ECE 385

Fall 2020

Final Project

Transit Support Chatbot in NIOS II and FPGA

Vishnu Jaganathan and Prateek Tenkale

Section AB9

Idea and Overview

For our project, we want to have a functional support chatbot that can read user input and output a response based on the user's intent. In order to make a chatbot, we have to use some sort of Machine Learning model, specifically a Neural Network. We could make a pure software-based chatbot, but implementing our neural network on an FPGA would greatly improve the performance by parallelizing many of our mathematical operations and optimizing the hardware for our functionality.

Our chatbot was designed to be a transit-support chatbot. We take the user's input, classify their intent, and return a pre-defined output response.

We settled on a simpler Machine Learning model: a classification neural network with a character embedding layer and a linear layer (simple matrix math with weights and biases). We accepted a 16 character input and had 2 output nodes that corresponded to the two classes in our dataset.

Our weights, biases, and character-embedding values were all trained with our dataset in a python notebook through PyTorch. Once we obtained the weights it was just a matter of implementing our network on the FPGA by creating the corresponding SystemVerilog modules.

We use the NIOS II with the eclipse console to interface with the user and handle input processing. The NIOS II pre-processes the user's input and feeds it into our Neural network through the Avalon interface.

Originally, we had trained a model that also had character embedding with a linear layer using 64 character input and a large dataset that consisted of 8 different classes. It looked like it was working in our testbench simulation when no SoC was in place. However, when we actually coded this model on the FPGA with the SoC configuration, there were not enough resources on the FPGA.

In our proposal, we considered using a Recurrent Neural Network (RNN) to implement this chatbot. An RNN is ideal when making a chatbot because it models the way our brain handles sequences of data, and it is able to implement dependencies on sequences of words. However, after doing a bit of research on implementation, we quickly realized that it would be way too complex to implement an RNN in an FPGA. There are mathematical functions like tanh which likely would have used too many resources on FPGA if we wanted to implement it without losing precision. Already with our current implementation, the FPGA neared its limit in combinational units. Had we a more robust FPGA, we would have looked into this. All of these concerns led us to believe that implementing an RNN-based chatbot on FPGA would be impossible given our current hardware.

List of Features

Baseline features:

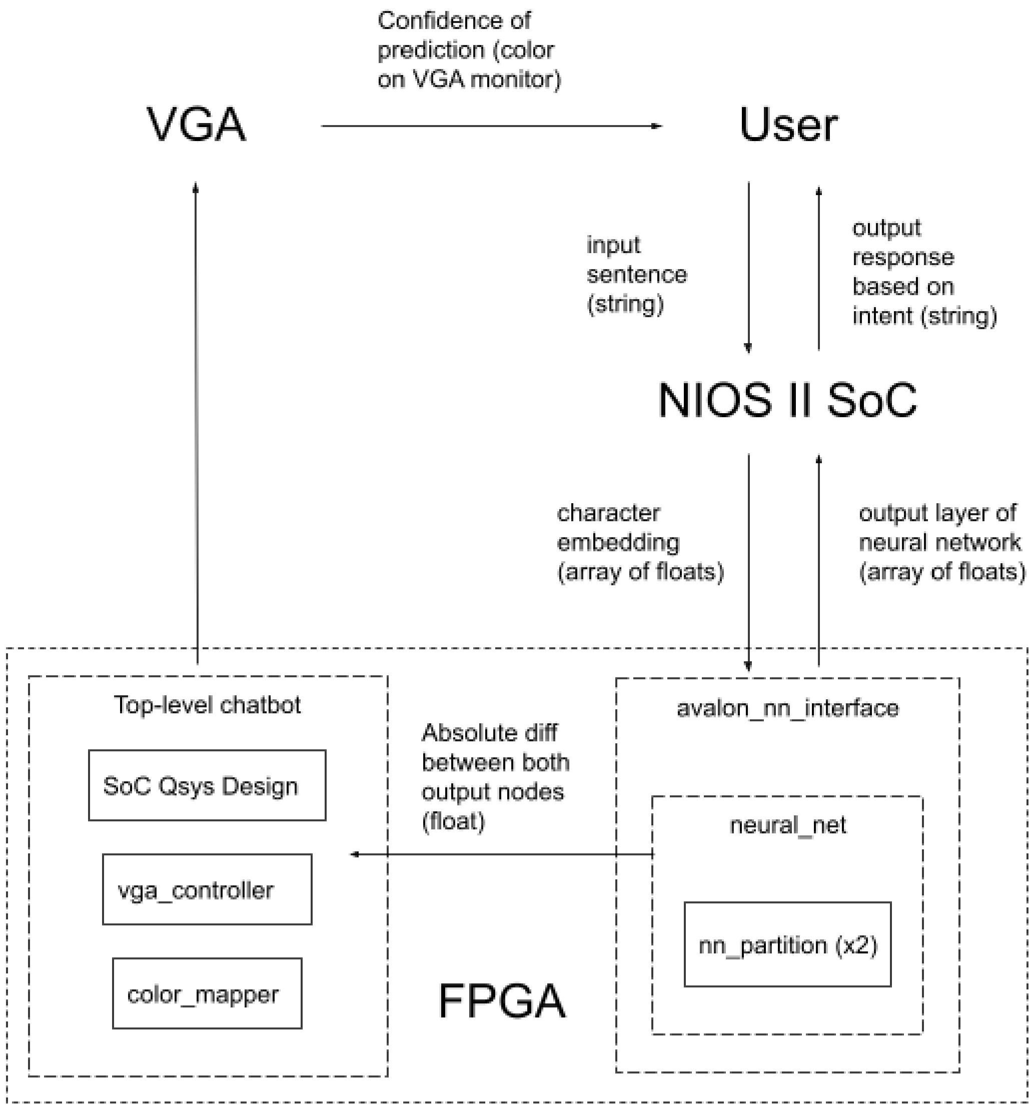
- I/O on SoC configuration through console
- Input preprocessing on SoC. Substitute character ASCII values with corresponding character-embedding floats with a dictionary
- Intent Classification Neural Network on FPGA
- Output processing depending on results of the Neural Network

Additional features:

- Confidence of Prediction with VGA. Red means not confident at all, Yellow indicates , and Green means that the chatbot is very confident

