Hardware Accelerator for the Training of Neural Networks

James Erik Groving Meade
s173476

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

This thesis proposes a novel hardware architecture to accelerate the training of neural networks with small batch sizes. The accelerator uses a modular, parameterizable and computationally well-balanced design to successfully an implement high-performance online training of neural networks. By using parallelism at the neuron level, the accelerator was able to achieve a speedup of 17.3 against a PyTorch CPU implementation of a specific neural network architecture. The accelerator also performs nearly as fast as a PyTorch GPU implentation of the network that used a batch size of 50 during training.

This thesis also highlights the importance of high-precision calculation for training. The highest accuracy attained by the accelerator on the MNIST dataset was 85.845%, which is a result of 18-bit fixed point precision being unable to successfully converge to a local optima due to the accumulation of precision error causing the degradation of training accuracy after a few epochs.

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring an M.Sc. in Computer Science and Engineering.

This thesis deals with the design of a hardware accelerator for the training of neural networks. Low-level design is one of my greatest passions and thus it has been an absolute privilege to have been given the opportunity to combine hardware with the surging field of machine learning.

Lyngby, 28-June-2019

James Erik Groving Meade
s173476

Acknowledgements

First and foremost, I would like to sincerely thank my advisor Jens Sparsø for his time and guidance throughout the entire duration of my thesis. Our regular meetings helped me to stay grounded and to think critically about my project.

I would also like to thank my former classmate, Cheng Fu, who is currently a PhD student at the University of California, San Diego. My conversations with him at the beginning of my foray into this thesis helped me establish my footing.

GitHub Repository Organization

As the scope of this thesis was fairly wide, several different technologies and programming languages were used in the project. As such, this section was added to help guide the reader around the organization of the GitHub repository [1]. The organization of the GitHub repository is present on the homepage of the repository. It may also be read in textform by viewing the README.md file.

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

Contents

Chapter 1

Introduction

Neural networks have seen a surge in popularity ever since a neural network, coined AlexNet, decisively won the ImageNet Large Scale Visual Recognition Challenge in 2012, achieving 10.8% higher accuracy then the next best solution [KSH12]. It was the only neural network entry in the entire competition; and the victory stunned much of the academic world.

This moment has been generally regarded as the spark that ignited the massive surge in academic interest toward neural networks and statistical machine learning. In the 7 years since, research regarding neural networks has yet to slow down as real-world applications of neural networks continue to be found. To name a few, neural networks are currently in use for facial recognition at Facebook [TYRW14], translation for Microsoft [XHQ+16], spam filters for Google's Gmail [gma15] and endless more.

In order for these neural networks to have such stellar accuracy on tasks such as image classification, they must first learn from labeled data in a process known as training. The neural network training process has an incredibly high level of inherent parallelism, and thus GPUs have emerged as the device of choice to train neural networks. GPU-based training takes advantage of data-level parallelism to train networks by assigning individual inputs in a training batch to different cores, which then all perform the same computations on different data. This coarsely-grained approach to training works well for large batch sizes. However, since these GPU models use data parallelism for speedup, GPU model training performance degrades for small batch sizes and are even slower than CPU models when it comes to online learning. Online learning is when a single labeled data sample is fed to the network at a time, or in other words, when the batch size is equal to 1 [MR17].

Today's solutions for training neural networks with small batch sizes do not take advantage of the fine-grain parallelism available in neural networks. This thesis presents a hardware accelerator that uses fine-grain parallelism at the neuron level to achieve high training performance. The accelerator results in much faster performance compared to the current solutions available for training neural networks with small batch sizes.

This thesis proposes a novel hardware architecture for the training of neural networks. While the focus of the thesis is the architectural design of the hardware accelerator, a basic understanding of neural networks is helpful. As such, Chapter 2 reviews the basics of neural networks and surveys related work on designing hardware to optimize for neural network computation. Chapter 3 describes the software model that was implemented to verify and further understand the algorithms to be used for the hardware model. Chapter 4 covers the hardware model and implementation of the accelerator. Next, Chapter 5 documents the testing methods used to functionally verify the hardware model. Chapter 6 presents the results of the thesis and Chapter 7 provides analysis of these results. Chapter 8 discusses the project as a whole and what future work could be done to improve the project. Finally, Chapter 9 presents the conclusion of the thesis.

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,

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 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

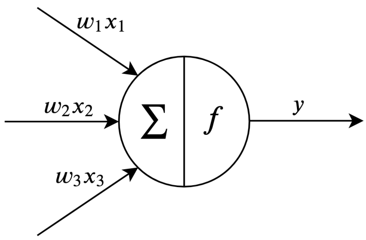

Figure 2.1: A neuron with 3 inputs; bias term omitted for simplicity.

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

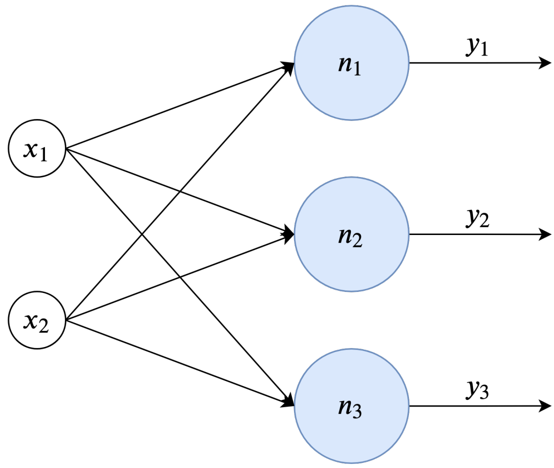

Figure 2.2: A fully-connected layer with 3 neurons, each receiving an input vector of size 2 from the input layer.

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.

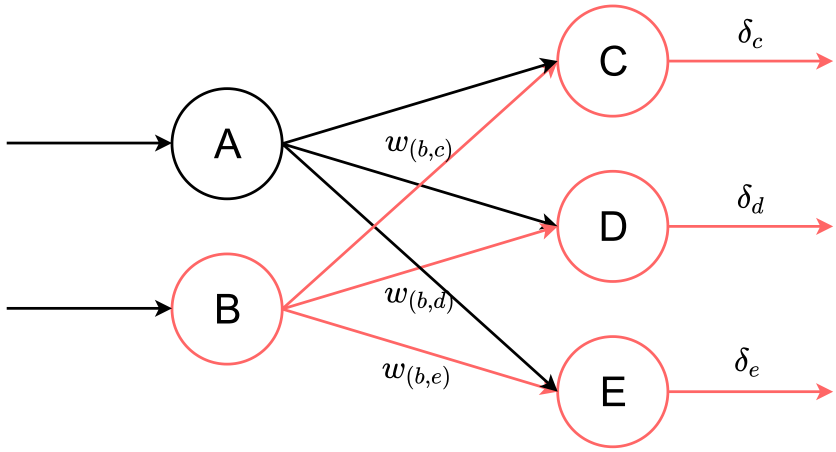

Figure 2.3: Example of backpropagating the loss gradient to the previous layer. Values used in backpropagating the loss to neuron B shown in red.

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 $\frac{\delta \mathcal{L}}{\delta \mathrm{net}}$, 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.

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

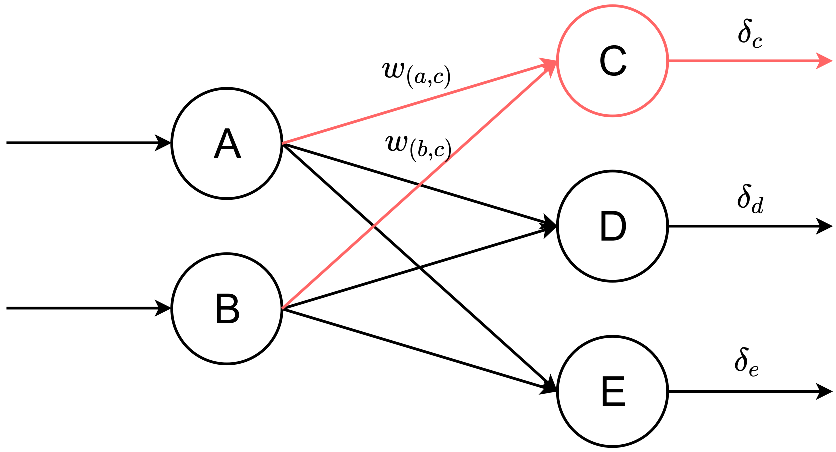

Figure 2.4: Example of computing weight gradients. Relevant values shown in red.

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:

$$\begin{aligned} m &\quad \text{—} \quad \text{the momentum parameter} \\ v &\quad \text{—} \quad \text{`velocity'} \\ \eta &\quad \text{—} \quad \text{the learning rate} \\ dw &\quad \text{—} \quad \text{the loss gradient for some weight or bias } w. \end{aligned}$$

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 [AAB+15], and PyTorch which is developed by Facebook [PGC+17]. Keras is another popular framework that has introduced the most popular syntax style for describing neural networks [C+15]. Caffe was one of the earliest public frameworks available and remains in widespread use today [JSD+14]. These frameworks are generally relatively simple to use and deliver 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.

Chapter 3

Software Model

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 Appendix F as well as online on GitHub in the SWModel folder.

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.

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.

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.

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()`.

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.

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.

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 $\frac{2}{\text{fan\_in}}$.

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.

```
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%.

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 Appendix F.

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 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

```
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.

(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

```
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

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.

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 [1]. 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
```

[1] https://github.com/google/googletest

```
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.

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

```
> ./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. The source code can be found in the *FPGA* folder of the GitHub repository or in Appendix B.

4.1 Hardware Setup

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 online training (training using 1 labeled data item at a time, a batch size of 1); offline training using larger batches is not supported by this hardware model.

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.

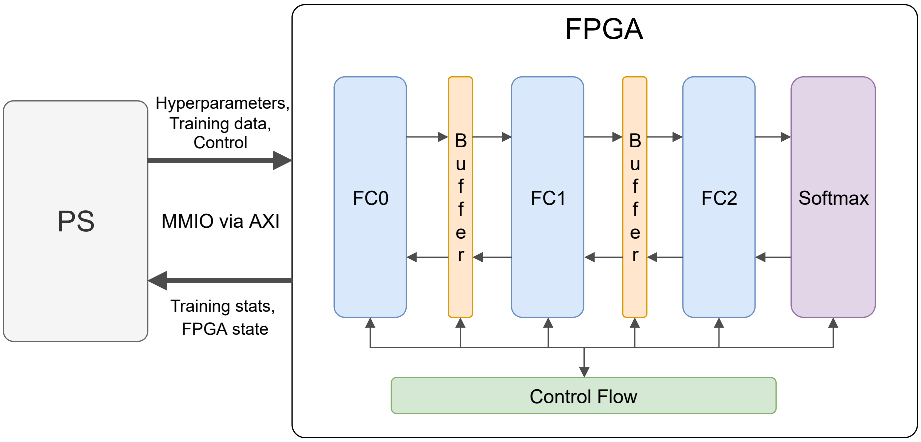

Figure 4.1: Architecture of the hardware accelerator

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 4.8.4.

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.

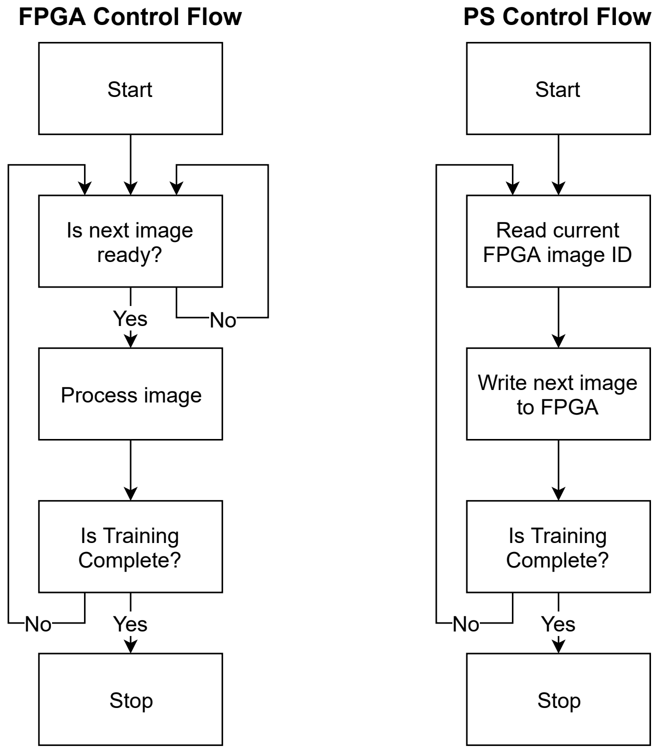

Figure 4.2: The high-level control flow of the training process

4.5 Training Process

The training process begins with the PS writing a 1 to the `start` register. This signals to the FPGA to start training whenever data becomes available. From the FPGA side, if the image in MMIO has ID equal to the FPGA's current image ID plus 1 (modulo image set size), then the image is ready. Otherwise, the FPGA will wait for the next image to be written. This process is summed up in Figure 4.2. The FPGA loops back around to 0 once the image set size is reached. The image set size is assigned via MMIO.

When the FPGA has processed the last image in the set during an epoch, the epoch counter is incremented and training stats will be available for the PS to read. If the epoch counter has reached the set number of training epochs, then training will stop, otherwise, the next training epoch will begin.

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 [Xila]. 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 B, 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 4.3. As DSP slices are limited, 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

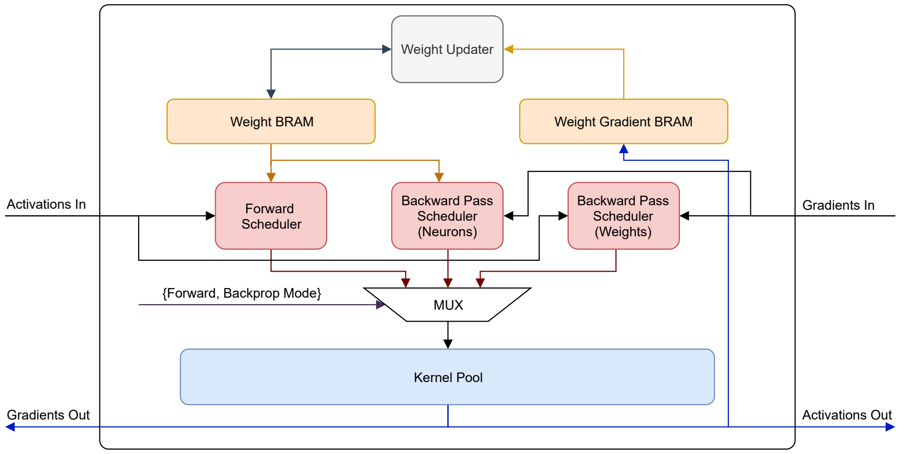

Figure 4.3: Architecture of the fully connected layer

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 4.4. 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. During both forward and backward passes, a kernel works on a single specific neuron until computation for that neuron has been completed, after which if there is still more work to do, will become computing for another neuron.

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 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

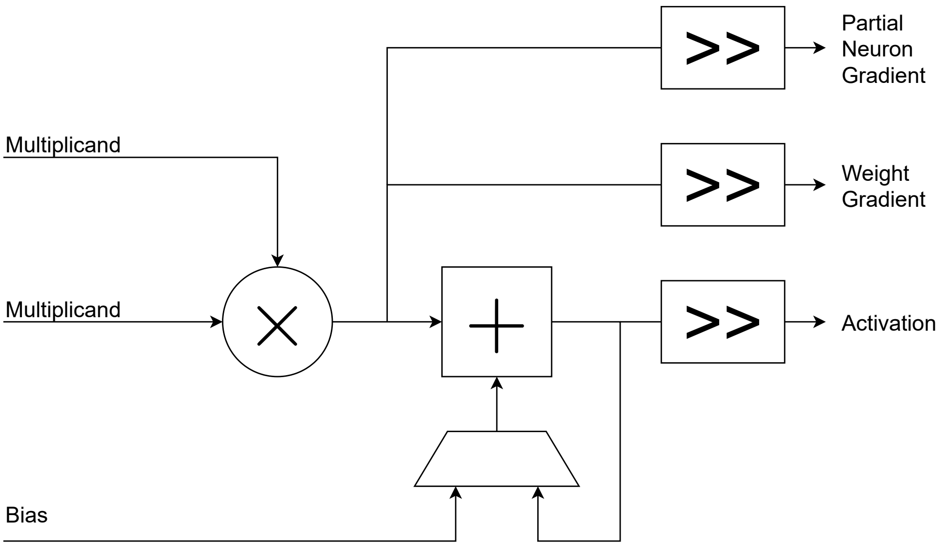

Figure 4.4: Architecture of the kernel, saturation checking not shown.

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.

Backpropagation Priority When the backward pass is in progress and the input neuron gradient's to a fully-connected layer become valid, there are two tasks that become ready to be performed. The first task is to use the valid input gradients to backpropagate neuron gradients to the previous layer. The second task is to calculate the weight gradients for the current layer. In this case, backpropagating the neuron gradients is given priority, because that way, once the gradients are backpropagated, the previous layer can also start performing its backward pass. Note that for the first layer, it is not possible to further backpropagate to previous neurons, so when the first layer receives its valid gradients, then it simply calculated its own weight gradients and then finishes.

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 E 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 layer must 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 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.

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

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 ($\frac{64 \times 98}{8}$) 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 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 4.5. There

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 4.5: The interlayer activation buffer

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 width of 1 word and is used during the backward pass, particularly during the weight-gradient calculation phase.

4.7.3 Softmax Layer

To implement training of the neural network, meaningful gradients needed to be calculated for the output layer neurons. Cross-entropy loss, one of the most popular loss functions in deep learning, was chosen for this network. As such, the softmax function (also described in Chapter 2) needed to be implemented. The softmax function is shown again in equation 4.1 for convenience.

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\Sigma_{j=1}^{C} e^{x_j}} \tag{4.1}$$

The dataflow of the softmax layer is shown in figure 4.6. The softmax layer has 10 inputs in this network for MNIST digit classification. This layer is fully pipelined, so while there is a relatively long latency, the computation can still be performed quickly.

The implemented softmax function is also referred to as a numerically stable softmax. By subtracting a constant from the exponents, the final probabilities will not be affected. This is why the first step of the softmax layer is to subtract the maximum value input from all inputs. This then results in low, stable exponents being fed to the exponential function.

Since there is no support for the exponential function using fixed-point inputs in the Xilinx IP core repository, logits are first converted to 32-bit floating point numbers. After this, the exponential function for each input is calculated. The e^x core uses 1 DSP slice. The exponential function output is then converted from floating point back to fixed point. At this point, e^x is known for all the inputs, so all the numerators required for $\sigma(\mathbf{x})_i$ are known. To calculate the denominator, these values must also be summed up, so this occurs in the next stage of the layer. Finally, the numerators are divided by this denominator to finish the softmax process of converting the outputs from logits to probabilities.

4.8 PS – FPGA Communication

The processing system and the FPGA communicate via an AXI4 bus. In this AXI bus, the PS is the master and the FPGA is the slave.

4.8.1 AXI Implementation for the PS

From the PS side, communication is performed as shown by the highlighted red line in figure 4.7. The AXI communication base address is 0x40000000 and spans until 0x7FFFFFFF. Having done this, by mapping a pointer to this location on */dev/mem*, data that is written to or read from addresses within this region of memory will invoke an AXI bus transaction. This was set up by adding the *Zynq7 Processing System Version 5.5* IP core to a block diagram in Vivado, and defining an address range in the address editor as shown in Figure 4.8.

Once the address range and Zynq IP core has been added to the block diagram, C code to run on the PS must be written. There are only 2 things that need

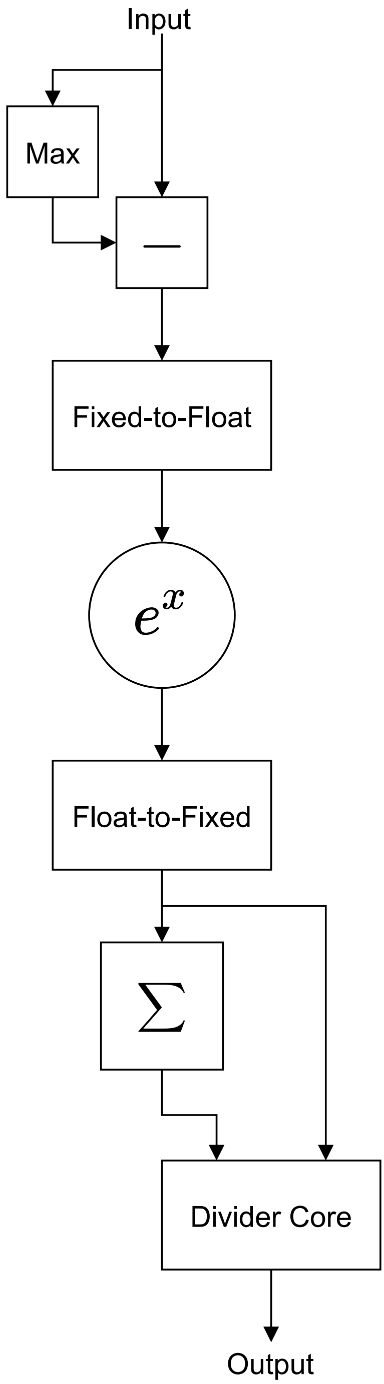

Figure 4.6: Architecture of the softmax layer

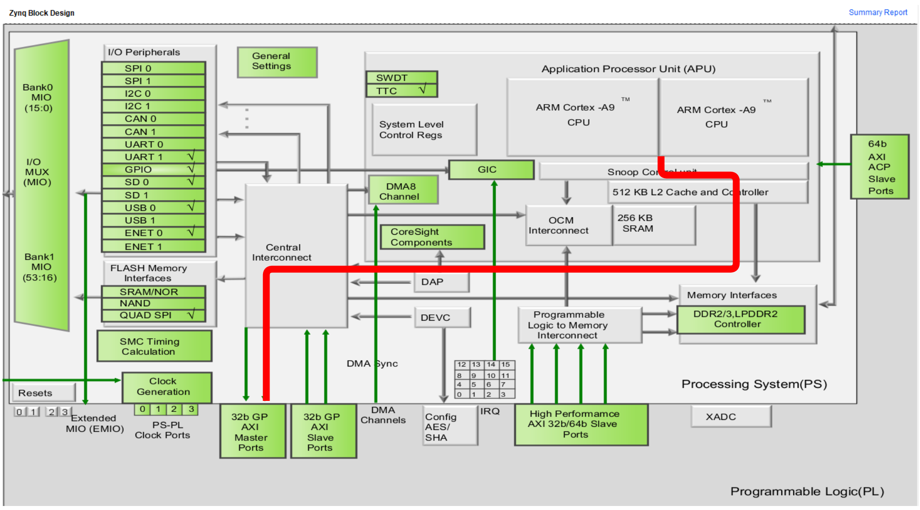

Figure 4.7: Communication from the PS to the FPGA

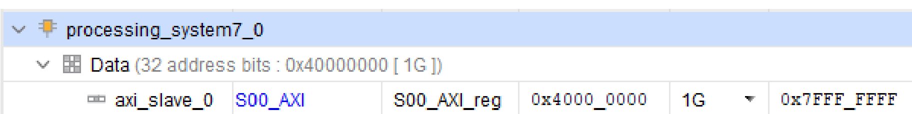

Figure 4.8: Specifying the Address Range for the AXI Bus

to be done to be able to start performing AXI bus transactions. The first step is to create a file handle by opening */dev/mem* with the proper flags set. Then one only needs to memory map a pointer to this region. In the example shown in Listing 4.2, the pointer is of type volatile, because the order of reads and writes is critical during the training process when images are transferred to the FPGA.

```c
int handle = open("/dev/mem", O_RDWR | O_SYNC);
volatile ddr_data_t* ddr_ptr = mmap(NULL, 134217728,
    PROT_READ | PROT_WRITE, MAP_SHARED, handle, 0x40000000);
```

Listing 4.2: Code to allow the PS to perform AXI transactions with the FPGA

4.8.2 Running Programs from the PS

Several programs have been created for this project. Currently on the SD-Card, there are 3 programs: `train`, `train_small`, and `inference_only`. The source code, written in C, for these programs can be found in Appendix C or in the *ps_code* folder of the project repository. The `train` program runs full-scale training on the entire MNIST dataset, splitting the data into training and testing sets of size according to a user-define. The `train_small` program uses a very small subset of the MNIST dataset to demonstrate the model's ability to have the network successfully learn an entire dataset. Finally, `inference_only` performs inference on every image in the MNIST dataset. A portion of terminal output from 5 epochs of training when running the `train` program is shown in Listing 4.3.

```
@@@ Loading MNIST images...
@@@ Loading complete!

@@@ EPOCH 1
@@@ Training Images: 2950/5000
Accuracy: 58.999997%
@@@Test Images: 43120/65000
Accuracy: 66.338462%
Active Cycles: 89107985  Idle Cycles: 228978233
Active Cycle Percentage: 28.013784%
Elapsed time: 6.36157 seconds

@@@ EPOCH 2
@@@ Training Images: 3623/5000
Accuracy: 72.460002%
@@@Test Images: 45904/65000
Accuracy: 70.621538%
```

```
Active Cycles: 178215969         Idle Cycles: 458029353
Active Cycle Percentage: 28.010575%
Elapsed time: 6.36297 seconds

@@@ EPOCH 3
@@@ Training Images: 4003/5000
Accuracy: 80.059999%
@@@Test Images: 45345/65000
Accuracy: 69.761539%
Active Cycles: 267323953         Idle Cycles: 687061998
Active Cycle Percentage: 28.010046%
Elapsed time: 6.36260 seconds

@@@ EPOCH 4
@@@ Training Images: 4172/5000
Accuracy: 83.440000%
@@@Test Images: 55452/65000
Accuracy: 85.310769%
Active Cycles: 356431937         Idle Cycles: 916126394
Active Cycle Percentage: 28.009084%
Elapsed time: 6.36324 seconds

@@@ EPOCH 5
@@@ Training Images: 4260/5000
Accuracy: 85.200000%
@@@Test Images: 53002/65000
Accuracy: 81.541538%
Active Cycles: 445539921         Idle Cycles: 1145184941
Active Cycle Percentage: 28.008611%
Elapsed time: 6.36312 seconds
```

Listing 4.3: Training output from the `train` program

4.8.3 AXI Implementation for the FPGA

To implement the FPGA side of AXI commmunication, a block diagram to interface with the *Zynq7 Processing System* was created. Next, a custom IP core was created using Vivado's base AXI4 slave and then customized to meet the design needs of the project. The block diagram was then completed as shown in Figure 4.9. As can be seen, the Zynq's master AXI output is connected to an AXI interconnect which is then connected to the AXI slave port of the custom AXI module. Once the block diagram is complete, a wrapper for the block-diagram was generated and instantiated inside the top module of the design.

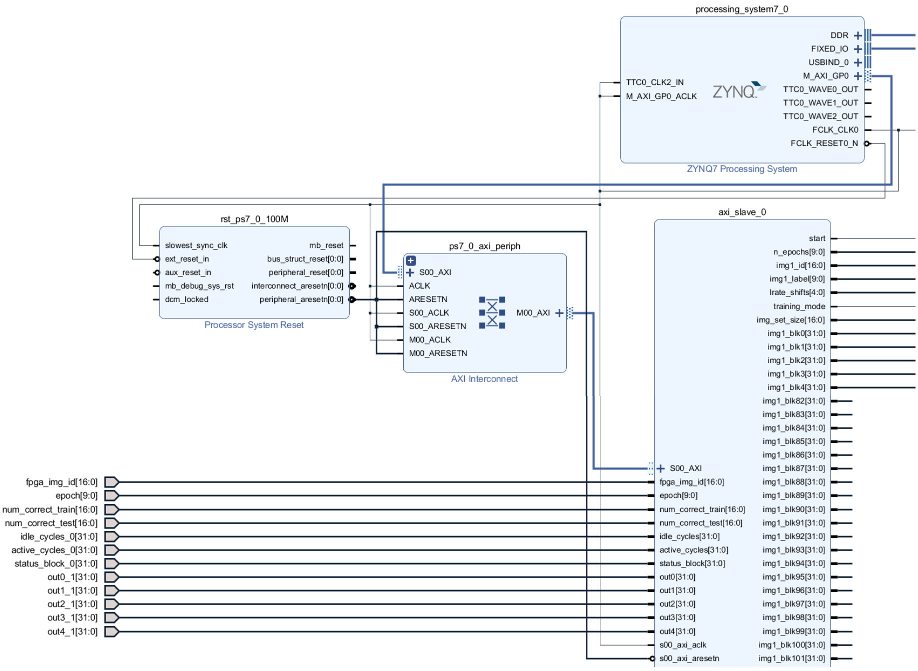

Figure 4.9: Block diagram to establish connection from the FPGA side to the AXI bus from the PS

The final part of the FPGA AXI implementation was to modify the generated AXI module. The module was generated in Verilog, so it differed slightly from the rest of the SystemVerilog project. It is also for this reason that there are 196 separate 32-bit registers to contain image data, since Verilog does not support 2D packed arrays as ports to modules. The Verilog code for implementing the memory map described in section 4.8.4 can be found on GitHub in the *ip_repo/axi_slave_1.0/hdl* folder.

4.8.4 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 PetaLinux

The boot image on the SD card has been modified to run Xilinx's PetaLinux. This was done by using the 2016.4 prebuilt PetaLinux image as a base from Xilinx's PetaLinux website [Xilb]. The image has been slightly modified by changing the */etc/init.d/rcS* file to have the PS acquire a certain IP address and to mount the SD card to the filesystem. The *BOOT.bin* for booting Petalinux on the PS is created by writing a first stage bootloader *elf* file created by the Xilinx SDK for the Vivado project, the bitstream generated by Vivado, and the *u-boot.elf* file from the 2016.4 PetaLinux image. The boot image is created by using Xilinx SDK's "Create Boot Image" utility. Aside from these changes to the 2016.4 SD card image, the other files in the image are not changed.

Cross-Compiling for PetaLinux Since the PS is an ARM-based processor, all C code for this project must be cross-compiled before it can be run on the PS. This is done by using the Linaro cross-compiling toolchain. A C file may be compiled to run on the PS using the below command.

```
> arm-linux-gnueabi-gcc <source_file> -o <executable>
```

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.

Chapter 5

Hardware Model Testing and Verification

Verification is a vital part of hardware design. For this project, all relevant functionality in the FPGA was verified by simulation.

5.1 Simulation

During FPGA development, four different testbenches were created to test the functionality of the design. The first three were module level testbenches to test the scheduler, fully-connected layer, and softmax layers. The fourth testbench tests the entire FPGA design, thus verification of the fourth testbench means that all modules are functional. Therefore, this section will discuss verification of the fourth testbench, which is a full test of the network: `neural_net_top_tb.sv`, found in the directory *FPGA/FPGA.srcs/sim_1/new* of the GitHub repository as well as in Appendix B.

5.1.1 Project Modifications to Simulate of the Design

To conduct the full-scale test of the network, an input needed to be provided to the network on which to perform training. This was done by using a BRAM to store a random input generated by the same Python script (`weight_coeff.py`) used to generate the weights for the weight BRAMS. When the simulation runs, the input to the network comes from this input BRAM rather than from the PS.

5.1.2 Testing Environment

The Vivado Simulator was used to perform simulation of the hardware. The testbench was run through Vivado's Tcl shell. During the testing process, a simulation would be ran and then diagnostic data in a test file could be The below commands run from the Vivado Tcl show how to open the project, run the testbench for 50000 ns, and have all output written to a file.

```
Vivado% open_proj FPGA.xpr
Scanning sources...
Finished scanning sources
open_project: Time (s): cpu = 00:00:11 ; elapsed = 00:00:13
    . Memory (MB): peak = 322.016 ; gain = 71.828
Vivado% launch_simulation > sim_out
Vivado% run 50000 >> sim_out
```

5.1.3 Simulation Output

Verification and debugging was simple through the use of informative simulation output files. The project formatted signal data to be easy to read through the use of `$display` statements. An example of this is shown in listing 5.1. This example is from `neural_net_top.sv` and prints the current cycle number and FC2 output and gradient data; `sf` and `sf2` are scaling factors for activations and gradients, respectively. These scaling factors allow the fixed-point Q format values to be displayed as their floating point equivalent. These types of display statements are ubiquitous in the modules of the project. Once verified, most of these display statements were commented out to prevent clutter of the simulation output file.

```
`ifdef DEBUG
integer clk_cycle;
```

```
integer it;

always_ff @(posedge clk) begin
  if (reset) begin
    clk_cycle   <= 0;
  end
  else begin
    clk_cycle   <=  clk_cycle + 1'b1;
  end
  $display("\n\n------ CYCLE %04d ------", clk_cycle);

  $display("---FC2 GRADIENTS---");
  $display("img_label: %d", img_label);
  for (it = 0; it < `FC2_NEURONS; it = it + 1) begin
    $display("%02d:\t%f", it,
      $itor($signed(fc2_gradients[it])) * sf2);
  end

  $display("--- FC2 OUT ---");
  $display("fc2_buf_valid: %01b" , fc2_buf_valid);
  for (it= 0; it < `FC2_NEURONS; it=it+1) begin
    $display("%02d: %f", it,
      $itor($signed(fc2_act_o_buf[it])) * sf);
  end
end
`endif
```

Listing 5.1: Example debug code for simulating the functionality of the hardware model

With this output redirected to a text file, signal data per cycle can be easily found. Furthermore, jumping to a previous or next cycle is quick. For example, in Vim, this can be done with by pressing ‘N’ or ‘n’, respectively. This method of debugging with Vim is shown in Figure 5.1, displaying the output generated from the code in Listing 5.1.

