Hardware Accelerator for the Training of Neural Networks

James Erik Groving Meade

Kongens Lyngby 2019

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

The goal of the thesis is to ...

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with ...

The thesis consists of ...

Lyngby, 28-June-2019

Not Real

James Erik Groving Meade

Acknowledgements

I would like to thank my....

Contents

Date accessed for bib

Chapter 1

Introduction

The following text is a message to the student and should be removed during the writing process.

Please note the following instructions regarding an MSc thesis outlined in the study handbook:

"During the first month, the student is to submit a project plan outlining the objective of the thesis and justification for same to his/her supervisor. In the project plan, the student is also to take into account the overarching learning objectives listed above. When submitting the thesis, the student is to enclose a separate document presenting the original project plan and a revision of same, where appropriate. In addition, the document is to include a brief auto-evaluation of the project process."

To learn more about the rules for an MSc thesis, please consult the rules for your own MSc programme at `http://sdb.dtu.dk`.

1.1 Project plan

We note that the contents of the project plan is also something we would like to see in the introductory chapter of your thesis. In fact, you can reuse your final project plan (possibly extended) as the introduction. If you prefer to write an introduction from scratch, it is, of course, important that it is consistent with the final project plan.

1.2 The “separate document”

It is also important to note that the separate document containing

- original project plan
- possibly revised project plan.
- brief self-evaluation

mentioned above will be passed on to the external examiner and since it contains the learning goals and the objectives for your thesis, it will be taken into account when your thesis is assessed.

CHAPTER 2

Background

2.1 Neural Networks

A neural network is a machine learning tool ideal for conducting supervised learning. As a relatively recent field, the application of neural networks has rapidly extended across many domains, such as facial recognition at Facebook [TYRW14], translation for Microsoft [XHQ+16], spam filters for Google's Gmail [gma15] and more. As such, it continues to be a hot topic in today's world of research.

2.1.1 The Neuron

The *neuron* is the basic computational unit of a neural network. A *layer* is comprised of one or more neurons. The computation performed by a neuron is shown below.

$$\text{net} = \mathbf{w} \cdot \mathbf{x} + b \tag{2.1}$$

$$y = f(\text{net}) \tag{2.2}$$

The *fan-in* to a neuron is the amount of elements in the input vector $\mathbf{x} = x_1, x_2, \ldots, x_n$. For each element, there is a corresponding parameter referred to

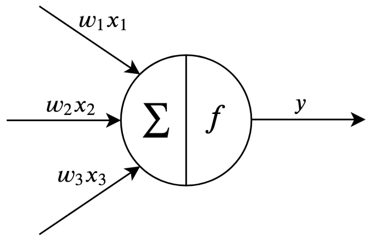

Figure 2.1: A neuron with 3 inputs; bias term omitted for simplicity.

as a *weight*. The weights of a neuron form the weight vector $\mathbf{w}$. The neuron also has an offset b which helps with normalization. The neuron's net is first computed as shown in equation 2.1, and then the output, or activation, is computed according to the neuron's activation function. This is shown visually in figure 2.1.

Weight Initialization Proper weight initialization is paramount to successfully training a neural network. Firstly, weights cannot be all initialized to 0, for this will result in the same gradient for all weights, and thus all weights will be updated in the same manner. This would effectively mean that the network would become a function of a singular weight.

The most naïve way to initialize weights would to assign each weight a random value between some range. In most cases, this is good enough for the network to converge to a relatively optimal solution so long as the range is not to extreme. A recent popular and effective way to initialize the weights is through He Initialization, which randomly initializes weights using a normal distribution with a mean of 0 and a variance of $\frac{2}{\text{fan\_in}}$ [HZRS15].

2.1.2 Fully-Connected Layers

A fully-connected layer is a vector of neurons. All neurons in a fully-connected layer receive the same input vector. This vector is the previous layer's output. A fully-connected layer with 3 neurons receiving input from an input layer is shown in figure 2.2. The output is a vector comprising of the outputs of each neuron. Each neuron output is calculated using the M-sized input vector as

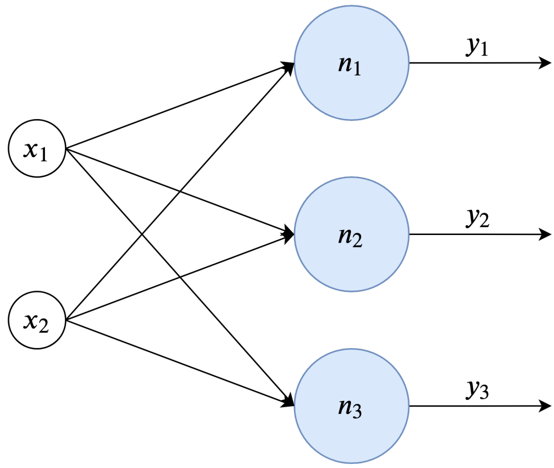

Figure 2.2: A fully-connected layer with 3 neurons, each receiving an input vector of size 2 from the input layer.

shown in equation 2.3 and added to output vector $\mathbf{y}$.

$$y_i = f_{\text{act}}\left(b + \sum_{j=1}^{M}(w_j x_j)\right) \tag{2.3}$$

$$\mathbf{y} = \{y_1, y_2, \ldots, y_n\} \tag{2.4}$$

2.1.3 Activation Functions

Without activation functions, the neural network would simply devolve to a linear classifier. Activation functions provide neural networks with the non-linearity to solve complex classification problems. Two of the most common activation functions are the rectified linear unit (ReLU) and the softmax function. These are the two activation functions that were chosen to be used in the software and hardware models of this thesis.

ReLU ReLU is a powerful activation function that has found widespread use due to its mathematical simplicity. The ReLU function is shown in equation 2.5.

$$y = \max(0, x) \tag{2.5}$$

Notably, the ReLU function is much easier to compute compare to the sigmoid or hyperbolic tangent functions, which both use the exponential function. The ReLU function also quite frequently performs just as well if not better compared to other activation functions. One of the reasons is because it does not suffer as much from the vanishing gradient problem [GBB11]. The vanishing gradient problem is encountered during training using backpropagation, which uses the chain rule from calculus, briefly covered in section 2.1.5. Since gradients will always be less than 1 for most loss functions, the gradients become geometrically smaller with each layer. Since ReLU only saturates in one direction, ReLU networks will be more sparse, in the sense that many of the neurons will have an output of 0.

ReLU-based neural networks also tend to reach convergence quicker than neural networks using the sigmoid or the hyperbolic tangent functions. It also results in a sparsely activated network, in that since the neuron output is 0 if the net is negative, that many neurons in the network will have an output of 0. This is also similar to how biological neurons also follow a sparse firing model, and has shown to be effective [GBB11].

Conversely, since active neurons in ReLU network are sparse, this brings rise to another potential problem, the "Dying ReLU Problem." This problem occurs when the sparsity increases to the point where a large majority of the neurons in the network become inactive during training and ultimately never become active again. Fortunately, this problem can be ameliorated with proper weight initialization [LSSK19].

Softmax The softmax function converts a vector of logits to a vector of probabilities. It has seen widespread use in neural networks that are used to predict the class of an input. The softmax function is shown in equation 2.6.

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\Sigma_{j=1}^{C} e^{x_j}} \tag{2.6}$$

In this function, x_i is the net of neuron i from the layer. Generally, the softmax function is used in the last layer to generate probabilities for multi-class problems. Each neuron in the layer represents a class, so the size of the last layer is equivalent to the number of classes, C. In much of the literature, the softmax portion of a neural network is referred to as the softmax layer as opposed to simply being the activation function of the neuron nets in the last layer.

2.1.4 Cross-Entropy Loss

Cross-entropy loss is a probabilistic loss function and as such, is frequently paired with the softmax activation function. This allows for the probabilities output from the softmax function to be used as inputs for calculating the cross-entropy loss. Cross-entropy loss is computed using probabilities and is shown in equation 2.7.

$$\mathcal{L}(\mathbf{x}) = \sum_{i=1}^{N} q(x_i) \log(p(x_i)) \tag{2.7}$$

In this function, $q(x_i)$ is the true probability of x belonging to class i, therefore, $q(x_i) = 1$ when x is of class i and 0 otherwise; $p(x_i)$ is equal to the predicted probability.

2.1.5 Backpropagation

Backpropagation is a method in which the weights of a network can be trained on a dataset by propagating the loss (also referred to as gradient in gradient descent) from the output layer backward through the network. There are three computational steps to be made during backpropagation: propagating loss gradients to the previous layer, using loss gradients for neurons in a layer to calculate individual weight gradients, and then finally to update the weights.

