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 investigates the performance of SpiNNaker machines for deep learning 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.

- here a paragraph about the results

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 prototype developed for benchmarking and Section 5 presents the benchmarks and its results. In Section 6 the results of the benchmark are discussed, as well as the development process. Section 7 contains the conclusion, while Section 8 outlines the next steps for further increasing the performance of SpiNNaker by enhancing the prototype.

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 the context of the conducted benchmark presented in Section 5. 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).

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

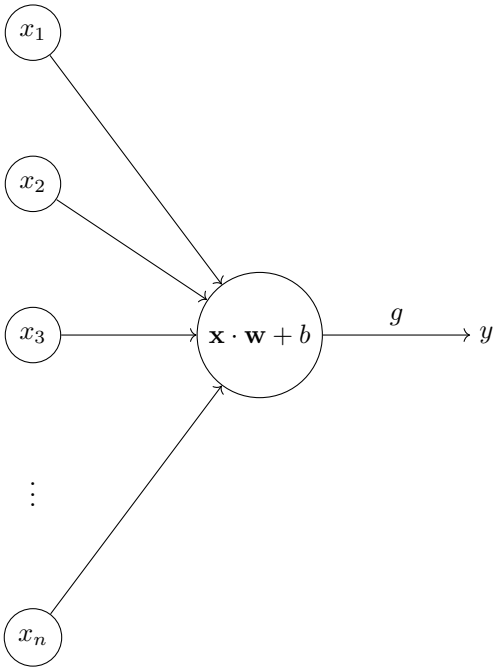

Figure 1: Schema of a perceptron.

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 con-

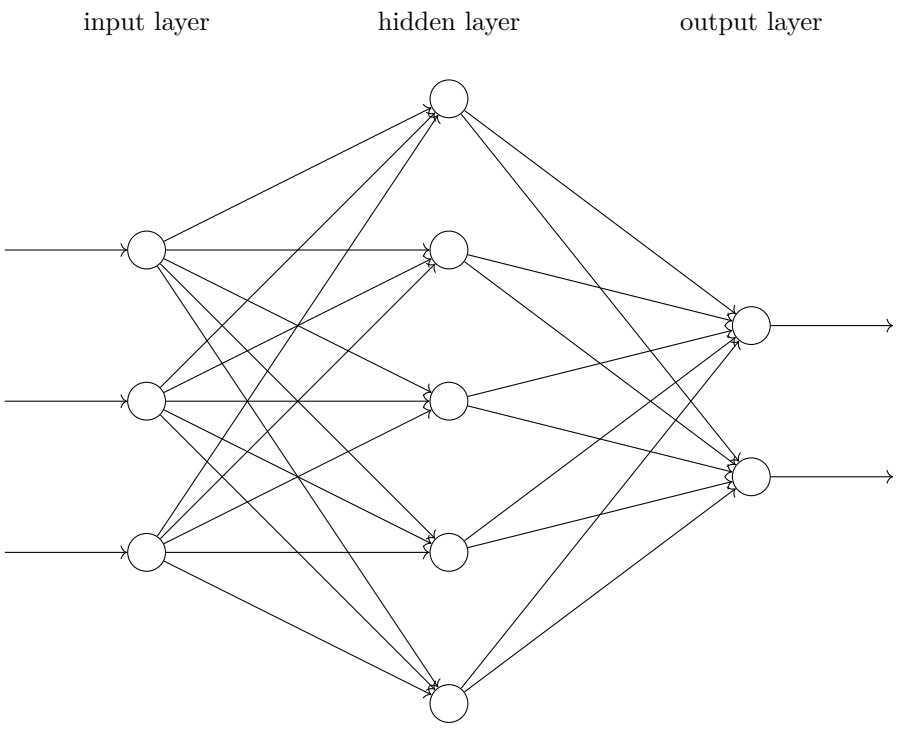

Figure 2: Schema of a MLP or feedforward neural network.

nected 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 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 $f^{(1)}, f^{(2)}, \ldots, f^{(l)}$, 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).