5.1.4 Correctness of Simulated Outputs

A Python script was written to aid in verification of the hardware simulation. This script, `fpga_forward_backward_pass_test.py`, is located in Appendix D as well as in the *misc* folder of the project on GitHub.

This script parses the Xilinx coefficient files for the input to the network, as well as for all the neurons in all the layers and converts them to floating point

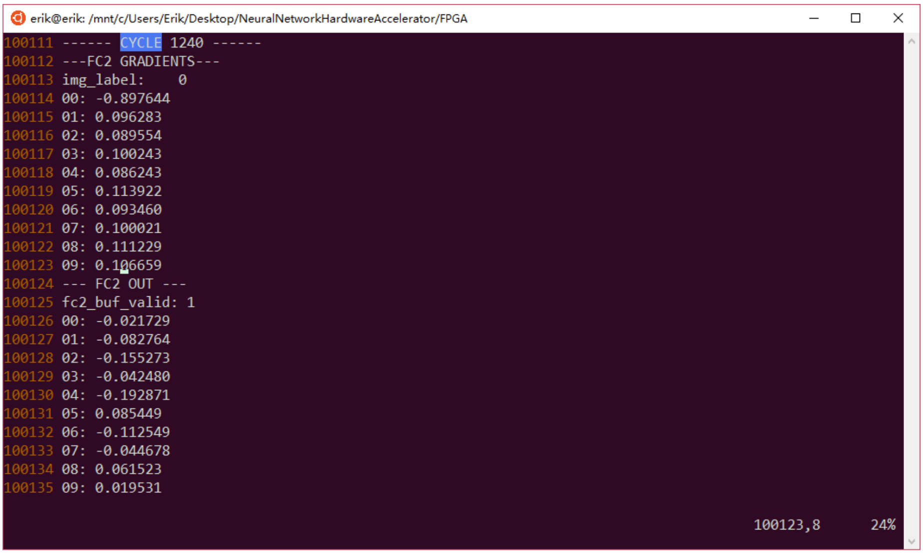

Figure 5.1: Jumping from cycle to cycle to view debugging data using Vim

numbers. The script then performs the forward pass using the parsed weights and prints the outputs. The script then computes the backward pass and also prints out all neuron and weight gradients. The hardware model can then be verified by checking that the outputs at each stage of computation align with the Python script. Note that values are not compared for equality, but for relative correctness, since the Python script uses floating point and the hardware model uses 18-bit fixed point.

Furthermore, to ensure that the script's computed outputs and gradients are correct, the script also implements gradient checking tests for itself. With this assurance, the script's computed output and gradients were successfully verified as correct and thus could be used as a baseline against which to compare the hardware model. Note that the gradient check testing for the testing script was based on the gradient checks implemented for the software model in Chapter 3, and example gradient check tests are shown in Listing 5.2.

```
> python3 fpga_forward_backward_pass_test.py
../FPGA/FPGA.srcs/sources_1/ip/fc0_weights_1.17.coe
../FPGA/FPGA.srcs/sources_1/ip/fc1_weights2_1.17.coe
../FPGA/FPGA.srcs/sources_1/ip/fc2_weights_1.17.coe
Calculated gradient:    -0.003531676695546401
Numerical gradient:     -0.003531676693313557

Calculated gradient:    -0.006374946805618298
```

```
Numerical gradient:     -0.0063749433798498956

Calculated gradient:    0.0006677415295515585
Numerical gradient:     0.0006677441533042838
```

Listing 5.2: Gradient checks for randomly chosen weights in the Python verification script that uses inputs and weights read from the Xilinx coefficient files of the BRAMs in the hardware model. Only three non-zero gradients shown for brevity.

Forward Pass Verification The Python script was used to verify the correctness of the forward pass of the FPGA layer by layer. Forward pass layer outputs for softmax layer from the simulation and script are compared side-by-side in Listing 5.3. Since the softmax output depends on the outputs from FC0, FC1, and FC2, these layer outputs are not shown for the sake of space. From these tests, the simulated forward pass outputs of the hardware model are shown to be correct. The full outputs for every layer may be seen in the *HW_ Verification* folder of the GitHub repository.

```
SIMULATION                      PYTHON SCRIPT
Neuron       Activation         Neuron       Activation
00           0.102348           0            0.10235213231346099
01           0.096283           1            0.09627338472902423
02           0.089554           2            0.08953177512141873
03           0.100243           3            0.1002532810000227
04           0.086243           4            0.08622264587243636
05           0.113922           5            0.11397991662882098
06           0.093460           6            0.09343919203092571
07           0.100021           7            0.10005157131620508
08           0.111229           8            0.11123918032286678
09           0.106659           9            0.10665692066481845
```

Listing 5.3: Softmax output. All 10 Neuron outputs shown.

Backward Pass Verification The backward pass was verified in the same way as the forward pass, though there are many more gradients than outputs. There is 1 gradient for each neuron and weight, totalling over 80,000 gradients. The forward pass outputs and backward pass gradients can be seen in their entirety in the *hardware_ verification* folder of the GitHub repository.

The gradients in the backward pass all stem from the output layer gradients which come from the softmax function. The steps for deriving the gradients of the output layer is a softmax based neural network are explained in more detail in Chapter 2, though the gradients are essentially the softmax output visible in Listing 5.3 except that the neuron representing the inputs class label is subtracted by 1.

Listing 5.4 shows randomly selected weight gradients from each of the fully-connected layers. The weight gradients depend on the neuron gradients, thus the neuron gradients for that layer must be correct for the weight gradients to be correct; because of this, only weight gradients are shown in the figure, though neuron gradients are also available for viewing in the *hardware_verification* folder. As can be seen, the gradients are calculated to relatively high accuracy. This level of accuracy is directly correlated to the fact that the gradients are all Q1.17, maximizing the amount of fractional bits. Note that the 1 integer bit is required to represent the output layer gradient (since the input class label neuron is subtracted by 1), so the radix cannot be moved any further.

```
SIMULATION                      PYTHON SCRIPT
FC0
Neuron   Weight   Gradient      Neuron   Weight   Gradient
09       593       0.000763     09       593       0.00076932259
19       711       0.006874     19       711       0.00688892029
37       412      -0.006149     37       412      -0.00613842723
57       128       0.000610     57       128       0.00061567956
74       485      -0.000282     74       485      -0.00027281649

FC1
Neuron   Weight   Gradient      Neuron   Weight   Gradient
02       051      -0.003815     02       051      -0.00380934976
19       097      -0.019463     19       097      -0.01948172921
24       035      -0.013214     24       035      -0.01325251269
37       094       0.016045     37       094       0.01610831241
51       030       0.016563     51       030       0.01659535729

FC2
Neuron   Weight   Gradient      Neuron   Weight   Gradient
01       043       0.015907     01       043       0.01595727359
03       002       0.016861     03       002       0.01688975169
04       057       0.005745     04       057       0.00578264697
08       023       0.024437     08       023       0.02451471064
09       024       0.000542     09       024       0.00055094484
```

Listing 5.4: 5 randomly selected weight gradients from each of the fully connected layers

Chapter 6

Results

6.1 Benchmarking Models and Structure

Some of the results in this chapter are based on evaluating the hardware model (HWM) against other models implementing the same neural network. The other models include my software model (SWM), PyTorch running on the CPU (PyCPU), and PyTorch running on the GPU (PyGPU). My software model performs the same computations as the hardware model, so this provides insight to speedup over CPU without computational optimizations. The PyTorch CPU and GPU models then compare my hardware accelerator against heavily-optimized neural network frameworks. A training epoch in the following experiments is defined as performing learning on the 60,000 training images of the MNIST dataset. Inference experiments measure time to perform inference on all 70,000 images in the MNIST dataset. All experiment data can be found in Appendix A.

6.2 Parallelism in GPU-based Training

Figure 6.1 shows PyGPU speedup for 1 training epoch over the 60,000 training images for varying batch sizes. As one might notice, speedup grows at approximately the same rate as batch size. This is a textbook display of data parallelism; where it is clear that the images in the batch are processed individually by separate CUDA cores. The slope is able to stay linear even at massive batch sizes because a larger batch size means fewer updates to the weights. This results in the overhead from a larger batch size being negated by the decrease in amount of weight updates. The amount of forward and backward passes, which can be completely parallelized, remains the same. The weight update, the serial portion of training, is only performed once per batch. This means that a batch size of 1 performs 5 times more weight updates than a batch size of 5, which performs 20 times more weight updates than a batch size of 100. Therefore it is for this reason to see such a speedup.

From the figures, it is clear that the GPU model takes advantage of data-level parallelism to achieve performance, as epoch time is a near linear function of batch size. As a result, since the GPU-based implementation uses a coarser form of parallelism compared to the HWM, it would be illogical to benchmark speedup against the GPU with a batch size of 1 since the GPU's parallelism depends on batch size. Therefore, in the following experiments, the PyGPU model has been benchmarked using a batch size of 50 unless otherwise specified. It should be noted that the PyGPU model also performs 49 fewer weight updates compared to the other models as a result of this. Moreover, each weight update on a GPU would require reductions of partial gradient results from the CUDA kernels, so this should be taken into consideration when observing the following performance benchmarks.

6.3 Evaluation Hardware

The hardware model is evaluated using a ZedBoard equipped with a Zynq-7000 XC7Z020 SoC. The SWM and PyCPU both run on a Intel Core i7-4720HQ CPU. The GPU is an Nvidia GeForce GTX 970M equipped with 6 GB of GDDR5 RAM.

6.4 Performance

One of the most important metrics for an accelerator is runtime performance. While this hardware model is primarily focused on training, experiments to determine performance for both training and inference modes have both been conducted and are shown in this section.

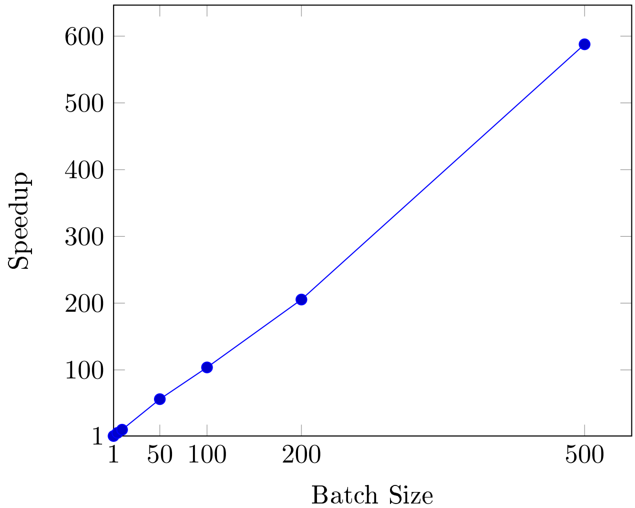

Figure 6.1: Speedup for 1 training epoch when training using different batch sizes for PyGPU. Speedup is calculated using the PyGPU model with a batch size of 1 as the reference.

6.4.1 Training

The average time for 1 training epoch has been recorded for each of the neural network models. The result is shown in Figure 6.2. This graph shows that the accelerator massively outperforms CPU models. Figure 6.3 shows the speedup of the models, using PyCPU as a baseline. Notably, the HWM achieves a speedup of of nearly equal to that of the PyGPU model.

6.4.2 Inference

Inference performance was also measured for each of the models. The result is shown in Figure 6.4. This graph shows that the accelerator also outperforms CPU models for inference, though falls short of the GPU model. Figure 6.5 shows the speedup of the models, using PyCPU as a baseline. The HWM achieves a speedup of 2.282 compared to the PyCPU model.

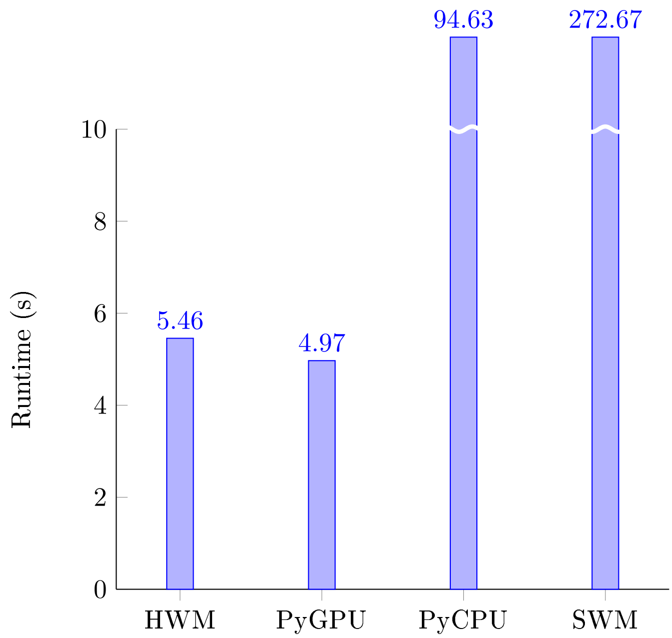

Figure 6.2: Training runtime for various network models

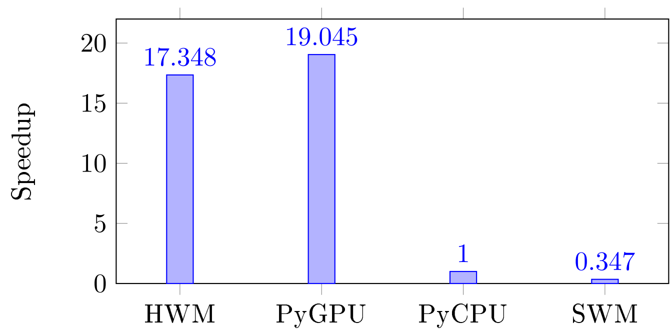

Figure 6.3: Training speedup using the PyCPU as a baseline

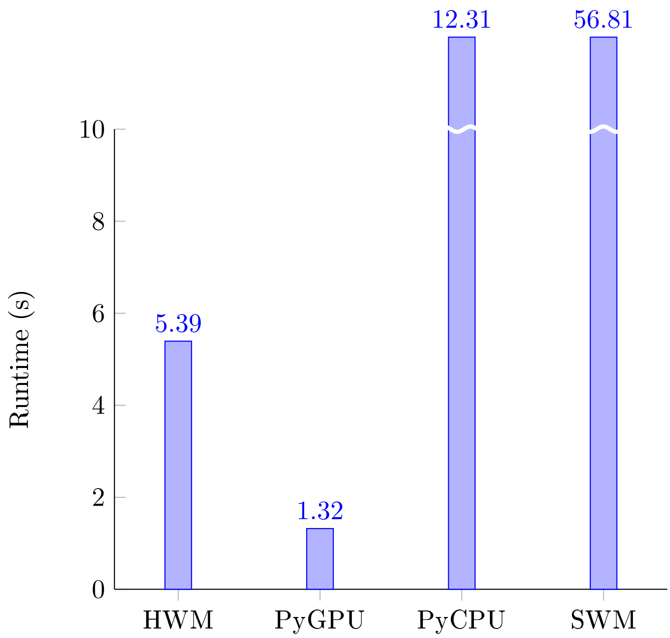

Figure 6.4: Inference runtime for various network models

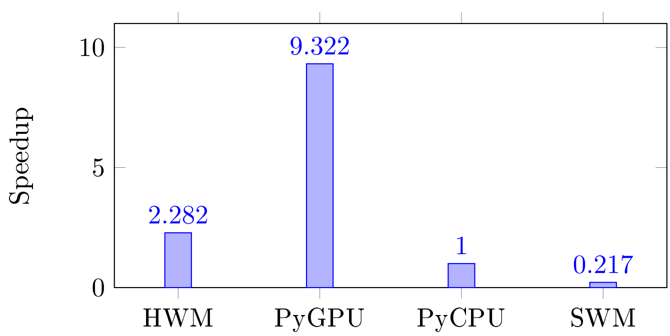

Figure 6.5: Inference speedup using the PyCPU as a baseline

	Active Cycle Percentage
Inference	25.13%
Training	69.20%

Table 6.1: Active Cycle Percentages for inference and training.

6.4.3 Active/Idle Cycles

To determine the impact of using MMIO via AXI bus to transfer image data between the PS and the FPGA, active and idle cycles were measured during training and inference. An active cycle is defined as a cycle on the FPGA during which at least one of the layers was computationally active. An idle cycle is thus defined as a non-active cycle.

An experiment was performed to measure active cycle percentages for the HWM during inference and training. The dataset was the entire MNIST dataset in both cases. The active cycle percentage for inference and training are shown in Table 6.1.

This experiment was performed to evaluate if the sending of input over MMIO was the bottleneck of the system. As can be clearly seen from the table, the MMIO transfer of training data was indeed the bottleneck. Furthermore, since backpropagation requires roughly double the amount of work compared to inference, it makes sense that training (which is both inference and backpropagation) is roughly a factor of 3 more active.

6.5 Training Accuracy

This section details the accuracy of the training process using the hardware accelerator. Varying training dataset sizes were chosen during the training process as the reduced precision training resulted in non-convergent training. As such, the training accuracy experiment conducted modified two variables: the learning rate and the training dataset size.

The tested learning rates were 2^{-7}, 2^{-8}, 2^{-9}, and 2^{-10} (0.0078, 0.0039, 0.00195, and 0.000977). This is because the hardware model performs the learning rate multiplication by using bitshifts. The experiments recorded the peak test data set accuracy during the training process. Note that the test dataset size for each run is 70000 minus the size of the training dataset. The results are shown in Figure 6.6. In this experiment, the highest test set accuracy, 85.845%, was

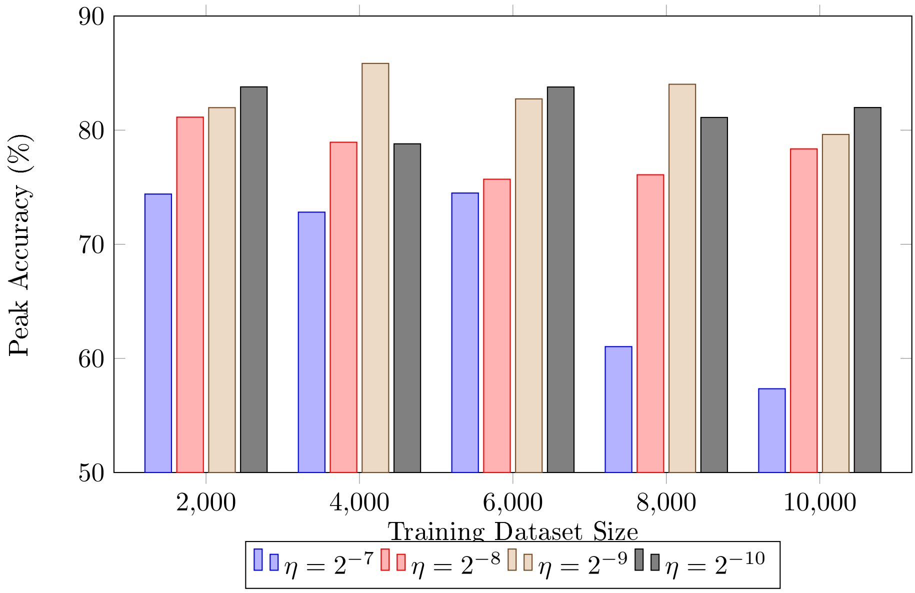

Figure 6.6: Maximum training accuracy reached for various learning rate and training set sizes.

achieved with a learning rate of $\eta = 2^{-9}$ and with a dataset of size 4,000. When using the same neural network architecture as the HWM, the SWM converges to 97.6% test set accuracy.

6.5.1 Stability of Training

As previously mentioned, due to the relatively low training precision of 18-bit fixed-point, the training process does not converge to a maximum training accuracy, but rather it will reach a maximum training accuracy, and then accuracy will degrade as precision errors accumulate over the training process. Training statistics for the first 10 epochs of the most optimal training configuration from Figure 6.6 illustrate this phenomenon and are shown in Figure 6.7.

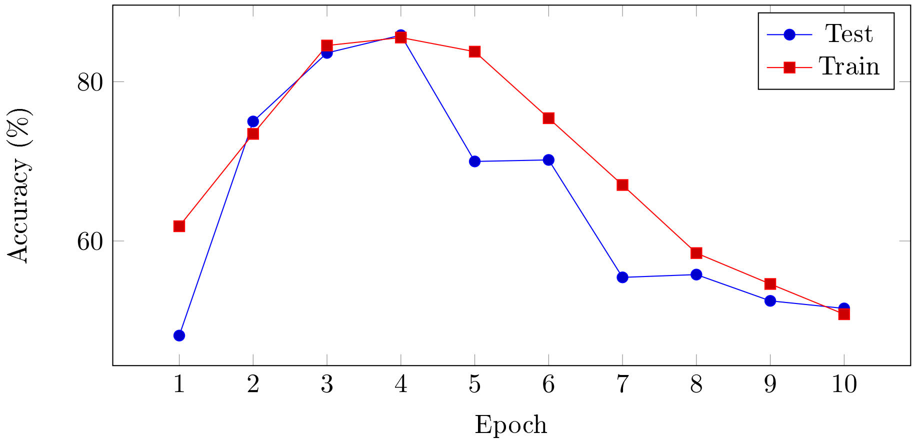

Figure 6.7: Epoch-by-epoch training data for an HWM configuration. Clearly visible degradation of accuracy instead of convergence after epoch 4.

6.6 Implemented Design

The design implemented for the FPGA is shown in Figure 6.8. As expected, the FC1 layer is by and large the most resource intensive, as it utilizes 196 kernels. It is interesting to observe the clustering of individual layer modules, while the interlayer activation buffer for FC0 and FC1 is widely spread out through the FPGA. This would indicate that this interlayer activation buffer was frequently routed to as a midpoint between FC0 and FC1.

It should be noted that implementation is a non-deterministic process and every design run should result in a slightly different implemented design. However, general trends for routing of the design tend to persist throughout multiple runs, despite the non-determism of the placing and routing algorithms.

6.6.1 Resource Usage, Power, and Timing

The resource usage of the hardware model is shown in Table 6.2. As can be seen, the DSP slice is the scarcest resource, with LUTs and BRAMs also heavily being used. Overall, high utilization of the FPGA resources were made to optimize the performance of the accelerator as much as possible.

According to the Vivado report, the total on-chip power of the design is 2.798

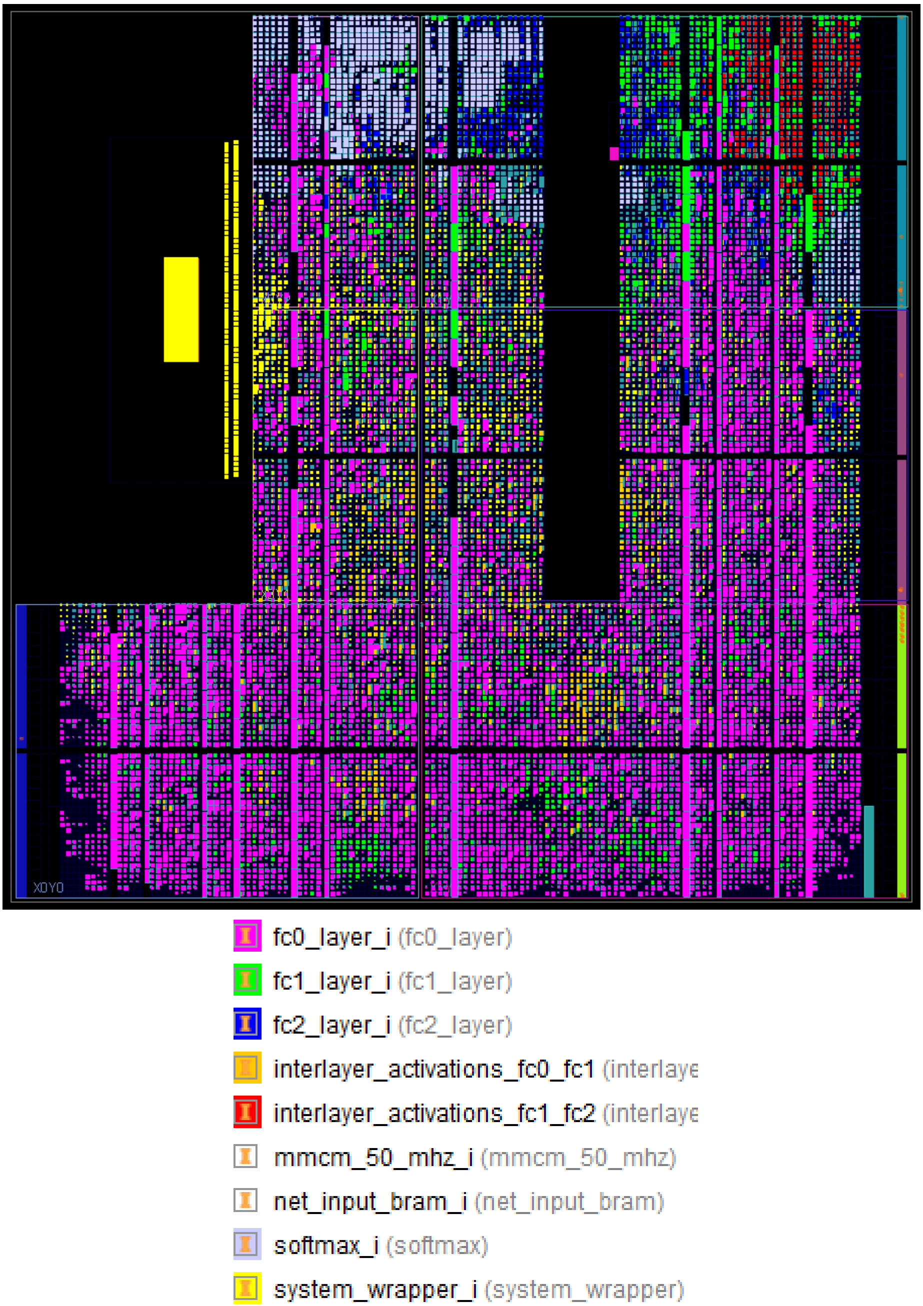

Figure 6.8: The implemented design of the hardware model

Resource	Utilization	Available	Utilization %
LUT	41132	53200	77.32
FF	54097	106400	50.84
BRAM	107.5	140	76.79
DSP	215	220	97.73

Table 6.2: Resource usage of the implemented design

Watts. While this number is reported with 'Low' confidence by Vivado, this wattage is far lower than typical GPU power consumptions. Power measurements have not been made for the internal GPU during these experiments, though measurements using the Torch framework were made and reported GPU average power at 94.19 Watts while performing training using the AlexNet architecture.The power measurements were performed with an Nvidia Tesla K20M GPU with 5 GB of GDDR5 SDRAM, a top-of-the-line GPU [Che16]. This is several magnitudes higher than the FPGA based solution in this thesis.

The implemented design for the hardware model is clocked at a frequency of 50 MHz. Placement and routing are both able to successfully complete with 0 timing violations. There was no need to improve frequency because the current performance bottleneck of this design stems from data transfer over the AXI bus and not from FPGA computational speed.

CHAPTER 7

Analysis

7.1 Allocating Computationl Kernels for Performance

7.1.1 Allocating Based on Only Forward Pass Analysis

When computing an optimal allocation of kernels to the fully-connected layers, it was enough to account only for the forward pass. This is because in the backward pass, there are 2 times the kernel is used, during previous layer neuron gradient calculation, and during weight gradient calculation. In each case, the amount of multiplications is equal to the sum of the fan-ins of all the neurons. For the forward pass, every neuron receives an input from every neuron in the previous layer, so the amount of MACs will be:

$$MACs = \#\text{previous layer neurons} \times \#\text{current layer neurons} \tag{7.1}$$

For the backward pass, each backpropagated neuron gradient to a previous layer requires an MAC on all the neurons in the current layer. This must be done for each neuron in the previous layer, thus the total amount of MACs for backpropagating neuron gradients is also equivalent to Equation 7.1.

Layer	Fan-in per Neuron	# Neurons	MACs
FC0	784	98	76832
FC1	98	64	6272
FC2	64	10	640

Table 7.1: MACs per layer during the forward pass

For computing all the weight gradients in a layer, every weight for every neuron must be multiplied by a gradient. Each neuron in the current layer will have a # previous layer neurons weights, as that is the fan-in for each neuron. Thus, there same amount of multiplications is also equal to the expresion in Equation 7.1.

The backward pass also has a weight updating step, however, this uses bitshifts and not DSP slices to multiply the weight gradient by the learning rate. As such, the backward pass in this model uses exactly twice the amount of multiplications as the forward pass, so the optimal allocation of kernels is optimal for both the forward and backward pass. Note however, that even if the update step used multiplication, the amount of extra multiplications would be 1 for every weight, which would still be a multiple of the amount of multiplciations in the forward pass.

7.1.2 Distribution of the Kernels

Table 7.1 shows the fan-in, number of neurons, and thus the number of MACs per layer during the forward pass.

Furthermore, recall that a kernel operates on only 1 neuron at a time. Therefore, the amount of kernels allocated should be either a multiple or a factor of the amount of neurons such that work can be evenly distributed across the kernels. Given the 220 DSP slices on the FPGA, this becomes an optimization problem such that the amount of time for each layer to finish computing their outputs should be roughly the same. This would allow for pipelined input processing if offline training for large batch-sizes were to be implemented.

FC0 has 120.05 times more MACs than FC2. FC1 has 9.8 times more MACs than FC2. To balance runtime per layer, FC0 should then have roughly 120.05 times more DSPs than FC2 and so on. Using this information, we can write an equation for the amount of MACs and use substitution to come up with an ideal allocation scheme if allocate partial DSPs and ignore the fact that kernels work on 1 neuron at a time.

$$\mathrm{DSP}_{FC0} = 120.05 DSP_{FC2} \tag{7.2}$$
$$\mathrm{DSP}_{FC1} = 9.8 DSP_{FC2} \tag{7.3}$$
$$\#\ \mathrm{DSPs} = 220 = \mathrm{DSP}_{FC0} + \mathrm{DSP}_{FC1} + \mathrm{DSP}_{FC2} \tag{7.4}$$

Substituting into equation 7.4 using equations 7.2 and 7.3:

$$\tag{7.5}$$
$$220 = 120.05 DSP_{FC2} + 9.8 DSP_{FC2} + DSP_{FC2} \tag{7.6}$$
$$220 = 129.85 DSP_{FC2} \tag{7.7}$$
$$DSP_{FC2} = 1.69 \tag{7.8}$$

Substituting the result from 7.8 into equations 7.2 and 7.3:

$$\mathrm{DSP}_{FC0} = 120.05 \times 1.60 = 203.36 \tag{7.9}$$
$$\mathrm{DSP}_{FC1} = 9.8 \times 1.69 = 16.60 \tag{7.10}$$

Thus if the 220 DSPs could be divided up ignoring all previous restrictions, the DSPs should be allocated according to Equations 7.8, 7.9 and 7.10. However, this is not possible, and DSPs are indivisible and the amount of kernels per layer should be a factor or multiple of the number of neurons, but it provides an maximum upper bound for performance.

Starting with layer FC0, which has 98 neurons, we should delegate 196 kernels. This is quite close to the optimal 203.36 computed above, thus layer FC0 should be allocated 196 kernels, which is $\frac{196}{203.36} = 96.38\%$ of the maximum upper bound. Continuing to layer FC1, which has 64 neurons, the optimal allocation is 16.6. The closest factor of 64 is thus 16 so 16 kernels are allocated, resulting in the same computation time as FC0 with $\frac{16}{16.6} = 96.38\%$ of the upper bound. Finally, layer FC2, with 10 neurons, rounding down and allocating 1 kernel would result in only $\frac{1}{1.69} = 59.17\%$ of the upper bound performance. Since a few kernels were freed up a from rounding down in FC0 and FC1, FC2 could be allocated 2 kernels, which allows it to finish faster than the optimally balanced latency for the 3 layers.

The final allocation of kernels is shown in Table 7.2. A pipelined solution is only as fast as its slowest step. Since FC0 and FC1 is the farthest away from the optimal upper bound, this solution performs at 96.38% of the theoretical upper bound for performance. It is worth re-mentioning that this upper bound is not actually possible since it assumes DSPs as divisible and that kernels can arbitrarily switched from neuron to neuron mid-computation, which would require finer-parallelism than what is supported in this architecture. Also note that since 214 of the 220 DSPs are used, the softmax layer was also able to use a DSP for calculation of the exponential function.

Layer	# Kernels
FC0	196
FC1	16
FC2	2
Total	214

Table 7.2: Kernel allocation between the fully-connected layers.