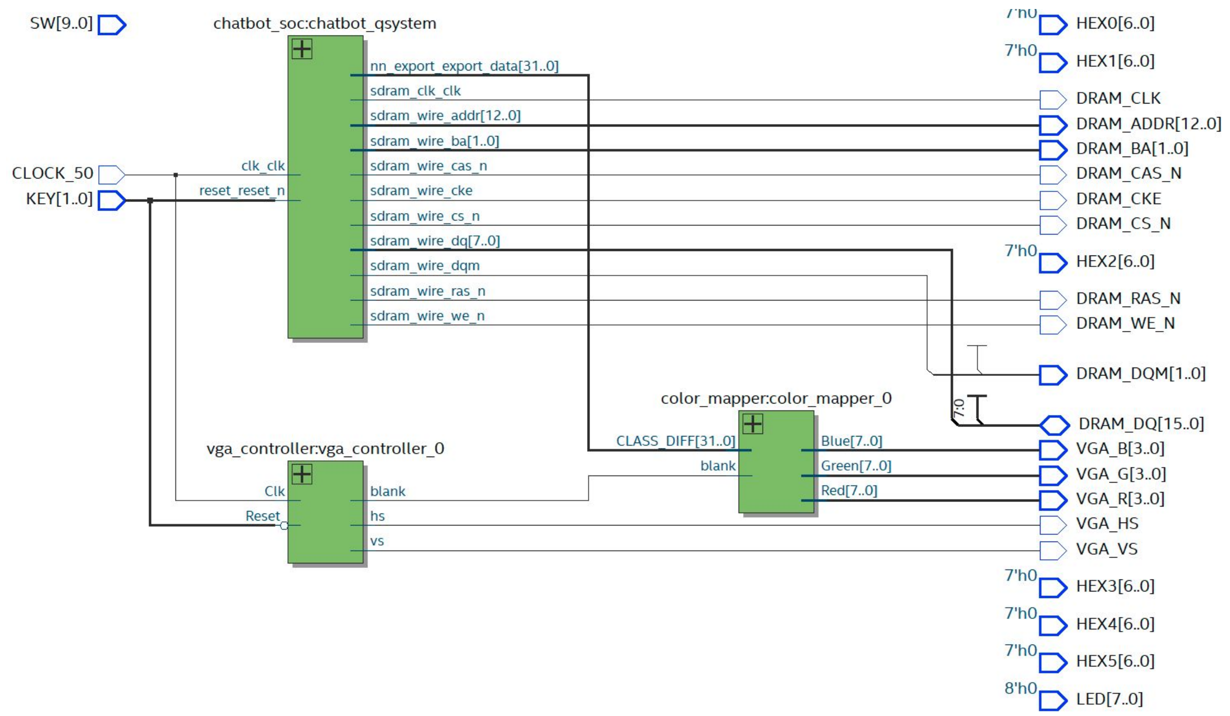
Block Diagrams

High-level Diagram of Dataflow for Chatbot

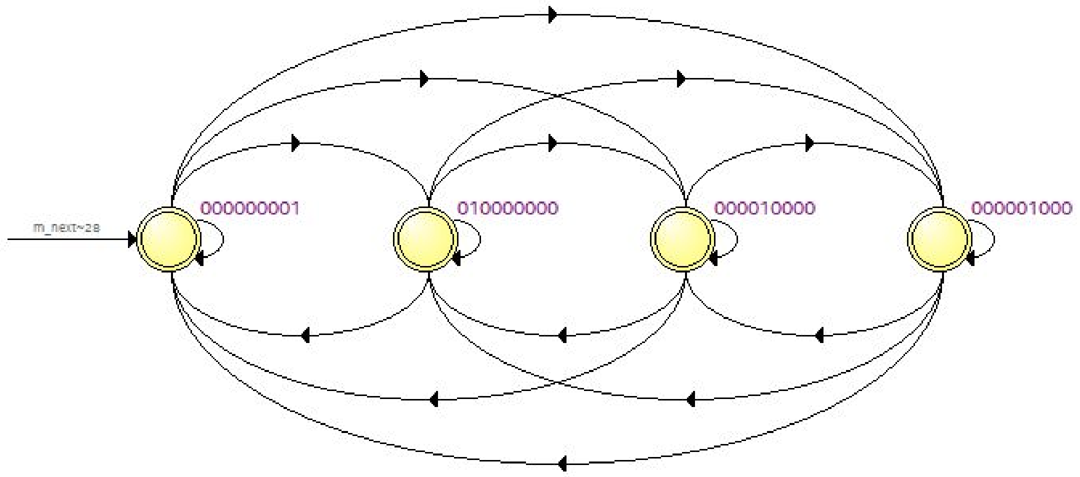


Confidence of prediction (color on VGA monitor)
VGA
User
input sentence (string)
output response based on intent (string)
NIOS II SoC
character embedding (array of floats)
output layer of neural network (array of floats)
Top-level chatbot
SoC Qsys Design
vga_controller
color_mapper
Absolute diff between both output nodes (float)
avalon_nn_interface
neural_net
nn_partition (x2)
FPGA

