FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

DEPARTMENT OF COMPUTER SCIENCE

Modelling learning systems in artificial and spiking neural networks

MSc thesis in Computer Science

Author
Jens Egholm Pedersen <xtp778@alumni.ku.dk>

Supervisor
Martin Elsman <mael@di.ku.dk>

Abstract

Spiking neural networks receive increasing attention due to their advantages over traditional artificial neural networks. They have proven to be energy efficient, biological plausible, and up to 10^5 times faster if they are simulated on analogue (neuromorphic) chips. Artificial neural network libraries use computational graphs as a pervasive representation, however, spiking models remain heterogeneous and difficult to train.

Using the hypothetico-deductive method, the thesis posits two hypotheses that examines whether 1) there exists a common representation for both neural networks paradigms, and whether 2) spiking and non-spiking models can learn a simple recognition task. The first hypothesis is confirmed by specifying and implementing a domain-specific language, that generates semantically similar spiking and non-spiking neural networks. Through three classification experiments, the second hypothesis is shown to hold for non-spiking models, but cannot be proven for the spiking models.

The thesis contributes three findings: 1) a domain-specific language for modelling neural network topologies, 2) a preliminary model for generalisable learning through backpropagation in spiking neural networks, and 3) a method for transferring optimised non-spiking parameters to spiking neural networks.

The latter contribution is promising because the vast machine learning literature can spill-over to the emerging field of spiking neural networks and neuromorphic computing. Future work includes improving the backpropagation model, exploring time-dependent models for learning, and adding support for neuromorphic chips.

Contents

Acknowledgements

I would like to thank my supervisor Martin Elsman for his patience and counsel. Christian Pehle and Sebastian Schmitt from the Kirchhoff-Institute for Physics at the University of Heidelberg deserves my gratitude for their time and guidance and for their contributions to the Myelin codebase. Both Nicolaj Dauggaard and Jesper Mogensen at the Unit for Cognitive Neuroscience at the University of Copenhagen have been instrumental in the conceptual phases of the thesis, and I am grateful for the physical and mental frame they provided during the thesis work. I would also like to recognise Kerem Kurban for providing internet in Sicily. Finally, I am indebted to Ole Egholm Jackson and Kristin Kaltenhäuser for their help in structuring and revising the final versions of the thesis.

Chapter 1

Introduction

The field of machine learning is evolving rapidly, and has in some recognition tasks surpassed human-level precision [57]. This acceleration is propelled by the advances in artificial neural networks, which recently defeated a human in the advanced real-time strategy game Starcraft II [12] [57, 43, 56]. Maass dubs ANNs second generation neural networks (NNs), because they supercede the first generation networks based on the perceptron. He believes they themselves will be superceded by a third generation of neuron models that closely resemble biology. Unlike neurons in ANNs that follow well-behaved continuous functions, biological neurons communicate by spikes of electricity over time [34].

Because of their biological similarities, third generation NNs are of great interest to (cognitive) neuroscientists [11, 7, 15]. Compared to experimental studies involving living neural substrate, it is significantly cheaper and faster to build neural models either as pure simulations [10, 15] or as analogue circuits that resemble the physical structure of neural networks [68, 58]. Furthermore, researchers have complete control over virtual models to pause, lesion, or even disassemble at will.

Neuromorphic computation is a paradigm that aims to exploit this new generation of network models, by constructing circuits that encode information in spikes over time instead of digital signals [15, 2]. The neuromorphic neuron model can be built in silicon and have shown to accelerate the performance of NNs by a factor of up to 10^5 [2, 58].

A challenge for third generation networks is the relatively poor understanding of learning processes within spiking neurons [63, 68]. This topic is subject to intense research, and there is a growing body of work that attempts to validate the theories through simulated experiments [27, 62]. Within the field of machine learning learning is a well-researched topic, and in the absence of clear neurophysiological learning models, it is a common approach to explore learning algorithms from machine learning in the simulated neural systems [32, 58, 68, 13, 54]. The landscape for neural simulations are, however, heterogeneous and the simulated models typically imply a number of assumptions (such as neuron parameters and model topology), that renders the experiments near-incommensurable [2, 32, 54]. The outcome is that the experimental findings are difficult to validate and re-integrate with theoretical models [54, 2, 7].

This thesis sets out to explore spiking neural networks (SNNs) and their potential for the field of machine learning, focusing on two major challenges for the third generation models: homogeneous modelling and learning.

The thesis is built around the hypothetico-deductive model, in which falsifiable hypotheses are formulated, tested and evaluated. The following two sections will present the hypotheses, the methods for evaluating the hypotheses and finally the thesis structure.

1.1 Hypotheses

This thesis examines two hypotheses:

1. The Volr DSL can translate into spiking and non-spiking neural networks such that the network topologies are retained.
2. Using training it is possible for spiking and non-spiking models to solve an MNIST recognition task.

The hypotheses are driven by two inquiries around modelling and learning.

Hypothesis 1: DSL modelling The first hypothesis tests that the neural networks generated by the DSL are modelled correctly, and translated—without significant deviations—to second and third generation neural networks. Consistent translations are important to ensure correct and reproducible experiments, but are also vital to further the understanding of spiking neural networks: correct rendition bridges the semantics of artificial and spiking neural networks. This indirectly allows users of the DSL to draw on the vast literature of second generation NNs.

To test this hypothesis, it is necessary to derive a common abstraction for NNs of both second and third generation, and to provide proof that the abstraction can be converted into functioning spiking and non-spiking neural models.

For that purpose a compiler will be built that translates neural models into two target paradigms: 1) a second generation NN based on the data-parallel language Futhark, and 2) a third generation NN based on the neural simulator Neural Simulation Toolkit (NEST). The models are expected to resemble each other in topology. That means that they consist of identical collections of nodes, edges and connectivity descriptions.

Hypothesis 2: Learning Second generation NNs are generally capable of learning pattern-recognition problems [56]. It is known that learning also occurs within neural systems, so a similar behaviour is expected in third generation NNs. The second hypothesis verifies that this property exists in both spiking and non-spiking neural networks. If it does not, the domain specific language (DSL) has failed to capture the learning capacities of the second generation networks. Additionally, the hypothesis provides a mean of comparison between the two paradigms.

To test the hypothesis, it is necessary to prove that learning occurs in both paradigms. Three experiments were designed to test this: two trivial logical gates, NAND and XOR, and a recognition task of handwritten digits (MNIST). The experiments will be executed in both second and third generation environments. Afterwards, the results will be compared based on the ability to predict the correct outcome, as well as the speed and quality of learning.

1.2 Thesis structure

The next chapter builds the theoretical basis of the thesis, by defining and establishing relevant theoretical concepts. Then the design and implementation of the DSL will be elaborated on. Chapter 4 presents the experimental setup that aims to test the hypotheses, and the following chapter 5 presents and analyses the results of the experiments. The last chapter discusses the findings of the thesis, concludes on the hypotheses, and finally reflects on the approach of the thesis before proposing areas of future work.

Chapter 2

Theory

This chapter elaborates on the theoretical foundations of the thesis. It is divided into four sections:

1. NNs from perceptron models to ANNs and SNNs
2. learning within NNs,
3. cognitive theories and
4. neuromorphic hardware.

2.1 Neural networks

NN is a broad term that originates in the neuronal models from the biological brain [11]. The general architecture of neural systems can be explained as circuits of neurons connected through weighted edges [56, 11].

In this abstract sense, a neuron is a computational unit that takes a number of signals (inputs) and processes them through a function f, that outputs a single value [16]. Composed in a network, neurons can *compute* complex non-linear functions [16, 11].

In a more concrete sense, NNs compute over either continuous (e.g., voltage and numbers) or discrete signals [56, 57]. Discrete models served as the foundation for the first generation of neural networks [56, 34]. They are based on the perceptron model as seen in Equation 2.1, also known as the McCulluch-Pitts neuron model [16].

$$\sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

2.1.1 Neural networks as directed graphs

These first NNs collect neurons in groups that connect to other groups in a sequence [56]. Figure 2.1 shows an example of such a network.

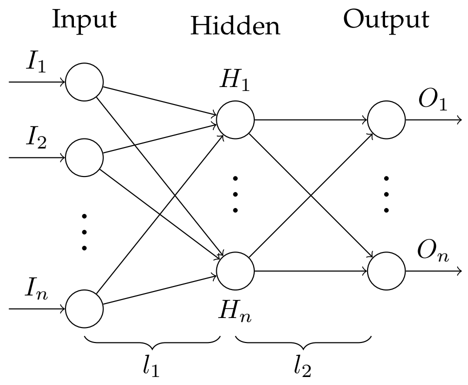

Figure 2.1: An example neural network of depth 3 with two layers (l_1, l_2) and a single hidden neuron group (H).

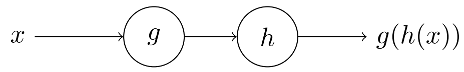

Figure 2.2: Another representation of the network in Figure 2.1, where each layer is considered a function ($l_1 = g$, $l_2 = h$), and the output is derived by composing functions sequentially over the input (x).

The number of groups determines the depth of a network graph [56]. Each group applies a non-linear transformation to the input that is forwarded to the next layer, and so on [4, 56]. From a computational point of view, a neuron group is simply a computational unit, which allows NNs to be abstracted as circuits of units connected in a directed graph [11, 16, 56]. This view can be simplified as shown in Figure 2.2, such that each neuron group (node) is considered a function [53]. Here the output is generated by the sequential composition of activation functions over the input x.

Neuron groups are sometimes referred to as *layers* in the literature, but from a computational perspective it is simpler to view layers as functions, such that they include the output activations for the next neuron group [4]. In this thesis, a layer is defined as computational units that transform input with a non-linear function to produce some output. Thus the network in Figure 2.1 consists of two layers.

Neuron groups or layers that are not directly connected to the input or output of the network are traditionally denoted as 'hidden' because they are only stimulated indirectly [56].

In this representation the 'input' is a vector, whose length is equal to the number of input neurons in the first layer. Conversely, the 'output' is a vector whose length is controlled by the number of output neurons in the network. An NN can then be understood as a function f that maps an input vector to an output vector of arbitrary size [56].

Neuron models typically enrich the input signals (x) with a linear equation

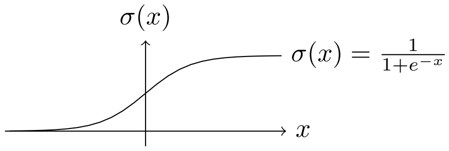

Figure 2.3: A sigmoidal (soft step) function.

as shown in Equation 2.2, where σ is the neuron function [57, 56]. For a single neuron x, the output signal is calculated through a weight (w) and a bias (b). Weights and biases allow the model to adapt the relative importance of each input neuron, thus allowing the model to *train* to a given domain [57, 56].

For each neuron x in the layer j, the output x_j can be calculated given the activation value, weight and bias from the previous layer (i) as shown in Equation 2.2. Here u_j is the weighted sum of the output from the previous layer i, before applying the activation function σ.

$$x_j = \sigma(u_j) = \sigma\left(\left[\sum_{i=1}^{n} w_{i,j} x_i\right] + b_j\right) \tag{2.2}$$

2.1.2 Second generation neural networks

Second generation neural networks augment the perceptron model by a) allowing continuous output values of a neuron and b) parametrising the computation of the neuron by adding an *activation function* that determines the output of the neuron [34]. *Sigmoidal* functions are commonly used for activation functions because they resemble the perceptron step function while retaining continuity (see Figure 2.3) [34].

A number of variations for sigmoidal activation functions exist such as the hyperbolic tangent ($tanh$) and the rectified linear unit (ReLU, see Equation 2.3).

They are applied either in a feed-forward or recurrent (cyclic) manner, where the recurrent variant performs temporal transformations[1] [57].

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{2.3}$$

Normalisation

To align with the activation output domain $[0, 1]$, the input data is typically normalised [4]. One naïve approach is to divide by the maximum value of the input, but that produces a linear effect, which can provide challenges for some networks during classifications. Instead, a sigmoidal function is typically used, because of its resemblance to the step function which polarises the

[1]It is possible to *unfold* recurrent networks to resemble the circular processes until a certain point, achieving a similar effect to temporal signal transformation, see [41].

input values. In differentiable neural networks this incentivises the network to favour extreme values, thus helping its accuracy in some cases. One common variant for normalisation is the softmax function, defined below given the input vector x with N elements:

$$\sigma(x) = \frac{e^x}{\sum_{n=1}^{N} e^x} \tag{2.4}$$

2.1.3 Third generation neural networks

Constructing a network of neuron models essentially creates a non-linear response to a given numerical vector [56]. This transformative view can be adopted to biological (third generation) SNNs, where the data being transferred are no longer vectors, but *spikes* of electrical signals over time [11, 16, p. 32]. In biological networks there is a temporal dimension, in that neurons produce and fire spikes asynchronously to other neurons in the same group [16].

Lapicque worked on a conductance model in 1907 that could describe this process dubbed the *integrate-and-fire* model. The model essentially integrates received current over time, and if the integrated current reaches a certain voltage threshold V_{thr}, the neuron fires [11, 16]. In biological neurons this also implies a spatial dimension, because the current is sent through neurons that extend in space [11].

The biological cell body (soma) separates the neuron cell from the exterior with a membrane. The soma receives impulses from a number of dendrites, and emits spikes through an axon, when the currents across the cell membrane exceeds the voltage threshold (V_{thr}) [11]. The geometry of the components influence the time as well as the amount of current it requires to send impulses through the neuron [16]. This has been modelled to a high degree of precision in the Hudgkin-Huxley model [11]. While it is more precise than models below, it is more complex [11, p. 195], and rarely used in simulations [2, 11, 15].

An idealised version of this is given in the Dirac (δ) function in Equation 2.5 [11, p. 404]. For all values it approaches 0, except when its argument is 0 where it will grow infinitely. In a trial starting at time 0, and ending at time t, this is the equivalent of summing up the n neuronal events that occurred in the duration of the trial. The total area of these 'spikes' sum together to 1 over time t.

$$\rho(t) = \int_0^T \delta(t) = \sum_{i=1}^{n} \delta(t - t_i) = 1 \tag{2.5}$$

This idealised representation is a common mathematical approximation of a sequence of activation functions [11, 16], and the foundation for the *third* generation neural networks, where the computational unit is discrete events over time, instead of continuous-valued (as in second generation NNs) [34].

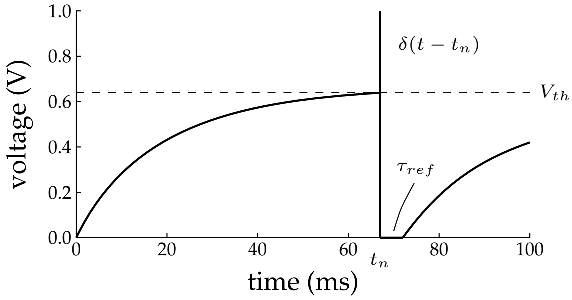

Figure 2.4: A model of how a constant, low input current produces a buildup of voltage inside an integrate-and-fire neuron, which eventually produces a spike. After spiking the neuron enters a refractory period, τ_{ref}, where no voltage is integrated.

The spiking model is based on a neuron that builds voltage over time, until it reaches a threshold voltage (V_{th})m and emits a spike ($\delta(t - t_n)$) [11, 16]. The spike carries a charge, and is received by a post-synaptic neuron as input current, which, in turn, decides whether to fire [11].

After spiking, the voltage inside a neuron is reset to a value (τ_{reset}), from which it begins to accumulate charge again. In a brief period after the activation the neuron enters a period of refraction, where injected voltage is not accumulated, denoted by τ_{ref}, illustrated in Figure 2.4 [16, p. 82].

Lapicque's model has been elaborated in the *leaky integrate-and-fire* (LIF) model, which introduces a numerical "leak" into the model, that acts as a type of memory for the neuron integration [16, 15]. In the leaky model, input voltages decays exponentially over time, meaning that the present voltage depends more strongly on recent input current [16, p. 85].

$$\frac{dv}{dt} = -\frac{1}{\tau_{RC}}(v - cr) \tag{2.6}$$

The LIF equation is given in Equation 2.6, where v is the membrane voltage difference between the interior, and exterior of the neuron membrane, c is the input current, r is the ionic current (or leak) of charge across the membrane, and τ_{RC} is the membrane time constant that determines how quickly the neuron decays to its resting state [11, 16]. As the voltage builds up inside the neuron, r will scale the rate of growth [16]. By tuning the leak, it is possible to control the time with which previous voltages are 'forgotten' [16].

Similar to second generation neural networks, neurons receive input from n input neurons. The connections are commonly referred to as synapses, and are similarly weighted, as well as translated by a bias (see Equation 2.2), such that they contribute differently to the accumulated voltage, given by Equation 2.7 [11].

$$x_j = \sigma(u_j) = \sum_{i=1}^{n} w_i \delta(t - t_i) + b_j \tag{2.7}$$

2.1.4 Coding spikes

Spikes convey information in the form of amplitude, duration, and inter-spike intervals [11]. A number of methods exist to decode the information in the spikes, by a combination of the three parameters [11, 15, 13, 54]. This thesis will focus on so-called rate models, which are commonly used because of their simplicity [16]. Recording neuron spikes over time provides an array of timestamps called a spike train [15]. Rate models count the number of spikes in such a train and divide them by the duration of the trial to produce the spike *rate*, shown in Equation 2.8 [11, 16]. Semantically this is the equivalent of averaging the number of spikes propagated in that interval, and provides the basis for a numerical interpretation of a neuron's output [16].

$$r = \frac{n}{T} = \frac{n}{\rho(t)} = \frac{1}{T}\int_0^T \delta(t) \tag{2.8}$$

When encoding numerical information to spikes, it is useful to express the spikes stochastically, such that one scalar determines the probability that a neuron spikes over time [11]. Assuming that the spikes are independent from each other, this probability can be expressed by a probability distribution such as the Poisson distribution [11]. The Poisson distribution determines the probability of a number of events occurring in an interval, given that the events are known to happen at a fixed rate [11]. It is defined in Equation 2.9 where n is the number of events and λ is the rate with which events happen [11]. Figure 2.5 shows the probability that the number of observed events (k) matches the poisson rate (λ).

$$P(n) = \lambda^n \frac{e^{-\lambda}}{n!} \tag{2.9}$$

To align digital representations with neural spikes, signals are encoded and decoded when entering and leaving the SNN [11]. To compare between non-spiking and spiking networks it is necessary to provide a coding scheme that transfers between the two representations.

During the following, it is assumed that the input represents a constant input current, and that the spike propagation follows the distribution above. The full proof for the following argument is available in Appendix A.

When simulating a LIF neuron at time t, the firing rate depends on the neuron firing threshold V_{thr}, some input current $v(t)$, a maximum firing rate r_{max}, and an activation of the input, following the ReLU activation function (Equation 2.10) and the linear scaling function in Equation 2.2:

$$x_i^l = max\left(0, \sum_j^{N^{l-1}} = W_{ji}^l x_j^{l-1} + b_i^j\right) \tag{2.10}$$

The output is injected as current into the neuron, which will fire if the potential exceeds the threshold. Any surplus charge ϵ is discarded when the

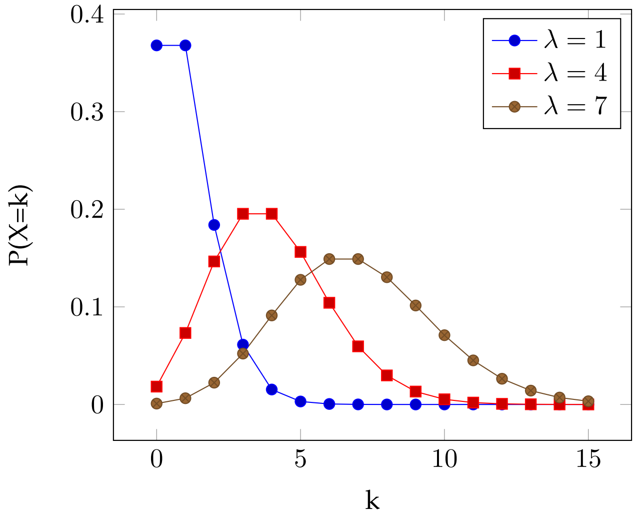

Figure 2.5: The probability of a number of events (k) occurring, given the poisson rates (λ) of 1 (blue), 4 (red) and 7 (brown).

neuron is reset after a spike, and will downscale the firing rate. The spiking rate for a neuron i at time t in the first layer can now be defined as:

$$r_i^1(t) = x_i^1 r_{max} \frac{V_{thr}}{V_{thr} + \epsilon_i^1} - \frac{v_i^1(t)}{t(V_{thr} + \epsilon_i^1)} \tag{2.11}$$

If the input and simulation time step is small, the surplus error ϵ goes towards 0, and can be ignored [54]. For large simulations over long periods of time, however, this assumption has proven to be an issue for Diehl et al. and Rueckauer et al. But for minor networks Equation 2.11 shows a linear relationship between the spike rate and the input [54].

Transferring normalised input from ANN to SNN, then, is a question of determining the exact scaling factor between the ANN input, and the SNN spiking rate.

2.2 Learning

Defining an agent as a system that can act on previous knowledge [56], learning in the context of an agent refers to "the process of gaining information through observation" [59].

Following the above abstraction of neural networks as computations over vectors, "learning" can be understood as the development of consistent patterns, given the same input. Within the machine learning literature, this is commonly referred to as *prediction*. In practice this is expressed in terms of general functions or *rules* in a network [56, p. 704.].

Within machine learning, systems are typically classified into supervised, unsupervised , and reinforced learning systems.

Supervised learning relies on a set of expected outputs which the learning agent must predict, given some input [56]. The agent is told how 'wrong' it was, so it can adapt accordingly. Learning typically happens in a *training* phase, where the agent is allowed to build its internal representation [56]. This representation is later tested in the *testing* phase, where the model is asked to infer based on previously unseen data [56].

Unsupervised learning asks the agent to learn without having any idea of error margin [56]. Rather, the agent is asked to *explore* a domain in search of patterns, which then form the basis for future predictions or classifications [56].

Reinforced learning reinforces the agent through rewards, and discourages it through punishments [56]. Contrary to supervised learning the rewards and punishments are not instructing the agent on what the output should be, but rather how well it performed the task, leaving the agent to infer rules or behaviours by itself [56, p. 873].

The process of learning can either be *inductive* or *deductive* [56, p. 704]. The latter requires a basis in rule-based systems from which new knowledge can be deduced, while the former requires a measurement of success [56, p. 705]. Such a measurement is typically referred to as the *error* or *loss* function, because it shows how much the prediction deviated from the expectation (goal) [56].

Learning in neural networks has shown to be possible within all three types of learning [57, 56], but deduction is rarely seen in the literature, because it is cumbersome to express neural networks through logic transformations [46].

Because of its simplicity and widespread use, this thesis will focus on supervised inductive learning.

2.2.1 Errors in learning

It is worth noting that NNs may learn rules that are not optimal [56]. This can happen in one of two ways: either the network is structurally incapable of learning the domain, or the learned rule is incorrect [56, 15].

