CUDA-based MLP Neural Network for Handwritten Digit Recognition

James Read
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
jread6@gatech.edu

***Abstract*—In this report, I present an artificial neural network implemented in CUDA C++ code that learns to recognize handwritten digits from the MNIST dataset. The network has 2 fully-connected layers, with 300 neuron units in the hidden layer. I implement the batched forward and backward passes of the network in CUDA code to take advantage of the high parallelism offered in GPU processing. Performance, training results and limitations are discussed.**

I. Introduction

Both the forward and backward passes were broken down into a sequence of functions, each of which were implemented in custom CUDA code. The operations include:

- Matrix-matrix multiplication
- Vector-matrix addition
- Element-wise matrix multiplication
- Categorical cross-entropy loss and its derivative
- ReLU activation function and its derivative
- Transposing a matrix
- Summing along the rows of a matrix
- Matrix-matrix subtraction
- Softmax and its derivative
- Gradient clipping

Unfortunately do to time constraints I could not include the simulation of using an Ferroelectric FET (FeFET) as the neurons like I originally proposed. The network has limited test-set accuracy and relatively slow training speed so I plan to refine the original network operation before including the FeFET as a neuron. Regardless, I gained a lot of practice writing CUDA code and plan to use what I have learned working on this project in future research. In the next section I will break down the high-level operation of the network.

II. Network and operation

The network I simulated was inspired from [1], the original composer of the MNIST dataset. The network is a multi-layer perceptron (MLP), which is essentially two fully connected layers. During the forward pass, 28 x 28 gray scale handwritten digits inputs are are flattened to a vector of length 784 and are multiplied by the first layer of weights. After this multiplication, biases are added and a non-linear activation function (ReLU) is applied. Then, the resulting vector is multiplied by the second layer of weights and biases are added. The result of the network is 10 values which are converted to probabilities by the softmax function. Each of the 10 outputs then represents the probability of the digit in the image being 0-9 respectively.

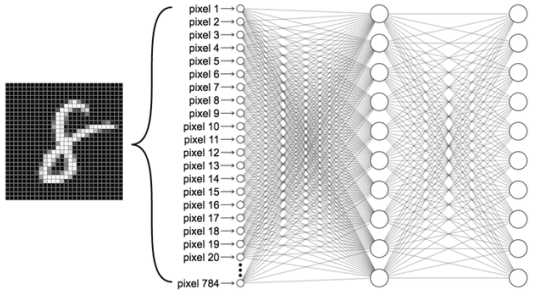

Fig. 1. Visual depiction of the multi-layer perceptron used to recognize handwritten digits. The second layer has 300 neurons and the final layer has 10.

In the backward pass, the loss of the network is calculated (how wrong the outputs were) and gradient descent is used to generate updates to each weight and bias in the network. These updates are calculated using a chain of derivatives that ultimately determine the significance that each parameter (weight or bias) has on the overall loss. For more information see [2]. Each derivative necessary to calculate the update values are executed on the GPU, and are implemented in the ”backward” function.

The MNIST dataset provides 60,000 handwritten digits for training and 10,000 for testing. I initialized the weights of the network randomly between 0-1 and normalized the input images to the range 0-1 before training.

III. Results

To improve the speed of the network I implemented batched forward and backward passes. This means that instead of running only one image at a time, a batched number of images can be run in parallel. This batched operation can greatly

TABLE I
Performance of Inference and Training

Batch Size	50	1000
Inference	0.81s	0.422s
Training	13.38s	2.81s

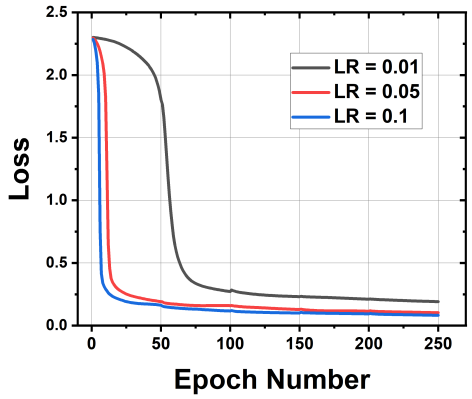

Fig. 2. Loss of the network during training with a batch size of 1000 and differing learning rates. The loss quickly decreases and then reduces more slowly with more training epochs.

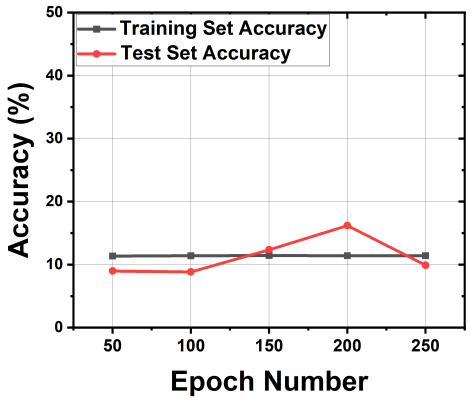

Fig. 3. Accuracy of the network on the training and testing datasets with a batch size of 1000 and learning rate of 0.1. Despite the reduction in loss over the training period, the network fails to achieve accuracy above 15% on the testing dataset.

benefit from the parallelism of GPUs and was an important goal of the project. See Fig. I for a comparison of the time it takes for one epoch of training and the inference operation of the network. Inference is the evaluation of the entire test set (10,000 images) and training is defined as training for one epoch (train once on all 60,000 images). We see that by increasing the batch size, the GPU is further utilized and can provide huge speedups, especially in the more computationally expensive backward pass. Increasing the batch size from 50 to 1000 improves the speed of about 5x.

To visualize the training process, Fig. 2 shows the decrease in the computed loss during training, and Fig. 3 shows the accuracy on the testing and training datasets with a learning rate of 0.1. Unfortunately, the network struggles to achieve high accuracy despite many training epochs. If I had more time to tweak the network, I would have included regularization techniques and dataset distortion to aid the network in escaping local minima during the optimization process. One technique I did include however was gradient clipping, where updates are clipped to a maximum value if they exceed a threshold. This was included to limit the exploding gradient problem.

IV. Code

The code is uploaded publicly to https://github.com/jread6/ECE-6122_CUDA_MLP To run the code, first compile with ”make cuda” in the main directory. To run the program type the following command with the following arguments in the terminal: ”./nn [number of hidden units] [batch_size] [number of training epochs (0 for inference only)] [load weights (bool)]”. The user can specify the number of hidden units in the network, batch size, number of training epochs, and whether to load weights saved periodically during the training process. The weights are stored in the folder ”current_weights”.

V. Potential improvements

The accuracy of the network could be further improved by using a special weight initialization technique such as Xavier initialization [3] and a learning rate optimization technique such as Adam optimization [4]. Additionally, it may have been beneficial to normalize the inputs between -1 and +1 instead of 0 and 1 to encourage more bi-directional updates of the parameters.

VI. Conclusion

In this report I give a breakdown of the neural network I implemented in C++ CUDA. The key takeaways are that batched forward and backward propagation provide huge speedup improvements, however the network needs to be further tweaked to achieve state-of-the art accuracy on the test dataset. Due to time constraints I leave the investigation of the FeFET device as a neuron to a future research project.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] S. Ruder, “An overview of gradient descent optimization algorithms,” 2016. [Online]. Available: https://arxiv.org/abs/1609.04747

[3] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. PMLR, 13–15 May 2010, pp. 249–256.

[4] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2014. [Online]. Available: https://arxiv.org/abs/1412.6980