GPU Accelerated Multilayer Perceptron

Max Robinson max.robinson@jhu.edu
Johns Hopkins University,
Baltimore, MD 21218 USA

1. Introduction

2. Background

This section describes the details behind how Multilayer perceptrons and the back propagation algorithms work.

2.1 Feedforward Neural network and Multilayer Perceptrons

A feedforward multilayer perceptron neural network is a structure for a learning algorithm that is uses connected nodes in a directed manner to solve the problems of classification or regression. Specifically feedforward neural networks are usually multilayered. These networks are typically structured as layers of nodes with non-linear activations functions that are connected to another layer of nodes.

When connecting nodes from layer to layer, every node from a layer is connected to every other node in the next layer. This is known as being a "fully connected" layer. Each connection from one node to another has a weight w_{ij}. A bias node with the value of 1 is also introduced at each layer, but no nodes connect to this bias, as it serves only to help serve as a threshold value (Rumelhart, Hinton, & Williams, 1986).

For a node, a non-linear activation function is used to determine and constrain the val ue of the inputs before passing on the value. It is important that this activation function be non-linear as this is what allows a network to learn non-learn models.

For this implementation, the non-linear activation function used is the logistic function. To calculate the value of a given node in the network, a linear combinations of the input values multiplied by the weight on the connection is taken as shown in equation 1 (Rumelhart et al., 1986).

$$x_j = \sum_i y_i w_{ij} \tag{1}$$

In equation 1, x_j is the jth unit or node and y_i is the ith input into j and w_{ij} is the weight between unit i and j. This linear combination is then put through the nodes activation function (Rumelhart et al., 1986) show in equation 2.

$$y_j = \frac{1}{1 + e^{-x_j}} \tag{2}$$

[0] Some pieces of this project writeup are derived from previous work by Max Robinson for CS499

In equation 2, y_j is the output of node j, with input value of x_j as described above. Again, notice that the activation function is non-linear.

To get the total error for the network, which is how difference the output of the network is from the expected output of the network, is described using a form of squared error. Formally E can be described by equation 3(Rumelhart et al., 1986).

$$E = \frac{1}{2}\sum_c \sum_j (y_{j,c} - d_{j,c})^2 \tag{3}$$

In equation 3, c is an index over the dataset, and j is an index over the output units. For example if there could be two output nodes of the network. y is the output of the network for a given output node. d is the desired state for that output node. This sum gives the total error of the state on a forward pass through the network. More simply $E = \frac{1}{2}(y_j - d_j)^2$ is describes the error at a given node j given an predicted output y_j and desired output d_j.

2.2 Back Propagation Algorithm

To train a feedforward neural network a technique called back propagation is used, which was developed by Rumelhart, Hinton, and Williams (Rumelhart et al., 1986). Back propagation is used to incrementally update all of the weights in the neural network based on the error of the output of the network. This technique uses Gradient Descent by following the partial derivative of the error function with respect to the inputs and outputs to optimize the network. This update is done in two parts, first calculating the update rule for the output layer, and then calculating the weight updates for the hidden layers. The update rule for the output layer is described in equation **??**.

$$\Delta w_{ij} = \alpha(d_j - y_j)y_j(1 - y_j)x_{ij} \tag{4}$$

$$\Delta w_{ij} = \alpha\delta_k x_{ij}, \delta_k = (d_j - y_j)(1 - y_j)y_j \tag{5}$$

Here, alpha is the learning rate, d_j is the correct output for node j and y_j is the estimated output of the network for the jth output node, and x ji was the input value of the node before the activation function was applied. This Δw_{ji} is then applied to all nodes j connect to output node i. This follows gradient descent because the first and second parts of the equation are the deriviates of the error function with respect to the outputs and the outputs with respect the activation function respectively (Rumelhart et al., 1986).

$$\sum_{k\epsilon downstream(j)} \delta_k w_{ij} \tag{6}$$

This sum is then combined with the derivative of the nodes activation function. This gives the total derivative update described by equation 7.

$$\delta_j = (y_j)(1 - y_j) \sum_{k\epsilon downstream(j)} \delta_k w_{ij} \tag{7}$$

$$\Delta w_{ji} = \alpha\delta_j x_{ji} \tag{8}$$

The update rule for weights for hidden layers is then described by equation 8. The update rule in equation 8 applies to all hidden nodes and weights. These updates are applied to the network following the typical gradient descent pattern. Once the network has finished training, the forward propagation part of the process of the network is used to get estimated outputs.