A NN is limited in complexity by its number of nodes, since one neuron is expressed through its activation function [11, 56]. Such a structural limit cannot be solved by any other means than augmenting the network [56].

Seeing neural networks as complex non-linear systems, with a number of parameters for the weights and biases, the network can be visualised as a point in a high-dimensional space [56]. Provided that the network is sufficiently complex, the learned rules can still fail to achieve a good accuracy because the system falls into local minima [56].

A similar problem occurs when the model only trains on data that is not representable for the more general domain [56]. This type of "overfitting" can be avoided by exploiting the training and testing phases from above, where networks only train on *parts* of the available data [56, 57]. The remaining data

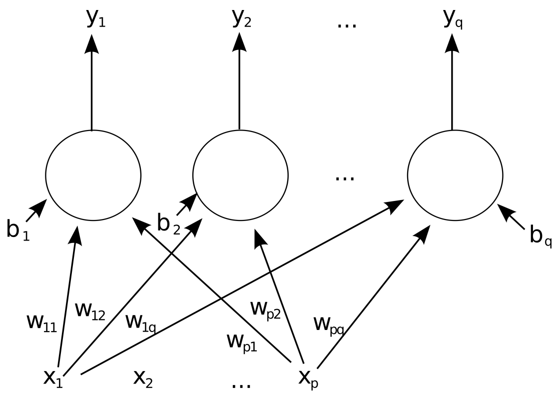

Figure 2.6: A visualisation of weights and biases in a single-layered neural network, given the input x and output y [37].

is applied in the testing phase to validate the generalisation of the model. The limits between data for training and data for testing are not agreed upon, but a 80% training/20% testing split seems to be the default [56, 57].

2.2.2 Backpropagation

In the search for optimal weight/bias configurations, such that prediction errors are minimised [55], the network weights constitute the search space, and the loss function is the subject of optimisation [56].

One loss function to minimize for a supervised network is described in Equation 2.12, where the actual output x is compared to the target (desired) output t, for all n output neurons [56].

$$E = \frac{1}{2}\sum_{i=1}^{n} |x_i - t_i|^2 \tag{2.12}$$

As has been shown, feedforward networks perform this calculation through the sequential application of the weighted activation functions in each layer. One method to minimise this error, is to calculate the gradients of the network layers and iteratively walk in the opposite direction of the error, a technique called gradient descent [55, 56]. Gradient descent requires that the network activation functions are differentiable, such that the gradient of E with respect to the layer weights $(w_1 \cdots w_l)$ can be found, and iteratively adjusted [53], as shown in equation 2.13. Figure 2.6 visualises how the weights relate to the output in a single-layer network.

$$\Delta E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \cdots, \frac{\partial E}{\partial w_l}) \tag{2.13}$$

Deriving the error function from Equation 2.12 gives:

$$\delta_j = y_j - t_j \tag{2.14}$$

The derivation of E_n for an output neuron n from layer i to layer j depends on the weight w_{ji} via the activation input u_j (see equation 2.2). The chain rule for partial derivations can now be applied [4]:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial u_j}\frac{\partial u_j}{\partial w_{ji}} \tag{2.15}$$

and replace the following terms

$$\delta_j = \frac{\partial E_n}{\partial u_j} \qquad z_i = \frac{\partial u_j}{\partial w_{ji}} \tag{2.16}$$

to get

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial u_j}\frac{\partial u_j}{\partial w_{ji}} = \delta_j z_i \tag{2.17}$$

Observing that δ_j can be further expanded to the sum of all units k that receives input from unit j such that

$$\delta_j = \frac{\partial E_n}{\partial u_j} = \sum_k \frac{\partial E_n}{\partial u_k}\frac{\partial u_k}{\partial u_j} \tag{2.18}$$

then Equation 2.16 can be substituted into Equation 2.18 while applying the chain rule to produce:

$$\begin{aligned} \delta_j &= \sum_k \frac{\partial E_n}{\partial u_k}\frac{\partial u_k}{\partial u_j} \\ &= \sum_k \delta_k \frac{\partial u_k}{\partial \sigma_j}\frac{\partial \sigma_j}{\partial u_k} \\ &= \sum_k \delta_k w_{kj} \sigma'_j \\ &= \sigma'_j \sum_k w_{kj}\delta_k \end{aligned} \tag{2.19}$$

This more general form of backpropagation can be chained through layers, where the outer most δ_j term—also called the layer "error"—is defined as in Equation 2.14.

In each layer weights are updated as shown in equation 2.20, to approach an optimal configuration. For biases, the derivative $\partial a_j \;/\; \partial b_j = 1$ such that bias updates are given by Equation 2.21. Here γ is a factor that controls the speed with which the weights are corrected (learning rate).

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i} \tag{2.20}$$

$$\Delta b_i = -\gamma \delta_j \tag{2.21}$$

The learning rate exists to avoid "shooting over" optimal points [56]. Typically a momentum is added to the learning rate, depending on the norm of δ_j, to scale the adaptation to the degree of error, see [61, 71]. Several other techniques have been invented to circumvent the problem of local minima and optimise the learning of the model [71, 57], but they will not be covered here.

2.2.3 Weight and bias normalisation

When applying backpropagation in SNNs, it is important to be aware of the dissonance between the gradient, differentiable activation models and the LIF models [13, 54]. The approximated coding scheme from Section 2.1.4 assists in the translation of the input, but it is reasonable to normalise the weights and biases to avoid vanishing gradients. Rueckauer et al. [54] propose a scheme in which weights are scaled according to the activation potential of the previous layer, over the activation potential a of the current layer:

$$w^l \rightarrow w^l \frac{a^{l-1}}{a^l} \tag{2.22}$$

Unfortunately this model is prone to outliers, and they propose a robust normalisation scheme, where a is set to a percentile of the activation values of a layer [54].

2.3 Cognitive theories

The theoretical development of neural networks has been heavily inspired by the mammalian brain [56, 43]. At the same time, cognitive systems have been studied extensively, outside of computer science. However, a convincing connection between cognitive processes and neurophysiology has yet to be made [64, 43, 52, 64, 68]. One way to approach this missing theoretical link, is to study the rehabilitation of brains with damages (so called lesions) [52, 39, 47]. Depending on the location and rehabilitation of the lesions, theories can be tested on their ability to correctly predict the empirical evidence [39, 38].

Mogensen studied such lesions and arrived at a theoretical framework he dubbed the reorganisation of elementary functions (Reorganisation of Elementary Functions (REF)). The REF theory divides the brain into localized and highly specialised, information processing elementary functions (EF). These basic modular functions are contained within substructures in the brain, that interact with each other to form more advanced processing units, dubbed algorithmic strategies (AS) [39].

An algorithmic strategy combines the capacity of multiple EFs into a single response [39, 38]. EFs and ASs interplay to create what Mogensen calls the 'surface phenomena', which manifests the behaviour of the system [39]. Surface phenomena are the product of applying an AS to a particular problem. Mogensen assumes that a given AS is evaluated for every success or failure of the surface behaviour predicted by that AS [39]. Such evaluation either strengthens the AS's association with the given behaviour scenario, or it weakens it. According to Mogensen numerous strategies exist, that are similar, even almost identical, to each other in function. When one AS fails to perform in a given task, a search for a more suitable stragegy is triggered [38]. Mogensen believes this to be the basis of cognitive rehabilitation.

The REF theory states a need for parallel systems that perform close to identical tasks. These 'task units' can replace each other if necessary, but their slight differences require them to retrain to the new domain if that happens. This redundancy is not found in traditional machine learning NNs.

2.4 Neuromorphic computing

Neuromorphic hardware is based on the idea of NNs where the activation units are modelled outside the classical von Neumann architecture, either in integrated circuits or in simulated environments [2, 5, 58].

This approach permits the simulators to work several hundred of magnitudes faster than regular ANNs, but at the cost of precision and noise [27, 58]. The precision problems occur because of hardware limitations where the typical weight is restricted to a few bits, compared to larger ANNs [27, 33]. The noise problems are caused by noise in the integrated circuit components [33, 48].

The technology is still relatively young and suffers from a number of practical problems. For instance, networks above a couple of thousand neurons remain problematic with the current technology [58]. Training is also challenging because of the embedded components. Both memory and processing power is limited, which makes is practically difficult to store enough data to recall the spiketrains and calculate the, sometimes complex, weight update rules [2].

One practical approach to combat this problem is to model and simulate the neuromorphic system *outside* the hardware, in a system with sufficient resources. Because the simulated systems have the same topology, the optimised model parameters can be directly transferred to the hardware. This paradigm is dubbed 'learning-to-learn', and have already been shown to produce decent results [58, 2]. However it has still not been generalised to support arbitrary machine learning models.

The state-of-the-art neuromorphic platforms are presented in Section 3.1.2 along with their implementation details.

Chapter 3

A DSL for neural networks modelling

This chapter presents the DSL Volr.

Before presenting and specifying the language itself, it is beneficial to examine existing work within the programming and simulation of second and third generation NNs. Following the existing work, a detailed list of requirements are provided to accurately scope the DSL. Finally, the implementation of the DSL is presented and accompanied by code and test examples.

3.1 Related work

A vast amount of work has been put into the development of software for simulating neural networks. This section covers recent work within the simulation of second and third generation NNs, and extracts relevant findings for use in Section 3.2, covering the requirements of the DSL.

The following list does not claim completeness, due to the fragmented and fast-paced nature of the field. To encompass as many relevant findings as possible, the subsequent references are based on a number of often-cited sources, primarily based on books [4, 56, 15, 33, 43, 45, 53, 55], review papers [57, 5, 35, 68, 26], and articles, which will be referenced below.

3.1.1 Second generation

The perhaps most notable product for this type of networks is the Tensorflow framework [1]. Tensorflow is an application programming interface (API) for the description and execution of directed graph structures, that connects varying activation functions and learning mechanisms through the common abstraction of tensors [36]. Tensorflow is the result of a large collaboration of multiple companies and organisations, who have developed a comprehensive library of both code as well as infrastructure and extensive support for hardware acceleration [36].

The primary advantage of Tensorflow comes from its foundation in tensors as a general abstraction, that can be applied to a wide array of problems [1]. Other frameworks have adapted a similar approach, such as PyTorch [51], scikit-learn [60], Microsoft Cognitive Toolkit (CNTK) [9], Caffe [19] and Theano [65].

The frameworks Lasagne and Keras are effectively higher-level abstraction built on top of Theano and Tensorflow respectively [14, 29]. They both provide imperative APIs for constructing models in steps, while including useful utilities for preprocessing data.

Whereas scikit-learn, CNTK, Tensorflow and Theano target NNs in general, PyTorch and Caffe are frameworks that specifically targets deep NNs [51, 19]. However, they all rely on second generation NN architecture.

To provide a comparison between them, a number of examples are provided below. Listing 3.1 shows a network with a single fully connected layer in Caffe, built to recognise handwritten digits from the popular MNIST dataset [71]. Caffe is verbose compared to the full network definitions in PyTorch (Listing 3.2), Keras (Listing 3.3) and Lasagne (Listing 3.12), but provides additional configuration options for the setting of weights and biases in individual layers.

Listing 3.1: A network layer for the MNIST task in Caffe.

```
layer {
  name: "ip1"
  type: "InnerProduct"
  param { lr_mult: 1 }
  param { lr_mult: 2 }
  inner_product_param {
    num_output: 500
    weight_filler { type: "xavier" }
    bias_filler { type: "constant" }
  }
  bottom: "pool2"
  top: "ip1"
}
```

Listing 3.2: MNIST network in PyTorch.

```
self.fc1 = nn.Linear(28*28, 128)    # Topology
self.fc2 = nn.Linear(128, 10)
x = x.view(-1, 28, 28)              # Activations
x = F.relu(self.fc1(x))
X = self.fc2(x)
```

Listing 3.3: MNIST network in Tensorflow using the Keras API.

```
keras.Sequential([
  keras.layers.Flatten(input_shape=(28, 28)),
  keras.layers.Dense(128, activation=tf.nn.relu),
  keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Listing 3.4: MNIST network in Theano using the Lasagne API.

```
l_in = lasagne.layers.InputLayer(shape=(None, 1, 28, 28),
           input_var=input_var)
```

```
l_hid = lasagne.layers.DenseLayer(l_in, num_units=128,
          nonlinearity=lasagne.nonlinearities.rectify)
l_out = lasagne.layers.DenseLayer(
          l_hid, num_units=10,
          nonlinearity=lasagne.nonlinearities.softmax)
```

In terms of learning, the frameworks are diverse, although gradient descent and auto-differentiation (where transformations are automatically registered and later derived using the chain rule) are among the most common features (seen in Tensorflow, PyTorch, CNTK, Theano and Caffe).

Finally, the Open Neural Network Exchange Format (ONNX) is an open data format for the representation of ANN learning models [44]. ONNX is interesting in this context because, like all the frameworks above, it describes networks as directed graphs, defined by nodes of a certain dimension (shape) connected through edges with certain activations (operations).

Neural networks in Futhark

The data-parallel language Futhark is designed to produce efficient parallel code, suitable for parallel application [24, 18]. Futhark is a functional language which operates with higher-order functions as well as modules [17, 25]. As argued above, second generation NNs can be understood as compositions of functions, which makes Futhark a suitable language for the implementation of NNs.

Futhark supports hardware acceleration through the OpenCL framework, and can integrate with other languages, such as Python through the PyOpenCL interface [30].

Tran developed a Futhark library for machine learning, which allows the construction of densely connected layers and gradient descent training [66].

3.1.2 Third generation

The landscape for third generation software is less homogeneous, which is primarily due to the fact that the field is still young [34, 2]. Secondly, there are two different approaches to the evaluation of SNNs: through simulation on general purpose hardware or specialised analogue (neuromorphic) hardware [34, 10, 2].

For each platform — digital or otherwise — a complete programming environment is developed from scratch, because of the degree of specialisation [68, 33]. This section covers the most important technical details of the environments.

SNN simulators

Based on the review of Blundell et al. [5], this paper will discuss the following third generation SNN simulators: PyNN [10], NEST [21], NEURON [8], Brian [22] and Nengo [15].

PyNN is a "simulator-independent language" [50] that compiles to both simulated and accelerated architectures [10]. Technically PyNN is not a simulator but acts as an interface to any third generation backend [10]. PyNN currently interfaces with Brian, NEST, BrainScaleS and SpiNNaker and is more than 10 years old [10], older than most neuromorphic chips. Their APIs were designed a priori and lacked a number of crucial elements, which the hardware designers sought to resolve by augmenting the interface [48, 50]. The result is a fragmented environment that supports basic morphologies, in which each experiment requires retrofitting to execute correctly on the respective backend [50].

Nengo is a neural simulation environment for large-scale neural models, with a focus on graphical modelling [15]. The Nengo project is based on the Neural Engineering Framework (NEF) that offers a concise language for describing third generation simulations [3], along with the limited rendering of logical computations (such as logical gates and basic mathematical functions) into approximated NN structures [16, 15]. Nengo supports a wide range of backends—non-spiking networks through Tensorflow, simulated spiking networks through its custom OpenCL engine and hardware accelerated networks through the neuromorphic platform SpiNNaker— but has a limited repertoire of models compared to other simulators [42].

Listings 3.5 and 3.6 shows how basic spiking models are defined using Nengo and PyNN.

Listing 3.5: A simple LIF MNIST population network in Nengo.

```
pop_1 = nengo.Ensemble(nengo.LIF(100), 2)
pop_2 = nengo.Ensemble(nengo.LIF(10),  1)
nengo.Connection(pop_1, pop_2)
```

Listing 3.6: A simple LIF MNIST network in PyNN.

```
pop_1 = nest.Create('iaf_exp_cond', 100)
pop_2 = nest.Create('iaf_exp_cond',  10)
nest.Connect(pop_1, pop_2, 'all_to_all')
```

PyNN and Nengo both attempt to converge platform differences into one single API, and offer high-level description of networks with support for detailed configuration. Nengo also offers an approximated model that can be evaluated in Tensorflow [26], but it does not translate to other simulators like PyNN: models written for one simulator cannot be transferred to another [42].

The NEST simulator supports both point neurons and compartmentalised models to support speed as well as sophisticated neuron geometry [21]. NEST claims to focus on "dynamics, size and structure rather than on the detailed morphological and biophysical properties of individual neurons" [21]. It has modelled a large number of neuron models and optimisations [5].

NEURON targets complex and detailed simulations of multi-chamber models, and attempts to model all aspects of the biophysical properties [8]. Brian can be located somewhere between NEST and NEURON in terms of adapt-ability and flexibility because it allows users to inject their own models through custom equations in plain text [22].

Rueckauer et al. [54] implemented a "Spiking neural network conversion toolbox" that converts second generation NNs into SNNs. As shown in Section 2.1.4 they approach this by estimating the firing rate of LIF neurons through fixed current inputs, assuming even Poisson distributed signals throughout the network [54].

Neuromorphic hardware

Based on the review of Walter, Röhrbein, and Knoll [68] and the work from Lin et al. [33], the following section classifies neuromorphic hardware into two categories: either as digital interpretations of neural components, or as analogue emulations of neural tissue.

Digital neuromorphic chips digitise neural signals and mimic neuron behaviour either through the regular von Neumann architecture, or via custom digital components [68]. SpiNNaker is an example of the former, where a number of ARM processors are equipped with controllers for handling timers and interrupts [68]. This permits SpiNNaker to compute arbitrary logic, while retaining a large degree of parallelism [2]. IBM's TrueNorth and Intel's Loihi are examples of neuromorphic hardware with custom digital components [68, 33]. TrueNorth consist of 4096 independently operating neurosynaptic cores, each implementing 256 digital neurons in silicon [68, 6]. The Loihi seems similar to the TrueNorth chip, with the difference that its 128 neuromorphic cores feature programmable synaptic learning rules [33].

Analogue neuromorphic chips construct circuits that equal those of biological neurons [68]. BrainScaleS [58], Neurogrid [67], and ROLLS (Reconfigurable On-line Learning Spiking) [68] are examples of such chips.

BrainScaleS is built on the High Input Count Analog Neural Network (HICANN) chip, that contains up to 512 neurons depending on the hardware configuration [48]. Several HICANN chips can be integrated to allow the simulation of larger networks, where dedicated FPGAs set weights for each neuron and communicate with other FPGAs on chip [68]. Neurogrid models around 10^6 two-compartment neurons, where the dendritic tree is separated from the neuron 'soma' [68]. The spikes are transmitted digitally through RAM [68]. The ROLLS processor consists of 256 analogue silicon neurons with $\sim 1.3 \cdot 10^5$ synapses, but with fixed synaptic weights [68].

3.2 DSL requirements

This section elaborates on four functional requirements for a DSL that will allow the testing of the thesis hypothesis. The requirements steer the specification as defined in Section 3.3 and later the implementation details in Section 3.4.

1. Semantic consistency The overarching goal of the DSL is to allow the translation of NN descriptions into semantically similar models in backend runtime environments. In other words, a network described in the DSL should carry the same semantic structure when translated to second or third generation implementations.

Because of the diverse and incompatible nature of the spiking neural network landscape, this is non-trivial but necessary if the models are to be validated across NN paradigms. This requirement is approached empirically, by illustrating examples in both generations and validate whether they achieved the desired degree of external validity.

2. Translation to second and third generation A second requirement is the translation of the DSL into two runtime environments to permit a sufficient degree of generalisation. It is required that the DSL can translate code that evaluate networks in second generation, simulated third generation, as well as analog third generation (neuromorphic).

3. Learning It is furthermore required that the DSL supports a form of learning, to illustrate the expected theoretical adaptation. For this purpose, supervised learning through backpropagation is sufficient for the purpose of this thesis.

4. Well-typed As a final functional requirement, the DSL is designed to ensure consistency and disallow any networks that are not well-formed at compile time.

None of the above mentioned environments fulfil all four requirements. Models built in Nengo and PyNN can be evaluated in both second and third generation environments, but Nengo does not offer consistent semantics between the backends and PyNN only allows for a partial translation into the neuromorphic platforms. However, PyNN supports a consistent API to describe models that, at least topologically, translate to both simulated and accelerated backends.

The following two sections will present the design and implementation of Volr, while drawing on the requirements above to assess whether and how they are met.

3.3 DSL specification

This sections presents the implementation for the Volr DSL. The main purpose of Volr is to define clear and reproducible experiments whose semantics are retained regardless of the runtime environment. It is built for the concise specification and straightforward translation of neural models into both artificial, as well as spiking, network models. Volr focuses solely on the topology of

$$
\begin{array}{lrcl}
\text{expressions} & e & ::= & n \\
 & & | & \mathtt{let}\ x = e\ \mathtt{in}\ e' \\
 & & | & \mathtt{dense}\ n\, m \\
 & & | & e \oplus e' \\
 & & | & e \ominus e' \\
 & & & \\
\text{values} & v & ::= & \mathtt{net}\ n\ m \\
 & & & \\
\text{types} & \tau & ::= & \mathtt{int}\ |\ \mathtt{net}\ n\ m
\end{array}
$$

Figure 3.1: Expressions, values and types of the Volr language.

$$\frac{}{\Gamma \vdash n : \mathbf{int}} \quad (e1) \qquad\qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad (e2)$$

$$\frac{}{\Gamma \vdash \mathbf{dense}\ l\ r : \mathbf{net}\ l\ r} \quad (e3)$$

$$\frac{\Gamma \vdash e : \mathbf{net}\ l\ m \qquad \Gamma \vdash e' : \mathbf{net}\ m\ n}{\Gamma \vdash e \oplus e' : \mathbf{net}\ l\ n} \quad (e4)$$

$$\frac{\Gamma \vdash e : \mathbf{net}\ l\ r \qquad \Gamma \vdash e' : \mathbf{net}\ l\ r'}{\Gamma \vdash e \ominus e' : \mathbf{net}\ l\ (r + r')} \quad (e5)$$

$$\frac{\Gamma \vdash e : \mathbf{net}\ l\ r \qquad \Gamma[x : \mathbf{net}\ l\ r] \vdash e' : \mathbf{net}\ l'\ r'}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \mathbf{net}\ l'\ r'} \quad (e6)$$

Figure 3.2: Type rules in Volr.

networks, thus separating the network description from generation-specific properties of neurons or neuron populations.

The first requirement on semantic consistency explained in the previous section is met through an unambiguous syntax, heavily inspired by lambda calculus [49]. Figure 3.1 shows the BNF notation for expressions, values and types in Volr. Figure 3.2 lists typing rules for the correct interpretation of the expressions.

The constant expression n is an integer that evaluates to the type **int** ($e1$). Similar to traditional functional languages, the **let** binding binds the string constant x to the expression e when evaluating e' [49]. That can later be referenced in the e' expression through the string x as shown in $e2$.

The **dense** expression describes a fully connected neural network layer, and is the smallest building block in the language. This aligns with the previous understanding of a layer, where a **dense** network layer can be understood as a number of inputs, that are densely connected to a number of outputs (see Section 2.1.1). To calculate the output of the layer, the layer bias is added to the weighted input (output from the previous layer) and given to the activation function. Taking into account the definition of layers as functions over

```
dense 2 2
```

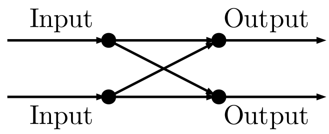

Figure 3.3a: A network with a stimulus containing two channels. The stimulus is fully connected to a population with an excitatory weight of 1. Each circular node represents a single neuron.