Calculating the Loss Gradients in the Output Layer For the first part of backpropagation, we must use the partial derivative of the loss function with respect to each of the neuron outputs to begin backpropagation. Note that the cross-entropy loss is calculated directly from the probabilities from the softmax function of the last layer. Therefore, the loss must derive the loss function with respect to the probabilities, and then must derive the softmax function in order to attain $\frac{\delta \mathcal{L}}{\delta \mathrm{net}_o}$ for the neurons in the last layer. The calculus is omitted for brevity, but the final result is clean and simple, as shown in equation 2.8 [sm-].

$$\frac{\delta \mathcal{L}}{\delta \mathrm{net}_{o,i}} = p_i - y_i \tag{2.8}$$

This equation calculates the partial derivative of the loss with respect to the net of the last layers output neuron. p_i is the probability computed from the softmax function and y_i is the true probability. Thus, if the input sample belong to class i, y_i is equal to 1, otherwise y_i is 0. Once the initial gradient for each neuron in the last layer has been calculated, backpropagation of the loss through the previous layers is possible.

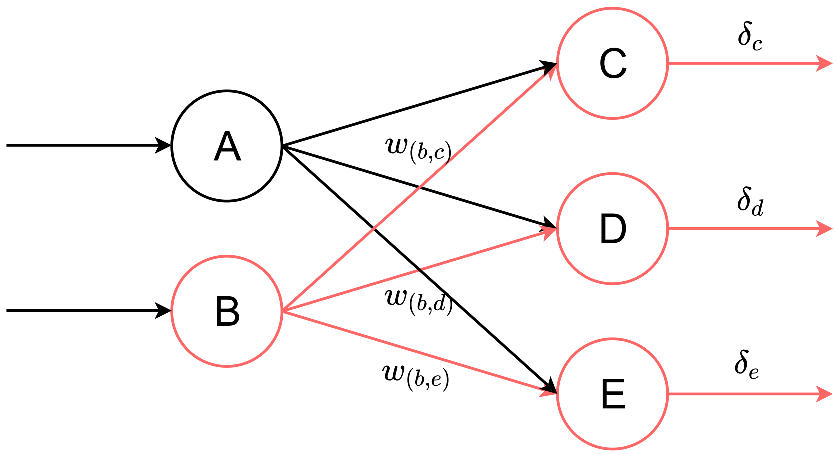

Figure 2.3: Example of backpropagating the loss gradient to the previous layer. Values used in backpropagating the loss to neuron B shown in red.

Backpropagating the Loss Gradient The strength of backpropagation is being able to use the chain rule to calculate gradients from previous layers. At a high-level, a neuron in a previous layer's output will affect the nets of neurons in the next layer. Since each activation is multiplied by a weight, the affect on the net is determined by a weight. For example, if a neuron's activation a_o increases by ϵ, then each of the next layer's neuron nets will increase by $w \times \epsilon$, where w is the weight for that connection. This connection is also somtimes referred to as a *synapse*, a term inspired from neuroscience.

An example illustrating this is shown in figure 2.3. The gradients for the nets of C, D, and E are represented by δ. The gradient of a net is commonly referred to as the *sensitivity* of a neuron. Subsequently, the weights on the synapses are also shown. With this knowledge, we can calculate $\frac{\delta \mathcal{L}}{\delta B}$ as shown in equation 2.9.

$$\frac{\delta \mathcal{L}}{\delta B} = \delta_c w_{(b,c)} + \delta_d w_{(b,d)} + \delta_e w_{(b,e)} \tag{2.9}$$

In more formal mathematical terms, if we know the $\frac{\delta \mathcal{L}}{\delta \text{net}}$, or δ, for each neuron in a layer with n neurons, then we can calculate the gradient for any neuron i's activation in the previous layer containing m neurons as shown in equation 2.10.

$$\frac{\delta \mathcal{L}}{\delta m_i} = \sum_{j=1}^{n} \delta_j w_{(i,j)} \tag{2.10}$$

The sensitivity for the neurons in layer m can then be computed using the derivative of the activation function. Since this thesis only uses ReLU, the derivative is simple to calculated and shown in equation 2.11. Note that the ReLU derivative is undefined at 0, however, in practical cases using a derivative of 0 works fine.

$$f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \\ \text{undefined} & x = 0 \end{cases} \tag{2.11}$$

Calculating Weight Gradients Once the sensitivity δ of neuron is known, calculating the gradients of individual weights and biases is possible. From a high-level, if we increase weight w by ϵ, then the product term of the net for the neuron will be $(w + \epsilon)a_i$, a net increase of $a_i \times \epsilon$. Therefore, the gradient for a weight is dependent on how large the weight's corresponding activation is. That means the weight corresponding to a large activation will have a much larger gradient than a weight corresponding to a small activation.

Returning to the previous example, the figure has now been updated to show how weight gradients for neuron C, this is shown in figure 2.4. The gradients for the 2 connecting weights are calculated as shown below. A_o and B_o are the activations of neuron A and B, respectively. As one would expect, the gradient of a weight is dependent on the magnitude of the neuron activation it is multiplied with, and the sensitivity of the neuron whose net it is summed with.

$$\begin{aligned} \frac{\delta \mathcal{L}}{\delta w_{(a,c)}} &= \delta_c A_o \\ \frac{\delta \mathcal{L}}{\delta w_{(b,c)}} &= \delta_c B_o \end{aligned}$$

Updating the Weights Once $\frac{\delta \mathcal{L}}{\delta w}$ is known for every single weight, the final step of backpropagation is to update the weights. This is performed by scaling the gradient for the weight by a value, known as the learning rate, η, and then subtracting it from the weight, since this will move the weight in the direction that lowers the loss. This is shown in equation 2.12.

$$w_{new} = w_{old} - \eta \frac{\delta \mathcal{L}}{\delta w} \tag{2.12}$$

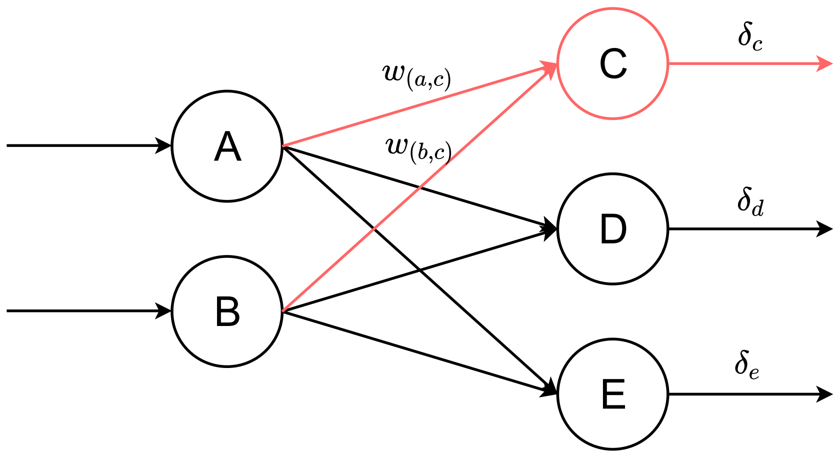

Figure 2.4: Example of computing weight gradients. Relevant values shown in red.

2.1.6 Hyperparameters

There are many hyperparameters to consider when designing a neural network. As described in section 2.1.5, the learning rate determines how of an impact the loss gradient has when updating the weight. Related to the learning rate is another hyperparameter known as momentum. The term is inspired from physics and has the effect of incorporating past updates in a geometrically decreasing fashion. We first define a few parameters:

m	—	the momentum parameter
v	—	'velocity'
η	—	the learning rate
dw	—	the loss gradient for some weight or bias w.

The momentum-based update can then be mathematically represented in the following manner:

$$
\begin{aligned}
v &= (m \times v) - (\eta \times w) \\
w &= w + v
\end{aligned}
$$

Each time each time we update w, previous updates update will also have an effect. A typical value for momentum is 0.9.

The final hyperparameter to be discussed in this section is batch size. Batch size determines the amount of forward and backward passes should be computed before performing a weight update. There is no typical value for the batch size and the optimal batch sizes varies largely by dataset and problem type.

2.2 Deep-Learning Frameworks

Deep-learning has come into the spotlight in the past few years and as such, many popular and robust frameworks have been developed. Some of the most popular frameworks are TensorFlow which is developed by Google, and PyTorch which is developed by Facebook. Other popular frameworks include Keras and Caffe. These frameworks are generally relatively simple to use while delivering high performance.

2.2.1 PyTorch

For this thesis, PyTorch has been chosen as the framework to construct a model against which to benchmark my results. PyTorch offers a simplistic interface to build highly customizable neural networks. In addition, it also has support for GPU-training, thus both CPU and GPU benchmarks can be obtained.

2.3 Related Work

With the surge in popularity of neural networks, there has been a lot of research focusing on improving the performance of inference and training. Zhao et al. developed a data-streaming solution (F-CNN) using an FPGA to perform 32-bit floating point training and inference. They used a CPU to communicate weights, addresses, and training data over a PCI-E bus, ultimately obtaining roughly a 4 times speedup over a CPU implementation and a 7.5 times more power-efficient compared to a GPU implementation [ZFL+16].

