Deep Learning on SpiNNaker

Master Thesis

Jonas Fassbender jonas@fassbender.dev

In the course of studies
High Performance Computing with Data Science

For the degree of
Master of Science

The University of Edinburgh

First supervisor: Caoimhín Laoide-Kemp
EPCC, University of Edinburgh

Second supervisor: Dr Kevin Stratford
EPCC, University of Edinburgh

Third supervisor: Dr Alan Stokes
APT, University of Manchester

Edinburgh, August 2020

Declaration

I declare that this dissertation was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Jonas Fassbender
August 2020

Abstract

Contents

List of Figures

Listings

1. Introduction

Deep learning is revolutionizing the world. It has become part of our daily lives as consumers, powering major software products—from recommendation systems and translation tools to web search (LeCun et al., 2015). Major breakthroughs in fields like computer vision (Krizhevsky et al., 2012) or natural language processing (Hinton et al., 2012) were achieved through the use of deep learning. It has emerged as a driving force behind discoveries in numerous domains like particle physics (Ciodaro et al., 2012), drug discovery (Ma et al., 2015), genomics (Leung et al., 2014) and gaming (Silver et al., 2016).

Deep learning has become so ubiquitous that we are changing the way we build modern hardware to account for its computational demands. From the way edge devices like mobile phones or embedded systems are built (Deng, 2019) and modern CPUs (Perez, 2017) to specialized hardware designed only for deep learning models, such as Google's tensor processing unit (TPU) (Jouppi et al., 2017) or NVIDIA's EGX Edge AI platform (Boitano, 2020). Whole state-of-the-art supercomputers are built solely for deep learning. An example would be a supercomputer built by Microsoft for OpenAI, which is part of the Azure cloud (Langston, 2020).

Hardware manufacturer are faced with a major challenge in meeting the computational demands arising from inference, and more importantly, training deep learning models. OpenAI researchers have estimated that the computational costs of training increases exponentially; approximately every 3.4 months the cost doubles (Amodei et al., 2019). Amodei et al. (2019) claims the deep reinforcement learning agent AlphaGo Zero—the successor of the famous AlphaGo program, which was able to beat Go world champion Lee Sedol (Silver et al., 2017)—to be the system with the highest computational demands of approximately 1850 petaflop/s-days. AlphaGo Zero was trained for 40 days on a machine with 4 TPUs (Silver et al., 2017). With the end of Moore's Law (Loeffler, 2018), chip makers have to get creative in scaling up computing, the same way machine learning researchers are scaling up their models (Simonite, 2016). Therefore production and research into new hardware designs for deep learning are well on the way.

Another field which has high computational demands for very specific tasks and algorithms is computational neuroscience. Computational neuroscience has long been linked to deep learning, which has its origin in research done by neuroscientists (Goodfellow et al., 2016; McCulloch and Pitts, 1943). While in the recent past deep learning research has been more focused on mathematical topics like statistics and probability theory, optimization or linear algebra, researchers are again looking to neuroscience to further improve the capabilities of deep learning models (Marblestone et al., 2016).

But the algorithms developed by computational neuroscientists are not the only aspect drawing attention from the deep learning community. Computational neuroscience has a long standing history of developing custom hardware for the efficient modeling of the human brain, so called neuromorphic computing. Neuromorphic computing—a computer architecture inspired by the biological nervous system—has been around since the 1980s (Mead, 1989). Today, neuromorphic computers are being developed to meet the demands for efficient computing needed to run large-scale spiking neural networks used for modeling brain functions (Furber, 2016). While being developed mainly for the task of modeling the human brain, deep learning has been linked to neuromorphic computing, especially in the context of commercial usability (Gomes, 2017). Both the low energy demands of neuromorphic computers—such as IBM's True North (Cassidy et al., 2013) or The University of Manchester's Spiking Neural Network Architecture (SpiNNaker) (Furber et al., 2006)—and their scalability and massive-parallelism are intriguing for two very important use cases of deep learning:

(i) edge computing, for example robotics and mobile devices, (ii) supercomputers and the cloud-era (Gomes, 2017).

This thesis's original goal was an investigation of the performance of SpiNNaker machines for deep learning. We wanted to conduct a benchmark by training the state-of-the-art computer vision model ResNet-50 (He et al., 2015) under the closed division rules of the MLPerf training benchmark (Mattson et al., 2019). In order to benchmark ResNet-50 on SpiNNaker, a prototypical implementation was developed as part of this thesis. Unfortunately, our implementation fell far short of being able to run a state-of-the-art computer vision model as complex as ResNet-50. The main problem during the development process was time and this project is another example of Hofstadter's Law.[1] We were not able to finish all the features needed for ResNet-50. The original work plan would have sufficed, but we spent too much time trying to fix problems, which were unaccounted for in the work plan. Nonetheless, we gained valuable experience. Instead of presenting a benchmark, this thesis investigates the problems encountered during the development process and tries to communicate bottlenecks found and misconceptions made. Hopefully this thesis can be a building block for the future efforts of implementing deep learning on SpiNNaker. The developed prototype is licensed under the GNU GPLv3.0 licence and is available—including this thesis and other documentation, for example a report of the research and planning phase—under `https://github.com/jofas/master_thesis`.

Section 2 presents the background of this thesis. An introduction to deep learning is given in Section 2.1, as well as an overview of the benchmark in Section 2.2. Section 2.3 describes the SpiNNaker architecture and compares it to current deep learning hardware. Related work can be found in Section 3. Section 4 presents the architecture of the developed prototype and discusses the problems encountered during the implementation. Section 4.1 describes the relevant aspects of the SpiNNaker programming model and the SpiNNaker toolchain. In Section 4.2, the architecture of the protoype is presented. Section 4.3 will outline and discuss the problems encountered and what we did in order to solve them. In Section 5, ideas are presented which could potentially solve the problems of the prototype. Section 6 contains the conclusion, while Section 7 outlines a few general next steps and ideas for implementing deep learning on SpiNNaker, based on observations made and knowledge gained during the efforts of enabling deep learning on SpiNNaker.

2. Background

This section summarizes the background knowledge needed for the following sections. First, a short introduction to deep learning is given in Section 2.1. The main focus lies on the basic concepts and those concepts important for computer vision. Next, Section 2.2 outlines our initial ideas for benchmarking the prototype and SpiNNaker against other deep learning libraries and accelerators. Lastly, the SpiNNaker neuromorphic computer architecture is described in Section 2.3. SpiNNaker is also compared against the two state-of-the-art hardware solutions for deep learning that currently produce the best performance in training and inference. Namely general purpose graphical processing units (GPGPUs) and Google's tensor processing unit (TPU).

1. Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law (Hofstadter, 1979).

2.1 An Introduction to Deep Learning

While it may seem that deep learning is a recent development in the field of artificial intelligence—due to all the recently announced breakthroughs (Senior et al., 2020; Vinyals et al., 2019; OpenAI, 2019; Murphy, 2019)—it has actually existed since the 1940s (Goodfellow et al., 2016). McCulloch and Pitts (1943) first described the McCulloch-Pitts neuron as a simple mathematical model of a biological neuron, which marks the origin of what is known today as deep learning.

Even though deep learning models today are still called *artificial neural networks* (due to their historical context), they are quite different from *spiking neural networks* (which SpiNNaker was designed to run efficiently). While the former has been described as "just nonlinear statistical models" (Hastie et al., 2009), the latter incorporated findings about biological neurons and is therefore more closely related to how the nervous system works (Maass, 1997). Spiking neural networks are mostly used for simulation, unlike deep learning models, which are mostly used for inference.

The history of deep learning can be broken down into three distinct phases. Only during the last of these phases was the methodology called deep learning (Goodfellow et al., 2016). Arguably the reason why deep learning seems to be a new development. The first phase, where deep learning was known as cybernetics, ranged from the 1940s to the 1960s (Goodfellow et al., 2016). As stated above, it was the time when the first biologically inspired representations of neurons were developed. Rosenblatt (1958) presents the first model, a single trainable artificial neuron known as the perceptron (see Figure 1).

Today's perceptron receives a real-valued n-vector $\mathbf{x}$ of *input* signals and builds the dot product with another real-valued n-vector known as *weights* $\mathbf{w}$: $\mathbf{x} \cdot \mathbf{w} = \sum_{i=1}^{n} x_i w_i$. The *bias* b is added to the dot product. $\mathbf{x} \cdot \mathbf{w} + b$ is then passed to the *activation function* g—some fixed transformation function appropriate for the application domain. $y = g(\mathbf{x} \cdot \mathbf{w} + b)$ is the output of the perceptron.

During *supervised learning*, we have a set of *examples*. Each example consists of an *input* vector $\mathbf{x}$ and a associated *label* y generated by an unknown function $f^*(\mathbf{x})$. A perceptron can be trained to approximate $f^*(\mathbf{x})$. We can describe a perceptron as the mathematical function

$$y = f(\mathbf{x}; \mathbf{w}, b) = g(\mathbf{x} \cdot \mathbf{w} + b). \tag{1}$$

$f(\mathbf{x}; \mathbf{w}, b)$ is known as a *(statistical) model* with $\mathbf{w}$ and b as its *trainable parameters*, which are trained/learned in order to approximate f^* with f. How a network of perceptrons—a more complex statistical model better suited for real world applications—is trained via backpropagation and gradient descent, will be explained below.

The second historical phase of deep learning is known as connectionism (1980s-1990s) (Goodfellow et al., 2016). Its main contributions to today's knowledge were the backpropagation algorithm (Rumelhart et al., 1986a) and the approach of parallel distributed processing (Rumelhart et al., 1986b,c), which provided a mathematical framework around the idea that a large number of simple computational units (e.g. the perceptron) could achieve intelligent behavior when connected together (Goodfellow et al., 2016). Backpropagation enabled the training of networks of perceptrons—artificial neural networks.

The quintessential artificial neural network is the *multilayer perceptron* (MLP), also called a *feedforward neural network* (see Figure 2) (Goodfellow et al., 2016). The MLP consists of multiple perceptrons organized in *layers*. Layers are connected successively such that the output of each of its perceptrons reaches all perceptrons in the next layer. Such a layer is known to be *fully-connected* or *dense*. No cycle exists between perceptrons; the MLP is a directed acyclic graph. Unlike the

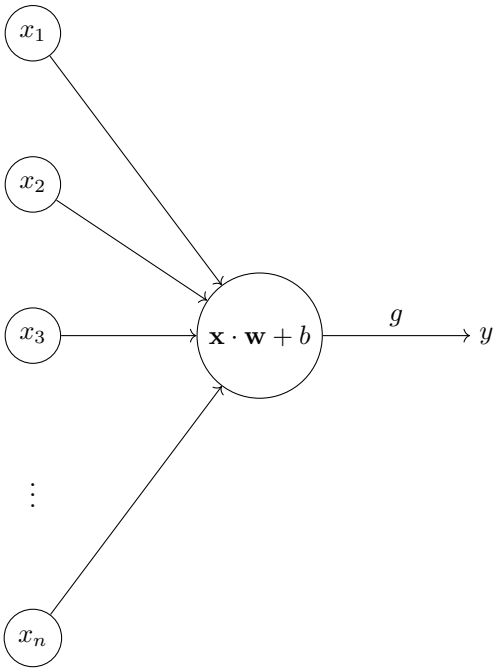


Figure 1: Schema of a perceptron.

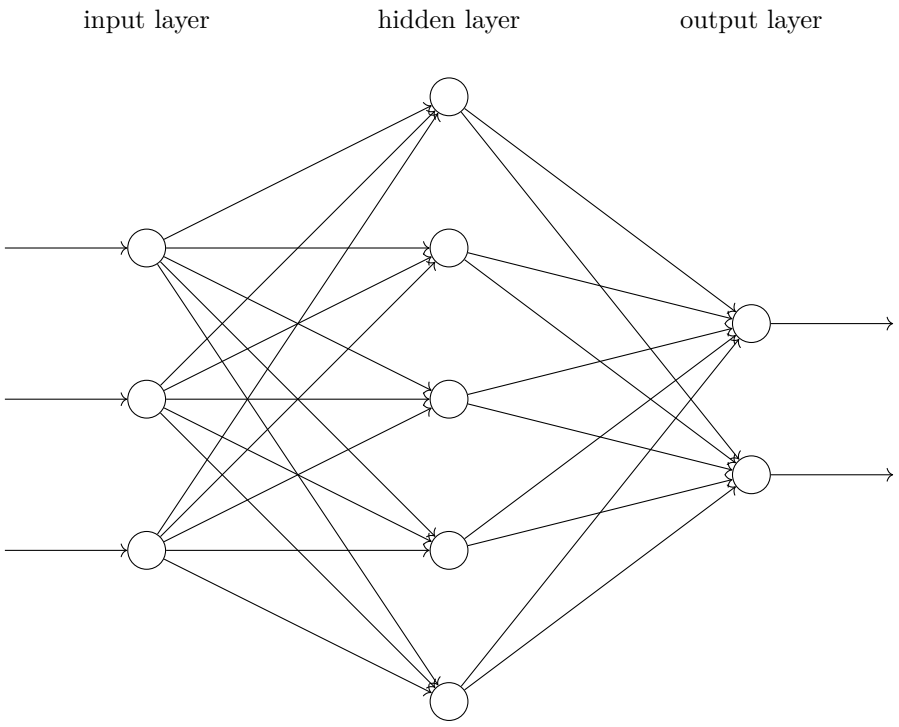


Figure 2: Schema of a MLP or feedforward neural network.

single layer perceptron, the MLP has at least one *hidden layer*. A hidden layer is a layer between the input and output layers (see Figure 2).

A MLP can also be represented as a statistical model $f(\mathbf{x};\theta)$. Computing $f(\mathbf{x})$—also called *inference* or the *forward pass*—can be described as a layer-wise composition of functions $f^{(1)}, f^{(2)}, \dots, f^{(l)}$, each function $f^{(i)}, i < l$ being a hidden layer and $f^{(l)}$ being the output layer. The perceptron has the weight vector $\mathbf{w}$ and the bias b as its parameters (see Equation 1). The parameters of a layer are the combination of $\mathbf{w}$ and b for each of its perceptrons. For example, if the first hidden layer contains m perceptrons and $\mathbf{x}$ is a n-vector, then the parameters of $f^{(1)}$ would be a matrix $\mathbf{W} : n \times m$ and a m-vector $\mathbf{b}$. The output of layer $f^{(1)}$ would be a m-vector computed as follows:

$$f^{(1)}(\mathbf{x};\mathbf{W},\mathbf{b}) = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b}). \tag{2}$$

The second layer takes the output of the first and so forth. The forward pass of the MLP is computed as:

$$y = f^{(l)}(f^{(l-1)}(\dots f^{(1)}(\mathbf{x}))). \tag{3}$$

The backpropagation algorithm is a way to train the parameters of a MLP (or other deep learning models) so that it approximates the unknown function f^* which generates the labels of the examples we have in our data set. The data set used for training a model is called the *training*

set. In addition to the training set there normally exists a *test set* with examples the model has not seen before (examples not in the training set). The test set is used to determine the generalization performance of the model. Backpropagation is an algorithm that allows to train a deep learning model with *(stochastic or batch) gradient descent*. For example, $\hat{y} = f(\mathbf{x})$ and y is the true label (y and $\hat{y}$ are k-vectors), the error of f is computed using a *loss function* L, for example mean squared error: $1/k \sum_{i=1}^{k}(y_k - \hat{y}_k)^2$. In order to get the *gradients* of the weights of the output layer we calculate the derivative of the loss according to each weight w_{ij} in $\mathbf{W}$ with the chain rule:

$$\frac{\delta L}{\delta w_{ij}} = \frac{\delta L}{\delta g}\frac{\delta g}{\delta h}\frac{\delta h}{\delta w_{ij}}, \tag{4}$$

h being $\mathbf{W}^\top f^{(l-1)} + \mathbf{b}$.

w_{ij} is updated by performing some form of gradient descent:

$$w_{ij}^{+} = w_{ij} - \mu \sum_{k=1}^{m} \frac{\delta L_k}{\delta w_{ij}}. \tag{5}$$

Which kind of gradient descent depends on m. m represents the amount of training examples seen, before the weights are updated. If m equals one, (5) would be called stochastic gradient descent. If m would encompass the whole training set, the equation would be gradient descent. Is m somewhere in-between one and the whole training set, one speaks of batch or mini-batch gradient descent. A deep learning model is normally trained by passing the whole training set multiple times through the model. Each pass over the whole training set is called an *epoch*. μ in (5) is called the *learning rate*.

The same procedure is applied to the following hidden layers. The total loss of the next hidden layer is given as:

$$L^{(l-1)} = \sum_{i=1}^{n} \frac{\delta L}{\delta f_i^{(l-1)}} = \sum_{i=1}^{n} \frac{\delta L}{\delta g}\frac{\delta g}{\delta h}\frac{\delta h}{\delta f_i^{(l-1)}}, \tag{6}$$

$f_i^{(l-1)}$ being the i-th perceptron of the hidden layer $l-1$.

Hornik et al. (1989) demonstrated that a non-linear MLP (the activation functions are non-linear transformations of $h(x) = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$) can overcome the famous XOR problem of a single layer perceptron demonstrated in Minsky and Papert (1969). Another major contribution of the phase of connectionism was the neocognitron (Fukushima, 1980), the origin of today's *convolutional neural networks* (CNNs)—which are the state-of-the-art approach for building computer vision models—and the application of the backpropagation algorithm to fully automate the training of CNNs (LeCun et al., 1989).