```
let s₁ = dense 1 1 in
let s₂ = dense 1 1 in
(s₁ ⊖ s₂) ⦶ dense 2 1
```

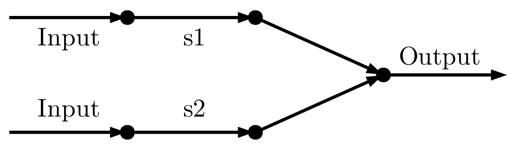

Figure 3.3b: An illustration of a simple binary network, whose two parallel layers share the middle population of size 2.

vectors as described previously, each neuron accounts for the activation of a single dimension in the input/output vectors. In turn, n and m illustrate the *dimensionality* of the network, such that the number of dimensions in the input is truncated (or expanded) to the dimensionality of the output layer.

The $\ominus$ (parallel) operator parallelises two networks by duplicating the input from the previous layer and merging the outputs into a single layer ($e5$). The input from the previous layer is replicated into both e and e', such that the input dimension of the network *must* be shared by the two layers (l). The output from the network is stacked such that each neuron from each population corresponds to one output neuron ($e_{out} + e'_{out}$). This is done to preserve the meaning of each parallel population, where a truncation would loose information.

Semantically the parallel operator provides the ability for a network to perform specialised functions, based on the same stimulus. In the context of neural systems such specialisations are frequently used to balance correctness with compactness: it is cheaper in terms of neurons, and more efficient in terms of accuracy, to allow two subnetworks to specialise than to have one large generalised network [15].

The $\oplus$ (sequential) operator binds two networks sequentially, such that the output layer of the first network becomes the input layer of the second network. This binding is similar to the **dense** operation before, in that the neurons are connected densely, but differ because the operator can connect parallel networks. To connect networks sequentially, it is expected that the output dimensionality of the first network (m) equals that of the input of the second network, as shown in $e4$.

Taken together these constructs can express simple neural networks and the properties of their connections. Figure 3.4 shows a number of example networks that visualises four examples of networks.

```
dense 100 50
  ⦶ dense 50 10
```

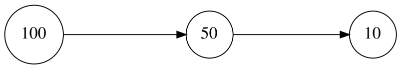

Figure 3.3c: A larger network that can process the MNIST dataset as 10x10 pixel images. The nodes are populations of neurons, where the input corresponds to the pixel size ($10 \cdot 10 = 100$) and the output to the possible classes (0 - 9). Input and output are implicit.

```
let l₁ = dense 2 4 in
let l₂ = dense 2 4 in
let o = dense 8 1 in
  (l₁ ⊖ l₂) ⦶ o
```

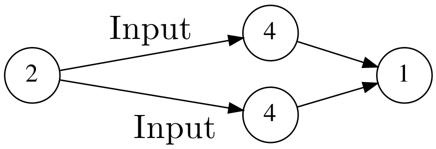

Figure 3.4: An example where a two-dimensional input is split into two nodes and later merged into a node of a single neuron.

3.4 DSL implementation

The compiler for the DSL is implemented in the functional language Haskell [23]. Currently, Volr translates its network models into two runtime environments (backends) based on OpenCL through Futhark and SNNs simulations through PyNN and NEST. However, using the learning-to-learn paradigm above, the PyNN implementations opens for the possibility to transfer the optimised models into neuromorphic hardware such as BrainScaleS.

Futhark was chosen because it is concise and offers useful abstractions that cleanly compose functional models [24]. Considering NNs as a structure of feedforward and feedback functions, Futhark is an elegant solution for the task.

PyNN was chosen for its general purpose API that translates to NEST, but also supports translation into neuromorphic platforms like BrainScaleS.

Figure 3.5 shows the workflow starting with the compilation of the network model, down to the runtime evaluation on each backend. The following section explains the diagram one component at a time.

3.4.1 The DSL compiler

The Haskell compiler consists of five different parts: An abstract syntax tree (AST), an evaluator, a language parser and two code generators for ANNs and SNNs.

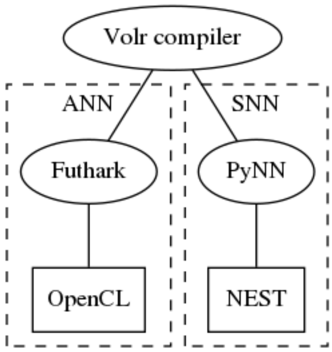

Figure 3.5: The workflow from the Volr compiler to ANN simulations in Futhark and SNN simulations in NEST.

Listing 3.7: The Volr AST in Haskell

```
data Term
  = TmNet Int
  | TmPar Term Term
  | TmSeq Term Term
  | TmRef String
  | TmLet String Term Term
  deriving (Eq, Show)
```

The AST reflects the expressions given in the specification (see Section 3.3) and is shown in Listing 3.7. It is accompanied by a simple type system (in Listing 3.8) that similarly maps to the types given above.

Listing 3.8: Volr type system in Haskell

```
data Type
  = TyNetwork Int Int
  | TyInt
```

The evaluator component evaluates the expression tree into a reference-free model, checking the type integrity in the process.

Listing 3.9 shows the type checking that also occurs in the evaluator step. If the model is malformed, an error is generated to explain why the model could not evaluate correctly. Tests for the type checks and the evaluator are written to ensure the correctness of the compilation. They are elaborated in Rection 3.5 on page 30.

Listing 3.9: Part of the type checking code in Haskell.

```
typeOf :: Term -> EvalState Type
typeOf term =
  case term of
    TmNet n m -> return $ TyNetwork n m
    TmSeq t1 t2 -> do
      leftOut <- sizeRight t1
      rightIn <- sizeLeft t2
      if leftOut == rightIn then do
        leftIn <- sizeLeft t1
        rightOut <- sizeRight t2
```

```
        return $ TyNetwork leftIn rightOut
      else
        throwError $ "Type error: Incompatible network sizes. " ++
                     "Output " ++ (show leftOut) ++ " should " ++
                     "be equal to input " ++ (show rightIn)
```

The language parser, built with the help of the monadic parser combinator library Megaparsec [28], interprets textual input into the AST. The main component of the parser is shown in Listing 3.10 and the whole parser is available in Appendix B on page 56.

Listing 3.10: MNIST network in Theano using the Lasagne API.

```
parseTerm :: Parser Term
parseTerm = (lexeme $ choice
  [ TmNet <$> (symbol "Net" *> integer) <*> integer
  , TmPar <$> (symbol "Par" *> (parens parseTerm)) <*> (parens parseTerm)
  , TmSeq <$> (symbol "Seq" *> (parens parseTerm)) <*> (parens parseTerm)
  , TmRef <$> (symbol "Ref" *> (name))
  , TmLet <$> (symbol "Let" *> (name)) <*> (symbol "=" *> parseTerm)
                           <*> (symbol "in" *> parseTerm)
  ]) <* (optional eof)
```

Finally two code generators are implemented for the translation into second and third generation networks. This translation will be presented along the implementation for the two backends.

3.4.2 Futhark backend

This thesis builds on the work by Tran [66], who implemented a functional library in Futhark for deep learning. The library models NN layers as records with three fields: a function for forward propagation, a function for backward propagation and a cache for weights. The backward propagation uses gradient descent to find optimal weight configurations. Because of its functional nature, layers can simply be joined sequentially by function composition.

The library from [66] has been modified to fit the additions in the thesis, and is available online (see links in Appendix B). In particular, a *replicate* and a *merge* layer has been added, to account for the parallel operator ($\ominus$). A connection that allows two parallel networks to connect to other layers has also been added, although it simply composes tuples of the layer structure (see the file `neural_networks.fut` in Appendix B, page 57).

Replicate layer In practice this layer connects to two other networks, and densely replicates the output to each layer. This happens by storing two separate collections of weights, such that each forward connection is assigned the correct value. Backpropagation works by calculating the error correction on the two sets of weights. The final error sent to the previous layer is the average of the error for each neuron. The algorithm is shown in a shortened form in Listing 3.11. Here the two errors and weights are independently calculated. The weights are stored in the layer as-is, but the errors are averaged before they are passed to the previous layer.

Listing 3.11: Part of the forward and backward propagation algorithms for the replicate layer. Abbreviated for clarity.

```
-- Forward propagation
let forward (act:[]t -> []t)
            (training:bool)
            ((t1, t2): weights)
            (input:input) : (cache, output) =
  let feedForward ((w, b): std_weights t): (tup2d t, arr2d t) = [...]
  let (c1, r1) = feedForward t1
  let (c2, r2) = feedForward t2
  in ((c1, c2), (r1, r2))

-- Backward propagation
let backward (act: []t -> []t)
             (first_layer: bool)
             (apply_grads: apply_grad t)
             ((w1, w2): weights)
             ((c1, c2): cache)
             ((e1, e2): error_in) : b_output =
  let (error1, w1) = backProp w1 c1 e1
  let (error2, w2) = backProp w2 c2 e2
  [...]
  in (average_sum_matrix [error1, error2], (w1, w2))
```

This interpretation fits with the original specification, after which the parallel notation duplicates the 'work' in two separate networks. The complete code for the replication, and the code for translating the DSL abstractions into Futhark, is shown in Appendix B.

Merge layer The merge layer is significantly simpler than the replication layer, because it merely concatenates the inputs from two parallel layers into one single layer. For that reason it also does not contain any weights. In the case of backprogapation, the errors are rerouted back to the population from which the neuron originated. All optimisation logic is left for the regular dense or replicated layers.

Listing 3.12: Functions for forward and backward propagation in the merge layer.

```
-- Forward propagation
let forward  (_:[]t -> []t)
             (_:bool)
             (_: weights)
             ((i1, i2):input) : (cache, output) =
  ((), concat i1 i2)

-- Backward propagation
let backward (_:[]t -> []t) (l1_sz:i32)
             (_:bool)
             (_:apply_grad t)
             (_:weights)
             (_:cache)
             (error_concat:error_in) : b_output =
  ((error_concat[0:l1_sz], error_concat[l1_sz:]), ())
```

This interpretation is also aligned with the original specification, because it allows the parallel populations to propagate their output independent of each other.

3.4.3 PyNN backend

The interpretation to PyNN is done in two steps: a conversion from the DSL into an SNN representation, and a translation between that representation into backend-specific NEST code in Python. The steps are decoupled to enforce the same semantics on the code generation for the neuromorphic and simulation backends. While PyNN is designed as a simulator-dependent API, it is unlikely that the backends will fully support it in the near future (see Section 3.1.2).

SNN translation step

Before Python code for PyNN can be generated, a number of assumptions have to be met. In particular the types of connections (referred to as projections in PyNN) and neuron models have to be described with a full set of neuron parameters. The modelling of this is in Haskell. The neuron parameters for a LIF neuron can be seen in Listing 3.13 with their corresponding default values below.[1]

Listing 3.13: A LIF neuron with exponential decay and conductance-based synapses, modelled in Haskell.

```haskell
data NeuronType =
    = IFCondExp {
        _v_rest :: Float, -- ^ resting potential
        _cm :: Float, -- ^ membrane capacitance
        _tau_m :: Float, -- ^ membrane time constant
        _tau_refrac :: Float, -- ^ refractory time
        _tau_syn_E :: Float, -- ^ excitatory synaptic time constant
        _tau_syn_I :: Float, -- ^ inhibitory synaptic time constant
        _e_rev_E :: Float, -- ^ excitatory reversal potential
        _e_rev_I :: Float, -- ^ inhibitory reversal potential
        _v_thresh :: Float, -- ^ spike initiation threshold
        _v_reset :: Float, -- ^ reset value for membrane potential after a spike
        _i_offset :: Float -- ^ offset current
    }

if_cond_exp :: NeuronType
if_cond_exp = IFCondExp {
    _v_rest = -65.0,
    _cm = 1.0,
    _tau_m = 20.0,
    _tau_refrac = 0.0,
    _tau_syn_E = 5.0,
    _tau_syn_I = 5.0,
    _e_rev_E = 0.0,
    _e_rev_I = -70.0,
    _v_thresh = -50.0,
    _v_reset = -65.0,
    _i_offset = 0.0
}
```

These neuron models form the basis of populations, which is determined by the neuron model and the number of neurons in the population. Populations can be understood as neuron groups or nodes in the network graph.

[1] The current neuron models and their default parameters are taken from PyNN's standard models, available at `http://neuralensemble.org/docs/PyNN/standardmodels.html`.

Along spike sources, which generate spikes over time, they are the basic components in a spiking neural network graph. The definition of nodes as populations of neurons are shown in Listing 3.14.

Listing 3.14: The definition of a node as either a population or a spike source.

```
data Node = Population {
        _numNeurons :: Integer,
        _neuronType :: NeuronType
    }
```

The nodes in the spiking graph are connected through edges shown in Listing 3.15.

Listing 3.15: The definition of an edge as a projection between two nodes with a certain effect.

```
data Edge =
      Projection {
          _effect :: ProjectionEffect,
          _input :: Node,
          _output :: Node
```

The type of the edges are determined by projection effects, which in the current implementation is fixed to describe a dense projection (`AllToAll` in PyNN), whose weights are static and do not change during the simulation.

The model presented here allows to completely reproduce the connection graph of the DSL description. The only difference between the two is the lack of biases and activation functions during the feedforward step in SNNs.

PyNN backend

The final step in the translation of the DSL to PyNN code maps the SNN model from above to executable Python code. A separate neural network library, VolrPyNN, has been developed for this purpose, and is included in Appendix B on page 62. Similarly to the Futhark backend, the Python backend models learning through backpropagation, but instead of using the traditional feedforward activation functions, the simulation backend (NEST) is used to generate the spike data. This requires that the layer knows which derivation function to apply in the backpropagation step. To simplify the code, ReLU is the default gradient model for all layers. The Python library also contains the merge, dense, replicate layer, but because the connections appear through PyNN projections, the parallel layer can be omitted.

Another divergence from the Futhark backend is the normalisation of the output data through softmax (see Equation 2.4 on page 7). This is a commonly used technique for spiking networks to ensure that the output stays positive, despite neurons that do not receive any inputs [54]. This softmax normalisation is only applied in the feedforward step, and does not interfere with the backpropagation.

The backpropagation algorithm is implemented in the dense layer, and a snippet is shown in Listing 3.16.

Listing 3.16: Part of the backpropagation algorithm implemented in PyNN.

```
# Calculate activations for output layer
input_decoded = np.array(self.input_cache)
output_activations = np.matmul(input_decoded, self.weights)

# Calculate output gradients and layer delta
output_gradients = self.gradient_model.prime(output_activations + self.biases)
delta = np.multiply(error, output_gradients)

# Calculate layer backprop and weights, bias updates
backprop = np.matmul(delta, self.weights.T)
weights_delta = np.matmul(input_decoded.T, delta)
(new_weights, new_biases) = optimiser(self.weights, weights_delta,
                                      self.biases, error.sum(axis=0))

self.set_weights(new_weights)
self.biases = new_biases

# Return layer errors
return backprop
```

Listing 3.17 shows an example of the network **`dense`** `100 100` ⦶ **`dense`** `100 10`, compiled to a PyNN model in Python. When activating the input node `node0` the connections are fed through the network to the `node2` output node. Each projection is initiated with random normal distributed weights, similar to the Futhark networks (see Appendix B.3).

Listing 3.17: A simple MNIST network in the PyNN backend from the network in figure 3.4 on page 24. The neuron parameters for the LIF populations have been omitted.

```
p1 = pynn.Population(100, pynn.IF_cond_exp())
p3 = pynn.Population(100, pynn.IF_cond_exp())
p5 = pynn.Population(10, pynn.IF_cond_exp())
layer0 = v.Dense(p1, p3)
layer1 = v.Dense(p3, p5)
l_decode = v.Decode(p5)
model = v.Model(layer0, layer1, l_decode)
```

3.5 DSL verification

To verify that the DSL implementation was successful, and that the models perform as expected when evaluated on the backends, a number of tests were written and automated. This sections elaborates on the reasoning and design of the tests, which are divided into two categories: unit tests and integration tests.

All tests are available online, see Appendix B.

3.5.1 Unit tests

Volr compiler

Each expression construct in the compiler—and their combinations—are tested as to whether the expected output is produced, such that the evaluator is guaranteed to generate well-formed code. An example is shown in Listing 3.19, in which a unit test verifies that the let binding of the constant `x` correctly evaluates to the network `(dense 1 1)`.

Listing 3.18: Part of the evaluation code in Haskell.

```
eval' :: Term -> EvalState Term
eval' term =
  case term of
    TmNet n m -> return $ TmNet n m
    TmSeq t1 t2 -> do
      t1' <- eval' t1
      t2' <- eval' t2
      return $ TmSeq t1' t2'
```

Listing 3.19: A unit test for the correct evaluation of a let binding.

```
it "can evaluate a let binding with a reference" $ do
  let e = TmLet "x" (TmNet 1 1) (TmRef "x")
  eval e `shouldBe` Right (TmNet 1 1)
```

Futhark backend

The Futhark backend was tested using unit tests (using `futhark-test` [18]) for each layer, activation function and loss function. Tests for the dense layer already existed in the library used ([66]). Tests for the merge and replicate layers were added using manual calculations of the expected gradients, as shown in Listing 3.20. The input should produce the expected weight gradients (`output`) here. Further tests for the different combinations of parallel layers were also added.

Listing 3.20: A test for the correct calculation of the backwards weight gradient during backpropagation in a replicate layer

```
-- ==
-- entry: replicate_backward_w
-- input {[[1.0, 2.0, 3.0, 4.0],
--         [2.0, 3.0, 4.0, 5.0],
--         [3.0, 4.0, 5.0, 6.0]]
--
--        [[1.0,  2.0,  3.0,  4.0],
--         [5.0,  6.0,  7.0,  8.0],
--         [9.0, 10.0, 11.0, 12.0]]
--
--         [1.0, 2.0, 3.0]}
--
-- output {[[-25.60,  -36.90,  -48.20,  -59.50],
--          [-59.00,  -87.40, -115.80, -144.20],
--          [-92.40, -137.90, -183.40, -228.90]]}

entry replicate_backward_w input w b =
  let ws = ((w, b), (w, b))
  let (cache, output) = replicate.forward true ws input
  let (_, ((w',_), (_, _))) = replicate.backward false updater ws cache output
  in w'
```

PyNN

The backend-agnostic code of PyNN was tested using the `Pytest` framework [31]. This includes test for the activation functions, spike normalisation functions, error functions, as well as general Python structure.

Especially the model class in the PyNN backend contains error-prone code, because it deals with stateful backends. Unit tests are therefore particularly crucial. The test shown in Listing 3.21 shows an example of such a test that, in this case, validates that the simulator is properly reset between runs:

Listing 3.21: Unit test for PyNN model to validate that the model correctly updates weights

```
def test_nest_model_backwards_reset():
    p1 = pynn.Population(2, pynn.IF_cond_exp())
    p2 = pynn.Population(2, pynn.IF_cond_exp())
    l1 = v.Dense(p1, p2, v.ReLU(), decoder = v.spike_count_normalised, weights =
        1)
    m = v.Model(l1)
    xs1 = np.array([1, 1])
    ys1 = np.array([0, 1])
    xs2 = np.array([1, 1])
    ys2 = np.array([0, 1])
    # First pass
    target1 = m.predict(xs1, 50)
    m.backward([0, 1], lambda w, g, b, bg: (w - g, b - bg))
    expected_weights = np.array([[1, -1], [1, -1]])
    assert np.allclose(l1.get_weights(), expected_weights)
    # Second pass
    target2 = m.predict(xs2, 50)
    m.backward([1, 0], lambda w, g, b, bg: (w - g, b - bg))
    expected_weights = np.array([[-1, -1], [-1, -1]])
    assert np.allclose(l1.get_weights(), expected_weights)
```

The PyNN code for the NEST backend was also tested with `Pytest` [31]. A unit test for the backpropagation was written using numerical gradient descent with a simulated feedforward step. In the test, the layer weights are slightly skewed to approximate the movement along a gradient, which, in the test, was based on a sigmoid function. The resulting changes in the backpropagated error should be minuscule, provided that the algorithm has been implemented correctly. A snippet of the test is shown below.

Listing 3.22: Part of the numerical gradient test for the densely connected layer in PyNN.

