intervals. A spike is fired on the synaptic input, that encodes the interval the angle of the pole maps to. For example, take the 12 equal intervals from -12 to +12 degrees, each input represents one interval, so x_1 is $[-12, 10]$, x_2 is $[-10, -8]$, this goes on to x_{12} which is $[10, 12]$. For instance, if the angle of the pole is 3.14, this angle maps to $[2, 4]$ and a spike is fired on x_8, the synaptic input representing $[2, 4]$. This is the encoding of the angle is one state variable, the next step is to encode multiple state variables. When encoding multiple state variables, each state variable gets their own group of synaptic inputs. For example, take the 12 equal intervals again, now add the displacement of the cart as the second state variable. In the cart-pole problem the cart displacement ranges from -2.4 to 2.4 meters, these can be divided into 3 intervals with the endpoints $[-2.4, -1, 1, 2.4]$. Then, synaptic inputs x_1 to x_{12} represent the intervals of the pole angle and synaptic inputs x_{13} to x_{15} represent the intervals of the cart displacement. Encoding intervals as single spikes is called a 1-hot code, it is also possible to encode an interval as multiple spikes, which is called an m-hot code. In Figure 5.5, examples of a 1-hot code and m-hot codes are given. In the example of Figure 5.5, there are three state variables present. In Figure 5.5a, a 1-hot code can be seen with spikes on input 2 of the first state variable, input 3 of the second state variable and input 8 of the third state variable. In Figure 5.5b, the 3-hot code equivalent can be seen, where two input spikes are added on the two inputs below. Furthermore, for each set of state variable synaptic inputs, two inputs are added. Figure 5.5c gives an example of a different 3-hot code with different input spikes. For clustering, a 1-hot code is deficient, due to clustering methods relying on input similarity to make clusters. For 1-hot coded spikes, inputs have no similarity, since each encoding has its own synaptic input. By using an m-hot code, where m is always odd, this deficiency can be resolved. Compare, for example, Figure 5.5b and Figure 5.5c, here the inputs on the second state variable have a difference of 1, which means two spikes in common. The inputs on the third state variable have a difference of 2, which means one spike in common. This adds a similarity metric to the input, values that have two spikes in common are similar, values that have one spike in common are less similar, but still similar and values that have no spikes in common are dissimilar.

For weight updating in the C-TNN, the most fundamental form of STDP is utilized. Weights are updated according to the rules in Table 4.1. However, this table is missing descriptions of each action, therefore an expanded version can be found in Table 5.1. A visual of the parameters used to update synaptic weights can be found in Figure 5.6. In Figure 5.6, adjustment of weight w_{ij} in the synapse is determined by the timing of spikes on synaptic input x_i and output z_j. If the input spike precedes the output spike or arrives at the same time as the output spike, then the weight is increased by the value μ_c, up to maximum weight w_{max}. Otherwise, the weight is decreased by μ_b down to the minimum of 0. The next three rows in Table 5.1 cover the remaining cases. If there is an input spike, but no output spike, the weight is increased by μ_s, this is usually a small value (smaller than μ_c) or 0. If there is an output spike,

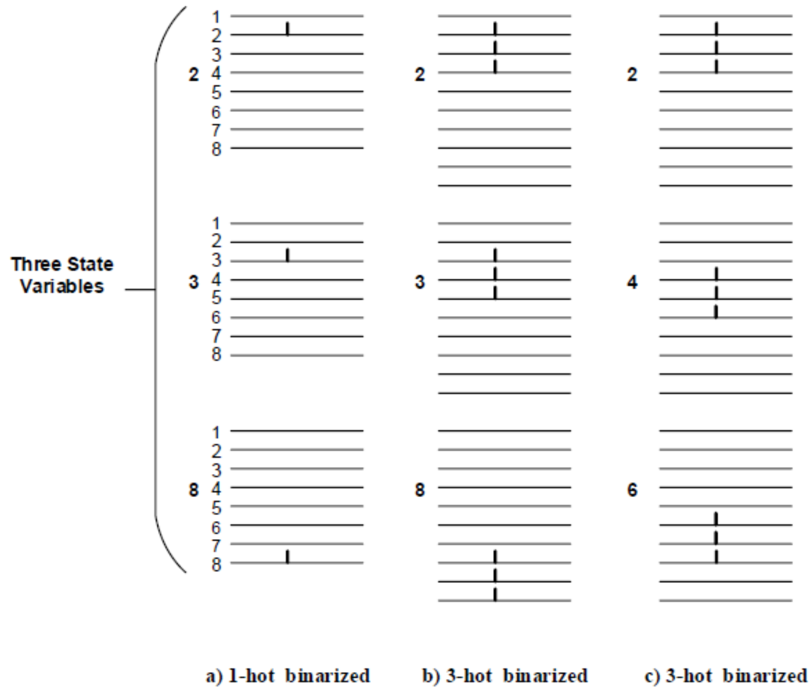


Figure 5.5: An example of 1-hot code and m-hot code from [35].

but no input spike, the weight is decreased by μ_b. If there are no input and output spikes the weight is not updated. The description *capture* is used, because if there is an input that lets a neuron fire an output on z_j. The weights on synapses connected to z_j that receive an input spike are increased and others are decreased. The input pattern has been *captured* by z_j, and z_j starts to form a cluster with this pattern as its initial centroid. The description *back-off* is the opposite of *capture*, the synapses that have no contribution to producing an output are told to *back-off*. The description *search* is used for synapses that should *search* for new clusters to form, because they are receiving input spikes, but no output spike is fired. The description *no-op* is used for when *no operation* happens on the weight. If $\mu_s > 0$, then *search* mode is enabled and new clusters may be *captured* if there is a large shift in input patterns. If in a cluster its members do not trigger the firing of an output spike, *search* enables other synapses outside of the cluster to gradually increase in weight. Eventually, the synapses outside of the cluster trigger the firing of an output spike and a new cluster is *captured*.

Input conditions		Weight Update	Description
$x_i \neq \infty$	$x_i \leq z_j$	$\Delta w_{ij} = +\mu_c$	*capture*
$z_j \neq \infty$	$x_i > z_j$	$\Delta w_{ij} = -\mu_b$	*backoff*
$x_i \neq \infty$ $z_j = \infty$		$\Delta w_{ij} = +\mu_s$	*search*
$x_i = \infty$ $z_j \neq \infty$		$\Delta w_{ij} = -\mu_b$	*backoff*
$x_i = \infty$ $z_j = \infty$		$\Delta w_{ij} = 0$	*no-op*

Table 5.1: Overview of STDP weight updating rules in the C-TNN of Smith's TNN Implementations

In Figure 5.7, an example of capturing clusters in the C-TNN using STDP is given. Here the red coloring indicates spikes fired on the marked synaptic inputs, the yellow coloring indicates the synapses that accumulate potential, the green coloring indicates the neuron that fires an output spike and the step is denoted by s. In this example, $w_{init} = 3$, $w_{max} = 4$, $\theta = 2$, $\mu_c = 1$ and $\mu_b = 1$. At $s = 0$, spikes are fired on synaptic inputs x_0 and x_1. The synapses connected to these inputs start accumulating potential and θ of z_0 and z_1 is crossed at the same time. WTA inhibition chooses the lower index neuron as winner, therefore z_0 fires an output spike. STDP captures this input pattern on the synapses connected to z_0, x_0 and x_1 by increasing the weight and the synapse connected to is z_0 and x_2 is told to back off by decreasing the weight. The adjusted weights can be seen at $s = 1$. At $s = 1$, spikes are fired on

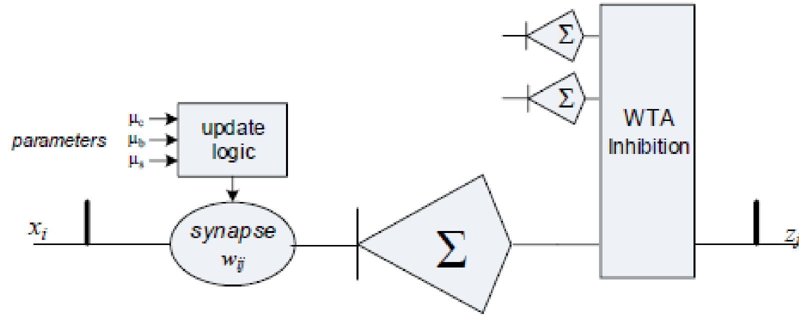


Figure 5.6: Visual overview of weight updating according to STDP in the C-TNN from [35].

synaptic inputs x_1 and x_2. The synapses connected to these inputs start accumulating potential. Due to the lower weight on synapse z_0x_2 the body potential at $s = 1, t = 0$ for z_0 is 1. Since synapse z_0x_2 has a weight lesser than or equal to $\frac{w_{max}}{2}$, it does not accumulate potential at $t = 0$. The synapses of z_1, however, do accumulate enough potential to cross θ at $s = 1, t = 0$, therefore z_1 fires an output spike. STDP captures this input pattern on the synapses connected to z_1, x_1 and x_2 by increasing the weight and the synapse connected to is z_1 and x_0 is told to back off by decreasing the weight. The adjusted weights can be seen at $s = 2$. After repeating this process for two more steps, it can be seen that the synapses not activated to accumulate potential start to back off and that input patterns are captured in clusters by the synapses that are activated.

s = 0

	z0	z1
x0	3	3
x1	3	3
x2	3	3

s = 1

	z0	z1
x0	4	3
x1	4	3
x2	2	3

s = 2

	z0	z1
x0	4	2
x1	4	4
x2	2	4

s = 3

	z0	z1
x0	4	2
x1	4	4
x2	1	4

s = 4

	z0	z1
x0	4	1
x1	4	4
x2	1	4

Figure 5.7: An example of how clusters are captured in the C-TNN using STDP.

