CONSTRAINT PROGRAMMING IN SPIKING NEURAL NETWORKS

a thesis submitted to
The University of Kent
in the subject of computer science
for the degree
of phd.

By
Huy Le Nguyen
October 2022

Abstract

Spiking Neural Networks (SNNs) are a class of event-driven and low-power Artificial Neural Networks which aim to closely mimic the computational dynamics which are observed in biological nervous systems. These networks employ different architectural designs and computational characteristics, when compared to the more commonly used Analogue Neural Networks (ANNs). As a consequence, existing tried-and-true Machine Learning methods which have proven effective for training ANNs may not directly work on SNNs, but require new interpretations or approximations to be applicable. It is currently still unclear how SNNs can deliver on the promises of high-performance computing at reduced energy costs.

The work in this thesis addresses the problem of efficiently training SNNs on traditional von Neumann hardware platforms. Although supervised learning rules that allow SNNs to learn spatio-temporal spike-pattern mappings have been developed and studied for a variety of problem domains, the computational paradigm of these methods can be broadly categorised into iterative or one-batch methods, with their own advantages and limitations. The research conducted here aims to combine computational properties from both of these two families of methods, in order to derive hybrid learning algorithms which exhibit improved learning efficiency.

First, we introduce a novel learning rule for supervised training of single-layer SNNs to solve precise input-output spike train mapping problems. This algorithm first converts the learning task into the form of a Constrained Satisfaction Problem (CSP), with the aim of computing the precise *step size* with which the spike-mapping problem can be solved with a single update step. In practice, the constraints of performing computation in continuous time means that the

method still require a number of updates to converge, however the required number of learning iterations will be several times fewer than with traditional iterative learning. We will show that the proposed algorithm is viable and efficient through extensive numerical simulations.

Next, we apply the proposed learning rule to supervised learning tasks with spike-count encoding, in which only the number of output spikes are specified and not their exact timings. Encoding the network output with a spike-count have become the norm in classification tasks, since it has reduced computational requirements for inference. Here, our algorithm demonstrates competitive generalisation accuracy and improved convergence speed on common data classification benchmarks, in comparison to existing methods in the literature. Additionally, we perform an experiment in order to measure the maximal learning capacity of the algorithm in spike-count learning problems.

Finally, we present an extension of the proposed algorithm to perform unsupervised feature extraction in networks with convolutional layers. When used sequentially with the original algorithm, we are able to partially address the main weakness of the CSP weight update approach, which is its inability to train multilayer architectures. The application of our algorithms to the convolutional network architecture is examined in depth, highlighting some of their strengths and weaknesses. Using these findings, we apply the methods to three well-known image classification benchmark problems.

Acknowledgements

I would like to express my most sincere gratitude to:

My supervisor, Dr. Dominique Chu, for his friendship, guidance, enthusiasm, unerring adherence to quality, and seemingly endless patience.

The School of Computing at the University of Kent, and all of its staff, for providing all the facilities and resources to support my research.

My parents, grandparents, and my brother, who have always treated my troubles as their own, and have provided constant support throughout this process.

My partner, Laura Costin, without whom I would not have the courage to pursue my further studies.

And last but not least to all of my friends and colleagues, who have made this a vibrant and stimulating journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Preface

The field of Machine Learning applies *learning algorithms* to data in order to automatically build *computational models*. In supervised ML, the purpose of the model is to describe the relationship between input and output data, such that the resulting model approximates the underlying data-generating function. In unsupervised ML, the model aims to extract and store some information which is hidden in the data, for example to separate the data points into a number of clusters based on their statistical similarities. Successful ML models are those which are able to describe the training data accurately, and also to maintain high performance when applied to new, previously unseen data (to *generalise*).

One of the most prolific areas of modern ML is the study of Artificial Neural Networks, which takes inspiration from computational neuroscience in order to design models of *artificial neurons*. An artificial neuron is a fundamental computational node which typically performs a simple computation, for example a linear combination of its inputs expressed as a dot product. Neurons are then connected to each other to form larger and larger models, which allows the network as a whole to perform complicated and high-dimensional computations.

The vast majority of modern neural network models fall into the category of *Analogue Neural Networks* (ANNs), which are mainly made up of neurons which both receive and generate continuous values Maass (1997); Bengio et al.

(2009). ANN models have been applied to a variety of difficult problem areas, for example Computer Vision (Krizhevsky, Sutskever and Hinton 2012; Simonyan and Zisserman 2014), Natural Language Processing (Mikolov et al. 2013; Vaswani et al. 2017; Ghojogh and Ghodsi 2020), Medical Diagnosis (Elveren and Yumuşak 2011; Catalogna et al. 2012; Barwad, Dey and Susheilia 2012), and many others. Large networks with millions of parameters are now routinely trained, and it is a growing challenge to keep up with the rising cost of energy consumption required to simulate, train, and deploy neural networks (Li et al. 2016; Alyamkin et al. 2018). ANNs have become so commonly used, that the name is often used interchangeably with Artificial Neural Networks. Here, we will use ANNs to denote specifically networks of continuously-activated neurons.

Spiking Neural Networks (SNNs) are a more recent type of architecture which offer a promising alternative to ANNs. Utilising *spiking neuron models* as the primary computational node, SNNs closely imitate the processing and dynamics of biological neurons. Unlike the analogue artificial neurons, spiking neuron models communicate information via discrete, all-or-nothing events called *spikes*. It has been theorised that a single simplified spiking neuron model exhibits greater computational power, information density, and inference speed when compared to its analogue counterpart (Maass 1997; Borst and Theunissen 1999; Thorpe, Fize and Marlot 1996). Furthermore, recent developments in spiking neural simulators and ultra energy-efficient neuromorphic emulators have promised to leverage the event-driven computation of SNNs for novel neural network implementations (Roy, Jaiswal and Panda 2019; DeBole et al. 2019; Orchard et al. 2021). These characteristics suggest that SNNs may offer potential benefits for building smaller and faster neural networks while maintaining high performance and precision (Thorpe, Delorme and Van Rullen 2001; Young et al. 2019).

Despite their relative prevalence in recent studies, there are several factors which are currently limiting the wide-spread adoption of SNNs in real-world applications. One major challenge is the inefficiencies of SNN simulation on traditional von Neumann hardware platforms. Computations in SNNs are carried out in an online manner, where the training data is processed sequentially and the state of each neuron evolves asynchronously over time. While this means that SNNs are amenable to highly parallel simulation strategies, the hardware architecture

must support fine-grained granularity while taking into account the communication overhead (Davies et al. 2018). Executing these temporal computations on CPUs and GPUs can be much more difficult when compared to the offline linear algebra approaches employed by traditional ANNs. Additionally, most SNNs are expressed as a system of Ordinary Differential Equations (ODEs) with no analytical solutions, and thus require numerical approximations (Gerstner et al. 2014; Valadez-Godínez, Sossa and Santiago-Montero 2020). Efforts to overcome these limitations constitute an active and ongoing field of SNN research (Ros et al. 2006; Brette et al. 2007; Naveros et al. 2014, 2017; Valadez-Godínez, Sossa and Santiago-Montero 2020; Qu et al. 2020).

Another significant challenge for modern SNN applications is the lack of a general-purpose learning algorithm (Grüning and Bohte 2014). For modern ANNs, gradient-descent learning via the Back-Propagation Through Time (BPTT) algorithm is a powerful procedure which is both problem-agnostic and model-agnostic (Linnainmaa 1976; Rumelhart, Hinton and Williams 1986; Bengio et al. 2009; Lillicrap and Santoro 2019). One faces a number of difficulties when applying gradient-descent for SNNs: firstly, the discrete spike activation found in spike-based models is generally not differentiable (Li et al. 2021). To circumvent this problem, a number of surrogate approximations of the spike derivative have been proposed, with varying degrees of complexity (Bohte, Kok and La Poutré 2000; Zenke and Ganguli 2018; Shrestha and Orchard 2018; Lee et al. 2020). Secondly, BPTT incorporates a number of mechanisms which prove particularly challenging for neuromorphic hardware implementations. One example is that neuromorphic chips have limited resolution and no access to external memory, which is problematic for bidirectional weight transport during the backward propagation of gradients (Bengio et al. 2015; Christensen et al. 2022). Because of these well-known difficulties, traditional BPTT is famously incompatible with the design of current neuromorphic hardware, and *on-chip* gradient-descent remains a difficult challenge (Kwon et al. 2020; Renner et al. 2021).

1.2 Thesis Contributions

The contributions of this thesis are three novel learning algorithms for training SNNs. The algorithms are titled *Discrete Threshold Assumption* (DTA). The details of each algorithm are as follows:

1. The DTA algorithm is first introduced in order to solve supervised spike-based learning problems where the input and target output spike sequences are both specified by the learning problem. The algorithm trains spiking neurons to reproduce the target output sequence when the neuron is presented with the corresponding input.

2. The DTA-B algorithm is then introduced in order to solve supervised learning problems where only the input sequences and a target spike count are specified by the learning problem. It is up to the algorithm to determine appropriate timings for the learned output spikes, in addition to the problem of updating the synaptic weights in order to generate said spikes.

3. The DTA-C algorithm is then introduced in order to solve unsupervised feature extraction tasks for image data. The algorithm must train a number of convolutional maps to recognise salient features in the training images.

All three learning algorithms proposed here achieve their learning objectives through a novel learning mechanism, which combines the computational characteristics of two different families of training methods in the literature: iterative methods based on the classical Widrow-Hoff learning rule (Ponulak and Kasiński 2010; Yu et al. 2013; Mohemmed et al. 2012; Gardner and Grüning 2016) and single-batch neural network optimisation methods (Tapson and van Schaik 2013; Tapson et al. 2013; Lee, Kukreja and Thakor 2016; Boucher-Routhier, Zhang and Thivierge 2021). Our algorithms implement an iterative learning regime wherein the SNNs incrementally improve over a number of update steps, however each update is computed by a Constraint Programming solver step. In exchange for higher computational requirements during each learning step, the number of learning iterations is greatly reduced in our method, and we observe an overall reduction in runtime when compared to standard iterative learning approaches.

Through this approach, we demonstrate that the solution weights of SNNs in a variety of supervised and unsupervised learning tasks can be solved to a reasonable performance while only requiring a relatively small number of training data presentations. This has implications on the efficiency of learning on traditional hardware platforms, since the computational requirements of neural network simulation are greatly reduced when compared to traditional iterative learning approaches which typically require many repeated presentations of the data to converge. On the ubiquitous MNIST classification task, our approach demonstrates the capability to achieve good convergence and performance in under one complete presentation of the dataset, which is a marked advantage over single-batch optimisation methods where exactly one dataset presentation is used for learning.

Traditional single-batch neural network optimisation methods, such as Extreme Learning Machines (Eliasmith and Anderson 2003; Huang, Zhu and Siew 2006), compute solution weights for the whole training set in a single step and are fundamentally limited by memory resources. While other single-batch incremental optimisation methods do exist in the literature (Tapson and van Schaik 2013; Liang et al. 2006; Widrow et al. 2013), these methods were primarily designed for ANNs or rate-based SNNs. In comparison, our learning algorithms are designed to incrementally solve three different SNN learning problems utilising full temporal coding. Additionally, such methods also typically utilise a randomised hidden layer of tens of thousands neurons, which is difficult to efficiently simulate on traditional architectures. We show how this hidden layer can be replaced with a much smaller spiking convolutional layer trained with a very small number of data samples, thus reducing the overall computational burden.

In summary, our novel approach aims to demonstrate how learning in SNNs can be carried out more efficiently by treating the weight adjustment problem as a constraint satisfaction problem. This improved efficiency is shown through reduced overall runtime, training data requirements, and smaller network size, all of which are important considerations for training spike-based models on traditional hardware architectures.

1.3 Thesis Outline

The rest of the thesis is organised as follows:

- In Chapter 2, an overview of the relevant background information is provided. Here, the computational dynamics and information coding of biological neurons are first examined, and the discussion then extends to analogue and spiking neurons. The construction of neural networks from individual neurons is also discussed, with illustrations of fully-connected and convolutional networks. Then, a review of the relevant supervised and unsupervised learning methods for SNNs are presented.

- Chapter 3 proposes the DTA algorithm for learning precisely timed spikes. First, a brief summary of the literature surrounding Constraint Satisfaction Problems (CSPs) is given, which is of fundamental importance to our method. Then, the process of converting a given problem to a set of constraints for CSP solving is introduced, and the design of the algorithm is discussed in detail. We then compare the memory capacity, learning accuracy, and convergence speed of the DTA algorithm with that of three other learning methods in the literature, using extensive numerical simulations and synthetic training data. The noise robustness, runtime, and effects of various hyper-parameters are also examined.

- Chapter 4 proposes the DTA-B algorithm for supervised learning tasks involving a target spike-count. The algorithm is compared against two other methods in the literature, using standard data classification benchmark problems. A hybrid SNN architecture is also used here to benchmark the algorithm on the MNIST dataset (LeCun and Cortes 2010). Additionally, we measure the maximal capacity of the algorithm to learn a large number of input classes with a single spiking neuron.

- Chapter 5 proposes the DTA-C algorithm for unsupervised feature extraction in SNNs containing convolutional layers. The proposed network architecture and algorithm are described in detail, and the approach is benchmarked on three standard image classification problems from the literature. The generalisation performances and effects of all hyper-parameters

are closely analysed, in order to highlight the strengths and weaknesses of the approach.

- Chapter 6 presents a discussion of the proposed constrained optimisation approach, providing an outline of the key contributions, current limitations, and possible future research directions pertaining to the above algorithms.

Chapter 2

Review of Spiking Neural Networks

This chapter provides a brief summary of the relevant biological background regarding neural network models, as well as the concepts and theories from Computational Neuroscience and Machine Learning that are used or expanded on in this thesis.

Section 2.1 introduces the computational dynamics and principles of biological neurons, and presents a discussion of how information is encoded in biological nervous systems. Section 2.2 provides an overview of traditional ANNs, starting with the definition of threshold-activated and continuous-activated neuron models. The discussion continues on to a review of neural network architectures, describing the common methods for building a network-level model. Section 2.3 presents a review of SNNs: firstly, three neuron models are introduced: the Integrate-and-Fire (IF) model, the Leaky Integrate-and-Fire (LIF) model, and the Spike Response Model (SRM). Secondly, an introduction is given to cover existing SNN learning methods in the literature which are relevant to the design of the algorithms proposed in the later research chapters.

2.1 Biological Background

The study of neural networks draws much inspiration from mathematical models developed in the field of computational neuroscience. This section presents a brief overview of the construction and neural dynamics of biological neurons, as well as how information can be represented in biological neural systems.

2.1.1 The Biological Neuron

Neurons are generally regarded as the main computational cells of the brain, whose computation is carried out using electrical and biochemical signals (Trappenberg 2009). There are several different types of neurons, however their basic function is simple: a neuron receives input signals from other cells, and if the inputs excite the cell enough, then the neuron emits an output action potential (commonly called a *spike*) which is propagated to other neurons to carry the signal forward. Spikes are broadly considered to be the fundamental currency of neural information processing and communication, because they can travel over large distances through the nervous system (Dayan and Abbott 2005). Commonly, the neuron sending spikes is called a *presynaptic neuron*, and the neuron receiving spikes is called a *postsynaptic neuron*.

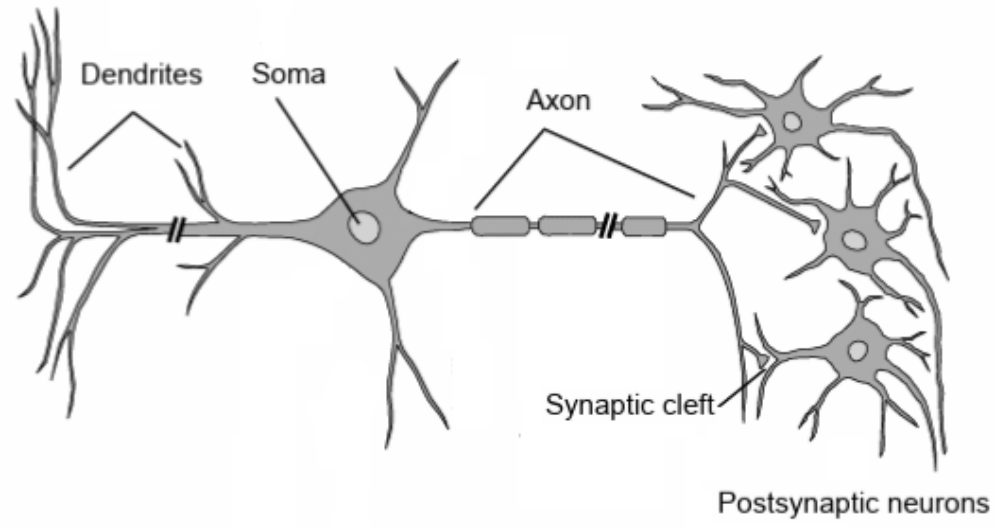

Figure 1: Model of a biological neuron, containing the soma, dendrites, and axon. Adapted from Figure 2.1 in Trappenberg (2009).

Cell Structure

The basic structure of a neuron can be generally divided into three components: the *dendrites*, a *soma*, and an *axon*. A stereotypical illustration of a neuron is given in Figure 1. Dendrites are branching tree-like structures which receive input signals from presynaptic cells. The soma is the main cell body, which contains the nucleus and other internal components which maintain the function of the cell. The axon is a nerve bundle serving as the output structure of the neuron. When a spike is generated at the soma, it travels down the entire length of the axon.

An important computational property of the neuron is called the *membrane potential*, which is the difference in electrical potential between the inside and outside of the cell. Input signals into a cell induce a change in the membrane potential of the cell, called a *postsynaptic potential* (PSP). The primary communication interfaces between the presynaptic axon and postsynaptic dendrites are structures called *synapses*. There are two types of synapses: a chemical synapse converts an incoming action potential into a PSP by the release of neurotransmitters in the presynaptic cell which are absorbed by the postsynaptic cell. An electrical synapse performs passive transmission of electrical voltage through intracellular gap junctions, which does not require an input action potential but does still induce a PSP (Debanne and Russier 2017).

Spike Generation

The ability of a neuron to vary its membrane potential by integrating PSPs is crucial to its computation. The membrane potential maintains a negative charge at rest, due to a greater concentration of negative ions within the cell. This equilibrium potential is maintained by a combination of active and passive ion channels on the membrane of the cell. Once the neuron is excited by a PSP, the membrane potential is depolarized, so that its charge becomes more positive. If the potential reaches a certain 'firing threshold', the process of generating a postsynaptic spike is triggered.

A spike is generated in three distinct sequential steps: firstly, voltage-activated ion channels become active and further depolarize the cell, which brings the membrane potential to a positive charge. Secondly, a period of rapid hyper-polarization

brings the membrane potential to be more negative than the resting voltage. While the membrane potential is hyper-polarized, it is said to be in a *refractory period* where it is almost impossible to generate another spike. Finally, the membrane potential gradually returns to the resting voltage state.

Both membrane potential integration and spike generation in biological neurons are temporal processes (an example is illustrated in Figure 2). As such, the PSP integration, depolarization, hyper-polarization, and the refractory period do not occur instantly, but over some periods of time. It is also important to note that spike generation has been observed to be *noisy*: the same input may not elicit the same spike train across different trials, and a neuron receiving no input may spontaneously spike (Gerstner et al. 2014).

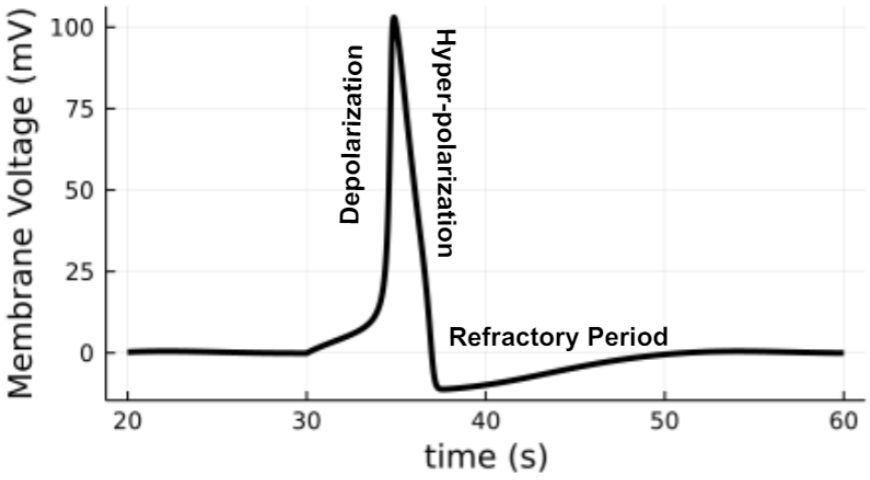

Figure 2: Generation of an action potential in a biological neuron. Here, the resting membrane potential of the neuron is zero.

Synaptic Strength

An important characteristic of neuronal communication is the *strength* of the synaptic connections between a pre- and post-synaptic neuron. The synaptic strength refers to the average amount of PSP that is produced by a presynaptic action potential, which can be regulated by a number of factors. For chemical synapses, this includes the number of dendritic connections between the presynaptic and postsynaptic cell, the number of synaptic sites that connect each axonal

and dendritic terminal, and also the density of neurotransmitter receptors on each synapse. In electrical synapses, the connection strength is dependent on the conductance of the gap junction, as well as the resistance of the cell membrane (Welzel and Schuster 2019).

The modification of synaptic connection strengths between neurons is called *synaptic plasticity*. Activity-dependent synaptic plasticity is believed to underlie experience-induced learning, behaviour modification, and memory formation in the brain (Citri and Malenka 2008; Ho, Lee and Martin 2011; Martin et al. 2000). Synaptic connections can be enhanced or depressed by plasticity, with such changes occurring over milliseconds to minutes (short-term plasticity), or hours to days (long-term plasticity) (Dan and Poo 2004; Citri and Malenka 2008). Extensive experimental evidences have been presented to support the existence of a wide variety of synaptic plasticity mechanisms, and it is known that most mammalian excitatory synapses simultaneously demonstrate multiple different forms of plasticity (Katz and Miledi 1968; Bliss and Lømo 1973; Rosahl et al. 1993; Zucker, Regehr et al. 2002; Whitlock et al. 2006).

2.1.2 Neural Information Coding

While it is generally accepted that the principal function of neurons is to process and communicate information, the question of *how* this information is represented in the brain (the 'neural code') remains a challenging topic without clear consensus. The majority of computational neural models assume that action potentials are stereotyped, such that individual spikes do not vary significantly in shape or amplitude (Kandel et al. 2000; Dayan and Abbott 2005). Putting aside the correctness of this assumption (for alternative opinions see de Polavieja et al. (2005); Debanne, Bialowas and Rama (2013); Maley (2018); Zbili and Debanne (2019)), the main consequence is that there must be some other property of action potentials which is fundamental to information representation, such as the number of spike events or their timings. As a direct result, many spiking neuron models do not model the shape of spike events, but instead characterize them by the spike timing only.

Each neuron can be considered an information channel with limited capacity,

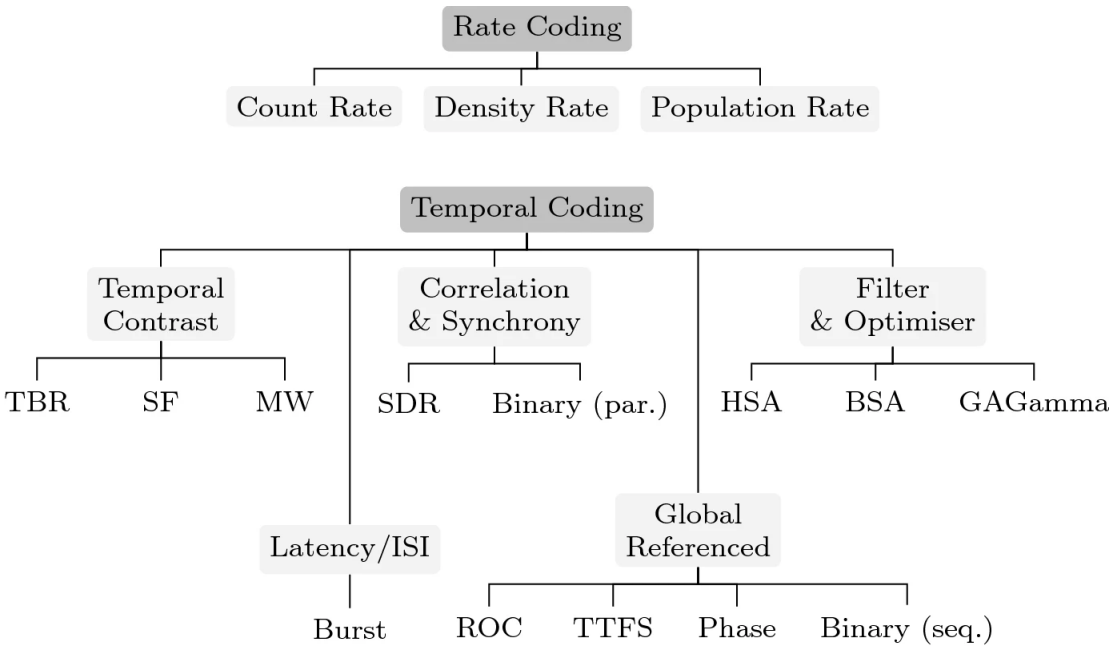

Figure 3: Neural coding schemes, categorised as either rate-coding or temporal-coding. Figure originally published in Auge et al. (2021). Abbreviated coding schemes are: Threshold-based Representation (TBR), Step-Forward (SF), Moving-Window (MW), Sparse Distributed Representation (SDR), Hough Spiker Algorithm (HSA), Ben's Spiker Algorithm (BSA), Rank Order Coding (ROC), Time-to-first Spike (TTFS). SDR, ROC, and TTFS coding schemes are discussed in further detail later in this chapter, and TTFS coding is used extensively in later research chapters.

and the relevant issues to neural coding include how much input information is encoded by a neuron (MacKay and McCulloch 1952; Panzeri et al. 1999; Zeldenrust et al. 2017), and also what information is encoded (Chichilnisky 2001; Paninski 2004; Schwartz et al. 2006). Research have proposed a variety of *neural coding schemes*, which can be broadly separated into two basic categories: rate-coding and temporal-coding (Figure 3).

Rate-Coding

In a rate coding scheme, the firing rate of the neuron is assumed to contain all of the information. In this view, the exact neuron spike timings are of little relevance. The idea of rate coding dates back to Adrian (1926), which shows a correlation between stimulus strength and the firing rate measured in the frog muscle. Similarly, Henry, Dreher and Bishop (1974) demonstrates correlations between the firing rate of cells in the cat visual cortex and the axis, orientation, and direction of moving lights. In London et al. (2010), *in vivo* measurements of the rat cortex suggests that there is substantial intrinsic noise in the spike generation process, which indicates that an average firing rate can represent information more reliably than the timing of spikes.

Rate coding schemes can be distinguished into several sub-categories:

- *Count rate* is the most common rate coding scheme, where the firing rate is defined simply as the number of spike events during a time unit of measurement T. Here, the timing of individual spikes bears no significance to information representation, as the same firing rate can be observed regardless of whether the spikes are evenly spread out from $0 \rightarrow T$, or are more closely clustered in time.

- *Density rate* measures the firing rate of a neuron over several independent simulations of the same input. All measurements are then summed and averaged over the number of repetitions. While this method is useful for analysis of neural activity, it is not a biologically plausible neural encoding approach. Consider the example given by Gerstner and Kistler (2002) of a frog attempting to catch a fly: the frog only has access to a single

computation of the fly's trajectory, instead of an average over multiple such instances.

- *Population rate* encodes information in the average firing rate of a population of (typically identical) neurons. Due to the noisy and probabilistic nature of spike generation, a neuron integrating rate-coded inputs require a longer temporal average duration T to achieve a more accurate evaluation. By averaging the output spike rate over many redundant neurons receiving the same input, population rate coding tackles the uncertainty introduced from using short integration windows.

Temporal Coding

In a temporal coding scheme, it is assumed that the timing of spikes play a crucial role in information representation (Thorpe, Delorme and Van Rullen 2001). The idea for temporal coding was first suggested by Thorpe (1990) as a fast-processing alternative to rate-coding. In Thorpe, Fize and Marlot (1996), experimental evidence suggested that the speed of the human visual system is too fast to be explained by the temporal averaging process in rate codes. A similar result was demonstrated by Johansson and Birznieks (2004) for primary sensory neurons on human fingertips, and it was also shown that the relative first spike timings are information-dense and fast enough to explain their experimental data. Additionally, it has been shown that for fast-fluctuating stimulus, temporal coding can be reliable and highly precise (Mainen and Sejnowski 1995).

Temporal-coding and rate-coding are inherently related: consider the example of an *instantaneous rate code*, which describes a spike train as a rapidly changing firing rate. A classical method to evaluate the instantaneous rate is to perform a low-pass filter over the output spike train using a smoothing function: if the spikes are close together, a larger instantaneous rate will be measured (French and Holden 1971; Pauluis and Baker 2000). In this approach, information is still described in terms of the spike rate, however the temporal density of the spike timings matter. As such, temporal codes and rate codes are not mutually exclusive, and temporal coding schemes simply consider that the spike timing carries additional information which is not captured by the average firing rate.

Compared to rate coding, temporal coding schemes are broader in definitions, ranging from codes which rely on only the first spike produced by a population of neurons, or the precise timings of individual events in a spike train. Several differentiations in temporal coding can be made as follows:

- *Time-to-first-spike* (TTFS) encodes the strength of a stimulus as the difference between the presynaptic signal onset time and the first postsynaptic spike time. Typically, a larger stimulus will result in an earlier spike time. This encoding scheme has been demonstrated in the salamander visual system (Gollisch and Meister 2008). In an idealised neural network setting, each encoding neuron is limited to a single spike per stimulus, and the encoding is realised by the linear relationship $t = T - T \times x$ where T is the encoding duration, x is some scalar-valued input normalised between 0 and 1, and t is the encoded spike time. Due to this linear implementation, TTFS encoding is also commonly referred to as *delay encoding* or *latency encoding*.

- *Rank order coding* (ROC) encodes information in the order with which multiple encoding neurons elicit their first spike. Unlike TTFS, a ROC code does not consider the precise timing of the spikes, and can be thought of as a discrete normalisation filter (Auge et al. 2021). As a result, certain information is lost when using ROC coding, such as the precise distances between each pair of stimulus. Another consideration is whether to allow two neurons to have the same rank (Galluppi and Furber 2011).

- *Precisely timed spikes* encodes an input stimulus using the exact timings of all spikes generated by a neuron. This coding scheme has also been referred to as a *fully temporal code* (Grüning and Bohte 2014). The fully temporal code is a generalisation of TTFS and ROC coding, which can encode a very large number of unique spatio-temporal patterns of stimuli. Theoretically, it has been shown that a code containing multiple spikes can increase the diversity, richness, and capacity of information representation in temporal coding schemes (Borst and Theunissen 1999; Ponulak and Kasiński 2010). However, the added complexity of encoding with multiple spikes comes with corresponding challenges, and there is yet no general consensus to how fully temporal codes should be used in artificial neural networks.

- *Population-based temporal coding* schemes come from a group of theories which consider the *temporal coordination* of spikes across populations of neurons. An early example is Polychronous Groups, which is a special case of fully temporal coding where multiple groups of neurons elicit the same exact spike train synchronised in time (Izhikevich 2006). Another example are Sparse Distributed Representation coding, where information is encoded by a relatively small group of neurons at any given time (Olshausen and Field 2004).

Rate and temporal coding schemes provide different benefits and challenges. While temporal coding can explain the behaviour of fast-reacting neurons, rate-coding is highly robust against fluctuations and noise. It is expected that further explorations into neural coding techniques will assist future developments of artificial neural models.

2.2 Analogue Neural Networks

2.2.1 A Brief History

The history of abstract mathematical neuron models, and networks of such neurons, dates back to the McCulloch & Pitts neuron model (McCulloch and Pitts 1943). Years later, Rosenblatt (1958) proposed the Perceptron learning algorithm in order to modify the weights of a McCulloch & Pitts neuron to compute linear functions. These works were instrumental to the later studies of the field of Neural Networks.

Generally, computational models of neurons used for building ML applications will consist of a number of input channels (a mathematical analogue of dendrites), an internal state variable (membrane potential), and an output channel (axon). Additionally, each input channel is associated with a *weight* variable, which represents the synaptic strength. The basic construction of a McCulloch & Pitts neuron, and all other neuron models discussed in this chapter, will follow the same high-level structure given above.

Formally, the computation of a McCulloch & Pitts neuron can be written as:

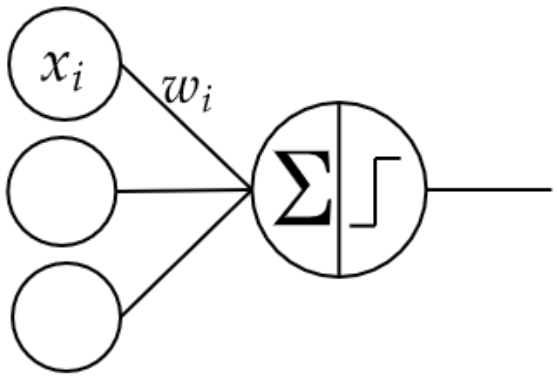

Figure 4: Model of a McCulloch & Pitts neuron. Inputs x_i arriving to channel i are weighted by w_i and summed, then thresholded to generate a binary output. The thresholding operation can be interpreted as applying a Heaviside step function as the activation function.

$$y = \Theta\left(\sum_{i=1}^{N} w_i x_i - b\right) \tag{1}$$

Here, the output y of the neuron when given N inputs is computed in two stages. Firstly, the input arriving to the i-th input channel x_i is multiplied by channel weights w_i, and summed with all the other weighted inputs. This weighted sum can be thought of as the internal state of the neuron. Secondly, the neuron state is passed to an *activation function*, in order to generate the output. In the case of the McCulloch & Pitts neuron, this activation takes the form of the Heaviside step function $\Theta(x)$:

$$\Theta(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

As such, the McCulloch & Pitts neuron is a *digital* computational node which produces a binary output. The *bias* term b is optionally added to the internal state, which has the effect of adjusting the activation threshold of the neuron.

The geometric interpretation of this digital neuron can be seen when Equation

1 is written in vector form:

$$y = \Theta(\mathbf{x} \cdot \mathbf{w}) \tag{2}$$

Here, $\mathbf{x}$ denotes the vector of inputs arriving to each channel, $\mathbf{w}$ denotes the vector of weights, and the $\cdot$ operator denotes the vector dot product. Under this interpretation, $\mathbf{w}$ defines a linear decision hyperplane in the N-dimensional input space. This splits the space into two, and thus an input vector $\mathbf{x}$ can be *classified* into either one of two possible classes, depending on which side of the hyperplane it is in. Note that here b is folded into $\mathbf{w}$ as an additional weight w_0, which has the effect of moving the hyperplane away from the origin by adding a constant value to the internal state of the neuron.

While a McCulloch & Pitts neuron is able to classify a set of linearly separable input vectors, it is unable to solve problems which are linearly non-separable. A famous example of this limitation is the Boolean Exclusive-Or (XOR) problem, which cannot be solved by a single Perceptron or a single-layer network of such neurons. However, multilayer feed-forward networks of Perceptrons have been shown to be capable of generating any Boolean functions, subject to certain constraints in the network size (Bishop et al. 1995).

2.2.2 Continuously-Activated Neuron Model

Unlike the McCulloch & Pitts neuron or Perceptron networks, modern ANNs typically compute highly non-linear functions. They achieve this by utilising *continously-activated neuron models*, which we refer to simply as *analogue neurons*. The basic computation of an analogue neuron is fundamentally similar to that of a McCulloch & Pitts neuron: first the inputs are weighted and summed, then the result is passed to an activation function to generate the output. However, the distinguishing factor is that analogue neurons utilise a *continuous* activation function, instead of a step function.

One of the earlier analogue models used in the literature is the sigmoidal neuron, in which the activation function takes the form of a logistic sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

The use of the continuous-valued activation function allows the sigmoidal neuron to compute function mappings between continuous input and output domains. Specifically, σ maps from the interval $(-\infty, \infty)$ to $(0, 1)$. It has been shown that, under very mild conditions, networks of sigmoidal neurons containing a single hidden layer can uniformly approximate any continuous functions (Cybenko 1992; Hornik 1991; Llanas, Lantarón and Sáinz 2008; Ma et al. 2020). This *universal approximation property* is one of the fundamental results driving modern ANN research.

However, interpreting the output of a sigmoidal neuron is not as straightforward compared to the Perceptron, with which the discrete output can be directly inferred as the predicted class label. Assuming a two-class scenario with a single neuron, the direct interpretation approach is to consider outputs less than 0.5 to represent one class prediction, and outputs at least 0.5 to represent the other. The more common approach is to interpret the outputs of multiple neurons as the probabilities of class membership. Each neuron is assigned one input class, and a larger neuron output indicates a higher probability of the input belonging to the designated class. To this end, the network is usually arranged such that its outputs approximate a Bayesian posterior distribution over the data (Bishop et al. 1995).

Formally, the state variable of a sigmoidal neuron is a weighted sum of its N input channels, with an added 'bias' term denoted as b. The state variable is then used in an activation function σ, in order to generate a real-valued output between 0 and 1:

$$y = \sigma\left(\sum_{i=1}^{N} w_i x_i + b\right) \tag{4}$$

Here, the weight vector $\mathbf{w}$ describes the steepness of the decision surface (if a direct interpretation is used for inference), and the bias b defines the position of the function.

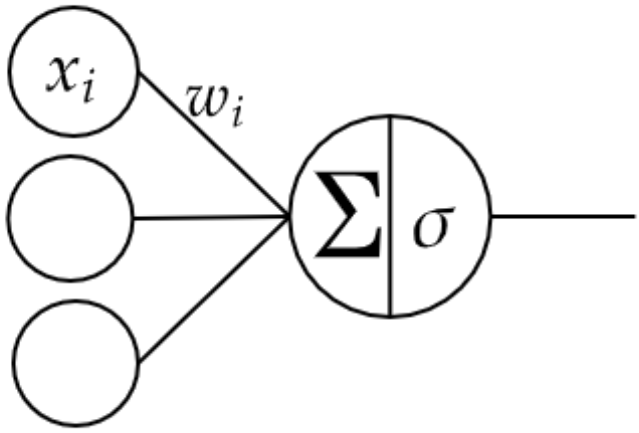

Figure 5: Model of a sigmoidal neuron. Inputs x_i are summed, then the sigmoidal function σ is applied as the transfer function, to generate a continuous output between 0 and 1.

2.2.3 Neural Network Design

Neurons are connected to each other to form a neural network, which enables the modelling of more complicated functions than achievable by a single neuron. Figure 6 illustrates some example decision boundaries generated by networks of Perceptrons, which becomes more complex as more neurons are added to the network (Bishop et al. 1995).

A typical modern ANN contains a number of *layers*. Each layer of a network contains neurons, which are arranged into some specified structure, for example a one-, two-, or three-dimensional grid. With some exceptions, most networks contain only two layers which are visible to the outside environment: the *input layer* which receives the input stimulus, and the *output layer* from which the computation of the network can be observed. Optionally, the network may also include a number of *hidden layers* between the input and output layers. The purpose of these hidden layers is to transform the input signal into some intermediate representations, through the weighted linear combination of inputs and the non-linear activation function.

Most authors will refer to the dimensionality of an ANN by the *height* and

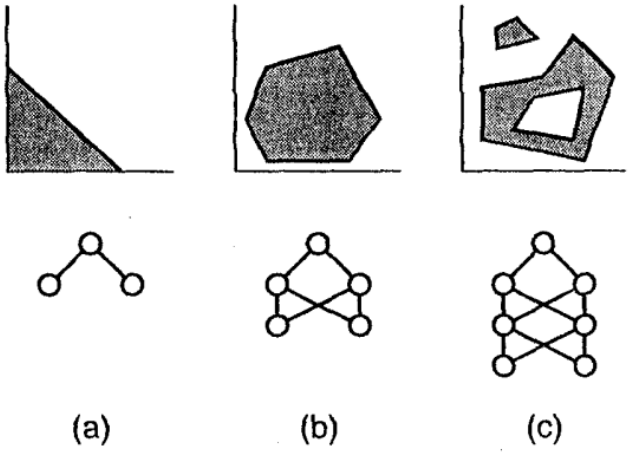

Figure 6: Decision boundaries generated by networks containing Perceptrons, for continuous two-dimensional inputs. **A**: A one-layer network with a single output neuron (bottom) generates a linear decision boundary (top). **B**: A two-layer network containing one hidden layer, generating a convex decision boundary. **C**: a three-layer network containing two hidden layers, generating non-convex and disjoint decision boundaries (Bishop et al. 1995).

the *width*. The height of a network denotes the number of layers in the model, which by convention excludes the input layer which only receives the input and does not perform any computation. The width of a model loosely refers to size of each layer: a network with a larger number of neurons in each layer is said to be wider.

Given the universal approximation property of neural networks with a single hidden layer, one may in principle expect to easily represent more complex functions by simply adding more hidden neurons to the architecture (Cybenko 1992; Hornik 1991). However, there are several practical issues with using shallow and wide neural networks, when compared to deep and narrow neural networks. One of the theoretical arguments is that some classes of functions cannot be approximated by shallow networks, or cannot do so without the network width growing exponentially (Bengio et al. 2009). Well-known examples of such problems include:

- stabilisation of non-linear control systems: achievable with two hidden layers but not one (Sontag 1991).

- inverse problems with discontinuous domains: achievable with two hidden layers but not one (Da Silva et al. 2017).
- radial functions on $\mathbb{R}^d$: achievable with one hidden layer but requires width which is exponential in d (Eldan and Shamir 2016).

An argument for using deeper architectures is the ability for networks with many hidden layers to represent the input signal at many levels of abstractions. Consider for example the task of interpreting an input image in Figure 7. In the first hidden layer, the image may be transformed into a collection of low-level features, such as information about the location of edges and corners. Neurons in the second layer could then combine the features into more complex shapes, and so on. In this manner, high-level abstractions in the data, such as MAN or SITTING, can be systematically and hierarchically constructed by combining lower-level features (Bengio et al. 2009). However, one of the main problems with deep networks is that they can be much more difficult to train. The most common method to train deep networks, gradient-descent with back-propagation, can encounter the problems of exploding, vanishing, or unstable gradient which means the earlier layers of the network improve very slowly compared to the subsequent layers (Nielsen 2015).

