Residual Connections in Spiking Neural Networks

Skipping deeper:
Unveiling the Power of Residual Connections in Multi-Spiking Neural Networks

Àlex De Los Santos Subirats

Delft University of Technology

Residual Connections in Spiking Neural Networks

Skipping deeper:
Unveiling the Power of Residual Connections in Multi-Spiking Neural Networks

by

Àlex De Los Santos Subirats

Student Name	Student Number
De Los Santos Subirats	5090520

Faculty: Electrical Engineering, Mathematics and Computer Science, Delft

Cover: Cover art generated by Dream AI
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

The current report of *Residual Connections in Spiking Neural Networks - Skipping deeper: Unveiling the Power of Residual Connections in Multi-Spiking Neural Networks* presents the work done for my master's graduation project. Research was conducted within the Computer Vision Lab of TU Delft, under the supervision of Dr. J.C. van Gemert, Dr. N. Tömen, and A. Micheli, the later of the two having been my daily co-supervisors.

I selected this topic because I was interested in a challenge and working on implementing something new, and after A. Micheli's lecture on Spiking Neural Networks as part of the "Computer Vision by Deep Learning Seminar" I was fascinated by this novel concept of spiking neural networks. For that, I would like to thank her for that lecture and Dr. J.C. van Gemert for teaching that seminar.

During these past few months, I have learned and struggled to implement these networks at a level of detail I have never experienced before. Gaining a deeper understanding of the calculations behind SNNs and traditional ANNs. For their help and guidance during these times I would like to thank Dr. N. Tömen and A. Micheli, it has been a pleasure to work with you and I hope our paths cross again in the future. Furthermore, I would like to thank Dr. M. Weinmann for joining the thesis committee as an external member.

Finally and on a more personal note I would like to thank my family and friends for their support during this time. Thank you to my parents for instilling in me a love for science and computers, and a work ethic that has been of great help during my academic career until now. Thank you to my little brother who I am incredibly proud of. I would like to thank my wonderful girlfriend whom I love very much for all her daily love and support, thank you for being part of my life. Finally, I would like to acknowledge my friends and old roommates of the great Palazzo house who have made the experience of this master's, and my bachelor's before that some of the best years of my life.

Àlex De Los Santos Subirats
Delft, August 2024

Contents

Preface **i**

Nomenclature **iv**

1 Introduction **1**

Academic Article **13**

2 Neural Networks **13**

2.1 Image Classification Problem and Datasets . . . 13
2.2 Training Process . . . 13
2.3 Network Layers . . . 14
2.3.1 Fully Connected Layers and Multi-Layer Perceptrons . . . 14
2.3.2 Convolutional Layers and Convolutional Neural Networks . . . 15
2.4 backpropagation . . . 18
2.4.1 Loss Calculation . . . 18
2.4.2 Gradient Descent . . . 18
2.4.3 backpropagation Calculations, The Chain Rule . . . 19
2.5 Hyperparamater Optimization . . . 21
2.6 Overfitting . . . 21

3 Spiking Neural Networks **23**

3.1 Neuromorphic Computing . . . 23
3.2 Spike Encoding . . . 23
3.3 Forward Pass . . . 24
3.3.1 Neurons . . . 24
3.3.2 Intuition Behind Spike Cascading . . . 25
3.4 Dead Neuron Problem . . . 25
3.5 Training Spiking Neural Networks . . . 26
3.5.1 Shadow Training . . . 26
3.5.2 Spike backpropagation and Surrogate Gradients . . . 26
3.5.3 Spike Time backpropagation . . . 28
3.6 Our Implementation . . . 28
3.6.1 Our Neurons . . . 28
3.6.2 Spike Time Calculations In The Forward pass . . . 29
3.6.3 Spike Time Loss Function . . . 30
3.6.4 Closed Form Solution . . . 30
3.6.5 Backward pass . . . 31

4 Residual Connections **33**

4.1 What Is A Residual Connection? . . . 33
4.2 Why They Are Used? . . . 33
4.3 Residual Connections in Spiking Neural Networks . . . 34
4.3.1 How Are They Implemented In Literature? . . . 34
4.3.2 Banishing Spikes . . . 34
4.3.3 Why Residual Connections Are Important In Spiking Neural Networks? . . . 34
4.3.4 How They Can Be Implemented In Our Scenario? . . . 35
4.3.5 Forward Pass With Residual Connections . . . 35
4.3.6 backpropagation With Residual Connections . . . 35
4.3.7 Implementation Of The Residual Connection . . . 36
4.3.8 Improvement To The Existing Code Base . . . 37

4.3.9 Addition Of Learning Rate Schedulers 38
References **39**
5 Source Code Example **41**

Nomenclature

Abbreviations

Abbreviation	Definition
NNs	Neural Networks
ANNs	Artificial Neural Networks
SNNs	Spiking Neural Networks
MLP	Multi Layer Perceptron
CNN	Convolutional Neural Network

Nomenclature

Nomenclature	Definition
Jump connection	The residual connection channel that skips d layers.
Residual connection	The residual connection channel that has flowed though the d layers.

1 Introduction

In recent times the world of artificial intelligence has seen an incredible amount of progress in the recent years, tasks that were once considered extremely hard, such as image recognition and generation, and large natural language models amongst other things have been achieving incredible results. However, in most of these fields, the marvel that is the human brain continues to outperform current neural networks (NN) and deep learning (DL) models. This is especially the case in visual recognition tasks, where a human brain can recognize objects and patterns with truly astonishing accuracy.

That raises the question, if the brain is so good at this sort of task, why not try and copy it? This very question is what has led to the development of *Spiking Neural Networks*. These networks attempt to model the way the human brain works by using *Neurons* that output *Spikes* over time to other neurons to generate an output. This idea of copying the brain, as one might expect, is as difficult as one might expect. Significant technical and mathematical problems need to be considered when implementing SNNs. Over the last few years, these problems have been slowly overcome, resulting in the state-of-the-art neuron model at the time of writing: a network with multi-spiking precise time neurons that can learn through precise time backpropagation [2]. It was the release of this paper, and the development of the neuron described in it, as well as its limitations that gave rise to this thesis. The networks built with this neuron were limited in depth by silent outputs as the network grew deeper. In order to tackle this problem, we hypothesized that the addition of residual connections to this architecture would allow for deeper and better networks.

This thesis is structured in two main blocks. The first is an academic paper containing the main contribution and results of this thesis, while the second is a general overview of the field that covers all required topics to understand the aforementioned paper. We will first go over some relevant neural network knowledge, followed by a section on spiking neural networks (SNNs), and finally, a section on residual connections, where we go over what residual connections are, their uses, their past implementations in the literature and finally how we went about implementing them.

Residual Connections in Spiking Neural Networks (SNN)

Àlex De Los Santos Subirats
TU Delft
Mekelweg 5, 2628 CD Delft
alex.delossantossubirats@student.tudelft.nl

Abstract

In recent years the emergence of Spiking Neural Networks (SNNs) has shown that these networks are a promising alternative to traditional Artificial Neural Networks (ANNs) due to their low-power computing capabilities and noise robustness. Nevertheless, in recent approaches, they have either strayed away from spike time backpropagation, used discrete time, single spike neurons, and/or limited themselves to shallow networks. These approaches limit the potential of SNNs by sacrificing significant aspects of what makes them special in order to have a functional network.

It is for this reason that we believe that, the implementation of residual connections, allowing a model that has multi-spiking neurons, precise time and spike time backpropagation is the path to allow these networks to truly shine. As it will solve one of this model's most severe limitations which prevent it from being used to build deeper networks, the banishing of spikes at a deeper depth. Hence we aim to remedy this problem by feeding inputs from further up the network to revitalize the spike counts at deeper layers.

In this paper, we explore the implementation of residual connections in precise time multi-spiking neural networks as a way of solving the disappearing spikes problem that inhibits spike time backpropagation. We explore two alternatives in the implementation of the residual connection and we analyze how they affect the accuracy of the network as the depth increases in both Multi-Layer Perceptrons and Convolutional Neural Networks. We also developed an architecture to allow for swapping of the fuse function in order to take full advantage of the flexibility that precise time provides us. Results show that the implemented residual connections allow for deeper training, which has the potential to aid in the network's performance, although in some cases some hurdles remain.

1. Introduction

The field of computer vision has long been dominated by conventional neural networks, also called artificial neural networks. For the sake of clarity in this paper we will refer to them as Artificial Neural Networks (ANNs). Yet, in recent years there has been a parallel interest in so-called Spiking Neural Networks (SNNs), these networks attempt to more closely resemble brain biology and are comprised of spiking neurons instead of linear operations [26]. These networks in theory show a lot of promise as they imitate the way that neurons fire in biological brains [25]. In other words, these networks process information through sparse asynchronous events called spikes.

This design brings the main benefit that they can be implemented in brain-like neuromorphic hardware, for a fast and ultra-low power solution to power-efficient neural networks [1, 10, 18, 21]. While digital components for the implementation of neuromorphic systems do exist [1, 10] these systems perform with better processing time and power consumption in specialized hardware.

In order to further understand SNNs we should look at the advantages and disadvantages of past approaches to these networks [19].

1.1. Advantages Of SNNs

This approach of attempting to copy biology has a very promising set of advantages The main motivation for the use of SNNs is the fact that brains show an enviable performance in real-world tasks, and show such performance with an incredibly small amount of power consumption when compared to ANNs [19]. It is therefore not surprising that there has been a push to attempt to recreate their structure and work to use those properties for tasks such as image recognition.

Once trained, these networks are also able to output their results faster than ANNs as can be seen in [8]. This can be useful in implementations where the information is time-sensitive. Building on that they have shown a robustness to noise when compared to their ANN counterparts [15] which has great benefits and allows the network to better deal with noisy inputs.

Adding to that, the lower power consumption of these systems as seen in [8, 24], make them ideal for applications in limited power environments such as battery-powered robots or drones. It is also interesting to note that their inspiration, the human brain, has the ability to perform similar tasks with an ever lower power consumption [22].

Due to the nature of the network and its inspiration, it is a natural match for neuromorphic computational devices, running these networks in such devices makes the above factors stand out [19].

1.2. Downsides Of SNNs

These networks also come with problems when it comes to their implementation, which has led many approaches in the past to make compromises or use workarounds.

The first downside is the fact that SNN approaches fall behind in terms of performance on the traditional benchmark datasets [4, 13, 14]. This can be attributed to a problem with the existing benchmark datasets, as they were designed with ANNs in mind.

The second downside with SNNs is the fact transfer function is sometimes none differentiable, which makes the application of backpropagation techniques used in ANNs difficult [19].

In order to solve these complications with the transfer function some approaches [8] have opted for limiting the amount of spikes a neuron can output to a single one. This makes backpropagation easier but has the downside of limiting the amount of information that a neuron can transmit. In this paper, we will be using a multi-spike network that allows for several spikes per neuron on each layer.

A very common compromise that past approaches do, in particular those adding residual connections to SNNs is the use of discrete time [16, 23]. This is done for two main reasons, firstly it makes backpropagation simpler, and secondly, it allows for the use of binary logic operations in the residual connection fuse point. On the other hand, similar to limiting the spikes per neuron this causes the network to be able to transmit less precise information.

Finally, and more relevantly for the purposes of this writing is the issue with *disappearing spikes* limiting the depths of the networks, as is the case in [2]. This phenomenon refers to each layer producing fewer spikes than the previous layer until no spikes are being outputted beyond a certain layer. In our approach, just as is the case in the BATS approach [3], these banishing spikes are the source highlighted by our use of *spike time backpropagation*. This means that backpropagation is based on the difference between the desired spike time and the obtained spike time. In this scenario no backpropagation can be applied if there is no spike, and hence no corresponding spike time. Despite this problem *spike time backpropagation* is a very desirable method due to its accuracy, as no the backpropagation is performed on the actual values flowing through the network without the use of workarounds, as would be the case if we used methods such as shadow training or surrogate gradients.

1.3. Contributions

Our main goal is to propose a method of implementing residual connection that can allow spike time backpropagation methods such as [2] to function at deeper depths. We expect that this will improve the network's performance. In other words, we aim to explore the possibility of deeper and more accurate Spiking Neural Networks by answering the following questions: *Do residual connections allow for training of deeper SNN? What is the best way to implement said residual connections?* and *Do these deeper SNNs see an increase in performance?*.

In this work, we explore the possible methods of implementing residual connections in SNN based on the neurons developed by [3]. We test these connections at different depths, datasets and architectures. Our main contributions are:

1. Creation of an architecture to allow for the use of residual connections in SNN.
2. The development of two novel fuse functions which are designed to work in multi-spiking continuous time in both MLP and CNN architectures.
3. The training of deeper networks thanks to the extra neuron activity provided by these fuse functions.
4. We test the implementation of a mean shift delay to the proposed fuse functions.

The rest of the article is structured as follows. Section 2 discusses past approaches to SNNs and residual connections. Section 3 covers the mathematical principles of the neurons form [3] that we use as well as the modifications done to accommodate residual connections. Finally in sections 4 and 5 we discuss the experiments we performed and their results. Finally, we discuss our results, the possible future research avenues and a conclusion.

2. Related Work

Having reviewed the current state of the field, we will now examine the most relevant papers related to our approach and delineate our contributions in relation to these works. These attempts have faced distinct challenges, such as imprecise backpropagation [5], the limited amount of spikes for each neuron [8], discrete-time [16, 23], and limited depth [2].

In the past, deep SNNs have been implemented, such as in the case of [5]. In this approach, they find two main issues with the backpropagation calculations. First is that SNNs use delta functions in order to represent their output spikes, and to be able to perform traditional backpropagation you require the derivative of the output with respect to the weight, and a delta function is non-differentiable. This can be addressed by the use of substitute or approximate gradients, nevertheless, it will be less accurate. Second, they have been shown to suffer from what has been termed the "weight transport" problem [7]. For both these issues workarounds can be used, however, these lead to a less precise transfer of information in the backwards pass.

Another notable approach is the Fast & Deep method suggested in [8], this paper provides a closed-form solution to the spike times by restraining the synaptic and membrane times. This exact expression of the spike times allows for fast and accurate backpropagation, they do this by iterating the ordered pre-synaptic spikes (spikes a layer receives). The precision of this network allowed it to be implemented in analogue neuromorphic processors. Crucially this method was the first to allow for exact backpropagation through spikes without requiring any assumptions about the relation between weights and spikes, allowing for novel and more direct training techniques for SNNs upon which other papers such as [2] have built. However, this Fast & Deep approach only encoding allowed for a neuron to fire a single spike in each forward pass, so again it is limited.

Building on the last paper and attempting to fix the single spike per neuron issue is the BATS approach seen in [2], this paper's neuron implementation is the one upon which we build our own residual network approach. This method allowed for the multiple spikes per neuron and hence the use of rate encoding rather the latency encoding. Yet, suffered from the banishing spikes problem, only allowing for shallow networks, limiting its effectiveness and applications.

Finally, it is important to note previous attempts to implement residual connections in spiking neural networks such as [12, 16, 23]. These approaches have shown promising results with the use of binary functions to join the different inputs from two input layers and have shown that the addition of a delay, especially if it is learnt can be beneficial to the network's performance. In particular, the case of [12] demonstrates that such networks are capable of learning more complex image recognition tasks, such as CIFAR [13] and ImageNet [20]. However, these approaches only have been implemented in discrete timings for the spikes which makes them require modifications to be implemented in precise time.

Hence our goal is to build on the [2] multi-piking precise time neuron with precise spike time backpropagation by adding residual connections to it in order to allow for deeper networks.

3. Approach

In this research, we intend to build an exact-time multi-spiking neural network architecture with residual connections in both MLP and CNN architectures. We aim to investigate if these changes can allow for the training of deeper networks and how those deeper networks perform. In this we are inspired by [9] with the aim to solve the limitations of the [3] approach.

In this section we shall cover the neurons used in our approach, the different ways we implement the residual connection, the delay used, and the learning rate schedulers developed.

3.1. Neurons

Like [2] we use a network of Current-Based Leaky Integrate-and-Fire neurons with a soft reset of the membrane potential [2, 6, 8]. This means that the membrane potential of neuron j in layer l referred as $u^{(l,j)}$ is given by the following equations:

$$u^{(l,j)}(t) = \sum_{i=1}^{N^{(l-1)}} w_{i,j}^{l} \sum_{z=1}^{n^{(l-1,i)}} \epsilon(t - t_z^{(l-1,i)}) - \sum_{z=1}^{n^{(l,j)}} \eta(t - t_z^{(l,j)})$$

And then:

$$\frac{du^{(l,j)}}{dt} = -\frac{1}{\tau} u^{(l,j)}(t) + g^{(l,j)}(t) - \underbrace{\vartheta \delta(u^{(l,j)}(t) - \vartheta)}_{\text{reset}}$$

$$\frac{dg^{(l,j)}}{dt} = -\frac{1}{\tau} g^{(l,j)} + \underbrace{\sum_{i=1}^{N^{(l-1)}} w_{i,j}^{(l)} \sum_{z=1}^{n^{(l-1,i)}} \delta(t - t_(^l - 1, i))}_{\text{Pre-synaptic spikes}}$$

where:

$$\delta = \begin{cases} +\infty & x = 0 \\ 0 & otherwise \end{cases}$$

- δ is a delta function that is used as the spike with $\int_{-\infty}^{+\infty} \delta(x)\, dx = 1$.
- $N^{(i)}$ represents the number of neurons at layer l.
- $n^{(l,j)}$ is the number of spikes of neuron j in layer l.
- $t_k^{(l,j)}$ is the k_{th} spike at of neuron $n^{(l,j)}$.
- $w_{i,j}^{l}$ is the weight that represents the intensity of the connection between neuron i in layer $l-1$ and neuron j in layer l.
- $g^{(l,j)}$ is the post-synaptic current, τ and τ_s the membrane and synaptic time constants that control the decay of the membrane potential and the post-synaptic current respectively.

- ϑ is the activation threshold of the neuron at which a post-synaptic spike is emitted, and then resets $u^{(l,j)}$ by reducing the potential by $\eta(t)$.

with $\epsilon(t)$ being the Post-Synaptic Potential (PSP) kernel determining how the neuron reacts to pre-synaptic times, and $\eta(t)$ being the refractory kernel determining the reset behaviour after a neuron spike.

3.2. Gradient Calculations

We will now briefly outline the process used to calculate the gradient for the weights of this network. The fact that the spike timing now has a closed-form solution is what allows these exact gradient calculations (this is the same solution used in [2]). A weight $w^{(l)}_{i,j}$ between neuron pre-synaptic neuron i and post-synaptic neuron j at layer l receives an error $\delta^{(l,j)}_k$ though backpropagation, this spike error is calculated in the same way as in [2]. This lead to a change $w^{(l)}_{i,j} \rightarrow w^{(l)}_{i,j} - \lambda \Delta w^{(l)}_{i,j}$

$$\Delta w^{(l)}_{i,j} = \sum_{k=1}^{n^{(l,j)}} \frac{\partial \mathcal{L}}{\partial t^{(l,j)}_k} \frac{\partial t^{(l,j)}_k}{w^{(l)}_{i,j}} = \sum_{k=1}^{n^{(l,j)}} \delta^{(l,j)}_k \frac{\partial t^{(l,j)}_k}{w^{(l)}_{i,j}}$$

In short, the gradient depends on two main components:

The first one is the loss associated with the weight $\delta^{(l,j)}_k$, which contains both intra-neuron and inter-neuron dependencies (how spikes from other neurons and the previous spikes before k of this neuron have effected the error). The calculations behind obtaining said value are covered in equation 2 for the res

The second one consists of $\frac{\partial t^{(l,j)}_k}{w^{(l)}_{i,j}}$, so how this weight has affected the timing of the spike k, the precise formula for $\frac{\partial t^{(l,j)}_k}{w^{(l)}_{i,j}}$ can be found in the [2] paper and in this paper's accompanying thesis document.

