Neural Network Hardware Implementation for Handwritten Digit Classification

Slayter Teal, A20131271
ECEN 4303
Fall 2021

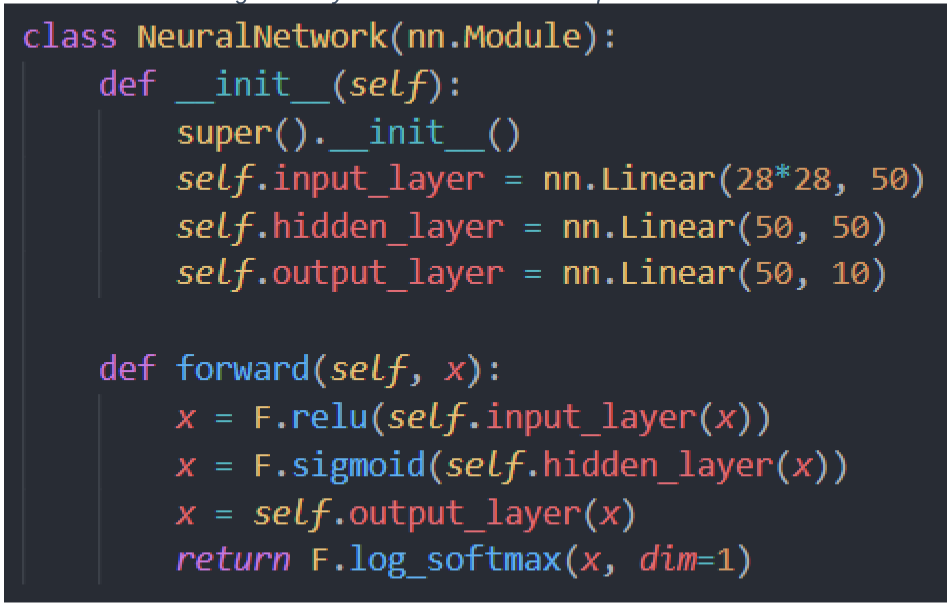
Progress Overview

During the time since the initial report, I have made decent progress with regards to this project. Implementation of the python-based neural network was considerably straight forward. I extracted the weights and biases into separate .weight/.bias files respectively as well as downloading the MNIST dataset. The model training achieved a 97.3% effectiveness rating after running through three epochs. I have elected to use EDA playground for my Verilog implementations making myself familiar with its tools and simulation methods. Regarding Phase 3, I have held off with the Verilog implementation in part due to my other design projects and to allow the in-class lectures to cover adders and multipliers.

Python Implementation

I made thorough use of sentdex's youtube series regarding deep learning and neural networks as he works through building a handwritten digit classification model. Using pytorch and torchvision I implemented a three layer neural network consisting of an input, hidden, and output layer. Per the project specifications the model's size is 784-50-10. For the model I used ReLU for both activation functions of the hidden and output layer. My reasoning being that the ReLU is more hardware friendly (although for the behavioral Verilog I might explore using sigmoid).

Figure 1: Python Neural Network Implementation

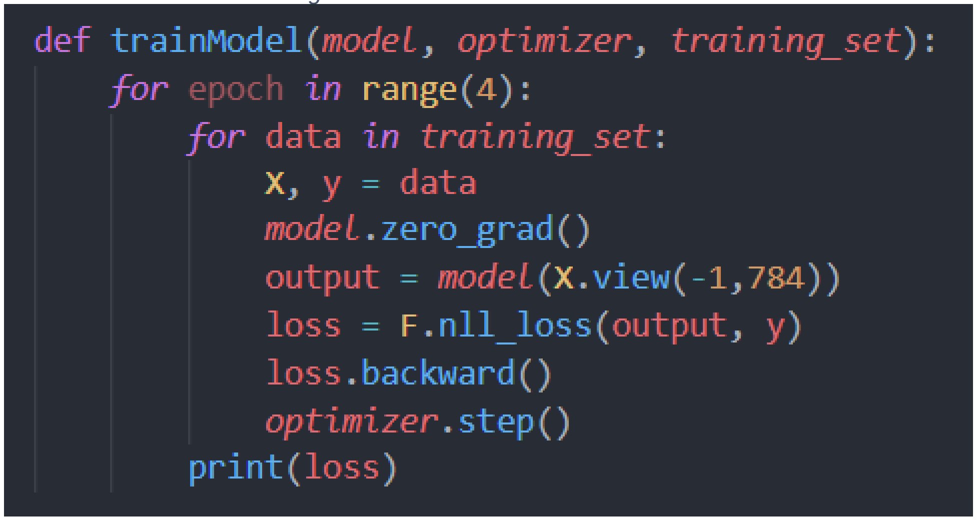


```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_layer = nn.Linear(28*28, 50)
        self.hidden_layer = nn.Linear(50, 50)
        self.output_layer = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(self.input_layer(x))
        x = F.sigmoid(self.hidden_layer(x))
        x = self.output_layer(x)
        return F.log_softmax(x, dim=1)
```

I used a forward approach to training the model, running for approximately four Epochs achieving a 97.3% accuracy. I used an optimizer provided through pytorch to aid in fine tuning the system.