The Alternative Computing Technologies lab at the Georgia Institute of Technology has begun promising work on a framework called TABLA, that generates FPGA code for a multitude of machine learning models. The framework is based on creating individual processing engines inside processing units and thus creating a more generalized design by having schedulers assign work to these processing units. Training and inference have been tested with promising results on a Xilinx Zynq-7000 SoC and the group are aiming to make it public in the near future [MPA+16].

The majority of work based on accelerators for neural networks has been focused on inference. Google's Tensor Processing Unit (TPU) has found real-world success by performing inference on networks using Google's TensorFlow Lite framework [JYP+17]. This framework sacrifices precision for speed and obtains great performance with little punishment to overall accuracy, due to inference being less accuracy reliant than training. The Eyeriss is another hardware accelerator chip that has been developed to improve inference. It is designed by the Eyeriss Project team at the Massachusetts Institute of Technology. Its primary contribution is a novel dataflow optimization for convolutional neural networks called RS, for row stationary. This optimization allows for a high percentage of data reuse during inference [CES16].

The overwhelming majority of research regarding hardware acceleration for neural networks is focused on inference. Consequently, aside from the aformentioned F-CNN FPGA-based framework for training neural network, there has not been much research investigating neural network training on an FPGA. This work differs from the F-CNN as it investigates the effectiveness of using fixed-point arithmetic instead of floating point during training, allowing for extra speed at the sacrifice of accuracy. Furthermore, this work proposes that the main advantage of using a training accelerator is for datasets that train optimally with smaller batch sizes.

Perhaps make use of Courbariaux: Results from Courbariaux et al. show that training has much stricter accuracy requirements than inference.

Chapter 3

Software Model

figure for code snippets or nah?

3.1 Overview

This section documents the general-purpose neural network framework that was written in C++ for this thesis. There is an example program that trains on the MNIST dataset and documents epoch-by-epoch training statistics. MNIST is a dataset of handwritten digits, containing 60,000 training images and 10,000 test images. The source code for the software model can be found in the appendix as well as online on github.[1]

[1] `https://github.com/erikgroving/NeuralNetworkHardwareAccelerator/tree/master/SWModel`.

3.2 Motivation

The software neural network framework was written so that the FPGA hardware model could be benchmarked against a CPU-based model that performs neural network inference and backward passes using the same method as the hardware model. This benchmark could be used to evaluate the performance of the hardware model. In addition, it could be benchmarked against professional open-source deep-learning frameworks that make use of advanced algebraic methods to perform computation such as matrix multiplication that inherently offer more efficiency. Furthermore, by developing a software model, the algorithmic integrity of the proposed network was able to verified and tested in an expedient manner by using a well-known testing framework, Google Test. Finally, if high floating-point precision were needed for training a network, then the software model could be used to learn the weights and parameters, and then subsequently be loaded into the weight BRAM of the FPGA hardware model.

3.3 Design

3.3.1 Layers

The software model was designed to be flexible such that any neural network architecture may be constructed so long as the layer types were implemented. The model currently supports 2D convolutional, fully connected, and pooling layers.

All layers are derived from a base class, `Layer`. Certain methods such as `forward()` and `backward()` must be implemented by all derived classes. There is then a `Net` class that contains a `vector` of `Layer` objects. This allows for a flexible design, as one only need add layers to the `Net` object. Furthermore, the model can easily be extended to other layer types so long as the layer type derives from `Layer`.

The non-linear activation function used in the model is ReLU because the derivative is trivial to compute. Compared to the sigmoid function, ReLU is much more computationally feasible for an FPGA hardware implementation, and thereofre, ReLU was used in the software model so that both models would use the same activation function.

3.3.2 Training

The Softmax Function and Computing Loss Gradients The network uses an implicit Softmax function for the last layer since this converts the logits in the last layer to numbers that can be interpreted as probabilities, ideal for image classification.

The loss gradients for the neurons in the last layer are computed using multi-class cross entropy loss. Therefore, only one probability will account for loss, however, since each probability is an output from the softmax function which takes in all neuron outputs as input, all neurons in the last layer will have a loss gradient.

The derivative of this loss function is needed to perform backpropagation. We define $\mathcal{L}_i$ as the loss for neuron i in the last layer and z_i as the output of neuron i. We also introduce y_i, which is 1 if x is an instance of class i and 0 otherwise. We can then compute the loss gradient for neuron i in the last layer quite simply as follows:

$$\frac{\delta \mathcal{L}_i}{\delta z_i} = z_i - y_i$$

Batch Size The software model supports batch training and thus a batch size is to be specified when creating an instance of a new network.

Learning Rate and Momentum The software model learns using stochastic gradient descent. As such, the network is configured with a learning rate and momentum. The learning rate may be manually readjusted during training epochs.

3.4 Source Code Structure

The software model contains a Makefile and three folders: *data*, *src* and *test*. The *data* folder contains the MNIST binary data files, and is loaded by the example program that trains on the MNIST dataset. The *src* folder contains the source code of the neural network framework. The *test* folder contains test made using the Google Test C++ testing framework. The Makefile is used to build the source as well as tests. This section will detail the source files in the *src* folder that are core to the software model framework. The files *main.cpp* and *parse_data{.cpp, .h}* will be described in section 3.5 that focuses on usage.

Name	Type	Description
`in`	`uint32_t`	Size of the input to the neural network.
`out`	`uint32_t`	Size of the output of the neural network.
`bs`	`uint32_t`	Size of the batch size to be used when training the net.
`lr`	`double`	The learning rate to be used during training of the network. Can be set and read using the functions `setLearningRate()` and `getLearningRate()`.
`momentum`	`double`	The momentum to be used when performing updates to the weights and biases of the network.

Table 3.1: Description of parameters for the constructor `Net` class.

net{.cpp, .h} These files contain the definition of the `Net` class, the highest-level class of the network. After initializing a `Net` object, layers can be added to the neural network by calling the `addLayer()` method which will add a `Layer` object to a `vector`. The `Net` class also stores intermediate activations from the current inference, which are required when performing backward pass to calculate loss gradients. The key parameters to the `Net` object are set in its constructor, and are defined in table 3.1.

The `Net` class has a method `inference()` that computes the forward pass for a batch of inputs, thus the argument is a 2-d `vector`, with each outer index corresponding to an input. The `()` operator has also been overloaded to call `inference()`. This is all that is needed to compute a forward pass.

To compute the backward pass, `computeLossAndGradients()` should be called first. This method takes in the label data as a `vector` for the inputs as an argument and computes the loss gradients for the outer layer of the network. Next, a call to `backpropLoss()` should be made; this method propagates the outer layer loss gradients back through the neural network. After the loss has been back-propagated, weights of each `Neuron` in the network should be updated by calling `update()`. Previously cached forward pass activation data should then be cleared with a call to `clearSavedData()`.

layer.h This file contains the `Layer` class, which serves as the base class for all the different types of layer classes in the framework. It contains virtual methods `forward()` and `backward()`, representing the forward and backward pass functionality that must be implemented. All layer classes must also contain a `getType()` method to identify the layer type, as well as methods for `updateWeights()`, `clearData()`, and `getOutput()`.

Name	Type	Description
dim	uint32_t	Dimensions of the input. The dimension is assumed square, meaning that rows = dim and columns = dim.
filt_size	uint32_t	Dimension of the filter used for the convolution, dimension also assumed square.
stride	uint32_t	Size of the stride
padding	uint32_t	Padding used for convolution.
in_channels	uint32_t	Amount of channels in the input.
out_channels	uint32_t	Amount of channels in the output.

Table 3.2: Description of parameters for the `ConvLayer` class.

convolutional{.cpp, .h} These files contain the definition of the `ConvLayer` class, which implements a 2D-convolutional layer, and derives from the `Layer` class. A unique method to the `ConvLayer` class is the `getWindowPixels()` method, which returns the pixels inside the filter window, and is used when computing both the forward and backward passes. The class' constructor and key parameters are described in table 3.2.

fullyconnected{.cpp,.h} These files define the `FullyConnected` class. The class only has two defining parameters in its constructor: `in` and `out`, which are of type `uint32_t` and specify the input and output size to the layer, respectively. It derives from the base `Layer` class, so methods such as `forward()` and `backward()` are also implemented.

pooling{.cpp,.h} These files define the `PoolingLayer` class. The class derives from `Layer` and performs a 2D 2×2 max pooling operation. There are three main parameters for the class: `dim_i`, `dim_o`, and `channels`. The parameters `dim_i` and `dim_o` specify the dimension of the input and output feature vectors. Since the layer currently only performs 2×2 max pooling, `dim_o` will always be half of `dim_i`, though if different types of pooling filters were to be supported, then `dim_o` would be necessary. The `channels` parameter is used to specify the number of channels of size `dim_i` × `dim_i` present in the input.

neuron{.cpp, .h} These files define the `Neuron` class. The `Neuron` class is the computational building block of the fully connected and convolutional layers. The fan-in of the neuron is specified in the constructor. Weights should be initialized using the `initWeights()` method, which implements He initialization [HZRS15]. He initialization randomly initializes weights using a normal distribution with a mean of 0 and a variance of $\frac{2}{\mathrm{fan\_in}}$.