Another method to design larger and more complex neural networks is to use *parallel* architectures. Unlike a conventional network, where information flows in one central path, in a parallel neural network information may travel through several different paths simultaneously. There are mainly two possible methods to implement data-path parallelism in neural networks (Figure 8):

- Implement multiple identical neural networks, each receiving a different subset of the whole set of input signals. Some of the networks may become specialists for specific input features or input regions. The outputs of all networks must be combined in some subsequent layers (Littmann, Meyering and Ritter 1992; Yang and Peng 2017; Luo et al. 2021).
- Implement multiple non-identical neural networks, each receiving the full input signal. The hyper-parameter differences between the networks determine the desired computation for each information path. An example is a

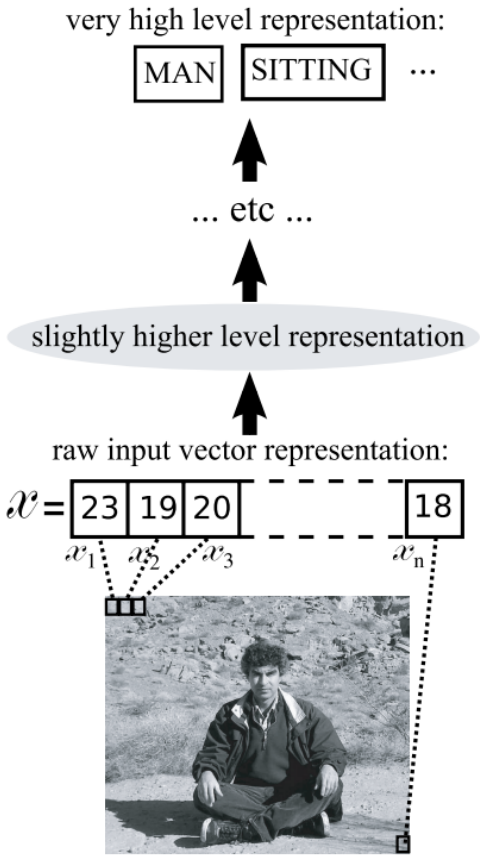

Figure 7: Different levels of representation of an input image (Bengio et al. 2009).

convolutional network with multiple parallel convolutional or pooling layers containing receptive fields of different sizes, which will detect input features of different scales (Wu et al. 2019).

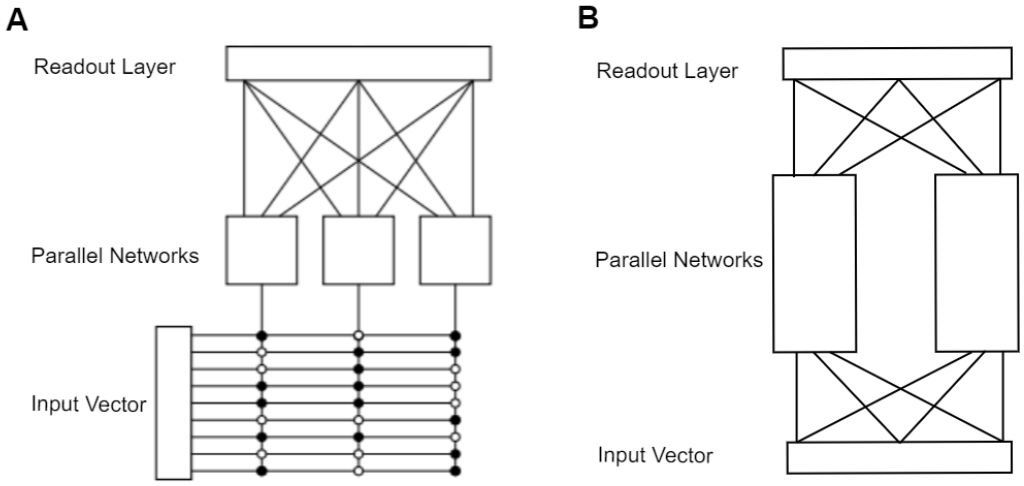

Figure 8: Parallel network architectures. **A**: A parallel network consisting of identical sub-networks which receives different parts of the input vector. Figure adapted from (Littmann, Meyering and Ritter 1992). **B**: A parallel network consisting of non-identical sub-networks, each receiving the same input vector.

2.2.4 Fully-Connected Feed-Forward Neural Networks

One of the most commonly used ANN architectures is a fully-connected neural network, which has become the workhorse of modern ML. One of the main advantages of a fully-connected network is that it is *structure agnostic* (Ramsundar and Zadeh 2018). This means that the fully-connected architecture makes no assumptions about the structure of the input, and so fully-connected ANNs are often the earliest models built, regardless of the application area.

A fully-connected ANN is characterised by the use of *fully-connected* layers. Here, each neuron in a layer propagates its output values to every neuron in the next layer. An example of a fully-connected ANN is illustrated in Figure 9. While networks of this type are very flexible, the full connectivity between each layer results in a very large number of network parameters (weights and biases),

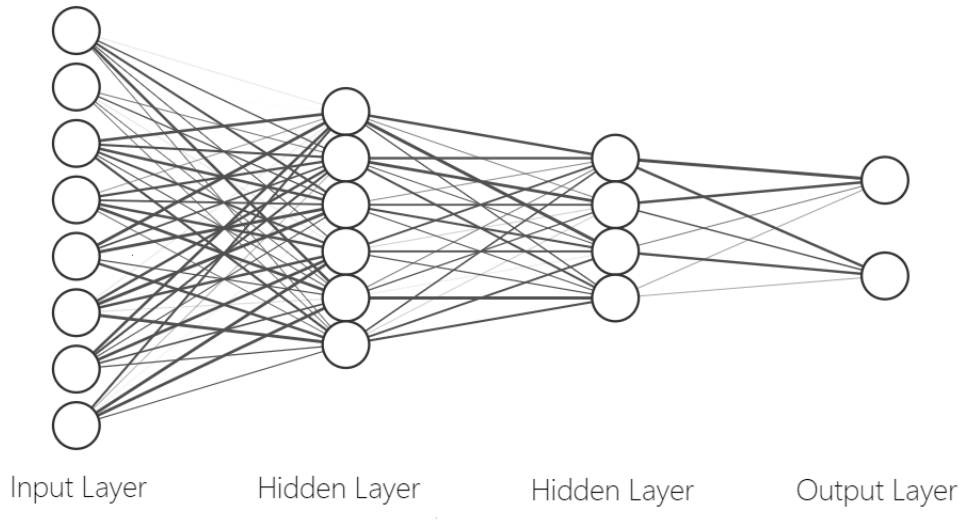

Figure 9: A fully-connected neural network with 4 layers, including the input layer.

which in turn requires significant computational effort in order to train. This is a significant drawback, and fully-connected neural networks are often outperformed by other network architectures which are specialised to exploit the structure of a given problem.

2.2.5 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have become the standard for ML problems involving pattern recognition. This is because CNNs perform a *convolution* operation in its hidden layers, which greatly reduces the number of trainable weights and biases when compared to a fully-connected network. In turn, this decreases the computational effort required to train CNNs, which allows more complex tasks and datasets to be solved (Bengio et al. 2009).

Historically, the convolution operation is widely used in the fields of mathematics and signal processing (Domínguez 2015). The basic idea is to *filter* a signal by applying a kernel function to it, and output a signal which is shifted by the kernel. Formally, the convolution operation $*$ can be written as the integral of the product of the original signal $f(t)$ and the kernel function $g(t)$:

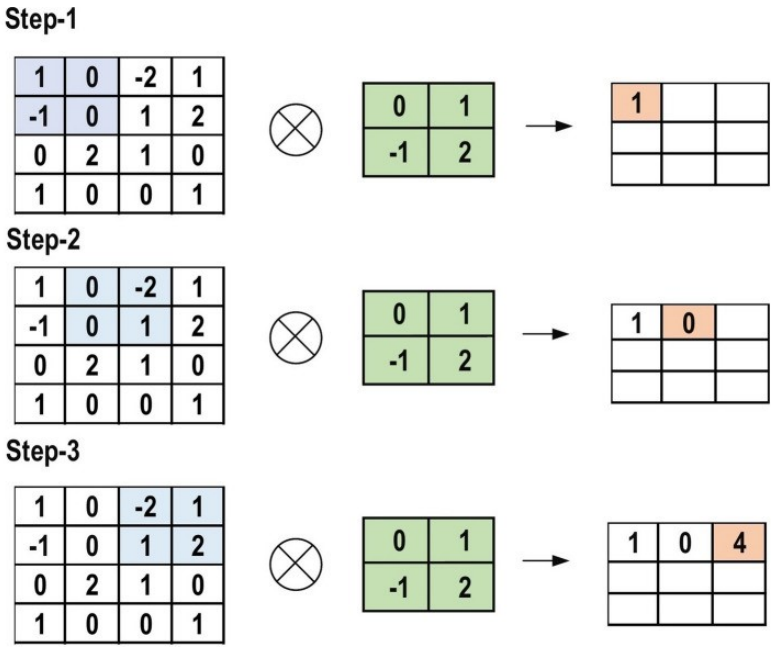

Figure 10: 2-dimensional convolution operation. **Left column**: input image, with each pixel encoded as an integer value. **Middle column**: the convolutional kernel being applied to the image. **Right column**: the resulting filtered convolutional map. Figure adapted from Alzubaidi et al. (2021).

$$(g * f)(t) = \int_{-\infty}^{\infty} g(\tau) f(t - \tau) d\tau \tag{5}$$

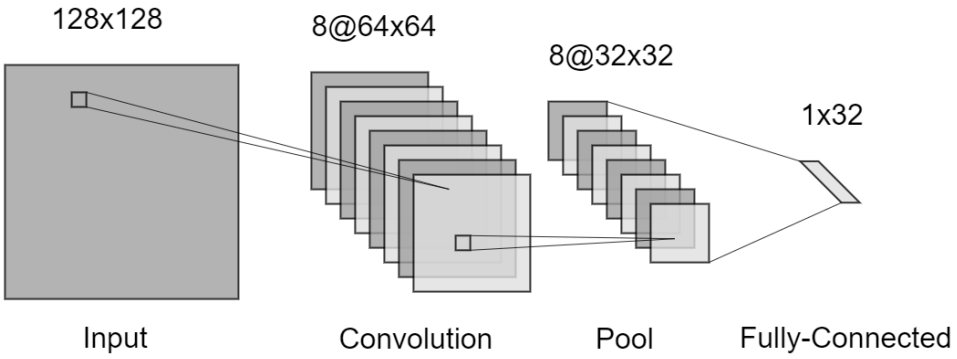

Figure 11: An example Convolutional Neural Network architecture.

The defining characteristic of CNNs is the use of *convolutional layers*. A convolutional layer contains a number of *convolutional maps*, wherein each map is the result generated by convolving a unique filter with the input image. Here, convolution is typically performed spatially in two dimensions: vertically and horizontally across an image. Standard examples of convolutions in image processing include the application of Ridge kernel functions to perform edge detection, or Gaussian kernels for image smoothing or blurring. An illustration of 2D convolution is represented in Figure 10, and an example convolutional layer with 8 maps can be seen in Figure 11.

Two central properties of convolutional layers is *local connectivity* and *weight-sharing*. Local connectivity signifies that each neuron in a convolutional layer is only connected to a specific portion of the neurons in the previous layer. The region of the input that a convolutional neuron is connected to is referred to as the *receptive field*. In CNNs, the receptive field is always the same size as the convolutional kernel of the current map. For example, a convolutional map with a kernel of size 5×5 means that each neuron in this map receives inputs from a 5×5 area of the input. Weight-sharing means that every neuron in a convolutional map shares the same input weights. Practically, one can think of the shared weights in a convolutional map as the convolutional kernel.

Formally, the convolution of kernel $g(t)$ over a two-dimensional image X can

be written as:

$$y(i,j) = \sum_{u=0}^{H} \sum_{v=0}^{W} X(i+u, j+v) g(k_i - u - 1, k_j - v - 1) \tag{6}$$

Here, $0 < i < H$ and $0 < j < W$ denote the vertical and horizontal positions of the image. H and W denote the height and width of the image, respectively. k_i and k_j denote the vertical and horizontal positions of the filter, respectively. Note that for some positions $i + u$ and $j + v$, these indices will be outside the domain of the image. One common approach is to assume that $X(i+u, j+v) = 0$ at these values, which is called *padding*. In addition, *stride* parameters s_i and s_j can be specified by substituting $X(s_i i + u, s_j j + v)$ into Equation 6, which allows the operation to skip a number of indices when generating the output map. More detailed tutorials for implementing convolution operations in CNNs can be found in (Hunsberger 2018; Alzubaidi et al. 2021).

Typically, a convolutional layer of a CNN will be immediately followed by a *pooling layer*. Pooling layers perform spatial sub-sampling of the convolutional maps, which serve the function of reducing the dimensionality of the information. They do this by combining the adjacent values in a convolutional map into a single value, most commonly by performing a maximum or averaging operation. Similarly to a convolution operation, a pooling operation has local connectivity, a pooling kernel size, and a stride parameter. Formally, a max-pooling operation can be written as:

$$y(i,j) = \max_{u,v=0}^{s_i, s_j} X(s_i i + u, s_j j + v) \tag{7}$$

It is important to note that pooling layers only perform a simple and predefined mathematical operation, and so they do not have trainable parameters. This is in contrast to convolutional layers, in which the parameters must be trained in order for the layer to learn useful transformations such as edge or corner detection.

2.3 Spiking Neural Networks

One of the main interpretations of analogue neuron models is that their continuous activation values can be viewed as an abstract representation of the average neuronal firing rate. As such, the computation of analogue neurons fall into the rate-coding category, and by extension feed-forward ANNs cannot easily process information by temporal-coding. By contrast, spiking neuron models communicate only by discrete, all-or-nothing action potentials encoded in time. This characteristic computation allows models of SNNs to receive and process information using some form of temporal-coding. However, the best coding scheme for any particular SNN architecture or application remains a matter of significant debate and ongoing research within the literature Grüning and Bohte (2014); Guo et al. (2021).

This section will review the spiking models and training algorithms in the SNN literature which are relevant to this current thesis.

2.3.1 A Brief History of Spiking Models

While spiking neuron models have existed for many decades (Hodgkin and Huxley 1952; Nagumo, Arimoto and Yoshizawa 1962), the use of SNNs in ML applications is a relatively recent development compared to ANN research. The earliest examples of neuronal modelling were used in the field of neuroscience to accurately capture the biochemical processes and complex dynamics of spike generation (Hodgkin and Huxley 1952; Koch and Segev 1989; Sterratt et al. 2011). These models were highly detailed, and as such they were less analytically tractable and very expensive to simulate at scale. To date, many models of spiking neurons have been proposed, each with varying degrees of abstraction and biological realism (Figure 12). The majority of spiking models used in modern ML treat the various structures of a biological neuron (importantly, the dendrites, soma, and axon), as a single computational node. In general, computational spiking neuron models aim to only capture the temporal spiking dynamics of biological neurons, instead of the underlying biochemistry.

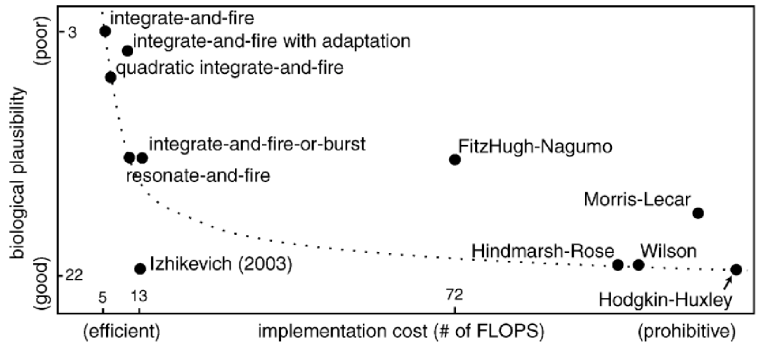

Figure 12: Model complexity vs simulation efficiency for various spiking neuron models. Figure adapted from Izhikevich (2004).

The Integrate-and-Fire model (IF) is one of the earliest spiking neuron models (Lapicque 1907). The basic computation is two-fold: firstly, at the time of an input spike arriving to the neuron, the membrane potential correspondingly increases or decreases. Secondly, if the membrane potential reaches a predefined and static spiking threshold, the neuron generates an output spike. Similarly to many modern spiking models, the IF neuron assumes that the shape of the output spike is not important, and thus an output spike is characterised only by the time of occurrence.

An IF neuron can be conceptualised as a capacitor C that stores charge over time. Formally, the membrane potential is written as:

$$C\frac{dV(t)}{dt} = I(t) \tag{8}$$

Here, $V(t)$ denotes the membrane potential of the neuron. $I(t)$ denotes an input electrical current, which can be an external current. Now, we only need to define the spiking mechanism by choosing a threshold ϑ. At some time t with which $V(t) = \vartheta$, the neuron generates a spike and $V(t)$ is immediately reset to a resting potential $V_{\text{rest}} = 0$. Some IF models may also choose to include an absolute refractory period, wherein immediately after spiking the membrane potential is held at V_{rest}, thus it is impossible to immediately spike again.

Since the membrane potential is not a perfect insulator, if there is no input then the membrane potential should decay over time. Combining this phenomenon

with the IF model yields the *Leaky Integrate-and-Fire* (LIF) neuron model, which is one of the most common models for building SNNs. The LIF neuronal dynamics are described by the following equation:

$$\tau_m \frac{dV(t)}{dt} = -V(t) + RI(t) \tag{9}$$

Here, R denotes the membrane resistance, and $\tau_m = RC$ denotes the membrane time constant. The LIF model has been instrumental to discovering some of the different properties of computation and processing in neural systems (Burkitt 2006). The model is sufficiently simple to be analytically tractable, and equally importantly, to be used for building medium to large-scale SNNs.

2.3.2 Spike Response Neuron Model

Experiments throughout this thesis will use the Spike Response Model (SRM), which is a generalisation of the LIF model. The SRM is best defined as a flexible framework which allows researchers to easily modify properties of the neuronal computation, without having to make major adjustments to the mathematical notation. As such, the model which is used here is obtained from mapping the LIF neuron to the SRM framework (see Gerstner et al. (2014)). One of the main differences between the LIF neuron and this version of the SRM model is that the SRM is formulated using kernel filters, instead of differential equations.

Typically, an input current induced by an action potential is modelled as an exponentially decaying kernel e^{-t/τ_s}, with synaptic time constant τ_s. Solving the differential equation of the LIF model yields the PSP kernel λ for a single input spike:

$$\lambda(t) = V_{\text{norm}} \left(e^{-t/\tau_m} - e^{-t/\tau_s} \right) \tag{10}$$

Here, τ_m denotes the membrane time constant. We also obtain an explicit expression for the membrane potential:

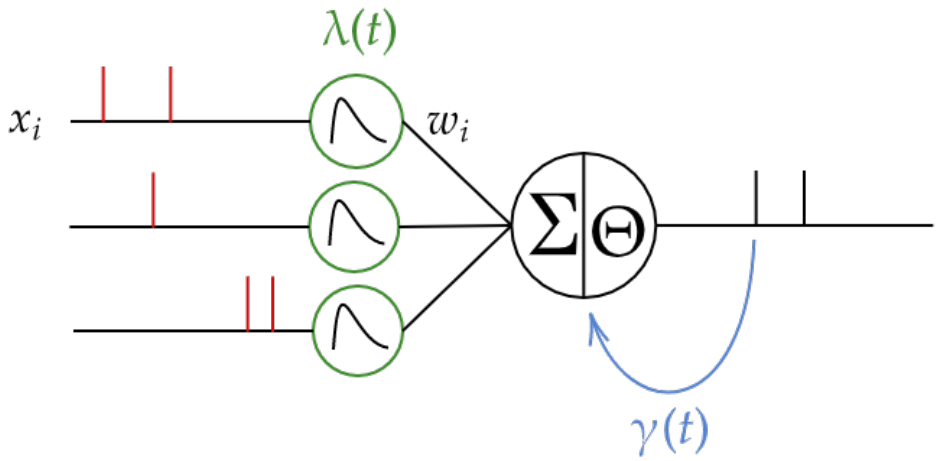

Figure 13: Model of a SRM neuron. Each sequence of input spikes x_i arriving to channel i is temporally convolved with a post-synaptic potential (PSP) kernel $\lambda(t)$ to generate a time-dependent PSP current. The sum weighted PSP currents of all input channels yields the membrane potential $V(t)$. If $V(t)$ crosses a certain threshold, an output spike is generated. Each output spike has a reset effect on the membrane potential via the reset kernel $\gamma(t)$.

$$V_0(t) = \sum_{i=1}^{N} w_i \sum_{t_j^i < t} \lambda\left(t - t_j^i\right) \tag{11}$$

Here, the input spike sequences arriving to a channel i is denoted as $x_i = [t_1^i, t_2^i, ...]$, with the spike times indexed by integer j. In experiments, we say that the sequences of spikes arriving to N channels form an *input pattern*, written as $\mathbf{x} = [x_1, x_2, ..., x_N]$. At a time t, if $V(t) = \vartheta$, then an output spike is generated at time $t^o = t$. The sequence of output spikes generated in response to $\mathbf{x}$ is denoted as $O = [t_1^o, t_2^o, ..., t_l^o]$, with the spike times indexed by l. In the SRM model, reset after each output spike is modelled by a kernel γ, as follows:

$$\gamma(t) = e^{\frac{t}{\tau_m}} \tag{12}$$

Note that the kernel functions $\lambda(t)$ and $\gamma(t)$ are zero for $t < 0$. The shapes of these functions are illustrated in Figure 14. Combining Equations 10, 11, and 12 yields the expression for the membrane potential *with reset*:

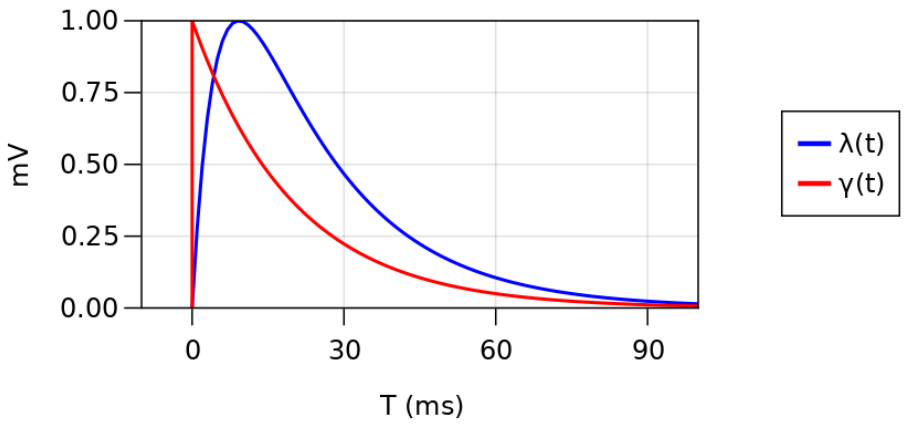

Figure 14: The shapes of the PSP kernel λ and the reset kernel γ in the neuron model, with $\tau_m = 20.0$ and $\tau_s = 5.0$.

$$V(t) = \sum_{i=1}^{N} w_i \sum_{t_j^i < t} \lambda\left(t - t_j^i\right) - \vartheta \sum_{t_l^o < t} \gamma(t - t_l^o) \tag{13}$$

2.3.3 Supervised Learning for SNNs

The goal of supervised learning using SNNs is as follows: given a set of fixed input patterns $X = [\mathbf{x}_1, \mathbf{x}_2, ...]$ and a set of desired outputs $Y = [y_1, y_2, ...]$ both indexed by integer p, compute a set of SNN weights with which the network output in response to $\mathbf{x}_p$ is as close to y_p as possible, for all indices p. Additionally, if the inputs are labelled according to a number of categories, then all inputs belonging to the same category should share the same desired output. While this objective is straightforward, in practice there are a number of complications.

The first difficulty is the choice of the target output y_p for each input pattern $\mathbf{x}_p$. As an example, we can consider one of the simplest applications of SNNs is to reproduce spike train data obtained from biological recordings, which is often called as the *spike-mapping* problem, or learning *Precisely-Timed Spikes* (PTS) (Memmesheimer et al. 2014). In this scenario, the target output is defined as a sequence of desired spike times $y_p = [t_1^{\mathrm{d}}, t_2^{\mathrm{d}}, ...]$, and no decisions have to be made about the computation of the SNN. One challenge of the PTS learning approach

is that the difference between desired and actual output spike trains must be computed, which is a signal processing problem (Victor 2005).

In general, there are currently two main approaches to spike-based classification, as follows:

1. In the first approach: for each input category index c, define a target output sequence $y_c = [t^{\mathrm{d}}_{c,1}, t^{\mathrm{d}}_{c,2}, ...]$. The training problem now becomes a special case of the PTS learning task, wherein input patterns belonging to the same category are assigned to the same target spike sequence. In earlier works, the target spike trains y_c contain only one spike time (Bohte, Kok and La Poutré 2000; Bohte, Kok and La Poutre 2002; Schrauwen and Van Campenhout 2004; McKennoch, Liu and Bushnell 2006), which can result in the network being susceptible to noise (Shrestha and Song 2017). More recent methods have defined target spike trains containing multiple spikes, which is more noise robust but at the cost of higher learning difficulty (Ponulak and Kasiński 2010; Florian 2012; Yu et al. 2013; Sporea and Grüning 2013; Gardner and Grüning 2016). An important limitation with this classification approach is that it is unclear how one should choose the spike times in each target sequence. It is possible to choose the sequence arbitrarily or randomly, however it is clear that due to the temporal nature of SNN computation, if a desired output spike occurs before the first input spike of any pattern, then the problem has no solutions.

2. In the second approach: for each input category index c, define a target *number of output spikes* $|y_c|$, where $|x|$ denotes the cardinality of x. The exact target spike timings are not specified. This approach can be referred to as *Spike Count Learning* (SC). The learning problem is now not only to replicate the target spike train, but also to find an appropriate target sequence for each input pattern, such that the number of spikes matches $|y_c|$. This approach is more robust than classification with PTS, since the learning can automatically find appropriate spike times for which there is a solution to the problem Gütig and Sompolinsky (2006); Gutig (2016); Shrestha and Orchard (2018). However, this requirement also adds significant difficulty and complexity to the design of the learning algorithm.

The remainder of this section will introduce several SNN learning algorithms which are directly relevant to the design of novel learning algorithms in subsequent research chapters.

Finite-Precision Learning Algorithm

The Finite-Precision (FP) learning algorithm was introduced by Memmesheimer et al. (2014) in order to train single SRM neurons to solve PTS learning problems with multiple target output spikes. In this algorithm, successful learning is characterised by each output spike occurring within a tolerance window around a desired spike time, set by hyper-parameter ϵ. The authors propose that there are three kinds of errors which may occur during learning:

1. An actual output spike time t^{o} occurs outside of a tolerance window.
2. A tolerance window does not have an output spike occurring within it.
3. A tolerance window has more than one output spikes occurring within it.

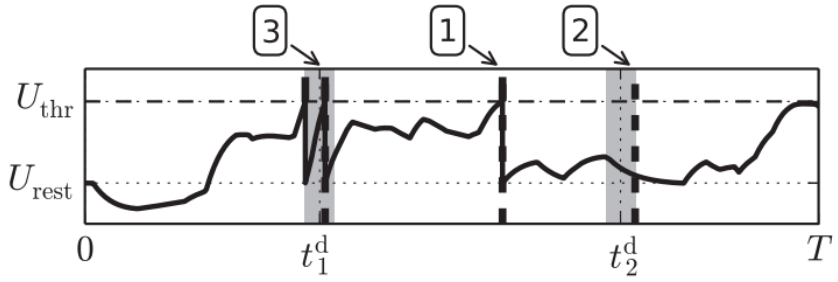

Figure 15: Illustration of the three types of errors in the FP learning algorithm. Gray areas are tolerance windows around desired spike times. Figure adapted from Memmesheimer et al. (2014).

In order to arrive at a solution without any of the above errors, the FP learning algorithm performs a number of iterative updates to the synaptic weights $\mathbf{w} = [w_1, w_2, ..., w_N]$ of the form:

$$\Delta w_i = \pm \eta \lambda(t_{\text{error}}) \tag{14}$$

Here, $\lambda(t)$ is the PSP kernel (Equation 10), η denotes a positive learning rate parameter, and t_{error} is the time of the *earliest* error of the three types above. If the error is of type (1), t_{error} is the time of the erroneous spike. If the error is of type (2), t_{error} is set as the end of the tolerance window $t^{\text{d}} + \epsilon$. If the error is of type (3), then t_{error} is set as the time of the extraneous spike (the second spike within the window).

Importantly, the decision to only include the time of the first error in the weight update means that the algorithm is not utilising any of the information about subsequent erroneous spikes. The authors state that this is done to "avoid nonlinear accumulation of errors due to interaction between output spikes" (Memmesheimer et al. 2014). This phenomenon of interacting output spikes will henceforth be referred to as *learning interference*, and has been summarised as follows: "when one of the actual output spikes is learning to be close to the desired one, weight updating inevitably changes not only the firing time of the current spike but also the firing times of the other spikes" (Xu et al. 2013). Despite the relatively simplistic weight adjustment approach, the FP algorithm was an important learning rule that was able to demonstrate very high learning capabilities in the PTS learning task.

Algorithms Derived from the Widrow-Hoff Rule

There exists a family of well-known SNN supervised learning algorithms for solving multi-spike PTS learning tasks, which share significant similarities in their synaptic weight update rules. Representative examples include the Remote Supervised Method (ReSuMe) (Ponulak and Kasiński 2010), the Spike Pattern Association Neuron (SPAN) (Florian 2012), the Precise-Spike Driven method (PSD) (Yu et al. 2013), the Instantaneous-Error (INST) rule and the Filtered-Error (FILT) rule (Gardner and Grüning 2016). There are some differences in notation and exact derivation method, however the majority of these learning rules can be said to be derived from the classical Widrow-Hoff learning rule, more commonly known in literature as the Delta rule for gradient-descent (Widrow and Lehr 1990):

$$\Delta w_i = \eta x_i (y^{\text{d}} - y^{\text{o}}) \tag{15}$$

However, the Delta rule was designed for ANNs where the input x_i, target output y^{d} and actual output y^{o} are all real-valued. For application to SNN learning, the rule must be adapted to process spikes. Here, we follow the derivation set out by Florian (2012). Specifically, this approach models the spike trains as a sum of Dirac delta functions of the form:

$$x(t) = \sum_{t_j^i} \delta(t - t_j^i) \tag{16}$$

$$y^{\mathrm{d}}(t) = \sum_{t_l^{\mathrm{d}}} \delta(t - t_l^{\mathrm{d}}) \tag{17}$$

$$y^{\mathrm{o}}(t) = \sum_{t_k^{\mathrm{o}}} \delta(t - t_l^{\mathrm{o}}) \tag{18}$$

However, the products of Dirac delta functions are mathematically problematic for gradient computations. To resolve this difficulty, several methods (Florian 2012; Yu et al. 2013) have proposed to convert the above definitions to continuous functions using the convolution operation:

$$\tilde{x}(t) = \kappa(t) * x(t) = \sum_{t_j^i} \kappa(t - t_j^i) \tag{19}$$

$$\tilde{y}^{\mathrm{d}}(t) = \kappa(t) * y^{\mathrm{d}}(t) = \sum_{t_l^{\mathrm{d}}} \kappa(t - t_l^{\mathrm{d}}) \tag{20}$$

$$\tilde{y}^{\mathrm{o}}(t) = \kappa(t) * y^{\mathrm{o}}(t) = \sum_{t_k^{\mathrm{o}}} \kappa(t - t_k^{\mathrm{o}}) \tag{21}$$

Here, $\kappa(t)$ denotes a kernel function. With these definitions, a spiking interpretation of the Delta learning rule is obtained:

$$\Delta w_i(t) = \eta \left(\tilde{y}^{\mathrm{d}}(t) - \tilde{y}^{\mathrm{o}}(t) \right) \tilde{x}(t) \tag{22}$$

Computing the integral of the above equation yields the batch version of the learning rule:

$$\Delta w_i = \eta \left(\sum_{t^{\mathrm{d}}} \sum_{t_i} \kappa(t^{\mathrm{d}} - t_i) - \sum_{t^{\mathrm{o}}} \sum_{t_i} \kappa(t^{\mathrm{o}} - t_i) \right) \tag{23}$$

In essence, the SPAN, ReSuMe, PSD, INST, and FILT rules all utilise variations of the above synaptic adjustment equation. There are some minor differences in each method, with the most important one being the choice of the kernel $\kappa(t)$. It is important to note that Equation 23 has only been proven to converge in the scenario where there is precisely one desired output spike and one actual input spike (Ponulak 2006), however algorithms of this type have reported successful learning in multi-spike settings. Additionally, these algorithms are susceptible to the problem of learning interference, as noted by (Gardner and Grüning 2016).

CONE Algorithm

The Convex-Optimised Synaptic Efficacies (CONE) algorithm is a single-batch supervised learning method for PTS problems which represents a different approach to SNN learning (Lee, Kukreja and Thakor 2016). Unlike the previously introduced algorithms, the CONE method is non-iterative. This means that the solution weights are evaluated using a single computation of the training data, instead of using iterative presentations of each sample.

The underlying idea of the CONE algorithm is to convert the PTS learning problem into a Constrained Optimisation Problem (COP), the solution of which can then be computed in one go by relying on established methods in the field of numerical optimisation. Note that Memmesheimer et al. (2014) previously proposed to map a PTS problem to a COP, which is very similar in principle. Generally, the COP is built using two types of constraints. The first type of constraint states that the membrane potential should be equal to the threshold at each desired spike time. The second type of constraint states that the threshold should be below threshold at every other time. Formally, for an input pattern $\mathbf{x}$ and output spike sequence y, this is written as:

$$V(t_l^{\mathrm{d}}) = \vartheta, \text{ for all } t_l^{\mathrm{d}} \in y \tag{24}$$

$$V(t) = \vartheta, \text{ for all } t \notin y \tag{25}$$

Equations 24 and 25 define a PTS task which is mapped to a Constrained Satisfaction Problem (CSP). By definition, a COP requires both a CSP formulation and an objective function to be optimised. In the CONE method, this objective takes the following form:

$$\min : \eta_1 \frac{||\mathbf{w}||_1}{||\mathbf{w}_{\max}||_1} + \eta_2 \frac{||\mathbf{w}||_2}{||\mathbf{w}_{\max}||_2} + \eta_3 \frac{B(\mathbf{w})}{B_{\max}} \tag{26}$$

The above objective function is minimising three terms, and η_1, η_2, and η_3 are regularisation parameters that tune the contribution of each term to the overall objective. $||\mathbf{w}||_1$ and $||\mathbf{w}||_2$ denote the $\mathcal{L}_1$ and $\mathcal{L}_2$ norms of the solution weight vector, respectively. Vector $\mathbf{w}_{\max}$ denotes the maximum value of the weight vector. Minimising the function $B(\mathbf{w})$ has the effect of increasing the membrane potential directly after a desired spike time:

$$B(\mathbf{w}) = \sum_{t_j^{\mathrm{d}} \in y} (V(t_j^{\mathrm{d}}) - \rho\vartheta)^2, \rho > 1 \tag{27}$$

In essence, the properties of the solution can be tuned by the objective terms. By adjusting the ratio of η_1 and η_2, the sparsity of the solution (whether few weights contribute to each spike or many weights) can be adjusted in order to obtain better overall noise robustness. By increasing the term η_3, the optimisation favours solutions in which the membrane potential continues to increase after the desired spike time, instead of just touching the spiking threshold then immediately decreasing. This allows the solution to become more robust to small downward noise perturbations caused by input jitter noise, missing spikes, or membrane potential noise.

In Lee, Kukreja and Thakor (2016), the above COP is optimised by using standard black-box solvers implementing the interior-point method. As a result, optimal solution weights which satisfy the constraints set out by the PTS can

be obtained without repeated presentation of the training data. The authors compare the CONE algorithm to a class of single-batch ANN algorithms which have been called *neural synthesis* methods (Tapson et al. 2013). Similarly to the CONE algorithm, neural synthesis methods also aim to solve for network weights in a non-iterative manner, however it is a notable difference that such methods are generally not capable of learning multiple precisely timed spikes (Eliasmith and Anderson 2003; Huang, Zhu and Siew 2006; Tapson and van Schaik 2013; Kulkarni and Rajendran 2018; Cohen et al. 2016, 2017; Boucher-Routhier, Zhang and Thivierge 2021). Learning rules which are capable of generating multiple spikes are desirable, because multi-spike neural coding schemes have been shown to have better computational properties compared to single-spike learning rules (Borst and Theunissen 1999; Ponulak and Kasiński 2010).

Tempotron and Multi-Spike Tempotron

The Tempotron and Multi-Spike Tempotron algorithms aim to solve SC learning tasks using gradient-descent (Gütig and Sompolinsky 2006; Gutig 2016). As such, both of these algorithms are capable of automatically finding appropriate spike times for an output neuron, given the desired number of spikes. In the case of the Tempotron, the algorithm is limited to either zero or one output spikes. As such, a single neuron trained with the Tempotron algorithm is limited to binary classification. In contrast, the Multi-Spike Tempotron has no upper limit on the number of target output spikes.

Each learning iteration of the Tempotron for binary classification is organised as follows. Firstly, the input pattern is simulated, and the resulting output (spike or no spike) is compared against the label. If the classification is incorrect, the algorithm performs two computation steps. In the first step, an update time t^* is identified: if the neuron elicits no spikes, then the update time is set as the time of the maximal subthreshold membrane voltage; if the neuron elicits a spike, then the update time is set as the output spike time. In the second step, a weight update is applied of the following form:

$$\Delta w_i = \pm \eta \lambda(t^*) \tag{28}$$

Here, the sign of the weight update depends on the learning scenario: positive updates to generate a desired spike, and negative updates to remove an erroneous spike. It is evident that the Tempotron algorithm is very similar to the FP algorithm described in the previous section. The important difference is the identification of the update time: for the Tempotron, the desired spike time is automatically set as the time the membrane potential is closest to the threshold. This mechanism for setting the update time t^* is what allows the Tempotron to automatically search for appropriate values of desired spike times.

The Multi-Spike Tempotron (MST) can be seen as a generalisation of the Tempotron learning rule to multiple spikes. One of the biggest differences is the suggestion of the Spike-Threshold-Surface (STS). The STS is a function which maps the spiking threshold value to the spike count generated by an SRM neuron. This models an inverse relationship: when the neuron is simulated by a lower threshold value than the static value ϑ, the spike count increases. Importantly, the STS is characterised by a sequence of critical threshold values $\vartheta^* = [\vartheta_1^*, \vartheta_2^*, ...]$, indexed by integer k. Each ϑ_k^* corresponds with the threshold value where the spike count jumps from $k-1$ to k. Formally, this is written as (Gutig 2016):

$$\vartheta_k^* = \sup\{\vartheta \in \mathbb{R} : \mathrm{STS}(\vartheta) = k\}, k \in \mathbb{N} \tag{29}$$

An illustration of the STS function is shown in Figure 16. The MST learning algorithm involves two steps: in the first step, the critical value for ϑ_k^* is found given a desired spike count $|y_c| = k$. In the second step, the neuronal weights are modified by gradient descent, which has the effect of adjusting the shape of the STS function, bringing ϑ closer to ϑ_k^* and thus allowing the neuron to generate the correct spike count. The full mathematical derivation of the weight update step can be found in the Supplementary Materials of Gutig (2016).

2.3.4 Unsupervised Learning for SNNs

The goal of unsupervised learning is relatively open-ended when compared to supervised learning: Given a set of inputs $X = [\mathbf{x}_1, \mathbf{x}_2, ...]$ without any labels, the learning should automatically uncover some structure or relationship hidden in

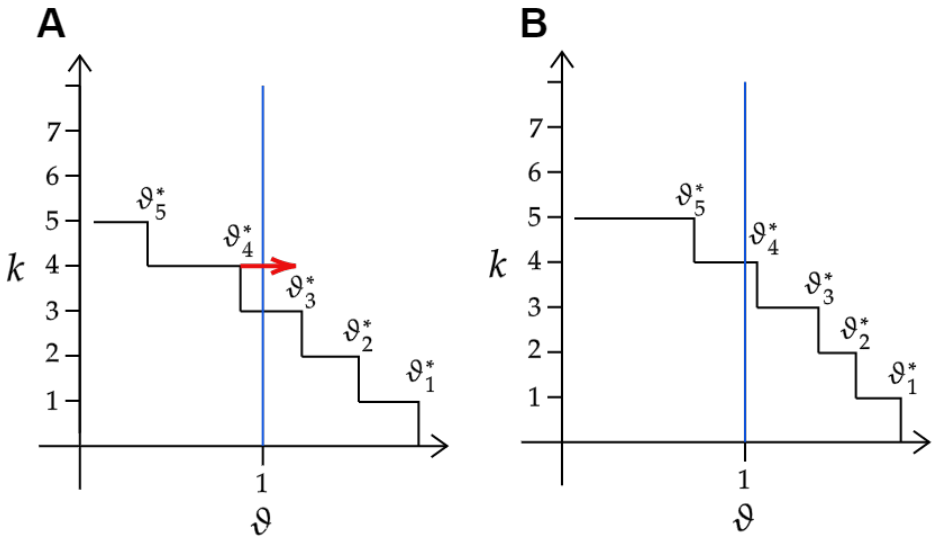

Figure 16: STS function of a LIF neuron given some fixed input pattern and random initial weights. **A**: before learning, the voltage threshold $\vartheta = 1$ corresponds with $k = 3$ output spikes. In order to increase k to 4, the voltage value corresponding to ϑ_4^* is moved closer to ϑ by gradient descent. **B**: after a number of learning updates, ϑ_4^* has crossed ϑ, which reflects that the neuron now elicits $k = 4$ spikes.

the data. In common unsupervised learning tasks, a neural network is typically tasked with learning a *transformation* of the input data into some *useful representation*. In problems such as clustering, the learned representation may be useful by themselves. In other applications such as feature extraction, the transformed input can then be used as input to other (often supervised) problems.

The vast majority of unsupervised learning algorithms for SNNs belong to the family of *Spike-Timing Dependent Plasticity* (STDP) methods. STDP has two main properties: firstly, synaptic plasticity only occurs based on locally available information. This means that a neuron does not adjust its weights based on the processing of some other neuron which it is not directly connected to. Secondly, STDP-based learning rules typically implement some form of *Hebbian learning* (Hebb 1949). In traditional Hebbian learning, the connection between two neurons is strengthened if the postsynaptic neuron fires shortly after the presynaptic neuron has spiked. Commonly, STDP rules may also contain an anti-Hebbian component, wherein the connection between two neurons is weakened if the presynaptic neuron fires after the postsynaptic spike. This form of STDP is often referred to as *pair-wise* and *asymmetric* (Figure 17).

Formally, the STDP function takes the form $\Delta w_i = \eta_+ e^{-t/\tau_+}$ if the relationship between the pre- and post-synaptic spike is $t = t_{\text{post}} - t_{\text{pre}}$ and $t > 0$. Here, η_+ is a magnitude adjustment parameter, and τ_+ is a time constant parameter. Similarly, for $t \leq 0$ the function is written as $\Delta w_i = -\eta_- e^{t/\tau_-}$. It is important to note that while STDP learning has been observed in biological nervous systems, there is evidence which suggests that the pair-wise function is not enough to give a full explanation of biological STDP (Morrison, Diesmann and Gerstner 2008; Bi and Wang 2002).