For weight updating in the R-TNN, three-factor learning rules are employed, where the first two factors are the same as STDP, the input spike time and output spike time, and the third factor is a reward signal broadcast to the network from the environment. A visual of the parameters used to update synaptic weights can be found in Figure 5.8. An explanation of each STDP parameter now follows, note that a "step" is one cycle of inputting spikes to the system and letting it produce an output. Weights in both networks of the system get updated after a step. When using the system in simulations, a simulation consists out of multiple steps or cycles. A fundamental part of the weight update process is a time window (ω), the window is a certain amount of steps. Weight w_{ij} is only updated if x_i has a spike on a step within the window.

Variables:

- w_{ij}: the weight of the synapse, this gets adjusted up and down. The minimum weight is 0 and maximum is w_{max}. Weights are initialized at a default value w_{init}.
- c_{ij}: a counter that counts up at each step until it saturates at ω_ρ or ω_π. The counter is reset to 0 if there is a spike on x_i.
- e_{ij}: a binary flag or boolean that is set to *true* if input x_i and output z_j spike on the same step. It is set to *false* if z_j does not spike on a step that x_i spikes.
- R: the reward signal, +1 if there is a reward, -1 if there is a punishment, 0 if there is none. R-TNN weights are not updated when $R = 0$.

Reward (ρ) Parameters:

- ω_ρ: update window size.
- ρ_0^+: maximum weight increase amount.

- ρ_0^-: maximum weight decrease amount.

Punishment (π) Parameters:

- ω_π: update window size.
- π_0^+: maximum weight increase amount.
- π_0^-: maximum weight decrease amount.

The weight increase and decrease amount decay over the length of the window. This means a weight is increases or decreased by the maximum amount when $c_{ij} = 0$. Counter c_{ij} increases linearly, this gives update values for w_{ij}:

- $\rho_{ij}^+ = \rho_0^+ - \frac{\rho_0^+}{\omega_\rho} c_{ij}$
- $\rho_{ij}^- = \rho_0^- - \frac{\rho_0^-}{\omega_\rho} c_{ij}$
- $\pi_{ij}^+ = \pi_0^+ - \frac{\pi_0^+}{\omega_\pi} c_{ij}$
- $\pi_{ij}^- = \pi_0^- - \frac{\pi_0^-}{\omega_\pi} c_{ij}$

Due to the neurons in the R-TNN outputting a 1-hot code, at most one neuron spikes. When a reward is a non-zero reward is received during a step, all synaptic weights are updated. For any weight w_{ij}, if $c_{ij} < \omega$ it is updated according to the following rules:

- If $R = 1$:
  - If $e_{ij} = 1$, then $\Delta w_{ij} = \rho_{ij}^+$.
  - If $e_{ij} = 0$, then $\Delta w_{ij} = \rho_{ij}^-$
- If $R = -1$:
  - If $e_{ij} = 0$, then $\Delta w_{ij} = \pi_{ij}^+$.
  - If $e_{ij} = 1$, then $\Delta w_{ij} = \pi_{ij}^-$

Figure 5.8: Visual overview of weight updating according to three factor learning rules in the T-TNN from [35].

5.2. Cart-Pole Problem

As in [35], the TNN system is used as a control system for the cart-pole problem. The cart-pole problem is a classical control problem with a system which is highly nonlinear, unstable and dynamic. The goal is to balance a two-dimensional inverted pendulum attached with a hinge (the pole) to a cart that can be pushed left or right. A visual overview of the cart-pole problem and its parameters can be found in Figure 5.9. The list of parameters with descriptions and values:

- Mass of the cart, $M = 0.711$kg
- Mass of the pole, $m = 0.209$kg

- Gravity, $g = 9.8$m/sec^2
- Force on the cart, $F = \pm 10$N
- Length of the pole, $l = 0.326$m
- Time step, time of one cycle/step, $\tau = 0.02$ sec. intervals

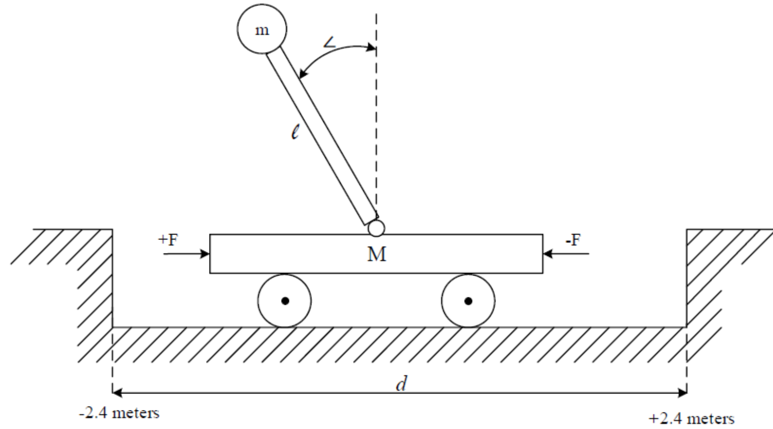


Figure 5.9: Visual representation of the Cart-Pole problem [35]

The angle of the pole ($\angle$) is restricted to a range of -12 to $+12$ degrees. The displacement (d) of the cart is restricted to a range of -2.4 to 2.4 meters. Both $\angle$ and d can be calculated, but calculating their second derivatives and then integrating. The equations for the second derivatives of $\angle$ and d can be found in Equation 5.1 and Equation 5.2 respectively. These equations are taken from [11]. In [11], the equations from [3] are improved upon due to mistakes. To quote Florian: “The control of a cart-pole system is widely used as a benchmark problem for testing the efficiency of reinforcement learning algorithms. It seems to have been first used as a test problem in adaptive control by [29], [28] and became a more famous problem since its use in the paper of [3]. Google Scholar reports about 500 papers citing this paper, and about 100 papers containing the words ‘cart pole’ or ‘cartpole’. There are, however, two mistakes in the equations from [3] that describe the dynamics of the cart pole. One mistake introduces a difference between the reported equations and the equations describing a correct physical model, and the other mistake is probably a typo.” [11]. The equations used in [35] are taken from [31], the equations in [31] are based off [29] and [3]. Therefore, the equations from [11] are used to ensure correctness.

$$\ddot{\angle} = \frac{g \sin\angle - \cos\angle \left(\frac{-F - ml\dot{\angle}^2 \sin\angle}{M+m} \right)}{l\left(\frac{4}{3} - \frac{m\cos^2\angle}{M+m}\right)} \tag{5.1}$$

$$\ddot{d} = \frac{F + ml\left(\dot{\angle}^2 \sin\angle - \ddot{\angle}\sin\angle\right)}{M+m} \tag{5.2}$$

After both $\ddot{\angle}$ and $\ddot{d}$ are calculated, integration is done as follows:

$\dot{d} = \dot{d} + \tau\ddot{d}$
$d = d + \tau\dot{d}$
$\dot{\angle} = \dot{\angle} + \tau\ddot{\angle}$
$\angle = \angle + \tau\dot{\angle}$

For each step of the system and environment, integration is done in the order as listed above. Every time step, the system receives the state of the environment as input. The system can take one or

multiple state variables, these are $\angle$, d and their first derivatives. The first version of the system only take $\angle$ as a state variable encoded as spikes, later versions take multiple state variables, first $\dot{d}$ is added as second state variable, then d as third [35]. For a trained system, after processing the input, the system checks in the C-TNN which cluster this input belongs. An output spike is generated from the neuron representing this cluster and this spike is propagated to the R-TNN. The R-TNN produces an output based on the cluster from the C-TNN. This output is decoded and determines whether the cart should be pushed left ($-F$) or right ($+F$) for the state of the system at that time step. The environment takes this output, the second derivatives are calculated with the direction to push the cart in according to Equation 5.1 and Equation 5.2. Then, $\angle$ and d are updated according to the integrations listed above. After updating the environment, the system and environment move forward to the next time step and the process is repeated. An overview of the whole system can be found in Figure 5.10.