3. GPU Implementation with CUDA

Implementing neural networks can typically be done in two fashions. The first is to loop through the weights and the inputs at each layer and calculate the output for each node one by one for the next layer. Similarly for the back propagation, the error terms and appropriate delta values for each weight can be calculated individually for each weight. This approach works but tends to be slow since loops must be run to calculate all elements involved individually.

The second approach, and the one taken for this implementation, is to represent the weights and layer outputs in the networks as matrices. Representing the weights in this fashion allows for use of matrix multiplication to speed up calculations of network output. Additionally, the vector and matrix approach enables implementation on GPU architecture since many vector and matrix operations can be parallelized on a GPU. For this implementation, CUDA was chosen as the API to use to implement a multilayer perceptron on a GPU.

3.1 Network Architecture

To take advantage of multiple cores and threads on the GPU, the network architecture was structured into two parts: three weight matrices, and four vectors to represent layer outputs plus the input vector. The weight matrices were stored as column based matrices to be compatible with CUDA.

Each row in the matrix represents all of the weights from an input node (e.g. A) to all the nodes in the next layer. More importantly, each column in the matrix represent all the weights that are inbound on a single node in the next layer. In equation 9, $H1$ has inputs from all input nodes with weights A_{H1}, B_{H1}, C_{H1}, D_{H1}

$$\begin{bmatrix} A_{input} & B_{input} & C_{input} & D_{input} \end{bmatrix} \begin{bmatrix} A_{H1} & A_{H2} & A_{H3} \\ B_{H1} & B_{H2} & B_{H3} \\ C_{H1} & C_{H2} & C_{H3} \\ D_{H1} & D_{H2} & D_{H3} \end{bmatrix} = \begin{bmatrix} H1_{output} & H2_{output} & H3_{output} \end{bmatrix} \tag{9}$$

Equation 9 shows an example how a 1x4 input vector multiplied by a 4x3 matrix of weights provides a 1x3 output vector for the next layer. The 1x4 input vector is representative of a layer in the network. The 4x3 matrix represents the weights that connect the input nodes from the input vector to the next layer of 3 nodes.

3.2 Computation

To compute and train a multilayer perceptron for this implementation, there are two parts to the computation: the forward pass and the back propagation. All computation takes place on the GPU device, except for where explicitly stated that it does not.

3.2.1 Forward Pass

Once the network has been created as a set of matrices the forward pass of the neural network becomes a series of matrix multiplications, and kernel calls. The matrix multiplication takes an input vector and a weight matrix and produces the next layers raw output. The raw output of the next layer is equivalent to the x_j in equation 1. The CUDA cuBLAS library *cublasSgemm* function is used to compute the matrix multiplication efficiently.

After the x_js have been computed the activation function still needs to be applied to get the final output of the layer. To compute efficiently compute the output of the activation function in parallel, a custom CUDA kernel is applied to apply the activation function seen in equation 2. This kernel applies the activation function to each data point individually in parallel.

After both the matrix multiplication and activation function have been applied, the output of that layer is complete. The output from the layer just computed is then used as input into the next layer and the same steps are executed.

Once all layers have been computed the last layer represents the output values of the network. During training, all intermediate layer output values are held onto as they are required for updating the network during back propagation.