Goodfellow et al. (2016) claims that the third and current phase of deep learning—where the name deep learning was established—starts with Hinton et al. (2006) describing a new learning algorithm called greedy layer-wise pretraining, which they applied to deep belief networks. Greedy layer-wise pretraining was soon generalized to work with other deep artificial neural network architectures (Ranzato et al., 2006; Bengio et al., 2007). While these papers may have resulted in the term deep learning, they were not the reason for the resurrected interest in this methodology. The two most important factors are the increase of available data and computation. The former enables better generalization (Goodfellow et al., 2016), while the latter allows training bigger models (more

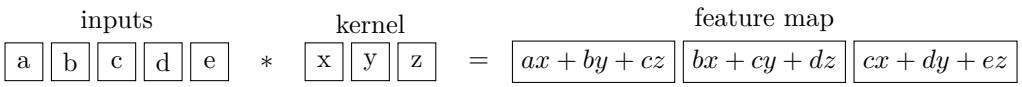


Figure 3: Example of a 1D cross-correlation operation with a kernel size of three, a single channel, a single filter, a stride of one and valid padding.

hidden layers—the *depth* of the neural networks increased) which can solve more complex problems (Bengio and LeCun, 2007; Goodfellow et al., 2016).

Like the perceptron, "neurons" in a *convolutional layer* are inspired by findings of neuroscientists. In this case by research done by Hubel and Wiesel about the mammalian visual cortex (Hubel and Wiesel, 1959, 1962, 1968). CNNs are just deep learning models which have at least one convolutional layer. They are applied to problems which have a grid-like topology, like time-series (1D), images (2D) or videos (3D) (Goodfellow et al., 2016).

Unlike dense layers of perceptrons, convolutional layers do not apply a full matrix multiplication $\mathbf{W}^\top \mathbf{x}$ but instead a linear operation $*$ called convolution. A one dimensional discrete convolution can be described as:

$$s(i) = (x * w)(i) = \sum_n x(i+n)w(n). \tag{7}$$

Equation 7 is not really a convolution but is referred to as *cross-correlation*. Unlike true convolution, cross-correlation is not commutative (Goodfellow et al., 2016). However, commutativity is not a factor in practice, so many deep learning libraries, like Keras (Chollet et al., 2015) or the prototype developed for this thesis implement cross-correlation rather than true convolution. Convolution will refer to cross-correlation below.

In the case of deep learning, x is a nD array called the *input* and w is another nD array referred to as the *kernel*. The kernel elements are the trainable parameters (Goodfellow et al., 2016). In Equation 7, x and w are one dimensional. If we let x be a m-vector, the function $x(i)$ is defined as:

$$x(i) = \begin{cases} x_i & \text{if } 1 \leq i \leq m \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

n is the size of the kernel in the first dimension. Figure 3 shows an example of how the output of a 1D convolutional layer is computed. Figure 4 shows the schema of the convolutional layer performing the operation from Figure 3. The result of a convolution can be transformed by an activation function like the perceptron and the concept of the bias applies also.

Normally a convolutional layer does not consist of a single convolution, but applies multiple kernels to the output of the previous layer. A single convolution is called a *filter* and a layer consists of a predefined number of filters, each with its own kernel (and optionally a bias) (Brownlee, 2019a). The output of a convolutional layer is often called a *feature map* (Goodfellow et al., 2016). Even though an image may seem to be a two dimensional structure of pixels, in most cases it is actually three dimensional, the third dimension being the RGB color values for each pixel. The third dimension of the three RGB colors are called the *channels* (Goodfellow et al., 2016). For example, we have a data set of images with 256×256 pixels and three channels (red, green and blue). We pass the image to a convolutional layer with a 3×3 kernel shape and 64 filters. A kernel consists of

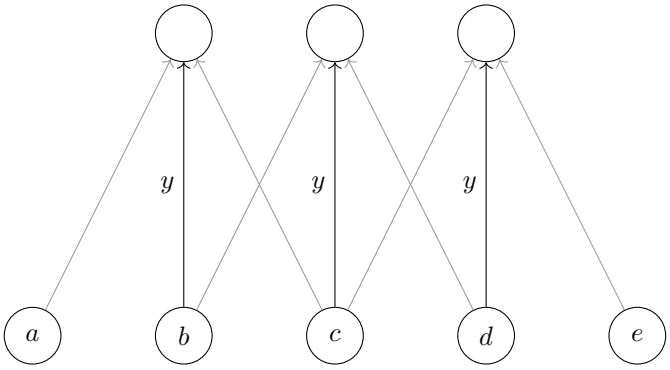


Figure 4: Schema for the convolutional layer performing the convolution shown in Figure 3. Each neuron represents one convolution. The schema shows the property of shared weights and sparse connectivity (Goodfellow et al., 2016). The black edges all have the same associated weight y, while one can see that there are much less edges compared to a dense layer shown in Figure 2.

18 elements, the kernel size (for the two spatial dimensions) times the three channels of each pixel. The shape of the feature map of that layer—if we assume "same" padding (see below)—would be $256 \times 256 \times 64$, so the next layer would have 64 channels.

There are two more notable concepts of convolutional layers: *stride* and *padding*. The former refers to skipping convolutions in order to reduce the computational cost at the expense of less exact feature extraction (patterns may not be detected by the model due to the increased inaccuracy). The latter is a way of dealing with vanishing spatial dimensions of the feature map if we only perform convolutions on "valid" inputs ($1 \leq i \leq m$ in Equation 8). "Valid" padding refers to the fact that the input has no padding. The feature map of the convolutional layer will have its kernel size minus one less neurons than its input (see Figure 4). "Same" padding would be to add enough zeros evenly above and below the valid input (along each spatial dimension) so that the feature map of the convolutional layer will have the same spatial dimensions as its input (see Figure 5 (Goodfellow et al., 2016).

Along convolutional layers, CNNs often have *pooling layers*. A pooling layer summarizes locally with the goal of making the CNN invariant to small translations of the input (Goodfellow et al., 2016), making the model less prone to *overfitting*—the state a model is in if it performs well on the training set but does not generalize well to unseen examples. *Max pooling*, for example, takes some local neighborhood of the input, exactly like a convolutional layer, and returns the maximum value of that neighborhood.

2.2 Benchmarking Deep Learning Systems for Computer Vision

In 2010 the annual (until 2017) ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was launched and has become the most famous benchmark for computer vision models, producing many well-known deep learning models like AlexNet in 2012 (Krizhevsky et al., 2012), VGG16 in 2014 (Simonyan and Zisserman, 2014) and the ResNet models in 2015 (He et al., 2015). The ILSVRC—like the name suggests—is based on the ImageNet data set consisting of more than 14 million images (Russakovsky et al., 2015). One task of the ILSVRC benchmark is image classification.

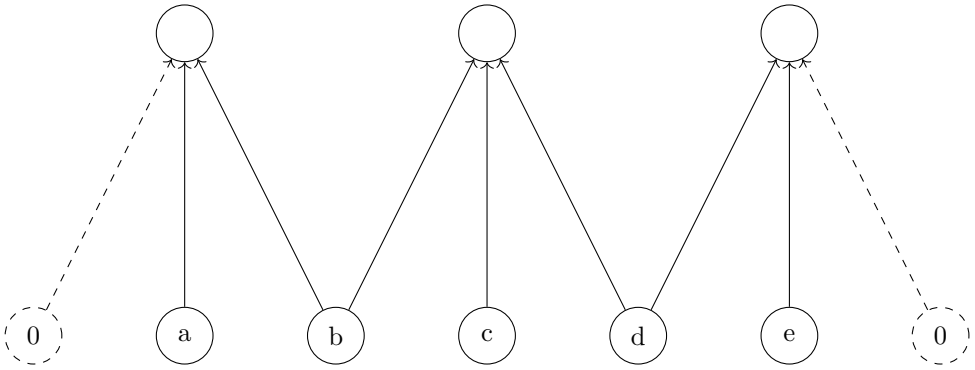


Figure 5: Example showing a layer with same padding and a stride of two.

During image classification the model is trained on 1000 categories (1.2 million images), without overlapping labels (each image has a single label, e.g. "dog") (Russakovsky et al., 2015). The top-1 ($y = \text{argmax } f(\mathbf{x})$) accuracy is measured on a test set of 150,000 images and winner is the model with the highest top-1 accuracy.

While a benchmark like the ILSVRC produces new insights into computer vision and keeps the community up to date on what is possible, deep learning has another issue on which a benchmark can shed light: training/inference speed of hardware and software systems. The MLPerf benchmark was developed to tackle this problem, so stakeholders can make informed decisions and to provide the industry—like hardware vendors, cloud providers and machine learning engineers—with a fair standard to rely on (Mattson et al., 2019). One task of the MLPerf training benchmark is training the ResNet-50 model (see below) on the image classification task from the ILSVRC 2012, until it reaches a top-1 accuracy of 74.9 percent. The wallclock time is measured and serves as the result for the benchmarked system (Mattson et al., 2019). Currently the fastest solution, from the latest MLPerf training benchmark v0.6, is Google's cloud system based on Tensorflow and one TPUv3 pod (1024 TPUv3s) (MLPerf, 2019; Stone, 2019). The benchmark we wanted to conduct in order to compare our implementation against others, is based on the image classification task of the MLPerf training benchmark, making it easy to compare SpiNNaker and our prototype to other state-of-the-art deep learning systems.

The winner of the image classification task of the ILSVRC 2015 was an ensemble of residual nets (ResNets) introduced in He et al. (2015). The ensemble generated a top-5 accuracy (true label in the five highest outputs of the ensemble) of 96.4 percent. ResNets are a revolution in the sense that they are not only better classifiers than previous models, they also can be significantly deeper (He et al., 2015). He et al. (2015) presents a 152-layer deep network, eight times deeper than a "very deep convolutional network" (VGG11–VGG19) (Simonyan and Zisserman, 2014; He et al., 2015). ResNets can be so deep, without losing their ability of convergence and without degradation (saturated accuracy and higher training error with increased depth) (He et al., 2015), by introducing residual blocks with shortcut connections (see Figure 6). He et al. (2015) hypothesizes that residual blocks ease the learning of the model. These shortcut connections do not increase the complexity of the model. No additional parameters are added to the model and nothing changes during backpropagation. Only the negligible operation where $\mathbf{x}$ is added to the output of the residual

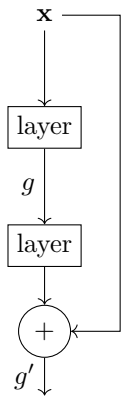


Figure 6: Schema of a residual block with two layers. $\mathbf{x}$ is added to the output of the last layer of the residual block, before the result is passed through its activation function g'.

block must be performed during the forward pass. He et al. (2015) shows comprehensive tests on how residual blocks decrease degradation by comparing ResNets against their counterparts with the same architecture, but without shortcut connections. The models without shortcut connections show a higher training error than their ResNet counterpart.

As stated above, the image classification task of the MLPerf training benchmark is to train ResNet-50 (50, because it has 50 layers) until it reaches a top-1 accuracy of 74.9 percent on the test set and to measure the wallclock time it took to reach that goal. Figure 7 shows an example block from the ResNet-50 model, while Figure 8 shows its architecture. The model takes a $224 \times 224 \times 3$ image as its input and first passes it through a convolutional layer with a relatively big kernel of 7×7 and a max pooling layer. Both times the spatial dimensions are halved by applying a stride of two, so the first residual block receives a $56 \times 56 \times 64$ feature map as its input. The model consists of multiple residual blocks. Some of them have a stride of two. Each time the input is halved that way, channels are doubled. This should keep computational cost the same for each block (He et al., 2015).

2.3 SpiNNaker as a Neuromorphic Computer Architecture

Spiking Neural Network Architecture (SpiNNaker, for short) is a massively parallel neuromorphic computer system designed to run spiking neural networks with up to one billion neurons (and a trillion synapses) in real-time (Painkras et al., 2013). As stated in Section 1, neuromorphic computing is the approach of developing hardware inspired by the biological nervous system (Mead, 1989). Today, neuromorphic computer architectures range from very fast and energy efficient but inflexible direct electronic models (neurons in hardware) (Indiveri et al., 2011) to very flexible but energy demanding systems based on common consumer hardware and software (neurons in software) (Plesser et al., 2007). SpiNNaker sits somewhere in between. On the one hand, flexibility is achieved by implementing neurons in software. On the other hand, speed is achieved by massive-parallelism and energy efficiency by using energy efficient processors, rather than fast ones (Furber and Bogdan, 2020).

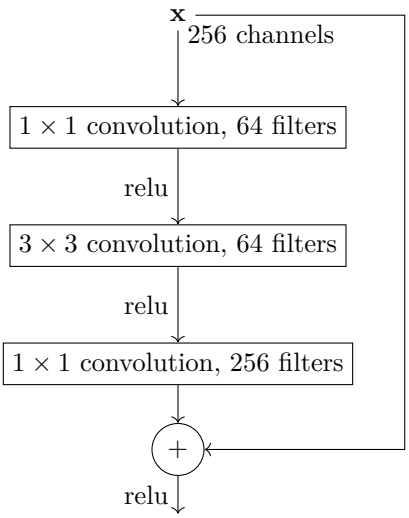


Figure 7: Example of a residual block in ResNet-50.

The SpiNNaker system's basic building block is the SpiNNaker chip, a multiprocessor chip consisting of 18 ARM968 cores (Furber and Bogdan, 2020) and a Network-on-Chip (NoC) system for communication between the cores (Furber and Temple, 2007). Each core can run up to 1000 spiking neurons, which communicate with each other over spikes—small packages with a maximum size of 72 bits which are sent over the NoC (Furber and Temple, 2007; SpiNNaker, 2020a). Each core has a 64 Kb DTCM—data tightly-coupled memory—for the application data and fast access to it. The 32 Kb ITCM stores the instructions executed by the core (Furber and Bogdan, 2020). All cores on a chip share access to 128 Mb SDRAM—synchronous dynamic random access memory—which has a higher capacity than DTCM but is also a lot slower (Furber and Bogdan, 2020; SpiNNaker, 2020b).

By today's standards 18 cores do not qualify as a massively-parallel system. Therefore, a SpiNNaker machine consists of multiple chips connected together in a 2D triangular (six edges per router instead of four) torus (Furber and Bogdan, 2020). The biggest SpiNNaker machine is the SpiNNaker1M supercomputer in Manchester with over one million cores. The SpiNNaker1M consists of 10 cabinets, each with five card frames holding 24 SpiNN-5 boards. A SpiNN-5 board has 48 SpiNNaker chips, which means the SpiNNaker1M has 1,036,800 theoretical cores, assuming no faulty cores (Furber and Bogdan, 2020).[2] Images of the SpiNNaker hardware can be found in Appendix A.

SpiNNaker is quite different from common deep learning accelerators, like general purpose graphics processing units (GPGPUs) or Google's TPU (Jouppi et al., 2017). Common deep learning libraries like Tensorflow (Abadi et al., 2015), Keras (Chollet et al., 2015) or PyTorch (Paszke et al., 2019) implement deep learning on a layer basis, which means by multiplying *tensors* (nD arrays), rather than implementing deep learning on a neuron level (Goodfellow et al., 2016). Therefore, the common industry approach to building hardware accelerators for deep learning is to facilitate

2. During the introduction of the SpiNNaker1M 1,010,285 cores were working (UoMCompsci, 2019).

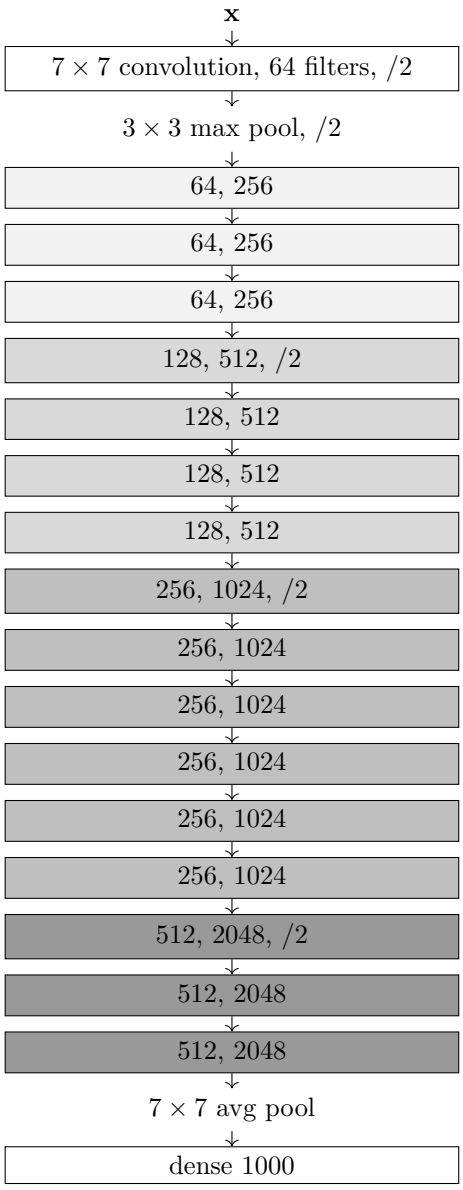


Figure 8: Schema of ResNet-50. Each block in the middle represents one residual block shown in Figure 7. The first number shows the amount of filters the first two layers of a block have, while the second number shows the filters of the last layer of the block. /2 indicates that a stride of two is applied (spatial dimensions are halved—each convolutional and pooling layer has "same" padding). Whenever the filters are doubled (indicated by the varying grey scales), the shortcut layer is linearly projected to match the higher channels.

fast matrix multiplication, which represents the majority of computation needed for training and inference. The leading systems, when it comes to throughput and speed according to the MLPerf training benchmark (MLPerf, 2019), are Google's TPU (Jouppi et al., 2017) and NVIDIA's GPU architecture Volta (Durant et al., 2017). Volta's successor Ampere was released in 2020, which, according to NVIDIA, is much more powerful (Krashinsky and Giroux, 2020). Both architectures leverage instruction level parallelism, sacrificing significant chip space for specialized units performing a fused multiply-add-accumulate matrix operation (MAC). NVIDIA calls these units tensor cores. The Tesla V100 has 640 tensor cores, each performing one 4×4 MAC per clock cycle, a theoretical peak performance of 125 terraflop/s (Markidis et al., 2018). The TPUv1 comes with 256×256 MACs performing 8 bit (unsigned) integer operations (Jouppi et al., 2017). The TPUv2 added support for mixed precision floating point operations (16 bit multiply with 32 bit add and accumulate, same as the tensor core of the Volta architecture) (Kennedy, 2017; Markidis et al., 2018). The SpiNNaker cores do not have a MAC unit, being designed to run spiking neurons efficiently, rather than lots of matrix multiplications. That means the prototype presented in Section 4 must put more focus on leveraging SpiNNaker's massive parallelism, rather than relying on fast instruction level parallelism. For example with optimized domain decomposition and smarter algorithms than matrix multiplication.