7.2 Cycle Analysis

This section calculates the computational cycles required for computing the forward and backward passes. Cycles spent pipelining or performing non-computational work are not included in this analysis. FP and BP are the amount of cycles needed to compute the forward pass and backward pass respectively. The amount of cycles for fully-connected layers in the forward pass can be roughly represented by $\frac{\#\text{MACs}}{\#\text{kernels}}$.

$$FP_{FC0} = \frac{76832}{196} = 392 \text{ cycles} \tag{7.11}$$

$$FP_{FC1} = \frac{6272}{16} = 392 \text{ cycles} \tag{7.12}$$

$$FP_{FC2} = \frac{640}{2} = 320 \text{ cycles} \tag{7.13}$$

For the softmax layer, there are several computational steps. For simplicity, conversions from fixed to floating point and vice versa are not included nor is the max circuit and the subsequent subtraction of the max. The softmax forward pass perform the exponential function, a sum of 10 values, and then a fixed point divison. The exponential function requires 20 cycles, the summation requires 10 cycles and the dividor takes 46 cycles. This results in $FP_{softmax} = 20 + 10 + 46 = 76$ cycles. FP is computed in Equation 7.16.

$$FP = FP_{FC0} + FP_{FC1} + FP_{FC2} + FP_{softmax} \tag{7.14}$$

$$FP = 392 + 392 + 320 + 76 \tag{7.15}$$

$$FP = 1180 \text{ cycles} \tag{7.16}$$

Computing the backward pass is a bit more involved. Note that backpropagation of from the softmax layer to FC2 takes 1 cycle so it is not included. Each fully-connected layer first backpropagates neuron gradients for the previous layer and then computes the weight gradients, however, the previous layer can start computing the backward pass as soon as the neuron gradients are ready. Thus, the time for backpropagation to finish is not based on layer computation time, but instead BP can be calculated by using neuron gradient computation time. Recall that the amount of computational cycles required to perform backpropagation of neuron gradients (NG) and computation of weight gradients (WG) is the same. As described in Chapter 4, updating the weights (UW) requires 2 cycles per-weight rather than 1-cycle per weight. Thus the UW step requires twice as many cycles as the others. The longest path during backpropagation is thus backpropagating neuron gradients from FC2 to FC1, from FC1 to FC0 and then performing WG in FC0 followed by UW.

$$BP = NG_{FC2} + NG_{FC1} + WG_{FC0} + UW_{FC0} \tag{7.17}$$

Substituting the forward pass cycles for the layers, as they are equivalent:

$$BP = 320 + 392 + 392 + (2 \times 392) \tag{7.18}$$

$$BP = 1888 \tag{7.19}$$

Thus, the amount of cycles spent performing computation during 1 training cycle is:

$$\text{Cycles} = FP + BP = 1180 + 1888 \tag{7.20}$$

$$\text{Cycles} = 3068 \text{ cycles} \tag{7.21}$$

This analysis has shown that the forward pass has 1,180 cycles of computation, and that one iteration of training contains 3,068 cycles of computation. Indeed, by viewing the output simulation file, this is confirmed as the forward pass finishes at cycle at cycle 1,235 and the training iteration concludes at cycle 3,145. The discrepancy in simulated cycles to computational cycles comes from the overhead of other actions in the design such as a max circuit, a pipelined addition reduction for neuron gradients, buffering of data between layers, non-computational data pipelining in layers, and floating-fixed type conversions. With a clock period of 20 nanoseconds (50 MHz) and ignoring data transfer overhead and delay, this would mean that 1 training iteration should take: $3145 \times 20ns = 62900ns$ or approximately 62.9 microseconds.

This result can be compared with the experimental results achieved. One training epoch was measured to take 5.455 seconds, as shown in Figure 6.2. Using the above calculated cycle results, at 62.9 microseconds per training iteration, training over a 60,000 image dataset we have:

$$t = 62.9\text{e}^{-6} \times 60000 = 3.774 \text{ seconds} \tag{7.22}$$

Recall however, how the active cycle percentage during training was only 69.2%. Thus, the analytically derived computation time should account for this:

$$t = 3.774 \times \frac{1.0}{0.692} \tag{7.23}$$

$$t = 5.454 \text{ seconds} \tag{7.24}$$

Compared to the experimentally measured epoch time of 5.455 seconds, the analytically computed training time of 5.454 seconds using cycle analysis is nearly exactly the same. The experimental data has thus successfully validated the cycle analysis.

7.3 Improving Performance

As was shown in Table 6.1 of the results section, the active cycle percentage for training is only 69.20%. Thus the first step to improve runtime performance would be to make data available for the FPGA to process faster. A suggested approach would be to use DRAM to stream data to the FPGA as done in other projects such as the neural network inference hardware accelerator proposed by Qiao et. al [QSX+16].

If an active cycle percentage of near 100% can be achieved from doing this, the next step to would be to optimize performance on the FPGA. The quickest route to doing this would be to improve the clock frequency. The design was clocked at a relatively low-frequency since it was not the bottleneck for performance. As such, the clock frequency was not investigated heavily during the design of the FPGA architecture, since improving this clock frequency would only increase the amount of time that the layers in the FPGA spend idling.

7.4 Granularity for Neural Network Computation

A key difference between training using this accelerator and training using GPUs is that this accelerator uses a much more fine-grained level of parallelism. While GPUs use data-level parallelism, this design uses neuron-level parallelism. Some attempts have been made to implement finer-level parallelism training on GPUs by Jiang et. al, though only yielded modest improvements of 1.58 to 2.19 times the speedup [JZL+18].

As a result, if one were to use the PyGPU solution to perform online training, then the GPU is 2.95 times slower than the CPU solution, which was 17.35 times slower than the hardware model (results in [**?**, inline]). Therefore, for online training, using fine-grained parallelism at the neuron level is the only place to find speedup, as data-level parallelism is not possible during online training where the batch size is only 1.

7.5 Ideal Learning Rate vs. Precision

One of the key intricacies in optimizing hyperparameters for the training process was balancing an ideal learning rate against increased precision error. Normally one need not think about precision when choosing a learning rate. However, in this accelerator, choosing what perhaps would be a more ideal learning rate might actually induce higher training error if it is too small since a smaller learning rate means fewer bits of gradient data are kept. For this project, this is compounded even further by the general notion that online training performs best with smaller learning rates.

For example, a learning rate of 0.001 results in a much better solution than a learning rate of 0.016 when a batch size of 1 is used for the implemented network architecture. However, the smaller in the implemented accelerator, this would mean an additional 4 bitshifts to the right when updating the weights. This results in a weight update that will have 4 fewer bits of information. In this project, the best training solution was found using 9 bitshifts to the right, or a learning rate of 0.00195. The weight gradients in this project are of number format Q1.17. Shifting this to the right 9 bits results in a Q1.8 number. This means that each and every weight update in the network only have 8 bits of information.

Vanishing Gradient Problem To add on even further to the aforementioned loss of information is that many of the gradients are already quite small. This is largely in part to a phenomenon referred to as the vanishing gradient problem. The vanishing gradient problem in neural networks refers to the fact that as one backpropagates further and further through the network, the gradients become smaller and smaller. To illustrate this phenomenon, the distributions of non-zero gradients in the implemented network has been plotted in Figure 7.1. As can be seen from the figure, gradients become smaller as backpropagation progresses from FC2 to FC0. Thus, as the gradients become smaller, they also become harder to represent using the limited precision of the hardware model. In an architecture where many bits of information are already being lost due to the learning rate, the vanishing gradient problem exacerbates the precision-induced error during training even more.

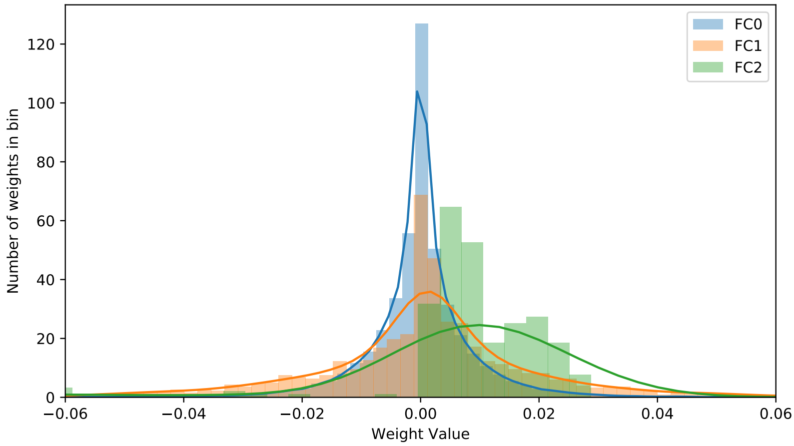

Figure 7.1: Weight gradient distribution for the 3 fully-connected layers after performing a single backward pass on the network using the input used to verify the hardware model from Chapter 5.

7.6 Potential Solutions for the Lack of Precision

As 18-bit fixed-point computation has been shown to be too imprecise to perform training on this network, potential solutions to this problem should be observed. It is the author's opinion that future accelerators for the training of neural networks would be best implemeneted by using 32-bit floating point as was done in the F-CNN accelerator by Zhao et. al [ZFL+16].

However, if 32-bit floating point is infeasible, or if accuracy is to be traded off for improved speed, area, and storage of weights, then investigations into designing accelerators using 16-bit or 24-bit floating-point computation could be made.

If the architecture must use fixed point, then the author would suggest first investigating 32-bit fixed-point computation with varying radices. If storage restrictions permit and the design performs multiplications using DSP slices, then a maximum of 36 bits would be supported for completing a multiplication using 2 DSPs in 1 cycle. This stems from the fact that 18 bits is the maximum width for one of the ports in a DSP-multiply. Otherwise, multiplication could also be a multi-cycle computation to trade off time for improved precision.

7.7 Weight Storage

The implemented neural network was specifically designed such that the weights and weight gradients could fit in the BRAM. Since 76.79% of the BRAM was utilized, the implemented network is representative of what architectures may be supported in BRAM as it comes close to hitting the upper limit of network size that can be supported entirely using BRAM. For networks larger than the implemented neural network for this thesis, other solutions such as a streaming weight and weight gradient datapath to DRAM would be required.

In addition, since the precision of the weight and weight gradients in this project proved to be inadequate for convergence to a local optimum during training, it should be noted that increasing precision would also increase BRAM utilization. This network would be able to use a maximum of 23-bits of precision for the weights while still fitting into BRAM at roughly 98.12% utilization. If more bits are needed for successful training then the network architecture would have to be made smaller or the hardware architecture would need to use a streaming datapath solution.

Chapter 8

Discussion

8.1 Overall Performance

Regarding the performance, the accelerator has outperformed all compared CPU benchmarks. It performs online training with a speedup of 17.35 compared to the PyTorch CPU model. Considering how the PyTorch GPU achieved a 19.05 speedup using a batch size of 50, the accelerator was nearly able to keep pace. Furthermore, the GPU model does not use fine-level parallelism, so the accelerator achieves the highest speedup of all models for training with a batch size of 1.

8.2 Finely-Grained Parallelism

Training of neural networks in today's world is done almost exclusively using GPUs and occasionally using CPUs. This is a stark contrast compared to inference, for which many different chips such as Google's TPU have been developed [JYP+17]. However, as this thesis has shown, for neural network training problems that do not have vast amount of data parallelism available, there is no highly optimized solution. As such, the accelerator developed during the process of this thesis shows a massive potential for this side of training since it takes advantage of the finely-grained parallelism available at the neuron-level, something not done by options available in today's world.

8.3 Limitations

8.3.1 Precision

Precision is a major limitation of training for the current design. It is the reason why the training process is not able to smoothly converge to a local optimum. This results in contradicting desires to have more bits of information available in weights gradients while at the same time having a low learning rate.

8.3.2 Data Transfer Rate

Another major limitation of this work is the method of transferring training data by using a memory-mapped interface between the PS to the FPGA. This approach was used for convenience, however, as the FPGA active cycle results from Table 6.1 showed, this approach is inefficient and became the largest bottleneck of performance for the design.

8.4 Future Work

While the potential for application-specific hardware accelerators training has been demonstrated in this thesis, there is a lot of potential for future work to improve the project.

Increased Precision As was demonstrated in the results section, training a neural network requires high precision computation. This is especially true for deeper neural networks as a result of the vanishing gradient problem. Therefore, increasing the precision, either via changing to floating point or using more bits in fixed point would be a great improvement.

Larger Batch Sizes Online training is only applicable to certain datasets. While the usefulness of an accelerator for online training has been shown, there are also many datasets that converge faster by using a larger batch size and off-line training. In addition, a larger batch size provides a more accurate gradient of the actual loss function of the training set.

Since the amount of data-level parallelism increases with the batch size, it becomes increasingly harder to compete with the performance of GPUs. Furthermore, a solution to storing activations in memory to compute the backward must be designed. That being said, using a larger batch size would also open up the possibility to taking advantage of data-level parallelism and using an array of training accelerators. In such a setup, both data-level and neuron-level parallelism would be working together.

Additional Layer Types This design only implemented the fully-connected and softmax layer types. There are many other types of layers for neural networks, and this project could be expanded by implementing other layer types such as convolutional or pooling layers, which are frequently used in image recognition.

Backward Pass for Biases In the interest of time and since the input data is already fairly normalized, only the backward pass for weights was computed. A rather quick improvement to the project would be to implement the backward pass for biases, so that the network architecture could be applied to non-normalized datasets as well. The gradient for a bias is simply the gradient of the net, as it is added directly to the net. Therefore, the bias gradients are already known in the hardware, and all the would be done would be to add BRAMs for the biases and slightly modify the update phase to update the biases.

Additional Activation Functions In both the software and hardware models for this project, the ReLU function was chosen specifically due to its computational simplicity, quick convergence during training, and its ability to converge to strong local optima. That being said, there are still many other activation functions in the realm of neural networks that also achieve strong training results. As the dataset and network architecture changes, so may the the most optimal activation function. Other activations functions such as the sigmoid function, leaky ReLU, hyperbolic tangent, and many others may be preferred to ReLU under certain circumstances. These functions would require extra hardware support though, and thus would require more computational resources to implement. As a result, one should expect that the performance of the accelerator would not be quite so high as with the ReLU activation function.

Implement Streaming DDR Interface for FPGA Adding a streaming data interface for training data would reduce the amount of cycles during which the FPGA idles. This would be a strong improvement for performance. Adding a streaming data DDR interface for weights and activations would allow networks with larger footprints to be supported by the hardware model. Both of these modifications would be an overall improvement to the model.

Generated HDL for a Pre-Specified Network Architecture As one of the design goals was to be modular, if the streaming data interface were to be implemented, then it would be feasible to define a network architecture in a configuration file and create a program to generate HDL files for that network architecture. This would allow for a flexible, modular, FPGA-based framework that could implement any type of network, so long as the layer-types of that network were supported.

Chapter 9

Conclusion

This thesis addressed the dearth of performance-optimized solutions for conducting online training of neural networks by proposing a novel hardware architecture. The proposed hardware accelerator achieves high performance by exploiting the fine-grain parallelism present at the neuron level during the training process.

After providing a background of neural networks, a software model was implemented to verify the chosen training algorithm for the hardware model. This provided the algorithmic foundation of the hardware model.

We then provided an explanation of the architecture and implementation of the hardware model. By carefully allocating computational kernel among the fully connected layers, a balanced computational scheme allowed for highly parallelized computation. The implemented design was a modular solution that resulted in a flexible design. Furthermore, full-scale and modular testbenches along with a convenient testing process resulted in the successful verification of the design.

The final design for the hardware accelerator was clocked at 50 MHz and satisfies all timing requirements. Furthermore, the implemented design uses low-power compared to GPUs. Experimental results of the hardware accelerator show that the proposed solution achieves a speedup of 17.35 compared to the next best online training model. At the same time, the accelerator is nearly as fast as the

GPU model performing training with a batch size of 50. The experiments also revealed that the bottleneck of the solution was from the MMIO communication between the PS and FPGA rather than the training on the FPGA. The results also showed that 18 bits of fixed-point precision is not enough to successfully converge to a local optimum during training, rather the training process will degrade in performance after a few epochs as precision error accumulates.

We then analyzed and discussed the experimental results, highlighting the need for more computational precision while showing the massive potential gains in performance from utilizing fine-grained parallelism.

Ultimately, this thesis has shown the feasibility of designing a hardware accelerator that uses neuron-level parallelism for the online training of neural networks. Furthermore, there are many potential future optimizations and improvements to the model that would increase both performance and functionality.

Appendix A

Experiment Data

Appendix A contains the experiment data used in the Results chapter of the thesis.

Training: Time Per Epoch (60,000 images)

Model	**Runtime (s)**	**Speedup**
HWM	5.455	17.348
SWM	272.67	0.347
PyCPU	94.633	1
PyGPU (Batch size: 50)	4.969	19.05

Training: Time Per Epoch. Model: PyGPU. Varying Batch Sizes (60,000 images)

Batch Size	**Runtime (s)**	**Speedup**
1	279.14	1
5	49.892	5.594965125
10	26.747	10.43646016
50	4.9689	56.17822858
100	2.6911	103.7285868
200	1.359	205.4039735
500	0.479	582.7640919

Inference Performance (70,000 images)

Model	**Runtime (s)**	**Speedup**
HWM	5.392	2.28208457
SWM	56.808	0.216606816
PyCPU	12.305	1
PyGPU (Batch size: 50)	1.32	9.321969697

Epoch-by-epoch Data for Unstable Training (70,000 images)

Epoch	**Test Accuracy (%)**	**Train Accuracy (%)**
1	48.12	61.85
2	75.012	73.45
3	83.62	84.55
4	85.85	85.55
5	69.99	83.78
6	70.18	75.42
7	55.43	67.03
8	55.78	58.48
9	52.48	54.6
10	51.53	50.8

Maximum Accuracy Results for Varying Learning Rates and Training Dataset Sizes

LRate (Shifts)	Dataset Size	Top Test (%)	Train (%)
7	1000	74.779	74.2
7	2000	74.401	77.5
7	3000	74.061	73.866
7	4000	72.814	65.85
7	5000	79.351	68.06
7	6000	74.49	68.78
7	7000	70.641	69.957
7	8000	61.03	69.22
7	9000	60.957	68.077
7	10000	57.34	53.4
8	1000	81.714	87.9
8	2000	81.14	84.25
8	3000	82.696	79.5
8	4000	78.94	77.82
8	5000	78.64	79.68
8	6000	75.698	70.1
8	7000	74.581	71.629
8	8000	76.088	72.47
8	9000	71.04	72.77
8	10000	78.36	73.21
9	1000	67.01	66.3
9	2000	81.97	84.95
9	3000	79.31	79.57
9	4000	85.845	85.549
9	5000	85.789	85.26
9	6000	82.737	84.2
9	7000	85.56	85.65
9	8000	84.02	85.69
9	9000	83.6	85.2
9	10000	79.615	84.34
10	1000	74.4	84.2
10	2000	83.79	86.5
10	3000	83.19	82.87
10	4000	78.8	77.72
10	5000	85.3	83.44
10	6000	83.78	81.78
10	7000	81.3	82.94
10	8000	81.11	84.425
10	9000	79.1	83.2
10	10000	81.98	80.59

Appendix B

Hardware Model Code

This appendix contains the SystemVerilog code used to implement the hardware accelerator. Display statements have been omitted as they serve no functional purpose to the implementation nor is there any particular insight to be gained from them. They may of course still be viewed at the GitHub repository. Note that some style has been modified make code fit on the page.

B.1 Source Files

B.1.1 sys_defs.vh

```
`ifndef __SYS_DEFS_VH__
`define __SYS_DEFS_VH__

// Precision defines
`define PREC                        18
`define MULT_BITS                   36
`define ACT_INT_BITS                6
`define ACT_FRAC_BITS               12
`define GRAD_INT_BITS               1
`define GRAD_FRAC_BITS              17
`define ONE                         18'h1_ffff
`define MAX_VAL                     18'h1_ffff
```

```
`define MIN_VAL                   18'h2_0000

// FC0 defines
`define FC0_N_KERNELS             196
`define FC0_PORT_WIDTH            98
`define FC0_NEURONS               98
`define FC0_FAN_IN                10'd784
`define FC0_KERNEL_FAN_IN         10'd392
`define FC0_MID_PTR_OFFSET        10'd784
`define FC0_ADDR                  10
`define FC0_BIAS_ADDR             1

// FC1 defines
`define FC1_N_KERNELS             16
`define FC1_ADDR                  10
`define FC1_PORT_WIDTH            8
`define FC1_PORT_WIDTH_TIMES2     16
`define FC1_PORT_WIDTH_TIMES3     24
`define FC1_BRAM                  1
`define FC1_NEURONS               64
`define FC1_BIAS_ADDR             2
`define FC1_FAN_IN                10'd98
`define FC1_STEP2                 10'd196
`define FC1_STEP3                 10'd294
`define FC1_MID_PTR_OFFSET        10'd392
`define FC1_MID_PTR_END           10'd784
`define FC1_HALF_NEURONS          32

// FC2 defines
`define FC2_BRAM                  1
`define FC2_NEURONS               10
`define FC2_FAN_IN                64
`define FC2_N_KERNELS             2
`define FC2_ADDR                  10
`define FC2_BIAS_ADDR             3
`define FC2_MID_PTR_OFFSET        320
`define FC2_HALF_NEURONS          5


// Backward pass defines
`define FC0_LOOPS                 1

`define FC1_MODE_SWITCH           4
`define FC1_LOOPS                 8

`define FC2_MODE_SWITCH           5
`define FC2_LOOPS                 10

`endif
```

B.1.2 neural_net_top.sv

```
`timescale 1ns / 1ps
`include "sys_defs.vh"

module neural_net_top(
  inout                       [14:0]DDR_addr,
  inout                       [2:0]DDR_ba,
  inout                       DDR_cas_n,
  inout                       DDR_ck_n,
  inout                       DDR_ck_p,
  inout                       DDR_cke,
  inout                       DDR_cs_n,
  inout                       [3:0]DDR_dm,
  inout                       [31:0]DDR_dq,
  inout                       [3:0]DDR_dqs_n,
  inout                       [3:0]DDR_dqs_p,
  inout                       DDR_odt,
  inout                       DDR_ras_n,
  inout                       DDR_reset_n,
  inout                       DDR_we_n,
  inout                       FIXED_IO_ddr_vrn,
  inout                       FIXED_IO_ddr_vrp,
  inout                       [53:0]FIXED_IO_mio,
  inout                       FIXED_IO_ps_clk,
  inout                       FIXED_IO_ps_porb,
  inout                       FIXED_IO_ps_srstb,

  input                       rst,
  input               [7: 0]  sw_in,
  input                       clock_in,
  output   logic      [7: 0]  led_o
  );

  logic                                             fab_clk;
  logic                                             clk;
  logic                                             forward;
  // Logics for the fc0 layer
  logic                                             fc0_start;
  logic [1: 0][`PREC - 1: 0]                        fc0_activation_i;
  logic                                             fc0_valid;
  logic                                             fc0_valid_i;
  logic [`FC0_NEURONS - 1: 0][`PREC - 1: 0]         fc0_activation_o ;
  logic [`FC0_NEURONS - 1: 0][6: 0]                 fc0_neuron_id_o ;
  logic                                             fc0_valid_act_o;
  logic                                             fc0_busy;
  logic [`FC0_NEURONS - 1: 0][`PREC - 1: 0]         fc0_gradients;
  logic                                             fc0_grad_valid;


  // Logics for the fc1 layer
  logic                                             fc1_start;
  logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0] fc1_activation_i ;
  logic                                             fc1_valid_i;
  logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0] fc1_activation_o;
```

```
logic [`FC1_N_KERNELS - 1: 0][5: 0]          fc1_neuron_id_o ;
logic                                        fc1_valid_act_o;
logic                                        fc1_buff_rdy;
logic                                        fc1_busy;
logic                                        fc1_grad_valid;

 // Logics for the fc2 layer (the last fc layer)
logic                                        fc2_start;
logic                                        fc2_buff_rdy;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0] fc2_activation_i;
logic                                        fc2_valid_i;
logic                                        fc2_busy;

logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0] fc2_activation_o;
logic [`FC2_N_KERNELS - 1: 0][3: 0]          fc2_neuron_id_o;
logic                                        fc2_valid_o;
logic [`FC2_NEURONS - 1: 0][`PREC - 1: 0]   fc2_act_o_buf;
logic                                        fc2_buf_valid;

// Backward pass logics
logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0] fc0_b_gradient_i;
logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0] fc0_b_activation_i;
logic [9: 0]                                 fc0_b_activation_id_i;
logic [9: 0]                                 fc0_b_activation_id_o;
logic                                        fc0_b_valid_i;
logic                                        fc0_b_start;
logic                                        fc0_b_start_r;
logic [3: 0]                                 fc0_loops;
logic [`FC0_NEURONS - 1: 0][`PREC - 1: 0]   fc0_gradients_i;
logic                                        fc0_gradients_rdy;
logic [6: 0]                                 fc0_n_loop_offset;
logic                                        fc0_bp_done;
logic                                        fc0_update;
logic                                        fc0_update_done;



logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0] fc1_b_gradient_i;
logic [`PREC - 1: 0]                         fc1_b_activation_i;
logic [6: 0]                                 fc1_b_activation_id_i;
logic [6: 0]                                 fc1_b_activation_id_o;
logic [`FC1_N_KERNELS - 1: 0][5: 0]          fc1_b_neuron_id_i;
logic                                        fc1_b_valid_i;
logic                                        fc1_b_start;
logic                                        fc1_b_start_r;
logic [3: 0]                                 fc1_loops;
logic [`FC1_NEURONS - 1: 0][`PREC - 1: 0]   fc1_gradients;
logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0] fc1_gradients_i;
logic                                        fc1_gradients_rdy;
logic [5: 0]                                 fc1_n_offset;
logic [5: 0]                                 fc1_n_loop_offset;
```

```
logic                                           fc1_bp_mode;
logic                                           fc1_bp_done;
logic                                           fc1_update;
logic                                           fc1_update_done;



logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0] fc2_b_gradient_i;
logic [`PREC - 1: 0]                            fc2_b_activation_i;
logic [5: 0]                                    fc2_b_activation_id_i
    ;
logic [5: 0]                                    fc2_b_activation_id_o
    ;
logic [`FC2_N_KERNELS - 1: 0][3: 0]             fc2_b_neuron_id_i;
logic                                           fc2_b_valid_i;
logic                                           fc2_b_start;
logic                                           fc2_b_start_r;
logic [3: 0]                                    fc2_loops;
logic [`FC2_NEURONS - 1: 0][`PREC - 1: 0]       fc2_gradients;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0] fc2_gradients_i;
logic                                           fc2_gradients_rdy;
logic [3: 0]                                    fc2_n_offset;
logic                                           fc2_bp_mode;
logic                                           fc2_bp_done;
logic                                           fc2_update;
logic                                           fc2_update_done;

logic [7: 0]                                    img1_unpacked[784];
logic                                           new_img;
logic [9:0]                                     epoch;
logic [16:0]                                    img_id;
logic [4: 0]                                    lrate_shifts;
logic [4: 0]                                    lrate_shifts_bus;
logic [31: 0]                                   active_cycles;
logic [31: 0]                                   idle_cycles;
logic                                           training_mode;
logic                                           training_mode_bus;
logic [16:0]                                    img_set_size;

logic [16:0]                                    img1_id;
logic [16: 0]                                   prev_img_id;
logic [9:0]                                     img1_label;
logic [9:0]                                     n_epochs;
logic [16:0]                                    num_correct_test;
logic [16:0]                                    num_correct_train;
logic                                           start;
logic                                           start_bus;

// Layer States
logic [2: 0]                                    fc0_state;
logic [2: 0]                                    next_fc0_state;
logic [2: 0]                                    fc1_state;
logic [2: 0]                                    next_fc1_state;
logic [2: 0]                                    fc2_state;
logic [2: 0]                                    next_fc2_state;
```

```
logic all_idle;


logic [9:0]                                      input_addr;
logic [`PREC - 1:0]                              net_input_bram_dout_a;
logic [`PREC - 1:0]                              net_input_bram_dout_b;
logic [`PREC - 1:0]                              input_data_a;
logic [`PREC - 1:0]                              input_data_b;
logic [9:0]                                      img_label;
logic                                            img_rdy;
logic                                            epoch_fin;
logic                                            correct;
logic [12:0]                                     fc0_ptr_a;
logic [12:0]                                     fc0_ptr_b;
logic [9:0]                                      fc0_addr_a;
logic [9:0]                                      fc0_addr_b;
logic [`FC2_NEURONS - 1:0][`PREC - 1:0]          fc2_out;
logic [4:0][`PREC - 1:0]                         max1;
logic [2:0][`PREC - 1:0]                         max2;
logic [1:0][`PREC - 1:0]                         max3;
logic [`PREC - 1:0]                              max4;
logic [`PREC - 1:0]                              max;
logic [4:0]                                      max_valid;
logic [7:0]                                      led_o_r;
logic                                            sm_valid_o;
logic [`FC2_NEURONS - 1:0][`PREC - 1:0]          sm_grad_o;
logic [31:0]                                     status_block;


localparam sf   = 2.0**-12.0;
localparam sf2  = 2.0**-17.0;

// Backward pass states
localparam WEIGHT_MODE = 0;
localparam NEURON_MODE = 1;

// Layer states
localparam FORWARD  = 1;
localparam WAITING  = 2;
localparam BACKWARD = 3;
localparam UPDATE   = 4;
localparam IDLE   = 5;


mmcm_50_mhz mmcm_50_mhz_i (
  .clk_in1(fab_clk),
  //.clk_in1(clock_in),
  .clk_out1(clk)
);

logic [7:0] sw_i; // so simulation uses net_input_bram
assign sw_i = sw_in;