2.4 Chapter Summary

This chapter provides a review of the fundamental principles of ANNs and SNNs which are relevant to this thesis: beginning with an overview of biological neurons and neural coding, the definition and construction of networks of analogue neurons, and finally the relevant spiking neural models and learning algorithms.

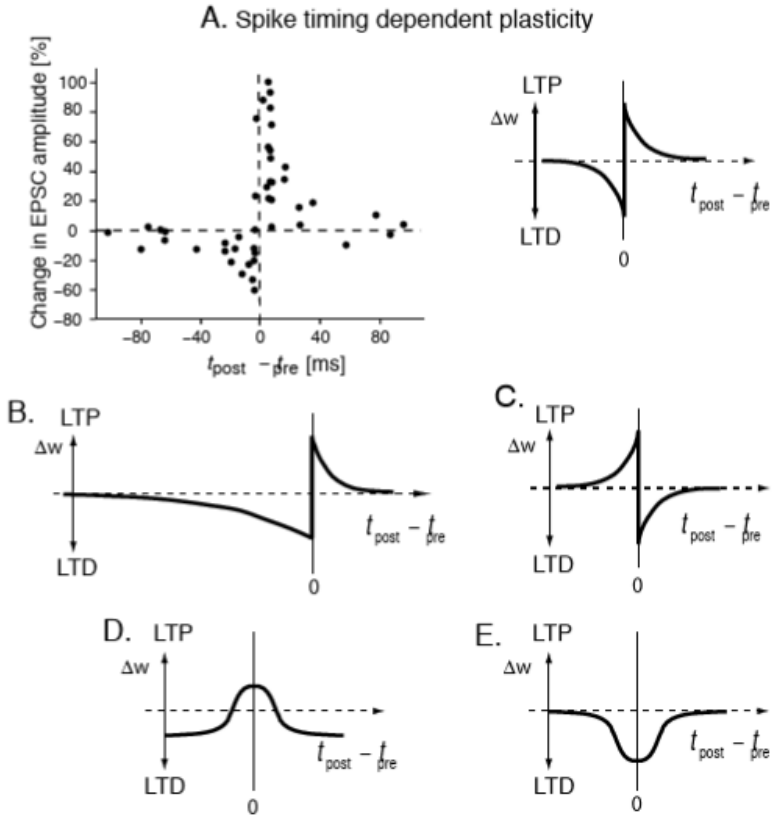

Figure 17: STDP mechanisms in the brain, replicated from Trappenberg (2009). **A**: The experimentally measured changes in excitatory postsynaptic current for various time differences between pre- and post-synaptic neurons. **B**-**C**: asymmetric STDP. **D**-**E**: symmetric STDP.

The overview of temporal coding schemes in Section 2.1 reveals clear advantages for temporal-coding over rate-coding. However, it is reasonable that in a biological nervous system, the type of coding depends on the application. For example, the components of the nervous system which are responsible for fast environment-sensing tasks should rely on temporal-coding, whereas other components which require high accuracy and robustness to noise should rely on average rate-coding. In a neural network, the specific coding scheme must be considered from the perspectives of both *input coding* and *output coding*. For example, a SNN model may be designed such that real-valued inputs into the model are transformed into spikes via ROC, whereas the output layer spikes are inferred by TTFS coding in order to obtain a classification. In practice, both the input and output coding methods must be considered by authors for each application, in order to obtain the best performance. However, such considerations are typically made by trial and error, instead of by theoretical metrics such as information density.

Section 2.2 outlines the computation of analogue neurons, and how such neurons are connected together to form neural networks. It is important to note that many of the structural principles behind designing ANNs also applies to SNNs. For example, Spiking Convolutional Neural Networks (SCNNs) are an important part of the current literature, which are beginning to show promising results in a number of application areas. Similarly, one could design a deep or parallel SNN in a similar manner to a deep or parallel ANN.

Section 2.3 reviews the computation of spiking neurons, as well as several learning algorithms which are important to the design of the learning rules introduced in this thesis. In particular, the computational characteristics of the CONE algorithm and the Widrow-Hoff-based learning rules will be combined in the design of the DTA algorithm (Chapter 3). The principles of the STS function, which provides the target spike times for a given spike count that corresponds with the minimum disturbance principle, is adapted for the DTA-B algorithm (Chapter 4). Finally, the computation of the DTA-C algorithm utilises competition-based learning commonly found in STDP algorithms.

Chapter 3

Linear Constrained Optimisation for Learning Precisely-Timed Spikes

3.1 Introduction

So far the typical approach to solving a PTS learning problem is to define a loss function as the difference between the desired and actual spike sequences, resolve the spike non-differentiability of the loss gradient by using a surrogate approximation, and then iteratively improve the SNN through gradient-descent (Zenke and Ganguli 2018; Gardner and Grüning 2016; Shrestha and Orchard 2018). Xu et al. (2013) outlined two major difficulties with the gradient-based approach in multi-spike mapping: firstly, it is unclear how to define the loss function when the number of actual and desired spikes are different. Recent works such as Shrestha and Orchard (2018) addressed this problem by utilising a continuous spike train distance measure as the loss function, for example the van Rossum distance (Rossum 2001). The second issue is the *learning interference* (LI) problem, which was introduced in Chapter 2. It has been suggested that learning interference accumulates with the number of output spikes, and for successful learning there should be sufficient distance between desired spike timings (Gardner and Grüning 2016). While it is currently unclear how LI can be completely solved during iterative

learning, several existing algorithms have shown the ability to overcome or reduce the effects of interference, which enables these methods to eventually converge to a solution (Memmesheimer et al. 2014; Gardner and Grüning 2016).

Geometric links between SNN learning and constrained optimisation problems have been made in Lee, Kukreja and Thakor (2016); Chou, Chung and Lu (2018); Mancoo, Keemink and Machens (2020). Accordingly, several methods have been proposed where the success of learning is defined by the ability of the SNN to satisfy a CSP, instead of minimising an explicit loss function (Memmesheimer et al. 2014; Lee, Kukreja and Thakor 2016; Luo et al. 2019). Out of these methods, the CONE algorithm has shown the capability to solve multi-spike PTS problems without requiring repeated presentations of the input (see Section 2.3.3). This approach has implications for the efficiency of learning, since the simulation requirements during training are greatly reduced in exchange for a more complex, one-batch approach to weight calculations.

Nonetheless, the CONE approach has a number of problems. Firstly, the method is designed for applications in discrete time, and is not applicable to continuous-time or event-based emulators. This approach is a challenge for learning situations which may require ultra-precise solutions, not only due to the discretisation but also because the number of inequality constraints scale directly with the number of discrete time 'bins'. Secondly, the method is not tested on any industry-standard datasets, which makes it difficult to perform comparisons. By extension, it is unclear what the exact learning capabilities of the algorithm are compared to known metrics (Memmesheimer et al. 2014). Thirdly, it is unclear how well the approach scales as the size of the problem increases. In particular, it is well known that other batch methods which compute solution weights non-incrementally can incur significant memory issues as the training dataset size increases (Tapson and van Schaik 2013).

3.1.1 Linear Programming with Constraints

Here, we give a brief overview of linear programming methods, since they are integral to solving CSPs. Linear programming is a subset of classical mathematical programming methods (coh 1978), in which the general form of the problem is

written as

$$\begin{aligned} &\text{maximise/minimise } Z(x_1, x_2, ..., x_n) \\ &\text{subject to } g_1(x_1, x_2, ..., x_n) = 0 \\ &\qquad\qquad g_2(x_1, x_2, ..., x_n) = 0 \\ &\qquad\qquad ... \\ &\qquad\qquad g_m(x_1, x_2, ..., x_n) = 0 \end{aligned}$$

Here, $Z()$ is the *objective function*, x are the n *decision variables*, and $g()$ are the m *constraints*. Linear programming constitute the special case where $Z()$ and $g()$ are all linear functions. Additionally, in a linear program the variables are often of some continuous domains, and the constraints may include inequalities as well as equalities.

The explanation above describes a *Constraint Optimisation Problem* (COP). Here, the problem is twofold: firstly, the decision variables have to be chosen such that the constraints are satisfied. Secondly, the variables must be chosen such that the objective is optimised. We note now that CSPs specifically only require the first part of the problem to be solved. That is, if a region of points in variable space is identified as feasible, then *any* point on this region is accepted as the solution, and correspondingly there may be an infinite number of feasible solutions. In a one must then traverse the feasible region to find a single optimal solution.

Figure 18 demonstrates an example CSP problem with two linear constraints, written as

$$\begin{aligned} 6x_1 + 3x_2 \leq 24 \\ 2x_1 + 3x_2 \leq 12 \end{aligned}$$

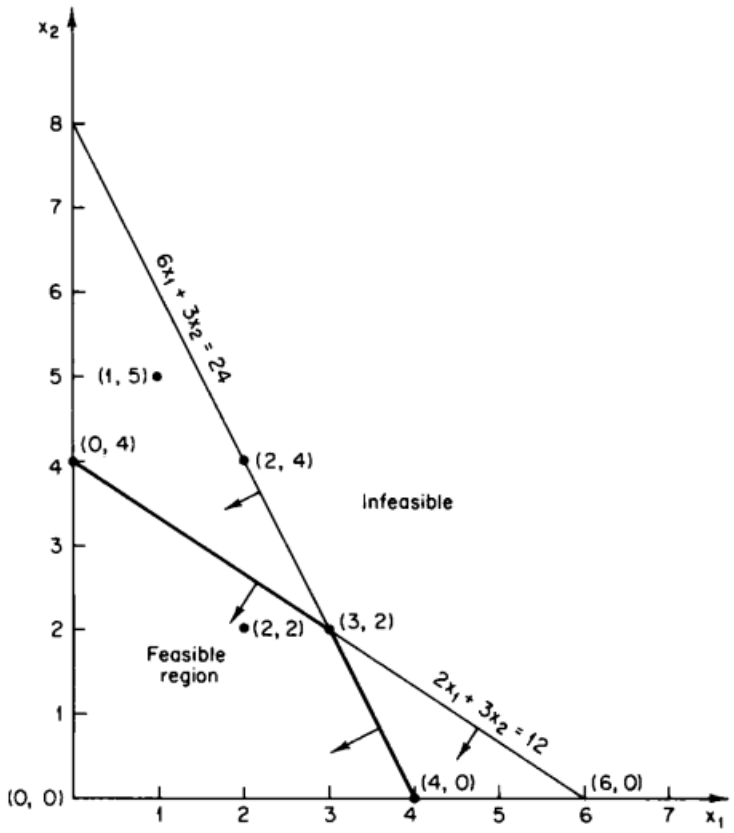

Figure 18: An example CSP problem. The two lines show the two linear constraints of the problem. When the constraints are inequalities ($\leq$), the feasible region lie inside of the thicker portions of the lines. Figure originally published in (coh 1978).

Additionally, we also have the domain constraints $x_1, x_2 \geq 0$, which is standard for linear programming problems. Here, we see that the feasible region in variable space forms a two-dimensional convex polytope, described by the set of vertices {(0, 0), (0, 4), (3, 2), (4, 0)}. Here, we obtain this feasible region by a graphical solution, however this is fundamentally limited to two dimensions due to the difficulty of graphing more than two decision variables. Beyond two dimensions, an industry-standard approach is the *simplex* method (Velinov and Gicev 2018). This involves setting some variables in the constraint to zero, and solving the constraints as a system of linear equations. For example, in the above constraints the vertex (0, 4) can be obtained by setting x_1 to zero and solving for x_2. Because of this operation, vertices of the feasible polytope is typically called the basic feasible solutions (Matoušek and Gärtner 2007). In the simplex method, the optimal feasible solution is then found by traversing the set of basic feasible solutions. In *interior-point* methods, optimisation is performed instead by traversing the interior of the feasible region (Tanneau, Anjos and Lodi 2021).

Further details of how these methods perform optimisation is beyond the scope of this thesis, as we will only encode the spiking problem as CSPs. Nonetheless, we will use an interior-point solver throughout the upcoming experiments. This is because various attempts were made at introducing objective functions into our formulation, which is discussed later on in Chapter 6. For use in CSPs, the linear programming solver will simply terminate once the first feasible solution is found.

3.2 Motivations & Chapter Layout

This chapter proposes a multi-spike learning algorithm to solve the PTS learning problem, which is called the Discrete Threshold Assumption (DTA) method. The algorithm is designed with a number of goals in mind:

1. Perform multi-spike learning in continuous time.
2. Combine computational characteristics of incremental learning and one-batch learning.
3. Overcome the learning interference problem.

4. Demonstrate robustness to input noise.

5. Minimise the simulation requirement during learning.

To this end, the DTA method uses the CONE algorithm as the basic inspiration. Each learning iteration of the DTA algorithm contains two steps: in the first step, the PTS problem is mapped to a CSP containing equality and inequality constraints. In the second step, the method utilises an industry-standard interior-point solver to compute solution weights. At this high level description, the DTA algorithm is very similar to CONE (Section 2.3.3). However, there are several characteristics that set the DTA method apart from the CONE algorithm. Firstly, our approach computes iterative updates to the solution weights in several solver steps, as opposed to directly computing the solution in one solver step as in Lee, Kukreja and Thakor (2016). This allows the neuron to learn one input pattern at a time, which has implications on memory complexity since the learning does not need to load the constraints from the entire dataset all at once. Secondly, the DTA method does not compute the weights directly, but instead introduces auxiliary optimisation variables which can then be used to calculate the weight adjustments. This approach can greatly reduce the number of optimisation variables in practical settings where there are fewer constraints than there are synaptic weights, thus further increasing the efficiency of the solver step. Thirdly, the DTA algorithm utilises a CSP without any objective function, however we introduce an expression for the weight update that allows tuning of the solution weight distribution.

The rest of the chapter is organised as follows. Section 3.3 describes the formulation of the DTA algorithm. The next four sections present various experiments and analyses in order to understand the learning capabilities and characteristics of the DTA algorithm in detail: Section 3.4 provides a learning demonstration of the algorithm, and measures the capacity of the algorithm to solve standard spike-mapping tasks with randomly generated input and output sequences. Section 3.5 studies the effects of several crucial parameters on the performance of the algorithm. Section 3.6 discusses the noise robustness of the algorithm, and shows that noise sensitivity can be improved by minor alterations to the weight update equation. Section 3.7 investigates the runtime of the algorithm in comparison to

existing methods in the literature. Finally, Section 3.8 analyses the computation of the algorithm with respect to the learning interference problem.

3.3 Method Description

The learning scenario is formalised as follows: given a fixed input pattern $\mathbf{x} = [x_1, x_2, ..., x_N]$ where $x_i = [t_1^i, t_2^i, ...]$ is the input spike sequence arriving to the i-th input channel of an SRM neuron, and a desired output spike sequence $\mathbf{d} = [t_1^{\mathrm{d}}, t_2^{\mathrm{d}}, ...]$, the algorithm computes a weight vector $\mathbf{w}$ such that the neuron generates an actual output sequence $\mathbf{o} = \mathbf{d}$ in response to $\mathbf{x}$.

The DTA algorithm solves this learning task in a number of iterative weight updates, where in each learning iteration the weight adjustment vector $\Delta\mathbf{w} = [\Delta w_1, \Delta w_2, ..., \Delta w_N]$ is computed by solving a Linear CSP. There are three components to the CSP: (1) the constraints, (2) the optimisation variables, and (3) the variable domains. Once all the components are defined, the CSP is solved using the Tulip interior-point solver (Tanneau, Anjos and Lodi 2021).

3.3.1 Problem Constraints

For simplicity, we assume here that the neuron is only learning the desired spike times for a single input pattern. At a given learning iteration, the initial neuron weights are denoted as $\mathbf{w}$. For a solution weight vector $\mathbf{w}^* = \mathbf{w} + \Delta\mathbf{w}$ to be feasible, the following equality and inequality constraints have to be satisfied:

$$\vartheta = V(t^{\mathrm{d}}), \text{ for all } t^{\mathrm{d}} \in \mathbf{d} \tag{30}$$

$$\vartheta > V(t), \text{ for all } t \notin \mathbf{d} \tag{31}$$

With the membrane potential $V(t)$ defined using the SRM equations (Equations 13, 10, and 12) as follows:

$$V(t) = \sum_{i=1}^{N} w_i^* \sum_{t_j^i < t} \lambda\left(t - t_j^i\right) - \vartheta \sum_{t_l^{\mathrm{d}} < t} \gamma(t - t_l^{\mathrm{d}})$$

Note that the neuronal reset times are fixed to the desired vector $\mathbf{d}$, which is mathematically equivalent to the High Threshold Projection method used in Memmesheimer et al. (2014), but only with notational difference. Optionally, the reset term $\gamma(t)$ can be moved from the membrane potential to the threshold. This does not change the computation of the neuron (Gerstner and Kistler 2002), however allows the constraints to be expressed as computationally efficient matrix multiplications.

In continuous time, there are clearly an infinite number of inequality constraints. However, Memmesheimer et al. (2014) showed that due to the strong temporal correlations in the membrane potential, the effective number of inequalities is finite. In the DTA method, we consider the only relevant inequalities to be the incorrect spike events of the current learning iteration, written as $\mathbf{a} = \{t : t \in \mathbf{o}, t \notin \mathbf{d}\}$. Additionally, we assume that no output spike can be learned with infinite precision, which is reasonable given that the system will be solved using numerical solvers. This implies $\mathbf{a} = \mathbf{o}$, and we can rewrite Equation 31 as:

$$\vartheta > V(t^{\mathrm{o}}), \text{ for all } t^{\mathrm{o}} \in \mathbf{o} \tag{32}$$

Equations 30 and 32 yield a CSP with $|\mathbf{o}| + |\mathbf{d}|$ linear constraints. Note that by this definition, the DTA method is not guaranteed a good solution in a single learning iteration, as threshold inequalities at any other times not in $\mathbf{o}$ are not enforced. However, any new inequality constraint violations that are a result of the current weight update will be considered in the next learning iteration.

3.3.2 Weight Update Method

The most common primal-dual interior-point methods exhibit a worst-case complexity which primarily scales with the number of optimisation variables (Wright 1997). As such, optimising Δw_i directly may scale unfavourably as N increases.

In addition, limiting the number of variables reduces the search space and in turn, the combinatorial complexity of the problem. To this end, we introduce new optimisation variables ζ, as well as a weight update equation as follows:

$$\Delta w_i = \sum_{l=1}^{|\mathbf{d}|} \zeta_l^{\mathrm{d}} \sum_{t^i \in x_i} \kappa(t_l^{\mathrm{d}} - t^i) + \sum_{k=1}^{|\mathbf{o}|} \zeta_k^{\mathrm{o}} \sum_{t^i \in x_i} \kappa(t_k^{\mathrm{o}} - t^i) \tag{33}$$

In essence, Equation 33 is a spiking interpretation of the Widrow-Hoff rule (Widrow and Lehr 1990), similarly to Equation 23. $\kappa(t)$ denotes a causal learning kernel, which for example can be the PSP kernel $\lambda(t)$ (Equation 10). Crucially, the learning rate variable in the Equation 23 has been replaced with unknown optimisation variables ζ_l^{d} and ζ_k^{o}. In general, increasing the value of ζ_l^{d} will also increase $V(t_l^{\mathrm{d}})$, and the reverse is also true.

The weight update in Equation 33 has several advantages. Firstly, in this form the CSP now contains the same number of variables as the number of constraints, which is more efficient when $N > |\mathbf{o}| + |\mathbf{d}|$. This is the case in many practical learning scenarios with a few dozen target spikes but hundreds or thousands of neuronal weights (Gardner and Grüning 2016; Gutig 2016; Xiao et al. 2019; Shrestha and Orchard 2018). Secondly, the form of weight update is controlled by the choice of $\kappa(t)$, which allows the user a degree of control over the weight distribution of the solution. This has a similar effect to the regularisation parameters in the CONE method (Equation 26), but without requiring an explicit objective function which needs to be evaluated numerically. As such, this fits well into the goal of the algorithm to minimise the simulation requirements of learning.

With the above constraints and variables, the resulting CSP can in theory be solved by a Linear Programming solver implementing any standard optimisation technique, such as simplex or interior-point methods. However, it is very important to include a fall-back strategy into the algorithm. This is because by introducing auxiliary optimisation variables ζ, the complexity and size of the search space have been reduced, but this also means the feasibility of a solution becomes partly dependent on the current spike times $\mathbf{o}$. This is because every additional spike in $\mathbf{o}$ introduces a new weight adjustment term in Equation 33, which increases the degrees of freedom the system has in search space. Consequently, this means that if a problem is infeasible, the current solution weights must be able

to continue exploring parameter space. In turn, this fall-back update allows the next learning iteration to perform optimisation with a different sequence $\mathbf{o}$ than the current, infeasible one.

To this end, if the solver identifies that the current problem is infeasible, then a fall-back weight update is applied with all $\zeta \leftarrow \eta$, where η is a learning rate parameter. Note that this fall-back update is equivalent to other Widrow-Hoff-based approaches (Section 2.3.3) and so is susceptible to the LI problem. However, our simulation results indicate that even for very difficult PTS problems, this fall-back is very rarely required (Section 3.4).

3.3.3 Domain Constraints

For stable optimisation, it can be helpful to specify domain constraints for the optimisation variables Wright (1997); Anjos and Burer (2008); Tanneau, Anjos and Lodi (2021). In our system, we find that without explicit domain bounds for ζ_l^{d}, the magnitude of weight adjustments can explode to very large values. Typically, this scenario occurs when equality constraints are satisfied from above, instead of from below. As such, the following domain constraint is added to the CSP for each ζ_l^{d} variable:

$$lb^{\mathrm{d}} \leq \zeta_l^{\mathrm{d}} \leq ub^{\mathrm{d}} \tag{34}$$

Here, lb^{d} and ub^{d} denotes the lower and upper bound parameters, respectively. The optimal values for lb^{d} and ub^{d}, as well as their effect on the overall stability of learning will be discussed in Section 3.5.

It is important to note that the ζ_k^{o} variables, corresponding to the weight adjustments at actual spike times, are unconstrained in this system. Because of temporal correlations in the membrane potential, an upper limit on ub^{d} effectively constrains the minimum values for lb^{o}. In our simulations, we find that including additional domain constraints for ζ_k^{o} increases the effort required for parameter optimisation without any significant benefits in terms of learning stability or runtime.

3.3.4 Algorithm Summary

A complete summary of the algorithm is given in Algorithm 1.

Algorithm 1: DTA Algorithm for learning Precisely Timed Spikes. The algorithm takes in a set of inputs $\mathbf{X}$ and target outputs $\mathbf{Y}$, and compute a synaptic weight vector $\mathbf{w}$ such that all the outputs of each input pattern $\mathbf{x}_p$ is the same as the target y_p.

Data: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_p\}, \mathbf{Y} = \{y_1, y_2, ..., y_p\}$

initialise neuron with weights w;
$accuracy \leftarrow 0.0$;
while *epoch* $<$ *maxEpoch or accuracy* $<$ *1.0* **do**
    shuffle X;
    foreach $\mathbf{x}_p$ *in* X **do**
        compute output $\mathbf{o}$;
        compute Δw (Eq. 33, 30 & 32);
        $w^* \leftarrow w + \Delta w$;
        $w \leftarrow w^*$;
    end
    $accuracy \leftarrow \frac{\sum_{p=1}^{P} \Theta(vRD(\mathbf{o},\mathbf{d}) - vRD_{\min})}{P}$ (Eq. 37 & 37);
end

3.4 Learning Performance

In Memmesheimer et al. (2014), it was suggested that when the input pattern and desired output sequence are randomly generated, there is an upper limit to the capacity of an SRM neuron to correctly implement the PTS problem. Specifically, as the input duration T increases, the probability that the problem is solvable decreases. Furthermore, the authors define a critical limit T_α where there is a 50% probability that a feasible solution exists. The authors establish a theoretical estimate of T_α as:

$$T_\alpha \approx \frac{N\sqrt{\tau_m \tau_s}}{\frac{-2\nu_{\text{out}}\sqrt{\tau_m\tau_s}}{1+\nu_{\text{out}}\sqrt{\tau_m\tau_s}} \log\left(\frac{\nu_{\text{out}}\sqrt{\tau_m\tau_s}}{1+\nu_{\text{out}}\sqrt{\tau_m\tau_s}}\right)} \tag{35}$$

Here, ν_{out} denotes the average spike rate of the randomly generated target output. This memory capacity limit provides a good benchmark for PTS learning performance, and the primary objective of this section is to demonstrate that the DTA learning rule is able to reach the limit defined by Equation 35.

3.4.1 Experimental Setup

The experimental setup of the random PTS task is as follows. The input pattern is created by drawing spike sequences of duration T for each of the $N = 500$ input channels from a homogeneous Poisson process with mean rate $\nu_{\text{in}} = 0.005$ for each of the $N = 500$ input channels. Desired output spike sequences are generated similarly with rate $\nu_{\text{out}} = 0.005$. Desired spikes are not allowed to occur earlier than the τ_m.

Learning Scenarios

There are two distinct experimental scenarios that we use for memory capacity measurements:

1. The neuron learns a single long input pattern of duration T. The duration starts at $T = 1000$, then increases in increments of 200 until less than half of all trials converge. The duration at the stopping point is taken as T_α.

2. The neuron learns multiple short input patterns of duration $T = 400$. The number of patterns starts at 1, then increases one at a time until less than half of all trials converge. The total combined duration of all patterns is taken as T_α.

According to Memmesheimer et al. (2014), an SRM neuron exhibits approximately the same PTS learning capacity (Equation 35) in either of the above learning scenarios. For each capacity measurement, 50 independent trials are run with different input and desired output sequences. For each measurement, the DTA algorithm is run for a maximum of 100 epochs. An epoch is the presentation of all the input patterns, which is split into a number of iterations wherein each iteration presents a single pattern at random.

Spike Distance Metric

In all experiments, convergence is decided using the van Rossum distance metric Rossum (2001); Schrauwen and Van Campenhout (2007); Paiva, Park and Principe (2009):

$$vRD(\mathbf{o},\mathbf{d}) = \sqrt{\sum_{i,j} e^{\frac{-|o_i-o_j|}{\tau_v}} + \sum_{i,j} e^{\frac{-|d_i-d_j|}{\tau_v}} - 2\sum_{i,j} e^{\frac{-|o_i-d_j|}{\tau_v}}} \tag{36}$$

Here, τ_v is the metric time constant which we set as $\tau_v = \tau_m$. vRD is a strictly positive distance metric, and $vRD(\mathbf{o},\mathbf{d}) = 0$ indicates $\mathbf{o} = \mathbf{d}$. Based on analyses presented in Rossum (2001); Satuvuori and Kreuz (2018), we set $\tau_v = 100$ which provides a good sensitivity to both short-term spike jitter and missing/additional spikes. This parameter tuning is necessary because we will be measuring the learning accuracy across a wide scale of the input duration T, and care must be taken not to let the distance value be dominated by very small differences between the actual and desired spike times.

It is practically impossible to achieve perfect convergence when the temporal dimension is continuous, due to limitations of computer precision during numerical optimisation. Hence, we must designate a parameter to denote 'sufficiently good' convergence for simulations. We choose this parameter to be the average distance between each pair of actual and desired output spikes, denoted as Δt_ϵ, and set $\Delta t_\epsilon = 1$ throughout this chapter. This means we consider an output sequence to be converged if each output spike is (on average) within 1 time unit of their desired output timing. This allows us to calculate a minimum distance value, denoted as $vRD_{\min}(T, t_\epsilon)$. Generally, if the average value of $vRD(\mathbf{o},\mathbf{d})$ over the whole training set is smaller than $vRD_{\min}(T, \Delta t_\epsilon)$, then we consider that the learning has converged.

However, the value of Equation 36 does not only rely on the average distance between pairs of target and desired spikes, but also the number of output spikes. In order to determine a formula for $vRD_{\min}(T, t_\epsilon)$, we perform a numerical experiment, as follows. First, a Poisson spike train of duration T is generated as the 'template'. Then, we apply a Gaussian spike jitter with mean zero and standard deviation t_ϵ to each spike in the generated sequence. We then compute the vRD

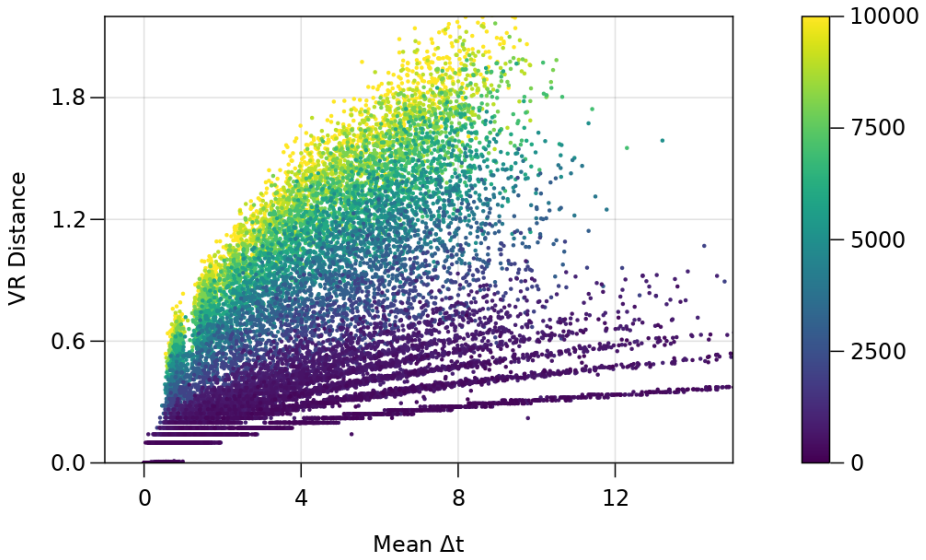

Figure 19: Dependence of van Rossum distance values on the mean Gaussian spike displacement (x-axis) and pattern duration T (colorbar) with filter time constant $\tau_v = 100$.

between the template and noisy spike trains, and record the resulting distance value. This was repeated many times while varying the value for T and t_ϵ. The result is a point cloud as shown in Figure 19. We then fit a Multiple Linear Regression model to the data, where the independent variables are T and Δt_ϵ, which yields the following relation:

$$vRD_{\min}(T, t_\epsilon) = 0.08t_\epsilon + 0.0001T \tag{37}$$

For the remainder of this chapter, all experiments utilise the minimum vRD value in Equation 37 to determine convergence.

Learning Demonstration

First, we run an initial experiment for demonstration purposes. The experimental setup is the random spike mapping task described above. An example is shown in Figure 20: on panel A, the neuron is trained on a single input pattern of duration $T = 2000$ with five desired output spikes. On panel B, the same input pattern

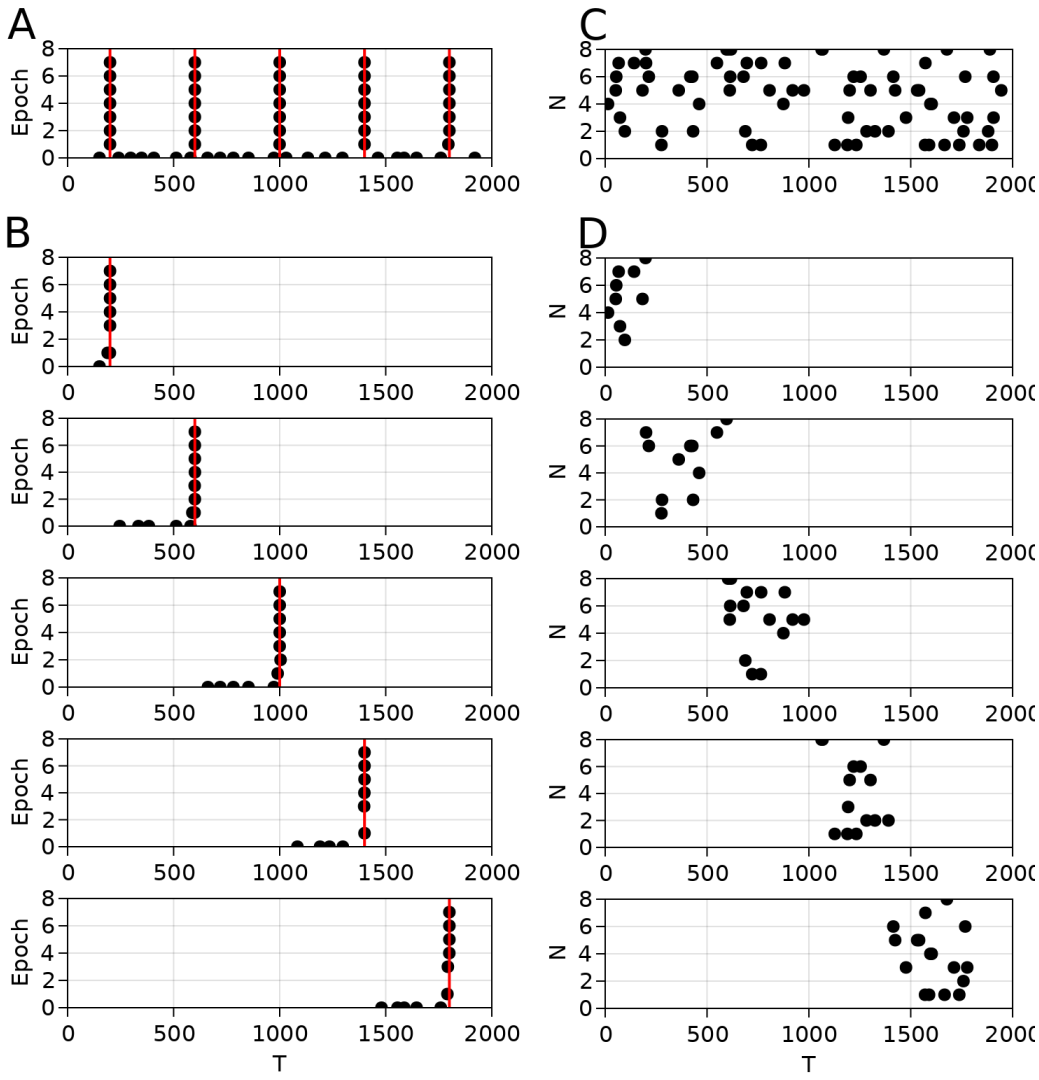

Figure 20: Output spikes throughout DTA learning in an example PTS task with randomly generated inputs. **A**: learning progress of a single long input pattern. **B**: The same long input pattern is split into five patterns of equal length, presented to the algorithm one at a time. Red lines are desired spike times. **C**-**D**: The input patterns presented to the learning neurons corresponding to **A** and **B**, respectively.

is now split into five patterns of length $T = 400$, which is passed one at a time to the DTA algorithm for learning. 100 independent trials are run with different realisations of the input pattern.

In the long input scenario, the DTA algorithm converges in 1.7 ± 0.09 epochs on average, with a mean runtime of 0.12 ± 0.04 seconds. In the short input pattern scenario, the algorithm converges in 3.1 ± 0.1 epochs on average, with a mean runtime of 0.18 ± 0.06 seconds. These results indicate that the short input scenario is generally more difficult to solve. This is reasonable, since the algorithm is presented with only a subset of the problem constraints at each learning iteration.

3.4.2 Simulation Results

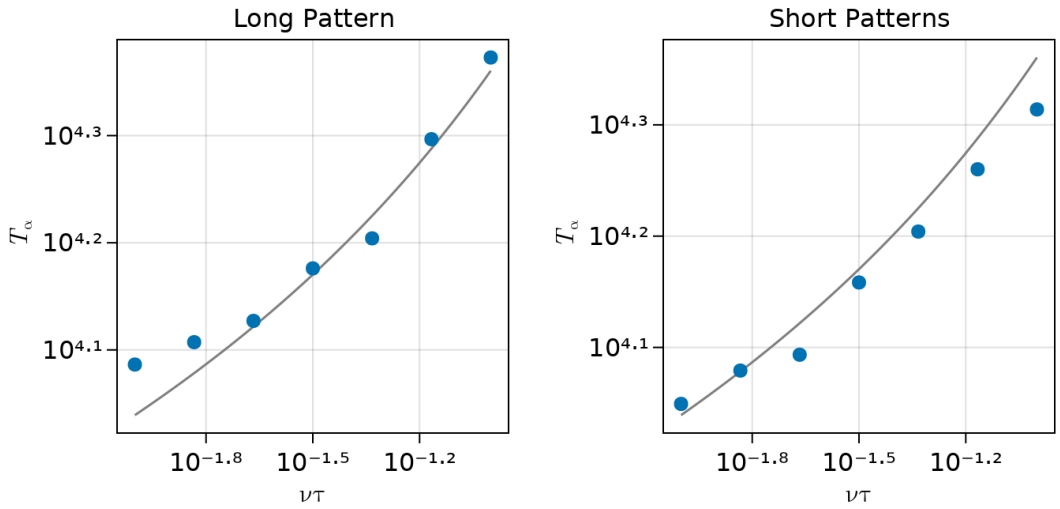

Figure 21: Memory capacity measurements (blue dots) of the DTA algorithm for $(lb^{\mathrm{d}}, ub^{\mathrm{d}}) = (0, 0.07)$ and $\kappa(t) = \lambda(t)$. (**Left**) Memory capacity in the long pattern learning scenario. (**Right**) Capacity in the short patterns scenario. Gray lines are theoretical bounds as computed by Equation 35.

Here, we measure the memory capacity of the DTA algorithm under both learning scenarios. Following from Memmesheimer et al. (2014), here we also vary $\tau = \sqrt{\tau_m \tau_s}$ as the independent variable, while keeping the ratio $\frac{\tau_m}{\tau_s} = 4$ constant in all measurements. In all trials, the DTA algorithm is trained for a maximum

of 100 epochs.

Results are shown in Figure 21. For both the long and short pattern scenarios, the measured capacity fits the theoretical bound reasonably well. Importantly, it is observed that the capacity in the short pattern scenario is worse than the long pattern scenario for every value of τ. This is expected, however increasing the maximum number of epochs to 200 did not significantly improve these results. Additionally, the memory capacity in the long pattern scenario is slightly higher than the theoretical bound for smaller values of $\nu_{\text{out}}\tau$ in our measurements. While the theoretical approximation is expected to diverge for very small values of $\nu_{\text{out}}\tau$, the capacity difference here is much larger than in the original work (Memmesheimer et al. 2014). A possible explanation for this result is the relatively small number of input channels, since the theoretical bound is most accurate in the very large N regime.

Importantly, the number of learning iterations where the solver returned an infeasible result was very low. Analysis of all of the learning trials at capacity indicates that the average number of infeasible iterations take up less than 1% of the total number of learning iterations to convergence, on average. The highest number of infeasible trials was observed for $\nu_{\text{out}}\tau = 0.032$, wherein the number of infeasible trials are approximately 2%. Note that the statistics stated here exclude any trials where the DTA algorithm did not converge (no solution). For non-converging trials alone, the number of infeasible trials are almost 100%. One explanation for this result is that the randomly generated input and output sequences have no exact solution, however it is difficult to test this hypothesis. Note that for each data point, the number of non-converging trials are approximately 50%, since the capacity is measured when half of all trials are failing.

3.5 Effect of Parameters

This section examines how the learning kernel function $K(t)$ (in Equation 33) and the domain constraint bounds (Equation 34) affect the learning performance of the DTA algorithm. In general, we find that there are optimal choices for each of these parameters, and thus careful optimisation is necessary.

The experimental setup here is the same memory capacity measurement procedure as detailed in the previous section, with both the long and short input scenarios. Unless stated otherwise, all experiments in this section use parameters $\tau_m = 20, \tau_s = 5, N = 500, \nu_{\text{in}} = \nu_{\text{out}} = 0.005$.

3.5.1 Choice of Learning Kernel

In order to study how $\kappa(t)$ affects learning, we performed memory capacity measurements for three different learning kernel functions which appear in the SNN literature. These functions are written as follows:

$$\kappa_{\text{STDP}}(t) = \exp\left(\frac{-t}{\tau_m}\right)\Theta(t) \tag{38}$$

$$\kappa_{\text{PSP}}(t) = \lambda(t) \tag{39}$$

$$\begin{aligned}\kappa_{\text{FILT}}(t) &= \begin{cases} V_{\text{norm}}\left(C_m \exp\left(\frac{-t}{\tau_m}\right) - C_s \exp\left(\frac{-t}{\tau_s}\right)\right), & \text{if } t > 0 \\ V_{\text{norm}}\left(C_m - C_s\right)\exp\left(\frac{t}{\tau_m}\right), & \text{otherwise} \end{cases} \\ C_m &= \frac{\tau_m}{\tau_m + \tau_s} \\ C_s &= \frac{\tau_s}{\tau_m + \tau_s} \end{aligned} \tag{40}$$

Here, $\kappa_{\text{PSP}}(t)$ is simply the PSP integration kernel (Equation 10), which is used in the PSD and INST learning rules (Yu et al. 2013; Gardner and Grüning 2016). The $\kappa_{\text{FILT}}(t)$ kernel was introduced in Gardner and Grüning (2016), which has a non-causal form where input channels that spiked shortly *after* an output spike can be potentiated. $\kappa_{\text{STDP}}(t)$ is an single-exponential STDP curve, which we find to provide better results than the more common two-exponential kernel (Ponulak and Kasiński 2010). The shapes of these learning kernels are shown in Figure 22E.

The effect of $\kappa(t)$ on the resulting weight distribution of the solution is demonstrated in Figure 22B-D. Here, an SRM neuron is trained to learn four desired spike times in response to an input pattern with one input spike per channel. The figures show the solution weights after one iteration of the DTA algorithm,

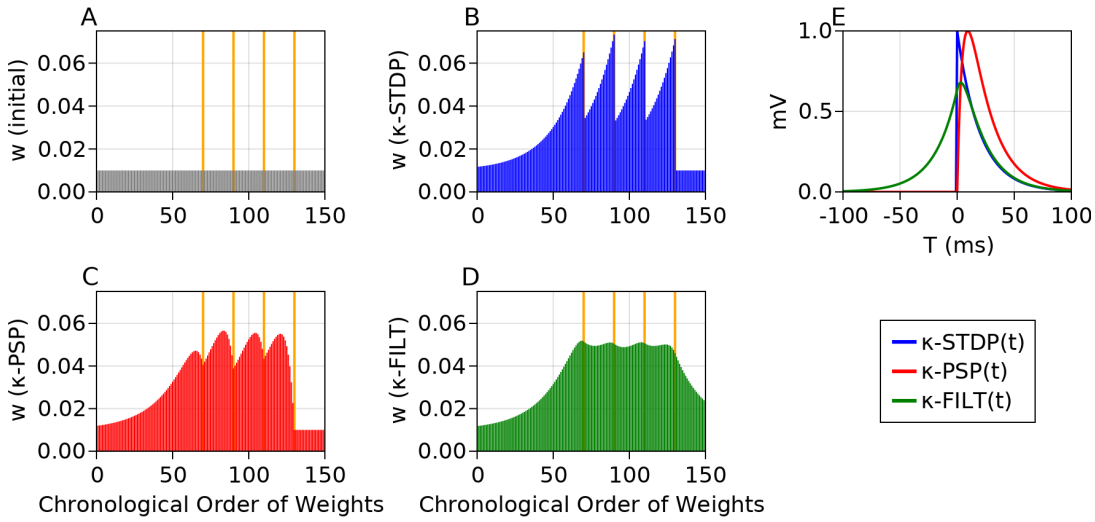

Figure 22: Weight distributions of an SRM neuron **A**: Before learning, and **B**-**D**: after one iteration of the DTA algorithm learning four desired spike times. **E**: The temporal shape of the three $\kappa(t)$ learning kernels.

after which convergence was observed using all three kernels. The different peak heights of the weight distributions in each figure illustrates the magnitudes of the four variables ζ_l^{d} computed by the solver.