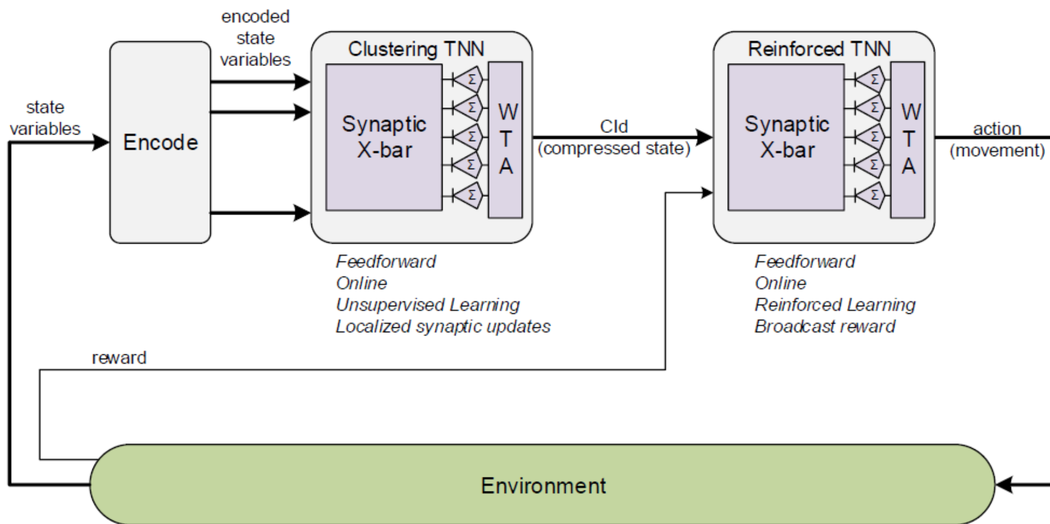


Figure 5.10: Overview of the whole system with TNN and cart-pole environment from [35]

A simulation of the TNN system and cart-pole problem entails running multiple episodes of the system and the cart-pole problem. A simulation episode ends when the environment goes out of bounds, $\angle < -12$, $\angle > 12$, $d < -2.4$ or $d > 2.4$, or the episode reaches the maximum number of steps, in [35] the maximum number of steps is $10,000$ (equal to 200 seconds of simulated time). Each simulation episode starts with a randomly generated starting angle of the pole and the cart at $d = 0$. The randomly generated angle is bound to -2 and $+2$ degrees. The random number generator for the angle uses a seed to make the research reproducible. The primary results metric to gather from simulations is the number of successful steps in each episode, this means how long the environment stays within bounds. First the system warmed up using a number of training episodes, then the simulation continues for a number of test episodes. Weight updating continues during the test phase, as this is an online reinforcement learning system. In [35], Smith has found that the specific seed has a significant effect on the results because different seeds lead to different sequences of angles. This means that for each seed the training and test phase is different. This can result in better results for a certain seed and worse results for another. Therefore, results are gathered for multiple seeds and the average number of successful steps for each seed is given.

Note that within the TNN system, time units are used for the temporal aspect. These are different from the time steps of the complete system with the environment. Time steps are defined by τ, such as in Section 5.2.

5.3. Implementation of Simulator and Experiment Environment

The goal of this thesis is to create a compiled code event driven simulator for TNN systems. This simulator is coded in C++, since it is typically known to be faster and more efficient than MATLAB. Smith has done his simulations in MATLAB, which is over all well known for its computational capabilities,

however it is not as efficient as other programming languages. Therefore, for this thesis, the simulator of the TNN system and environment is programmed in a different, more efficient language. The simulator is built using C++, which is a programming language with highly optimizable code. The decision to go with C++, over C or Rust, is that C++ has more libraries than C for modern functions, such as mathematical libraries and libraries for dynamic storage objects. This makes C++ more flexible to program in. When coding in Rust, the developer has to comply with strict rules and constraints in order for the code to compile, this is to prevent issues such as memory bugs or segmentation faults. This adds complexity during the coding process. In C++, these issues are the responsibility of the programmer. For the scope of this thesis, C++ is a better fit, since the the lack of rules and constraints compared to Rust helps write code more quickly. The codebase for the simulator is built from the ground up and the code is designed in a way that all simulator components are as modular as possible. If future researchers are interested in the topic of TNNs, the simulator can be expanded upon to fit their research.

6
Design and Implementation

This chapter goes over the design and implementation of the Temporal Neural Network system and environment. It explains the setup of both components and how each component is implemented, it explains in depth how the Temporal Neural Networks are built in C++, it explains in depth how the elements around the Temporal Neural Network system are implemented, it describes how to setup the environment and network, and how to run simulations with the simulator. Section 6.1 discusses the setup of the different simulator components. Section 6.2 explains the implementation of the TNN system of this thesis' simulator. Section 6.3 presents the implementation of the full simulator, it describes the implementation of the environment, it goes over each step before the simulation and it explains each step during a simulation.

6.1. Setup of System and Environment

The codebase of the simulator is divided into two folders, one for the actual simulator with all the code relevant to the TNNs and running the simulation and one for the environment around the TNN system. Outside of the folders there are four configuration files. One for the setup of the network, one for the mapping of connections between synaptic inputs and the first layer (C-TNN) and between the two layers (C-TNN and R-TNN), one for the STDP parameters and one to read weight from for each synapse, if the user specifies to read weights from this configuration file. In the environment folder, there is an option to dump the results of a specific simulation episode, these are then dumped to a .csv file, which after running the simulation can be found in the root folder.

6.2. Implementation of Temporal Neural Networks

This section describes the implementation of the Temporal Neural Networks of the complete system. Subsection 6.2.1 lays out how spikes are implemented in C++. Subsection 6.2.2 explains how TNN neurons are built using C++. Subsection 6.2.3 discusses the layers that make up the structure of the network, Subsection 6.2.4 describes the configuration of the network, Subsection 6.2.5 explains the encoding and decoding between system and environment and Subsection 6.2.6 discusses the implementation of STDP and weight updating in the simulator.

6.2.1. Spikes

Before the implementation of the network and its components can be discussed, an explanation of the implementation of spikes is needed. In this simulator's implementation of TNNs, spikes are implemented as booleans. If there is a spike on synaptic input x_i at time t, then in the C++ simulator the boolean for input x_i is set to *true* at time t. If time is not t, then the boolean of x_i is *false*. Spike integrity is upheld by setting the booleans at the start of a unit of time to *true* and at the end of unit of time to *false*. For example, if input x_0 fires a spike at $t = 1$, then at $t = 0$, x_0 remains false, at the start of $t = 1$, x_0 is set to true, at the end of $t = 1$, x_0 is set to false and the rest of the time units x_0 remains false.

6.2.2. Structure of Neurons

In this simulator's implementation, neurons have a set of synapses, a threshold potential, the RIF response function and an output boolean. Synapses have a pointer to the boolean of a synaptic input, a weight and functions to read and update the weight. Each synapse of the neuron is connected to a synaptic input of the TNN system through a pointer. If a synaptic input of the system spikes, the boolean is set to *true*, the synapse receives the spike due to the synapse using a pointer. If the neuron checks whether potential should be accumulated from its synapses, it checks whether the boolean the pointer points to is true. A function in the neuron checks whether there are spikes on its synapses. If there is a spike at time t and $w_{ij} > \frac{w_{max}}{2}$, then potential is accumulated from that synapse at time t. If $w_{ij} \leq \frac{w_{max}}{2}$, then potential is accumulated from that synapse at $t + 1$. Potential is accumulated according to the RIF response function as depicted in Figure 5.2. Each unit of time a function checks whether the sum of potentials from the neuron's synapses crosses the threshold. If the threshold is crossed at time t, then an output spike is generated at time t, so the output boolean is set to *true* at time t. After all spikes have been processed during a time unit, all neuron outputs are set to *false* to maintain spike integrity over time.

6.2.3. Structure of Network

The TNN system has two layers, the C-TNN and the R-TNN. In the simulator, layers consist out of the neurons in the layer, a flag that denotes if it is a C-TNN or R-TNN, functions to do WTA inhibition and it contains all STDP parameters of that that layer. This section explains all parts mentioned except for the STDP parameters, that is reserved for Subsection 6.2.6. The STDP parameters are added to the layer instead of individual neurons, due to layers being implemented later than neurons. Neuron structure was already established and difficult to edit when layers were implemented.

The functions in neurons that check for spikes, potential accumulation and output spike generation are all called from the layer. The layer first checks all neurons for spikes, then for potential accumulation and then for output spikes. These checks are done one by one on each neuron in the order of the indices. These functions are first called for the first layer and then for the second layer, since the first layer processes spikes and has to fire an output for the second layer to process. Neurons in the second layer have their synapses point to the output booleans of the first layer.

WTA inhibition in the layer is 1-WTA inhibition. Every time an output spike is fired from a layer, a flag gets check to see if a spike has previously been output from this layer. This flag is the WTA flag and gets set when the first output spike is fired from a layer. If any neurons in the layer fire output spikes after the flag is set, the spikes are not propagated. If two neurons in the same layer fire output spikes at the same time, the logic from Section 5.1 is used, where the higher body potential neuron is the winner and if both have equal body potentials the lower index wins.

The layer contains a flag to denote whether it is a C-TNN or an R-TNN. When calling STDP weight update functions to update weights for the neurons and their synapses in that layer. The flag is checked to see which weight update rules and parameters should be used. This is also a reason for why the STDP parameters are on the layer level. As mentioned in the first paragraph of this subsection, further explanation of the STDP mechanism can be found in Subsection 6.2.6.

6.2.4. Network Configuration

The configuration of the TNNs is read from four configuration files. The first configuration file is the network configuration. It contains the number of layers in the network, the number of neurons per layer, the number of synaptic inputs of the TNN system, the threshold potential of the neurons per layer, whether default weights should be read from another configuration file and all encoding parameters. The encoding parameters are the number of state or environment variables, m for the m-hot code and the intervals of the state variables.

The second configuration file is the configuration of the mapping between the synaptic inputs and the first layer (C-TNN), and the mapping between the two layers (C-TNN and R-TNN). Here can be specified whether each connection should be specified individually by hand or whether the network is fully connected and that each connection is automatically generated. For the mapping between the synaptic inputs and first layer, the number of synaptic inputs, the number of neurons in the first layer and the default weight of each synapse must be specified. For the mapping between the two layers,

the number of neurons in the first layer, the number of neurons in the second layer and the default weight of each synapse must be specified. If the fully connected option is enabled, then the simulator automatically makes connections according to these parameters.