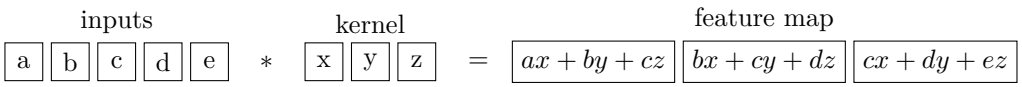

Figure 3: Example of a 1D cross-correlation operation with a kernel size of three, a single channel, a single filter, a stride of one and valid padding.

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 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

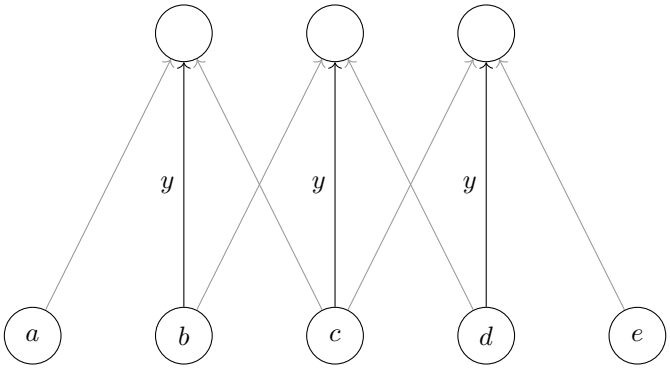

Figure 4: Schema for the convolutional layer performing the convolution shown in Figure 3. Each neuron represents one convolution. The schema shows the property of shared weights and sparse connectivity (Goodfellow et al., 2016). The black edges all have the same associated weight y, while one can see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

of a predefined number of filters, each with its own kernel (and optionally a bias) (Brownlee, 2019). 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 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.

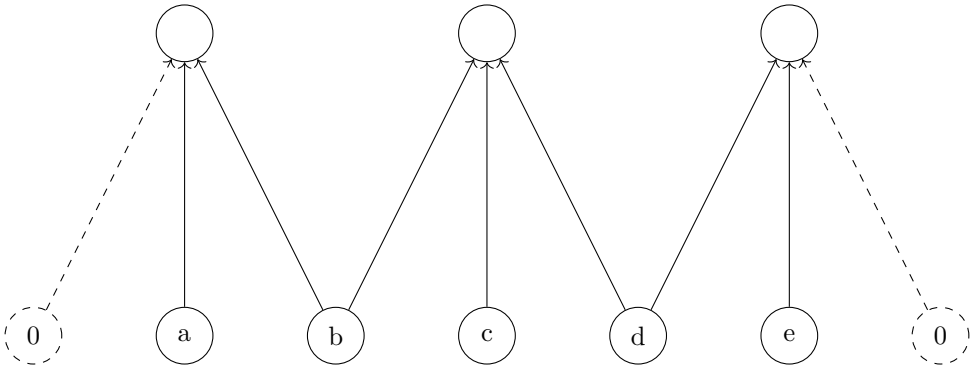

Figure 5: Example showing a layer with same padding and a stride of two.

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. 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). Our benchmark, presented in Section 5, 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.,

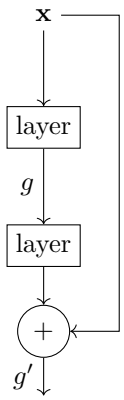

Figure 6: Schema of a residual block with two layers. $\mathbf{x}$ is added to the output of the last layer of the residual block, before the result is passed through its activation function g'.

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 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

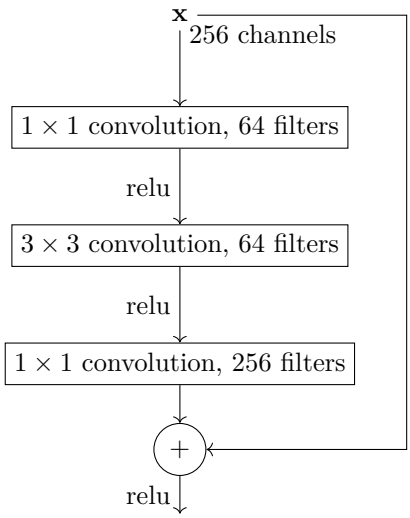

Figure 7: Example of a residual block in ResNet-50.