```
# Calculate numerical gradients
numerical_gradients = compute_numerical_gradient(xs, ys)
# Calculate 'normal' gradients
gradients = compute_gradients(xs, ys)
# Calculate the ratio between the difference and the sum of vector norms
ratio = np.linalg.norm(gradients - numerical_gradients) /\
           np.linalg.norm(gradients + numerical_gradients)
assert ratio < 1e-07
```

Continuous integration

Continuous integration (CI) is a tool to automatically trigger the unit tests, and is typically associated with updates in a version control system. The projects in this thesis are all exploiting this to continually verify that the software does not regress. Whenever changes are committed and published to their respective repositories on GitHub, the unit tests are executed. Unit tests for NEST are, however too computationally expensive for the continuous integration service due and are left out.

3.5.2 Integration tests

Integration tests exists to verify that the entire toolchain is functioning. It is not intended to test the correctness of the individual parts, but rather that they correctly integrate with each other. The tests are performed inside a containerised environment, as a method to ensure a homogeneous environment.

Integration tests have been performed during the construction of the software, but no automated tests are in place.

The tests assert that the DSL compiler is capable of compiling the models into code for the individual backends that executes and provides the correctly formatted result. The actual values are not verified, because the integration tests assume that the projects are well-behaved independently.

Chapter 4

Experimental setup

This chapter describes the experimental setup that aims to validate the implementation and test the hypotheses introduced in Section 1.1. Firstly, the assumptions and parameters that are the basis of the simulations are described. Second, datasets and methods that are used to test the hypotheses are elaborated.

4.1 Neuron parameters

Since the neural network topologies between the backends are shown to be similar, the parameters related to this topology are shared as well. By applying the theory from Rueckauer et al. [54] (see Section 2.1.4), this section will explain how input data, network weights, and network biases can be transferred from ANNs to SNNs. It will then proceed to describe the setupwhich prepares the scene for the experimental results in the following chapter.

As explained in Section 2.1.4, normalised input in ANNs can be inserted into SNNs with the help of a linear transformation. This transformation has been examined empirically, by constructing a simple one-neuron network, and injecting a constant current over time. By running a number of experiments, it is possible to measure the integrated current in the neuron and the amount of spikes it produces over time.

Figure 4.1 shows one such experiment, in which the membrane potential of a single neuron is plotted over time. The neuron is given a constant input current of 1.3 nA, and produces 5 spikes during the 100 ms simulation time. As expected in Section 2.1.4, the interspike interval is constant.

A neuron only spikes when its membrane potential exceeds its excitation threshold V_{thr}, but depends on a number of parameters to describe the neuron conductivity, current decay time and so on. In PyNN, NEST and BrainScaleS, these parameters can be programatically defined. Table 4.1 shows the default parameters for NEST (also shown in Listing 3.13). The τ_{syn} parameters denote the decay time for the input spike currents. Similarly, the membrane time constant, τ_m, expresses the time it takes for the neuron membrane to decay to its resting state (V_{rest}) if no other input arrives. For the LIF model used in

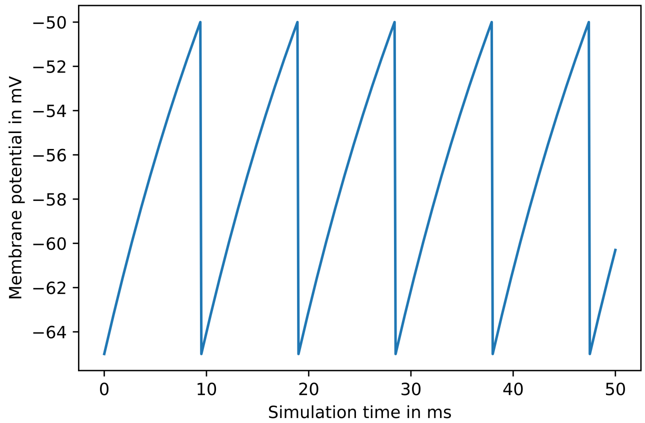

Figure 4.1: Membrane potential for a LIF neuron given a constant input current of 2 nA, simulated over 100 ms.

C_m	1	nF	Capacity of the membrane.
I_{offset}	0	nA	Offset current.
V_{rest}	-65	mV	Resting membrane potential.
V_{reset}	-65	mV	Reset potential after a spike.
V_{rev}^{E}	0	mV	Reverse potential for excitatory input.
V_{rev}^{I}	-70	mV	Reverse potential for inhibitory input.
V_{thr}	-50	mV	Spike threshold.
τ_m	20	ms	Membrane time constant.
τ_{refrac}	0.1	ms	Duration of the refractory period.
τ_{syn}^{E}	5	ms	Decay time of the excitatory synaptic conductance.
τ_{syn}^{I}	5	ms	Decay time of the inhibitory synaptic conductance.

Table 4.1: The names, default values and description of the neuron parameters in PyNN, NEST and BrainScaleS.

PyNN, all τ parameters decay exponentially [10]. The V_{rev} parameters explain the potential to integrate into the neuron, when either excitatory or inhibitory input arrives.

With the exception of I_{offset}, which is the constant input current, the parameters are kept constant in Volr, and used in all spiking backends to avoid spurious influences.

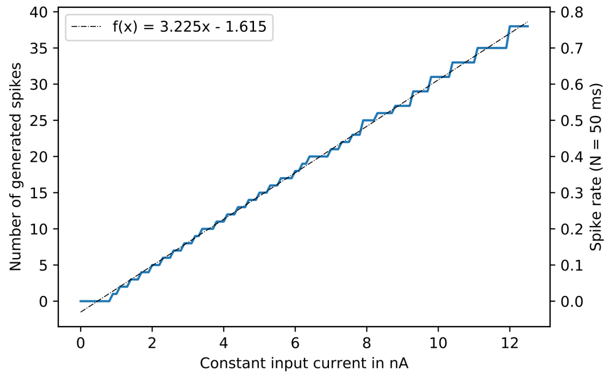

Figure 4.2: Spike count and spike rates of a single neuron from 120 simulations of 50 ms duration with varying input current offset ($[0;12]$). A linear regression ($r^2 = 0.9977$) shows the best-fit linear model.

4.2 Parameter translation

As described in Section 2.1.4, it is necessary to discover the exact linear relationship between ANN activations and SNN activations. This section provides the empirical basis for the input and weight normalisation rates that are used in the remainder of the thesis.

The code for the experiments and visualisations are available in Appendix C.

Figure 4.2 plots the spike count and spike rate against the constant input current, using the neuron parameters from Table 4.1. 120 simulations were performed with offset currents ranging from 0 to 12 nA, using a resolution of 0.1 nA.

The relationship shows that there is an approximate linear correlation, when the input current is kept below 12 and above 1. Outside this range the relationship becomes unstable. Towards 0 it flattens out and produces no spikes. Towards and beyond 12, it begins to resemble a non-differentiable step function. Cropping the 'unstable' parts of the graph away, produces spike-counts in the interval $[1;38]$ and spike rates in the interval $[0.02;0.76]$. In order to allow compositions of populations, as required by the DSL, future populations will have to scale their stimuli to fit the same intervals. Otherwise the assumption about a linear correlation between input stimuli and output rates collapses, and the differentiation becomes imprecise. In turn, this would result in bad prediction rates for the model, because the model cannot properly learn from the backpropagation errors.

To illustrate this point in a deeper network, Figure 4.3 shows three populations —each with a single neuron—chained through synapses to the first population, whose spike rates are seen above. Each datapoint is a separate

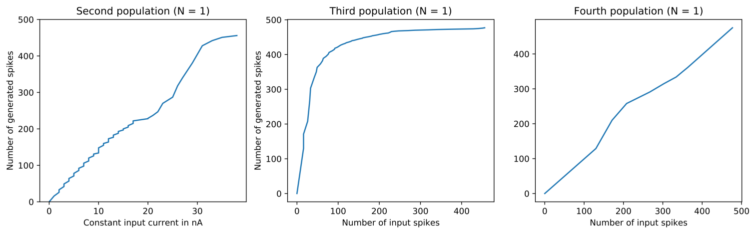

Figure 4.3: Spike counts for a second, third and fourth population in a chained network of single-neuron populations with a constant synaptic weight of 1.

simulation over 50ms with a fixed current offset for the first population. The first population is synaptically connected to the second population, which integrates the spike currents, until that population fires and so on. Without weight normalisation, the correlation is visibly unstable. All weights and biases between the populations are set to a constant value of 1 and 0 respectively.

The correlations are clearly not linear. To avoid this, it is possible to adjust the weights between the populations through an approximation of the weight normalisation scheme from Rueckauer et al. [54], shown in Equation 2.22. The normalisation is based on the maximum activation of the previous layer. Figure 4.2 illustrates that this relation is linear. However, in deeper layers the activation is scaled by the number of previous neurons, since the post-synaptic potentials accumulates. In the case of the NEST and BrainScaleS backends, the post-synaptic potentials are solely depending on the synaptic weights [21, 58]. In other words, the energy of the spikes does not depend on the number of post-synaptic neurons. For the post-synaptic neurons this means that the input potential for a neuron is linearly correlated with the number of presynaptic connections.

A number of experiments have been conducted to find the appropriate scaling values, and, given a population of size N^l and its preceding population of size N^{l-1}, a normalisation value of $w_l = 0.065/N^{l-1}$ has been shown to provide a stable linear approximation of spike rates in a network. Figure 4.4 shows the same populations as in Figure 4.3 with the same constant weights of 1, but with the normalisation term applied.

To prove this in larger networks, the same normalisation term was applied to an MNIST topology of (**`dense`** `100 100` ⦶ **`dense`** `100 10`). Figure 4.5 shows the averaged spike rate for each neuron population, with normalised weights. The figure shows that a near-linear relationship exists, even for larger neuron populations.

For network training in the remainder of the thesis, this normalisation will be applied after the weights have been calculated through regular backpropagation. This separates the weight normalisation from the actual SNN weights, such that the optimisation operates on the idealised linear spike rate model. Listing 4.1 shows the separation of concerns within the densely connected

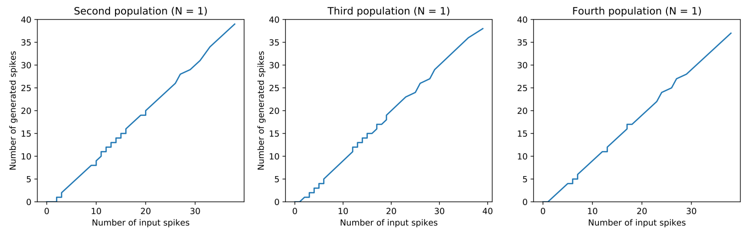

Figure 4.4: Spike counts for the second, third and fourth population in a chained network of single-neuron populations, adjusted for previous neuron activation with the normalisation term $0.065/N^{l-1}$.

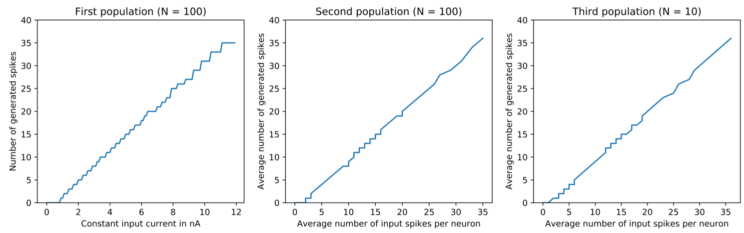

Figure 4.5: Spike count for the first, second and third population in an MNIST network, adjusted for previous neuron activation with the normalisation term $0.065/N^{l-1}$.

spiking layer, in which the raw weights are stored directly, and the normalised weights are set for the neuron projection that connects the input and output populations.

Listing 4.1: Weight normalisation in the spiking dense layer.

```
def set_weights(self, weights):
    self.weights = weights
    normalised = self._normalise_weights(weights)
    self.projection.set(weight=normalised)
```

4.3 Problem sets

Three problem sets will be tested: the NAND ($\neg(A \land B)$) and XOR ($\oplus$) logical gates, as well as the Modified National Institute of Standards and Technology (MNIST) database. The NAND and XOR problems are trivial for ANNs to learn, and are used as a means to test and compare the rudimentary learning capacities of the NEST backend.

The NAND and XOR experiments will be based on the same network topology (**dense** 2 4 ⦶ **dense** 4 2). All backends will execute the experiment with randomly initialised weights. However, the spiking backends will be evaluated a second time with imported weights and biases from the optimised Futhark networks. This is interesting because Futhark is expected to outperform the SNNs, and since the network topology is shared, network parameters can be inserted 1:1. In theory this should improve the initial training of the spiking models and lead to an increased accuracy.

The weights and biases from the optimised Futhark model will only be imported into NEST, which then trains the weights to fit the spiking neuron model.

The MNIST dataset is widely used for training neural networks to classify images of digits between 0 and 9. It is also commonly used for implementation benchmarks [57, 58], with the best networks scoring an error rate of 0.21% [70]. MNIST consists of a collection of 60,000 training images and 10,000 testing images of handwritten digits [71].

To predict the MNIST digits two networks will be constructed. MNIST images contain 784 pixes (28x28), but to avoid too complex simulations it is necessary to limit the network size. The images have been cropped and scaled to 10x10 pixels, such that the initial network layer can be scaled to 100 neurons. The topology for the sequential model is **`dense`** `100 100` ⦶ **`dense`** `100 10`.

To test the parallel structures of the DSL, a second and parallel network will be constructed. The network will resemble the sequential model, but consist of two separate parallel subnetworks (**`dense`** `20 10`), that is merged to produce an output of 20 neurons. The full model is as follows: **`dense`** `100 20` ⦶ `(`**`dense`** `20 10` ⊖ **`dense`** `20 10)` ⦶ **`dense`** `20 10`. The idea of the model is that the two parallel subsystems can learn semantically different tasks, and the final layer will be able to 'choose' which subnetwork to use, based on its weights.

4.4 Experiment method

All the above mentioned experiments are classification tasks, and the labels are encoded as one-hot vectors. To compare the network output with the labels, the argmax value of the network output is taken and converted to a one-hot vector of the same shape as the label data.

To avoid one-off effects such as local minima or (un)fortunate weight initialisation, all experiments have been repeated 10 times. The results reported below are accumulations of the prediction accuracies and errors from the runs.

Weights have been initialised in the models using a normal distribution with a mean of 1 and a standard deviation of 1.

The experiments use a 80/20 training/testing split with a fixed learning rate of 0.1, and the batch size has been set to 64.

To make the experiments as reproducible as possible, they have all been initialised with constant random seeds. Since all randomness in Futhark is based on this seed, all results are constant and standard deviations are effectively 0. This is not the case in PyNN, where the randomness is highly backend-specific. A configuration for setting the initial seed exists (`rng_seed_seeds`)—

and have been set for all experiments—but PyNN does not fully support the randomness configurations in NEST [21].

Chapter 5

Results

This chapter consists of two parts: the first part presents the data of the experiments described above as-is, and the second part discusses the results in the light of the thesis hypotheses. The aim of this chapter is to verify or discard the hypotheses, and will conclude with a summary of the thesis results. This conclusion will be discussed in a broader context in the subsequent chapter.

Details on the system that performed the benchmarks as well as library versions and experiment code is available in Appendix D.

During the experimental phase, namely the evaluation of the parallel model on NEST, an incompatibility of the DSL with the PyNN framework was discovered. During the initialisation of the weights PyNN threw an exception in the merge layer. The error was due to the fact that a single population receives weights from multiple populations, which requires a *view* or a subset of the population. In PyNN this is done using indices, and a consistency check is carried out to verify that the index is within a certain range. However, the parallel layer requires that the indices are configured out of order, resulting in a failed index verification check.

While all the models compile and pass the structural tests of PyNN, the experiments using the Merge layer fail to complete. As a consequence the parallel spiking experiments are unavailable, and are marked as N/A in the result tables below. A complete description of the error trace is available in Appendix E on page 78.

The metric used to compare the performance of the results is the prediction accuracy of the models. The one-hot outputs are compared to the labels, and given N experiments and K true hits, the accuracy is defined as K/N. Thus, an accuracy of 1 is a perfect score, and 0 indicates that none of the labels have been correctly predicted.

5.1 NAND

Both the NAND and XOR experiments are built by compiling the expression **`dense`** `2 4` ⦶ **`dense`** `4 2`. The input data is generated by randomly sam-

Backend	Random weights	Transferred weights
Futhark	1.000	-
NEST	0.370 ± 0.040	0.69 ± 0.216

Table 5.1: Mean accuracies and standard deviations of the NAND experiment.

pling two numbers from the set $\{0, 1\}$. The resulting NAND value was then used as the target label. The networks were injected with a total of 512 data points, corresponding to 8 batches.

Data from the NAND experiments are shown in Table 5.1. Futhark achieves a 100% accuracy rate as expected. However, the rates are below chance level (0.75 for NAND) in NEST, both during the random weight initialisation and while transferring weights from Futhark. It is noteworthy that the accuracy increases significantly when the Futhark weights are injected.

Standard deviations for NEST are small in the case of random weights, meaning that the variance between the experiment accuracies are small. This variance becomes significantly larger when the weights are transferred into NEST from Futhark. This could be explained by the fact that the optimised model in Futhark does not correlate with an equal 'plateau' in NEST. In turn this indicates that the gradient model for NEST is different, although not necessarily worse. The NEST prediction rate, however, proves that the gradient model is incomplete at best.

Figure 5.1 shows a plot of the average backpropagated errors of the model. The gradient errors are interesting to explore because the relative values reveal the average performance of the model (smaller is better), and the absolute values describe how well the model is able to cancel out errors entirely. The plot shows that the errors never approaches zero, but are instead hovering around 45.

In the present experiment the errors decrease initially, but begin to increase after the third batch. The model with imported weights shows a higher accuracy in the beginning, but rapidly decreases to the same level as the model with randomly initialised weights. The declining error rates show that the model is capable of learning to a certain degree. The later reversal means that the optima for the gradient models does not align with the spiking model. This can either be caused by an initially effective, but erroneous gradient model, or an imprecise translation scheme from the raw input data to input currents.

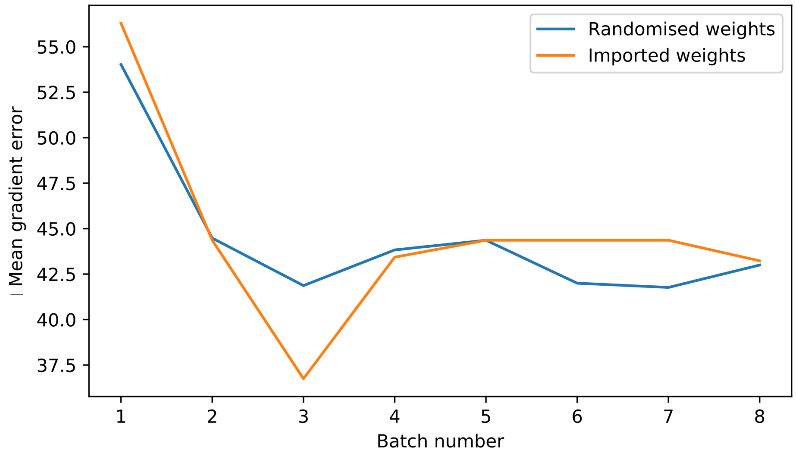

Figure 5.1: Mean gradient error rates for the NAND network simulated in NEST. The rates are produced by averaged over 10 simulations.

5.2 XOR

Tabel 5.2 shows the results of the XOR experiment. The experiment execution and generation of data is similar to the NAND experiment above, except from the label generation that was based on the NAND logical gate.

The chance level of the XOR experiment is at 0.5, lower than for the NAND experiment. Futhark reaches the same accuracy rate of 100%. Notably, NEST performs above chance level, but only with a small margin. The weight improvements when transferring the Futhark parameters are smaller than what was seen in the NAND experiment.

Figure 5.2 illustrates the summed error rates for the XOR experiment, and a similar behaviour as in the NAND experiment can be observed: the error decreases initially, but increases again after the third batch has been processed. The experiment shows a similar learning behaviour as above, although with a higher accuracy. In combination with the imprecise gradient models, this is likely due to the nature of the XOR task. In the case of the NAND task, only 25% of the data trains the second 'on' bit in the one-hot vector. With a constant learning rate, this means that the model is not allowed to move sufficiently in

Backend	Random weights	Transferred weights
Futhark	1.000	-
NEST	0.530 ± 0.038	0.585 ± 0.033

Table 5.2: Mean accuracies and standard deviations of the XOR experiment.

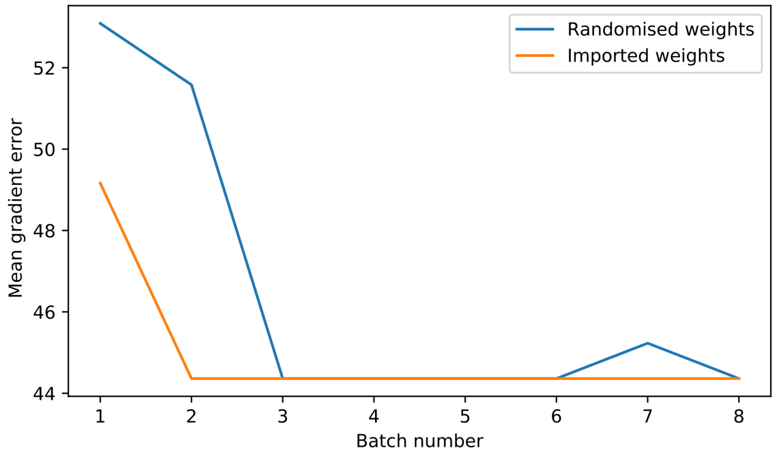

Figure 5.2: Mean gradient error rates for the XOR network simulated in NEST. The rates are produced by averaged over 10 simulations.

the direction of the gradient, before another data points pulls the model in another direction. For the XOR task this is different because 50% of the data trains the second on bit.

The gradient error rates for the XOR network plotted in Figure 5.2 shows same picture as with the NAND error rates: they decline initially, but grows after the third batch has been processed.

5.3 Sequential MNIST

Table 5.3 and Table 5.4 display the mean accuracies of the sequential and parallel MNIST experiments respectively. The state-of-the-art models for MNIST has achieved an accuracy rate of 99.88% [70]. As a comparison, the chance level for MNIST is 0.1 because there are 10 possible output labels.

While the Futhark models was tested on the full 70000 images of the MNIST data, the data for the NEST experiments were reduced to 8192 data points (128 batches) to avoid excessive runtimes. Despite this reduction in size the NEST experiments took well over 6 hours to complete.

Futhark performs equally well in the two experiments with a mean accuracy of 0.710. However, this result is still far below contemporary results. The state-of-the-art models are to a large part based on convolutions instead of non-linear activations, although some models achieve similarly high accuracies with only densely connected layers.

The current experiments are challenged by a) the data compression, b) a constant learning rate and a model that is too small to accurately capture the complexity of the problem domain.

Backend	Random weights	Transferred weights
Futhark	0.710	-
NEST	0.147 ± 0.044	0.098 ± 0.006

Table 5.3: Mean accuracies and standard deviations for the sequential MNIST experiment.

Backend	Random weights	Transferred weights
Futhark	0.710	-
NEST	N/A	N/A

Table 5.4: Mean accuracies and standard deviations for the parallel MNIST experiment.

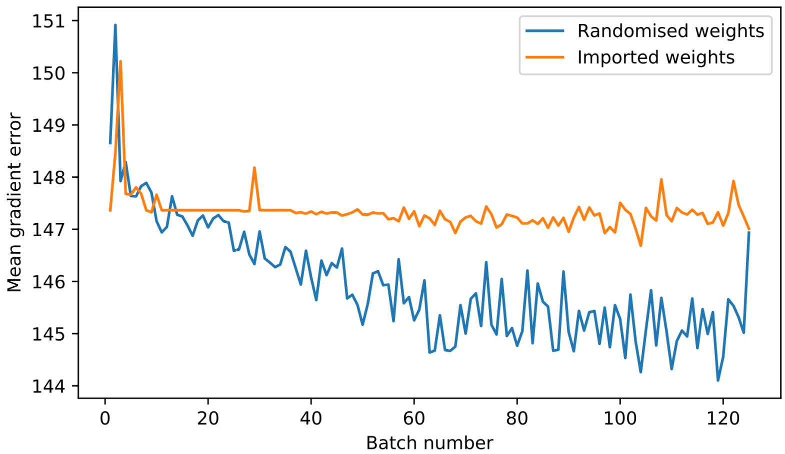

Figure 5.3: Summed gradient error rates for the sequential MNIST network simulated in NEST. The rates are produced following each batch and are averaged over 10 simulations.

The sequential results on NEST performs close to chance level with randomly initialised weights perform close to chance level. The standard deviation is quite small, indicating that the results are consistent across runs. However, the 0.047% is negligible and proves that the gradient model and input translations are incorrect.

It is noteworthy that the model with the transferred rates performs worse than the randomly initialised weights. This is unexpected and questions the assumption that the weights are correctly transferred between the ANN and SNN models, especially because the standard deviations are so small that it cannot be ascribed to chance.

Figure 5.3 shows the summed gradient error rates for the MNIST models. Unlike the above experiments the rates do not improve initially. Instead they worsen shortly in the beginning, again indicating that they are not training towards the correct gradients. The constant high gradient error values aligns with this interpretation by stagnating and thus failing to find better minima.

There is a clear difference between the model with randomised weights and with non-randomised weights. It is counterintuitive that the errors in the model with the imported parameters performs the worst. If this was consistent with the NAND and XOR experiments, the accuracies would be better initially, but trail off after the advantage of the imported weights disappears. One possible explanation is that the imported weight model found a consistent local minima that it could not escape from. It is even feasible that the minima is global in the linear non-spiking gradient model, but that the coding scheme fails to translate the minima into the correct weights.

Looking at the data from the experiments in depth, a large number of 'dead neurons' can be found. 'Dead neurons' are neurons whose weights have decreased to a point where they are not able to fire anymore. This is detrimental to the result because of the argmax classification that brings the network closer to chance level. Because the gradients are calculated based on the ReLU model, such a low neuron activity level is difficult to compensate for, since the activations approaches 0, where the ReLU model is non-differentiable.

The low mean accuracies when performing the NEST experiments suggest that the approximated gradient spiking model is flawed. The visualisations show that the initial models are capable of performing some form of learning, but that the learning is imprecise and quickly looses momentum. This can be attributed to a large number of causes, but in the light of the consistently high backpropagation error rates, this is likely due to the aforementioned flawed gradient model and too many dead neurons.

However, the error rates in the weight transfer models are smaller than, or close to, those in the random weight initialisation models. The weak improvements does not decisively prove that there is a relation between the models in Futhark and the models in NEST, but it does not disprove it.

Regarding learning, the spiking models are showing signs of improvements, albeit inconsistent signs. Both the NAND and the XOR experiments are achieving a consistently lower error rate. The randomised MNIST model shows the same consistent development, but this development is difficult to reason about in the light of the weight transfer model, where the error gradient roughly remains constant.

Chapter 6

Discussion and future work

This thesis sat out to explore SNNs and their future relevance to the field of machine learning. A DSL for neural models was presented, along with two supporting machine learning libraries for the training of second and third generation NNs. To validate that DSL, two hypotheses were put forward and experiments were designed to attempt to falsify these hypotheses. Finally, a theory for the translation of model parameters for SNNs was developed and tested empirically.

Three experiments, each executed on two different backends, were conducted to prove two things: that the DSL Volr can translate into second and third generation neural networks and adapt to a well-known recognition task, using backpropagation learning.

The experimental results prove that the DSL concepts are translatable between the NN paradigms, and that the DSL can generate executable programs that retain the abstract network topologies.

The results further show, that some form of learning was taking place in the experiments with SNNs. However, flaws in the gradient approximation model and the spike rate coding scheme, suggests that the model learns consistently wrong patterns and produces a large quantity of dead neurons. The experimental results do not disprove that training within SNNs is possible, but further adaptations to the gradient and coding models are required.

Table 6.1 concludes the findings of the thesis:

While this thesis focused on the theoretical foundations of the DSL, its

Hypothesis 1	Translation to ANN	Confirmed
Hypothesis 1	Translation to SNN	Confirmed
Hypothesis 2	Learning an MNIST task in Futhark	Confirmed
Hypothesis 2	Learning an MNIST task in NEST	Unconfirmed

Table 6.1: A summary of the thesis findings.

hypotheses were conceived with the assumption that the technical tools for constructing and simulating spiking neural networks were in place. During the experimental phase of the project, this assumption proved to be wrong. For future research, an early benchmark of framework candidates for DSL backends should be part of early initial feasibility studies, to avoid similar obstacles. For this reason hypothesis 2 could not conclusively be confirmed or disproved. The corresponding experiment could not be conducted.

Research within machine learning—surrounding NNs in particular—offers a large number of optimisation techniques, which could be employed to improve the above models. It is common to operate with a momentum in the learning rate, such that larger errors cause larger adaptations [20, 61]. It is also popular to add a layer normalisation scheme to force the layers to adhere to a certain property such as a sigmoid distribution. This could be attempted to avoid the large number of dead neurons, because the layers can be normalised into a distribution that minimises the likelihood of zero signals.

In the context of optimisation, the spike rate models are another possibility for improvement. The present rate models ignore the amplitude, inter-spike intervals as well as sub-threshold activity, and are effectively discarding valuable information. A possible next step can be to explore other coding schemes that allow for more stable and concise gradient models.

To improve training and learning of the models, alternative paradigms can be explored that do not rely on differentiability. One example is evolution learning, in which the search for optima is more time consuming compared to gradient descent, but can be performed without a gradient model.

Another measure to avoid dead neurons is to explore different activation functions. The currently used ReLU function is flawed in two ways: it is not differentiable around zero and it does not penalise values close to zero. Other activation functions, such as the sigmoid function, or a linear term that favours larger weights, could disincentivise dead neurons.

The PyNN interface has proven to be less stable and scaleable than anticipated based on its widespread use and documentation. However, the architecture of the DSL permits generalisation to other backends, and the Haskell compiler is entirely independent from the experiment results. Since the underlying abstract NN model stays constant, it is possible to compile to other targets such as BrainScaleS or SpiNNaker. This could also be used to omit the unstable PyNN interface and compile directly to NEST.

A paper yet to be published by Tavanaei et al. reviews techniques for applying deep learning techniques in SNNs. The authors conclude that ANNs still perform significantly better than SNNs when it comes to recognition tasks, but that the gap is closing. Further research on the thesis could include the studies from Tavanaei et al. and use them to improve the DSL.

The thesis also picked up on the subject of cognitive science, because the field is highly invested in the prospects of the simulation of neurophysiology. Regarding the cognitive REF theory in the context of this thesis, however, conclusions have to be drawn carefully. The composition of neural components with a DSL are an essential first step towards being able to not just construct, but also understand, the NN models and its semantics. It is, however, necessary to analyse far bigger and more complex systems before any connections can be made.

As a final perspective, - are many merits to the models that seamlessly compile between platforms, particularly in the context of neuromorphic computation. Using a better performing model for the parameter translation, ANN models can be compiled directly to neuromorphic hardware. Translating functionally complete logical gates (like the NAND gate) into neural circuits, implies that any logical circuit can be translated. Tasks like recognition models for faces or objects, as well as algorithmic problems like sorting, are just a few out many possible applications. In short, if the accuracies can be improved and translated into neuromorphic hardware, it would accelerate the current computational capacities of the von Neumann machine architectures by a factor of at least 100.

Appendix A

Transfer functions for spiking neurons

This appendix walks through the proof for Equation 2.11, as given by Rueckauer et al. in [54].

Given the stepwise activation function from Equation 2.1, the neuron can be said to spike with [13, 54, p. 3]:

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

In turn the occurrence of a spike for a neuron i at timestep t can be calculated by the integration of input current at every simulation time step, where v_i^l is the membrane potential for the neuron [54, p. 3]:

$$\Theta_{t,i}^l = \Theta(v_i^l(t-1) + \zeta_i^l(t) - V_{thr}), \tag{A.2}$$

Recalling that a neuron i at layer l receives post-synaptic impulses from j neurons from layer $l-1$, the input current ζ_i^l for neuron i at layer l can be seen as the linear equation [54, p. 3]:

$$\zeta_i^l(t) = V_{thr} \left(\sum_{j=1}^{N^{l-1}} w_{ij}^l \Theta_{t,j}^{l-1} + b_i^l \right), \tag{A.3}$$

where V_{thr} is the neuron threshold.

Neurons integrate $\zeta_i^l(t)$ until the threshold V_{thr} is reached, where a spike is emitted and the membrane potential is reset to 0 (see Figure 2.4). The membrane current $v_i^l(t)$ can then be modelled as

$$v_i^l(t) = \left(V_i^l(t-1) + \zeta_i^l(t)\right)\left(1 - \Theta_{t,i}^l\right) \tag{A.4}$$

Assuming that the input current is above zero ($\zeta_i^1 > 0$) and that it remains constant through time, there will be a constant number of timesteps n_i^1 between spikes in the neuron i, and the neuron threshold will always be exceeded by the same amount:

$$\epsilon_i^l = v_i^1(n_i^1) - V_{thr} = n_i^1 \cdot \zeta_i^1 - V_{thr} \tag{A.5}$$

Assuming the same constant input current ζ such that $\sum_{t'}^{t} 1 = t/\Delta t$, and realising that the number of spikes N in a simulation of duration t is $N(t) = \sum_{t'=1}^{t} \Theta_{t'}$, the membrane potential can be obtained by summing over the simulation duration t [54]:

$$\begin{aligned}\sum_{t'}^{t} v(t') &= \sum_{t'=1}^{t} v(t'-1)(1-\theta_{l'}) + (1-\Theta_{t'}) \\ &= \sum_{t'=1}^{t} v(t'-1)(1-\theta_{l'}) + \zeta(\frac{t}{\Delta t} - n)\end{aligned} \tag{A.6}$$

The layer and neuron indices are omitted for clarity.

By further rearranging Equation A.6 and defining the maximum firing rate as $r_{max} = 1/\Delta t$, the average firing rate N/t can now be calculated by dividing with the simulation time [54]:

$$\begin{aligned}\frac{1}{\zeta t}\sum_{t'}^{t} v(t') &= \frac{1}{\zeta t}\sum_{t'=1}^{t} v(t'-1)(1-\Theta_{l'}) + \zeta(\frac{t}{\Delta t} - N) \\ \frac{1}{\zeta t}\sum_{t'}^{t} v(t') &= \frac{1}{\Delta t} - \frac{N}{t} + \frac{1}{\zeta t}\sum_{t'=1}^{t} v(t'-1)(1-\Theta_{l'}) \\ \frac{1}{\zeta t}\sum_{t'}^{t} v(t') + \frac{N}{t} &= r_{max} + \frac{1}{\zeta t}\sum_{t'=1}^{t} v(t'-1)(1-\Theta_{l'}) \\ \frac{N}{t} &= r_{max} + \frac{1}{\zeta t}\sum_{t'=1}^{t} (v(t'-1)(1-\Theta_{l'}) - v(t')) \\ r = \frac{N}{t} &= r_{max} - \frac{1}{\zeta t}\sum_{t'=1}^{t} (v(t') - v(t'-1)(1-\Theta_{l'}))\end{aligned} \tag{A.7}$$

Since the input current is constant, the value of the membrane potential before a spike is always the same, and is always an integer multiple of the input ζ. Defining $n \in \mathbb{N}$ as the number of simulation steps needed to cross the threshold V_{thr}, then

$$\frac{1}{\zeta t}\sum_{t'}^{t} v(t'-1)\Theta_{t'} = \frac{1}{\zeta t}(n-1)\zeta N = r(n-1) \tag{A.8}$$

Realising that

$$\sum_{t'=1}^{t} v(t') - v(t'-1) = v(t) - v(0) \tag{A.9}$$

Equation A.7 simplifies to:

$$
\begin{aligned}
r &= r_{max} - \frac{1}{\zeta t}\sum_{t'=1}^{t}\left(v(t') - v(t'-1)(1-\Theta_{l'})\right) \\
&= r_{max} - \frac{v(t)-v(0)}{\zeta t} - \frac{1}{\zeta t}\sum_{t'=1}^{t} v(t'-1)\Theta_{l'} \\
&= r_{max} - \frac{v(t)-v(0)}{\zeta t} - r(n-1) \\
&= r_{max} - \frac{v(t)-v(0)}{\zeta t} - rn - r \\
r &= \frac{1}{n}\left(r_{max} - \frac{v(t)-v(0)}{\zeta t}\right)
\end{aligned} \tag{A.10}
$$

Finally, the residual charge $\epsilon \in \mathbb{R}$ is defined as the surplus charge at the time of a spike:

$$
\epsilon = n\zeta - V_{thr} \tag{A.11}
$$

and remembering that the first layer at constant input $\zeta^l = V_{thr}x^1$, the average spike rate for that layer can now be defined with re-introduced neuron and layer indices:

$$
\begin{aligned}
r_i^1(t) &= \frac{1}{n_i^1}\left(r_{max} - \frac{v_i^1(t) - v_i^1(0)}{\zeta t}\right) \\
&= \frac{1}{(\epsilon_i^1 + V_{thr})/\zeta}\left(r_{max} - \frac{v_i^1(t) - v_i^1(0)}{\zeta t}\right) \\
&= \frac{\zeta}{\epsilon_i^1 + V_{thr}}\left(r_{max} - \frac{v_i^1(t) - v_i^1(0)}{\zeta t}\right) \\
&= \frac{V_{thr}x_i^1}{\epsilon_i^1 + V_{thr}} r_{max} - \left(\frac{\zeta}{\epsilon_i^1 + V_{thr}} \frac{v_i^1(t) - v_i^1(0)}{\zeta t}\right) \\
&= x_i^1 r_{max}\frac{V_{thr}}{V_{thr} + \epsilon_i^1} - \frac{v_i^1(t)}{t(V_{thr} + \epsilon_i^1)}
\end{aligned} \tag{A.12}
$$

Appendix B

Volr implementation code

The code in this appendix is available for completeness (and newer versions) at the website `https://github.com/`. This includes the tests and scripts used to execute the experiments in the thesis. The code is split into four different locations:

- The Volr compiler in Haskell:
  `https://github.com/volr/compiler`
- The Python neural network library:
  `https://github.com/volr/volrpynn`
- The Futhark neural network library:
  `https://github.com/jegp/deep_learning`
- The thesis itself, including experimental setups:
  `https://github.com/jegp/thesis`

B.1 Haskell compiler

B.1.1 Evaluator.hs