The class implements all necessary computational elements for a neuron in a neural network. During a forward pass, a neuron's net and activation are computed with `computeNet()` and `computeActivation()` respectively. When computing the backward pass, the gradients for the neuron's weights are computed using `calculateGradient()`. Weights can be subsequently updated using the `updateWeights()` function. Finally, all gradient data can be cleared using `clearBackwardData()`.

3.5 Usage

This section will show how the software model may be used for image classification. In the following example, the software model will be trained to classify handwritten digits from the MNIST database. Each image is a handwritten digit of size 28×28. The relevant files specific to this example are *main.cpp* and *parse_data.cpp*.

Load the Training and Testing Data The first step to any neural network problem is to load the training and testing dataset. The MNIST dataset is provided as binary files and helper functions to load the data have been provided in *parse_data.cpp*. Training and testing data can be loaded as shown below.

```
std::vector< std::vector<double> > trainX;
std::vector<int> trainY;
std::vector< std::vector<double> > testX;
std::vector<int> testY;
trainX = readImages("data/train-images.idx3-ubyte");
trainY = readLabels("data/train-labels.idx1-ubyte");
testX = readImages("data/t10k-images.idx3-ubyte");
testY = readLabels("data/t10k-labels.idx1-ubyte");
```

Create a `Net` Instance The next step is to create a `Net` object with the relevant hyperparameters to be used for the neural network. The below code accomplishes this.

```
int       input_size  = 28*28;
int       output_size = 10;
int       batch_size  = 200;
```

```
double  momentum    = 0.9;
double  lr          = 0.01;
Net net(input_size, output_size, batch_size, lr, momentum);
```

Create Layer Objects and Add them to the `Net` Object After the `Net` object has been created, layers need to be added to the network. Two configuration options are present in *main.cpp*; one implements a 7-layer convolutional neural network, and the other implements a 4-layer fully connected neural network. The below code snippet shows how the 7-layer convolutional neural network is implemented. The software model was designed with simplicity in mind, so the below code is relatively straightforward to follow.

```
Layer* conv1 = new ConvLayer(28, 3, 1, 1, 1, 8);
Layer* pool1 = new PoolingLayer(28, 14, 8);
Layer* conv2 = new ConvLayer(14, 3, 1, 1, 8, 16);
Layer* pool2 = new PoolingLayer(14, 7, 16);
Layer* fc1 = new FullyConnected(16*7*7, 64);
Layer* fc2 = new FullyConnected(64, 10);

net.addLayer(conv1);
net.addLayer(pool1);
net.addLayer(conv2);
net.addLayer(pool2);
net.addLayer(fc1);
net.addLayer(fc2);
```

Train the Net In *main.cpp*, a function `trainNet()` has been implemented, which trains the net using batch training. The actual training for a given batch only requires 5 lines of code, and is shown below.

```
net(in_batch);
net.computeLossAndGradients(out_batch);
net.backpropLoss();
net.update();
net.clearSavedData();
```

Build and Run the Model Compile the code by running `make` in the *SW-Model* directory. The model will then train for the amount of epochs specified in the call to the `trainNet()` function in `main()`. Since the model is initialized with random weights, the final result of training is non-deterministic. Output similar to the output shown in figure 3.1 can be expected. In this case, the fully connected model was used, and train to a maximum accuracy of 97.62%. it is also worth noting the expected differences in loss and accuracy between the training and test datasets. This discrepancy is expected as the network never learns from the test dataset. The difference between test and training dataset accuracy is normally used to quantify how well the network is able to generalize from the training dataset.

3.6 Testing

To ensure the correctness of the software model, several test suites were created during development. Source code for the test suites can be found in the *test* folder as well as in the appendix.

source code in appendix

3.6.1 Test Suites

Four test suites were created during the development of the software model. The test cases were written to test features as they were developed. As such, the tests include neuron functionality, forward pass for fully connected and convolutional layers, and finally a gradient checking test suite to verify the backward pass. This section elaborates on the test suites that were used during development.

Neuron Testing The neuron test suite, found in *neuron_test.cpp*, contains one primary test case that sets the weights of a neuron, computes the activation, and verifies that the activation is correct.

Fully Connnected Forward Pass The test case for a fully connected layer's forward pass is located in *fullyconnected_test.cpp*. The test case creates a `FullyConnected` layer that has 3 inputs and 4 outputs. The weights are then set and an input is sent forward through the layer. Each of the 4 outputs are then verified to be correct.

Convolutional Forward Pass There is a test case to verify the convolutional forward pass located in *conv_test.cpp*. The test creates a convolutional layer

```
Running software model...
Starting Accuracy
Total correct: 1022 / 10000
Accuracy: 0.1022

Epoch: 0
--- Training Stats ---
Total correct: 54914 / 60000
Accuracy: 0.915233
Loss: 0.290908
--- Test Stats ---
Total correct: 9183 / 10000
Accuracy: 0.9183
Loss: 0.280574

Epoch: 1
--- Training Stats ---
Total correct: 56213 / 60000
Accuracy: 0.936883
Loss: 0.218062
--- Test Stats ---
Total correct: 9390 / 10000
Accuracy: 0.939
Loss: 0.214584

...

Epoch: 36
--- Training Stats ---
Total correct: 59168 / 60000
Accuracy: 0.986133
Loss: 0.0516957
--- Test Stats ---
Total correct: 9762 / 10000
Accuracy: 0.9762
Loss: 0.0845137
```

Figure 3.1: An expected output from using the software model on the provided MNIST dataset. Epochs 2-35 omitted for brevity. In this training run, the network reached a maximum test set accuracy of 97.62%.

that takes a 2×2 feature vector with 2 channels, uses a 3×3 filter for convolution, uses a stride and padding of 1, and produces 2 output channels. Weights and inputs were the arbitrarily assigned and the forward pass was computed and verified against the output that had been previously calculated manually.

Gradient Checking It would be very tedious and error-prone to debug the backward pass of a neural network using manual calculations, thus the general standard method of testing the gradients computed during a backward pass is to use gradient checking. Note that during the backward pass, all the loss gradients for every single weight and bias are calculated. For every weight (and bias), the partial derivative $\frac{\delta \mathcal{L}}{\delta w_i}$ is computed. Gradient checking verifies that the mathematically computed analytic derivative aligns with a numerically estimated derivative [Kar]. The numerical gradient can be computed as follows:

$$\frac{\delta \mathcal{L}(w_i)}{\delta w_i} = \frac{\mathcal{L}(w_i + \epsilon) - \mathcal{L}(w_i - \epsilon)}{2\epsilon}$$

The partial derivative of the loss with respect to a certain weight w_i can thus be estimated by calculating the loss after incrementing w_i by a small ϵ, calculating the loss after decrementing w_i by ϵ, and then dividing the difference by 2ϵ. As long as ϵ is rather small, the derivatives should be near exact. In these test cases, $\epsilon = 10^{-4}$. Once we have the analytic and numerical gradient, we can compute the relative error as shown below:

$$\text{Relative gradient error} = \frac{|\mathcal{L}'(w_i)_a - \mathcal{L}'(w_i)_n|}{\max\ (|\mathcal{L}'(w_i)_a|,\ |\mathcal{L}'(w_i)_n|)}$$

If the relative error is below a certain threshhold, then it is safe to assume the gradient has been calculated correctly. In this test suite, the relative error threshhold must be lower than 10^{-7}.

The two test cases in *gradient_check_test.cpp* perform gradient checks for a fully connected network and for a convolutional neural network. The fully connected network gradient check test creates a neural network with an architecture shown in figure 3.2.

The test then creates 10 random inputs, each having a random label. Each input sample is fed forward through the network and analytic gradients are computed for each weight. The numerical gradient is then subsequently computed for a random weight. The random weight can belong to any neuron and any layer. This process of choosing a random weight, calculating the numerical gradient, comparing it to the analytic gradient is then repeated 100 times. The test asserts that the relative error is less than 10^{-7} each time. A portion of the computed analytic and numerical gradients are shown in figure 3.3.

```
int      input_size  = 100;
int      output_size = 2;
int      batch_size  = 1;
double   momentum    = 0.9;
double   lr          = 0.001;
Net net(input_size, output_size, batch_size, lr, momentum);


Layer* fc1 = new FullyConnected(input_size, 98);
Layer* fc2 = new FullyConnected(98, 64);
Layer* fc3 = new FullyConnected(64, output_size);

net.addLayer(fc1);
net.addLayer(fc2);
net.addLayer(fc3);
```

Figure 3.2: Layer created for the fully connected gradient check test.

```
Layer: 2, Neuron: 0,  Weight: 31
Analytic Gradient: -0.0638284 Numerical Gradient: -0.0638284

Layer: 0, Neuron: 93, Weight: 71
Analytic Gradient: -0.156235  Numerical Gradient: -0.156235

Layer: 1, Neuron: 34, Weight: 29
Analytic Gradient: -1.22615   Numerical Gradient: -1.22615

Layer: 1, Neuron: 12, Weight: 43
Analytic Gradient: 0.376021   Numerical Gradient: 0.376021
```

Figure 3.3: Results from the fully connected test using randomly sampled weights to perform gradient checking

```
int    input_size   = 8*8;
int    output_size  = 2;
int    batch_size   = 1;
double momentum     = 0.9;
double lr           = 0.001;
Net net(input_size, output_size, batch_size, lr, momentum);

Layer* conv1 = new ConvLayer(8, 3, 1, 1, 1, 3);
Layer* pool1 = new PoolingLayer(8, 4, 3);
Layer* conv2 = new ConvLayer(4, 3, 1, 1, 3, 6);
Layer* fc1   = new FullyConnected(4*4*6, output_size);

net.addLayer(conv1);
net.addLayer(pool1);
net.addLayer(conv2);
net.addLayer(fc1);
```

Figure 3.4: Layer created for the convolutional layer gradient check test.