Simulation results obtained from memory capacity measurements (Figure 23) indicate that the learning performance of the DTA algorithm is highly dependent on $\kappa(t)$. In both learning scenarios, κ_{PSP} provided the best results. κ_{STDP} demonstrated high performance in the long pattern scenario, but has much worse performance in the short pattern scenario. Increasing the maximum number of epochs from 100 to 500 did not significantly increase this performance. The worst capacity results are demonstrated while using κ_{FILT}, which is surprising since this learning kernel has demonstrably better performance compared to κ_{PSP} when used in a Widrow-Hoff-based learning approach (Gardner and Grüning 2016).

3.5.2 Choice of Domain Constraints

Here, we explore the effect of the domain constraint parameters lb^{d} and ub^{d} on the DTA algorithm during learning.

The experimental conditions are as follows. Firstly, we generate 100 random

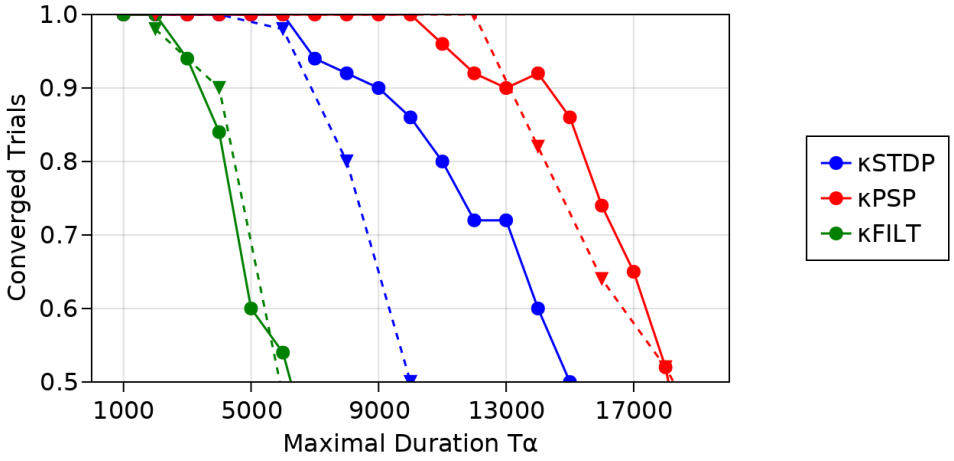

Figure 23: Memory capacity of the DTA algorithm with different learning kernel functions. Solid lines show the number of converged trials in the long pattern scenario, dashed lines are for the short pattern scenario. In general, κ_{PSP} provided the best results.

input patterns and desired output sequences with duration $T = 10000$, and train SRM neurons to learn a single input-output mapping in each trial (single pattern scenario). For each input-output pair, the DTA algorithm is run *without* the domain constraints as the baseline trial. A grid-search parameter optimisation was performed for $-0.1 \leq lb^{\mathrm{d}} \leq 0.1$ and $-0.1 \leq ub^{\mathrm{d}} \leq 0.1$ in increments of 0.02. Note that any impossible parameter settings ($lb^{\mathrm{d}} \geq ub^{\mathrm{d}}$) are ignored in the results. Each parameter combination was tested on the same 100 random PTS problems as the baseline trials. The optimisation and baseline trials are compared in terms of the percentage of converged trials, the average number of learning epochs to convergence, and the total algorithm runtime. κ_{PSP} is used as the learning kernel in all simulations.

	With Constraints	**Without Constraints**
Converged Trials (%)	87.65	94
Epochs to Convergence	9.58 ± 4.30	7.12 ± 2.72
Algorithm Runtime (s)	6.82 ± 3.93	15.44 ± 17.95

Table 1: Overall comparison of learning performance for the DTA algorithm when trained with or without the domain constraints. Note that the results for Epochs to Convergence and Algorithm Runtime only includes converged trials.

Overall, we find that the domain constraints are not necessary for convergence. Out of the 100 PTS problems, there were zero instances where the algorithm converged with the constraints, but the baseline trial did not converge. Some of the parameter combinations resulted in lower percentages of converged trials compared to the baseline. Importantly, such combinations all fall in the parameter range $lb^{\mathrm{d}} \geq ub^{\mathrm{d}} - 0.04$, indicating that these domain bounds are too restrictive. Additional analysis confirm that experimental trials in this parameter range are dominated by infeasible learning iterations. When comparing only the trials that converge, the trials with domain constraints also exhibited worse convergence speed compared to the baseline (Table 1).

However, we also find that the average runtime of the trials with domain constraints are significantly faster when compared to the baseline trials. Further analysis reveals that approximately 18% of the baseline trials showed unusually high runtime, and that these trials computed very large ζ_l^{d} values in the first learning iteration, which resulted in many undesired output spikes at the second iteration. This phenomenon in the baseline trials greatly affected both the simulation time and the learning time, but not the convergence percentage. We did not observe this in any learning trials with the domain constraints, with any parameter combinations tested here. These results indicate a trade-off between convergence and learning stability, as the inclusion of the domain constraints results in worse convergence, but better overall runtime.

Additionally, we find that the optimal values for the domain constraints are $lb^{\mathrm{d}} = 0$ and $ub^{\mathrm{d}} = 0.1$. This means that the lowest average runtime is achieved when computed values of ζ_l^{d} are positive.

3.6 Noise Robustness

In this section, we investigate the robustness of the DTA algorithm in the presence of noise. Specifically, while the learning kernel κ_{PSP} has the highest learning accuracy and memory capacity out of the three learning kernels tested in Section 3.5, updating weights proportionally to the PSPs produces solutions which are not robust to noise. Here, we show that better solutions can be obtained by tuning the time constants of the learning kernel function.

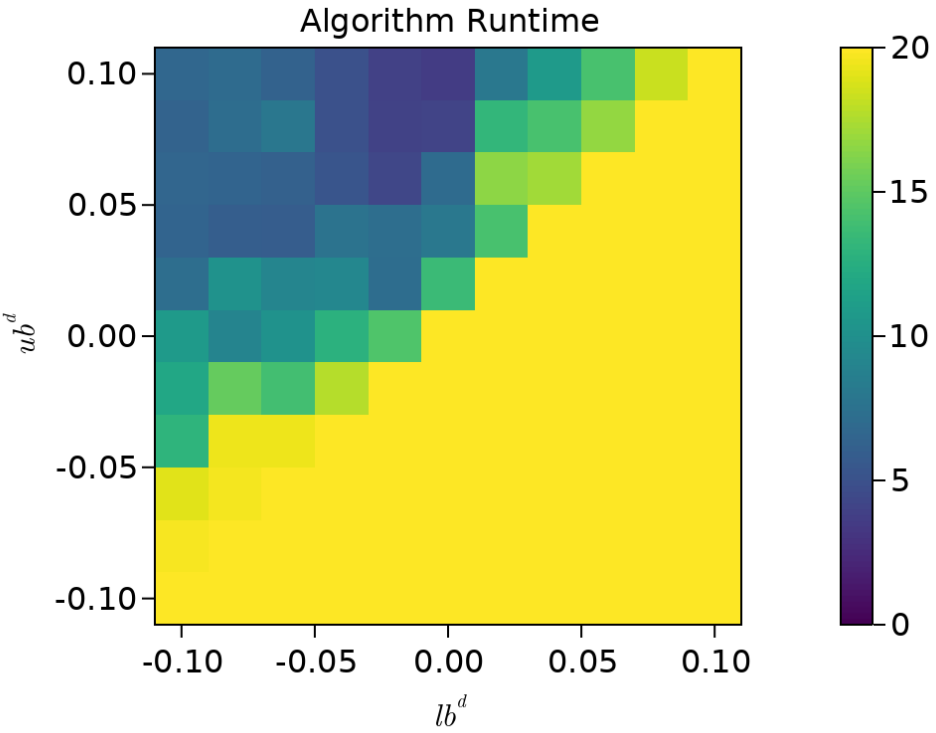

Figure 24: Average runtime of the DTA algorithm with different values of the domain constraints lower bound lb^{d} and upper bound ub^{d}.

3.6.1 Experimental Setup

According to Lee, Kukreja and Thakor (2016), there are two characteristics to a noise-robust solution for a PTS problem: the first component is the average membrane potential trajectory at spike time. Since the input channels which spike closest to a desired output spike incur the smallest weight adjustment with κ_{PSP}, the learning is prone to produce solutions where the membrane potential meets the threshold then immediately decays (Figure 25). The resulting solutions are susceptible to downward perturbations in the membrane potential as a result of input noise. The second characteristic is the sparsity of the solution, meaning how many synapses contribute to a spike. Solutions produced by κ_{PSP} minimise the $\mathcal{L}_2$ norm, where all synapses contribute based on their sum PSPs. Instead, solutions which minimise $\mathcal{L}_1$ norm will have only the most significant synapse contribute. Lee, Kukreja and Thakor (2016) shows that sparser solutions exhibit lower average subthreshold membrane potentials, which can be more robust to small upward perturbations.

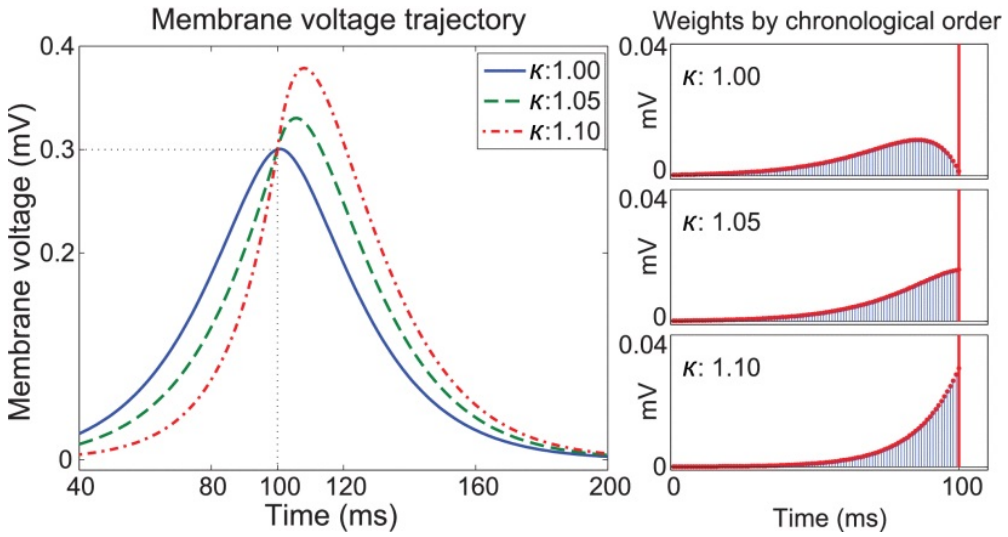

Figure 25: Membrane potential trajectories without reset (left) produced by different weight solutions (right), with a single desired output spike time at time 100. Replicated from Lee, Kukreja and Thakor (2016), and κ denotes a regularisation term in the CONE method.

In order to tune the DTA solutions for better noise robustness, we now introduce a new learning kernel κ^*_{PSP} to the DTA formulation, which can be written

as follows:

$$\kappa^*_{\mathrm{PSP}}(t) = V_{\mathrm{norm}} \left(e^{\frac{-(t-t^i_j)}{\tau^*_m}} - e^{\frac{-(t-t^i_j)}{\tau^*_s}} \right)$$
$$\tau^*_m = \frac{\tau_m}{\phi}$$
$$\tau^*_s = \frac{\tau_s}{\phi}$$

Here, we have simply replaced time constants in κ_{PSP} with new values which are controlled by a hyper-parameter ϕ. Figure 26 demonstrates the effect of ϕ on the shape of the learning kernel. Setting $\phi > 1$ has two effects: firstly, κ^*_{PSP} will peak closer to $t = 0$ compared to κ_{PSP}. This produces steeper membrane potential trajectories at spike time, which increases the robustness against downward noise. Additionally, the width of the kernel function is smaller, which results in a sparser solution and protects against upward noise. Our research question now involves determining whether this strategy is (1) effective against input noise, and (2) maintains high DTA learning performance.

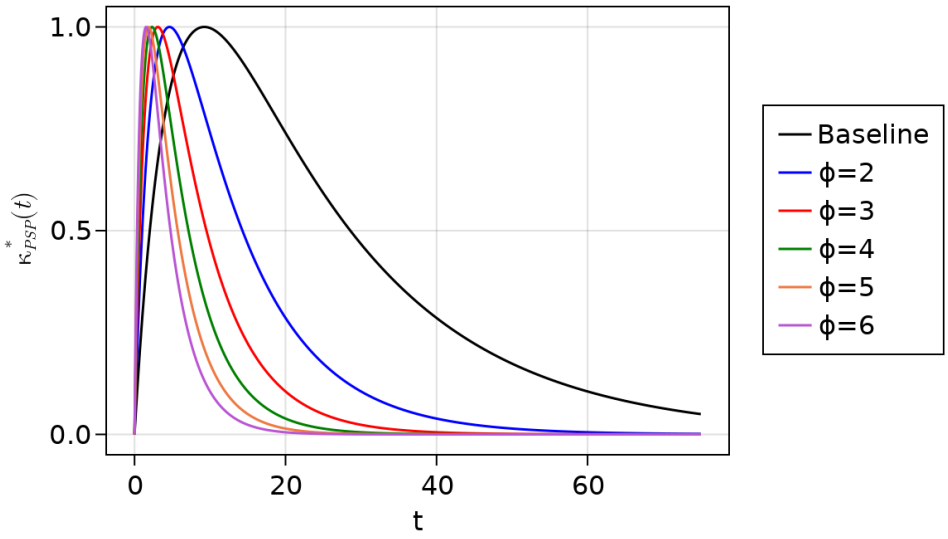

Figure 26: Effect of parameter ϕ on the learning kernel κ^*_{PSP}.

The experimental setup is as follows. Firstly, we generate 100 random input

patterns and desired output sequences with $T = 1000$, for the single pattern learning scenario. We then create 'noisy' input patterns by applying a Gaussian spike jitter to each input spike, with zero mean and standard deviation σ. Finally, we perform DTA training on the original input patterns using $\kappa^*_{\text{PSP}}(t)$ while varying $2 \leq \phi \leq 6$, and record the performance of the solutions on the noise-corrupted patterns.

3.6.2 Simulation Results

Simulation results are shown in Figure 27. Panel A plots the vRD metric against increasing degrees of Gaussian noise. Overall, increasing the value of parameter ϕ contributed significantly to the noise robustness of solutions, as expected. With $\phi \geq 2$, the solutions are robust against even high degrees of Gaussian noise. Panel B plots the percentage of converged trials for various values of parameter ϕ. Here, it can be seen that setting $\phi > 4$ will significantly compromise the ability of the DTA algorithm to converge to a solution. This result is also expected, as weight solutions with large ϕ have very large weights and steep membrane potential trajectories, which may cause the neuron to spike twice in quick succession when learning a single output time.

There are two important implications of the above results. Firstly, we can confirm the hypothesis that by tuning the shape of the learning kernel function, the resulting distribution of the solution weights can be influenced for better robustness to noise. Maintaining high accuracy while being robust to noise is a desirable property for deployment on neuromorphic platforms, which are known to suffer from computational variability induced by manufacturing inaccuracies and thermal instabilities (Wunderlich et al. 2019). Secondly, there is a clear limit to the extent to which the learning kernel can be tuned by the ϕ parameter, without compromising accuracy. In the best case scenario of $\phi = 4$, the algorithm is on average 3 times more robust to input jitter noise than the baseline (based on vRD values) while maintaining the ability to reliably converge. For applications that require significantly better noise robustness than is demonstrated here, more complex solutions for tuning the learning kernel may be required.

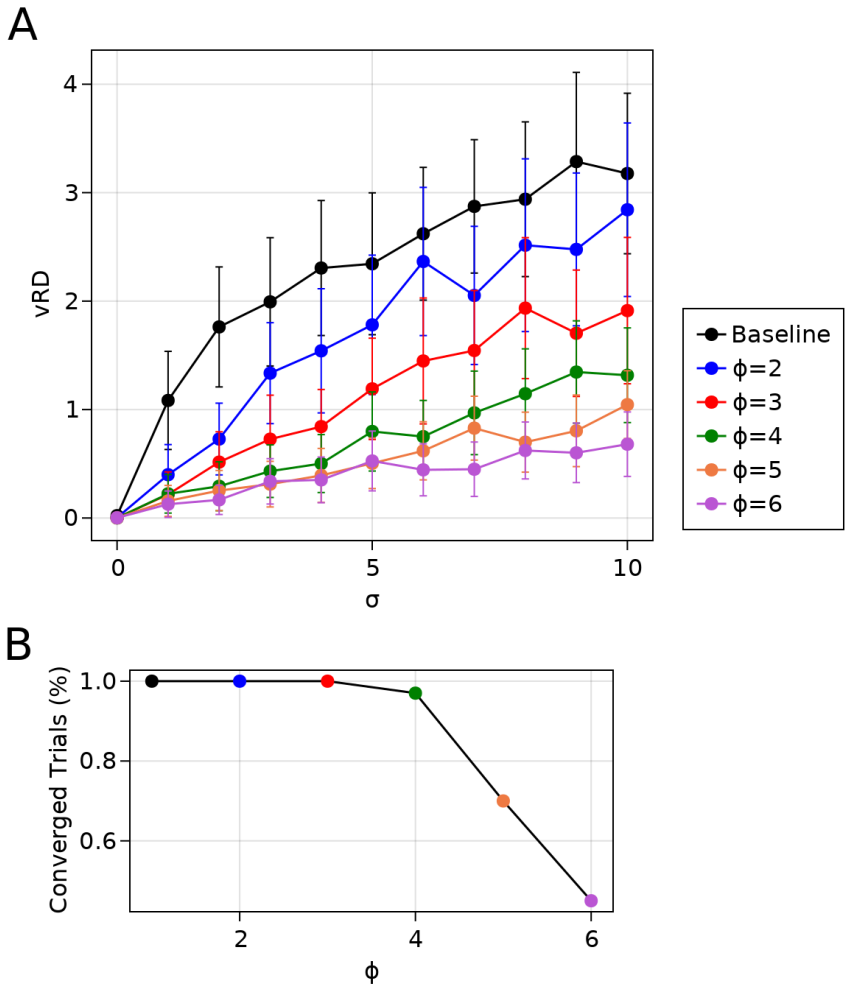

Figure 27: **A**: Performance of DTA solutions for the PTS task on noise-corrupted inputs, shown for varying degrees of Gaussian spike jitter with standard deviation σ. **B**: Percentage of converged trials using the learning kernel κ^*_{PSP} while varying the parameter ϕ.

3.7 Algorithm Runtime

In this section, we compare the runtime of the DTA algorithm to that of existing methods in the literature. The primary objective is to demonstrate that the DTA method can provide a significant benefit in terms of convergence speed and runtime.

3.7.1 Experimental Setup

In particular, we are interested in how the DTA method compares against Widrow-Hoff-based methods such as the ReSuMe, PSD, and FILT algorithms (Ponulak and Kasiński 2010; Yu et al. 2013; Gardner and Grüning 2016). Due to these methods having a very similar form of weight update to Equation 33, they provide a good benchmark for the effectiveness of the constraint solver step in our algorithm. Note that our implementation of the ReSuMe method utilises the learning kernel $\kappa_{\text{STDP}}(t)$.

To reduce the computational burden, we will only compare the DTA method against one Widrow-Hoff-based method from the literature. As such, we first perform parameter optimisation and compare the ReSuMe, PSD, and FILT algorithms in the memory capacity learning task (Section 3.4). The resulting learning rate used in the Widrow-Hoff-based methods is $\eta = \frac{0.12}{|\mathbf{d}|}$. We find this optimal value works equally well for ReSuMe, PSD, and FILT. The inclusion of a normalisation term in η was originally suggested in Gardner and Grüning (2016), and significantly improved the learning performance for all three algorithms (Figure 28). Overall, the FILT method produces the best results, and so we use this algorithm as comparison to the DTA method.

The experimental setup is the long input pattern scenario as described in Section 3.4. In general, we vary the pattern duration T while measuring the average number of epochs to convergence, per-epoch runtime, and total runtime of the algorithms. For each value of T, 50 independent trials were run. The DTA algorithm is trained for maximum 100 epochs, with parameters $\kappa(t) = \kappa_{\text{PSP}}(t)$, $(lb^{\text{d}}, ub^{\text{d}}) = (0, 0.1)$, and $\eta = \frac{0.01}{|\text{d}|}$ (for the fall-back update). The FILT algorithm is trained for maximum 500 epochs with parameter $\eta = \frac{0.12}{|\mathbf{d}|}$. Overall, we did not

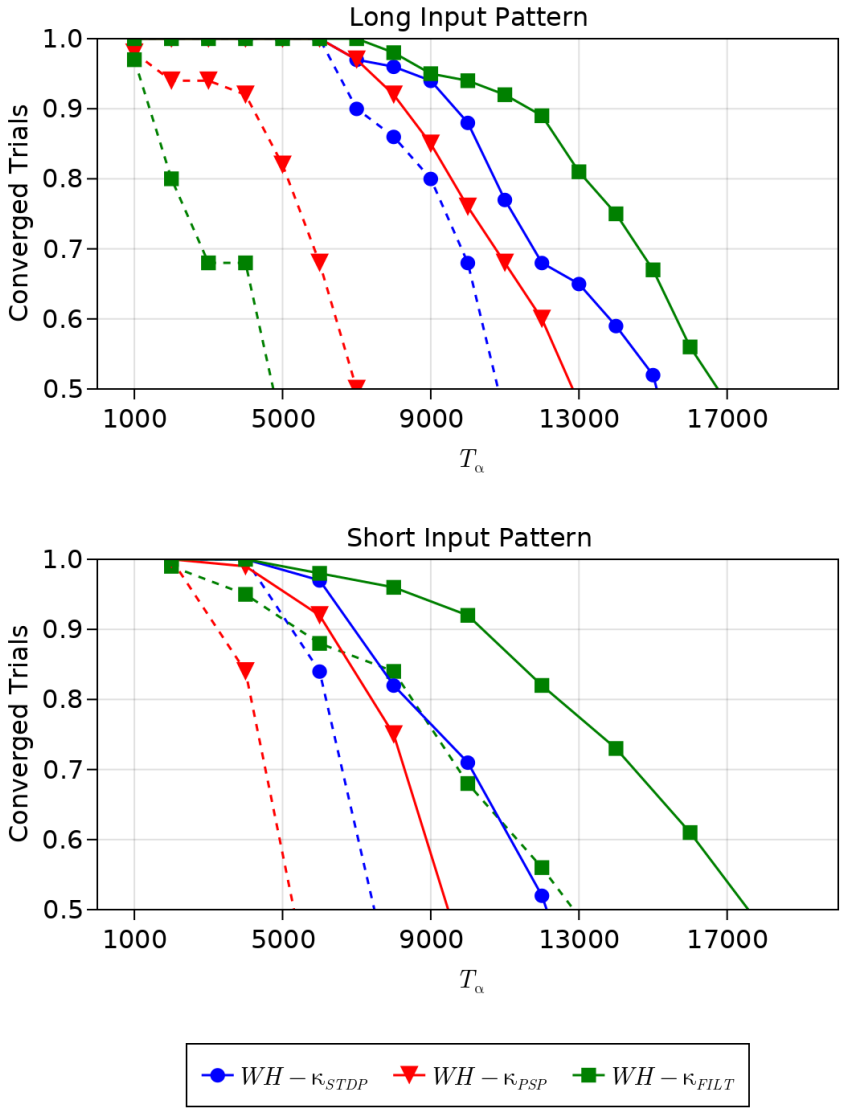

Figure 28: Memory capacity measurements of Widrow-Hoff-based learning algorithms, for the long (top panel) and short (bottom panel) input pattern scenarios. Solid lines denote trials where the learning rate is normalised by the number of desired output spikes, dashed lines denote trials where the learning rate is not normalised. Overall, the FILT learning kernel produced the best results for Widrow-Hoff-based learning.

find any trials where one algorithm converged on the learning problem but the other algorithm did not. Any trials where both algorithms failed to converge were discarded and rerun, as this was not indicative of runtime to convergence.

3.7.2 Simulation Results

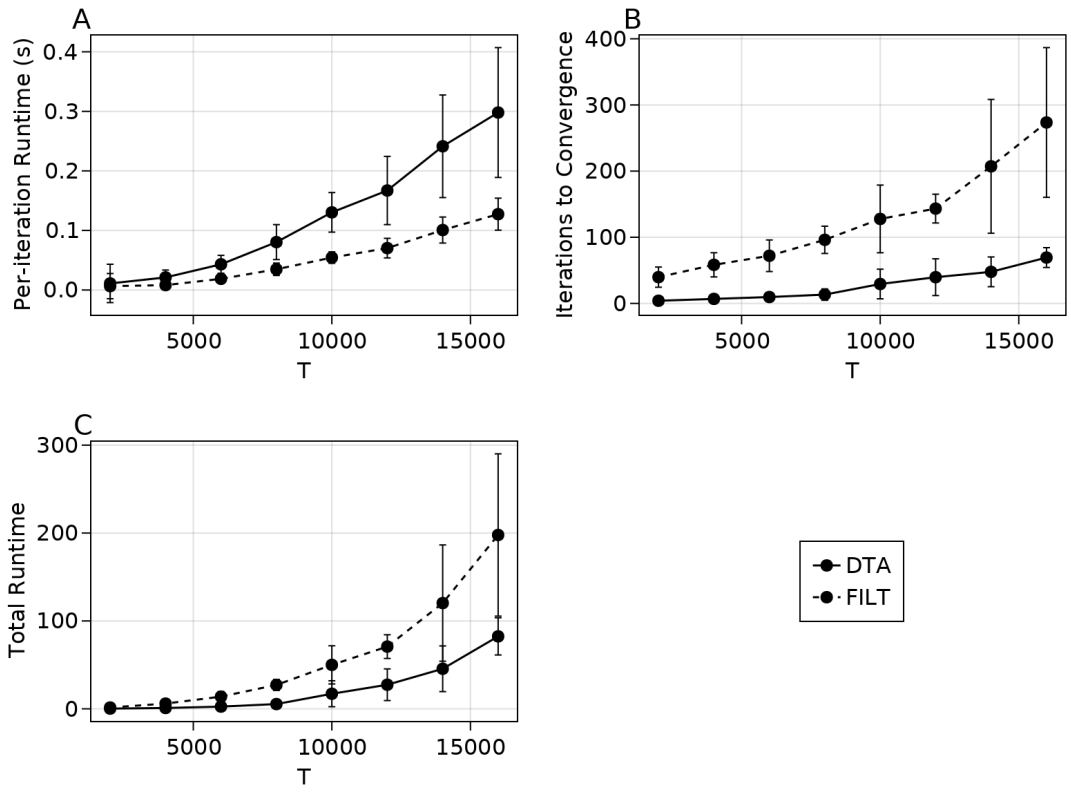

Figure 29: Runtime comparisons between the DTA and FILT algorithms, for various values of pattern duration T in the single input pattern scenario.

Simulation results are shown in Figure 29. On average, the DTA algorithm is on average 4.08 times faster than the FILT algorithm, in terms of total runtime (panel A). The time difference is largest for small input duration, and decreases as T increases. With $T = 16000$, we observe an average of 2.39 times the runtime improvement compared to the FILT algorithm. Additionally, our algorithm requires on average 6.18 times fewer learning iterations to reach convergence. A similar trend to the total runtime is observed here, with the largest difference

observed for small T (panel C). In an average learning iteration, the DTA algorithm is 2.30 times slower than the FILT algorithm, and this time difference can be entirely attributed to the solver step (panel B). The per-iteration simulation time is confirmed to be the same for both algorithms, regardless of T.

The above results indicate that the learning efficiency of the DTA algorithm is overall a net improvement compared to the FILT algorithm. This achieves the fourth goal of the algorithm, however the difference was overall not as large as expected. Given the relatively high difficulty to implement the DTA algorithm when compared to an implementation of the FILT algorithm, there is a significant trade-off in model complexity and efficiency which must be considered for practical applications. It is also worth noting that the runtime of the DTA algorithm is in essence entirely dependent on the efficiency of the solver, which may be an undesirable property.

3.8 Analysis of Learning Interference

In this section, the DTA algorithm is compared against the FILT algorithm on a simple 'toy' learning problem, in order to investigate the effects of LI on each algorithm. These results can be of potential benefit for two reasons: the first reason is that the effects of LI are not well-characterised in the literature. While it has been hypothesised that LI can affect the convergence rate of learning, this has not yet been confirmed Xu et al. (2013). The second reason is that it is not yet clear that the DTA algorithm can overcome this problem during learning.

3.8.1 Experimental Setup

Learning interference is defined as the phenomenon of adjusting the timing of one output spike, inevitably affecting the timing of other output spikes. Here, we demonstrate this problem with a toy problem: a learning scenario wherein a neuron with two input channels is being trained to elicit two target output spikes, denoted as t_1^{d} and t_2^{d}. There is precisely one input spike arriving at each input channel, which we denote as $t^{i=1}$ and $t^{i=2}$. The spike timings are set up so that:

$$t^{i=1} < t_1^{\mathrm{d}} < t^{i=2} < t_2^{\mathrm{d}}$$

As such, only $t^{i=1}$ contributes to generating the first output spike t_1^{d}. We assume also that the distances between $t^{i=1}, t_1^{\mathrm{d}}$ and $t^{i=2}, t_2^{\mathrm{d}}$ are both far enough apart that a solution exists (if $t^{i=1}$ and t_1^{o} are two close together then the first weight will grow large enough to elicit two output spikes after the first input, and so no solutions exist). Figure 30A demonstrates this toy problem.

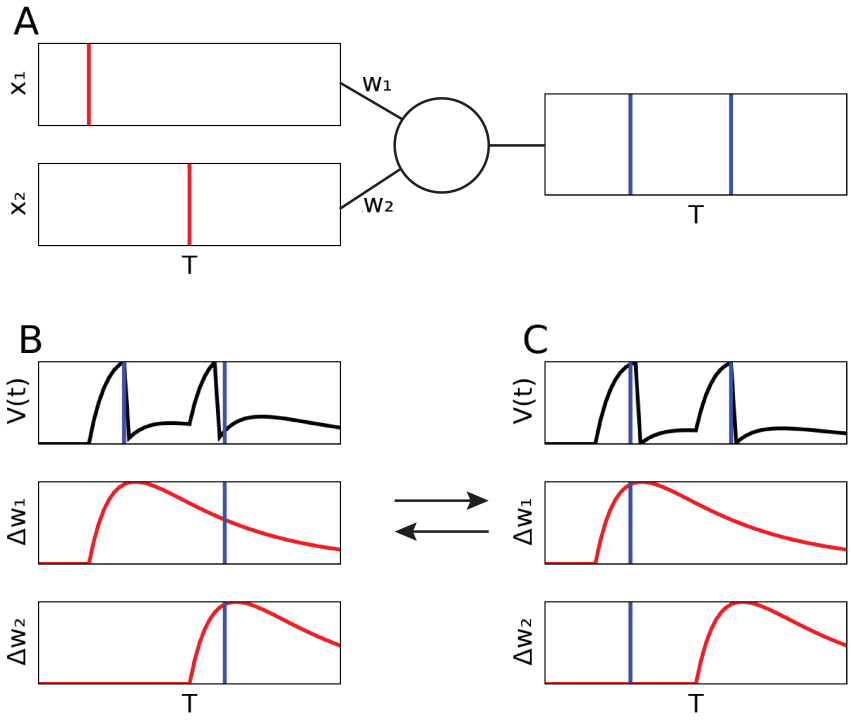

Figure 30: Illustration of learning interference with FP learning algorithm. **A**: the Two-Spikes problem with two input spikes (red lines) and two desired output spikes (blue lines). **B**: (top) First spike is converged, second spike is earlier than desired. Hence, $-\Delta w_i$ is applied which changes both weights; (bottom) intersection between red and blue lines is $\kappa_{\mathrm{PSP}}(t^d)$ for each weight. **C**: because of the previous weight update, the second spike is converged but the first is now later than desired. Adjusting the first spike with Δw_i then affects the second spike.

3.8.2 Learning Demonstration

In order to demonstrate the learning interference problem on a conceptual level, we can first apply the FP learning algorithm in a step-by-step manner. The straightforward weight adjustment rule in this algorithm (Section 2.3.3) provides a simple illustration of the problem.

The example scenario involves a neuron which is already close to converging to a solution, by setting weights such that $t_1^{\mathrm{o}} = t_1^{\mathrm{d}}$, and $t_2^{\mathrm{o}} - \epsilon = t_1^{\mathrm{d}}$, where ϵ is a positive variable with a small value (Figure 30)B. In the first step of the FP algorithm, we set $t_{\mathrm{error}} = t_2^{\mathrm{o}}$ and apply a negative weight adjustment of the form in Equation 14. Since both weights have nonzero PSPs at t_2^{o}, both output spikes occur later in time, after the first update (Figure 30C). As such, adjusting the time of t_2^{o} towards its desired value had the unintended effect of moving the time of t_2^{o} away from its desired value. This fits with our definition of learning interference.

3.8.3 FILT Algorithm

In this experiment, we will apply the FILT algorithm to the above learning scenarios, and record the progress through weight space. All results shown here are for $\Delta t = t_1^{\mathrm{d}} - t^{i=1} = t_2^{\mathrm{d}} - t^{i=2} = 7$, however additional simulations were run for $5 < \Delta t < 9$ without reaching any different conclusions. Similarly to the learning demonstration done with the FP algorithm, the initial weights are set such that the first output spike is already converged, since this scenario demonstrates the effects of LI most easily.

Simulation data for the FILT algorithm is shown in Figure 31. The top panel demonstrates results for $t_2^{\mathrm{d}} - t_1^{\mathrm{d}} = 10$, which will be referred to as Scenario A. The bottom panel demonstrates results for $t_2^{\mathrm{d}} - t_1^{\mathrm{d}} = 40$, which will referred to as Scenario B. Results for values between $10 \leq t_2^{\mathrm{d}} - t_1^{\mathrm{d}} \leq 60$ are also run, however no further conclusions could be drawn outside of the data shown here. The background contour and colour shows the loss landscape of the problem, with the vRD used as the loss value. The plotted line shows the position of the weights in two-dimensional weight space after each learning iteration, with a red starting point and white end point. The FILT algorithm converges in every permutation

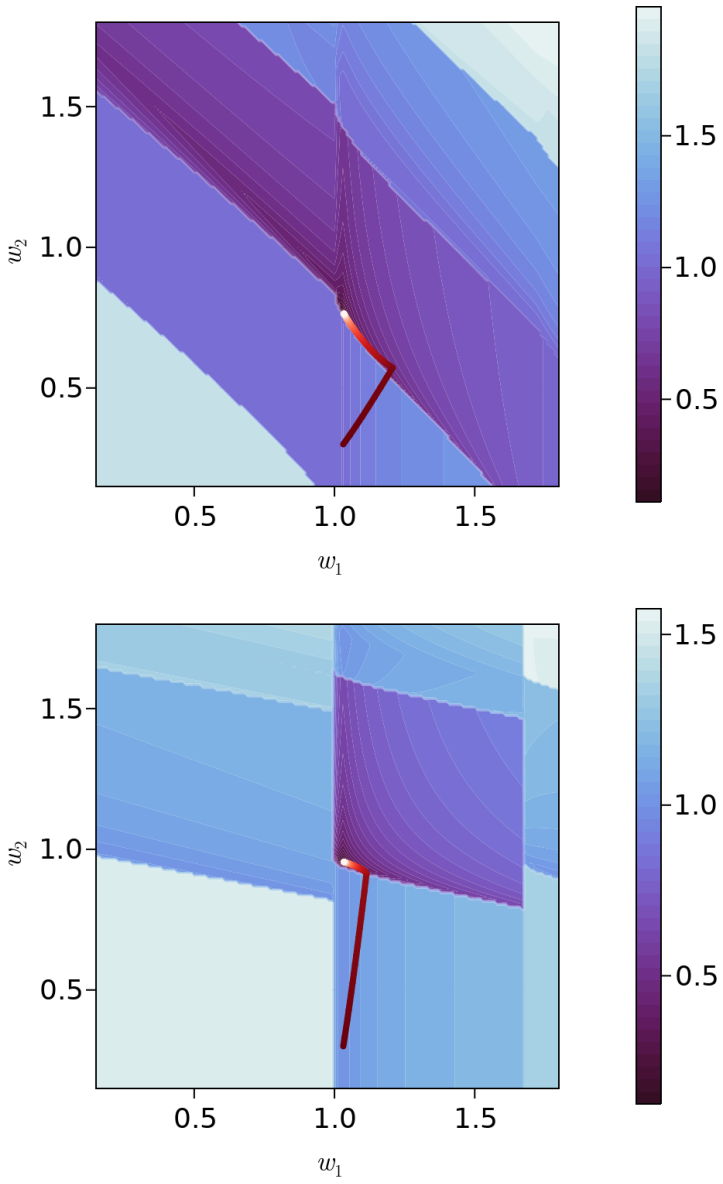

Figure 31: The effects of learning interference on the FILT algorithm throughout learning, illustrated using the 'toy' PTS problem. Colours and contours represent the van Rossum distance values mapped to the weight space. Plotted line denotes the progression of the learning from start (red) to finish (white). **Top plot**: $t_2^{\mathrm{d}} - t_1^{\mathrm{d}} = 10$. **Bottom plot**: $t_2^{\mathrm{d}} - t_1^{\mathrm{d}} = 40$. The contours show discrete regions of the loss value, which corresponds to the loss with (from lightest to darkest): 0 output spike, 1 output spike, and 2 output spikes generated by the neuron. Here, we observe that the FILT algorithm finds the optimal solution, but does not take a direct path.

of the problem parameters.

The primary result is that there are two phases to the weight progression of the FILT algorithm during learning. In the first phase, the weight vector initially travels in a direction where the vRD value actually *increases*. Then, there is a turning point where the loss sharply drops, and the learning transitions into the second phase where the weight vector travels directly towards the optimal solution. The explanation for the first phase of learning is that the second output spike does not exist, hence a net positive weight update is applied to both weights, during which learning interference draws w_1 away from its converged value. This continues until the weights are large enough for a second spike to occur, which corresponds with the turning point entering the second phase of learning, and the immediate drop in loss value.

The secondary result is that the effect of LI on w_1 is larger for smaller values of $t_2^{\mathrm{d}} - t_1^{\mathrm{d}}$. This is an expected result, since it is logical that output spikes which are temporally close have much larger overlaps in their PSPs, which results in larger interference effects during weight adjustment. This phenomenon was also already noted by Gardner and Grüning (2016), and is consistent throughout the rest of the simulation data. Importantly, it was also observed that the FILT algorithm converged in fewer learning iterations for larger values of $t_2^{\mathrm{d}} - t_1^{\mathrm{d}}$ where there is small interference, even though the distance that the learning must travel through weight space is larger. This supports the hypothesis that learning interference has an adverse effect on the convergence rate of multi-spike learning.

3.8.4 DTA Algorithm

Simulations of the toy problem were performed for the DTA algorithm, under the same experimental conditions as with the FILT algorithm. The first result is that under all permutations of the simulation parameters Δt, the DTA algorithm reported convergence within one learning iteration. While this immediately demonstrates that the DTA algorithm overcomes the effects of learning interference, it would be more interesting to investigate the underlying mechanisms of the algorithm that lead to this result. To this end, we write the DTA weight update equation in this learning scenario as follows:

$$\begin{aligned}\Delta w_i &= a_i + b_i + c_i \\ a_i &= \zeta_1^{\mathrm{d}} \lambda(t_1^d - t^i) \\ b_i &= \zeta_2^{\mathrm{d}} \lambda(t_2^d - t^i) \\ c_i &= \zeta_1^{\mathrm{o}} \lambda(t_1^o - t^i)\end{aligned}$$

Here, a_i and b_i represent the weight update terms made by the DTA algorithm to channel i corresponding to t_1^d and t_2^d, respectively. Similarly, c_i represents the weight update term corresponding to the actual output spike t_1^o which exists at the start of learning. Importantly, application of the DTA algorithm in this learning scenario will be performed with the additional constraint $\zeta_1^{\mathrm{o}} = 0$. There are two reasons for this: firstly, because the initial weights are set such that $t_1^o = t_1^d$, it follows that $\lambda(t_1^o - t_1^i) = \lambda(t_1^d - t_1^i)$ and c_i is redundant, as any non-zero solution value of ζ_1^{o} can be absorbed by ζ_1^{d}. The second reason is purely for demonstration purposes, as this makes illustrating the results easier with only two update terms a_i and b_i.

Simulation data for the same A and B scenarios as demonstrated in Figure 31 is shown in Figure 32. Instead of showing the weight progression of the sole learning iteration, we instead opted to plot the two vectors representing each update term: $\mathbf{a} = [a_1, a_2]$ (dashed line) and $\mathbf{b} = [b_1, b_2]$ (dotted line). The solid line shows the total $\Delta\mathbf{w}$ vector. Here, the computation of the DTA algorithm as a result of the two update terms is demonstrated. When compared to Figure 31, we observe that the first phase of the weight progression in the FILT algorithm matches that of $\mathbf{b}$, however the first spike time is already converged and so $\mathbf{a} = [0, 0]$. In the DTA algorithm, the equality constraint corresponding to t_1^d results in a negative value of ζ_1^{d}, which is reflected in the vector $\mathbf{a}$.

These results illustrate the way that the DTA algorithm overcomes learning interference. While the form of the DTA update equation is very similar to the existing algorithms derived from the WH rule, the DTA algorithm considers all weight update terms *simultaneously*, in a global and correlated fashion. In contrast, the weight update rule in the FILT algorithm are considered as a sum of uncorrelated update terms.