```

```
assign start         = sw_i[0] ? 1'b1 : start_bus;
assign training_mode = sw_i[0] ? 1'b1 : training_mode_bus;
assign forward       = fc0_state == FORWARD || fc1_state == FORWARD || fc2_state == FORWARD;
assign all_idle      = (fc0_state == IDLE) && (fc1_state == IDLE) && (fc2_state == IDLE);
assign img_rdy       = (img1_id == (img_id + 1'b1)) | (img1_id == 0 && img_id == img_set_size);
assign new_img       = start & all_idle & img_rdy;
assign epoch_fin     = sw_i[0] ? 1'b0 : epoch == n_epochs;

logic reset_i;
logic reset;
always_ff @(posedge clk) begin
  reset_i       <= rst;
  lrate_shifts  <= sw_i[0] ? 5'd7 : lrate_shifts_bus;
end

always_ff @(posedge clk) begin
  if (reset || !start) begin
    idle_cycles     <= 0;
    active_cycles   <= 0;
  end
  else begin
    idle_cycles     <= idle_cycles + all_idle;
    active_cycles   <= all_idle ? active_cycles : active_cycles + 1'b1;
  end
end

BUFG BUFG_reset(.I(reset_i), .O(reset));

always_ff @(posedge clk) begin
  if (reset) begin
    input_addr  <= 0;
    fc0_start   <= 0;
  end
  else if (fc0_state == FORWARD & !fc0_start && ~epoch_fin) begin
    fc0_start   <= 1'b1;
    input_addr  <= 0;
  end
  else if (fc0_state == FORWARD & fc0_start) begin
    input_addr  <= input_addr + 1'b1;
  end
  else begin
    fc0_start   <= 1'b0;
    input_addr  <= 0;
  end
end
```

```
  assign fc0_addr_a = (forward) ? input_addr << 1 :
      fc0_b_activation_id_i << 1;
  assign fc0_addr_b = fc0_addr_a + 1'b1;


  net_input_bram net_input_bram_i (
    .addra(fc0_addr_a),
    .clka(clk),
    .dina(18'b0),
    .douta(net_input_bram_dout_a),
    .ena(1'b1),
    .wea(1'b0),

    .addrb(fc0_addr_b),
    .clkb(clk),
    .dinb(18'b0),
    .doutb(net_input_bram_dout_b),
    .enb(1'b1),
    .web(1'b0)
  );




  always_ff @(posedge clk) begin
    if (reset) begin
      prev_img_id <= img_set_size;
    end
    else begin
      prev_img_id <= img_id;
    end


    if (reset || (img_id == 0 && prev_img_id != 0)) begin
      num_correct_train <= 0;
      num_correct_test  <= 0;
    end
    else if (correct) begin
      num_correct_train <= (~training_mode) ?
                  num_correct_train : num_correct_train + 1'b1;
      num_correct_test  <= (training_mode) ?
                  num_correct_test : num_correct_test + 1'b1;
    end

    if (reset) begin
      epoch    <= 0;
    end
    else if (img_id == 0 && prev_img_id != 0) begin
      epoch    <= epoch + 1'b1;
    end
  end


  always_comb begin
    input_data_a  <= sw_i[0] ? net_input_bram_dout_a  :
```

```
               {6'b0, img1_unpacked[fc0_addr_a], 4'b0};
    input_data_b   <= sw_i[0] ? net_input_bram_dout_b :
               {6'b0, img1_unpacked[fc0_addr_b], 4'b0};
  end


  always_ff @(posedge clk) begin
    if (reset) begin
      fc0_valid          <= 0;
      fc0_valid_i        <= 0;
    end
    else begin
      fc0_valid          <= fc0_start;
      fc0_valid_i        <= fc0_valid;
      fc0_activation_i   <= {input_data_b, input_data_a};
    end
  end


  assign fc0_b_activation_i    = {{`FC0_NEURONS{input_data_b}}, {`
      FC0_NEURONS{input_data_a}}};
  assign fc0_gradients_rdy   = fc0_grad_valid;
  // Start when backward is good and gradients are ready. Only do
      backprop once
  assign fc0_b_start = fc0_state == BACKWARD;
  bit [7: 0] q, r;
  always_ff @(posedge clk) begin
    if (reset) begin
      fc0_b_start_r    <= 1'b0;
    end
    else begin
      fc0_b_start_r    <= fc0_b_start;
    end

    // Loop over fan in
    if (reset) begin
      fc0_loops    <= 0;
    end
    else if (fc0_state != BACKWARD) begin
      fc0_loops    <= 0;
    end
    else if (fc0_b_activation_id_i == (`FC0_KERNEL_FAN_IN - 1))
        begin
      fc0_loops    <= fc0_loops + 1'b1;
    end

    if (reset) begin
      fc0_b_activation_id_i <= 0;
    end
    else if (fc0_state != BACKWARD) begin
      fc0_b_activation_id_i <= 0;
    end
    else if (fc0_b_start) begin
      fc0_b_activation_id_i <= (fc0_b_activation_id_i == (`
          FC0_KERNEL_FAN_IN - 1'b1)) ?
```

```
                   0 : fc0_b_activation_id_i + 1'b1;
    end

    for (q = 0, r = `FC0_PORT_WIDTH; q < `FC0_PORT_WIDTH; q=q+1, r=r+1) begin
      fc0_gradients_i[q]    <= fc0_gradients[q];
    end
    fc0_b_activation_id_o     <= fc0_b_activation_id_i << 1;
  end
  always_comb begin
    case(fc0_state)
      FORWARD:
        next_fc0_state  = fc1_buff_rdy & training_mode
                                    ? WAITING   :
                    fc1_buff_rdy & ~training_mode
                                    ? IDLE     : FORWARD;
      WAITING:
        next_fc0_state  = (fc0_gradients_rdy)   ? BACKWARD  : WAITING;
      BACKWARD:
        next_fc0_state  = (fc0_bp_done)      ? UPDATE   : BACKWARD;
      UPDATE:
        next_fc0_state  = (fc0_update_done)   ? IDLE     : UPDATE;
      IDLE:
        next_fc0_state  = (new_img | sw_i[0])   ? FORWARD   : IDLE;
      default:
        next_fc0_state  = IDLE;
    endcase
  end
  always_ff @(posedge clk) begin
    if (reset) begin
      fc0_state    <= IDLE;
    end
    else begin
      fc0_state    <= next_fc0_state;
    end
  end

  assign fc0_update = fc0_state == UPDATE;
  // FC0
  fc0_layer fc0_layer_i (
    // inputs
    .clk(clk),
    .rst(reset),
    .forward(forward),
    .update(fc0_update),
    .activations_i(fc0_activation_i),
    .valid_i(fc0_valid_i & forward),
    .lrate_shifts(lrate_shifts),

    // backward pass inputs
    .b_gradient_i(fc0_gradients_i),
    .b_activation_i(fc0_b_activation_i),
    .b_activation_id(fc0_b_activation_id_o),
```

```
    .b_valid_i(fc0_b_start_r),

    // outputs
    .activation_o(fc0_activation_o),
    .neuron_id_o(fc0_neuron_id_o),
    .valid_act_o(fc0_valid_act_o),
    .fc0_busy(fc0_busy),
    .bp_done(fc0_bp_done),
    .update_done(fc0_update_done)
  );

  always_ff @(posedge clk) begin
    if (reset) begin
      fc1_start   <= 1'b0;
    end
    else begin
      fc1_start   <= fc1_state == FORWARD & fc1_buff_rdy;
    end
  end
  interlayer_activation_buffer
  #(.N_KERNELS_I(`FC0_NEURONS),
    .N_KERNELS_O(`FC1_N_KERNELS),
    .ID_WIDTH(7),
    .BUFF_SIZE(`FC0_NEURONS),
    .LOOPS(4))
  interlayer_activations_fc0_fc1 (
    // inputs
    .clk(clk),
    .rst(reset),

    .start(fc1_start),
    .activation_i(fc0_activation_o),
    .neuron_id_i(fc0_neuron_id_o),
    .valid_act_i(fc0_valid_act_o & forward),
    .b_ptr(fc1_b_activation_id_i),
    // outputs
    .activation_o(fc1_activation_i),
    .valid_o(fc1_valid_i),

    .b_act_o(fc1_b_activation_i),

    .buff_rdy(fc1_buff_rdy)
  );



  assign fc1_gradients_rdy  = fc1_grad_valid;
  assign fc1_n_offset       = (fc1_loops >= `FC1_MODE_SWITCH) ?
      fc1_loops - 4 : fc1_loops;
  // Start when backward is good and gradients are ready. Only do
      backprop once
  assign fc1_b_start        = fc1_state == BACKWARD;
  bit [5: 0] o, p;
  always_ff @(posedge clk) begin
    if (reset) begin
```

```
      fc1_b_start_r   <= 1'b0;
      fc1_bp_mode    <= 1'b0;
    end
    else begin
      fc1_b_start_r   <= fc1_b_start;
      fc1_bp_mode    <= fc1_loops >= `FC1_MODE_SWITCH ? WEIGHT_MODE
          : NEURON_MODE;
    end

    // Loop over fan in
    if (reset) begin
      fc1_loops    <= 0;
    end
    else if (fc1_state != BACKWARD) begin
      fc1_loops    <= 0;
    end
    else if (fc1_b_activation_id_i == (`FC0_NEURONS - 1)) begin
      fc1_loops    <= fc1_loops + 1'b1;
    end

    if (reset) begin
      fc1_b_activation_id_i <= 0;
    end
    else if (fc1_state != BACKWARD) begin
      fc1_b_activation_id_i <= 0;
    end
    else if (fc1_b_start) begin
      fc1_b_activation_id_i <= (fc1_b_activation_id_i == (`
          FC1_FAN_IN - 1'b1)) ?
                    0 : fc1_b_activation_id_i + 1'b1;
    end

    for (p = 0, o = `FC1_PORT_WIDTH; p < `FC1_PORT_WIDTH; p=p+1, o=
        o+1) begin
      fc1_gradients_i[p]     <= fc1_gradients[(fc1_n_offset << 3) +
          p];
      fc1_gradients_i[o]     <= fc1_gradients[((fc1_n_offset << 3) +
           p) | 6'd32];
      fc1_b_neuron_id_i[p]  <= (fc1_n_offset << 3) + p;
      fc1_b_neuron_id_i[o]  <= ((fc1_n_offset << 3) + p) | 6'd32;
    end
    fc1_b_activation_id_o    <= fc1_b_activation_id_i;
  end

  always_comb begin
    case(fc1_state)
      FORWARD:
        next_fc1_state  = fc2_buff_rdy & training_mode
                          ? WAITING    :
                  fc2_buff_rdy & ~training_mode
                          ? IDLE     : FORWARD;
      WAITING:
        next_fc1_state  = (fc1_gradients_rdy)    ? BACKWARD   :
            WAITING;
      BACKWARD:
```

```
        next_fc1_state  = (fc1_bp_done)     ? UPDATE  : BACKWARD;
      UPDATE:
        next_fc1_state  = (fc1_update_done)  ? IDLE    : UPDATE;
      IDLE:
        next_fc1_state  = (new_img | sw_i[0])  ? FORWARD   : IDLE;
      default:
        next_fc1_state  = IDLE;
    endcase
  end
  always_ff @(posedge clk) begin
    if (reset) begin
      fc1_state   <= IDLE;
    end
    else begin
      fc1_state   <= next_fc1_state;
    end
  end

  assign fc1_update = fc1_state == UPDATE;
  // FC1
  fc1_layer fc1_layer_i (
    // inputs
    .clk(clk),
    .rst(reset),
    .forward(forward),
    .update(fc1_update),
    .activations_i(fc1_activation_i),
    .valid_i(fc1_valid_i & forward),
    .lrate_shifts(lrate_shifts),

    // backward pass inputs
    .b_gradient_i(fc1_gradients_i),
    .b_activation_i({`FC1_N_KERNELS{fc1_b_activation_i}}),
    .b_activation_id(fc1_b_activation_id_o),
    .b_neuron_id_i(fc1_b_neuron_id_i),
    .b_valid_i(fc1_b_start_r),
    .bp_mode(fc1_bp_mode),

    // outputs
    .activation_o(fc1_activation_o),
    .neuron_id_o(fc1_neuron_id_o),
    .valid_act_o(fc1_valid_act_o),
    .fc1_busy(fc1_busy),
    .bp_done(fc1_bp_done),
    .update_done(fc1_update_done),

    // backward pass outputs
    .pl_gradients(fc0_gradients),
    .pl_grad_valid(fc0_grad_valid)
  );


  always_ff @(posedge clk) begin
    if (reset) begin
```

```
      fc2_start   <= 1'b0;
    end
    else begin
      fc2_start   <= fc2_state == FORWARD & fc2_buff_rdy;
    end
  end


  interlayer_activation_buffer
  #(.N_KERNELS_I(`FC1_N_KERNELS),
    .N_KERNELS_O(`FC2_N_KERNELS),
    .ID_WIDTH(6),
    .BUFF_SIZE(`FC1_NEURONS),
    .LOOPS(`FC2_NEURONS))
  interlayer_activations_fc1_fc2 (
    // inputs
    .clk(clk),
    .rst(reset),

    .start(fc2_start),
    .activation_i(fc1_activation_o),
    .neuron_id_i(fc1_neuron_id_o),
    .valid_act_i(fc1_valid_act_o & forward),
    .b_ptr(fc2_b_activation_id_i),
    // outputs

    .activation_o(fc2_activation_i),
    .valid_o(fc2_valid_i),

    .b_act_o(fc2_b_activation_i),

    .buff_rdy(fc2_buff_rdy)
  );

  always_comb begin
    case(fc2_state)
      FORWARD:
        next_fc2_state  =     fc2_buf_valid & training_mode
                            ? WAITING   :
                              fc2_buf_valid & ~training_mode
                            ? IDLE    : FORWARD;
      WAITING:
        next_fc2_state  = (fc2_gradients_rdy) ? BACKWARD  : WAITING ;
      BACKWARD:
        next_fc2_state  = (fc2_bp_done)       ? UPDATE  : BACKWARD;
      UPDATE:
        next_fc2_state  = (fc2_update_done)   ? IDLE    : UPDATE;
      IDLE:
        next_fc2_state  = (new_img | sw_i[0]) ? FORWARD   : IDLE;
      default:
        next_fc2_state  = IDLE;
    endcase
  end
  always_ff @(posedge clk) begin
```

```
    if (reset) begin
      fc2_state    <= IDLE;
    end
    else begin
      fc2_state    <= next_fc2_state;
    end
  end




  assign fc2_n_offset = (fc2_loops >= `FC2_MODE_SWITCH) ? fc2_loops - 5 : fc2_loops;

  // Start when backward is good and gradients are ready. Only do backprop once
  assign fc2_b_start = fc2_state == BACKWARD;
  always_ff @(posedge clk) begin
    if (reset) begin
      fc2_b_start_r    <= 1'b0;
      fc2_bp_mode    <= 1'b0;
    end
    else begin
      fc2_b_start_r    <= fc2_b_start;
      fc2_bp_mode    <= fc2_loops >= `FC2_MODE_SWITCH ? WEIGHT_MODE : NEURON_MODE;
    end

    // Loop over fan in
    if (reset) begin
      fc2_loops    <= 0;
    end
    else if (fc2_state != BACKWARD) begin
      fc2_loops    <= 0;
    end
    else if (fc2_b_activation_id_i == (`FC1_NEURONS - 1)) begin
      fc2_loops    <= fc2_loops + 1'b1;
    end


    if (reset) begin
      fc2_b_activation_id_i <= 0;
    end
    else if (fc2_state != BACKWARD) begin
      fc2_b_activation_id_i <= 0;
    end
    else if (fc2_b_start) begin
      fc2_b_activation_id_i <= fc2_b_activation_id_i + 1'b1;
    end
    fc2_gradients_i        <= {fc2_gradients[fc2_n_offset + 5], fc2_gradients[fc2_n_offset]};
    fc2_b_neuron_id_i      <= {fc2_n_offset + 5, fc2_n_offset};
    fc2_b_activation_id_o  <= fc2_b_activation_id_i;
  end
```

```
assign fc2_update = fc2_state == UPDATE;
// FC2, fed directly from FC1 due to the small size
fc2_layer fc2_layer_i (
  // inputs
  .clk(clk),
  .rst(reset),
  .forward(forward),
  .update(fc2_update),
  .activations_i(fc2_activation_i),
  .valid_i(fc2_valid_i & forward),
  .lrate_shifts(lrate_shifts),

  // backward pass inputs
  .b_gradient_i(fc2_gradients_i),
  .b_activation_i({fc2_b_activation_i, fc2_b_activation_i}),
  .b_activation_id(fc2_b_activation_id_o),
  .b_neuron_id_i(fc2_b_neuron_id_i),
  .b_valid_i(fc2_b_start_r),
  .bp_mode(fc2_bp_mode),

  // outputs
  .activation_o(fc2_activation_o),
  .neuron_id_o(fc2_neuron_id_o),
  .valid_act_o(fc2_valid_o),
  .fc2_busy(fc2_busy),
  .bp_done(fc2_bp_done),
  .update_done(fc2_update_done),

  // backward pass outputs
  .pl_gradients(fc1_gradients),
  .pl_grad_valid(fc1_grad_valid)
);






bit [`FC2_N_KERNELS - 1: 0] m;
logic prev_fc2_buf_valid;
always_ff @(posedge clk) begin
  if (reset) begin
    prev_fc2_buf_valid   <= 0;
    fc2_act_o_buf        <= 0;
  end
  else begin
    prev_fc2_buf_valid   <= fc2_buf_valid;
    for (m = 0; m < `FC2_N_KERNELS; m=m+1) begin
      if (fc2_valid_o && forward) begin
        fc2_act_o_buf[fc2_neuron_id_o[m]]   <= fc2_activation_o[m
            ];
      end
    end
  end

```

```
    if (reset) begin
      fc2_buf_valid   <= 1'b0;
    end
    else if (fc2_valid_o) begin
      fc2_buf_valid   <= fc2_neuron_id_o[`FC2_N_KERNELS - 1] == `FC2_NEURONS - 1;
    end
    else if (fc2_state == IDLE) begin
      fc2_buf_valid   <= 1'b0;
    end
  end

  always @(posedge clk) begin
    if (fc2_buf_valid) begin
      fc2_out <= fc2_act_o_buf;
    end
  end




  // LED Logic
  bit [3: 0] k;
  bit [3: 0] j, t;
  always_ff @(posedge clk) begin
    if (reset) begin
      max         <= 0;
      max_valid   <= 0;
    end
    else if ({fc2_buf_valid, prev_fc2_buf_valid} == 2'b10) begin
      for (k = 0; k < 5; k=k+1) begin
        max1[k] <= $signed(fc2_act_o_buf[2*k]) > $signed(fc2_act_o_buf[2*k+1]) ?
          fc2_act_o_buf[2*k] : fc2_act_o_buf[2*k + 1];
      end
      max_valid     <= {max_valid[3: 0], 1'b1};
    end
    else begin
      max_valid[0]  <= 1'b0;

      max2[0]       <= $signed(max1[0]) > $signed(max1[1]) ? max1[0] : max1[1];
      max2[1]       <= $signed(max1[2]) > $signed(max1[3]) ? max1[2] : max1[3];
      max2[2]       <= max1[4];
      max_valid[1]  <= max_valid[0];

      max3[0]       <= $signed(max2[0]) > $signed(max2[1]) ? max2[0] : max2[1];
      max3[1]       <= max2[2];
      max_valid[2]  <= max_valid[1];

      max4          <= $signed(max3[0]) > $signed(max3[1]) ? max3[0] : max3[1];
      max_valid[3]  <= max_valid[2];
```

```
      max             <= max4;
      max_valid[4]  <= max_valid[3];


    end
    if (reset) begin
      led_o_r    <= 0;
      correct    <= 1'b0;
    end
    else if (max_valid[4]) begin
      correct   <= fc2_act_o_buf[img_label] == max;
      for (t = 0; t < `FC2_NEURONS; t=t+1) begin
        if (fc2_act_o_buf[t] == max && t != img_label) begin
          correct  <= 1'b0;
        end
      end
      for (j = 0; j < 8; j=j+1) begin
        led_o_r[j]  <= fc2_act_o_buf[j] == max;
      end
    end
    else begin
      correct    <= 1'b0;
    end
    led_o[7:0]   <= led_o_r[7: 0];
  end


  softmax softmax_i (
    .clk(clk),
    .reset(reset),
    .start(max_valid[4]),
    .max(max),
    .act_in(fc2_act_o_buf),

    .valid_o(sm_valid_o),
    .grad_o(sm_grad_o)
  );

  bit [3: 0] u;
  always_ff @(posedge clk) begin
    if (reset) begin
      fc2_gradients_rdy      <= 0;
    end
    else if (all_idle) begin
      fc2_gradients_rdy      <= 1'b0;
    end
    else if (sm_valid_o) begin
      fc2_gradients_rdy      <= 1'b1;
    end

    if (sm_valid_o) begin
      for (u = 0; u < `FC2_NEURONS; u=u+1) begin
        fc2_gradients[u]   <= (fc2_act_o_buf[img_label] == `MIN_VAL)
             ? 0 :
```

```
                              sm_grad_o[u];
        end
        fc2_gradients[img_label]  <= (fc2_act_o_buf[img_label] == `
            MAX_VAL) ? 0 :
                              $signed(sm_grad_o[img_label]) - $signed(`ONE)
                                  ;
      end
    end


    assign status_block = {5'b0, led_o_r, fc0_state, fc1_state,
        fc2_state, forward, fc0_start,
               fc1_start, fc2_start, fc0_busy, fc1_busy, fc2_busy,
                   new_img,
               all_idle, img_rdy};


    logic [31:0]img1_blk0_0;
    logic [31:0]img1_blk100_0;
    logic [31:0]img1_blk101_0;
...


system_wrapper system_wrapper_i
     (DDR_addr,
    DDR_ba,
    DDR_cas_n,
    DDR_ck_n,
    DDR_ck_p,
    DDR_cke,
    DDR_cs_n,
    DDR_dm,
    DDR_dq,
    DDR_dqs_n,
    DDR_dqs_p,
    DDR_odt,
    DDR_ras_n,
    DDR_reset_n,
    DDR_we_n,
    fab_clk,
    FIXED_IO_ddr_vrn,
    FIXED_IO_ddr_vrp,
    FIXED_IO_mio,
    FIXED_IO_ps_clk,
    FIXED_IO_ps_porb,
    FIXED_IO_ps_srstb,
    active_cycles,
    epoch,
    img_id,
    idle_cycles,
    img1_blk0_0,
    img1_blk100_0,
...
    img1_id,
    img1_label,
```

```
  img_set_size,
  lrate_shifts_bus,
  n_epochs,
  num_correct_test,
  num_correct_train,
  {fc2_out[1][17:2], fc2_out[0][17:2]},
  {fc2_out[3][17:2], fc2_out[2][17:2]},
  {fc2_out[5][17:2], fc2_out[4][17:2]},
  {fc2_out[7][17:2], fc2_out[6][17:2]},
  {fc2_out[9][17:2], fc2_out[8][17:2]},
  start_bus,
  status_block,
  training_mode_bus);

  always_ff @(posedge clk) begin
    if (reset) begin
      img_id      <= img_set_size;
      img_label   <= 0;
    end
    else if (new_img) begin
      img_id      <= img1_id;
      img_label   <= img1_label;
    end
    if (new_img) begin
      img1_unpacked[0]   <= img1_blk0_0[7:0];
      img1_unpacked[1]   <= img1_blk0_0[15:8];
      img1_unpacked[2]   <= img1_blk0_0[23:16];
      img1_unpacked[3]   <= img1_blk0_0[31:24];
      img1_unpacked[4]   <= img1_blk1_0[7:0];
...
    end
  end

endmodule
```

B.1.3 fc0_layer.sv

```
`timescale 1ns / 1ps
`include "sys_defs.vh"

module fc0_layer(
	input clk,
	input rst,
	input forward,
	input update,
	input  [1:0][`PREC - 1:0] activations_i,
	input valid_i,
	input  [4:0] lrate_shifts,

	input [`FC0_NEURONS - 1:0][`PREC - 1:0] b_gradient_i,
	input [`FC0_N_KERNELS - 1:0][`PREC - 1:0] b_activation_i,
	input [9:0] b_activation_id,
	input  b_valid_i,

	output logic [`FC0_NEURONS - 1:0][`PREC - 1:0] activation_o,
	output logic [`FC0_NEURONS - 1:0][6:0] neuron_id_o,
	output logic valid_act_o,
	output logic fc0_busy,
	output logic bp_done,
	output logic update_done
);

logic	[`FC0_PORT_WIDTH - 1:0][`PREC - 1:0]	data_in_a;
logic	[`FC0_PORT_WIDTH - 1:0][`PREC - 1:0]	data_in_b;
logic	[`FC0_PORT_WIDTH - 1:0][`PREC - 1:0]	data_out_a;
logic	[`FC0_PORT_WIDTH - 1:0][`PREC - 1:0]	data_out_b;

logic	[`FC0_N_KERNELS - 1:0][`PREC - 1:0]	weights;
logic	[`FC0_ADDR - 1:0]			head_ptr;
logic	[`FC0_ADDR - 1:0]			mid_ptr;
logic	[`FC0_ADDR - 1:0]			addr_a;
logic	[`FC0_ADDR - 1:0]			addr_b;
logic	[`FC0_BIAS_ADDR - 1:0]			bias_ptr;

logic	[1:0][`PREC - 1:0]			sch_activations;
logic						sch_valid;
logic	[1:0][`PREC - 1:0]			bram_activations;
logic						bram_valid;
logic	[`FC0_N_KERNELS - 1:0][`PREC - 1:0]	kern_activations;
logic						kern_valid;

logic	[`FC0_N_KERNELS - 1:0][`PREC - 1:0]	bias;
logic	[`FC0_N_KERNELS - 1:0][`PREC - 1:0]	kern_bias;
logic	[255:0]					bias_container;
logic						sch_has_bias;
logic						bram_has_bias;
logic						kern_has_bias;
logic	[`FC0_NEURONS - 1:0][6:0]		neuron_id;
logic	[`FC0_N_KERNELS - 1:0][6:0]		kern_neuron_id;
```

```
  logic    [`FC0_N_KERNELS - 1: 0]                        last_weight;

  logic    [`FC0_N_KERNELS - 1: 0]                        valid;
  logic    [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]   kern_activation_o
      ;
  logic    [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]   activation_o_rel;
  logic    [`FC0_N_KERNELS - 1: 0][6: 0]               kern_neuron_id_o;


  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      b_gradient;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      b_gradient_pl;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      b_kern_grad;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      b_act;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      b_act_pl;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      b_kern_act;

  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      b_kern_grad_o;
  logic [`FC0_N_KERNELS - 1: 0]                      b_kern_valid_o;
  logic [2: 0]                                       b_valid;
  logic [3: 0][9: 0]                                 b_act_id;

  logic                                              b_weight_we;

  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      kern_mult1;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      kern_mult2;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      weight_grad_o;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      weight_grad;
  logic [1: 0][9: 0]
      fc0_weight_grad_addr;
  logic [1: 0][9: 0]
      fc0_weight_grad_addr_offset;
  logic [`FC0_NEURONS - 1: 0]                       act_o_sign;
  logic [`FC0_N_KERNELS - 1: 0][`PREC: 0]
      update_weights_sat;
  logic [`FC0_N_KERNELS - 1: 0][`PREC - 1: 0]      update_weights;
  logic [10: 0]                                      update_ptr;
  logic [9: 0]                                       update_addr_a;
  logic [9: 0]                                       update_addr_b;
  logic [9: 0]                                       w_addr_a;
  logic [9: 0]                                       w_addr_b;
  logic [9: 0]                                       wg_addr_a;
  logic [9: 0]                                       wg_addr_b;
  logic                                              w_we;
  logic                                              wg_we;

  logic                                              sch_valid_i;
  localparam WEIGHT_MODE = 0;
  localparam NEURON_MODE = 1;
  logic bp_mode;
  assign bp_mode = WEIGHT_MODE;
  always_ff @(posedge clk) begin
    if (rst) begin
      sch_valid     <= 0;
    end
    else begin
```

```
      sch_valid      <= valid_i;
    end
    sch_activations <= activations_i;
  end

  assign sch_valid_i = (forward) ? valid_i : b_valid_i;

  // Scheduler for the fully connected layer
  fc_scheduler #(.ADDR(`FC0_ADDR), .BIAS_ADDR(`FC0_BIAS_ADDR),
  .MID_PTR_OFFSET(`FC0_KERNEL_FAN_IN), .FAN_IN(`FC0_FAN_IN))
  fc0_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)
  );



  always_ff @(posedge clk) begin
    if (rst) begin
      bram_valid     <= 0;
      bram_has_bias    <= 0;
      fc0_busy     <= 0;
    end
    else begin
      bram_valid     <= sch_valid;
      bram_has_bias    <= sch_has_bias;
      fc0_busy     <= valid_i;
    end
    bram_activations   <= sch_activations;
  end



  always_ff @(posedge clk) begin
    if (rst) begin
      update_ptr   <= 0;
    end
    else if (update) begin
      update_ptr   <= update_ptr + 1'b1;
    end
    else begin
      update_ptr   <= 0;
    end
  end
```

```


  assign update_done     = update_ptr == 11'd783;
  assign update_addr_a   = update_ptr[10: 1] << 1;
  assign update_addr_b   = update_addr_a + 1'b1;
  assign w_addr_a        = (update) ? update_addr_a  : addr_a;
  assign w_addr_b        = (update) ? update_addr_b  : addr_b;
  assign wg_addr_a       = (update) ? update_addr_a  : fc0_weight_grad_addr[0];
  assign wg_addr_b       = (update) ? update_addr_b  : fc0_weight_grad_addr[1];
  assign w_we            = (update) ? update_ptr[0]  : 1'b0;  // write when odd
  assign wg_we           = (update) ? 1'b0        : b_weight_we;
  assign addr_a          = (head_ptr << 1);
  assign addr_b          = (head_ptr << 1) + 1'b1;
  bit [7: 0] a,c;
  always_comb begin
    weight_grad = 0;
    for (a = 0, c =`FC0_PORT_WIDTH; a < `FC0_PORT_WIDTH; a = a + 1, c=c+1) begin
      case(lrate_shifts)
        5'd7: begin
          weight_grad[a] = {{7{weight_grad_o[a][`PREC - 1]}}, {weight_grad_o[a][`PREC - 1: 7]}};
          weight_grad[c] = {{7{weight_grad_o[c][`PREC - 1]}}, {weight_grad_o[c][`PREC - 1: 7]}};
        end

        5'd9: begin
          weight_grad[a] = {{9{weight_grad_o[a][`PREC - 1]}}, {weight_grad_o[a][`PREC - 1: 9]}};
          weight_grad[c] = {{9{weight_grad_o[c][`PREC - 1]}}, {weight_grad_o[c][`PREC - 1: 9]}};
        end
        5'd11: begin
          weight_grad[a] = {{11{weight_grad_o[a][`PREC - 1]}}, {weight_grad_o[a][`PREC - 1: 11]}};
          weight_grad[c] = {{11{weight_grad_o[c][`PREC - 1]}}, {weight_grad_o[c][`PREC - 1: 11]}};
        end
        5'd10: begin
          weight_grad[a] = {{10{weight_grad_o[a][`PREC - 1]}}, {weight_grad_o[a][`PREC - 1: 10]}};
          weight_grad[c] = {{10{weight_grad_o[c][`PREC - 1]}}, {weight_grad_o[c][`PREC - 1: 10]}};
        end
        default: begin
          weight_grad[a] = {{8{weight_grad_o[a][`PREC - 1]}}, {weight_grad_o[a][`PREC - 1: 8]}};
          weight_grad[c] = {{8{weight_grad_o[c][`PREC - 1]}}, {weight_grad_o[c][`PREC - 1: 8]}};
        end
      endcase
```

```
      update_weights_sat[a]   = $signed(data_out_a[a]) - $signed(weight_grad[a]);
      update_weights_sat[c]   = $signed(data_out_b[a]) - $signed(weight_grad[c]);
    end
  end

  bit [7:0] d;
  always_comb begin
    for (d = 0; d < `FC0_N_KERNELS; d=d+1) begin
      if (update_weights_sat[d][`PREC:`PREC - 1] == 2'b01) begin
        update_weights[d]   = `MAX_VAL;
      end
      else if (update_weights_sat[d][`PREC:`PREC - 1] == 2'b10) begin
        update_weights[d]   = `MIN_VAL;
      end
      else begin
        update_weights[d]   = update_weights_sat[d][`PREC - 1:0];
      end
    end
  end


  // BRAM for the weights of the fully connected layer
  fc0_weight_bram_controller fc0_weight_bram_controller_i (
    // inputs
    .clk(clk),
    .rst(rst),

    .addr_a(w_addr_a),
    .data_in_a(update_weights[97:0]),
    .en_a(1'b1),
    .we_a(w_we),

    .addr_b(w_addr_b),
    .data_in_b(update_weights[195:98]),
    .en_b(1'b1),
    .we_b(w_we),

    // outputs
    .data_out_a(data_out_a),
    .data_out_b(data_out_b),
    .neuron_id(neuron_id)
  );

  assign b_weight_we = &b_kern_valid_o;

  assign fc0_weight_grad_addr_offset[0] = 0;
  assign fc0_weight_grad_addr_offset[1] = fc0_weight_grad_addr_offset[0] + 1'b1;
  assign fc0_weight_grad_addr[0]        = fc0_weight_grad_addr_offset[0] + b_act_id[3];
  assign fc0_weight_grad_addr[1]        = fc0_weight_grad_addr_offset[1] + b_act_id[3];
```

```

  assign bp_done = fc0_weight_grad_addr[1] == `FC0_FAN_IN - 1'b1;

  fc0_weight_gradients fc0_weight_gradients_i (
    .addra(wg_addr_a),
    .clka(clk),
    .dina(b_kern_grad_o[97: 0]),
    .douta(weight_grad_o[97: 0]),
    .ena(1'b1),
    .wea(wg_we),

    .addrb(wg_addr_b),
    .clkb(clk),
    .dinb(b_kern_grad_o[195: 98]),
    .doutb(weight_grad_o[195: 98]),
    .enb(1'b1),
    .web(wg_we)
  );

  assign bias = 0;


  always_ff @(posedge clk) begin
    if (rst) begin
      kern_valid       <= 0;
      kern_has_bias    <= 0;
    end
    else begin
      kern_valid       <= bram_valid;
      kern_has_bias    <= bram_has_bias;
    end
    kern_activations   <= {{`FC0_NEURONS{bram_activations[1]}}, {`
        FC0_NEURONS{bram_activations[0]}}};
    kern_bias          <= 0;//bias;
    kern_neuron_id     <= {2{neuron_id}};
    weights            <= {data_out_b, data_out_a};
  end


  assign kern_mult1   =   (forward) ? weights          : b_kern_grad;

  assign kern_mult2   =   (forward) ? kern_activations : b_kern_act
      ;

  // Computational kernel for the fully connected layer
  genvar i;
  generate
    for (i = 0; i < `FC0_N_KERNELS; i=i+1) begin
      fc_kernel #(.FAN_IN(`FC0_KERNEL_FAN_IN), .ID_WIDTH(7))
          fc_kernel_i (
        // input
        .clk(clk),
        .rst(rst),
        .activation_i(kern_mult2[i]),
        .weight(kern_mult1[i]),
```

```
        .bias(18'b0),
        .neuron_id_i(kern_neuron_id[i]),
        .has_bias(kern_has_bias),
        .valid_i(kern_valid),
        .b_valid_i(b_valid[2]),
        .bp_mode(bp_mode),
        // output
        .b_gradient_o(b_kern_grad_o[i]),
        .b_valid_o(b_kern_valid_o[i]),
        .activation_o(kern_activation_o[i]),
        .neuron_id_o(kern_neuron_id_o[i]),
        .valid_o(valid[i])
      );
    end
  endgenerate

  bit [7: 0] b;
  always_ff @(posedge clk) begin
    if (&valid) begin
      for (b = 0; b < `FC0_NEURONS; b = b + 1) begin
        act_o_sign[neuron_id_o[b]]   <= activation_o_rel[b][`PREC - 1];
      end
    end
  end


  assign valid_act_o  = &valid;
  assign neuron_id_o  = kern_neuron_id_o[`FC0_NEURONS - 1: 0];

  bit [8: 0] m, n;
  always_comb begin
    for (m = 0, n = `FC0_NEURONS; m < `FC0_NEURONS; m=m+1, n=n+1) begin
      activation_o_rel[m] = $signed(kern_activation_o[m]) + $signed(kern_activation_o[n]);
      activation_o[m] = activation_o_rel[m][`PREC - 1] ? 0 : activation_o_rel[m];
    end
  end


   bit [7: 0] q, w;
  // Backward pass logic
  always_ff @(posedge clk) begin
    for (q = 0, w = `FC0_NEURONS; q < `FC0_NEURONS; q = q + 1, w = w+1) begin
      b_gradient[q]   <= act_o_sign[q] ? 0 : b_gradient_i[q];
      b_gradient[w]   <= act_o_sign[q] ? 0 : b_gradient_i[q];
    end
    b_gradient_pl  <= b_gradient;
    b_kern_grad    <= b_gradient_pl;

    b_act          <= b_activation_i;
    b_act_pl       <= b_act;
```

```
    b_kern_act     <= b_act_pl;


    b_act_id       <= {b_act_id[2:0], b_activation_id};
    b_valid        <= {b_valid[1: 0], b_valid_i};
  end
endmodule
```

B.1.4 fc0_weight_bram_controller.sv

```
`timescale 1ns / 1ps

`include "sys_defs.vh"

module fc0_weight_bram_controller (
    input                                                   clk,
    input                                                   rst,

    input   [`FC0_ADDR - 1: 0]                              addr_a,
    input   [`FC0_PORT_WIDTH - 1: 0][`PREC - 1: 0]          data_in_a,
    input                                                   en_a,
    input                                                   we_a,

    input   [`FC0_ADDR - 1: 0]                              addr_b,
    input   [`FC0_PORT_WIDTH - 1: 0][`PREC - 1: 0]          data_in_b,
    input                                                   en_b,
    input                                                   we_b,

    output logic [`FC0_PORT_WIDTH - 1: 0][`PREC - 1: 0] data_out_a,
    output logic [`FC0_PORT_WIDTH - 1: 0][`PREC - 1: 0] data_out_b,
    output logic [`FC0_PORT_WIDTH - 1: 0][6: 0]         neuron_id

    );

    bit [6: 0]  i, j;
    always_ff @(posedge clk) begin
        for (i = 0; i < `FC0_PORT_WIDTH; i=i+1) begin
            neuron_id[i]    <= i;
        end
    end

    fc0_weights_bram fc0_weights_bram_i (
        .addra(addr_a),
        .clka(clk),
        .dina(data_in_a),
        .douta(data_out_a),
        .ena(en_a),
        .wea(we_a),

        .addrb(addr_b),
        .clkb(clk),
        .dinb(data_in_b),
        .doutb(data_out_b),
        .enb(en_b),
        .web(we_b)
    );

endmodule
```