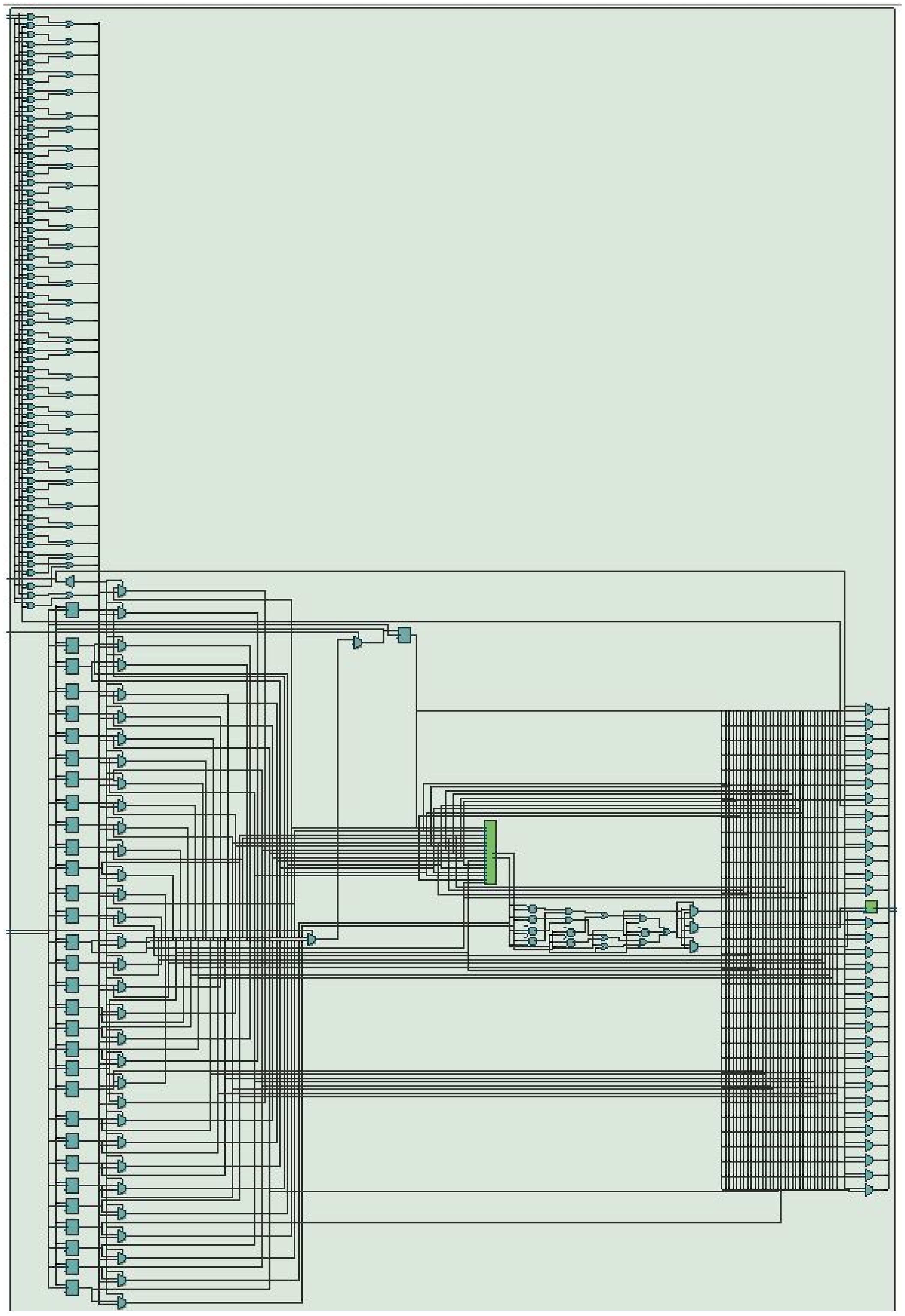
Top level RTL Diagram



State Diagram in Quartus



RTL View of avalon_nn_interface



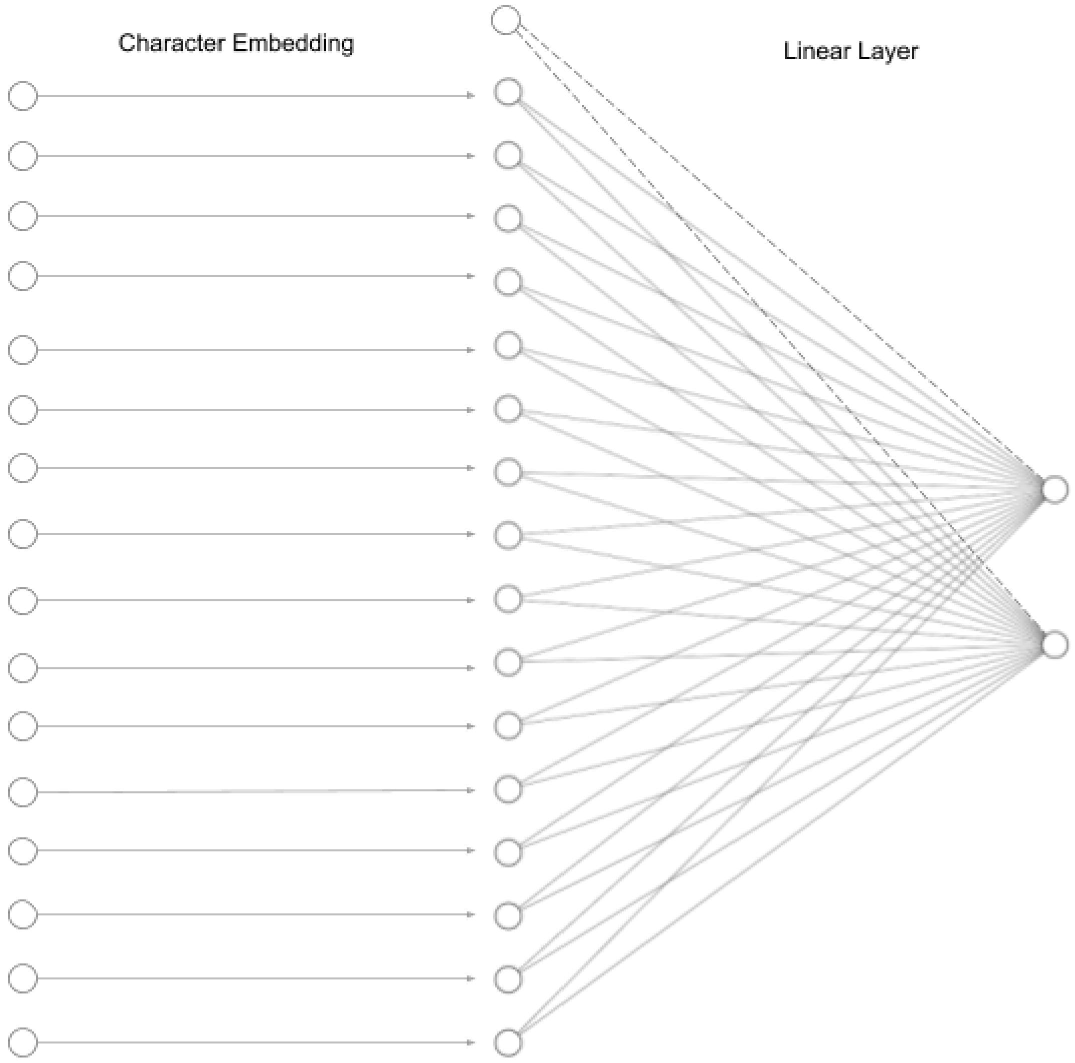
Written Description

Neural Network Architecture

Our neural network took in 16-character input, and had 2 output nodes that corresponded to the two labels in our dataset. We had to convert our 16-character input to their corresponding ASCII values. If our input was less than 16 characters, we would pad it with null values. If it was greater than 16 characters, we would only look at the first 16.

In between the 16-node input layer and 2-node output layer, there was a 1D embedding layer and a linear/fully-connected layer. Each ASCII value from the 16-node input layer maps to a particular character embedding value (vocabulary size of 128), and each of those 16 embedding values were multiplied by a weights matrix and added to a bias vector, creating two values.

Our architecture can be described in the diagram below:



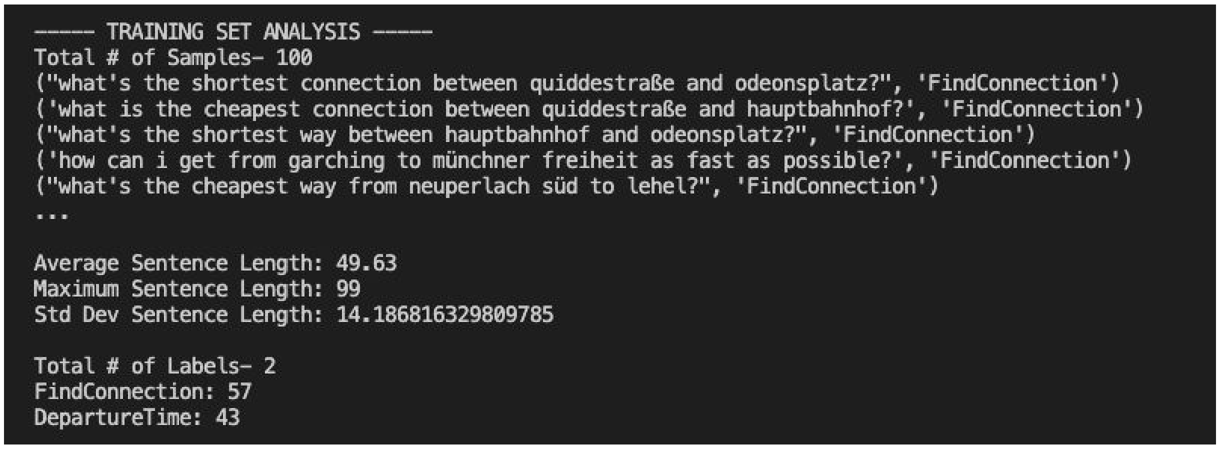
There is a 16-node input layer that has a 1:1 mapping with the 16-node hidden layer, and the corresponding floats in the 16-node hidden layer are multiplied by a 16x2 weights matrix and added to a 2x1 bias vector. There are no activation layers on this neural network. The dotted lines represent the bias units.