3.3. Loss Function

We use a loss function that aims to minimize the distance between the desired spike count and the number of spike the neuron outputted. Hence the loss function is:

$$\mathcal{L} = \frac{1}{2} \sum_{j=1}^{n^{(o)}} \left(y_j - n^{(o,j)} \right)^2$$

Where:

- y_j is the desired amount of output spikes for neuron i.
- $n^{(o,j)}$ is the obtained number of output spikes from neuron j in the output layer o.
- $n^{(o)}$ is the number of neurons in the output layer o.

The target number of spikes on the correct and incorrect labels are constant hyperparameters usually set to a ratio of 1/10.

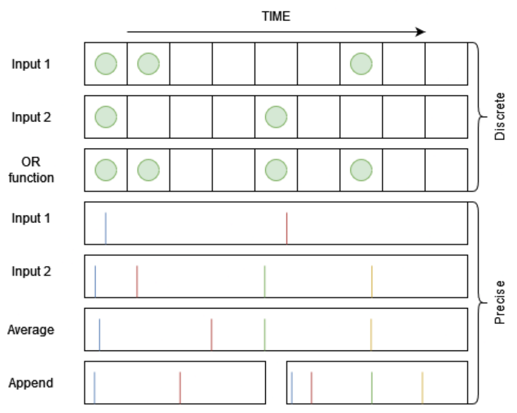


Figure 1. In this figure you can see on the top how a fuse function in a residual connection in discrete can be a logical function (in this case OR). While in the case of precise time, we have a lot more flexibility. In the "Average" entry, we display the result of applying the **average fuse function** to the two precise time inputs, while in the "Append" case we show a representation of applying the **append fuse function** to those same precise inputs.

3.4. Residual Connections

Our approach builds on [3] by adding residual connections to a network with precise and exact backpropagation on neurons that can fire multiple times. The aim is to allow for the networks to learn when using deeper architectures with the goal of having this extra depth aid in achieving better performance in more complex problems.

These residual connections are implemented in the form of *Residual Layers* that help add the outputs of a layer further up the network into the input for the *Residual Layer* and then this layer computes an output based on both the *Previous Layer* and the *Jump Layer*. The reason for this is that the addition or fusing of spikes in precise time is more complex than when you use discrete timing and we therefore can not use a binary logic function like in [23] as can be seen in figure 1.

These residual connections allow for precise time and multiple spikes just as the [3] implementation does.

3.4.1 Fuse Functions

There are many ways to combine of the inputs from the *Jump* and *Previous* layers, in this section we will go over the options and the justifications for each approach. The first two describe approaches for fully connected layers while the last one describes an approach for a convolutional architecture.

The first of these *fuse functions*, is called the **average fuse function**. As the name implies it is a simple averaging out of the spike times, the results of this fuse function can be seen in figure 1 for a single neuron. In our implementation, the spikes are represented by a matrix of shape (*b*,*n*, *s*); where *b* is the batch size, *n* is the number of output neurons from the previous layer, and *s* is the maximum number of spikes a neuron can output. For this *fuse function* it is necessary that both inputs be the same *b* and *n*. If that is given the procedure is simple: for each set of output spikes for a neuron in the *previous layer* we check it with the equivalent neuron in the *Jump Layer*. Since for a given neuron in a given batch, the spikes are sorted from earliest to latest we take inspiration from both [23] and regular residual connections checking both neurons have a spike in that position, if they do we take the average of both spike times if only one of them has a spike we take that spike time (similar to an OR gate), and if none of them have given a spike we do not output a spike. This function is simple and takes inspiration from past approaches using boolean fuse functions [23], however, it has the downside of information loss as with the averaging out operation you fuse two precise spike times, that could be carrying different information, into one.

The second *fuse function* is one we shall refer to as the **Append fuse function** seen in figure 1. As the name implies we append the outputs from both the *Jump* and *Previous* layers into a single output of size with a number of neurons the size of the sum of the number of neurons from both layers. In this way we avoid the problem of information loss of the previous approach, this, however, requires more weight and therefore increases training time and cost.

Finally, the **Convolutional fuse functions** are the versions of the above **append and average fuse functions** but designed for a convolutional architecture. These functions are similar to the previously described fuse functions, except for some changes to make them fit the architecture and that in the case of the **append convolutional function**, instead of appending on the neuron dimensions we append on the channel dimensions of the convolutional layer. As an example, this means that if the *Previous Layer* has 30 channels and the *Jump Layer* has 15 then the residual layer will receive an input of 45 channels.

The math behind the implementation for the **append fuse function** and its **convolutional** counterpart are described in equations 1 for the forward pass, and equation 2 for the loss associated with the k_{th} spike of neuron j in layer l, formally referred to as $\delta_k^{(l,j)}$. In comparison to the equations of the non-residual forward and backward pass equations described in [3] the main difference is the input from the jump connection at layer $l-d$ in the forward pass and a change in the inter-neuronal component of the backward pass equation. Equation 2 shows the forward pass for a residual layer l that receives inputs from layer $l-1$ and jump layer $l+d$. An effort has been made to keep the same notation as in the [2] paper, similarly equation 2 shows the backward pass from a layer l that has a jump to a layer $l+d$.

For the layers without residual connections, they use a version of equation 2 without the **Inter Neuronal (part 2)** component.

$$u^{(l,j)}(t) = \sum_{i=1}^{N^{(l-1)}} w_{i,j}^{l} \sum_{z=1}^{n^{(l-1,i)}} \epsilon(t - t_z^{(l-1,i)})$$
$$+ \underbrace{\sum_{u=1}^{N^{(l-d)}} w_{u,j}^{l} \sum_{z=1}^{n^{(l-d,u)}} \epsilon(t - t_z^{(l-d,u)})}_{\text{Input from the jump connection at layer } l-d} - \sum_{z=1}^{n^{(l,j)}} \eta(t - t_z^{(l,j)}) \tag{1}$$

$$\delta_k^{(l,j)} := \frac{\partial \mathcal{L}}{\partial t_k^{(l,j)}} = \phi_k^{(l,j)} + \mu_k^{(l,j)} =$$
$$\underbrace{\frac{\left(\sum_{i=1}^{N^{(l+1)}} \sum_{z=1}^{n^{(l+1,i)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l+1,i)}} \frac{\partial t_z^{(l+1,i)}}{\partial t_k^{(l,j)}}\right)}{2}}_{\phi_k^{(l,j)} \rightarrow \textbf{Inter Neuronal (part 1)}}$$
$$+ \underbrace{\frac{\left(\sum_{i=1}^{N^{(l+d)}} \sum_{z=1}^{n^{(l+d,i)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l+d,i)}} \frac{\partial t_z^{(l+d,i)}}{\partial t_k^{(l,j)}}\right)}{2}}_{\phi_k^{(l,j)} \rightarrow \textbf{Inter Neuronal (part 2)}} \tag{2}$$
$$+ \underbrace{\sum_{z=k+1}^{n^{(l,j)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l,j)}} \frac{\partial t_z^{(l,j)}}{\partial t_k^{(l,j)}}}_{\mu_k^{(l,j)} \rightarrow \textbf{Intra Neuronal}}$$

where:

- $\mathcal{L}$ is the loss value obtained from calculating the loss of the last forward pass using the method described in section 3.3.
- $\delta_k^{(l,j)}$ loss associated with the k_{th} spike of neuron j on layer l.
- $N^{(i)}$ represents the number of neurons at layer l.
- $n^{(l,j)}$ is the number of spikes of neuron j in layer l.
- $t_k^{(l,j)}$ is the k_{th} spike at of neuron $n^{(l,j)}$.
- $w_{i,j}^{l}$ is the weight that represents the intensity of the connection between neuron i in layer $l-1$ and neuron j in layer l.

In the case of the **average fuse function** and its **convolutional** pair the math is more complex as the effect from the different layers depends on the relation between the number and times of the spikes from the two input layers as described earlier.

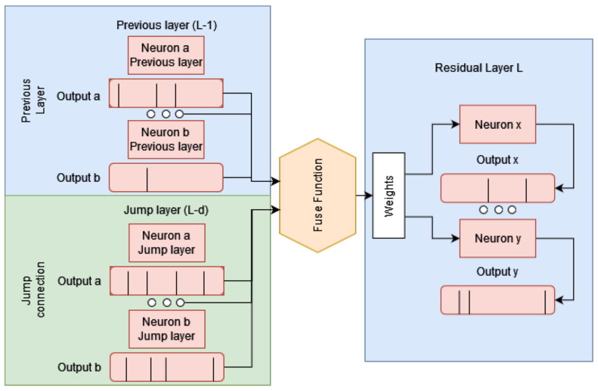


Figure 2. In this figure we show how the fuse function would affect the input from the *Previous layer* and *Residual layer* as a means to feed it into the residual layer. Note that the inputs are fused before the weights of the layer so that the network and that depending on the fuse function the number of wights changes (in this diagram there would be two wights in the *Average* function and 4 in the case of the *Append*).

For all of these fuse functions the possibility of adding a delay, as they do in [16], is very intuitive. This gives the possibility of this delay being made a learned variable, this nonetheless, is beyond the scope of this paper.

An overlook of how these fuse functions interact with the architecture and how the residual layers are implemented can be seen in figure 2.

3.5. Delay

We implement the possibility of a time delay for the spikes from the jump connection. The delay amount is calculated such that the average of both spike clusters are located at the same point in time. This is done by calculating the mean of the spike times from both the residual and jump connections mt_r and mt_j respectively and then calculating the difference to delay all jump connection spikes by the given amount.

The goal with this is to attempt to give equal conditions to both sets of spikes so that neither one nor the other suffers the risk of being ignored due to arriving too early or too late. Yet, as seen in the results this was not the case.

3.6. Learning Rate Scheduler

Finally, it is worth noting that we implemented a more advanced learning rate scheduler than the one in [2], which simply reduces the learning rate by a fixed amount every N epochs. In our case, we use two different learning rate schedulers, one that reduces the learning rate when the slope of the last five epochs is higher than -0.1. The second one reduces the learning rate by the same fraction when the lowest training loss epoch was more than five epochs away.

4. Methodology

In this section, we will go over how we tested the previously mentioned residual layers. For the purpose of testing the performance of these networks at different depths, we shall use the following benchmark datasets: MNIST, EMNIST, Fashion MNIST, and CIFAR-10.

4.1. Multi Layer Perceptrons

We decided to test on *Multi Layer Perceptrons* as they are one of the simplest architectures and would serve as a proof of concept as well as a solid point of comparison with the approach in [3]. Unlike in the [3] approach we used layers consisting of 600 neurons each.

When testing the *Fully Connected Residual Layer* we used a standard multi-layer architecture with residual connections skipping two layers, like it was done in the [9] paper. With the goal of testing how these residual connections perform, we compared them on the four above-mentioned datasets with different depths.

Since we are interested in how the residual connections affect the network's ability to be deeper we test using the following depth: 1, 4 (5), 9, and 13. The reason for these numbers is the following. Depth of 1 is the baseline as used in [2], the depth of 4 (5 in the case of EMNIST) is the maximum depth to which the networks without residual connections can be stretched, then 9 and 13 are depths that have additional residual connections, as we are using a jump of 3 layers every 4 as they do in [9].

4.2. Convolutional Neural Network

After the MLP tests, we decided to test on an *Convolutional Neural Network* due to their properties of translation invariance and their ability to increase their receptive field as depth increases. We believe that these allow them to better shine with the added depth that residual connections allow.

Unlike in the [3] approach we used kernels of size 5x5 with 16 channels each, we chose these values because we wanted to increase the receptive field, but that meant we had to limit the number of channels when compared to standard implementations, in which the number of channels tends to be values such as 64 channels and increase with the depth of the layer [17]. Our CNN network consists of an input layer, an n amount of layers, a feedforward layer and finally an output layer. We chose this architecture as it was the simplest CNN architecture and would allow us to test the effects of the residual connections without external variables.

When testing the performance of the *Convolutional Layer* and how residual connections affect the performance of multi-spiking exact time spiking neural networks we will test them using both fuse functions and with one, three and four layers. The single-layer case is meant as a baseline for the current state of the art, while the three-layer is a baseline for the performance on the max depth the non-residual network can achieve, finally the four-layer network is used to show that the extra depth added thanks to the residual connections improves results.

5. Results

In this section, we will present the experimental results in relation to our primary research questions. To answer them we have tested out fuse functions at different depths with the datasets MNIST, EMNIST, Fashion MNIST, and CIFAR-10. As mentioned in section 4 we use the MLP experiments in order to test our baseline assumptions and then attempt to exploit the benefit of our implementation in the CNN architecture.

5.1. Do Residual Connections Allow For Training Of Deeper SNN?

In this section, we seek to investigate if the implemented residual connections help solve the decaying spikes problem. The fact that we can train with deeper networks is self-evident in the result as we get an accuracy above random chance and learning. As we expected we also saw that the addition of residual connections kept all the layers in the network producing output spikes. It is important to note that in order to avoid the appearance of these "dead" layers the frequency of residual connections and the length of the jump needs to be similar to the max depth the network can achieve without residual connections.

5.2. What Is Best Approach To Implement Residual Connections?

As mentioned there are many possible ways to implement these residual connections in a precise time SNN. In this paper, we have tested two approaches and the effects of adding delay. In this section, we will go over the results of the experiments performed and discuss what fuse function should be used and how the addition of delay should be handled. For this, we will mainly focus on the results obtained from the MLP experiments as those allow for testing in a shorter time frame and due to lack of time many of these experiments could not be performed on a CNN architecture, in addition to this MLP can provide a good baseline on how depth affects the ability of the network to learn and how it affects overfitting.

5.2.1 What Fuse Function Should Be Used?

When deciding what fuse function should be used one should take into account what architecture it is being used in, when testing both functions we saw a discrepancy between how they compared in MLP architectures and how they compared in CNN architectures.

Layers	Datasets	No Residual	Average	Append
1	MNIST	98.2	NA	NA
4	MNIST	97.6	**97.8**	97.2
9	MNIST	No BP	96.9	**97.0**
13	MNIST	No BP	95.1	**97.1**
1	EMNIST	75.3	NA	NA
4	EMNIST	**82.6**	78.6	81.3
5	EMNIST	76.4	80.1	**82.7**
9	EMNIST	No BP	78.7	**83.1**
13	EMNIST	No BP	71.6	**75.5**
1	Fashion	88.8	NA	NA
4	Fashion	88.6	**89.9**	88.8
9	Fashion	No BP	**87.2**	80.7
13	Fashion	No BP	75.8	**77.9**
1	CIFAR-10	25.8	NA	NA
4	CIFAR-10	20.9	23.2	**24.4**
9	CIFAR-10	No BP	19.4	**20.6**
13	CIFAR-10	No BP	16.0	**16.3**

Table 1. **Test accuracy** of the different **MLP** experiment. "NA" means that the experiments can not be run due to residual connections needing more than one layer. "No BP" refers to no backpropagation as there are no spikes in the output. The bold entries represent the best results at a given depth.

Layers	Datasets	No Residual	Average	Append
1	MNIST	98.9	NA	NA
4	MNIST	99.6	99.8	**99.8**
9	MNIST	No BP	99.2	**99.8**
13	MNIST	No BP	98.0	**99.8**
1	EMNIST	76.7	NA	NA
4	EMNIST	**88.4**	80.2	85.5
5	EMNIST	80.1	81.4	**86.2**
9	EMNIST	No BP	79.2	**88.5**
13	EMNIST	No BP	73.0	**80.9**
1	Fashion	91.6	NA	NA
4	Fashion	99.1	98.5	**99.4**
9	Fashion	No BP	**98.9**	88.7
13	Fashion	No BP	87.2	**87.2**
1	CIFAR-10	30.6	NA	NA
4	CIFAR-10	30.7	30.0	**33.1**
9	CIFAR-10	No BP	29.3	**31.3**
13	CIFAR-10	No BP	22.0	**22.8**

Table 2. **Training accuracy** of the different **MLP** experiment. "NA" means that the experiments can not be run due to residual connections needing more than one layer. "No BP" refers to no backpropagation as there are no spikes in the output. The bold entries represent the best results at a given depth.

In the MLP test of table 1 we saw that the **average fuse function** tends to perform better, initially it would be tempting to see that this is due to the lower number of weights

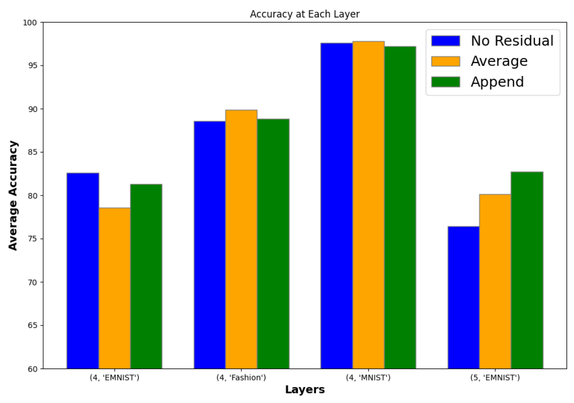


Figure 3. **Test accuracy** obtained at the depth where the **MLP** network still has output spikes, in this figure we can see that the addition of residual connections overall improves results, especially at deeper layers. **No Residual** is shown in blue, **Average** is shown in yellow, and **Append** is shown in green.

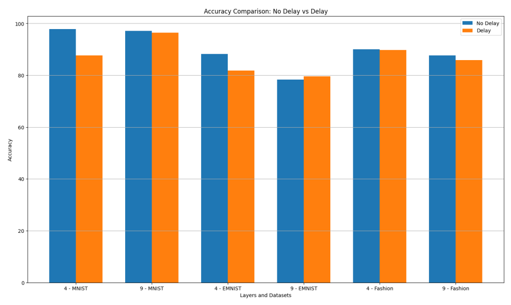


Figure 4. **Test accuracy** results of adding delay on a 4 and 9-layer **MLP** network.

used in the **average function** when compared to the **append fuse function** causing it to overfit less. Yet, this does not seem to be the case as the training data results 2 should then be better for Append. That then leaves us with the option that the additional weights are not needed, as the task is relatively simple, and hence they are mainly acting as a source of noise that is decreasing the accuracy. We also believe that the lower accuracy at these increased depths could be due to the hyperparameters selected being imported from [3] and therefore focused on networks of one or two layers.