B.1.5 fc_kernel.sv

```
`timescale 1ns / 1ps
`include "sys_defs.vh"

module fc_kernel #(
    parameter FAN_IN = 0,
    parameter ID_WIDTH = 0
)(
    input                             clk,
    input                             rst,
    input   [`PREC - 1: 0]            activation_i,
    input   [`PREC - 1: 0]            weight,
    input   [`PREC - 1: 0]            bias,
    input   [ID_WIDTH - 1: 0]         neuron_id_i,
    input                             has_bias,
    input                             valid_i,
    input                             b_valid_i,
    input                             bp_mode,

    output logic [`PREC - 1: 0]       b_gradient_o,
    output logic                      b_valid_o,
    output logic [`PREC - 1: 0]       activation_o,
    output logic [ID_WIDTH - 1: 0]    neuron_id_o,
    output logic                      valid_o
);

    logic [31: 0]                 dsp_o;

    logic [ID_WIDTH - 1: 0]       neuron_id;
    logic [ID_WIDTH - 1: 0]       prev_neuron_id_i;
    logic                         valids;
    logic [31: 0]                 kernel_in;


    logic [35: 0]                 mult_res;
    logic [33: 0]                 mac_res;

    logic                         last;
    logic                         prev_valid_i;
    logic [8: 0]                  cnt;


    localparam WEIGHT_MODE = 0;
    localparam NEURON_MODE = 1;


    always_ff @(posedge clk) begin
        if (valid_i) begin
            cnt     <= (cnt == FAN_IN - 1) ? 0 : cnt + 1'b1;
            last    <= cnt == FAN_IN - 1;
        end
        else begin
            cnt     <= 0;
            last    <= cnt == FAN_IN - 1;
```

```
        end
    end

    always_ff @(posedge clk) begin
        prev_neuron_id_i     <= neuron_id_i;
    end

    always_ff @(posedge clk) begin
        if (bp_mode == WEIGHT_MODE) begin
            if ({mult_res[35], &mult_res[34: 30]} == 2'b10) begin
            // negative saturation
                b_gradient_o     <= `MIN_VAL;
            end
            else if ({mult_res[35], |mult_res[34: 30]} == 2'b01)
                begin
            // positive saturation
                b_gradient_o     <= `MAX_VAL;
            end
            else begin
                b_gradient_o     <= mult_res[29: 12];
            end
        end
        else begin
             if ({mult_res[35], mult_res[34]} == 2'b10) begin
             // negative saturation
                b_gradient_o     <= `MIN_VAL;
            end
            else if ({mult_res[35], mult_res[34]} == 2'b01) begin
            // positive saturation
                b_gradient_o     <= `MAX_VAL;
            end
            else begin
                b_gradient_o     <= mult_res[34: 17];
            end
        end

        b_valid_o        <= b_valid_i;
    end



    always_ff @(posedge clk) begin
        if (last) begin
            activation_o     <= dsp_o[31: 14];
            neuron_id_o      <= prev_neuron_id_i;
            valid_o          <= 1'b1;
        end
        else begin
            valid_o          <= 1'b0;
        end
    end

    assign kernel_in     = has_bias ? {14'b0, bias} : dsp_o;
    assign mult_res      = $signed(weight) * $signed(activation_i);
```

```
    assign mac_res       = $signed(mult_res[35:3]) + $signed(
        kernel_in);

    always_ff @(posedge clk) begin
        if ({mac_res[33], &mac_res[32: 31]} == 2'b10) begin
        // negative saturation
            dsp_o    <= 32'h8000_0000;
        end
        else if ({mac_res[33], |mac_res[32: 31]} == 2'b01) begin
        // positive saturation
            dsp_o    <= 32'h7FFF_FFFF;
        end
        else begin
            dsp_o    <= mac_res[31: 0];
        end
    end
endmodule
```

B.1.6 fc_scheduler.sv

```
`timescale 1ns / 1ps

module fc_scheduler #(
    parameter ADDR              = 0,
    parameter BIAS_ADDR         = 0,
    parameter MID_PTR_OFFSET    = 0,
    parameter FAN_IN            = 0
)(
    input                               clk,
    input                               rst,
    input                               forward,
    input                               valid_i,

    output logic    [ADDR - 1: 0]       head_ptr,
    output logic    [ADDR - 1: 0]       mid_ptr,
    output logic    [BIAS_ADDR - 1: 0]  bias_ptr,
    output logic                        has_bias

);
    logic                       start;
    logic   [ADDR - 1: 0]       h_thresh;
    logic   [ADDR - 1: 0]       next_head_ptr;
    logic   [ADDR - 1: 0]       next_mid_ptr;
    logic   [BIAS_ADDR - 1: 0]  next_bias_ptr;
    logic                       prev_forw;
    logic                       mode_switch;


    assign h_thresh         = MID_PTR_OFFSET - 2;
    assign mode_switch      = prev_forw ^ forward;

    assign next_head_ptr = (mode_switch || !start)   ? 0 :
                (!valid_i) ? head_ptr   : head_ptr + 1'b1;
    assign next_mid_ptr = (mode_switch || !start) ? MID_PTR_OFFSET :
                (!valid_i) ? mid_ptr : mid_ptr + 1'b1;
    assign next_bias_ptr = (mode_switch || !start)   ? 0 :
                (!valid_i) ? bias_ptr   : bias_ptr + 1'b1;



    always_ff @(posedge clk) begin
        head_ptr     <= next_head_ptr;
        mid_ptr      <= next_mid_ptr;
        prev_forw    <= forward;
    end

    logic [ADDR - 1: 0] bias_cntr;
    always_ff @(posedge clk) begin
        if (rst) begin
            bias_cntr    <= 0;
        end
        else if (valid_i && forward) begin
```

```
            bias_cntr   <= (bias_cntr == FAN_IN - 1) ? 0 :
                bias_cntr + 1'b1;
        end
        else begin
            bias_cntr   <= 0;
        end
    end


    always_ff @(posedge clk) begin
        if (rst) begin
            has_bias    <= 0;
            bias_ptr    <= 0;
        end
        else if (valid_i && bias_cntr == 0 && forward) begin
            has_bias    <= 1'b1;
            bias_ptr    <= next_bias_ptr;
        end
        else begin
            has_bias    <= 1'b0;
            bias_ptr    <= bias_ptr;
        end
    end

    always_ff @(posedge clk) begin
        if (rst) begin
            start   <= 1'b0;
        end
        else if (valid_i && !start) begin
            start   <= 1'b1;
        end
        else if (valid_i && head_ptr == h_thresh) begin
            start   <= 1'b0;
        end
        else if (mode_switch) begin
            start   <= 1'b0;
        end
    end

endmodule
```

B.1.7 interlayer_activation_buffer.sv

```
`timescale 1ns / 1ps

module interlayer_activation_buffer #(
    parameter N_KERNELS_I = 0,
    parameter N_KERNELS_O = 0,
    parameter ID_WIDTH = 0,
    parameter BUFF_SIZE = 0,
    parameter LOOPS = 0
)(
    input                                              clk,
    input                                              rst,
    input                                              start,
    input [N_KERNELS_I - 1: 0][`PREC - 1: 0]           activation_i,
    input [N_KERNELS_I - 1: 0][ID_WIDTH - 1: 0]        neuron_id_i,
    input                                              valid_act_i,
    input  [ID_WIDTH - 1: 0]                           b_ptr,


    output logic [N_KERNELS_O - 1: 0][`PREC - 1: 0] activation_o,
    output logic                                       valid_o,
    output logic [`PREC - 1: 0]                        b_act_o,
    output logic                                       buff_rdy
);
    logic   [ID_WIDTH - 1: 0]                      buff_ptr;
    logic   [BUFF_SIZE - 1: 0][`PREC - 1: 0]       buffer;
    logic                                          read_o;
    logic   [LOOPS: 0]                             loop_cnt;


    bit [ID_WIDTH: 0] i;
    always_ff @(posedge clk) begin
        if (rst) begin
            buff_rdy     <= 0;
        end
        else if (valid_act_i) begin
            if (!read_o && neuron_id_i[N_KERNELS_I - 1] ==
                BUFF_SIZE - 1) begin
                buff_rdy     <= 1'b1;
            end
        end
        if (valid_act_i) begin
            for (i = 0; i < N_KERNELS_I; i=i+1) begin
                buffer[neuron_id_i[i]]   <= activation_i[i];
            end
        end
        if (read_o) begin
            buff_rdy         <= 1'b0;
        end
    end

    always_ff @(posedge clk) begin
        if (rst) begin
            read_o       <= 1'b0;
```

```
                buff_ptr     <= 0;
            end
            else if (buff_rdy && start && !read_o) begin
                read_o       <= 1'b1;
                buff_ptr     <= 0;
            end
            else if (read_o) begin
                read_o       <= ~((buff_ptr == (BUFF_SIZE - 1'b1)) & (
                    loop_cnt == LOOPS - 1));
                buff_ptr     <= (buff_ptr == (BUFF_SIZE - 1'b1)) ? 0 :
                    buff_ptr + 1'b1;
            end

            if (rst) begin
                loop_cnt     <= 0;
            end
            else if(~read_o) begin
                loop_cnt     <= 0;
            end
            else if (buff_ptr == BUFF_SIZE - 1'b1) begin
                loop_cnt     <= loop_cnt + 1'b1;
            end
        end

        bit [ID_WIDTH - 1: 0] j;
        always_ff @(posedge clk) begin
            for (j = 0; j < N_KERNELS_O; j=j+1) begin
                activation_o[j] <= buffer[buff_ptr];
            end
            valid_o <= read_o;
        end

        always_ff @(posedge clk) begin
            b_act_o <= buffer[b_ptr];
        end
    endmodule
```

B.1.8 fc1_layer.sv

```
`timescale 1ns / 1ps
`include "sys_defs.vh"

module fc1_layer(
  input                                             clk,
  input                                             rst,
  input                                             forward,
  input                                             update,
  input  [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]      activations_i,
  input                                             valid_i,
  input  [4: 0]                                     lrate_shifts,


  input [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]       b_gradient_i,
  input [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]       b_activation_i,
  input [6: 0]                                      b_activation_id,
  input [`FC1_N_KERNELS - 1: 0][5: 0]               b_neuron_id_i,
  input                                             b_valid_i,
  input                                             bp_mode,


  output logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]  activation_o,
  output logic [`FC1_N_KERNELS - 1: 0][5: 0]          neuron_id_o,
  output logic                                        valid_act_o,
  output logic                                        fc1_busy,
  output logic                                        bp_done,
  output logic                                        update_done,

  output logic [`FC0_NEURONS - 1: 0][`PREC - 1: 0]    pl_gradients,
  output logic                                        pl_grad_valid

);

  logic   [`FC1_PORT_WIDTH - 1: 0][`PREC - 1: 0]  data_in_a;
  logic   [`FC1_PORT_WIDTH - 1: 0][`PREC - 1: 0]  data_in_b;
  logic   [`FC1_PORT_WIDTH - 1: 0][`PREC - 1: 0]  data_out_a;
  logic   [`FC1_PORT_WIDTH - 1: 0][`PREC - 1: 0]  data_out_b;

  logic   [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]   weights;
  logic   [`FC1_ADDR - 1: 0]                      head_ptr;
  logic   [`FC1_ADDR - 1: 0]                      mid_ptr;
  logic   [`FC1_BIAS_ADDR - 1: 0]                 bias_ptr;

  logic   [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]   sch_activations;
  logic                                           sch_valid;
  logic   [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]   bram_activations;
  logic                                           bram_valid;
  logic   [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]   kern_activations;
  logic                                           kern_valid;
```

```
    logic   [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]   bias;
    logic   [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]   kern_bias;
    logic   [255: 0]                                bias_container;
    logic                                           sch_has_bias;
    logic                                           bram_has_bias;
    logic                                           kern_has_bias;
    logic   [`FC1_N_KERNELS - 1: 0][5: 0]           neuron_id;
    logic   [`FC1_N_KERNELS - 1: 0][5: 0]           kern_neuron_id;
    logic   [`FC1_N_KERNELS - 1: 0]                 last_weight;

    logic   [`FC1_N_KERNELS - 1: 0]                 valid;

    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     kern_activation_o;

    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     b_gradient;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     b_gradient_pl;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     b_kern_grad;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     b_act;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     b_act_pl;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     b_kern_act;

    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     b_kern_grad_o;
    logic [`FC1_N_KERNELS - 1: 0]                   b_kern_valid_o;
    logic [2: 0]                                    b_valid;
    logic [3: 0][6: 0]                              b_act_id;
    logic [3: 0][`FC1_N_KERNELS - 1: 0][5: 0]       b_neuron_id;

    logic                                           b_kern_valid;
    logic                                           b_weight_we;

    logic                                           sch_bp_mode;
    logic                                           bram_bp_mode;
    logic                                           kern_bp_mode;
    logic                                           kern_bp_mode_o;

    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     kern_mult1;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     kern_mult2;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     weight_grad;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     weight_grad_o;
    logic [1: 0][9: 0]                              fc1_weight_grad_addr;
    logic [1: 0][9: 0]                              fc1_weight_grad_addr_offset;
    logic [`FC1_NEURONS - 1: 0]                     act_o_sign;
    logic [`FC1_N_KERNELS - 1: 0][`PREC: 0]         update_weights_sat;
    logic [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]     update_weights;

    logic [10: 0]                                   update_ptr;
    logic [9: 0]                                    update_addr_a;
    logic [9: 0]                                    update_addr_b;
    logic [9: 0]                                    w_addr_a;
    logic [9: 0]                                    w_addr_b;
```

```
logic [9: 0]                                              wg_addr_a;
logic [9: 0]                                              wg_addr_b;
logic                                                     w_we;
logic                                                     wg_we;
logic                                                     sch_valid_i;

localparam WEIGHT_MODE = 0;
localparam NEURON_MODE = 1;

always_ff @(posedge clk) begin
  if (rst) begin
    sch_valid     <= 0;
    sch_bp_mode   <= 0;
  end
  else begin
    sch_valid     <= valid_i;
    sch_bp_mode   <= bp_mode;
  end
  sch_activations <= activations_i;
end


assign sch_valid_i = (forward) ? valid_i : b_valid_i & bp_mode ==
     NEURON_MODE;

// Scheduler for the fully connected layer
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)
);




always_ff @(posedge clk) begin
  if (rst) begin
    bram_activations   <= 0;
    bram_valid         <= 0;
    bram_has_bias      <= 0;
    fc1_busy           <= 0;
    bram_bp_mode       <= 0;
  end
  else begin
    bram_activations   <= sch_activations;
```

```
      bram_valid         <= sch_valid;
      bram_has_bias      <= sch_has_bias;
      fc1_busy           <= valid_i;
      bram_bp_mode       <= sch_bp_mode;
    end
  end




  always_ff @(posedge clk) begin
    if (rst) begin
      update_ptr   <= 0;
    end
    else if (update) begin
      update_ptr   <= update_ptr + 1'b1;
    end
    else begin
      update_ptr   <= 0;
    end
  end


  assign update_done     = update_ptr == 11'd783;
  assign update_addr_a   = update_ptr[10: 1] << 1;
  assign update_addr_b   = update_addr_a + 1'b1;
  assign w_addr_a        = (update) ? update_addr_a   : head_ptr;
  assign w_addr_b        = (update) ? update_addr_b   : mid_ptr;
  assign wg_addr_a       = (update) ? update_addr_a   :
      fc1_weight_grad_addr[0];
  assign wg_addr_b       = (update) ? update_addr_b   :
      fc1_weight_grad_addr[1];
  assign w_we            = (update) ? update_ptr[0]   : 1'b0;  //
      write when odd
  assign wg_we           = (update) ? 1'b0        : b_weight_we;

  bit [4: 0] a,c;
  always_comb begin
    for (a = 0, c =`FC1_PORT_WIDTH; a < `FC1_PORT_WIDTH; a = a + 1,
         c=c+1) begin
      case(lrate_shifts)
        5'd7: begin
          weight_grad[a] = {{7{weight_grad_o[a][`PREC - 1]}}, {
              weight_grad_o[a][`PREC - 1: 7]}};
          weight_grad[c] = {{7{weight_grad_o[c][`PREC - 1]}}, {
              weight_grad_o[c][`PREC - 1: 7]}};
        end

        5'd9: begin
          weight_grad[a] = {{9{weight_grad_o[a][`PREC - 1]}}, {
              weight_grad_o[a][`PREC - 1: 9]}};
          weight_grad[c] = {{9{weight_grad_o[c][`PREC - 1]}}, {
              weight_grad_o[c][`PREC - 1: 9]}};
        end
        5'd11: begin
```

```
        weight_grad[a] = {{11{weight_grad_o[a][`PREC - 1]}}, {weight_grad_o[a][`PREC - 1: 11]}};
        weight_grad[c] = {{11{weight_grad_o[c][`PREC - 1]}}, {weight_grad_o[c][`PREC - 1: 11]}};
      end
      5'd10: begin
        weight_grad[a] = {{10{weight_grad_o[a][`PREC - 1]}}, {weight_grad_o[a][`PREC - 1: 10]}};
        weight_grad[c] = {{10{weight_grad_o[c][`PREC - 1]}}, {weight_grad_o[c][`PREC - 1: 10]}};
      end
      default: begin
        weight_grad[a] = {{8{weight_grad_o[a][`PREC - 1]}}, {weight_grad_o[a][`PREC - 1: 8]}};
        weight_grad[c] = {{8{weight_grad_o[c][`PREC - 1]}}, {weight_grad_o[c][`PREC - 1: 8]}};
      end
    endcase
    update_weights_sat[a]   = $signed(data_out_a[a]) - $signed(weight_grad[a]);
    update_weights_sat[c]   = $signed(data_out_b[a]) - $signed(weight_grad[c]);
  end
end

bit [7: 0] d;
always_comb begin
  for (d = 0; d < `FC1_N_KERNELS; d=d+1) begin
    if (update_weights_sat[d][`PREC:`PREC - 1] == 2'b01) begin
      update_weights[d]   = `MAX_VAL;
    end
    else if (update_weights_sat[d][`PREC:`PREC - 1] == 2'b10) begin
      update_weights[d]   = `MIN_VAL;
    end
    else begin
      update_weights[d]   = update_weights_sat[d][`PREC - 1: 0];
    end
  end
end

// BRAM for the weights of the fully connected layer
fc1_weight_bram_controller fc1_weight_bram_controller_i (
  // inputs
  .clk(clk),
  .rst(rst),

  .addr_a(w_addr_a),
  .data_in_a(update_weights[7: 0]),
  .en_a(1'b1),
  .we_a(w_we),

  .addr_b(w_addr_b),
  .data_in_b(update_weights[15: 8]),
  .en_b(1'b1),
```

```
    .we_b(w_we),

    // outputs
    .data_out_a(data_out_a),
    .data_out_b(data_out_b),
    .neuron_id(neuron_id)
  );


  /*
  biases_fc1_blk_mem_gen_1 biases_fc1_blk_mem_gen_1_i (
    .addra(bias_ptr),
    .clka(clk),
    .dina(256'b0),
    .douta(bias),
    .ena(1'b1),
    .wea(1'b0)
  );*/
  assign bias = 0;
  assign b_weight_we = &b_kern_valid_o & kern_bp_mode_o ==
      WEIGHT_MODE;

  assign fc1_weight_grad_addr_offset[0] = ({6'b0, b_neuron_id
      [3][0][5:3]} << 6) +
                        ({6'b0, b_neuron_id[3][0][5:3]} << 5) +
                        ({6'b0, b_neuron_id[3][0][5:3]} << 1);
  assign fc1_weight_grad_addr_offset[1] =
      fc1_weight_grad_addr_offset[0] + `FC1_MID_PTR_OFFSET;

  assign fc1_weight_grad_addr[0] = fc1_weight_grad_addr_offset[0] +
       b_act_id[3];
  assign fc1_weight_grad_addr[1] = fc1_weight_grad_addr_offset[1] +
       b_act_id[3];

  fc1_weight_gradients fc1_weight_gradients_i (
    .addra(wg_addr_a),
    .clka(clk),
    .dina(b_kern_grad_o[7: 0]),
    .douta(weight_grad_o[7: 0]),
    .ena(1'b1),
    .wea(wg_we),

    .addrb(wg_addr_b),
    .clkb(clk),
    .dinb(b_kern_grad_o[15:8]),
    .doutb(weight_grad_o[15:8]),
    .enb(1'b1),
    .web(wg_we)
  );


  previous_layer_gradient_adder previous_layer_gradient_adder_i (
    // inputs
    .clk(clk),
    .rst(rst),
```

```
    .forward(forward),
    .grad_i(b_kern_grad_o),
    .neuron_id_i(b_act_id[3]),
    .valid_i(&b_kern_valid_o),
    .bp_mode_i(kern_bp_mode_o),

    // outputs
    .pl_gradients(pl_gradients),
    .pl_grad_valid(pl_grad_valid)
  );

  always_ff @(posedge clk) begin
    if (rst) begin
      kern_valid       <= 0;
      kern_has_bias    <= 0;
    end
    else begin
      kern_valid       <= bram_valid;
      kern_has_bias    <= bram_has_bias;
    end
    kern_activations   <= bram_activations;
    kern_bias          <= 0;//bias;
    kern_neuron_id     <= neuron_id;
    kern_bp_mode       <= bram_bp_mode;
    kern_bp_mode_o     <= kern_bp_mode;
    weights            <= {data_out_b, data_out_a};
  end


  // 3 modes of use in kernel
  // forward: weight * activations
  // weight gradient: gradient * activations
  // neuron gradient: weight * gradient
  assign kern_mult1 = (forward) ? weights :
              (bram_bp_mode == WEIGHT_MODE) ? b_kern_grad : weights
                  ;

  assign kern_mult2 = (forward) ? kern_activations :
              (bram_bp_mode == WEIGHT_MODE) ? b_kern_act :
                  b_kern_grad;


  // Computational kernel for the fully connected layer
  genvar i;
  generate
    for (i = 0; i < `FC1_N_KERNELS; i=i+1) begin
      fc_kernel #(.FAN_IN(`FC1_FAN_IN), .ID_WIDTH(6)) fc_kernel_i (
        // input
        .clk(clk),
        .rst(rst),
        .activation_i(kern_mult2[i]),
        .weight(kern_mult1[i]),
        .bias(18'b0/*kern_bias[i]*/),
        .neuron_id_i(kern_neuron_id[i]),
        .has_bias(kern_has_bias),
```

```
        .valid_i(kern_valid),
        .b_valid_i(b_valid[2]),
        .bp_mode(bp_mode),
        // output
        .b_gradient_o(b_kern_grad_o[i]),
        .b_valid_o(b_kern_valid_o[i]),
        .activation_o(kern_activation_o[i]),
        .neuron_id_o(neuron_id_o[i]),
        .valid_o(valid[i])
      );
    end
  endgenerate

  bit [6: 0] b;
  always_ff @(posedge clk) begin
    if (rst) begin
      act_o_sign   <= 0;
    end
    else if (&valid) begin
      for (b = 0; b < `FC1_N_KERNELS; b = b + 1) begin
        act_o_sign[neuron_id_o[b]]    <= kern_activation_o[b][`PREC
            - 1];
      end
    end
  end

  bit [10: 0] j;
  always_comb begin
    for (j = 0; j < `FC1_N_KERNELS; j=j+1) begin
      activation_o[j] = kern_activation_o[j][`PREC - 1] ? 0 :
          kern_activation_o[j];
    end
  end

  assign bp_done = wg_we && wg_addr_a == `FC1_MID_PTR_OFFSET - 1;
  assign valid_act_o = &valid;


  bit [5: 0] q;
  // Backward pass logic
  always_ff @(posedge clk) begin
    for (q = 0; q < `FC1_N_KERNELS; q = q + 1) begin
      b_gradient[q]    <= act_o_sign[b_neuron_id_i[q]] ? 0 :
          b_gradient_i[q];
    end
    b_gradient_pl  <= b_gradient;
    b_kern_grad    <= b_gradient_pl;

    b_act          <= b_activation_i;
    b_act_pl       <= b_act;
    b_kern_act     <= b_act_pl;


    b_act_id       <= {b_act_id[2:0], b_activation_id};
    b_neuron_id    <= {b_neuron_id[2:0], b_neuron_id_i};
```

```
    b_valid        <= {b_valid[1: 0], b_valid_i};
  end
endmodule
```

B.1.9 previous_layer_gradient_adder.sv

```
`timescale 1ns / 1ps

module previous_layer_gradient_adder (
  input                                               clk,
  input                                               rst,
  input                                               forward,
  input [`FC1_N_KERNELS - 1: 0][`PREC - 1: 0]         grad_i,
  input [6: 0]                                        neuron_id_i,
  input                                               valid_i,
  input                                               bp_mode_i,

  output logic [`FC0_NEURONS - 1: 0][`PREC - 1: 0]    pl_gradients,
  output logic                                        pl_grad_valid
);

  localparam WEIGHT_MODE = 0;
  localparam NEURON_MODE = 1;

  logic [7: 0][`PREC - 1: 0]  stage1_grad;
  logic                       stage1_valid;
  logic [6: 0]                stage1_neuron_id;
  logic                       stage1_bp_mode;

  logic [3: 0][`PREC - 1: 0]  stage2_grad;
  logic                       stage2_valid;
  logic [6: 0]                stage2_neuron_id;
  logic                       stage2_bp_mode;

  logic [1: 0][`PREC - 1: 0]  stage3_grad;
  logic                       stage3_valid;
  logic [6: 0]                stage3_neuron_id;
  logic                       stage3_bp_mode;

  logic [`PREC - 1: 0]        stage4_grad;
  logic                       stage4_valid;
  logic                       prev_stage4_valid;
  logic [6: 0]                stage4_neuron_id;
  logic                       stage4_bp_mode;

  logic                       prev_bp_mode_i;

  bit [4: 0] i, j, m, n, o, p;
  always_ff @(posedge clk) begin
    prev_bp_mode_i   <= bp_mode_i;

    // Stage 1 of Adder
    if (rst) begin
      stage1_grad        <= 0;
      stage1_valid       <= 0;
      stage1_neuron_id   <= 0;
      stage1_bp_mode     <= 0;
    end
    else begin
```

```
        for (i = 0, n = 0; i < 16; i = i + 2, n = n + 1) begin
          stage1_grad[n]   <= $signed(grad_i[i]) + $signed(grad_i[i +
              1]);
        end
        stage1_valid       <= valid_i & (bp_mode_i == NEURON_MODE);
        stage1_neuron_id   <= neuron_id_i;
        stage1_bp_mode     <= bp_mode_i;
      end

      // Stage 2 of Adder
      if (rst) begin
        stage2_grad        <= 0;
        stage2_valid       <= 0;
        stage2_neuron_id   <= 0;
        stage2_bp_mode     <= 0;
      end
      else begin
        for (j = 0, o = 0; j < 8; j = j + 2, o = o + 1) begin
          stage2_grad[o]   <= $signed(stage1_grad[j]) + $signed(
              stage1_grad[j + 1]);
        end
        stage2_valid       <= stage1_valid;
        stage2_neuron_id   <= stage1_neuron_id;
        stage2_bp_mode     <= stage1_bp_mode;
      end


      // Stage 3 of Adder
      if (rst) begin
        stage3_grad        <= 0;
        stage3_valid       <= 0;
        stage3_neuron_id   <= 0;
        stage3_bp_mode     <= 0;
      end
      else begin
        for (m = 0, p = 0; m < 4; m = m + 2, p = p + 1) begin
          stage3_grad[p]   <= $signed(stage2_grad[m]) + $signed(
              stage2_grad[m + 1]);
        end
        stage3_valid       <= stage2_valid;
        stage3_neuron_id   <= stage2_neuron_id;
        stage3_bp_mode     <= stage2_bp_mode;
      end


      // Stage 4 of Adder
      if (rst) begin
        stage4_grad        <= 0;
        stage4_valid       <= 0;
        prev_stage4_valid  <= 0;
        stage4_neuron_id   <= 0;
        stage4_bp_mode     <= 0;
      end
      else begin
```

```
      stage4_grad        <= $signed(stage3_grad[0]) + $signed(
          stage3_grad[1]);
      stage4_valid       <= stage3_valid;
      prev_stage4_valid  <= stage4_valid;
      stage4_neuron_id   <= stage3_neuron_id;
      stage4_bp_mode     <= stage3_bp_mode;
    end


    // Stage 5
    if (rst || forward) begin
      pl_gradients   <= 0;
    end
    else if (stage4_valid & stage4_bp_mode == NEURON_MODE) begin
      pl_gradients[stage4_neuron_id]   <= $signed(pl_gradients[
          stage4_neuron_id]) + $signed(stage4_grad);
    end

    if (rst) begin
      pl_grad_valid    <= 0;
    end
    else if ({prev_stage4_valid, stage4_valid} == 2'b10) begin
      pl_grad_valid    <= 1'b1;
    end
    else if (forward) begin
      pl_grad_valid    <= 1'b0;
    end
  end
endmodule
```

B.1.10 fc1_weight_bram_controller.sv

```
`timescale 1ns / 1ps

`include "sys_defs.vh"

module fc1_weight_bram_controller (
    input                                                   clk,
    input                                                   rst,

    input   [`FC1_ADDR - 1: 0]                              addr_a,
    input   [`FC1_PORT_WIDTH - 1: 0][`PREC - 1: 0]          data_in_a,
    input                                                   en_a,
    input                                                   we_a,

    input   [`FC1_ADDR - 1: 0]                              addr_b,
    input   [`FC1_PORT_WIDTH - 1: 0][`PREC - 1: 0]          data_in_b,
    input                                                   en_b,
    input                                                   we_b,

    output logic [`FC1_PORT_WIDTH - 1: 0][`PREC - 1: 0] data_out_a,
    output logic [`FC1_PORT_WIDTH - 1: 0][`PREC - 1: 0] data_out_b,
    output logic [`FC1_N_KERNELS - 1: 0][5: 0]           neuron_id

    );

    bit [`FC1_PORT_WIDTH - 1: 0]  i, j;
    always_ff @(posedge clk) begin
        for (i = 0, j = 8; i < `FC1_PORT_WIDTH; i=i+1, j=j+1) begin
            if (addr_a < `FC1_FAN_IN) begin
                neuron_id[i]    <= i;
                neuron_id[j]    <= i + `FC1_HALF_NEURONS;
            end
            else if (addr_a < `FC1_STEP2) begin
                neuron_id[i]    <= i + `FC1_PORT_WIDTH;
                neuron_id[j]    <= i + `FC1_PORT_WIDTH + `
                    FC1_HALF_NEURONS;
            end
            else if (addr_a < `FC1_STEP3) begin
                neuron_id[i]    <= i + `FC1_PORT_WIDTH_TIMES2;
                neuron_id[j]    <= i + `FC1_PORT_WIDTH_TIMES2 + `
                    FC1_HALF_NEURONS;
            end
            else begin
                neuron_id[i]    <= i + `FC1_PORT_WIDTH_TIMES3;
                neuron_id[j]    <= i + `FC1_HALF_NEURONS + `
                    FC1_PORT_WIDTH_TIMES3;
            end
        end
    end

    fc1_weights_bram_0 fc1_weights_bram_0_i (
        .addra(addr_a),
        .clka(clk),
        .dina(data_in_a),
```

```
        .douta(data_out_a),
        .ena(en_a),
        .wea(we_a),

        .addrb(addr_b),
        .clkb(clk),
        .dinb(data_in_b),
        .doutb(data_out_b),
        .enb(en_b),
        .web(we_b)
    );

endmodule
```