3. Related Work

Like Gomes (2017) states, implementing deep learning on neuromorphic chips has been a goal for some time. This section will outline two approaches to implementing deep learning models on neuromorphic hardware. One being the SNN toolbox (Rueckauer et al., 2017), the other being an implementation of CNNs on IBM's TrueNorth system (Esser et al., 2016).

The SNN toolbox takes pre-trained deep learning models and translates them into spiking neural networks. Its front-end supports a wide range of different input formats from various deep learning libraries, including Keras, Tensorflow, PyTorch and Caffe (Jia et al., 2014), while its back-end supports different spiking neural network simulators like Brian2 (Stimberg et al., 2019). The back-end also supports the simulator-independent language PyNN (Davison et al., 2009), enabling running the converted models on SpiNNaker, which supports PyNN as its front-end for spiking neural networks. Furthermore, direct mappings to other neuromorphic computers like Intel's Loihi (Davies et al., 2018) are supported by the back-end (SNN toolbox, 2020). The SNN toolbox supports complex CNNs like VGG16 or Inception-v3 (Szegedy et al., 2015; Rueckauer et al., 2017). Rueckauer et al. (2017) shows that using the converted version of LeNet (LeCun et al., 1989) on the MNIST data set (LeCun et al., 2020) and BinaryNet (Courbariaux and Bengio, 2016) on CIFAR-10 (Krizhevsky, 2009) requires half the operations of the original CNNs without considerable loss in accuracy. Unfortunately, for bigger problems, namely VGG16 and Inception-v3 on the ImageNet data set, the converted models have a much lower accuracy than their original counterpart (63.9 percent accuracy for the original VGG16 and only 49.6 for the converted model) (Rueckauer et al., 2017). Another caveat of the SNN toolbox is the fact that it only supports inference and not training. Training is the far more complex task computationally.

IBM's TrueNorth neuromorphic architecture has the goal of achieving energy efficiency and performance through scalability, similar to SpiNNaker. Esser et al. (2016) presents Eedn (energy-efficient deep neuromorphic networks). Eedn is an approach to generating CNNs on the TrueNorth system, enabling both inference and training. TrueNorth—unlike SpiNNaker—uses one bit spikes (Esser et al., 2016), which means it is substantially different from contemporary consumer hardware

and deep learning accelerators. Eedn is needed to translate the CNN in order to make it run on TrueNorth. SpiNNaker on the other hand, as stated above, is only a collection of low power ARM cores connected over a NoC. Spikes on SpiNNaker are communicated with small multicast packets (up to 72 bits, see Section 2.3) (Furber and Bogdan, 2020). These packets can be used to transfer any information from one core to another. This makes it much easier to implement deep learning on SpiNNaker, because focus lies more on how to deconstruct the model and map it onto the cores, rather than having to translate the model into a SpiNNaker-specific format. Nonetheless, Eedn models show promising results. Esser et al. (2016) presents tests on 6 well known, industry-strength data sets with the Eedn models having approximately the same accuracy as the original models. The throughput of the TrueNorth system is promising as well. Esser et al. (2016) shows that TrueNorth is able to process between 1,200 and 2,600 $32 \times 32 \times 3$ images per second.

4. Deep Learning on SpiNNaker

This section will describe the developed deep learning prototype in detail. While Section 2.3 gave a short overview over the SpiNNaker hardware, Section 4.1 will present a short introduction to the SpiNNaker software toolchain and programming model. Section 4.2 will present the architecture of the prototype. Lastly, Section 4.3 will describe the hardships and problems encountered and mistakes made during the development process.

4.1 The SpiNNaker Programming Model

Parallel programming is hard (Lee, 2011), especially on novel hardware architectures like SpiNNaker (Brown et al., 2015). SpiNNaker provides layers of software abstractions over the hardware to make programming and exploiting the capabilities of it as easy as possible (Furber and Bogdan, 2020).

The SpiNNaker hardware is designed to tackle problems which can be decomposed into many small, autonomous units without a central computational overseer (Brown et al., 2015). These problems are commonly known as embarrassingly parallel problems (Foster, 1995). The software toolchain lets the user describe their program as a graph. Each vertex of the graph represents one unit of computation (e.g. a bunch of spiking neurons or a perceptron in our case) and directed edges represent the communication between the units (Furber and Bogdan, 2020).

There are two type of graphs, application graphs and machine graphs. A vertex in the machine graph, referred to as a machine vertex, is directly mapped to a single SpiNNaker core. An application graph is an abstraction over a machine graph. Application vertices have a number of atoms, each being an atomic unit of computation. The atoms of an application vertex are distributed onto machine vertices, such that each machine vertex contains a disjunct subset of the atoms of the corresponding application vertex. This makes programming and scaling easier and also facilitates proper resource exploitation. For example by mapping 1,000 small spiking neurons—atoms of a neuron population represented as a application vertex—onto a single core, instead of having them distributed across 1,000 cores, which would be the case if they were to be implemented directly as machine vertices (Furber and Bogdan, 2020). For the prototype we used a machine graph, since it is easier to implement. Less code is required and the difficulties of multiplexing and demultiplexing are avoided. The development process was guided by the UNIX rule of optimization: prototype before polishing (Raymond, 2003), or in Donald Knuth's words: "premature optimization is the root of all evil" (Knuth, 1974).

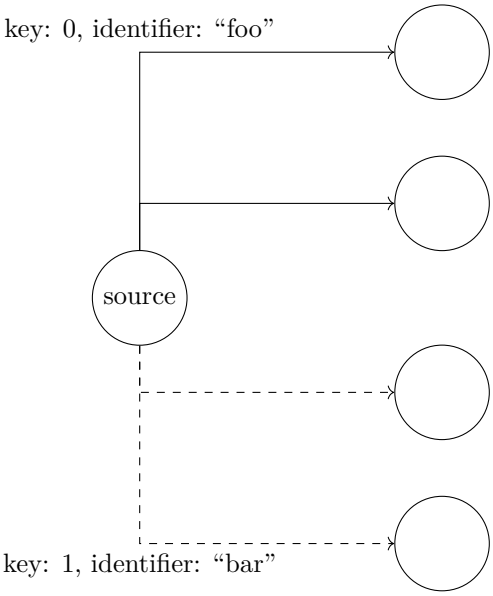


Figure 9: Example of a machine vertex “source” connected to four other machine vertices over two outgoing edge partitions.

The SpiNNaker toolchain is mostly written in the Python programming language. The SpiNNaker machine is connected to a host device via Ethernet (Rowley et al., 2019). In order to execute a program on SpiNNaker, one writes a Python script which generates a graph. It must be either an application graph or a machine graph. Machine vertices cannot be added to an application graph and vice versa. The toolchain—running on the host which also executes the script generating the graph—takes a lot of boilerplate away from the user and takes care of the execution of the program on the connected SpiNNaker machine. The toolchain goes through a stage of mapping the graph onto the available cores, before data generation (where e.g. parameters of the vertex are loaded into SDRAM, so that the vertex on the machine is able to access them) and finally loading and running the application (Furber and Bogdan, 2020).

Each vertex is represented as a Python object—instantiated from the appropriate classes—and has an associated binary—the program to be executed by the machine (Furber and Bogdan, 2020). The source code of the binary to be executed on the machine is written in the C programming language and compiled with the gcc compiler from the GNU ARM embedded toolchain (ARM, 2020; Rowley et al., 2019). Machine vertices are not common C programs. They do not own the control flow but instead are event-based, like the ECMAScript programming language (ECMA, 2020). The SpiNNaker1 API provides the operating system executing the vertex and serves as the mechanism for registering software callback functions, triggered when a certain event occurs (Furber and Bogdan, 2020). The two events used by the vertices of this prototype were: (i) receiving a packet and (ii) a periodic update event, called every x microseconds. During testing, we set the update event to be called every five milliseconds, which is rather slow but true to our design philosophy of prototyping before polishing.

Machine vertices communicate with each other via MC (multicast) packets. A MC packet has two or three segments: (i) one control byte, (ii) 4 byte key and (iii) optionally 4 byte payload. This makes a MC packet either 40 or 72 bits long (Furber and Bogdan, 2020). A MC packet is sent via the directed edges between vertices. Each edge has an associated outgoing edge partition. An outgoing edge partition has one source vertex and n destination vertices and—in the case of the machine graph—one unique routing key (allocated by the toolchain). The routing key—a 32 bit unsigned integer—is unique in the sense that no other outgoing edge partition will have the same key. Otherwise routing would fail, because packets would be send to the wrong destinations or dropped when no matching router entry for the key is found. If the source vertex sends a MC packet it uses the key of the outgoing edge partition. The packet will reach all destination vertices of the outgoing edge partition. A vertex can have multiple outgoing edge partitions, as is shown in Figure 9 (Furber and Bogdan, 2020). If the source vertex from Figure 9 sends a MC packet with key zero, the two upper vertices will receive the packet, whereas with key one the two lower vertices would receive the packet.

The toolchain offers support for live IO, enabling external devices (like robots, or in our case the host) to interact with the application running on SpiNNaker. Interaction happens, again, via MC packets[3] by adding extra vertices for input and output to the graph. Live input is enabled by the `ReverseIPTagMulticastSource` (RIPTMCS) machine vertex and live output by the `LivePacketGatherer` (LPG) (Furber and Bogdan, 2020). The toolchain provides a `LiveEventConnection` for the external device, which supplies the appropriate abstractions over the networking. Like the SpiNNaker1 API, the `LiveEventConnection` provides an event-based interface with callbacks.

4.2 The Prototype

The underlying assumption made for developing the prototype was, that because SpiNNaker was designed to run spiking neurons, it would be a natural fit to implement deep learning on a neuron level as well. Besides that, neurons are an easy-to-understand abstraction over the mechanisms of deep learning and rather straightforward to implement. This design decision was made against the trend of both deep learning research and state-of-the-art deep learning libraries. The former more commonly abstracts over layers while the latter implements deep learning as a computational graph (Goodfellow et al., 2016). Problems with this assumption are discussed in Section 4.3. The API of the prototype was designed to resemble the API of the Keras deep learning library (Chollet et al., 2015). An example comparison can be seen in Listing 1.

The prototype exposes the `Model` class to the user (see Listing 1, line 5 and 18). The model is the main interface to SpiNNaker and—as the name clearly suggests—functions as the representation of a deep learning model. It is designed to be used the same way as Keras's `Sequential` model (see Listing 1, line 12). The second user interface is the layers—the building blocks of a model. Layers are added sequentially to a model instance by calling the model's `add` method (see Listing 1, lines 19–22). A layer has at most one preceding and one succeeding layer (see Figure 10). While this suits most common needs, modern deep learning models, like the inception nets (Szegedy et al., 2014), have layers connected to multiple preceding and succeeding layers. Also the shortcut connections of the ResNets (see Section 2.2) are not straightforward to implement with the sequential API. Keras

3. MC packets are the abstraction presented to the user. In fact, communication between the host and the SpiNNaker machine are handled by the EIEIO (external internal event input output) protocol (Rast et al., 2015). The EIEIO protocol sits on-top of the SpiNNaker-internal SDP (SpiNNaker datagram protocol), which itself sits on-top of UDP (Furber et al., 2014).

```python
import tensorflow as tf
import numpy as np

# prototype library
from spiDNN import Model
from spiDNN.layers import Input, Dense

# random test set
X = np.random.rand(500, 64)

# the keras model
keras_model = tf.keras.Sequential()
keras_model.add(tf.keras.layers.Dense(128, input_shape=(64,)))
keras_model.add(tf.keras.layers.Dense(128, activation='relu'))
keras_model.add(tf.keras.layer.Dense(10, activation='softmax'))

# the equivalent model for SpiNNaker
spinn_model = Model()
spinn_model.add(Input(64))
spinn_model.add(Dense(128))
spinn_model.add(Dense(128, activation='relu'))
spinn_model.add(Dense(10, activation='softmax'))

# this call ensures both models have the same parameters
model.set_weights(keras_model.get_weights())

# predict the results for the random test set (with random weights)
p_ = kmodel.predict(X)
p = model.predict(X)

error = np.absolute(p - p_)

# difference in prediction can happen, due to floating point errors
assert np.amax(error) < 1e-4
```

Listing 1: Example code comparing inference with Keras to inference with the prototype. The code would result in a model akin to the one shown in Figure 10.

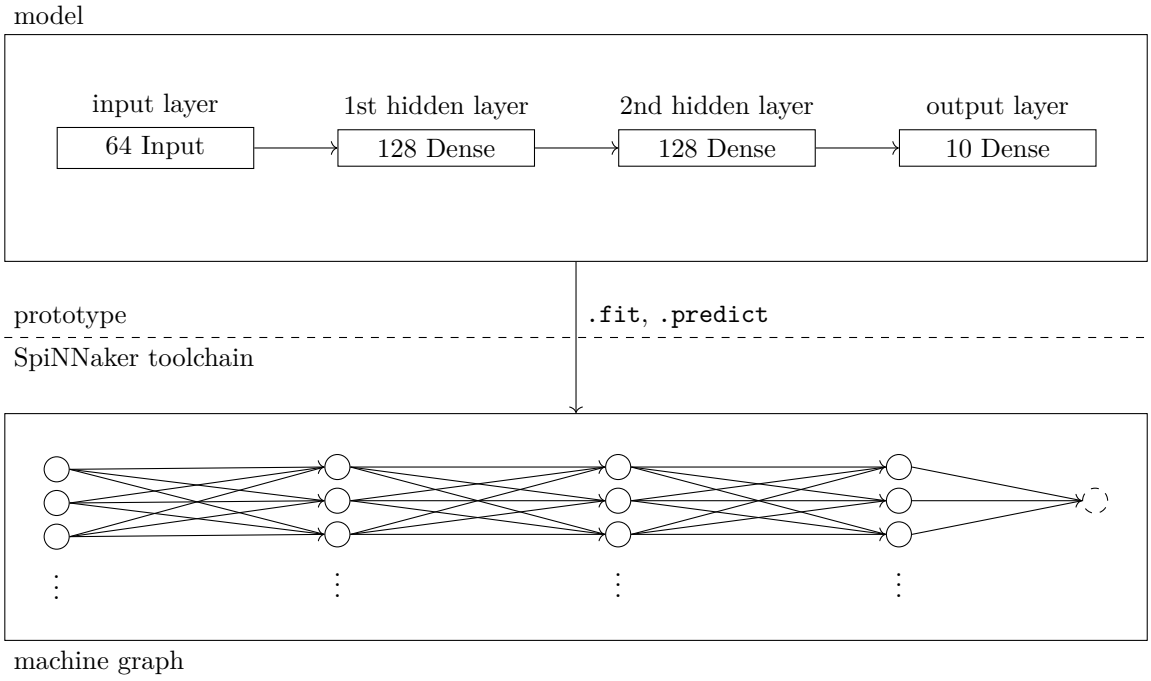


Figure 10: Illustration of how a machine graph is generated by the prototype. The dashed circle represents an auxiliary machine vertex, in this case the LPG. This machine graph would be generated when the `predict` method of the model is called.

therefore exposes another API, its functional API. The prototype was not developed to this stage and only supports sequential models.

The model stores the trainable parameters (the weights of the layers), which can be accessed via the `set_weights` and `get_weights` methods of the model class (see Listing 1, line 25). The parameters are generated during the `add` call by the added layer, since weights are different for different layer types and can depend on the preceding layer. For example, a dense layer (see Section 2.1) generates the weight matrix $\mathbf{W} : n \times m$ and the bias m-vector. n is the number of neurons in the preceding layer, m the number of neurons of the dense layer. Convolutional layers do not depend on the previous layer for their weights. For example a 1D convolutional layer generates a 3D array $\mathbf{W} : \mathit{kernel\ size} \times \mathit{channels} \times \mathit{filters}$ and a bias $\mathit{filters}$-vector. $\mathbf{W}$ are the kernels for each filter (see Section 2.1). The kernel of a 2D convolutional layer simply has one more spatial dimension, so the 2D layer would generate a 4D array as weights. The input layer does not generate any weights. The format of the weights is again inspired by Keras and enables seamless interoperability between a SpiNNaker deep learning model and the subset of the Keras API supported by the prototype (see Listing 1, line 25). Having our prototype so closely resemble Keras had the great advantage of enabling precise integration testing. We could simply build two equal models, one in Keras and one with our prototype, and compare their outputs against each

other (see Listing 1, lines 29–36). The same goes for testing backpropagation, where we could simply train both models and compare the updated weights (see Listing 2).

