TECHNICAL UNIVERSITY OF CRETE
School of Electrical and Computer Engineering

DIPLOMA THESIS

Modelling Error Gradients in Deep Learning Methods Using Reconfigurable Hardware

by

Michail Fragkiadakis Theodorouleas

Thesis Committee:
Professor Apostolos Dollas (Supervisor)
Professor Dionisios Pnevmatikatos
Associate Professor Ioannis Papaefstathiou (AUTH)

A thesis submitted in fulfillment of the requirements for the degree of Diploma in Electrical and Computer Engineering.

December 2018

Abstract

During the training process of a Neural Network there is significant amount of resources that remains unused due to data dependencies, waiting for the forward pass and the error backpropagation to complete. Decoupled neural interfaces using gradient error modelling were introduced in order to overcome this pitfall allowing each layer to be updated before the backpropagation is complete and it is provided with the error gradient.

In this diploma thesis we examine the parallelisation of decoupled neural interfaces operations when implemented on Field Programmable Gate Array (FPGA), configurations that can decrease training time as well as the effects of decoupled neural interfaces regarding the training ability and the accuracy of the network.

In this thesis we adjudge that:

- Integration of synthetic gradient error does not have negative effect in accuracy and representational strength.
- The addition of synthetic gradient error causes the addition of noise in the training error which results in training error regularisation. This is beneficial for the training process as it broadens the exploration of error space, decreases the generalisation error of the neural network and prevents from overfitting the training dataset.
- Synthetic gradient error modelling can accomplish decrease of training time only in particular cases.
- The combination of synthetic and true gradient error increases the number of neural layers error correction updates and accelerate the convergence rate of the training error.
- Despite the fact that the combination of synthetic and true gradient error increases the training pass latency, the overall training time can decrease due to the higher error convergence rate.

Περίληψη

Κατά τη διάρκεια της διαδικασίας εκμάθησης ενός Νευρωνικού Δικτύου, σημαντικός όγκος πόρων παραμένει αχρησιμοποίητος, εξαιτίας εξαρτήσεων πληροφορίας, περιμένοντας την ολοκλήρωση του υπολογισμού της εξόδου και της όπισθεν διάδοσης του λάθους. Οι "Αποσυζευγμένες" (μη συζευγμένες) Νευρωνικές Διεπαφές (Decoupled Neural Interfaces) με χρήση μοντελοποίησης κλίσης σφάλματος έχουν προταθεί για να ξεπεράσουν αυτό το εμπόδιο, επιτρέποντας σε κάθε επίπεδο να ενημερώνεται προτού η όπισθεν διάδοση ολοκληρωθεί, ώστε να του γνωστοποιηθεί η πραγματική κλίση σφάλματος.

Σε αυτή την διπλωματική εργασία εξετάζουμε την παραλληλοποίηση των διεργασιών των αποσυζευγμένων νευρωνικών διεπαφών κατά την υλοποίησή τους σε Field Programmable Gate Arrays, διατάξεις που μπορούν να μειώσουν τον χρόνο εκμάθησης, όπως επίσης και την επίδραση των αποδεσμευμένων νευρωνικών διεπαφών όσον αφορά την ικανότητα εκμάθησης και ακρίβειας του δικτύου.

Η συνεισφορά της παρούσας εργασίας είναι:

- Η ενσωμάτωση της μοντελοποίησης κλίσης σφάλματος δεν έχει αρνητική επίπτωση στην ακρίβεια και στην ικανότητα αναπαράστασης.
- Η προσθήκη της μοντελοποίησης κλίσης σφάλματος προκαλεί την προσθήκη θορύβου στο σφάλμα εκμάθησης το οποίο έχει ως αποτέλεσμα την κανονικοποίηση του. Αυτό είναι ευεργετικό για την διαδικασία εκμάθησης, καθώς διευρύνει την εξερεύνηση του χώρου σφάλματος, μειώνει το σφάλμα γενίκευσης του νευρωνικού δικτύου και αποτρέπει τον περιορισμό στη λύση των δεδομένων εκμάθησης (overfitting).
- Η μοντελοποίηση κλίσης σφάλματος μπορεί να πετύχει μείωση του χρόνου εκμάθησης μόνο υπό συγκεκριμένες συνθήκες.
- Ο συνδυασμός της μοντελοποίησης με την πραγματική κλίση σφάλματος αυξάνει τον αριθμό ενημερώσεων διόρθωσης σφάλματος των νευρωνικών επιπέδων και επιταχύνει τον ρυθμό σύγκλισης σφάλματος.
- Παρόλο του ότι ο συνδυασμός μοντελοποίησης και πραγματικής κλίσης σφάλματος αυξάνει τον χρόνο ανά κύκλο εκμάθησης, ο συνολικός χρόνος εκμάθησης μειώνεται, εξαιτίας του υψηλότερου ρυθμού σύγκλισης σφάλματος.

Acknowledgements

First of all, I would like to thank Dr. Antonios Nikitakis, MHL laboratory research assistant for the thesis topic proposal, his valuable advice and our collaboration throughout the entire journey of this research. I would also like to express my deepest gratitude to Prof. Ioannis Papaefstathiou and Prof. Apostolos Dollas who have been my supervisors for their trust in my abilities and their excellent guidance, that have been more than helpful and crucial for the fulfillment of this thesis, as well as Prof. Dionisios Pnevmatikatos for participating in the committee .

Contents

Chapter 1

Introduction

Supervised Neural networks (NNs) are a main paradigm of the artificial intelligence field that has provided answers to many problems that were impossible to solve with traditional engineering approaches. This revealed a new engineering approach aiming in learning from data rather than defining a solution. A main factor that has contributed to the success of supervised learning using NNs is the growth of dataset size, a key point to the training process. This raised the need to design efficient algorithms for the processing of large size datasets.

Although backpropagation is the dominant algorithm for training supervised NNs, it has a big drawback due to the fact that both forward pass and backpropagation are computed sequentially. Forward pass is carried out by passing the input data through every layer of the network, connecting the output of one, to the input of another, while backpropagation is the application of the chain rule, which is based on sequential computation of a complex derivative, thus this two processes cannot happen in parallel as they both use the same memory, the layers parameters. The nature of NN results in significant amount of lost processing resources, remaining inactive during the training process before error correction can take place.

Decoupled neural interfaces (DNI) is one of the models that have been introduced to address this problem with the addition of a single layer gradient error predictor called synthetic gradient (SG). This predictor provides an estimation of the training error gradient for a layer sooner than backpropagation is complete. This predictor receives the output of a layer along with the training example labels as an input in order to produce an estimation of the gradient error of the next layer which is needed for error correction using only the local information of the layers output and the training labels. The predictor is updated afterwards, according to the true gradient (TG) of the error that results from backpropagation. This means that SG can be integrated in any type of NN as it uses the output of the hidden layers, with no dependencies on how this is produced. When neural layers are provided with a gradient error (the SG), they can perform weights update before backpropagation is complete. In this way, operations of the hidden layers of a NN can be parallelised, which results in unblock from the waiting for error gradient to be computed from the rest of the network. This reduces communication time and data dependencies between seperate layers of the network and can lead in acceleration of the training process in deep architectures.

The scientific contribution of this thesis constitutes of two parts. First, we review the DNI model and examine its performance regarding the affection of the learning ability and accuracy of the NN. We also examine the time complexity and parallelisation of DNI operations compared to that of operations of normal backpropagation. Second, we continue in the main purpose of our research, investigating the possibility of expanding DNI model by combining SG with TG which in order to increase the error decreasing rate of the NN layers during training.

The experiments we executed during our research were conducted on a fully connected, feedforward, mini-batch NN with three hidden layers that, in its initial form, uses backpropagation for error correction during the training process. It is one of the fundamental architectures of neural networks that can present the layers lock problem without introducing further issues. Despite that, all of the models we investigate in this thesis can be applied in different NN architectures. All of the models presented, are implemented on the Field-Programmable Gate Array (FPGA) Xilinx Zynq UltraScale+ board (xczu9eg-ffvc900-1-i-es) in order to define the time complexity. The performance of the training process was analysed through functional simulation of each of the model, designed in Python.

Following the introduction chapter, in chapter two we present the theoretical background of machine learning and neural networks. Prior work that has been done regarding the training of neural networks, which is the main part of neural networks we cover in our research, is described in chapter three. In chapter four we explain the modelling and functional simulations of the architectures that we explored and we continue in chapter five, with the presentation of the analysis and design we proceed for the implementation of the hardware architectures of our models as well as the results of our research. Chapter six consists of the conclusion and the proposed direction of future work.

Chapter 2

Theoretical Background

2.1. Machine Learning

One formal rule can be easily comprehensible for a human. The memorization and combination of too many formal rules can be a difficult task, while for computers it is quiet easy. This is why conventional use of computers was very successful in finding solutions in problems that could be described by a list of formal, mathematical rules.

A big challenge for computers, was to deal with problems that cannot be easily described by such rules and as a result a step by step solution cannot be defined, as well as, to be able to solve problems by processing natural data in their raw form. This raised the need for more sophisticated algorithms, that could create and optimise a solution rather than just implement defined solutions. The goal for these algorithms is to get an abstract description function of the problem and find the parameters that accomplish the best solution.

Learning is any process by which a system improves its performance. Humans learn through observation, previous mistakes or advice. This is the motivation to create machine learning algorithms. In cases we are not able to describe a solution for a problem, we can instead create an algorithm that can learn it through examples or through a trial and error procedure.

"We define machine learning as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty (such as planning how to collect more data!)." [1]

Machine Learning is concerned with computer programs that automatically improve their performance through experience. It is the field of predictive modelling that primarily concerned with minimizing the error of a model or making the most accurate predictions possible, at the expense of interpretability.

2.2. Supervised learning

A fundamental model for learning from examples introduced by Vapnik [2] consists of:

1. The generator of the data (examples), G.
2. The target operator (also called supervisor's operator or for simplicity supervisor), S.

3. The learning machine, LM.

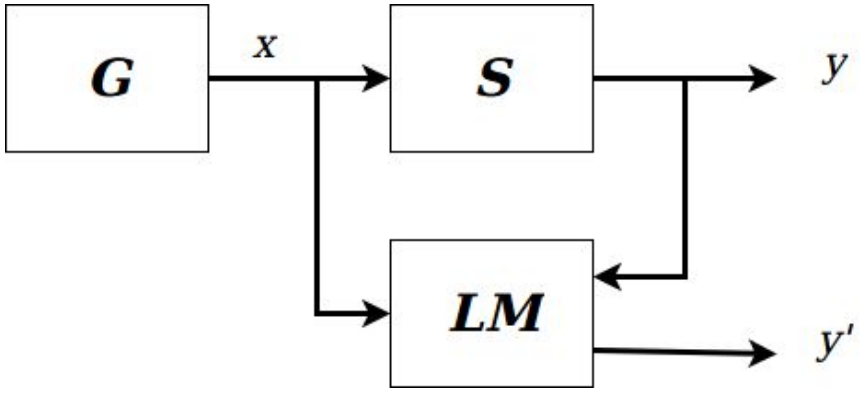


***Figure 2.1**: Supervised learning system*

The generator (G) produces example inputs of the environment of the supervisor. These examples are considered to be independently and identically distributed although we do not own knowledge of this distribution. We consider these inputs a vector xX and the unknown distribution F(x). In real life problems these can have more variables. These examples are input of the supervisor which produces an output y for every example. The relation between x and y is unknown, but we suppose it does exist and this is what we are trying to define. If this relation is random, it would be impossible for a prediction to exist.

The pairs (x,y) are the training set that learning machine observes during the learning process. During this process, the learning machine constructs an operator that predicts the output yi of the supervisor for a given input xi. The prediction of the LM (output) is the y'. The training error is defined as the distance between y and y' and the goal of the learning process is for the LM to modify its input-output relation in response to this error in order to minimise it [2]. In most supervised learning training implementations the supervisor module S is not an actual operator that the learning machine is trying to imitate. It is a set of pairs (x,y) of a known examples. This is why a key point of such a system for achieving its task is the number of available examples for the training process [3].

Many of the learning algorithms that have achieved significant results have not had any important improvement for decades. The main development that has led to increase of accuracy is the size of the training datasets.

2.3. Artificial Neural Networks

2.3.1. Structure

Artificial neural networks are one of the most important paradigms of machine learning models that exist today, with a great success in problems like classification, voice and image recognition and signal denoising. The fundamental module of a NN is the perceptron which was introduced as a linear threshold unit [4] in 1943. The idea that perceptron could help in learning representations was established in 1958 [5]. Perceptron is a simple unit that has many inputs and produces a single output. The inputs are combined and the output is computed according to the activation function of the perceptron which defines weather the output should be 0 or 1.

In fact, the perceptron is a classifier that produces a binary decision according to its inputs.

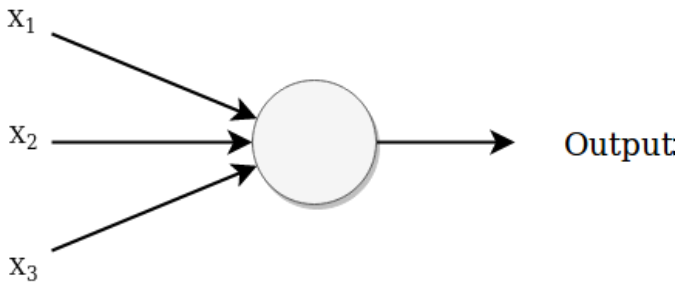


***Figure 2.2**: Perceptron*

The combination of the inputs x is a sum with respect to some weights w of importance. The activation function applies a threshold on this sum and is set to active if it is above this:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \quad (1)$$

This function can be simplified by replacing the sum of the product of the two vectors with the inner product and moving the threshold to the left side defining it as the bias b which is the bias of the neuron:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \quad (2)$$

The use of a threshold has the drawback of non continuous function. A small change in one weight can cause the activation function to meet the threshold and the flip of the output. Furthermore a derivative of the activation function is necessary for reasons explained in chapter 2.3.4.

For the reasons above, the threshold function is replaced with a squishification, nonlinear function. The first squishification function used is the sigmoid function:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \quad (3)$$

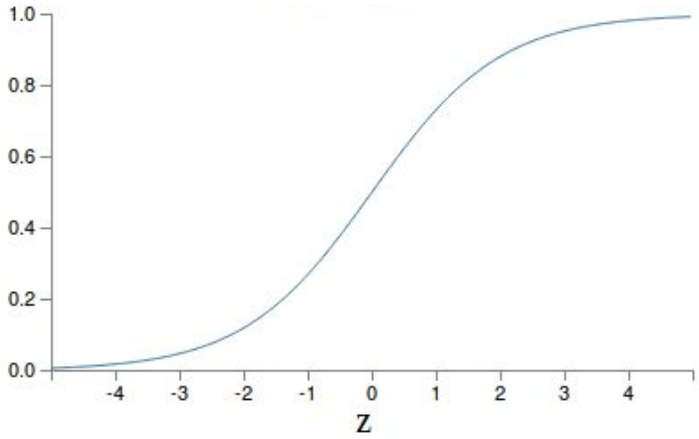


***Figure 2.3**: The sigmoid function*

The sigmoid meets the requirements of continuity and the existence of derivative while it is a bounded function. The most important advantage that was revealed with the use of nonlinear functions though, is that NNs that use such activation functions can represent solutions for both linear and nonlinear problems. In this way complex solutions for non trivial problems are feasible. The mathematical representation of the sigmoid perceptron is

$$y = \sigma(wx + b) \quad (4)$$

Where w is the vector of weights, x is the vector of inputs, b the vector of biases and y is the vector of the output. Perceptrons combined in parallel form a neural layer that produces a sequence of real-valued outputs, called activations. A layer of N perceptrons functions as a classifier of 2^N classes. As we can see in figure 2.3, the number of weights required for a layer occurs from the number of inputs times the number of outputs.

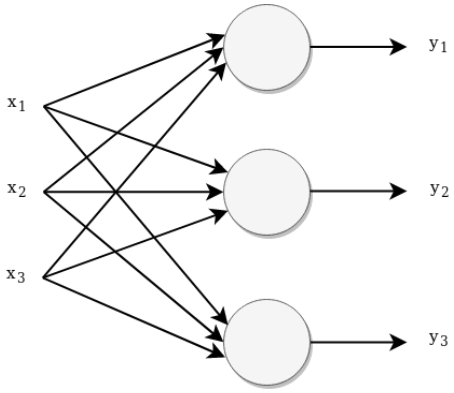


***Figure 2.4**: Neural layer*

Combining such layers seemed promising for complex classifiers but there was still no way for defining appropriate internal parameters to accomplish that. It was in 1974 that backpropagation was introduced as a solution for that [6]. Backpropagation is an algorithm that could achieve a minimisation process for prediction error resulting from a complex system. Now, combination of many such layers with the activations of a layer fed as an input to another was feasible as there was a way of tuning this whole system. Such combination forms a NN. For that reason NNs are also determined as multilayer perceptrons. It is actual a mathematical function, a mapping of inputs to outputs, but this is done through a complex system with a large number of parameters. There are many ways that layers can be connected between them and many different architectures have been demonstrated.

Here is an example of a feedforward, fully connected neural network. (Feedforward means that activations propagations is done only in one direction, from input to output. Fully connected is a network in which every output of a layer is connected to all the inputs of the next layer.

