Hardware Accelerator for the Training of Neural Networks

James Erik Groving Meade

Kongens Lyngby 2019

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

Abstract

The goal of the thesis is to ...

Preface

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

The thesis deals with ...

The thesis consists of ...

Lyngby, 27-June-2019

James Erik Groving Meade

Acknowledgements

I would like to thank my....

Contents

Chapter 1

Introduction

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

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

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

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

1.1 Project plan

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

1.2 The "separate document"

It is also important to note that the separate document containing

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

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

Chapter 2

Background

2.1 Neural Networks

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

2.1.1 The Neuron

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

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

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

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

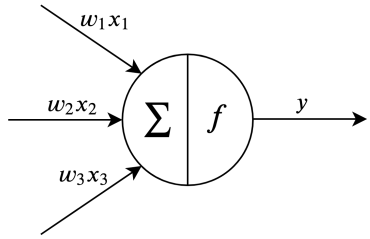

Figure 2.1: A neuron with 3 inputs; bias term omitted for simplicity.

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

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

The most naïve way to initialize weights would to assign each weight a random value between some range. In most cases, this is good enough for the network to converge to a relatively optimal solution so long as the range is not to extreme.

He initialization with proper paper cite

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

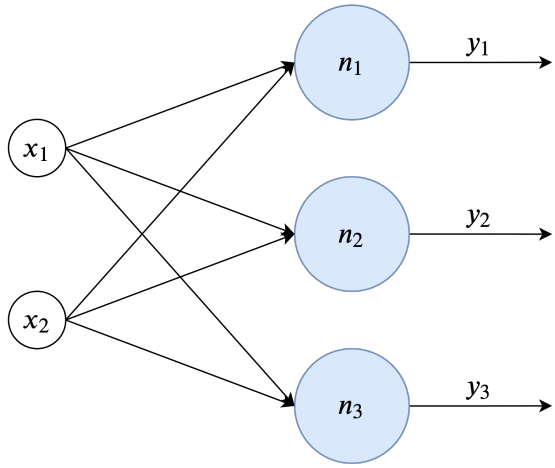

Figure 2.2: A fully-connected layer with 3 neurons, each receiving an input vector of size 2 from the input layer.

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 last layer. Each neuron in the last layer represents a class, so the size of the last layer is equivalent to the number of classes, C.

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.

Backpropagating the Loss Gradient

Calculating Weight Gradients

Updating the Weights

2.1.6 Hyperparameters

Learning Rate

Batch Size

Momentum

Training Epochs

2.2 Deep-Learning Frameworks

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

2.2.1 PyTorch

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

2.3 Related Work

Chapter 3

Software Model

figure for code snippets or nah?

3.1 Overview

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

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

3.2 Motivation

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

3.3 Design

3.3.1 Layers

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

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

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

3.3.2 Training

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

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

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

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

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

Learning Rate and Momentum The software model learns using stochastic gradient descent. As such, the network is configured with a learning rate and momentum. The learning rate may be manually readjusted during training epochs. Momentum is a learning technique in that previous updates to a parameter should impact the update in in a geometrically decreasing fashion. We first define a few parameters:

m	—	the momentum parameter
v	—	'velocity'
lr	—	the learning rate
dx	—	the loss gradient for some weight or bias x.

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

$$v = (m \times v) - (lr \times dx)$$
$$x = x + v$$

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.

We can observe that each time we update x, the previous updates update will have an effect, with the most recent updates having more effect than older ones. A typical value for momentum is 0.9.

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.

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

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.

`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.

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.

3.6 Testing

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

source code in appendix

```
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%.

3.6.1 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 (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

```
1  int      input_size   = 100;
2  int      output_size  = 2;
3  int      batch_size   = 1;
4  double   momentum     = 0.9;
5  double   lr           = 0.001;
6  Net net(input_size, output_size, batch_size, lr, momentum);
7
8
9  Layer* fc1 = new FullyConnected(input_size, 98);
10 Layer* fc2 = new FullyConnected(98, 64);
11 Layer* fc3 = new FullyConnected(64, output_size);
12
13 net.addLayer(fc1);
14 net.addLayer(fc2);
15 net.addLayer(fc3);
```

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

the loss after decrementing w_i by ϵ, and then dividing the difference by 2ϵ. As long as ϵ is rather small, the derivatives should be near exact. In these test cases, $\epsilon = 10^{-4}$. Once we have the analytic and numerical gradient, we can compute the relative error as shown below:

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

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

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

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

The convolutional gradient checking test is set up in the same manner as the

```
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

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

3.6.2 Building and Running the Test Suites

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

```
> make all_tests
```

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

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

```
1  int     input_size    = 8*8;
2  int     output_size   = 2;
3  int     batch_size    = 1;
4  double  momentum      = 0.9;
5  double  lr            = 0.001;
6  Net net(input_size, output_size, batch_size, lr, momentum);
7
8  Layer* conv1 = new ConvLayer(8, 3, 1, 1, 1, 3);
9  Layer* pool1 = new PoolingLayer(8, 4, 3);
10 Layer* conv2 = new ConvLayer(4, 3, 1, 1, 3, 6);
11 Layer* fc1   = new FullyConnected(4*4*6, output_size);
12
13 net.addLayer(conv1);
14 net.addLayer(pool1);
15 net.addLayer(conv2);
16 net.addLayer(fc1);
```

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

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

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

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

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

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

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

Chapter 4

Hardware Model and Implementation

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

Refer to github, appendix, and project link

4.1 Specifications

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

Cite the datasheets

4.2 The Implemented Neural Network

MACs of the networks, distribution of kernels, probably to go in analysis

figure of network.

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.