The third configuration file is the configuration of the STDP weight update parameters. It contains the update values for both the C-TNN and the R-TNN. For the C-TNN, there is *capture* (μ_c), *backoff* (μ_b), *search* (μ_s) and the maximum value for weights in the C-TNN (w_{max}). For the R-TNN, there is reward potentiation (ρ_0^+), reward depression (ρ_0^-), reward window (ω_ρ), punishment potentiation (π_0^+), punishment depression (π_0^-), punishment window (ω_π) and the maximum value for weights in the R-TNN (w_{max}). Potentiation is a term used for weight increase and depression is a term used for weight decrease.

The fourth configuration file is optional and it contains weights for all the synapses in the network. If specified in the network configuration to read weights from this file, then the weight of each individual synapse can be set manually. This is useful for testing ideal network states, where all weights have converged to a number. In the network configuration it can be specified per layer to read the weights or not, this means that one layer can have default weights, while the other has pre set weights, adding more configurability to the complete system.

The network is built by reading the network configuration and then generating neurons and layers based on the specifications in the configuration. The neurons and layers are then connected according to the specifications in mapping configuration. The STDP configuration file is read and then a look-up table is made. During simulations, when weights need to be updated, this look-up table is accessed for the parameters to update the weight by. If the specification to read weights from the weight configuration is enabled, then before the simulations are started, after each connection has been established, weights of all synapses are set according to the weight configuration.

6.2.5. Encoding and Decoding

To input the state of the environment to the network, input encoding is required. In the network configuration file, the number of state variables and the m-hot code need to be specified. Then, for each state variable the intervals need to be defined. These intervals are then read and turned into a look-up table, where each state variable gets its own look-up table. During a step, a function is called to take the state of the environment and to encode this as spikes based on the intervals defined in the network configuration. The number of spikes to encode is based on the number of state variables and m in the m-hot code specification. The spikes are then readied on their respective synaptic inputs and fired on time unit $t = 0$, when the TNN system starts processing. The spikes are processed by the C-TNN, the C-TNN determines what cluster the input belongs to and an output spike is fired from the neuron that belongs to the cluster. Based on the output from the C-TNN, the R-TNN determines which output spike to fire. This output spike is then decoded in the environment, in the case of the cart-pole problem this is a direction to push the cart in.

6.2.6. Spike Timing Dependent Plasticity and Weight Updating

After decoding and updating the state of the environment, the weights of both the C-TNN and the R-TNN need to be updated. For the C-TNN, a function in the layer is called to go over each synapse and to check the five conditions that are listed in Table 5.1. Each synapse's weight is then updated according to the conditions of the input and output spikes of that synapse. Neurons have a function to call which updates the weight of a synapse given its index and the parameters to update the synapse with. This function is called for each synapse of each neuron in the C-TNN.

For the R-TNN, the process is more complicated. The R-TNN layer keeps track of binary flags (e_{ij}) and decay counters (c_{ij}) for all synapses in its synaptic crossbar. A function is called to go over each synapse, take the binary flag of this synapse and the reward broadcast from the environment. If the reward is not zero, identify whether the weight needs to be updated according to the rules of reward potentiation, reward depression, punishment potentiation or punishment depression. The synapse's weight is updated according to the determined rules. The function in the respective neuron is called to update the weight of the synapse. For the R-TNN, this function not only takes the STDP configuration parameters for the R-TNN, it additionally takes c_{ij} from the layer as parameter. This is required to calculate whether there was an input spike within the window (ω). This process is repeated for each

synapse of each neuron in the R-TNN.

6.3. Implementation of Simulator

This section describes the implementation of the other components of the simulator, the environment around the network, pre-simulation setup of environment and network and simulation running. Subsection 6.3.1 explains the implementation of the environment,

6.3.1. Environment Around Neural Network

The environment around the network consists out of the state control system, in this case the cart-pole problem, various parameters to keep track of training and test simulation episodes, parameters to keep track of result metrics and a function to determine the reward that is broadcast to the R-TNN. This thesis uses the cart-pole problem, therefore the environment is explained from the perspective of the cart-pole problem. Note that, due to the modular implementation, separate folders for simulator and environment, a user of the simulator could take away the cart-pole environment and plug in their own environment. The user then only has to specify the network according to their research or experiment and define encoding and decoding in the network configuration required for their environment.

The state in the environment contains all cart-pole parameters, a step function to update the state of the cart-pole system, a reset function to reset the state for a new simulation episode and a function to select a random starting angle for when the state is reset. After the TNN system has produced an output and the environment has decoded this output, the step function is called to update the state of the cart-pole system. The state is updated using the parameters and equations from Section 5.2. After a simulation episode ends due to failure or reaching the cycle limit, the reset function is called to move the cart back to $d = 0$, and the random pole angle function is called to generate a starting angle between -2 and $+2$ degrees. Additionally, the state contains the seed to generate random angles from.

Further, the environment contains the number of simulation episodes to run for the training phase and the test phase. Additionally, it contains a parameter to specify a specific episode to dump to a .csv file. The function that determines the reward is also present in the environment. Reward conditions are experiment specific, in the case of Smith and [35], a positive reward is broadcast for every 500 successful simulation steps, a negative reward is broadcast for failure of the cart-pole system and in all other cases the reward is zero. The environment also contains vectors to track the length of each episode in number of steps and actual time.

6.3.2. Pre-Simulation Setup of Environment and Network

Before any simulations are executed, the code undergoes a setup stage, which is essential for initializing the environment and preparing the necessary components. This process begins with the initialization of the environment and its various components. Following this, a .csv file is created to store the results of a selected simulation episode.

Next, the network configuration file is read, and look-up tables for the state variables are created. Neurons and layers are then generated based on the network configuration. The setup process includes a check to determine whether the network configuration requires reading weights from a weight configuration file. If so, the weight configuration file is read, and the weights for the specified layers are set accordingly.

Then, the STDP configuration file is read, and corresponding look-up tables for the STDP update parameters are created. Subsequently, the mapping configuration file is read, and synaptic inputs are generated. These inputs are connected to the C-TNN, and the C-TNN is then connected to the R-TNN.

Throughout the setup process, the specifications for all configurations are printed for users to verify that everything is correctly configured before the simulation begins. Once all these steps are completed, the function that runs the simulation episodes is called, and the actual simulation process starts.

6.3.3. Simulation Running

The function responsible for running all simulation episodes begins by reading the necessary parameters before starting the actual simulations. This section first discusses these parameters, followed by

a detailed description of the simulation episode process.

Initially, the function checks with the environment to determine if a training phase is required. If training is necessary, the training episodes are executed first. The environment's state is then reset to generate a random starting angle, and the starting time is recorded to allow timing of each episode. A step counter is also initialized to track the duration of each episode. During each simulation episode, the process unfolds as follows:

If the current step is the first of the simulation, a function initializes vectors that track the times of input and output spikes, as well as binary flags (e_{ij}) and decay counters (c_{ij}). For subsequent steps, the spike input and output time vectors are reset, while the binary flags and decay counters remain unchanged, because the binary flags and decay counters track processes that occur over multiple steps.

Next, the state of the environment is obtained. If it is the first step, the starting state is printed for the user. The state is then mapped to an interval and encoded into spikes, which are passed to the TNN system for processing. The spikes are set to fire at the initial time unit $t = 0$, and synaptic input booleans are reset to false to prevent unintended spikes.

The system then enters a loop for each time unit within the TNN, where several operations occur: spikes are fired on their respective synaptic inputs, spike times are captured at C-TNN synapses, neurons in the C-TNN are checked for potential accumulation and threshold crossing, and output spikes are fired if the threshold is crossed. These spikes are then captured in the R-TNN, where similar checks are performed. If the R-TNN generates an output, the TNN system stops, and the output is returned to the simulation function. If no output is generated, the network's booleans are reset to maintain spike integrity, and the process repeats until an output is produced.

Once an output is generated, it is decoded within the environment, and the step function is called with the decoded output, for this thesis a direction to push the cart. The step function then produces a boolean value indicating whether the state is out of bounds, signaling the failure of the cart-pole system if true.

Following this, a reward is determined and broadcast to the R-TNN, and all weights in the network are updated. The step counter is incremented, and checks are performed to determine if the step limit has been reached or if the out-of-bounds condition is met. If either condition is true, the simulation episode is halted.

Upon termination of the episode, the function prints whether the simulation reached the cycle limit or if the cart-pole system failed, along with the final state of the environment. The end time of the episode is recorded, and if the simulation was not in the training phase, the episode time and the number of steps are saved to the environment. This is done to report results metrics to the user after all simulations are done. After saving episode metrics, the episode counter is incremented, and if the simulation was in the training phase and all training episodes have been completed, the function switches to the test phase. The environment is then reset for the next episode, and the starting time and step counter are reset. All neurons in the TNN system are also reset for the next simulation. If the out-of-bounds condition is not met, the neurons are reset for the next step of the episode, and the process repeats until the episode concludes.

After all episodes have been simulated, the function prints the average number of cycles per episode and the average time per episode for all test episodes. Additionally, the final weights of all synapses are printed as these metrics serve as important results.