We settled on this neural network architecture because we believe that it was easy to understand and be resourceful on the FPGA.

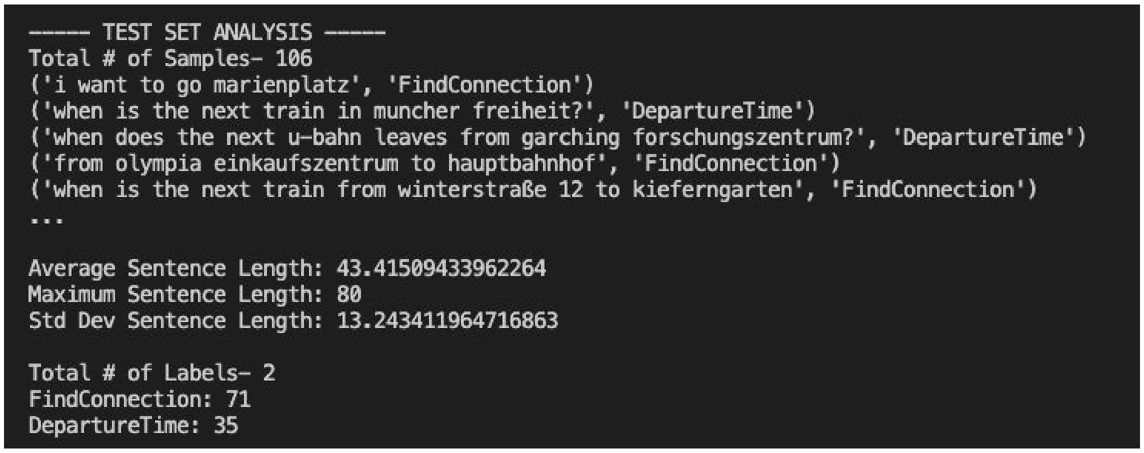
Dataset Analysis

Our data set classified intents for a transit support chatbot in Germany. There were two labels across the entire dataset: “FindConnection”, which means that the user is trying to find a route, and “DepartureTime”, which means that the user is trying to find the next available train/bus on a particular connection.

This was the best dataset that we could find which had only 2 labels. We originally had a transit support chatbot with 8 labels (we really liked this dataset because it had 4000+ samples), but with the VGA and SoC there were not enough resources on the FPGA to implement a 64x8 NN. We tried bringing it down to an 8x8 NN and that still wouldn’t compile. Eventually, we switched to an entirely different dataset which only had 2 output labels



```
----- TRAINING SET ANALYSIS -----
Total # of Samples- 100
("what's the shortest connection between quiddestraße and odeonsplatz?", 'FindConnection')
('what is the cheapest connection between quiddestraße and hauptbahnhof?', 'FindConnection')
("what's the shortest way between hauptbahnhof and odeonsplatz?", 'FindConnection')
('how can i get from garching to münchner freiheit as fast as possible?', 'FindConnection')
("what's the cheapest way from neuperlach süd to lehel?", 'FindConnection')
...

Average Sentence Length: 49.63
Maximum Sentence Length: 99
Std Dev Sentence Length: 14.186816329809785

Total # of Labels- 2
FindConnection: 57
DepartureTime: 43
```



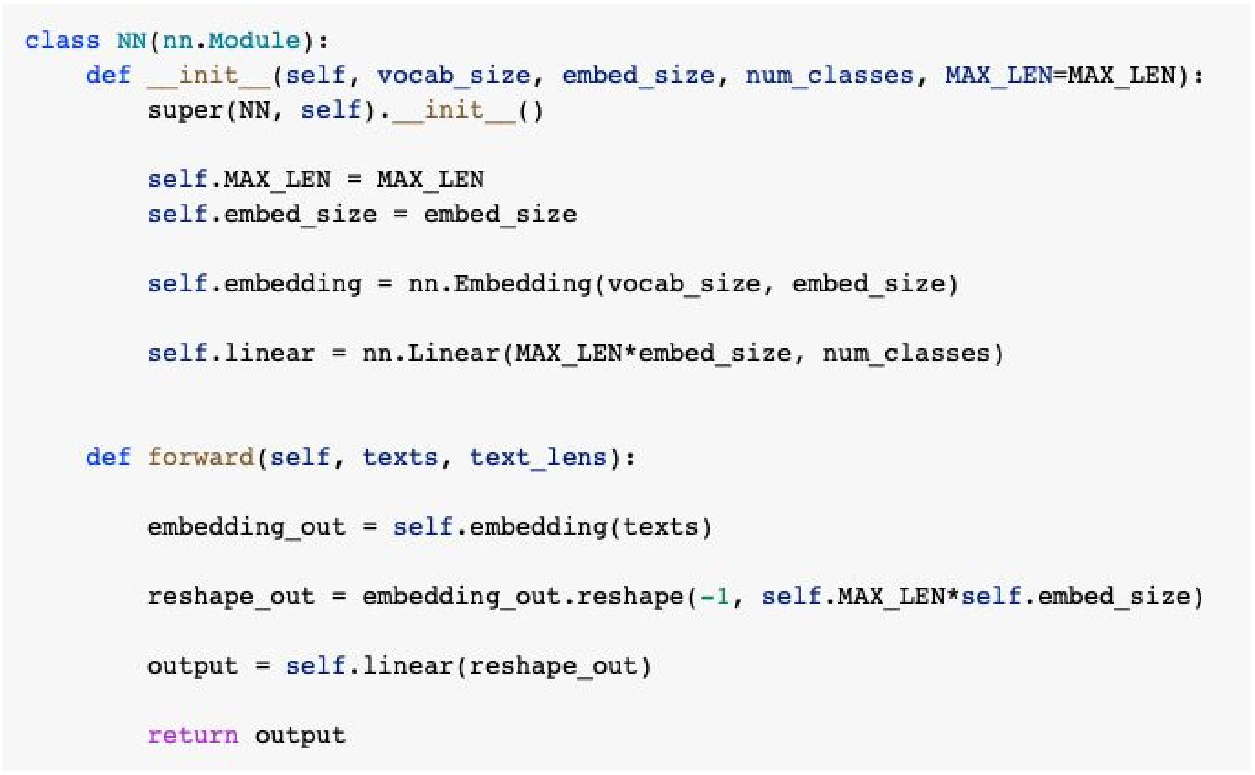
```
----- TEST SET ANALYSIS -----
Total # of Samples- 106
('i want to go marienplatz', 'FindConnection')
('when is the next train in muncher freiheit?', 'DepartureTime')
('when does the next u-bahn leaves from garching forschungszentrum?', 'DepartureTime')
('from olympia einkaufszentrum to hauptbahnhof', 'FindConnection')
('when is the next train from winterstraße 12 to kieferngarten', 'FindConnection')
...

Average Sentence Length: 43.41509433962264
Maximum Sentence Length: 80
Std Dev Sentence Length: 13.243411964716863

Total # of Labels- 2
FindConnection: 71
DepartureTime: 35
```

The average sentence length of entire dataset can be taken through weighted avg: (106*43.415+ 100*49.63)/206 approximately equal to 46.432.