On the other hand, the **append function** performed better in the CNN experiments in figure 5 and table 3 as it has more weights and less loss of information. Nevertheless, the **average function** is not lagging far behind and has less weight.

Due to the above-mentioned when using more complex image recognition tasks, we recommend the use of a CNN with the **Append fuse function**, as it seems to give the best results, on the other hand, when doing an MLP task we suggest the use of an **average fuse function** as it is less prone to overfitting. Nevertheless, for different cases some trial and error might be required, and even the possibility of creating a new fuse function implementing domain knowledge of the given problem to fit into the architecture. For this reason, we designed the code base to be able to accommodate new fuse functions.

5.2.2 Should Non-learnable Delay Be Used?

In short, as seen in table 4 the delay was not useful and using it produced a drop in accuracy all across the board. This is not surprising as it is not a learnable delay, however, implementing a learnable delay fell outside the scope of this project. Due to the results in table 4, we stopped testing the delay in other scenarios, with the purpose of focusing our attention on other parts of the proposal. As it didn't show a significant improvement and caused a significant drop in accuracy in most tested cases.

5.3. Do These Deeper Spiking Neural Networks Perform Better?

As can be seen in tables 1 and 2 we see an increase in accuracy when the residual connections are added as seen in figure 3 and the depth is increased. This is especially noticeable in table 2 where the MLPs overfit in the training data, showing that it is indeed capable of learning more complex problems with depth.

However, the most relevant results are in the results obtained in the CNN experiments (seen in figure 5 and table 3), this is because CNNs have been known to increase in accuracy with depth as their receptive field increases. Hence we expected the best results in relation to depth in this model. It is important to note that no pooling layers were used in the architecture as a means of simply testing the residual connections without concern for any external influence from other factors.

When comparing the network without residuals stretched as deep as it will go to the single-layer baseline we see that there is an expected decrease in accuracy. We believe this to be due to the network having a harder time getting output spikes at this depth. However, as expected when residual connections are added the accuracy increases even without increasing the depth. This shows that the increased spike activity in the network caused by the residual connections is beneficial to the network's performance and accuracy.

Finally, when increasing the depth to four, a depth in which without residual connections there is no backpropagation, we see generally a further increase in accuracy. This is especially the case in the experiments with the EMNIST

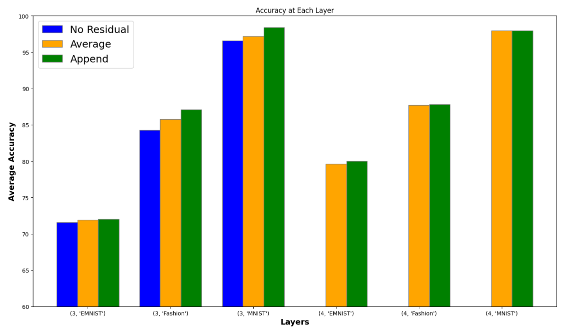


Figure 5. Comparison of the **testing accuracy** plot for an **CNN** architecture in the different datasets and fuse functions as the depth increases. **No Residual** is shown in blue, **Average** is shown in yellow, and **Append** is shown in green.

dataset. Additionally to what was shown before, this points toward the fact that the added depth allowed by the residual connections does indeed have a positive effect on the accuracy of the network.

Despite this, none of these deeper approaches have been able to beat the hyperparameter optimized single layer replication of the [2] paper. We believe this is due to the lack of hyperparameter optimization on the side of the residual connection neuron parameters, as well as some tuning being required in non-residual hyperparameters. With the goal of addressing this, we attempted to optimize the hyperparameters, yet due to long run times and the limited time available for this project, we could not do an optimization of all hyperparameters. Hence we decided to focus on changing the distribution of the weight initialization and implementing better learning rate schedulers, we believed that these would be parameters that would have a large effect on the accuracy of the network. However, this process was unsuccessful in getting to surpass the single-layer results. In this situation, the hyperparameter optimization is extremely time-consuming as a single run of a CNN to convergence on the machine we are using is about 7 days.

5.4. Hyperparameter Optimization

Since the Hyperparameter from [2] had been optimized with shallow networks in mind, we considered that in order to achieve a higher accuracy we would also need to perform some optimization. Nevertheless, given the time limitation and long run times of the deeper networks (especially the CNNs in table 3) we were limited on the number of hyperparameters we could optimize, hence we decided to focus on adding a learning rate scheduler and investigation the effects of changing the initial weight distribution. The results of this can be seen in 4.

Even so, although some changes did have an effect on the

Layers	Datasets	No Residual	Average	Append
1	MNIST	98.7	NA	NA
3	MNIST	96.6	98.4	97.2
4	MNIST	No BP	98.0	98.0
1	EMNIST	84.4	NA	NA
3	EMNIST	71.6	72.0	71.9
4	EMNIST	No BP	80.0	79.6
1	Fashion	90.0	NA	NA
3	Fashion	84.3	87.1	85.8
4	Fashion	No BP	87.8	87.7

Table 3. **Test accuracy** results for the **CNN** architecture tests. The same terminology as in tables 1 and 2 is used.

Dataset	LR scheduler	(-1,1)	(-1,2)	(-5,5)
MNIST	Slope	**97.9**	96.0	96.7
MNIST	No Slope	96.3	97.3	96.8
EMNIST	Slope	**83.8**	30.0	73.3
EMNIST	No Slope	66.5	78.1	73.7
Fashion	Slope	73.1	63.2	80.2
Fashion	No Slope	**86.6**	86.4	61.2
CIFAR-10	Slope	**19.6**	16.4	17.3
CIFAR-10	No Slope	18.4	15.5	15.6

Dataset	LR scheduler	(-1,1)	(-1,2)	(-5,5)
MNIST	Slope	**98.0**	97.5	96.0
MNIST	No Slope	97.2	97.5	96.4
EMNIST	Slope	79.6	78.6	72.2
EMNIST	No Slope	79.5	**80.3**	74.8
Fashion	Slope	**87.7**	87.7	79.7
Fashion	No Slope	86.7	84.1	78.2
CIFAR-10	Slope	19.6	16.4	17.3
CIFAR-10	No Slope	**19.9**	19.6	18.6

Table 4. **Test accuracy** of the hyperparameter optimization attempts on a 4-layer **CNN** architecture with the **append fuse function** on the top table and the **average fuse function** in the lower table. In each dataset, the entry marking the best hyperparameter combination has been highlighted.

accuracy we see that increasing the initial weight distribution makes the network more unstable, and since the (-1,1) uniform weight distribution has the most consistently high results we recommend using the (-1,1) uniform distribution with the slope learning rate scheduler, as it gives some of the more consistent results.

5.5. How Do They Preform On A More Complex Dataset?

Finally, we wanted to see if these deeper networks could aid in the learning of a more complex dataset. For this purpose, we performed some tests on the CIFAR-10 dataset [13]. We tested both the MLP and CNN architectures on the dataset and obtained the results shown in tables 1 and 2 for MLP results, and 4 for the CNN results. Sadly we have been unable to achieve the results we desired, although the addition of residual connections and some additional depth does seem to have a positive effect on the network's accuracy. We attribute this underperformance to a few reasons. The first is the lack of time for the long experiments and computational times required, as with these networks being the most complex ones each epoch takes the most time out of the other datasets and they require more epochs to converge. Secondly, we speculate that the lack of pooling layers throughout our architecture has a negative effect, these layers although implemented in the code base were not used as we wanted to focus on the effect of the residual connections. Thirdly this was the only of the tested datasets containing coloured RGB images without a uniform background, we hypothesise that this property of that dataset combined with the spike encoding inherited from [3] could be a feasible explanation for the decrease in accuracy. Intending to remedy it, we attempted two solutions such as grey-scaling the image, and adding a minimum amount of intensity that a pixel needed in order to produce an input spike. Yet none of these proved to improve the performance to any significant extent.

Finally is the fact that the CNNs are of relatively shallow depth, which combined with the lack of pooling layers limits the receptive field size quite considerably.

6. Discussion

In this section there are three points we would like to discuss in regards to this paper, the methodology choices and results.

Firstly, is that due to the fact that the goal of this paper was to evaluate how residual connections affected the performance of the SNN neurons developed by [3] we used the same datasets as a means of aiding the result comparison between the two methods. While these datasets are widely regarded as excellent benchmarks for comparison due to their standard use in image recognition, compatibility with various architectures, and ease of use, it is important to recognize that they were not designed with event-based systems in mind. As a result, they do not fully leverage the advantages offered by SNNs. We believe that results that do not focus on comparison to other methods should, in the future, focus on datasets composed of data such as event camera images or audio datasets.

Secondly, in reference to the long run times of the network, it is important to note that these runs were computed in a traditional computer as we had no access to a neuromorphic chip, hyperparamater optimization might be aided incredibly by the reduced runtime achieved from using such technology.

Finally, when comparing our results to those obtained by the [3] paper it is important to note that in an effort to save computation time, we use fewer neurons on each layer in the MLP architectures and fewer channels in the CNN architectures. It is because of this that we made our own baseline runs to use as a point of comparison. Additionally, regarding the CIFAR-10 results we want to note that neither in the [3] or [2] paper were there any attempts on datasets of this complexity. To the best of our knowledge, there are currently no papers that use neurons of this type on datasets such as CIFAR-10.

7. Future Work

There are a few areas of this approach that we believe have room for improvement for any future researcher who chooses to pursue it. These are as follows.

Firstly, continue the hyperparameter optimization in order to show if the deeper CNNs can beat the single-layer CNN, we believe that the parameters that could have the greatest effect, given that the initial weight distribution did not fix it, are the residual neuron threshold parameters.

Secondly, the implementation of a learnable delay and the testing of its performance. Although many implementations recently seem to not be focusing on delay, as is the case with [11].

Thirdly, attempt these experiments with event camera datasets, or other event-based data as it would give more accurate information in the fields where SNNs are meant to shine. Finally, the implementation of other fuse functions to see if they have a better effect on the accuracy.

Finally, we believe that while the images are encoded into spikes works for the purposes of grey-scale images without background it falls short when it comes to encoding more complex datasets such as CIFAR [13] or ImageNet [20]. Hence we believe that if this model is to be used in those datasets an improvement on the [3] image encoding is required.

8. Conclusion

In conclusion, we have explored two ways of implementing fuse residual connection on precise time multi-spiking neural networks. We show that the addition of these connections allows for the training of deeper networks and that these deeper networks have their strengths when compared to shallower networks. Namely, a higher ability to fit more complex data in MLP and a better increase of accuracy in CNNs both when depth was increased and when the residual connections were added. Overall we believe that the results show promise and potential, and prove that residual connections in SNN can be beneficial to the accuracy of the network. Yet there are still some hurdles to overcome and areas to further explore as described in section 7

We have, however, been unable to beat the accuracy of the single-layer CNN although we remain convinced that it is possible if they both had a similar level of hyperparameter optimization. To our disappointment, this process was not feasible due to the long run times of the deeper CNNs combined with our time constraints for this project.

References

[1] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems*, 34(10):1537–1557, 2015. 1

[2] Florian Bacho and Dominique Chu. Exploring trade-offs in spiking neural networks. *Neural Computation*, 35(10):1627–1656, 2023. 2, 3, 4, 5, 6, 9, 10

[3] Florian Bacho and Dominique Chu. Exploring tradeoffs in spiking neural networks, 2023. 2, 3, 4, 5, 6, 8, 10

[4] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*, pages 2921–2926. IEEE, 2017. 2

[5] Wei Fang, Zhaofei Yu, Yanqi Chen, Tiejun Huang, Timothée Masquelier, and Yonghong Tian. Deep residual learning in spiking neural networks. *Advances in Neural Information Processing Systems*, 34:21056–21069, 2021. 2, 3

[6] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002. 3

[7] Stephen Grossberg. Competitive learning: From interactive activation to adaptive resonance. *Cognitive science*, 11(1):23–63, 1987. 3

[8] J. Göltz, L. Kriener, A. Baumbach, S. Billaudelle, O. Breitwieser, B. Cramer, D. Dold, A. F. Kungl, W. Senn, J. Schemmel, K. Meier, and M. A. Petrovici. Fast and energy-efficient neuromorphic deep learning with first-spike times. *Nature Machine Intelligence*, 3(9):823–835, Sept. 2021. 1, 2, 3

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 3, 6

[10] Hagar Hendy and Cory Merkel. Review of spike-based neuromorphic computing for brain-inspired vision: biology, algorithms, and hardware. *Journal of Electronic Imaging*, 31(1):010901–010901, 2022. 1

[11] Yifan Hu, Lei Deng, Yujie Wu, Man Yao, and Guoqi Li. Advancing spiking neural networks toward deep residual learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2024. 10

[12] Yangfan Hu, Huajin Tang, and Gang Pan. Spiking deep residual networks. *IEEE Transactions on Neural Networks and Learning Systems*, 34(8):5200–5205, 2021. 3

[13] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. 2, 3, 9, 10

[14] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010. 2

[15] Chen Li, Runze Chen, Christoforos Moutafis, and Steve Furber. Robustness to noisy synaptic weights in spiking neural networks. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020. 1

[16] Yudong Li, Yunlin Lei, and Xu Yang. Rethinking residual connection in training large-scale spiking neural networks. *arXiv preprint arXiv:2311.05171*, 2023. 2, 3, 6

[17] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 33(12):6999–7019, 2021. 6

[18] Eustace Painkras, Luis A Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R Lester, Andrew D Brown, and Steve B Furber. Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, 48(8):1943–1953, 2013. 1

[19] Michael Pfeiffer and Thomas Pfeil. Deep learning with spiking neurons: Opportunities and challenges. *Frontiers in neuroscience*, 12:409662, 2018. 1, 2

[20] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2015. 3, 10

[21] Sebastian Schmitt, Johann Klähn, Guillaume Bellec, Andreas Grübl, Maurice Guettler, Andreas Hartel, Stephan Hartmann, Dan Husmann, Kai Husmann, Sebastian Jeltsch, et al. Neuromorphic hardware in the loop: Training a deep spiking network on the brainscales wafer-scale system. In *2017 international joint conference on neural networks (IJCNN)*, pages 2227–2234. IEEE, 2017. 1

[22] Biswa Sengupta and Martin B Stemmler. Power consumption during neuronal computation. *Proceedings of the IEEE*, 102(5):738–750, 2014. 2

[23] Yimeng Shan, Xuerui Qiu, Rui jie Zhu, Ruike Li, Meng Wang, and Haicheng Qu. Or residual connection achieving comparable accuracy to add residual connection in deep residual spiking neural networks, 2023. 2, 3, 4, 5

[24] Martino Sorbaro, Qian Liu, Massimo Bortone, and Sadique Sheik. Optimizing the energy consumption of spiking neural networks for neuromorphic applications. *Frontiers in neuroscience*, 14:516916, 2020. 2

[25] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural networks*, 111:47–63, 2019. 1

[26] Jilles Vreeken et al. Spiking neural networks, an introduction. 2003. 1

2 Neural Networks

To understand *Spiking Neural Networks* (SNNs) it is first paramount to understand conventional Neural Networks, also referred to as *Artificial Neural Networks* (ANNs). For the purposes of this thesis, we will only focus on *Fully Connected Layers* and *Convolutional Layers* as they are the most commonly used networks for image recognition and the ones that have been transformed into SNNs in this thesis.

On the most basic level, neural networks are a set of ordered layers in which the outputs of one are used as the inputs of the next. Each of these layers has a set of *weights* that act as variables in a function that takes the input and turns it into the output. An error is calculated from this output using a *Loss Function*, which is used to calculate how to adjust the weights with the aim of obtaining a better result by working backwards and calculating the derivative of the weights of a current layer in respect to the errors of the one after it through a process called *backpropagation*.

2.1. Image Classification Problem and Datasets

Image classification is a common problem for which to use Neural Networks. The objective of Image Classification algorithms is to classify or label images that have not been previously provided to the algorithm, without human intervention.

To do this you run the training loop on a subset of images in the dataset called the training set and then test how well this newly trained network generalizes on unseen data with the test set. These sets in the case of supervised learning consist of an image/data, a label for said data, and possibly some metadata.

In this project, we use three main image classification datasets: MNIST [16], EMNIST [5], and Fashion-MNIST[29]. All of these datasets are labelled datasets and are split into training and testing subsets, as explained in the paper the selection of these datasets was due to our goal of comparing our method with existing SNN methods, such as [2].

MNIST is one of the simplest image classification tasks that exist in the field. It consists of a train set of 60,000 images and a test set of 10,000. All the images are in greyscale and are of handwritten digits 0-9.

EMNIST is similar to MNIST but has been extended with handwritten letters. It has 697,932 training samples and 116,323 test samples that can be one of 62 labels.

Fashion MNIST also has a similar formal but instead of handwriting it attempts to classify articles of clothing into 10 groups. It has 60,000 training samples and 10,000 testing samples.

2.2. Training Process

To achieve a network that produces the desired outcomes the weights of the network need to be set to a correct value employing a training process. This training process is divided into steps over which the entire dataset is iterated over called **epochs**. Due to memory and computation limitations of different machines the data in these epochs is usually split into **batches**.

There are two types of epochs: training epochs, and testing epochs; these go over the training and testing splits of the dataset respectively. During the training epochs for each batch of data a forward and backward pass is performed; in the forward pass the data is processed through the network with the current values of the weights of the layers described in section 2.3 and an output is produced, this output is then compared to the desired output and from the difference, calculated as the **loss** value, backpropagation (described in section 2.4) the weights are updated to something that is believed to produce a better result, with these weight values the process is then repeated with the next batch. This goes on until convergence is reached, which refers to when the loss value of the training set is no longer decreasing.

2.3. Network Layers

In this section, we will cover two of the most popular types of layers used in neural network architectures, the **Multi-Layer Perceptron** and the **Convolutional layers** as well as their main respective components, the **Perceptrons** and the **Kernel** respectively. These layers can be used in combination with one another, and in fact in the case of convolutional architectures they include a feedforward layer just before the output layer that is composed of **Perceptrons**

2.3.1. Fully Connected Layers and Multi-Layer Perceptrons

Fully connected layers are the basis for Multi-Layer Perceptrons, as they consist of a stack of n perceptron layers with m_i perceptrons each. For each of these layers, input of the current layer $i \in \{1, ..., n-2\}$ is the output of the previous layer $i-1$ and the output of i is the input of $i+1$. For layers 0 (the input layer) and layer $n-1$) (the output layer) they receive the input feed into the model and output the model's output respectively. These take the form of an array of numbers with size m_0 for the input and m_{n-1} for the output. For classification problems $m_{n-1} = \#labels$ as one-shot encoding is used. This means that the output of a perceptron in the output layer represents the probability that the input should be assigned the corresponding label.

Perceptron

A perceptron is one of the most basic instances of a neural network. They consist of a very simple function $f(x) = x * w + b$ where x is the input, w is the weight of the perceptron and b is the bias. This simple equation repeated hundreds or thousands of times is what powers the fully connected layers. To create a perceptron layer the w is transformed into an array of weights $w = w_1, w_2, ..., w_n$ where n is the number of weights in the layer and the same thing is done with the biases $b = b_1, b_2, ..., b_n$. This layer is then fed an input x of form $x = x_1, x_2, ..., x_n$ to obtain an output, this output is then cascaded to the next layer in the network, as can be seen in figure 2.1, where the diagram on an MLP with two hidden layers and its internal connections is shown.