3.2.2 Back Propagation

For the implementation the Back Propagation algorithm is chosen to make updates to the network. The loss function implemented is a squared error loss function and all deltas and error functions are based on a gradient descent based approach based on the squared error loss function. The equations to describe the update functions are found in equations 4 - 8

To implement back propagation on a GPU several kernels are used to calculate the necessary pieces step by step in parallel. The first step is to calculate the δ_k error terms for the last layer. This is done in parallel using a custom kernel that applies the δ_k function seen in equation 5. The results provides the necessary components to update the last weight matrix.

For each hidden layer, starting from the layers closest to the output layer, the contribution of each node to the error for each output node is calculated. The contribution to error is specified by equation 6. This is calculated by iterating through the hidden layer and calculating the dot product of that row of the weight matrix and the δ_ks that were previously calculated. The *cublasSdot* function is used to compute the dot product. After the contribution to error is computed the δ_j values or error values for hidden nodes is computed with a custom kernel. The custom kernel applies equation 7 to each node output in parallel. These steps are done for each hidden layer.

Once all error terms have been calculated the weights of the network are updated. Updating the weights of the network applies equations 8 and 5. The weights are updated by enqueuing a series of custom kernels that update each column of the weight matrix with

Table 1: Program CLI options

Option	Value	Explanation	Required
archFile	string	file path to arch file	yes
weights	string	file path to weights file	no
training	string	file path to training data	no
groundTruth	string	file path to ground truth output data	yes, if training
evaluation	string	evaluation file path for data to evaluate	yes, if not training
output	string	output File path for weights	no
epochs	int	number of training epochs	no
alpha	float	learning parameter	no

the appropriate x_{ji} value as each column of the weight matrix is associated with a node. Weights are updated in place in the matrix.

4. The Program

The implementation of the Multilayer perceptron is compiled to an executable call "network.exe". To run the program the user must specify input parameters to the program. Table 1 specifies the command line parameters available to specify when running the program. Figure 1 shows the program output if the program is run with no parameters.

Figure 1: Program help output

```
Usage is: ./network.exe –archFile <> –weights <optional> –training <trainingDataFile> –
groundTruth <gtFile> –evaluation <dataFileForEval> –output <networkWeightSaveFile>
–alpha <.1> –epochs <200>
```

To run the program in any way, the program requires an architecture file. This file specifies what the network architecture for the multilayer perceptron is. The architecture file is a one line csv file that has four values specifying, in order, the size of the input layer, first hidden layer, second hidden layer, and output layer. Currently the program only supports multilayer perceptrons with this two hidden layer architecture. The size of the layers can be any size less than or equal to 1024 and greater than 0, though smaller values are suggested.

A weights file can be given to the network to specify a a set of weights to use for each layer. The file is a three line csv file. The first line should have $inputSize*hiddenlayer1Size$ number of float values. The second line should have $hiddenlayer1Size * hiddenlayer1Size$ number of float values. The third line should have $hiddenlayer2Size * outputLayerSize$ number of float values. Each values is read into the weight matrix in column order.

If no weights file is specified a random set of weights is created to fit the architecture of the network. The weights are initialized on the GPU using the cuRAND CUDA library. The weights are then shifted using a kernel function to be between the values of -0.5 and 0.5.

When training a network, the “–training” flag and the “–GroundTruth” flags must have values that point to a training file and corresponding ground truth file. The training file must be a csv file where each line is the size of the input layer to the network. The ground truth file must be a csv file where each line is the size of the output layer in the network. Ground Truth files should can not be larger than what can be loaded into host memory. The ground truth and training file should match up on a line by line basis (i.e. line 5 in training has a correspond line 5 in the ground truth file).

Training stops once the number of specified epochs is reached. If the number of epochs is not specifies, training runs for a default of 100 epochs. An epoch consists of a forward pass and back propagating through the network once for each data point in the dataset. For every forward pass through the network during training there is a back propagation.

When training, the program outputs the average error per epoch as well as statistics about how quickly each part of the training process is. Figure 2 shows a sample output of the network while training.