Obviously, making a 16 character input NN is not ideal for this dataset. We tried to make a 64x2 and 32x2 NN with the same overall architecture, but neither of those were able to compile on the FPGA because of limited resources.

Model Training

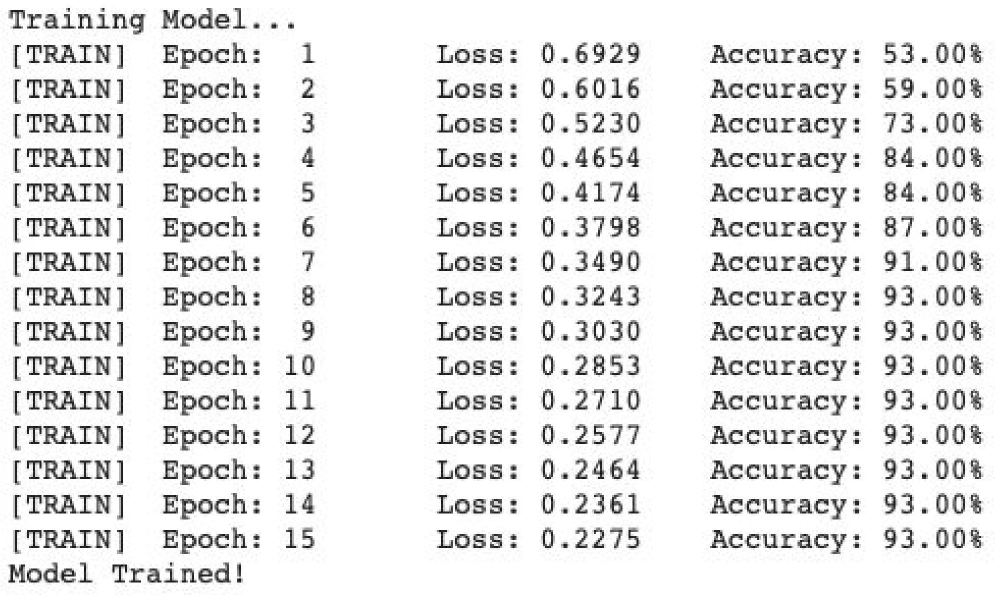
Our model was trained on a Python notebook with PyTorch. We defined our neural network class like this:



```python
class NN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_classes, MAX_LEN=MAX_LEN):
        super(NN, self).__init__()

        self.MAX_LEN = MAX_LEN
        self.embed_size = embed_size

        self.embedding = nn.Embedding(vocab_size, embed_size)

        self.linear = nn.Linear(MAX_LEN*embed_size, num_classes)


    def forward(self, texts, text_lens):

        embedding_out = self.embedding(texts)

        reshape_out = embedding_out.reshape(-1, self.MAX_LEN*self.embed_size)

        output = self.linear(reshape_out)

        return output
```

We also had to create another ‘Dataset’ python class to load the data from our json file and preprocess the text. This ‘Dataset’ class had methods that were designed to work with the PyTorch library functions.

We trained our model with a batch size of 5 and 15 epochs. These were our results when we trained it:

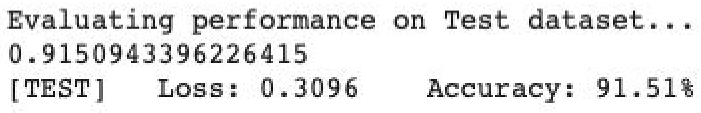


```
Training Model...
[TRAIN]  Epoch:  1       Loss: 0.6929    Accuracy: 53.00%
[TRAIN]  Epoch:  2       Loss: 0.6016    Accuracy: 59.00%
[TRAIN]  Epoch:  3       Loss: 0.5230    Accuracy: 73.00%
[TRAIN]  Epoch:  4       Loss: 0.4654    Accuracy: 84.00%
[TRAIN]  Epoch:  5       Loss: 0.4174    Accuracy: 84.00%
[TRAIN]  Epoch:  6       Loss: 0.3798    Accuracy: 87.00%
[TRAIN]  Epoch:  7       Loss: 0.3490    Accuracy: 91.00%
[TRAIN]  Epoch:  8       Loss: 0.3243    Accuracy: 93.00%
[TRAIN]  Epoch:  9       Loss: 0.3030    Accuracy: 93.00%
[TRAIN]  Epoch: 10       Loss: 0.2853    Accuracy: 93.00%
[TRAIN]  Epoch: 11       Loss: 0.2710    Accuracy: 93.00%
[TRAIN]  Epoch: 12       Loss: 0.2577    Accuracy: 93.00%
[TRAIN]  Epoch: 13       Loss: 0.2464    Accuracy: 93.00%
[TRAIN]  Epoch: 14       Loss: 0.2361    Accuracy: 93.00%
[TRAIN]  Epoch: 15       Loss: 0.2275    Accuracy: 93.00%
Model Trained!
```

Once we had the trained embedding dictionary and weights, we exported them to a text file so that we could use another python script to read the text files and generate the corresponding SystemVerilog code.

Model Testing & Evaluation

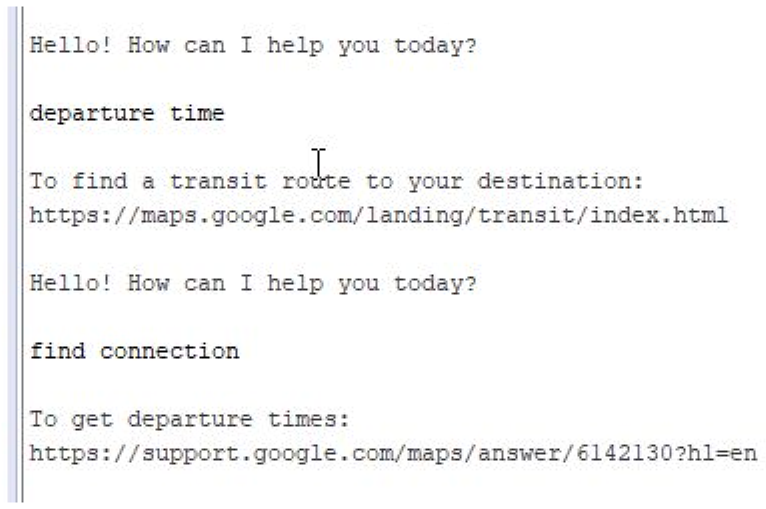
When we ran our model in software on the test dataset, these were our results:



```
Evaluating performance on Test dataset...
0.9150943396226415
[TEST]   Loss: 0.3096    Accuracy: 91.51%
```

We were actually really surprised by a 91.5% accuracy, considering that our test set had more samples than our training set. When we tried re-training it, our accuracy would be anywhere between 65-85%. Part of the reason for these varying accurracies is because we only have 100 training samples in our dataset, so the accuracy overall will depend a little bit on luck.