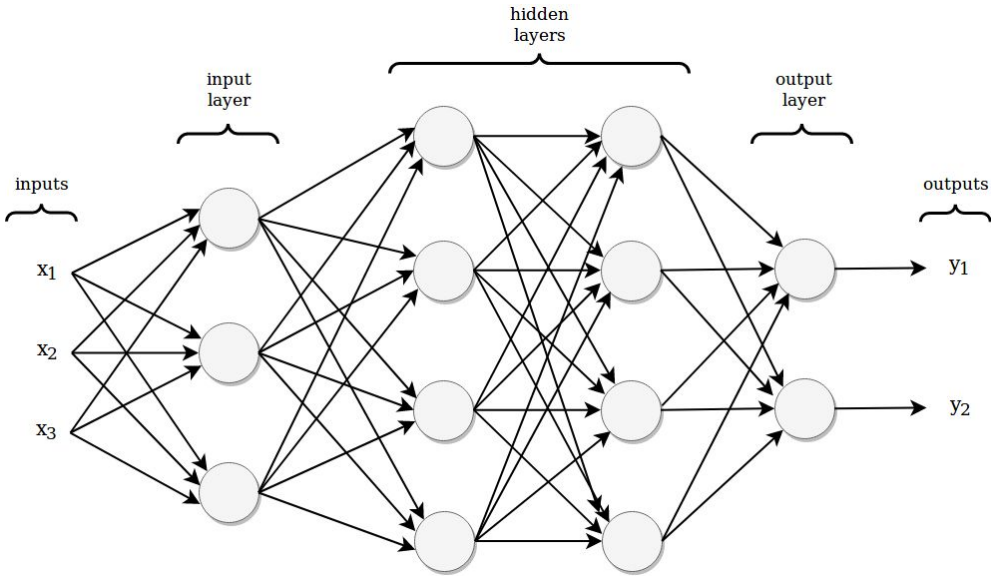


***Figure 2.5**: Fully connected neural network*

Inputs are the data we feed in the network for processing in its raw form. The first layer is the input layer of the network. It has dimension and values of the form of the input data. The intermediate layers are the hidden layers, it is the place where internal, abstract representations of the data are created by the model during the learning process. The last layer is the output layer, which has a dimension that can form the desired representation of the solution. The outputs are the activations of the perceptrons of the output layer.

If a layer given some inputs, can produce a set of decisions that will be fed in a next layer, this means that the next layer will make a more complex decision depending on the decisions of the previous one.

When the NN receives an input, sequential calculations take place and the activations of every perceptron of every layer propagate forward, towards next layers, deeper into the network. This is the forward pass (FP) operation.

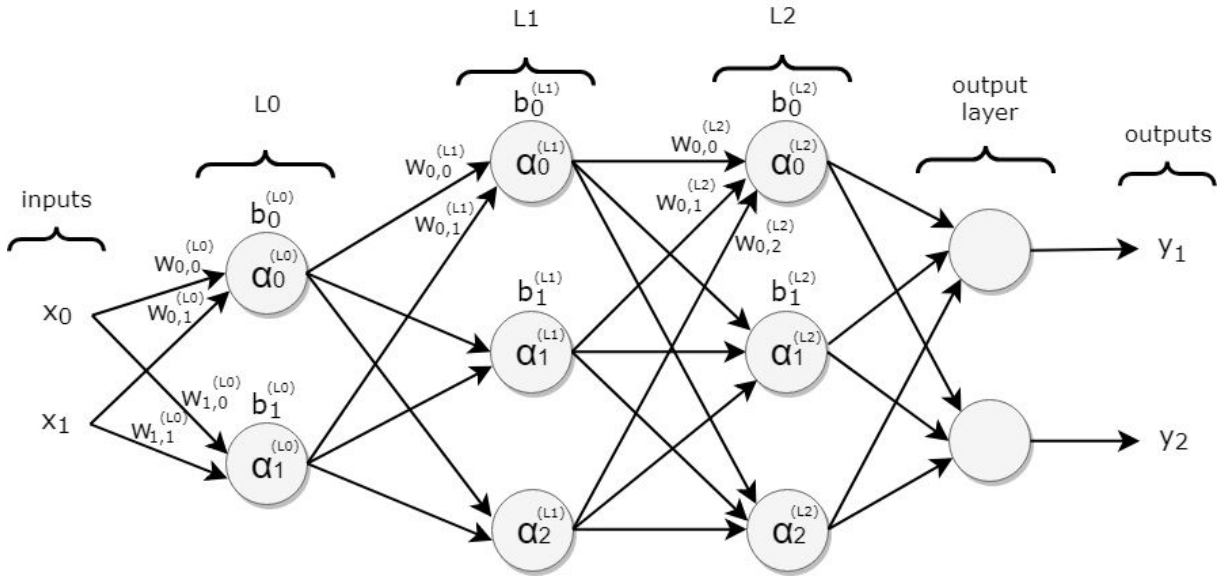


***Figure 2.6**: Forward pass*

According to the notation of the network above, the activation $a_0^{(L0)}$ of the first neuron of the layer L_0 is:

$$z_0^{(L0)} = x_0 w_{0,0}^{(L0)} + x_1 w_{0,1}^{(L0)} + b_0^{(L0)} \qquad (5)$$

$$a_0^{(L0)} = \sigma(z_0^{(L0)}) \qquad (6)$$

where $w_{0,i}^{(L0)}$ are the weights, $b_0^{(L0)}$ the bias, $z_0^{(L0)}$ is the output of the perceptron and σ the squishification function of the neuron. We calculate $a_1^{(L0)}$ the same way and then the activations are propagated to the layer L1.

$$z_1^{(L0)} = x_0^{(L0)} w_{1,0}^{(L0)} + x_1^{(L0)} w_{1,1}^{(L0)} + b_1^{(L0)} \qquad (6)$$

$$a_1^{(L0)} = \sigma(z_1^{(L0)}) \qquad (7)$$

This procedure is repeated until the activations of the output layer are calculated. The general function for calculating the forward pass (also named activations) a of the neuron j in layer L is:

$$a_j^{(L)} = \sigma\left(\sum_i^N a_i^{(L-1)} w_{j,i}^{(L)} + b_j^{(L)} \right) \qquad (8)$$

where N is the number of inputs of the neuron,equal to the number of activations of the previous layer.

The forward pass equations differ according to the architecture and the way knobs of the network are connected but the procedure is the similar. Every layer receives an input and produces an output according to its weights, bias and the squisification function. This output is the input for the another layer that repeats this process.

2.3.2. Deep Neural Networks and Deep Learning

In 1089, the function approximation theorem [7] proved that a neural network with a single hidden layer could approximate any continuous function but there was no proof for the required number of parameters and training examples. Through many experiments on the architecture of NNs (number, width and connection configuration of the layers) it was revealed that multiple processing layers can learn representations of data with multiple levels of abstraction. These architectures allow the decomposition of a complex input, into many different, abstract representations that each one is described by a layer of the network. For this reason these architectures were named deep neural networks. A neural network is defined as "deep" when it consists of more than one hidden layer.

It became feasible for the computer to have a deep understanding of the perceived information, like the understanding we have of an image we are looking at, or even deeper than a human get. This was the idea that led to a new approach machine learning that by learning from experience, interprets the world by decomposing it in simple concepts that are comprehensible to computers and achieve multiple levels of abstraction. Combining them in a hierarchical way enables computers to understand complicated, real world concepts.

> *"If we draw a graph showing how these concepts are built on top of each other,*
> *the graph is deep, with many layers.*
> *For this reason, we call this approach to AI deep learning."* [8]

A different meaning of the term "deep learning" arises from the deep understanding of the information that is accomplished in this way, that can lead in the solution of complicated problems. Being able to learn through experience discharges the designers from specifying all the knowledge that is required for a solution, overcomes the restrictions of knowledge based feature engineering approaches and makes this deep understanding attainable. This can lead to solutions that use knowledge that was never defined by the designer. This can be considered to be a worthwhile definition of artificial intelligence.

2.3.3. Training

Learning is process by which weighting parameters of a NN are adapted through some process correction in order to implement a corresponding mapping between inputs and output of the network [9].

In supervised NNs, we define the learning process as the training process in which known examples are given as inputs to the NN and an output is produced. Comparing the output of the NN with the desired output of the example we compute the training error according to a loss function that determines the distance between the example and the predicted outputs. The training error is then fed back to the NN which adapts its internal weights, aiming to minimise the training error.

The training error is computed in the output of the network but each of its internal parameters had a different impact on that. The way that this learning system can "distribute" this error to all the internal parameters of the network, depending on the partial contribution each one had to the final error, is by using the backpropagation algorithm.

Given its partial error, each parameter is adapted by applying gradient descent. This appeared to be a very efficient way of finding the appropriate input-output relation of a NN in order to define a representation function for solutions in complex problems [10].

2.3.4. Gradient Descent, Chain Rule & Backpropagation

Gradient descent is a first order algorithm for finding the minimum of a function using the gradient of that function.

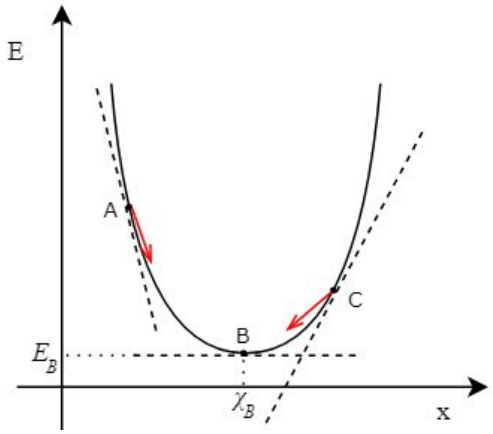


***Figure 2.7**: Gradient descent*

Consider the function $E(x)$ above. Point B is the the minimum of the function we are looking for. This means that we seek the appropriate x_B such that $E(x_B) = E_B$. Point B is a minimum, which means that $dE(x_B)/dx = 0$. Gradient descent algorithm checks the gradient of the function for a given x in order to determine this is point like A or like C. Points with

$E(x)>E_B$ for $x<x_B$ have a negative gradient while points $E(x)>E_B$ for $x<x_B$ have a positive gradient. For both of these cases, it is known in which direction a minima exists and we can proceed in making a step towards this direction, which is actually direction opposite of the gradient.

For a random given function parameter value x_c, we find the new appropriate X_{new} that will the error E function to make a step closer to the minimum according to the gradient descent algorithm:

$$X_{new} = x_c - a\, x_c\, dE(x_c)/dx \qquad (9)$$

The parameter α in the equation above is the learning rate that defines how "big" will be the distance we want to move towards the minima. It is important to specify an appropriate α so that our step is neither too small, so it takes too long to reach the function minima, nor too big because we might overshoot the minima point.

During the training process of a NN we can consider its input/output relation as a function with the internal weights as an input and training error E as an output. By applying the gradient descent algorithm on that NN function $E(w)$ we are trying to minimise the training error. This is a very simple example though. In NNs there are hundreds of weights and biases parameters combined both in parallel and in sequence to produce the output E so we need to find the relative proportion to the change of every weight and bias that affects mostly the error function decrease. In order to apply the gradient descent algorithm for every parameter of the NN we need the partial derivative of the E with respect to this parameter. We can compute that by using the chain rule.

The chain rule is a formula for calculating a partial derivative of a complex function. Given function $f(x)$ and $g(f(x))$ we calculate the partial derivative of g with respect to x using the following rule.

$$g'(f(x)) = g'(f(x)) \cdot f'(x) \qquad (10)$$

$$\Rightarrow \quad dg/dx = dg/df \cdot df/dx \qquad (11)$$

We start by applying the chain rule in a single input of a single neuron of the NN. In order to do that, we need to represent the neuron as a computational graph as follows.

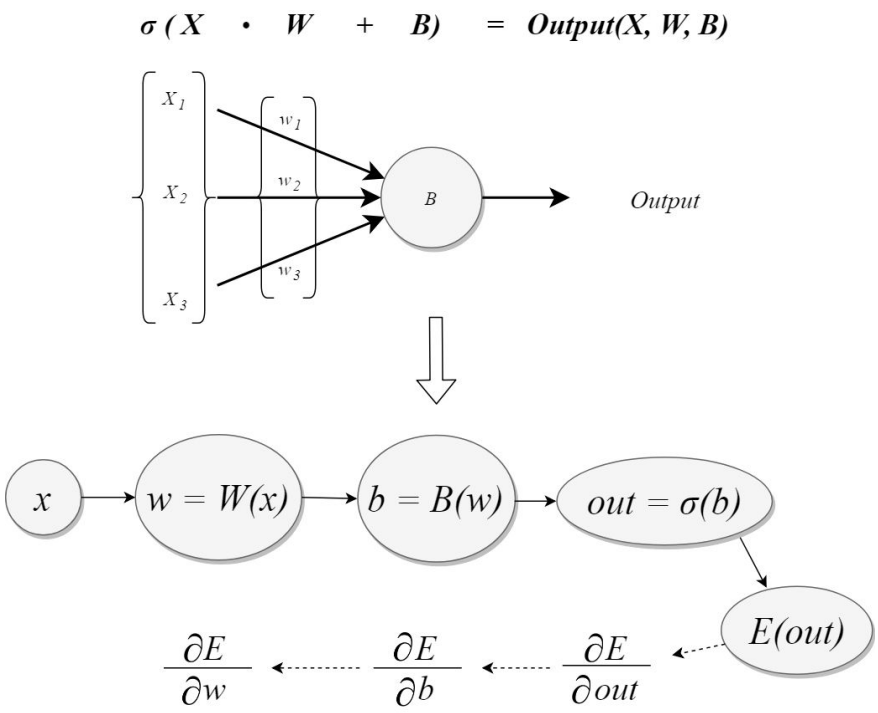


***Figure 2.8**: Chain rule on a neuron*

We consider the output of the neuron like this presented in chapter 2.3.1 as a function of inputs *X,* weights *W*, biases *B* and squisification function σ and an error function *E* of the output. *E* is the function we are trying to find a minimum for, by calculating the minimum with respect to every parameter, except the input X, by using the gradient descent algorithm. We do not do this for the parameter *X* as we wish to minimize the error in general, independently from the input. To do that, we need the partial derivatives of *b* and *w*. The single parameter derivative $\partial E/\partial out$ is easy to compute for a known function *E* as it is equal to the derivative of *E*. Then we continue for the rest of the partial derivatives by applying the chain rule iteratively:

$$E'(\sigma(b)) = E'(out) \cdot \sigma'(b) \Leftrightarrow \partial E/\partial b = \partial E/\partial out \cdot \partial \sigma/\partial b \quad (12)$$

$$E'(\sigma(B(w))) = E'(\sigma(b)) \cdot B'(w) \Leftrightarrow \partial E/\partial w = \partial E/\partial b \cdot \partial B/\partial w \quad (13)$$

Note that we first need to find the partial derivative of the parameter that is “closer” to the output and then we can do the same thing step by step for parameters that are far from the output.

Applying the chain rule to find the partial derivative of the training error with respect to the internal parameters in complex NNs is called backpropagation. The error derivative that

is computed in the output of the network it is propagated, step by step, back to the internal parameters.

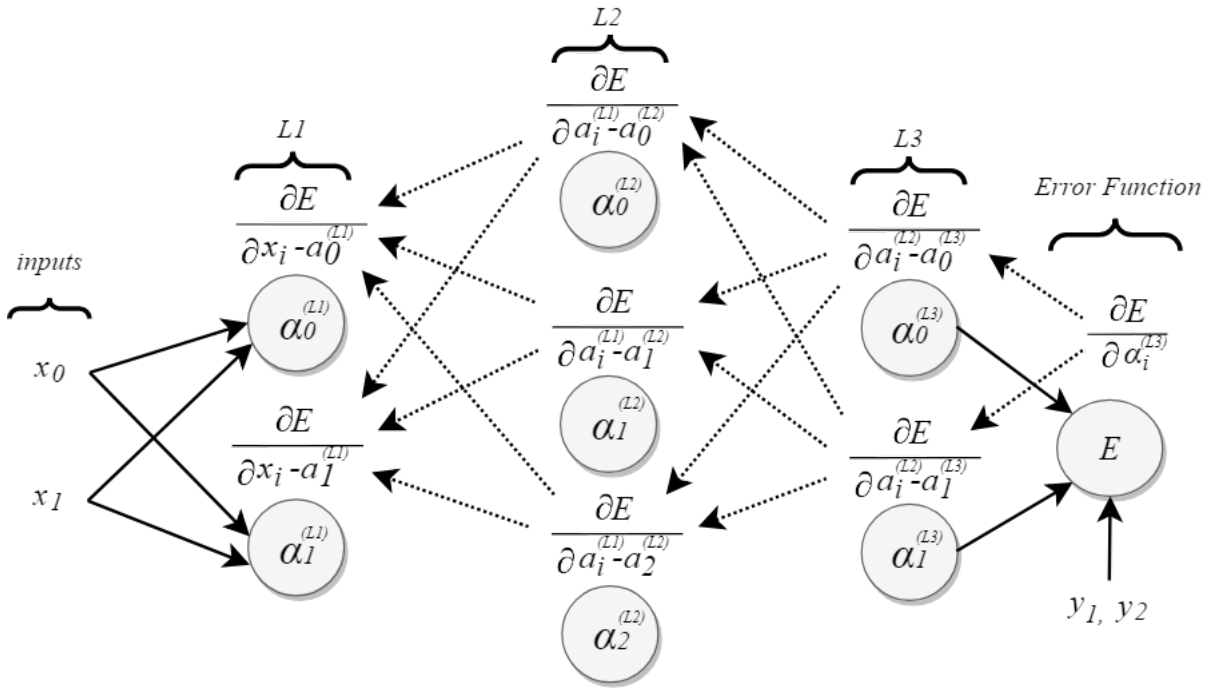


***Figure 2.9**: Backpropagation*

In this way, we calculate the partial derivative of the error in respect with every activation. We can use that in order to calculate the partial error of every one of them, meaning "how much" a specific activation contributes to the error. [11]
First, we calculate the error for layer L3:

$$\delta_i^{(3)} = \alpha_i^{(3)} \cdot E \cdot \partial E / \partial \alpha_i^{(3)} \qquad (14)$$

Then we can apply the chain rule for the weights and bias of the neuron to correct them according to the error of the activation,

$$\delta w_i = w_i \, \delta_i \quad \partial \, \delta_i / \partial \, w_i$$
$$w_i^{new} = w_i^{current} - \delta w_i \qquad (15)$$

$$\delta b_i = b_i \, \delta_i \quad \partial \, \delta_i / \partial \, b_i$$
$$b_i^{new} = b_i^{current} - \delta b_i \qquad (16)$$

and continue propagating backwards to layer L2.

$$\delta_i^{(2)} = \alpha_i^{(2)} E \quad \partial E / \partial \alpha_i^{(2)} \qquad (17)$$

This process is repeated until all the parameters of the network are updated according to the current error. The goal is to minimize the overall error E_{ov} upon the different examples of the training set, according to the loss function E of the predicted and the desired outputs y_{ex} and y_{ex}'.

$$E_{ov} = \sum_{ex} E(y_{ex,}\ y_{ex}') \qquad (18)$$

This usually is achieved by parsing the whole training set iteratively until E_{ov} has reached a desired level. This iterative solution is easy to generalise and can replace the analytic solution of a NN function that is limited by the need of linearity in the function in order to be applied. [11]

2.3.5. Training Objectives

When designing and experimenting on the training process of a neural network we aim to reach the desired levels in the following three objectives:

1. Convergence speed/learning ability is the rate of training error decrease. When taking steps along the shortest path to the minima, the error decreases faster than having some deviation from that. Furthermore, making bigger steps towards these directions increases the convergence speed. On the other hand, big steps may causes overshooting the global minima. This means that the fastest way is not always the optimal because this can limit the generalisation of the solution (overfitting on the training examples). Learning ability could become weak if the training procedure stucks in a local minima rather than the global one which is supposed to be the best case.