B.1.11 fc2_layer.sv

```
`timescale 1ns / 1ps
`include "sys_defs.vh"

module fc2_layer(
  input                                                     clk,
  input                                                     rst,
  input                                                     forward,
  input                                                     update,
  input   [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]             activations_i,
  input                                                     valid_i,
  input   [4: 0]                                            lrate_shifts,


  input  [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]              b_gradient_i,
  input  [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]              b_activation_i,
  input  [5: 0]                                             b_activation_id,
  input  [`FC2_N_KERNELS - 1: 0][3: 0]                      b_neuron_id_i,
  input                                                     b_valid_i,
  input                                                     bp_mode,


  output logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]        activation_o,
  output logic [`FC2_N_KERNELS - 1: 0][3: 0]                neuron_id_o,
  output logic                                              valid_act_o,
  output logic                                              fc2_busy,
  output logic                                              bp_done,
  output logic                                              update_done,

  output logic [`FC1_NEURONS - 1: 0][`PREC - 1: 0]          pl_gradients,
  output logic                                              pl_grad_valid

);

  logic    [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]    data_in;
  logic    [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]    data_out;

  logic    [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]    weights;
  logic    [`FC2_ADDR - 1: 0]                       head_ptr;
  logic    [`FC2_ADDR - 1: 0]                       mid_ptr;
  logic    [`FC2_BIAS_ADDR - 1: 0]                  bias_ptr;

  logic    [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]    sch_activations;
  logic                                             sch_valid;
  logic    [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]    bram_activations;
  logic                                             bram_valid;
  logic    [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]    kern_activations;
  logic                                             kern_valid;

  logic    [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]    bias;
```

```
logic    [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]    kern_bias;
logic                                             sch_has_bias;
logic                                             bram_has_bias;
logic                                             kern_has_bias;
logic    [`FC2_N_KERNELS - 1: 0][3: 0]            neuron_id;
logic    [`FC2_N_KERNELS - 1: 0][3: 0]            kern_neuron_id;
logic    [`FC2_N_KERNELS - 1: 0]                  last_weight;


logic    [`FC2_N_KERNELS - 1: 0]                  valid;

logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       b_gradient;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       b_gradient_pl;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       b_kern_grad;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       b_act;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       b_act_pl;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       b_kern_act;

logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       b_kern_grad_o;
logic [`FC2_N_KERNELS - 1: 0]                     b_kern_valid_o;
logic [2: 0]                                      b_valid;
logic [3: 0][5: 0]                                b_act_id;
logic [3: 0][`FC2_N_KERNELS - 1: 0][3: 0]         b_neuron_id;

logic                                             b_kern_valid;
logic                                             b_weight_we;

logic                                             sch_bp_mode;
logic                                             bram_bp_mode;
logic                                             kern_bp_mode;
logic                                             kern_bp_mode_o;

logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       kern_mult1;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       kern_mult2;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       weight_grad;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       weight_grad_o;
logic [`FC2_N_KERNELS - 1: 0][9: 0]
    fc2_weight_grad_addr;

logic [`FC2_N_KERNELS - 1: 0][`PREC : 0]
    update_weights_sat;
logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]       update_weights;

logic                                             prev_b_kern_valid
    ;

logic [10: 0]                                     update_ptr;
logic [9: 0]                                      update_addr_a;
logic [9: 0]                                      update_addr_b;
logic [9: 0]                                      w_addr_a;
logic [9: 0]                                      w_addr_b;
logic [9: 0]                                      wg_addr_a;
logic [9: 0]                                      wg_addr_b;
logic                                             w_we;
logic                                             wg_we;
```

```
  logic                                                sch_valid_i;

  localparam WEIGHT_MODE = 0;
  localparam NEURON_MODE = 1;

  always_ff @(posedge clk) begin
    if (rst) begin
      prev_b_kern_valid   <= 0;
    end
    else begin
      prev_b_kern_valid   <= &b_kern_valid_o;
    end
  end

  always_ff @(posedge clk) begin
    if (rst) begin
      sch_activations <= 0;
      sch_valid       <= 0;
      sch_bp_mode     <= 0;
    end
    else begin
      sch_activations <= activations_i;
      sch_valid       <= valid_i;
      sch_bp_mode     <= bp_mode;
    end
  end


  assign sch_valid_i = (forward) ? valid_i : b_valid_i & bp_mode ==
      NEURON_MODE;
  // Scheduler for the fully connected layer
  fc_scheduler #(.ADDR(`FC2_ADDR), .BIAS_ADDR(`FC2_BIAS_ADDR),
  .MID_PTR_OFFSET(`FC2_MID_PTR_OFFSET), .FAN_IN(`FC2_FAN_IN))
  fc2_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)
  );




  always_ff @(posedge clk) begin
    if (rst) begin
      bram_activations   <= 0;
      bram_valid         <= 0;
      bram_has_bias      <= 0;
```

```
      fc2_busy            <= 0;
      bram_bp_mode        <= 0;
    end
    else begin
      bram_activations    <= sch_activations;
      bram_valid          <= sch_valid;
      bram_has_bias       <= sch_has_bias;
      fc2_busy            <= valid_i;
      bram_bp_mode        <= sch_bp_mode;
    end
  end



  always_ff @(posedge clk) begin
    if (rst) begin
      update_ptr   <= 0;
    end
    else if (update) begin
      update_ptr   <= update_ptr + 1'b1;
    end
    else begin
      update_ptr   <= 0;
    end
  end


  assign update_done     = update_ptr == 11'd639;
  assign update_addr_a   = update_ptr[10: 1] << 1;
  assign update_addr_b   = update_addr_a + 1'b1;
  assign w_addr_a        = (update) ? update_addr_a   : head_ptr;
  assign w_addr_b        = (update) ? update_addr_b   : mid_ptr;
  assign wg_addr_a       = (update) ? update_addr_a   :
      fc2_weight_grad_addr[0];
  assign wg_addr_b       = (update) ? update_addr_b   :
      fc2_weight_grad_addr[1];
  assign w_we            = (update) ? update_ptr[0]   : 1'b0;  //
      write when odd
  assign wg_we           = (update) ? 1'b0        : b_weight_we;

  always_comb begin
    case(lrate_shifts)
      5'd7: begin
        weight_grad[0] = {{7{weight_grad_o[0][`PREC - 1]}}, {
            weight_grad_o[0][`PREC - 1: 7]}};
        weight_grad[1] = {{7{weight_grad_o[1][`PREC - 1]}}, {
            weight_grad_o[1][`PREC - 1: 7]}};
      end

      5'd9: begin
        weight_grad[0] = {{9{weight_grad_o[0][`PREC - 1]}}, {
            weight_grad_o[0][`PREC - 1: 9]}};
        weight_grad[1] = {{9{weight_grad_o[1][`PREC - 1]}}, {
            weight_grad_o[1][`PREC - 1: 9]}};
      end
```

```
      5'd11: begin
        weight_grad[0] = {{11{weight_grad_o[0][`PREC - 1]}}, {weight_grad_o[0][`PREC - 1: 11]}};
        weight_grad[1] = {{11{weight_grad_o[1][`PREC - 1]}}, {weight_grad_o[1][`PREC - 1: 11]}};
      end
      5'd10: begin
        weight_grad[0] = {{10{weight_grad_o[0][`PREC - 1]}}, {weight_grad_o[0][`PREC - 1: 10]}};
        weight_grad[1] = {{10{weight_grad_o[1][`PREC - 1]}}, {weight_grad_o[1][`PREC - 1: 10]}};
      end
      default: begin
        weight_grad[0] = {{8{weight_grad_o[0][`PREC - 1]}}, {weight_grad_o[0][`PREC - 1: 8]}};
        weight_grad[1] = {{8{weight_grad_o[1][`PREC - 1]}}, {weight_grad_o[1][`PREC - 1: 8]}};
      end
    endcase

    update_weights_sat[0]   = $signed(data_out[0]) - $signed(weight_grad[0]);
    update_weights_sat[1]   = $signed(data_out[1]) - $signed(weight_grad[1]);
  end

  bit [7: 0] d;
  always_comb begin
    for (d = 0; d < `FC2_N_KERNELS; d=d+1) begin
      if (update_weights_sat[d][`PREC :`PREC - 1] == 2'b01) begin
        update_weights[d]   = `MAX_VAL;
      end
      else if (update_weights_sat[d][`PREC :`PREC - 1] == 2'b10) begin
        update_weights[d]   = `MIN_VAL;
      end
      else begin
        update_weights[d]   = update_weights_sat[d][`PREC - 1: 0];
      end
    end
  end

  // BRAM for the weights of the fully connected layer
  fc2_weight_bram_controller fc2_weight_bram_controller_i (
    // inputs
    .clk(clk),
    .rst(rst),

    .addr_a(w_addr_a),
    .data_in_a(update_weights[0]),
    .en_a(1'b1),
    .we_a(w_we),

    .addr_b(w_addr_b),
    .data_in_b(update_weights[1]),
```

```
    .en_b(1'b1),
    .we_b(w_we),

    // outputs
    .data_out(data_out),
    .neuron_id(neuron_id)

  );
  /*
   biases_fc2_blk_mem biases_fc2_blk_mem_i (
    .addra(bias_ptr),
    .clka(clk),
    .dina(32'b0),
    .douta({bias[1], bias[0]}),
    .ena(1'b1),
    .wea(1'b0)
  );
  */
  assign b_weight_we = &b_kern_valid_o & kern_bp_mode_o ==
      WEIGHT_MODE;
  assign fc2_weight_grad_addr[0] = ({6'b0, b_neuron_id[3][0]} << 6)
       + b_act_id[3];
  assign fc2_weight_grad_addr[1] = ({6'b0, b_neuron_id[3][1]} << 6)
       + b_act_id[3];

  fc2_weight_gradients fc2_weight_gradients_i (
    .addra(wg_addr_a),
    .clka(clk),
    .dina(b_kern_grad_o[0]),
    .douta(weight_grad_o[0]),
    .ena(1'b1),
    .wea(wg_we),

    .addrb(wg_addr_b),
    .clkb(clk),
    .dinb(b_kern_grad_o[1]),
    .doutb(weight_grad_o[1]),
    .enb(1'b1),
    .web(wg_we)
  );


  bit [2: 0] z;
  always_ff @(posedge clk) begin


    // Calculating gradients for the neurons of the previous layer
    if (rst || forward) begin
      pl_gradients  <= 0;
    end
    else if (&b_kern_valid_o & kern_bp_mode_o == NEURON_MODE) begin
      pl_gradients[b_act_id[3]]  <= $signed(pl_gradients[b_act_id
          [3]]) +
                        $signed(b_kern_grad_o[0]) +
                        $signed(b_kern_grad_o[1]);
```

```
    end

    if (rst) begin
      pl_grad_valid   <= 0;
    end
    else if (&b_kern_valid_o & {kern_bp_mode_o, kern_bp_mode} == 2'b10) begin
      pl_grad_valid   <= 1'b1;
    end
    else if (forward) begin
      pl_grad_valid   <= 1'b0;
    end
  end



  always_ff @(posedge clk) begin
    if (rst) begin
      kern_valid      <= 0;
      kern_has_bias   <= 0;
    end
    else begin
      kern_valid      <= bram_valid;
      kern_has_bias   <= bram_has_bias;
    end
    kern_activations  <= bram_activations;
    kern_bias         <= 0;//bias;
    kern_neuron_id    <= neuron_id;
    kern_bp_mode      <= bram_bp_mode;
    kern_bp_mode_o    <= kern_bp_mode;
    weights           <= data_out;
  end



  // 3 modes of use in kernel
  // forward: weight * activations
  // weight gradient: gradient * activations
  // neuron gradient: weight * gradient
  assign kern_mult1 = (forward) ? weights :
              (bram_bp_mode == WEIGHT_MODE) ? b_kern_grad : weights;

  assign kern_mult2 = (forward) ? kern_activations :
              (bram_bp_mode == WEIGHT_MODE) ? b_kern_act : b_kern_grad;

  // Computational kernel for the fully connected layer
  genvar i;
  generate
    for (i = 0; i < `FC2_N_KERNELS; i=i+1) begin
      fc_kernel #(.FAN_IN(`FC2_FAN_IN), .ID_WIDTH(4)) fc_kernel_i (
        // input
        .clk(clk),
        .rst(rst),
```

```
          .activation_i(kern_mult2[i]),
          .weight(kern_mult1[i]),
          .bias(kern_bias[i]),
          .neuron_id_i(kern_neuron_id[i]),
          .has_bias(kern_has_bias),
          .valid_i(kern_valid),
          .b_valid_i(b_valid[2]),
          .bp_mode(bp_mode),
          // output
          .b_gradient_o(b_kern_grad_o[i]),
          .b_valid_o(b_kern_valid_o[i]),
          .activation_o(activation_o[i]),
          .neuron_id_o(neuron_id_o[i]),
          .valid_o(valid[i])
        );
      end
    endgenerate
    assign bp_done = wg_we && wg_addr_a == `FC2_MID_PTR_OFFSET - 1;
    assign valid_act_o = &valid;


     // Backward pass logic
    always_ff @(posedge clk) begin
      b_gradient     <= b_gradient_i;
      b_gradient_pl  <= b_gradient;
      b_kern_grad    <= b_gradient_pl;

      b_act          <= b_activation_i;
      b_act_pl       <= b_act;
      b_kern_act     <= b_act_pl;


      b_act_id       <= {b_act_id[2:0], b_activation_id};
      b_neuron_id    <= {b_neuron_id[2:0], b_neuron_id_i};
      b_valid        <= {b_valid[1: 0], b_valid_i};
    end
  endmodule
```

B.1.12 fc2_weight_bram_controller.sv

```
`timescale 1ns / 1ps

`include "sys_defs.vh"

module fc2_weight_bram_controller (
    input                                                       clk,
    input                                                       rst,

    input   [`FC2_ADDR - 1: 0]                                  addr_a,
    input   [`PREC - 1: 0]                                      data_in_a,
    input                                                       en_a,
    input                                                       we_a,

    input   [`FC2_ADDR - 1: 0]                                  addr_b,
    input   [`PREC - 1: 0]                                      data_in_b,
    input                                                       en_b,
    input                                                       we_b,

    output logic [`FC2_N_KERNELS - 1: 0][`PREC - 1: 0]         data_out,
    output logic [`FC2_N_KERNELS - 1: 0][3: 0]                 neuron_id

    );

    logic [`FC2_BRAM - 1: 0][`PREC - 1: 0]  data_out_a;
    logic [`FC2_BRAM - 1: 0][`PREC - 1: 0]  data_out_b;

    assign data_out = {data_out_b, data_out_a};

    bit [`FC2_BRAM - 1: 0]  i, j;
    always_ff @(posedge clk) begin
        if (rst) begin
            neuron_id        <= 0;
        end
        else begin
            for (i = 0, j = 1; i < `FC2_BRAM; i=i+1, j=j+1) begin
                neuron_id[i]     <= addr_a[9: 6];     // abuse the
                    fact that fan in is a power of 2
                neuron_id[j]     <= addr_a[9: 6] + `FC2_HALF_NEURONS
                    ;
            end
        end
    end

    fc2_weights_bram fc2_weights_bram_i (
        .addra(addr_a),
        .clka(clk),
        .dina(data_in_a),
        .douta(data_out_a[0]),
        .ena(en_a),
        .wea(we_a),

        .addrb(addr_b),
        .clkb(clk),
```

```
        .dinb(data_in_b),
        .doutb(data_out_b[0]),
        .enb(en_b),
        .web(we_b)
    );

endmodule
```

B.1.13 softmax.sv

```
`timescale 1ns / 1ps

module softmax(
  input                                                   clk,
  input                                                   reset,
  input                                                   start,
  input [`PREC - 1: 0]                                    max,
  input [`FC2_NEURONS - 1: 0][`PREC - 1: 0]               act_in,

  output logic                                            valid_o,
  output logic [`FC2_NEURONS - 1: 0][`PREC - 1: 0]        grad_o
  );


  logic [`FC2_NEURONS - 1: 0][23: 0]   act_in_norm;
  logic [`FC2_NEURONS - 1: 0][23: 0]   fixed_exp_res;
  logic [`FC2_NEURONS - 1: 0][31: 0]   act_in_norm_float;
  logic [31: 0]                        float_o;
  logic [31: 0]                        float_exp_o;
  logic                                float_valid_o;
  logic                                float_exp_valid_o;
  logic [23: 0]                        fixed_exp_o;
  logic [23: 0]                        fixed_exp_sum;
  logic                                fixed_exp_valid_o;
  logic [3: 0]                         fp_in_ptr;
  logic [3: 0]                         fixed_exp_ptr;
  logic [3: 0]                         div_ptr;
  logic                                in_prog;
  logic [47: 0]                        div_o;
  logic                                div_valid_i;
  logic                                div_valid_o;

  bit [3: 0] i;
  always_ff @(posedge clk) begin
    if (start) begin
      for (i = 0; i < `FC2_NEURONS; i=i+1) begin
        act_in_norm[i]  <= $signed(act_in[i]) - $signed(max);
      end
    end
  end

  always_ff @(posedge clk) begin
    if (~in_prog) begin
      fp_in_ptr <= 4'b0;
    end
    if (in_prog && fp_in_ptr != `FC2_NEURONS) begin
      fp_in_ptr <= fp_in_ptr + 1'b1;
    end
  end

  fp_to_float fp_to_float_i (
    .s_axis_a_tdata(act_in_norm[fp_in_ptr]),
    .s_axis_a_tvalid(in_prog & fp_in_ptr != `FC2_NEURONS),
```

```
    .aclk(clk),

    .m_axis_result_tdata(float_o),
    .m_axis_result_tvalid(float_valid_o)

  );

  float_exp float_exp_i (
    .s_axis_a_tdata(float_o),
    .s_axis_a_tvalid(float_valid_o),
    .aclk(clk),

    .m_axis_result_tdata(float_exp_o),
    .m_axis_result_tvalid(float_exp_valid_o)
  );

  float_to_fp float_to_fp_i (
    .s_axis_a_tdata(float_exp_o),
    .s_axis_a_tvalid(float_exp_valid_o),
    .aclk(clk),

    .m_axis_result_tdata(fixed_exp_o),
    .m_axis_result_tvalid(fixed_exp_valid_o)
  );

  always_ff @(posedge clk) begin
    if (~in_prog) begin
      fixed_exp_sum    <= 0;
    end
    if (fixed_exp_valid_o) begin
      fixed_exp_sum    <= $signed(fixed_exp_sum) + $signed(
          fixed_exp_o);
    end
  end

  always_ff @(posedge clk) begin
    if (~in_prog) begin
      fixed_exp_ptr           <= 4'b0;
      fixed_exp_res[fixed_exp_ptr]  <= 0;
    end
    else if (fixed_exp_valid_o) begin
      fixed_exp_ptr           <= fixed_exp_ptr + 1'b1;
      fixed_exp_res[fixed_exp_ptr]  <= fixed_exp_o;
    end
  end


  always_ff @(posedge clk) begin
    if (~in_prog) begin
      div_ptr <= 0;
    end
    else if (div_ptr != `FC2_NEURONS && fixed_exp_ptr == `
        FC2_NEURONS) begin
      div_ptr <= div_ptr + 1'b1;
    end
```

```
  end

  assign div_valid_i = (fixed_exp_ptr == `FC2_NEURONS) & (div_ptr
      != `FC2_NEURONS);

  fixed_divider fixed_divider_i (
    .s_axis_divisor_tdata(fixed_exp_sum),
    .s_axis_divisor_tvalid(div_valid_i),

    .s_axis_dividend_tdata(fixed_exp_res[div_ptr]),
    .s_axis_dividend_tvalid(div_valid_i),

    .aclk(clk),

    .m_axis_dout_tdata(div_o),
    .m_axis_dout_tvalid(div_valid_o)

  );

  logic [3: 0] grad_ptr;
  always_ff @(posedge clk) begin
    if (~in_prog) begin
      grad_ptr  <= 0;
    end
    else if (div_valid_o) begin
      grad_ptr  <= grad_ptr + 1'b1;
    end
  end

  always_ff @(posedge clk) begin
    if (div_valid_o) begin
      grad_o[grad_ptr]  <= div_o[`PREC - 1: 0];
    end

    valid_o <= ((grad_ptr == `FC2_NEURONS) & in_prog);

  end

  always_ff @(posedge clk) begin
    if (reset) begin
      in_prog <= 1'b00;
    end
    else if (start) begin
      in_prog <= 1'b1;
    end
    else if (valid_o) begin
      in_prog <= 1'b0;
    end
  end
endmodule
```

B.2 Testbenches

B.2.1 neural_net_top_tb.sv

```
`timescale 1ns / 1ps

module neural_net_top_tb(
    );

    logic clock;
    logic reset;

    neural_net_top neural_net_top_i (
        .clock_in(clock),
        .rst(reset),
        .sw_in(8'h01),
        .led_o()
    );


    always begin
        #5
        clock = ~clock;
    end

    initial begin
        clock = 1'b0;
        reset = 1'b1;
        @(negedge clock);
        reset = 1'b1;
        @(negedge clock);
        @(negedge clock);
        @(negedge clock);
        @(negedge clock);
        @(negedge clock);
        reset = 1'b0;

        #100000;
        $finish;
    end
endmodule
```

B.2.2 softmax_tb.sv

```
`timescale 1ns / 1ps

module softmax_tb(

    );

    logic                               clock;
    logic                               reset;
    logic                               start;
    logic [15: 0]                       max;
    logic [`FC2_NEURONS - 1: 0][15: 0]  act_in;

    logic valid_o;
    logic [`FC2_NEURONS - 1: 0][15: 0]  grad_o;


    assign act_in = {
        16'h1234,
        16'h0735,
        16'hdf28,
        16'hf801,
        16'hf206,
        16'h1842,
        16'h1842,
        16'h2182,
        16'h0321,
        16'h0a18
    };

    assign max = 16'h2182;

    softmax softmax_i (
        .clk(clock),
        .reset(reset),
        .start(start),
        .max(max),
        .act_in(act_in),

        .valid_o(),
        .grad_o()

    );

    always begin
        #5
        clock = ~clock;
    end

    initial begin
        clock = 1'b0;
        reset = 1'b1;
        @(negedge clock);
        reset = 1'b1;
```

```
        @(negedge clock);
        reset = 1'b0;
        @(negedge clock);
        @(negedge clock);
        start = 1'b1;
        @(negedge clock);
        start = 1'b0;


        #100000;
        $finish;
    end
endmodule
```

B.2.3 fc1_scheduler_tb.sv

```
`timescale 1ns / 1ps
`include "sys_defs.vh"


module fc1_scheduler_tb(

    );

    logic clock;
    logic reset;
    logic forward;
    logic has_bias;

    logic   [`FC1_N_KERNELS - 1: 0]         valid_i;

    logic   [`FC1_ADDR - 1: 0]              head_ptr;
    logic   [`FC1_ADDR - 1: 0]              mid_ptr;
    logic   [`FC1_BIAS_ADDR - 1: 0]         bias_ptr;




    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 (
        .clk(clock),
        .rst(reset),
        .forward(forward),
        .valid_i(&valid_i),

        .head_ptr(head_ptr),
        .mid_ptr(mid_ptr),
        .bias_ptr(bias_ptr),
        .has_bias(has_bias)
    );

    always begin
        #5 clock = ~clock;
    end


    initial begin
       clock = 1'b0;
       reset = 1'b1;
       forward = 1'b1;
       valid_i = 0;
       @(negedge clock);
       reset = 1'b1;
       @(negedge clock);
       reset = 1'b0;
       valid_i = {`FC1_N_KERNELS{1'b1}};

    end
```

```
endmodule
```

B.2.4 fc1_layer_tb.sv

```
`timescale 1ns / 1ps
`include "sys_defs.vh"
module fc1_layer_tb(

    );

    logic clock;
    logic reset;
    logic forward;

    logic  [`FC1_N_KERNELS - 1: 0][15: 0]        activations_i;
    logic  [`FC1_N_KERNELS - 1: 0]               valid_i;
    logic  [`FC1_N_KERNELS - 1: 0][15: 0]        activation_o;
    logic  [`FC1_N_KERNELS - 1: 0][4: 0]         neuron_id_o;
    logic  [`FC1_N_KERNELS - 1: 0]               valid_act_o;




    fc1_layer fc1_layer_i (
        // inputs
        .clk(clock),
        .rst(reset),
        .forward(forward),
        .activations_i(activations_i),
        .valid_i(valid_i),

        // outputs
        .activation_o(activation_o),
        .neuron_id_o(neuron_id_o),
        .valid_act_o(valid_act_o)
    );

    always begin
        #5
        clock = ~clock;
    end

    initial begin
        clock = 1'b0;
        reset = 1'b1;
        forward = 1'b1;
        valid_i = 0;
        @(negedge clock);
        reset = 1'b1;
        @(negedge clock);
        reset = 1'b0;
        valid_i = {`FC1_N_KERNELS{1'b1}};
        activations_i = {`FC1_N_KERNELS{16'h2000}};

        #100000;
        $finish;
    end
```

```
endmodule
```

Appendix C

Processing System Code

This appendix contains the source code for the programs that run on the PS. It should be noted that *inference_only.c* and *train.c* are quite similar. This code was written in C and cross-compiled for ARM.

C.1 train.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdint.h>
#include "parse_mnist.h"
#include <unistd.h>
#include <math.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>

#define FORWARD     1
#define WAITING     2
#define BACKWARD    3
#define UPDATE      4
#define IDLE        5
#define SET_SIZE    70000
```

```c
#define TRAIN_SIZE  4000
#define START_LRATE 9

typedef struct ddr_data {
    // written to by fpga          Offset  Desc
    uint32_t    fpga_img_id;       // 0    fpga image ptr
    uint32_t    epoch;             // 1
    uint32_t    num_correct_train; // 2
    uint32_t    num_correct_test;  // 3
    uint32_t    idle_cycles;       // 4
    uint32_t    active_cycles;     // 5
    uint32_t    status;            // 6    contains status info

    // written to by arm
    uint32_t    start;             // 7    start looping
    uint32_t    n_epochs;          // 8    upper limit on epochs
    uint32_t    learning_rate;     // 9    # of right shifts
    uint32_t    training_mode;     // 10   train or just forward pass
    uint32_t    img_set_size;      // 11   size of dataset
    uint32_t    img_id;            // 12   arm image ptr
    uint32_t    img_label;         // 13
    uint32_t    img[196];          // 14
    int16_t     out[10];

} ddr_data_t;

void state_enc_to_str(uint32_t state, char* enc);
void parse_mnist_data(char* filename, uint32_t** mnist_images);
void print_debug_data(volatile ddr_data_t* ddr_ptr);

int main() {
  uint32_t magic_number;
  uint32_t id, test_idx, epoch, corr_tr, corr_test;
  uint32_t** train_images;
  uint32_t** test_images;
  uint32_t* train_labels;
  uint32_t* test_labels;

  int handle = open("/dev/mem", O_RDWR | O_SYNC);
  ddr_data_t* ddr_ptr = mmap(NULL, 134217728, PROT_READ |
      PROT_WRITE, MAP_SHARED, handle, 0x40000000);


  uint32_t* ptr = (uint32_t*)ddr_ptr;
  magic_number = ptr[400];
  printf("@@@ Checking Magic Number\n");
  if (magic_number != 0xFADEDBEE) {
    printf("@@@ Memory was read incorrectly.\n");
    return -1;
  }
  printf("@@@ Magic number: %08x\n", magic_number);
  printf("@@@ Magic number successfully read.\n");

  // Load MNIST images into memory
```

```
  printf("@@@ Loading MNIST images...\n");
  train_images = parse_mnist_images("data/train-images.idx3-ubyte");
  train_labels = parse_mnist_labels("data/train-labels.idx1-ubyte");
  test_images = parse_mnist_images("data/t10k-images.idx3-ubyte");
  test_labels = parse_mnist_labels("data/t10k-labels.idx1-ubyte");
  printf("@@@ Loading complete!\n");

  struct timespec sleep;
  sleep.tv_sec = 0;
  sleep.tv_nsec = 1000;

  // Start training!
  ddr_ptr->start = 0;
  usleep(100);
  ddr_ptr->start = 1;
  ddr_ptr->n_epochs = 5;
  ddr_ptr->learning_rate = START_LRATE;
  ddr_ptr->training_mode = 1;
  ddr_ptr->img_set_size = SET_SIZE - 1;
  struct timeval start, end;
  gettimeofday(&start, NULL);
  do {
    id     = (ddr_ptr->fpga_img_id + 1) % SET_SIZE;
    epoch    = ddr_ptr->epoch;
    // Print data if epoch just finished
    if ((id == 0) && epoch != 0) {
      gettimeofday(&end, NULL);
      corr_tr      = ddr_ptr->num_correct_train;
      corr_test    = ddr_ptr->num_correct_test;
      printf("\n\n@@@ EPOCH %d\n@@@ Training Images"
          ": %d/%d\nAccuracy: %f%%\n"
          "@@@Test Images: %d/%d\n"
          "Accuracy: %f%%\n", epoch, corr_tr, TRAIN_SIZE,
          (float)(corr_tr/(float)TRAIN_SIZE) * 100., corr_test,
              SET_SIZE - TRAIN_SIZE,
          ((float)corr_test/(float)(SET_SIZE - TRAIN_SIZE)) * 100.)
              ;


      uint32_t active = ddr_ptr->active_cycles;
      uint32_t idle = ddr_ptr->idle_cycles;
      printf("Active Cycles: %d\t Idle Cycles: %d\n", active, idle)
          ;
      printf("Active Cycle Percentage: %f%%\n", 100.*(float)active
          / ((float)idle + (float)active));
      printf("Elapsed time: %.5f seconds\n", (end.tv_sec - start.
          tv_sec) + ((end.tv_usec - start.tv_usec) * 1e-6));
      gettimeofday(&start, NULL);
    }

    ddr_ptr->training_mode = (id < TRAIN_SIZE);


```

```
    if (id < 60000) {
      memcpy((void*)ddr_ptr->img, train_images[id], sizeof(uint32_t
          ) * 196);
      ddr_ptr->img_label  = train_labels[id];
    }
    else {
      test_idx = id - 60000;
      memcpy((void*)ddr_ptr->img, test_images[test_idx],  sizeof(
          uint32_t) * 196);
      ddr_ptr->img_label  = test_labels[test_idx];
    }
    nanosleep(&sleep, NULL);
    ddr_ptr->img_id    = id;

  } while (epoch < ddr_ptr->n_epochs);
}

void state_enc_to_str (uint32_t state, char* enc) {

  if (state == IDLE) {
    sprintf(enc, "IDLE");
  }
  else if (state == FORWARD) {
    sprintf(enc, "FORWARD");
  }
  else if (state == WAITING) {
    sprintf(enc, "WAITING");
  }
  else if (state == BACKWARD) {
    sprintf(enc, "BACKWARD");
  }
  else if (state == UPDATE) {
    sprintf(enc, "UPDATE");
  }
}

void print_debug_data(volatile ddr_data_t* ddr_ptr) {

  uint32_t start, fpga_img_id, img_id, img_label;
  uint32_t status;
  uint32_t led_o_r, fc0_state, fc1_state, fc2_state, forward,
      fc0_start, fc1_start;
  uint32_t fc2_start, fc0_busy, fc1_busy, fc2_busy, new_img,
      all_idle, img_valid;
  char fc0_state_str[40];
  char fc1_state_str[40];
  char fc2_state_str[40];
  uint32_t corr_tr, corr_test;
  float output[10];

  printf("\n@@@ CURRENT STATE \n");
  fpga_img_id = ddr_ptr->fpga_img_id;
  img_id      = ddr_ptr->img_id;
  img_label   = ddr_ptr->img_label;
  start       = ddr_ptr->start;
```

```
  // parse the status data
  status      = ddr_ptr->status;
  img_valid   = status & 0x1;
  all_idle    = (status >> 1) & 0x1;
  new_img     = (status >> 2) & 0x1;
  fc2_busy    = (status >> 3) & 0x1;
  fc1_busy    = (status >> 4) & 0x1;
  fc0_busy    = (status >> 5) & 0x1;
  fc2_start   = (status >> 6) & 0x1;
  fc1_start   = (status >> 7) & 0x1;
  fc0_start   = (status >> 8) & 0x1;
  forward     = (status >> 9) & 0x1;
  fc2_state   = (status >> 10) & 0x7;
  fc1_state   = (status >> 13) & 0x7;
  fc0_state   = (status >> 16) & 0x7;
  led_o_r     = (status >> 19) & 0xFF;
  corr_tr     = ddr_ptr->num_correct_train;
  corr_test   = ddr_ptr->num_correct_test;
  state_enc_to_str(fc0_state, fc0_state_str);
  state_enc_to_str(fc1_state, fc1_state_str);
  state_enc_to_str(fc2_state, fc2_state_str);

  float max_out    = -100;
  int max_out_id   = 0;
  for (int i = 0; i < 10; i++) {
    output[i] = (float)(ddr_ptr->out[i]) / pow(2, 10);
    if (output[i] > max_out) {
      max_out = output[i];
      max_out_id = i;
    }
  }

  printf("fpga_img_id: %d\t\timg1_id: %d\n", fpga_img_id, img_id);
  printf("img1_label: %d\t\tmax_out: %d\t\tled_o: %08x\n",
      img_label, max_out_id, led_o_r);
  printf("Output:\n");
  for (int i = 0; i < 10; i++) {
    printf("%d: %f\n", i, output[i]);
  }


}
```

C.2 inference_only.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdint.h>
#include "parse_mnist.h"
#include <unistd.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>

#define FORWARD   1
#define WAITING   2
#define BACKWARD  3
#define UPDATE    4
#define IDLE    5
#define SET_SIZE  70000
#define TRAIN_SIZE  70000

typedef struct ddr_data {
  // written to by fpga          Offset    Desc
  uint32_t  fpga_img_id;         // 0      fpga image ptr
  uint32_t  epoch;               // 1
  uint32_t  num_correct_train;   // 2
  uint32_t  num_correct_test;    // 3
  uint32_t  idle_cycles;         // 4
  uint32_t  active_cycles;       // 5
  uint32_t  status;              // 6      contains status info

  // written to by arm
  uint32_t  start;               // 7      start looping
  uint32_t  n_epochs;            // 8      upper limit on epochs
  uint32_t  learning_rate;       // 9      # of right shifts
  uint32_t  training_mode;       // 10     train or just forward pass
  uint32_t  img_set_size;        // 11     size of dataset
  uint32_t  img_id;              // 12     arm image ptr
  uint32_t  img_label;           // 13
  uint32_t  img[196];            // 14
  int16_t   out[10];

} ddr_data_t;

void state_enc_to_str(uint32_t state, char* enc);
void parse_mnist_data(char* filename, uint32_t** mnist_images);
void print_debug_data(volatile ddr_data_t* ddr_ptr);

int main() {
  uint32_t magic_number;
  uint32_t id, test_idx, epoch, corr_tr, corr_test;
  uint32_t** train_images;
  uint32_t** test_images;
```

```
  uint32_t* train_labels;
  uint32_t* test_labels;

  int handle = open("/dev/mem", O_RDWR | O_SYNC);
  volatile ddr_data_t* ddr_ptr = mmap(NULL, 134217728, PROT_READ |
      PROT_WRITE, MAP_SHARED, handle, 0x40000000);


  uint32_t* ptr = (uint32_t*)ddr_ptr;
  magic_number = ptr[400];
  printf("@@@ Checking Magic Number\n");
  if (magic_number != 0xFADEDBEE) {
    printf("@@@ Memory was read incorrectly.\n");
    return -1;
  }
  printf("@@@ Magic number: %08x\n", magic_number);
  printf("@@@ Magic number successfully read.\n");

  // Load MNIST images into memory
  printf("@@@ Loading MNIST images...\n");
  train_images = parse_mnist_images("data/train-images.idx3-ubyte")
      ;
  train_labels = parse_mnist_labels("data/train-labels.idx1-ubyte")
      ;
  test_images = parse_mnist_images("data/t10k-images.idx3-ubyte");
  test_labels = parse_mnist_labels("data/t10k-labels.idx1-ubyte");
  printf("@@@ Loading complete!\n");

  struct timespec sleep;
  sleep.tv_sec = 0;
  sleep.tv_nsec = 1000;

  // Start training!
  ddr_ptr->start = 0;
  usleep(10);
  ddr_ptr->start = 1;
  ddr_ptr->n_epochs = 2;
  ddr_ptr->training_mode = 0;
  ddr_ptr->img_set_size = SET_SIZE - 1;
  struct timeval start, end;
  gettimeofday(&start, NULL);
  do {
    id    = (ddr_ptr->fpga_img_id + 1) % SET_SIZE;
    epoch   = ddr_ptr->epoch;
    // Print data if epoch just finished
    if ((id == 0) && epoch != 0) {
      gettimeofday(&end, NULL);

      corr_tr     = ddr_ptr->num_correct_train;
      corr_test   = ddr_ptr->num_correct_test;
      printf("\nImages"
          ": %d/%d\nAccuracy: %f%%\n", corr_test, 70000,
          ((float)corr_test/70000.) * 100.);