Supported layers are: (i) input, (ii) dense and (iii) 1D convolutional layers. A flatten layer (used for flattening a feature map into a one dimensional vector so it can be processed by a dense layer) is implicitly implemented (see Listing 3). Besides layers, the prototype supports the following activation functions:

- identity (default when activation unspecified):

$$f(x) = x \tag{9}$$

- tanh:

$$f(x) = tanh(x) \tag{10}$$

- sigmoid:

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{11}$$

- relu (rectified linear unit):

$$f(x) = max(0, x) \tag{12}$$

- softmax:

$$f(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{N} \exp(x_j)} \tag{13}$$

The only supported optimizer (algorithm for training the weights) is gradient descent with a constant learning rate. Gradient descent is probably the most common optimization algorithm for deep learning (Goodfellow et al., 2016). Supported loss functions, which are optimized, are:

- mean squared error:

$$L(y, \hat{y}) = 1/k \sum_{i=1}^{k} (y_k - \hat{y}_k)^2 \tag{14}$$

- categorical cross-entropy:

$$L(y, \hat{y}) = -\sum_{i=1}^{k} y_i \log \hat{y}_i \tag{15}$$

- binary cross-entropy (y and $\hat{y}$ must be scalar):

$$L(y, \hat{y}) = -y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \tag{16}$$

```
import tensorflow as tf
import numpy as np

# prototype library
from spiDNN import Model
from spiDNN.layers import Input, Dense

# random training set
X = np.random.rand(500, 64)
y = np.random.rand(500, 10)

# the keras model
keras_model = tf.keras.Sequential()
keras_model.add(tf.keras.layers.Dense(128, input_shape=(64,)))
keras_model.add(tf.keras.layers.Dense(128, activation='relu'))
keras_model.add(tf.keras.layer.Dense(10, activation='softmax'))

# the equivalent model for SpiNNaker
spinn_model = Model()
spinn_model.add(Input(64))
spinn_model.add(Dense(128))
spinn_model.add(Dense(128, activation='relu'))
spinn_model.add(Dense(10, activation='softmax'))

# this call ensures both models have the same parameters
model.set_weights(keras_model.get_weights())

# train both models
#
# unlike the prototype, Keras compiles its models, before they can be trained.
# Since the prototype does not offer a optimizer interface yet, it does not
# have a compile method
kmodel.compile(loss='mean_squared_error',
               optimizer=tf.keras.optimizers.SGD(learning_rate=0.01))
kmodel.fit(X, y, epochs=5, batch_size=32, shuffle=False)

model.fit(
  X, y, loss='mean_squared_error', epochs=5, batch_size=32, learning_rate=0.01)

error = [x - x_ for x, x_ in zip(model.get_weights(), kmodel.get_weights())]

# difference in weights can happen, due to floating point errors
for e in error:
    assert np.amax(np.absolute(e)) < 0.1
```

Listing 2: Example code comparing training with Keras to training with the prototype. The code would result in a model (not the machine graph) akin to the one shown in Figure 10.

```
import tensorflow as tf
import numpy as np

# prototype library
from spiDNN import Model
from spiDNN.layers import Input, Dense, Conv1D

# 64 features, each with 4 channels
input_shape = (64, 4)

# random test set
X = np.random.rand(500, *input_shape)

# the keras model
keras_model = tf.keras.Sequential()
# this layer has 20 filters and a kernel size of 3. Valid padding and a
# stride of 1 are default
keras_model.add(tf.keras.layers.Conv1D(20, 3, input_shape=input_shape))
keras_model.add(tf.keras.layers.Conv1D(5, 3, activation='relu',
                                       padding='same', strides=2))
keras_model.add(tf.keras.layers.Flatten())
keras_model.add(tf.keras.layer.Dense(10, activation='softmax'))

# the equivalent model for SpiNNaker
spinn_model = Model()
spinn_model.add(Input(*input_shape))
spinn_model.add(Conv1D(20, (3,)))
# The feature map of this layer is implicitly flattened,
# because the next layer is a dense layer
spinn_model.add(Conv1D(5, (3,), activation='relu',
                       padding='same', stride=2))
spinn_model.add(Dense(10, activation='softmax'))

# this call ensures both models have the same parameters
model.set_weights(keras_model.get_weights())

# predict the results for the random test set (with random weights)
p_ = kmodel.predict(X)
p = model.predict(X)

error = np.absolute(p - p_)

# difference in prediction can happen, due to floating point errors
assert np.amax(error) < 1e-4
```

Listing 3: Example code comparing inference with 1D CNNs in Keras to inference with the prototype.

1D convolutional layers have support for same and valid (default) padding and they can be strided (see Section 2.1 and Listing 3, lines 30 and 31).

As stated above, a deep learning model is decomposed into neurons in order to execute it on SpiNNaker. Neurons are part of the internal APIs of the prototype and not exposed to the user. The neuron of a dense layer is a perceptron, which can be seen in Figure 1. Each neuron of a convolutional layer performs one single convolution operation with all filters of the layer. For example, when we look at Figure 3, $ax + by + cz$ would be one convolution with one filter. If the layer has a second filter with a kernel w, v, u, the neuron would produce a second value $aw + bv + cu$. The input a, b, c stays the same. A convolutional layer is schematized in Figure 4.

Figure 10 shows how to operate the prototype. The user defines the sequential deep learning model, like described above. Only when the inference or training operations are called is the model translated into a SpiNNaker machine graph and executed on the connected machine. For inference one has to call the `predict` method of the model (see Listing 1, line 29) and for training the `fit` method (see Listing 2, lines 37 and 38).

Algorithm 1 : `predict` method

1: create extractor layer
2: reset layers
3: setup SpiNNaker
4: initialize machine vertices
5: establish forward pass connections
6: establish connection between output layer and extractor
7: setup live IO
8: execute model on SpiNNaker (run forever) {the live IO connection stops the execution}
9: stop the SpiNNaker machine
10: close live IO
11: **return** predictions {predictions collected by live IO}

Algorithm 1 shows what happens during the `predict` method. First, an auxiliary layer for the live IO is created, the extractor layer. The extractor layer has a single machine vertex, a LPG instance (see Section 4.1) for streaming the predictions off SpiNNaker, back to the host. The LPG is also shown as the auxiliary vertex in Figure 10. The first layer—which must always be an input layer—handles streaming the data onto the machine. Neurons of a input layer are simple wrappers around the RIPTMCS machine vertices provided by the SpiNNaker toolchain (see Section 4.1).

The layer interface specifies a `reset` method, which is called for each layer in Algorithm 1, line 2. This method resets the neurons of a layer (simply deletes them). If, for example, the model was trained before the call to the `predict` method—probably the most common workflow in deep learning—the neurons from the training graph would still exist. Beyond the training these neurons are meaningless, because they were part of a different run on SpiNNaker and must be deleted (when the toolchain is reset it deletes its machine graph instance, but the vertex objects remain). They could be deleted right before the `fit` or `predict` method exits, but for unit testing it is convenient to still have the neurons beyond a run on SpiNNaker.

After the machine and the SpiNNaker toolchain are set up, the machine vertices (neurons) are initialized (see Algorithm 1, line 4). The layer interface always sits between the model and the neurons. The model only ever calls methods of the layers. This is a design pattern in software engineering called the layered pattern (not to be confused with the layers provided by our prototype)

(Morlion, 2018). This design pattern is a common way to abstract—e.g. TCP/IP or the OSI model (Tanenbaum and Wetherall, 2013)—and keeps code complexity down, makes ownership clear and helps writing well-defined interfaces. For the initialization of the neurons, the layer interface provides a method `init_neurons`. This method generates the neurons, which are stored in the `neurons` property of each layer (a Python list). Furthermore, the `init_neurons` method adds the neurons to the machine graph of the SpiNNaker toolchain.

After the neurons are created and initialized, the connections for the forward pass are generated (see Algorithm 1, lines 5 and 6). The layer interface offers the `connect_incoming` and `connect_outgoing` functions for this. Incoming and outgoing refer to the called layer, respectively. For example, the forward pass is built with calling `connect_incoming` for each hidden layer, the output layer and the auxiliary layer for extracting the predictions. Each of these layers is connected with the preceding layer. All layers except convolutional layers connect each of their neurons with every neuron of the layer connected with. Connecting convolutional layers is more complicated, because they also need to take care of stride and padding.

The last thing to be set up, before execution can start, is the live IO connection for streaming the observations onto the board and the predictions off it (see Algorithm 1, line 7). The communication protocol chosen was the most naive possible solution, staying true to our development principle (see above). We called the protocol ping-pong protocol, because the communication pattern resembles table tennis, the host being one player, the SpiNNaker machine the other. The host always sends data onto the board (ping) and receives some sort of result (pong). After receiving the pong, another ping is sent by the host or the execution is stopped when no more data is there to send. During inference the observations are streamed onto the board as the ping events. The deep learning model processes each observation and returns the predictions of the output layer as the pong event. Each prediction is collected by the live IO callback function for receiving data and returned from the `predict` method (see Algorithm 1, line 11).

The ping-pong protocol is easy to implement and easy to reason about. Its main disadvantage is performance. One can easily see the problem. Only a single layer is processing an observation at a time. All other layers wait. This is a huge waste of time, especially for very deep model architectures. A possible solution to this problem is discussed in Section 5.

As stated above, weights are actually owned by the model, not the layer objects. The weights are directly injected into the neurons during their generation and initialization. Conceptually, a perceptron (neuron of a dense layer) owns the weights associated with its incoming connections. The filters of the convolutional layers are simply shared between all neurons of the layer, so each neuron has a copy.

The received packets have to be matched somehow to the weights the payload of the packet is multiplied by. All packets sent by the different components of the prototype are with payload, so 72 bit big. Each packet consists of a control byte and two words, key and payload (see Section 4.1). Each neuron knows how many neurons in the previous layer it is connected to and knows their keys within a certain partition. The simple MLP shown in Figure 10 only has a single partition—the "forward" partition.[4] During data generation (see Section 4.1) the data structured generated by the SpiNNaker toolchain can be queried and the machine graph traversed in order to find out the corresponding routing keys. The routing keys of the outgoing edge partitions of the neurons from the previous layer are collected. The keys are guaranteed to be consecutive. For example, the first neuron of the input layer of the MLP from Figure 10 (64 neurons big) would have zero as its key.

4. Not actually one partition, but lots of outgoing edge partitions with the same identifier (see Section 4.1). Below, a partition will always refer to a set of outgoing edge partitions with the same identifier.

The second neuron one. The first neuron of the first hidden layer would have 64 as key and so forth. How we changed the key allocation process to make this guarantee will be discussed below.

After having collected the keys of the neurons from the previous layer, the *minimum key* is determined and passed as argument to the machine vertex running on SpiNNaker. The weights of the neuron are also passed as an argument to the machine vertex. The weights are a simple array of floats. Now if a packet is received, the weight corresponding to the connection is simply determined by indexing the weight array with the index being the key of the received packet minus the minimum key. The same procedure works for the backward pass as well. For convolutional neurons it is somewhat more complex, because they receive multiple channels from their preceding neurons. The index is therefore determined with an additional array of channel counters—for each connection a channel counter is created. A channel counter is incremented each time a value is received from its corresponding connection. The channel counter is then additionally used to determine the correct index of the weight the payload of the packet must be multiplied with.

Algorithm 2 : `receive_forward`(key, payload) event of a perceptron machine vertex

```
1: index := key - minimum key
2: signals[index] := payload
3: receive counter += 1
```

Algorithm 2 shows the receive event of a perceptron machine vertex. Unlike stated above, the payload is stored in an array called "signals", instead of multiplying it directly with its corresponding weight and summing it up in the perceptron's potential (see Algorithm 3, lines 2–6). If only inference is done with the deep learning model, this could be optimized to save precious memory. Unfortunately, the received signals must be stored for computing the gradients during the backward pass. To keep development simple, plain machine vertices for inference and their trainable counterparts were kept as close to each other as possible. Another point in favor of storing the signals, rather than processing them directly in the receive event, is the fact that SpiNNaker supports floats only in software and floating point operations are therefore expensive (they take a lot of cycles) (Furber and Bogdan, 2020). Receive events must be processed as fast as possible to keep pressure off the router, which blocks until the packet is received. Blocking the router causes back-pressure, which leads to packet loss (discussed in Section 4.3).

Algorithm 3 : `update`() event of a perceptron machine vertex

```
1: if receive counter == N then
2:     potential := 0
3:     for (i = 0; i < N; i++) do
4:         potential += signals[i] · weights[i]
5:     end for
6:     potential += weights[N] {add the bias}
7:     potential := g(potential) {apply activation function}
8:     send(forward key, potential) {send potential to next layer}
9:     receive counter := 0
10: end if
```

The prototype must support multiple partitions. For basic inference—without a softmax layer (see below)—it is only one partition, as stated above. For the backward pass during training on the

other hand, neurons are not only connected to the neurons of the succeeding layer, but also to the ones of the preceding layer. If these backward connections were also part of the "forward" outgoing edge partition of the neuron, the potential would reach the neurons in the preceding layer as well. In the other direction, the error passed backwards would reach the neurons in the succeeding layer. It would be hard to determine if a packet is a potential (meant to be sent forward) or an error (meant to be sent backward). Furthermore, this would mean that a lot of MC packets would simply be dropped by the receiving vertex, because only one set of neurons handles potentials, while the other only handles errors. This would put unnecessary strain onto the communication fabric.

Not only does the backward pass needs its own partition, there is also an activation function which behaves differently to the other activation functions implemented. The activation function is the softmax activation function (see Equation 13). Softmax depends on the output of the other neurons of the same layer, producing a normalized version of the potential. Softmax is a common activation function for the output layer, making the output a probability distribution over the K output neurons (Goodfellow et al., 2016).

The first implementation of the machine vertices handled softmax in the next layer or on the host, if the output layer had softmax as activation function. Most certainly faster than the current version, it led to awkward code with more branches. The amount of overhead in the code and the fact that information of one layer (its activation function) has to be shared with another layer was deemed too complex at this stage of the prototype. The softmax activation function was implemented early in the development process, when we tried to avoid code complexity at all cost. The current version of the prototype handles softmax via an intra-layer partition, the "softmax" partition. The neurons of a softmax layer are fully connected. Instead of passing the potential forward (see Algorithm 3, line 8) it is passed sideways to all neurons in the same layer. The potentials received on the connections of the softmax partition are summed in a variable "denominator". When all packets from one pass on the softmax partition are received (all potentials from the neurons in the same layer), the potential of the neuron is divided through the denominator and passed forward to the neurons of the next layer (or the host, respectively).

When we first made the change to add a second partition (softmax was implemented before backpropagation) we realized that the SpiNNaker toolchain does allocate its keys per machine vertex. For example, if we assume the first hidden layer of the deep learning model in Figure 10 to have softmax as its activation function, the keys for the two partitions of the first neuron would be 64 (forward partition) and 65 (softmax partition). The problem with this is that we need our partitions to be consecutive (see above). Luckily the toolchain provides a way to override the default key allocator.

We have overridden the default key allocator with a first-touch global partition key allocator. Every time a directed edge is added to the machine graph (e.g. as is done in Algorithm 1, lines 5 and 6) it is mapped to its place in the key space, such that partitions continue to be consecutive (global partition over the outgoing edge partitions of each neuron). First-touch has two meanings. On the one hand it means that the first partition will have the lower keys in the key space. On the other hand the source neuron of the directed edge added will have the next biggest key of that partition. This way we guarantee that, if the connections from the first neuron of a layer are created before those of the second neuron, the second neuron will have the key of the first neuron plus one.

The intra-layer connections for the softmax partition are established during initialization of the neurons (see Algorithm 1, line 4). This means they are touched by the toolchain prior to the connections of the forward partition (see Algorithm 1, lines 5 and 6). If we look at our example from above, where the first hidden layer of the deep learning model shown in Figure 10 has softmax

as its activation function, the keys of the softmax partition would range from 0 to 127, inclusively (one key per neuron in the first hidden layer). The forward keys for the input layer would be 128 to 191, the forward keys for the first hidden layer would be 192 to 319 and so forth. Our first-touch global partition key allocator scales up to an arbitrary amount of partitions (partitions are obviously bound by the key space being 2^{32} items big), so it is no problem to handle all four partitions of the prototype: (i) the forward partition, (ii) the softmax partition, (iii) the backward partition and (iv) the previously unmentioned "kernel update" partition. The kernel update partition works the same way as the softmax partition. It is an intra-layer partition to update the filters of a convolutional layer during the backward pass, since the weights of the convolutional layer are shared between its neurons (see Section 2.1).

After implementation of the forward pass and the activation functions, the backward pass was implemented, enabling training a MLP. The forward pass of trainable neurons works the same with plain inference neurons, thanks to our development choices (see above). Training a deep learning model on SpiNNaker is done via the `fit` method. The method takes a training set and its labels as two input parameters. Furthermore, one has to define the parameters of the training session. These parameters are: (i) the loss function to be optimized, (ii) the number of iterations over the whole training set (the epochs), (iii) the batch size for gradient descent and (iv) the learning rate (see Section 2.1). The parameters of the training session are passed to the trainable neurons on the board. The training set and its labels are streamed onto the board, same as inference (without the labels).

The structure of the machine graph for a trainable deep learning model is more complicated than that of simple inference models. Not only due to the added backward connections, but also the auxiliary layers are more complex. First, besides the training data, the labels have to be streamed onto the board as well. This is simply done by adding another input layer to the layer representation of the model. The labels are needed to compute the loss. In order to compute the loss, another layer type was implemented, the loss layer. An instance of a loss layer is connected to the output layer and the label input layer. The loss layer only has a single neuron which computes the loss, based on the outputs and the labels, and passes its derivatives backwards to the neurons of the output layer. The overall loss is also streamed off the board. The overall loss is displayed to the user in the console, together with a progress bar. The input layer has no parameters. Therefore it can be excluded in the backward pass. The first hidden layer is connected—backwards—to the LPG used for streaming data off of SpiNNaker. Its outputs are used as the pong event. Once the first hidden layer has computed its gradients the backward pass is complete and the next forward pass can begin, so the next example is streamed onto the board. All the other layers are connected backwards to their preceding layers.