2. Representational strength/accuracy is the performance of the NN upon data different from these that the it was trained with. This simulates the performance in a real application where inputs will actually be unknown. Representational strength is the accuracy of the prediction on previous unseen data. For this reason we use only a part of our dataset for the training. This is the training set. The part of the dataset that was not processed by the network before, is used to measure the accuracy. This is the test set. The more general the solution found during the training process is, the better accuracy we can achieve on the test set data. The accuracy can be defined using various metrics according to the application. In some cases false positives of false

negatives are more important from the percentage accuracy, so an appropriate metric is needed.

3. Computational complexity/training time is the latency of a training pass of a NN. A training pass is the procedure of processing the needed number of examples for the training error to be computed and the execution of the error correction update. Depending on the architecture of the network, the loss computation and the update method varies on how computational complexity of a training pass. Less complex training methods can process a training dataset faster, making the training process less time consuming.

Chapter 3

Relevant Research

Neural networks have proved to be capable of finding solutions in a huge range of problems more efficient than it has never been before. Therefore there has been a significant raise of interest in the following domains. First, in the creation of appropriate network configurations, regarding the number and the width of layers, that can achieve best results for different applications. Second, in the effort to make both inference and training process faster. NNs use huge amounts of data and iterate hundreds or even thousands times during the training procedure. This means that even small speedup improvements can achieve significant overall acceleration. Third, in reducing operation cost for large scale applications. The purpose of this thesis is to investigate models that achieve acceleration in the training process, for this reason, in this chapter we present prior research and applications that have been done towards this direction.

3.1. Batch/Mini-batch Gradient Descent

Batch and mini-batch gradient descent are variations of the gradient descent algorithm in order to decrease the frequency of backpropagation passes and layer updates. Batch gradient descent makes forward passes of the entire training set and calculates an average error while mini-batch gradient descent splits the training dataset into small batches and calculates an error for each one of them. Then the backpropagation pass takes place using this single error [12].

This methods achieve massive decrease of time complexity because the update lock of the neural layers takes place a lot more rarely. Furthermore, computational complexity decreases due to parallel computation of different examples during the forward pass. Finally, the averaging of gradient error makes the error correction smoother which generalises the solution and prevents it from stucking in local minimas, increasing the representational strength of the model. However, the synchronization cost of mini-batch training is potentially still too large for large scale applications.

3.2. Noise During Training

A common problem in deep neural networks is the case where the training error decreases while test error remains the same or even gets larger. This phenomenon is called overfitting the training data and it can happen when a model learns the details and noise in the training data instead of a general solution.

One way of preventing overfitting is to increase the size of training data, but this is not always feasible. For this reason regularisation techniques were developed. Regularisation is any modification to a learning algorithm to reduce its test error without reducing its training error.

A common way of applying regularisation is by injecting noise in the network during the training. Techniques like this are the Dropout, presented in [13], or alternations of dropout like [14], [15] and [16]. In [17] it was suggested that noise injected in the networks weights creates a more fault tolerance solution than normal training while also increasing the accuracy of the solution while [18] proposed that noise injection in the training sets labels enhances the learning ability of a network by helping training overcome local minimas.

3.3. Parallel and Distributed Deep Learning

As the size of neural networks and dataset sizes was getting larger, the need of parallelisation and distribution of processing among multiple machines araised. In order for distributed processing to be feasible, there is need for existence of operations that can be parallelised. Parallelisation of deep learning methods expands in the following three areas.

Data parallelism is the distribution of data among different processors when this is too large to fit in a single one or in order to achieve data processing acceleration.

Model parallelism is the distribution of modules of the model among different processors. In cases of very deep and wide neural networks for example, different layers can be utilised along different machines.

Pipelining is parallelisation by overlapping computations (i.e. between one layer and the next one) or by partitioning a NN and assigning different parts in different processors that are more efficient for each process.

For data parallelisation powerful machines are often used in order to achieve parallel computations. Matrix operations are one of the most usual operations in NNs and they are

quiet suitable for that. For this reason GPUs proved to achieve a huge speedup in the computational latency of NNs. In [19] there was an operation time speedup up to 72 for a neural network of one million parameters. Distributed memories and super scale computers are also used for similar purposes. Methods like [20] have achieved data parallelisation resulting in speedup up to 6 by scaling upon 7 processors for a three layer NN of 10512 parameters. Model parallelism approaches aim in distributing different part of a neural network in multiple processors so that these are running independently. Such methods reduce the computation time and probably memory (since the network is not stored in a single place) but it has high demands in communication among the machines for synchronisation. Pipelining aims in distributing the operations of different layers. The main issue of pipelining is the data dependencies that should be respected and the synchronisation. In [21] there a 1.9 and 3.3 times end-to-end speed-up with the use of pipeline parallelisation with respect to layers upon 2 and 4 GPGPUs.

Combination of model, data and pipelining parallelism have proved to be very successful as it can overcome the drawbacks of every approach. A hybrid parallelism example of AlexNet [22] achieved a speedup of up to 6.25 times for 8 GPUs over one by combining model and data parallelism while DistBelief [23] combined all three types of parallelism achieving a speedup up to 12× using 81 machines.

3.4. Layers Lock During Training

The training of a NN is achieved through iteration of procedure described in figure 3.1 bellow, which is composed of four tasks. (i) Every layer of the network, given an input, produces an output according its FP function. (ii) When step (i) is complete and the output of network is formed, the training error of the entire network is computed according to a loss function. (iii) The training error is propagated backwards into the network and every layer computes its partial error. (iv) Finally, layers proceed to error correction, using the gradient descent algorithm, in order to update their weights.

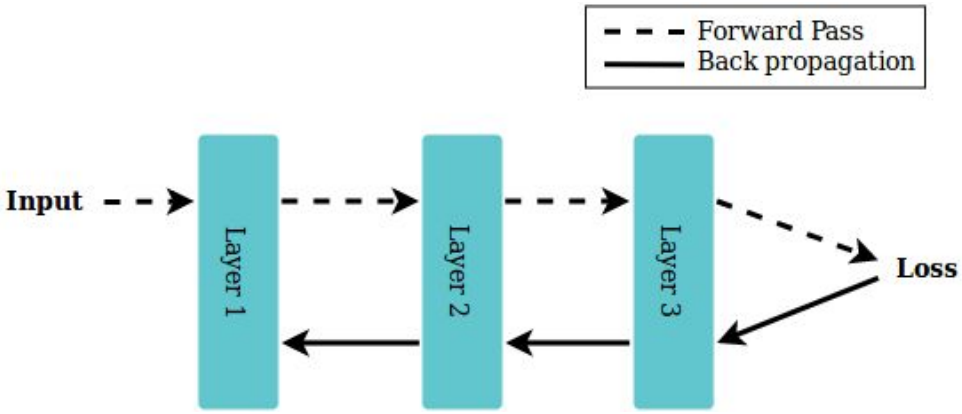


***Figure 3.1**: Training cycle of a neural network*

This process results in several types of locking. **Forward lock** - during step (i), each layer remain inactive until previous layers have produced an output, forming the input it needs in order to process it and produce its own output. **Backward lock** - during step (ii) and (iii), layers cannot compute their own partial error until the error has been produced, propagated and provided to them. **Update lock** - layers cannot perform error correction update until (i) to (iii) are complete. These problems described above, cause a major loss in both processing resources and time.

There have been several approaches aiming in the elimination of the different types of lock. Backward lock has been resolved by allowing loss to be broadcast directly to the layers like [24] and [25]. For resolving update lock, real-time recurrent learning was introduced in [26] or [27] but these methods require maintaining the full gradient of the current state with respect to the parameters which makes it inherently not scalable.

Decoupled neural interfaces (DNI) overcome both backward and update lock, performing error correction of layers using local information only. They were introduced in [28] and further discussed in [29], proposing the replacement of the backpropagation algorithm with function approximators that predict the error gradient of a layer *N+1* using output of layer *N* along with the training example labels. This information, is local to layer *N* and available right after this layer has performed a FP, and as a result, error correction update can start immediately, without waiting for FP of the rest of the layers, loss computation and backpropagation to take place. This makes it ideal for large scale and distributed NNs as it overcomes the major problems of such applications, multiple processors communication and data dependencies. Exploration of DNI model is the main subject of the current thesis and it will be thoroughly explained in chapter 4.3.

3.5. Deep learning on FPGAs

Most models for parallel and distributed NN computing like these presented in chapter 3.4 aim in the exploitation of powerful and expensive machines. This approach leads to high demand in processing and power resources which is not efficient when seeking solution for large scale applications.

In recent years Field-Programmable Gate Arrays (FPGAs) have come to a point of large processing power, enough to make them an interesting alternative for deep learning methods. First, the morphology of NNs suits in the large degree of pipelining and parallelism of the FPGAs. Although other processors, like GPUs have proved to be great solution for parallel computation, FPGAs are ideal for executing independent processes in parallel which gives freedom in exploring different speedup algorithms that do not strictly depend on parallelisation of similar computations. Second, FPGAs can achieve distribution and localisation of memory accesses which can save a lot of time during a NN training process. Finally, the significant high performance per watt makes them ideal for large scale and distributed systems. These have led in several applies of deep learning models for speeding up both training and inference process on FPGAs like [30], which took advantage of memory localisation of FPGA to improve performance and process parallelisation for network partitioning among multiple FPGAs and [31] that aim in increasing throughput using parallel operators and a coarse-grain pipeline on a FPGA. In addition there have bee applications on distributed NNs like [32], a model implemented on FPGA that reduced communication cost in exchange with increase of computational complexity which led to overall speed up due to advantage of FPGAs in computational strength, as well as [33] that used tile techniques, FIFO buffers, and pipelines to minimize memory transfer operations, and reuse of the computing units to implement the large-size NNs.

Despite the fact that FPGAs seem very efficient in NN applications considering performance, the hardware design approach along with the big compilation time makes them stiff for prototyping purposes. Nevertheless modern tool for FPGA programming such as Vivado HLS and OpenCL have reached decent levels of abstraction. This can lead to implementations with an accurately corresponding software functional simulation so that prototypes exploration and parameters tuning can be done in top level and flexible programming languages.

Chapter 4

Modelling

DNIs were originally designed in order to solve problems of NNs that are distributed along multiple machines like communication reduction and prevention of layers lock (chapter 3.4). In the current thesis we review the performance capability of training a neural network using synthetic error gradient, the essential component of a DNI, and we also discuss further applications using DNIs on how this it can contribute to the acceleration of the training process as well as to the increase of accuracy. At first we evaluate the performance of the DNI model. Then, we continue by investigating how layers update using synthetic gradient can be combined with the normal backpropagation update in order to achieve acceleration of error convergence during training process.

The assessment of the models investigated in this research is based on measuring both efficiency and computational complexity. The efficiency is determined as the learning ability and the representational strength (accuracy) by measuring the error levels during the learning process as well as the accuracy on test dataset after training is complete. The exploration of models behavior was achieved with the use of Python functional simulations. This offered flexibility on experimenting with modifications in the models architectures and parameters tuning in a high level programming language, as well as easy representation of the metrics with the use of Matplotlib, a Python 2D plotting library [34].

Time complexity analysis of the reviewed models was carried out by measurement of simulated operation time of the FPGA implementation through RTL simulation in Vivado HLS. Time analysis is presented in chapter 5.

4.1. Experiment Datasets

In order to define the efficiency of the models that we review in our research, we experimented with training and test of our model NN upon the two datasets of handwritten digits. The Optical Recognition of Handwritten Digits Dataset and the MNIST Dataset that are presented below.

4.1.1. UCI Optical Recognition of Handwritten Digits Dataset

Optical Recognition of Handwritten Digits Dataset [35] created by E. Alpaydin and C. Kaynak. It consists of 5620 8x8 pixels images of handwritten digits created by collecting 250 samples from 44 writers. Images were created using using a pressure sensitive table that produces 500x500 pixels images with pixel integer values between 0 and 500. The samples were then normalised in the range 0 to 100 which made the representation invariant to translations and scale distortions. Finally the samples were spatially resampled in 8 points using simple linear interpolation. This gives us a constant length feature vector of 64 integers.

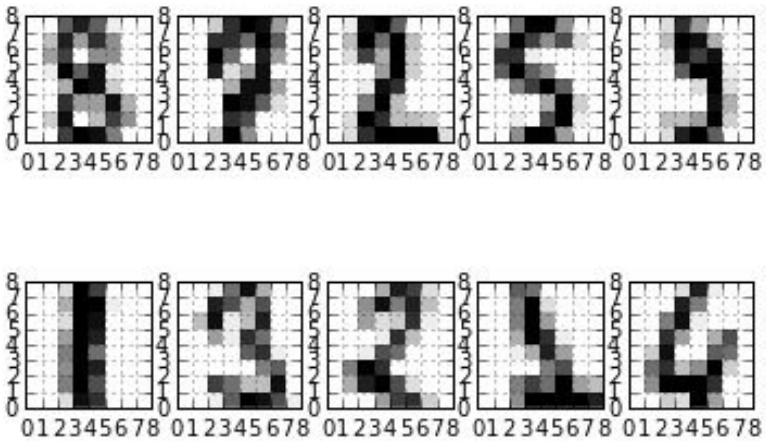


***Figure 4.1**: Samples of Optical Recognition of Handwritten Digits dataset*

For our experiments in Python, the acquisition of the dataset was done using the sklearn.datasets.load_digits tool from sklearn library [36]. This tool retrieves a part of the Optical Recognition of Handwritten Digits Dataset in the form of example data 1797x64 integer elements array and integer array of 1797 elements for the data labels. The labels array is then converted into One-Hot encoding array in order to be compatible with our NN architecture. We split the dataset randomly into 1258 samples for training set and 539 for test set with the use of sklearn train_test_split tool [37].

4.1.2. MNIST Handwritten Digits Dataset

MNIST a dataset of handwritten digits dataset [38] created by Yann LeCun, Corinna Cortes and Christopher J.C. Burges offered for experimenting on learning techniques and pattern recognition methods on real-world data. It is consists of 28x28 images of handwritten digits that are more realistic than digits dataset thus more challenging. The 70,000 examples dataset contains samples from approximately 250 writers. This dataset was created by combining two of NIST's databases: Special Database 1 and Special Database 3. Pixels are organized row-wise and have integer values between 0 and 255. 0 means background (white), 255 means foreground (black).

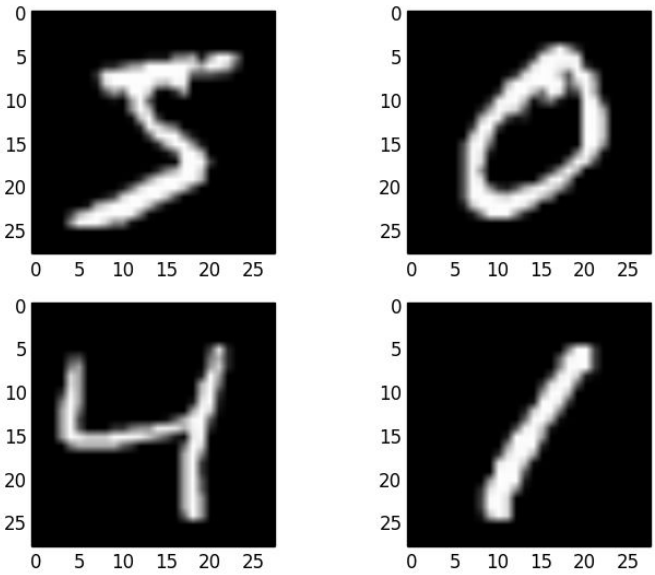


***Figure 4.2**: Samples of the MNIST dataset*

For our experiments in Python, the acquisition of the dataset was done using the sklearn.datasets.fetch_mldata tool from sklearn library [39]. This tool retrieves datasets from the online repository [40] in the form of 70000x784 integer elements array for the examples data and integer array of 70000 elements for the data labels. The labels array is then converted into One-Hot encoding array in order to be compatible with our NN architecture.. The integer labels array is then converted into One-Hot encoding in order to be compatible with our NN architecture. We split the dataset randomly into 49000 samples for training set and 21000 for test set with the use of sklearn train_test_split tool [38].

4.2. Initial Model V0

DNIs are designed to be able to be integrated in any kind of NN architecture. In this thesis, the experiments were carried out on a fully connected feedforward neural network that consists of three layers. This NN is designed to implement a handwritten images recognition classifier which uses images as an input, transformed into a feature vector with of integer values between 0 and 100. It is a supervised learning NN, so during training, every example that is given as an input, comes along with an one hot encoded label vector of ten elements that indicates the true class of the example. Training is performed in mini-batches of 10 examples each.