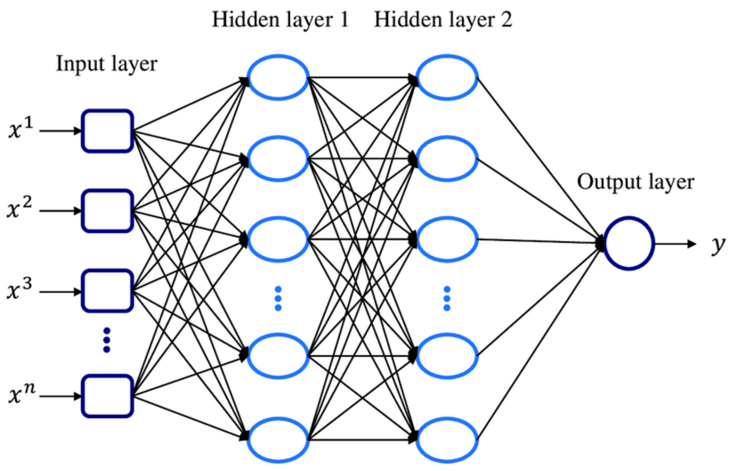


Figure 2.1: Diagram of a two hidden layer fully connected MLP from [24]. In it you can see the inputs represented by the x values and the output represented by the y value, as we are looking at a fully connected MLP each perceptron is connected to every element of the previous and next layer.

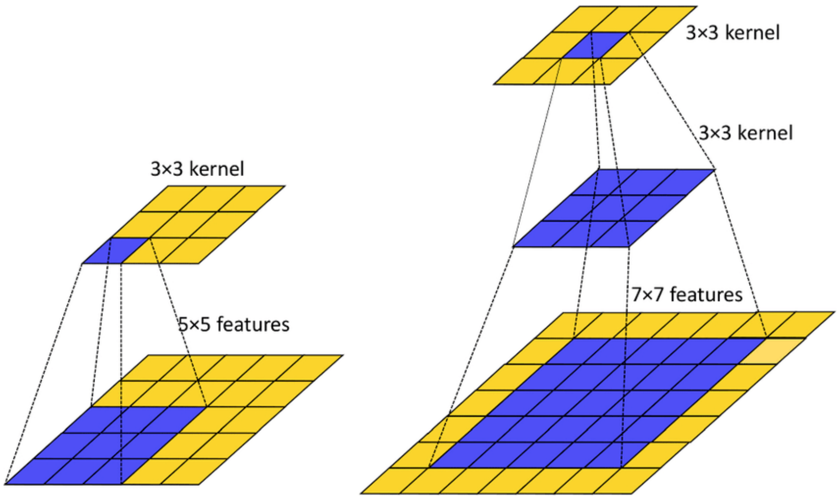


Figure 2.2: Visual representation of how the receptive field increases in the simple case of a 2D convolution with a 3x3 kernel.[14]

2.3.2. Convolutional Layers and Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a staple of image recognition networks as they can detect patterns in images. They focus on transmitting information from groups of inputs (in our case pixels) and therefore unlike MLP take into account neighbouring pixels.

These networks are able to detect “shapes” and “patterns” in the image as long as they fit within the networks **Receptive field size**.

Receptive field size is the size of the region of the input that affects a kernel feature. In the case of a single kernel layer, we can calculate it very easily as the receptive field is the size of the kernel. On the other hand, when layers are stacked the receptive field depends on the size of the staked kernels as well as the stride each kernel uses. This can be calculated recursively using the following formula $r_i = s_{i-1}r_{i-1} + (k_{i-1}s_{i-1})$ with r being the receptive field, s being the stride, and k being the kernel size. A visualization of this with a 3x3 kernel can be seen in figure 2.2.

Another crucial property that CNNs have is that they can find these patterns regardless of where they are located in the image, this is called translation invariance. For example, if the network learns that a solid black area of 5x5 pixels means a higher chance of a certain label then the absolute location of this area in the picture will not harm the network's capabilities to recognise it.

Kernel

Much like the fully connected layer has the *Perceptron* a convolutional layer has a *Kernel*, these are in essence the detectors of a CNN. They take the form of a matrix of size (m, m, c) with m being a hyperparameter and c being the number of channels, much like with the perceptron we shall focus on the single channel example as one can easily extrapolate that to multiple channels. Mathematically speaking a kernel works as a set of perceptrons (usually without the bias component) that take a set of m^2 inputs and multiply each with one of its weights before adding them all together in a single output, then takes a step of stride s before repeating the process, an example of a kernel operation on a padded matrix and its result can seen in 2.4. The application of this operation in an image reduces the size of the image as can be seen in the second image of figure 2.4, due to this, it is common to apply some form of padding to the sides of the image so that it maintains the size after the operation.

By changing the weights of the kernel it can learn to detect different patterns across the image, some examples of this property that are easy to see by humans can be seen in figure 2.3. In actual CNNs the patterns tend to be more complex and harder for humans to interpret as can be seen in [31]. Whatever the case it allows the network to gather information on pixel values in relation to the values of the pixels around them.

In conclusion, for the goals of this thesis, it is more important to understand that the following key ideas.

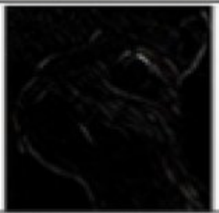
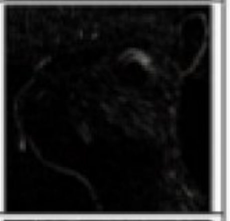
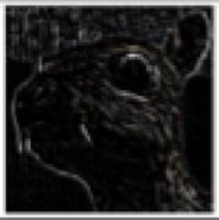
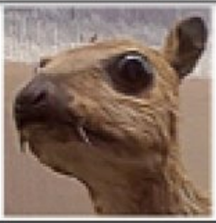
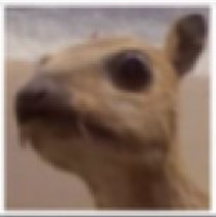
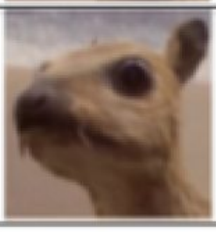
Operation	Filter	Feature map
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 1 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Guassian blur	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 1 & 1 \end{bmatrix}$	

Figure 2.3: Visual representation of the effects applying a kernel can have on an image [6]. As shown by using different values in the kernel, one can detect regions of the image, such as edges. With larger kernels one can detect larger shapes, hence the importance of *receptive fields*.

Kernels look for "shapes" and "patterns" in the data by looking not only at the value of a pixel but also at the values of the pixel around it. The complexity and size of what these kernels are able to detect increases with the depth of the CNN, this is due to the depth causing an increase in the **Receptive field size** of the network. Unlike MLPs, this extra depth does not cause overfitting as they are learning properties that are general and translation invariant. Overfitting is discussed more in-depth in section 2.6.

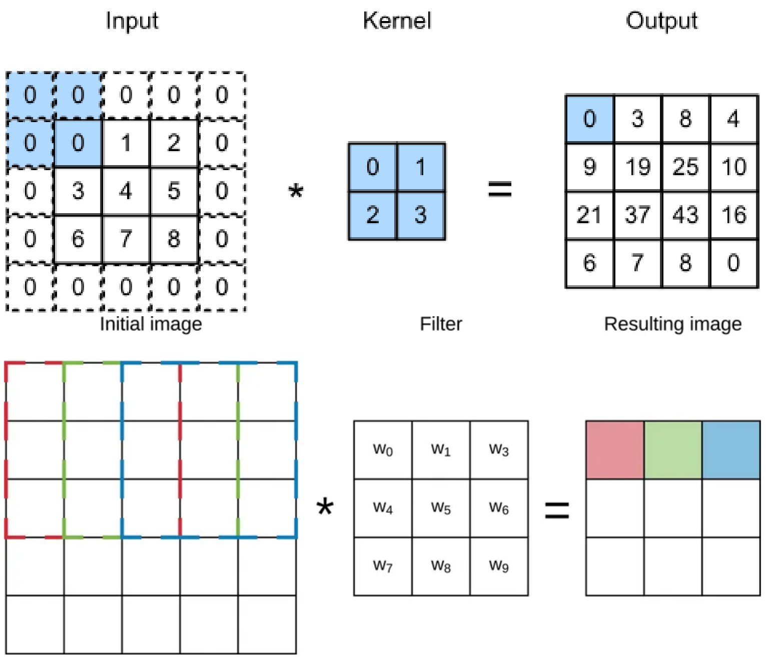


Figure 2.4: An example and visualization of how convolution operations function in relation to the addition of padding. As you can note the first image has padding in the input and the second image does not, hence the output of the first example remains the same while the output of the second is reduced in size [19, 21].

Pooling layers

Pooling layers act in a similar way to convolutional layers and are a common part of the CNN architecture, their goal is to help pool together the information from the previous layers and expand the *receptive field*. They function by performing an operation with the values inside the *kernel* area that outputs a single output. In theory, there could be a nearly infinite number of pooling functions but only three are used commonly. These are max pooling, which takes the max value from the pooled area; min pooling, which takes the minimum value from inside the pooled area; and finally average pooling, taking the average of the pooled values. An example of the operation with max and average pooling can be seen in figure 2.5.

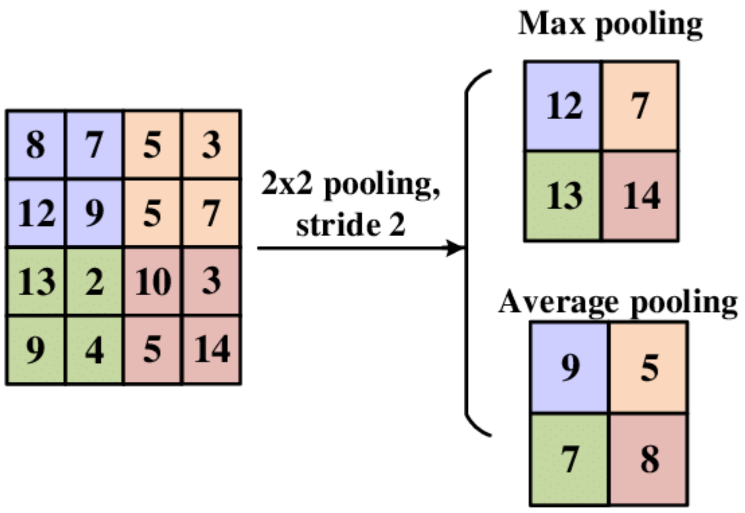


Figure 2.5: Example of a 2x2 pooling layer with a stride of 2 being applied to a 4x4. In max pooling the highest of the four values while the average pooling takes the average of the four values [30]

2.4. backpropagation

As mentioned earlier, backpropagation is the process by which neural networks learn from their mistakes and adjust their weights to values that will produce a better result in the next iteration. As the name indicates this process begins from the loss value or error and then works backwards through all the layers in the network.

2.4.1. Loss Calculation

The loss function is the function that takes the predicted values and compares them to the expected values giving a higher loss the further the values are from the expected values. This loss value is the starting point of the backpropagation and hence is what the network is aiming to minimize when updating its weights. The most commonly used loss function for classification function is the cross-entropy loss function, shown below.

$$L(p,t) = -\sum_{x \in classes} (p(x) * log(q(x)))$$

Where $p(x)$ is the one-shot probability of label x and $q(x)$ is the model's predicted probability distribution. In this paper, we will use a different loss function that is better suited for our spike-based output and will be explained in 3.

2.4.2. Gradient Descent

Gradient descent is the optimization algorithm used with the goal of finding the local minimum of a derivable function. In the case of deep learning, it is used to find the combination of weight values that produce the lowest loss in the loss function.

The goal is to reach a satisfactory or ideally the global minimum of the loss function by taking small approximation steps as seen in figure 2.7 and 2.6. This is done by gradually updating the weights of the network based on a hyperparameter called "learning rate" λ using the following formula, with $\triangle w$ being the gradient for the variable in question that would cause the greatest reduction in the value of the loss function (this $\triangle w$ gradient can be obtained through backpropagation calculations covered in the following section 2.4.3).

$$w_{\rightarrow} w - \lambda \triangle w$$

This constant determines how big each step should be and it is one of the most crucial parameters for the performance of the functions. This is because as mentioned above gradient descent locates the *local* optimum of the loss function, therefore as shown in figure 2.6 a learning rate that is too high will lead the

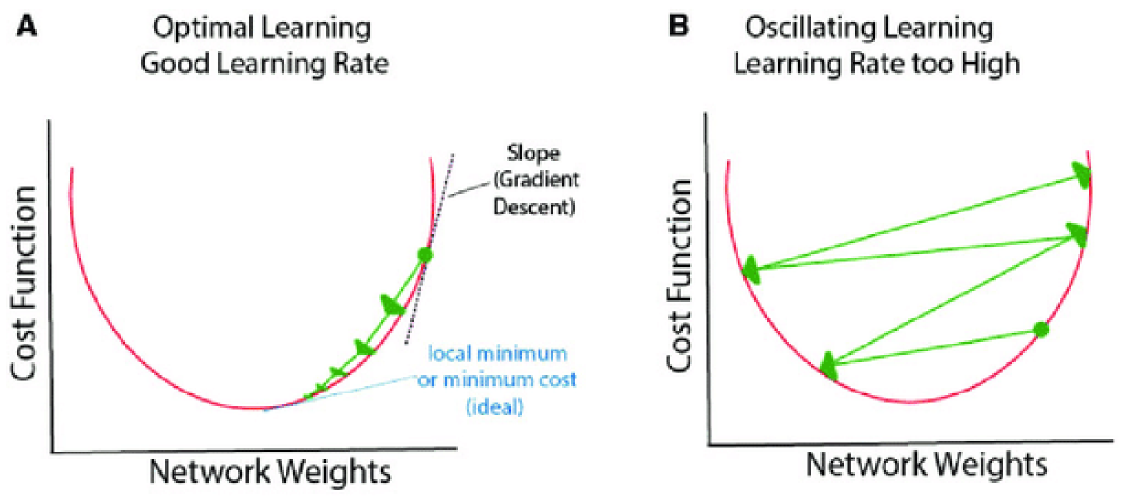


Figure 2.6: Comparison between two different values of learning rate in a very simple loss function. Case A shows a proper learning rate and how it leads to a local minima of this simple function. Case B shows a learning rate that is too high and therefore is not able to settle down on the local minimum.

skipping over the optimal, while a learning rate that is too low will lead to finding a very local minimum and be unable to take a large enough *step* to leave said minimum. Choosing the proper learning rate involves both domain knowledge and trial and error.

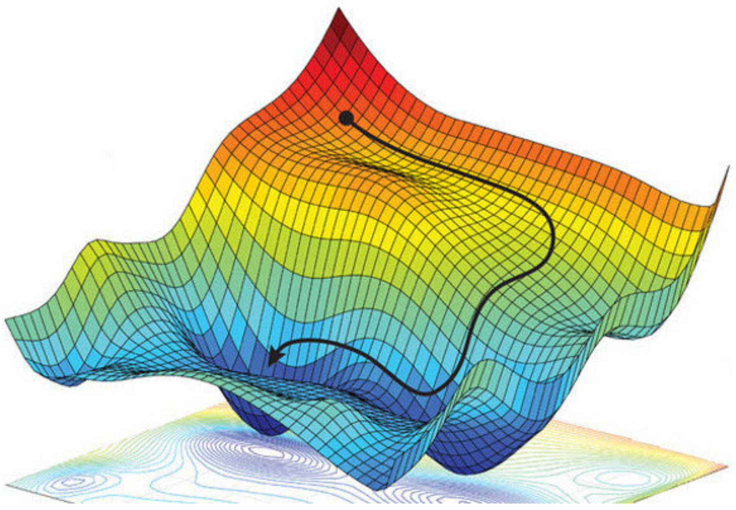


Figure 2.7: Illustration of gradient descent on a two-dimensional solution space [1]. As shown in this figure, the intuition behind it is to take steps towards the "direction" that the loss decreases the most and eventually you will find a local optimum. How good this local optimum is in relation to the global optimum is usually determined by the learning rate value as well as the initial weight distribution/values.

2.4.3. backpropagation Calculations, The Chain Rule

In this section, we will go over the calculations and mathematical principles that are behind the process of backpropagation. The chain rule is a calculus formulation that states $f'(g(x)) = f'(g(x))g'(x)$. Note that, for simplicity's sake, in this section when discussing the math we will use a chain of perceptrons and a single univariate input. If we take a single perceptron with a nonlinearity $\sigma(z)$. So in the case of a single input and label pair (x, t) with regularization parameter to the loss $R = \frac{1}{2}w^2$ and hyperparameter λ so the equations for this system become:

$$z = xw + b \rightarrow y = \sigma(z) \rightarrow \mathcal{L} = \frac{1}{2}(y-t)^2$$

$$R = \frac{1}{2}w^2 \rightarrow \mathcal{L}_{reg} = \mathcal{L} + \lambda R$$

$$\begin{tikzcd}
x \arrow[dr] & & & t \arrow[d] & \\
b \arrow[r] & z \arrow[r] & y \arrow[r] & \mathcal{L} \arrow[r] & \mathcal{L}_{\text{reg}} \\
w \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

Figure 2.8: Equations for the described equation system and the flow diagram. In this case, we want to change the bias b and the weight w with the goal of minimizing the loss with the regularization term applied $\mathcal{L}_{reg}$

Given this forward pass, we will now explore how to determine the adjustments required to the values of the learned variables W and b with the objective of minimizing the regularized loss $\mathcal{L}_{reg}$. How these variables all relate to one another can be seen in the set of equations in figure 2.8 and their accompanying information flow diagram. What we are most interested in knowing is how a change in a variable, for example, the weight w and b, affects the loss so that we can increase or decrease it appropriately; formally we aim to find:

$$\frac{\partial \mathcal{L}_{reg}}{\partial W} = \bar{w}$$

$$\frac{\partial \mathcal{L}_{reg}}{\partial b} = \bar{b}$$

Please note the notation $\bar{x}$ that we will be using from now on as a way to refer to the derivative of the loss function with respect to a variable x.

To do this we use the chain rule starting from $\bar{\mathcal{L}_{reg}} = 1$ and work our way backwards.

$$\bar{\mathcal{L}_{reg}} = 1$$

$$\bar{R} = \bar{\mathcal{L}_{reg}} \frac{dL_{reg}}{dR} = \bar{\mathcal{L}_{reg}} \lambda$$

$$\bar{\mathcal{L}} = \bar{\mathcal{L}_{reg}} \frac{dL_{reg}}{d\mathcal{L}} = \bar{\mathcal{L}_{reg}}$$

$$\bar{y} = \bar{\mathcal{L}} \frac{d\mathcal{L}}{dy} = \bar{\mathcal{L}}(y-t)$$

$$\bar{z} = \bar{y} \frac{dy}{dz} = \bar{y}\sigma'(z)$$

$$\bar{w} = \bar{z} \frac{dz}{dw} + \bar{R} \frac{dR}{dw} = \bar{z}x + \bar{R}w$$

$$\bar{b} = \bar{z} \frac{dz}{db} = \bar{z}$$

Having obtained the value of $\bar{w}$, we can change w appropriately, as we know how a change in the value of w will affect the value of the loss. If we update the weight by a fraction λ of $\bar{w}$ as covered in the *gradient descent* section 2.4.2 this should cause the next forward pass calculation to have a different and more accurate result. This process is repeated for each batch in every epoch, the loss function is complex and hard to predict, for this reason, small steps are taken each time depending on the gradient and the learning rate.