Algorithm 4 gives an overview of how a deep learning model is trained on SpiNNaker. The prototype only supports streaming each example individually. The observation $\mathbf{x}_i$ is passed through the model and the loss is computed based on the outputs and the labels by the loss layer (see Algorithm 4, lines 3 and 4). For example, let the loss function be mean squared error (see Equation 14). The loss is partially derived for each output neuron. The output of the ith neuron is computed as $\hat{y}_i = g(h(f^{(l-1)}))$, g being the activation function, $h(\mathbf{x}) = \mathbf{x} \cdot \mathbf{w} + b$. The partial derivatives in this case are given by:

$$\frac{\delta L}{\delta \hat{y}_i} = 2(\hat{y}_i - y_i). \tag{17}$$

Algorithm 4 : high-level overview of training a deep learning model on SpiNNaker

```
1: for 1,…,epochs do
2:   for x_i ∈ X, i = 1, ..., |X| do
3:     forward pass x_i
4:     compute loss
5:     pass error backwards and compute gradients
6:     if (i mod batch size) == 0 || epoch complete then
7:       update weights with (5)
8:     end if
9:     if last epoch complete then
10:      write weights back to SDRAM
11:    end if
12:  end for
13: end for
```

$\delta L/\delta \hat{y}_i$ is passed backwards to the ith neuron of the output layer and the backward pass begins (see Algorithm 4, line 5). The ith output neuron then applies the chain rule in order to compute its gradients and the error that is passed backwards to each neuron of the preceding layer (see Equation 4). $\delta L/\delta \hat{y}_i$ is multiplied with $\delta \hat{y}_i/\delta h$ (the derivative of the activation function) to get the *neuron error*. In order to update the weights, the neuron error is multiplied by the partial derivation of h to each weight of $\mathbf{w}$: $\delta h/\delta w_j = f_j^{(l-1)}$. The same is done for the bias: $\delta h/\delta b = 1$.

The error passed backwards to the jth neuron of the previous layer is given as the neuron error multiplied by the derivation of the signal received by the jth neuron: $\delta h/\delta f_j^{(l-1)} = w_j$. The jth neuron of layer $f^{(l-1)}$ sums its total error over all errors received from the neurons of the succeeding layer. This process is continued until the neurons of the first hidden layer have computed their gradients and passed their errors backwards as the pong event to the host. The host knows the backward pass is complete and the next forward pass begins.

Once a batch of backward passes is complete or the epoch has finished—in the case that the training set size is not divisible by the batch size—the weights of each neuron of the deep learning model are updated with Equation 5—the sum over the gradients computed in each backward pass multiplied by the learning rate (see Algorithm 4, lines 6–8). Lastly, the updated weights have to be extracted from the SpiNNaker machine again, back to the host. As stated above, the weights are passed onto the machine during the data generation phase of the SpiNNaker toolchain and are stored in SDRAM. Because SDRAM is slow, the weights are copied into DTCM (see Section 2.3). The fast and private DTCM memory of each SpiNNaker core can not be accessed from anywhere but the core itself. Therefore the updated weights must be written back to the parameter region in SDRAM, where the host can access them. This is a very slow operation and is only done after the last example of the whole training session has been processed (see Algorithm 4, lines 9–11). A constant latency of two seconds has been added to the host. The host sleeps for two seconds before it extracts the weights from the machine, to make sure the updated weights are all successfully copied back to SDRAM.

Last phase of the implementation was spend with implementing 1D convolutional layers. Convolutional layers were determined—during the research and planing phase—to be the hardest part of the implementation. This turned out to be true, but not for the expected reasons. We thought

that the their general approach to multiple channels and filters would lead to a complicated implementation. Rather than having troubles with mapping messages to weights and updating the kernel, the mapping of a convolutional layer onto neurons turned out to be awkward and much time was spend on getting padding and strides right, which were both estimated to be quick and easy to implement.

In the end, padding and strides turned out to be easy to implement, once the correct formula was found. Unfortunately mistakes were made during development, mostly with calculating offsets for padded neurons, when padding was set to "same" (see Section 2.1). Instead of wasting precious resources by creating more neurons in the preceding layer, which only ever send zero as signal, variables for upper and lower offsets were calculated for each neuron. For example, the left-most neuron of the layer shown in Figure 5 has a lower offset of one, whereas the right-most neuron has an upper offset of one. An offset of one means, that the neuron does not receive from *kernel size* many neurons from the previous layer, but from *kernel size* minus offset many neurons. For the left-most neuron with a lower offset of one, this means that its minimum key (see above) does not match to the first value of each kernel, but to the second. For the right-most neuron with an upper offset this simply means the counter which indicated that the forward pass has completed (see Algorithm 1, line 1) must incorporate the fact that one neuron from the previous layer is missing (same goes for lower padding).

Multichannel input and multi-filter output turned out to be straight-forward to implement. Simply sending the output of each filter in succession turned out to be no problem. On the receiving side a simple counter per neuron was created (see above). Flattening a convolutional layer turned out to be no problem as well. Flattening means that the filter-dimension of the feature map is reduced, so instead of producing a matrix (in the case of 1D convolution), the layer actually produces a vector. This is needed in order to connect a convolutional layer to a dense layer, because perceptrons cannot handle multichannel inputs. The 1D convolutional layer of the prototype were flattened by utilizing the SpiNNaker toolchain. The toolchain provides the ability to give one outgoing edge partition multiple keys. In this case each outgoing edge partition between the neurons of the convolutional layer and the dense layer in the forward direction has $filters$ many keys. Every filter sends with its own key. This gives the illusion to the dense layer, that it actually is connected to $filters \cdot n\_neurons$ many neurons in the previous layer. In truth it is only connected to $n\_neurons$ many neurons in the previous layer and each one sends $filters$ many times.

Backpropagation for convolutional layers is more complex than it is for simple dense layers, for two reasons: (i) each filter produces an error which has to be passed backwards and (ii) weights are shared between the neurons of a convolutional layer. Handling backpropagation for convolutional layers have led to the most complex code of the prototype.

Having to deal with multiple filters in the succeeding layer, which all produce an error which has to be passed backwards, makes it more difficult to match between received packets (in the backward direction) and the filter of the receiving neuron which generated the error. Again the problem was solved with counters. Unfortunately the receiving event for the backward pass in our implementation has become very complex and computationally too expensive. As stated above, the router blocks until a packet is received (the receive event callback finished). If the receive event callback takes a long time, the router will be blocked longer, which will lead to back-pressure on the communication fabric. Back-pressure will lead to dropped packets (discussed in Section 4.3).

Shared weights are implemented like the softmax activation function, with a fully-connected intra-layer partition, the kernel update partition (see above). Once the backward pass is complete, the gradients are send to the other neurons of the layer and simply summed up by each neuron.

Afterwards the gradients are the same for each neuron. Once the gradients are shared, the errors are passed backwards.

4.3 Problems

This section describes the mistakes made during development and the problems encountered. The only real problem encountered during development was time. We had the ambitious goal of implementing ResNet-50, a state-of-the-art deep learning model, on SpiNNaker. SpiNNaker is a novel neuromorphic computer architecture. Without prior programming experience on SpiNNaker and with limited domain knowledge of deep learning, the effort such an endeavor would take was underestimated. Without complications, the work plan developed during the research and planning phase, would have sufficed. In retrospect it was a naive assumption, that we would not encounter complications with the power of stalling implementation progress and this project has been another example of Hofstadter's Law (see Section 1). In the end, minor deviations from the work plan added up and time pressure ensued. Besides the minor deviations, a major problem with the prototype was discovered, which we were unable to fix. This problem has been mentioned in Section 4.2 before and concerns dropped packets.

The minor deviations encountered during the development process (listed in chronological order) were:

1. A bug in the LPG (see Section 4.1)
2. Interface changes made to the RIPTMCS during development (see Section 4.1)
3. A bug discovered during the key generation process for more than one key (as is the case for flattened convolutional layers, see Section 4.2)
4. Problems with sockets when trying to establish a second live IO session (e.g. when calling `predict` after `fit`)
5. Underestimation of the difficulties of implementing strides and "same" padding for convolutional layers

The first problem encountered was a bug in the LPG. Between the research and planning phase and the dissertation phase there was another phase of this project planned. This phase's goal was familiarization with the SpiNNaker programming model. During this phase an example program implementing Conway's game of life (Gardener, 1970; Furber and Bogdan, 2020) was changed to make it work with live IO. The original code can be found at SpiNNaker (2020c). The implementation using live IO can be found at Fassbender (2020a). The original version saves the state of each cell (machine vertex) on the SpiNNaker machine and extracts the data once the execution is finished (Furber and Bogdan, 2020; SpiNNaker, 2020c).

We changed the program such that each cell streams its state off the board. This did not work once the number of cells exceeded a certain threshold. No packets were received by the `LiveEventConnection` (see Section 4.1) and the router on the chip with the LPG showed strange warnings of dropped packets. These warnings were strange because the amount of dropped packets reported far exceeded the total amount of packets sent during the whole execution. SpiNNaker has no memory protection (no segmentation faults when trying to access memory not owned by the accessing machine vertex). Any machine vertex can override data owned by other vertices in

SDRAM and we correctly thought that the LPG was writing to memory owned by the router. After reducing the code of the LPG, we could determine strange behavior in the buffer that stores received packets for sending them to the host. Once the LPG has sent its first EIEIO message to the host (see Section 4.2) the buffer showed that it was full, even though it did not contain packets anymore. While we could isolate the problem to the generation and sending process of the EIEIO message, we could not find the bug in there and had to rely on the SpiNNaker core team to find the bug and fix it.

Finding and fixing this bug took away approximately one week (the first) of the dissertation phase and the live IO version of Conway's game of live was not yet finished. While the states were now streamed off SpiNNaker, the initial states were not yet streamed onto the machine (a vital part of the prototype which we had to familiarize ourselves with). Further interfaces and mechanisms we had to learn were how to tell SpiNNaker to run forever (see Algorithm 1, line 8) and how to stop it from the live IO callbacks. Finishing the implementation of Conway's game of life took another half week of the dissertation phase. The issue, a description and a link to the solution can be found at (Fassbender, 2020b).

The interface changes made for the next release of the toolchain were easily fixable. The SpiNNaker core team removed the direct support for RIPTMCS in a machine graph. Only a single parameter was changed in order to restore the direct support again (instead of having to use constructs only used in application graphs). The change made to the toolchain can be found at Fassbender (2020c).

The same goes for the bug discovered during the key allocation process, once an outgoing edge partition has more than one key, as is the case with flattened convolutional layers (see Section 4.2). For example, a flattened convolutional layer has five filters. Each neuron therefore has five keys. The allocation process of the toolchain was overridden by providing a constraint that tells the toolchain which key to allocate (a fixed key constraint). For the flattened layer, five fixed key constraints were passed to the toolchain, which should return a list with five keys (unsigned integers). Let the first neuron of the convolutional layer have the constraints for keys one to five. What the toolchain returned was: `[1, 5, 0, 0, 0]`. As it turned out, the keys are put into a pre-allocated numpy array (van der Walt et al., 2011) of size five. The constraints are iterated. A fixed key constraint can return multiple keys, depending on its mask (Furber and Bogdan, 2020). We only ever used `0xFFFFFFFF` as mask, which, in combination with the fixed key, will only return the fixed key. So for each iterated constraint, an offset is calculated, depending on how many keys it has generated. Unfortunately it was forgotten to sum the offsets up. The first key (one) generates an offset of one. The next key (two) is written to the numpy array at the index of the offset, so to index one. The second constraint generates an offset of one again. But instead of adding it to the previous offset, the previous offset is overwritten. The offset stays one. Therefore the third constraint (key is three) is written again to index one, overwriting the key two. This continues all the way to five. Adding two lines that add up the offset instead of overwriting it solved the problem. The solution can be found at Fassbender (2020d). Both, the interface change and the bug together, were solved in a single day.

An issue during testing was the fact that we were unable to get a second live IO session running in the same Python script, after the first one was properly closed. This made it impossible to call `predict` after `fit`, which is a necessary feature for a working deep learning library for SpiNNaker. Otherwise the user would need to export the updated weights to file and start a new script which loads the weights, before using the model for inference. The problem arises due to a socket not being properly closed, or rather not properly deallocated by the operating system of the host. In

order to fix this problem we tried to tell the operating system to reuse the socket. While this removed the error, it led to inconsistent behavior of the second live IO session. With the reused socket, the second live IO session sometimes sent packets to the machine and sometimes did not.

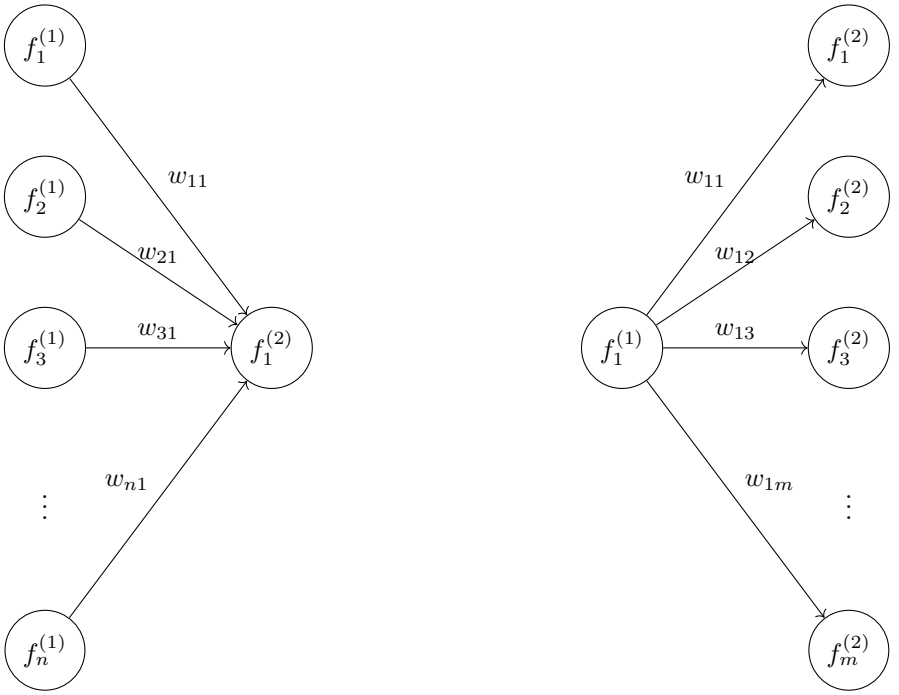
The fact that we were unable to get a second working live IO session limited our testing abilities in two ways. First, we could not `predict` with a trained model (except by exporting the updated weights to file first, as described above). Second, we could never execute the whole test suite, but had to call each function, from every test script, by hand. While this has been an inconvenience, the overall time spent trying out the workaround described above and executing the test suite by hand has not exceeded a whole working day. The issue can be found at (Fassbender, 2020e).

The four issues discussed above were all dealing with the SpiNNaker toolchain. Dealing with these issues took approximately two weeks of time during the dissertation phase. The last of the minor deviations does not concern itself with the toolchain, but has its source in mistakes made during the implementation of the prototype. As stated in Section 4.2, the time and effort it would take to get the stride and padding of a convolutional layer to work was underestimated. The problem has two sides: (i) the time it took to find the right solution and (ii) how difficult it would be to integrate strides and padding into the machine vertex. The former could have been prevented by spending more time on finding the right formula. Convolutional layers were implemented after forward and backward pass of the MLP. Time—unaccounted for in the work plan—was spent debugging cases, where padding and offset were computed wrongfully. Getting the formula for computing the offset for strided convolutional layers with "same" padding right took approximately one week. The difficulties of having to integrate padding into the machine vertices have been discussed in Section 4.2 (integrating the offset into the forward pass). The problem with strides arose during the backward pass, where we have to match the key a packet is sent with to its correct position, so the errors arrive at their proper place (where they have actually happened). In order to get the matching mechanism right, the stride of the succeeding layer has to be known. Sharing information between layers is costly concerning code complexity, something we have learned early during the development phase while implementing softmax (see Section 4.2) and backpropagation for MLPs (see below). Backpropagation with convolutional layers has been the most complex task (and the last) implemented in the prototype and it is the main cause of the major problem limiting supported size of deep learning models by the prototype: dropped packets.

While we have discussed the general backpropagation mechanism of the prototype in Section 4.2, we have not looked at the communication structure of the backward pass. The reason for that is, that there were three different communication structures implemented. All three trying to somehow avoid dropped packets or, in one case, to avoid code complexity which made another solution unmaintainable.

The problem of dropped packets has been known since the implementation of the live IO version of Conway's game of life. The execution of the game of life stopped with an error, once too many vertices were used. The problem is that, while the SpiNNaker communication fabric is great at handling lots of packets, the supported peak of packets being on the fabric at the same time is not high. Too many packets on the fabric at the same time causes back-pressure and the routers simply drop packets. Therefore packets must be distributed across time.

The SpiNNaker1 API provides a function to the machine vertex called `spin1_set_timer_tick_and_phase`, which allows the user to define a timer offset. The timer offset tells the machine vertex when to start its first update event. Without the timer offset, the update events of all vertices on the machine are called at the same time. During the update event, a cell of Conway's game of life shares its state with its neighbors. A perceptron, during the forward pass, may send its potential



(a) Incoming connections for neuron $f_1^{(2)}$.

(b) Outgoing connections for neuron $f_1^{(1)}$.

Figure 11: Conceptually, weights can be associated with edges.

during its update event, depending on whether it has received all signals from the previous layer (see Algorithm 3). If all vertices send at the same time, the peak number of packets on the communication fabric is obviously high. With the timer offset one can distribute the peak across time.

The timer offset is computed based on the placement of the neuron on the board. Each core of a SpiNNaker machine is uniquely identifiable by a three element tuple (x, y, z), where x and y are the spatial position of a SpiNNaker chip in the 2D torus topology of the machine (see Section 2.3) and $0 \leq z \leq 17$ is the id of the core on that chip. During data generation (see Section 4.1), the neuron gets the information about its placement from the toolchain. A maximum timer offset is provided by the prototype. It is the time between update events, divided by ten. 90 percent of the time between update events leaves enough time for a machine vertex with the maximum offset to send its packets along the longest path of the topology and the packet being received before the receiver's next update event. The formula for computing the maximum offset is taken from the implementation of a Poisson spike source neuron from the SpiNNaker front-end for spiking neural networks (SpiNNaker, 2020d). The maximum offset is divided by the cores per chip is multiplied by z. This way each neuron on a chip has a different offset.