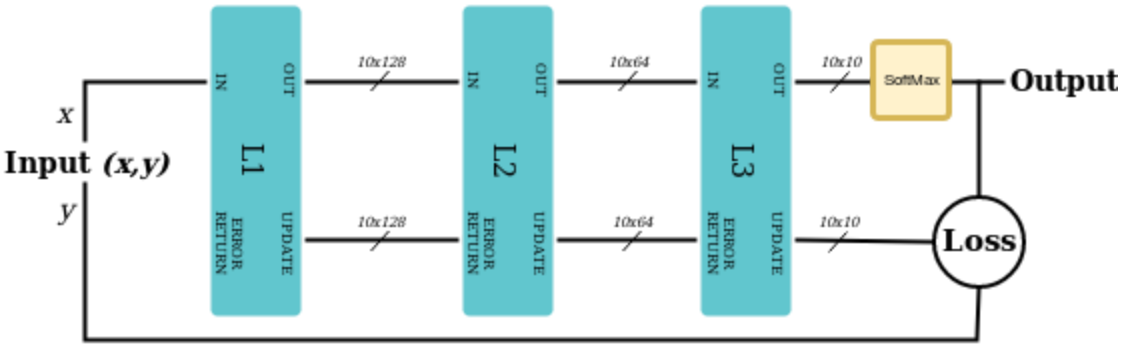


***Figure 4.3**: Experiment neural network*

We aim in training the NN above to produce an output of One Hot Encoding among ten classes. Softmax is a normalisation function used on the output. While the NN output is weights of unknown range for every one of the ten classes, softmax transforms the representation into the probability distribution over the classes. If vector z size of K is the output of an One Hot Encoding classifier, Softmax transforms each element of the output into the probability rate among the classes and it is defined as:

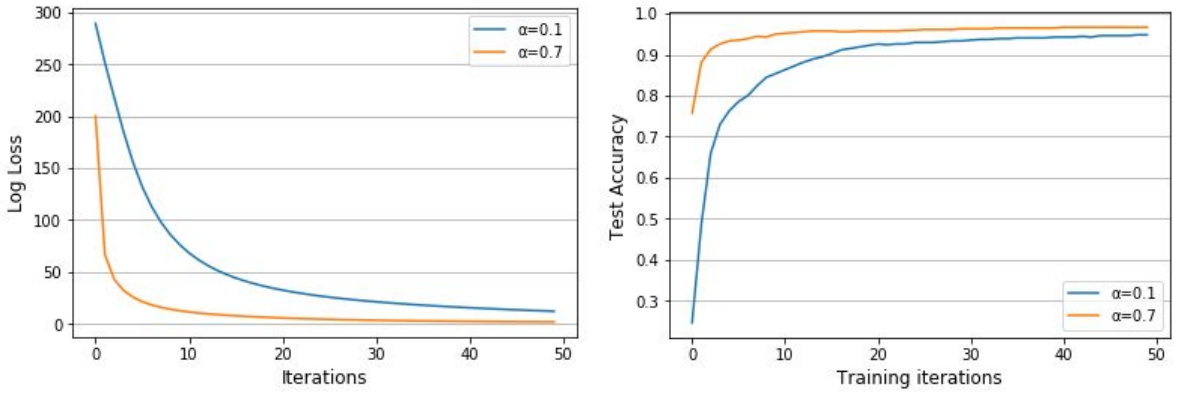
$$\sigma(z)_i = e^{z_j} / \sum_{k=1}^{K} e^{z_k} \quad \text{for} \quad j = 1,\ldots,K \qquad (19)$$

Loss function of the NN is the Logarithmic loss:

$$-\sum_{k=1}^{K} y_k \log(y'_k) \qquad (20)$$

where K is the number of classes, y is the true class and y' is the prediction of the NN.

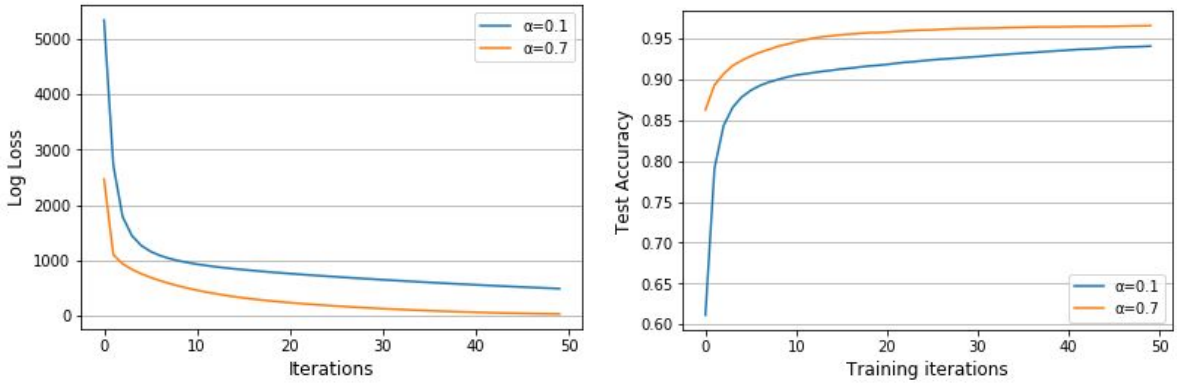
In the figures below we present the results from the performance experiments that we carried out on our initial model V0. Experiments were carried out using the two datasets that were reviewed in chapter 4.1 by keeping track of both training loss during training and accuracy upon the test set per training iterations.



(a) ***(b)***

Figure 4.4*: Performance on the UCI Dataset*

Training error (a) and test set accuracy (b) of initial model V0 on UCI Dataset for different learning rates.



(a) ***(b)***

Figure 4.5*: Performance on the* MNIST Dataset

Training error (a) and test set accuracy (b) of initial model V0 on MNIST Handwritten Digits Dataset for different learning rates.

The experiments that we show above were conducted with the use of two different layers learning rates, α=0.1 and α=0.7 in order to understand the difference in behaviour and improvement of enhancements that were applied In general. We consider these two values as the upper and lower bounds for our experiments case. 0.7 is the largest learning rate we managed to use without overshooting the solution and 0.1 is a relatively very small learning rate. In this ways we can speculate that our results respond to a general solution as well that results would be similar even in cases where a varying learning rate is used. This will be our baseline performance for comparison with the models we review in next two chapters.

4.3. Decoupled Neural Interfaces Model V1

DNI model was introduced in order to allow neural layer to perform error correction update using only local information, with no need for waiting the FP of entire NN, loss computation and backpropagation to complete. This is achieved with the use of SG error method, an approximation of the function implied by backpropagation. In order to update weights θ_i of module i in backpropagation, we compute the partial error of that weight using the following function,

$$\partial L / \partial \theta_i = f_{Bprop}((h_i, x_i, y_i, \theta_i),...) \partial h_i / \partial h_i / \partial \theta_i \quad (21)$$

where h are activations, x are inputs, y is supervision, and L is the overall loss to minimise. DNIs replaces this formula with an estimation of that, in order to use local information only.

$$\simeq {}'f_{Bprop}(h_i) h_i / \partial \theta_i \quad (22)$$

This leaves dependency only on h_i which achieves the decoupling of the layer as it does no longer have to remain locked until forward pass, loss estimation and backpropagation are complete.

Synthetic gradient (SG) is actual a function approximation of the backpropagation function which is implemented as a single linear layer NN that uses activations hi of layer N as an input in order to compute a prediction of the error gradient of the N+1 layer. The training of SG module is performed along with the training of the NN. When forward pass and backpropagation are complete, the error gradient of layer $N+1$ is available for layer N and SG uses it as supervision in order to compare it with the predicted output according a loss function. This loss is used by the SG module to apply gradient descent and update its weights. In this way the approximation of gradient error is becoming more accurate during the training process of the NN. DNI is a normal neural layer combined with a SG module.

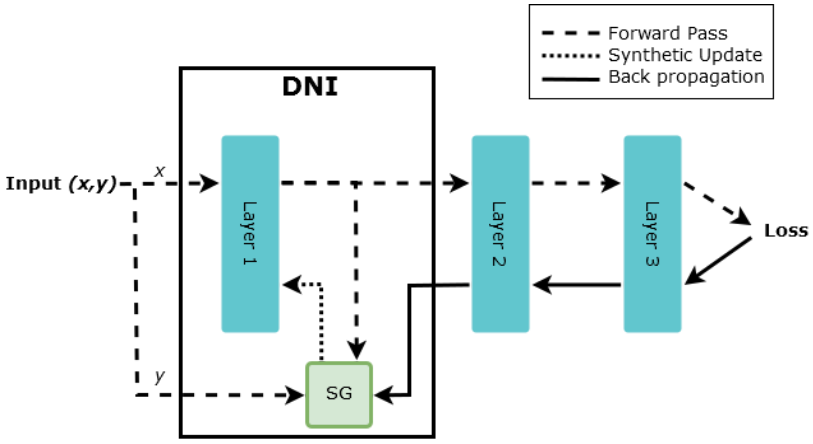


***Figure 4.6**: Decoupled neural interface*

Given an Input, the network propagates it by performing forward pass. After Layer 1 has complete a forward pass, the activations that were produced are fed in both SG and Layer 2 while SG also receives the training labels. Forward pass can continue normally for the rest of the network while SG computes the prediction of the error gradient of the next layer. After it is complete, Layer 1 uses this prediction to perform weights update. This is called synthetic update because a predicted (synthetic) gradient error is used rather than the actual one. With the completion of backpropagation, SG is provided with the true error gradient from Layer 2 in order to apply gradient descent and update its weights.

The first experimental model (V1) was created by we extending the baseline NN of chapter 4.2, replacing layers 1 and 2 with decoupled neural interfaces like these presented above, replacing the true gradient error with the synthetic gradient for the layers error correction update. We consider that there is no need to be extended Layer 3 in a DNI as late layers of a network are not subject to update lock as the time remaining inactive is small compared to the time the layer spends on its own operations.

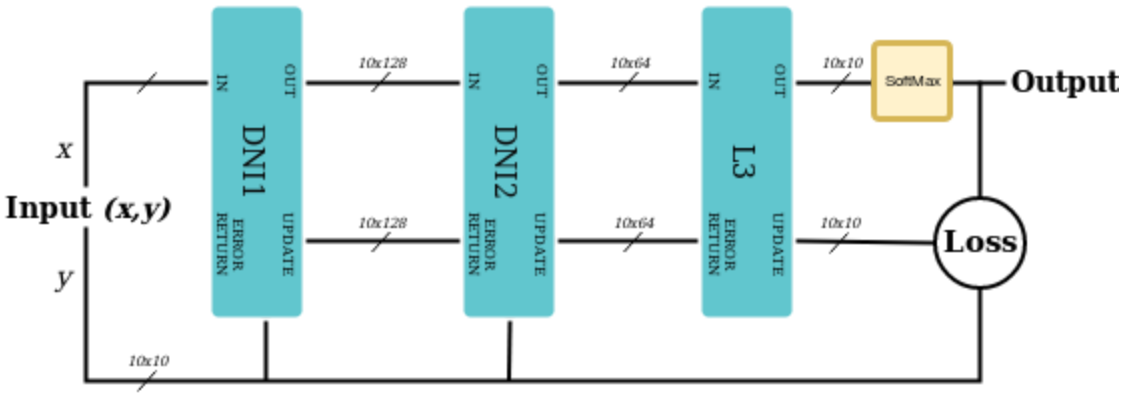
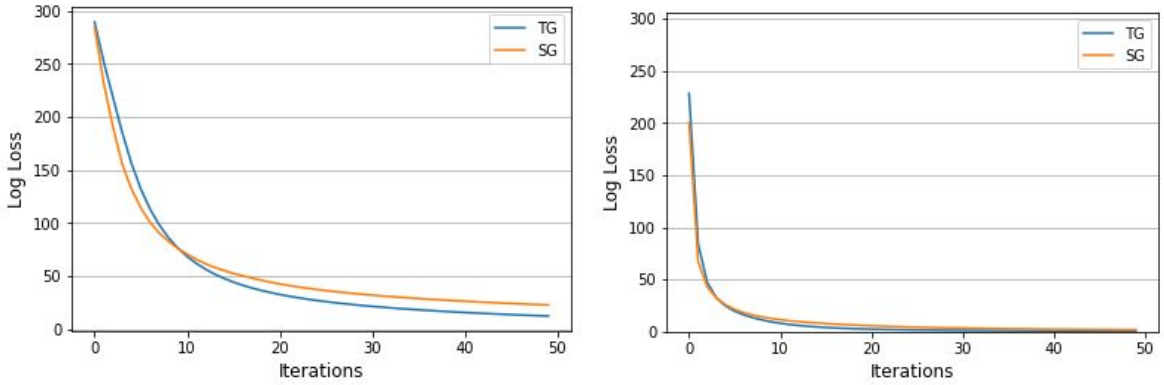


***Figure 4.7**: Model V1*

In order to evaluate that the replacement of error gradient with the synthetic gradient will not decrease the performance of the training process we compared the results of our new NN with these of the baseline architecture presented in chapter 4.2. The comparison is done using both UCI and MNIST datasets. The results are presented in the following two chapters.

4.3.1. Performance on the UCI Dataset



(a) ***(b)***

***Figure 4.8**: Training error*

Comparison of training error of true gradient (V0) and synthetic gradient (V1) for applying gradient descent with layers learning rate α=0.1 (a) and α=0.7 (b).

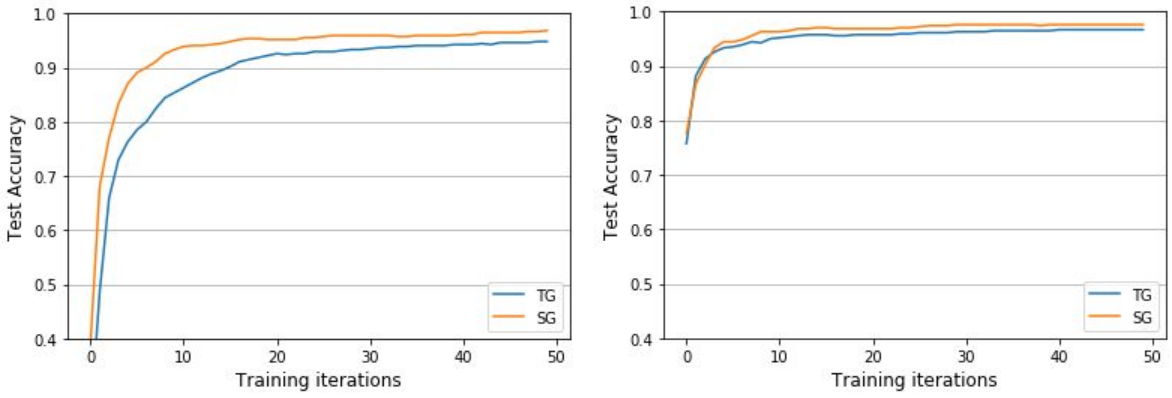
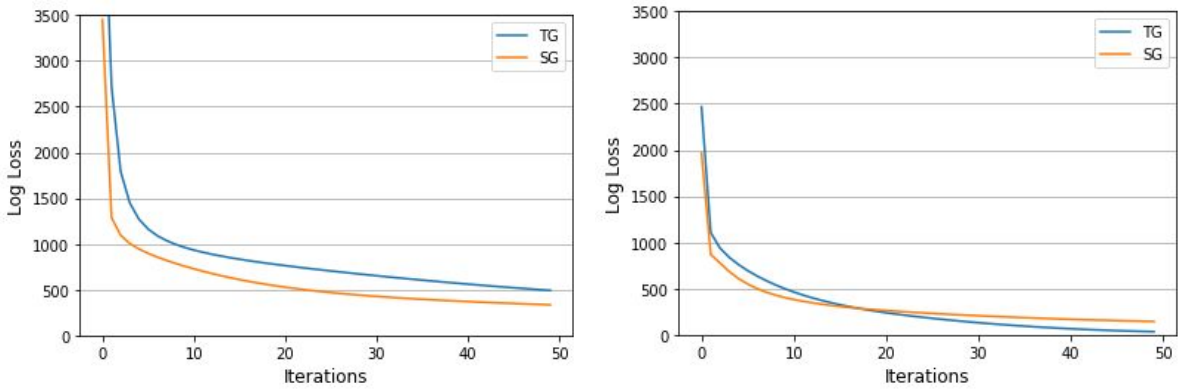


***Figure 4.9**: Test accuracy*

Comparison of accuracy on the test set of true gradient (V0) and synthetic gradient (V1) with layers learning rate α=0.1 (a) and α=0.7 (b).

Figure 4.8 shows that the error converges in a similar way but a little slower when using SG. In contrast, figure 4.9 we see that, even though the error convergence is worse, SG achieves better performance regarding accuracy, especially in case (a), for a small learning rate. This is caused due to the noise that is injected during the training by the estimation error of SG which results in a more general solution. In (b), SG also results in better accuracy but the difference is smaller because both SG and TG achieve high accuracy very fast, so SG is still appropriate for replacing TG but there is small further positive impact.

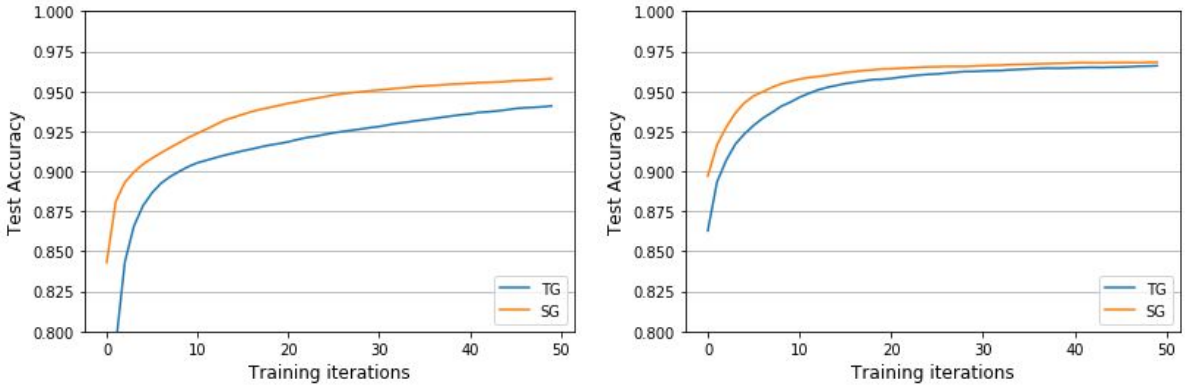
4.3.2. Performance on the MNIST Dataset



(a) ***(b)***

***Figure 4.10**: Training error*

Training error of true gradient (V0) and synthetic gradient (V1) for applying gradient descent with layers learning rate α=0.1 (a) and α=0.7 (b).



(a) ***(b)***

***Figure 4.11**: Test accuracy*

Accuracy on the test set of true gradient (V0) and synthetic gradient (V1) with layers learning rate α=0.1 (a) and α=0.7 (b).

In figures 4.10 and 4.11 we see that replacement of TG with SG has better impact for MNIST dataset than this appeared in UCI dataset that was presented in previous chapter. SG achieves both faster error convergence and better performance regarding accuracy. MNIST

consist of more complex data than UCI, with larger information per instance (MNIST images are 64x64 pixels, while UCI only 8x8). As a result, the noise injected has a bigger positive impact. The improvement is smaller for large learning rates (b) than for small learning rates (a), the same way like it happens for UCI dataset.