```

```
      uint32_t active = ddr_ptr->active_cycles;
      uint32_t idle = ddr_ptr->idle_cycles;
      printf("Active Cycles: %d\t Idle Cycles: %d\n", active, idle);
      printf("Active Cycle Percentage: %f%%\n", (float)active / ((float)idle + (float)active));
      printf("Elapsed time: %.5f seconds\n\n", (end.tv_sec - start.tv_sec) + ((end.tv_usec - start.tv_usec) * 1e-6));
      gettimeofday(&start, NULL);
    }

    if (id < 60000) {
      memcpy((void*)ddr_ptr->img, train_images[id], sizeof(uint32_t) * 196);
      ddr_ptr->img_label  = train_labels[id];
    }
    else {
      test_idx = id - 60000;
      memcpy((void*)ddr_ptr->img, test_images[test_idx],  sizeof(uint32_t) * 196);
      ddr_ptr->img_label  = test_labels[test_idx];
    }

    nanosleep(&sleep, NULL);
    ddr_ptr->img_id   = id;

  } while (epoch < ddr_ptr->n_epochs);
}
```

C.3 parse_mnist.c

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint32_t** parse_mnist_images(char* filename) {
  FILE* f;
  uint8_t* u8;
  uint8_t** images;
  uint32_t** images32;
  uint32_t magic_number;
  uint32_t n_items;
  uint32_t rows;
  uint32_t cols;

  f = fopen(filename, "rb");

  fread(&magic_number, 4, 1, f);
  fread(&n_items, 1, 4, f);
  fread(&rows, 1, 4, f);
  fread(&cols, 1, 4, f);

  u8 = (uint8_t*)&magic_number;
  magic_number = u8[3] + (u8[2] << 8) + (u8[1] << 16) + (u8[0] <<
      24);
  u8 = (uint8_t*)&n_items;
  n_items = u8[3] + (u8[2] << 8) + (u8[0] << 16) + (u8[0] << 24);
  u8 = (uint8_t*)&rows;
  rows = u8[3] + (u8[2] << 8) + (u8[0] << 16) + (u8[0] << 24);
  u8 = (uint8_t*)&cols;
  cols = u8[3] + (u8[2] << 8) + (u8[0] << 16) + (u8[0] << 24);

  images = malloc(sizeof(uint8_t*) * n_items);

  for (int i = 0; i < n_items; i++) {
    images[i] = malloc(sizeof(uint8_t) * 784);
    fread(images[i], 784, 1, f);
  }

  images32 = malloc(sizeof(uint32_t*) * n_items);
  for (int i = 0; i < n_items; i++) {
    images32[i] = malloc(sizeof(uint32_t) * 196);
    for (int j = 0; j < 196; j++) {
      images32[i][j] = images[i][4*j] + (images[i][4*j+1] << 8) +
            (images[i][4*j+2] << 16) + (images[i][4*j+3] << 24);
    }
  }

  fclose(f);
  return images32;
}

uint32_t* parse_mnist_labels(char* filename) {
```

```
  uint8_t* ptr;
  uint8_t* labels;
  uint32_t* labels32;
  uint32_t magic_number;
  uint32_t n_items;
  uint8_t label;
  FILE* f;
  f = fopen(filename, "rb");

  fread(&magic_number, 4, 1, f);
  ptr = (uint8_t*)&magic_number;
  magic_number = ptr[3] + (ptr[2] << 8) + (ptr[1] << 16) + (ptr[0]
      << 24);
  fread(&n_items, 1, 4, f);
  ptr = (uint8_t*)&n_items;
  n_items = ptr[3] + (ptr[2] << 8) + (ptr[0] << 16) + (ptr[0] <<
      24);
  labels = malloc(sizeof(uint8_t) * n_items);
  labels32 = malloc(sizeof(uint32_t) * n_items);
  for (int i = 0; i < n_items; i++) {
    fread(&labels[i], 1, 1, f);
  }

  for (int i = 0; i < n_items; i++) {
    labels32[i] = labels[i];
  }

  fclose(f);
  return labels32;
}
```

Appendix D

Hardware Testing Code

This Appendix contains the code that was use to verify the simulated output of the hardware model. This code was written in Python.

D.1 fpga_forward_backward_pass_test.py

```
import glob
import math
import random
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import seaborn as sns
from itertools import chain

integer_bits = 6
int_bits_grad = 1

# Set activations in
fc0_fan_in = 28*28
activations_i = []
f = open('../FPGA/FPGA.srcs/sources_1/ip/rand_input_6.12.coe')
next(f)
next(f)
for line in f:
```

```
  act = 0
  bit_val = 2 ** (integer_bits - 1)

  if (line[0] == '1'):
    act -= bit_val
  bit_val /= 2.

  for bit in line[1:]:
    if (bit == '1'):
      act += bit_val
    bit_val /= 2.
  activations_i.append(act)

# FC0 layer
fc0_n_neurons = 98
fc0_fan_in = 28*28
fc0_bram = 1
fc0_neurons_per_bram = fc0_n_neurons / fc0_bram
fc0_neurons = [[] for i in range(fc0_n_neurons)]
n_offset = 0


# read fc0
path = '../FPGA/FPGA.srcs/sources_1/ip/fc0_weights_1.17.coe'
for fname in glob.glob(path):
  print(fname)
  f = open(fname, 'r')
  next(f)
  next(f)
  for line in f:
    for i in range(fc0_n_neurons):
      curr_neuron = i
      st_bit = i * 18
      end_bit = (i + 1) * 18

      bit_str = line[st_bit: end_bit]
      weight_val = 0
      bit_val = 2 ** (int_bits_grad - 1)

      if (bit_str[0] == '1'):
        weight_val -= bit_val
      bit_val /= 2.

      for bit in bit_str[1:]:
        if (bit == '1'):
          weight_val += bit_val
        bit_val /= 2.
      fc0_neurons[fc0_n_neurons - (curr_neuron + 1)].append(
          weight_val)


# FC1 layer
fc1_n_neurons = 64
fc1_fan_in = 98
fc1_bram = 8
```

```
fc1_neurons_per_bram = fc1_n_neurons / fc1_bram
fc1_neurons = [[] for i in range(fc1_n_neurons)]
n_offset = 0


# read fc1
path = '../FPGA/FPGA.srcs/sources_1/ip/fc1_weights2_1.17.coe'
for fname in glob.glob(path):
  print(fname)
  f = open(fname, 'r')
  next(f)
  next(f)
  for line in f:
    for i in range(8):
      curr_neuron = n_offset + (7 - i)
      st_bit = i * 18
      end_bit = (i + 1) * 18

      bit_str = line[st_bit: end_bit]
      weight_val = 0
      bit_val = 2 ** (int_bits_grad - 1)

      if (bit_str[0] == '1'):
        weight_val -= bit_val
      bit_val /= 2.

      for bit in bit_str[1:]:
        if (bit == '1'):
          weight_val += bit_val
        bit_val /= 2.
      fc1_neurons[curr_neuron].append(weight_val)
    if len(fc1_neurons[n_offset]) == fc1_fan_in:
      n_offset += 8

# FC2 layer
fc2_n_neurons = 10
fc2_fan_in = fc1_n_neurons
fc2_bram = 1
fc2_neurons_per_bram = fc2_n_neurons / fc2_bram
fc2_neurons = [[] for i in range(fc2_n_neurons)]
n_offset = 0


# read fc2
path = '../FPGA/FPGA.srcs/sources_1/ip/fc2_weights_1.17.coe'
for fname in glob.glob(path):
  print(fname)
  f = open(fname, 'r')
  next(f)
  next(f)
  curr_neuron = n_offset
  for line in f:
    weight_val = 0
    bit_val = 2 ** (int_bits_grad - 1)

```

```
    if (line[0] == '1'):
      weight_val -= bit_val
    bit_val /= 2.

    for bit in line[1:]:
      if (bit == '1'):
        weight_val += bit_val
      bit_val /= 2.
    fc2_neurons[curr_neuron].append(weight_val)
    if len(fc2_neurons[curr_neuron]) == fc2_fan_in:
      curr_neuron += fc2_bram
  n_offset += 1

fc0_output = []
for neuron in fc0_neurons:
  n_out = 0
  for j in range(len(activations_i[0: fc0_fan_in])):
    n_out += activations_i[j + 0*fc0_fan_in] * neuron[j]
  fc0_output.append(max(n_out, 0))


fc1_output = []
for neuron in fc1_neurons:
  n_out = 0
  for j in range(len(fc0_output)):
    n_out += fc0_output[j] * neuron[j]
  fc1_output.append(max(n_out, 0))

fc2_output = []
for neuron in fc2_neurons:
  n_out = 0
  for j in range(len(fc1_output)):
    n_out += fc1_output[j] * neuron[j]
  fc2_output.append(n_out)

gradients = []
sm_output = []
sm_sum = 0.
max_v = max(fc2_output)
for output in fc2_output:
  sm_sum += math.exp(output - max_v)

for output in fc2_output:
  sm_output.append(math.exp(output - max_v) / sm_sum)

for output in sm_output:
  gradients.append(output)

print(gradients[0])
gradients[0] -= 1
loss = -math.log(gradients[0] + 1.)



weight_grad = [[] for i in range(fc2_n_neurons)]
```

```
for i in range(len(fc2_neurons)):
  for j in range(len(fc2_neurons[i])):
    weight_grad[i].append(gradients[i] * fc1_output[j])


fc1_grad = [0 for i in range(fc1_n_neurons)]
for i in range(len(fc2_neurons)):
  for j in range(len(fc2_neurons[i])):
    fc1_grad[j] += gradients[i] * fc2_neurons[i][j]

fc1_w_grad = [[] for i in range(fc1_n_neurons)]
for i in range(len(fc1_neurons)):
  for j in range(len(fc1_neurons[i])):
    de_dnet = fc1_output[i] > 0
    fc1_w_grad[i].append(de_dnet * fc1_grad[i] * fc0_output[j])

fc0_grad = [0 for i in range(fc0_n_neurons)]
for i in range(len(fc1_neurons)):
  for j in range(len(fc1_neurons[i])):
    de_dnet = fc1_output[i] > 0
    fc0_grad[j] += (de_dnet * fc1_grad[i]) * fc1_neurons[i][j]

fc0_w_grad = [[] for i in range(fc0_n_neurons)]
for i in range(len(fc0_neurons)):
  for j in range(len(fc0_neurons[i])):
    de_dnet = fc0_output[i] > 0
    fc0_w_grad[i].append(de_dnet * fc0_grad[i] * activations_i[j])

# conduct 10 gradient check tests
for i in range(10):

  #pick random weight in layer 0
  n_idx = random.randint(0, len(fc0_neurons) - 1)
  w_idx = random.randint(0, len(fc0_neurons) - 1)
  print('Calculated gradient:\t' + str(fc0_w_grad[n_idx][w_idx]))
  eps = 1e-4
  fc0_neurons[n_idx][w_idx] += eps
  fc0_output = []
  for neuron in fc0_neurons:
    n_out = 0
    for j in range(len(activations_i[0: fc0_fan_in])):
      n_out += activations_i[j + 0*fc0_fan_in] * neuron[j]
    fc0_output.append(max(n_out, 0))


  fc1_output = []
  for neuron in fc1_neurons:
    n_out = 0
    for j in range(len(fc0_output)):
      n_out += fc0_output[j] * neuron[j]
    fc1_output.append(max(n_out, 0))

  fc2_output = []
  for neuron in fc2_neurons:
    n_out = 0
```

```
    for j in range(len(fc1_output)):
      n_out += fc1_output[j] * neuron[j]
    fc2_output.append(n_out)

  gradients = []
  sm_output = []
  sm_sum = 0.
  max_v = max(fc2_output)
  for output in fc2_output:
    sm_sum += math.exp(output - max_v)

  for output in fc2_output:
    sm_output.append(math.exp(output - max_v) / sm_sum)

  for output in sm_output:
    gradients.append(output)


  gradients[0] -= 1
  loss2 = -math.log(gradients[0] + 1.)


  fc0_neurons[n_idx][w_idx] -= (2 * eps)
  fc0_output = []
  for neuron in fc0_neurons:
    n_out = 0
    for j in range(len(activations_i[0: fc0_fan_in])):
      n_out += activations_i[j + 0*fc0_fan_in] * neuron[j]
    fc0_output.append(max(n_out, 0))


  fc1_output = []
  for neuron in fc1_neurons:
    n_out = 0
    for j in range(len(fc0_output)):
      n_out += fc0_output[j] * neuron[j]
    fc1_output.append(max(n_out, 0))

  fc2_output = []
  for neuron in fc2_neurons:
    n_out = 0
    for j in range(len(fc1_output)):
      n_out += fc1_output[j] * neuron[j]
    fc2_output.append(n_out)

  gradients = []
  sm_output = []
  sm_sum = 0.
  max_v = max(fc2_output)
  for output in fc2_output:
    sm_sum += math.exp(output - max_v)

  for output in fc2_output:
    sm_output.append(math.exp(output - max_v) / sm_sum)

```

```
  for output in sm_output:
    gradients.append(output)


  gradients[0] -= 1
  loss3 = -math.log(gradients[0] + 1.)
  print('Numerical gradient:\t' + str((loss2 - loss3) / (2*eps)) +
      '\n')


print('\n--- FC0 OUT ---')
print('Neuron\t\tActivation')
for i in range(len(fc0_output)):
  print(str(i) + "\t\t\t" + str(fc0_output[i]))

print('\n--- FC1 OUT ---')
print('Neuron\t\tActivation')
for i in range(len(fc1_output)):
  print(str(i) + "\t\t\t" + str(fc1_output[i]))

print('\n--- FC2 OUT ---')
print('Neuron\t\tActivation')
for i in range(len(fc2_output)):
  print(str(i) + "\t\t\t" + str(fc2_output[i]))

print('\n--- SOFTMAX OUT ---')
print('Neuron\t\tActivation')
for i in range(len(sm_output)):
  print(str(i) + "\t\t\t" + str(sm_output[i]))

print('\n--- FC2 NEURON GRADIENTS ---')
for i in range(len(gradients)):
  print(str(i) + ": " + str(gradients[i]))


print('\n--- FC2 WEIGHT GRADIENTS ---')
for i in range(len(weight_grad)):
  print("Neuron " + str(i))
  for j in range(len(weight_grad[i])):
    print(str(j) + ": " + str(weight_grad[i][j]))

print ('\n--- FC1 NEURON GRADIENTS ---')
for i in range(len(fc1_grad)):
  print(str(i) + ": " + str(fc1_grad[i]))

print('\n--- FC1 WEIGHT GRADIENTS ---')
for i in range(64):
  print("Neuron " + str(i))
  for j in range(98):
    print(str(j) + ": " + str(fc1_w_grad[i][j]))


print ('\n--- FC0 NEURON GRADIENTS ---')
for i in range(len(fc0_grad)):
  print(str(i) + ": " + str(fc0_grad[i]))
```

```

print('\n--- FC0 WEIGHT GRADIENTS ---')
for i in range(98):
  print("Neuron " + str(i))
  for j in range(784):
    print(str(j) + ": " + str(fc0_w_grad[i][j]))


#Plot weight gradient distribution
no_zero_fc0 = []
no_zero_fc1 = []
no_zero_fc2 = []

for i in range(len(fc0_w_grad)):
  for grad in fc0_w_grad[i]:
    if grad != 0.0:
      no_zero_fc0.append(grad)

for i in range(len(fc1_w_grad)):
  for grad in fc1_w_grad[i]:
    if grad != 0.0:
      no_zero_fc1.append(grad)

for i in range(len(weight_grad)):
  for grad in weight_grad[i]:
    if grad != 0.0:
      no_zero_fc2.append(grad)

print('Fc0: ' + str(len(no_zero_fc0) / (fc0_n_neurons * fc0_fan_in)))
print('Fc1: ' + str(len(no_zero_fc1) / (fc1_n_neurons * fc1_fan_in)))
print('Fc2: ' + str(len(no_zero_fc2) / (fc2_n_neurons * fc2_fan_in)))

no_zero_fc0 = np.asarray(no_zero_fc0)
no_zero_fc1 = np.asarray(no_zero_fc1)
no_zero_fc2 = np.asarray(no_zero_fc2)

sns.distplot(no_zero_fc0, label='FC0', bins = 70, norm_hist=True)
sns.distplot(no_zero_fc1, label='FC1', bins = 70, norm_hist=True)
sns.distplot(no_zero_fc2, label='FC2', bins = 70, norm_hist=True)
plt.xlabel('Weight Value')
plt.ylabel('Number of weights in bin')
plt.legend()
plt.xlim(-0.06, 0.06)
plt.show()
```

APPENDIX E

Weight Generation

This appendix contains the code that was used to generate the Xilinx coefficient files to initialize the weight BRAMs in the hardware model using He Initialization. This code was written in Python.

E.1 weight_coeff.py

```
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 F

Software Model

This appendix contains the code that was used to implement the software model for this project. The software model was written in C++.

F.1 Header Files

F.1.1 net.h

```
#ifndef __NET_H
#define __NET_H

#include <vector>
#include <stdint.h>
#include "layer.h"

class Net {
    std::vector<Layer*> layers;
    std::vector< std::vector< std::vector<double> > > activations;
    std::vector< std::vector<double> > batch_output;
    std::vector< std::vector<double> > ol_gradient;
    uint32_t batch_size;
    uint32_t input_size;
    uint32_t output_size;
```

```
    double learning_rate;
    double momentum;

public:

    std::vector< std::vector<double> > operator() (std::vector < std::vector<double> > input);
    void addLayer(Layer*);
    std::vector< std::vector<double> > inference(std::vector< std::vector<double> > input);
    double computeLossAndGradients(std::vector<int> labeled);
    void backpropLoss();
    void update();
    void clearSavedData();

    std::vector<double> convLogitToProb(std::vector<double> logits);
    std::vector<double> getPredictions();

    const uint32_t getBatchSize() const { return batch_size; }
    std::vector< std::vector<double> > getOlGradient() const { return ol_gradient; }
    std::vector< std::vector< std::vector<double> > > getActivations() const { return activations; }
    std::vector<Layer*> getLayers() { return layers; }
    void setLearningRate(double lr) { learning_rate = lr; };
    const double getLearningRate() const& { return learning_rate; };

    Net(uint32_t in, uint32_t out, uint32_t bs, double lr, double momentum);
    Net(const Net& net);
    ~Net();
};

#endif
```

F.1.2 layer.h

```
#ifndef __LAYER_H
#define __LAYER_H

#include "neuron.h"
#include <vector>

#define FULLY   0
#define CONV    1
#define POOL    2

class Layer {
public:
    bool last_layer;
    virtual void forward(std::vector<double>) = 0;
    virtual void forward(std::vector<double>, bool) = 0;
    virtual std::vector< std::vector<double> > backward(std::vector
        < std::vector<double> >,
      std::vector< std::vector<double> >,
      std::vector< std::vector<double> >) = 0;
    virtual void updateWeights(double lr, double momentum) = 0;
    virtual void clearData() = 0;
    virtual const std::vector<double>& getOutput() = 0;
    virtual std::vector<Neuron>& getNeurons() = 0;
    virtual void setNeurons (const std::vector<Neuron>& n) = 0;
    virtual int getType() = 0;
    virtual ~Layer() {};
};

#endif
```

F.1.3 convolutional.h

```
#ifndef __CONVOLUTIONAL_H
#define __CONVOLUTIONAL_H

#include <vector>
#include <stdint.h>
#include "layer.h"
#include "neuron.h"

class ConvLayer : public Layer {

    uint32_t dim;
    uint32_t filt_size;
    uint32_t stride;
    uint32_t padding;
    uint32_t in_channels;
    uint32_t out_channels;
    uint32_t dim_o;

    std::vector<Neuron> neurons;
    std::vector<double> output;

public:
    ConvLayer (uint32_t dim, uint32_t filt_size, uint32_t stride,
        uint32_t padding, uint32_t in_channels, uint32_t
        out_channels);
    ConvLayer(const ConvLayer& A);
    ~ConvLayer ();

    void forward(std::vector<double>);
    void forward(std::vector<double>, bool in);
    std::vector< std::vector<double> > backward (std::vector< std::
        vector<double> >,
      std::vector< std::vector<double> >,
      std::vector< std::vector<double> >) ;
    void updateWeights(double lr, double momentum);
    void clearData();
    std::vector<Neuron>& getNeurons() { return neurons; };
    const std::vector<double>& getOutput() { return output; };
    const uint32_t getDim() const { return dim; }
    const uint32_t getInChannels() const { return in_channels; }
    const uint32_t getOutChannels() const { return out_channels; }
    const uint32_t getFiltSize() const { return filt_size; }
    std::vector<double> getWindowPixels(const std::vector<double>&
        input, int row, int col);
    int getType() { return CONV; }

    void setNeurons (const std::vector<Neuron>& n) {neurons = n;}


};

#endif
```

F.1.4 fullyconnected.h

```
#ifndef __FULLYCONNECTED_H
#define __FULLYCONNECTED_H

#include <stdint.h>
#include <vector>

#include "neuron.h"
#include "layer.h"

class FullyConnected : public Layer {

    uint32_t input_size;
    uint32_t output_size;
    std::vector<Neuron> neurons;
    std::vector<double> output;

public:
    FullyConnected(uint32_t in, uint32_t out);
    FullyConnected(const FullyConnected& x) {
        input_size = x.input_size;
        output_size = x.output_size;
        neurons = x.neurons;
    }
    ~FullyConnected() {}

    void forward(std::vector<double> input);
    void forward(std::vector<double> input, bool last_layer);

    std::vector< std::vector<double> > backward(
        std::vector< std::vector<double> > gradients_ps,
        std::vector< std::vector<double> > in_activations,
        std::vector< std::vector<double> > out_activations);

    void updateWeights(double lr, double momentum);
    void clearData();

    void setNeurons (const std::vector<Neuron>& n) {neurons = n;}

    const std::vector<double>& getOutput() { return output; }
    std::vector<Neuron>& getNeurons() { return neurons; }
    int getType() { return FULLY; }
};

#endif
```

F.1.5 pooling.h

```cpp
#ifndef __POOLING_H
#define __POOLING_H

#include <stdint.h>
#include <vector>
#include "layer.h"
#include "neuron.h"

class PoolingLayer : public Layer {
    uint32_t dim_i;
    uint32_t dim_o;
    uint32_t channels;
    std::vector<Neuron> placeholder;
    std::vector<double> output;

public:
    PoolingLayer(uint32_t d_i, uint32_t d_o, uint32_t c) :
      dim_i(d_i), dim_o(d_o), channels(c),
      output(std::vector<double>(d_o * d_o * c, 0)) {last_layer = false;}
    PoolingLayer(const PoolingLayer& p) {
        dim_i = p.dim_i;
        dim_o = p.dim_o;
        channels = p.channels;
        output = p.output;
    }
    ~PoolingLayer() {};


    void forward(std::vector<double>);
    void forward(std::vector<double>, bool);
    std::vector< std::vector<double> > backward(std::vector< std::vector<double> >,
      std::vector< std::vector<double> >,
      std::vector< std::vector<double> >);

    std::vector<double> getWindowPixels (const std::vector<double>& input,
                          uint32_t ch, uint32_t row, uint32_t col);

    void updateWeights(double lr, double momentum) {};
    void clearData() {}
    const std::vector<double>& getOutput() { return output; }
    std::vector<Neuron>& getNeurons() { return placeholder; }
    void setNeurons (const std::vector<Neuron>& n) {};
    int getType() { return POOL; };
};


#endif
```

F.1.6 neuron.h

```
#ifndef __NEURON_H
#define __NEURON_H

#include <stdint.h>
#include <vector>

class Neuron {
    std::vector<double> weights;
    std::vector<double> gradient_per_weight;
    std::vector<double> momentum_per_weight;
    double              offset_gradient;
    double              offset_momentum;
    double              offset;
    double              de_dnet;
    uint32_t            fan_in;
    double              net;
    double              activation;

public:

    Neuron(uint32_t in);
    Neuron(const Neuron& n);
    ~Neuron();

    void initWeights(); // He initialization
    double computeNet(std::vector<double> input);
    double computeActivation();
    void calculateGradient(double grad, std::vector<double> act_in,
                           double act_out, bool last_layer);
    void updateWeights(double lr, double momentum);
    void clearBackwardData();

    const double& getActivation() { return activation; }
    const double& getSensitivity() { return de_dnet; }
    void setOffset(double offset) { this->offset = offset; }
    const double& getOffset() { return offset; }
    void setWeights(std::vector<double> weights) { this->weights =
        weights; }
    const std::vector<double>& getWeights() { return weights; }
    const std::vector<double>& getGradients() { return
        gradient_per_weight; }
};

#endif
```

F.1.7 parse_data.h

```
#ifndef __PARSE_DATA_H
#define __PARSE_DATA_H

#include <string>
#include <vector>
#include <stdint.h>

std::vector<int> readLabels(std::string filename);
std::vector< std::vector<double> > readImages(std::string filename);

#endif
```

F.2 Source Files

F.2.1 main.cpp

```cpp
#include <iostream>
#include <random>
#include <time.h>
#include <chrono>

#include "convolutional.h"
#include "fullyconnected.h"
#include "pooling.h"
#include "parse_data.h"
#include "layer.h"
#include "net.h"

double printAccuracy(Net& net, std::vector< std::vector<double> >& in, std::vector<int>& out);
void trainNet(Net& net, std::vector< std::vector<double> >& in, std::vector<int>& out,
  std::vector< std::vector<double> >& in_test, std::vector<int>& out_test, int n_epochs,
  int epochs_per_change, double geometric_rate);
int main () {

    std::cout << "Running software model...\n";

    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");
    int n_epochs = 50;

    int input_size = 28*28;
    int output_size = 10;
    int batch_size = 1;
    double momentum = 0.9;
    double lr = 0.001;
    Net net(input_size, output_size, batch_size, lr, momentum);

    trainX = std::vector< std::vector<double> > (trainX.begin(), trainX.begin() + 60000);
    trainY = std::vector<int> (trainY.begin(), trainY.begin() + 60000);


    testX = std::vector< std::vector<double> > (testX.begin(), testX.begin() + 10000);
```

```
    testY = std::vector<int> (testY.begin(), testY.begin() + 10000)
        ;

/*
    Convolutional configuration

    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);
*/

    Layer* fc1 = new FullyConnected(28*28, 98);
    Layer* fc2 = new FullyConnected(98, 64);
    Layer* fc3 = new FullyConnected(64, 10);

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

    trainNet(net, trainX, trainY, testX, testY, n_epochs, 25, .1);

    printAccuracy(net, testX, testY);
}

void trainNet(Net& net, std::vector< std::vector<double> >& in, std
    ::vector<int>& out,
            std::vector< std::vector<double> >& in_test, std::
                vector<int>& out_test, int n_epochs,
            int epochs_per_change, double geometric_rate) {
    std::cout << "Starting Accuracy" << std::endl;
    printAccuracy(net, in_test, out_test);
    std::cout <<std::endl;
    clock_t start, end, diff;
    start = clock();
    for (int i = 0; i <= n_epochs; i++) {
        double train_loss = 0.0;
        int batch_size = net.getBatchSize();
        int lb = 0;
        int ub = batch_size;
        int size = in.size();
        while (ub <= size) {

            /* Get the batch */
            std::vector< std::vector<double> >::iterator startX =
                in.begin() + lb;
```

```
            std::vector< std::vector<double> >::iterator endX = in.begin() + ub;
            std::vector<int>::iterator startY = out.begin() + lb;
            std::vector<int>::iterator endY = out.begin() + ub;

            std::vector< std::vector<double> > in_batch(startX, endX);
            std::vector<int> out_batch(startY, endY);
            /* Train by batch size! */
            net(in_batch);
            train_loss += net.computeLossAndGradients(out_batch);

            net.backpropLoss();
            net.update();
            net.clearSavedData();

            lb += batch_size;
            ub += batch_size;
        }
        end = clock();
        diff = end - start;
        std::cout << "Epoch: " << i << std::endl;
        std::cout << "\n--- Training Stats ---\n";
        train_loss = printAccuracy(net, in, out);
        std::cout << "Loss: " << train_loss / (double)in.size() << std::endl;
        std::cout << "\n--- Test Stats ---\n";
        double test_loss = printAccuracy(net, in_test, out_test);
        std::cout << "Elapsed time: " << (float)diff / CLOCKS_PER_SEC << std::endl;
        std::cout << "Loss: " << test_loss / (double)in_test.size() << std::endl << std::endl;
        if ( (i + 1) % epochs_per_change == 0) {
            std::cout << "Learning rate changed from " << net.getLearningRate();
            net.setLearningRate(net.getLearningRate() * geometric_rate);
            std::cout << " to " << net.getLearningRate() << std::endl << std::endl;
        }
    }
}

double printAccuracy(Net& net, std::vector< std::vector<double> >& in, std::vector<int>& out) {
    auto result = net(in);
    int corr = 0;
    for (size_t i = 0; i < result.size(); i++) {
        int max_idx = 0;
        double max = result[i][0];
        for (size_t j = 1; j < result[i].size(); j++) {
            if (result[i][j] > max) {
                max_idx = j;
                max = result[i][j];
            }
```

```
        }
        if (max_idx == out[i]) {
            corr++;
        }
    }
    double loss = net.computeLossAndGradients(out);
    net.clearSavedData();
    std::cout << "Total correct: " << corr << " / " << result.size
        () << std::endl;
    std::cout << "Accuracy: " << (double)corr / result.size() <<
        std::endl;
    return loss;
}
```