During the forward pass the peak number of packets is low. Figure 11 shows how neurons are connected and how weights are conceptually bound to the edges. For example, the output of neuron $f_1^{(1)}$ is multiplied with weight w_{11}, when it is received by neuron $f_1^{(2)}$, which computes (1) during

the forward pass. Weights are owned by the destination neuron of each edge. So $f_1^{(2)}$ owns all the weights $w_{11}, w_{21}, \ldots, w_{n1}$. The output of each neuron is passed to all the neurons in the following layer—if the following layer is a dense layer—or to a subset of neurons, in the case of a convolutional layer as successor layer. Again from the example shown in Figure 11, the output of $f_1^{(1)}$ would be passed to the neurons $f_1^{(2)}, f_2^{(2)}, \ldots, f_m^{(2)}$. The forward pass is elegantly solved using a multicast message. $f_1^{(1)}$ has an outgoing edge partition "forward" with all the edges shown in Figure 11b. It sends its output with a single call to the SpiNNaker API to all the neurons it is connected to and the communication fabric takes care of multicasting the output to the neurons in the next layer. Therefore, the peak number of packets on the fabric would be exactly $n \cdot m$ (only one layer sends at a time, thanks to the ping-pong protocol presented in Section 4.2).

For a convolutional layer the peak number of packets is $n \cdot m \cdot filters/stride$. Each filter produces a different result (see Section 2.1). For each filter, one call to the SpiNNaker API is necessary in order to multicast its output to the next layer. To reduce the added pressure of having multiple filters, we introduced a constant latency into the sending loop. This keeps the peak number of packets lower by spreading the packets across time, like using the timer offset (see above).

For development we had access to one SpiNN-5 board (see Section 2.3 and Appendix A). The SpiNN-5 board has 48 chips, which means it has 864 cores. One core per chip is reserved for the router, so a maximum of 816 cores can be used for neurons. Extra monitor cores can be enabled to decrease loading time before the execution starts and for packet re-injection of dropped packets (see below), which decreases the number of cores per chip down to 15 (Furber and Bogdan, 2020). Listing 4 shows an excerpt from the test suite. Inference with two models, one MLP and one CNN, is tested. Both models have enough neurons to fill the whole board (816 cores). Neither one experiences problems with dropped packets.

Backpropagation puts much more pressure onto the communication fabric. During backpropagation, neuron $f_1^{(2)}$ in Figure 11a computes its neuron error (see Section 4.2). In order to get the error that is passed backwards, the neuron error is multiplied by each weight. The error passed from $f_1^{(2)}$ to $f_1^{(1)}$ is the neuron error of $f_1^{(2)}$ multiplied with w_{11}. The error passed to $f_2^{(1)}$ is the neuron error multiplied with w_{21} and so forth. The problem is, neuron $f_1^{(2)}$ owns all the weights $w_{11}, w_{21}, \ldots, w_{n1}$. $f_1^{(1)}$ does not know about w_{11} and cannot access it. There were three approaches to solving the problem: (i) one-to-one backward partitions, (ii) shared parameters and (iii) multicasting each error.

The first approach was to compute each error passed backwards from $f_1^{(2)}$ inside the neuron and send each error successively, via a one-to-one outgoing edge partition. This way, $f_1^{(2)}$ would have n outgoing edge partitions, each with one destination. In comparison, $f_1^{(1)}$ has one outgoing edge partition in the forward direction with n destinations. This approach has one major disadvantage: pressure on the routing table. Each outgoing edge partition has one unique key (see Section 4.1). This way, $m \cdot n$ keys are needed only for layer two shown in Figure 11, instead of m keys for all the outgoing edge partitions of the layer in the forward direction. Each router has a 1024 word routing table (Navaridas et al., 2009). Even though the SpiNNaker toolchain offers routing table compression (Heathcote, 2016), lots of small partitions will cause too many entries. The number of entries does not fit into the routing table, even for small networks. It was clear that this approach does not offer a scalable solution and was abandoned quickly. It took approximately half a week to develop and test this version of backpropagation.

Since we knew about the problem of dropped packets, we continued searching for a solution with minimal packets, rather than the most naive and easiest to implement. The next approach represents an unsuccessful trade-off between code quality and scalability. Since we did not experience dropped packets during the forward pass on a MLP filling all the available cores on a SpiNN-5 board (see Listing 4), we deemed that sending $n \cdot m$ packets works well enough (tested with a maximum n of 50 and m of 300, which makes the peak number of packets 15,000). In order to achieve one call to the SpiNNaker API per neuron for sending backwards, we introduced shared parameters. Shared parameters means that $f_1^{(1)}$ now owns and maintains a copy of $w_{11}, w_{12}, \ldots, w_{1m}$, which are owned by the neurons of layer two. If the next layer is a convolutional layer, the neurons own a copy of the whole kernel of the succeeding layer. $f_1^{(2)}$ only has to send its neuron error backwards. It reaches every neuron in layer one. The neurons in layer one compute their error by multiplying the received neuron error with w_{i1}, now that they own a copy of the weight. This way the same number of packets are sent forward and backward. Forward and backward partition between two layers are simply reversed (outgoing edge partition of $f_1^{(1)}$ in the forward direction has the m destinations in the succeeding layer, while the outgoing edge partition of $f_1^{(2)}$ in the backward direction has n destinations in the preceding layer).

As stated in Section 4.2, we early on realized the code complexity arising from sharing information between layer objects in the context of the softmax activation function. Therefore, we tried to avoid breaking the layered design pattern we chose for the prototype. Compared to softmax, sharing weights between neurons is a severe violation of this pattern. Furthermore, it violated our design philosophy: prototype before you polish (see Section 4.1).

Another negative aspect of sharing parameters is the fact that the computational effort of the backward pass and the amount of memory needed for the whole model are doubled. w_{11} now has to be updated by two different neurons, its conceptual owner $f_1^{(2)}$ and the neuron which owns a copy $f_1^{(1)}$. If w_{11} is only updated by the owner, the error of $f_1^{(1)}$ would be computed wrongly, because the neuron error of $f_1^{(2)}$ is multiplied by the outdated weight. Therefore, $f_1^{(1)}$ has to perform gradient descent not only for the weights it owns, but also for the copies of the weights of the succeeding layer. The copied weights and their gradients have to be stored by $f_1^{(1)}$ as well. The overall memory needed by the deep learning model (ignoring the constant memory needed for variables) is doubled.

Sharing parameters was deemed good enough of a compromise between scalability and code complexity during the backward pass implementation of the MLP. We were able to train a model similar to the MLP shown in Listing 4 to learn XOR (see Listing 5), without dropped packets. Therefore, we modeled our implementation of 1D convolutional layers based on shared parameters. In retrospect, this turned out to be the wrong decision and can be accounted to the miscalculation of the effort it would take to implement convolutional layers (see Section 4.2). The complexity of having to update weights of a convolutional layer not only intra-layer, but also across layers turned out to be too complex to implement in the time allocated for this thesis. Trying to solve the issue arising with padding and strides described above in addition to a backward pass algorithm for convolutional layers which turned out to be much higher in complexity than the backward pass algorithm for dense layers took all the remaining time intended for implementation and most of the time reserved as a buffer (three weeks buffer reserved for improving the implementation and for benchmarking, out of thirteen weeks of dissertation phase). In the end, we were not able to get the backward pass working for convolutional layers with shared parameters.

When it became clear that shared parameters are too complex to implement and maintain, we changed our approach for communication during the backward pass again, trying to tackle

Algorithm 5 : Backward pass for the approach of multicasting each error

```
1: compute neuron error
2: for i = 1, ..., n do
3:    error := neuron error · w_ij {j is the constant index of the neuron executing this algorithm}
4:    send error backwards to all connected neurons in the preceding layer
5:    (optionally) add latency in-between iterations, in order to spread the packets in time
6: end for
```

the problem arising from the complexity of the code. We chose to compute the error passed backwards inside $f_1^{(2)}$. This approach is like our first approach, without one-to-one partitions. The partition structure stays the same as the structure of the forward pass and the structure of the shared parameters approach for the backward pass. This approach puts massive pressure on the communication fabric. Algorithm 5 shows the backward pass procedure for communicating the errors backwards. The problem arises in Algorithm 5, line 4. For example, if $f_1^{(2)}$ from Figure 11 sends the error intended for $f_1^{(1)}$ backwards, it will reach all n neurons it is connected to from the preceding layer. The other neurons $f_2^{(1)}, f_3^{(1)}, \ldots, f_n^{(1)}$ all receive a packet and drop it, since it is not their error. In the second iteration $f_2^{(1)}$ receives the error and all other neurons drop their multicast packet. This is continued n times. The peak number of packets increases from $m \cdot n$ to $m \cdot n^2$. Of those n^2 packets, $n(n-1)$ are simply dropped at the receiving end. These packets do not serve a purpose in themselves and are only used to increase a counter, so the receiving neuron knows if the received error is the one intended for it. If we look at the MLP from Listing 4, lines 24–31, it has a 300 neuron layer, followed by a 50 neuron layer. A forward pass between these two layers needs 15,000 packets (see above). The backward pass from the 50 neuron layer to the 300 neuron layer multiplies this with a factor of 300. The peak number of packets increases from 15,000 packets to 4,500,000. 4,485,000 packets are dropped at the receiving end. Over 99 percent of all packets sent during the backward pass are unused.

With multicasting errors backwards, we were able to get backpropagation working for convolutional layer. Listing 6 shows the test we used for implementing backpropagation. We computed the gradients by hand, controlled them with the Keras model and tried to implement the backward pass accordingly. We were unable to make this test work with shared parameters. With multicasting errors, we managed to pass the test. The biggest CNN trained with this approach can be seen in Listing 7. Beyond this size, dropped packets stalled the execution. Even for such a small deep learning model, we experienced exploding gradients (up to the point where weights turned to NaN) (Brownlee, 2019b), whereas the Keras model used for comparing did not. This could have three reasons: (i) Keras has some mechanism to avoid exploding gradients the author does not know about, (ii) floating operations on SpiNNaker are numerically unstable and (iii) the most likely reason being a bug in the prototype.

Another major problem of multicasting errors backwards, besides the number of unused packets straining the communication fabric, is the amount of time spent by the receive callback. As stated in Section 4.2, while a packet is received by a core, the router on that chip blocks. Therefore, receive callbacks must be fast, to keep the time the router is blocked down. Receiving multicasted errors backwards takes time, because the receiver must find out, if the received packets is intended for it. As stated above, this is done by increasing a counter. For dense layers, receiving backwards is not too complex. Neurons of a convolutional layer on the other hand spend a lot of time in

their receive callback. They not only have to increase the counter and see, whether the packets received is one of its errors. Also, they have to deal with the extra complexity of having not only to receive one error from the sending neuron. They need to receive as many packets from a neuron in the succeeding layer, as this layer has filters (channels of the succeeding layer), multiplied by the amount of filters of the succeeding layer. Therefore, matching received packets takes time, pressuring the communication fabric even more. Ironically, in order to match received packets, the neuron needs to know about the stride of the succeeding layer, so even the less complex solution compared to shared parameters needs to violate the layered design pattern.

The SpiNNaker toolchain offers a utility called a packet re-injector (Furber and Bogdan, 2020). The re-injector is a machine vertex running on each chip. It collects dropped packets by the blocked router and stores them. Later, once the router is not blocked anymore, it re-injects the packets into the router. There is only one register in the SpiNNaker hardware for dropped packets (Furber and Bogdan, 2020). If a second packet is dropped, before the re-injector can take the first dropped packet from the register, the first packet will be lost, without possibility to recover. Even with the re-injector and increasing the time between update events from five microseconds to ten, multicasting errors backwards was not able to scale in a meaningful way.

After trying out multicasting errors backwards, without much success and seeing its scaling errors in practice, we ran out of time. We could not solve the problem of dropped packets and were unable to get the more complex approach to the backward pass—shared parameters—working. We were unable to get the prototype to a state where it would have sufficed for implementing ResNet-50 with it—the model we originally intended to benchmark our prototype with (see Section 2.2). The missing features are: (i) 2D convolutional layers, (ii) pooling layers and (iii) shortcut connections (see Section 2.2).

5. Discussion

In this section we will discuss further issues with and enhancements to the prototype. Besides the problem of not being able to implement a version of the backward pass which would enable a model on the scale of ResNet-50, we discovered some issues with our general approach to implementing deep learning on SpiNNaker and with our hypothesis, that neurons are a good choice for domain decomposition of a deep learning model (see Section 4.2). Untested solutions for the problem with dropped packets are presented, as is an idea for better domain decomposition, which could solve (or at least mend) two major issues with the prototype at once: (i) a lower peak number of packets and (ii) the previously undiscussed issue, that decomposing a deep learning model into neurons implemented as machine vertices on SpiNNaker leads to resource invariance and therefore to poorer performance. Performance has not been discussed previously, simply due to the fact that a working solution must be achieved first, before it can be made fast. But since the goal of this thesis is to communicate shortcomings of our prototype and our hypothesis (see Section 1)—so the next efforts of implementing deep learning on SpiNNaker can build on the knowledge gained during this work—we have to communicate the observations made about performance, as well as complexity and functionality.

We chose neurons as abstractions because we thought them straightforward to implement. For perceptrons, this turned out to be correct. Implementing a convolutional layer based on neurons—where each neuron does one convolution with all filters of the layer (see Section 4.2)—turned out to be quite complex. The major complexity arises during the backward pass and strides and padding being hard to integrate into the neurons (see above).

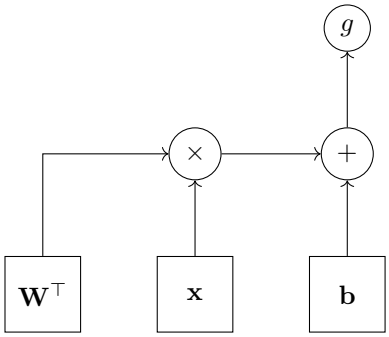
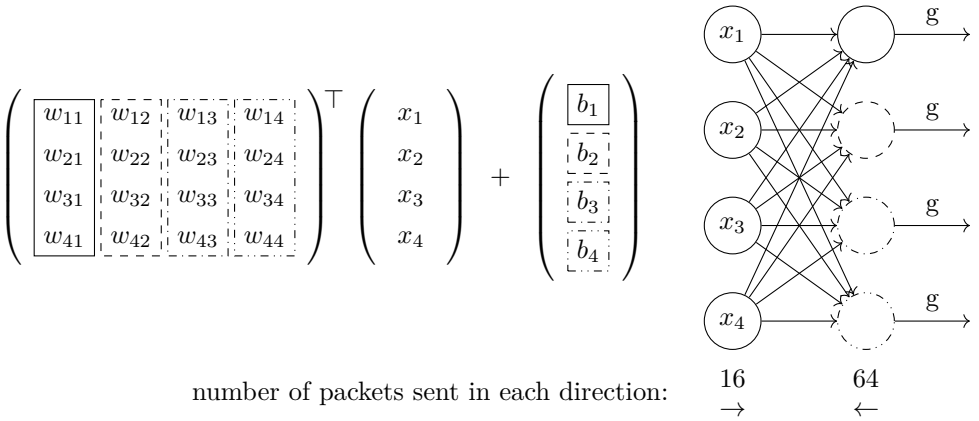


Figure 12: Schema of the computational graph for the forward pass of a single dense layer (see Equation 2). Rectangle nodes are variables (or sub-graphs, e.g. in the case where $\mathbf{x}$ is the output of a hidden layer), while circle nodes represent operators.

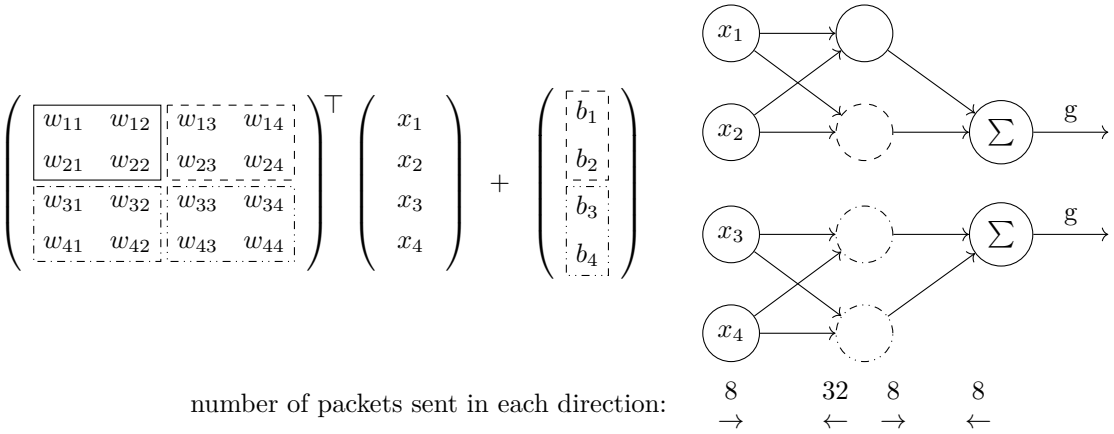
Besides not being as straightforward to implement and not as easy to reason about as was expected, our prototype has another flaw: resource invariance. This can be attributed to the early stage this project is in. The SpiNNaker toolchain offers a way to make neurons more resource aware, namely application graphs. Application graphs can combine multiple neurons into one core, fully utilizing its resources (see Section 4.1). The problem with application graphs in this context is, that they cannot handle neurons (atoms for the application graph) which will not fit into a single core. For example, the middle layer of one of the last three residual blocks of ResNet-50 (see Section 2.2) performs a 3×3 convolution on a feature map with 512 channels and has 512 filters. Storing the weights alone takes 9,216 Kb, far exceeding the available 64 Kb of DTCM of a single core (see Section 2.3). This is another problem which can be added to the missing features for ResNet-50. Deep learning neurons can grow very big and it can happen that they have to be split up. Conceptually this is solvable by utilizing the SpiNNaker toolchain, which offers a concept called shared keys, allowing multiple outgoing edge partitions to have the same key. This way a receiving neuron would not know that it is receiving from different cores.