4.3.3. Results

The graphs of training error per training iterations *(Figure 4.8)* and (*Figure 4.10)* show that the replacement of TG update with SG update generally leads in similar error convergence. Our experiments show that the single layer of the SG module can effectively predict the gradient error of a hidden unit of a neural network. In addition, in the graphs of test accuracy per iteration *(Figure 4.9)* and *(Figure 4.11)* we see that the model of SG results in better performance on the test set. SG reaches the levels of accuracy of V0 in fewer iterations and it also achieves higher maximum accuracy even in cases where training error was larger for SG like *(Figure 4.9.a)* and *(Figure 4.11.b)*. This means that the fact that SG is an estimation of TG and as a result it is not identical, it adds a noise in the training procedure which leads to better generalisation of the found solution. As a result the replacement of TG with the SG has a positive impact in the performance of the neural network.

In the table below we present the speedup in training steps we achieve with the use of synthetic gradient instead of the true gradient, regarding the test set accuracy for the two different datasets we used for the experiments.

	UCI		MNIST	
Accuracy	α=0.1	α=0.7	α=0.1	α=0.7
0,9	1,33	1	2,2	3
0,91	1,26	1,1	2,44	2,3
0,92	1,4	1,2	2,53	2
0,93	1,37	1,25	2,52	1,9
0,94	1,39	1,33	2,6	1,7
0,95	1,42	2,12	2,8	1,76
0,96	1,44	3	2,92	1,72
0,97	1,45	2	3	1,7

***Table 4.1**: Training steps speedup of model V1*

4.4. Combining True and Synthetic Gradient Error Model V2

In this chapter we investigate how synthetic gradient error can be used in order to accelerate the training process. We propose combining SG with normal backpropagation aiming to increase the error convergence rate by performing the layers weights updates with the use of both true and predicted error gradients in every training iteration. For this reason we extend the DNI so that, when backpropagation is complete and the layer receives a gradient error from the next layer in order update the weight of SG module, layer also averages the true error gradient with the SG error gradient estimation in order to perform weight update. Now in every training pass (inference-loss-backpropagation) each layer updates its weights using both true and synthetic error gradient.

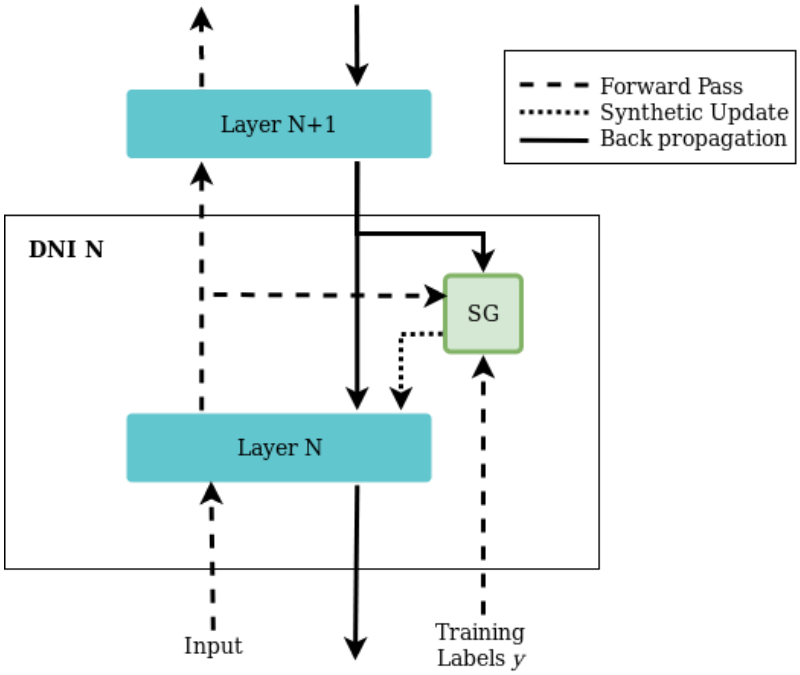


***Figure 4.12**: DNI combined with normal update*

1. Input is received from the network input of from layer N-1 output
2. Layer N performs forward pass
3. Output of layer N is available for layer N+1 and SG module
4. SG module starts prediction process in order to produce an estimation of layer N+1 error gradient
5. The forward pass of the network continues normally

6. When forward pass, error computation and backpropagation of the network is finished to the point of layer N+1, layer N receives the error gradient of layer N+1 along with an estimation of the error gradient of layer N+1 produced by the SG module
7. Layer N performs weights update using the average of the true and the predicted error gradient of layer N+1
8. SG also uses error gradient of layer N+1 to apply weights update

In the next two chapters, we compare the training process performance of our new model V2 with the performance of our baseline model explained in chapter 4.2 upon both UCI and MNIST datasets.

4.4.1. Performance on the UCI Dataset

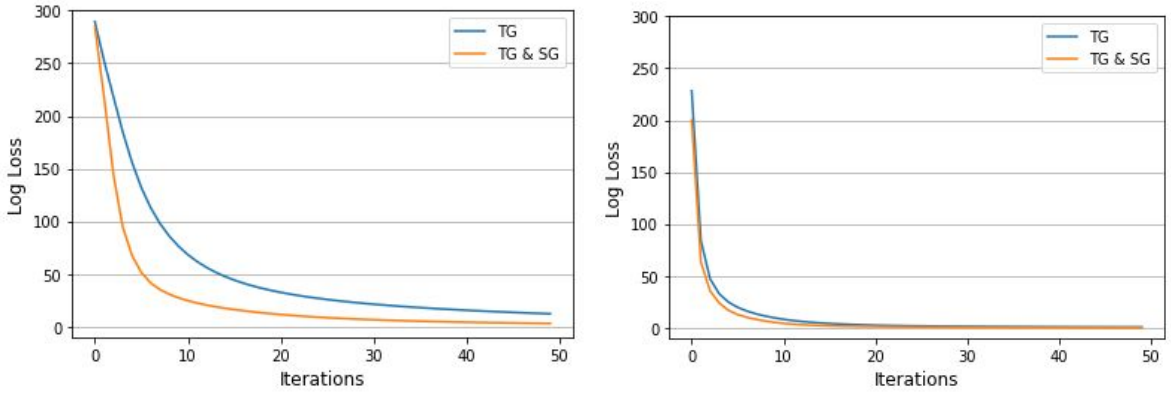
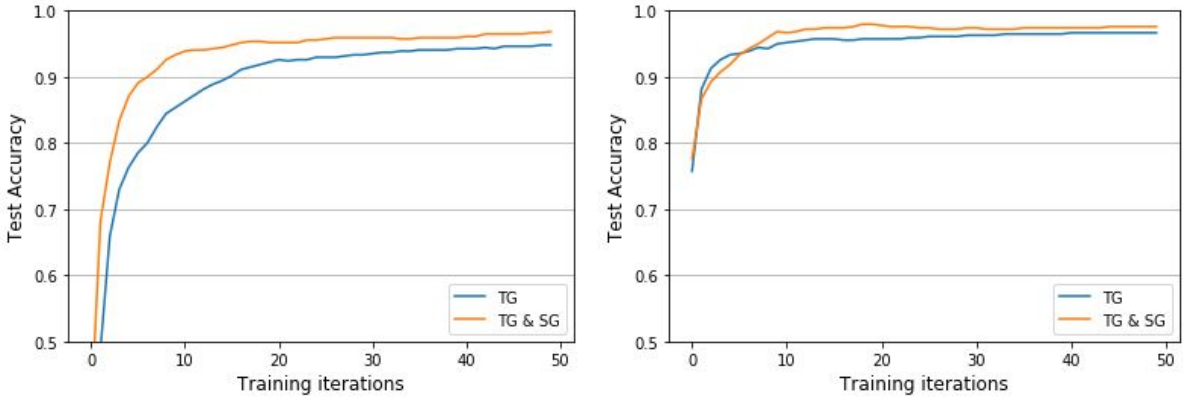


***Figure 4.13**: Training error*

Training error of true gradient (V0) and combination of true and synthetic gradient (V2) for applying gradient descent with layers learning rate α=0.1 (a) and α=0.7 (b).



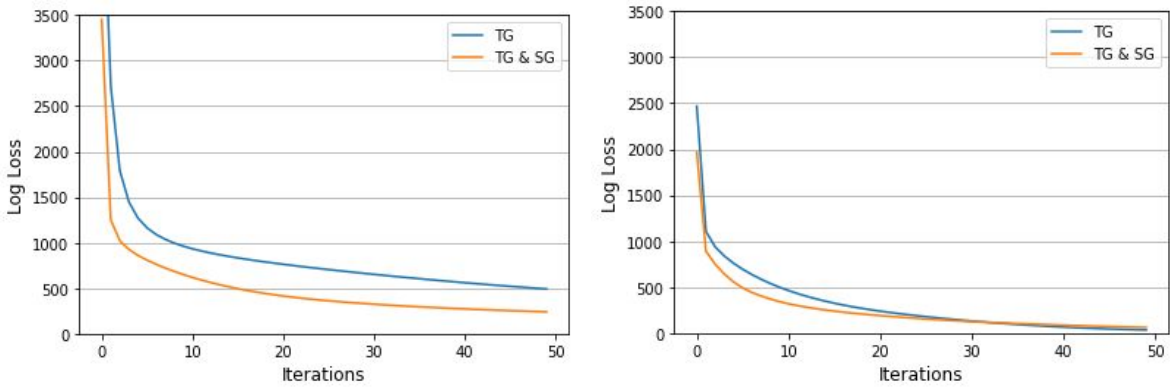
(a) ***(b)***

***Figure 4.14**: Test accuracy*

Accuracy on the test set of true gradient (V0) and combination of true and synthetic gradient (V2) with layers learning rate α=0.1 (a) and α=0.7 (b).

The experiments results above show that in case of small learning rate, the combination of true and synthetic gradient can achieve a significant acceleration in error convergence and it also results in higher accuracy. For large learning rates the positive effect is smaller but still, we can achieve higher accuracy when using the combination of true and synthetic gradient.

4.4.2. Performance on the MNIST Dataset



(a) ***(b)***

***Figure 4.15**: Training error*

Training error of true gradient (V0) and combination of true and synthetic gradient (V2) for applying gradient descent with layers learning rate α=0.1 (a) and α=0.7 (b).

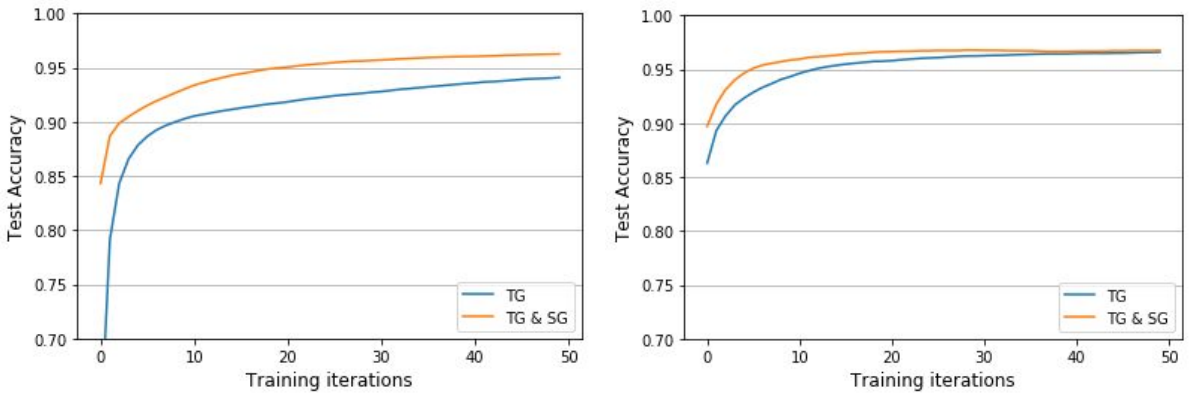


***Figure 4.16**: Test accuracy*

Accuracy on the test set of true gradient (V0) and combination of true and synthetic gradient (V2) with layers learning rate α=0.1 (a) and α=0.7 (b).

In the experiments on the MNIST dataset, we can see that the positive effect of synthetic gradient is bigger for a more complex dataset like this. In both cases of small and large learning rates we achieve acceleration of error convergence while the test set accuracy is also higher. This means that the increase of error corrections combined with the regularisation due to the SG estimation noise results in prevention of overfitting the training set and it also reaches the levels of accuracy of V0 in less training iterations and the overall accuracy is higher for both small and large learning rates cases.

4.4.3. Results

The experiments we presented in previous two chapters show that the combination of TG and SG can achieve a significant acceleration in the error convergence. The effect is bigger in cases where a small learning rate is used. In the training on the MNIST dataset *(Figure 4.15.a)* we see that SG needs down to ⅔ of the iterations that TG needs to converge to the same level. This does not have a negative effect in training fitting as it also achieves higher accuracy *(Figure 4.16.a)*. When large learning rate is used the error convergence is similar but the performance on the test set is still higher. We see that the noise of SG has the same positive effects as it had in model V1 that explained in the previous chapter, while the combination with the true gradient results in even faster error convergence.

In the table below we present the steps speedup we achieve with the use of model V2 compared to the initial model V0, regarding the test set accuracy for the two different datasets we used for the experiments.

	UCI		MNIST	
Accuracy	α=0.1	α=0.7	α=0.1	α=0.7
0,9	2,25	0,5	1,57	3
0,91	2,3	0,7	2,8	2
0,92	2.66	1,4	3.14	2,1
0,93	3,18	1,2	3.4	2,33
0,94	3,125	1,35	3,5	2,25
0,95	3,19	2	3,75	2,12
0,96	3,22	2,7	3,9	2,25
0,97	3,23	2	4	2,3

***Table 4.2**: Training steps speedup of model V2*

The increase of the performance is achieved for both of our experiment datasets and for every layers learning rate we used. This shows that this speedup can be achieved in general. In the following chapter we are showing the performance of our models when implemented in hardware in order to measure the operation time and have a clear comparison metric.

Chapter 5

Hardware Implementation

Both of the models discussed in this thesis, DNI (V1) and combination of SG with TG (V2) , aim in accelerating the training process. In the current thesis, we aim to accomplish that from an algorithmic approach, with parallelisation of the network forward pass and backpropagation with the layers synthetic updates. For this reason FPGA seems to be an ideal hardware to deploy our models and define the time performance.

Each model was implemented as a hardware architecture in Vivado_HLS and synthesised for a Xilinx Zynq UltraScale+ board (xczu9eg-ffvc900-1-i-es) [41]. This board is capable of fitting each one of our three entire example models (further analysis in chapter 5.2) with surplus resources remaining for further design optimisation which is ideal for distributed situations providing great power efficiency. Furthermore, this board contains a DDR4 memory that can cover the input bandwidth required for our models (further analysis in chapter 5.4). The operation time measurement was done through Vivado HLS RTL simulation in order to have a clear and accurate comparison between the different models.

In this chapter we present the hardware architectures designs of our models, the initialisation process and the hardware resources demand for each model as well as the design verification method we use. Then we continue by analysing and comparing the time complexity of the reviewed models and the result of the training process of each model, compared in the time domain.

5.1. Hardware Architectures

In this chapter we present the architecture designs for each of the models we reviewed in the previous chapter and we explain the functionality of the containing components. The deployment of the designs was done in Vivado High Level Synthesis platform using Vivado C++ hardware description language [42].

5.1.1. Model V0 Architecture

In this chapter we present the architecture of our initial model that was simulated in chapter 4.2. This is a mini batch, three layer fully connected normal backpropagation NN.

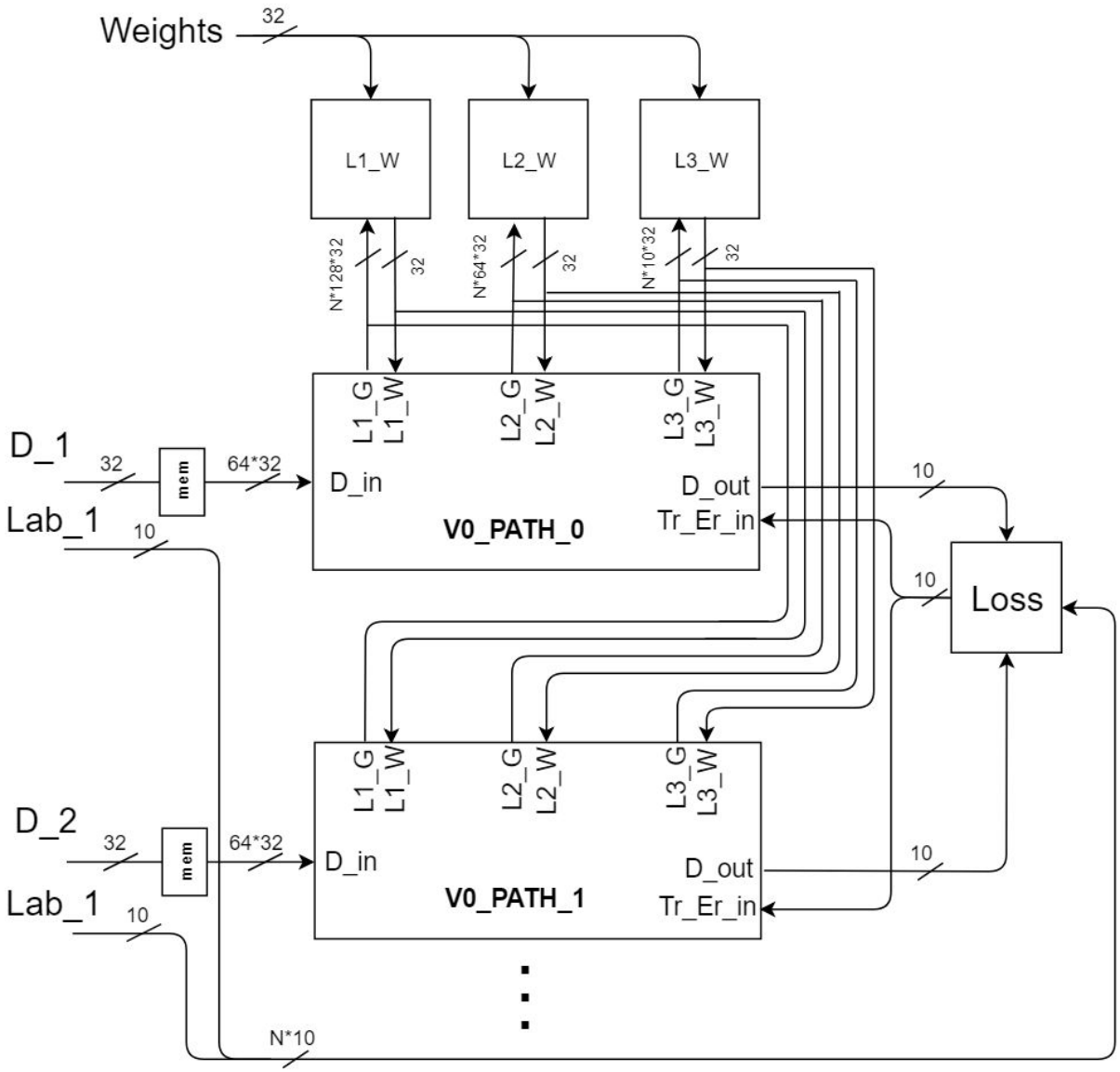


***Figure 5.1**: Top Level diagram of model V0*

As explained in chapter 3.1 a mini batch NN has a single set of weights for every layer while it proceeds several examples (batch) in parallel in every pass. In the end of training examples proces, an average loss is computed that is fed back to the network. For every layer, we compute the mean partial error of all the examples of the batch and the weights of the layer are updated according to the mean error. In the diagram above, we define ***N*** as the batch number which is 10 for our architectures. This can extend in bigger batches for different applications. Modules V0_PATH are the units that compute the forward pass and backpropagation for each example of the training batch. Loss is the unit that computes the loss upon all the training examples. L1_W to L3_W are the weights units, that perform error correction with respect to the average error of each example for every layer and perform the memory read/write. Every module will also be explained later in detail.

