Hardware Accelerator for the Training of Neural Networks

James Erik Groving Meade

DTU

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

Not Real

James Erik Groving Meade

Acknowledgements

I would like to thank my....

Contents

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 and Motivation

2.1 A Brief Introduction of Neural Networks

2.2 Related Work

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 the appendix as well as online on github.[1]

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

---

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

open-source deep-learning frameworks that make use of advanced algebraic met-
hods 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 deri-
vative is trivial to compute. Compared to the sigmoid function, ReLU is much
more computationally feasible for an FPGA hardware implementation, and the-
reofre, 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 softmax function is as shown below:

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

In this function, x_i is the output of neuron i in the last layer.

The loss gradients for the neurons in the last layer are computed using multi-class cross entropy loss. Cross entropy loss is computed using probabilities and is defined as:

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

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. Conversely, $p(x_i)$ is equal to the predicted probability resultant from the softmax function. 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:

$$
\begin{array}{lll}
m & \text{---} & \text{the momentum parameter} \\
v & \text{---} & \text{'velocity'} \\
lr & \text{---} & \text{the learning rate} \\
dx & \text{---} & \text{the loss gradient for some weight or bias } x.
\end{array}
$$

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

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

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 `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

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.

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

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.

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 $dim\_i \times dim\_i$ 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 $\frac{2}{fan\_in}$.

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.

```
1  std :: vector < std :: vector < double > > trainX ;
2  std :: vector < int > trainY ;
3  std :: vector < std :: vector < double > > testX ;
4  std :: vector < int > testY ;
5  trainX = readImages ( " data / train - images . idx3 - ubyte " );
6  trainY = readLabels ( " data / train - labels . idx1 - ubyte " );
7  testX = readImages ( " data / t10k - images . idx3 - ubyte " );
8  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.

```
1  int      input_size  = 28*28;
2  int      output_size = 10;
3  int      batch_size  = 200;
4  double   momentum     = 0.9;
5  double   lr           = 0.01;
6  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.

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

```
 6  Layer* fc2 = new FullyConnected(64, 10);
 7
 8  net.addLayer(conv1);
 9  net.addLayer(pool1);
10  net.addLayer(conv2);
11  net.addLayer(pool2);
12  net.addLayer(fc1);
13  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.

```
1  net(in_batch);
2  net.computeLossAndGradients(out_batch);
3  net.backpropLoss();
4  net.update();
5  net.clearSavedData();
```

Add a trained example using the new formatted output.

3.6   Testing

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

source code in appendix

3.6.1   Test Suites

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

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

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

Convolutional Forward Pass   There is a test case to verify the convolutional forward pass located in *conv_test.cpp*. The test creates a convolutional layer that takes a 2×2 feature vector with 2 channels, uses a 3×3 filter for convolution, uses a stride and padding of 1, and produces 2 output channels. Weights and inputs were the arbitrarily assigned and the forward pass was computed and verified against the output that had been previously calculated manually.

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

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

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

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

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}.

```
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 3.1: Layer created for the fully connected gradient check test.

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 3.1.

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 3.2.

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 3.3.

```
 1  Layer: 2, Neuron: 0,   Weight: 31
 2  Analytic Gradient: -0.0638284 Numerical Gradient: -0.0638284
 3
 4  Layer: 0, Neuron: 93, Weight: 71
 5  Analytic Gradient: -0.156235  Numerical Gradient: -0.156235
 6
 7  Layer: 1, Neuron: 34, Weight: 29
 8  Analytic Gradient: -1.22615   Numerical Gradient: -1.22615
 9
10  Layer: 1, Neuron: 12, Weight: 43
11  Analytic Gradient: 0.376021   Numerical Gradient: 0.376021
```

Figure 3.2: Results from the fully connected test using randomly sampled weights to perform gradient checking

```
 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 3.3: Layer created for the convolutional layer gradient check test.

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:

```
1   > 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 3.4

---

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

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

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

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

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

[----------] Global test environment tear-down
[==========] 6 tests from 4 test cases ran. (3329 ms total)
[  PASSED  ] 6 tests.
```
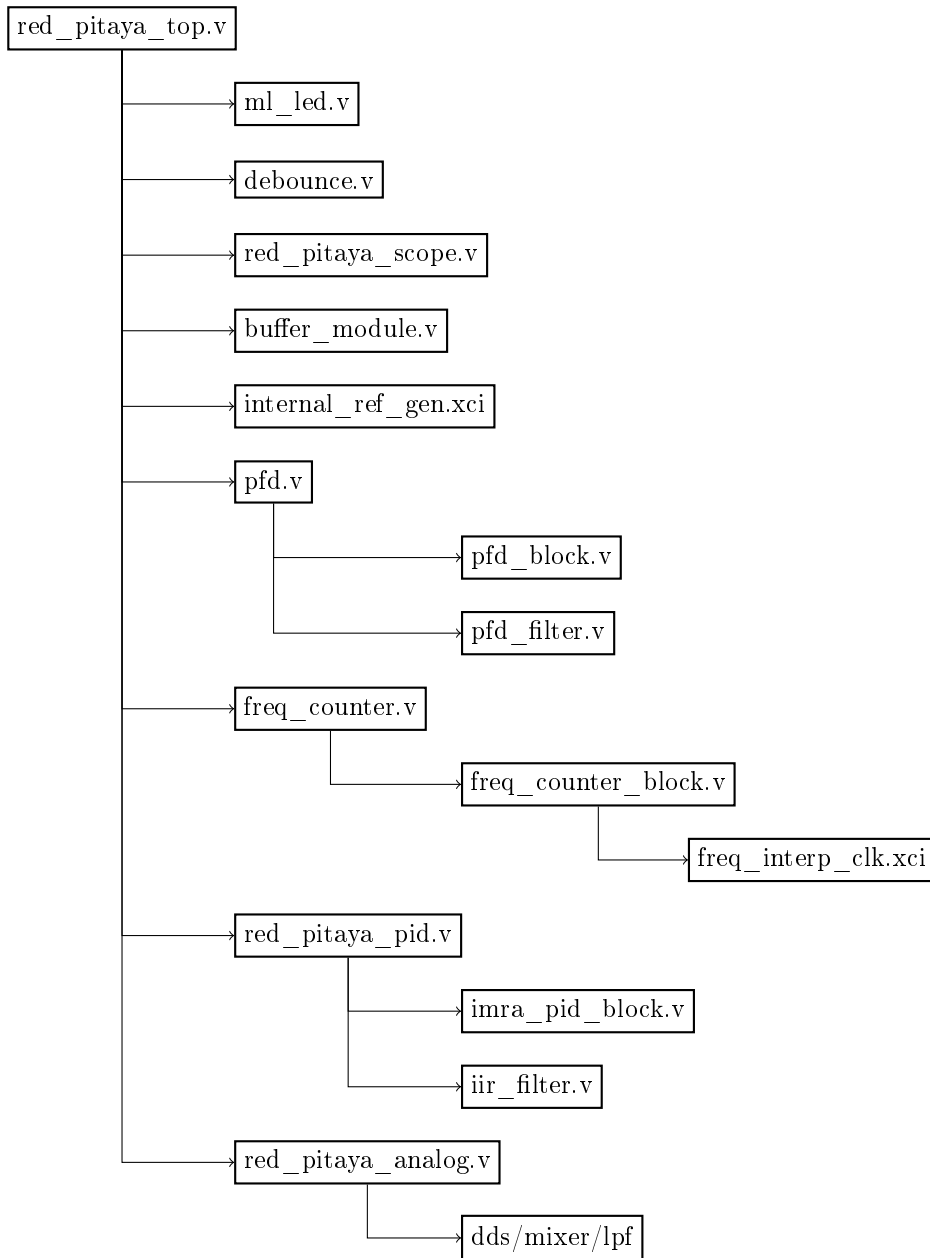
Figure 3.4: Test coverage output using the Google Test C++ testing framework to verify the correctness of the software model for both forward and backward passes.

Hardware Model and Implementation

Figure 4.1: Hierarchy of the FPGA code.

CHAPTER 5

Hardware Model Testing

Results

CHAPTER 7

Analysis

CHAPTER 8

Future Work

CHAPTER 9

Discussion

CHAPTER 10

Conclusion

APPENDIX A

Stuff

This appendix is full of stuff ...

Bibliography

[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.