The convolutional gradient checking test is set up in the same manner as the fully connected gradient checking test, except that the network structure is different. The network is now a **convolutional layer** — **pooling layer** — **convolutional layer** — **fully connected layer**. The input is randomized 8x8 data, and convolutional layers use 3×3 filters with a padding and stride set to 1. The first convolutional layer has 3 output channels and the second convolutional layer has 3 input channels and 6 output channels. The code used to create the network is shown in figure 3.4.

3.6.2 Building and Running the Test Suites

The test suites requires Google Test to compile. Google Test can be downloaded online at github [2]. The *googletest* directory should then be placed under the *SWModel* folder. The test suite can then be compiled using the provided Makefile and the following command:

```
> make all_tests
```

This will produce an executable in the *SWModel* directory called **all_tests**. The test suites can be run by invoking the executable. The output is shown in figure 3.5

[2] `https://github.com/google/googletest`

```
> ./all_tests
Running main() from ./googletest/src/gtest_main.cc
[==========] Running 6 tests from 4 test cases.
[----------] Global test environment set-up.
[----------] 1 test from ConvTest
[ RUN      ] ConvTest.TestForward
[       OK ] ConvTest.TestForward (1 ms)
[----------] 1 test from ConvTest (11 ms total)

[----------] 1 test from FCTest
[ RUN      ] FCTest.TestForward
[       OK ] FCTest.TestForward (0 ms)
[----------] 1 test from FCTest (10 ms total)

[----------] 2 tests from NeuronTest
[ RUN      ] NeuronTest.InitWeights
[       OK ] NeuronTest.InitWeights (0 ms)
[ RUN      ] NeuronTest.SetWeightsAndGetOutput
[       OK ] NeuronTest.SetWeightsAndGetOutput (0 ms)
[----------] 2 tests from NeuronTest (29 ms total)

[----------] 2 tests from GradientTest
[ RUN      ] GradientTest.FCGradientCheck
[       OK ] GradientTest.FCGradientCheck (950 ms)
[ RUN      ] GradientTest.ConvGradientCheck
[       OK ] GradientTest.ConvGradientCheck (2260 ms)
[----------] 2 tests from GradientTest (3223 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 4 test cases ran. (3329 ms total)
[  PASSED  ] 6 tests.
```

Figure 3.5: Test coverage output using the Google Test C++ testing framework to verify the correctness of the software model for both forward and backward passes.

Chapter 4

Hardware Model and Implementation

This chapter details the hardware designed during this Master's thesis to accelerate neural network training. The current hardware implements both training and inference acceleration for the neural network architecture described in section 4.2.

Refer to github, appendix, and project link

4.1 Specifications

The hardware model was implemented using a ZedBoard. The ZedBoard is a development board equipped with a Zynq-7000 XC7Z020 SoC. The Zynq series has both a processing system and programmable logic, where the processing system is a ARM Cortex-A9 based processor (hereafter referred to as the "PS") and the programmable logic is an Artix-7 series FPGA. Bitstreams for the FPGA were generated using Vivado 2018.3 and PetaLinux boot images for the PS were created using Xilinx SDK. The hardware description language (HDL) code for the project was primarily written in SystemVerilog. The programs run on the PS were written in C.

Layer Name	Input Size	Output Size
FC0	784 (28×28)	98
FC1	98	64
FC2	64	10
Softmax	10	10

Table 4.1: The hidden and output layers in the implemented neural network

4.2 The Implemented Neural Network

The classical MNIST handwritten digit dataset was chosen as the problem setting for the hardware model as a proof-of-concept. This problem has been chosen to verify the value in designing accelerators that take advantage of the finer-grained parallelism present in neural networks. The network consists of an input layer, 3 fully-connected layers, and a softmax output layer. The input layer is a 28×28 grayscale image of a handwritten digit. The dimensions of the rest of the layers in the network are shown in table 4.1. Layers whose name starts with FC are fully-connected layers.

Note that in this implementation, while biases are supported for forward computation, they are not used as the MNIST dataset is already fairly normalized. As such, the biases read are always 0 during the forward pass, and during the backward pass, no updates or gradients are calculated for the bias. Note that the gradient of the bias would just be the gradient of the neuron, unless the neuron had a ReLU activation function with negative net, so implementing this update would be trivial as all neuron gradients are already calculated. In addition, the current implementation only supports training with a batch size of 1.

4.3 Design Goals

There were a few key principles that guided the overall design process throughout the development of the hardware accelerator. A core tenet was to maintain the project such that in the future HDL could be generated for training a network of any architecture so long as the desired layer types had an implementation. As a result, all layers have been modularized and internal components are parameterized. Designing in a modular and parameterizable fashion also allows for quick and easy readjustments to the neural network architecture if needed.

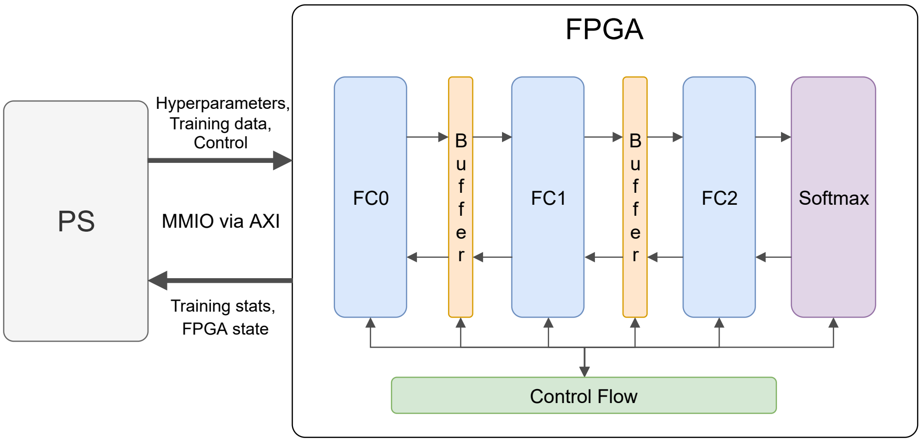

Figure 4.1: Architecture of the hardware accelerator

In addition, optimal usage of resources available was prioritized. For example, the limiting FPGA resource was the amount of digital signal processing slices (DSPs). Therefore, the FPGA design optimized the distribution of DSPs over other resources as opposed to saving an extra Block RAM (BRAM) block.

4.4 Overall Architecture

In the hardware model, both the Zynq's PS and the FPGA were used to facilitate a cohesive and efficient architecture to accelerate neural network computation. The overall system architecture can be seen in figure 4.1.

Through memory-mapped I/O, the PS transfers neural net hyperparameters, training data, and control signals to the FPGA. The FPGA transfers training statistics and state data back to the PS. The interface is further described in sections 4.8 and 4.8.1.

Inside the FPGA, the neural network described in section 4.2 is implemented. Layers are connected in both forward and backward directions in order to support training. There are three types of primary modules in the top-level of the FPGA: fully-connected layers, interlayer activation buffers, and the softmax layer. In addition, there is a general control flow in the top-level with which all the primary modules interact.

4.5 Training Process

4.6 Computational Precision

In this implementation, a bit-width of 18 was chosen for all weight gradients and activations. This value was chosen because the multiplication portion of DSP slices have an input multiplicands with bit widths of 25 and 18

cite dsp

. This thesis uses the Q number format to define precision types. For example, Q10.6 would mean that a 16-bit value has 10 integer bits and 6 fractional bits [cen01]. For this accelerator, activations have a precision of Q6.12. Weights and weight gradients both have a precision of Q1.17. These values were chosen through experimental analysis of minimum and maximum activation, weight, and gradients values using the software model described in chapter 3.

4.7 Module Architecture

As mentioned in the design goal section, one of the tenets of this design was to allow for modularity and parameterization, such that changing a network architecture would not require too much work. As such, there are a few global parameters defined, such as the amount of bits specified for the fixed-point precision. There are also parameters defined for each of the fully-connected layers. These parameters can all be found in the *sys_defs.vh* file in the Appendix

fpga appendix

, or on GitHub.