Ports of architecture V0:

Weights (32-bits):	Input for initialisation of the layer's weights.
D_1 to D_N (32-bits):	Input of training examples. The training examples are integer vectors of 64 elements. Each element of the vectors is read sequentially.
Lab_1 to Lab_N (10-bits):	Input of training labels.

5.1.1.1. V0_PATH

V0_Path_1 to V0_PATH_N are the units compute the forward pass and backpropagation for each one of the N examples of the training batch.These units operate synchronously in parallel using the layers weights values as an input. Weights are broadcasted by the L_W units and V0_PATH units read them simultaneously. V0_PATH implements the NN configuration of our model (number dimensions of layers).

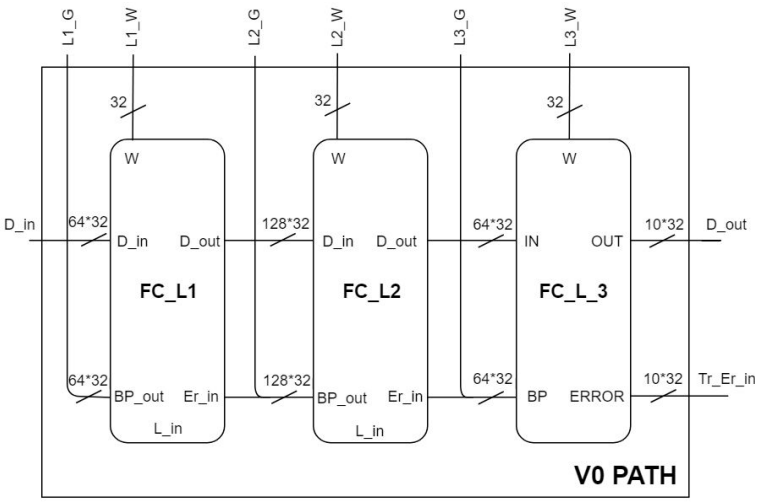


***Figure 5.2**: PATH of model V0*

Ports of V0_PATH:

D_in (64 * 32-bits):	Input of training examples.
D_out (10-bits):	Classification result output of the network.
L1_G (64 * 32-bits):	Error gradient of layer 1 output for the layer update.
L1_W (32-bits):	Layer 1 weights input for the forward pass.
L2_G (128-bits):	Error gradient of layer 1 output for the layer update.
L2_W (32-bits):	Layer 2 weights input the forward pass.
L3_G (64 * 32-bits):	Error gradient of layer 1 output for the layer update.
L3_W (32-bits):	Layer 3 weights input the forward pass.
Tr_Er_In (10*32-bits)	Training error input for backpropagation.

The FC_L units perform the basic fully connected layer operations as explained in chapter 2.3.1. Each FC_L unit implements the forward pass (chapter 2.3.1, equation 8) according to its input D_in, layer's weights W and passes the result in its output D_out. The backpropagation (chapter 2.3.4, equation 13) is implemented according to the input Er_in and the result is passed in the output BP.

5.1.1.2. Weights Units

The Weights units L1_W, to L3_W handle the broadcast of weights and biases to the layers and also perform the error correction update.

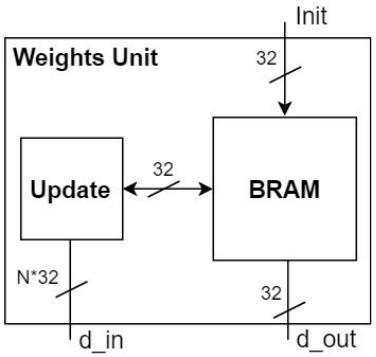


***Figure 5.3**: Weight units of model V0*

Ports of Weights Unit:

Init (32-bits):	Input for weights initialisation.
d_in (N * 32-bits):	Input of layer error gradient of each batch example for weights update.
d_out (32-bits):	Layer's weights broadcasting output.

The Update module collects the error gradient of the layer that was produced from the backpropagation for each of the *N* examples of the training batch. The average error gradient is computed and then the weights update is performed as explained in chapter 2.3.4. BRAM is the memory where the values of weights of the layer are stored. Size of BRAM is equal to the number of parameters required for the layer.

5.1.1.3. Loss unit

Loss unit collects the output of every path of the network in order to compare it with training labels and produce the training error. The training error is computed according to the network's loss function which was presented in chapter 4.2. The computation of the error for each of the *N* training batch examples is independent and operates in parallel.

5.1.1.4. Mem units

Mem units are buffers that collect the elements of every training example that are read sequentially from the D ports. When the entire example vector is fetched, it is fed into the PATH unit for the forward pass to begin.

5.1.2.Model V1 Architecture

In this chapter we present the architecture of the DNI model V1 that was simulated in chapter 4.3. This is a NN similar with that of model V0 where we have replaced layer 1 and 2 with decoupled neural interfaces.

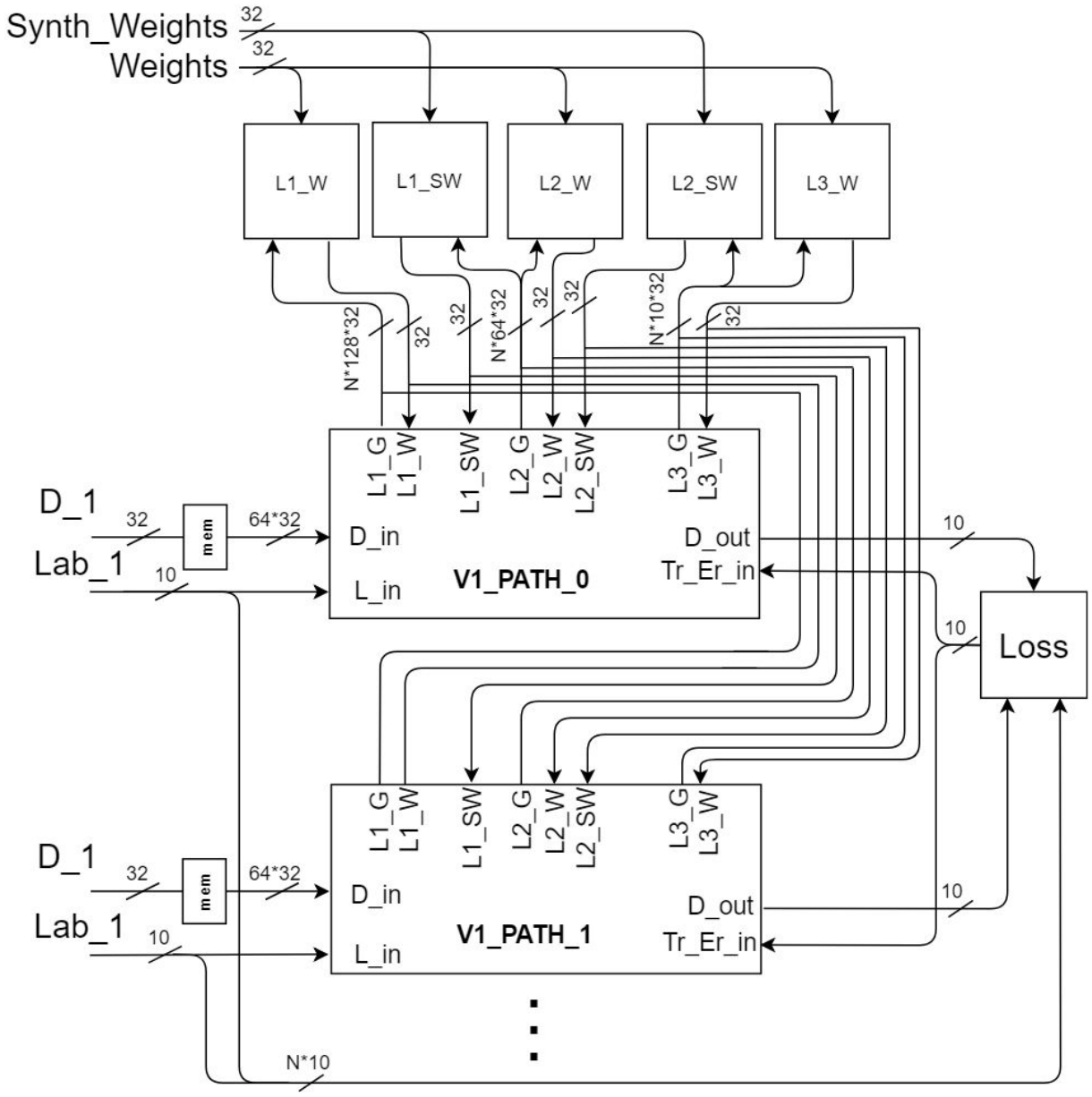


***Figure 5.4**: Top level diagram of model V1*

5.1.2.1. PATH_V1 Unit

As explained in chapter 4.3, model V1 is an extension of V0 by the replacement of neural layers 1 and 2 with decoupled neural interfaces DNI 1 and DNI 2. V1_PATH unit contains an FC_L unit like this explained in chapter 5.1.1.1 for layer 3 of the network and two DNI units for layers 1 and 2 that will be described in the next chapter.

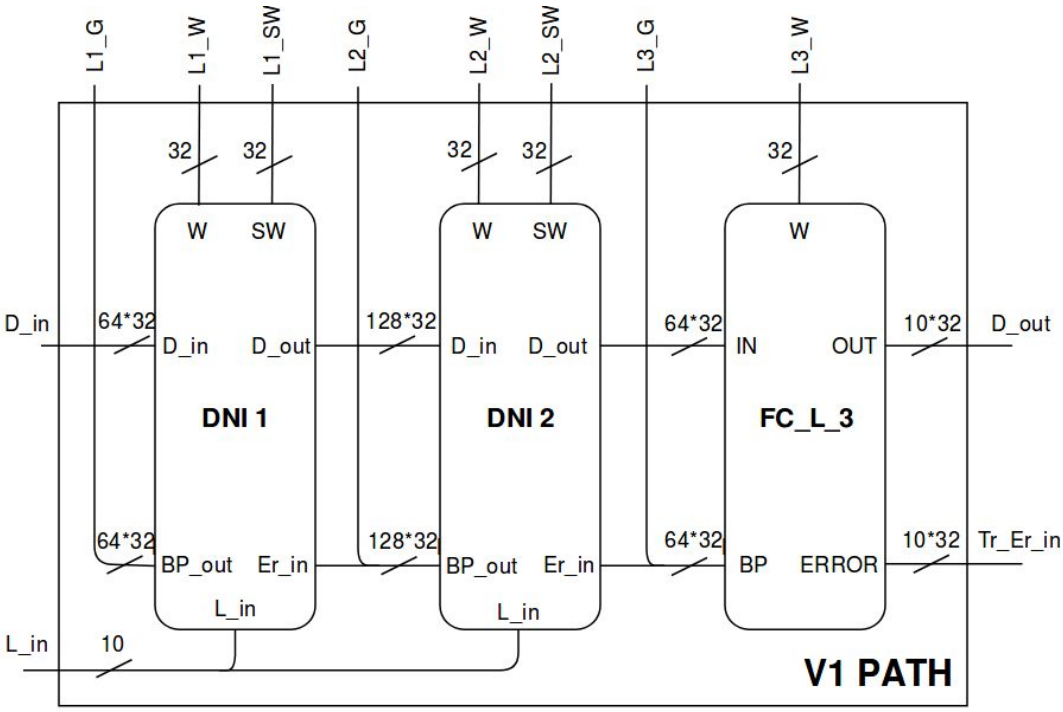


***Figure 5.5**: PAth unit of model V1*

Ports of V1_PATH:

D_in (64 * 32-bits):	Input of training examples.
L_in (10-bits):	Input of the training labels.
D_out (10-bits):	Classification result output of the network.
Tr_Er_In (10*32-bits)	Training error input for backpropagation.

L1_G (64 * 32-bits):	Error gradient of layer 1 output for the layer update.
L1_W (32-bits):	Layer 1 weights input for the forward pass.
L2_G (128-bits):	Error gradient of layer 1 output for the layer update.
L2_W (32-bits):	Layer 2 weights input the forward pass.

L3_G (64 * 32-bits): Error gradient of layer 1 output for the layer update.
L3_W (32-bits): Layer 3 weights input the forward pass.

L1_SW (32-bits): DNI 1 synthetic weights input for the computation of synthetic gradient.
L2_SW (32-bits): DNI 2 synthetic weights input for the computation of synthetic Gradient.

5.1.2.2. DNI Unit

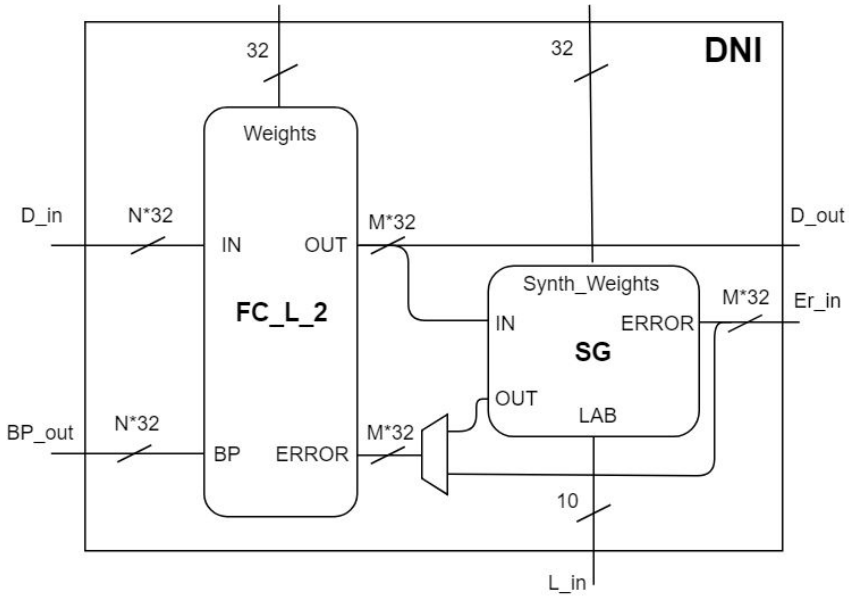


***Figure 5.6**: DNI unit*

5.1.2.3. Common Components with Model V0 Architecture

The architecture of model V1 is an extension of the architecture of model V0, thus there are several components that are common. Weights units L1_W to L3_W are identical with these of architecture of model V0 and they were explained in chapter 5.1.1.2. Loss function unit is also the same for this architecture and it is explained in chapter 5.1.1.3.

The units L1_SW and L2_SW are the units where weights for the SG modules are stored. They are weight units same like these presented in chapter 5.1.1.2 and they are used in order to store and update the parameters that the Synthetic Gradient unit uses. Synthetic gradient unit is a typical fully connected neural layer, similar to the FC_L units. It also operates in mini batches and as a result there is a single weight unit for the N PATH_V1 units that contain two SG modules each, for layer 1 an 2.

5.1.3 .Model V2 Architecture

The architecture of model V2 is identical with the architecture of V1. The difference applies in the control of the units and the sequence of operations. We can understand that difference only through the timing simulation which is presented and explained in chapter 5.3.3.

5.2. FPGA Resources

In the following tables we present the comparison of hardware resources for deploying each of the tested models on the FPGA. The models were deployed according to the dimensions of both UCI and MNIST datasets. This resources values are provided through the Vivado HLS Synthesis and the are referred to the percentage of utilisation of the target FPGA.

The UCI dataset consists of 8x8 pixels images, therefore the input of the NN is a vector of 64 elements while MNIST dataset which consists of 28x28 pixels runs on a NN of 784 elements input. The rest of the network's dimensions remain the same for both experiments.

Model	LUT (%)	FF (%)	DSP (%)	BRAM (%)
V0	14	5	8	12
V1	15	5	8	17
V2	17	6	9	20

(a)

Model	LUT (%)	FF (%)	DSP (%)	BRAM (%)
V0	14	5	8	30
V1	15	5	8	37
V2	17	6	9	48

(b)

Table 5.1*: FPGA resources for UCI (a) and MNIST (b) datasets, three layers FCNN.*

LUT: LookUp Table
FF: Flip Flop
DSP: Digital Signal Processor
BRAM: Block RAM.

V0: NN presented in chapter 4.1
V1: NN presented in chapter 4.3
V2: NN presented in chapter 4.4

5.3. Design verification

Modern FPGA designing tools like Vivado HLS have reached a significant level of abstraction. This gives us the ability to create hardware designing source code along with software simulations with accurate corresponding in components and functionality. This provides flexibility in exploring models and modifications in the software that can be easily configured on the hardware implementations as well as verifying behaviour of hardware architectures according to the software modelling.