While our chatbot works very well for input sentences that are similar to the dataset, we noticed that it is not nearly as accurate for generic inputs. For example, we tried to see what intent was classified for “departure time” and “find connection”, and our neural network was totally off:


```
Hello! How can I help you today?

departure time

To find a transit route to your destination:
https://maps.google.com/landing/transit/index.html

Hello! How can I help you today?

find connection

To get departure times:
https://support.google.com/maps/answer/6142130?hl=en
```

Our neural network definitely would have been more accurate if we had a more extensive and used word-based embedding (which is how the embedding layer is typically used) instead of character embedding.

FPGA Neural Network Implementation

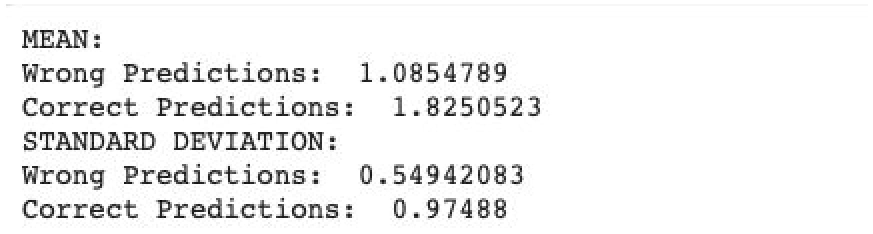
In order to implement our Neural Network on the FPGA, we decided it would be best to use the NIOS II SoC to interface with the user. The NIOS II handles text preprocessing and maps each character to its corresponding embedding value. The linear layer of our neural network was implemented in SystemVerilog. We used the avalon interface similar to lab 9 to feed the embedding values into the pure hardware-based neural network. Implementation details can be found in 'avalon_nn_interface.sv.'

One tricky part was dealing with floats, which we hadn't done during the semester. Our embedding values, weights, and biases were all floating point values that needed to be represented in SystemVerilog. We also had to perform add, multiply operations on the float in Verilog with separate modules, since quartus could not synthesize the real type. We used the IEEE754 standard to represent a 32 bit float with 1 sign bit, 8 bits of exponent, and 23 bits of mantissa. We had to implement the add and multiply operations between these representations by ourselves, which is described in the modules section.

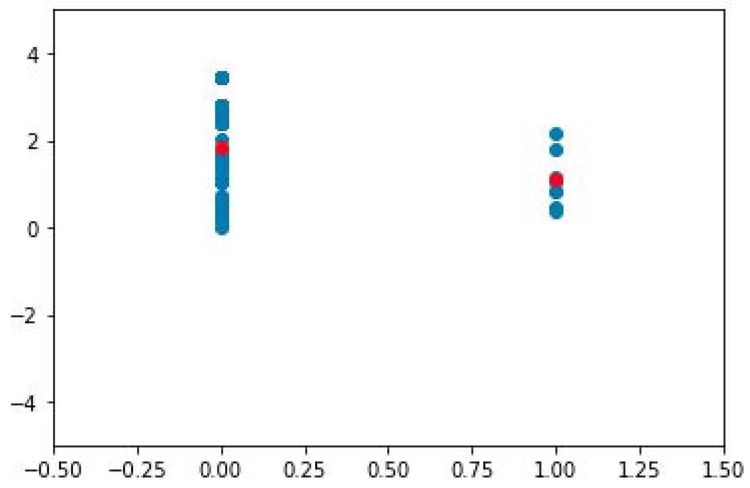
Confidence of Prediction

The 'Confidence of Prediction' for our chatbot was based on the 2 output node values. When a neural network is making a classification, it takes the largest value of all of its output nodes to determine the label. We realized that if an output node was particularly high or low, then the neural network was trained to give high/low values in order to make the correct classification. Therefore, the absolute difference between the two output nodes would likely be a good representation of how confident the neural network is in its prediction. If 1 output node was significantly larger than another one, then the neural network has been trained to make that prediction. Likewise, if two output nodes are relatively similar, then the neural network is probably unsure about which one of those could be the correct class.

Based on this idea, we wanted to see if there was any correlation between correctness of prediction and the difference in two output nodes. These were our results:



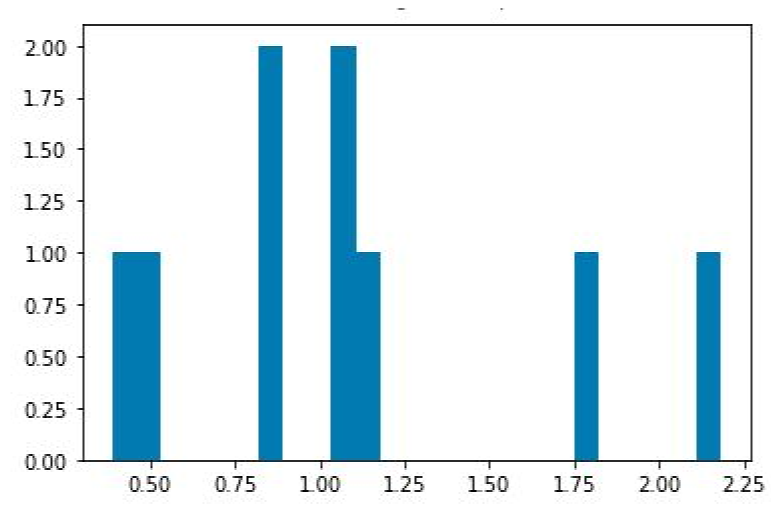
```
MEAN:
Wrong Predictions:  1.0854789
Correct Predictions:  1.8250523
STANDARD DEVIATION:
Wrong Predictions:  0.54942083
Correct Predictions:  0.97488
```