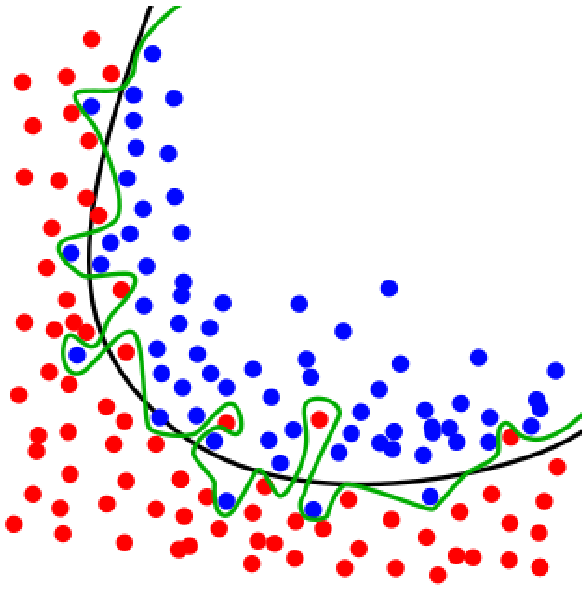


Figure 2.9: A figure showing how overfitting can look in a 2D space. The green line shows overfitting while the black line is more accurate to the general distribution of the data

2.5. Hyperparamater Optimization

A crucial factor influencing the performance of neural networks is the selection and tuning of hyperparameters. Hyperparameters are values external to the model itself, and they control the learning process, determining how the model is trained and how well it will perform. These parameters can significantly affect the model's ability to learn patterns in the data and generalize to new, unseen data. One illustrative example of the importance of hyperparameters is the learning rate, which was discussed in detail in section 2.4.2. The learning rate is a key hyperparameter in the gradient descent algorithm that governs the step size at each iteration while moving towards a minimum of the loss function. Choosing an appropriate learning rate is critical; a value too large can lead to overshooting the optimal solution, while a value too small can result in a slow convergence process, or getting stuck in a local minimum.

However, the learning rate is just one of many hyperparameters that need carefully selected. Other hyperparameters that tend to have a large impact on the performance include but are not limited to weight initialization methods, batch size, number of layers, number of neurons per layer, activation functions, and dropout rates. Each of these parameters plays a role in the network's ability to learn from data and requires fine-tuning in order to achieve optimal performance, and they usually have different values depending on the dataset and architecture.

Selecting the optimal values for these hyperparameters is a complex task. The process typically involves exploring the hyperparameter space, which can be vast and high-dimensional. This exploration is often conducted through a combination of intuition, trial and error, and domain knowledge. Researchers and practitioners may rely on experience and theoretical insights as a means of narrowing down the range of potential values. In my case, I used the experience of my supervisors who advised me that the learning rate scheduler and the weight distributions were the factors that were most likely contributing to a loss of accuracy.

2.6. Overfitting

Overfitting is the phenomenon in which a model adapts to the training data to such a degree that it hurts the performance of the model on unseen data (such as the test set). The main cause for this, especially as it concerns this thesis is the extra degrees of freedom that the network has in combination with the extra training epochs.

As an example let's take a look at Multi-Layer Perceptrons are linear classifiers that draw a line to split the data with a function, hence the more layers and the bigger those layers are the more variables the function has and the more flexible it can be in order to fit to the data. An example of this can be seen in figure 2.9 where even though the green line will have a higher accuracy in the shown data it would perform worse than the black line in unseen data given the distribution of the blue and red dots.

In this thesis, we employ both MLPs and CNNs, making it crucial to explore why MLPs are more prone to overfitting, while CNNs exhibit a lower tendency for this issue. Having established a foundational understanding of these architectures, we can now delve into the factors that contribute to the higher likelihood of overfitting in MLPs compared to CNNs.

MLPs focus only on the value of the pixel and do not take into account the location of said pixel in the image or the values of the neighbouring pixels. What we mean by that is that an MLP would achieve the same result if you were to choose a random way of scrambling the pixel locations in the image, as long as you always scramble it the same way. This means that if given enough weights and layers an MLP could theoretically perfectly split the training data groups perfectly adjusted to the training data of the classification problem. This fit however will probably not reflect the reality of the general data. In figure 2.9 you can see a clear example of overfitting on the training data of a two-class classification problem.

On the other hand, due to the use of kernels and pooling layers, CNNs do not have this problem, as they learn to detect certain local pixel patterns, as described in section 2.3.2. These patterns are a lot more likely to be repeated over several images, and since regardless of where the pattern is located the network will be able to identify it, the network learns general information of what it is looking for and not specific properties of the training set, hence avoiding overfitting.

3 Spiking Neural Networks

With a solid understanding of Artificial Neural Networks, we can now transition to exploring Spiking Neural Networks. As the name might suggest the SNNs are a type of neural network that instead of working with numbers (as described in section 2) uses spikes over time in order to transmit the information. This is more accurate to biological brain functioning and can be faster and more energy efficient [9, 27]. On the other hand, they also come with some downsides, mainly because the non-differential transfer function they use in the forward pass does not allow for traditional propagation methods.

3.1. Neuromorphic Computing

When talking about SNNs it is important to understand that one of the key advantages of it is that they can efficiently run on neuromorphic computational devices. These devices have specialized hardware in which one can embed an SNN and train it faster and with less energy consumption than on a traditional machine. However, in this thesis, we did not implement it on a neuromorphic machine due to lack of resources.

3.2. Spike Encoding

As one might assume, the inputs and outputs of these networks consist of spikes. Since most data is not naturally made out of spikes over time the question of spike encoding then is "*How do we transform none spike information into spikes and vice versa?*".

There are two main ways of doing the none-spike to spike data transformation in the case of conventional image data: temporal or latency encoding and rate encoding. In other words, one stores information based on the time between spikes while the other stores information in the amount of times received in a period of time. Let's explore how each of these methods would go about encoding the data of a single channel pixel such as in the case of the input from the MNIST benchmark dataset [17]:

- The first encodes the intensity of the input pixel by producing a single spike per pixel value and making the delay of that spike to be inversely (or directly) proportional to the intensity of the original value.
- While the second encodes this information by generating multiple spikes per value with the number of spikes being proportional to the pixel intensity.

In the case where we want to interpret the output spikes of an SNN network, there are two main options for image classification labelling: Time To First Spike (TTFS) and Spike Count (SC). Time To First Spike (TTFS) the predictions of the network are the ones corresponding to the first output neuron that produced a spike. In Spike Count (SC) the selected label is the one that is the most active.

While in the past both approaches have been used, in [2] rate encoding is used. Similarly, in the approach covered in this thesis, we chose to use a spike count, also referred to as rate encoding. This is done in order to take better advantage of our implementation's multi-spiking neuron capability.

3.3. Forward Pass

In this section we will cover everything that one needs to know in order to understand a forward pass of a SNN and the different design decisions that can be made; special focus will be given to the properties chosen for our approach. We will first go over a general overview of a neuron in an SNN and all the different types of neurons that exist while explaining what neuron is used in our implementation, we will then follow that up with some more general intuition on how these neurons interact with each other when set up in a SNN.

3.3.1. Neurons

Neurons are the key to spiking neural networks, they perform a similar function than perceptrons do in ANNs. In order to understand the spiking behaviour of a neuron it is important to understand the concepts of *Spikes* and *Membrane potentials*.

A *Spikes* is the lowest unit of information in an SNN, just like output and input values are in the context of an ANN. However, unlike numerical values in ANNs, they do not transmit the information based on "what they are", as all spikes are the same in terms of intensity, but in terms of "When they happen". In other words, information is communicated through the temporal channel.

The *Membrane potential* of a neuron is the internal state of said neuron. This value goes up by a certain amount when the neuron receives a spike, decays constantly over time, and when it reaches a certain threshold v_{th} or v_{th} the neuron outputs a spike to the neurons it is connected to in the next layer and resets its membrane potential. This can be seen in the visualization 3.1.

There are four main properties to keep in mind when talking about a neuron and its membrane potential are the following:

- The first is whether the system uses exact time or discrete time. In a lot of implementations of SNNs, discrete-time is used due to the fact that it simplifies a lot of the calculations. The way this is done is by splitting the time into segments and instead of having a precise and exact time for each spike they store each spike in a time step. The use of discrete time does come with its own set of downsides, mainly that by limiting the values the spike times can take you are also limiting how precise the information the network can relay, for this reason in our implementation we have opted to use precise time.
- The second is if the neuron uses a leaky membrane potential. A membrane potential is considered leaky if it decreases over time when it does not receive spikes, this means that in the case of a leaky potential in order for the neuron to output a spike it has to receive the spikes close together, while in the case of a completely non-leaky neuron, the time between spikes does not matter. For the implementation described in this thesis, we have chosen to use leaky neurons as they allow for more flexibility in the network.
- The third is if the neuron is multi-spiking, if a neuron can spike more than once then this affects the computations behind membrane potential, as now it is no longer affected by the spikes that input the neuron (called pre-synaptic spikes), but also by the spikes that have outputted the neuron (post-synaptic spikes). If a neuron can output more than one spike then the neuron can communicate more complex information, therefore we have chosen (thanks to the advances shown in [2]) to go for a network of multi-spiking neurons.
- The final property to keep in mind is whether the neuron has a hard or soft reset of the membrane potential. This only applies in the case of multi-spiking neurons and it refers to how the membrane potential is brought down from its activation threshold v_{th}after the neuron has outputted a spike. In the case of a hard reset the membrane potential is brought back to its initial value v_{rest}, while in the case of a soft reset an amount $\eta(t)$ is subtracted from the current potential $v \to v - \eta(t)$ when $v > v_{th}$ this can lead to values above or below v_{rest} and by tuning the function η a lot more flexibility is given for tuning the network. For this reason, we chose a soft reset in our approach.

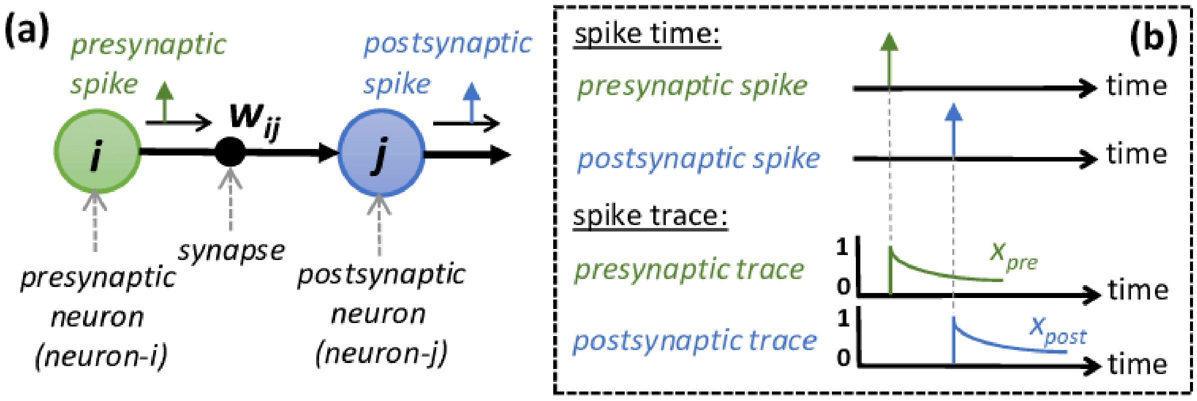


Figure 3.2: Representation of the relation between the weight, and the neurons it connects. The weight determines how much the spike increases the membrane potential. This is repeated for every pair of neurons that have a connection to each other [22]

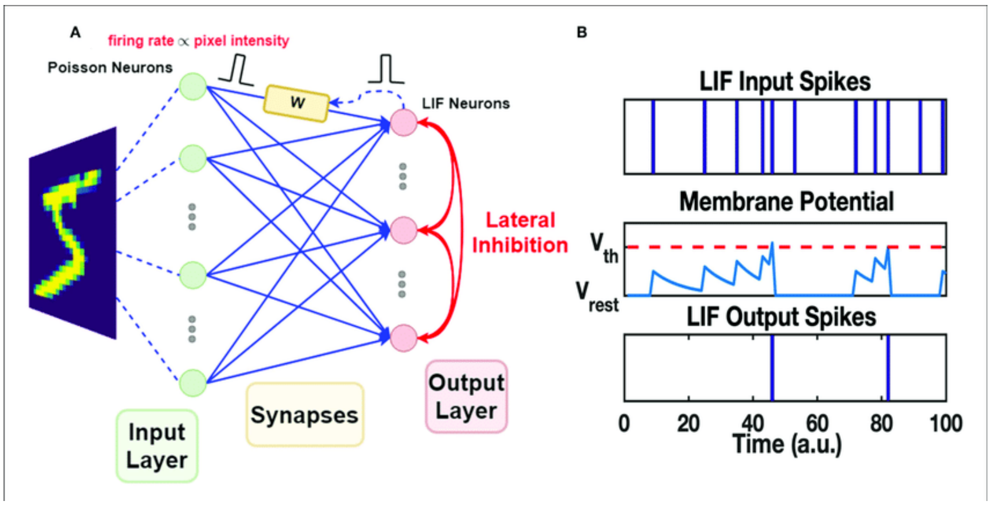


Figure 3.1: Visual representation of the activation function of a neuron in s SNN taken from [10]. This figure uses a leaky neuron with a hard reset, while my approach uses a leaky neuron with a soft reset, meaning that the membrane potential would not necessarily go to v_{rest}, but the current membrane potential value minus a reset amount.

3.3.2. Intuition Behind Spike Cascading

With an understanding of neurons, their various types, and their interrelationships, we can now explore how neurons interact during the forward pass of an SNN.

As mentioned before a neuron outputs a spike when its membrane potential reaches the activation threshold, in our case since we are using leaky neurons this means that the neuron has received enough spikes in a short enough amount of time from neurons from the layers above it. How many of these spikes are needed to produce an output spike depends on the neurons the spikes are coming from, as the membrane potential increases in direct relation to the weight $w_{i,j}^{(l)}$ as seen in figure 3.2 (it is interesting to note that these weights can also be negative). Keeping this in mind the idea is that these weights, after a number of epochs will produce a neural path based on strong connections in neuron spikes through the network [28]; just like neural pathways are created in the human brain [8].

3.4. Dead Neuron Problem

Before we move on to how to train SNN this is an adequate point to cover a recurring problem known as the dead neuron problem. This problem refers to a situation where the weights of a neuron are not updated because of issues with the spike S and weight w derivation $\frac{\partial S}{\partial w}$, this mainly but mainly stems from the fact that the spikes are discontinuous. Formally there are 3 cases when taking the spike derivation:

1. The membrane potential is below the threshold. $\frac{\partial S}{\partial w} = 0$
2. The membrane potential is above the threshold $\frac{\partial S}{\partial w} = 0$
3. The membrane potential is equal to the threshold $\frac{\partial S}{\partial w} = \infty$

In cases one and two adding them to the chain rule causes $\frac{\partial \mathcal{L}}{\partial w} = 0$ causing there to be no change to the weight, and therefore no learning to it or any of the previous neurons. In the improbable case three adding ∞ to the chain rule causes it to swap out any meaningful gradient. This is visualized in figure 3.3 a.

In our implementations this is tackled by using the spike times, so $\frac{\partial t_s}{\partial w}$ instead of the above described $\frac{\partial S}{\partial w}$. This solves it because while the spikes are not continuous time is. There are however other ways around this problem and they all have advantages and disadvantages to be aware of.

3.5. Training Spiking Neural Networks

The main ways of training an SNN network as covered by [7] are the following:

- Shadow Training: This method consists of training an ANN and then once it is trained transforming it into an SNN by means of encoding the ANN weights as SNN weights.
- Surrogate Gradients: In surrogate gradients, you use a surrogate function to replace the spikes, as in the case of [18].
- Spike Time: spike time is a method of training SNNs that use the actual spike times, as in the case of [4].

Let's cover them in a bit more detail as although in this project we only use *spike time backpropagation* it is important to understand the context it is being proposed in and the other options available in order to see the benefits of our selected method.

3.5.1. Shadow Training

Shadow training is a method of training SNN that avoids the problem of how to train an SNN by sidestepping the need to train an SNN. The way they do this is by instead training an ANN and then once this network is trained they convert it and its weights into a SNN. This method has produced good results in the field of image classification, however as one can intuit their training process is quite inefficient and raises questions about redundancy.

Other than that they also show issues that make them not as promising as other training methods:

- The first is that since the training is done on an ANN the datasets used do not usually involve the use of the temporal dynamics of SNN and relegate them to ordinary benchmark datasets designed for ANN, This however is a problem many approaches face and is hard to solve due to the lack of datasets designed to highlight the aforementioned property.
- The second downside is that since ANNs use highly precise values converting them into SNN requires, in the case of discrete-time SNNs, a large amount of simulation time steps. This can undermine the power and latency advantages that SNNs provide.
- Finally and more crucially the training method attempts to make the SNN an approximation of an ANN, and therefore the SNN will not reach or surpass the ANN. Some approaches such as [23] use a hybrid approach where the ANN is used to set an initial value for the SNN and then perform backpropagation on the SNN, this however still requires one of the other covered SNN training methods.