```
module Volr.Evaluator where

import Control.Applicative
import Control.Monad.Except
import Control.Monad.State.Lazy

import Data.Either
import qualified Data.Map.Strict as Map

import Volr.AST

type Error = String

data TermState = TermState { types :: Context, store :: Store }
  deriving (Eq, Show)
type EvalState = ExceptT Error (State TermState)

emptyState = TermState Map.empty Map.empty

eval :: Term -> Either Error Term
```

```haskell
eval term = evalState (runExceptT (evalTyped term)) emptyState
  where
    evalTyped term = do
      untyped <- eval' term
      typeOf untyped *> return untyped

eval' :: Term -> EvalState Term
eval' term =
  case term of
    TmNet n m -> return $ TmNet n m
    TmSeq t1 t2 -> do
        t1' <- eval' t1
        t2' <- eval' t2
        return $ TmSeq t1' t2'
    TmPar t1 t2 -> do
        t1' <- eval' t1
        t2' <- eval' t2
        return $ TmPar t1' t2'
    TmRef n -> do
        state <- get
        case store state Map.!? n of
          Nothing -> throwError $ "Could not find reference of name " ++ n
          Just m -> return m
    TmLet name t1 t2 -> do
        state <- get
        t1' <- eval' t1
        put $ state { store = Map.insert name t1' (store state) }
        t2' <- eval' t2
        put $ state
        return t2'

-- | Tests whether a given term is a value
isVal :: Term -> Bool
isVal (TmNet _ _) = True
isVal _ = False

typeOf :: Term -> EvalState Type
typeOf term =
  case term of
    TmNet n m -> return $ TyNetwork n m
    TmSeq t1 t2 -> do
      leftOut <- sizeRight t1
      rightIn <- sizeLeft t2
      if leftOut == rightIn then do
        leftIn <- sizeLeft t1
        rightOut <- sizeRight t2
        return $ TyNetwork leftIn rightOut
      else
        throwError $ "Type error: Incompatible network sizes. Output " ++
                     (show leftOut) ++ " should be equal to input " ++ (show
                          rightIn)
    TmPar t1 t2 -> do
      left1 <- sizeLeft t1
      left2 <- sizeLeft t2
      if left1 == left2 then do
        right1 <- sizeRight t1
        right2 <- sizeRight t2
        return $ TyNetwork left1 (right1 + right2)
      else
        throwError $ "Type error: Parallel networks must share input sizes, got "
             ++
                     (show left1) ++ " and " ++ (show left2)
    TmLet name t1 t2 -> do
      state <- get
      t1' <- eval' t1
      let innerState = state { store = Map.insert name t1' (store state) }
      evalState (return $ typeOf t2) innerState
```

```haskell

sizeLeft :: Term -> EvalState Int
sizeLeft term =
  case term of
    TmNet m _ -> return m
    TmSeq t1 t2 -> sizeLeft t1
    TmPar t1 t2 -> sizeLeft t1
    TmRef n -> do
      state <- get
      case store state Map.!? n of
        Nothing -> throwError $ "Unknown reference " ++ n
        Just e -> sizeLeft e
    _ -> throwError $ "Cannot extract size from term " ++ (show term)

sizeRight :: Term -> EvalState Int
sizeRight term =
  case term of
    TmNet _ m -> return m
    TmSeq t1 t2 -> sizeRight t2
    TmPar t1 t2 -> (+) <$> sizeRight t1 <*> sizeRight t2
    TmRef n -> do
      state <- get
      case store state Map.!? n of
        Nothing -> throwError $ "Unknown reference " ++ n
        Just e -> sizeRight e
    _ -> throwError $ "Cannot extract size from term " ++ (show term)
```

B.1.2 EvaluatorSpec.hs

```haskell
module Volr.EvaluatorSpec (main, spec) where

import Control.Monad.Except
import Control.Monad.State.Lazy
import Data.Either
import qualified Data.Map.Strict as Map

import Test.Hspec

import Volr.AST
import Volr.Evaluator

main :: IO()
main = hspec spec

spec :: Spec
spec = do
  describe "The evaluator" $ do
    it "can evaluate sequential connection of two networks" $ do
      let e = TmSeq (TmNet 1 1) (TmNet 1 1)
      eval e `shouldBe` Right e
    it "can evaluate parallel connection of two networks" $ do
      let e = TmPar (TmNet 1 1) (TmNet 1 1)
      eval e `shouldBe` Right e
    it "can connect a network to a parallel network" $ do
      let e = TmSeq (TmNet 1 2) (TmPar (TmNet 2 3) (TmNet 2 4))
      eval e `shouldBe` Right e
    it "can fail to connect a network to a parallel network with malformed input size" $ do
      let e = TmSeq (TmNet 1 2) (TmPar (TmNet 1 2) (TmNet 2 4))
      eval e `shouldSatisfy` isLeft
    it "can connect a parallel network to a network" $ do
      let e = TmSeq (TmPar (TmNet 2 3) (TmNet 2 4)) (TmNet 7 5)
      eval e `shouldBe` Right e
    it "can fail to connect a parallel network to a network with malformed input size" $ do
```

```
      let e = TmSeq (TmPar (TmNet 2 3) (TmNet 2 4)) (TmNet 8 5)
      eval e `shouldSatisfy` isLeft
    it "can evaluate a let binding" $ do
      let e = TmLet "x" (TmNet 1 2) (TmSeq (TmRef "x") (TmNet 2 1))
      eval e `shouldBe` Right (TmSeq (TmNet 1 2) (TmNet 2 1))
    it "can evaluate a let binding with a reference" $ do
      let e = TmLet "x" (TmNet 1 1) (TmRef "x")
      eval e `shouldBe` Right (TmNet 1 1)
    it "can fail to evaluate a let binding with a missing reference" $ do
      eval (TmRef "x") `shouldSatisfy` (isLeft)
    it "can evaluate a let binding and discard the inner context" $ do
      let e = TmLet "x" (TmNet 1 1) (TmRef "x")
      let s = execState (runExceptT $ eval' e) emptyState
      s `shouldBe` emptyState
    it "can fail to evaluate two sequential connections with unmatched sizes" $
         do
      let e = TmSeq (TmNet 1 2) (TmNet 1 1)
      eval e `shouldSatisfy` isLeft
```

B.1.3 Parser.hs

```
module Volr.Parser where

import Control.Applicative hiding (many, some)
import Control.Monad.State.Lazy

import Data.Bifunctor
import Data.Functor
import qualified Data.Map as Map
import Data.Maybe (isJust)

import Text.Megaparsec
import qualified Text.Megaparsec.Char as Char
import qualified Text.Megaparsec.Char.Lexer as Lexer
import qualified Text.Megaparsec.Pos as Pos

import Volr.AST
import Volr.Evaluator

type SyntaxError = ParseError (Token String) String
type Parser = Parsec String String

parse :: String -> Either String Term
parse code =
  first show (runParser parseTerm "" code) >>= eval

parseTerm :: Parser Term
parseTerm = (lexeme $ choice
  [ TmNet <$> (symbol "Net" *> integer) <*> integer
  , TmPar <$> (symbol "Par" *> (parens parseTerm)) <*> (parens parseTerm)
  , TmSeq <$> (symbol "Seq" *> (parens parseTerm)) <*> (parens parseTerm)
  , TmRef <$> (symbol "Ref" *> (name))
  , TmLet <$> (symbol "Let" *> (name)) <*> (symbol "=" *> parseTerm)
                           <*> (symbol "in" *> parseTerm)
  ]) <* (optional eof)

integer :: Parser Int
integer = lexeme Lexer.decimal

parens :: Parser a -> Parser a
parens = between (symbol "(") (symbol ")")

name :: Parser String
name = lexeme (many Char.alphaNumChar)

```

```
symbol :: String -> Parser String
symbol = Lexer.symbol sc

lexeme :: Parser a -> Parser a
lexeme = Lexer.lexeme sc

sc :: Parser ()
sc = Lexer.space Char.space1 lineCmnt blockCmnt
  where
    lineCmnt  = Lexer.skipLineComment "//"
    blockCmnt = Lexer.skipBlockComment "/*" "*/"
```

B.2 Futhark library

B.2.1 neural_network.fut

```
import "nn_types"
import "activation_funcs"

module type network = {

  type t

  --- Combines two networks into one
  val connect_layers 'w1 'w2 'i1 'o1 'o2 'c1 'c2 'e1 'e2 'e22:
                    NN i1 w1 o1 c1 e22 e1 (apply_grad t) ->
                    NN o1 w2 o2 c2 e2 e22 (apply_grad t) ->
                    NN i1 (w1, w2) (o2) (c1,c2) (e2) (e1) (apply_grad t)

  -- Runs two networks in parallel
  val connect_parallel 'w1 'w2 'i1 'i2 'o1 'o2 'c1 'c2 'ei1 'eo1 'ei2 'eo2:
                    NN i1 w1 o1 c1 ei1 eo1 (apply_grad t) ->
                    NN i2 w2 o2 c2 ei2 eo2 (apply_grad t) ->
                    NN (i1, i2) (w1, w2) (o1, o2) (c1, c2) (ei1, ei2) (eo1, eo2
                         ) (apply_grad t)

  --- Performs predictions on data set given a network,
  --- input data and activation func
  val predict 'w 'g 'i 'e1 'e2 '^u 'o  : NN ([]i) (w) ([]o) g e1 e2 u ->
                                         []i ->
                                         activation_func o ->
                                         []o

  --- Calculates the accuracy given a network, input,
  --- labels and activation_func
  val accuracy 'w 'g 'e1 'e2 'i '^u 'o : NN ([]i)  w  ([]o) g e1 e2 u ->
                                         []i ->
                                         []o ->
                                         activation_func o ->
                                         (o -> i32) ->
                                         t

  --- Calculates the absolute loss given a network, input, labels,
  --- a loss function and classifier aka activation func
  val loss 'w 'g 'e1 'e2 '^u 'i 'o : NN ([]i) w ([]o) g e1 e2 u ->
                                     []i ->
                                     []o ->
                                     loss_func o t ->
                                     activation_func o ->
                                     t