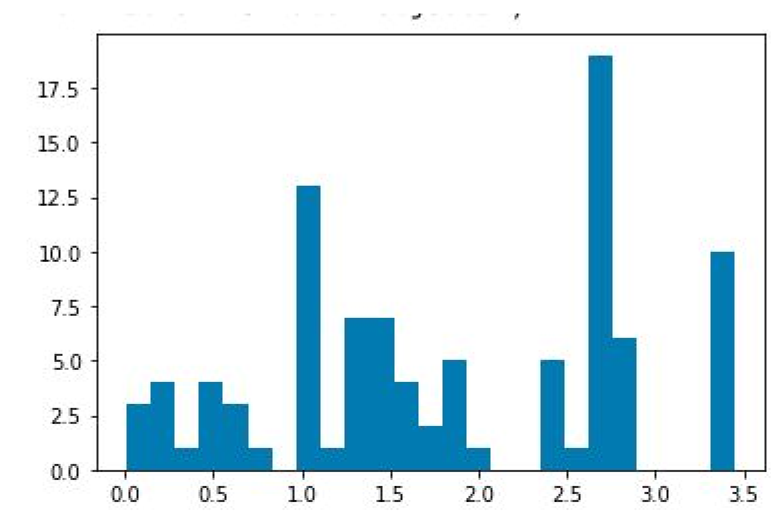
On the x-axis, 0 corresponds to all the correct predictions (97 of them), while 1 corresponds to the incorrect predictions (9). We expected to see much lower differences on the incorrect predictions, but based on this scatter plot, there was no significant correlation between correctness of prediction and output node difference. This could be due to the fact that we only had 9 test samples that were incorrect.

To get a better understanding, we wanted to see what the distribution of output node differences was like on our data. We looked at two histograms: one for correct predictions, the other for incorrect predictions.

Incorrect predictions:



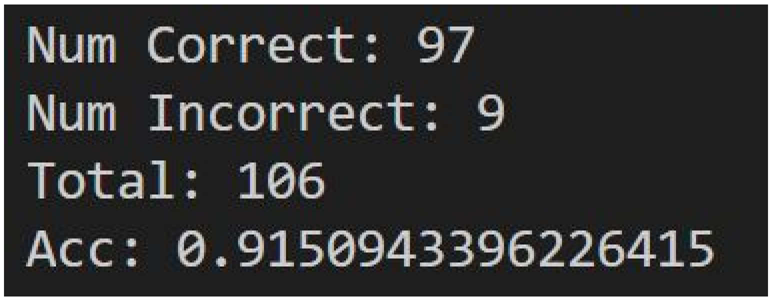
Correct predictions:



Based on this data, we decided that a difference > 1.25 would indicate ‘very confident,’ a difference between 0.25 - 1.25 would indicate ‘unsure’, and a difference < 0.25 would indicate ‘not confident at all.’ These values correspond to green, yellow, and red on our VGA monitor

FPGA vs. Python

We wanted to compare the accuracy of our FPGA-based chatbot to the Python-based chatbot on the dataset. We did this by manually feeding in all 106 test samples and recording the intent classification for each one in a text file. We had another text file which had the actual intents of those test samples, and when we compared the results we found no drop-off in accuracy on the FPGA-based chatbot



```
Num Correct: 97
Num Incorrect: 9
Total: 106
Acc: 0.9150943396226415
```

The FPGA-based made the exact same inferences as the Python-based chatbot despite a slight loss of precision.

For a small network, performance on the FPGA was around 52x faster than the python code. This number may also have factors of additional overhead of data loading in python and the NIOS delay in the FPGA baked into it. Another advantage of the FPGA is the lower power usage of the neural network system. In general CPUs and GPUs consume a large amount of power when dealing with matrix multiplies. An FPGA with dedicated combinational logic is able to perform these more efficiently.

Module Descriptions

Module: chatbot.sv
Inputs: CLOCK_50, [1:0] KEY, [9:0] SW
Outputs:VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B, [7:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [1:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
InOut: [15:0] DRAM_DQ
Description: This module is the top-level entity of our Quartus project. It contains the chatbot soc and the VGA signals to display the color indicating confidence of prediction. All logic for making inferences is held within this SOC module and it is also used to interface with the NIOS.
Purpose: Wrapper to fit the pin assignments in Quartus, and send correct VGA signals to monitor. This top level module does not contain any logic and just maps ports to where they should go to interface with pins / VGA.

Module: chatbot_soc.sv
Inputs: clk_clk, reset_reset_n
Outputs: [31:0] nn_export_export_data, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sram_wire_cke, sdram_wire_cs_n, [7:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n
InOut: [15:0] sdram_wire_dq
Description: This module is the SOC configuration from platform designer that interfaces the C code to the hardware. All of the inputs and outputs are the same as usually found on this

platform designer chip except the nn_export_export_data, which sends the absolute difference between the 2 output nodes for color_mapper to determine confidence of prediction
Purpose: The purpose of this module is to take the clock and reset from fpga pins and give back the nn_export_export_data, which contains the absolute difference between the 2 output nodes. This also houses the avalon_nn_interface.sv that takes care of making an inference based on input. Overall this module's generated code is what is responsible for the function of connecting together the majority of the program.

Module: avalon_nn_interface.sv
Inputs: CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, [3:0] AVL_BYTE_EN, [3:0] AVL_ADDR, [31:0] AVL_WRITEDATA
Outputs: [31:0] AVL_READDATA, [31:0] EXPORT_DATA
Description: This module is the 32 word, 32 bit per word data register. It uses combinational logic to read data and clock driven logic to write it. The outputs from the neural_net hardware component are written to the register. EXPORT_DATA is modified through combinational logic
Purpose: The purpose of this module is to interface between the Neural Network and the C code. This module takes in read / write commands from the C code in the form of array operations, and lets the neural_net module read register values and write to the register synchronously with its clock.

Module: neural_net.sv
Inputs: [31:0] X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14, X_15, X_16
Outputs: [31:0] O_1, O_2
Description: This module defines the weights and biases that are used in the linear layer of our neural network, and outputs the result, all with combinational logic. Contains 'nn_partition' modules that do the actual multiplication/addition operations.
Purpose: Take the 16 floats as input, define weights and biases, and output the resulting float for each node. Wrapper for nn_partition

Module: nn_partition.sv
Inputs: [31:0] X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_10, X_11, X_12, X_13, X_14, X_15, X_16,
W_1, W_2, W_3, W_4, W_5, W_6, W_7, W_8, W_9, W_10, W_11, W_12, W_13, W_14, W_15, W_16,
bias
Outputs: [31:0] out
Description: This module multiplies each input value by its corresponding weight, adds the bias, and outputs the results. The dot product sum is calculated through a hierarchical add to parallelize the addition. All multiplications are parallelized. Uses Float_Mul and Float_Add
Purpose: Perform dot product between weights vector and input vector and add bias.

Module: Float_Add.sv
Inputs: [31:0] a, b

Outputs: [31:0] c
Description: Consists of combinational logic that adds the two floats. First the mantissas are shifted so that the smaller exponent matches up to the larger one. Then addition / subtraction is performed to get the resulting mantissa. The sign bit and exponents bits are the sign and exponent bits of the larger float. The mantissa is then normalized through an if else block that finds the most significant 1 bit and shifts it to the high bit position, adjusting the exponent accordingly.
Purpose: Perform floating point addition of two IEEE 754 floats. These are arranged in a hierarchical manner in the nn_partition module to perform the sum of the dot product.

