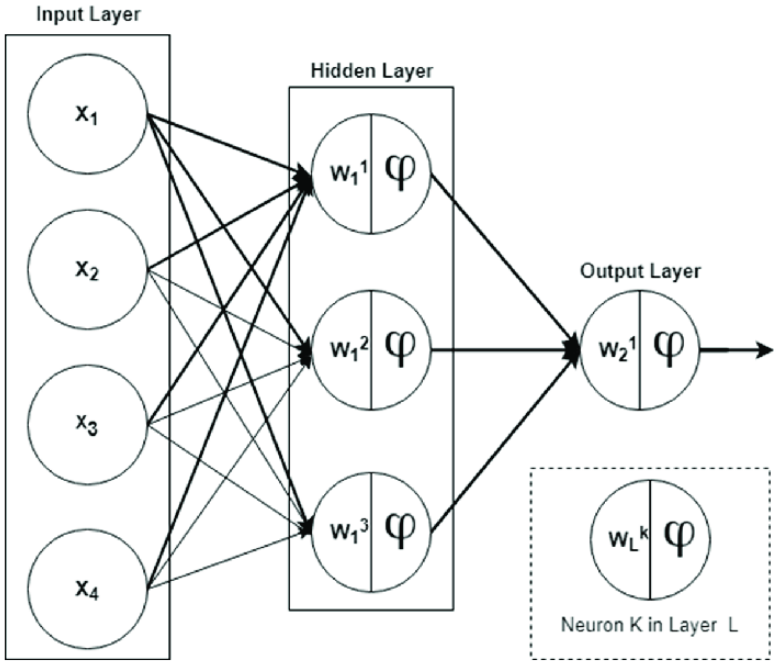
EE217 Project Report

Matthew Choi
SID: 862034854
Keerthi Korisal
SID: 862034133

Overview of project:

The project plan was to build a Feed Forward Neural Network(FFNN) trained using backpropagation. Taking a stock price dataset of NVIDIA Corporation(NVDA) from Yahoo! finance, we used monthly stock data to predict various characteristics of the stock. The inputs used were weekly open, close, high and low stock data. The FFNN and backpropagation algorithm is made from scratch on the GPU. We tried testing different outputs but weren't able to accurately predict the values we expected. We tried testing price increase/decrease, percent difference, and next month price prediction, but were not satisfied with the results we got.



Example FFNN implement

Implementation:

The picture above shows an example of a FFNN, although we use a slightly different layout. Our FFNN was built using 3 layers: Input, hidden, and output layer. We take in four inputs, the high, low, open, and close price. We received one output and we tested a couple of different outputs to predict. The hidden layer has 5 nodes.

For the feedforward network we have weights mapped between each layer, meaning we have a set of weights between the inputs and hidden layer and a set of weights between the hidden and output layer. This provides us with a matrix of weights, which we initialize with random values. We perform a matrix multiplication with our weights and inputs to receive a matrix of our

weighted sums. We apply a sigmoid to these weighted sums, providing us with values for each neuron in the hidden layer. We multiply this matrix of hidden values to a set of weights between the hidden and output to give us our output value. Again, we apply a sigmoid to the output.

For backpropagation, we take the error of the output by taking the difference between our predicted value and the expected value and apply a learning rate, which in our case is 0.5. We then update our hidden weights by subtracting the old hidden values and subtracting the product of the error and the hidden neuron values. To update the input weights, we take the old input weights and subtract the product of the error, inputs, and old hidden weights.

To train the network, we repeat the feedforward and backpropagation process a certain number of times, until the weights adjust enough to provide us with our desired outputs. We trained the network by running the FFNN 1000 times for each set of inputs.

To implement both, we required a couple kernel functions, which involved a matrix multiply, subtract, and sigmoid function. We used the matrix multiply code we implemented in our lab 3 code and made a simple function for subtracting our matrices and for applying a sigmoid.

Project Status:
We were able to create functions for forward passing and backpropagation that can take in 4 desired inputs, update the weights using a back propagation function written from scratch, and provide us with an output or the prediction of the FFNN. Unfortunately, we tried passing in a couple of different types of expected outputs, but we weren't able to find a way to accurately predict desired results. We had a bit of trouble understanding the feedforward network, and we didn't have enough time to fix our network enough to be able to predict an accurate output.

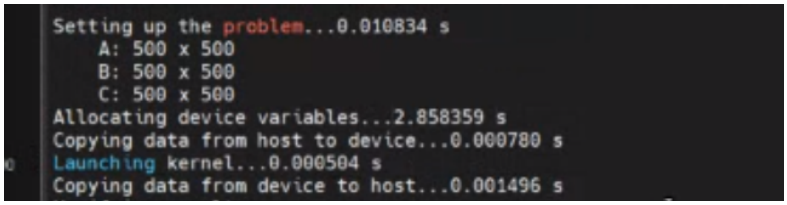
The functions we included do perform as intended. When passing in our matrix multiply, sigmoid, and subtract function, each performs as expected. The overall framework of the FFNN works as well. It is able to take 4 inputs and is able to perform a forward propagation operation then followed by a back propagation operation. It is able to update the weight for each layer and is able to forward and back propagate till the cost function is minimized.

We think the dataset we used might have not been the best to use with our NN after analyzing the results. Due to time constraints, we were unable to test out with other datasets.

Evaluation/Results:
Evaluation / Results. Timing of parallel code vs serial code. Timing of various implementations. Bottleneck analysis using profiling. Screenshots of your project running. Etc…

When implementing our FFNN, we tested the timing of the kernel launch with respect to our matrix multiply, since the majority of our network relies on matrix multiplication operations.

We also attempted using CUBLAS sgemm library for your project, however we did some testing and found that the CUBLAS kernel took a much longer time for the same matrix size.



Our own matrix multiplication implementation

CUBLAS matrix multiplication implementation

As seen in the first picture, the kernel takes 0.005s to launch. Compared to 0.645s when using CUBLAS.

Documentation:
To compile the project, you just have to run the Makefile. The make file should output an ./basic-ffnn file. After successful compilation, the project can be executed using ‘./basic-ffnn 5 4 1’. The 5 4 1 correspond to m, n, and k values which map to the FFNN's hidden, input and output layer nodes. The input, training and testing dataset for the FFNN are already preloaded as part of the main.cu file. The expected results of the FFNN should be training iterations of the training data and finally testing iterations using the testing data. The output of each training and testing iterations will be displayed.

Tasks/Contribution:

Task	Breakdown
Matrix multiply kernel function	Matthew Choi - 50%, Keerthi Korisal - 50%
Sigmoid kernel function	Matthew Choi - 50%, Keerthi Korisal - 50%
Forward propagation	Matthew Choi - 50%, Keerthi Korisal - 50%
Back propagation	Matthew Choi - 50%, Keerthi Korisal - 50%
Project report	Matthew Choi - 50%, Keerthi Korisal - 50%

*We worked on the project at the same time and put in equal effort on each task for this project.