Figure 2: Sample program output during training

```
Average Error for Epoch #8: 0.010930
Epoch Time: 128.874817ms
Average Times of compute:
Iteration: 0.6860ms Forward Pass: 0.0374ms BackProp: 0.6172ms
```

The program is also able to use the network to provide output for a dataset. If an evaluation file is provided, the network reads in the evaluation file and only performs the forward pass of the network for each data point. Evaluation files should be structured the same as training data files, and must not be larger than what can be loaded into host memory. The network output is then collected and saved in csv format to a ”ResultsFile.txt”. Each line of the results file is an output from the network that corresponds to that line of the evaluation file as input data.

At the end of any execution, evaluation or training, if an output file is specified by the “–output” option, the weights of the network are written to that file. If no file is specified, the weights are written to a file called “weights_{}.txt” with “{}” replaced by the current linux epoch time.

5. Experiment

To test that the multilayer perceptron works a network was training on a dataset built for face recognition to see how accurate it can be. Additionally, the speed of the implemented network is compared the that of a commercial CPU based implementation, TensorFlow (?).

5.1 Accuracy

For our accuracy test, the dataset consists of 1x128 vectors that were created using OpenFace (Amos, Ludwiczuk, & Satyanarayanan, 2016) on a celebrity faces image dataset. The images first were passed through a face detection algorithm implemented using dlib, an open source c++ algorithm library. After faces were detected, OpenFace was used to embed the faces into a 1x128 dimensional vector.

The dataset was labeled to as a classification problem were the goal is to be able to identify two celebrities and categories those who are not the celebrities as "other". The two celebrities to identify were Anderson Cooper and Scarlett Johansson. The dataset was label as a 1's hot encoding dictating that the output layer of the network is size 3.

To train the network, the dataset was shuffled and two-thirds split off to be used for training while one-third is used for evaluation. The network had a network architecture of 128, 15, 10, 3. The network was trained for 300 epochs and a learning parameter of $alpha = .1$.

5.2 Speed

To compare the speed of the multilayer perceptron implementation in this paper, a multilayer perceptron was implemented using TensorFlow on a CPU. The implementations are compared with two different network architectures. One network has an architecture of 128, 15, 10, 3, and the other has an architecture of 128, 300, 200, 3.

The TensorFlow implementation for a multilayer perceptron is similar to the GPU implementation in this paper but there are two key differences. First, the TensorFlow implementation uses a softmax cross entropy loss function, while the GPU implementation uses squared error. Second, the TensorFlow implementation uses a slightly different optimization function, the Adam optimizer (Kingma & Ba, 2015). The Adam optimizer is a gradient-based optimization technique that uses momentum as well as other parameters to help with the optimization process.

Speed for each multilayer perceptron implementation is compared by looking at the average speed to complete an epoch and the average time to feedforward and back prop through the network.

6. Results and Analysis

This section details the results and analysis for each experiment run.

6.1 Accuracy

To measure accuracy, the error rate is calculated from the predicted classifications and the known correct classifications for the evaluation set. Equation 10 describes how to calculate error rate.

$$errorrate = \frac{\#ofmisclassifiedtestinstances}{totalnumberoftestinstances} \tag{10}$$

The network program itself does not directly calculate error rates, but instead reports the results per data point to a file. To calculate the error rate a small python program is used to compare the output data from the network to the ground truth test data. Network outputs are interpreted using an argmax approach for a one's hot encoding. Of the outputs from the network nodes, the node with the highest values is considered to be equal to 1, and all other values are 0.

After processing the network outputs, the error rate was about 0.02128 or about 2%. There were 94 data points in the test set, so this means that 2 out of the 94 evaluation data points were misclassified. This shows that the network was definitely able to learn to distinguish the two celebrities from the others most of the time.