  --- activation function wrappers
  val identity : activation_func ([]t)
  val sigmoid  : activation_func ([]t)
```

```
  val relu     : activation_func ([]t)
  val tanh     : activation_func ([]t)
  val softmax  : activation_func ([]t)

  --- helper functions for calculating accuracy
  val argmax : []t -> i32
  val argmin : []t -> i32

}

module neural_network (R:real): network with t = R.t = {

  type t = R.t

  module act_funcs = activation_funcs R


  let connect_layers 'w1 'w2 'i1 'o1 'o2 'c1 'c2 'e1 'e2 'e
                     ({forward=f1, backward=b1,
                        weights=ws1}: NN i1 w1 o1 c1 e e1 (apply_grad t))
                     ({forward=f2, backward=b2,
                        weights=ws2}: NN o1 w2 o2 c2 e2 e (apply_grad t))
                      : NN i1 (w1,w2) (o2) (c1,c2) (e2) (e1) (apply_grad t) =

    {forward = \(is_training) (w1, w2) (input) ->
                          let (c1, res)  = f1 is_training w1 input
                          let (c2, res2) = f2 is_training w2 res
                          in ((c1, c2), res2),
     backward = \(_) u (w1,w2) (c1,c2) (error) ->
                          let (err2, w2') = b2 false u w2 c2 error
                          let (err1, w1') = b1 true u w1 c1 err2
                          in (err1, (w1', w2')),
     weights = (ws1, ws2)}

  let connect_parallel 'w1 'w2 'i1 'i2 'o1 'o2 'c1 'c2 'ei1 'eo1 'ei2 'eo2
                     ({forward=f1, backward=b1,
                        weights=ws1}: NN i1 w1 o1 c1 ei1 eo1 (apply_grad t))
                     ({forward=f2, backward=b2,
                        weights=ws2}: NN i2 w2 o2 c2 ei2 eo2 (apply_grad t))
                      : NN (i1, i2) (w1, w2) (o1, o2) (c1, c2) (ei1, ei2) (eo1,
                           eo2) (apply_grad t) =

    {forward = \(is_training) (w1, w2) (i1, i2) ->
               let (c1, res1) = f1 is_training w1 i1
               let (c2, res2) = f2 is_training w2 i2
               in ((c1, c2), (res1, res2)),
     backward = \(_) u (w1, w2) (c1, c2) (e1, e2) ->
               let (err0, w0') = b1 false u w1 c1 e1
               let (err1, w1') = b2 false u w2 c2 e2
               in ((err0, err1), (w0', w1')),
     weights = (ws1, ws2)}

  let predict  'i 'w 'g 'e1 'e2 'u 'o
               ({forward=f, backward=_, weights=w}:NN ([]i) w ([]o) g e1 e2 u)
               (input:[]i)
               ({f=class, fd = _}:activation_func o)  =

    let (_, output) = f false w input
    in map (\o -> class o) output


  let accuracy [d] 'w 'g 'e1 'e2 'u 'i 'o (nn:NN ([]i) w ([]o) g e1 e2 u)
                                          (input:[d]i)
                                          (labels:[d]o)
                                          (classification:activation_func o)
                                          (f: o -> i32) : t =

```

```
    let predictions  = predict nn input classification
    let argmaxs      = map2 (\x y -> (f x,f y)) labels predictions
    let total        = reduce (+) 0 (map (\(x,y) -> i32.bool (x == y))
                                         argmaxs)
    in R.(i32 total / i32 d)


  let loss [d] 'w 'g 'e1 'e2 'u 'i 'o (nn:NN ([]i) w ([]o) g e1 e2 u)
                                      (input:[d]i)
                                      (labels:[d]o)
                                      ({f = loss, fd = _}: loss_func o t)
                                      (classification:activation_func o) =

    let predictions = predict nn input classification
    let losses      = map2 (\p l -> loss p l) predictions labels
    in R.sum losses

  --- Breaks if two or more values have max values?
  --- Question is which index should be chosen then?
  let argmax [n] (X:[n]t) : i32 =
    reduce (\n i -> if unsafe R.(X[n] > X[i]) then n else i) 0 (iota n)

  let argmin [n] (X:[n]t) : i32 =
    reduce (\n i -> if unsafe R.(X[n] < X[i]) then n else i) 0 (iota n)

  --- activation function wrappers
  let identity = act_funcs.Identity_1d
  let sigmoid  = act_funcs.Sigmoid_1d
  let relu     = act_funcs.Relu_1d
  let tanh     = act_funcs.Tanh_1d
  let softmax  = act_funcs.Softmax_1d

}
```

B.2.2 replicate.fut

```
import "layer_type"
import "../nn_types"
import "../util"
import "../weight_init"
import "../../../diku-dk/linalg/linalg"

-- | Split input into several layers
module replicate (R:real) : layer_type with t = R.t
                                  with input_params = (i32)
                                  with activations  = activation_func ([]R.t)
                                  with input        = arr2d R.t
                                  with weights      = (std_weights R.t,
                                       std_weights R.t)
                                  with output       = tup2d R.t
                                  with cache        = (tup2d R.t, tup2d R.t)
                                  with error_in     = tup2d R.t
                                  with error_out    = arr2d R.t = {

  type t             = R.t
  type input         = arr2d t
  type weights       = (std_weights t, std_weights t)
  type output        = tup2d t
  type cache         = (tup2d t, tup2d t)
  type error_in      = tup2d t
  type error_out     = arr2d t
  type b_output      = (error_out, weights)

  type input_params = (i32)
  type activations  = activation_func ([]t)
```

```

type replicate_nn = NN input weights output
                    cache error_in error_out
                    ((std_weights t) -> (std_weights t) -> (std_weights t))

module lalg   = mk_linalg R
module util   = utility R
module w_init = weight_initializer R

let empty_cache : (arr2d t, arr2d t) = ([[]],[[]])
let empty_error : error_out = [[]]

-- Forward propagation
let forward (act:[]t -> []t)
             (training:bool)
             ((t1, t2): weights)
            (input:input) : (cache, output) =
  let f ((w, b): std_weights t): (tup2d t, arr2d t) =
    let res      = lalg.matmul w (transpose input)
    let res_bias = transpose (map2 (\xr b' -> map (\x -> (R.(x + b'))) xr) res b)
    let res_act  = map (\x -> act x) (res_bias)
    let cache    = if training then (input, res_bias) else empty_cache
    in (cache, res_act)
  let (c1, r1) = f t1
  let (c2, r2) = f t2
  in ((c1, c2), (r1, r2))

-- Backward propagation
let backward (act: []t -> []t)
             (first_layer: bool)
             (apply_grads: apply_grad t)
             ((w1, w2): weights)
             ((c1, c2): cache)
             ((e1, e2): error_in) : b_output =
  let b ((w, b): std_weights t) ((input, inp_w_bias): tup2d t)
       (error: arr2d t) : (arr2d t, std_weights t) =
    let deriv    = (map (\x -> act x) inp_w_bias)
    let delta    = transpose (util.hadamard_prod_2d error deriv)
    let w_grad   = lalg.matmul delta input
    let b_grad   = map (R.sum) delta
    let (w', b') : std_weights t = apply_grads (w,b) (w_grad, b_grad)

    --- Calc error to backprop to previous layer
    let error' : arr2d t =
      if first_layer then
       empty_error
      else
       transpose (lalg.matmul (transpose w) delta)
    in (error', (w', b'))

  let (error1, w1) = b w1 c1 e1
  let (error2, w2) = b w2 c2 e2

  let zero = R.from_fraction 0 1
  let fact = (R.from_fraction 1 2)
  let average_sum_matrix [l][m][n] (tensor: [l][m][n]t) : arr2d t=
    util.scale_matrix (reduce util.add_matrix (replicate m (replicate n zero)) tensor) fact

  in (average_sum_matrix [error1, error2], (w1, w2))

let init (m:input_params) (act:activations) (seed:i32) : replicate_nn =
  let w = w_init.gen_random_array_2d_xavier_uni (m,m) seed
  let b = map (\_ -> R.(i32 0)) (0..<m)
  in {forward  = forward act.f,
      backward = backward act.fd,
```

```
        weights  = ((w, b), (w, b))}

}
```

B.2.3 merge.fut

```
import "layer_type"
import "../nn_types"
import "../util"
import "../weight_init"
import "../../../diku-dk/linalg/linalg"


-- | Merges an array of layers
module merge (R:real) : layer_type with t = R.t
                                   with input_params = (i32, i32)
                                   with activations  = activation_func ([]R.t)
                                   with input        = tup2d R.t
                                   with weights      = ()
                                   with output       = arr2d R.t
                                   with cache        = ()
                                   with error_in     = arr2d R.t
                                   with error_out    = tup2d R.t = {

  type t            = R.t
  type input        = tup2d t
  type weights      = ()
  type output       = arr2d t
  type cache        = ()
  type error_in     = arr2d t
  type error_out    = tup2d t
  type b_output     = (error_out, weights)

  type input_params = (i32, i32)
  type activations  = activation_func ([]t)

  type merge_tp = NN input weights output
                     cache error_in error_out (apply_grad t)

  module lalg   = mk_linalg R
  module util   = utility R
  module w_init = weight_initializer R

  -- Forward propagation
  let forward  (_:[]t -> []t)
               (_:bool)
               (_: weights)
               ((i1, i2):input) : (cache, output) =
    ((), map2 concat i1 i2)

  -- Backward propagation
  let backward (_:[]t -> []t) (l1_sz:i32)
               (_:bool)
               (_:apply_grad t)
               (_:weights)
               (_:cache)
               (error_concat:error_in) : b_output =
    (unzip (map (split l1_sz) error_concat), ())

  let init ((l, _):input_params) (act:activations) (_:i32) : merge_tp =
    {forward  = forward act.f,
     backward = backward act.fd l,
     weights  = ()}

}
```

B.3 Python library

B.3.1 layer.py

```python
"""
The layers of VolrPyNN which must all define a method for a backward-pass
through the layers (to update the layer weights), as well as getting, setting and
storing weights.
"""
import abc
import numpy as np
import volrpynn as v
from volrpynn.util import get_pynn as pynn

class Layer():
    """A neural network layer with a PyNN-backed neural network population and a backwards
       weight-update function, based on existing spikes"""

    def __init__(self, pop_in, pop_out, gradient_model,
             decoder=v.spike_count_linear):
        """
        Initialises a densely connected layer between two populations output
        Args:
        pop_in -- The input population
        pop_out -- The output population
        gradient_model -- An ActivationFunction that calculates the neuron gradients
                          given the current spikes and errors from this layer
        decoder -- A function that can code a list of SpikeTrains into a numeric
                   numpy array
        """
        self.pop_in = pop_in
        self.pop_out = pop_out
        self.output = None
        self.weights = None

        # Store gradient model
        if not isinstance(gradient_model, v.ActivationFunction):
            raise ValueError("gradient_model must be an activation function")

        self.gradient_model = gradient_model

        # Store decoder
        assert callable(decoder), "spike decoder must be a function"
        self.decoder = decoder

    @staticmethod
    def _is_tuple(data, name, allow_none=False):
        """Tests that the given data is a tuple. If not, raise an exception
        using 'name'"""
        if (not data and not allow_none) and \
           (not isinstance(data, (tuple, list)) or len(data) != 2):
            raise ValueError(name + " must be tuple of length two")

    @abc.abstractmethod
    def backward(self, error, optimiser):
        """Performs backwards optimisation based on the given error and
        activation derivative"""

    @abc.abstractmethod
    def get_biases(self):
        """Returns the layer biases, or a list of zeros of the same shape as the
        output layer if the layer does not have biases"""

    @abc.abstractmethod
```

```python
    def get_output(self):
        """Returns a numpy array of the decoded output"""

    def get_weights(self):
        """Returns the weights as a matrix of size (input, output)"""
        return self.weights

    def reset_cache(self):
        """Resets the cached inputs and outputs for batch gradients"""
        self.input_cache = []
        return self

    def restore_weights(self):
        """Restores the current weights of the layer"""
        self.set_weights(self.get_weights())
        return self.weights

    @abc.abstractmethod
    def set_biases(self, biases):
        """Sets the biases of the network layer"""

    @abc.abstractmethod
    def set_weights(self, weights):
        """Sets the weights of the network layer"""

    @abc.abstractmethod
    def store_spikes(self):
        """Stores the spikes of the current run"""

class Decode(Layer):
    """A layer that only decodes the spike trains without any activation passes"""

    def __init__(self, pop_in, decoder=v.spike_softmax):
        super(Decode, self).__init__(pop_in, None, v.UnitActivation(), decoder)
        self.weights = np.ones(pop_in.size)

    def backward(self, error, optimiser):
        return error

    def get_biases(self):
        return np.zeros((self.pop_in.size))

    def get_output(self):
        return self.decoder(self.output)

    def set_biases(self, biases):
        return biases

    def set_weights(self, weights):
        return self.weights

    def store_spikes(self):
        self.output = np.array(self.pop_in.getSpikes().segments[-1].spiketrains)
        return self

class Dense(Layer):
    """A densely connected neural layer between two populations,
       creating a PyNN all-to-all connection (projection) between the
       populations."""

    def __init__(self, pop_in, pop_out, gradient_model=v.ReLU(), weights=None,
                 biases=0, decoder=v.spike_count_linear,
                 translation=v.LinearTranslation(), projection_pos=None,
                 projection_neg=None):
        """
        Initialises a densely connected layer between two populations
```

```python
        """
        super(Dense, self).__init__(pop_in, pop_out, gradient_model, decoder)

        self.input = None
        self.input_cache = []
        self.translation = translation

        # Create a projection between the input and output populations
        if not projection_pos:
            connector = pynn().AllToAllConnector(allow_self_connections=False)
            projection_pos = pynn().Projection(pop_in, pop_out, connector,
                    receptor_type='excitatory')
            projection_neg = pynn().Projection(pop_in, pop_out, connector,
                    receptor_type='inhibitory')
        self.projection_pos = projection_pos
        self.projection_neg = projection_neg

        # Prepare spike recordings
        self.pop_in.record('spikes')
        self.pop_out.record('spikes')

        # Assign given weights or default to a normal distribution
        if weights is not None:
            self.set_weights(weights)
        else:
            random_weights = np.random.normal(0, 1, (pop_in.size, pop_out.size))
            self.set_weights(random_weights)

        if isinstance(biases, np.ndarray):
            self.biases = biases
        elif biases:
            self.biases = np.repeat(biases, pop_out.size)
        else:
            self.biases = np.zeros(pop_out.size)

    def backward(self, error, optimiser):
        """Backward pass in the dense layer

        Args:
        error -- The error in the output from this layer as a numpy array
        optimiser -- The optimiser that calculates the new layer weights, given
                     the current weights and the gradient deltas

        Returns:
        A tuple of the cached spikes from the first (input) layer and the errors
        """
        assert callable(optimiser), "Optimiser must be callable"

        if len(self.input_cache) == 0:
            raise RuntimeError("No input data found. Please simulate the model" +
                               " before doing a backward pass")

        # Calculate activations for output layer
        input_decoded = np.array(self.input_cache)
        output_activations = np.matmul(input_decoded, self.weights)

        # Calculate output gradients and layer delta
        output_gradients = self.gradient_model.prime(output_activations + self.
            biases)
        delta = np.multiply(error, output_gradients)

        # Calculate layer backprop and weights, bias updates
        backprop = np.matmul(delta, self.weights.T)
        weights_delta = np.matmul(input_decoded.T, delta)
        (new_weights, new_biases) = optimiser(self.weights, weights_delta,
                                              self.biases, error.sum(axis=0))

```

```
        # NEST cannot handle too large weight values, so this guard
        # ensures that the simulation keeps running, despite large weights
        new_weights = np.nan_to_num(new_weights)
        new_weights[new_weights > 100] = 100.0
        new_weights[new_weights < -100] = -100.0

        self.set_weights(new_weights)
        self.biases = new_biases

        # Return errors changes in backwards layer
        return backprop

    def get_biases(self):
        return self.biases

    def get_output(self):
        return self.decoder(self.output)

    def get_weights_normalised(self):
        return self.projection.get('weight', format='array')

    def set_biases(self, biases):
        self.biases = biases

    def set_weights(self, weights):
        if type(weights) == int:
            weights = np.zeros((self.pop_in.size, self.pop_out.size)) + weights
        self.weights = weights
        normalised = self.translation.weights(weights, self.pop_in.size)
        positive = normalised.copy()
        positive[positive < 0] = 0
        negative = normalised.copy()
        negative[negative > 0] = 0
        self.projection_pos.set(weight=positive)
        self.projection_neg.set(weight=negative * -1)

    def store_spikes(self):
        segments_in = self.projection_pos.pre.get_data('spikes').segments
        self.input = np.array(segments_in[-1].spiketrains)
        self.input_cache.append(self.decoder(self.input))
        segments_out = self.projection_pos.post.get_data('spikes').segments
        self.output = np.array(segments_out[-1].spiketrains)
        return self

class Merge(Layer):
    """A merge layer that takes a tuple of input layers and
    uniforms them into a single output population by connecting them densely.
    In practice this happens by simply forwarding the spikes to another
    population view. No data processing is done in this layer."""

    def __init__(self, pop_in, pop_out, gradient_model=v.ReLU(), weights=None,
                 decoder=v.spike_count):
        super(Merge, self).__init__(pop_in, pop_out, gradient_model, decoder)
        Layer._is_tuple(pop_in, "Input populations")

        if pop_in[0].size + pop_in[1].size != pop_out.size:
            raise ValueError("Population input sizes must equal population output
                    size")

        if not weights:
            weights = (None, None)

        self.top_size = self.pop_in[0].size
        self.bot_size = self.pop_in[1].size

        assembly = pynn().Assembly(pop_in[0], pop_out[1])
        connector = pynn().AllToAllConnector(allow_self_connections=False)
```

```
        projection1 = pynn().Projection(assembly, pop_out, connector)

    def backward(self, error, optimiser):
        top_size = self.top_size
        top_errors = np.array([x[:top_size] for x in error])
        bot_errors = np.array([x[top_size:] for x in error])
        return (l1_error, l2_error)

    def get_biases(self):
        return np.zeros((self.pop_out.size))

    def get_weights(self):
        return (None, None)

    def set_biases(self, biases):
        return biases

    def set_weights(self, weights):
        return self.weights

    def reset_cache(self):
        pass

    def set_weights(self, weights):
        pass

    def store_spikes(self):
        pass

class Replicate(Layer):
    """A replicate layer that takes a single population and copies the outputs
    to the two output populations. In practice this happens by creating two dense
    layers between the input population and the output populations."""

    def __init__(self, pop_in, pop_out, gradient_model=v.ReLU(), weights=None,
                 biases=0, decoder=v.spike_count_normalised):
        super(Replicate, self).__init__(pop_in, pop_out, gradient_model, decoder)
        Layer._is_tuple(pop_out, "Output populations")
        Layer._is_tuple(weights, "Replicate layer weights", allow_none=True)

        if not pop_out[0].size == pop_out[1].size or \
                (pop_out[0].size != pop_in.size):
            raise ValueError("Output populations must be of the same size as
                 input")

        connector = pynn().AllToAllConnector(allow_self_connections=False)
        projection1 = pynn().Projection(pop_in, pop_out[0], connector)
        projection2 = pynn().Projection(pop_in, pop_out[1], connector)

        self.layer1 = v.Dense(pop_in, self.pop_out[0],
                              gradient_model=gradient_model,
                              weights=1, # Reset later
                              biases=biases,
                              decoder=decoder, projection=projection1)
        self.layer2 = v.Dense(pop_in, self.pop_out[1],
                              gradient_model=gradient_model,
                              biases=biases,
                              weights=1, # Reset later
                              decoder=decoder, projection=projection2)

        # Assign given weights or default to a normal distribution
        if weights is not None:
            self.set_weights(weights)
        else:
            random_weights = np.random.normal(0, 1.0, (2, pop_in.size, pop_out
                 [0].size))
            self.set_weights(random_weights)
```

```

    def backward(self, error, optimiser):
        Layer._is_tuple(error, "Backwards error in replicate layer")
        l1_error = self.layer1.backward(error[0], optimiser)
        l2_error = self.layer2.backward(error[1], optimiser)
        return (l1_error + l2_error) / 2 # Return the mean

    def get_biases(self):
        return (self.layer1.get_biases(), self.layer2.get_biases())

    def get_output(self):
        l1_output = self.layer1.get_output()
        l2_output = self.layer2.get_output()
        return np.array([l1_output, l2_output])

    def get_weights(self):
        return self.weights

    def reset_cache(self):
        self.layer1.reset_cache()
        self.layer2.reset_cache()

    def set_weights(self, weights):
        self.weights = weights
        self.layer1.set_weights(weights[0])
        self.layer2.set_weights(weights[1])

    def store_spikes(self):
        self.layer1.store_spikes()
        self.layer2.store_spikes()
```

B.3.2 test_dense.py

```
import volrpynn.nest as v
import pyNN.nest as pynn
import numpy as np
import pytest

@pytest.fixture(autouse=True)
def setup():
    pynn.setup()

def test_nest_dense_create():
    p1 = pynn.Population(12, pynn.IF_cond_exp())
    p2 = pynn.Population(10, pynn.IF_cond_exp())
    d = v.Dense(p1, p2, v.ReLU())
    expected_weights = np.ones((12, 10))
    actual_weights = d.projection.get('weight', format='array')
    assert not np.allclose(actual_weights, expected_weights) # Should be normal distributed
    assert abs(actual_weights.sum()) <= 24

def test_nest_dense_shape():
    p1 = pynn.Population(12, pynn.SpikeSourcePoisson(rate = 10))
    p2 = pynn.Population(10, pynn.IF_cond_exp())
    d = v.Dense(p1, p2, v.ReLU(), weights = 1)
    pynn.run(1000)
    d.store_spikes()
    assert d.input.shape == (12,)
    assert d.output.shape[0] == 10

def test_nest_dense_projection():
    p1 = pynn.Population(12, pynn.SpikeSourcePoisson(rate = 10))
    p2 = pynn.Population(10, pynn.IF_cond_exp())
    p2.record('spikes')
```