4.7.1 Fully-Connected Layers

The fully-connected layer modules implement both forward and backward passes. The general architecture is shown in figure 4.2. As DSP slices are limited,

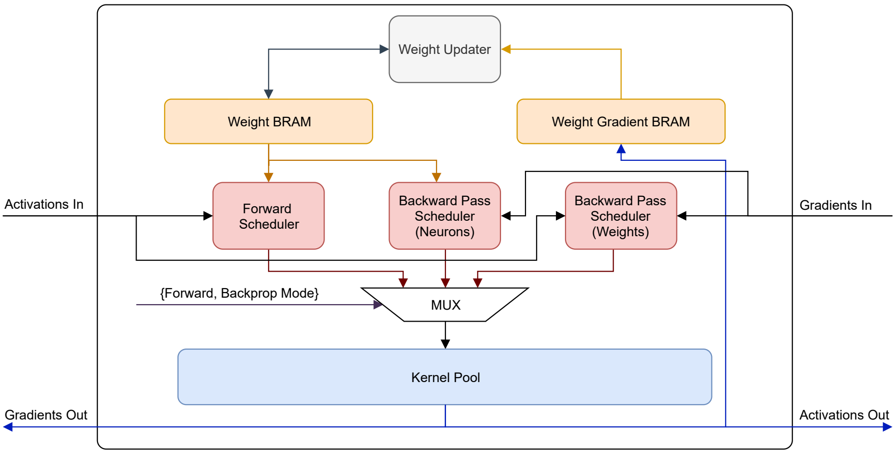

Figure 4.2: Architecture of the fully connected layer

both the forward and backward computational units make use of the same resources to compute multiplications, known as the kernel pool. There are 4 modes of computation in the fully-connected layer: forward pass, backpropagating neuron gradients, computing weight gradients, and updating the weights. Of these 4 modes of computation, all except updating the weights make use of the kernel pool. This is because updating the weights makes uses of bit shifting instead of multiplication to multiply gradients by the learning rate.

The forward pass multiplies weights and input activations to produce output activations. Backpropagating neuron gradients multiplies weights by current layer input gradients to produce previous layer gradients as output. The weight gradient computation mutliplies input activation from the forward pass by the current layer gradient, and then writing the computed gradient to the weight gradient BRAM.

Since being flexible and modular was one of the design goals, all the fully-connected layers use the same kernel and scheduler modules, with different parameters in the instantiation.

Scheduling Each of the computational modes needs to have a scheduler to generate addresses to be read and guide the computation. For this, the a generalized scheduler module was implemented. The scheduler uses two pointers starting from the head and middle of the BRAM, and iterates through the entirety during the forward pass. Since the weight BRAMs of each layer are different, certain parameters are assigned for instantiations of the scheduler are also different.

```
fc_scheduler #(.ADDR(`FC1_ADDR),
        .BIAS_ADDR(`FC1_BIAS_ADDR),
        .MID_PTR_OFFSET(`FC1_MID_PTR_OFFSET),
        .FAN_IN(`FC1_FAN_IN))
        fc1_scheduler_i (
        //inputs
        .clk(clk),
        .rst(rst),
        .forward(forward),
        .valid_i(sch_valid_i),
        //outputs
        .head_ptr(head_ptr),
        .mid_ptr(mid_ptr),
        .bias_ptr(bias_ptr),
        .has_bias(sch_has_bias)
);
```

Listing 4.1: The instantiation of the scheduler for the FC1 layer

The instantiation of the scheduler shown in listing 4.1 is similar across all the 3 fully connected layers, with only the parameters in the instantiations differing. The outputs are the pointers whose starting addresses are at the head and middle of the weight BRAM. In addition, there is a bias pointer and a signal to indicate if there is a bias.

Kernel The same kernel module is used in all the fully-connected layers. A high-level architecture of the computational kernel is provided in figure 4.3. Note that saturation checking is not shown in the figure for simplicity, though it is implemented and verified. The scarcest computational resource in this FPGA architecture are the DSP slices, thus the kernel has been designed so that all required forms of multiplication are supported in the network. This is why there are 3 different outputs.

From the figure, the top output is the neuron gradient. This multiplies a weight with a gradient. Multiplying two Q1.17 values results in a Q2.34 product, which must be checked for saturation in the top 2 bits and the bottom 17 bits must be truncated to obtain a Q1.17 output.

The middle output is the weight gradient. The weight gradients computation multiplies a gradient with an activation. As gradients are Q1.17 and activations are Q6.12, the output is Q7.29. To convert the resultant Q7.29 result to the

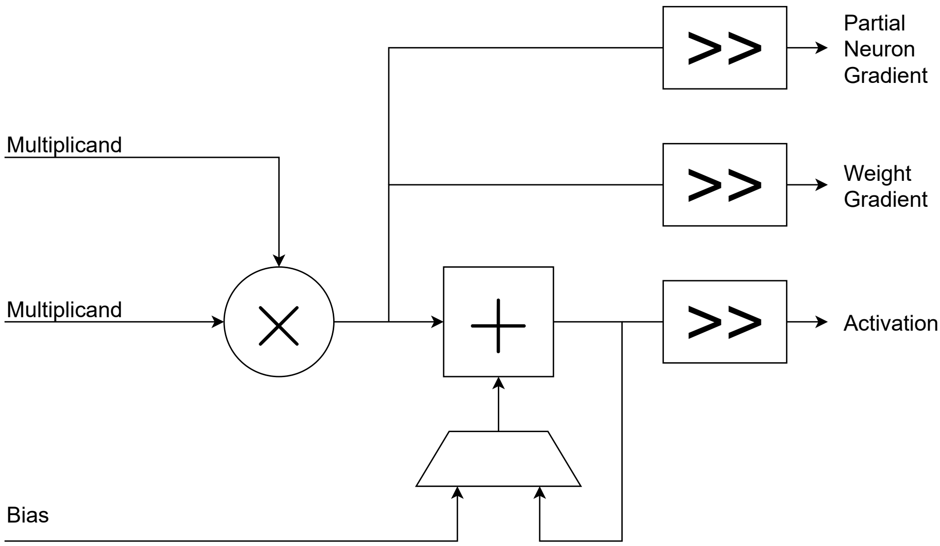

Figure 4.3: Architecture of the kernel, saturation checking not shown.

desired Q1.17 format required for a weight gradient, the top 7 bits must be checked for saturation and the bottom 12 bits truncated.

Finally, the bottom output is the net output calculated during the forward pass. This value becomes valid after performing n MACs, where n is the fan-in of a neuron. An input activation and corresponding weight are multiplied and added to either a bias or the running sum for the current in-progress net calculation.

Since the forward pass multiplies Q6.12 activations by Q1.17, the multiplied result is 36-bits, Q7.29. Since for some layers fan-in can be quite large, extra precision is used during the accumulation phase. The accumulated sum uses 32 bits: Q6.26, thus the internal sum must be checked for saturation and truncation of the bottom 3 bits for each MAC. The conversion from the internal sum precision of Q6.26 to the net output of Q6.12 is a simple truncation of the bottom 14 bits.

With this amount of internal precision during the forward pass, truncation error during the internal summation is sufficiently minimized. This is because the largest fan-in in this network is 784 in the FC0 layer. For each internal MAC, the 27th fractional bit onward is truncated when converted from the product with precision Q7.29 to the internal sum precision of Q6.26. Assuming that the 27th bit equally probable to be 0 or 1, then the 27th bit is 1 approximately half of the time. This means that the average truncation per MAC is $0.5 \times 2^{-27} = 2^{-28}$.

Given the 784 MACs of layer FC0, the expected truncation error is $784 \times 2^{-28} = 2.92 \times 10^{-6}$. The final truncation error from Q6.26 to Q6.12 truncates from the 13th fractional bit, resulting in an expected truncation error of $2^{-14} = 6.1 \times 10^{-5}$. Note that worst-case error is simply double the expected error, since it would assume that a 1-bit is truncated every time. Thus, truncation error from internal summation is expected to be more than 20 times less than the truncation to the 18-bit net output. Therefore, truncation error is successfully minimized during the internal summation of net computation during the forward pass.

The kernel is parameterized to support reuse in all the layers. The kernel has 2 parameters that are specified upon instantiation: neuron fan-in and the amount of bits needed to represent a neuron ID in the layer.

Weight Updates During the weight update phase, scaled weight gradients are added to the corresponding weight. This phase is implemented by iterating over the weight BRAM and weight gradient BRAM in the fully-connected layers. Scaling down the gradient by a learning rate is accomplished by using right bitshifts, which allows for efficient computation without any real sacrifice in training accuracy, as learning rates are generally arbitrary; frequently chosen as some power of 10.