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).

The SpiNNaker system's basic building block is the SpiNNaker chip, a multiprocessor chip consisting of 18 ARM968 cores and a Network-on-Chip (NoC) system for communication between the cores (Furber and Temple, 2007; Furber and Bogdan, 2020). 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).

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

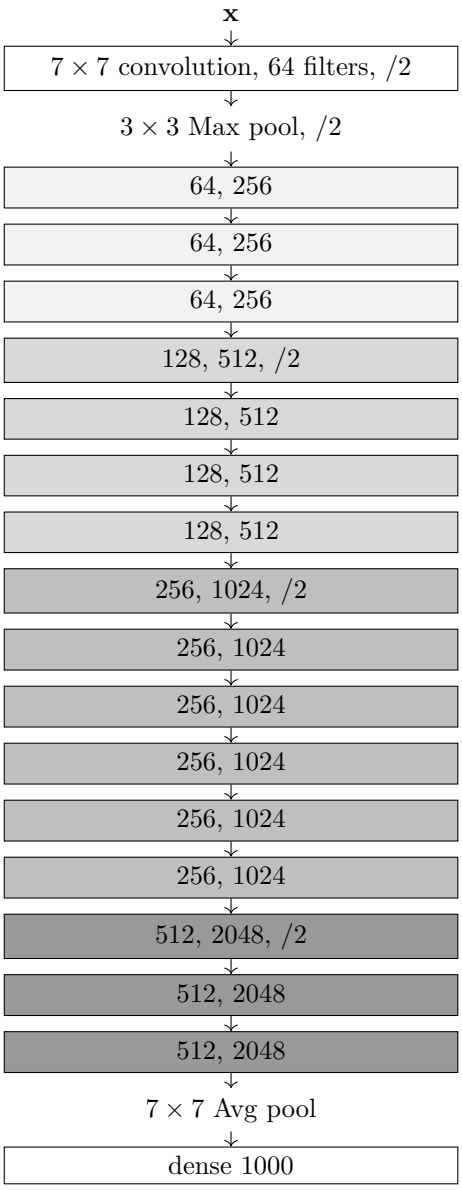

Figure 8: Schema of ResNet-50. Each block in the middle represents one residual block shown in Figure 7. The first number shows the amount of filters the first two layers of a block have, while the second number shows the filters of the last layer of the block. /2 indicates that a stride of two is applied (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.

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 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 peek 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 desire for some time. This section will outline two approaches of 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 or Caffe (Jia et al., 2014), while its back-end supports different spiking neural network simulators like Brian2 (Stimberg et al., 2019), the simulator independent language PyNN (Davison et al., 2009) or direct mappings to neuromorphic computers like SpiNNaker or Intel's Loihi (Davies et al., 2018; SNN toolbox, 2020). It 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 two times less operations than 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 an accuracy of 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, which is the far more complex task computationally.

IBM's TrueNorth neuromorphic architecture has the goal of achieving energy efficiency and performance through scalability, same as SpiNNaker. Esser et al. (2016) presents Eedn (energy-efficient deep neuromorphic networks). Eedn is an approach of generating CNNs on the TrueNorth

system, enabling both inference and training. TrueNorth—unlike SpiNNaker—uses one bit spikes (Esser et al., 2016), which means its substantially different from contemporary consumer hardware and deep learning accelerators and Eedn is needed to translate the CNN in order to make it run on TrueNorth. SpiNNaker on the other hand, like stated above, is just a collection of low power ARM cores connected over a NoC. Spikes on SpiNNaker a communicated with small multicast packets (up to 72 bits, see above) (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 shine light onto 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—a machine vertex—is directly mapped to a single SpiNNaker core. An application graph is an abstraction over a machine graph. Application vertices have atoms. Atoms are atomic units of computation. The atoms of an application vertex are distributed onto machine vertices. 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. 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).