7

Experimental Setup, Results and Discussion

This chapter discusses the experimental setup of this thesis, it presents the results of this thesis and it provides a discussion with points of improvement. Section 7.1 describes the set up of the TNN system and the cart-pole problem. Section 7.2 discusses the experimental setup to verify the C++ implementation of the cart-pole equations. Section 7.3 explains the experimental setup of the TNN system and cart-pole simulations. Section 7.4 presents the results for the verification of the C++ implementation of the cart-pole equations. Section 7.5 presents the results of the TNN system simulation experiment. Section 7.6 presents the results of Spike Timing Dependent Plasticity in the TNN system and discusses them. Section 7.7 discusses additional improvements for this thesis.

7.1. Setup of Environment: Cart-Pole Problem

For this thesis, as mentioned in Section 6.3, the environment is setup for the cart-pole problem. For initial experiments, the pole angle is the state variable to be encoded as input for the TNN system. The output of the TNN system decodes to a direction to push the cart in, a negative force to push the cart left and a positive force to push the cart right (Figure 5.9). This output is used to call the step function of the environment to update the state of the cart-pole system to the next time step. The state of the system is updated using the equations from Section 5.2 (Equation 5.1 and Equation 5.2) implemented in C++.

7.2. Experimental Setup: Verification of Cart-Pole Equations

Before any experiments using the TNN system can be done, the functionality of the equations implemented in C++ need to be validated. A working implementation of reinforcement learning using the same equations as Smith in [35], can be found in [42]. Farama Foundation's Gymnasium [42] is a fork of OpenAI's Gym, where the team that has been maintaining Gym since 2021 has moved all future development to Gymnasium after OpenAI handed over maintenance to an outside team. Gymnasium is an API standard for reinforcement learning with a diverse collection of reference environments. They provide environments for users to implement their reinforcement learning paradigms in Python. Part of Gymnasium is the ClassicControl library which implements controllers for classic control problems using real physics, one of these being the cart-pole problem. The equations used for the cart-pole system in Gymnasium are the ones from [11] and working controllers using neural networks have been built using the Gymnasium's ClassicControl environment [2]. An experiment is done where the cart is pushed in various directions for a number of steps, both using Gymnasium's Python implementation of the cart-pole equations and this thesis' C++ implementation of the equations. Then, numbers for the angle of the pole, the displacement of the cart, their first and second derivatives are compared. If they are equal or close to equal, then the C++ implementation of the cart-pole equations is verified to work. For the experiment, the same starting angle and cart displacement are used. The cart is pushed three

times per round of comparison, each round having a different order of directions the cart is pushed in. Three pushes is chosen to maintain practicality, while still ensuring that the system is tested for symmetry and to test the cumulative effect of force on the cart. Two pushes might not be sufficient to expose all potential errors or the full range of system dynamics, while four or more pushes may not provide significantly more information, given that the key dynamic behaviors can be tested with sequences of three pushes.

7.3. Experimental Setup: Simulations

An experiment is done where Smith's Ideal T-Learning system [35] is simulated in this thesis' simulator. The state variable processed by the TNN system is the pole angle. The pole angle is encoded by the range $[-12, 12]$ divided into 16 equal ranges using a 3-hot code. The output of the TNN system is then decoded to a direction to push the cart in. Smith's Ideal T-Learning system is a TNN system where all weights are pre set to their ideal values. This means that clusters have already been formed in the C-TNN and that the R-TNN outputs the correct direction to push the cart in for each cluster. The setup of the ideal weights for the C-TNN can be found in Table 7.1 and for R-TNN in Table 7.2. Here synapses are identified by their synaptic input (x_i) and neuron output (z_i). In Table 7.1, the weights are configured in such a way that each input interval is captured by a cluster. In Table 7.2, the weights are configured so that clusters that encode angles smaller than 0 produce an output spike that decodes to push the cart left ($-F$) and that angles equal to or bigger than 0 produce an output spike that decoded to push the cart right ($+F$). Weights are set according to the minimum (0) or maximum weight (w_{max}). So for example, in Table 7.1, the synapses of z_0 capture the the interval $[-12, -10.5]$, since $[-12, -10.5]$ is encoded as an input on x_0, x_1 and x_2. In Table 7.2, the TNN system recognizes this cluster and produces an output spike that is decoded to $-F$, push the cart left.

	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9	z_{10}	z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
x_0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x_1	8	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x_2	8	8	8	0	0	0	0	0	0	0	0	0	0	0	0	0
x_3	0	8	8	8	0	0	0	0	0	0	0	0	0	0	0	0
x_4	0	0	8	8	8	0	0	0	0	0	0	0	0	0	0	0
x_5	0	0	0	8	8	8	0	0	0	0	0	0	0	0	0	0
x_6	0	0	0	0	8	8	8	0	0	0	0	0	0	0	0	0
x_7	0	0	0	0	0	8	8	8	0	0	0	0	0	0	0	0
x_8	0	0	0	0	0	0	8	8	8	0	0	0	0	0	0	0
x_9	0	0	0	0	0	0	0	8	8	8	0	0	0	0	0	0
x_{10}	0	0	0	0	0	0	0	0	8	8	8	0	0	0	0	0
x_{11}	0	0	0	0	0	0	0	0	0	8	8	8	0	0	0	0
x_{12}	0	0	0	0	0	0	0	0	0	0	8	8	8	0	0	0
x_{13}	0	0	0	0	0	0	0	0	0	0	0	8	8	8	0	0
x_{14}	0	0	0	0	0	0	0	0	0	0	0	0	8	8	8	0
x_{15}	0	0	0	0	0	0	0	0	0	0	0	0	0	8	8	8
x_{16}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	8
x_{17}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8

Table 7.1: Ideal weights for the C-TNN

All simulations are done with random starting angles, generated by a pseudo-random number generator. These starting angles are between $-2° \leq \angle \leq 2°$. Smith uses the MATLAB pseudo-random number generator and has found that the specific seed has a significant effect on the results. Smith uses eight random seeds to avoid cherry-picking a good seed, however these 8 seeds are not given. The random number generator engine used in this thesis is the Mersenne Twister [25]. This engine is used in C++ for generating pseudo-random numbers from a uniform distribution. The seeds used for this pseudo-random number generator are the numbers 0 to 7. The C++ Mersenne Twister engine ensures a high quality of randomness and a long period. Even with small seeds like 0 to 7, the generator is still able to produce sufficiently varied random sequences. This ensures that the experiment is not biased by

	z_0	z_1
x_0	0	8
x_1	0	8
x_2	0	8
x_3	0	8
x_4	0	8
x_5	0	8
x_6	0	8
x_7	0	8
x_8	8	0
x_9	8	0
x_{10}	8	0
x_{11}	8	0
x_{12}	8	0
x_{13}	8	0
x_{14}	8	0
x_{15}	8	0

Table 7.2: Ideal weights for the R-TNN

the choice of seeds, as each seed produces a well-distributed random sequence within the generator's capabilities. Given the unknowns in Smith's methodology, and the practical constraints of exploring every possible seed, choosing a small, manageable range like 0 to 7 is therefore the most practical approach. Additionally, using a simple and well-defined range of seeds, like 0 to 7, ensures the ease of reproducibility of this experiment. All other parameters are equal to the ones listed in Section 5.2.

Results between this thesis and Smith's [35] are compared on simulated time length of keeping the simulation within bounds. Within bounds is defined as $-12° \leq \angle \leq 12°$ and $-2.4 \leq d \leq 2.4$, for the pole angle ($\angle$) and the cart displacement (d).

7.4. Results: Cart-Pole Equations

The configuration of the cart-pole system used by Gymnasium can be found in Table 7.3, for the validation of the C++ cart-pole equations the same configuration is used. The results for the validation of the cart-pole equations can be found in Appendix A. Here R is a cart push to the right and L is a cart push to the left. In Table A.1, it can be seen that the values for all state variables are equal when pushing the cart right three times and when pushing the cart left three times. In Table A.2, the same can be seen, all values for all state variables are equal when pushing the cart right, then left, then right, as well as for the mirrored variant. For each mirrored variant, it can be seen that values are equal, but negative, therefore Table A.3 does not include the mirrored variants. In Table A.3, the same can be seen again, all values for all state variables are equal when pushing the cart right once and then left two times, and when pushing the cart right two times and then left once.

Parameter	Value
$\angle$ at $t = 0$	0
$\dot{\angle}$ at $t = 0$	0
d at $t = 0$	0
$\dot{d}$ at $t = 0$	0
F	±10
M	0.1
m	1
l	0.5
g	9.8
τ	0.02

Table 7.3: Cart-Pole mechanical equations verification experiment parameters.

7.5. Results: Simulations

In [35], for the Ideal T-Learning system, Smith reports results of 6,000 to 7,500 successful steps per simulation episode on average. Smith's simulations are done with a maximum number of 10,000 steps for a simulation episode, which is equal to 200 seconds of simulated time. That means for Smith's Ideal T-Learning system the cart-pole system can be balanced for 120 to 150 seconds of simulated time.

The results for the simulation using the same parameters as in Section 5.2 can be found in Figure 7.1. Here it can be seen that the cart-pole system can only be successfully balanced for 5.5 to 6.5 seconds. This is far off the 120 to 150 seconds Smith reports. This discrepancy may be attributed to the hypothesis that Smith's MATLAB implementation utilizes a more granular time step. Since Smith likely did not construct the cart-pole system from scratch, but rather used an existing simulation library, the time step in his implementation might be more fine-grained than explicitly reported, resulting in an extended duration of pole balancing on the cart in the cart-pole system.