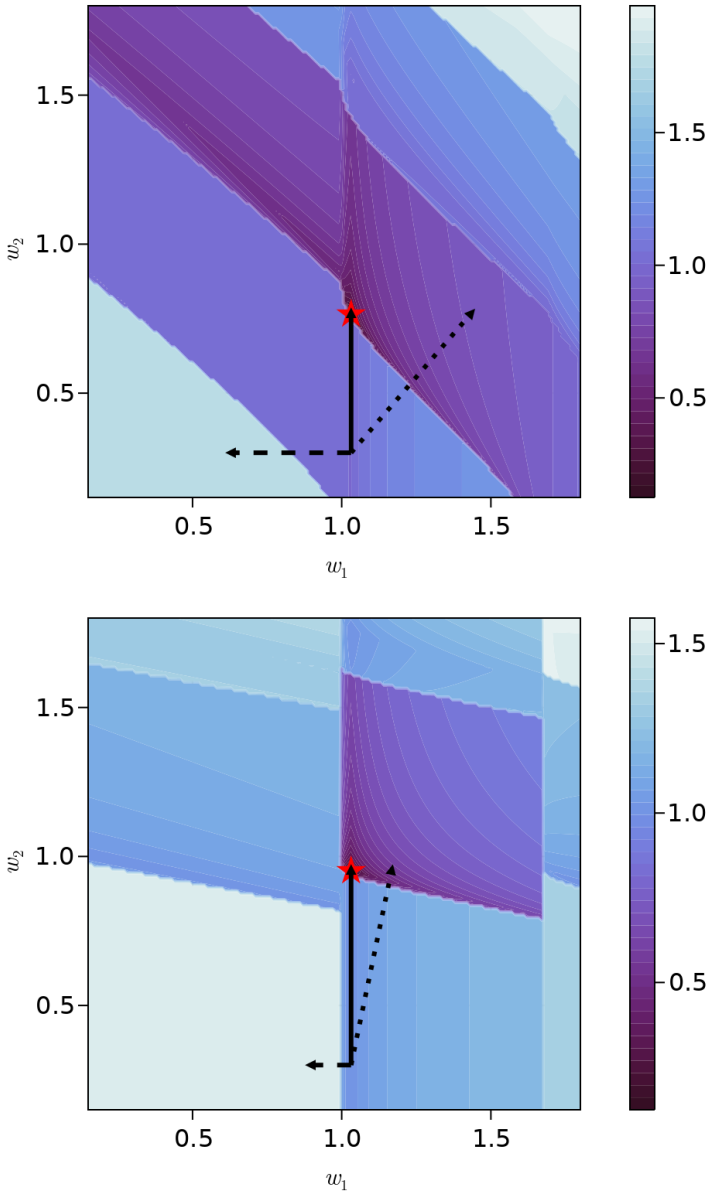

Figure 32: The weight computation of the DTA algorithm in the toy learning problem, with the dashed and dotted arrows demonstrating the two update terms corresponding with the two desired output spikes. **Top plot**: $t_2^{\mathrm{d}} - t_1^{\mathrm{d}} = 10$. **Bottom plot**: $t_2^{\mathrm{d}} - t_1^{\mathrm{d}} = 40$

3.9 Discussion

This chapter proposes the DTA learning algorithm for solving multi-spike PTS problems for single-layer SNNs. The main novel contribution of this method is to compute a set of variables ζ which scales the extent of the weight update terms corresponding to each output spike, by solving an optimisation problem. This is in contrast to iterative approaches to computing weight updates wherein a static learning rate value is used to scale all weight update terms equally. Our experiments suggest that the DTA method is viable and efficient, achieves high memory capacity, and is able to overcome the learning interference problem.

κ_{PSP} appears to be the overall best learning kernel in terms of accuracy. Not only does it exhibit the best memory capacity results, tuning the time constants had a larger effect on the noise robustness of solutions compared to tuning the time constants κ_{STDP} and κ_{FILT}. However, the poor memory capacity results achieved with κ_{FILT} are perplexing, as this kernel function is the most recent in the literature, and has documented advantages over κ_{PSP} in Widrow-Hoff-based learning (Gardner and Grüning 2016). One possible hypothesis for this result is the acausal properties of this kernel function, wherein channels that do not contribute to an output spike can be potentiated. In the DTA algorithm, weight adjustment magnitudes ζ can be much larger than the typical Widrow-Hoff-based learning rate, which may cause the non-contributing channels to generate undesired spikes after an update step.

An alternative approach to tuning the properties of κ_{PSP} for better noise robustness is simply to shift the learning kernel in the temporal dimension. In initial investigations, we introduced an additional parameter $s > 0$ to be used as $\kappa_{\text{PSP}}(t) = \kappa_{\text{PSP}}(t - s)$. This had the effect of moving the kernel peak closer to zero, thus making the membrane potential trajectory steeper. The resulting noise robustness results were not as good as optimising parameter ϕ as in Section 3.6. A more complex method is to tune the kernel time constants τ_m^* and τ_s^* separately, however this increases the computational requirements of parameter optimisation.

One distinct advantage that our method has over the CONE algorithm is the ability to process the input patterns incrementally, rather than as one batch. In practical learning settings, it may be unrealistic to assume that all the input data is

available at the beginning of learning. In this scenario, the CONE algorithm must calculate a completely new solution. Because our method computes incremental weight updates, the existing solution can be used as the new initial weights as new data becomes available. In particular, this is common in transfer learning scenarios where the application dataset changes (Weiss, Khoshgoftaar and Wang 2016; Niu et al. 2020).

While the learning efficiency improvements that we observe here can be attributed to the fast convergence speed of the algorithm, another contributing factor is the simulation time. In our experiments, simulations are performed in an event-based manner using spike-event queues, which works best for sparse inputs. It is thus possible that with a more efficient simulation strategy, the runtime difference observed in Section 3.7 is lessened. However, we note that the improved convergence speed of our algorithm does not only reduce the simulation time, but also other necessary components of a learning iteration, such as the computational requirement to calculate the spike train distance metric.

One of the central assumptions in the design of the DTA method is that applicable supervised learning problems are in the scenario where the number of synaptic weights of the neuron is larger than the number of constraints. This means that the computational efficiency of the proposed approach may not scale favourably when the assumption is violated. Simulation results in Section 3.7 suggests that this is the case, with the runtime difference between the DTA and FILT algorithms decreasing as the input pattern duration (and the number of constraints) increases. Here, the runtime measurements only continue until the memory capacity bound defined in Memmesheimer et al. (2014). While the DTA algorithm demonstrates approximately half of the total runtime compared to the FILT algorithm at this point, it is reasonable to expect that as the input duration increases the FILT algorithm may eventually be a better choice in terms of efficiency.

One of the problems of the DTA algorithm in its current state is the assumption made in each learning iteration that the set of inequality constraints are sufficient to produce a correct solution. Evidently, the algorithm requires a number of learning iterations to converge for difficult problems, meaning this assumption is violated multiple times during learning. In a discrete-time simulation

strategy, the number of inequality constraints are finite, and thus similarly to the CONE algorithm we can guarantee convergence within a single learning iteration (Lee, Kukreja and Thakor 2016). To alleviate this problem, we also conducted experiments in continuous simulation time, however with additional inequality constraints inserted at discrete intervals throughout the input duration T. As expected, the smaller discretisation resulted in faster convergence speed (in terms of required epochs), however the larger number of constraints and optimisation variables result in much lower runtime efficiency.

The most important weakness of the DTA algorithm is that it is not applicable to multi-layer SNN learning. This is because the algorithm requires both the output times (in the constraints) and the input times (in Equation 33) to be specified. In a multi-layer SNN, the spike times of an intermediate layer are not specified by the learning problem, and as such learning cannot be performed in any layer. This is a critical problem, as we cannot expect a sufficiently complex problem to be solvable by a single-layer SNN. One possible solution is to use the 'kernel trick': passing the input to an intermediate layer of higher dimensionality using random weights. This approach has been successfully demonstrated by many *analytical* SNN learning methods, which also cannot perform multi-layer training (Eliasmith and Anderson 2003; Huang, Zhu and Siew 2006; Tapson et al. 2013; Tapson and van Schaik 2013; Cohen et al. 2016, 2017; Kulkarni and Rajendran 2018; Boucher-Routhier, Zhang and Thivierge 2021). However, we did not consider this a viable method since our CPU-bound simulation strategies greatly limit parallelisation of spike propagation, which would be prohibitive for a randomised hidden layer of potentially tens of thousands of neurons.

3.10 Hardware Information

In this research chapter, we have measured the runtime of the DTA algorithm for analysis and comparison to other learning algorithms in the literature. For the sake of completeness, the details of the physical hardware and software used to generate these measurements are detailed below in Table 2. Note that all neuronal simulations and learning were performed on CPU.

	Component	**Detail**
Software	Programming Language	Julia 1.6.2
Hardware	CPU	
	CPU clock speed	
	RAM	
	RAM speed	

Table 2: Software and hardware information used for neuronal simulation, learning, and time measurements in Chapter 3.

Chapter 4

Spike Count Learning with DTA

4.1 Introduction

In the context of supervised learning for classification, it can be desirable to make predictions based on the number of spikes, rather than the precise spike timing. One of the reasons for this choice is the ability to delegate the decision of how to generate the desired spike train from the user onto the learning process, which was previously discussed in Chapter 2. However, there are a number of further reasons why SC learning can be advantageous over PTS learning.

One benefit of SC learning is the ease at which inference can be carried out, which has implications for the computational complexity of a model. To put very simply, comparing a spike count integer to an integer label is a relatively straightforward task compared to computing the difference between two spike trains. In practice, with a PTS learning setup one would typically have to implement some form of spike distance metric, in order to determine which target spike sequence the actual output is most similar to. In turn, this adds additional implementation complexity to the SNN model, as well as extra computational requirements. This additional cost is especially important in real-world applications with large datasets, and where hardware resources and inference speed are important considerations.

Another reason for utilising SC learning is reliability. In Li and Yu (2020), the authors demonstrated a learning algorithm on a synthetic learning task where

an SRM neuron is tasked to perform multi-class classification with each category assigned to a different target spike count. Here, it was shown that after training, the output of the neuron to the samples in a category forms an approximately Gaussian distribution over the output spike count, which is centred around the target label. This means that even if the neuron fails to reproduce the desired number of spikes, it may still generate a spike count which is close to the label. An implication of this result is that by setting target spike counts which are not close together, a high classification accuracy can be maintained even if the problem is too difficult to solve exactly as the labels specify Li and Yu (2020). In comparison to PTS learning, we have observed in Section 3.8.2 that even a single missing spike which is expected to exist at some precise time can cause the spike train distance value to significantly change, which could potentially compromise the prediction accuracy in the classification setting.

4.2 Motivations & Chapter Layout

In this chapter, the DTA-B algorithm is proposed, which is a direct extension of the DTA algorithm (Chapter 3) designed for supervised learning tasks involving a target spike count. The overall goals of the experiments in this chapter are as follows:

1. Extend the DTA algorithm to SC learning with an arbitrary target number of spikes.
2. Measure the memory capacity of DTA-B in synthetic benchmark tasks.
3. Benchmark DTA-B performance on standard real-world benchmark tasks.
4. Compare the DTA-B performance with existing methods in the literature.

In order to identify appropriate values for the desired spike times, we adapt and incorporate the STS function into the DTA-B method. The STS was introduced in Chapter 2 and originally proposed by Gutig (2016). We will show that this approach can be simplified and applied to the DTA weight update method in order to predict desired output spike times. Once the spike times have been computed,

the original DTA algorithm from Chapter 3 can be used without modification in order to generate the spikes.

The rest of this chapter is organised as follows. In Section 4.3, the DTA-B method for SC learning is described in detail, which largely involves an explanation of the modified STS computation. Section 4.4 provides a learning demonstration of the proposed method on synthetic datasets. Additionally, results are presented in order to provide a first comparison of the method to two other learning algorithms in the literature. In Section 4.5, the runtime of the algorithm is investigated, and the computational complexity of the modified STS operation is discussed. In Section 4.6, numerical experiments are conducted to investigate the ability of the algorithm to learn a large number of output classes. In Section 4.7, the algorithm is applied for the first time on small but standard benchmark datasets which are representative of real-world learning scenarios. Here, several options to modify the architecture of the SNN model is also investigated. Finally, in Section 4.8 the model is applied to the MNIST dataset, which elucidates on the model performance when provided with a large amount of training data belonging to multiple input categories.

4.3 Method Description

The learning scenario is formalised as follows: given a fixed input pattern $\mathbf{x} = [x_1, x_2, ..., x_N]$ where $x_i = [t_1^i, t_2^i, ...]$ is the input spike sequence arriving to the i-th input channel of an SRM neuron, and a desired *spike count* denoted as $|\mathbf{y}|$, the learning problem involves computing a set of weights $\mathbf{w}$ with which the output spike sequence o that the neuron generates is such that $|\mathbf{o}| = |\mathbf{y}|$. Here, $|x|$ denotes the cardinality of x.

Each learning iteration of the DTA-B algorithm is split into two computational steps. In the first step, we apply a process which will be referred to as the *Dynamic Threshold* procedure. The outcome of this process is to identify $|\mathbf{y}|$ target spike times, denoted as $\mathbf{y}^* = [t_1^*, t_2^*, ..., t_{|\mathbf{y}|}^*]$. In the second computation step, we substitute $\mathbf{y}^*$ in place of the desired spike sequence used in the DTA algorithm (Equations 30 and 33), and compute the solution weights as described in Chapter 3.

4.3.1 Dynamic Threshold Procedure

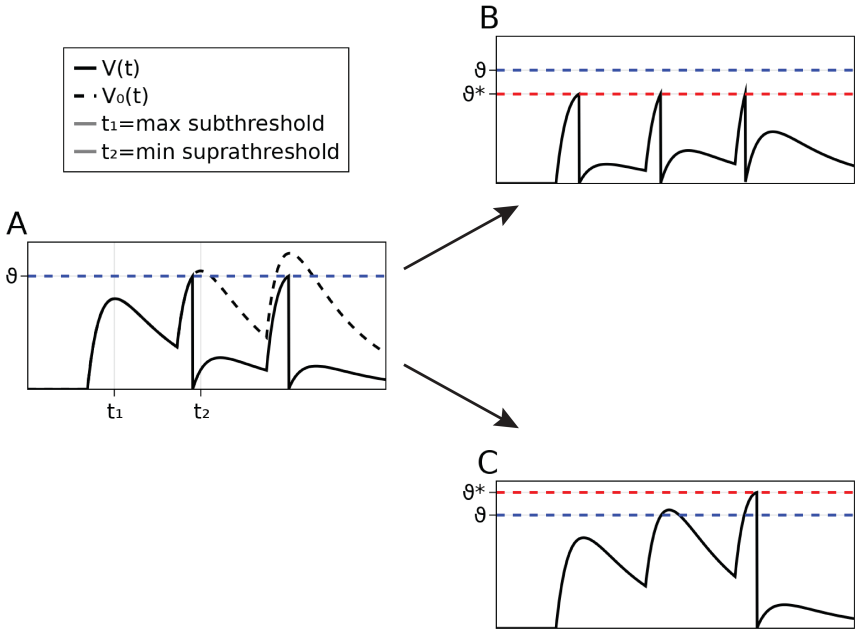

Figure 33: Dynamic threshold procedure for determining desired spike timings for output neurons. **A**: Membrane potential $V(t)$ (solid line) and membrane potential without reset $V_0(t)$ (dashed line) of a neuron with two output spikes. The appropriate time to generate a new spike is t_1, since $V(t_1)$ is closest to ϑ, thus requires the smallest weight adjustment. **B**: decreasing threshold yields an extra spike at t_1. **C**: increasing threshold removes the spike at t_2.

The underlying idea of the Dynamic Threshold procedure is the similar to the basic principle of the STS. In the scenario where $|\mathbf{o}| \neq |\mathbf{y}|$, we raise or lower the spiking threshold from ϑ to a new value ϑ^* with which the neuron generates exactly $|\mathbf{y}|$ output spikes. The identification of an appropriate value for ϑ^* involves interval halving in the interval $[0, 10\vartheta)$, stopping when the correct spike count is observed. An illustration of this procedure is shown in Figure 33, for identifying appropriate transitions between 2 spikes to 1 or 3 output spikes.

When compared to the original procedure proposed by Gutig (2016), our Dynamic Threshold process has two important differences. Firstly, our approach has a relatively loose definition for the target threshold ϑ^*, compared to the critical points ϑ_k^* used by the MST. In the original version, the process of interval halving was stopped when near to critical value ϑ_k^*, and subsequently an additional root

solving step was used to identify ϑ_k^* as numerically close as possible to the real value defined by Equation 29. This is because the weight update calculations in the MST algorithm requires highly accurate ϑ_k^* in order to derive a gradient. In the DTA algorithm, there is no such requirement. As a result, any threshold ϑ^* which generates spike times satisfying the condition $|\mathbf{o}| = |\mathbf{y}|$ is sufficient. This has implications on the efficiency of the algorithm, since the interval halving process is very computationally expensive: each halving step requires one simulation of the neuron with the new threshold.

Secondly, a fall-back mechanism was introduced into the Dynamic Threshold procedure to handle failure. Specifically, if the membrane potential is negative at every point in the neuron simulation (all weights negative), the Dynamic Threshold procedure will be unable to converge to a suitable ϑ^* solution for any $|\mathbf{y}| > 0$. This is due to two reasons: firstly, the interval halving step is only performed in a positive interval. Secondly, the neuron resting potential is zero, so by definition of the neural dynamics a negative threshold cannot be defined. Note that the original MST procedure did not handle this failure state in any way. When the Dynamic Threshold procedure has no solutions, our approach is to set a single desired spike situated at the very end of the input pattern, written as $\mathbf{y}^* = [T]$. As a result, the DTA-B algorithm will attempt to make positive weight adjustments in order to achieve a positive membrane potential. We find the choice of this desired spike time works reasonably well for both our synthetic and real-world benchmarks since at this time the probability of the input PSPs being non-zero is very high. An alternative choice for a desired spike time in this scenario is the time of the largest (negative-valued) membrane potential peak, however it is difficult to find this timing without analysing the entire history of input PSPs.

It is important to summarize the main similarities and differences between the DTA-B and the MST algorithm (Chapter 2). In general, the similarity between the two algorithms is conceptual: both are designed with the idea that the optimal target output spike times can be computed by applying a dynamic threshold ϑ^* to the neuron, following the principle of minimum disturbance. The main differences are as follows:

- Different definition of ϑ^*: In the MST, there is only one possible ϑ^* value per learning iteration. In DTA-B, any value from a range of possible threshold

values are chosen during interval halving.

- Different process to determine ϑ^*: In MST, ϑ^* is found by interval halving and then root-solving, in DTA-B the process is only interval halving.
- Different weight adjustment computation: in MST, the obtained spikes $\mathbf{o}$ under dynamic threshold is used in a recursive analytical derivation of the gradient of the membrane potentials at target and desired spike times. In DTA-B, $\mathbf{o}$ is used as times for inequality constraints of the CSP.
- Fallback mechanism for dynamic threshold: in MST, the algorithm fails when the membrane potential is negative at all times during simulation. In DTA-B, this is recoverable.

4.3.2 Algorithm Summary

A complete summary of the DTA-B algorithm is given in Algorithm 2.

Algorithm 2: DTA Algorithm for learning Spike Counts. The algorithm takes in a set of inputs $\mathbf{X}$ and target output spike counts $\mathbf{Y}$, and compute a synaptic weight vector $\mathbf{w}$ such that all the outputs of each input pattern $\mathbf{x}_p$ is of the same length as target count $|\mathbf{y}|_p$. The function δ denotes the Dirac delta function.

Data: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_p\}, \mathbf{Y} = \{|y_1|, |y_2|, ..., |y_p|\}$

initialise neuron with weights w;
$accuracy \leftarrow 0.0$;
while *epoch* $<$ *maxEpoch or accuracy* $<$ *1.0* **do**
    shuffle X;
    foreach $\mathbf{x}_p$ *in* X **do**
        compute output $|\mathbf{o}|$;
        compute ϑ^* where $|\mathbf{o}| = |\mathbf{y}_p|$ by Dynamic Threshold;
        compute Δw (Eq. 33, 30 & 32);
        $w^* \leftarrow w + \Delta w$;
        $w \leftarrow w^*$;
    end
    $accuracy \leftarrow \frac{\sum_{\mathbf{x}_p \in \mathrm{X}} \delta(|\mathbf{o}| - |\mathbf{y}_p|)}{P}$;
end

4.4 Learning Performance

This section provides a benchmark of the DTA-B algorithm on synthetic data, wherein the input training patterns are randomly generated using Poisson point processes in the same manner as the experiments in Chapter 3.

In addition to providing a learning demonstration, a secondary aim of this section is to benchmark the DTA-B algorithm against existing SC learning methods. We choose two algorithms here for comparison purposes: firstly, the MST (Gutig 2016) will provide a baseline point of comparison, which is useful to highlight the effects of the simplified Dynamic Threshold procedure. However, the MST learning rule was designed to solve a very specific learning problem known as *aggregate-label learning*, and we do not expect it to perform remarkably well in our experimental conditions. To this end, we also compare the DTA-B algorithm

with the EMLC learning rule (Li and Yu 2020). This is a more recent method which claims to have superior learning performance and convergence speed when compared to the MST. The EMLC algorithm achieves this by two design characteristics: firstly, the learning rule also performs a simplified version of the original STS function, where only one output spike time is adjusted at each learning iteration. Secondly, their weight update rule does not contain any recursive computations, and thus is significantly simpler than the MST weight update. Unlike the MST algorithm, the efficacy of the EMLC rule has been demonstrated on a similar experimental setup to the one we use here.

4.4.1 Experimental Setup

In each experimental trial, a set of input patterns with $N = 500$ synaptic channels are randomly generated with a Poisson rate of $\nu = 0.005$ over a time window $T = 50$. As the input patterns are generated, they are evenly and equally split into five input classes, which are labelled with target spike counts $|\mathbf{y}|$ between 1 and 5. Successful convergence is decided by 100% classification accuracies on the entire training set. Importantly, learning performance is measured here using a similar process to the memory capacity measurements made in Section 3.4: 50 independent trials were run for each algorithm, and during each trial the number of input patterns is iteratively increased by 5 each time successful convergence is reported (1 additional pattern for each input class), thus increasing the learning load. A learning trial concludes at the first iteration when less than 50% of all trials have failed to converge, and the total number of patterns at this point is taken as the learning performance.

For all neurons, the initial synaptic weights are drawn from a Gaussian distribution with mean and standard deviation both set to 0.01, and this initial condition is controlled across all three algorithms. All algorithms are run for a number of learning epochs. Each epoch is split into a number of learning iterations, and in each iteration only one input pattern is randomly selected for learning. A learning epoch concludes when all input patterns in the training set have been processed. For the DTA-B algorithm, the learning is carried out for a maximum of 100 epochs. Note that the DTA-B algorithm is applied here *without*

any domain constraints. Initial investigations indicated that learning instabilities are also present here, in the form of large values of the optimisation variables ζ that cause the number of spikes to sharply increase. However, the number of target spikes is relatively small in this experiment when compared to the PTS learning scenario, and the instabilities did not have as adverse an effect on the overall runtime.

The EMLC and MST algorithms were trained for a maximum of 200 learning iterations. Before the experiment was performed, a grid search parameter optimisation was carried out for both the MST and EMLC algorithm, in order to find optimal learning parameter values. As a result, the learning rate parameter is set to $\eta = 0.009$ for the EMLC algorithm, and $\eta = 0.001$ for the MST algorithm, which provided the best average convergence speed when learning a set of 10 input patterns split equally into 5 classes. A momentum acceleration coefficient was also part of the grid search, which was set to $\mu = 0$ for the EMLC algorithm, and $\mu = 0.2$ for the MST algorithm.

4.4.2 Simulation Results

	Number of Patterns
DTA-B	135
EMLC	110
MST	70

Table 3: Learning performance comparison between DTA, EMLC, and MST algorithms.

Simulation results are shown in Table 3. Overall, the DTA-B algorithm achieved 23% better learning performance compared to the EMLC rule, and 93% better compared to the MST algorithm. Since the number of learning epochs are quite limited for each learning rule, this result should not be considered to be indicative of the maximal memory capacity of the algorithms. It is entirely possible that given a very large number of learning iterations, all algorithms here demonstrate similar memory capacity results. However, due to constraints in computational resources we were unable to increase the maximum number of learning

epochs much higher, especially for the MST algorithm which is very computationally expensive.

Instead, the above results indicate that the DTA-B algorithm exhibits better learning convergence speed compared to both the EMLC and MST learning algorithms. When comparing between the DTA-B algorithm and the MST algorithm, the DTA-B algorithm converges in approximately 9 times fewer learning epochs, on average. When compared to the EMLC algorithm, the performance difference is more gradual, and increases with the number of input patterns. With only 5 input patterns in the training set, the DTA-B and EMLC algorithm converges in approximately 11 and 25 epochs, respectively (EMLC is 127% slower to converge). With 50 input patterns in the training set, the DTA-B and EMLC convergence epoch is approximately 25 and 90, respectively (260% difference).

Additionally, we recorded metrics on the number of fall-back weight adjustments made by the DTA-B algorithm during this experiment. On average, the percentage of fall-back updates are 7.2% across all learning trials. Additional trials were run with longer input pattern duration ($T = 100, 150$) and larger number of weights ($N = 1000, 1500$), and we did not observe any significant differences in the number of fall-back updates compared to the original experimental settings.

4.5 Algorithm Runtime

In this section, we investigate the runtime complexity of the proposed DTA-B learning algorithm through numerical experiments. In particular, we are interested in the following research questions:

1. Investigate the overall runtime differences between the DTA-B, EMLC, and MST algorithms.

2. Investigate the computation time required by the Dynamic Threshold procedure in comparison to the original (Gutig 2016).

3. Determine how the Dynamic Threshold runtime changes as the input duration T increases.

4.5.1 Overall Runtime Comparison

In this experiment, we aim to investigate the overall runtime of the DTA-B algorithm in comparison to the EMLC and MNIST methods. The experimental setting utilises randomly generated Poisson input patterns split into five classes, with the same hyper-parameters to that of Section 4.4. In particular, we are interested in measuring the total runtime of each algorithm as the number of inputs in the training set increases. Only learning trials which have successfully converged to a solution are included in the measurements, since we are interested in the runtime to convergence.

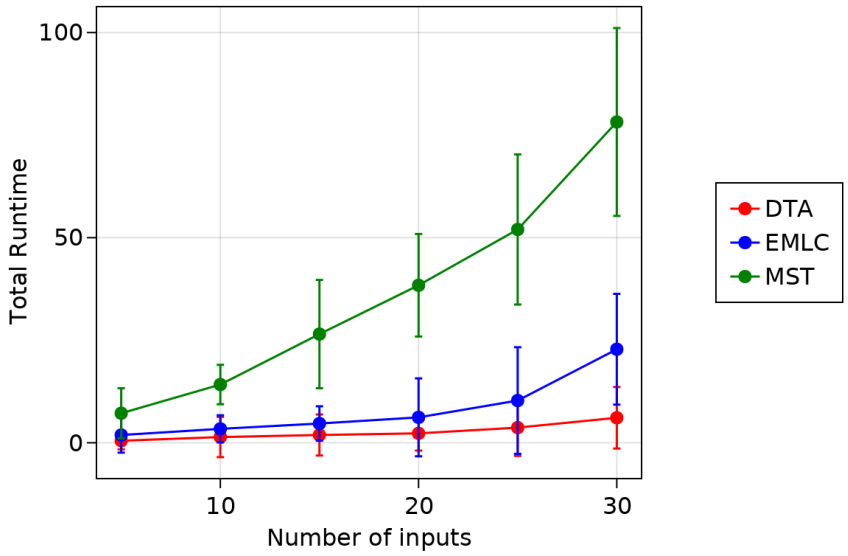

Figure 34: Comparison of the overall runtimes of the DTA, EMLC, and MST learning algorithms in a spike count learning task with 5 input categories. The total number of inputs is shown on the x axis.

Simulation data is demonstrated in Figure 34. Overall, the DTA-B algorithm exhibited the best overall runtime out of the three algorithms. Here, the MST algorithm exhibits average runtime which is 13.7 times slower compared to the DTA-B algorithm, which is consistent throughout the different sizes of the input set. However, we note that the MST rule is not designed for classification tasks involving multiple short input patterns, or necessarily even for fast convergence speed. As such, the performance difference between DTA-B and the EMLC algorithm is a much more useful statistic, especially since our setup matches the

experimental conditions in Li and Yu (2020) reasonably well. When compared to the DTA-B method, the EMLC exhibits runtime which is between 2.4 and 3.8 times slower in terms of runtime, with an average difference of 3 times.

The results shown here establishes that the DTA-B algorithm is capable of outperforming the EMLC algorithm in both overall convergence speed and overall runtime. This is in spite of the fact that the EMLC algorithm has a much simpler weight update, which does not require the computationally expensive Dynamic Threshold step, or an interior-point solver step. In the range of parameter values tested here, the approximate difference in runtime between the two algorithms is consistent throughout, and the data does *not* indicate that as the size of the training set increases, the performance difference becomes smaller. This is in contrast to the results established in Section 3.7 regarding the DTA-B runtime in PTS learning problems, where the DTA improvements become less significant as the input duration increases.

4.5.2 Dynamic Threshold Runtime

In this experiment, we aim to investigate the runtime of the Dynamic Threshold procedure, as described in Section 4.3. Importantly, we will compare the per-epoch runtime of the Dynamic Threshold procedure against that of the original process proposed in Gutig (2016), in order to characterise whether the simplifications made in the proposed process have had an effect on the overall efficiency of the algorithm.

The experimental setup is as follows. In each trial, we randomly generate one Poisson input pattern of rate $\nu_{\text{in}} = 0.005$, and set the target spike count to $|\mathbf{y}| = 5$. The DTA-B and MNIST algorithms are applied to learn this pattern for a single iteration each, and we measure the runtime of the Dynamic Threshold procedure as well as the runtime of the weight update steps in each method.

Simulation data are demonstrated in Figure 35, which shows the differences between the DTA and MST algorithms for various values of the input duration T, for both the runtime of the dynamic threshold step and the weight update step. For the MST algorithm, we observe that similar runtimes are observed for the dynamic thresholding step and the weight update step. For the DTA algorithm, we

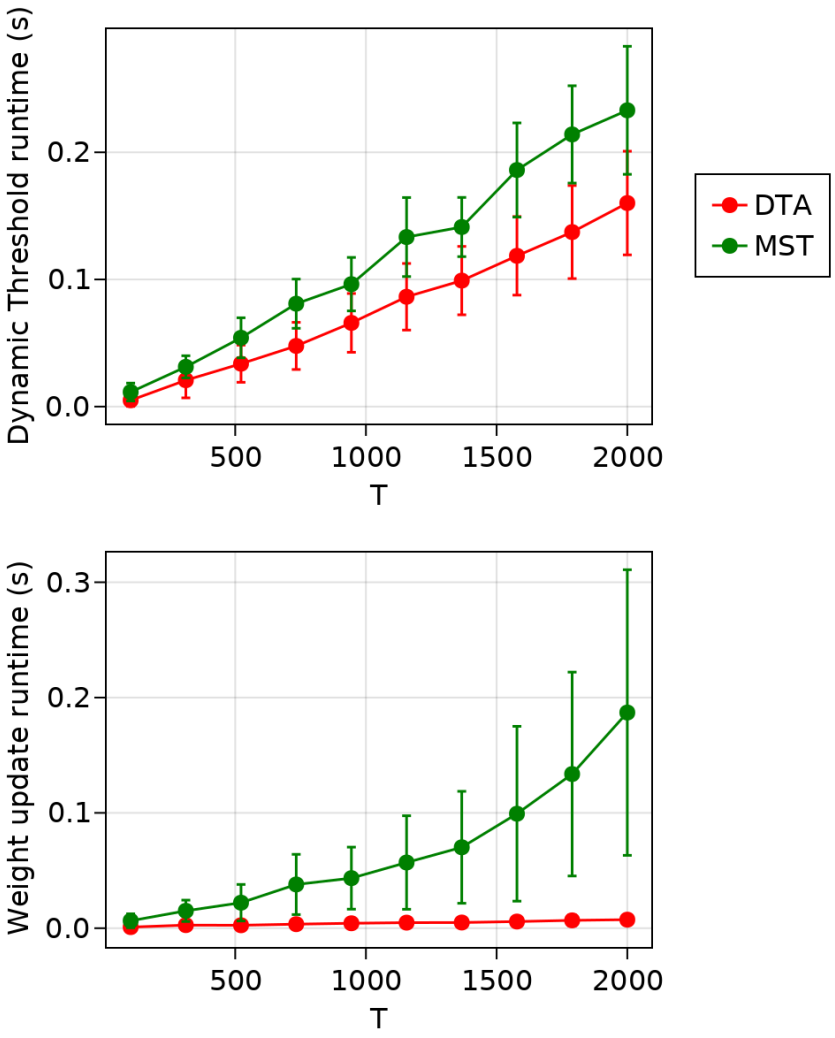

Figure 35: Per-epoch runtime comparisons between the DTA-B and MST learning algorithms. Measurements are performed for both steps in the learning process: Dynamic Threshold and weight update calculations.

observe that the thresholding step constitute a major computational bottleneck of the algorithm, with the weight update step exhibiting near constant runtime over the tested range of T. On average, our Dynamic Threshold process demonstrates a 37% reduction in runtime, compared to the original procedure. However, our process appears to exhibit a linear time complexity, similarly to the original.

4.6 Memory Capacity

Unlike the theoretical results and numerical evidence set out by Memmesheimer et al. (2014) to prove that a maximal learning capacity exists for spiking neurons in the PTS learning task, there are no such results when it comes to learning tasks involving spike counts. Part of the difficulty in establishing this type of result is in the flexible nature of the learning task. There are a number of different approaches to set up a SC classification problem, in which the variables which can vary include the input pattern duration, the number of target output spikes, the number of input patterns in each class, and the number of input classes. Furthermore, this is without considering the architectural parameters such as the neuron time constants, the number of input afferents, and so on. While a theoretical analysis of the overall capacity of SRM neurons to solve SC tasks is beyond the scope of the current work, it is nonetheless an interesting problem to characterise the DTA-B algorithm in further depth, in order to better understand its learning capabilities.

To this end, this section presents an experiment which attempts to measure the ability of the algorithm to solve a specific variant of the SC task. In particular, the learning scenario we are interested in is one where the algorithm learns only one input pattern per input category, but is trained to correctly learn as many input classes as possible. Our aim is to understand the ability of the algorithm to train neurons to precisely generate a large range of output statistics (spike counts), while the average input statistics (input firing rate and input duration) remain constant.

4.6.1 Experimental Setup

The capacity of the algorithm to train neurons in this task is denoted as C_α. We propose to measure C_α as follows. For each measurement, we will run 50 independent trials. In each learning trial, we begin with an input training set of one pattern which has a target spike count of one. If the DTA-B algorithm can train a single SRM neuron on this classification task, then we add a second input pattern to the training set, which has a target spike count of two. All input patterns are generated using as Poisson point processes with spiking rate $\nu_{\text{in}} = 0.005$. Each time the input training set grows, the previously trained neuron is used as the starting point of another training process. This continues until the algorithm does not converge within 100 learning epochs, and the trial ends. When half of all learning trials have failed to converge, the number of input classes in the training set is measured as C_α.

Importantly, the experiment will characterise the learning capabilities of the algorithm with respect to several different parameters by performing a grid search. Each permutation of the parameters result in one measurement. The list of parameters, as well as their search ranges, are shown in Table 4.

Parameter Name	**Search Values**
τ_m	[10, 20, 30]
τ_m/τ_s	[2, 4, 6, 8]
N	[100, 316, 1000, 3162]
T	[10, 32, 100, 316, 1000]

Table 4: Parameter values for C_α measurements.

4.6.2 Simulation Results

Simulation results of C_α measurements are shown in Figure 36. Importantly, in the large N limit ($N = 10000$) we find that the learning capacity C_α can be expressed as an exponentially decaying function of a parameter τ, where $\tau = \sqrt{\tau_m \tau_s}$ is called the PSP correlation time constant. Interestingly, in Memmesheimer et al. (2014) the authors established that the parameter τ is crucial to explain the maximal capacity of SRM neurons in PTS learning tasks.

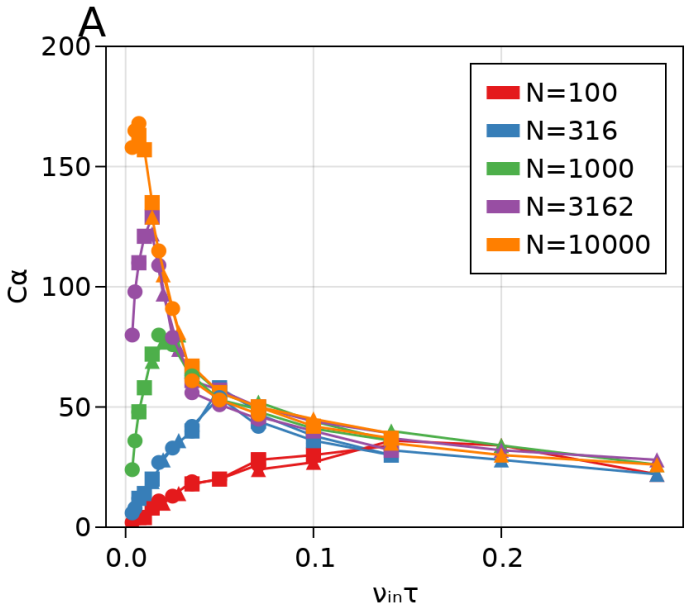

Figure 36: **A**: Decoding capacity C_α, plotted against $\nu_{\text{in}}\tau$ where ν_{in} is the (constant) input spike rate and τ is the PSP correlation time $\sqrt{\tau_m \tau_s}$. In general, C_α can be expressed as an exponentially decaying function of τ . Symbols (circle, square, triangle) respectively represent different series measured with $\tau_m = 10, 20, 40$. The range of the number of synaptic weights N is chosen on the log scale $(10^2, 10^2.5, 10^3, ...)$.

Specifically, we find that C_α decreases with increasing τ. This means τ should be small in order to maximise the capacity, and thus the neural dynamics that maximises C_α in the large N limit is such that the membrane potential is able to rise rapidly upon receiving an input spike, and also decays rapidly afterwards. The neuron then operates in a fashion which is reminiscent of coincidence-detection, where each output spike is only generated by coincident inputs during a very small time window. Furthermore, as τ_m decreases, the time duration of the reset pulse (Equation 12) becomes shorter, which allows the neuron to more easily elicit spikes in quick succession. Additionally, outside of the large N limit the capacity quickly deviates from the behaviour described above. Assuming N is fixed, for τ smaller than a critical value, C_α instead increases with increasing τ. We find that the critical value also forms an exponentially decaying function which depend on the average number of spikes in an input pattern (calculated as $NT\nu_{\text{in}}$), as shown in Figure 36B.

The above result is consistent with the conclusions published in Memmesheimer et al. (2014) for the PTS task, and in Rubin, Monasson and Sompolinsky (2010) for the binary (two-class) SC task. In both of these studies, the memory capacity decreases as τ increases, assuming that N is large. This then suggests that regardless of the specific learning task, or indeed of whether learning is single-spike or multi-spike, the ability for the membrane potential to rapidly respond to new inputs or forget past stimulus is central to the computational capacity of spiking neuron models. However, it is important to reiterate that the above numerical results are valid only for characterising the capabilities of the DTA-B algorithm to train neurons within a limited number of training epochs. This is the main difference between our conclusions and the ones drawn in Memmesheimer et al. (2014) and Rubin, Monasson and Sompolinsky (2010), in that we are not attempting to uncover the computational properties of the neuron in this multi-spike SC learning task, but only of the algorithm.

4.7 Benchmark Performance: UCI Datasets

In this section, the ability of the proposed SC learning approach to generalise from training data to unseen sample is evaluated using two standard benchmark

datasets which can be accessed from the UCI Machine Learning repository (Dua and Graff 2017). Additionally, we will also consider a simplistic option for addressing a significant weaknesses of the DTA-B algorithm, specifically its inability to back-propagate in order to train deeper architectures.

4.7.1 Dataset Descriptions

Fisher's Iris Dataset

The Iris dataset Fisher (1936) is a classic data classification problem involving 150 samples, which are evenly divided into three classes of different species of the *iris* plant: *setosa, vesicolor,* and *virginica*. Two of the input classes are not linearly separable. Each sample consists of four continuous and real-valued features: petal width, petal length, sepal width, and sepal length. This dataset is one of the most commonly used for benchmarking supervised machine learning and neural network algorithms. Of the 150 samples in the dataset, 50% is chosen at random as the training set, and the other 50% is used to measure generalisation performance in the test phase.

Wisconsin Breast Cancer Dataset

The Wisconsin Breast Cancer (WBC) dataset (Wolberg 1992) is a binary data classification problem containing 699 samples split into two different classifications of breast cancer tumour: benign and malignant. The dataset is imbalanced, containing 458 and 241 samples in each respective class. This dataset has been used for analysis of machine learning algorithms, because imbalanced data can cause problems during learning (Mohammed et al. 2020). Note that 16 samples in this dataset contained missing values, which were discarded for a final 683 samples. Each sample consists of 9 discrete-valued features. For training and testing, a 50-50 split is used.

4.7.2 Network Architecture

Two different fully-connected feed-forward SNN architectures are proposed here to solve the above classification tasks: the first is a one-layer network and the second is a two-layer network with a population of classifiers in the hidden layer.

Learning Layers

The one-layer SNN is made up of a single output layer of SRM neurons, where the number of neurons is the same as the number of input classes (3 for Iris and 2 for WBC). Each neuron in the layer is assigned one of the input classes, and if presented with the corresponding input the neuron is trained to generate $|\mathbf{y}|$ output spikes. If the neuron is presented with an input not belonging to its designated class, then it is trained to remain silent. A parameter optimisation was performed to optimise $|\mathbf{y}|$ before learning, which showed that classification accuracy increased with $|\mathbf{y}|$, but does not improve above $|\mathbf{y}| = 10$, so this value is set as the target spike count.

The two-layer SNN is made up of an output layer of SRM neurons which is trained exactly the same as the output layer in the one-layer SNN. The hidden layer contains one population of SRM neurons for each input class, with each neuron in a population also trained to generate $|\mathbf{y}|$ spikes when seeing the designated class, remaining silent otherwise. The basic idea of the hidden layer is to train multiple classifiers, which make the classification boundary more robust in the case that the learning fails to train one of the classifier neurons. For the Iris dataset, the hidden layer contains two neurons in each population (total 6 neurons), and for the WBC dataset there are three neurons for each population (also total 6). Note that the this approach is a rather primitive construction of a two-layer network which we do not expect to scale up to larger datasets. However, it would be interesting to see if training populations of classifiers is effective to improve the generalisation performance.

Since the number of classification neurons is equal to the number of input classes, the classification of the network is represented by the neuron which exhibits the highest number of output spikes during a pattern presentation. This classification is then compared to the label to determine the accuracy. In addition

to this, for the one-layer SNN we also tried a setup with a single neuron learning all three classes, where the neuron is trained to generate 1 spike when seeing the first input class, 2 spikes for the second input class, and so on. However, we find that in this setup the learning performance is (approximately 10%) worse compared to using one neuron to represent each class.