6.2 Speed

The GPU implementation in this paper, called "network.exe", and the TensorFlow implementation were trained on the same set of data as the accuracy experiment. Average compute time is the time taken to do a forward pass through the network, calculate errors, and do a back propagation on a single training data point. The FP time is the time it takes to complete only the forward pass through the network. The overall average epoch time is the time taken to do a forward pass and back propagation for every data point in the training set once. The back propagation time is not explicitly reported due to constraints in the TensorFlow implementation that does not give insight in the back propagation portion of the algorithm.

Table 2 has the results for running the training and capturing timing metrics for training. The results show that for overall epoch time for both network architectures, the TensorFlow implementation is quite a bit faster. For the smaller network it is about two times faster and for the larger network it is about 23 times faster overall. However, for the forward pass time, network.exe was about three times faster than the TensorFlow implementation. This means that most of the disparity in compute times is in the back propagation portion of the algorithm.

Given the implementation of the network.exe, and the complexity of training a multilayer perceptron this sort of makes sense. The simplest part of training a multilayer perceptron is the forward pass. As described above in sections 3.1 and 3.2.1, the weights are represented as matrices and the forward pass is a series of matrix multiplications. Matrix multiplications on the GPU are done using an optimized parallelized CUDA library. This would make it the most likely to run faster than on a CPU even with basic linear algebra libraries. The highly tested and developed CUDA libraries provide great optimization and speed in this area.

The back propagation part of the algorithm is where the most custom code and kernels are run. This is where the biggest time differences show. There are a few likely reasons for this GPU implementation to be slower. One is data transfer and memory allocations taking place on the GPU. Second is a not having optimized weight update and error calculations. Third is that TensorFlow is also using a compiler called XLA (https://www.tensorflow.org/xla) which is an accelerated linear algebra compiler to optimize.

When running the multilayer perceptron, the GPU implementation described in this paper ends up allocating and deallocating memory on the GPU frequently. For example, calculating the errors for each layer in the network and using them for back propagation requires allocation of addition GPU memory to hold the errors. The allocated space is also then deallocated to ensure no memory is leaked. There are several places in the code where space is allocated and deallocated that might slow down the computation process.

Table 2: Program CLI options

Implementation	Network Arch.	Avg Epoch Time	Avg Compute Time	FP Time
network.exe	128,15,10,3	127.4913ms	0.6365ms	0.0426ms
TensorFlow	128,15,10,3	64.4191ms	0.3302ms	0.14902ms
network.exe	128,300,200,3	1914.3486ms	10.1816ms	0.0558ms
TensorFlow	128,300,200,3	84.9011ms	0.4649ms	0.1597ms

In addition, data must be moved to and from the GPU for each new data point. This requires copying data where the TensorFlow implementation does not need to.

The back propagation implementation for this paper is also probably not optimized properly. There are several places where loops are used to help calculate various parts or enqueue computation. For example, to calculate the contribution to error, a loop is used to call a cublasSdot function. Using a loop to call the cublasSdot function, while better than doing a looped dot product, is likely not as optimized as possible.

The "weightUpdate" kernel is also enqueued using a loop. While each kernel is enqueued and run in parallel, there might be a more efficient way to update all the weights in the network. Rather than enqueuing multiple kernels, it might be possible to write one kernel to update all the weights at once. The difficult part of this is that each column in the kernel must have the same x_{ji} applied to it.

Lastly, the TensorFlow API is a highly optimized commercial library even for implementation on a CPU. TensorFlow uses a compiler XLA (https://www.tensorflow.org/xla), that optimizes linear algebra operations on all different hardware, including CPUs. These optimizations enable the forward pass and back propagation to be as fast as possible and creates machine code that optimizes loops and kernel launches (https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html).

7. Future Work

8. Conclusion

References

Amos, B., Ludwiczuk, B., & Satyanarayanan, M. (2016). Openface: A general-purpose face recognition library with mobile applications. Tech. rep., CMU-CS-16-118, CMU School of Computer Science.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Rumelhart, D., Hinton, G. E., & Williams, R. (1986). Learning representations by back-propagating errors.. *323*, 533–536.