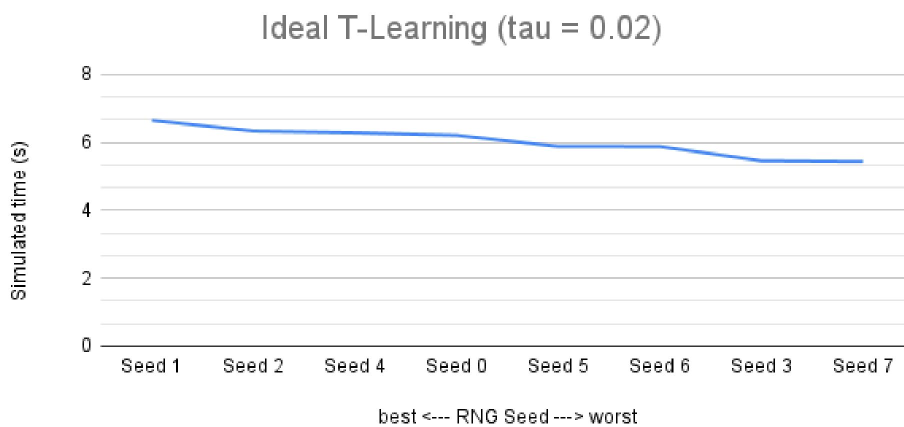


Figure 7.1: Simulation results for Ideal T-Learning with tau = 0.02

To test this hypothesis, τ is halved for the next simulation. The changed parameters for the next simulation are $\tau = 0.01$ which is half of $\tau = 0.02$, and to maintain a maximum of 200 seconds of simulated time the maximum step count is doubled from 10,000 to 20,000. The rest of the parameters remain the same. The results of the next simulation can be found in Figure 7.2. Here it can be seen that the cart-pole system can be successfully balanced for 17 to 23 seconds of simulated time. This is a step in the right direction, however this is still not close to 120 to 150 seconds of simulated time.

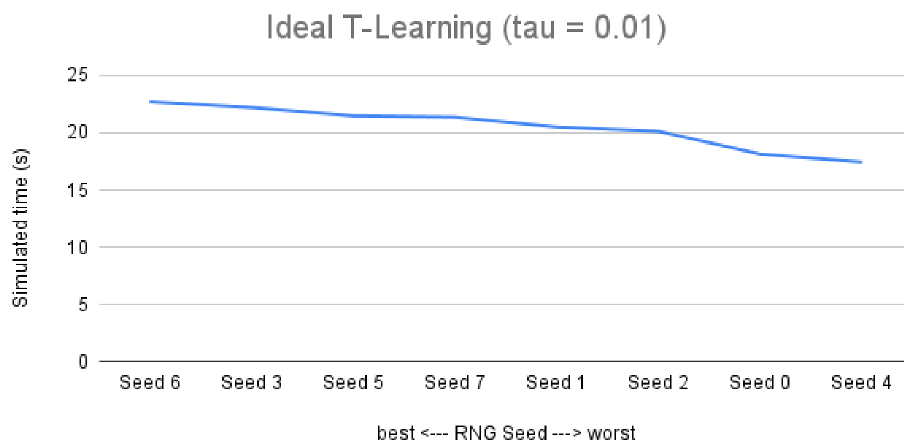


Figure 7.2: Simulation results for Ideal T-Learning with tau = 0.01

Therefore, τ is halved again for the next simulation. The changed parameters for the next simulation

are $\tau = 0.005$, and the maximum number of steps is 40,000, other parameters remain unchanged. The results of the next simulation can be found in Figure 7.3. Here it can be seen that the cart-pole system is successfully balanced for 60 to 70 seconds of simulated time. This duration is nearly half of the 120 to 150 seconds of simulated time reported by Smith.

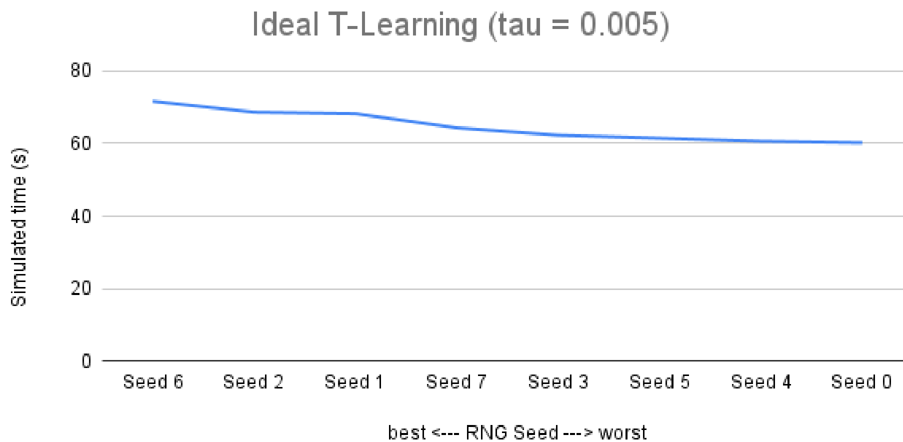


Figure 7.3: Simulation results for Ideal T-Learning with tau = 0.005

To obtain results comparable to Smith's, τ is halved one more time. The changed parameters for the next simulation are $\tau = 0.0025$, and the maximum number of steps is 80,000, other parameters remain unchanged. The results of the next simulation can be found in Figure 7.4. Here it can be seen that the cart-pole system is successfully balanced for 130 to 150 seconds. This is on par with Smith's reported results.

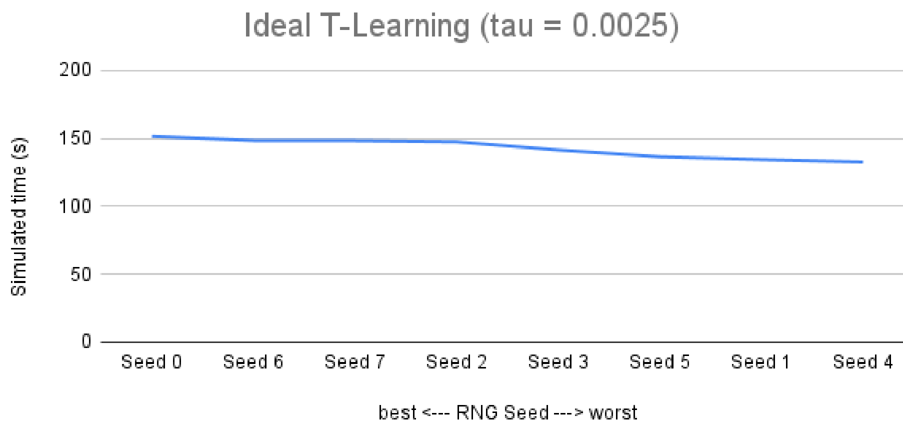


Figure 7.4: Simulation results for Ideal T-Learning with tau = 0.0025

By lowering τ to 0.0025, the cart-pole system is pushed and updated eight times more frequently compared to $\tau = 0.02$. As a result, the system integrates more frequently over the same simulated time, enabling the controller to respond more swiftly and accurately. This allows the controller to better determine the optimal moment to apply force in the opposite direction, thereby minimizing the swing of the pole and preventing it from tipping excessively towards the side it is falling on.

7.6. Results: Experiments with Spike Timing Dependent Plasticity

When simulating the TNN system with default weights, instead of pre-trained weights, the C-TNN's weights should form clusters to capture input patterns for different pole angles and the R-TNN should converge such that for a cluster the cart is pushed in the direction it balances the pole. At $\tau = 0.02$,

clustering in the C-TNN worked, however it had limited success and weights in the R-TNN converge to w_{max}. The hypothesis for the C-TNN is that simulation episode length is too short to capture clusters effectively. For example, when running simulations with default weights at $tau = 0.02$, for seed 0, the average episode length is 0.1 seconds of simulated time. The hypothesis for the R-TNN is that the cart-pole system always fails before a positive reward can be broadcast. Additionally, due to the C-TNN not forming effective clusters, it cannot learn what the correct direction is to push the cart.

At $\tau = 0.0025$, the C-TNN captures clusters effectively. When running simulations with default weights at $\tau = 0.0025$, for seed 0, 13 out of 16 neurons in the C-TNN capture clusters, all capturing a different input angles. The STDP parameters used are the same as Smith in [35], these can be found in Table 7.4. Simulation is done with 30 episodes of training and then 50 test episodes, the training process continues in the test phase, all in accordance to Smith's simulation parameters in [35]. All other parameters are equal to the Ideal T-Learning experiment. In Table 7.5, it can be seen that neurons z_0 to z_{12} make clusters for different input angles. Neurons z_{13}, z_{14} and z_{15} have not captured an input pattern. It can also be seen that z_{10} and z_{12} have not fully converged yet, but for both neurons three weights have converged to the maximum, with all other weights (close to) zero, effectively capturing a cluster. To give some examples of angle intervals captured, neuron z_0 captures the ninth interval $[0, 1.5]$, neuron z_2 captures the third interval $[-9, -7.5]$ and neuron z_8 captures the twelfth interval $[4.5, 6]$. The only intervals not captured are $[7.5, 9]$, $[9, 10.5]$ and $[10.5, 12]$. Which means this intervals are rarely input to the TNN system. A potential cause for this can be that the system is biased to fail on the left side (range $[-12, 0]$).