```
    d = v.Dense(p1, p2, v.ReLU(), weights = 1)
    pynn.run(1000)
    spiketrains = p2.get_data().segments[-1].spiketrains
    assert len(spiketrains) == 10
    avg_len = np.array(list(map(len, spiketrains))).mean()
    # Should have equal activation
    for train in spiketrains:
        assert abs(len(train) - avg_len) <= 1

def test_nest_dense_reduced_weight_fire():
    p1 = pynn.Population(1, pynn.IF_cond_exp(i_offset=10))
    p2 = pynn.Population(2, pynn.IF_cond_exp())
    d = v.Dense(p1, p2, v.ReLU(), weights = np.array([[1, 0]]))
    pynn.run(1000)
    spiketrains1 = p1.get_data().segments[-1].spiketrains
    spiketrains2 = p2.get_data().segments[-1].spiketrains
    assert spiketrains1[0].size > 0
    assert spiketrains2[0].size > 0
    assert spiketrains2[1].size == 0

def test_nest_dense_increased_weight_fire():
    p1 = pynn.Population(1, pynn.SpikeSourcePoisson(rate = 1))
    p2 = pynn.Population(1, pynn.IF_cond_exp())
    p2.record('spikes')
    d = v.Dense(p1, p2, v.ReLU(), weights = 2)
    pynn.run(1000)
    spiketrains = p2.get_data().segments[-1].spiketrains
    count1 = spiketrains[0].size
    pynn.reset()
    p1 = pynn.Population(1, pynn.SpikeSourcePoisson(rate = 1))
    p2 = pynn.Population(1, pynn.IF_cond_exp())
    p2.record('spikes')
    d = v.Dense(p1, p2, v.ReLU(), weights = 2)
    pynn.run(1000)
    spiketrains = p2.get_data().segments[-1].spiketrains
    count2 = spiketrains[0].size
    assert count2 >= count1 * 2

def test_nest_dense_chain():
    p1 = pynn.Population(12, pynn.SpikeSourcePoisson(rate = 100))
    p2 = pynn.Population(10, pynn.IF_cond_exp())
    p3 = pynn.Population(2, pynn.IF_cond_exp())
    p3.record('spikes')
    d1 = v.Dense(p1, p2, v.ReLU())
    d2 = v.Dense(p2, p3, v.ReLU())
    pynn.run(1000)
    assert len(p3.get_data().segments[-1].spiketrains) > 0

def test_nest_dense_restore():
    p1 = pynn.Population(12, pynn.IF_cond_exp())
    p2 = pynn.Population(10, pynn.IF_cond_exp())
    d = v.Dense(p1, p2, v.ReLU(), weights = 2)
    d.set_weights(-1)
    t = v.LinearTranslation()
    assert np.array_equal(d.projection.get('weight', format='array'),
             t.weights(np.ones((12, 10)) * -1, 12))
    d.projection.set(weight = 1) # Simulate reset()
    assert np.array_equal(d.projection.get('weight', format='array'),
            np.ones((12, 10)))
    d.restore_weights()
    assert np.array_equal(d.projection.get('weight', format='array'),
            t.weights(np.ones((12, 10)) * -1, 12))

def test_nest_dense_backprop():
    p1 = pynn.Population(4, pynn.IF_cond_exp())
    p2 = pynn.Population(2, pynn.IF_cond_exp())
    l = v.Dense(p1, p2, v.UnitActivation(), weights = 1, decoder = lambda x: x)
```

```
    old_weights = l.get_weights()
    l.input_cache = np.ones((1, 4)) # Mock spikes
    errors = l.backward(np.array([[0, 1]]), lambda w, g, b, bg: (w - g, b - bg))
    expected_errors = np.zeros((2, 4)) + 4
    assert np.allclose(errors, expected_errors)
    expected_weights = np.tile([1, -3], (4, 1))
    assert np.allclose(l.get_weights(), expected_weights)

#def test_nest_dense_batch_gradient():
#    p1 = pynn.Population(4, pynn.IF_cond_exp(**v.DEFAULT_NEURON_PARAMETERS))
#    p2 = pynn.Population(3, pynn.IF_cond_exp(**v.DEFAULT_NEURON_PARAMETERS))
#    weights = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
#    bias = np.array([1, 2, 3])
#    l = v.Dense(p1, p2, v.UnitActivation(), weights=weights, biases=bias)
#    old_weights = l.get_weights()
#    xs = np.array([[1.0, 2.0, 3.0, 4.0]])
#    ys = np.dot(xs, weights)
#    expected_error = np.array([[1408.0, 1624.0, 1840.0, 2056.0]])
#    l.input_cache = xs
#    print(xs)
#    e = l.backward(ys, lambda a, b, c, d: (a, c))
#    assert np.allclose(e, expected_error)

def test_nest_dense_numerical_gradient():
    # Test idea from https://github.com/stephencwelch/Neural-Networks-Demystified
        /blob/master/partSix.py
    # Use simple power function
    f = lambda x: x**2
    fd = lambda x: 2 * x
    e = 1e-4

    weights1 = np.ones((2, 3)).ravel()
    weights2 = np.ones((3, 1)).ravel()

    p1 = pynn.Population(2, pynn.IF_cond_exp())
    p2 = pynn.Population(3, pynn.IF_cond_exp())
    p3 = pynn.Population(1, pynn.IF_cond_exp())
    l1 = v.Dense(p1, p2, v.Sigmoid(), decoder = lambda x: x)
    l2 = v.Dense(p2, p3, v.Sigmoid(), decoder = lambda x: x)
    m = v.Model(l1, l2)
    error = v.SumSquared()

    def forward_pass(xs):
        "Simple sigmoid forward pass function"
        l1.input_cache = xs
        l1.output = l2.input_cache = v.Sigmoid()(np.matmul(xs, l1.weights))
        l2.output = v.Sigmoid()(np.matmul(l2.input_cache, l2.weights))
        return l2.output

    def compute_numerical_gradient(xs, ys):
        "Computes the numerical gradient of a layer"
        weights1 = l1.get_weights().ravel() # 1D
        weights2 = l2.get_weights().ravel()
        weights = np.concatenate((weights1, weights2))
        gradients = np.zeros(weights.shape)

        def initialise_with_distortion(index, delta):
            distortion = np.copy(weights)
            distortion[index] = distortion[index] + delta
            l1.set_weights(distortion[:len(weights1)].reshape(l1.weights.shape))
            l2.set_weights(distortion[len(weights1):].reshape(l2.weights.shape))
            forward_pass(xs)

        # Calculate gradients
        for index in range(len(weights)):
            initialise_with_distortion(index, e)
            error1 = -error(l2.output, ys)
```

```
            initialise_with_distortion(index, -e)
            error2 = -error(l2.output, ys)
            gradients[index] = (error2 - error1) / (2 * e)

        # Reset weights
        l1.set_weights(weights1.reshape(2, 3))
        l2.set_weights(weights2.reshape(3, 1))

        return gradients

    def compute_gradients(xs, ys):
        class GradientOptimiser():
            counter = 2
            gradients1 = None
            gradients2 = None
            def __call__(self, w, wg, b, bg):
                if self.counter > 1:
                    self.gradients2 = wg
                else:
                    self.gradients1 = wg
                self.counter -= 1
                return (w, b)
        output = forward_pass(xs)
        optimiser = GradientOptimiser()
        m.backward(error.prime(l2.output, ys), optimiser)
        return np.concatenate((optimiser.gradients1.ravel(), optimiser.gradients2.ravel()))

    # Normalise inputs
    xs = np.array(([3,5], [5,1], [10,2]), dtype=float)
    xs = xs - np.amax(xs, axis=0)
    ys = np.array(([75], [82], [93]), dtype=float)
    ys = ys / 100

    # Calculate numerical gradients
    numerical_gradients = compute_numerical_gradient(xs, ys)
    # Calculate 'normal' gradients
    gradients = compute_gradients(xs, ys)
    # Calculate the ratio between the difference and the sum of vector norms
    ratio = np.linalg.norm(gradients - numerical_gradients) /\
              np.linalg.norm(gradients + numerical_gradients)
    assert ratio < 1e-07
```

Appendix C

Neuron rate models

The following code simulates the data and produces the plots in Section 3.5.

C.1 Single-neuron population rate experiments

```python
import volrpynn.nest as v
import numpy as np
import pyNN.nest as pynn
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt
get_ipython().magic('matplotlib inline')

parameters = {"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E
    ":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}
pynn.setup()

# Setup initial population
p1 = pynn.Population(1, pynn.IF_cond_exp(**parameters))
p1.record(['spikes', 'v'])

def simulate(offset):
    for recorder in pynn.simulator.state.recorders:
        recorder.clear()
    pynn.reset()
    p1.set(i_offset=offset)
    pynn.run(50)
    return p1.get_data()

def membrane_simulate(offset, pop):
    simulate(offset)
    b = pop.get_data()
    return b.segments[0].filter(name='v')[0]

def plot_membrane_simulate(offset, pop):
    current = membrane_simulate(offset, pop)
    spikes = len(pop.get_data().segments[0].spiketrains[0])
    plt.gca().set_title('Spikes: ' + str(spikes))
    plt.plot(np.arange(0, 50.1, 0.1), current)

def spikes_simulate(offset, pop):
    simulate(offset)
    b = pop.get_data()
    return len(b.segments[0].spiketrains[0])

# Membrane current plot
plot_membrane_simulate(2, p1)
plt.gca().set_title('')
plt.gcf().set_size_inches(6, 4)
plt.gca().set_xlabel('Simulation time in ms')
plt.gca().set_ylabel('Membrane potential in mV')
plt.savefig('membrane.svg')

# Spike rate regression model
xs = np.arange(0, 12.6, 0.02)
spikes = [spikes_simulate(x, p1) for x in xs]
reg = LinearRegression().fit(xs.reshape(-1, 1), spikes)
print(reg.coef_, reg.intercept_)
pred_y = reg.predict(xs.reshape(-1, 1))
r2_score(spikes, pred_y)

# Plot spike count and rate for first population
plt.gca().plot(xs, spikes)
plt.gca().set_ylabel('Number of generated spikes')
plt.gca().set_xlabel('Constant input current in nA')
plt.gca().set_title('')
plt.gcf().set_size_inches(6, 4)
plt.plot(xs, pred_y, color='black', linewidth=0.6, label="f(x) = 3.225x - 1.615",
     linestyle="-.")
plt.legend()
ylim1, ylim2 = plt.gca().get_ylim()
ax2 = plt.gca().twinx()
```

```
ax2.set_ylim(ylim1 / 50, ylim2 / 50)
ax2.set_ylabel('Spike rate (N = 50 ms)')
plt.savefig('spike_rate.svg')


# Deeper layer
p2 = pynn.Population(1, pynn.IF_cond_exp(**parameters))
proj2 = pynn.Projection(p1, p2, pynn.AllToAllConnector())
p3 = pynn.Population(1, pynn.IF_cond_exp(**parameters))
proj3 = pynn.Projection(p2, p3, pynn.AllToAllConnector())
p4 = pynn.Population(1, pynn.IF_cond_exp(**parameters))
proj4 = pynn.Projection(p3, p4, pynn.AllToAllConnector())
p2.record(['v', 'spikes'])
p3.record(['v', 'spikes'])
p4.record(['v', 'spikes'])

def spikes_simulate_deep(offset, weight_function):
    proj2.set(weight=weight_function(offset, p1.size, p2.size))
    proj3.set(weight=weight_function(offset, p2.size, p3.size))
    proj4.set(weight=weight_function(offset, p3.size, p4.size))
    simulate(offset)
    return (p1.get_data(), p2.get_data(), p3.get_data(), p4.get_data())

def to_spikes(d):
    return d.segments[0].spiketrains[0].size

def to_potential(d):
    return d.filter('v')

# Plot spike counts with constant weights
data_constant = [spikes_simulate_deep(x, lambda r, x, y: 1) for x in xs]
spikes_constant = np.array([(to_spikes(x[0]), to_spikes(x[1]), to_spikes(x[2]),
     to_spikes(x[3])) for x in data_constant])

plt.figure(figsize=(15, 4))
ax = plt.subplot(131)
ax.set_ylim(0, 500)
ax.set_title('Second population (N = 1)')
plt.ylabel('Number of generated spikes')
plt.xlabel('Constant input current in nA')
plt.plot(spikes_constant[:, 0], spikes_constant[:, 1])
ax2 = plt.subplot(132)
ax.set_ylim(0, 500)
ax2.set_title('Third population (N = 1)')
plt.ylabel('Number of generated spikes')
plt.xlabel('Number of input spikes')
plt.plot(spikes_constant[:, 1], spikes_constant[:, 2])
ax3 = plt.subplot(133)
ax.set_ylim(0, 500)
ax3.set_title('Fourth population (N = 1)')
plt.xlabel('Number of input spikes')
plt.plot(spikes_constant[:, 2], spikes_constant[:, 3])
plt.savefig('spike_rate_not_weighted.svg')

# Spike rates with adjusted weights
data = [spikes_simulate_deep(x, lambda r, x, y: 0.065 / x) for x in xs]
spikes = np.array([(to_spikes(x[0]), to_spikes(x[1]), to_spikes(x[2]), to_spikes(
     x[3])) for x in data])

plt.figure(figsize=(15, 4))
ax = plt.subplot(131)
ax.set_ylim(0, 40)
ax.set_title('Second population (N = 1)')
plt.ylabel('Number of generated spikes')
plt.xlabel('Number of input spikes')
plt.plot(spikes[:, 0], spikes[:, 1])
ax2 = plt.subplot(132)
```

```
ax2.set_ylim(0, 40)
ax2.set_title('Third population (N = 1)')
plt.ylabel('Number of generated spikes')
plt.xlabel('Number of input spikes')
plt.plot(spikes[:, 1], spikes[:, 2])
ax3 = plt.subplot(133)
ax3.set_ylim(0, 40)
ax3.set_title('Fourth population (N = 1)')
plt.xlabel('Number of input spikes')
plt.plot(spikes[:, 2], spikes[:, 3])
plt.savefig('spike_rate_chain.svg')
```

C.2 MNIST neuron rate experiments

```
import volrpynn.nest as v
import numpy as np
import pyNN.nest as pynn
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt
get_ipython().magic('matplotlib inline')
parameters = {"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E
    ":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}
pynn.setup()

# Setup populations and projections

p1 = pynn.Population(100, pynn.IF_cond_exp(**parameters))
p2 = pynn.Population(100, pynn.IF_cond_exp(**parameters))
proj2 = pynn.Projection(p1, p2, pynn.AllToAllConnector())
p3 = pynn.Population(10, pynn.IF_cond_exp(**parameters))
proj3 = pynn.Projection(p2, p3, pynn.AllToAllConnector())
p1.record(['spikes', 'v'])
p2.record(['spikes', 'v'])
p3.record(['spikes', 'v'])

def simulate(offset):
    for recorder in pynn.simulator.state.recorders:
        recorder.clear()
    pynn.reset()
    p1.set(i_offset=offset)
    pynn.run(50)

def simulate_spikes(offset, weight_function):
    proj2.set(weight=weight_function(offset, p1.size, p2.size))
    proj3.set(weight=weight_function(offset, p2.size, p3.size))
    simulate(offset)
    return (p1.get_data(), p2.get_data(), p3.get_data())

def count_spikes(data):
    return np.array([s.size for s in data.segments[0].spiketrains]).mean()

def simulate_offsets(rates, weight_function):
    return np.array([simulate_spikes(rate, weight_function) for rate in rates])

# Define weight normalisation function
weight_function = lambda r, x, y: 0.065 / x
xs = np.arange(0, 12, 0.1)
data = simulate_offsets(ct, weight_function)

spikes = np.array([(count_spikes(d1), count_spikes(d2), count_spikes(d3)) for (d1
    , d2, d3) in data])

# Plot population rates
```

```
plt.figure(figsize=(15, 4))
ax = plt.subplot(131)
ax.set_ylim(0, 40)
ax.set_title('First population (N = 100)')
plt.ylabel('Number of generated spikes')
plt.xlabel('Constant input current in nA')
plt.plot(xs, spikes[:, 0])
ax2 = plt.subplot(132)
ax2.set_ylim(0, 40)
ax2.set_title('Second population (N = 100)')
plt.ylabel('Average number of generated spikes')
plt.xlabel('Average number of input spikes per neuron')
plt.plot(spikes[:, 0], spikes[:, 1])
ax3 = plt.subplot(133)
ax3.set_ylim(0, 40)
ax3.set_title('Third population (N = 10)')
plt.ylabel('Average number of generated spikes')
plt.xlabel('Average number of input spikes per neuron')
plt.plot(spikes[:, 1], spikes[:, 2])
plt.savefig('spike_rate_mnist.svg')
```

Appendix D

Experiment details

This appendix includes data on the environment, library versions used to execute the experiments and the generated scripts used to run the experiments. The scripts have all been generated by the Volr compiler that is included in Appendix B.

D.1 Hardware and library configurations

The benchmark was run on a Linux kernel version 4.18.0, with a Intel i7-8750H 6-core CPU and a Nvidia GeForce GTX 1060 GPU.

Regarding library versions, Futhark is fixed at 0.9.0, PyNN at 0.9.3 and NEST at 2.16.0.

D.2 Execution wrapper

A wrapper was written for the experiments to ensure a homogenous execution from the commandline as well as data injection and extraction via standard in/out.

Listing D.1: Execution wrapper for experiments

```
import json
import sys
import numpy as np
import volrpynn as v

class Main():
    """A runtime class that accepts a model and exposes a 'train' method
       to train that model with a given optimiser, given data via std in"""

    def __init__(self, model, parameters=None,
            translation=v.LinearTranslation()):
        self.model = model
        if isinstance(parameters, str):
            self._load_parameters_file(parameters)
        if isinstance(parameters, np.ndarray):
            self._load_parameters(parameters)
        if not isinstance(translation, v.Translation):
            raise ValueError('Translator must be a Translation')
        self.translation = translation

    def _load_parameters_file(self, file_name):
        parameters = np.load(file_name)
        self._load_parameters(parameters)

    def _load_parameters(self, parameters):
        for index in range(len(self.model.layers) - 1):
            layer = self.model.layers[index]
            weights, biases = parameters[:2]
            layer.biases = biases
            layer.set_weights(weights)
            # Continue with next element in tuple
            parameters = parameters[2:]

    def _load_data(self):
        if len(sys.argv) < 3:
            raise Exception("Training input and training labels expected as "+\
                            "either argument data or filenames")
        xs_text, ys_text = (sys.argv[1], sys.argv[2])
        if type(xs_text) == str and type(ys_text) == str:
            return (self._load_file(xs_text), self._load_file(ys_text))
        else:
            return xs_text, ys_text

    def _load_file(self, filename):
        with open(filename, 'r') as fp:
            return fp.read()

    def train(self, optimiser, xs=None, ys=None, split=0.8):
        """Trains and tests the model loaded in this class with the given
        optimiser, input data, expected output data and testing/training
        split

        Args:
        optimiser -- The optimisation algorithm that trains the model
        xs -- The input data, will later be normalised
        ys -- Expected categorical output labels
        split -- Testing/training split. Defaults to 0.8 (80%)

        Returns:
        A Report of the training and testing run
        """
        if not isinstance(xs, np.ndarray) or not isinstance(ys, np.ndarray):
            xs, ys = self._load_data()
            xs = np.array(json.loads(xs))
            ys = np.array(json.loads(ys))

```

```
        # Normalise data
        xs = self.translation.to_current(xs)

        # Split training/testing
        split = int(len(xs) * split)
        x_train = xs[:split]
        y_train = ys[:split]
        x_test = xs[split:]
        y_test = ys[split:]
        assert len(x_train) > 0 and len(x_test) > 0, "Must have at least 5 data
             points"
        _, errors, _ = optimiser.train(self.model, x_train, y_train, v.
             SoftmaxCrossEntropy())
        report = optimiser.test(self.model, x_test, y_test, v.
             ErrorCostCategorical())

        reportDict = report.toDict()
        reportDict['train_errors'] = errors # Include training errors

        # Add network weights and biases
        parameters = []
        for layer in self.model.layers[:-1]: # Exclude decode layer
            parameters.append(layer.get_weights().tolist())
            parameters.append(layer.biases.tolist())
        reportDict['parameters'] = parameters

        # Return a JSON version of the report to stdout
        print(json.dumps(reportDict))
```

D.3 Experiment code

The following sections present the code that was used to execute the experiments. Two files are provided per experiment: one for the Futhark backend and one for NEST.

D.4 NAND and XOR

```
import "../lib/github.com/HnimNart/deeplearning/deep_learning"
module dl = deep_learning f64
let x0 = dl.layers.dense (2, 4) dl.nn.relu 1
let x1 = dl.layers.dense (4, 2) dl.nn.relu 2
let x2 = dl.nn.connect_layers x0 x1

let nn = x2
let main [m] (input:[m][]dl.t) (labels:[m][]dl.t) =
  let batch_size = 128
  let train_l = i32.f64 (f64.i32 m * 0.8)
  let train = train_l - (train_l %% batch_size)
  let validation_l = i32.f64 (f64.i32 m * 0.2)
  let validation = validation_l - (validation_l %% batch_size)
  let alpha = 0.1
  let nn' = dl.train.gradient_descent nn alpha
            input[:train] labels[:train]
            batch_size dl.loss.softmax_cross_entropy_with_logits
  let acc = dl.nn.accuracy nn' input[train:train+validation]
     labels[train:train+validation] dl.nn.softmax dl.nn.argmax
  in (acc, nn'.weights)
```

```
import numpy as np
import volrpynn.nest as v
```

```
import pyNN.nest as pynn



p1 = pynn.Population(2, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p3 = pynn.Population(4, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p5 = pynn.Population(2, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
layer0 = v.Dense(p1, p3, weights=np.random.normal(1.0, 1.0, (2, 4)), biases=0.0)
layer1 = v.Dense(p3, p5, weights=np.random.normal(1.0, 1.0, (4, 2)), biases=0.0)
l_decode = v.Decode(p5)
model = v.Model(layer0, layer1, l_decode)

optimiser = v.GradientDescentOptimiser(0.1, simulation_time=50.0)
if __name__ == "__main__":
    v.Main(model).train(optimiser)
```

D.5 MNIST sequential

```
import "lib/github.com/HnimNart/deeplearning/deep_learning"
module dl = deep_learning f64
let x0 = dl.layers.dense (100, 100) dl.nn.relu 0
let x1 = dl.layers.dense (100, 10) dl.nn.relu 1
let x2 = dl.nn.connect_layers x0 x1

let nn = x2
let main [m] (input:[m][]dl.t) (labels:[m][]dl.t) =
  let batch_size = 128
  let train_l = i32.f64 (f64.i32 m * 0.8)
  let train = train_l - (train_l %% batch_size)
  let validation_l = i32.f64 (f64.i32 m * 0.2)
  let validation = validation_l - (validation_l %% batch_size)
  let alpha = 0.1
  let nn' = dl.train.gradient_descent nn alpha
            input[:train] labels[:train]
            batch_size dl.loss.softmax_cross_entropy_with_logits
  let acc = dl.nn.accuracy nn' input[train:train+validation]
     labels[train:train+validation] dl.nn.softmax dl.nn.argmax
  in (acc, nn'.weights)
```

```
import numpy as np
import volrpynn.nest as v
import pyNN.nest as pynn



p1 = pynn.Population(100, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p3 = pynn.Population(100, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p5 = pynn.Population(10, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
layer0 = v.Dense(p1, p3, weights=np.random.normal(1.0, 1.0, (100, 100)), biases=0.0)
layer1 = v.Dense(p3, p5, weights=np.random.normal(1.0, 1.0, (100, 10)), biases=0.0)
```