Input Encoding

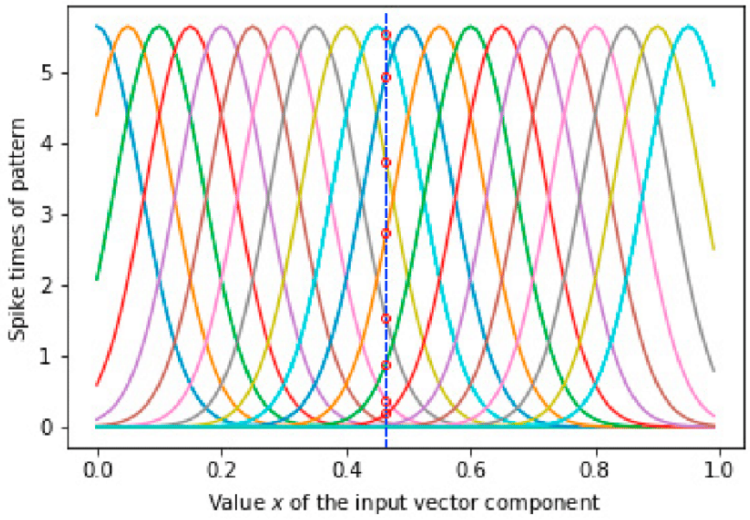

Figure 37: Gaussian population coding. Here, the input numeric value (dashed vertical line) is plotted against a number of overlapping Gaussian functions with varying means (solid lines). The y-values of the intersections between the dashed line and the solid lines are then taken as the (encoded) input spike times. In this way, a single numeric value is encoded as a population of input spikes. Further details of how the Gaussian functions are set up are given in the rest of this section. Figure originally published in Sboev et al. (2018).

Before learning, the real- and discrete-valued features of the datasets must be temporally encoded into spike times. Here, population encoding (Bohte, Kok and La Poutre 2002) with Gaussian receptive fields of the form $G(v, \mu_j, \sigma) = \exp(-(v - \mu_j)^2/2\sigma^2)$, wherein v is the feature value. The basic idea is to encode each input feature into a population of values, which makes the features more linearly separable (Figure 37). Each feature is represented by $M = 10$ identically shaped Gaussian functions centered at:

$$\mu_j = I_j^{\min} + \left(\frac{2i-3}{2}\right)\left(\frac{I_j^{\max} - I_j^{\min}}{M-2}\right), j \in \{1, 2, ..., M\}$$

Here, $I_j^{\max}$ and $I_j^{\min}$ represents the maximum and minimum of the j-th feature, respectively. The spread of the Gaussian functions are determined by:

$$\sigma = \frac{1}{\beta}\left(I_j^{\max} - I_j^{\min}\right)(M-2)$$

Here, $1 \leq \beta \leq 2$ is an adjustment factor, which is set to 1.5 because this value produced the best classification results. As such, each feature value v is converted into M output values $0 \leq G(v, \mu_j, \sigma) \leq 1$, which is then linearly converted into spike times as $t_j = T - G(v, \mu_j, \sigma)T$.

4.7.3 Experimental Setup

The experimental conditions is are as follows. 50 independent trials were conducted for the Iris dataset and 20 independent trials were conducted for WBC, each with a random split of the data into training and testing sets. All networks are trained for a maximum of 20 epochs. On the Iris dataset, learning is performed using the DTA, EMLC, and MST algorithms using both SNN architectures described above, which gives additional data points for analysis of the two-layer approach. On the WBC dataset, the learning accuracy of the DTA-B algorithm is compared with other published results in the literature, in order to provide a point of comparison with state-of-the-art supervised learning methods which are able to train multi-layer networks with back-propagation.

During each trial, the initial weights of the network as well as the order of pattern presentation are controlled across the different methods. Additionally, an early-stopping condition was put in place, where the learning is allowed to stop if classification accuracy on the test set reaches 100%. Before learning, a grid-search parameter optimisation was conducted using only 15 input samples. As a result, the learning parameter values for EMLC and MST are set as $\eta = 0.02, \mu = 0$ and $\eta = 0.009, \mu = 0.1$, respectively. The input pattern duration T was also optimised in the range of $0 < T < 100$, however in this range there were no observable

differences in the training accuracies of any of the learning algorithms.

4.7.4 Classification Results

Iris Classification Performance

Table 5: Training and test accuracy of the proposed method on the Iris flower dataset formulated as a spike count learning problem. Data represents 50 independent trials.

Method	Architecture	Train (%)	Test (%)
DTA-κ_{PSP}	40-3	96.8 ± 2.03	96.1 ± 2.61
EMLC (Li and Yu 2020)	40-3	89.6 ± 6.91	84.8 ± 6.10
MST Gutig (2016)	40-3	89.4 ± 2.62	86.3 ± 4.98
DTA-κ_{PSP}	40-6-3	97.3 ± 1.03	99.1 ± 1.45
EMLC (Li and Yu 2020)	40-6-3	93.6 ± 4.20	90.3 ± 2.42
MST Gutig (2016)	40-6-3	92.4 ± 1.98	91.1 ± 1.28

Training and generalisation accuracies for the Iris dataset are shown in Table 5. Here, the DTA-B algorithm achieves approximately 10% better generalisation accuracy compared to EMLC and MST using the one-layer SNNs. For the two-layer SNNs, the DTA-B algorithm achieves approximately 8% better generalisation accuracy compared to these algorithms, reaching a state-of-the-art final accuracy of 99.1%. We also observed an improvement in terms of convergence speed (Figure 38): to achieve the same level of generalisation accuracy that the DTA-B method achieves after one epoch of training, the EMLC and MST methods both require approximately seven epochs. After the first four training epochs, the DTA-B method achieves better performance than the final accuracy demonstrated by the EMLC or MST algorithms

The Friedman test (Sheldon, Fillyaw and Thompson 1996) was used on the generalisation performance, in order to determine whether or not there is a significant difference between the average ranks of the three algorithms in Table 5 under the null hypothesis. The calculated Q-statistic is $Q = 16.8$, which yields a corresponding p-value of $p = 0.0002$. Thus, we reject the null hypothesis that the three algorithms have no significant differences in generalisation accuracy. The Nemenyi test was used for post-hoc analysis to determine pairwise differences,

which reported significant differences between the DTA-B method, when compared to the EMLC and MST methods (both yielding p-values of 0.001). When comparing the EMLC and MST methods, the statistical difference was not significant ($p = 0.425$).

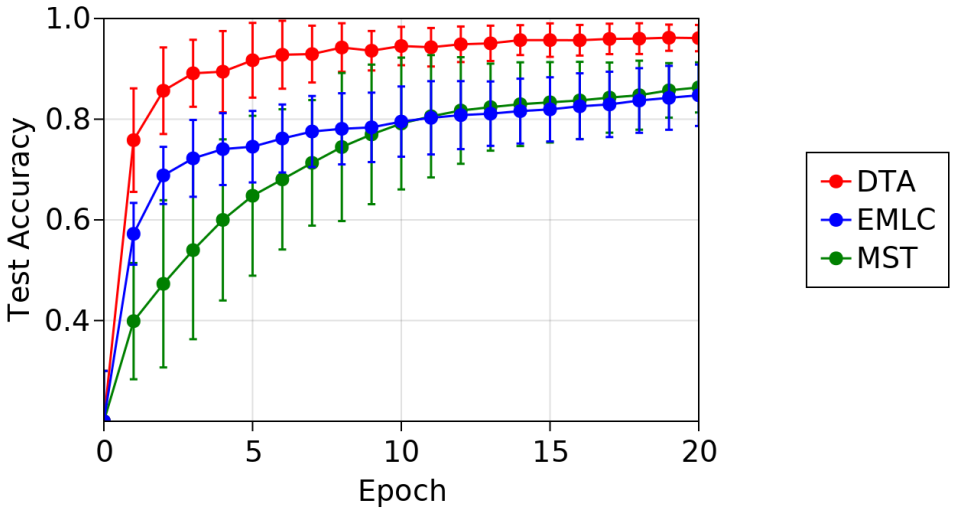

Figure 38: Generalisation speed of shallow networks on the Iris dataset, over 20 epochs of training. Error bars are standard deviations. Each data point represents 50 independent trials.

Notably, in this classification task our one-layer network performance is comparable to two-layer approaches in the literature. In particular, the DTA-B method achieve similar predictive accuracy when compared to Bohte, Kok and La Poutre (2002) (reported 96.1%), Sporea and Grüning (2013) (94%), and Gardner and Grüning (2021) (95.2%). The works noted here all utilise a multi-layer architecture with one hidden layer, containing 9-10 hidden neurons. In addition, Bohte, Kok and La Poutre (2002) and Gardner and Grüning (2021) uses $M = 12$ population input neurons to encode each feature, which is more than our $M = 10$. In terms of convergence speed, the DTA-B method also demonstrates an improvement compared to existing methods. For example, Gardner and Grüning (2021); Tavanaei and Maida (2019); Taherkhani et al. (2018) reported high ($> 94\%$) predictive accuracies after 30, 120, and 100 training epochs, respectively. On average, the one-layer network reaches this performance after the first 8 epochs, and the two-layer network reaches this performance after the first four epochs. It is also

important to note that while some of the works discussed here also performs classification with multiple output spikes, none of them utilise a SC learning setup, which may also be a major factor in the performance difference.

WBC Classification Performance

Table 6: Training and test accuracy of the proposed method on the Wisconsin Breast Cancer dataset formulated as a spike count learning problem. Data represents 20 independent trials.

Method	Architecture	Epochs	Train (%)	Test (%)
DTA	63-2	5	96.6 ± 1.20	96.7 ± 2.00
DTA	63-6-2	5	97.8 ± 0.52	98.0 ± 0.97
(Bohte, Kok and La Poutre 2002)	64-15-2	1500	97.6 ± 0.20	97.0 ± 0.60
Xie et al. (2016)	108-9-2	16	98.0 ± N/A	96.0 ± N/A
Gardner and Grüning (2021)	63-20-2	100	98.4 ± 0.04	97.1 ± 0.08

Simulation results for the WBC dataset are shown in Table 6. A similar trend is observed here, where the one-layer network demonstrates slightly below state-of-the-art classification performance, and the two-layer network demonstrates improved generalisation accuracy in comparison. In terms of convergence speed, both networks achieve above 96% generalisation accuracy after only 5 epochs of training, which is slightly faster than compared to Gardner and Grüning (2021). Additionally, predictive accuracies do not appear to improve significantly, even if training was performed for up to 20 epochs. Increasing the number of neurons in each hidden population of the three-layer network also did not improve performance (tested up to 5 neurons in each population, or 10 neurons total).

4.8 Benchmark Performance: MNIST Dataset

In this section, the proposed SC learning approach is applied to the MNIST dataset of handwritten digits (LeCun and Cortes 2010). With this larger dataset, the initial goal was to investigate whether the two-layer architecture in Section 4.7 will scale up to a more complicated problem. Initial investigations showed that

this was not the case, even with up to 20 neurons per population of classifiers in the hidden layer the method did not demonstrate above 92% accuracy. As such, a different type of 'hybrid' SNN architecture from the literature will be the subject of study here.

4.8.1 Dataset Description

The MNIST dataset of handwritten digits is an overwhelmingly popular problem in the computer vision sub-field of ML. The dataset has the advantages of having samples of relatively small dimensionality, but many such samples, and most importantly there exists an over-abundance of existing ML studies on this dataset, which provides ample sources for comparison purposes. The dataset consists of 60000 training and 10000 test images in gray-scale format of 28 by 28 pixels, split into ten classes labelled from 0 to 9. Both the training and test sets do not have balanced splits between each of the 10 classes.

4.8.2 Network Architecture

Since other methods which are not able to perform back-propagation do exist in the literature, there exists several different frameworks to allow researchers to test these algorithms in multi-layer networks. One relatively recent approach is a hybrid ANN-SNN framework called CSNN (Xu et al. 2018b). In this framework, the SNN learning algorithm is only applied on the (spiking) output layer to perform classification.

The CSNN framework combines traditional CNN with a SNN classifier. More formally, the architecture has two layers of rate-coded neurons, and two layers of spiking neurons. Computation through the network can be decomposed into three parts: feature extraction, temporal encoding, and classification. The technical details of this architecture are as follows:

First, we train a traditional rate-coded CNN, which provides feature extraction capabilities. The CNN only has three layers: a convolutional layer (6C5), a max-pooling layer (2P2), and a fully connected output layer with 10 neurons. We train this CNN using traditional back-propagation with cross-entropy error function for

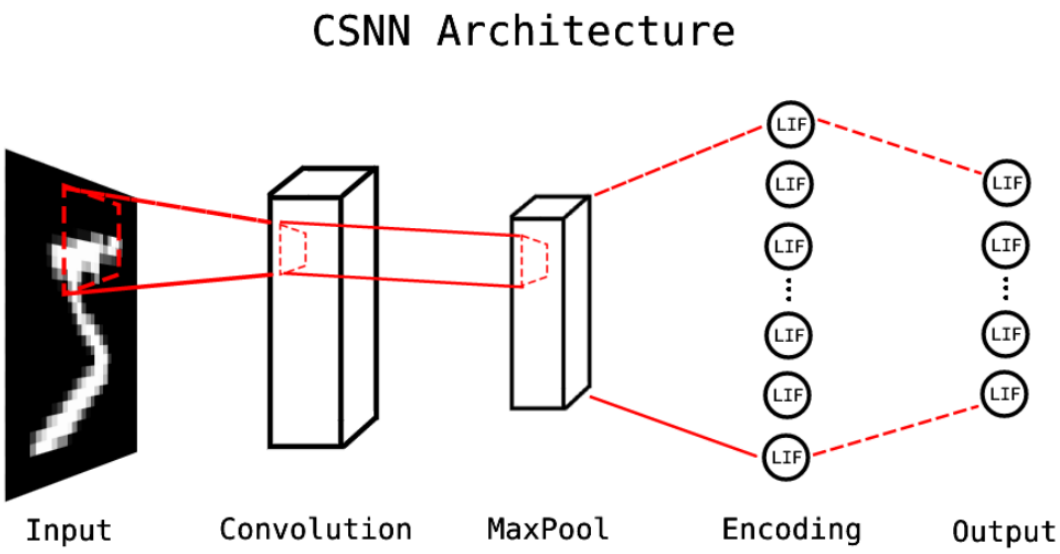

Figure 39: Overview of the CSNN architecture (Xu et al. 2018a). The Convolution and MaxPool layers are composed of rate-coded neurons, while the Encoding and Output layers are composed of spiking neurons. In our setup, the Encoding layer has 864 neurons and the Output layer has 10 neurons.

30 epochs, then the CNN parameters are fixed and the output layer discarded. The resulting partial-CNN model (the convolutional and pooling layer) performs extraction of invariant local feature maps from the input image.

The feature maps produced by the above partial-CNN must be converted into a spike-based encoding. To this end, the real-valued activations of the pooling layer are linearly mapped to spike times in a time window of length $T = 50$. The pooling layer feature maps are flattened to a vector of 864 activation values. We denote the i-th activation value A_i and the corresponding spike time t_i^{spike}. Encoded spike times are calculated as $t_i^{\text{spike}} = T - TA_i$. These timings are then used as spike times for the encoding layer of LIF neurons. Additionally, any encoding neurons with spike time $t_i^{\text{spike}} = T$ (corresponds to $A_i = 0$) do not spike, as their activation is considered too low to induce input spikes.

The encoding layer is fully connected to the output layer, which consists of ten spiking neurons which will be trained. Similarly to the previous section, each neuron is responsible for responding to a 'target' class with $|\mathbf{y}| = 10$ output spikes, remaining quiescent for all other classes. The neuron with the highest number of

spikes decides the classification. All spiking neurons are initialised with $\tau_m = 20.0$ and $\tau_s = 5.0$, with initial weights drawn from a Gaussian distribution with mean 0.01 and standard deviation 0.01.

4.8.3 Experimental Setup

Here, the DTA-B algorithm is compared against the EMLC and MST methods, over 10 independent learning trials. In each trial, the order of pattern presentation and the initial weights of neurons in the output layer are controlled to be the same. Additionally, the same pre-trained CNN layer is used to extract features in all trials. All networks are trained for 20 epochs of the full 60000-10000 split. A learning rate parameter of $\eta = 0.001$ was used for both the EMLC and MST methods and also as the fall-back learning rate for the DTA-B method, which was determined through manual tuning.

4.8.4 Classification Results

Classification results are shown in Table 7. Overall, the DTA-B method outperforms the EMLC and MST methods in both accuracy and convergence speed. Regarding the generalisation performance, the DTA-B method achieves approximately 5% better accuracy compared to EMLC, and approximately 10% better compared to MST. For good generalisation performance, it is often useful to consider early-stopping in order to prevent over-fitting to training data. If we consider a suitable early-stopping point to be within 1% of the best accuracies reported during learning, then the DTA-B method only requires approximately six training epochs, the EMLC method requires ten epochs, and the MST method requires 18 epochs. This means that the DTA-B method reaches convergence faster than compared to the EMLC or MST. Similarly, convergence to a good ($> 90\%$) solution is achieved by the DTA-B method in just 1 epoch, compared to 7 epochs for the EMLC method. The MST method does not achieve this accuracy after 20 epochs of training (Figure 40).

The Friedman test (Sheldon, Fillyaw and Thompson 1996) was used here to determine whether or not there is a significant difference between the average

Method	Train accuracy (%)	Test accuracy (%)
CSNN-DTA	97.57 ± 0.82	97.54 ± 0.44
CSNN-EMLC (Li and Yu 2020)	94.15 ± 0.26	92.27 ± 0.26
CSNN-MST (Gutig 2016)	89.98 ± 4.61	86.91 ± 4.26

Table 7: Performance comparison of CSNN trained with the MST, EMLC, and DTA-B methods on the MNIST dataset. Each data point is averaged over 10 independent trials.

ranks of the three algorithms in Table 7 under the null hypothesis. The calculated Friedman statistic is $F_F = 47.25$. With three treatments (methods) and 10 independent trials, F_F is distributed according to the F-distribution with 3 and 18 degrees of freedom, which for 5% significant level yields the critical value of 3.16. Since F_F is greater than the critical value, the null hypothesis can be rejected and the differences between the three methods is considered significant. The Nemenyi test was used for post-hoc analysis, which reported significant differences in the predictive performance between the DTA-B and EMLC methods (with p-value of 0.0199), and between the DTA-B and MST methods (with p-value of 0.001). These results indicate that the DTA-B method demonstrate statistically significant improvement in generalisation performance compared to EMLC and MST. Additionally, we note that even though the CSNN architecture was fundamentally similar, we did not compare our results with those of Xu et al. (2018a), because they did not train output neurons to fire multiple spikes.

In addition to comparing the DTA-B method with other spike-based approaches, we also obtained results from a traditional non-spiking CNN (dashed line in Figure 40). This CNN has one convolution and one pooling layer, the weights of these layers are initialised to be the same as those layers of the CSNN, and kept fixed throughout learning. The pooling layer is fully connected to an output layer of 10 rate-coded sigmoidal neurons, and we train this output layer weights with gradient-descent with cross-entropy loss. A small (maximum 0.1) amount of uniform noise is added to the CNN loss during training for regularisation. The order of pattern presentation is controlled to be the same as training the CSNN. This setup allows us to directly compare our spike-based method with the traditional ANN training approach. Overall, the DTA-B method demonstrates comparable generalisation accuracy as well as convergence speed to that of the CNN. After

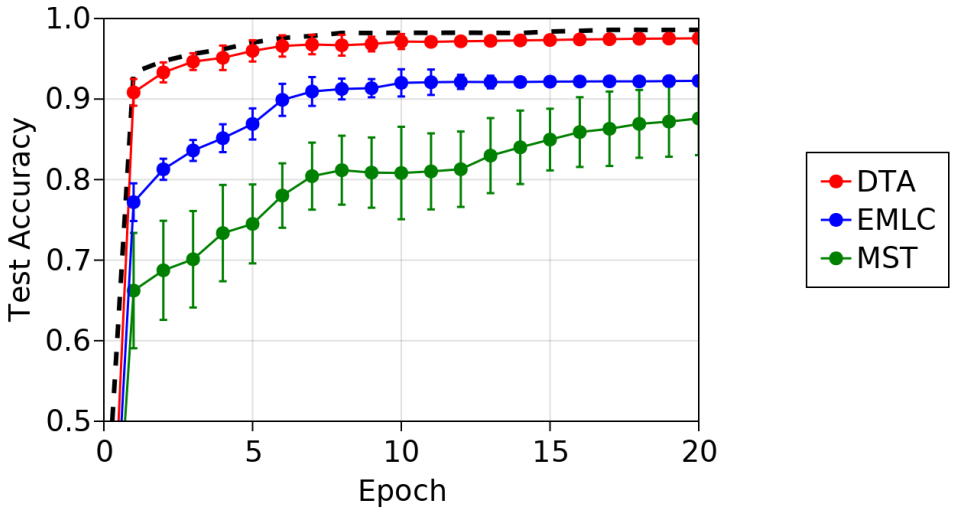

Figure 40: Generalisation accuracy on the MNIST dataset, over 20 epochs of training. Error bars are standard deviations. Each data point represents 10 independent trials. Black dashed lines represents average accuracy of a traditional rate-coded convolutional neural network with the convolution and pooling weights fixed to the same weights which are used in the CSNN.

one epoch of training, CSNN-DTA is approximately 2% worse than the CNN, and this gradually improves to a final difference of 1% after 20 epochs. In comparison, EMLC and MST are approximately 6% and 11% worse than the CNN in terms of final accuracy. We found no statistically significant difference between DTA and the rate-coded CNN in terms of final generalisation performance ($p = 0.507$). These results suggest that our method is competitive with traditional ANN learning, even without any regularisation strategies.

4.9 Discussion

This chapter proposes an extension to the DTA-B method which enables supervised learning where the desired output is a target number of output spikes. Given a target spike count, the approach to solving the supervised learning problem involves two distinct computational steps. In the first step, the Dynamic Threshold procedure (Section 4.3) is used to find appropriate timings for each of the desired spikes. The learning problem now reduces to a PTS learning scenario, and in

the second step a set of solution weights is computed using the DTA-B algorithm (Section 3.3).

In general, the proposed learning procedure demonstrates fast convergence speeds on SC learning problems. The method is compared with the seminal MST Gutig (2016) algorithm, and the more recent EMLC algorithm (Li and Yu 2020), and improved convergence is demonstrated on both synthetic and real-world datasets. Care must be taken to interpret this performance difference only as learning efficiency, and not as *the ability to converge*. That is, given a very large number of training epochs, it is possible that both the MST and EMLC learning algorithms demonstrate similar accuracies as the DTA-B algorithm in real-world classification tasks, or similar maximal memory capacities for synthetic benchmarks.

The most important weakness of the DTA-B algorithm is the inability to train SNNs of more than one layer. In Section 4.7, a simple two-layer architecture was proposed with populations of classifier neurons in the hidden layer, and neurons in both layers are trained identically in a layer-wise fashion. While the two-layer architecture demonstrated a small accuracy improvement compared to the single-layer network, this approach did not scale to the larger and higher-dimensional MNIST dataset, and the performance benefits gained on the IRIS and WBC datasets are also small.

The above weaknesses of the approach are the reason why a hybrid CSNN architecture was chosen for benchmarking the algorithm on the MNIST problem. Here, the experiment is only measuring the ability of the algorithm to train a classification layer, and not its ability to extract visual features from images. This approach is similar to (Li and Yu 2020; Xu et al. 2018b), where the CSNN is used to benchmark other learning algorithms which are limited to single-layer training. While good classification results are reported on the MNIST dataset, it is evident that a fully spiking architecture is preferable since hybrid network architectures pose additional challenges for deployment on neuromorphic hardware.

Chapter 5

Unsupervised Feature Learning with DTA

5.1 Introduction

So far the most significant problem of the DTA and DTA-B algorithms is their inability to train multi-layer neural networks. Because the algorithms do not make use of incremental fixed-size weight updates derived from a loss function, we cannot perform end-to-end back-propagation learning in a similar manner to the current state-of-the-art learning approaches (Zenke and Ganguli 2018; Shrestha and Orchard 2018; Gardner and Grüning 2021). With many recent works in the literature already demonstrating the superior learning performance of multi-layer SNNs in a variety of practical applications (Sengupta et al. 2019; Zhang and Li 2020; Li et al. 2021; Kim and Panda 2021), the drawback of being able to train only single-layer SNNs appear to be very problematic for the applicability of the proposed methods to more complex problem domains.

One simple option to design a multi-layer SNNs with the constraints of our algorithm is to utilise a randomised hidden layer containing a large number of neurons. The idea here is to randomly project the data onto a higher dimension, which may untangle the correlations in the input and make the categories more easily separable. Examples of this approach for both ANN and SNN implementations can be seen in Tapson and van Schaik (2013); Kulkarni and Rajendran

(2018); Cohen et al. (2016, 2017); Boucher-Routhier, Zhang and Thivierge (2021). However, the efficacy of the random projection increases with the number of neurons in the hidden layer, and typically this can be ten times the dimensionality of the input (Tapson and van Schaik 2013). The high degree of parallelism required for efficient simulation of such a wide SNN is difficult to achieve on CPUs (or even GPUs). As such, we will consider alternative options in this chapter.

One promising approach to non-backpropagated learning is to apply the DTA approach to a multi-layer SNN in a layer-wise fashion (Kheradpisheh et al. 2018; Vaila, Chiasson and Saxena 2020). Here, the network layers are trained one at a time in sequence, from the first hidden layer to the output layer. Training for one layer only begins after the previous network layer has finished its training loop. In practice, this approach has been demonstrated to some success by existing STDP-based unsupervised learning algorithms which are also unable to back-propagate errors (Vigneron and Martinet 2020). Utilising unsupervised learning in the hidden layers mean that the learning process can try to automatically discover useful intermediate representations in the data, which can then be propagated to a classification layer. Note that there exists STDP-inspired learning algorithms which are able to perform end-to-end back-propagation training instead of layer-wise learning, for example Tavanaei and Maida (2019). In the case of the DTA approach, layer-wise training of a fully-connected architecture is difficult, because we immediately encounter the non-trivial issue of having to solve for the target output spike times of a hidden layer *before* performing the weight update.

A direction of research which can provide a suitable architecture to the above problems is the study of Convolutional Spiking Neural Networks (CSNNs). Here, the layer-wise training method has demonstrated relatively good success in training CSNNs using layer-wise unsupervised STDP at the convolutional layers. The STDP is applied to extract visual features such as edges or corners from the image in a spike-based manner (Kheradpisheh et al. 2018; Tavanaei, Kirby and Maida 2018; Masquelier and Thorpe 2007; Vaila, Chiasson and Saxena 2020). Typically, the resulting visual features are then fed-forward and classified using a fully-connected output classification layer trained with a supervised learning algorithm. By exploiting the relatively straightforward computation and structure of the computational layers, these approaches successfully design simple and efficient

learning procedures.

5.2 Motivations & Chapter Layout

In this chapter, an extension of the DTA approach called DTA-C is proposed in order to perform unsupervised feature extraction in the convolutional layer of a CSNN network. The proposed algorithm is inspired by existing works in unsupervised STDP learning, and the weight update computation is significantly different to that of the DTA and DTA-B algorithms studied in the previous chapters.

The objectives of the DTA-C algorithm are:

1. Automatically learn a number of convolutional kernels, which are represented by the shared weights in each convolutional map.
2. The learned kernels should each be *selective* towards specific visual features, such as edges or corners of a particular orientation (horizontal, vertical, diagonal).
3. Each convolutional kernel should learn a *unique* visual feature in the input data, instead of all learning the same feature.

The remaining sections of this chapter are organised as follows. In Section 5.3, the CSNN architecture is described in detail. In Section 5.4, the training loop of the DTA-C algorithm is explained. Sections 5.5, 5.6, and 5.7 presents empirical benchmarks of the proposed approach and CSNN on three different image classification datasets: MNIST, EMNIST, and ETH-80. Here, the DTA generalisation accuracies are compared against other layer-wise learning approaches, as well as to other state-of-the-art performances reported on these datasets. In Section 5.8, the effects of various hyper-parameters on the accuracy of the network are examined, which demonstrates the conditions with which the model exhibits the best generalisation performance. Finally, in Section 5.9 we investigate the efficacy of the proposed approach for training CSNNs with two convolutional layers.

5.3 Network Architecture

This section describes the CSNN architecture used throughout this chapter. Here, the CSNN is built primarily using three layers: convolutional, pooling, and output classification. In Section 5.9, an additional, deeper CSNN with five layers is also trained, which contains two convolutional and two pooling layers, in addition to the output layer. Neurons in all layers are modelled as SRM neurons with dynamics set out in Equation 13. The details of each network layer are given below.

Convolutional Layer The convolutional layer is made up of M convolutional maps. Each convolutional map consists of a two-dimensional grid of neurons, with a constant number of neurons in each map. Each map performs a two-dimensional convolution operation similarly to the process described in Section 2.2.5, without any padding. Each convolutional neuron in a map receives input spikes from a $K \times K$ receptive field, with the synaptic weights (the convolutional kernel) shared across all neurons in a map. The receptive fields of adjacent neurons in each map are always maximally overlapped (stride parameter set to 1), such that the target feature can be detected in any location of the input image.

Additionally, each convolutional neuron is only allowed to generate *at most* one output spike for each input image (binary activation). This operational simplification is made for two reasons: firstly, the function of a convolutional neuron is only to determine whether or not a visual feature exists in its receptive field, so a binary output is sufficient. Secondly, restricting the number of output spikes greatly simplifies the training process. This condition can be enforced during simulation by making convolutional neurons undergo a refractory period after spiking, which lasts for the duration of the input pattern presentation.

Pooling Layer The pooling layer is made up of M pooling maps. Each pooling map receives inputs from only a single convolutional map, hence their purpose is implement two-dimensional spatial down-sampling of the convolutional activations. In our network, each pooling neuron propagates only the earliest input spike it receives from a 2×2 receptive field in the corresponding convolutional

layer. The receptive fields of adjacent pooling neurons do not overlap (stride parameter set to 2). As such, 75% of the input information arriving to each pooling neuron is discarded, and the size of the pooling layer is also 75% smaller than the convolutional layer in the vertical and horizontal dimensions. This operation approximates local max-pooling in traditional ANNs, where only the maximal real-valued output activation from the receptive field is propagated forward.

Output Classification Layer The output classification layer contains as many neurons as there are input classes. This layer performs classification with target spike counts. In particular, each neuron is trained to output $|\mathbf{y}|$ output spikes if the input belongs to the designated class, and to remain silent otherwise.

5.4 Method Description

The proposed CSNN is trained in a layer-wise fashion: first the convolutional layer is fully trained using the DTA-C algorithm for a number of training iterations. Once this process has finished, then the output layer is trained using the supervised DTA-B algorithm (Chapter 4). Pooling layers are not trained, since their computation does not require adjustment.

Similarly to the DTA and DTA-B learning algorithms, each learning iteration of the DTA-C method can be split into two main computational steps. In the first step, the learning problem is described in terms of a CSP containing equality and inequality constraints on the membrane potentials. In the second step, the CSP is solved using an interior-point solver, which yields a weight update that satisfies all constraints.

Here, we denote a convolutional neuron at location (r, c) in the d-th map as $n_{r,c,d}$. For learning purposes, we consider $n_{r,c,d}$ as a quadruple:

$$n_{r,c,d} = \{\mathbf{W}^d, \mathbf{X}_{r,c,d}, t^*_{r,c,d}, V(t^*_{r,c,d})\}$$

Here, $\mathbf{W}^d$ denotes the convolutional weight matrix of size $K \times K$ shared by all neurons in map d. The synaptic weight at location (i, j) in this matrix is denoted

as $w^d_{i,j}$. The $K \times K$ matrix $\mathbf{X}_{r,c,d}$ denotes the input spikes sequences within the receptive field of the neuron. An input sequence at location (i, j) in this matrix is denoted as $x^{i,j}_{r,c,d}$. Finally, $t^*_{r,c,d}$ denotes the time of the maximal membrane potential value $V(t^*_{r,c,d})$, which is also the time of an output spike if $V(t^*_{r,c,d}) = \vartheta$ during simulation.

5.4.1 Problem Constraints

A visual feature is a specific pixel arrangement of size $K \times K$, for example edges, lines, or corners. At the beginning of the learning, neurons in a convolutional map may respond to many different visual features, or to no feature in the input images. In order to develop selectivity and uniqueness in the learned convolutional kernels, we now introduce a concept which we call *target neurons*. Fundamentally, a target neuron is a specific neuron in a convolutional map, and in a given iteration each map will learn to respond to the input feature seen by its target neuron (selectivity). In order to define the constraints of the unsupervised learning problem, we first apply an iterative process to select target neurons.

At the beginning of a learning iteration, we first simulate the network with the input image. Then, we select the first target neuron as the neuron with the earliest spike time $t^*_{r,c,d}$ *across all* M *convolutional maps*. Since time is continuous in the neuron model, a unique target neuron can always be selected under this condition. The one exception to this rule occurs when there are no output spikes in any convolutional map. This can happen when initial neuronal weights are too close to zero for any neuron to elicit a spike, for example. In this scenario, the neuron with the largest $V(t^*_{r,c,d})$ is selected, in order for learning to continue. Under these two selection rules, we aim to select the *highest activated* neuron as the target neuron.

After a target neuron $n_{r^*,c^*,d}$ is selected, we then apply a competition mechanism which has been demonstrated in STDP learning (Vaila, Chiasson and Saxena 2020; Kheradpisheh et al. 2018): we *mark* all other neurons in map d, as well as any neurons in other maps which lie in a $K_{\text{inh}} \times K_{\text{inh}}$ spatial window centred on (r^*, c^*). Marked neurons are prevented from being selected as target neurons for

the remainder of the current learning iteration. This competition mechanism ensures that target neurons are spatially separated, which is crucial to encourage different convolutional maps to learn different features in the image (uniqueness).

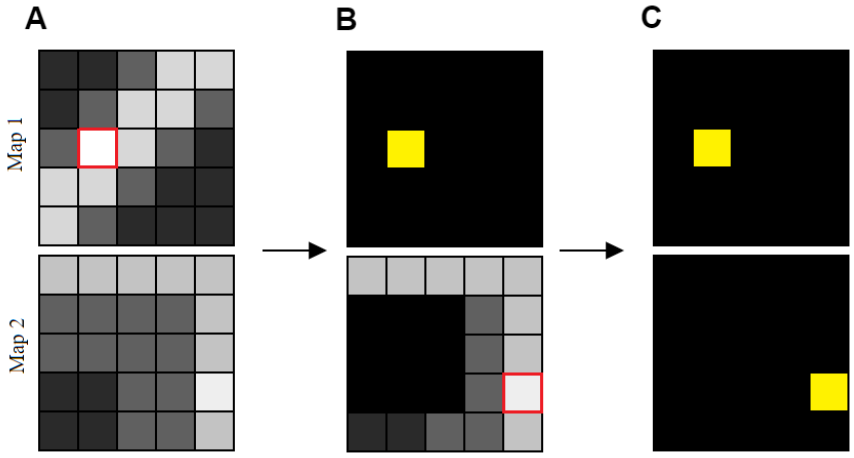

Figure 41: Simplified example of selection and competition steps to select target neurons in the convolutional layer. Here, we assume no neurons spike. Each convolutional map has 25 neurons, and the colour of each location denotes $V(t^*_{r,c,d})$ (lighter is larger membrane potential). Red squares in **A** and **B** denote the neuron at each step with the largest membrane potential maximum in the entire layer. Black cells denote neurons inhibited (marked) by competition.

The selection and competition steps described above are repeated until all neurons in the convolutional layer are either target neurons or marked neurons. Note that due to the competition mechanism, there can be at most one target neuron per convolutional map. We will now convert the learning problem into a set of constraints. To this end, the neurons in a map d are split into two disjoint sets: S^d_{targ}, which is a singleton set consisting of the target neuron; and S^d_{inh}, which contains all other spiking neurons of the map. Formally, these can be written as:

$$S^d_{\text{targ}} = \{n_{r^*,c^*,d}\} \tag{41}$$

$$S^d_{\text{inh}} = \{n_{r,c,d} : V\left(t^*_{r,c,d}\right) = \vartheta, r \neq r^*, c \neq c^*\} \tag{42}$$

The goal of the learning is to adjust the weights $\mathbf{W}^d$ such that the map is more selective towards the feature seen by the target neuron, and less selective towards

features seen by neurons in S^d_{inh}. We can now write the membrane potential constraints as:

$$C = \begin{cases} \vartheta + \epsilon = V(t^*_{r^*,c^*,d}) \\ \vartheta + \epsilon > V(t^*_{r,c,d}), \text{ for } n_{r,c,d} \in S^d_{\text{inh}} \end{cases} \tag{43}$$

Here, $\epsilon \in \mathbb{R}^+$ denotes a parameter which modulates the sparsity of spiking. Setting $\epsilon > 0$ allows non-target neurons which sees similar inputs to the target features to also spike, and we find that this is crucial for good performance. With $\epsilon = 0$, only target neurons may spike, which may lead to information loss due to hyper-selectivity. This means that the learned kernel is too specific, and neurons in the map fails to spike even when the observed feature is very similar to the learned feature. An example is illustrated in Fig. 42.

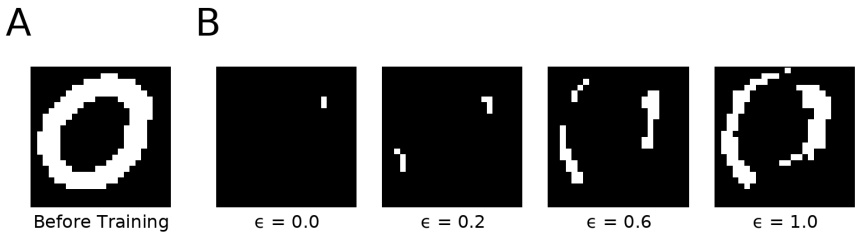

Figure 42: Effect of parameter ϵ on neuron activation in the convolutional layer. White pixels are spiking neurons in a map. **A**: Before learning, the map is selective towards all features in the image. **B**: neuron activation in the map after one learning iteration, with different values of ϵ.

5.4.2 Weight Update Method

With the problem constraints defined in Equation 43, a weight update can be applied in a similar manner to the DTA method (Equation 33). Note that weight updates are only applied to convolutional maps which contain a target neuron. The synaptic update at weight $w^d_{i,j}$ for a convolutional map d is written as follows:

$$\Delta w_{i,j} = \zeta^{\mathrm{d}} \sum_{t \in x^{i,j}_{r^*,c^*,d}} \kappa(t^*_{r^*,c^*,d} - t) + \sum_{n_{r,c,d} \in S^d_{\mathrm{inh}}} \zeta^{\mathrm{o}}_{r,c,d} \sum_{t \in x^{i,j}_{r,c,d}} \kappa(t^{\mathrm{o}}_k - t) \qquad (44)$$

Note that the CSP contains a single ζ^{d} variable corresponding to weight update term for the target neuron, and $|S^d_{\mathrm{inh}}|$ variables $\zeta^{\mathrm{o}}_{r,c,d}$ corresponding to the weight update terms for the marked neurons. In order for the neurons in the map to becomes more selective towards the feature seen by the target neuron $n_{r^*,c^*,d}$, and less selective towards visual features seen by other neurons, we impose additional domain constraints $\zeta^{\mathrm{d}} > 0$ and $\zeta^{\mathrm{o}}_{r,c,d} < 0$. This results in a positive weight update with respect to the feature seen by the target neuron, and a negative weight update otherwise. Unlike the DTA method, a fall-back weight update is not applied for the DTA-C algorithm in the case that the CSP solver returns an infeasible result. This is because the DTA fall-back update is only useful for the learning scenario where the problem data involves only one input pattern, which is not the case for the benchmark tasks presented in this chapter.

5.4.3 Algorithm Summary

A complete summary of the DTA-C algorithm is given in Algorithm 3.

Algorithm 3: DTA-C Algorithm for unsupervised convolutional feature extraction. The algorithm takes in a set of 2D input images $\mathbf{X}$, and trains the M convolutional maps to detect different visual features. $V(t*_{r,c,d})_{\max}$ denotes the maximum membrane potential in the entire layer. The abbreviation 's.t.' stands for 'such that'.

Data: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{\text{maxSamples}}\}$

initialise maps with weights $\mathbf{W}^d$;
shuffle $\mathbf{X}$;
foreach $\mathbf{x}_p \in \mathbf{X}$ **do**
    targetNeurons $= \emptyset$;
    while $V(t*_{r,c,d})_{\max} > 0$ **do**
        compute output spikes;
        if $V(t^*_{r,c,d})_{\max} = \vartheta$ **then**
            $r^*, c^*, d^* = \text{argmin}_{r,c,d} t^*_{r,c,d}$ s.t. $V(t^*_{r,c,d}) = \vartheta$
        else
            $r^*, c^*, d^* = \text{argmax}_{r,c,d} V(t^*_{r,c,d})$
        end
        Set $V(t^*_{r,c,d^*}) = 0, \forall r, c$;
        Set $V(t^*_{r^*,c^*,d}) = 0, \forall d$;
        targetNeurons $\leftarrow n_{r^*,c^*,d^*}$;
    end
end
convert targetNeurons to constraints (Eq. 43);
compute Δw (Eq. 44);
$w^* \leftarrow w + \Delta w$;
$w \leftarrow w^*$;

5.5 Benchmark Performance: MNIST Dataset

In this section, the generalisation performance of the proposed CSNN architecture and training methods on the MNIST dataset (LeCun and Cortes 2010) is investigated, and compared against that of state-of-the-art SNN training methods in the literature. The standard split of 60000 training and 10000 test images are used.

5.5.1 Simulation Protocol

A CSNN with a single convolutional layer is used to solve the MNIST classification task. In total, 10 independent trials were run. For each input image, the greyscale pixel value $s_{i,j}$ at location (i, j) is linearly encoded into a single input spike at time $T - T \times s_{i,j}$. A value of $s_{i,j}$ does not elicit an input spike. The encoded input spikes are then fed into the first convolutional layer for training using the DTA-C learning algorithm. The output layer contains 10 SRM neurons, which is trained to generate a spike count of $|\mathrm{y}| \in \mathbb{N}^+$ using the DTA-B algorithm. During each trial, the output layer is trained for a maximum of 2 epochs.

Due to constraints in simulation resources, a full parameter sweep was not possible. A good set of parameter settings were determined by manual tuning, and the final values used here are shown in Table 8.

Table 8: Parameter settings for CSNNs in the MNIST classification task.

Parameter	**Value**
τ_m	20.0
τ_s (convolutional)	10.0
τ_s (output)	5.0
K	5
$\|Y\|$	10
M	32
T	10.0
ϵ	0.1
K_{inh}	3

5.5.2 Simulation Results

Our generalisation performance is compared here against existing methods in the SNN literature. In particular, there are three types of methods which we are interested in making comparisons to. The first type of method includes methods which apply layer-wise unsupervised and supervised learning, which is most similar to the DTA-C and DTA-B approach. The second type includes one-batch algorithms which were introduced at the end of Section 2.3.3. Following from Tapson et al. (2013), we refer to these algorithms as *neural synthesis* methods. They share some

similar characteristics to the CONE and DTA-B algorithm, including the inability to back-propagate errors, and thus can be valuable for comparison purposes. The final type of methods of interest are end-to-end training methods based on back-propagation, which currently demonstrate the best SNN performance in the MNIST classification task.