The toolchain is written in the Python programming language. The SpiNNaker machine is connected to a host device via Ethernet (Rowley et al., 2019). The toolchain—running on the

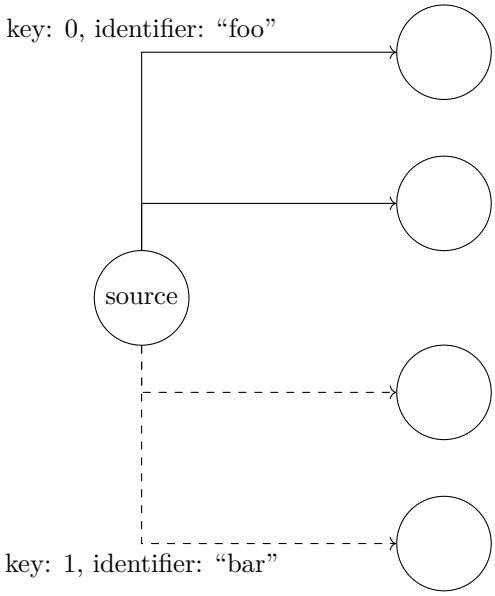

Figure 9: Example of a machine vertex “source” connected to four other machine vertices over two outgoing edge partitions. If the source vertex sends a MC packet with zero as key, the two upper vertices will receive the packet, whereas with one as key the two lower vertices would receive the packet.

host—is mainly responsible for the generation of the graph and its execution on the connected SpiNNaker machine. It 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) and finally 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 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 periodical update event, called every x cycles.

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 send 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 in the sense that packets will be send to the wrong destinations. 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 partition. A vertex can have multiple outgoing edge partitions (see Figure 9) (Furber and Bogdan, 2020).

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 and 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 straight-forward 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. 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). The second user interface are the layers—the building blocks of a model. Layers are added sequentially to a model instance by calling the model's `add` method. 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 straight forward to implement with the sequential API. Keras therefore exposes another API, its functional API. The prototype was not developed so far 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. 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 generates the weight matrix $\mathbf{W} : n \times m$ and the bias m-vector. n is the amount of neurons in the preceding layer, m the amount 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 formats of the weights is again inspired by Keras and enables seamless interoperability between a SpiNNaker deep learning

```
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 Keras to the prototype. The code would result in a model akin to the one shown in Figure 10.

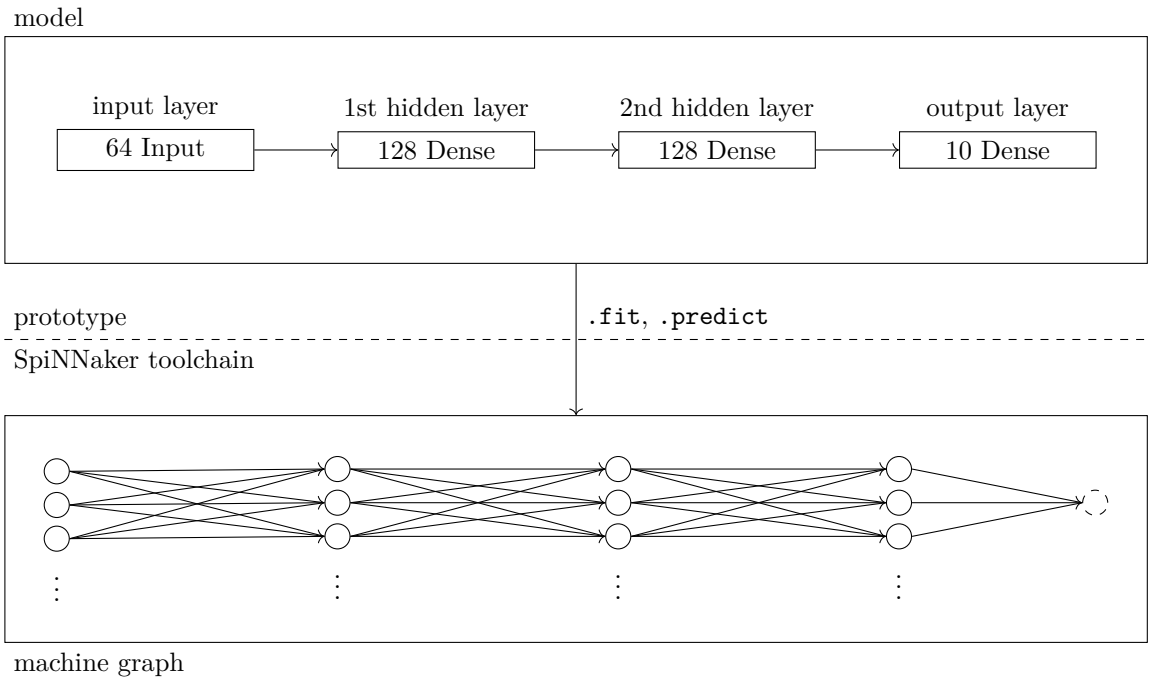

Figure 10: Illustration of how a machine graph is generated by the prototype. The dashed circle represents an auxiliary machine vertex, in this case the LPG. This machine graph would be generated when the `predict` method of the model is called.

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 29ff). The same goes for testing backpropagation, where we could simply train both models and compare the updated weights.

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 2). 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 descend is probably the most common optimization algorithm for deep learning (Goodfellow et al., 2016). 1D convolutional layers have support for same and valid (default) padding and they can be strided (see Section 2.1).

Like 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 and for training the `fit` method.

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 of SpiNNaker, back to the host. 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 resetted 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)