Module: Float_Mul.sv
Inputs: [31:0] a, b
Outputs: [31:0] c
Description: Consists of combinational logic to multiply two floats. First the sign bit is calculated as the XOR of the two input sign bits. Then the mantissas are multiplied, and the exponent bits are added together. Then the mantissa is normalized and the exponent is adjusted accordingly.
Purpose: Perform floating point multiplication of two IEEE 754 floats. Each element of the input X vector is multiplied with each element of the input W vector.

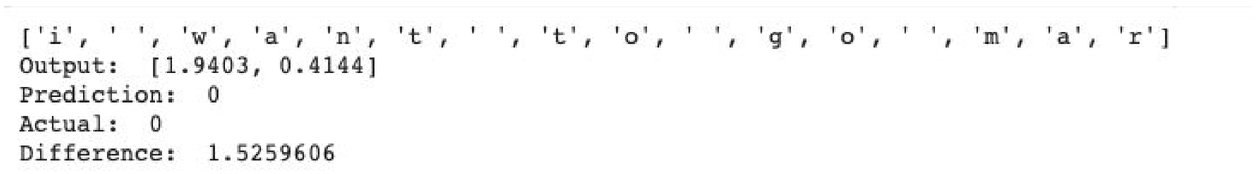
Module: VGA_controller.sv
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync
Description: This module iterates through every pixel in row-major order (go right ->left, then up -> down). It specifies that only 640x480 pixels will get displayed on the VGA monitor, and it produces a vertical sync pulse and horizontal sync pulse when it has finished iterating through this range of pixels. It updates at ½ the input clock (25MHz instead of 50 MHz) to stick with a 60 Hz refresh rate.
Purpose: Specify when the frame is finished being written through ‘blank’

Module: color_mapper.sv
Inputs: [31:0] CLASS_DIFF, blank
Outputs: [7:0] Red, [7:0] Green, [7:0] Blue
Description: This is essentially a function that returns the RGB value that should be written to the monitor depending on the difference between our two output nodes (CLASS_DIFF). If CLASS_DIFF > 1.25, return RGB for ‘green’, if CLASS_DIFF <= 1.25 && CLASS_DIFF > 0.25, return RGB for ‘yellow.’ Otherwise, return RGB for ‘red.’
Purpose: Define RGB value to be written to VGA monitor

Simulation

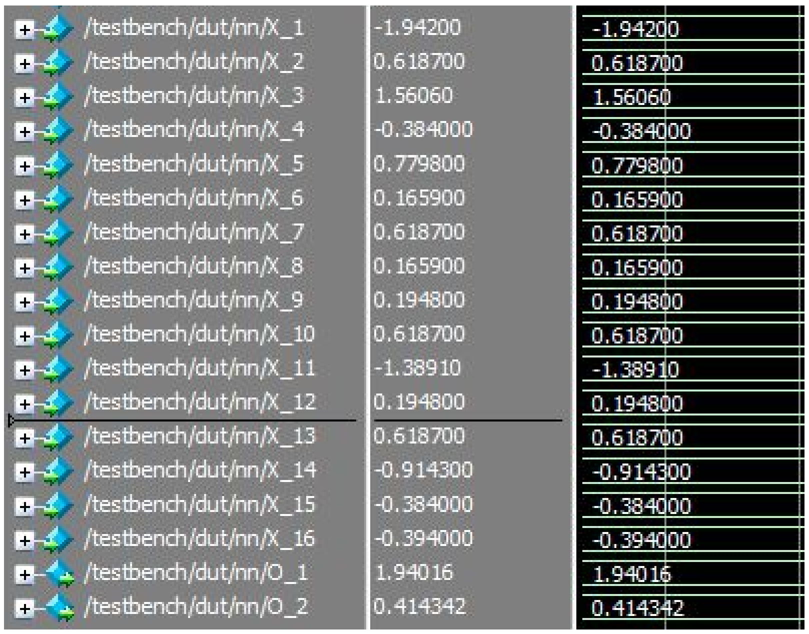
Input string: "i want to go marienplatz"

Result in software-based implementation:



```
['i', ' ', 'w', 'a', 'n', 't', ' ', 't', 'o', ' ', 'g', 'o', ' ', 'm', 'a', 'r']
Output:  [1.9403, 0.4144]
Prediction:  0
Actual:  0
Difference:  1.5259606
```

Result in Hardware simulation:



```
/testbench/dut/nn/X_1    -1.94200    -1.94200
/testbench/dut/nn/X_2    0.618700    0.618700
/testbench/dut/nn/X_3    1.56060     1.56060
/testbench/dut/nn/X_4    -0.384000   -0.384000
/testbench/dut/nn/X_5    0.779800    0.779800
/testbench/dut/nn/X_6    0.165900    0.165900
/testbench/dut/nn/X_7    0.618700    0.618700
/testbench/dut/nn/X_8    0.165900    0.165900
/testbench/dut/nn/X_9    0.194800    0.194800
/testbench/dut/nn/X_10   0.618700    0.618700
/testbench/dut/nn/X_11   -1.38910    -1.38910
/testbench/dut/nn/X_12   0.194800    0.194800
/testbench/dut/nn/X_13   0.618700    0.618700
/testbench/dut/nn/X_14   -0.914300   -0.914300
/testbench/dut/nn/X_15   -0.384000   -0.384000
/testbench/dut/nn/X_16   -0.394000   -0.394000
/testbench/dut/nn/O_1    1.94016     1.94016
/testbench/dut/nn/O_2    0.414342    0.414342
```

O_1 and O_2 correspond to the two output nodes

Design Resources and Statistics

LUT	26047
DSP	592
Memory (BRAM)	55296 bits
Flip-Flop	2977
Frequency	137.51 MHz
Static Power	96.18 mW
Dynamic Power	0.68 mW
Total Power	106.07 mW

Conclusion

We really enjoyed working on this final project, and we really liked how there was a lot of room to experiment. We thought it was a great way to use our backgrounds in Computer Science and apply what we learned in this class to Machine Learning. FPGA Accelerated Machine Learning Inference has huge real-world applications, and this project was a great introduction into that field.

Our biggest setbacks with this project was implementing our Python-based chatbot on the FPGA. Since we were using the NIOS II SoC to interface with the user, we did not have enough resources to implement a bigger neural network that could accept larger inputs and covered a larger range of intents. While the neural network that we implemented was accurate for the test dataset, that accuracy did not carry over to generic inputs. If we had a powerful FPGA, we would be able to make larger, more advanced neural networks that were very accurate.

We also could have improved accuracy by implementing word-based embedding instead of character-based embedding. We chose character-based embedding for our Neural Network because the embedding dictionary would be easy to implement in C. If we had word-based embedding, we would need to implement a hash table that mapped strings to floats to make it work in C.

Overall, this was a great project that gave us insight on how hardware acceleration works. Despite the limited scale of our resources, we prototyped a model that paralleled its software counterpart's accuracy and produced meaningful classification results. In our future studies, we hope to learn more about how more complicated neural models are implemented on FPGAs and try these ideas with more powerful devices.