One update phase takes two cycles. In the first cycle, the weights and their corresponding gradients for an address are read from the BRAMs. In the second cycle, the gradient is scaled down using bitshifting and added to the weight. The result is then written to the weight BRAM in the same cycle. The address will then be incremented and the process continues until all weights have been updated. The control logic is implemented by simply using a counter. The address to the BRAM contains all the bits except the bottom bit, so the address is incremented once every 2 cycles. The bottom bit of the counter is used to indicate the update phase and determine read and write enables on the BRAMs.

4.7.1.1 Individual Fully-Connected Layer Implementation

While the scheduler and kernels are the same across fully-connected layers, the weight BRAM specifications are not, since number of neurons, kernels and fan-in are different for each layer. For this reason, all fully-connected layers needed to have separate files defining them. If the weights and gradients were loaded and stored to from DRAM instead of BRAM, then the fully-connected layer could be parameterized.

Layer	# Kernels
FC0	196
FC1	16
FC2	2
Total	214

Table 4.2: Kernel allocation for the fully-connected layers in this implementation

Kernels Per Layer Given that the layers all have different amounts of MACs, the amount of computational kernels to allocate to each layer should be balanced to roughly even out the amount of cycles the computational phases require. There are 220 DSP slices available on the FPGA, and each kernel uses 1 DSP slice. In this design, 215 kernels were instantiated and the distribution is shown in table 4.2. The mathematical reasoning for this allocation is discussed in Chapter 7.

Weight BRAM Initialization The BRAMs cannot be initialized with all 0 values as that devolve to being a linear classifier since all weights would have the same gradients, as explained in Chapter 2. Therefore, the weight BRAMs have been pre-initialized with values generated using He Initialization [HZRS15]. The BRAMs are configured using Xilinx coefficient (COE) files. The initialization is performed using floating point, converted to Q1.17 binary format, and then written to a file in COE format using a python script. This script, `weight_coeff.py` may be found in Appendix

appendix for this

or in the *misc* folder of the github repository.

BRAM Structure The memory storage and throughput requirements differed between the layers. As such, the weight and gradient BRAMs are organized differently. All the BRAMs use the true-dual port RAM configuration of the Xilinx Block Memory Generator IP Core, version 8.4. Each of the layers run be able to read 1 weight per kernel per cycle during computational steps to prevent kernels from idling. Note that the weight and gradient BRAMs are organized in the same way.

The FC0 layer has 196 kernels and 98 neurons, thus 196 weights need to be read per cycle. This is accomplished by having two ports of width 98 weights

Address	Word Content
0	$w_{(0,0)}w_{(0,1)}\cdots w_{(0,96)}w_{(0,97)}$
1	$w_{(1,0)}w_{(1,1)}\cdots w_{(1,96)}w_{(1,97)}$
2	$w_{(2,0)}w_{(2,1)}\cdots w_{(2,96)}w_{(2,97)}$
...	...
782	$w_{(782,0)}w_{(782,1)}\cdots w_{(782,96)}w_{(782,97)}$
783	$w_{(783,0)}w_{(783,1)}\cdots w_{(783,96)}w_{(783,97)}$

Table 4.3: FC0 BRAM layout

wide. A weight is 18 bits, so the word length for each port is 1,764 bits wide. Furthermore, since the fan-in of each neuron is 784 (28×28), there are 784 words in this BRAM. In total, the FC0 layer requires 49 36K BRAMs for the weights and the gradients each, or 98 total. The BRAM layout is shown in table 4.3. The format for the weights listed in the word content is $w_{(i,j)}$, which meaning weight i of neuron j.

The FC1 layer has 16 kernels and 64 neurons. 16 weights must be read each cycle to supply the neurons, so two ports of width 8 words are used. This means that the bitwidth of each port is 144 bits. The fan-in of each of the 64 neurons is 98, so there are 784 ($\frac{64\times98}{8}$) words in each BRAM for the weights and gradients. This results in needing eight 36K BRAMs for the FC1 layer. The first 98 words contain the weights for neurons 0-7. The subsequent 98 weights contain the weights for neurons 8-15. This continues through the entire contents of the BRAM, concluding with the last 98 words containing the weights for neurons 56-63. Since not every neuron is represented in every word, the memory layout is slightly different and shown in table 4.4.

The FC2 layer is the smallest fully-connected layer in this hardware model, containing 10 neurons each with a fan-in of 64. There are 2 kernels, so each port on the BRAM has a word width equal to 1 weight, or 18 bits. The depth of the BRAM is 640, and can be entirely contained within one 36K BRAM. The layout is shown in table 4.5.

4.7.2 Interlayer Architecture

In this implemented hardware model, activations stored in interlayer buffers are stored directly in flipflops. This is because there are only 98 activations from FC0 to FC1 and 64 from FC1 to FC2. This results in 162 18-bit activations being stored in interlayer activation buffers, far within the resource limitations of the FPGA. The interlayer activation buffer module is also parameterized, so

Address	Word Content
0	$w_{(0,0)} w_{(0,1)} \cdots w_{(0,6)} w_{(0,7)}$
1	$w_{(1,0)} w_{(1,1)} \cdots w_{(1,6)} w_{(1,7)}$
...	...
97	$w_{(97,0)} w_{(97,1)} \cdots w_{(97,6)} w_{(97,7)}$
98	$w_{(0,8)} w_{(0,9)} \cdots w_{(0,14)} w_{(0,15)}$
99	$w_{(1,8)} w_{(1,9)} \cdots w_{(1,14)} w_{(1,15)}$
...	...
195	$w_{(97,8)} w_{(97,9)} \cdots w_{(97,14)} w_{(97,15)}$
196	$w_{(0,16)} w_{(0,17)} \cdots w_{(0,22)} w_{(0,23)}$
197	$w_{(1,16)} w_{(1,17)} \cdots w_{(1,22)} w_{(1,23)}$
...	...
293	$w_{(97,16)} w_{(97,17)} \cdots w_{(97,22)} w_{(97,23)}$
...	...
686	$w_{(0,56)} w_{(0,57)} \cdots w_{(0,62)} w_{(0,63)}$
687	$w_{(1,56)} w_{(1,57)} \cdots w_{(1,62)} w_{(1,63)}$
...	...
783	$w_{(97,56)} w_{(97,57)} \cdots w_{(97,62)} w_{(97,63)}$

Table 4.4: FC1 BRAM layout

Address	Word Content
0	$w_{(0,0)}$
1	$w_{(0,1)}$
...	...
63	$w_{(0,63)}$
64	$w_{(1,0)}$
65	$w_{(1,1)}$
...	...
127	$w_{(1,63)}$
...	...
576	$w_{(9,0)}$
577	$w_{(9,1)}$
...	...
639	$w_{(9,63)}$

Table 4.5: FC2 BRAM layout

Parameter Name	Brief Description
N_KERNELS_I	The width of the input write port. This is equivalent to the number of kernels of the previous layer.
N_KERNELS_O	The width of the output read port. This is equivalent to the number of kernels in the next layer after the buffer.
ID_WIDTH	The amount of bits needed to represent a neuron of the previous layer.
BUFF_SIZE	The amount of entries in the buffer.
LOOPS	Amount of times the buffer needs to be looped through for the next layer after the buffer to finish its computation.

Table 4.6: Parameters required for instantiation of the interlayer activation buffer.

Figure 4.4: The interlayer buffer

both buffers use the same SystemVerilog file. There are 5 parameters to be provided upon instantiation of the module, shown in table 4.6.

The architecture of the interlayer activation buffer is shown in figure 4.4. There is one write port which has a write width of N_KERNELS_I words. There are two read ports. The top one has a read width of N_KERNELS_O words and is used during the forward pass. The bottom one has a read widht of 1 word and is used during the backward pass, particularly during the weight-gradient calculation phase.

4.7.3 Softmax Layer

To have training be feasible, a proper loss function was required to calculate initial gradients for the output neurons. As such, cross entropy loss, one of the most popular loss functions in deep learning was chosen for this network. Cross-entropy loss is a statistical loss that uses probabilities as input, as shown below.