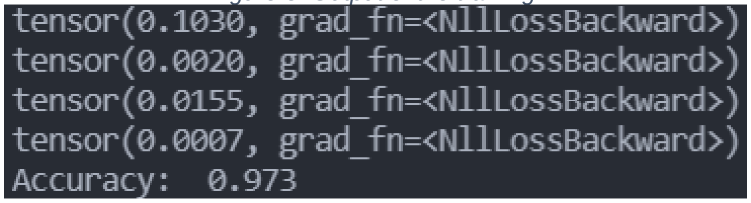
Figure 2: The code to train the model.



```python
def trainModel(model, optimizer, training_set):
    for epoch in range(4):
        for data in training_set:
            X, y = data
            model.zero_grad()
            output = model(X.view(-1,784))
            loss = F.nll_loss(output, y)
            loss.backward()
            optimizer.step()
        print(loss)
```

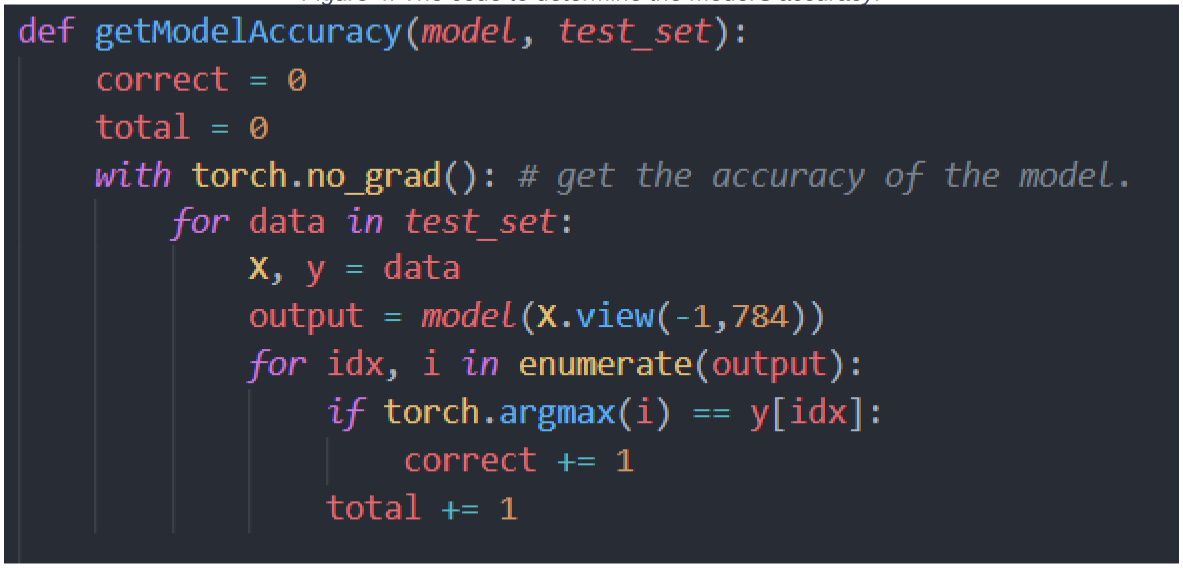
The output of the training as well as the accuracy measurement can be seen below:

Figure 3: Output of the training.



```
tensor(0.1030, grad_fn=<NllLossBackward>)
tensor(0.0020, grad_fn=<NllLossBackward>)
tensor(0.0155, grad_fn=<NllLossBackward>)
tensor(0.0007, grad_fn=<NllLossBackward>)
Accuracy:  0.973
```