Table 9: Generalisation performance on the MNIST image classification problem. Comparisons with state-of-the-art results from several method categories for training CSNNs are provided.

Model	**Learning Method**	**Accuracy (%)**
This work	DTA-C & DTA	98.42 ± 0.7
Tapson and van Schaik (2013)	Neural Synthesis	97.25
Wang et al. (2017)	Neural Synthesis	96.55
Tissera and McDonnell (2016)	Deep Neural Synthesis	99.19
Vaila, Chiasson and Saxena (2020)	STDP & Back-propagation	98.58
Tavanaei, Kirby and Maida (2018)	STDP & Back-propagation	98.60
Lee et al. (2020)	Back-propagation	99.59
Fang et al. (2021)	Back-propagation	**99.72**

Generalisation results averaging 10 independent trials, and comparison to state-of-the-art results from the above three approaches, are shown in Table 9. Overall, our CSNN achieves near state-of-the-art results, while utilising smaller network sizes compared to existing approaches. In particular, we achieve 0.77% lower accuracy compared to the fully-connected deep neural synthesis model proposed in Tissera and McDonnell (2016), however their network contains many more layers (16 compared to our 3). Similarly, our accuracy is 1.3% lower than demonstrated in Fang et al. (2021), but their network is larger, and additionally the neuronal time constants are also trained in their method, which further increases the number of network parameters. When compared to other layer-wise training methods, our results are within 0.2% of the recent models demonstrated in Tavanaei, Kirby and Maida (2018); Vaila, Chiasson and Saxena (2020), which also achieve this accuracy with additional convolutional or fully-connected hidden layers.

Notably, both the DTA-B and DTA-C learning algorithms demonstrated the ability to converge to a good solution while requiring very few training samples. In particular, the results demonstrated here are achieved with only 3 training samples

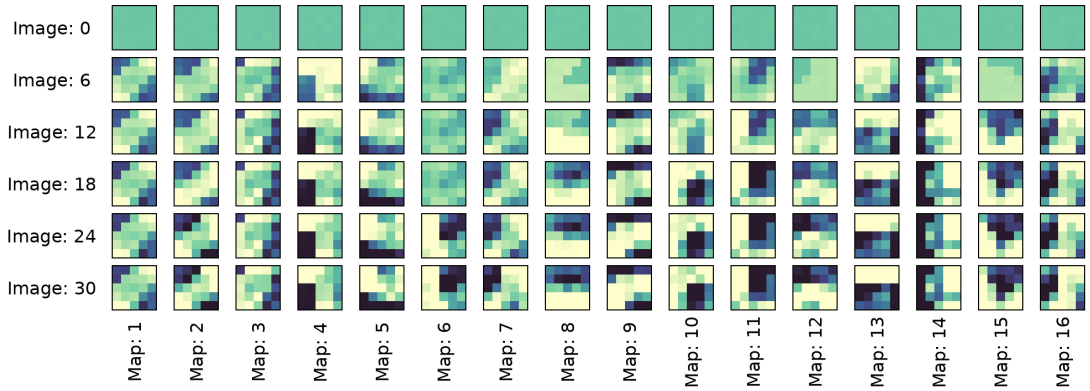

Figure 43: Evolution of 16 convolutional kernels throughout 30 learning iterations of samples from the MNIST dataset.

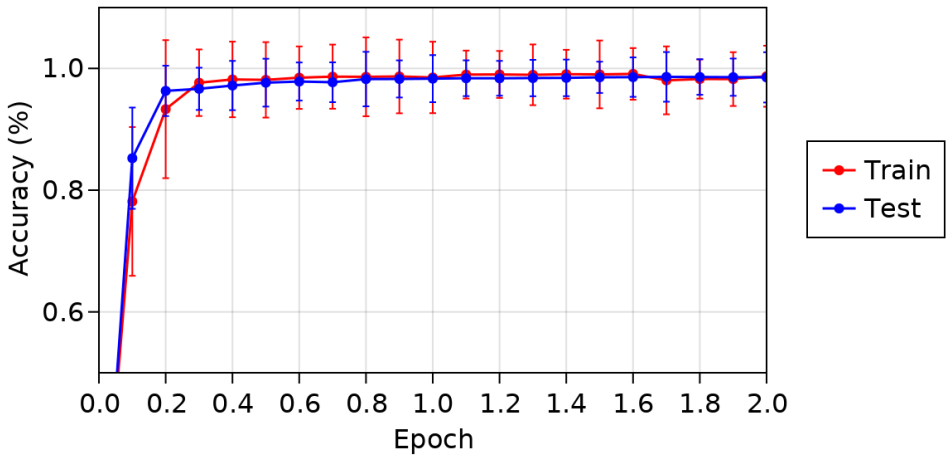

Figure 44: Learning accuracy of the proposed model throughout 2 learning epochs of the DTA-B algorithm on the MNIST dataset.

per input category (30 total training samples) to fully train the convolutional layer with the DTA-C algorithm. An illustration of the learned convolutional kernels throughout the unsupervised learning process is shown in Figure 43, which demonstrates that selectivity is formed in most convolutional maps with as few as 1 training sample per input category. By comparison, the first convolutional layer used in Mozafari et al. (2019) was trained using 100000 samples. A similar trend can be observed with the DTA-B algorithm for training the output layer of the network. In particular, the network converges to a good solution after approximately 40% of the total training set size, with only minimal improvements thereafter (on the order of 1% after 10 epochs).

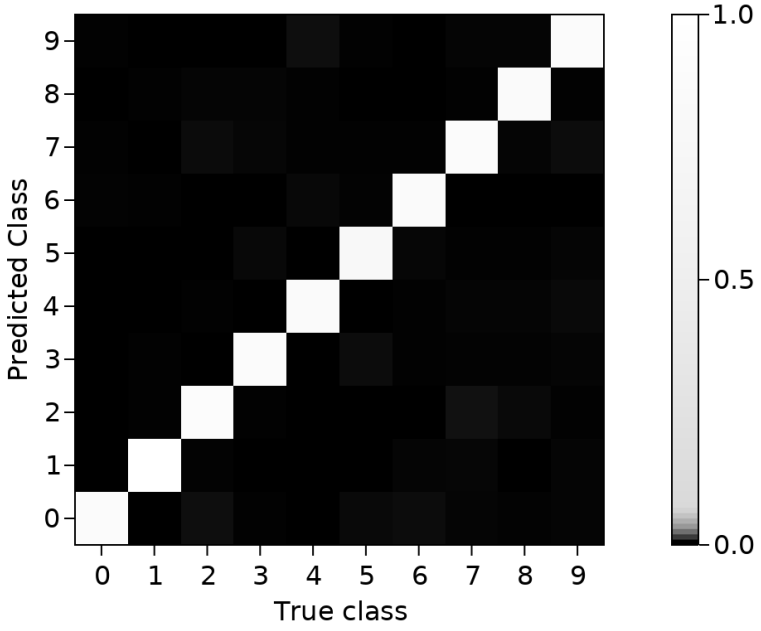

Figure 45: Confusion matrix of a randomly chosen trial of the MNIST dataset. The colour bar shows (logarithmic) colour mappings for the generalisation accuracy of each input category.

The confusion matrix of a randomly chosen trial after learning the MNIST dataset is shown in Figure 45. Results indicate that one of the most problematic input classes is '2', which the network confuses with '0' and '7' for approximately 0.9% and 0.7% of the classifications, respectively. Next, the digit '5' is confused as '0' and '4' for approximately 0.7% and 0.9% of the classifications. Finally, the class '4' is confused as '9' on 0.8% of the classifications. In general, this is

not a surprising result, since the miscategorised categories do share similar visual features, which can be difficult for the CSNN to correctly predict. There are other input classes that are miscategorised, however the rate of incorrect classification for these classes are only around 0.1%.

5.6 Benchmark Performance: E-MNIST Dataset

In this section, we investigate the generalisation performance of the proposed CSNN approach on the EMNIST dataset (Cohen et al. 2017). The EMNIST dataset is an extension of the MNIST dataset to handwritten letters. There are several different realisations of the dataset, for example with only letters or only digits, with balanced or unbalanced classes, but in general the size of the dataset is much larger when compared to MNIST. Unlike the MNIST dataset, this benchmark is more suitable as a stress test for the performance of the CSNN, because the higher problem complexity should pose a significant challenge for a network containing only three layers.

5.6.1 Simulation Protocol

Here, we use the balanced version of the dataset with 47 balanced input categories, containing digits, lowercase letters, and uppercase letters in the English alphabet. Note that this version of the dataset combines certain input categories due to their visual similarities, specifically the uppercase and lowercase versions of the letters ‘i’, ‘j’, ‘k’, ‘l’, ‘m’, ‘o’, ‘p’, ‘s’, ‘u’, ‘w’, ‘x’, ‘y’, ‘z’. For training and testing, we use the standard 112800-18800 training and testing split.

Parameter settings for the EMNIST task are shown in Table 10. Otherwise, the network architecture and experimental setup is the same as in Section 5.5.1.

5.6.2 Simulation Results

Generalisation performance averaged over 10 independent trials are shown in Table 11. Overall, the DTA methods demonstrate a competitive level of accuracy when compared to the state-of-the-art results achieved by existing CSNNs in the

Table 10: Parameter settings for CSNNs in the EMNIST classification task.

Parameter	**Value**
τ_m	20.0
τ_s (convolutional)	10.0
τ_s (output)	5.0
K	5
$\|Y\|$	10
M	64
T	10.0
ϵ	0.05
K_{inh}	3

Table 11: Generalisation performance on the EMNIST image classification problem. Comparisons with state-of-the-art results of other convolutional neural networks in the literature are provided.

Model	**Learning Method**	**Accuracy (%)**
This work	DTA-C & DTA	84.36 ± 0.3
Cohen et al. (2017)	Neural Synthesis	78.02
Vaila, Chiasson and Saxena (2020)	STDP & Back-propagation	85.35
Jin, Zhang and Li (2018)	Back-propagation (SNN)	**85.41**
Shawon et al. (2018)	Back-propagation (ANN)	**90.59**

literature. In particular, the reported accuracies are within 1% of the CSNN model reported by Vaila, Chiasson and Saxena (2020), which also performs layer-wise training. However, a direct comparison of these results are difficult due to the differences in experimental setup. In particular, the network used by Vaila, Chiasson and Saxena (2020) utilises fewer convolutional maps, however they have an additional hidden layer of 1500 neurons which is fully-connected to the output layer and trained by back-propagation. It is worth noting that the learning performances demonstrated by the spiking models in Table 11 are all lower than current state-of-the-art results for traditional convolutional ANNs, which have demonstrated above 90% accuracies (Shawon et al. 2018).

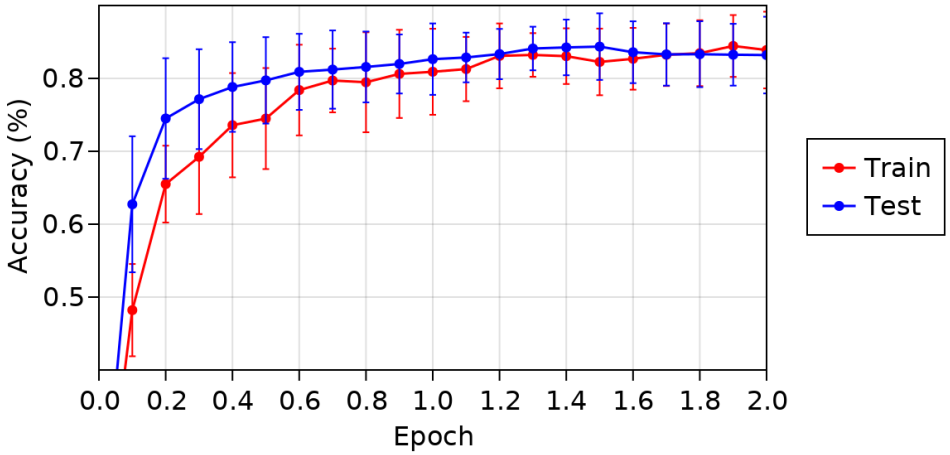

Figure 46: Learning accuracy of the proposed model throughout 2 learning epochs of the DTA-B algorithm on the EMNIST dataset. Note that the accuracies reported at partial epochs (for example 0.2 of an epoch) were measured using the complete training and testing set. This is in contrast with traditional experimental design where the accuracy is reported only at the end of a complete epoch. However, we wanted to obtain additional data points since the DTA-B algorithm is only run for 2 epochs total.

In terms of convergence speed, the DTA-C method manages to fully train the convolutional layer using only one input sample per category, or 47 training samples in total. This can be considered a significant improvement when compared to the 6000 training samples used for STDP training in Vaila, Chiasson and Saxena (2020). The average convergence speed for the DTA-B algorithm on this dataset is illustrated in Figure 46. Here, we observe that the best performance is reached

at approximately 1.5 training epochs, and does not improve significantly after this point (tested up to 10 epochs). The convergence speed is here is much slower than was demonstrated for the MNIST dataset, which is expected given the higher difficulty of the EMNIST problem.

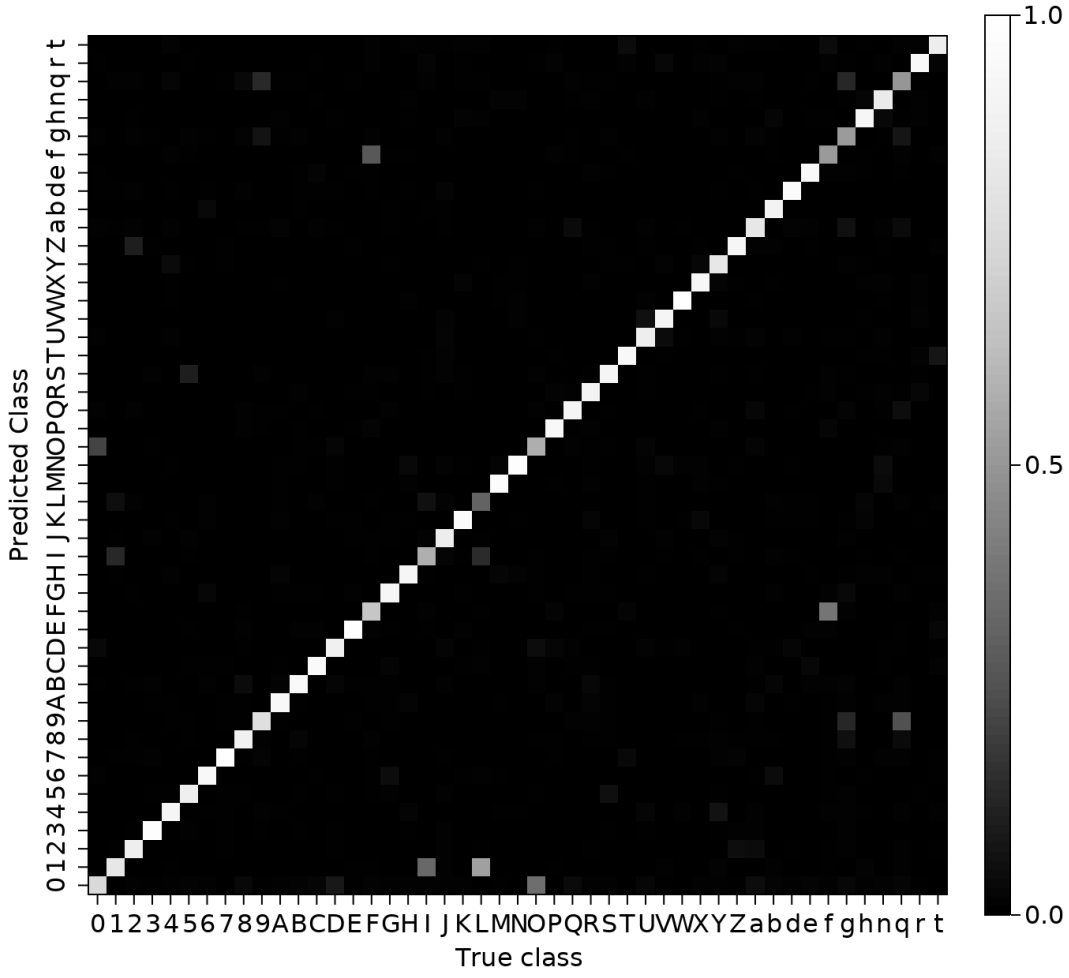

Figure 47: Confusion matrix of a randomly chosen trial of the EMNIST dataset. The colour bar shows colour mappings for the generalisation accuracy of each input category.

The confusion matrix of a randomly chosen learning trial is shown in Figure 47. When compared to the MNIST problem, we observe that the CSNN had difficulty separating several different input categories in this class. The most difficult class for the model was the letter 'L' with 30% accuracy, which is misclassified as the

digit '1' for approximately 52% of all predictions, and misclassified as the letter 'I' for approximately 13.75% of predictions. The another commonly misclassified class was the letter 'I' with 57% accuracy, which is misclassified as the digit '1' for 33% of all predictions. Overall, there are only 7 input categories which demonstrate below 70% classification accuracies, which interestingly are all letter-based classes: 'F', 'I', 'L', 'O', 'f', 'g', and 'q'.

5.7 Benchmark Performance: ETH-80 Dataset

In this section, we investigate the generalisation performance of the proposed CSNN approach on the ETH-80 dataset (Leibe and Schiele 2003). This dataset contains photographs of eight different object categories: 'apple', 'car', 'toy cow', 'cup', 'toy dog', 'toy horse', 'pear', and 'tomato'. There are 10 different object instances of each classes. In addition, each individual object is photographed from 41 different viewpoints from various angles and tilts. As such, there are in total 3280 samples split evenly into the eight input classes.

The ETH-80 dataset is used here to investigate two different capabilities of the algorithm. The first aim is to determine whether the training algorithm can generalise well in the low data regime where the number of training samples is much lower than with the MNIST or EMNIST datasets. The second aim is to investigate the ability of the model to handle classification tasks with extreme intra-class variability, which in this case refers to the large variations in viewpoints for the images in each category.

5.7.1 Experimental Setup

Five object instances of each input category (and the corresponding 41 samples of this each object) are randomly chosen for the training set, and the other half are used for the test set. This is similar to the setup used in (Kheradpisheh et al. 2018). Exploratory simulations were also run for the scenario where the train-test split is chosen uniformly random from the 3280 samples, however we find that the generalisation accuracy is reduced by approximately 1.5% in this scenario, compared to randomly choosing 5 object instances.

Since the input images in ETH-80 are colour photographs, the input encoding method used here is different to the linear encoding used for the greyscale images in MNIST and EMNIST. Firstly, the colour images are transformed into greyscale, and resized to 64×64 pixels. The main reason for this is because much of the input information in a simple photograph dataset of this scale are redundant for object recognition purposes. Instead, one possible approach is to reduce the training samples to only the information about their *contrast values*. To this end, following from (Vaila, Chiasson and Saxena 2020; Kheradpisheh et al. 2018; Kheradpisheh, Ganjtabesh and Masquelier 2016) the images are spatially convolved with Difference of Gaussian (DoG) filters. There are two DoG filters (ON- and OFF-centre filters), which are given by:

$$K_{\sigma_1,\sigma_2}(i,j) = \begin{cases} \frac{1}{2\pi\sigma_1^2} e^{\frac{-i^2+j^2}{2\sigma_1^2}} - \frac{1}{2\pi\sigma_2^2} e^{\frac{-i^2+j^2}{2\sigma_2^2}}, \text{ for } -3 \leq i,j \leq 3 \\ 0, \text{ otherwise} \end{cases} \tag{45}$$

The ON-center filter is generated with $\sigma_1 = 1, \sigma_2 = 2$, and the OFF-center filter is generated with $\sigma_1 = 2, \sigma_2 = 1$. The result of convolving the input image with each filter are two images which respectively contain positive and negative contrasts. An example of a convolved ETH-80 image is shown in Figure 48. The resulting contrast values are then linearly converted into spike time delays for each input channel, with a pixel value of 0 not generating an input spike. The two input matrices are concatenated along the depth dimension and fed into the CSNN. As such, each kernel in the convolutional layer are of size $K \times K \times 2$. The approach above reduces the 64×64 input pixels in an image to an average of only 700 input spikes, which greatly reduces the computational burden of simulation.

Parameter settings for the ETH-80 task are shown in Table 12. Otherwise, the network architecture and experimental setup is the same as detailed for the MNIST task (Section 5.5.1).

5.7.2 Simulation Results

Generalisation performance, averaged over 10 independent trials, are shown in Table 13. When compared to other layer-wise training methods based on STDP,

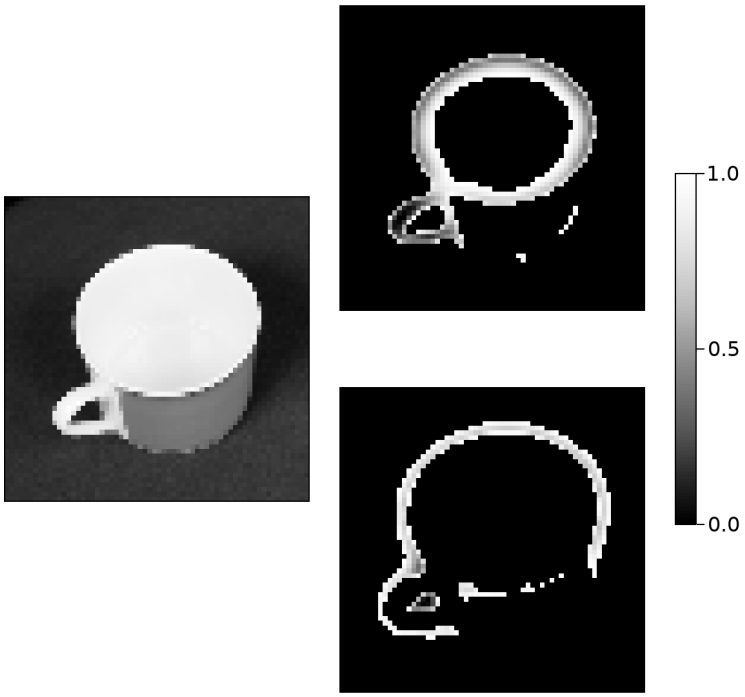

Figure 48: DoG encoding of an ETH-80 image of the 'cup' input category. Left: the original input image, transformed into grayscale. Right: the ON- and OFF-center filtered images. Colorbar represents pixel intensity value.

Table 12: Parameter settings for CSNNs in the ETH-80 classification task.

Parameter	**Value**
τ_m	20.0
τ_s (convolutional)	10.0
τ_s (output)	5.0
K	5
$\|Y\|$	10
M	16
T	100.0
ϵ	0.1
K_{inh}	9

Table 13: Generalisation performance on the ETH-80 image classification problem. Comparisons with state-of-the-art results of other CSNNs in the literature are provided.

Model	Learning Method	Accuracy (%)
This work	DTA-C & DTA	82.9 ± 1.1
Kheradpisheh et al. (2018)	STDP & SVM	82.8
Zhou et al. (2020)	STDP & SVM	82.9
Shi et al. (2021)	Back-propagation	**88.40**

the proposed model achieves competitive performance with a smaller architecture. Both CSNN models proposed by Kheradpisheh et al. (2018) and Zhou et al. (2020) contain multiple convolutional layers, in order to reach the same performance level demonstrated here. However, we our generalisation performance is approximately 6.34% lower compared to the end-to-end back-propagation method demonstrated in Shi et al. (2021). However, we note that they use a 80-20 percentage split for training and testing (as opposed to our 50-50 split), which may have contributed to this result. In addition, the convolutional layer in our CSNN was trained with 3 training samples per category (24 samples total). This is a significant improvement compared to the 40 samples per category used in (Kheradpisheh et al. 2018).

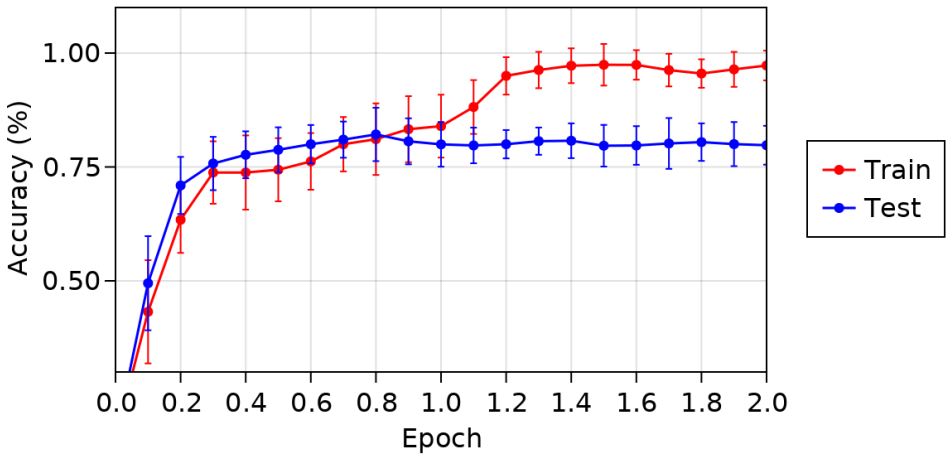

Figure 49: Learning accuracy of the proposed model throughout 2 learning epochs of the DTA-B algorithm on the ETH-80 dataset.

The training and generalisation accuracies of the model on the ETH-80 dataset throughout 2 epochs of training is shown in Figure 49. Here, the proposed training

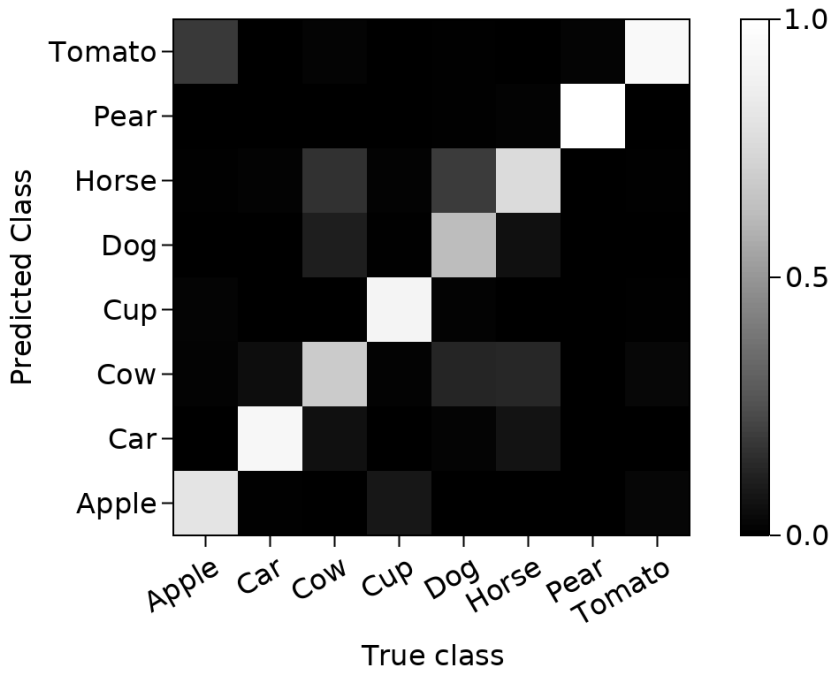

Figure 50: Confusion matrix of a randomly chosen trial of the ETH-80 dataset. The colour bar shows colour mappings for the generalisation accuracy of each input category.

method reaches the maximal accuracies at approximately 40 to 80% of a the first epoch. Additionally, we observe that the test accuracy at the end of 2 epochs is approximately 15% lower compared to the training accuracy, which means the model is significantly over-fitting to the data. Interestingly, this was not observed for the MNIST and EMNIST datasets, which could be due to the low-data regime that the learning process is operating in.

Additionally, we note that in Figure 49 the model appears to be over-fitting to the training data, as the training accuracy continues to increase while the test accuracy does not. In the traditional sense, over-fitting should cause the test accuracy to continue to decrease as training progresses. However, this assumption is difficult to make here because we have only obtained data points up to 2 epochs of training. Nonetheless, a slight accuracy test accuracy decrease is observed between 0.8 and 1.0 of an epoch of training.

The confusion matrix of a randomly chosen learning trial is shown in Figure 50. After the first 10% of a training epoch, the only two classes which demonstrate below 70% accuracies are 'toy cow' and 'toy horse' categories. While this suggests that the model can generalise well without many training data samples, the accuracies for these classes did not improve significantly past this point. Additionally, the class 'toy dog' demonstrates generalisation accuracy below 80%. These three classes are misclassified as each other: for example, the class 'toy cow' is misclassified as both 'toy horse' and 'toy dog' for approximately 10.5 and 15.6% of all classifications. Similar ratios of misclassifications are observed with the other two classes. Interestingly, further analysis of the misclassified samples reveals that the majority of them have front-back or top-down viewpoints. This is inline with the results published in Kheradpisheh et al. (2018), which noted specific angles in ETH-80 with similar outlines for inter-category samples.

5.8 Effects of Parameters

In this section, the effects of various learning and network parameters on the learning performances of the proposed CSNN model. Due to constraints in computational resources, we performed partial scans of the sparsity parameter ϵ, the number of convolutional maps M, the number of training samples for DTA-C, the

input duration T, and finally the target spike count $|y|$ for the DTA-B algorithm. During simulation, each of these parameters were varied while all other parameters were kept constant to their basic values shown in Tables 8, 10, and 12 for the MNIST, EMNIST, and ETH-80 tasks, respectively.

Overall, we find that all of the above have significant impacts on the learning performance of the model. In particular, there are optimal value ranges for ϵ, T, and $|y|$. However, the performance was not sensitive to small variations around their optimal values. For ϵ and $|y|$, similar optimal ranges are observed for all three datasets: $\epsilon \approx 0.1$ and $|y| \approx 10$. This is not the case for the input duration T: for MNIST and EMNIST a smaller input duration ($T \approx 10$) results in better performance, however for ETH-80 the optimal value is observed with $T \approx 100$. This is an interesting result, because in the regime of $T \approx \tau_m$ observed as optimal for MNIST/EMNIST, all layers of the network are performing approximately as perfect integrators, utilising only the spatial information of input spikes occurring in quick succession.

Interestingly, we find that using a large number of training samples for convolutional learning can adversely impact performance. This means that *over-training* occurs in our DTA-C algorithm, and thus an early-stopping criterion is crucial. This phenomenon has also been observed in other unsupervised learning methods Kheradpisheh et al. (2018); Vaila, Chiasson and Saxena (2020), and is particularly problematic for larger datasets where it is difficult to quickly verify the final accuracy. However, we may be able to detect over-training in our approach without training the output layer. On these datasets, we find that an optimally trained convolutional layer typically exhibits a bimodal distribution of weights with the peaks occurring on both sides of zero. Instead, an over-trained layer has a unimodal distribution typically with a negative-valued peak and a long positive tail. Intuitively, over-trained convolutional maps can be thought of as being too selective, with many regions of an input image not eliciting a spike in any layer (Fig. 52).

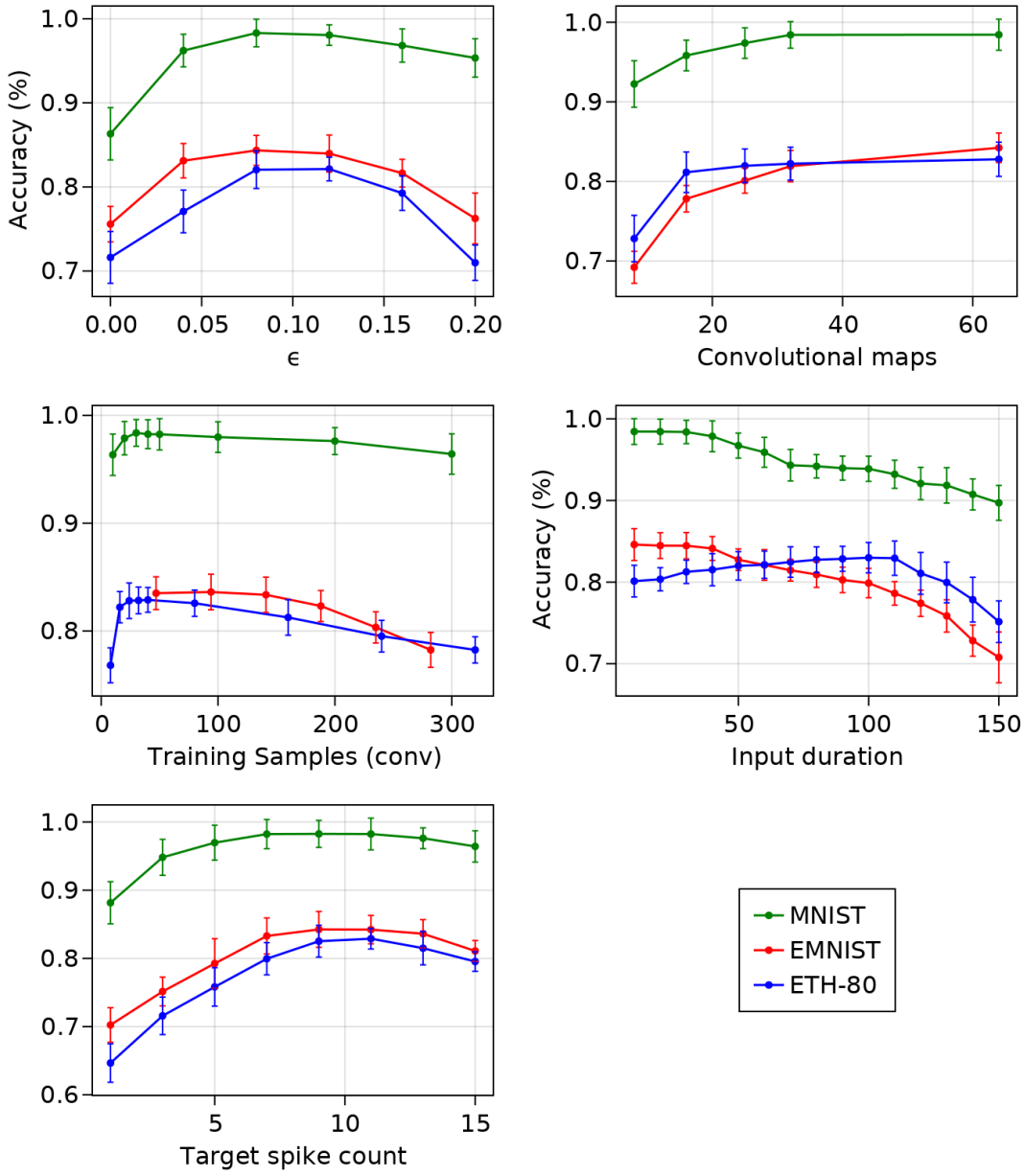

Figure 51: Effect of various parameters on the learning performance of the CSNN model on the MNIST, EMNIST, and ETH-80 classification tasks. Each data point represents five independent trials.

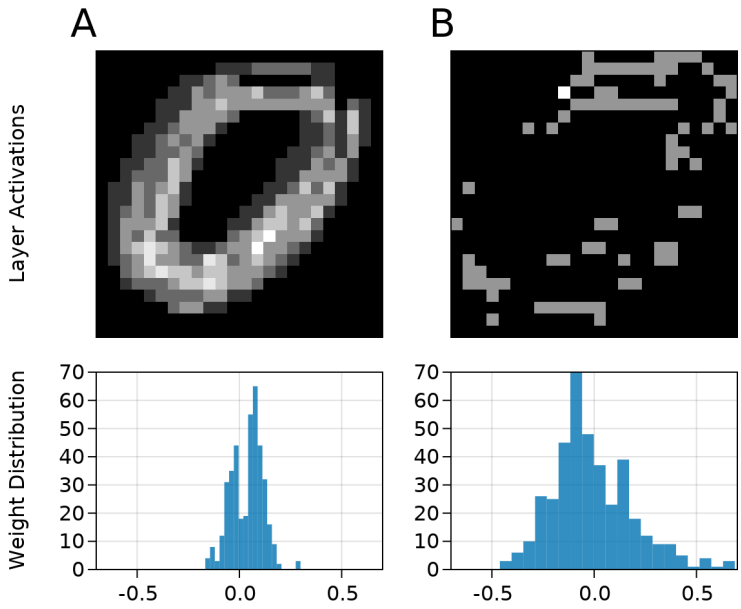

Figure 52: Effect of over-training in convolutional learning with DTA-C. Top row shows the spike activation on an MNIST image in the convolutional layer, bottom row shows the weight distribution of the convolutional layer. **A**: An optimally-trained convolutional layer, trained with 30 images. **B**: an over-trained convolutional layer, trained with 500 images.

5.9 Learning Performance of Deeper CSNNs

So far, benchmark simulations have shown that the DTA-C learning algorithm is a viable approach for training multi-layer CSNN architectures in image classification problems. However, the capabilities for layer-wise training methods to train multiple layers of convolutional feature extractors have been demonstrated by (Kheradpisheh, Ganjtabesh and Masquelier 2016; Kheradpisheh et al. 2018). The ability of the method to learn a hierarchical series of internal representations is an important consideration when solving more complicated learning tasks. Given that the proposed architecture does not manage to perfectly solve the EMNIST and ETH-80 classification problems, in this section we investigate the ability of the DTA-C algorithm to perform layer-wise training of deeper CSNNs with two convolutional layers.

5.9.1 Experimental Setup

The CSNNs now contain two convolutional layers, each followed by a pooling layer, and finally an output layer. The network architecture and parameters are otherwise exactly the same as described in Section 5.3. Due to constraints in simulation resources and the sizes of the datasets, a complete parameter scan was unrealistic. As such, the learning and network parameters are kept the same as described in Table 8, 10, and 12. The number of maps in the two convolutional layer in each network are also kept constant. This means for MNIST, the network contains two convolutional layers consisting of 32 maps each. Note that this is done to save computation time, and is in contrast to the typical approach wherein the subsequent convolutional layers are larger than previous ones (for example see Shrestha and Orchard (2018); Kheradpisheh et al. (2018)).

Initial simulations showed that the small number of training samples which were sufficient for training the first convolutional layer did not achieve good results when applied to the second convolutional layer, even though both layers are of the same size. Here, the number of training samples was manually tuned using 10% of of the available training samples in each datasets. As a result, we train the second convolutional layers for 15, 10, and 20 training samples per input category, for the

MNIST, EMNIST, and ETH-80 datasets respectively. This is not surprising, as Kheradpisheh et al. (2018) has previously suggested that subsequent convolutional layers in CSNNs can converge more slowly compared to earlier ones. Experimental conditions are otherwise identical to the previous sections. In each trial, the initial weights of the first convolutional layers and output layers are kept to be the same across the three- and five-layer CSNN models.

5.9.2 Simulation Results

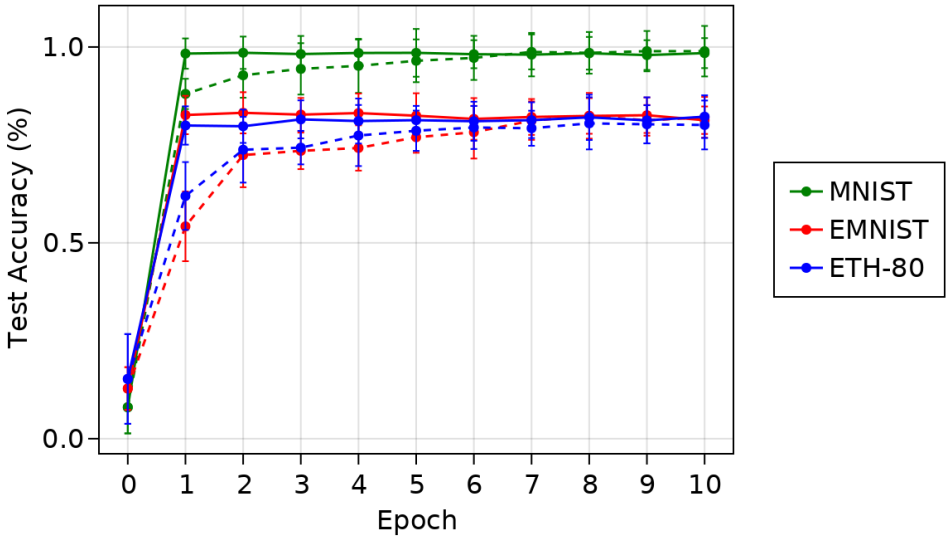

Figure 53: Generalisation accuracies of CSNN model with 1 convolutional layer (solid lines) and 2 convolutional layers (dashed lines) on the MNIST, EMNIST, and ETH-80 datasets

Simulation data are shown in Figure 53. Overall, the results are inconclusive. After 2 training epochs of the DTA-B algorithm, the five-layer CSNN achieved an approximately 5.7% lower generalisation accuracy in the MNIST task, 9.3% lower accuracy in the EMNIST task, and 6.5% lower accuracy in the ETH-80 task, when compared to the three-layer models. After 10 training epochs, we find that the five-layer CSNN achieved a 0.49% improvement in generalisation accuracy on MNIST, and a 0.64% improvement on the EMNIST dataset. However, we also observed a 1.2% *reduction* in accuracy on the ETH-80 dataset. As such, while

the final generalisation accuracies between the three- and five-layer networks are similar, the convergence speed of the five-layer models are much slower on average.

In general, we consider that the deeper CSNNs trained by DTA-C have not demonstrated sufficiently improved performance when compared to our previous three-layer models. Out of the three benchmark datasets, the models here have only demonstrated a marginal improvement in generalisation accuracies for the MNIST and EMNIST tasks. Furthermore, the additional convolutional and pooling layers have increased the simulation requirements, additional hyper-parameters, and slower convergence speeds are observed during learning.

Fundamentally, it is possible that the deeper network achieves better learning performance than demonstrated here, if a full parameter optimisation was performed for hyper-parameters in the second layer. The relevant parameters to optimise are the number of convolutional maps, the neuronal time constants, and the sparsity parameter ϵ. However, it is important to note that if the inclusion of a second convolutional layer requires a separate parameter optimisation to that of the first layer, then the same procedure will be necessary for a third convolutional layer, and so on. This evidently has implications on the scalability of the proposed approach to more complex problems.

5.10 Discussion

In this chapter, an extension to the DTA supervised learning approach was introduced in order to perform unsupervised feature extraction in Convolutional SNNs, called DTA-C. The proposed method successfully trains multi-layer SNNs to reach near state-of-the-art generalisation performance on the MNIST, EMNIST and ETH-80 image classification benchmarks. In general, the method is competitive against other layer-wise training algorithms in the literature, however the performance is not as good when compared to end-to-end training methods which perform gradient-descent by back-propagation. The important benefit of the DTA-C method is that the number of training samples required to achieve good performance is relatively low compared to STDP-based approaches. In this sense, the method may perform very well in low-data learning regimes.