STDP Parameter	Value	Description
θ_C	6	*threshold*
μ_c	1/16	*capture*
μ_b	1/16	*backoff*
μ_s	0	*search*

Table 7.4: STDP parameters used for the C-TNN

	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9	z_{10}	z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
x_0	0	0	0	0	0	0	0	0	0	0	8	0	1.125	5	5	5
x_1	0	0	0	0	0	0	0	0	0	0	8	0	8	5	5	5
x_2	0	0	8	0	0	0	0	0	0	0	8	0	8	5	5	5
x_3	0	0	8	0	0	0	0	0	0	8	0.5	0	8	5	5	5
x_4	0	0	8	0	8	0	0	0	0	8	0	0	1.125	5	5	5
x_5	0	0	0	0	8	0	0	8	0	8	0	0	1.125	5	5	5
x_6	0	8	0	0	8	0	0	8	0	0	0	0	1.125	5	5	5
x_7	0	8	0	8	0	0	0	8	0	0	0	0	1.125	5	5	5
x_8	8	8	0	8	0	0	0	0	0	0	0	0	1.125	5	5	5
x_9	8	0	0	8	0	8	0	0	0	0	0	0	1.125	5	5	5
x_{10}	8	0	0	0	0	8	8	0	0	0	0	0	1.125	5	5	5
x_{11}	0	0	0	0	0	8	8	0	8	0	0	0	1.125	5	5	5
x_{12}	0	0	0	0	0	0	8	0	8	0	0	8	1.125	5	5	5
x_{13}	0	0	0	0	0	0	0	0	8	0	0	8	1.125	5	5	5
x_{14}	0	0	0	0	0	0	0	0	0	0	0	8	1.125	5	5	5
x_{15}	0	0	0	0	0	0	0	0	0	0	0	0	1.125	5	5	5
x_{16}	0	0	0	0	0	0	0	0	0	0	0	0	1.125	5	5	5
x_{17}	0	0	0	0	0	0	0	0	0	0	0	0	1.125	5	5	5

Table 7.5: C-TNN Weights for seed 0 after STDP at $\tau = 0.0025$

At $\tau = 0.0025$, a majority of the weights in the R-TNN converge correctly. For 9 of the 13 caotured clusters, the direction to push the cart in output by the system balances the pole. The STDP parameters used are the same as Smith in [35], these can be found in Table 7.6. All other parameters are equal to the Ideal T-Learning experiment. For the decoding, an output spike from neuron z_0 is decoded to

push the cart right ($+F$) and an output spike from neuron z_1 is decoded to push the cart left ($-F$). In Table 7.7, it can be seen that the clusters encoded by synaptic inputs x_0, x_1, x_2 to x_8 and x_{10} have the correct direction to push the cart. For an ideal system the cart should be pushed right if the pole is on the right (angle range $[0, 12]$) and pushed left if the cart is on pole is on the left (angle range $[-12, 0]$). For example, synaptic input x_0 is connected to C-TNN output z_0, according to Table 7.5 this encodes the angle interval $[0, 1.5]$. Angle interval $[0, 1.5]$ is part of the range $[0, 12]$ and for this range the cart should be pushed right ($+F$). In Table 7.7, the synapses for x_0, the weights are $w_{00} = 8$ and $w_{01} = 0$, which means an output spike always fires from z_0, so the cart is pushed right ($+F$).

STDP Parameter	Value	Description
θ_R	2	*threshold*
ρ_0^+, ρ_0^-	3/2	*reward potentiation and depression*
ω_ρ	2	*reward window*
π_0^+, π_0^-	3/2	*punishment potentiation and depression*
ω_π	16	*punishment window*

Table 7.6: STDP parameters used for the R-TNN

	z_0	z_1
x_0	8	0
x_1	8	0
x_2	8	8
x_3	0	8
x_4	0	8
x_5	0	8
x_6	8	0
x_7	0	8
x_8	8	0
x_9	7.0625	5.5625
x_{10}	3.59375	6.40625
x_{11}	5	5
x_{12}	8	8
x_{13}	5	5
x_{14}	5	5
x_{15}	5	5

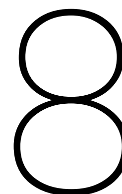
Table 7.7: R-TNN Weights for seed 0 after STDP at $\tau = 0.0025$

Other seeds outside of seed 0 have shown to be able to capture clusters effectively, however the R-TNN weights do not converge correctly. For example, for seed 1, 10 out of 16 neurons captured clusters and for only 5 of these clusters the cart is pushed in the direction to balance the pole. For seed 2, 11 out of 16 neurons captured clusters and for only 4 of these clusters the cart is pushed in the direction to balance the pole.

7.7. Discussion: Additional Improvements

Certain components of the environment implemented in this thesis' simulator should be part of the simulator code. These are the number of simulation episodes for the training and test phase, and the reward generation system for the R-TNN. If a real cart-pole system is to be connected to the TNN system to be used as a controller, then the number of episodes to balance the pole is nowhere to be defined in the physical system. Therefore, this should be part of the TNN simulator folder. This can be achieved by making a simulation configuration file, with parameters such as the number of training and test episodes. In a real-cart pole system, the physical environment also cannot generate a reward signal. Therefore, this also needs to be part of the TNN system code. The rules and conditions of the reward signal still need to be defined by the user, therefore these could be added to the simulation configuration, together with the number of training and test episodes.

Furthermore, the time step of the TNN system and the environment should be decoupled. If a real cart-pole system is to be connected to the TNN system to be used as a controller, the cart-pole system moves in real-time, while the TNN system can only read the state of the environment at a given rate. This should be present in the simulator as well. This can be implemented by making making separate time step configuration parameters for the TNN system and the environment. Another method of implementation is making the step of the environment a scaled down step of the TNN system, by dividing by a factor.



Conclusion

8.1. Summary

The research question of this thesis is: Can a simulator be developed in an environment that would enable large-scale experimentation and testing of TNN systems? This thesis has presented a compiled code event driven simulator for TNN systems. Past works [37], [34], [35] have made simulators in MATLAB, an environment which makes large-scale experiments difficult. This simulator is implemented in C++, which offers performance advantages over MATLAB. Unlike MATLAB, which can be slower, C++ provides more efficient execution and faster processing times.. Additionally, results with an increased time discretization were presented that match previous work [35] for pre-trained TNN systems. Furthermore, preliminary results were shown for reinforcement learning using TNN systems.

8.2. Future Work

In the future, the focus lies on making the simulator more decoupled. This means moving the parameters for training and test simulation episodes to the TNN system code, moving the reward generation to the TNN system code and to separate the time step of the TNN system and the environment. Other future work, after further decoupling the system, can be done by encoding multiple state variables, to see if the cart-pole system can be balanced for a longer simulated time, or to attach different environments.

References

[1] Larry F Abbott. “Lapicque’s introduction of the integrate-and-fire model neuron (1907)”. In: *Brain research bulletin* 50.5-6 (1999), pp. 303–304.

[2] Adibyte. *GitHub - adibyte95/CartPole-OpenAI-GYM: solution to cartpole problem of openAI gym with different approaches*. URL: `https://github.com/adibyte95/CartPole-OpenAI-GYM`.

[3] Andrew G Barto, Richard S Sutton, and Charles W Anderson. “Neuronlike adaptive elements that can solve difficult learning control problems”. In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 834–846.

[4] Yoshua Bengio et al. “Greedy layer-wise training of deep networks”. In: *Advances in neural information processing systems* 19 (2006).

[5] Guo-qiang Bi and Mu-ming Poo. “Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type”. In: *Journal of neuroscience* 18.24 (1998), pp. 10464–10472.

[6] Olivier Bichler et al. “Extraction of temporally correlated features from dynamic vision sensors with spike-timing-dependent plasticity”. In: *Neural networks* 32 (2012), pp. 339–348.

[7] Sander M Bohte, Joost N Kok, and Johannes A La Poutré. “SpikeProp: backpropagation for networks of spiking neurons.” In: *ESANN*. Vol. 48. Bruges. 2000, pp. 419–424.

[8] Hervé Bourlard and Yves Kamp. “Auto-association by multilayer perceptrons and singular value decomposition”. In: *Biological cybernetics* 59.4-5 (1988), pp. 291–294.

[9] Junyoung Chung et al. “Empirical evaluation of gated recurrent neural networks on sequence modeling”. In: *arXiv preprint arXiv:1412.3555* (2014).

[10] Jeffrey L Elman. “Finding structure in time”. In: *Cognitive science* 14.2 (1990), pp. 179–211.

[11] Razvan V Florian. “Correct equations for the dynamics of the cart-pole system”. In: *Center for Cognitive and Neural Studies (Coneural), Romania* (2007), p. 63.

[12] Wulfram Gerstner. “Time structure of the activity in neural network models”. In: *Physical review E* 51.1 (1995), p. 738.

[13] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[14] Geoffrey E Hinton, Terrence J Sejnowski, et al. “Learning and relearning in Boltzmann machines”. In: *Parallel distributed processing: Explorations in the microstructure of cognition* 1.282-317 (1986), p. 2.

[15] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[16] John J Hopfield. “Neural networks and physical systems with emergent collective computational abilities.” In: *Proceedings of the national academy of sciences* 79.8 (1982), pp. 2554–2558.

[17] Eugene M Izhikevich. “Simple model of spiking neurons”. In: *IEEE Transactions on neural networks* 14.6 (2003), pp. 1569–1572.

[18] Diederik P Kingma and Max Welling. “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (2013).

[19] Teuvo Kohonen. “Self-organized formation of topologically correct feature maps”. In: *Biological cybernetics* 43.1 (1982), pp. 59–69.