3.5.2. Spike backpropagation and Surrogate Gradients

Spike backpropagation and Surrogate Gradients are ways to attempt backpropagation while circumventing the dead neuron problem with the use an approximate gradient (as is the case in [18]) to replace $\frac{\partial S}{\partial w}$ with an approximation $\frac{\partial \tilde{S}}{\partial w}$. In the case of [18] for example they use a Sigmund function $\Theta(x)$ for $\tilde{\Theta}(x)$ in this case the notation used is $\frac{\partial S}{\partial w} = \Theta'(x) \rightarrow \frac{\partial \tilde{S}}{\partial w} = \tilde{\Theta}'(x)$. A visual representation of these surrogate gradients can be seen in figure 3.4.

$$\Theta(x) = \begin{cases} 1 \text{ if } x >= 0 \\ 0 \text{ else} \end{cases} \tag{3.1}$$

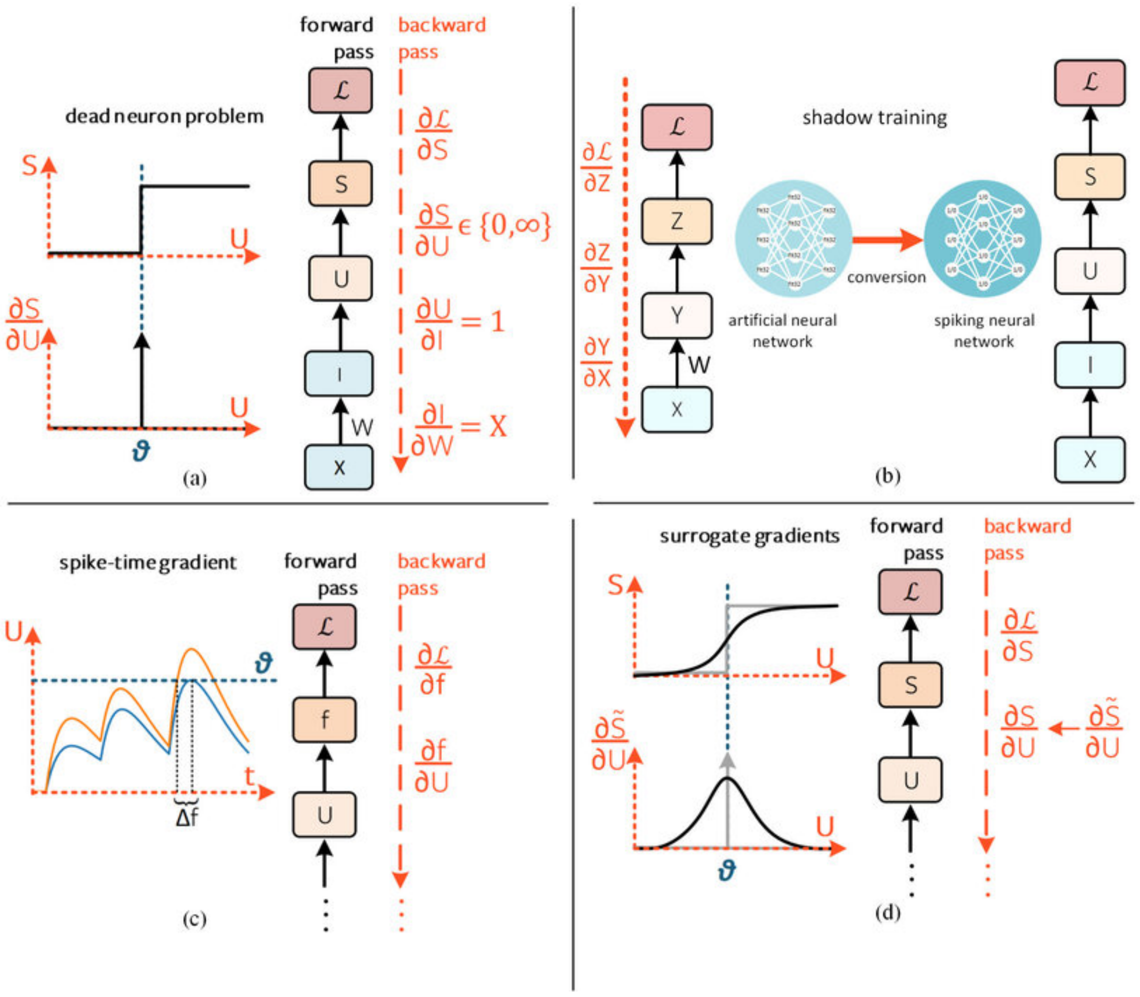

dead neuron problem
forward pass
backward pass
$\mathcal{L}$
S
U
I
X
W
$\frac{\partial \mathcal{L}}{\partial S}$
$\frac{\partial S}{\partial U} \in \{0,\infty\}$
$\frac{\partial U}{\partial I} = 1$
$\frac{\partial I}{\partial W} = X$
ϑ
(a)
shadow training
artificial neural network
conversion
spiking neural network
$\frac{\partial \mathcal{L}}{\partial Z}$
$\frac{\partial Z}{\partial Y}$
$\frac{\partial Y}{\partial X}$
Z
Y
(b)
spike-time gradient
$\frac{\partial \mathcal{L}}{\partial f}$
$\frac{\partial f}{\partial U}$
f
t
Δf
(c)
surrogate gradients
$\frac{\partial \tilde{S}}{\partial U}$
$\frac{\partial S}{\partial U} \leftarrow \frac{\partial \tilde{S}}{\partial U}$
(d)

Figure 3.3: Visual representation of the dead neuron problem and the three main solutions to it [7]. " (a) The dead neuron problem: the analytical solution of S/ U 0, ∞ results in a gradient that does not enable learning. (b) Shadow training: a non-spiking network is first trained and subsequently converted into an SNN. (c) Spike-time gradient: the gradient of spike time f is taken instead of the gradient of the spike generation mechanism, which is a continuous function as long as a spike necessarily occurs [3]. (d) Surrogate gradients: the spike generation function is approximated to a continuous function during the backward pass [11]. The left arrow (←) indicates function substitution. This is the most broadly adopted solution to the dead neuron problem."- as explained by [7]

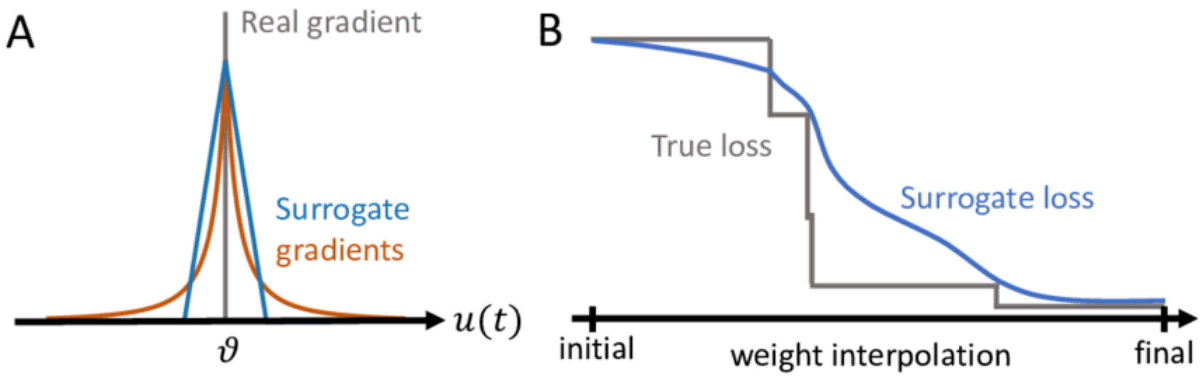


Figure 3.4: Comparison of the surrogate gradient and the real gradient. As you can see the loss of the gradient helps smooth out the loss function, and hence make the. [15]

$$\tilde{\Theta}(x) = \frac{1}{1+e^{-4x}} \rightarrow \tilde{\Theta}'(x) = 4\tilde{\Theta}(x) * (1 - \tilde{\Theta}(x)) \tag{3.2}$$

This allows them to perform propagation in the same way as ANNs do, this is visualized in 3.3 section d. However spikes still need to be outputted in order to update the weights, a downside our implementation also shares, as explained in section 4.3.2 and which we aim to correct with the residual connections.

3.5.3. Spike Time backpropagation

The final method of training SNNs is the use of Spike time backpropagation, this is the method used on our approach and therefore we will not go in-depth in this section as the rest of this chapter covers this training method and the equations used in our implementation. As mentioned in the dead neuron problem section 3.4 this method performs the backpropagation not in the spikes but in the spike times.

The intuition behind this method is that a change $\triangle w$ in a weight will cause a change in the membrane potential by $\triangle U$ and that will result in a change of the spike time t of $\triangle t$. When comparing to Spike backpropagation the main change is that we change $\frac{\partial S}{\partial w}$ of $\frac{\partial t}{\partial w}$. This does however mean that *neurons must emit a spike in order for their gradient to be calculated*, this problem that we call *silent neurons problem* is one of the main problems we intend to tackle in our implementation by the use of residual connection.

Other than that, using spike times to calculate the gradient also has another difficulty. That is the relation between the weight values and the spike times. Due to the nature of the neurons a small change in weights can cause a large change in the spike times as can be seen in 3.5 or it can even cause the neuron to stop outputting spikes altogether.

This method has another significant downside that this thesis aims to solve, and that is the fact that since the backpropagation is done on spike times; if there are no spikes at the output backpropagation is not possible. This is a problem we refer to as the banishing spike problem and I will cover it more in-depth and how it relates to deeper networks in section 4.3.2.

3.6. Our Implementation

In our implementation we use neurons multi-spiking precise time neurons capable of precise spike time backpropagation developed by [2]. In this section, we shall go over both the intuition and the math behind these neurons. The additional math related to residual connections and our relation to it will be explored in section 4.3.

3.6.1. Our Neurons

With that said let's start taking a look at the inner workings of our neurons by looking a the membrane potential equation. This equation determines when our spikes will occur and is composed of two main components. One that reacts to pre-synaptic spikes, and one that reacts to post-synaptic spikes. The reason for this is that since it can spike multiple times the spike times of future spikes do not only depend on the

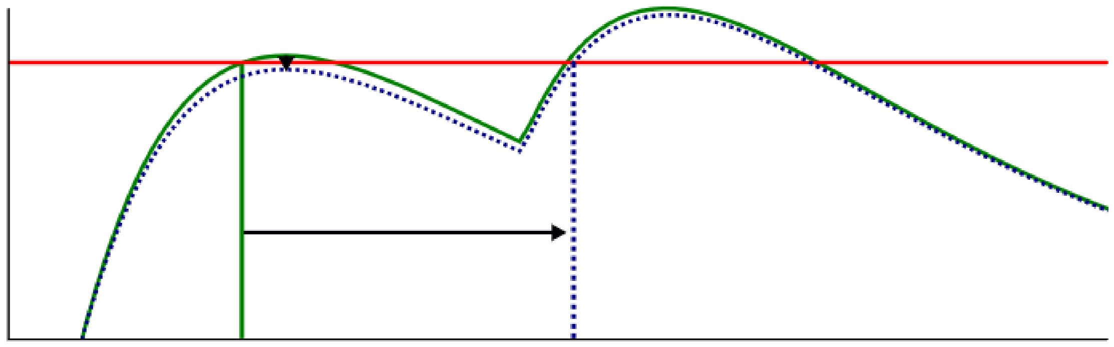


Figure 3.5: Visual representation of how a small change in the weight can have a large effect on the spike time. This is one of the causes that make backpropagation in spiking neural networks more challenging than in traditional Artificial Neural Networks.

spike the neuron receives but also on the spikes the neuron has emitted.

$$u^{(l,j)}(t) = \sum_{i=1}^{N^{(l-1)}} w_{i,j}^{l} \sum_{z=1}^{n^{(l-1,i)}} \epsilon(t - t_z^{(l-1,i)}) - \sum_{z=1}^{n^{(l,j)}} \eta(t - t_z^{(l,j)}) \tag{3.3}$$

with $\epsilon(t)$ representing the response of the neuron to a pre-synaptic spike and $\eta(t)$ being the neuron's reset response to emitting a spike. In the case of the neurons used for this thesis these equations are the following:

$$\epsilon(t) = \Theta(t)\frac{\tau\tau_s}{\tau - \tau_s}\left[exp\left(\frac{-t}{\tau}\right) - exp\left(\frac{-t}{\tau_s}\right)\right] \tag{3.4}$$

$$\eta(t) = \Theta(t)v_{th}exp\left(\frac{-t}{\tau}\right) \tag{3.5}$$

$$\Theta(x) := \begin{cases} 1 \text{ if } x > 0 \\ \\ 0 \text{ else} \end{cases} \tag{3.6}$$

Where τ is the membrane time constant τ_s is the synaptic time constant and v_{th} is the activation threshold. Let's take these equations one at a time. The first thing to note in both equations is that all spikes no matter the time set the $\Theta(t) = 1$. Another detail of importance is that since we use a leaky neuron the input of both functions is not the spike time but the difference between the current time and the time of the last spike. We also see that a more recent spike will increase the membrane potential more due to the $\exp\left(\frac{-t}{\tau}'\right)$ and $\exp\left(\frac{-t}{\tau_s}\right)$ components in $\tau(x)$.

3.6.2. Spike Time Calculations In The Forward pass

Spike time for a spike in a multi-spiking neuron depends on two things, the pre and post-synaptic spikes of the neuron, those referring to the spikes the neuron has received up to this point (pre-synaptic) and the spikes the neuron has outputted up to this point (post-synaptic). A way to understand this is to picture the neuron's membrane potential as a water bucket with a hole at the bottom and a pump that empties it once a certain level of water is reached. In this analogy the leakiness of the neuron would be the whole at the bottom of the bucket, a higher leakiness a larger whole and the more water (spikes) in a certain period of time you need to input to fill it to the threshold. This threshold is the v_{th} is the level at which the pump is activated, throwing water into the next buckets (the neuron outputs a spike), whether the bucket is left at the original water line or not is the hard or soft rest.

This is the reason why in the closed form solution for the spike time calculations seen in equation 3.9 the spike time $t_k^{(l,j)}$ depends on both the spike times of the previous layer $t_k^{(l-1,i)} \to \forall i \in N(l-1)$ as well as the previous spike times of itself $t_n^{(l,j)} \to n < k$.

3.6.3. Spike Time Loss Function

Similar to an ANN the loss function should take two inputs, what we obtained from the network and what we want to obtain (taken from the labels), and calculate a distance between them. However, in SNNs most traditional loss functions such as Mean Squared Error (MSE) do not apply. Therefore we use the following loss function:

$$\mathcal{L} = \frac{1}{2} \sum_{j=1}^{n^{(o)}} \left(y_j - n^{(o,j)} \right)^2 \tag{3.7}$$

That compares the number of output spikes of a given neuron in the output layer $n^{(o,j)}$ with the number of outputs it should be producing y_j. The value of y_j changes between two values in our implementation depending on whether the neuron corresponds to the correct label or not, these two values are set as parameters.

Another option for the loss function would be to use Time to first spike encoding and then perform the cross-entropy loss function on the selected label.

3.6.4. Closed Form Solution

The closed-form solution of the spike timing equations is what allows backpropagation to work without needing to use surrogate functions as it makes the spike time equations derivable. This is done in [9] and then built upon by [2] to allow for multiple spikes per neuron. they do this by constraining the value of the membrane time constant τ to be two times the value of the synaptic time constant ($\tau = 2\tau_s$).

In general, the Spike Response Model (SRM) mapping of a LIF with this constraint can be written as:

$$0 = -a_k^{(l,j)} \exp\left(\frac{-t_k^{(l,j)}}{\tau} \right)^2 + b_k^{(l,j)} \exp\left(\frac{-t_k^{(l,j)}}{\tau} \right) - c_k^{(l,k)} \tag{3.8}$$

and therefore this polynomial equitation can be solved for $t_k^{(l,j)}$, representing the spike time of the k_{th} spike in neuron j of layer l, by using:

$$t_k^{(l,j)} = \tau \ln \left[\frac{s a_k^{(l,j)}}{b_k^{(l,j)} + x_k^{(l,j)}} \right] \textbf{ with } x_k^{(l,j)} = \sqrt{(b_k^{(l,j)})^2 - 4 a_k^{(l,j)} c_k^{(l,j)}} \tag{3.9}$$

With $a_k^{(l,j)}, b_k^{(l,j)}, c_k^{(l,j)}$ being terms that change depending on the implementation; in this case:

$$a_k^{(l,j)} := \sum_{i=1}^{N^{(l-1)}} w_{i,j}^{(l)} \sum_{z} = 1^{n^{n(l-1,i)}} \Theta\left(t_k^{(l,j)} - t_z^{(l-1,i)} \right) \exp\left(\frac{t_z^{(l-1,i)}}{\tau_s} \right) \tag{3.10}$$

$$b_k^{(l,j)} := \sum_{i=1}^{N^{(l-1)}} w_{i,j}^{(l)} \sum_{z} = 1^{n^{n(l-1,i)}} \Theta\left(t_k^{(l,j)} - t_z^{(l-1,i)} \right) \exp\left(\frac{t_z^{(l-1,i)}}{\tau_s} \right) - \frac{v_{th}}{\tau} \sum_{z=1}^{n^{(l,j)}} \Theta\left(t_k^{(l,j)} - t_z^{(l,i)} \right) \exp\left(\frac{t_z^{(l,i)}}{\tau} \right) \tag{3.11}$$

$$c := c_k^{(l,j)} = \frac{v_{th}}{\tau} \tag{3.12}$$

Since $c_k^{(l,j)}$ is the same for every neuron we denote is at c for convenience. These equations are derivable, therefore gradient descent can be performed.

3.6.5. Backward pass

In the backward pass, we aim to modify values of the weights (labelled as synapses in figure 3.1) that connect the neurons in order to strengthen or weaken pathways between them to how the output of a neuron in layer $l-1$ have a higher or lower impact on the membrane potential of layer l. For that, we again need to calculate the gradient from the spike errors in order to know the best direction of weight change.

Weight Updating

A weight $w_{i,j}^{(l)}$ between neuron pre-synaptic neuron i at layer $l-1$ and post-synaptic neuron j at layer l receives an error $\delta_k^{(l,j)}$ through backpropagation, this spike error is calculated in the same way as in [2]. This lead to a change $w_{i,j}^{(l)} \to w_{i,j}^{(l)} - \lambda \Delta w_{i,j}^{(l)}$

$$\Delta w_{i,j}^{(l)} = \sum_{k=1}^{n^{(l,j)}} \frac{\partial L}{\partial t_k^{(l,j)}} \frac{\partial t_k^{(l,j)}}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{n^{(l,j)}} \delta_k^{(l,j)} \frac{\partial t_k^{(l,j)}}{\partial w_{i,j}^{(l)}} \tag{3.13}$$

With λ being the training rate hyperparameter.

Gradient Calculations

As a consequence of the above described closed form solution found in equation 3.9. From spike timing, it is now differential, which allows for a computation of an exact gradient as was eluded to in the equation 3.13. Just as in that case, we are dealing with a weight $w_{i,j}^{(l)}$ receiving an error $\delta_k^{(l,j)}$; let's now pick up from where we left off with the calculation of how $w_{i,j}^{(l)}$ effects $t_k^{(l,j)}$, since as said it is deliverable we get:

$$\frac{\partial t_k^{(l,j)}}{\partial w_{i,j}^{(l)}} = \sum_{z=1}^{n^{(l-1,i)}} \Theta\left(t_k^{(l,j)} - t_z^{(l-1,i)}\right) \left[f_k^{(l,j)} \exp\left(\frac{t_z^{(l-1,i)}}{\tau_s}\right) - h_k^{(l,j)} \exp\left(\frac{t_z^{(l-1,i)}}{\tau}\right) \right] \tag{3.14}$$

with

$$f_k^{(l,j)} := \frac{\partial t_k^{(l,j)}}{\partial a_k^{(l,j)}} = \frac{\tau}{a_k^{(l,j)}} \left[1 + \frac{c}{x_k^{(l,j)}} \exp\left(\frac{t_k^{(l,j)}}{\tau}\right) \right] \tag{3.15}$$

$$h_k^{(l,j)} := \frac{\partial t_k^{(l,j)}}{\partial b_k^{(l,j)}} = \frac{\tau}{x_k^{(l,j)}} \tag{3.16}$$

With these calculations the only element we are missing to perform the weight updating calculations from section 3.6.5 is the error.