The computation of the DTA-C algorithm is fundamentally similar to the previously proposed methods, in that each learning iteration first requires the specification of a set of linear constraints on the membrane potential of the neuron, and subsequently the weights are computed using interior-point solvers. The step for specifying the constraints was the largest difference between the DTA-C method compared to the previous approaches. In the DTA method, the constraints were well-defined by the precise-spike learning problem. Here, the problem constraints were obtained by the target neuron selection process. The fundamental idea of the process is similar to STDP-based unsupervised learning methods, wherein the most active neurons (earliest spike time) receive positive weight adjustments (Kheradpisheh et al. 2018; Vaila, Chiasson and Saxena 2020). Similarly to these existing methods, one of the problem with the DTA-C algorithm is that early-stopping must be applied in order to prevent over-training or over-fitting. It is a non-trivial problem to determine when the optimal point during training is to implement early-stopping, however we have observed that there are certain features of the convolutional weight distribution which may forecast over-training.

Similarly to the DTA and DTA-B methods, the DTA-C algorithm demonstrates good convergence speed when compared to similar methods existing in the literature. While our models do not demonstrate superior learning accuracy, the fast convergence suggests that one of the main applications for the proposed approaches are to pre-train SNNs. In particular, the method may be used with one sample per input category, and the resulting weights are tuned by some other method such as back-propagation. Doing so can speed up the overall time required to converge to a solution, which has benefits for the overall computational complexity. This highlights an additional benefit of using the DTA-C algorithm in a layer-wise training approach, which is flexibility. In particular, the DTA-C algorithm can be used for training the convolutional layer, and then the output layer can be trained using any other supervised learning method. Given that the algorithm fully trains a convolutional layer in under 10 seconds on any of the dataset tested here, the method for training the output layer can be swapped with some other approach which demonstrates better accuracy or robustness to noise, while decreasing the required learning complexity.

One problem with our experimental designs in this chapter, which is related to

the discussion above, is that it is difficult to decouple the learning performances of the DTA-B and DTA-C algorithm. That is, in order to improve the approach it is unclear which algorithm requires adjustments. One way to decouple the learning performances of these algorithms is to perform two additional experiments. The first experiment switches the DTA-B method for some other learning algorithm, for example the SLAYER method Shrestha and Orchard (2018), in order to determine whether the visual features extracted by the DTA-C algorithm still works well with other supervised learning algorithms. If better results are achieved, then the DTA-B algorithm is insufficient. The second experiment switches the DTA-C method for another unsupervised feature extraction algorithm, for example Vaila, Chiasson and Saxena (2020), in order to empirically compare the features learned by DTA-C with other approaches. If better results are achieved, then the learned features are insufficient.

There are two main problems to the DTA-C algorithm, in its current state. The first problem is that there are insufficient evidence to support the capabilities of the algorithm to train deeper networks. This greatly limits the application of the algorithm to more complex learning problems. The second problem is that this method is only capable of training multi-layer *convolutional* neural networks. With regards to the overall goal of applying the DTA approach to multi-layer architectures, it would still be ideal to apply the method to fully-connected deep networks, which are fundamentally problem-agnostic. While applications of convolutional neural networks do exist outside of computer vision, such networks are particularly well-poised to exploit *spatial* structure in the data, which may not always exist for all types of problems.

Another problem of the DTA-C algorithm is that it is not biologically plausible, in that a global signal is required in order to apply the target neuron selection process. As a consequence, this means that different neurons in the same convolutional layer must not only know which times the other neurons spiked, but also have access to the history of their membrane potentials. As such, in this learning scheme the network performs not only spatially global credit assignment, but also temporally global credit assignment (with knowledge of past and future states). Both of these global learning mechanisms are currently thought of as not having enough evidence in biological systems (Taherkhani et al. 2020). In addition, this

means that our approach cannot perform learning in an online and continuous manner as with STDP-based approaches. As such, the proposed method cannot be performed in an on-chip manner for neuromorphic platforms, which is a significant weakness for the computational and energy efficiency of the method, regardless of the improvements observed in terms of convergence speed.

Chapter 6

Discussion

6.1 Thesis Summary

In this thesis we have introduced three novel training algorithms to solve a variety of supervised and unsupervised SNN learning problems with temporal coding. The computation of each algorithm can be broken down into two sequential steps: In the first step, the learning problem is mapped into a Constraint Satisfaction Problem. This formulation involves defining two sets of linear constraints on the membrane potential of a neuron, which specify when the neuron should or should not generate an output spike. In the second learning step, a weight update is computed by solving the system of constraints using industry-standard linear optimisation solvers. Here, we chose an expression for the weight update equation which is similar to that of supervised learning methods derived from the classical Widrow-Hoff learning rule (Widrow and Lehr 1990; Ponulak and Kasiński 2010; Yu et al. 2013; Gardner and Grüning 2016), and the optimisation variables take the form of real-valued scalars which modulate the magnitude of each weight adjustment term. In theory, this constraint programming approach allows adjustment of the membrane potentials at spike time to specific values such that the neuron generate the desired spikes, in one update step. However, due to the continuous nature of temporal integration in the neuron model, the algorithms may require a number of update steps in order to converge.

The DTA algorithm is proposed here to solve precisely-timed spike mapping

problems, wherein the neuron must be trained to produce the correct spike sequence given some input. This is a fundamental learning task for spiking models, and state-of-the-art methods such as Shrestha and Orchard (2018) still utilises this problem for demonstration purposes. In Chapter 3, we go one step further and provide measurements of the maximal memory capacity of the DTA algorithm, following the metrics proposed in Memmesheimer et al. (2014). The capabilities for learning algorithms to achieve this theoretical estimate has been rarely demonstrated in the literature, which is partly due to the large computational requirements of simulation. Here, we demonstrate that our algorithm is capable of reaching the established theoretical bound on memory capacity, while requiring under 100 learning epochs to do so.

One of the fundamental difficulties in learning multiple precisely-timed spikes is the problem of learning interference, wherein an adjustment to the timing of one output spike affects the convergence of another spike. Several approaches have been suggested to overcome learning interference, from a general principle regarding the ratio of synaptic adjustments (Xu et al. 2013), to limiting the weight adjustments to one spike at each learning iteration (Memmesheimer et al. 2014). The constrained computation of the DTA algorithm naturally provides a different approach to handling interference, which we illustrate in a two-dimensional problem in Chapter 3.

The DTA-B algorithm is introduced in Chapter 4. Here, the learning problem involves the additional complexity of finding appropriate spike times before the CSP can be formulated. We show that the Spike-Threshold-Surface function originally proposed by Gutig (2016) provides a suitable solution to this problem, even when used outside of the original gradient derivations. Additionally, we make adjustments to the original procedure to compute the STS, in order to prevent failure and better match the computational characteristics of the DTA algorithm. The result is a 37% reduction in runtime when compared to the original procedure, and a much higher reduction in weight computation runtime when compared to the original recursive weight update method in Gutig (2016). The DTA-B algorithm is demonstrated on two classic data classification problems, wherein near state-of-the-art learning performance is observed.

From the results demonstrated by the DTA-B algorithm, it was clear that

while the method is efficient in terms of runtime, its most important limitation is the inability to train multi-layer SNNs. Consequently, the size and complexity of the learning problems with which the method is applicable are also very limited. In order to overcome this difficulty, we introduce the DTA-C algorithm in Chapter 5, in order to train the hidden layers in a convolutional network. This method is inspired by existing unsupervised learning rules with Spike-Timing Dependent Plasticity, for example (Kheradpisheh et al. 2018; Vaila, Chiasson and Saxena 2020). By limiting each convolutional neuron to a single output spike, and exploiting the relatively predictable activation of the convolution operation, the method successfully and automatically extracts salient features in the training images. The multi-layer architecture is successfully applied to several image classification benchmarks, and in particular demonstrates fast convergence speed for feature extraction, when compared to STDP-based approaches. This suggests that the DTA-C method perform well in learning scenarios where speed is important or training data is limited, such as layer-wise pre-training.

It can be quite difficult to categorise the proposed algorithms in the context of the wider literature. While the form of the weight update is similar to that of methods which perform gradient-descent, the approach is distinctly different from the established incremental learning paradigms wherein the synaptic update step is performed with a fixed step size at each iteration. The computation of the weight solver step is much closer to the CONE algorithm (Lee, Kukreja and Thakor 2016), and by extension it is also similar to other batch-computation approaches wherein the solution weights are analytically computed in either one or few update steps (Tapson and van Schaik 2013; Tapson et al. 2013; Boucher-Routhier, Zhang and Thivierge 2021). However, such methods all guarantee that their computation converges within exactly one presentation of the training data, which is a characteristic that our method does not adhere to. Removing the requirement for presenting all the data can be seen as a benefit: for example MNIST problem solved is by the 3-layer CSNN architecture using well under one epoch of training data.

6.2 Future Work

This section outlines some important limitations of the proposed constrained optimisation approach for SNN learning, and accordingly some directions for future research that follow from our results and discussion.

6.2.1 GPU Implementation

An important limitation of our SNN implementation used throughout this thesis is that all simulation and learning procedures are implemented on CPUs. This means that even for networks of a few hundred or thousand neurons in the hidden layer (for example the CSNN models), the ability to parallelise operations are greatly limited. In turn, this makes it difficult to investigate the runtime complexity of our event-based algorithms in comparison to state-of-the-art clock-based learning algorithms such as Shrestha and Orchard (2018). Additionally, this limits our ability to test computationally expensive learning algorithms, such as the Multi-Spike Tempotron (Gutig 2016), for many training iterations.

In general, there are two potential avenues for more efficient implementations of the algorithms and architectures used in this thesis. The first approach is to utilise an event-driven simulator with GPU computing capabilities. However, to the best of our knowledge, to date there are no open-source and easily accessible GPU simulators that fit these requirements (see Naveros et al. (2017) for a review of event-driven techniques). One could instead apply the DTA algorithm to a clock-driven SNN implementation on CPU, however this would be considered a trade-off given that the DTA algorithm has specific in-built mechanisms to handle continuous-time simulations.

If a GPU implementation of the neural network is possible then the second avenue for improving the DTA algorithm becomes available, which is to utilise GPU-based constraint solvers. The computation of a constraint solver is inherently sequential and not easily amenable for parallelisation. While parallel interior-point solvers exist in the literature, for example Gondzio and Sarkissian (2003); Rehfeldt et al. (2022), these software typically achieve parallelisation by exploiting special structure or geometries in the underlying optimisation problem,

which may not necessarily be applicable in our case. Simply choosing a suitable parallel interior-point solution is non-trivial, because solver design is a highly specialised area of research which lies in between mathematics and computer science.

6.2.2 Objective Functions for Constrained Optimisation

One alternative design of our proposed constraint programming approach is to re-introduce an objective function which then turns the CSP into a COP. While the CONE algorithm (Lee, Kukreja and Thakor 2016) was taken as inspiration for the DTA rule, we removed the objective function in favour of a pre-defined weight update equation (Equation 33). Our design has the benefit of allowing the optimisation procedure to compute a weight *update*, instead of a single one-batch solution. In addition, through the choice of the learning kernel function we are able to still influence the resulting properties of the solution weight distribution, thus retaining some of the flexibility of objective optimisation in CONE.

There are a number of limitations with our approach. Firstly, the very large flexibility provided by introducing well-defined objective contribution terms into an objective function is lost in the weight update equation. Fundamentally, many objective terms can be derived from well-understood criteria such as L_1 or L_2 norm minimisation, and easily introduced into the CONE algorithm with minimal changes. In the DTA algorithm, the ability to produce noise-robust solutions must be introduced through very careful design of the learning kernel function, which may also involve expensive parameter optimisations. This links into the second limitation, which is the fact that the learning kernels adjusted to provide a certain computational characteristic may also compromise the ability of the DTA algorithm to reliably converge or find feasible solutions (Section 3.6). In the CONE method, the nature of constrained optimisation (Tanneau, Anjos and Lodi 2021) means that guaranteeing a feasible solution and optimisation of the objective are decoupled. This means that the solver always will first find a feasible solution. In the DTA method, these two computational requirements are coupled, and thus the learning kernel function represents a single point of failure.

One possible research direction may be to re-introduce a simple objective function into the DTA learning. This is also partly the reason why we used interior-point solvers in our simulations, which are normally used for solving COPs. Initial investigations were conducted to test both L_1 or L_2 norm minimisation objectives, however the additional computations introduced by the weight update equation means that evaluation of weight-based objective terms become very computationally expensive. This is because the solution weights in the DTA algorithm are not expressed as a simple vector, but instead obtained by matrix multiplication and addition operations. To be viable, objective terms should optimise the values of the optimisation variables themselves.

6.2.3 Multi-Layer Fully-Connected SNNs

The fundamental limitation of our approach remains that it is unable to train multi-layer fully-connected networks. With the DTA-C algorithm we are able to address the training of Convolutional SNNs in signal processing tasks, however convolutional networks are highly specialised to problems with spatially local features within the training data. This means that the architecture is not problem-agnostic, and the applicability of our learning approach is still limited in data classification or regression tasks. In such scenarios, it would be better to utilise multi-layer networks with full connectivity.

The underlying reason why the DTA and DTA-B algorithms are limited to training single-layer SNNs is the requirement that both the input and target output of each individual neurons are known at training time. This is fundamentally problematic, because in a multi-layer network the target output of a hidden layer (and correspondingly the input to the next layer) are not specified by the learning problem. In its current form, the DTA and DTA-B algorithms have no mechanisms with which it is possible to automatically learn these hidden spike times. One potential solution is to set the hidden spike times as optimisation variables, and simultaneously solve for both spike times and weights. However, in this approach the CSP becomes both non-linear and non-convex, which is significantly more difficult to optimise. Initial investigations were performed to analyse the viability of such an algorithm, however even with a few hidden neurons the problem

reported infeasibility. Another problem that arises is the number of optimisation variables becomes unknown, unless each neuron in the hidden layers are allowed a limited number of activations.

6.3 Publications

- Chu, D. and Nguyen, H. L. (2021) 'Constraints on Hebbian and STDP learned weights of a spiking neuron', Neural Networks. Elsevier, pp. 192-200. doi: 10.1016/j.neunet.2020.12.012.
- Nguyen, H. L. and Chu, D. (2022). Incremental neural synthesis for spiking neural networks. In *IEEE Symposium Series on Computational Intelligence*

Bibliography

(1978). Chapter 3 review of linear programming. In J. L. Cohon, ed., *Multiobjective Programming and Planning*, *Mathematics in Science and Engineering*, vol. 140, Elsevier, pp. 27–67.

Adrian, E. D. (1926). The impulses produced by sensory nerve endings. *The Journal of Physiology*, 61(1), pp. 49–72, `https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1926.sp002273`.

Alyamkin, S. et al. (2018). 2018 low-power image recognition challenge. *arXiv preprint arXiv:181001732*.

Alzubaidi, L. et al. (2021). Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8(1), pp. 1–74.

Anjos, M. F. and Burer, S. (2008). On handling free variables in interior-point methods for conic linear optimization. *SIAM Journal on Optimization*, 18(4), pp. 1310–1325.

Auge, D., Hille, J., Mueller, E. and Knoll, A. (2021). A survey of encoding techniques for signal processing in spiking neural networks. *Neural Processing Letters*, 53(6), pp. 4693–4710.

Barwad, A., Dey, P. and Susheilia, S. (2012). Artificial neural network in diagnosis of metastatic carcinoma in effusion cytology. *Cytometry Part B: Clinical Cytometry*, 82(2), pp. 107–111.

Bengio, Y. et al. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1), pp. 1–127.

Bengio, Y., Lee, D.-H., Bornschein, J., Mesnard, T. and Lin, Z. (2015). Towards biologically plausible deep learning. *arXiv preprint arXiv:150204156*.

Bi, G.-Q. and Wang, H.-X. (2002). Temporal asymmetry in spike timing-dependent synaptic plasticity. *Physiology & behavior*, 77(4-5), pp. 551–555.

Bishop, C. M. et al. (1995). *Neural networks for pattern recognition*. Oxford university press.

Bliss, T. V. and Lømo, T. (1973). Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *The Journal of physiology*, 232(2), pp. 331–356.

Bohte, S. M., Kok, J. N. and La Poutre, H. (2002). Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1-4), pp. 17–37.

Bohte, S. M., Kok, J. N. and La Poutré, J. A. (2000). Spikeprop: backpropagation for networks of spiking neurons. In *ESANN*, vol. 48, Bruges, pp. 419–424.

Borst, A. and Theunissen, F. E. (1999). Information theory and neural coding. *Nature neuroscience*, 2(11), pp. 947–957.

Boucher-Routhier, M., Zhang, B. L. F. and Thivierge, J.-P. (2021). Extreme neural machines. *Neural Networks*, 144, pp. 639–647.

Brette, R. et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience*, 23(3), pp. 349–398.

Burkitt, A. N. (2006). A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biological cybernetics*, 95(1), pp. 1–19.

Catalogna, M. et al. (2012). Artificial neural networks based controller for glucose monitoring during clamp test.

Chichilnisky, E. (2001). A simple white noise analysis of neuronal light responses. *Network: computation in neural systems*, 12(2), p. 199.

Chou, C.-N., Chung, K.-M. and Lu, C.-J. (2018). On the algorithmic power of spiking neural networks. *arXiv preprint arXiv:180310375*.

Christensen, D. V. et al. (2022). 2022 roadmap on neuromorphic computing and engineering. *Neuromorphic Computing and Engineering*, 2(2), p. 022501.

Citri, A. and Malenka, R. C. (2008). Synaptic plasticity: multiple forms, functions, and mechanisms. *Neuropsychopharmacology*, 33(1), pp. 18–41.

Cohen, G., Afshar, S., Tapson, J. and Van Schaik, A. (2017). Emnist: Extending mnist to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*, IEEE, pp. 2921–2926.

Cohen, G. K. et al. (2016). Skimming digits: neuromorphic classification of spike-encoded images. *Frontiers in neuroscience*, 10, p. 184.

Cybenko, G. (1992). Approximation by superpositions of a sigmoidal function. *Math Control Signals Syst*, 5(4), p. 455.

Da Silva, I. N., Spatti, D. H., Flauzino, R. A., Liboni, L. H. B. and dos Reis Alves, S. F. (2017). Artificial neural networks. *Cham: Springer International Publishing*, 39.

Dan, Y. and Poo, M.-m. (2004). Spike timing-dependent plasticity of neural circuits. *Neuron*, 44(1), pp. 23–30.

Davies, M. et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1), pp. 82–99.

Dayan, P. and Abbott, L. F. (2005). *Theoretical neuroscience: computational and mathematical modeling of neural systems*. MIT press.

de Polavieja, G. G., Harsch, A., Kleppe, I., Robinson, H. P. and Juusola, M. (2005). Stimulus history reliably shapes action potential waveforms of cortical neurons. *Journal of Neuroscience*, 25(23), pp. 5657–5665.

Debanne, D., Bialowas, A. and Rama, S. (2013). What are the mechanisms for analogue and digital signalling in the brain? *Nature Reviews Neuroscience*, 14(1), pp. 63–69.

Debanne, D. and Russier, M. (2017). How do electrical synapses regulate their strength? *The Journal of physiology*, 595(13), p. 4121.

DeBole, M. V. et al. (2019). Truenorth: Accelerating from zero to 64 million neurons in 10 years. *Computer*, 52(5), pp. 20–29.

Domínguez, A. (2015). A history of the convolution operation [retrospectroscope]. *IEEE pulse*, 6(1), pp. 38–49.

Dua, D. and Graff, C. (2017). UCI machine learning repository.

Eldan, R. and Shamir, O. (2016). The power of depth for feedforward neural networks. In *Conference on learning theory*, PMLR, pp. 907–940.

Eliasmith, C. and Anderson, C. H. (2003). *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT press.

Elveren, E. and Yumuşak, N. (2011). Tuberculosis disease diagnosis using artificial neural network trained with genetic algorithm. *Journal of medical systems*, 35(3), pp. 329–332.

Fang, W. et al. (2021). Incorporating learnable membrane time constant to enhance learning of spiking neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 2661–2671.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2), pp. 179–188.

Florian, R. V. (2012). The chronotron: A neuron that learns to fire temporally precise spike patterns. *PloS one*, 7(8), p. e40233.

French, A. and Holden, A. (1971). Alias-free sampling of neuronal spike trains. *Kybernetik*, 8(5), pp. 165–171.

Galluppi, F. and Furber, S. (2011). Representing and decoding rank order codes using polychronization in a network of spiking neurons. In *The 2011 International Joint Conference on Neural Networks*, IEEE, pp. 943–950.

Gardner, B. and Grüning, A. (2016). Supervised learning in spiking neural networks for precise temporal encoding. *PloS one*, 11(8), p. e0161335.

Gardner, B. and Grüning, A. (2021). Supervised learning with first-to-spike decoding in multilayer spiking neural networks. *Frontiers in computational neuroscience*, 15, p. 25.

Gerstner, W. and Kistler, W. M. (2002). *Spiking Neuron Models Single Neurons, Populations, Plasticity*. Cambridge University Press.

Gerstner, W., Kistler, W. M., Naud, R. and Paninski, L. (2014). *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press.

Ghojogh, B. and Ghodsi, A. (2020). Attention mechanism, transformers, bert, and gpt: Tutorial and survey.

Gollisch, T. and Meister, M. (2008). Rapid neural coding in the retina with relative spike latencies. *science*, 319(5866), pp. 1108–1111.

Gondzio, J. and Sarkissian, R. (2003). Parallel interior-point solver for structured linear programs. *Mathematical Programming*, 96(3), pp. 561–584.

Grüning, A. and Bohte, S. M. (2014). Spiking neural networks: Principles and challenges. In *ESANN*, Bruges.

Guo, W., Fouda, M. E., Eltawil, A. M. and Salama, K. N. (2021). Neural coding in spiking neural networks: A comparative study for robust neuromorphic systems. *Frontiers in Neuroscience*, 15, p. 638474.

Gutig, R. (2016). Spiking neurons can discover predictive features by aggregate-label learning. *Science*, 351, pp. aab4113–aab4113.

Gütig, R. and Sompolinsky, H. (2006). The tempotron: a neuron that learns spike timing–based decisions. *Nature Neuroscience*, 9, pp. 420–428.

Hebb, D. O. (1949). *The organization of behavior: a neuropsychological theory*. Science editions.

Henry, G. H., Dreher, B. and Bishop, P. (1974). Orientation specificity of cells in cat striate cortex. *Journal of neurophysiology*, 37(6), pp. 1394–1409.

Ho, V. M., Lee, J.-A. and Martin, K. C. (2011). The cell biology of synaptic plasticity. *Science*, 334(6056), pp. 623–628.

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4), p. 500.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2), pp. 251–257.

Huang, G.-B., Zhu, Q.-Y. and Siew, C.-K. (2006). Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3), pp. 489–501.

Hunsberger, E. (2018). Spiking deep neural networks: Engineered and biological approaches to object recognition.

Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5), pp. 1063–1070.

Izhikevich, E. M. (2006). Polychronization: computation with spikes. *Neural computation*, 18(2), pp. 245–282.

Jin, Y., Zhang, W. and Li, P. (2018). Hybrid macro/micro level backpropagation for training deep spiking neural networks. *Advances in neural information processing systems*, 31.

Johansson, R. S. and Birznieks, I. (2004). First spikes in ensembles of human tactile afferents code complex spatial fingertip events. *Nature neuroscience*, 7(2), pp. 170–177.

Kandel, E. R. et al. (2000). *Principles of neural science*, vol. 4. McGraw-hill New York.

Katz, B. and Miledi, R. (1968). The role of calcium in neuromuscular facilitation. *The Journal of physiology*, 195(2), pp. 481–492.

Kheradpisheh, S. R., Ganjtabesh, M. and Masquelier, T. (2016). Bio-inspired unsupervised learning of visual features leads to robust invariant object recognition. *Neurocomputing*, 205, pp. 382–392.

Kheradpisheh, S. R., Ganjtabesh, M., Thorpe, S. J. and Masquelier, T. (2018). Stdp-based spiking deep convolutional neural networks for object recognition. *Neural Networks*, 99, pp. 56–67.

Kim, Y. and Panda, P. (2021). Optimizing deeper spiking neural networks for dynamic vision sensing. *Neural Networks*, 144, pp. 686–698.

Koch, C. and Segev, I. (1989). Methods in neural modeling. *MIT Press, Cambridge, MA*, 8, pp. 1659–1671.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Kulkarni, S. R. and Rajendran, B. (2018). Spiking neural networks for handwritten digit recognition—supervised learning and network optimization. *Neural Networks*, 103, pp. 118–127.

Kwon, D. et al. (2020). On-chip training spiking neural networks using approximated backpropagation with analog synaptic devices. *Frontiers in neuroscience*, p. 423.

Lapicque, L. É. (1907). Louis lapicque. *J physiol*, 9, pp. 620–635.

LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database. `http://yann.lecun.com/exdb/mnist/`.

Lee, C., Sarwar, S. S., Panda, P., Srinivasan, G. and Roy, K. (2020). Enabling spike-based backpropagation for training deep neural network architectures. *Frontiers in neuroscience*, p. 119.

Lee, W. W., Kukreja, S. L. and Thakor, N. V. (2016). Cone: Convex-optimized-synaptic efficacies for temporally precise spike mapping. *IEEE transactions on neural networks and learning systems*, 28(4), pp. 849–861.

Leibe, B. and Schiele, B. (2003). Analyzing appearance and contour based methods for object categorization. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, vol. 2, IEEE, pp. II–409.

Li, D., Chen, X., Becchi, M. and Zong, Z. (2016). Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus. In *2016 IEEE international conferences on big data and cloud computing (BDCloud), social computing and networking (SocialCom), sustainable computing and communications (SustainCom)(BDCloud-SocialCom-SustainCom)*, IEEE, pp. 477–484.

Li, S. and Yu, Q. (2020). New efficient multi-spike learning for fast processing and robust learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34, pp. 4650–4657.

Li, Y. et al. (2021). Differentiable spike: Rethinking gradient-descent for training spiking neural networks. *Advances in Neural Information Processing Systems*, 34, pp. 23426–23439.

Liang, N.-Y., Huang, G.-B., Saratchandran, P. and Sundararajan, N. (2006). A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE Transactions on neural networks*, 17(6), pp. 1411–1423.

Lillicrap, T. P. and Santoro, A. (2019). Backpropagation through time and the brain. *Current opinion in neurobiology*, 55, pp. 82–89.

Linnainmaa, S. (1976). Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2), pp. 146–160.

Littmann, E., Meyering, A. and Ritter, H. (1992). Cascaded and parallel neural network architectures for machine vision—a case study. In *Mustererkennung 1992*, Springer, pp. 81–87.

Llanas, B., Lantarón, S. and Sáinz, F. J. (2008). Constructive approximation of discontinuous functions by neural networks. *Neural Processing Letters*, 27(3), pp. 209–226.

London, M., Roth, A., Beeren, L., Häusser, M. and Latham, P. E. (2010). Sensitivity to perturbations in vivo implies high noise and suggests rate coding in cortex. *Nature*, 466(7302), pp. 123–127.

Luo, X., Qu, H., Zhang, Y. and Chen, Y. (2019). First error-based supervised learning algorithm for spiking neural networks. *Frontiers in neuroscience*, 13, p. 559.

Luo, Y. et al. (2021). Siamsnn: siamese spiking neural networks for energy-efficient object tracking. In *International Conference on Artificial Neural Networks*, Springer, pp. 182–194.

Ma, C., Wojtowytsch, S., Wu, L. et al. (2020). Towards a mathematical understanding of neural network-based machine learning: what we know and what we don't. *arXiv preprint arXiv:200910713*.

Maass, W. (1997). Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9), pp. 1659–1671.

MacKay, D. M. and McCulloch, W. S. (1952). The limiting information capacity of a neuronal link. *The bulletin of mathematical biophysics*, 14(2), pp. 127–135.

Mainen, Z. F. and Sejnowski, T. J. (1995). Reliability of spike timing in neocortical neurons. *Science*, 268(5216), pp. 1503–1506.

Maley, C. J. (2018). Toward analog neural computation. *Minds and Machines*, 28(1), pp. 77–91.

Mancoo, A., Keemink, S. and Machens, C. K. (2020). Understanding spiking networks through convex optimization. *Advances in Neural Information Processing Systems*, 33, pp. 8824–8835.

Martin, S., Grimwood, P. D., Morris, R. G. et al. (2000). Synaptic plasticity and memory: an evaluation of the hypothesis. *Annual review of neuroscience*, 23(1), pp. 649–711.

Masquelier, T. and Thorpe, S. J. (2007). Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS computational biology*, 3(2), p. e31.

Matoušek, J. and Gärtner, B. (2007). *Understanding and using linear programming*, vol. 33. Springer.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), pp. 115–133.

McKennoch, S., Liu, D. and Bushnell, L. G. (2006). Fast modifications of the spikeprop algorithm. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, IEEE, pp. 3970–3977.

Memmesheimer, R.-M., Rubin, R., Ölveczky, B. P. and Sompolinsky, H. (2014). Learning precisely timed spikes. *Neuron*, 82, pp. 925–938.

Mikolov, T., Chen, K., Corrado, G. and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:13013781*.

Mohammed, S. A., Darrab, S., Noaman, S. A. and Saake, G. (2020). Analysis of breast cancer detection using different machine learning techniques. In *International Conference on Data Mining and Big Data*, Springer, pp. 108–117.

Mohemmed, A., Schliebs, S., Matsuda, S. and Kasabov, N. (2012). Span: Spike pattern association neuron for learning spatio-temporal spike patterns. *International journal of neural systems*, 22(04), p. 1250012.

Morrison, A., Diesmann, M. and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biological cybernetics*, 98(6), pp. 459–478.

Mozafari, M., Ganjtabesh, M., Nowzari-Dalini, A., Thorpe, S. J. and Masquelier, T. (2019). Bio-inspired digit recognition using reward-modulated spike-timing-dependent plasticity in deep convolutional networks. *Pattern recognition*, 94, pp. 87–95.

Nagumo, J., Arimoto, S. and Yoshizawa, S. (1962). An active pulse transmission line simulating nerve axon. *Proceedings of the IRE*, 50(10), pp. 2061–2070.

Naveros, F. et al. (2014). A spiking neural simulator integrating event-driven and time-driven computation schemes using parallel cpu-gpu co-processing: a case study. *IEEE transactions on neural networks and learning systems*, 26(7), pp. 1567–1574.

Naveros, F., Garrido, J. A., Carrillo, R. R., Ros, E. and Luque, N. R. (2017). Event-and time-driven techniques using parallel cpu-gpu co-processing for spiking neural networks. *Frontiers in neuroinformatics*, 11, p. 7.

Nguyen, H. L. and Chu, D. (2022). Incremental neural synthesis for spiking neural networks. In *IEEE Symposium Series on Computational Intelligence*.

Nielsen, M. A. (2015). *Neural networks and deep learning*, vol. 25. Determination press San Francisco, CA, USA.

Niu, S., Liu, Y., Wang, J. and Song, H. (2020). A decade survey of transfer learning (2010–2020). *IEEE Transactions on Artificial Intelligence*, 1(2), pp. 151–166.

Olshausen, B. A. and Field, D. J. (2004). Sparse coding of sensory inputs. *Current opinion in neurobiology*, 14(4), pp. 481–487.

Orchard, G. et al. (2021). Efficient neuromorphic signal processing with loihi 2. In *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, IEEE, pp. 254–259.

Paiva, A. R., Park, I. and Principe, J. C. (2009). A reproducing kernel hilbert space framework for spike train signal processing. *Neural computation*, 21(2), pp. 424–449.

Paninski, L. (2004). Maximum likelihood estimation of cascade point-process neural encoding models. *Network: Computation in Neural Systems*, 15(4), p. 243.

Panzeri, S., Schultz, S. R., Treves, A. and Rolls, E. T. (1999). Correlations and the encoding of information in the nervous system. *Proceedings of the Royal Society of London Series B: Biological Sciences*, 266(1423), pp. 1001–1012.

Pauluis, Q. and Baker, S. N. (2000). An accurate measure of the instantaneous discharge probability, with application to unitary joint-event analysis. *Neural computation*, 12(3), pp. 647–669.

Ponulak, F. (2006). Resume-proof of convergence.

Ponulak, F. and Kasiński, A. (2010). Supervised learning in spiking neural networks with resume: Sequence learning, classification, and spike shifting. *Neural Computation*, 22, pp. 467–510.

Qu, P., Zhang, Y., Fei, X. and Zheng, W. (2020). High performance simulation of spiking neural network on gpgpus. *IEEE Transactions on Parallel and Distributed Systems*, 31(11), pp. 2510–2523.

Ramsundar, B. and Zadeh, R. B. (2018). *TensorFlow for deep learning: from linear regression to reinforcement learning.* " O'Reilly Media, Inc.".

Rehfeldt, D. et al. (2022). A massively parallel interior-point solver for lps with generalized arrowhead structure, and applications to energy system models. *European Journal of Operational Research*, 296(1), pp. 60–71.

Renner, A., Sheldon, F., Zlotnik, A., Tao, L. and Sornborger, A. (2021). The backpropagation algorithm implemented on spiking neuromorphic hardware. *arXiv preprint arXiv:210607030*.

Ros, E., Carrillo, R., Ortigosa, E. M., Barbour, B. and Agís, R. (2006). Event-driven simulation scheme for spiking neural networks using lookup tables to characterize neuronal dynamics. *Neural computation*, 18(12), pp. 2959–2993.

Rosahl, T. W. et al. (1993). Short-term synaptic plasticity is altered in mice lacking synapsin i. *Cell*, 75(4), pp. 661–670.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6), p. 386.

Rossum, M. v. (2001). A novel spike distance. *Neural computation*, 13(4), pp. 751–763.

Roy, K., Jaiswal, A. and Panda, P. (2019). Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575(7784), pp. 607–617.

Rubin, R., Monasson, R. and Sompolinsky, H. (2010). Theory of spike timing-based neural classifiers. *Physical Review Letters*, 105.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088), pp. 533–536.

Satuvuori, E. and Kreuz, T. (2018). Which spike train distance is most suitable for distinguishing rate and temporal coding? *Journal of neuroscience methods*, 299, pp. 22–33.

Sboev, A., Vlasov, D., Rybka, R. and Serenko, A. (2018). Spiking neural network reinforcement learning method based on temporal coding and stdp. *Procedia computer science*, 145, pp. 458–463.

Schrauwen, B. and Van Campenhout, J. (2004). Extending spikeprop. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*, vol. 1, IEEE, pp. 471–475.

Schrauwen, B. and Van Campenhout, J. (2007). Linking non-binned spike train kernels to several existing spike train metrics. *Neurocomputing*, 70(7-9), pp. 1247–1253.

Schwartz, O., Pillow, J. W., Rust, N. C. and Simoncelli, E. P. (2006). Spike-triggered neural characterization. *Journal of vision*, 6(4), pp. 13–13.

Sengupta, A., Ye, Y., Wang, R., Liu, C. and Roy, K. (2019). Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in neuroscience*, 13, p. 95.

Shawon, A., Rahman, M. J.-U., Mahmud, F. and Zaman, M. A. (2018). Bangla handwritten digit recognition using deep cnn for large and unbiased dataset. In *2018 International Conference on Bangla Speech and Language Processing (ICBSLP)*, IEEE, pp. 1–6.

Sheldon, M. R., Fillyaw, M. J. and Thompson, W. D. (1996). The use and interpretation of the friedman test in the analysis of ordinal-scale data in repeated measures designs. *Physiotherapy Research International*, 1(4), pp. 221–228.

Shi, C. et al. (2021). Deeptempo: a hardware-friendly direct feedback alignment multi-layer tempotron learning rule for deep spiking neural networks. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 68(5), pp. 1581–1585.

Shrestha, S. B. and Orchard, G. (2018). Slayer: Spike layer error reassignment in time. *Advances in neural information processing systems*, 31.

Shrestha, S. B. and Song, Q. (2017). Robustness to training disturbances in spikeprop learning. *IEEE transactions on neural networks and learning systems*, 29(7), pp. 3126–3139.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:14091556*.

Sontag, E. D. (1991). Feedback stabilization using two-hidden-layer nets. In *1991 American control conference*, IEEE, pp. 815–820.

Sporea, I. and Grüning, A. (2013). Supervised learning in multilayer spiking neural networks. *Neural computation*, 25(2), pp. 473–509.

Sterratt, D., Graham, B., Gillies, A. and Willshaw, D. (2011). *Principles of computational modelling in neuroscience*. Cambridge University Press.

Taherkhani, A., Belatreche, A., Li, Y. and Maguire, L. P. (2018). A supervised learning algorithm for learning precise timing of multiple spikes in multilayer spiking neural networks. *IEEE transactions on neural networks and learning systems*, 29(11), pp. 5394–5407.

Taherkhani, A. et al. (2020). A review of learning in biologically plausible spiking neural networks. *Neural Networks*, 122, pp. 253–272.

Tanneau, M., Anjos, M. F. and Lodi, A. (2021). Design and implementation of a modular interior-point solver for linear optimization. *Mathematical Programming Computation*.

Tapson, J. and van Schaik, A. (2013). Learning the pseudoinverse solution to network weights. *Neural Networks*, 45, pp. 94–100.

Tapson, J. C. et al. (2013). Synthesis of neural networks for spatio-temporal spike pattern recognition and processing. *Frontiers in neuroscience*, 7, p. 153.

Tavanaei, A., Kirby, Z. and Maida, A. S. (2018). Training spiking convnets by stdp and gradient descent. In *2018 International Joint Conference on Neural Networks (IJCNN)*, IEEE, pp. 1–8.

Tavanaei, A. and Maida, A. (2019). Bp-stdp: Approximating backpropagation using spike timing dependent plasticity. *Neurocomputing*, 330, pp. 39–47.

Thorpe, S., Delorme, A. and Van Rullen, R. (2001). Spike-based strategies for rapid processing. *Neural networks*, 14(6-7), pp. 715–725.

Thorpe, S., Fize, D. and Marlot, C. (1996). Speed of processing in the human visual system. *nature*, 381(6582), pp. 520–522.

Thorpe, S. J. (1990). Spike arrival times: A highly efficient coding scheme for neural networks. *Parallel processing in neural systems*, pp. 91–94.

Tissera, M. D. and McDonnell, M. D. (2016). Deep extreme learning machines: supervised autoencoding architecture for classification. *Neurocomputing*, 174, pp. 42–49.

Trappenberg, T. (2009). *Fundamentals of computational neuroscience*. OUP Oxford.

Vaila, R., Chiasson, J. and Saxena, V. (2020). A deep unsupervised feature learning spiking neural network with binarized classification layers for the emnist classification. *IEEE transactions on emerging topics in computational intelligence*.

Valadez-Godínez, S., Sossa, H. and Santiago-Montero, R. (2020). On the accuracy and computational cost of spiking neuron implementation. *Neural Networks*, 122, pp. 196–217.

Vaswani, A. et al. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Velinov, A. and Gicev, V. (2018). Practical application of simplex method for solving linear programming problems. *Balkan Journal of Applied Mathematics and Informatics*, 1(1), pp. 07–15.

Victor, J. D. (2005). Spike train metrics. *Current opinion in neurobiology*, 15(5), pp. 585–592.

Vigneron, A. and Martinet, J. (2020). A critical survey of stdp in spiking neural networks for pattern recognition. In *2020 International Joint Conference on Neural Networks (IJCNN)*, IEEE, pp. 1–9.

Wang, R. et al. (2017). Neuromorphic hardware architecture using the neural engineering framework for pattern recognition. *IEEE transactions on biomedical circuits and systems*, 11(3), pp. 574–584.

Weiss, K., Khoshgoftaar, T. M. and Wang, D. (2016). A survey of transfer learning. *Journal of Big data*, 3(1), pp. 1–40.

Welzel, G. and Schuster, S. (2019). A direct comparison of different measures for the strength of electrical synapses. *Frontiers in cellular neuroscience*, 13, p. 43.

Whitlock, J. R., Heynen, A. J., Shuler, M. G. and Bear, M. F. (2006). Learning induces long-term potentiation in the hippocampus. *science*, 313(5790), pp. 1093–1097.

Widrow, B. and Lehr, M. A. (1990). 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9), pp. 1415–1442.

Widrow, B., Greenblatt, A., Kim, Y. and Park, D. (2013). The no-prop algorithm: A new learning algorithm for multilayer neural networks. *Neural Networks*, 37, pp. 182–188.

Wolberg, W. (1992). Uci machine learning repository.

Wright, S. J. (1997). *Primal-dual interior-point methods*. SIAM.

Wu, X., Wang, Y., Tang, H. and Yan, R. (2019). A structure–time parallel implementation of spike-based deep learning. *Neural Networks*, 113, pp. 72–78.

Wunderlich, T. et al. (2019). Demonstrating advantages of neuromorphic computation: a pilot study. *Frontiers in neuroscience*, 13, p. 260.

Xiao, R., Yu, Q., Yan, R. and Tang, H. (2019). Fast and accurate classification with a multi-spike learning algorithm for spiking neurons. In *IJCAI*, pp. 1445–1451.

Xie, X., Qu, H., Liu, G., Zhang, M. and Kurths, J. (2016). An efficient supervised training algorithm for multilayer spiking neural networks. *PloS one*, 11(4), p. e0150329.

Xu, Q. et al. (2018a). Csnn: An augmented spiking based framework with perceptron-inception. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*.

Xu, Q. et al. (2018b). Csnn: an augmented spiking based framework with perceptron-inception. In *IJCAI*, pp. 1646–1652.

Xu, Y., Zeng, X., Han, L. and Yang, J. (2013). A supervised multi-spike learning algorithm based on gradient descent for spiking neural networks. *Neural Networks*, 43, pp. 99–113.

Yang, S. and Peng, G. (2017). D-pcn: Parallel convolutional neural networks for image recognition in reverse adversarial style. *CoRR*.

Young, A. R., Dean, M. E., Plank, J. S. and Rose, G. S. (2019). A review of spiking neuromorphic hardware communication systems. *IEEE Access*, 7, pp. 135606–135620.

Yu, Q., Tang, H., Tan, K. C. and Li, H. (2013). Precise-spike-driven synaptic plasticity: Learning hetero-association of spatiotemporal spike patterns. *Plos one*, 8(11), p. e78318.

Zbili, M. and Debanne, D. (2019). Past and future of analog-digital modulation of synaptic transmission. *Frontiers in cellular neuroscience*, 13, p. 160.

Zeldenrust, F., de Knecht, S., Wadman, W. J., Denève, S. and Gutkin, B. (2017). Estimating the information extracted by a single spiking neuron from a continuous input time series. *Frontiers in computational neuroscience*, 11, p. 49.

Zenke, F. and Ganguli, S. (2018). Superspike: Supervised learning in multilayer spiking neural networks. *Neural computation*, 30(6), pp. 1514–1541.

Zhang, W. and Li, P. (2020). Temporal spike sequence learning via backpropagation for deep spiking neural networks. *Advances in Neural Information Processing Systems*, 33, pp. 12022–12033.

Zhou, X., Song, Z., Wu, X. and Yan, R. (2020). A spiking deep convolutional neural network based on efficient spike timing dependent plasticity. In *2020 3rd International Conference on Artificial Intelligence and Big Data (ICAIBD)*, IEEE, pp. 39–45.

Zucker, R. S., Regehr, W. G. et al. (2002). Short-term synaptic plasticity. *Annual review of physiology*, 64(1), pp. 355–405.