[20] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[21] WB Levy and O Steward. "Temporal contiguity requirements for long-term associative potentiation/depression in the hippocampus". In: *Neuroscience* 8.4 (1983), pp. 791–797.

[22] Wolfgang Maass. "Networks of spiking neurons: the third generation of neural network models". In: *Neural networks* 10.9 (1997), pp. 1659–1671.

[23] Henry Markram et al. "Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs". In: *Science* 275.5297 (1997), pp. 213–215.

[24] Timothée Masquelier and Simon J Thorpe. "Unsupervised learning of visual features through spike timing dependent plasticity". In: *PLoS computational biology* 3.2 (2007), e31.

[25] Makoto Matsumoto and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30.

[26] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[27] Carver Mead and Mohammed Ismail. *Analog VLSI implementation of neural systems*. Vol. 80. Springer Science & Business Media, 2012.

[28] Donald Michie and RA Chambers. "'Boxes' as a model of pattern-formation". In: *The Origin of Life*. Routledge, 2017, pp. 206–215.

[29] Donald Michie and Roger A Chambers. "BOXES: An experiment in adaptive control". In: *Machine intelligence* 2.2 (1968), pp. 137–152.

[30] John Moody and Christian J Darken. "Fast learning in networks of locally-tuned processing units". In: *Neural computation* 1.2 (1989), pp. 281–294.

[31] Savinay Nagendra et al. "Comparison of reinforcement learning algorithms applied to the cart-pole problem". In: *2017 international conference on advances in computing, communications and informatics (ICACCI)*. IEEE. 2017, pp. 26–32.

[32] Marc'Aurelio Ranzato et al. "Efficient learning of sparse representations with an energy-based model". In: *Advances in neural information processing systems* 19 (2006).

[33] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[34] James E Smith. "A temporal neural network architecture for online learning". In: *arXiv preprint arXiv:2011.13844* (2020).

[35] James E Smith. "Implementing Online Reinforcement Learning with Temporal Neural Networks". In: *arXiv preprint arXiv:2204.05437* (2022).

[36] James E Smith. "Temporal Computer Organization". In: *arXiv preprint arXiv:2201.07742* (2022).

[37] James E Smith and Margaret Martonosi. *Space-time computing with temporal neural networks*. Springer, 2017.

[38] Paul Smolensky et al. "Information processing in dynamical systems: Foundations of harmony theory". In: (1986).

[39] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[40] Gunther S Stent. "A physiological mechanism for Hebb's postulate of learning". In: *Proceedings of the National Academy of Sciences* 70.4 (1973), pp. 997–1001.

[41] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[42] Mark Towers et al. "Gymnasium: A Standard Interface for Reinforcement Learning Environments". In: *arXiv preprint arXiv:2407.17032* (2024).

[43] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

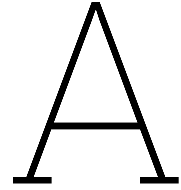
[44] Pascal Vincent et al. "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 1096–1103.

[45] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. “On early stopping in gradient descent learning”. In: *Constructive Approximation* 26.2 (2007), pp. 289–315.

[46] Rafael Yuste. “From the neuron doctrine to neural networks”. In: *Nature reviews neuroscience* 16.8 (2015), pp. 487–497.

[47] Yu-Dong Zhang et al. “Improved breast cancer classification through combining graph convolutional network and convolutional neural network”. In: *Information Processing & Management* 58.2 (2021), p. 102439.

[48] Jinming Zou, Yi Han, and Sung-Sau So. “Overview of artificial neural networks”. In: *Artificial neural networks: methods and applications* (2009), pp. 14–22.



Tables of Results: Cart-Pole Equations

	Python RRR	C++ RRR	Python LLL	C++ LLL
$\angle$ at $t = 0$	0	0	0	0
$\dot{\angle}$ at $t = 0$	-0.2926829268	-0.2926829268	0.2926829268	0.2926829268
$\ddot{\angle}$ at $t = 0$	-14.63414634	-14.63414634	14.63414634	14.63414634
d at $t = 0$	0	0	0	0
$\dot{d}$ at $t = 0$	0.1951219512	0.1951219512	-0.1951219512	-0.1951219512
$\ddot{d}$ at $t = 0$	9.756097561	9.756097561	-9.756097561	-9.756097561
$\angle$ at $t = 1$	-0.3353899289	-0.3353899289	0.3353899289	0.3353899289
$\dot{\angle}$ at $t = 1$	-0.5853658537	-0.5853658537	0.5853658537	0.5853658537
$\ddot{\angle}$ at $t = 1$	-14.63414634	-14.63414634	14.63414634	14.63414634
d at $t = 1$	0.003902439024	0.003902439024	-0.003902439024	-0.003902439024
$\dot{d}$ at $t = 1$	0.3902439024	0.3902439024	-0.3902439024	-0.3902439024
$\ddot{d}$ at $t = 1$	9.756097561	9.756097561	-9.756097561	-9.756097561
$\angle$ at $t = 2$	-1.006169787	-1.006169787	1.006169787	1.006169787
$\dot{\angle}$ at $t = 2$	-0.8798869825	-0.8798869825	0.8798869825	0.8798869825
$\ddot{\angle}$ at $t = 2$	-14.72605644	-14.72605644	14.72605644	14.72605644
d at $t = 2$	0.01170731707	0.01170731707	-0.01170731707	-0.01170731707
$\dot{d}$ at $t = 2$	0.5854473555	0.5854473555	-0.5854473555	-0.5854473555
$\ddot{d}$ at $t = 2$	9.760172654	9.760172654	-9.760172654	-9.760172654

Table A.1: Comparison of Python and C++ for RRR and LLL

	Python RLR	C++ RLR	Python LRL	C++ LRL
$\angle$ at $t = 0$	0	0	0	0
$\dot{\angle}$ at $t = 0$	-0.2926829268	-0.2926829268	0.2926829268	0.2926829268
$\ddot{\angle}$ at $t = 0$	-14.63414634	-14.63414634	14.63414634	14.63414634
d at $t = 0$	0	0	0	0
$\dot{d}$ at $t = 0$	0.1951219512	0.1951219512	-0.1951219512	-0.1951219512
$\ddot{d}$ at $t = 0$	9.756097561	9.756097561	-9.756097561	-9.756097561
$\angle$ at $t = 1$	-0.3353899289	-0.3353899289	0.3353899289	0.3353899289
$\dot{\angle}$ at $t = 1$	0	0	0	0
$\ddot{\angle}$ at $t = 1$	14.63414634	14.63414634	-14.63414634	-14.63414634
d at $t = 1$	0.003902439024	0.003902439024	-0.003902439024	-0.003902439024
$\dot{d}$ at $t = 1$	0	0	0	0
$\ddot{d}$ at $t = 1$	-9.756097561	-9.756097561	9.756097561	9.756097561
$\angle$ at $t = 2$	-0.3353899289	-0.3353899289	0.3353899289	0.3353899289
$\dot{\angle}$ at $t = 2$	-0.2945240641	-0.2945240641	0.2945240641	0.2945240641
$\ddot{\angle}$ at $t = 2$	-14.7262032	-14.7262032	14.7262032	14.7262032
d at $t = 2$	0.003902439024	0.003902439024	-0.003902439024	-0.003902439024
$\dot{d}$ at $t = 2$	0.1952054099	0.1952054099	-0.1952054099	-0.1952054099
$\ddot{d}$ at $t = 2$	9.760270496	9.760270496	-9.760270496	-9.760270496

Table A.2: Comparison of Python and C++ for RLR and LRL

	Python RLL	C++ RLL	Python RRL	C++ RRL
$\angle$ at $t = 0$	0	0	0	0
$\dot{\angle}$ at $t = 0$	-0.2926829268	-0.2926829268	-0.2926829268	-0.2926829268
$\ddot{\angle}$ at $t = 0$	-14.63414634	-14.63414634	-14.63414634	-14.63414634
d at $t = 0$	0	0	0	0
$\dot{d}$ at $t = 0$	0.1951219512	0.1951219512	0.1951219512	0.1951219512
$\ddot{d}$ at $t = 0$	9.756097561	9.756097561	9.756097561	9.756097561
$\angle$ at $t = 1$	-0.3353899289	-0.3353899289	-0.3353899289	-0.3353899289
$\dot{\angle}$ at $t = 1$	0	0	-0.5853658537	-0.5853658537
$\ddot{\angle}$ at $t = 1$	14.63414634	14.63414634	-14.63414634	-14.63414634
d at $t = 1$	0.003902439024	0.003902439024	-0.003902439024	-0.003902439024
$\dot{d}$ at $t = 1$	0	0	0.3902439024	0.3902439024
$\ddot{d}$ at $t = 1$	-9.756097561	-9.756097561	9.756097561	9.756097561
$\angle$ at $t = 2$	-0.3353899289	-0.3353899289	-1.006169787	-1.006169787
$\dot{\angle}$ at $t = 2$	0.2908302931	0.2908302931	-0.2945326253	-0.2945326253
$\ddot{\angle}$ at $t = 2$	14.54151466	14.54151466	14.54166142	14.54166142
d at $t = 2$	0.003902439024	0.003902439024	0.01170731707	0.01170731707
$\dot{d}$ at $t = 2$	0.1950375141	0.1950375141	0.1952044315	0.1952044315
$\ddot{d}$ at $t = 2$	-9.751875706	-9.751875706	-9.751973547	-9.751973547

Table A.3: Comparison of Python and C++ for RLL and RRL