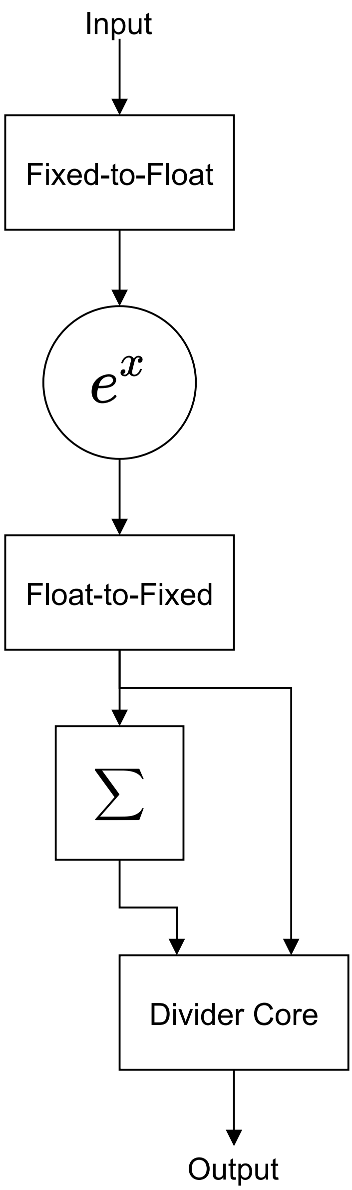

Figure 4.5: Architecture of the softmax layer

$$\mathcal{L}(y) = -\sum_{i=1}^{C} y_{i,c} \log(p_{i,c}) \tag{4.1}$$

In this equation,

Consider against the cross entropy loss defined in Chapter 3, define it in chapter 2 IMO

As explained in Chapter 2, the softmax functions converts logits to probabilities in following manner

4.8 PS-FPGA Communication

ps master fpga slave

4.8.1 Memory Map Layout

The memory map may be extended easily by adding or modifying address definitions to the AXI slave connected to the PS in the FPGA. In the PS code, one only need modify the `ddr_data` struct definition found in the C files of the PS source code. The memory map layout is defined in table 4.7. All addresses have a base address of 0x40000000. Note that values with addresses starting from 0x0 to 0x18 are registers written to by the FPGA. Values with addresses starting from 0x1C until 0x343 are written to by the PS. Output data from the FPGA is provided in the address range of 0x344 – 0x358.

4.9 Project Structure

PS – FPGA Memory Map

Offset	**Name**	**Brief Description**
0x0	fpga_img_id	The ID of the image that the FPGA is currently processing.
0x4	epoch	The current epoch in the FPGA
0x8	num_correct_train	The amount of correctly classified training images during the current epoch.
0xC	num_correct_test	The amount of correct classified test images during the current epoch.
0x10	idle_cycles	The amount of idle cycles in the FPGA since the start signal was received from the PS. An idle cycle is one in which none of the layers are performing any form of computation.
0x14	active_cycles	The amount of active cycles in the FPGA since the start signal was received from the PS. An active cycle is one in which at least one of the layers is performing computations.
0x18	status	A 32-bit register with many different flags from the FPGA, such as the layer states, for example.
0x1C	start	The start signal for training.
0x20	n_epochs	The number of epochs to train for in the FPGA
0x24	learning_rate	Value set to specify the amount of right shifts weight gradient should incur before updating a weight.
0x28	training_mode	Specifies whether the backward pass should be performed or not during computation.
0x2C	img_set_size	The size of the image set used during computation.
0x30	img_label	The label of the current image being computed.
0x34 - 0x343	img	Image data for the FPGA.
0x344 - 0x358	output	Output data from the last layer in the FPGA, before the softmax function is performed, so they are still logits in this case.

Table 4.7: Current memory map for communication between the PS and the FPGA.

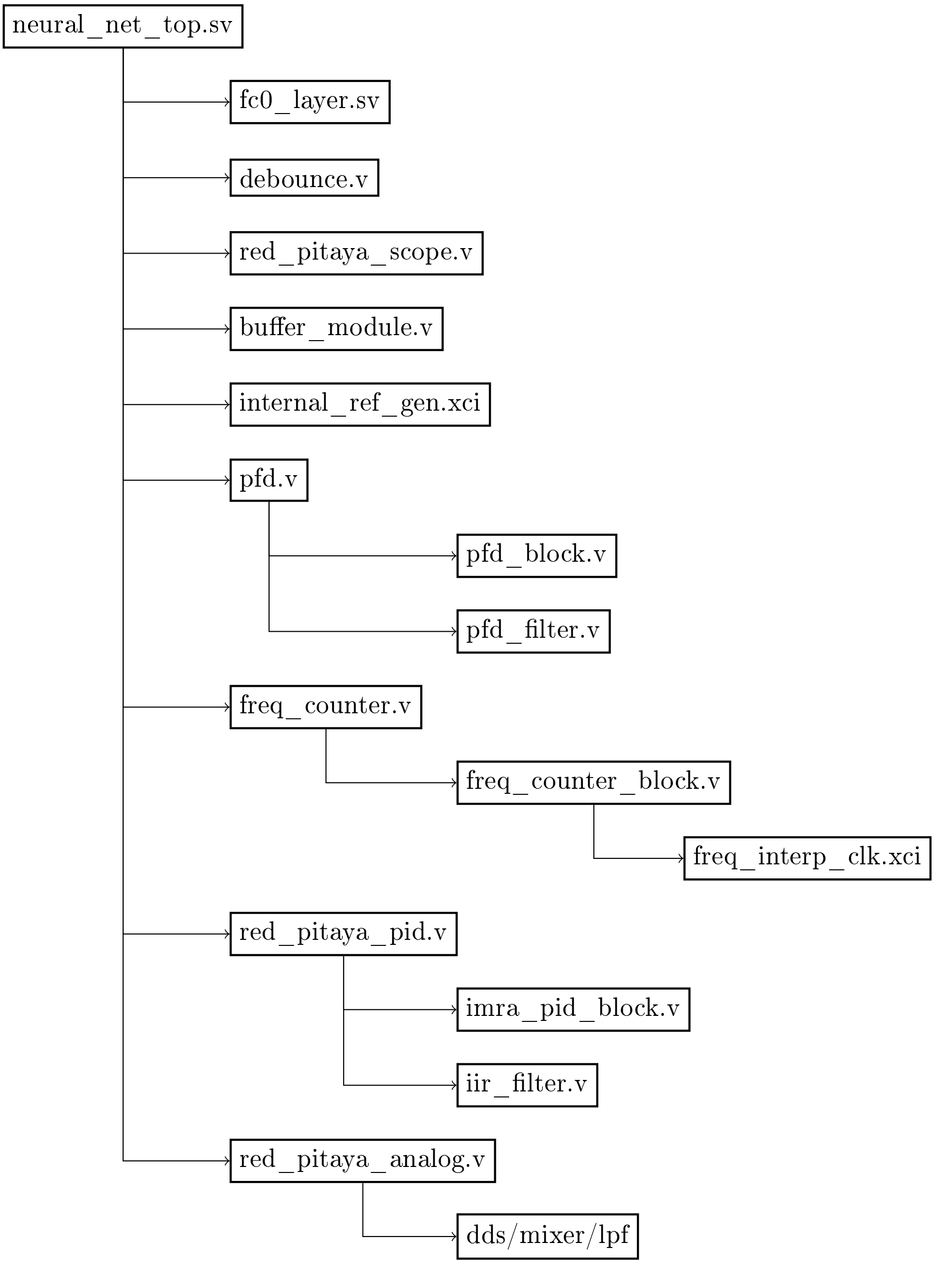

Figure 4.6: Hierarchy of the FPGA code used for the implementation of the network.

Chapter 5

Hardware Model Testing and Verification

5.1 Simulation

5.2 MMIO

Chapter 6

Results

6.1 Performance

6.2 clock freq

6.3 Fw/bw pass

6.4 Power

6.5 Resource Usage

CHAPTER 7

Analysis

7.1 Basis for Performance

7.2 Cycle Timing

7.3 Training learn rate precision vs. batch size optimal lr

Chapter 8

Discussion

8.1 Future Work

Chapter 9

Conclusion

Appendix A

Stuff

```python
from random import seed
from random import gauss
import math

n_neurons = 64
params_per_neuron = 98
r_width = 8
fan_in = 98

def intToBinaryString(x, l):
    str = ""
    neg = False
    if x < 0:
        x += 2 ** (l - 1)
        neg = True


    while x:
        if int(x) & 1:
            str = "1" + str
        else:
            str = "0" + str
        x = int(x / 2)

    if neg:
```

```
        str = "1" + str
    while (len(str) != 1):
        str = "0" + str
    return str

params = []
binary_params = []

for i in range(n_neurons):
    for j in range(params_per_neuron):
        param = gauss(0, math.sqrt(2/(fan_in - 1)))
        params.append(param)

for p in params:
    b = int(p * (2**17))
    binary_params.append(intToBinaryString(b, 18))
print(params[0])
print(binary_params[0])

contents = "memory_initialization_radix=2;\
    nmemory_initialization_vector=\n"
cnt = 0
for b in binary_params:
    contents += str(b)
    cnt += 1
    if cnt == r_width:
        contents += ",\n"
        cnt = 0
contents = contents[:-2] + ";"


f = open('output.coe', 'w')
f.write(contents)
f.close()
```

Appendix B

Stuff2

Bibliography

[cen01] ARM Info center. Arm developer suite axd and armsd debuggers guide, 2001. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0066d/CHDFAAEI.html`.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[gma15] The mail you want, not the spam you don't, Jul 2015.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[Kar] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/neural-networks-3`.

[LSSK19] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *CoRR*, abs/1903.06733, 2019.

[MPA+16] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. pages 14–26, 03 2016.

[sm-] The softmax function and its derivative. `https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/`.

[TYRW14] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. 09 2014.

[XHQ+16] Yingce Xia, Di He, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. *CoRR*, abs/1611.00179, 2016.

[ZFL+16] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. F-cnn: An fpga-based framework for training convolutional neural networks. *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 107–114, 2016.