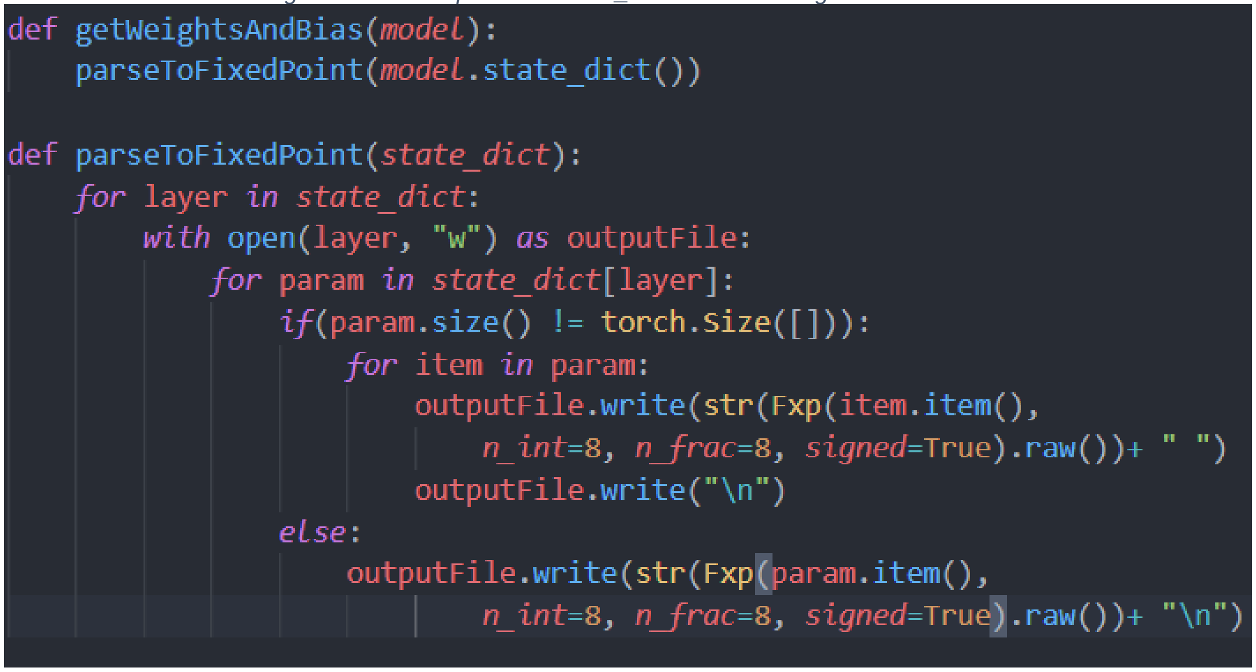
Figure 4: The code to determine the model's accuracy.



```python
def getModelAccuracy(model, test_set):
    correct = 0
    total = 0
    with torch.no_grad(): # get the accuracy of the model.
        for data in test_set:
            X, y = data
            output = model(X.view(-1,784))
            for idx, i in enumerate(output):
                if torch.argmax(i) == y[idx]:
                    correct += 1
                total += 1
```

To extract the weights and biases I made use of the fxpmath library (props to Michael Thompson for the discovery) to extract the state_dict from the model and parse it into a 16 bit fixed-point value.

Figure 5: Code to parse the state_dict and store weights and biases.



```python
def getWeightsAndBias(model):
    parseToFixedPoint(model.state_dict())

def parseToFixedPoint(state_dict):
    for layer in state_dict:
        with open(layer, "w") as outputFile:
            for param in state_dict[layer]:
                if(param.size() != torch.Size([])):
                    for item in param:
                        outputFile.write(str(Fxp(item.item(),
                            n_int=8, n_frac=8, signed=True).raw())+ " ")
                        outputFile.write("\n")
                else:
                    outputFile.write(str(Fxp(param.item(),
                            n_int=8, n_frac=8, signed=True).raw())+ "\n")
```

I would gladly provide the weights and biases from my implementation, however, for sake of not ballooning this document you'll have to take me on my word.

Next Steps and Timeline

Currently, I am dividing my time between this project and my senior design project; this being the case, I'll be shifting focus throughout the coming weeks. Fortunately, my senior design project is several weeks prior to the final due date of this project. With these things in consideration this is my timelines is as follows:

Due Date	Task	Description
11/6/21	Adder and Multiplier implementation	Define and Test an Adder and Multiplier implementation.
11/8-13/21	Slack Period	During this week I'll be shifting focus to my senior design project (hopefully bringing my contributions to completion). If necessary, I'll be preforming research on any nuances regarding Phase 3. This will likely involve meeting with Dr. Li and verifying the progress of Phase 3. I also plan on addressing memory storage of the data and/or weights/bias files.

11/17/21	Begin implementation of Phase 4	At this point I should have a good grasp of Phase 4 (if not already in early stages of development). Basic nodal structures should be built, and unit tested as development continues. By this point I should have a good grasp on how I plan on accessing the dataset/weights and bias files in Verilog.
11/27/21	Complete Phase 4	With Thanksgiving break I will have a surplus of time to focus on this project. By the end of Break I should have completed Phase 4, testing and all.
12/3/21	Dead Week	In the event I ran into a significant problem regarding Phase 4 implementation, this is the week to solve them.
12/7/21	Final Report	The final detailed report of the project contains the results of the Verilog simulation as well as the overall neural network classification hardware implementation.

Works Cited

PythonProgramming.net. “Introduction - Deep Learning and Neural Networks with Python and Pytorch p.1.” *YouTube*, uploaded by sentdex, 23 Sept. 2019, www.youtube.com/watch?v=BzcBsTou0C0.