Another example illustrating the resource invariance of neurons as machine vertices are bottlenecks. If one would execute a three layer MLP with 10, 1000 and 20 neurons per layer, the second layer would take 1000 cores, each core having a tenth of the computational effort compared to the 20 cores taken by the output layer. ResNet-50 is another example. Each time a residual block halves the spatial dimensions of its input (applies a stride of two, see Section 2.2), it doubles the number of filters (He et al., 2015). This should keep computational cost per layer identical. For our prototype, the computational effort would simply rise, each time the filters are doubled. The layers that are computationally more demanding receive less space on the board—less resources—than the computationally less demanding layers.

As stated in Section 4.2, implementing deep learning models as neurons is not the current trend in deep learning research and is not practiced by state-of-the-art deep learning libraries. These libraries represent models as a computational graph (Goodfellow et al., 2016; Abadi et al., 2015). Figure 12 shows the computational graph for the forward pass of a single dense layer (see Equation 2). A direct implementation of this compute graph on SpiNNaker would not make much sense. The matrix multiplication is by far the most complex operation, both computationally and



(a) How the computational graph of a dense layer is decomposed into neurons.



(b) How the computational graph of a dense layer could be decomposed in order to reduce the peak and overall number of packets by sparsifying the graph.

Figure 13: Illustration of how a dense layer could be decomposed in order to reduce the peak number of packets. The number of packets sent are based on multicasting errors backwards (see Section 4.3).

memory wise. $\mathbf{W}$ is m times bigger than $\mathbf{x}$ and n times bigger than $\mathbf{b}$. Furthermore, we have already discussed that neurons can grow too big. Abstracting over the computational graph of a layer instead, will only intensify this problem.

But the computational graph gives us a way to decompose a layer, other than neurons. Figure 13 shows how a better domain decomposition reduces the overall and the peak number of packets sent, both for the forward pass and the backward pass. Neurons—for the forward pass—can be seen as nothing other than the row-wise decomposition of the matrix multiplication between $\mathbf{x}$ and $\mathbf{W}^\top$, in combination with adding a single bias and putting the resulting scalar through the activation function (see Figure 13a). During the backward pass, neurons also perform the derivation of their parts (they integrate the derivation of the compute graph shown in Figure 12). The problem with this domain decomposition is, that each column of $\mathbf{W}$ (each row of $\mathbf{W}^\top$—Figure 13 displays the untransposed $\mathbf{W}$) depends on all the values of the vector $\mathbf{x}$ with which the multiplication is conducted. The result is—like the name suggests—a dense (fully-connected) graph (see Figure 13a). As we have shown in Section 4.3, the forward pass between two dense layers did not result in dropped packets, for up to 15,000 packets sent on a SpiNN-5 board (see Section 2.3). Problems arise during the backward pass with multicasted errors, where the packets sent are n times higher than during the forward pass.

The number of packets sent by each layer during either the forward or backward pass is a cuboid. One dimension is the number of neurons in the layer, the second is the number of connections (in that particular direction) the neurons have and the third dimension is the number of sends per pass. For example, the four neurons with the different dash patterns seen in Figure 13a all have four connections in the backwards direction and all send four times (multicasting errors backwards, see Section 4.3), the errors for each neuron in the previous layer. Therefore, the number of packets sent backwards by that layer is $4 \cdot 4 \cdot 4 = 64$.

Another way to decompose a layer is shown in Figure 13b. Here, each vertex owns not a column of $\mathbf{W}$, but a contiguous slice Both dimensions of the slice are the same, or as close to it than possible. The one dimensional domain decomposition of $\mathbf{W}^\top\mathbf{x}$ that we have called neurons so far is made two dimensional. This decomposition has the great property of sparsifying connections between the layers (see Figure 13b). Instead of full connectivity, each vertex is only connected to half of the neurons of the previous layer. This way, one dimension of the cuboid representing the sent packets is also halved. This decreases the peak packets sent backwards from 64 down to $4 \cdot 2 \cdot 4 = 32$ (see Figure 13b). The downside of the two dimensional decomposition is the fact that the forward pass is now more complicated. A vertex now does not produce an output which is ready for the next layer, but the outputs must be accumulated row-wise, in order to get the correct results of the matrix multiplication. This is why another layer is added, which (i) performs the accumulation, (ii) applies the activation function and (iii) passes the results forward to the next layer. In Figure 13b, we simply led the two of the four vertices performing the matrix multiplication own and add the biases to their outputs. Adding the bias could easily be done by the accumulator vertices (which would probably result in better readable and maintainable code). Also possible, to further reduce resources taken by a layer, would be to fuse the accumulator vertices with a subset of vertices from those performing the distributed matrix multiplication.

Concerning convolutional layers, decomposing them should be comparable to dense layers, the only difference being the obvious fact that matrix multiplication is replaced with a convolution. Feature maps are nothing but a higher dimensional output, compared to the vectors produced by dense layers and it should be possible to decompose a convolutional in the same manner. This would also be a first step towards solving the problem with bottlenecks for ResNet-50 (see above),

where the amount of filters is increased, but the amount of spatial dimensions is reduced. However, how the time spent in the receive callbacks can be reduced to a bare minimum remains an open question and is not directly solved by a better domain decomposition. A more memory intensive solution could be to buffer the received packets and sort and match them during the update event.

Using an application graph to fuse neurons into less machine vertices would be a big step to solving the problem of having a peak number of packets too high for SpiNNaker's communication fabric. Like the two dimensional domain decomposition, less machine vertices can reduce two dimensions of the cuboid of generated messages: (i) the number of machine vertices and (ii) the number of connections (if the previous layer also has less machine vertices than neurons). The main advantage of using the two dimensional domain decomposition over an application graph is the fact that it is more atomic. Neurons are always performing a whole row of $\mathbf{W}^\top \mathbf{x}$. If this row gets too big to fit into a single core, an application graph would not help. With the two dimensional domain decomposition, this would not happen, while it still offers the same ability to utilize the resources of SpiNNaker, simply by giving the vertices as much of the matrix as will fit. For this, the process of decomposing must be aware of the available resources (e.g. how many cores are at exposure). If one thinks this even further, the domain decomposition could even span across layers, for example by fusing sub-graphs of the overall computational graph of the deep learning model. This could also reduce the number of idle cores or even remove a whole pass from the communication fabric (e.g. when fusing two small, consecutive layers into one vertex no communication would need to go over the network but could simply be done in the core). Programs that optimize deep learning models like that—not for SpiNNaker—exist already. These programs are known as deep learning compilers (Li et al., 2020). There are already existing many different compilers, such as: Glow (Rotem et al., 2018), nGraph (Cyphers et al., 2018), TVM (Chen et al., 2018), Tensor Comprehensions (Vasilache et al., 2018) or XLA (Leary and Wang, 2017). All trying to make deep learning faster and more portable.

- with this streaming model I have no easy way to do batch normalization—implications for vanishing/exploding gradients during training
- Ping-pong protocol sucks performance wise
- optimizations from README
- gradients not shared after each pass but only before weight update

6. Conclusion

7. Next Steps

- profiling
- cost model
- multiple copies of the same network on the same machine → use all resources available
- better domain decomposition (SpiNNaker application graph or custom solution (application graph not helpful for neurons which become too big))

- smart algorithms vs. integrating with state-of-the-art libraries (investing time in stuff like SLIDE and the one paper by the Austrian guys about sparse connections explicitly mentioning SpiNNaker and neuromorphic chips or rather work on a trans-/compiler that efficiently translates linear algebra operations (like TF, PyTorch,...) onto SpiNNaker)
- integrate into compiler projects like Apache-TVM, XLA, Glow, nGraph, etc.
- implementing ONNX spec to make it easy for developers to use SpiNNaker (develop in PyTorch → run on SpiNNaker)

References

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, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. AI and Compute. `https://openai.com/blog/ai-and-compute/`, 2019.

ARM. GNU ARM Embedded Toolchain, 2020. URL `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm`.

Y. Bengio and Yann LeCun. Scaling learning algorithms towards ai. 01 2007.

Y. Bengio, Pascal Lamblin, D. Popovici, Hugo Larochelle, and U. Montreal. Greedy layer-wise training of deep networks. volume 19, 01 2007.

Justin Boitano. How NVIDIA EGX Is Forming Central Nervous System of Global Industries, 05 2020. URL `https://blogs.nvidia.com/blog/2020/05/15/egx-security-resiliency/`.

A. D. Brown, S. B. Furber, J. S. Reeve, J. D. Garside, K. J. Dugan, L. A. Plana, and S. Temple. Spinnaker—programming model. *IEEE Transactions on Computers*, 64(6):1769–1782, 2015.

Jason Brownlee. Crash Course in Convolutional Neural Networks for Machine Learning, 08 2019a. URL `https://machinelearningmastery.com/crash-course-convolutional-neural-networks/`.

Jason Brownlee. A Gentle Introduction to Exploding Gradients in Neural Networks, 07 2019b. URL `https://machinelearningmastery.com/exploding-gradients-in-neural-networks/`.

Andrew Cassidy, Paul Merolla, John Arthur, S.K. Esser, Bryan Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore Wong, Vitaly Feldman, Arnon Amir, Daniel Ben Dayan Rubin, Filipp Akopyan, Emmett McQuinn, W.P. Risk, and Dharmendra Modha. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. 08 2013. doi: 10.1109/IJCNN.2013.6707077.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018.

François Chollet et al. Keras. `https://keras.io`, 2015.

T Ciodaro, D Deva, Joao Seixas, and Denis Oliveira Damazio. Online particle detection with neural networks based on topological calorimetry information. *Journal of Physics: Conference Series*, 368, 06 2012. doi: 10.1088/1742-6596/368/1/012030.

Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016. URL `http://arxiv.org/abs/1602.02830`.

Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning, 2018.

M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

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. *Frontiers in Neuroinformatics*, 2:11, 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.011.2008. URL `https://www.frontiersin.org/article/10.3389/neuro.11.011.2008`.

Yunbin Deng. Deep learning on mobile devices - A review. *CoRR*, abs/1904.09274, 2019. URL `http://arxiv.org/abs/1904.09274`.

Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. Inside volta: The world's most advanced data center gpu, 05 2017. URL `https://developer.nvidia.com/blog/inside-volta/`.

ECMA. ECMA-262, 06 2020. URL `https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`.

Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, 2016. ISSN 0027-8424. doi: 10.1073/pnas.1604850113. URL `https://www.pnas.org/content/early/2016/09/19/1604850113`.

Jonas Fassbender. Conway's Game of Life with Live IO, 2020a. URL `https://github.com/jofas/master_thesis/tree/learn_programming_spinnaker/learn_spinn`.

Jonas Fassbender. Encountered an issue within the live_packet_gather.c file (flush_events seems to corrupt the circular_buffer 'with_payload_buffer'), 2020b. URL https://github.com/SpiNNakerManchester/SpiNNFrontEndCommon/issues/593.

Jonas Fassbender. 'ReverseIPTagMulticastSourceMachineVertex' was missing its 'n_keys' argument which I need, because I work with a machine graph and not an application graph, 2020c. URL https://github.com/SpiNNakerManchester/SpiNNFrontEndCommon/pull/629.

Jonas Fassbender. fixed wrong offset into key_array, 2020d. URL https://github.com/SpiNNakerManchester/PACMAN/pull/298.

Jonas Fassbender. "OSError: [Errno 98] Address already in use" when trying to run a second LiveEventConnection in the same program (after the first was closed properly), 2020e. URL https://github.com/SpiNNakerManchester/SpiNNFrontEndCommon/issues/646.

Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0201575949.

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 04 1980. doi: 10.1007/bf00344251. URL https://doi.org/10.1007%2Fbf00344251.

Steve Furber. Large-scale neuromorphic computing systems. *Journal of Neural Engineering*, 13, 08 2016. doi: 10.1088/1741-2560/13/5/051001.

Steve Furber and Petruț Bogdan. *SpiNNaker: A Spiking Neural Network Architecture*. 03 2020. ISBN 978-1-68083-653-0. doi: 10.1561/9781680836523.

Steve Furber and Steve Temple. Neural systems engineering. *Journal of the Royal Society, Interface / the Royal Society*, 4:193–206, 05 2007. doi: 10.1098/rsif.2006.0177.

Steve Furber, Steve Temple, and Andrew Brown. High-performance computing for systems of spiking neurons. 2, 01 2006.

Steve Furber, Francesco Galluppi, Steve Temple, and Luis Plana. The spinnaker project. *Proceedings of the IEEE*, 102:652–665, 05 2014. doi: 10.1109/JPROC.2014.2304638.

M. Gardener. Mathematical games: the fantastic combinations of john conway's new solitaire game "life. 1970.

Lee Gomes. Neuromorphic Chips Are Destined for Deep Learning—or Obscurity, 05 2017. URL https://spectrum.ieee.org/semiconductors/design/neuromorphic-chips-are-destined-for-deep-learningor-obscurity.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer New York, New York, NY, second edition edition, 2009. ISBN 9780387848570.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Jonathan David Heathcote. Building and operating large-scale spinnaker machines. 2016.

G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18:1527–54, 08 2006. doi: 10.1162/neco.2006.18.7.1527.

D.R. Hofstadter. *Godel, Escher, Bach: an eternal golden braid*. Harvester Press Limited, 1979. URL https://books.google.de/books?id=uvoRvgAACAAJ.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL http://www.sciencedirect.com/science/article/pii/0893608089900208.

D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, 160(1):106–154, 1962. doi: 10.1113/jphysiol.1962.sp006837. URL https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1962.sp006837.

D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243, 1968. doi: 10.1113/jphysiol.1968.sp008455. URL https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1968.sp008455.

David H. Hubel and Torsten N. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148:574–91, 1959.

Giacomo Indiveri, Bernabe Linares-Barranco, Tara Hamilton, André van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii Liu, Piotr Dudek, Philipp Häfliger, Sylvie Renaud, Johannes Schemmel, Gert Cauwenberghs, John Arthur, Kai Hynna, Fopefolu Folowosele, Sylvain SAÏGHI, Teresa Serrano-Gotarredona, Jayawan Wijekoon, Yingxue Wang, and Kwabena Boahen. Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, 5:73, 2011. ISSN 1662-453X. doi: 10.3389/fnins.2011.00073. URL https://www.frontiersin.org/article/10.3389/fnins.2011.00073.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski,

Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017. URL `http://arxiv.org/abs/1704.04760`.

Patrick Kennedy. Case study on the google tpu and gddr5 from hot chips 29, 08 2017. URL `https://www.servethehome.com/case-study-google-tpu-gddr5-hot-chips-29/`.

Donald E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6:261–301, 1974.

Ronny Krashinsky and Olivier Giroux. Inside the nvidia ampere architecture, 2020. URL `https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21730-inside-the-nvidia-ampere-architecture.pdf`.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

Jennifer Langston. Microsoft announces new supercomputer, lays out vision for future AI work, 05 2020. URL `https://blogs.microsoft.com/ai/openai-azure-supercomputer/`.

Chris Leary and Todd Wang. XLA: TensorFlow, compiled!, 02 2017. URL `https://www.youtube.com/watch?v=kAOanJczHA0`.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015. doi: 10.1038/nature14539.

Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. The MNIST database of handwritten digits, 07 2020. URL `http://yann.lecun.com/exdb/mnist/`.

Victor Lee. Parallel Computing: Opportunities and Challenges, 03 2011. URL `http://web.stanford.edu/class/ee380/Abstracts/110330-slides.pdf`.

Michael Leung, Hui Xiong, Leo Lee, and Brendan Frey. Deep learning of the tissue-regulated splicing code. *Bioinformatics (Oxford, England)*, 30:i121–i129, 06 2014. doi: 10.1093/bioinformatics/btu277.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey, 2020.

John Loeffler. No More Transistors: The End of Moore's Law, 11 2018. URL https://interestingengineering.com/no-more-transistors-the-end-of-moores-law.

Junshui Ma, Robert Sheridan, Andy Liaw, George Dahl, and Vladimir Svetnik. Deep neural nets as a method for quantitative structure–activity relationships. *Journal of chemical information and modeling*, 55, 01 2015. doi: 10.1021/ci500747n.

Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(97)00011-7. URL http://www.sciencedirect.com/science/article/pii/S0893608097000117.

Adam H. Marblestone, Greg Wayne, and Konrad P. Kording. Towards an integration of deep learning and neuroscience. *bioRxiv*, 2016. doi: 10.1101/058545. URL https://www.biorxiv.org/content/early/2016/06/13/058545.

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. *CoRR*, abs/1803.04014, 2018. URL http://arxiv.org/abs/1803.04014.

Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. Mlperf training benchmark, 2019.

Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, pages 115–133, 12 1943. doi: 10.1007/BF02478259. URL http://link.springer.com/10.1007/BF02478259.

Carver Mead. Analog vlsi and neural systems. 1989.

Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969.

MLPerf. Mlperf training v0.6 results, 07 2019. URL https://mlperf.org/training-results-0-6/.

Peter Morlion. Software Architecture: The 5 Patterns You Need to Know, 06 2018. URL https://dzone.com/articles/software-architecture-the-5-patterns-you-need-to-k.

Margi Murphy. Google says its AI can spot lung cancer a year before doctors, 05 2019. URL https://www.telegraph.co.uk/technology/2019/05/07/google-says-ai-can-spot-lung-cancer-year-doctors/.

Javier Navaridas, Mikel Luján, Jose Miguel-Alonso, Luis Plana, and Steve Furber. Understanding the interconnection network of spinnaker. pages 286–295, 01 2009. doi: 10.1145/1542275.1542317.

OpenAI. OpenAI Five Defeats Dota 2 World Champions, 04 2019. URL https://openai.com/blog/openai-five-defeats-dota-2-world-champions/.

Eustace Painkras, Luis Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David Lester, Andrew Brown, and Steve Furber. Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *Solid-State Circuits, IEEE Journal of*, 48:1943–1953, 08 2013. doi: 10.1109/JSSC.2013.2259038.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019. URL `http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Andres Felipe Rodriguez Perez. Intel Processors for Deep Learning Training, 11 2017. URL `https://software.intel.com/content/www/us/en/develop/articles/intel-processors-for-deep-learning-training.html`.