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

4.5 Layer Architecture

4.5.1 Fully Connected Layers

4.5.2 Softmax Layer

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

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

In this equation,

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

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

4.6 Interlayer Architecture

4.7 ARM-Zynq Communication

4.8 Memory Map Layout

4.9 Project Structure

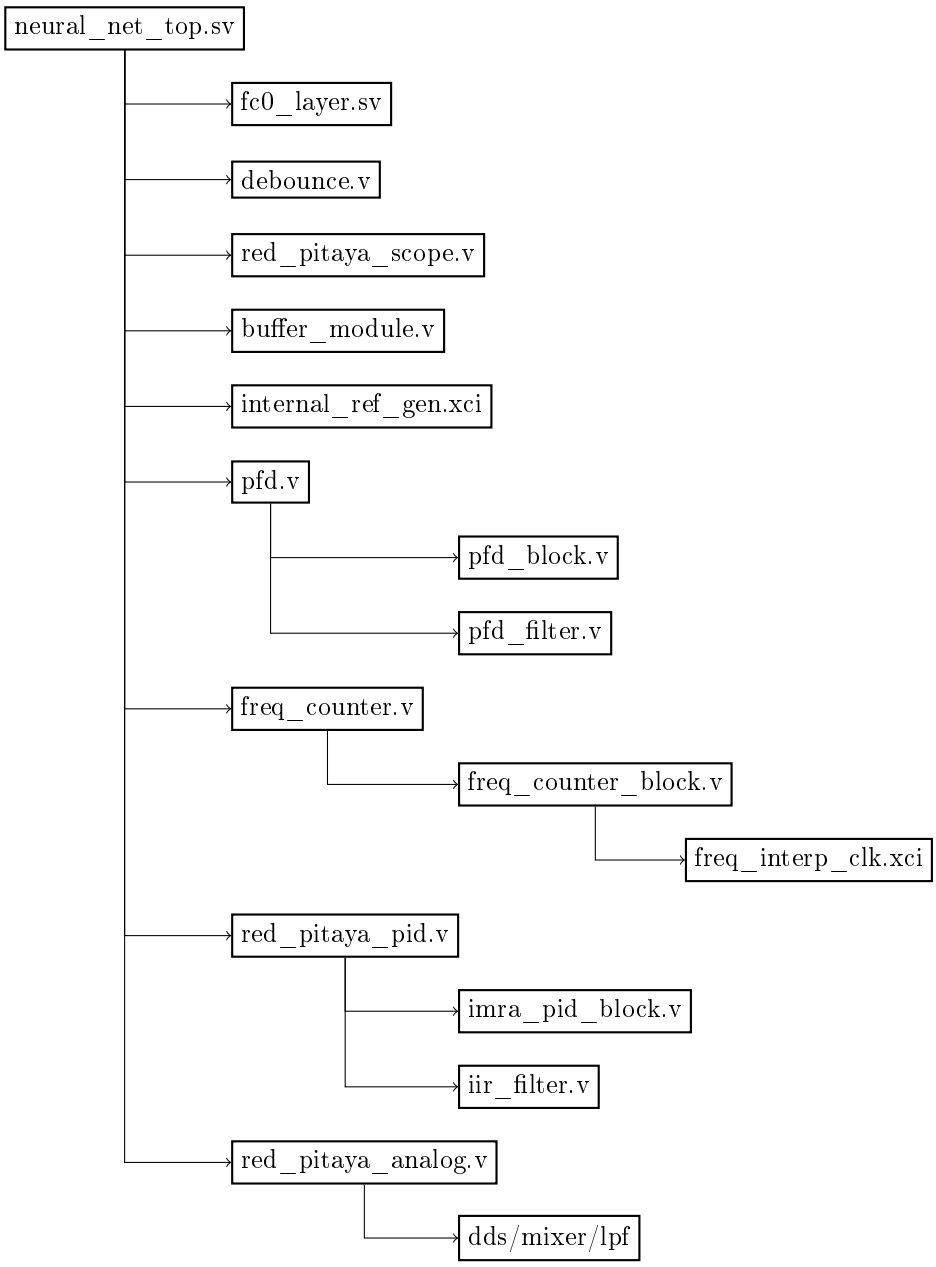

Figure 4.1: Hierarchy of the FPGA code used for the implementation of the network.

Chapter 5

Hardware Model Testing and Verification

5.1 Simulation

5.2 MMIO

Chapter 6

Results

6.1 Performance

6.2 Fw/bw pass

6.3 Power

6.4 Resource Usage

Chapter 7

Analysis

7.1 Basis for Performance

7.2 Cycle Timing

Chapter 8

Future Work

Chapter 9

Discussion

Chapter 10

Conclusion

Appendix A

Stuff

```
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:
```

```
26          str = "1" + str
27      while (len(str) != l):
28          str = "0" + str
29      return str
30
31  params = []
32  binary_params = []
33
34  for i in range(n_neurons):
35      for j in range(params_per_neuron):
36          param = gauss(0, math.sqrt(2/(fan_in - 1)))
37          params.append(param)
38
39  for p in params:
40      b = int(p * (2**17))
41      binary_params.append(intToBinaryString(b, 18))
42  print(params[0])
43  print(binary_params[0])
44
45  contents = "memory_initialization_radix=2;\
        nmemory_initialization_vector=\n"
46  cnt = 0
47  for b in binary_params:
48      contents += str(b)
49      cnt += 1
50      if cnt == r_width:
51          contents += ",\n"
52          cnt = 0
53  contents = contents[:-2] + ";"
54
55
56  f = open('output.coe', 'w')
57  f.write(contents)
58  f.close()
```

Appendix B

Stuff2

Bibliography

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

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

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

[Kar] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition.

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

[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.