Spike Errors

Next, we will examine the term $\delta_k^{(l,j)}$ in detail and discuss its calculation. $\delta_k^{(l,j)}$ is the error associated with the spike time $t^{(l,j)}$. As we are dealing with a neuron that can output multiple spikes this error is received from two locations and can be written in the following way:

$$\delta_k^{(l,j)} := \frac{\partial \mathcal{L}}{\partial t^{(l,j)}} = \phi_k^{(l,j)} + \mu_k^{(l,j)} = \underbrace{\sum_{i=1}^{N^{(l+1)}} \sum_{z=1}^{n^{(l+1,i)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l+1,i)}} \frac{\partial t_z^{(l+1,i)}}{\partial t_k^{(l,j)}}}_{\phi_k^{(l,j)} \to \textbf{ Inter Neuronal}} + \underbrace{\sum\nolimits_{z=k+1}^{n^{(l,j)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l,j)}} \frac{\partial t_z^{(l,j)}}{\partial t_k^{(l,j)}}}_{\mu_k^{(l,j)} \to \textbf{Intra Neuronal}} \tag{3.17}$$

The first, represented by $\phi_k^{(l,j)}$, is from the influence of the post-synaptic spikes of the neuron on pre-synaptic spikes, in other words from how the weights have influenced the neuron's behaviour and its effect on the neurons further down the network. This error is different in the output layer than in the other layers, as this is where the loss from the labels enters the system. In this layer o the $\phi_k^{(o,j)}$ is set to be the difference between the number of spikes the neuron should be outputting in based on the label y_i and the real number the neuron is outputting $n^{(o,j)}$. Hence these are the equations:

$$\phi_k^{(o,j)} := \frac{\partial \mathcal{L}}{\partial n^{(o,j)}} = y_i - n^{(o,j)} \tag{3.18}$$

On the other hand in the hidden neurons the error $\phi_k^{(l,j)}$ is the sum of all the errors $\delta_z^{(l+1,i)}$ backpropagated from spikes that spike $t_k^{(l,j)}$ has contributed to. Hence it is formally defined as:

$$\phi_k^{(l,j)} := \sum_{i=1}^{N(l,j)} \sum_{z=1}^{n^{(l+1,i)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l+1,i)}} \frac{\partial t_z^{(l+1,i)}}{\partial t_k^{(l,j)}} = \sum_{i=1}^{N(l,j)} \sum_{z=1}^{n^{(l+1,i)}} \delta_z^{(l+1,i)} \frac{\partial t_z^{(l+1,i)}}{\partial t_k^{(l,j)}} \tag{3.19}$$

and

$$\frac{\partial t_z^{(l+1,i)}}{\partial t_k^{(l,j)}} = \Theta\left(t_z^{(l+1,i)} - t_k^{(l,j)}\right) w_{j,i}^{(l+1)} \left[\frac{f_z^{(l+1,i)}}{\tau_s} \exp\left(\frac{t_k^{(l,j)}}{\tau_s}\right) - \frac{h_z^{(l+1,j)}}{\tau} \exp\left(\frac{t_k^{(l,j)}}{\tau}\right)\right] \tag{3.20}$$

note that $f_z^{(l+1,i)}$ and $h_z^{(l+1,j)}$ are given in equations 3.15 and 3.16 respectively. Therefore we now have everything we need to calculate the errors and the gradients extracted from said errors if we were not dealing with a multi-spiking neuron architecture.

In the second location, represented by $\mu_k^{(l,j)}$ the error comes is the influence the post-synaptic spikes have on the neuron's other post-synaptic spikes due to the reset functionality. The intuition behind these calculations is that spike k in neuron j of layer l will affect all later spikes in that neuron because it will reset the neuron potential. Therefore in a similar way to before $\mu_k^{(l,j)}$ is defined as the sum of all backpropagated errors generated by spikes the neuron generates after k. Formally we write it in the following way:

$$\mu_k^{(l,j)} := \sum_{z=k+1}^{n^{(l,j)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l,j)}} \frac{\partial t_z^{(l,j)}}{\partial t_k^{(l,j)}} = \sum_{z=k+1}^{n^{(l,j)}} \delta_z^{(l,j)} \frac{\partial t_z^{(l,j)}}{\partial t_k^{(l,j)}} \tag{3.21}$$

where

$$\frac{\partial t_z^{(l,j)}}{\partial t_k^{(l,j)}} = \frac{v_{th}}{\tau x_z^{(l,j)}} \exp\left(\frac{t_k^{(l,j)}}{\tau}\right) \tag{3.22}$$

With the combination of these two elements in the error, an error can be calculated to perform the backpropagation, as seen in section 3.6.5.

4 Residual Connections

In the field of deep learning residual connections are a standard piece of the puzzle and they have been successfully used to allow for the training of deeper networks, such as in the case of ResNet [12]. These connections are tried in tested in the field of ANNs but are having some issues being implemented into SNNs In this section we will cover what they are, why they are used, why they are relevant to implement in SNNs, and how they can be implemented in said networks.

4.1. What Is A Residual Connection?

A residual connection is a connection in a neural network that allows for information from an earlier layer to be transmitted into a layer lower down the network. This is usually used to skip two or three layers in the network to create what is called a block, a 2-layer example can be seen in figure 4.1.

Of special note for this project is what we will refer to as the fuse function. That is how the residual input $F(x)$ is fused with the skip connection x; in the case of the example given in figure 4.1 that would be a simple addition, this is the function used for the vast majority of approaches to residual connections as it has been shown to produce great results [12]. However, the reason this can be used is that the values of both x and $F(x)$ are numerical, this sort of simplistic function can not be used in SNNs where you have more concerns but also more flexibility.

4.2. Why They Are Used?

The theoretical idea behind these connections is that the output from the extra layers $F(x)$ in figure 4.1 will provide some sort of fine-tuning and additional information to the data in x in figure 4.1. These connections were also designed to help with the banishing and exploiting gradient problem that deep ANNs have.

In the past they have been used in ANN architectures like ResNet [12] and have empirically shown that they help the network converge faster. They are a standard part of the deeper networks used in deep learning and

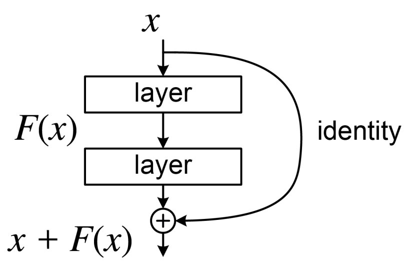


Figure 4.1: A typical example of a "*block*" in a residual network that uses a numerical sum as a fuse function, this is the case in [12], where this figure is from.

theoretically provide the network with an information channel that passes through fewer filters, providing it a channel in which to store the important and basic information [12].

4.3. Residual Connections in Spiking Neural Networks

With an understanding of both Spiking Neural Networks and Residual connections, we can now explore how the principles of residual connections can be applied to SNNs. In this part of the thesis, we shall first go over some examples of how they have been applied on networks with other types of neurons and then we will delve into why and how we implemented them using our neurons.

4.3.1. How Are They Implemented In Literature?

This approach is not the first time that there has been an attempt to add residual connections to Spiking Neural Networks, yet, to the best of my knowledge this is the first time that residual connections are implemented in multi-spiking neurons with precise time that use spike time backpropagation. In this section, we will go over some examples of implementations of residual connections in SNNs and what we can learn from them for this implementation.

In the past residual connections have usually been implemented on discreet time and then usually implemented using a boolean logic function to combine them [26, 18]. From these cases, we get the idea to attempt to fuse the information of the two inputs, an idea that eventually evolved into the **Average fuse function**.

A different approach as the one proposed by [13], although they also use discrete time they use a more complex way of fusing the inputs. This method and their goal to keep the transmitted information intact is the inspiration behind our **Append fuse function**.

Another paper of note is [13], the approach from this paper we can see that deep SNNs have the potential to be implemented successfully to allow for deeper spiking networks. This paper gives credence to our plan to implement residual connections to improve the depth and accuracy of the precise time multi-spiking neurons of the [2] paper.

Further information in the related existing literature on the topic can be found in the related work section of the attached academic paper.

4.3.2. Banishing Spikes

When working with SNN, in particular with implementations such as [2] and the one covered in this thesis, we must acknowledge that they suffer from a problem of decaying spikes and silent outputs. This causes problems due to the use of spike time backpropagation, which makes the network dependent on output spikes to update the weights. This in combination with the decrease in the average number of output spikes as the depth of the layer increases means that we are limited in the depth of a non-residual network.

The reason for this decrease in spikes as the depth increases is that, unless the weights of the synapses between two layers l and $l+1$ are on average inefficiently high, the number of spikes that a neuron needs to receive to produce and output spike will be greater than one; hence the number of spikes in outputted by layer $l \to N^l$ will be greater than the number for layer $l+1 \to N^{(l+1)}$.

An example of this is in our implementation, the network goes silent after four or five layers of depth, depending on the dataset. This limitation of depth is a crucial issue, as deep learning has shown that deeper networks are able to produce better results than their shallow counterparts [25]. In this thesis, we therefore aim to solve this problem using residual connections to increase the number of spikes received by deeper layers.

4.3.3. Why Residual Connections Are Important In Spiking Neural Networks?

Other than the same reasons behind adding them to ANNs there is an extra motivation to add them to SNNs in order to solve the above-mentioned spike decay problem. The addition of outputs from more active layers into the lower layers of the network can help to maintain spike activity as the network gets deeper, hence increasing the maximum deph.

4.3.4. How They Can Be Implemented In Our Scenario?

When implementing Residual connections in SNN with precise time you can not use methods such as simple addition (as is used in most ANNs such as [12]) or use a binary logic function as in the case of discrete-time SNNs (as in the case of [26]). On the other hand, it offers a lot more flexibility, in order to exploit this the suggested architecture has the ability to accommodate different fuse functions and I created three different fuse functions to test. These are described in the above paper in the *Fuse functions* section.

4.3.5. Forward Pass With Residual Connections

As expected the residual connection changes the forward pass calculations. The main change happens to the equations of 3.3 as it now looks like this in the case of the **append fuse function** with a jump length of d:

$$u^{(l,j)}(t) = \sum_{i=1}^{N^{(l-1)}} w_{i,j}^{l} \sum_{z=1}^{n^{(l-1,i)}} \epsilon(t - t_z^{(l-1,i)}) + \underbrace{\sum_{u=1}^{N^{(l-d)}} w_{u,j}^{l} \sum_{z=1}^{n^{(l-d,u)}} \epsilon(t - t_z^{(l-d,u)})}_{\text{Input from the jump connection at layer } l-d} - \sum_{z=1}^{n^{(l,j)}} \eta(t - t_z^{(l,j)}) \quad (4.1)$$

As it might seem obvious this uses more weights $(w_{i,j}^{l} + w_{u,j}^{l})$ in layer l as there are now more neurons that the neurons in layer l need to connect to.

In the case of the **average fuse functions** the equation for $u^{(l,j)}(t)$ 3.3 remains the same, nevertheless, the value of $t_z^{(l-1,u)}$ does change, as it is now affected by the values of $t_z^{(l-d,u)}$. Formally, in the case of an average residual connection with layer $l-d$ the following change would occur for every neuron in layer $l-1$ with neurons $i = 0, ..., x$ and each neuron in layer $l-d$ with neurons $u = 0, ..., x$ there are three cases:

1. The number of spikes emitted by both neurons are the same $n^{(l-1,i)} = n^{(l-d,u)}$
2. The number of spikes emitted by neuron i of layer $l-1$ is the smaller of the two $n^{(l-1,i)} < n^{(l-d,u)}$
3. The number of spikes emitted by neuron i of layer $l-1$ is the larger of the two $n^{(l-1,i)} > n^{(l-d,u)}$

In case one the number of spike times match $\forall t_z^{(l,i)}$ exists a $t_z^{(l,u)}$, therefore we can simply perform an average to compute the new spike times, mathematically this is as follows: $t_z^{(r,i)} \leftarrow \frac{t_z^{(l-1,i)} + t_z^{(l-d,i)}}{2}$.

In cases two and three the situation is mirrored depending on what neuron has the least amount of activation. It is for this reason that in this explanation with the aim to avoid repetition, I will only be considering case two as due to the spike decay problem 4.3.2 it is the most common, case number three is the same but reversed as in which spike first runs out of spikes to compare. While the spikes can be matched; meaning that for every $t_z^{(l-1,i)}$ that has a paired $t_z^{(l-d,i)}$ we use the same method as in case one. After one of them runs out of spikes we use the remaining spikes $t_z^{(r,i)} \leftarrow t_z^{(l-1,i)}$ or $t_z^{(l-d,i)}$. Formally the equation would then be:

$$u^{(r,j)}(t) = \sum_{i=1}^{N^{(r)}} w_{i,j}^{r} \sum_{z=1}^{n^{(r,i)}} \epsilon(t - t_z^{(r,i)}) - \sum_{z=1}^{n^{(r,j)}} \eta(t - t_z^{(r,j)}) \quad (4.2)$$

4.3.6. backpropagation With Residual Connections

The backward pass of the residual layers also has to be modified in order to accommodate for the equations 4.1 and 4.2 of the residual forward pass. The logic behind the equations remains the same as in chapter 3.6.5 nonetheless the math needs to change in order to accommodate the new connections.

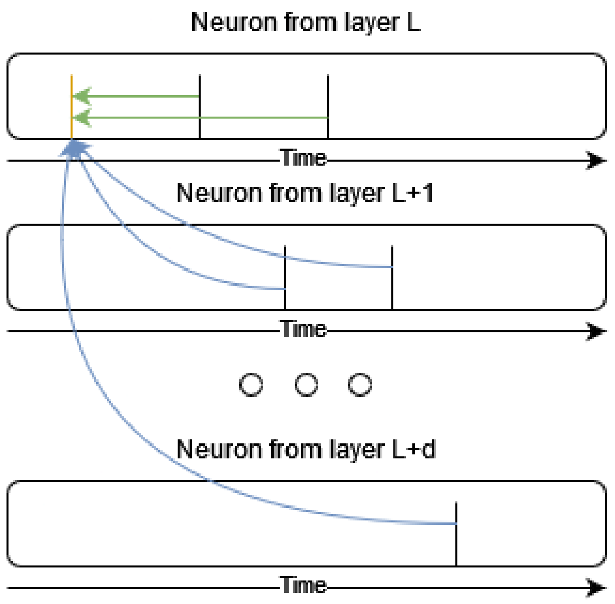


Figure 4.2: Visual representation of the error flow into a spike with time $t_1^{(l,j)}$ at later. For simplicity, in this diagram, we only show the error flows for a single neuron in each layer. In this representation, you can see that the spike time receives errors from both other spikes emitted later on in its neuron (here represented with orange arrows) and from spike times that it has influenced lower in the network (represented with blue arrows), in the case of this figure layer l is jump connection layer, so it receives errors from both the next layer and the residual layer.

The main change occurs in the weight update equation 3.17 as it now is defined as such:

$$\delta_k^{(l,j)} := \frac{\partial \mathcal{L}}{\partial t^{(l,j)}} = \phi_k^{(l,j)} + \mu_k^{(l,j)} =$$
$$\underbrace{\left(\sum_{i=1}^{N^{(l+1)}} \sum_{z=1}^{n^{(l+1,i)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l+1,i)}} \frac{\partial t_z^{(l+1,i)}}{\partial t_k^{(l,j)}} + \sum_{i=1}^{N^{(l+d)}} \sum_{z=1}^{n^{(l+d,i)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l+d,i)}} \frac{\partial t_z^{(l+d,i)}}{\partial t_k^{(l,j)}}\right) \times 0.5}_{\phi_k^{(l,j)} \rightarrow \textbf{Inter Neuronal}}$$
$$+ \underbrace{\sum_{z=k+1}^{n^{(l,j)}} \frac{\partial \mathcal{L}}{\partial t_z^{(l,j)}} \frac{\partial t_z^{(l,j)}}{\partial t_k^{(l,j)}}}_{\mu_k^{(l,j)} \rightarrow \textbf{Intra Neuronal}} \tag{4.3}$$

As you can see the change occurred in the inter-neuronal error component as the neuron now has a more direct effect on layers further down the network, hence the formula was changed to reflect that.

In figure 4.2 one can see how the error flows back from both the next layer and the layer that receives the output as a jump connection input.

4.3.7. Implementation Of The Residual Connection

Now that we have looked at how we went about implementing the residual connection and the fuse functions. An overview of it can be seen in figure 4.3. As you can see the inputs of the two channels of the residual connection are inputted into a fuse function, these are the fuse functions that can be swapped out depending on how you wish to mix the input of the previous and jump layer.

As described in the academic paper section of this thesis we utilize a function that takes the average spike times of the spike belonging to the two input streams, called the **Average fuse function**.

Secondly, I developed a function that uses an append method to feed the input streams of the two layers into the residual layer as if it were a single layer with a neuron count equal to the sum of the neurons of the two input layers (formally, $N^{pre} = N^{(l-1)} + N^{(l-d)}$), this we call the **Append fuse function**.

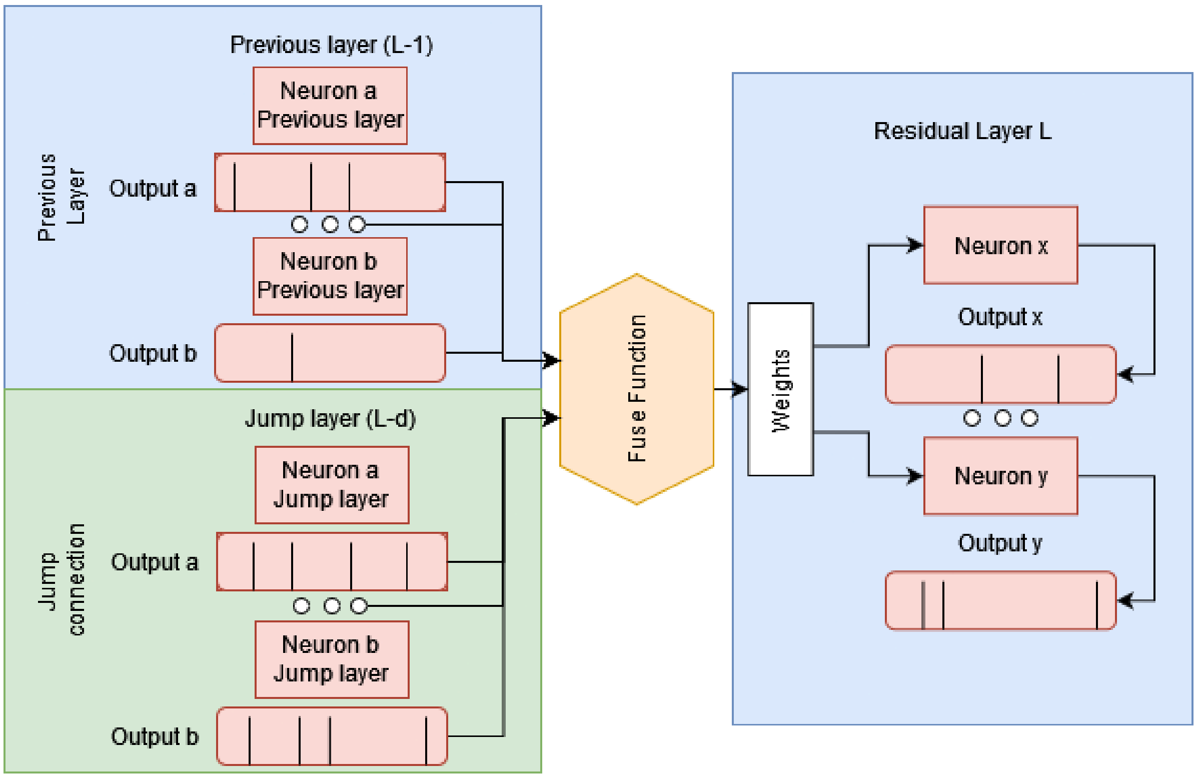


Figure 4.3: An overview of how the different layers in a residual connection interact with each other in the implementation. This architecture was designed with the intent of being as flexible as possible and allowing for the testing of different fuse functions to better take advantage of the flexibility given to us by the multi-spiking and precise time neurons of the network.