F.2.2 net.cpp

```
#include "net.h"
#include <string>
#include <math.h>
#include <algorithm>
#include <iostream>
#include "convolutional.h"
#include "fullyconnected.h"

void Net::addLayer(Layer* layer) {
    if (layers.size()) {
        layers[layers.size() - 1]->last_layer = false;
    }
    layers.push_back(layer);
    layers[layers.size() - 1]->last_layer = true;
}

std::vector< std::vector<double> > Net::operator() (std::vector <
    std::vector<double> > input) {
    return inference(input);
}

std::vector< std::vector<double> > Net::inference(std::vector< std
    ::vector<double> > input) {
    batch_output = std::vector< std::vector<double> >();
    activations = std::vector< std::vector< std::vector<double> > >
         ();
    for (size_t i = 0; i < layers.size() + 1; i++) {
        activations.push_back(std::vector< std::vector<double> >())
            ;
    }

    for (std::vector<double> in : input) {
        if (in.size() != input_size) {
            std::cout << "Input size does not match, expected: " +
                std::to_string(input_size) +
            ", got: " + std::to_string(in.size()) << std::endl;
            exit(1);
        }

        for (size_t i = 0; i < layers.size(); i++) {
            Layer*&l = layers[i];
            /*double max = *(std::max_element(in.begin(), in.end())
                );
            for (double& e : in) {
                e /= max;
            }*/
            activations[i].push_back(in);
            if (i == layers.size() - 1) {
                l->forward(in, true);
            }
            else {
                l->forward(in);
            }
```

```cpp
            in = l->getOutput();
        }


        if (in.size() != output_size) {
            std::cout << "Output size does not match, expected: " + std::to_string(output_size) + ", got: " + std::to_string(in.size());
            std::cout << std::endl;
            exit(1);
        }
        activations[layers.size()].push_back(in);
        std::vector<double> output = convLogitToProb(in);
        //std::vector<double> output = in;
        batch_output.push_back(output);
    }
    return activations[layers.size()];
    //return batch_output;
}

double Net::computeLossAndGradients(std::vector<int> labeled) {
    if (labeled.size() != batch_output.size()) {
        std::cout << "Labeled data size does not match the net's output size, expected: " +
                    std::to_string(batch_output.size()) + ", got: " + std::to_string(labeled.size()) << std::endl;
        std::cout << std::endl;
        exit(1);
    }
    ol_gradient = std::vector< std::vector<double> >();
    double loss = 0;
    // Compute cross entropy loss for each output
    // CrossEntropy loss -q(x) * log(p(x))
    // q(x) is true distribution, so it is 1 for our labeled data on the correct sample
    for (size_t i = 0; i < labeled.size(); i++) {
        std::vector<double> gradient(output_size, 0);
        unsigned short label = labeled[i];

        for (size_t j = 0; j < output_size; j++) {
            //double f_out = (tanh(batch_output[i][j] / 2.))/2. + 0.5;
            if (j == label) {
                //gradient[j] = batch_output[i][j] * (1 - batch_output[i][j]);
                gradient[j] = batch_output[i][j] - 1;

                //gradient[j] = (f_out - 1) * (f_out*(1-f_out));
            }
            else {
                //gradient[j] = -batch_output[i][j] * batch_output[i][label];
                gradient[j] = batch_output[i][j];
                //gradient[j] = (f_out) * (f_out*(1-f_out));
```

```
            }
        }
        loss += -log(batch_output[i][label]);
        ol_gradient.push_back(gradient);
    }


    // Compute mean square error
    /*for (size_t i = 0; i < labeled.size(); i++) {
        unsigned short label = labeled[i];
        std::vector<double> gradient(output_size, 0);
        for (size_t j = 0; j < output_size; j++) {
            double err, grad;
            if (j == label) {
                grad = 1 - batch_output[i][j];
                err = 0.5 * pow(grad, 2);
            }
            else {
                grad = 0 - batch_output[i][j];
                err = 0.5 * pow(grad, 2);
            }
            gradient[j] = grad;
            loss += err;
        }
        ol_gradient.push_back(gradient);
    }*/

    return loss;
}

// Backpropagate the gradients of the error
void Net::backpropLoss() {
    std::vector< std::vector<double> > gradients = ol_gradient;
    std::vector< std::vector<double> > sens;
    // Outer layer gradients is just the loss
    for (int i = layers.size() - 1; i >= 0; i--) {
        //std::cout << "Grad len: " << gradients[0].size() << "
            Layer len: " << layers[i]->getNeurons().size() << "
            Prev: " << activations[i].size() << " Next: " <<
            activations[i+1].size() << std::endl;
        sens = layers[i]->backward(gradients, activations[i],
            activations[i + 1]);
        // fully connected gradients
        // if fully connected check, on layers[i]
        gradients = std::vector< std::vector<double> >();
        Layer* l = layers[i];
        std::vector<Neuron> neurons = l->getNeurons();

        if (l->getType() == POOL) {
            gradients = sens;
        }
        else {
            for (size_t j = 0; j < sens.size(); j++) {
                if (l->getType() == FULLY) {
```

```
                // The gradient of neuron i in prev layer is the sum of the weights[i] * de_dnet of all
                // neurons in layer j
                std::vector<double> grad(neurons[0].getWeights().size(), 0);
                for (size_t k = 0; k < neurons.size(); k++) {
                    std::vector<double> weights = neurons[k].getWeights();
                    for (size_t l = 0; l < weights.size(); l++) {
                        grad[l] += sens[j][k] * weights[l];
                    }
                }
                gradients.push_back(grad);
            }
            else if (l->getType() == CONV) {
                ConvLayer* cl = dynamic_cast<ConvLayer* >(l);
                // If the sensitivities of i + 1 layer were from a convolution, then the
                // neurons for layer i only need to do weights[i] * de_dnet for
                // the relevant windows that the activation was in
                int dim = cl->getDim();
                int in_chan = cl->getInChannels();
                int out_chan = cl->getOutChannels();

                int num_neurons = dim * dim * in_chan;  // amount of gradients to give previous layer
                std::vector<double> grad(num_neurons, 0);

                for (int k = 0; k < num_neurons; k++) {
                    // for each window it goes to... need to know which weight to use
                    int chan = k / (dim * dim);
                    int row = (k - (chan * dim * dim)) / dim;
                    int col = (k - (chan * dim * dim + row * dim)) % dim;
                    int filt_size = cl->getFiltSize();
                    int filt_sq = filt_size * filt_size;
                    // Iterate over the neurons in the window for this gradient
                    int dim_sq = dim * dim;
                    int start_row = row - (filt_size / 2);
                    int end_row = row + (filt_size / 2);
                    int start_col = col - (filt_size / 2);
                    int end_col = col + (filt_size / 2);

                    /*std::cout << "Row: " << row << " Col: " << col << " Start row: " << start_row <<
                    " End row: " << end_row << " Start col: "
                    << start_col << " End col: " << end_col << " j: " << j
```

```
                        << " Sens size: " << sens.size() << std::endl;*/

                        for (int o = 0; o < out_chan; o++) {
                            int count = 0;
                            for (int m = start_row; m <= end_row; m++) {
                                for (int n = start_col; n <= end_col; n++) {
                                    if (m < 0 || m >= dim || n < 0 || n >= dim) {
                                        count++;
                                        continue;
                                    }
                                    int o_neur_idx = o * dim_sq + m * dim + n;
                                    int filt_offset = (filt_sq) - (count + 1);
                                    int weight_idx = (chan * filt_sq) + filt_offset;

                                    grad[k] += sens[j][o_neur_idx] * neurons[o_neur_idx].getWeights()[weight_idx];
                                    count++;
                                }
                            }
                        }
                    }
                    gradients.push_back(grad);
                }
            }
        }
    }
}
void Net::update() {
    double effective_lr = learning_rate / batch_size;
    for (int i = layers.size() - 1; i >= 0; i--) {
        layers[i]->updateWeights(effective_lr, momentum);
    }
}

std::vector<double> Net::convLogitToProb(std::vector<double> logits) {
    double sum = 0;
    double max = *std::max_element(logits.begin(), logits.end());
    for (auto l : logits) {
        sum += exp(l - max);
    }
    std::vector<double> prob;

    for (auto l : logits) {
        prob.push_back(exp(l - max) / sum);
    }

```

```
    return prob;
}

std::vector<double> Net::getPredictions() {
    std::vector<double> preds;
    for (size_t i = 0; i < batch_output.size(); i++) {
        int pred_class = 0;
        double pred_max = batch_output[i][0];
        for (size_t j = 1; j < output_size; j++) {
            if (batch_output[i][j] > pred_max) {
                pred_class = j;
            }
        }
        preds.push_back(pred_class);
    }
    return preds;
}

void Net::clearSavedData() {
    activations = std::vector< std::vector< std::vector<double> >
        >();
    batch_output = std::vector< std::vector<double> >();
    ol_gradient = std::vector< std::vector<double> >();
    /*for (int i = layers.size() - 1; i >= 0; i--) {
        layers[i]->clearData();
    }*/
    for (Layer* l : layers) {
        l->clearData();
    }
}

Net::Net(uint32_t in, uint32_t out, uint32_t bs, double lr, double
    moment) {
    layers = std::vector<Layer*>();
    input_size = in;
    output_size = out;
    batch_size = bs;
    learning_rate = lr;
    momentum = moment;
}

Net::Net(const Net& net) {
    layers = net.layers;
    input_size = net.input_size;
    output_size = net.output_size;
}

Net::~Net() {
    for (size_t i = 0; i < layers.size(); i++) {
        delete layers[i];
    }
}
```

F.2.3 convolutional.cpp

```cpp
#include "convolutional.h"

#include <iostream>

void ConvLayer::forward(std::vector<double> input) {
    uint32_t h_steps = 1 + ((dim + (padding * 2) - filt_size) / stride);

    if (input.size() != h_steps * h_steps * in_channels) {
        std::cout << "Wrong input size for convolutional layer\n";
        exit(1);
    }

    unsigned start = (filt_size / 2) - padding;

    for (unsigned int i = 0; i < out_channels; i++) { // channel of output
        for (unsigned int k = 0; k < dim_o; k++) { // row
            for (unsigned int l = 0; l < dim_o; l++) { // column
                int row = k * stride + start;
                int col = l * stride + start;
                std::vector<double> pixels = getWindowPixels(input, row, col);
                int out_idx = i * dim_o * dim_o + k * dim_o + l;
                neurons[out_idx].computeNet(pixels);
                output[out_idx] = neurons[out_idx].computeActivation();
            }
        }
    }

}

void ConvLayer::forward(std::vector<double> input, bool in) {
    forward(input);
}

std::vector<double> ConvLayer::getWindowPixels(const std::vector<double>& input, int row, int col) {
    std::vector<double> pixels;

    int start_row = row - (filt_size / 2);
    int end_row = row + (filt_size / 2);
    int start_col = col - (filt_size / 2);
    int end_col = col + (filt_size / 2);
    int dim_sq = dim * dim;

    for (unsigned int ch = 0; ch < in_channels; ch++) {
        for (int i = start_row; i <= end_row; i++) {
            for (int j = start_col; j <= end_col; j++) {
                int idx = ch * dim_sq + i * dim + j;
                if (i < 0 || i >= (int)dim || (j < 0 || j >= (int)dim)) {
```

```
                    // in padding region, just push 0
                    pixels.push_back(0);
                }
                else {
                    pixels.push_back(input[idx]);
                }
            }
        }
    }
    return pixels;
}

std::vector< std::vector<double> > ConvLayer::backward (std::vector< std::vector<double> > gradients,
                                                        std::vector< std::vector<double> > in_activations,
                                                        std::vector< std::vector<double> > out_activations) {

    std::vector< std::vector<double> > sensitivity;
    int dim_sq = dim_o * dim_o;

    unsigned start = (filt_size / 2) - padding;
    for (size_t i = 0; i < gradients.size(); i++) {
        std::vector<double> single_sens;
        for (size_t j = 0; j < out_channels; j++) {
            for (size_t k = 0; k < dim_o; k++) {
                for (size_t l = 0; l < dim_o; l++) {
                    // Neuron at row k, column l, in output channel j
                    int idx = l + k * dim_o + j * dim_sq;
                    int row = k * stride + start;
                    int col = l * stride + start;
                    std::vector<double> in_act = getWindowPixels(in_activations[i], row, col);
                    neurons[idx].calculateGradient(gradients[i][idx], in_act, out_activations[i][idx], last_layer);
                    single_sens.push_back(neurons[idx].getSensitivity());
                }
            }
        }
        sensitivity.push_back(single_sens);
    }
    return sensitivity;
}


void ConvLayer::updateWeights(double lr, double momentum) {
```

```
    for (Neuron& n : neurons) {
        n.updateWeights(lr, momentum);
    }
}



void ConvLayer::clearData() {
    for (Neuron& n : neurons) {
        n.clearBackwardData();
    }
}


ConvLayer::ConvLayer(uint32_t d, uint32_t fsize, uint32_t str, uint32_t pad, uint32_t in_ch, uint32_t out_ch) {
    last_layer = false;
    dim = d;
    in_channels = in_ch;
    out_channels = out_ch;
    filt_size = fsize;
    stride = str;
    padding = pad;

    uint32_t weights_per_fmap = filt_size * filt_size * in_channels;

    uint32_t steps = 1 + ((dim + (padding * 2) - filt_size) / stride);
    uint32_t num_neurons = steps * steps * out_channels;

    dim_o = steps;


    output.resize(num_neurons);
    neurons.reserve(num_neurons);

    for (uint32_t i = 0; i < num_neurons; i++) {
        Neuron n(weights_per_fmap);
        n.initWeights();
        neurons.push_back(n);
    }
}

ConvLayer::ConvLayer(const ConvLayer& a) {
    dim = a.dim;
    in_channels = a.in_channels;
    out_channels = a.out_channels;
    filt_size = a.filt_size;
    neurons = a.neurons;
    output = a.output;
}

ConvLayer::~ConvLayer() {

```

```
}
```

F.2.4 fullyconncted.cpp

```
#include "fullyconnected.h"

#include <iostream>

void FullyConnected::forward(std::vector<double> input) {
    for (size_t i = 0; i < output_size; i++) {
        neurons[i].computeNet(input);
        output[i] = neurons[i].computeActivation();
    }
}

void FullyConnected::forward(std::vector<double> input, bool
    last_layer) {
    if (last_layer) {
        for (size_t i = 0; i < output_size; i++) {
            output[i] = neurons[i].computeNet(input);
        }
    }
    else {
        forward(input);
    }
}

std::vector< std::vector<double> > FullyConnected::backward(
                    std::vector< std::vector<double> > gradients,
                    std::vector< std::vector<double> >
                        in_activations,
                    std::vector< std::vector<double> >
                        out_activations) {
    std::vector< std::vector<double> > sensitivity;
    for (size_t i = 0; i < gradients.size(); i++) {
        std::vector<double> sngl_sens;
        for (size_t j = 0; j < neurons.size(); j++) {
            neurons[j].calculateGradient(gradients[i][j],
                in_activations[i], out_activations[i][j],
                last_layer);
            sngl_sens.push_back(neurons[j].getSensitivity());
        }
        sensitivity.push_back(sngl_sens);
    }
    return sensitivity;
}

void FullyConnected::updateWeights(double lr, double momentum) {
    for (Neuron& n : neurons) {
        n.updateWeights(lr, momentum);
    }
}

void FullyConnected::clearData() {
    for (Neuron& n : neurons) {
        n.clearBackwardData();
    }
```

```
}

FullyConnected::FullyConnected(uint32_t in, uint32_t out) :
    input_size(in), output_size(out) {
    last_layer = false;
    neurons.reserve(out);
    output.resize(out);
    for (size_t i = 0; i < out; i++) {
        Neuron n(in);
        n.initWeights();
        neurons.push_back(n);
    }
}
```

F.2.5 pooling.cpp

```
#include "pooling.h"
#include <iostream>
#include <algorithm>

void PoolingLayer::forward(std::vector<double> in) {
    for (size_t c = 0; c < channels; c++) {
        for (size_t i = 0; i < dim_i - 1; i += 2) {
            for (size_t j = 0; j < dim_i - 1; j += 2) {
                int row_o = i / 2;
                int col_o = j / 2;
                int out_idx = c * dim_o * dim_o + row_o * dim_o +
                    col_o;

                std::vector<double> pixels = getWindowPixels(in, c,
                     i, j);
                double max = *std::max_element(pixels.begin(),
                    pixels.end());
                output[out_idx] = max;
            }
        }
    }
}

void PoolingLayer::forward(std::vector<double> in, bool first) {
    forward(in);
}

std::vector<double> PoolingLayer::getWindowPixels(const std::vector
    <double>& input,
                                                     uint32_t ch,
                                                         uint32_t
                                                         row,
                                                         uint32_t
                                                         col) {
    int offset = dim_i * dim_i * ch + dim_i * row + col;
    std::vector<double> pixels;
    pixels.push_back(input[offset]);
    pixels.push_back(input[offset + 1]);
    pixels.push_back(input[offset + dim_i]);
    pixels.push_back(input[offset + dim_i + 1]);

    return pixels;
}

std::vector< std::vector<double> > PoolingLayer::backward(
                         std::vector< std::vector<double> > gradients,
                         std::vector< std::vector<double> >
                             in_activations,
                         std::vector< std::vector<double> >
                             out_activations) {
    std::vector< std::vector<double> > sensitivity;
    for (size_t i = 0; i < gradients.size(); i++) {
        std::vector<double> sngl_sens(channels * dim_i * dim_i, 0);
```

```
        for (size_t c = 0; c < channels; c++) {
            for (size_t j = 0; j < dim_o; j++) {
                for (size_t k = 0; k < dim_o; k++) {
                    int offset_o = c * dim_o * dim_o + j * dim_o + k;
                    int row_i = j * 2;
                    int col_i = k * 2;
                    int offset_i = c * dim_i * dim_i + row_i * dim_i + col_i;
                    std::vector<double> pixels = getWindowPixels(in_activations[i], c, row_i, col_i);

                    double max = *std::max_element(pixels.begin(), pixels.end());

                    // write the gradients for each pixel
                    std::vector<double> window_gradients(4, 0);
                    for (size_t l = 0; l < pixels.size(); l++) {
                        if (pixels[l] == max) {
                            window_gradients[l] = gradients[i][offset_o];
                        }
                    }
                    sngl_sens[offset_i] = window_gradients[0];
                    sngl_sens[offset_i + 1] = window_gradients[1];
                    sngl_sens[offset_i + dim_i] = window_gradients[2];
                    sngl_sens[offset_i + dim_i + 1] = window_gradients[3];
                }
            }
        }
        sensitivity.push_back(sngl_sens);
    }
    return sensitivity;
}
```

F.2.6 neuron.cpp

```
#include "neuron.h"

#include <random>
#include <chrono>
#include <math.h>
#include <iostream>

Neuron::Neuron(uint32_t in) {
    fan_in = in;
    gradient_per_weight = std::vector<double> (fan_in, 0);
    momentum_per_weight = std::vector<double> (fan_in, 0);
    offset_gradient = 0;
    offset_momentum = 0;
}

Neuron::Neuron(const Neuron& n) {
    weights = n.weights;
    gradient_per_weight = n.gradient_per_weight;
    momentum_per_weight = n.momentum_per_weight;
    offset_gradient = n.offset_gradient;
    offset_momentum = n.offset_momentum;
    offset = n.offset;
    fan_in = n.fan_in;
    net = n.net;
    activation = n.activation;
}

Neuron::~Neuron() {

}

/**
 * Uses He initialization
 */
void Neuron::initWeights() {
    weights.reserve(fan_in);
    std::normal_distribution<double> distribution(0, sqrt(2. / (
        fan_in)));

    static unsigned seed = std::chrono::system_clock::now().
        time_since_epoch().count();
    static std::default_random_engine generator(seed);

    for (size_t i = 0; i < fan_in; i++) {
        weights.push_back(distribution(generator));
    }
    offset = distribution(generator);
}

/**
 * Compute net for the neuron
 */
double Neuron::computeNet(std::vector<double> input) {
```

```
    if (input.size() != weights.size()) {
        std::cerr << "Input size did not match size of weights. Input size: " <<
                        input.size() << " Weight size: " << weights.size() << std::endl;
        exit(1);
    }

    // Dot product and sum offset
    net = offset;
    for (size_t i = 0; i < fan_in; i++) {
        net += input[i] * weights[i];
    }

    return net;
}

/**
 * Compute output for neuron
 */
double Neuron::computeActivation() {
    activation = std::max(net, 0.);    // ReLU
    return activation;
}

void Neuron::calculateGradient(double grad, std::vector<double> act_in, double act_out, bool last_layer) {
    double dact_dnet = (!last_layer) ? (act_out > 0) : 1;
    de_dnet = grad * dact_dnet;

    for (size_t i = 0; i < fan_in; i++) {
        gradient_per_weight[i] += (de_dnet * act_in[i]);
    }
    offset_gradient += de_dnet;
}

void Neuron::updateWeights(double lr, double momentum) {
    // update weights
    for (size_t i = 0; i < fan_in; i++) {
        momentum_per_weight[i] = momentum * momentum_per_weight[i] + (lr * -gradient_per_weight[i]);
        weights[i] += momentum_per_weight[i];
    }
    offset_momentum = offset_momentum * momentum + (lr * -offset_gradient);
    offset += offset_momentum;
    clearBackwardData();
}

void Neuron::clearBackwardData() {
    gradient_per_weight = std::vector<double>(fan_in, 0);
    offset_gradient = 0;
}
```

F.2.7 parse_data.cpp

```cpp
#include "parse_data.h"
#include <fstream>
#include <iostream>

std::vector< std::vector<double> > readImages(std::string s) {
    std::ifstream f(s, std::ios::binary | std::ios::in);

    std::vector< std::vector<double> > data;
    uint8_t pixel;
    uint32_t rows;
    uint32_t cols;

    uint8_t buf[4];


    f.read((char*)&buf, 4);  // magic number
    f.read((char*)&buf, 4);  // number of images
    f.read((char*)&buf, 4);  // number of rows
    rows = buf[3] + (buf[2] << 8) + (buf[1] << 16) + (buf[0] << 24);
    f.read((char*)&buf, 4);  // number of cols
    cols = buf[3] + (buf[2] << 8) + (buf[1] << 16) + (buf[0] << 24);


    while(!f.eof()) {
        // read an image
        std::vector<double> img;
        for (unsigned i = 0; i < rows * cols; i++) {
            f.read((char*)&pixel, 1);
            double p = ((double)pixel / 255.) ;
            img.push_back(p);
        }
        if (!f.eof()) {
            data.push_back(img);
        }
    }
    f.close();
    return data;
}

std::vector<int> readLabels(std::string s) {
    std::ifstream f(s, std::ios::binary | std::ios::in);
    std::vector<int> labels;
    uint8_t label;
    int32_t res;
    f.read((char*)&res, 4);   // magic number
    f.read((char*)&res, 4);   // number of items
    f.read((char*)&label, 1);
    while(!f.eof()) {
        labels.push_back((int)label);
        f.read((char*)&label, 1);
    }
```

```
    f.close();
    return labels;
}
```

F.3 Testing Files

F.3.1 gradient_check_test.cpp

```
#include <gtest/gtest.h>

#include <iostream>
#include <random>
#include <chrono>

#include "../src/convolutional.h"
#include "../src/fullyconnected.h"
#include "../src/pooling.h"
#include "../src/parse_data.h"
#include "../src/layer.h"
#include "../src/net.h"

TEST(GradientTest, FCGradientCheck) {

    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);

    // Generate random input and output labels
    std::vector< std::vector<double> > in;
    std::vector<int> out;

    std::uniform_real_distribution<double> distribution(-1.0, 1.0);
    std::uniform_int_distribution<int> distribution_out(0,
        output_size - 1);
    static unsigned seed = std::chrono::system_clock::now().
        time_since_epoch().count();
    static std::default_random_engine generator(seed);
    int test_size = 10;
    for (int i = 0; i < test_size; i++) {
        std::vector<double> smpl;
        for (int j = 0; j < input_size; j++) {
            smpl.push_back(distribution(generator));
        }
        in.push_back(smpl);
        out.push_back(distribution_out(generator));
```

```
    }

    int grad_tests = 100;
    int num_layers = 3;
    srand(time(0));
    for (int i = 0; i < grad_tests; i++) {
        double sigma = pow(10, -4);
        auto out_ = net(in);
        net.computeLossAndGradients(out);

        net.backpropLoss();

        int l_idx = (rand() % num_layers);
        FullyConnected* fc = (FullyConnected* )net.getLayers()[
            l_idx];
        std::vector<Neuron>& neurons = fc->getNeurons();
        int n_idx = (rand() % neurons.size());
        auto neuron = neurons[n_idx];
        auto weights = neuron.getWeights();
        int w_idx = (rand() % neuron.getWeights().size());
        double grad = neuron.getGradients()[w_idx];
        net.clearSavedData();


        // + sigma loss
        weights[w_idx] += sigma;
        neurons[n_idx].setWeights(weights);
        fc->setNeurons(neurons);
        net(in);
        double loss_plus = net.computeLossAndGradients(out);
        net.clearSavedData();

        // - sigma loss
        weights[w_idx] -= (sigma + sigma);
        neurons[n_idx].setWeights(weights);
        fc->setNeurons(neurons);
        net(in);
        double loss_minus = net.computeLossAndGradients(out);
        net.clearSavedData();

        weights[w_idx] += sigma;
        neurons[n_idx].setWeights(weights);
        fc->setNeurons(neurons);

        double num_grad = (loss_plus - loss_minus) / (2 * sigma);

        double rel = std::max(num_grad > 0 ? num_grad : -num_grad,
            grad > 0 ? grad : -grad);
        rel = rel == 0. ? 1 : rel;
        double diff = (num_grad - grad) / rel;
        diff = (diff > 0) ? diff : -diff;
        ASSERT_LE(diff, 1e-7);
    }
}

```

```
TEST(GradientTest, ConvGradientCheck) {

    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);

    // Generate random input and output labels
    std::vector< std::vector<double> > in;
    std::vector<int> out;

    std::uniform_real_distribution<double> distribution(-1.0, 1.0);
    std::uniform_int_distribution<int> distribution_out(0,
        output_size - 1);
    static unsigned seed = std::chrono::system_clock::now().
        time_since_epoch().count();
    static std::default_random_engine generator(seed);
    int test_size = 10;
    for (int i = 0; i < test_size; i++) {
        std::vector<double> smpl;
        for (int j = 0; j < input_size; j++) {
            smpl.push_back(distribution(generator));
        }
        in.push_back(smpl);
        out.push_back(distribution_out(generator));
    }

    int grad_tests = 100;
    int num_layers = 3;
    srand(time(0));
    for (int i = 0; i < grad_tests; i++) {
        double sigma = pow(10, -4);
        auto out_ = net(in);
        net.computeLossAndGradients(out);
        net.backpropLoss();
        int l_idx = (rand() % num_layers);
        while (l_idx == 1) {     // don't select the pooling layer
            l_idx = (rand() % num_layers);
        }
        Layer* l = (FullyConnected* )net.getLayers()[l_idx];
        std::vector<Neuron>& neurons = l->getNeurons();
        int n_idx = (rand() % neurons.size());
        auto neuron = neurons[n_idx];
```

```
        auto weights = neuron.getWeights();
        int w_idx = (rand() % neuron.getWeights().size());
        double grad = neuron.getGradients()[w_idx];
        net.clearSavedData();

        // + sigma loss
        weights[w_idx] += sigma;
        neurons[n_idx].setWeights(weights);
        l->setNeurons(neurons);
        net(in);
        double loss_plus = net.computeLossAndGradients(out);
        net.clearSavedData();

        // - sigma loss
        weights[w_idx] -= (sigma + sigma);
        neurons[n_idx].setWeights(weights);
        l->setNeurons(neurons);
        net(in);
        double loss_minus = net.computeLossAndGradients(out);
        net.clearSavedData();

        weights[w_idx] += sigma;
        neurons[n_idx].setWeights(weights);
        l->setNeurons(neurons);

        double num_grad = (loss_plus - loss_minus) / (2 * sigma);

        double rel = std::max(num_grad > 0 ? num_grad : -num_grad,
            grad > 0 ? grad : -grad);
        rel = rel == 0. ? 1 : rel;
        double diff = (num_grad - grad) / rel;
        diff = (diff > 0) ? diff : -diff;
        ASSERT_LE(diff, 1e-7);
    }
}
```

F.3.2 conv_test.cpp

```
#include <gtest/gtest.h>

#include "../src/convolutional.h"

TEST(ConvTest, TestForward) {
    ConvLayer conv1(2, 3, 1, 1, 2, 2);

    std::vector<double> input = {   2, 3,
                                    1, 4,

                                    3, 1,
                                    5, 0 };

    std::vector<Neuron> neurons;

    for (double i = 0; i < 8; i++) {
        Neuron n(18);
        std::vector<double> weights = { 5, 1, 2,
                                        3, 3, 2,
                                        4, 1, 1,

                                        2, 3, 5,
                                        0, 1, 2,
                                        4, 2, 1};
        n.setWeights(weights);
        n.setOffset(4);
        neurons.push_back(n);
    }
    conv1.setNeurons(neurons);
    conv1.forward(input);

    std::vector<double> outputs = conv1.getOutput();

    ASSERT_EQ(outputs[0], 36.);
    ASSERT_EQ(outputs[1], 48);
    ASSERT_EQ(outputs[2], 42);
    ASSERT_EQ(outputs[3], 41);
    ASSERT_EQ(outputs[4], 36.);
    ASSERT_EQ(outputs[5], 48);
    ASSERT_EQ(outputs[6], 42);
    ASSERT_EQ(outputs[7], 41);
}
```

F.3.3 fullyconnected_test.cpp

```cpp
#include <gtest/gtest.h>

#include "../src/fullyconnected.h"

TEST(FCTest, TestForward) {
    FullyConnected fc(3, 4);

    std::vector<Neuron> neurons;
    for (double i = 0; i < 4; i++) {
        Neuron n(3);
        std::vector<double> weights = {i, i, i};
        n.setWeights(weights);
        n.setOffset(i + 1.5);
        neurons.push_back(n);
    }
    std::vector<double> input = {1, 2, 3};
    fc.setNeurons(neurons);
    fc.forward(input);

    std::vector<double> outputs = fc.getOutput();

    ASSERT_EQ(outputs[0], 0 + 1.5);
    ASSERT_EQ(outputs[1], 1 + 2 + 3 + 1 + 1.5);
    ASSERT_EQ(outputs[2], 2 * 1 + 2 * 2 + 2 * 3 + 2 + 1.5);
    ASSERT_EQ(outputs[3], 3 * 1 + 3 * 2 + 3 * 3 + 3 + 1.5);
}
```

F.3.4 neuron_test.cpp

```
#include <gtest/gtest.h>

#include "../src/neuron.h"

TEST(NeuronTest, InitWeights) {
    int fan_in = 11;
    Neuron n(fan_in);
    n.initWeights();
    ASSERT_EQ(n.getWeights().size(), fan_in);
}

TEST(NeuronTest, SetWeightsAndGetOutput) {
    int fan_in = 5;
    Neuron n(fan_in);

    std::vector<double> weights;
    std::vector<double> input;
    double offset = 13;
    for (int i = 0; i < fan_in; i++) {
        weights.push_back(i + 1);
        input.push_back(2 * i + 1);
    }

    n.setOffset(offset);
    n.setWeights(weights);

    double result = 1 * 1 + 2 * 3 + 3 * 5 + 4 * 7 + 5 * 9 + 13;

    n.computeNet(input);

    ASSERT_EQ(n.computeActivation(), result);
}
```

Appendix G

PyTorch Model

This appendix contains the code for the PyTorch version of the implemented neural network. This code was written in Python.

G.1 mnist_model.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
import numpy as np
import parse_data
from timeit import default_timer as timer
from sklearn.model_selection import train_test_split
import csv

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.fc0 = nn.Linear(28*28, 2*7*7)
        self.fc1 = nn.Linear(2*7*7, 64)
        self.fc2 = nn.Linear(64, 10)
```

```python

    def forward(self, x):
        in_size = x.size(0)
        x = x.view(in_size, -1)
        x = F.relu(self.fc0(x))
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=0)

use_cuda = torch.cuda.is_available()
#use_cuda = False
device = torch.device("cuda:0" if use_cuda else "cpu")

X_train, Y_train, X_test, Y_test = parse_data.loadData('mnist_train
    .csv', 'mnist_test.csv', device)

def finalTrainAndTest():
    start = timer()
    n_epochs = 100
    net = Net()
    net.cuda()
    lrate = 0.01
    momen = 0.9
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=lrate, momentum=0.0)
    #Test for number of epochs we found with above function
    for i in range(n_epochs):
        running_loss = 0.0

        batch_s = 1
        lb = 0
        ub = batch_s

        if i == 15:
            lrate = 1e-3
        elif i == 30:
            lrate = 1e-4
        elif i == 45:
            lrate = 1e-5

        for g in optimizer.param_groups:
            g['lr'] = lrate

        while ub <= len(X_train):
            optimizer.zero_grad()
            output = net(X_train[lb: ub])
            loss = criterion(output, Y_train[lb: ub])
            loss.backward()
            optimizer.step()
            lb += batch_s
            ub += batch_s
            running_loss += loss.item()

        num_correct = 0
        val_guess = net(X_test)
```

```
        loss = criterion(val_guess, Y_test)

        for j in range(len(Y_test)):
            if torch.argmax(val_guess[j]) == Y_test[j]:
                num_correct += 1

        acc = (num_correct / len(Y_test))

        test_loss = 0.0 + loss.item()

        print("Epoch: " + str(i) + ": " + str(acc))
        print("Training loss: " + str((batch_s * running_loss) /
            len(X_train)))
        print("Test loss: " + str(test_loss))
        end = timer()
        print("Training time: " + str(end - start) + " seconds\n")

finalTrainAndTest()
```

G.2 parse_data.py

```
import torch.utils.data as data_utils
import torch
import numpy as np
import pandas as pd

def loadData(train_filename, test_filename, device):
    train = pd.read_csv(train_filename, skiprows=0).values
    trainX = train[:, 1:].reshape(train.shape[0],1,28, 28).astype('float32')
    X_train = trainX / 255.0

    y_train = train[:,0]

    print(X_train.shape)
    print(y_train.shape)


    test = pd.read_csv(test_filename, skiprows=0).values
    testX = test[:, 1:].reshape(test.shape[0],1,28, 28).astype('float32')
    X_test = testX / 255.0

    y_test = test[:,0]

    X_train_tsr = torch.from_numpy(X_train)
    Y_train_tsr = torch.from_numpy(y_train)
    X_test_tsr = torch.from_numpy(X_test)
    Y_test_tsr = torch.from_numpy(y_test)

    if device == torch.device("cuda:0"):
        X_train_tsr = X_train_tsr.cuda()
        Y_train_tsr = Y_train_tsr.cuda()
        X_test_tsr = X_test_tsr.cuda()
        Y_test_tsr = Y_test_tsr.cuda()

    return X_train_tsr, Y_train_tsr, X_test_tsr, Y_test_tsr
```

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[C+15] François Chollet et al. Keras. `https://keras.io`, 2015.

[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.

[Che16] Xinbo Chen. Energy efficiency analysis and optimization of convolutional neural networks for image recognition. Master's thesis, Texas State University, may 2016.

[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. `https://gmail.googleblog.com/2015/07/the-mail-you-want-not-spam-you-dont.html`.

[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.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[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.

[JZL+18] Wenbin Jiang, Yangsong Zhang, Pai Liu, Geyan Ye, and Hai Jin. *FiLayer: A Novel Fine-Grained Layer-Wise Parallelism Strategy for Deep Neural Networks: 27th International Conference on Artificial*

Neural Networks, Rhodes, Greece, October 4–7, 2018, Proceedings, Part III, pages 321–330. 10 2018.

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

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[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.

[MR17] Pavlo M. Radiuk. Impact of training set batch size on the performance of convolutional neural networks for diverse datasets. *Information Technology and Management Science*, 20, 12 2017.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[QSX+16] Yuran Qiao, Junzhong Shen, Tao Xiao, Qianming Yang, Mei Wen, and Chunyuan Zhang. Fpga-accelerated deep convolutional neural networks for high throughput and energy efficiency: Fpga-accelerated deep convolutional neural networks. *Concurrency and Computation: Practice and Experience*, 05 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.

[Xila] Xilinx. *7 Series DSP48E1 Slice User Guide*. Xilinx.

[Xilb] XilinxWiki. Zynq 2016.4 release. `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842326/Zynq+2016.4+Release`.

[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.