```python
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
keras_model.add(tf.keras.layers.Conv1D(20, 3, input_shape=input_shape))
keras_model.add(tf.keras.layers.Conv1D(5, 3, activation="relu"))
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"))
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 2: Example code comparing 1D CNNs in Keras to the 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 holds 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). The `init_neurons` method furthermore 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 every 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.

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 6.

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}
```

Like 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.

How are received packets matched to the correct weight? All packets send 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[1]. During data generation (see Section 4.1) 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 all the neurons from the previous layer, connected to the neuron which is creating its mapping between packets and weights, are collected. The keys are guaranteed to be consecutive. For example, the first neuron of the input layer of the MLP from Figure 10 would have zero as its key. 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 difficult, 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 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 2ff). 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 backpressure, which leads to packet loss (discussed below).

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
```

1. Not actually one partition, but lots of outgoing edge partitions with the same identifier (see Section 4.1).

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, like 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 send forward) or an error (meant to send 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 than 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 now 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 where 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. Everytime 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 obvioulsy bound by the key space being 2^{32} items big), so it is no problem to handle all four parititions 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 amount 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 is 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. But what is the label input layer connected to? 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:

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

y being the k-vector encoding the true label and $\hat{y}$ being the k-vector result of the output layer.

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:

$$\frac{\delta L}{\delta \hat{y}_i} = 2(\hat{y}_i - y_i). \tag{15}$$

$\delta L / \delta \hat{y}_i$ 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. Like 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.

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
```

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 reasons why that was believed in the first place. We thought that the their general approach to multiple channels and filters would lead to complications on SpiNNaker. 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. Like stated above, the router blocks until a packet is received (the receive event callback finished). If the receive event callback takes long, 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

- time
- mention problem with LPG
- what is all missing (2D, Pooling and Shortcut connections)
- comm structure of the backward pass
- Too much time spend in receive callback
- interpreting neurons as domain decomposition over linear algebra compute graph
- backward pass: gradients computed two times so comm fabric is not overly used by unique partitions/lots of unused packages
- How I crushed nd-kernels into a single blog of weights
- backprop design change (shared weights $\rightarrow$ redundant packets (easier and nicer to implement)). Really stress this point and present all three things tried (e-mail).
- resource invariance (first prototype, deemed too difficult to put resources into it as well (just three months time))
- bottlenecks (one layer much higher computational effort but even less resources (10–1024–20)
- conv neurons of resnet way too big and are horrific scalers (small spatial input but big channels computationally much more effort but again, less resources)

5. Benchmark

6. Discussion

- present possible solutions for problems encountered
- space used inefficiently (cores and memory) → better domain decomposition
- 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

7. Conclusion

8. 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 2019. URL https://machinelearningmastery.com/crash-course-convolutional-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.

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.

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.

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.

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.

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.

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/.

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.

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/.

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.

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.

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/.

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.

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.