```
l_decode = v.Decode(p5)
model = v.Model(layer0, layer1, l_decode)

optimiser = v.GradientDescentOptimiser(0.1, simulation_time=50.0)
if __name__ == "__main__":
    v.Main(model).train(optimiser)
```

D.6 MNIST parallel

```
import "lib/github.com/HnimNart/deeplearning/deep_learning"
module dl = deep_learning f64
let x0 = dl.layers.dense (100, 20) dl.nn.relu 0
let x1 = dl.layers.replicate 20 dl.nn.relu 1
let x2 = dl.nn.connect_layers x0 x1
let x3 = dl.layers.dense (20, 10) dl.nn.relu 3
let x4 = dl.layers.dense (20, 10) dl.nn.relu 4
let x5 = dl.nn.connect_parallel x3 x4
let x6 = dl.nn.connect_layers x2 x5
let x7 = dl.layers.merge (10, 10) dl.nn.relu 2
let x8 = dl.nn.connect_layers x6 x7
let x9 = dl.layers.dense (20, 10) dl.nn.relu 9
let x10 = dl.nn.connect_layers x8 x9

let nn = x10
let main [m] (input:[m][]dl.t) (labels:[m][]dl.t) =
  let batch_size = 128
  let train_l = i32.f64 (f64.i32 m * 0.8)
  let train = train_l - (train_l %% batch_size)
  let validation_l = i32.f64 (f64.i32 m * 0.2)
  let validation = validation_l - (validation_l %% batch_size)
  let alpha = 0.1
  let nn' = dl.train.gradient_descent nn alpha
            input[:train] labels[:train]
            batch_size dl.loss.softmax_cross_entropy_with_logits
  let acc = dl.nn.accuracy nn' input[train:train+validation]
     labels[train:train+validation] dl.nn.softmax dl.nn.argmax
  in (acc, nn'.weights)
```

```
import numpy as np
import volrpynn.nest as v
import pyNN.nest as pynn



p1 = pynn.Population(100, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"
     v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I
     ":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p11 = pynn.Population(10, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"
     v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I
     ":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p13 = pynn.Population(20, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"
     v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I
     ":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p3 = pynn.Population(20, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"
     v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I
     ":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p5 = pynn.Population(20, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"
     v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I
     ":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p7 = pynn.Population(10, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"
     v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I
     ":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
```

```
p9 = pynn.Population(20, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
p15 = pynn.Population(10, pynn.IF_cond_exp(**{"tau_syn_I":5,"tau_refrac":0,"v_thresh":-50,"v_rest":-65,"tau_syn_E":5,"v_reset":-65,"tau_m":20,"e_rev_I":-70,"i_offset":0,"cm":1,"e_rev_E":0}))
layer0 = v.Dense(p1, p3, weights=np.random.normal(1.0, 1.0, (100, 20)), biases=0.0)
layer3 = v.Replicate(p3, (p5, p9), weights=(np.random.normal(1.0, 1.0, (20, 20)), np.random.normal(1.0, 1.0, (20, 20))), biases=0.0)
layer1 = v.Dense(p5, p7, weights=np.random.normal(1.0, 1.0, (20, 10)), biases=0.0)
layer2 = v.Dense(p9, p11, weights=np.random.normal(1.0, 1.0, (20, 10)), biases=0.0)
layer4 = v.Merge((p7, p11), p13)
layer5 = v.Dense(p13, p15, weights=np.random.normal(1.0, 1.0, (20, 10)), biases=0.0)
l_decode = v.Decode(p15)
model = v.Model(layer0, layer1, layer2, layer3, layer4, layer5, l_decode)

optimiser = v.GradientDescentOptimiser(0.1, simulation_time=50.0)
if __name__ == "__main__":
    v.Main(model).train(optimiser)
```

D.7 PyNN exception in weight initialisation

Listing D.2: PyNN exception when performing weight initialisation during a test in `test_merge.py`.

```
../../volrpynn/model.py:108: in predict    return self.simulate(time)
../../volrpynn/model.py:138: in simulate    self.reset_weights()
../../volrpynn/model.py:121: in reset_weights    layer.restore_weights()
../../volrpynn/layer.py:75: in restore_weights    self.set_weights(self.get_weights())
../../volrpynn/layer.py:349: in set_weights
    self.layer1.set_weights(weights[0])
../../volrpynn/layer.py:217: in set_weights
    self.projection.set(weight=normalised)
/usr/local/lib/python3.6/dist-packages/pyNN/common/projections.py:172: in set
    attributes = self._value_list_to_array(attributes)
/usr/local/lib/python3.6/dist-packages/pyNN/common/projections.py:208: in _value_list_to_array
    connection_mask = ~numpy.isnan(self.get('weight', format='array', gather='all'))
/usr/local/lib/python3.6/dist-packages/pyNN/common/projections.py:350: in get
    multiple_synapses=multiple_synapses)
/usr/local/lib/python3.6/dist-packages/pyNN/nest/projections.py:362: in _get_attributes_as_arrays
    addr = self.pre.id_to_index(src), self.post.id_to_index(tgt)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = Population(6, IF_cond_exp(<parameters>), structure=Line(dx=1.0, x0=0.0, y=0.0, z=0.0), label='population10')
id = 22

    def id_to_index(self, id):
        """
            Given the ID(s) of cell(s) in the Population, return its (their) index
            (order in the Population).

                >>> assert p.id_to_index(p[5]) == 5
```

```
            """
        if not numpy.iterable(id):
            if not self.first_id <= id <= self.last_id:
>               raise ValueError("id should be in the range [%d,%d], actually %d"
      % (self.first_id, self.last_id, id))
E               ValueError: id should be in the range [11,16], actually 22

/usr/local/lib/python3.6/dist-packages/pyNN/common/populations.py:691: ValueError
```

Glossary

agent An agent is a system that can act based on previous knowledge, and that can *learn* to adapt its actions. Used interchangeably with *system*. 11

ANN Artificial neural networks is a broad term for connected units with weighted edges. Each unit is roughly modelled over the biological neuron, in the way that they can receive a number of inputs (dendrites) but only provide a single output (axon). They also typicall use sigmoidal activation functions to calculate the unit responses. This broad definition covers both third and second generation neural networks, and is generally avoided througout the thesis to avoid ambiguity. 1, 4, 10, 15, 19, 26, 35, 37, 39, 46, 48–50

API A number of interaction-points for a piece of code, such as a library or framework, that are available for other programmers to communicate with. APIs are typically documented as lists of their modular components along with their purpose and usage. 17, 18, 20, 22, 29

ARM A family of reduced instruction set computing (RISC) for processing units that are widespread in smaller and mobile devices.. 21

AST An abstract syntax tree (AST) is a tree structure that represents a data model. ASTs are typically recursive.. 26

DSL A DSL is a language used to model concepts from a certain domain. DSLs are usually simpler than more general programming languages in that they contain fewer concepts and less complex syntax. 3, 17, 22, 23, 25, 28–30, 34, 37, 40, 48–50

FPGA A field-programmable gate array (FPGA) is a programmable integrated circuit, that can achieve high-speed computing by wiring physical blocks together to perform high-speed computations.. 21

machine learning Machine learning is a sub-field within artificial intelligence that is concerned with developing systems that "progressively improves their performance on a certain task" [69]. 1, 2, 11, 16, 48, 49

NN A neural network refers to a circuit of neurons, artificial or biological. plural. 1–6, 8, 11, 12, 15, 17–22, 25, 27, 48–50

OpenCL An open standard for cross-platform parallel programming, which allows software to be executed on CPUs, GPUs or other processors or hardware accelerators. See `https://www.khronos.org/opencl/`. 19, 20, 25

Python A widespread interpreted programming language with dynamic typing.. 19

RAM Random-access memory (RAM) is a temporary storage device that allows read and write access to arbitrary locations without significant delays compared to spinning disks. Typically used as a cache for instructions and memory from long-term storage.. 21

REF A theory and model for rehabilitation in patients with brain lesions, developed by [39]. An extension in the form of the REFGEN model was developed by Mogensen and Overgaard [40] to account for broader aspects of neurocognitive organisation.. 15

SNN A broad term for second or third generation neural networks whose nodes communicates via timed pulses or *spikes*. 2, 4, 7, 9, 10, 14, 19–21, 25, 26, 29, 30, 35, 37, 38, 40, 46, 48, 49

von Neumann architecture A computer architecture for universal computing machines that relies on a processing unit, a control unit and memory for storing data and instructions. Invented by John von Neumann in 1945.. 15, 21

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. "TensorFlow: A system for large-scale machine learning". In: *CoRR* abs/1605.08695 (2016). arXiv: `1605.08695`. URL: `http://arxiv.org/abs/1605.08695`.

[2] Sacha J van Albada, Andrew G Rowley, Johanna Senk, Michael Hopkins, Maximilian Schmidt, Alan B Stokes, David R Lester, Markus Diesmann, and Steve B Furber. "Performance Comparison of the Digital Neuromorphic Hardware SpiNNaker and the Neural Network Simulation Software NEST for a Full-Scale Cortical Microcircuit Model". In: *Frontiers in Neuroscience* 12 (2018), p. 291. ISSN: 1662-453X. DOI: `10.3389/fnins.2018.00291`. URL: `https://www.frontiersin.org/article/10.3389/fnins.2018.00291`.

[3] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis Dewolf, Terrence Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. "Nengo: A Python tool for building large-scale functional brain models". In: *Frontiers in neuroinformatics* 7 (Jan. 2014), p. 48.

[4] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006. ISBN: 978-0387-31073-2.

[5] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. "Code Generation in Computational Neuroscience: A Review of Tools and Techniques". In: *Frontiers in Neuroinformatics* 12 (2018), p. 68. ISSN: 1662-5196. DOI: `10.3389/fninf.2018.00068`. URL: `https://www.frontiersin.org/article/10.3389/fninf.2018.00068`.

[6] Artificial Brains. *DARPA SyNAPSE Program*. [Online; Accessed 02-10-2018]. 2018. URL: `http://www.artificialbrains.com/darpa-synapse-program`.

[7] Daniel Brüderle, Mihai Petrovici, Bernhard Vogginger, Matthias Ehrlich, and Al. "A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems". In: *Biological Cybernetics* 104 (2011), pp. 263–296.

[8] Ted Carnevale. "Neuron simulation environment". In: (2007). URL: `http://www.scholarpedia.org/article/Neuron_simulation_environment`.

[9] Microsoft Corporation. *The Microsoft Cognitive Toolkit*. [Online; Accessed 30-08-2018]. 2018. URL: `https://docs.microsoft.com/en-gb/cognitive-toolkit`.

[10] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. "PyNN: a common interface for neuronal network simulators". In: *Frontiers in Neuroinformatics* 2 (2009), p. 11. ISSN: 1662-5196. DOI: `10.3389/neuro.11.011.2008`. URL: `https://www.frontiersin.org/article/10.3389/neuro.11.011.2008`.

[11] Peter Dayan and L. F. Abbot. *Theoretical neuroscience - Computational and Mathematical Modeling of Neural Systems*. MIT Press, 2001. ISBN: 978-0-262-04199-7.

[12] Google DeepMind. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. [Online; accessed 28-01-2019]. 2019. URL: `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`.

[13] Peter U. Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing". In: *2015 International Joint Conference on Neural Networks (IJCNN)* (2015), pp. 1–8.

[14] Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, Diogo Moitinho de Almeida, Brian McFee, Hendrik Weideman, Gábor Takács, Peter de Rivaz, Jon Crall, Gregory Sanders, Kashif Rasul, Cong Liu, Geoffrey French, and Jonas Degrave. *Lasagne: First release.* Aug. 2015. DOI: `10.5281/zenodo.27878`. URL: `http://dx.doi.org/10.5281/zenodo.27878`.

[15] Chris Eliasmith. *How to Build a Brain: A Neural Architecture for Biological Cognition*. Oxford University Press, 2015. ISBN: 978-0-190-26212-9.

[16] Chris Eliasmith and Charles H. Anderson. *Neural Engineering: Computation, representation, and dynamics in neurobiological systems*. MIT Press, 2004. ISBN: 978-0-262-05071-5.

[17] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. “Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large”. In: *Proceedings of the ACM on Programming Languages* 2.ICFP (July 2018), 97:1–97:30. ISSN: 2475-1421.

[18] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. 2018. URL: `https://futhark-book.readthedocs.io`.

[19] Inc. Facebook. *Caffe2, A New Lightweight, Modular, and Scalable Deep Learning Framework*. [Online; Accessed 30-08-2018]. 2018. URL: `https://caffe2.ai`.

[20] G. B. Orr G. Montavon and K. Müller. *Neural Networks: Tricks of the Trade*. 1998. ISBN: 978-3-642-35289-8.

[21] Marc-Oliver Gewaltig and Markus Diesmann. “NEST (NEural Simulation Tool)”. In: *Scholarpedia* (2007). URL: `http://www.scholarpedia.org/article/NEST_%28NEural_Simulation_Tool%29`.

[22] Dan F. M. Goodman and Romain Brette. “Brian simulator”. In: *Scholarpedia* (2013). URL: `http://www.scholarpedia.org/article/Brian_simulator`.

[23] Haskell. *The Haskell Programming Language*. [Online; accessed 21-01-2019]. 2019. URL: `https://wiki.haskell.org/Haskell`.

[24] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), pp. 556–571.

[25] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. “High-performance defunctionalization in Futhark”. In: *Symposium on Trends in Functional Programming (TFP'18)*. Sept. 2018.

[26] Eric Hunsberger and Chris Eliasmith. “Spiking Deep Networks with LIF Neurons”. In: *CoRR* abs/1510.08829 (2015). arXiv: `1510.08829`. URL: `http://arxiv.org/abs/1510.08829`.

[27] G. Indiveri and S. C. Liu. “Memory and Information Processing in Neuromorphic Systems”. In: *Proceedings of the IEEE* 103.8 (Aug. 2015), pp. 1379–1397. ISSN: 0018-9219. DOI: `10.1109/JPROC.2015.2444094`.

[28] Mark Karpov. *Megaparsec on GitHub*. [Online; accessed 31-March-2018]. 2018. URL: `%5Curl%7Bhttps://github.com/mrkkrp/megaparsec%7D`.

[29] Keras. *The Python Deep Learning Library*. [Online; accessed 10-May-2018]. 2018. URL: `https://keras.io/`.

[30] Andreas Klöckner. *PyOpenCL documentation*. [Online; accessed 21-01-2018]. 2019. URL: `https://mathema.tician.de/software/pyopencl/`.

[31] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. *Pytest 4.0.1*. 2018. URL: `https://github.com/pytest-dev/pytest`.

[32] Junhaeng Lee, Tobi Delbrück, and Michael Pfeiffer. "Training Deep Spiking Neural Networks using Backpropagation". In: *CoRR* abs/1608.08782 (2016). arXiv: `1608.08782`. URL: `http://arxiv.org/abs/1608.08782`.

[33] Chit-Kwan Lin, Andreas Wild, Gautham N. Chinya, Tsung-Han Lin, Mike Davies, and Hong Wang. "Mapping Spiking Neural Networks Onto a Manycore Neuromorphic Architecture". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 78–89. ISBN: 978-1-4503-5698-5. DOI: `10.1145/3192366.3192371`. URL: `http://doi.acm.org/10.1145/3192366.3192371`.

[34] Wolfgang Maass. "Networks of spiking neurons: The third generation of neural network models". In: *Neural Networks* 10.9 (1997), pp. 1659–1671. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/S0893-6080(97)00011-7`. URL: `http://www.sciencedirect.com/science/article/pii/S0893608097000117`.

[35] Henry Markram, Eric R. Kandel, Paul M. Matthews, Rafael Yuste, and Christof Koch. "Neuroscience thinks big (and collaboratively)". In: *Nature Reviews Neuroscience* 14 (), p. 659. URL: `http://dx.doi.org/10.1038/nrn3578`.

[36] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[37] Mcstrother. *Visualisation of a neural network by Mcstrother from Wikimedia Commons*. [CC BY 3.0 (https://creativecommons.org/licenses/by/3.0)], from Wikimedia Commons. 2010. URL: `%5Curl%7Bhttps://commons.wikimedia.org/wiki/File:Single_layer_ann.svg%7D`.

[38] J. Mogensen. "Reorganization of Elementary Functions (REF) after brain injury: implications for the therapeutic interventions and prognosis of brain injured patients suffering cognitive impairments". In: *Brain Dam-*

age: Causes, Management and Prognosis 1 (2012). Ed. by A. J. Schäfer and J. Müller, pp. 1–40.

[39] J. Mogensen. "Reorganization of the Injured Brain: Implications for Studies of the Neural Substrate of Cognition". In: *Frontiers in Psychology* 2 (2011), p. 7. ISSN: 1664-1078. DOI: `10.3389/fpsyg.2011.00007`. URL: `https://www.frontiersin.org/article/10.3389/fpsyg.2011.00007`.

[40] J. Mogensen and M. Overgaard. "Reorganization of the Connectivity between Elementary Functions – A Model Relating Conscious States to Neural Connections". In: *Frontiers in Psychology* 8 (2017), p. 625. ISSN: 1664-1078. DOI: `10.3389/fpsyg.2017.00625`. URL: `https://www.frontiersin.org/article/10.3389/fpsyg.2017.00625`.

[41] Michael Mozer. "A Focused Backpropagation Algorithm for Temporal Pattern Recognition". In: 3 (Jan. 1995).

[42] Nengo. *Nengo documentation*. 2018. URL: `https://nengo.ai`.

[43] Nils J. Nilsson. *The Quest for Artificial Intelligence*. 1st. New York, NY, USA: Cambridge University Press, 2009. ISBN: 9780521122931.

[44] ONNX. *Open neural network exchange format*. [Online; accessed 10-May-2018]. URL: `https://onnx.ai/`.

[45] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan-Kaufmann, 1988. ISBN: 978-1-558-60479-7.

[46] Judea Pearl. *The Book of Why: The New Science of Cause and Effect*. Basic Books, 2018.

[47] Jens Egholm Pedersen. *Modelling learning systems - a DSL for cognitive neuroscientists*. Copenhagen University [Online; accessed 06-05-2018]. 2018. URL: `https://github.com/jegp/volr-report`.

[48] Thomas Pfeil, Andreas Grübl, Sebastian Jeltsch, Eric Müller, Paul Müller, Mihai A Petrovici, Michael Schmuker, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier. "Six Networks on a Universal Neuromorphic Computing Substrate". In: 7 (Feb. 2013), p. 11.

[49] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. ISBN: 978-0-262-16209-8.

[50] PyNN. *Documentation: Introduction*. [Online; accessed 10-May-2018]. 2018. URL: `https://neuralensemble.org/docs/PyNN/introduction.html`.

[51] PyTorch. *Tensors and Dynamic neural networks in Python with strong GPU acceleration*. [Online; accessed 10-May-2018]. 2018. URL: `https://pytorch.org/`.

[52] Ian H. Robertson and Jaap M. J. Murre. "Rehabilitation of Brain Damage: Brain Plasticity and Principles of Guided Recovery". In: *Psychological Bulletin* 125.5 (1999), pp. 544–575.

[53] Raul Rojas. *Neural Networks: A Systematic Introduction*. Springer, 1996.

[54] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. “Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification”. In: *Frontiers in Neuroscience* 11 (2017), p. 682. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00682. URL: https://www.frontiersin.org/article/10.3389/fnins.2017.00682.

[55] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Neurocomputing: Foundations of Research”. In: ed. by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chap. Learning Representations by Back-propagating Errors, pp. 696–699. ISBN: 0-262-01097-6. URL: http://dl.acm.org/citation.cfm?id=65669.104451.

[56] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, Dec. 2002. ISBN: 0137903952. URL: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%5C&path=ASIN/0137903952.

[57] J. Schmidhuber. “Deep Learning in Neural Networks: An Overview”. In: *Neural Networks* 61 (2015). Published online 2014; based on TR arXiv:1404.7828 [cs.NE], pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003.

[58] Sebastian Schmitt, Johann Klaehn, Guillaume Bellec, Andreas Grübl, Maurice Guettler, Andreas Hartel, Stephan Hartmann, Dan Husmann de Oliveira, Kai Husmann, Vitali Karasenko, Mitja Kleider, Christoph Koke, Christian Mauch, Eric Müller, Paul Müller, Johannes Partzsch, Mihai A. Petrovici, Stefan Schiefer, Stefan Scholze, Bernhard Vogginger, Robert A. Legenstein, Wolfgang Maass, Christian Mayr, Johannes Schemmel, and Karlheinz Meier. “Neuromorphic Hardware In The Loop: Training a Deep Spiking Network on the BrainScaleS Wafer-Scale System”. In: *CoRR* abs/1703.01909 (2017). arXiv: 1703.01909. URL: http://arxiv.org/abs/1703.01909.

[59] Oliver Schulte. “Formal Learning Theory”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2018. Metaphysics Research Lab, Stanford University, 2018.

[60] *scikit-learn: Machine Learning in Python*. [Online; accessed 30-08-2018]. 2018. URL: https://scikip-learn.org/.

[61] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. “On the importance of initialization and momentum in deep learning”. In: *International conference on machine learning*. 2013, pp. 1139–1147.

[62] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. “Deep learning in spiking neural networks”. In: *Neural Networks* 111 (Mar. 2019), pp. 47–63. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.12.002. URL: http://dx.doi.org/10.1016/j.neunet.2018.12.002.

[63] Amirhossein Tavanaei and Anthony Maida. "A Minimal Spiking Neural Network to Rapidly Train and Classify Handwritten Digits in Binary and 10-Digit Tasks". In: 4 (June 2015), pp. 1–8.

[64] Paul Thagard. "Cognitive Science". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2014. Metaphysics Research Lab, Stanford University, 2014.

[65] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: `http://arxiv.org/abs/1605.02688`.

[66] Duc Minh Tran. *Implementation of a deep learning library in Futhark*. BSc. thesis at the department of Computer science, Copenhagen University. Supervised by Troels Henriksen. 2018.

[67] Stanford University. *Brains in Silicon*. [Online; Accessed 02-30-2018]. 2018. URL: `https://web.stanford.edu/group/brainsinsilicon/LabPositions.html`.

[68] Florian Walter, Florian Röhrbein, and Alois Knoll. "Neuromorphic implementations of neurobiological learning algorithms for spiking neural networks". In: *Neural Networks* 72.Supplement C (2015). Neurobiologically Inspired Robotics: Enhanced Autonomy through Neuromorphic Cognition, pp. 152–167. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/j.neunet.2015.07.004`. URL: `http://www.sciencedirect.com/science/article/pii/S0893608015001410`.

[69] Wikipedia. *Machine learning — Wikipedia, The Free Encyclopedia*. [Online; accessed 31-March-2018]. 2018. URL: `%5Curl%7Bhttps://en.wikipedia.org/w/index.php?title=Machine_learning&oldid=833013504%7D`.

[70] C. Cortes Y. LeCun and C. J. C. Burges. *The MNIST database of handwritten digits*. [Online; Accessed 04-02-2019]. 2019. URL: `http://yann.lecun.com/exdb/mnist/`.

[71] Y. Bengio Y. LeCun L. Bottou and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998).

Index