Hans Plesser, Jochen Eppler, Abigail Morrison, Markus Diesmann, and Marc-Oliver Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. volume 4641, pages 672–681, 08 2007. doi: 10.1007/978-3-540-74466-5_71.

Marc'Aurelio Ranzato, Christopher Poultney, Sumit Chopra, and Yann Lecun. Efficient learning of sparse representations with an energy-based model. 01 2006.

Alexander Rast, Alan Stokes, Sergio Davies, Samantha Adams, Himanshu Akolkar, David Lester, Chiara Bartolozzi, Angelo Cangelosi, and Steve Furber. Transport-independent protocols for universal aer communications. 11 2015. doi: 10.1007/978-3-319-26561-2_79.

Eric Steven Raymond. The art of unix programming. Addison-Wesley, 2003.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph lowering compiler techniques for neural networks, 2018.

Andrew Rowley, Christian Brenninkmeijer, Simon Davidson, Donal Fellows, Andrew Gait, David Lester, Luis Plana, Oliver Rhodes, Alan Stokes, and Steve Furber. Spinntools: The execution engine for the spinnaker platform. *Frontiers in Neuroscience*, 13, 03 2019. doi: 10.3389/fnins.2019.00231.

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. *Frontiers in Neuroscience*, 11:682, 2017. ISSN 1662-453X. doi: 10.3389/fnins.2017.00682. URL `https://www.frontiersin.org/article/10.3389/fnins.2017.00682`.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986a. ISBN 026268053X.

David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, Cambridge, MA, USA, 1986b. ISBN 026268053X.

David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 2: Psychological and Biological Models*. MIT Press, Cambridge, MA, USA, 1986c. ISBN 0262132184.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Andrew Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577:1–5, 01 2020. doi: 10.1038/s41586-019-1923-7.

David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 10 2017. doi: 10.1038/nature24270.

Tom Simonite. Moore's Law Is Dead. Now What?, 05 2016. URL `https://www.technologyreview.com/2016/05/13/245938/moores-law-is-dead-now-what/`.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 09 2014.

SNN toolbox. Spiking neural network conversion toolbox – introduction, 07 2020. URL `https://snntoolbox.readthedocs.io/en/latest/guide/intro.html`.

SpiNNaker. SpiNNaker Project, 2020a. URL `http://apt.cs.manchester.ac.uk/projects/SpiNNaker/project/`.

SpiNNaker. SpiNNaker Chip, 2020b. URL `http://apt.cs.manchester.ac.uk/projects/SpiNNaker/SpiNNchip/`.

SpiNNaker. Conway's Game of Life, 2020c. URL `https://github.com/SpiNNakerManchester/SpiNNakerGraphFrontEnd/tree/master/spinnaker_graph_front_end/examples/Conways/partitioned_example_b_no_vis_buffer`.

SpiNNaker. Spike Source Poisson Vertex, 2020d. URL https://github.com/SpiNNakerManchester/sPyNNaker/blob/8a8cd3132f9d8e1b22354bb1da01f8840aaba29b/spynnaker/pyNN/models/spike_source/spike_source_poisson_vertex.py#L639.

Marcel Stimberg, Romain Brette, and Dan FM Goodman. Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314, aug 2019. ISSN 2050-084X. doi: 10.7554/eLife.47314. URL https://doi.org/10.7554/eLife.47314.

Zak Stone. Cloud TPU Pods break AI training records, 2019. URL https://cloud.google.com/blog/products/ai-machine-learning/cloud-tpu-pods-break-ai-training-records.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567.

A.S. Tanenbaum and D.J. Wetherall. *Computer Networks*. Pearson custom library. Pearson, 2013. ISBN 9781292024226. URL https://books.google.de/books?id=w_d5ngEACAAJ.

UoMCompsci. SpiNNaker 1 Million Celebration, 10 2019. URL https://www.youtube.com/watch?v=wcJWLH026P8.

S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.

Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

Appendices

A. Images of the SpiNNaker Hardware

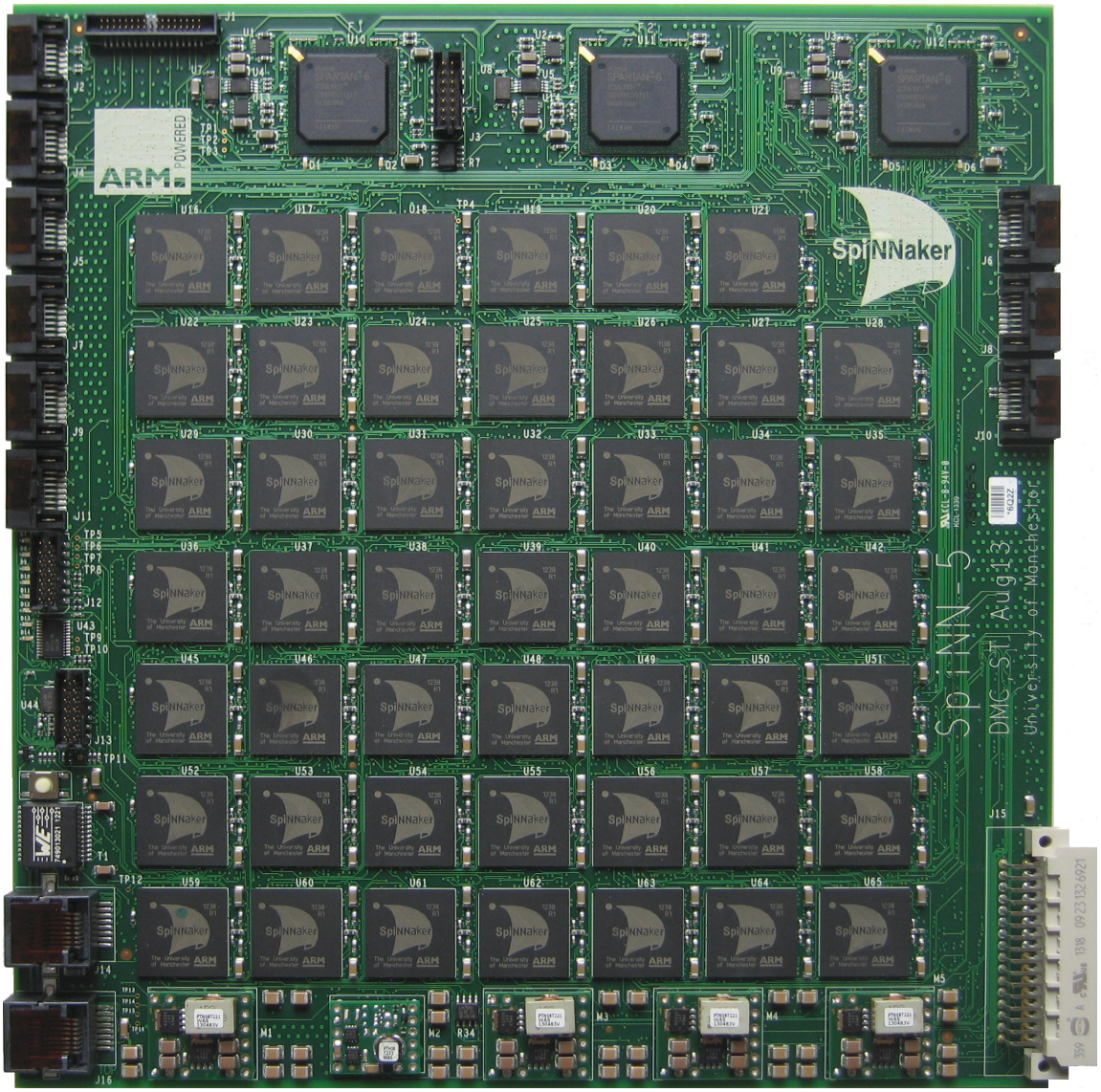


Figure 14: A single SpiNNaker (SpiNN-5) board. Image reproduced with permission from Alan Stokes.

Figure 15: The SpiNNaker1M machine in Manchester. Image reproduced with permission from Alan Stokes.

B. Excerpts from the Test Suite

```
import numpy as np
import tensorflow as tf

from spiDNN import Model
from spiDNN.layers import Input, Dense, Conv1D


N = 224
n_channels = 3
kernel_size = 3


def test_inference():
    X = np.random.rand(500, N)

    kmodel = tf.keras.models.Sequential()
    kmodel.add(tf.keras.layers.Dense(50, activation="relu", input_shape=(N,)))
    kmodel.add(tf.keras.layers.Dense(50, activation="softmax"))
    kmodel.add(tf.keras.layers.Dense(300, activation="tanh"))
    kmodel.add(tf.keras.layers.Dense(50, activation="sigmoid"))
    kmodel.add(tf.keras.layers.Dense(25))
    kmodel.add(tf.keras.layers.Dense(17, activation="softmax"))

    model = Model()
    model.add(Input(N))
    model.add(Dense(50, activation="relu"))
    model.add(Dense(50, activation="softmax"))
    model.add(Dense(300, activation="tanh"))
    model.add(Dense(50, activation="sigmoid"))
    model.add(Dense(25))
    model.add(Dense(17, activation="softmax"))

    model.set_weights(kmodel.get_weights())

    p = model.predict(X)
    p_ = kmodel.predict(X)

    error = np.absolute(p - p_)
    assert np.amax(error) < 1e-4


def test_inference_conv1d_1():
    input_shape = (N, n_channels)
    X = np.random.rand(500, *input_shape)
```

```
45
46      kmodel = tf.keras.models.Sequential()
47      kmodel.add(tf.keras.layers.Conv1D(
48          1, kernel_size, padding="same", input_shape=input_shape))
49      kmodel.add(tf.keras.layers.Conv1D(5, kernel_size * 4, padding="same"))
50      kmodel.add(tf.keras.layers.Conv1D(
51          16, kernel_size, padding="same", strides=2))
52      kmodel.add(tf.keras.layers.Flatten())
53      kmodel.add(tf.keras.layers.Dense(16))
54
55      model = Model()
56      model.add(Input(*input_shape))
57      model.add(Conv1D(1, (kernel_size,), padding="same"))
58      model.add(Conv1D(5, (kernel_size * 4,), padding="same"))
59      model.add(Conv1D(16, (kernel_size,), padding="same", stride=2))
60      model.add(Dense(16))
61
62      model.set_weights(kmodel.get_weights())
63
64      p = model.predict(X)
65      p_ = kmodel.predict(X)
66
67      error = np.absolute(p - p_)
68      assert np.amax(error) < 1e-4
```

Listing 4: Excerpt from the test suite showing two models (one MLP and one CNN), which take up all the capacity of a SpiNN-5 board.

```
1  import numpy as np
2  import tensorflow as tf
3
4  from spiDNN import Model
5  from spiDNN.layers import Input, Dense
6
7  from copy import deepcopy
8  from time import time
9
10 EPOCHS = 50
11 BATCH_SIZE = 4
12 LEARNING_RATE = 0.1
13
14 def test_mlp_backprop():
15     X = np.array([[.0, .0], [.0, 1.], [1., .0], [1., 1.]])
16     y = np.array([[.0, 1.], [1., .0], [1., .0], [.0, 1.]])
17
18     kmodel = tf.keras.models.Sequential()
19     kmodel.add(tf.keras.layers.Dense(
20         50, activation="relu", input_shape=(2,)))
21     kmodel.add(tf.keras.layers.Dense(50, activation="softmax"))
22     kmodel.add(tf.keras.layers.Dense(300, activation="tanh"))
23     kmodel.add(tf.keras.layers.Dense(50, activation="sigmoid"))
24     kmodel.add(tf.keras.layers.Dense(25))
25     kmodel.add(tf.keras.layers.Dense(2, activation="softmax"))
26
27     kmodel.compile(
28         loss="mean_squared_error",
29         optimizer=tf.keras.optimizers.SGD(learning_rate=LEARNING_RATE))
30
31     model=Model()
32     model.add(Input(2))
33     model.add(Dense(50, activation="relu"))
34     model.add(Dense(50, activation="softmax"))
35     model.add(Dense(300, activation="tanh"))
36     model.add(Dense(50, activation="sigmoid"))
37     model.add(Dense(25))
38     model.add(Dense(2, activation="softmax"))
39
40     model.set_weights(kmodel.get_weights())
41
42     unfitted_weights = deepcopy(model.get_weights())
43
44     spinn_start = time()
45     model.fit(
46         X, y, "mean_squared_error", epochs=EPOCHS, batch_size=BATCH_SIZE,
```

```
47              learning_rate=LEARNING_RATE)
48      spinn_end = time()
49
50      keras_start = time()
51      kmodel.fit(X, y, epochs=EPOCHS, batch_size=BATCH_SIZE, shuffle=False)
52      keras_end = time()
53
54      w = model.get_weights()
55      w_ = kmodel.get_weights()
56
57      error = [x - x_ for x, x_ in zip(w, w_)]
58      update = [x - x_ for x, x_ in zip(w, unfitted_weights)]
59
60      # make sure weights are updated in SDRAM before they are extracted
61      for u in update:
62          assert np.amax(np.absolute(u)) > 0.0
63
64      for e in error:
65          e_max = np.amax(np.absolute(e))
66          print(e_max)
67          assert e_max < 0.1
68
69      print("SpiNNaker took: {} seconds".format(spinn_end - spinn_start))
70      print("Keras took: {} seconds".format(keras_end - keras_start))
```

Listing 5: Excerpt from the test suite showing a network similar to the MLP from Listing 4 being trained. The MLP is trained to learn XOR, so the input and output dimensions are different from the model shown in Listing 4. The backward pass was implemented using shared parameters (see Section 4.3).

```
import numpy as np
import tensorflow as tf

from spiDNN import Model
from spiDNN.layers import Input, Dense


def test_training_conv1d_with_known_weights():
    input_shape = (2, 1)

    X = np.array([[0., 1.]]).reshape(1,2,1)
    y = np.array([[1.]])

    weights = [
        np.array([[[.1, .4]],
                  [[.2, .5]],
                  [[.3, .6]]]), np.array([.0, .0]),
        np.array([[[ .7, 1.3],
                   [ .8, 1.4]],
                  [[ .9, 1.5],
                   [1. , 1.6]],
                  [[1.1, 1.7],
                   [1.2, 1.8]]]), np.array([.0, .0]),
        np.array([[1.9],[2.0],[2.1],[2.2]]), np.array([.0])
    ]


    c1 = tf.keras.layers.Conv1D(2, 3, padding="same", input_shape=input_shape)
    c2 = tf.keras.layers.Conv1D(2, 3, padding="same")

    kmodel = tf.keras.models.Sequential()
    kmodel.add(c1)
    kmodel.add(c2)
    kmodel.add(tf.keras.layers.Flatten())
    kmodel.add(tf.keras.layers.Dense(1))

    kmodel.compile(
        loss="mean_squared_error",
        optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2))

    kmodel.set_weights(weights)

    kmodel.train_on_batch(X, y)

    model = Model()
    model.add(Input(*input_shape))
```

```
47      model.add(Conv1D(2, (3,), padding="same"))
48      model.add(Conv1D(2, (3,), padding="same"))
49      model.add(Dense(1))
50
51      model.set_weights(deepcopy(weights))
52
53      model.fit(X, y, "mean_squared_error", epochs=1, batch_size=1,
54                learning_rate=1e-2)
55
56      w = model.get_weights()
57      w_ = kmodel.get_weights()
58
59      error = [x - x_ for x, x_ in zip(w, w_)]
60      update = [x - x_ for x, x_ in zip(w, weights)]
61
62      for u in update:
63          assert np.amax(np.absolute(u)) > 0.0
64
65      for e in error:
66          e_max = np.amax(np.absolute(e))
67          print(e_max)
68          assert e_max < 0.1
```

Listing 6: Excerpt from the test suite showing the training of a CNN with known weights. It was used for implementing backpropagation for convolutional layer.

```
import numpy as np
import tensorflow as tf

from spiDNN import Model
from spiDNN.layers import Input, Dense


def test_training_conv1d():
    loss = "mean_squared_error"
    kernel_size = 3
    input_shape = (10, 3)

    X = np.random.rand(4, *input_shape)
    y = np.random.rand(4, 4)

    kmodel = tf.keras.models.Sequential()
    kmodel.add(tf.keras.layers.Conv1D(
        1, 3, padding="same", input_shape=input_shape))
    kmodel.add(tf.keras.layers.Conv1D(2, kernel_size - 1, padding="same"))
    kmodel.add(tf.keras.layers.Conv1D(2, kernel_size + 2, padding="same"))
    kmodel.add(tf.keras.layers.Flatten())
    kmodel.add(tf.keras.layers.Dense(y.shape[1]))

    kmodel.compile(
        loss=loss,
        optimizer=tf.keras.optimizers.SGD(learning_rate=0.1))

    model = Model()
    model.add(Input(*input_shape))
    model.add(Conv1D(1, (3,), padding="same"))
    model.add(Conv1D(2, (kernel_size - 1,), padding="same"))
    model.add(Conv1D(2, (kernel_size + 2,), padding="same"))
    model.add(Dense(y.shape[1]))

    model.set_weights(kmodel.get_weights())

    unfitted_weights = deepcopy(model.get_weights())

    model.fit(
        X, y, loss, epochs=1, batch_size=4,
        learning_rate=0.1)

    kmodel.fit(X, y, epochs=1, batch_size=4, shuffle=False)

    w = model.get_weights()
    w_ = kmodel.get_weights()
```

```
47
48      error = [x - x_ for x, x_ in zip(w, w_)]
49      update = [x - x_ for x, x_ in zip(w, unfitted_weights)]
50
51      for u in update:
52          assert np.amax(np.absolute(u)) > 0.0
53
54      for e in error:
55          e_max = np.amax(np.absolute(e))
56          print(e_max)
57          assert e_max < 1.
```

Listing 7: Excerpt from the test suite showing the biggest CNN trained with the prototye, without dropped packets.