In this thesis the modelling and the experiments of our different architectures functionality was implemented in Python. The architectures hardware deploy was implemented in Vivado HLS coding in Vivado C++. With the use of Vivado HLS c-simulation [43] we initialised our models with the known parameters values. Then we simulated the training process with known training examples while monitoring the output of each layer along with the training error. The same procedure was applied in Python functional simulations in order to examine the match of each internal parameter in every training step between the two runs. In this way we verified that both hardware design implementation and software functional simulation of every model discussed, have identical behaviour.

5.4. Timing Analysis

The timing analysis of our models was achieved through Vivado HLS RTL simulation of training operations of our the NNs architectures presented in the previous chapter. In this way we could define the interval of accepting a new input during training for every one of the three different models. The time length is defined in clock cycles that were acquired from the RTL simulation.

In order to understand the sequence of the actions that take place during the training pass we present the timing sequence of the operations of the discrete modules. This operations correspond to the discrete steps that take place during the training pass of a NN. We consider that the training pass begins from the point that an input has already been stored into the mem units described in chapter 5.1.1.4. The time it takes for an input to be fetched is same for all three architectures and it will be discussed in chapter 5.4.

Timing diagrams use the following annotation:

Operations:

L1, L2, L3:	Layer 1, 2 and 3 of the NN.
FW:	Layer forward pass. It is the function between ports D_in, W and D_out of the FC_L unit 5.1.1.1.
Loss:	Computation of the training loss. It is the operation of Loss unit 5.1.1.3.
BP:	Layer backpropagation. It is the function between ports Er_in and BP of the FC_L unit 5.1.1.1.
Up:	Layer weights and biases update. It is the operation of Weights unit 5.1.1.2.
SG:	Production of synthetic gradient error. It is the operation of SG unit
SG Up:	Update of the synthetic gradient weights. It is the operation of Synthetic Weights unit.

Colours:

Green:	Operations that use information produced from the forward pass and normal backpropagation.
Yellow:	Operations that use information produced by the SG modules.
Red:	Operations that use combination of backpropagation and SG modules Information.

Note: *Timing diagrams clock cycles axis is not linear for presentation reasons.*

The new input interval time is defined by the number of cycles that first layer starts the forward pass operation for the second time. This is the point when the NN has completed a full

training cycle. This is the interval we use in order to compare the time complexity of the difference models. The fetch of the input data would need 64 clock cycle. In the following chapters we are about to show that the training pass needs 10^4 range of clock cycles. For this reason, during the comparison of time complexity of the different architectures we assume that time to fetch the input is equal to zero.

5.4.1 Model V0

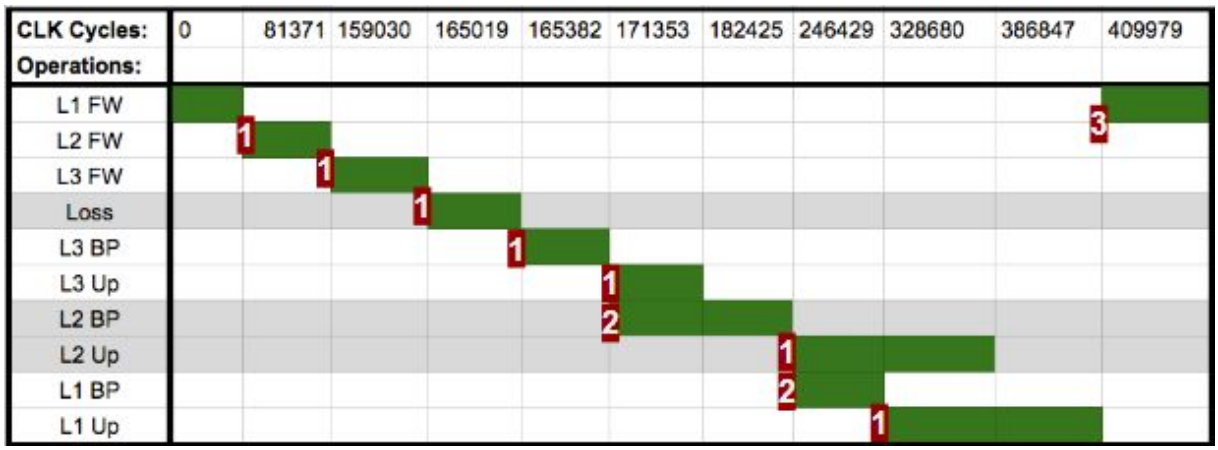


***Figure 5.8**: Timing diagram of model V0 architecture*

In the timing diagram of our initial architecture V0 we see that most of the operations of forward pass and normal backpropagation happen sequentially (points 1). The only operations that can happen in parallel are the backpropagation of a layer N with the weights update of a layer N-1 (points 2). The new input interval of the architecture is in point 3 and defined by the end of weights update of layer 1 in is 409979 clock cycles.

Diagram analysis:

[0]

L1 FW - Layer 1 forward starts.

[81371]

Result of layer 1 forward is available and it is provided in L2 FW.

L2 FW - Layer 2 forward starts.

[159030]

Result of layer 2 forward is available and it is provided in L3 FW.

L3 FW - Layer 3 forward starts.

[165019]

Result of layer 3 forward is available and it is provided in Loss.

Loss - Calculation of training error starts.

[165382]

The training error is available and it is provided to L3 BP for the backpropagation to start.

L3 BP - Layer 3 backpropagation starts.

[171353]

Result of layer 3 backpropagation is available and it is provided in L2 BP for the backpropagation to continue and in L3 Up in order to perform error correction update.

L3 Up - Layer 3 error correction update starts.

L2 BP - Layer 2 backpropagation using the true error gradient starts.

[246429]

Layer 2 true error is available and it is provided in L1 BP for the backpropagation to continue and in L2 Up in order to perform error correction update.

L1 BP - Layer 1 backpropagation using the true error gradient starts.

L2 Up - Layer 2 error correction update using the true error starts.

[328680]

Layer 1 error is available and it is provided in L1 Up in order to perform error correction update.

L1 Up - Layer 1 correction update starts.

[409979]

Layer 1 update correction is completed. The system is ready to accept new input.

5.4.2. Model V1

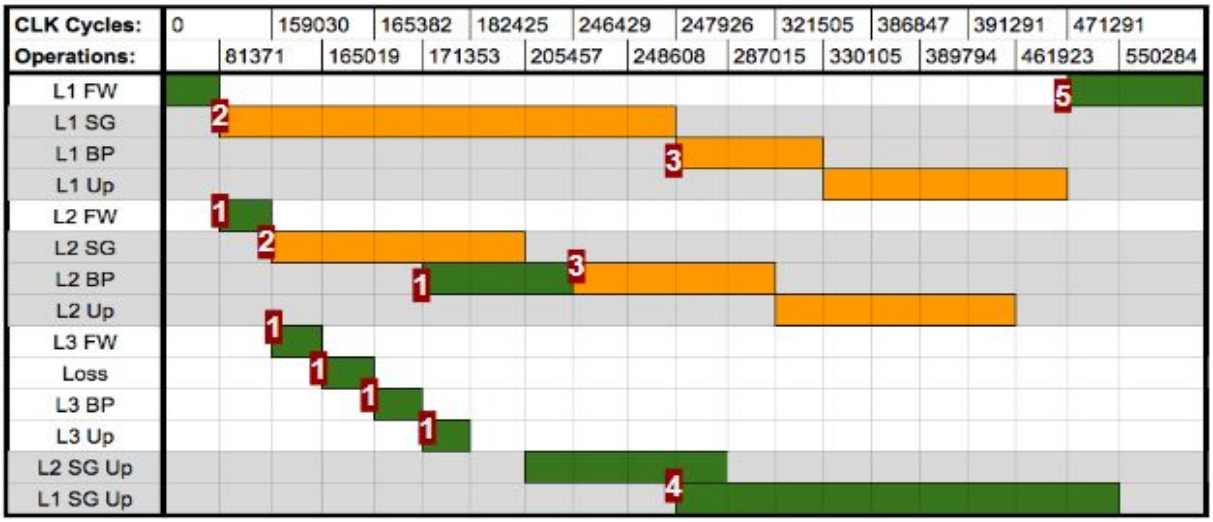


***Figure 5.9**: Timing diagram of model V1 architecture*

In the timing diagram of model V1 we can see that operations of synthetic gradient can be executed in parallel (points 2) but actually take much longer than the operations of backpropagation. This main delay occurs from the SG module of layer 1, which is actually a 138 to 128 fully connected layer. This means that L1 SG is the widest layer of our model, thus the forward pass and error correction update of layer 1 SG (point 4) are now, the two most time consuming operations of our architecture while the operations of forward pass and backpropagation are not affected (points 1). In point 3 we see that the backpropagation for error correction using the synthetic gradient occurs later than it would occur using the true error. As a result, the new input interval (point 5) of architecture V1 is 550284 clock cycles, 1,34 times longer than this of initial model architecture V0. Considering the speedup results of table 4.1, DNI could probably not achieve acceleration in a NN of such scale. A precise comparison of training time according to accuracy is presented in chapter 5.6

Diagram analysis:

[0]

L1 FW - Layer 1 forward starts.

[81371]

Result of layer 1 forward is available and it is provided in L1 SG and L2 FW.

L1 SG - Layer 1 synthetic gradient calculation starts.

L2 FW - Layer 2 forward starts.

[159030]

Result of layer 2 forward is available and it is provided in L2 SG and L3 FW.

L2 SG - Layer 2 synthetic gradient calculation starts.

L3 FW - Layer 3 forward starts.

[165019]

Result of layer 3 forward is available for Loss.

Loss - Calculation of training error starts.

[165382]

The training error is available and it is provided in L3 BP for the backpropagation to start.

L3 BP - Layer 3 backpropagation starts.

[171353]

Result of layer 3 backpropagation is available and it is provided in L2 BP for the backpropagation to continue and in L3 Up in order to perform error correction update. In addition, it is provided in L2 SG in order to perform error correction update of the SG module.

L3 Up - Layer 3 error correction update starts.

L2 BP - Layer 2 backpropagation using the true error gradient starts.

L2 SG - Layer 2 SG module is not available because it still produces the layer 2 synthetic error gradient. As a result error correction update cannot start.

[205457]

Layer 2 synthetic error gradient is available and it is provided in L2 BP in order to produce the layer synthetic error.
L2 SG Up - Layer 2 SG module is available and SG error correction update starts.
L2 BP - Layer 2 is not available because it still performs backpropagation using the true error gradient. L2 BP using the synthetic error gradient cannot start.

[246429]

Backpropagation of layer 2 using the true error gradient is completed and layer 2 is available.
L2 BP - Backpropagation using the synthetic error gradient starts in order to produce the layer synthetic error.

[321505]

Layer 2 synthetic error is available and it is provided in layer 2 in order to perform error correction update.
L2 Up - Layer 2 error correction update using the synthetic error starts.

[247926]

Layer 1 synthetic error gradient is available and it is provided in layer 1 in order to perform backpropagation.
L1 SG Up - L1 SG module is available and error correction update starts using the result of L2 BP which is already available.
L1 BP - Layer 1 backpropagation using the synthetic error gradient starts.

[330105]

Layer 1 synthetic error is available and it is provided in L1 Up in order to perform error correction update.
L1 Up - Layer 1 error correction update using the synthetic error starts.

[471291]

Layer 1 error correction update is completed. The system is ready to accept new input.

5.4.3. Model V2

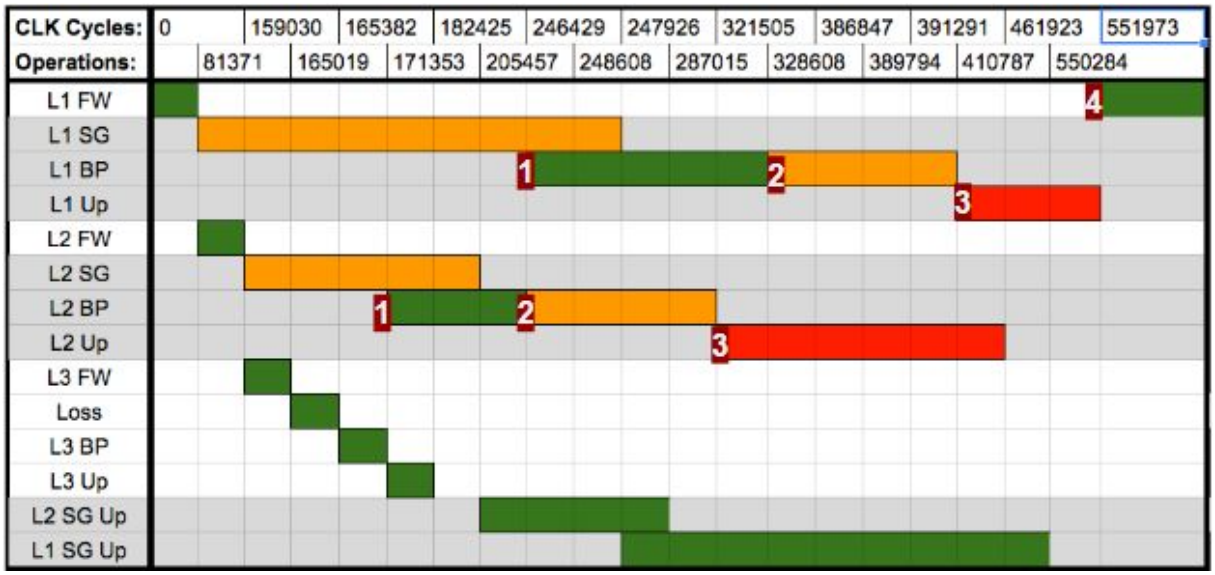


***Figure 5.10**: Timing diagram of model V2 architecture*

In the timing diagram of model V2 we realise that all operations of backpropagation can actually happen in parallel with the operations of DNIs with no significant additional time overhead. In points 1, we apply backpropagation using true error gradient before synthetic error gradient is ready and then, backpropagation using the synthetic error gradient (points 2). In addition, normal layers update are less time consuming than synthetic gradient operations, therefore the time complexity of normal layes update is completely hidden. Therefore in point 3 we apply weights update using both errors at the time point we did in architecture V1. As a result, this model has the same new input interval with DNIs NN V1. A major advantage though is that, in this model, layer 1 and 2 are updated using both information of backpropagation and synthetic gradient modules.

Diagram analysis:

[0]

L1 FW - Layer 1 forward starts.

[81371]

Result of layer 1 forward is available and it is provided in L1 SG and L2 FW.

L1 SG - Layer 1 synthetic gradient calculation starts.

L2 FW - Layer 2 forward starts.

[159030]

Result of layer 2 forward is available and it is provided in L2 SG and L3 FW.

L2 SG - Layer 2 synthetic gradient calculation starts.
L3 FW - Layer 3 forward starts.
[165019]
Result of layer 3 forward is available and it is provided in Loss.
Loss - Calculation of training error starts.
[165382]
The training error is available and it is provided in L3 BP for backpropagation to start.
L3 BP - Layer 3 backpropagation starts.
[171353]
Result of layer 3 backpropagation is available and it is provided in L2 BP for the backpropagation to continue and in L3 Up in order to perform error correction update. In addition, it is provided in L2 SG in order to perform error correction update of the SG module.
L3 Up - Layer 3 error correction update starts.
L2 BP - Layer 2 backpropagation using the true error gradient starts.
L2 SG - Layer 2 SG module is still producing the layer synthetic error gradient. As a result error correction update cannot start.
[205457]
Layer 2 synthetic error is available and it is provided in L2 BP in order to produce the layer synthetic error.
L2 SG Up - Layer 2 SG module is available and SG error correction update starts.
L2 BP - Layer 2 is not available because it is still performing backpropagation using the true error gradient. L2 BP using the synthetic error gradient cannot start.
[246429]
Layer 2 true error is available and it is provided in L1 BP for the backpropagation to continue and in layer 1 SG module in order to perform error correction update.
L1 BP - Layer 1 backpropagation using the true error gradient starts.
L1 SG Up - Layer 1 SG module is not available to perform error correction update using the true error gradient because it still produces the synthetic error.
L2 BP - Layer 2 is available and backpropagation using the synthetic error gradient starts in order to produce the layer synthetic error.
[247926]
Layer 1 synthetic error gradient is available and it is provided in layer 1 in order to perform backpropagation.
L1 BP - Layer 1 is not available because it still performs backpropagation using the true error gradient.
L1 SG Up - L1 SG module is available and error correction update starts using the result of L2 BP which is already available.
[328608]
Result of layer 1 backpropagation using the true error gradient is available.
L1 BP - Layer 1 is available and backpropagation using the synthetic error gradient starts.

[410787]
Result of layer 1 backpropagation using the synthetic error gradient is available. Both true and synthetic errors ara now available for layer 1 in order to perform error correction update.
L1 Up - Layer 1 error correction update using the average of true and synthetic error starts.
[321505]
Layer 2 synthetic error is available and it is provided in layer 2. Both true and synthetic errors are now available for layer 2 in order to perform error correction update.
L2 Up - Layer 2 error correction update using the average of true and synthetic error starts.
[550284]
Layer 1 error correction update is completed. The system is ready to accept new input.

5.4.5. Comparison of time complexity

In the following figure we present the timing diagram of the new input interval, as defined for each model in the previous chapters, in order to compare them.

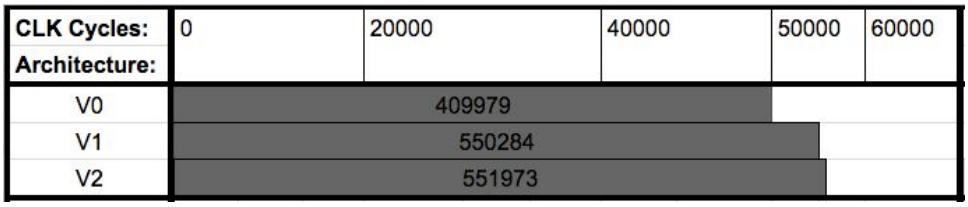


***Figure 5.11**: Timing comparison of models V0, V1 and V2*

5.5. Data Input Bandwidth

The implementation of our architectures was only conducted in simulation using the Xilinx Vivado HLS RTL Simulation. In this simulation, it is assumed that data we feed as an input to the FPGA are available without any latency. However, we have to consider the behaviour in a real world application where input data are provided in the FPGA by the DDR memory and ensure this will not cause a bottleneck in the training process.