Finally, there are the fuse functions used for the convolutional layers, called **Convolutional Append Fuse Function** and **Convolutional Average Fuse Function** these follow the same logic as their non-convolutional counterparts with some modifications to make them work in the architecture and in the case of the **Convolutional Append Fuse Function** the appending is made on the channel dimensions instead of the neuron dimension.

In our **fuse functions** as shown in the appendix 5 you can see we implement them to allow them all to be used with delay. These fuse functions had to be accompanied by changes to both the forward and backward pass calculations to accommodate them and the changing size of the inputs. In the source code it is also of note that I made use of CUPY [20], a Python library focused on optimization in GPUs to speed up the processing times.

The rest of the source code can be found in the projects https://github.com/AlexDeLos/bats.git.

4.3.8. Improvement To The Existing Code Base

In addition to the changes described in the above section and the paper attached to this thesis, I have also implemented several quality-of-life changes to help with the ability to replicate my work. These changes include but are not limited to allowing every script to be called from the root folder, with arguments representing hyperparameters such as the number of hidden layers, a learning rate scheduler, the length of the residual connection jumps, batch size, number of neurons per layer, the user of weights and biases to record the results, number of epochs, amongst many others.

Thanks to this implementation the script can now be called with the hyperparameters and architecture changes without having to do any modifications to the code. This was very useful in my case to generate sbatch files with the python script's hyperparameters being able to be determined when the script was called, unlike the original version of the code base where to change the hyperparameters one needed to edit the code in the files.

4.3.9. Addition Of Learning Rate Schedulers

Unlike the original implementation for some of my experiments, I use a more complex learning rate scheduler. For these there are two versions that I made, it is important to note that due to the lack of a validation dataset all learning rate reduction decisions are made based on the training loss, not the test loss.

Slope Scheduler the first of the implemented schedulers keeps track of the slope of best fitting linear regression of loss of the last t training epochs (in the case of my experiments 5 epochs). If the slope of this line is higher than a threshold (in the case of my experiments this hyperparamater was set to -0.1) it then lowers the learning rate by a given ratio (in the case of my experiments with the scheduler this amount was set to 0.75). The goal of this scheduler was to take a more general look at the state of the loss over the last few epochs in order to decide when to decrease the learning rate.

Last Lowest point Scheduler Was the other implemented scheduler and used a simpler logic than its slope-based counterpart. It looked at how many epochs ago the best loss result was and if that number was bigger than a given hyperparamater (in my experiments 5 epochs) then the learning rate was decreased by a ratio (in the case of my experiments with the scheduler this amount was set to 0.75).

References

[1] Alexander Amini et al. *Spatial Uncertainty Sampling for End-to-End Control*. May 2018.

[2] Florian Bacho and Dominique Chu. "Exploring Trade-Offs in Spiking Neural Networks". In: *Neural Computation* 35.10 (2023), pp. 1627–1656.

[3] Sander M Bohte, Joost N Kok, and Han La Poutre. "Error-backpropagation in temporally encoded networks of spiking neurons". In: *Neurocomputing* 48.1-4 (2002), pp. 17–37.

[4] Sander M. Bohté, Joost N. Kok, and Han La Poutré. "SpikeProp: backpropagation for networks of spiking neurons". In: *The European Symposium on Artificial Neural Networks*. 2000. URL: https://api.semanticscholar.org/CorpusID:14069916.

[5] Gregory Cohen et al. "EMNIST: Extending MNIST to handwritten letters". In: *2017 international joint conference on neural networks (IJCNN)*. IEEE. 2017, pp. 2921–2926.

[6] Suxia Cui et al. "Fish Detection Using Deep Learning". In: *Applied Computational Intelligence and Soft Computing* 2020 (Jan. 2020), pp. 1–13. DOI: 10.1155/2020/3738108.

[7] Jason K. Eshraghian et al. *Training Spiking Neural Networks Using Lessons From Deep Learning*. 2023. arXiv: 2109.12894 [cs.NE].

[8] Samanwoy Ghosh-Dastidar and Hojjat Adeli. "Spiking neural networks". In: *International journal of neural systems* 19.04 (2009), pp. 295–308.

[9] J. Göltz et al. "Fast and energy-efficient neuromorphic deep learning with first-spike times". In: *Nature Machine Intelligence* 3.9 (Sept. 2021), pp. 823–835. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00388-x. URL: http://dx.doi.org/10.1038/s42256-021-00388-x.

[10] Yilong Guo et al. "Unsupervised Learning on Resistive Memory Array Based Spiking Neural Networks". In: *Frontiers in Neuroscience* 13 (Aug. 2019). DOI: 10.3389/fnins.2019.00812.

[11] Robert Gütig and Haim Sompolinsky. "The tempotron: a neuron that learns spike timing–based decisions". In: *Nature neuroscience* 9.3 (2006), pp. 420–428.

[12] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[13] Yangfan Hu, Huajin Tang, and Gang Pan. "Spiking deep residual networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 34.8 (2021), pp. 5200–5205.

[14] Zhenhua Huang et al. "Making accurate object detection at the edge: review and new approach". In: *Artificial Intelligence Review* 55 (Mar. 2022). DOI: 10.1007/s10462-021-10059-3.

[15] Dario Izzo et al. *Neuromorphic Computing and Sensing in Space*. Dec. 2022. DOI: 10.48550/arXiv.2212.05236.

[16] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[17] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[18] Yudong Li, Yunlin Lei, and Xu Yang. *Rethinking Residual Connection in Training Large-Scale Spiking Neural Networks*. 2023. arXiv: 2311.05171 [cs.NE].

[19] Medium. *Kernels (Filters) in convolutional neural network (CNN), Let's talk about them*. 2021. URL: https://medium.com/codex/kernels-filters-in-convolutional-neural-network-cnn-lets-talk-about-them-ee4e94f3319 (visited on 04/13/2024).

[20] Ryosuke Okuta et al. "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations". In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[21] J Padarian, B Minasny, and AB McBratney. “Using deep learning to predict soil properties from regional spectral data”. In: *Geoderma Regional* 16 (2019), e00198.

[22] Rachmad Vidya W. Putra and Muhammad Shafique. *FSpiNN: An Optimization Framework for Memory- and Energy-Efficient Spiking Neural Networks*. July 2020.

[23] Nitin Rathi et al. *Enabling Deep Spiking Neural Networks with Hybrid Conversion and Spike Timing Dependent Backpropagation*. 2020. arXiv: `2005.01807 [cs.LG]`.

[24] Alireza Sarraf Shirazi and Ian Frigaard. “SlurryNet: Predicting Critical Velocities and Frictional Pressure Drops in Oilfield Suspension Flows”. In: *Energies* 14 (Feb. 2021), p. 1263. DOI: `10.3390/en14051263`.

[25] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural Networks* 61 (Jan. 2015), pp. 85–117. ISSN: 0893-6080. DOI: `10.1016/j.neunet.2014.09.003`. URL: `http://dx.doi.org/10.1016/j.neunet.2014.09.003`.

[26] Yimeng Shan et al. *OR Residual Connection Achieving Comparable Accuracy to ADD Residual Connection in Deep Residual Spiking Neural Networks*. 2023. arXiv: `2311.06570 [cs.CV]`.

[27] Martino Sorbaro et al. “Optimizing the energy consumption of spiking neural networks for neuromorphic applications”. In: *Frontiers in neuroscience* 14 (2020), p. 516916.

[28] Amirhossein Tavanaei et al. “Deep learning in spiking neural networks”. In: *Neural networks* 111 (2019), pp. 47–63.

[29] Han Xiao, Kashif Rasul, and Roland Vollgraf. “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms”. In: *CoRR* abs/1708.07747 (2017). arXiv: `1708.07747`. URL: `http://arxiv.org/abs/1708.07747`.

[30] Huo Yingge, Imran Ali, and Kang-Yoon Lee. “Deep Neural Networks on Chip - A Survey”. In: Feb. 2020, pp. 589–592. DOI: `10.1109/BigComp48618.2020.00016`.

[31] Matthew D Zeiler and Rob Fergus. “Visualizing and understanding convolutional networks”. In: *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*. Springer. 2014, pp. 818–833.

5
Source Code Example

In this Appendix we show some snippets of the source code. For the full code please go to the GitHub repository. The following snipets cover the code corresponding to the **fuse functions**.

```
def fuse_inputs_append(pre_input, jump_input, count_pre, count_jump, max_n_spike, delay =
    False) -> Tuple[cp.ndarray, cp.ndarray]:
    pre_is_inf = cp.all(cp.isinf(pre_input))
    jump_is_inf = cp.all(cp.isinf(jump_input))
    if delay and not(pre_is_inf or jump_is_inf):
        copy_pre_spike_per_neuron = cp.copy(pre_input)
        non_inf_values_pre = copy_pre_spike_per_neuron[cp.isfinite(copy_pre_spike_per_neuron)
            ]  % Select non-inf values
        average_non_inf_pre = cp.mean(non_inf_values_pre)
        copy_jump_spike_per_neuron = cp.copy(jump_input)
        non_inf_values_jump = copy_jump_spike_per_neuron[cp.isfinite(
            copy_jump_spike_per_neuron)]
        average_non_inf_jump = cp.mean(non_inf_values_jump)
        time_delay = average_non_inf_pre - average_non_inf_jump
        jump_input = jump_input + time_delay
    result_count = cp.append(count_pre, count_jump, axis=1)
    if jump_input.shape[2] != pre_input.shape[2]:
        jump_input = cp.pad(jump_input, ((0, 0), (0, 0), (0, pre_input.shape[2] - jump_input.
            shape[2])), mode='constant', constant_values=cp.inf)
    result_spikes = np.append(pre_input, jump_input, axis=1)
    return result_spikes, result_count
```

Listing 5.1: Python function for fusing inputs and appending spikes in an MLP architecture

```
def fuse_inputs(pre_input, jump_input, count_residual, count_jump, max_n_spike, delay = False
    ) -> Tuple[cp.ndarray, cp.ndarray]:
    if delay:
        copy_pre_spike_per_neuron = cp.copy(pre_input)
        non_inf_values_pre = copy_pre_spike_per_neuron[cp.isfinite(copy_pre_spike_per_neuron)
            ]  # Select non-inf values
        average_non_inf_pre = cp.mean(non_inf_values_pre)
        copy_jump_spike_per_neuron = cp.copy(jump_input)
        non_inf_values_jump = copy_jump_spike_per_neuron[cp.isfinite(
            copy_jump_spike_per_neuron)]
        average_non_inf_jump = cp.mean(non_inf_values_jump)
        time_delay = average_non_inf_pre - average_non_inf_jump
        jump_input = jump_input + time_delay

    result_count = cp.maximum(count_residual, count_jump)

    batch_size_res, n_of_neurons_res, max_n_spike_res = pre_input.shape
    batch_size_jump, n_of_neurons_jump, max_n_spike_jump = jump_input.shape

    not_inf_mask_res = cp.logical_not(cp.isinf(pre_input))
    not_inf_mask_jump = cp.logical_not(cp.isinf(jump_input))

```

```python
    inf_mask_res = cp.isinf(pre_input)
    inf_mask_jump = cp.isinf(jump_input)

    xor_combined_mask = cp.logical_xor(not_inf_mask_res, not_inf_mask_jump)
    or_combined_mask = cp.logical_or(not_inf_mask_res, not_inf_mask_jump)
    and_combined_mask = cp.logical_and(not_inf_mask_res, not_inf_mask_jump)

    #! for now if both are inf we take residual, we should take whichever is not inf
    get_non_infinite = cp.where(inf_mask_res, jump_input, pre_input)
    get_non_infinite = cp.where(inf_mask_jump, pre_input, get_non_infinite)
    result_times = cp.where(or_combined_mask, jump_input, pre_input)

    # we make the average of both inputs
    if batch_size_res != batch_size_jump:
        raise ValueError("The batch size of the residual and jump input must be the same")
    if n_of_neurons_res != n_of_neurons_jump:
        raise ValueError("The number of neurons of the residual and jump input must be the
            same")
    if max_n_spike_res != max_n_spike_jump:
        raise ValueError("The max number of spikes of the residual and jump input must be the
             same")

    result_times = cp.where(xor_combined_mask,
                            get_non_infinite,
                            cp.mean(cp.array([pre_input, pre_input]), axis=0))

    return result_times, result_count
```

Listing 5.2: Python function for fusing inputs and averaging spikes using the methods described.

```python
def aped_on_channel_dim(pre_spike_per_neuron, pre_n_spike_per_neuron, jump_spike_per_neuron,
                       jump_n_spike_per_neuron, shape_of_neurons, delay=False):
    batch_size, spikes, max_n_spikes = pre_spike_per_neuron.shape
    jump_batch_size, jump_spikes, jump_max_n_spikes = jump_spike_per_neuron.shape

    if delay:
        copy_pre_spike_per_neuron = cp.copy(pre_spike_per_neuron)
        non_inf_values_pre = copy_pre_spike_per_neuron[cp.isfinite(copy_pre_spike_per_neuron)
            ]
        average_non_inf_pre = cp.mean(non_inf_values_pre)
        copy_jump_spike_per_neuron = cp.copy(jump_spike_per_neuron)
        non_inf_values_jump = copy_jump_spike_per_neuron[cp.isfinite(
            copy_jump_spike_per_neuron)]
        average_non_inf_jump = cp.mean(non_inf_values_jump)
        time_delay = average_non_inf_pre - average_non_inf_jump
        jump_spike_per_neuron = jump_spike_per_neuron + time_delay

    if batch_size != jump_batch_size:
        raise RuntimeError("The batch sizes of the two inputs are not the same")
    pre_x, pre_y, pre_c = shape_of_neurons.get()

    if max_n_spikes != jump_max_n_spikes:
        print('Warning: the maximum number of spikes is not the same for the two inputs',
              max_n_spikes, jump_max_n_spikes)
        max_n_spikes = max(max_n_spikes, jump_max_n_spikes)
        padding_for_max_spikes_pre = ([0, 0], [0, 0], [0, max_n_spikes - pre_spike_per_neuron
            .shape[2]])
        padding_for_max_spikes_jump = ([0, 0], [0, 0], [0, max_n_spikes -
            jump_spike_per_neuron.shape[2]])
        pre_spike_per_neuron = cp.pad(pre_spike_per_neuron, padding_for_max_spikes_pre,
                                      mode='constant', constant_values=cp.inf)
        jump_spike_per_neuron = cp.pad(jump_spike_per_neuron, padding_for_max_spikes_jump,
                                       mode='constant', constant_values=cp.inf)

    pre_spike_per_neuron = cp.reshape(pre_spike_per_neuron, (batch_size, pre_x, pre_y, pre_c,
         max_n_spikes))
    jump_spike_per_neuron = cp.reshape(jump_spike_per_neuron, (jump_batch_size, pre_x, pre_y,
         pre_c, jump_max_n_spikes))
    pre_n_spike_per_neuron = cp.reshape(pre_n_spike_per_neuron, (batch_size, pre_x, pre_y,
        pre_c))
    jump_n_spike_per_neuron = cp.reshape(jump_n_spike_per_neuron, (jump_batch_size, pre_x,
        pre_y, pre_c))

    new_spike_per_neuron = cp.append(pre_spike_per_neuron, jump_spike_per_neuron, axis=3)
    new_spike_per_neuron = cp.reshape(new_spike_per_neuron, (batch_size, pre_x * pre_y * (
        pre_c + pre_c), max_n_spikes))

    new_n_spike_per_neuron = cp.append(pre_n_spike_per_neuron, jump_n_spike_per_neuron, axis
        =3)
    new_n_spike_per_neuron = cp.reshape(new_n_spike_per_neuron, (batch_size, pre_x * pre_y *
        (pre_c + pre_c)))
    return new_spike_per_neuron, new_n_spike_per_neuron
```

Listing 5.3: Python function handelling the fusing of spikes using the **convolutional append function**.

```python
def fuse_inputs_conv_avg(pre_input, pre_n_spike_per_neuron, jump_input,
    jump_n_spike_per_neuron, neurons_shape, delay) -> Tuple[cp.ndarray, cp.ndarray]:
    if delay:
        copy_pre_spike_per_neuron = cp.copy(pre_input)
        non_inf_values_pre = copy_pre_spike_per_neuron[cp.isfinite(copy_pre_spike_per_neuron)
            ]  # Select non-inf values
        average_non_inf_pre = cp.mean(non_inf_values_pre)
        copy_jump_spike_per_neuron = cp.copy(jump_input)
        non_inf_values_jump = copy_jump_spike_per_neuron[cp.isfinite(
            copy_jump_spike_per_neuron)]
        average_non_inf_jump = cp.mean(non_inf_values_jump)
        time_delay = average_non_inf_pre - average_non_inf_jump
        jump_input = jump_input+ time_delay

    result_count = cp.maximum(pre_n_spike_per_neuron, jump_n_spike_per_neuron)


    not_inf_mask_res = cp.logical_not(cp.isinf(pre_input))
    not_inf_mask_jump = cp.logical_not(cp.isinf(jump_input))

    inf_mask_res = cp.isinf(pre_input)
    inf_mask_jump = cp.isinf(jump_input)

    xor_combined_mask = cp.logical_xor(not_inf_mask_res, not_inf_mask_jump)
    or_combined_mask = cp.logical_or(not_inf_mask_res, not_inf_mask_jump)
    and_combined_mask = cp.logical_and(not_inf_mask_res, not_inf_mask_jump)

    #! for now if both are inf we take residual, we should take whichever is not inf
    get_non_infinite = cp.where(inf_mask_res, jump_input, pre_input)
    get_non_infinite = cp.where(inf_mask_jump, pre_input, get_non_infinite)
    result_times = cp.where(or_combined_mask, jump_input, pre_input)

    # return residual_input
    result_times = cp.where(xor_combined_mask,
                        get_non_infinite,
                      cp.mean(cp.array([ pre_input, pre_input ]), axis=0))

    return result_times, result_count
```

Listing 5.4: Python function handelling the fusing of spikes using the **convolutional average function**.