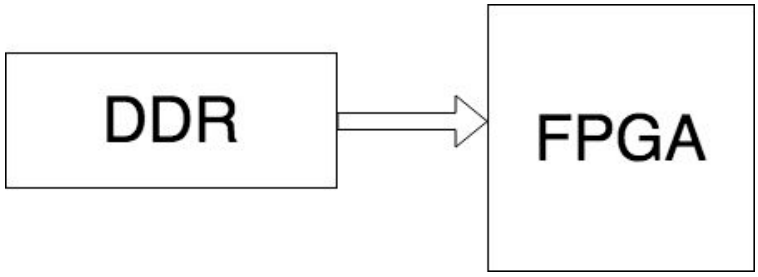


***Figure 5.7**: System Top Level*

Our initial NN V0 as well as the extended architecture V1 and V2 are designed to receive input data in two stages. First, it receives the initial values for the weights and biases parameters during the initialisation of the NN. In this case, for architecture V0, an input stream of 32 bits width is required while for architectures V1 and V2, two input streams of 32 bits are required which result in a maximum input bandwidth demand (new data in every clock cycle) of 1,6 and 3,2 GB/s respectively. Second, it receives the training examples during the training process. Each example is received through an input stream of 32 bits width along with a 10 bits input of the training label. The architectures are designed to process 10 examples in parallel in every training pass. This results in the need of 420 width input stream with maximum input bandwidth (new data in every clock cycle) of 4 GB/s which remains the same for all architectures.

The Zynq UltraScale+ board that we use provides four 128-bit/64-bit/32-bit HP AXI interfaces (up to 512 bits). These interfaces can accomplish theoretical input stream of 16 GB/s rate. This means that, for the system initialisation and examples fetch, the number of inputs and bandwidth would be more than sufficient. As a result during our simulation, we assume that we can have a new input in every clock cycle. Furthermore, parallel load of parameters could lead in faster initialisation but this is something that we did not apply in our research.

5.5.1. Architectures Initialisation

Initialisation of the architectures presented in previous chapters regards the storage of the initial parameters in the weights units that were described in chapter 5.1.1.2 and 5.1.2.3. Architecture V1 has three weights units, one for the parameters of each layer of the network, while architectures V1 and V2 have additional two weight units, one for the parameters of every SG module. We assume that the initial weights and biases parameters values are stored in files in the DDR memory and are passed sequentially into the FPGA in order to be stored into the weight units. The fact that parameters are read sequentially results in time complexity

of initialisation linear to the number of weights to be written. In the next table we present the number of parameters for each architecture.

Component	Inputs	Outputs	Weights	Biases
Layer 1	64	128	8192	128
Layer 2	128	64	8192	64
Layer 3	64	10	640	10
Total Parameters:	17226			

***Table 5.2**: Parameters of architecture V0*

Component	Inputs	Outputs	Weights	Biases
Layer 1	64	128	8192	128
Layer 2	128	64	8192	64
Layer 3	64	10	640	10
SG 1	138	128	17664	128
SG 2	74	64	4736	64
Layers Parameters:	17226			
SG Parameters:	22592			
Total Parameters:	39818			

***Table 5.3**: Parameters of architectures V1 and V2*

We have implemented two input ports for writing the initialization weights. One for the layers weights and a second one for the weights of SG modules (figure 5.4). As a result, assuming we would use one port of the DDR for the weights and a second one for the synthetic weights, the theoretical initialisation overhead for the extensions applied for architectures V1 and V2 is going to be the clock cycles required for storing the overhead number of parameters. In previous chapter, we showed that a single training pass of the NN V0 takes 409979 clock cycles to complete and ~550000 clock cycles for architectures V1 and V2 while the full training for the UCI dataset would take up to 6400 training passes. This means that initialisation time is 17226 clock cycles for V0 and 39818 for V1,V2 while overall operation time is ~$26234*10^5$ and ~$35321*10^5$ respectively. As a result, we considered the time overhead added for the initialisation of the extension architectures to be inconsiderable small.

5.5.2. Examples Data Fetch

In this thesis, we investigate ways that we can accelerate the training of a NN through acceleration of the data processing. In order to evaluate the need of that, we need to ensure that the bottleneck of the overall process applies in the processing of the data rather that the fetch of the data. For this reason we compare the clock cycles needed to fetch every training examples batch to the clock cycles needed for processing each training examples batch. The examples data fetch is fulfilled within 64 clock cycles for all architectures while processing of each example batch data is ~$40*10^4$ clock cycles for architecture V0 and ~$55*10^4$ clock cycles for architectures V1 and V2. We consider this to be a reasonable input data fetch time.

5.6. Results

In the previous chapter we concluded that there is increase of the new input interval of the NN in both architectures V1 and V2 compared with that of architecture V0. Conversely in chapter 4.4 experiments we show that this model achieves faster convergence than normal backpropagation. In order to make an accurate comparison of the two models training time we present the training error convergence and the accuracy of the NN according to the operation time. The time is calculated given the number of clock cycles and the estimated clock period that are provided by the Vivado HLS RTL simulation equal to 10ns.

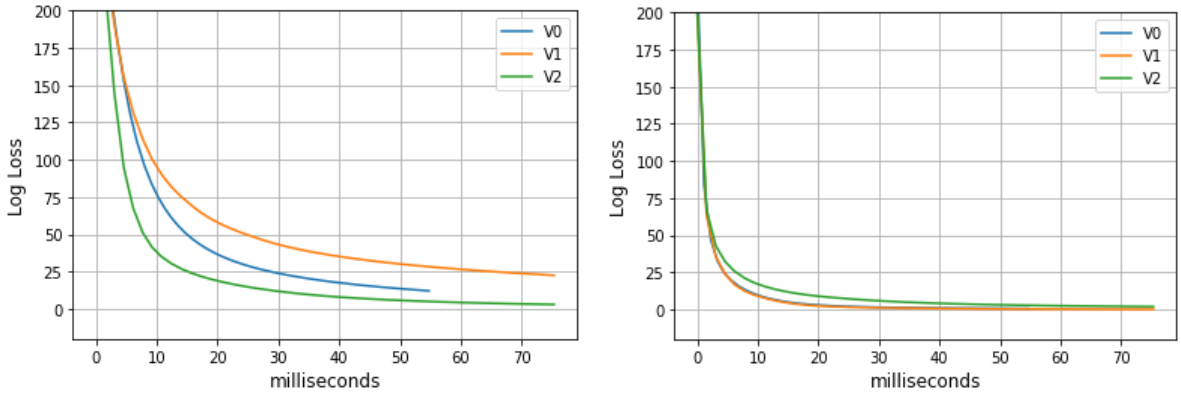
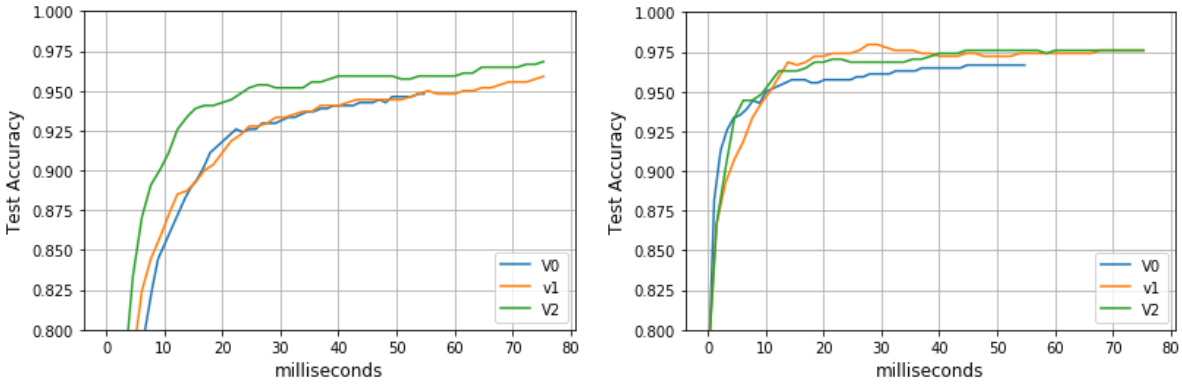


***Figure 5.12**: Training error in time domain*

Training error convergence comparison of models V0, V1 and V2 over milliseconds of operation with layers learning rate α=0.1 (a) and α=0.7 (b).



(a) ***(b)***

Figure 5.13*: Test accuracy in time domain*

Accuracy on the test set comparison of models V0, V1 and V2 over milliseconds of training operation with layers learning rate α=0.1 (a) and α=0.7 (b).

The measurements of accuracy according to training time shows that although models V1 and V2 result in longer new input interval time, they actually achieve faster training in overall. This is accomplished because the decrease of the training steps that are required along with the bigger time overhead applied in the architectures operation latency results in overall shorter time.

In the following table we present the speedup in operation time for certain accuracy levels of the architectures V1 and V2 against the initial architecture V0. The operation time speedup according to accuracy experiment can give us a clear view of the result of our attempt to speedup the training process.

Accuracy	V1	V2
0,91	1,00	1,68
0,92	0,93	1,76
0,93	1,05	2,21
0,94	1,06	2,37
0,95	1,03	2,43
0,96	1,04	2,70
0,97	1,04	3,00

(a)

Accuracy	V1	V2
0,93	1,00	1,00
0,94	0,80	1,25
0,95	1,13	1,22
0,96	4,03	3,15
0,97	4,22	4,00
0,98	5,00	5,02

(b)

Table 5.4*: Operation time speedup*

Operation time speedup, compared on RTL simulation time, of architectures V1, V2 versus architecture V0 with layers learning rate α=0.1 (a) and α=0.7 (b).

The results of architecture V1 training for small learning rate presented in table 5.4 show that synthetic gradient can efficiently replace the backpropagation algorithm. The time overhead that is added in the training cycle is counterbalanced by the decrease of needed steps for convergence. As a result V1 achieves neither speedup nor delay and can reach the levels of accuracy of initial model V0 in approximately same operation time. When a large learning rate is used table 5.4.b, V1 has achieves speedup of the training process. This happens because the regularisation effect added by the use of synthetic gadient is more important for cases of large learning rate. In this case, V1 needs much less steps to converge and as a result the overall training time for certain levels of accuracy is smaller. The effect is more tense for higher levels of accuracy achieving speedup of up to 5 for accuracy of 0,98.

The result of architecture V1 training for small learning rate (table 5.4.a) shows that model V2 can effectively achieve speedup of the training process as a result of reaching higher levels of accuracy using less training steps. The time overhead added by the extension of our architecture is counterbalanced by the speedup of training steps required. The effect is better than this of architecture V1 as it can achieve speedup of the training process for small learning rates. The effect is more important for high levels of accuracy reaching speedup up to 3 for accuracy of 0,97. When a large learning rate is used (table 5.4.b) the positive impact is even higher, as V2 can achieve overall higher accuracy of V0. As a result V2 achieves higher speedup for higher levels of accuracy reaching up to 5 for accuracy of 0,98. The fact that V2 performs better than normal backpropagation in both cases of small and big learning rates gives an insight that it could also achieve speedup of the training process for cases where a varied learning rate is used.

Chapter 6

Conclusions and Future Work

The purpose of this thesis was to investigate decoupled neural interfaces as a method for training deep neural networks. The investigation was held regarding the effect of the learning ability as well as a methodology that uses synthetic gradient for accelerating the training process of neural networks. We realised that replacement of backpropagation algorithm with DNIs can speed up the training process only in particular cases. In all cases though, the training speed was at least equal to this of the initial NN. On the other hand, DNIs can improve the performance of the training process, increasing the accuracy and representational strength due to the regularisation applied caused by the noise injected by the gradient error estimation of synthetic gradient.

Consequently, we showed that the combination of synthetic gradient with normal backpropagation gradient error (V2) can causes faster error convergence and faster increase of the accuracy. Due to the capability of parallelisation these two processes, the architecture that combines the synthetic gradient error and the backpropagation error results in almost same time complexity as the architecture that uses only the synthetic gradient error. The use of both errors during the training process achieves acceleration of the error convergences and as a result, speedup of the training process.

As future work, we propose the implementation of our architectures in hardware to and the expansion with low level design optimisation techniques like fine-grain parallelisation, pipelining etc. Further research on the use of DNIs could discover large neural network architectures where the use of DNIs could speed up the training process as well as deep architectures where models like these presented in this thesis could achieve higher acceleration. Finally the research could expand towards different neural network types that the synthetic gradient model would have a positive impact.

Bibliography

Books

[2] Vapnik V.. “Statistical learning theory”, 1998. New York: Wiley.

[6] Werbos, P. J.. “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”, 1974. Harvard University.

[8] Goodfellow I., Bengio Y., Courville A.. “Deep Learning”, 2016. MIT Press.
ISBN: 978-0262035613

[9] Aizenberg I., Aizenberg N.N., Vandewalle Joos P.L.. “Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications”, 2000. Springer US.

Publications

[1] Christian Robert, Machine learning, a probabilistic perspective, CHANCE 27 (2014), no. 2, 62–63.

[4] Gualtiero Piccinini, The first computational theory of mind and brain: A close look at mcculloch and pitts’s ”logical calculus of ideas immanent in nervous activity”, Synthese 141 (2004), no. 2, 175–215.

[3] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta, Revisiting unreasonable effectiveness of data in deep learning era, CoRR abs/1707.02968 (2017).

[5] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Review (1958), 65–386.

[7] Grzegorz Lewicki and Giuseppe Marino, Approximation of functions of finite variation by superpositions of a sigmoidal function, Appl. Math. Lett. 17 (2004), no. 10, 1147–1152.

[10] Yann Lecun, Lon Bottou, Yoshua Bengio, and Patrick Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE, 1998, pp. 2278–2324.

[11] Shun-ichi Amari, Backpropagation and stochastic gradient descent method, Neurocomputing 5 (1993), no. 3, 185–196.

[12] Sebastian Ruder, An overview of gradient descent optimization algorithms, CoRR abs/1609.04747 (2016).

[13] Samuel Rota Bul`o, Lorenzo Porzi, and Peter Kontschieder, Dropout distillation, in Balcan and Weinberger ICML 2016, pp. 99–107.

[14] Lei Jimmy Ba and Brendan J. Frey, Adaptive dropout for training deep neural networks, in Burges et al. 27th annual conference on neural information processing systems 2013, pp. 3084–3092.

[15] Diederik P. Kingma, Tim Salimans, and Max Welling, Variational dropout and the local reparameterization trick, CoRR abs/1506.02557 (2015).

[16] Hyeonwoo Noh, Tackgeun You, Jonghwan Mun, and Bohyung Han, Regularizing deep neural networks by noise: Its interpretation and optimization, CoRR abs/1710.05179 (2017).

[17] Alan F. Murray and Peter J. Edwards, Enhanced MLP performance and fault tolerance resulting from synaptic weight noise during training, IEEE Trans. Neural Networks 5 (1994), no. 5, 792–802.

[18] Chuan Wang and Jos´e C. Pr´ıncipe, Training neural networks with additive noise in the desired signal, IEEE Trans. Neural Networks 10 (1999), no. 6, 1511–1517.

[19] Rajat Raina, Anand Madhavan, and Andrew Y. Ng, Large-scale deep unsupervised learning using graphics processors, in Danyluk et al. ICML 2009, pp. 873–880.

[20] K. Ganeshamoorthy and D. N. Ranasinghe, On the performance of parallel neural network implementations on distributed memory architectures, in 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), 19-22 May 2008, Lyon, France , pp. 90– 97.

[21] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide, Pipelined back-propagation for context-dependent deep neural networks, in INTERSPEECH 2012, 13th Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012, pp. 26–29.

[22] Alex Krizhevsky, One weird trick for parallelizing convolutional neural networks, CoRR abs/1404.5997 (2014).

[23] Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, L´eon Bottou, and Kilian Q. Weinberger (eds.), Advances in neural information processing systems 25: 26th annual conference on neural information processing systems 2012. proceedings of a meeting held december 3-6, 2012, lake tahoe, nevada, united states, 2012.

[24] Balduzzi, D., Vanchinathan, H., and Buhmann, J. (2014). *Kickback cuts Backprop's red-tape: biologically plausible credit assignment in neural networks*. arXiv:1411.6191.

[25] Philip S. Thomas, Policy gradient coagent networks, Advances in Neural Information Processing Systems 24 (J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, eds.), Curran Associates, Inc., 2011, pp. 1944–1952.
[26] Ronald J. Williams and David Zipser, A learning algorithm for continually running fully recurrent neural networks, Neural Comput. 1 (1989), no. 2, 270–280.

[27] Yann Ollivier and Guillaume Charpiat, Training recurrent networks online without backtracking, CoRR abs/1507.07680 (2015).

[28] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu, Decoupled neural interfaces using synthetic gradients, CoRR abs/1608.05343 (2016).

[29] Wojciech Marian Czarnecki, Grzegorz Swirszcz, Max Jaderberg, Simon Osindero, Oriol Vinyals, and Koray Kavukcuoglu, Understanding synthetic gradients and decoupled neural interfaces, CoRR abs/1703.00522 (2017).

[30] Daniel Le Ly and Paul Chow, High-performance reconfigurable hardware architecture for restricted boltzmann machines, IEEE Trans. Neural Networks 21 (2010), no. 11, 1780–1792.

[31] Lok-Won Kim, Sameh Asaad, and Ralph Linsker, A fully pipelined FPGA architecture of a factored restricted boltzmann machine artificial neural network, TRETS 7 (2014), no. 1, 5:1–5:23.

[32] Urs Muller and A Gunzinger, Neural net simulation on parallel computers, 12 2018, pp. 3961 – 3966 vol.6.

[33] Chao Wang, Qi Yu, Lei Gong, Xi Li, Yuan Xie, and Xuehai Zhou, DLAU: A scalable deep learning accelerator unit on FPGA, CoRR abs/1605.06894 (2016).

Webpages

[34]*https://matplotlib.org/index.html*

[35]*http://archive.ics.uci.edu/ml*

[36]*https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html*

[37]*https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html*

[38] *http://yann.lecun.com/exdb/mnist/*

[39]*https://scikit-learn.org/0.19/modules/generated/sklearn.datasets.fetch_mldata.html#sklearn.datasets.fetch_mldata*

[40]*https://mldata.org*

Data Sheets

[41]*https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf*

[42]*https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html*

[43]***Vivado Design Suite User Guide High-Level Synthesis UG902 (v2012.2) July 25, 2012***