# Machine Learning Framework in CUDA C++

Vhal Dhimant Purohit

AMS 148 Final Project Report

## 1 Introduction

The field of machine learning is currently the fastest growing technical field[1]. Its applications range from the field of Bio-informatics, to NLP, and Computer Vision, to name a few amongst very many more. Despite the disparate domains that it can be applied to, one finds that at the heart of any good machine learning model is a good machine learning framework. This is for two major reasons:

1. Most supervised machine learning models consist of repeated iterations of what is known as the forward and backward pass[2]. It is tedious to calculate the values for these passes by hand, especially since different layer types and sizes result in different values. Any machine learning framework worth its salt comes with an implementation of Autograd, which can automatically calculate the gradients for the backward pass, effectively allowing researchers to focus on tuning models to their application, rather than having to spend time calculating gradients for the model's backward pass.
2. Secondly, while machine learning and backpropagation algorithms have been around since the sixties, they only started getting traction in the last 12 years. This is in great part due to the advent of GPU computing. A GPU is designed for specific computations and its design allows it to make these calculations at a lightning fast pace. But, writing code that involves GPU acceleration without the aid of a deep learning framework is both time consuming and very difficult. A good deep learning framework thus creates abstractions between the user and the language and provide an easy interface to create models.

I propose to create a simple artificial neural network framework in *CUDA* C++, having both GPU and CPU capabilities, using which a user can create any architecture based on, but not limited to a feed forward neural network, and have the possibility of defining their own layers, which they can use along with the already provided layers to create their own models. The architecture of this framework is explained in the following section.

## 2 Architecture

This framework is inspired by the design of *Keras*[3], such that the user need only define the model's layers and its hyper-parameters, and can directly call a function that trains a model using those hyper-parameters without having to worry about writing the code for forward and backward passes. The framework has the following functionalities.

### 2.1 Matrix

All functionalities are built on a class named Matrix. Various operations such as addition, difference, matrix multiplication and Hadamard product are defined for this class. It has the following attributes

- int h: height of the Matrix
- int w: width of the matrix
- double []elements: One dimensional double array that stores the matrix using two dimensional indexing

### 2.2 Layers

Currently, the implementation contains definitions for 3 layers

- Linear Layer: This is a fully connected layer and is currently the building block of the framework
- ReLU Layer[4]: This is a non linear activation layer that maps its input to zero if its value is negative and is an identity function otherwise
- Sigmoid Layer: Apart from being a non-linear activation, this also acts as the final layer for bilinear classification, since it maps its input to the domain $[0, 1]$, which can be interpreted as the probability of the input belonging to a class.

Apart from using these layers, users have the capability of defining their own layers. This is due to the fact that all layers in the framework inherit the Layer class, and implement the following functions:

- void get_name(): Used to fetch the name of the layer when printing the summary of the model
- void get_taip(): Used to fetch the type of the layer when printing the summary
- Matrix forward(Matrix &X): Used to compute the forward pass for the layer
- Matrix backward(Matrix &X): Used to compute the backward pass for the layer
- void step(float alpha): Used to update the weights of the layers according to the gradient values

Please note that the implementation of these functions may be left blank for a user defined layer as long as they have the correct parameters and return values, but they absolutely must be implemented. This is because they are all virtual functions in Layer (the base class for all layers), and must have an implementation in the derived classes for it to be possible to construct an object of the derived class.

### 2.3 Loss

Loss functions are also implemented as classes and must be inherited from Loss, the base class, containing the following virtual functions

- Matrix compute($Y_{hat}$,$Y$): computes the loss matrix. ($Y_{hat}$ is the matrix of predictions from the model and $Y$ is the ground truth matrix)
- Matrix Backward(): Computes the gradient of the loss with respect to the output of the model ($\frac{dL}{dY_{hat}}$)

Currently, the implementation only contains mean squared error (MSE) loss, but users can write their own loss functions as long as they implement the virtual functions mentioned above in the class that they define

### 2.4 Model

Model is the class that handles the bulk of the code. The user is supposed to create a Model object, add Layer pointers to the model using
void add(Layer* layer) and call train() on a model object by passing in the following parameters to it

- Matrix X: the feature matrix
- Matrix y: the ground truth matrix
- float alpha: the learning rate
- int epochs: the number of epochs to train the model
- Loss* L1: pointer to the Loss object
- int categorical: 1 if the ground truth is categorical and accuracy needs to be computed, and 0 otherwise
- int verbose: 1 if loss and accuracy need to be printed and 0 otherwise

train() works by computing the predictions using model.forward() (which in turn calls forward() for each layer in the model), computing the loss using L1->compute(), computing the gradient of the loss wrt the predictions using L1->backward() and then calling model.backward() and step() (similar to model.forward()) to update the weights.

### 2.5 Autograd(-ish)

Since all layers inherit the Layer class, and have backward() defined for themselves, model.backward() can simply loop over the layers backward, with $\frac{dL}{dY_{hat}}$ as the starting point and compute the gradient for all the weights by calling backward on a particular layer, and use the value that that layer returns as the input to the previous layer's backward() function. While this isn't strictly considered Autograd (given its limited ability to only calculate the gradients for the layers defined) it nonetheless saves the hassle of having to derive the backpropagation equations by hand, and gives the user the ability to create custom architectures.

### 2.6 GPU Capabilities

Finally, since the framework has been written in CUDA C++, it also contains GPU functions for Matrix operations, which have been defined such that they can be called just like any other function, without the user having to understand their inner workings, in effect creating a barrier of abstraction allowing the users to harness the power of GPGPU without having to understand how it functions. All the user needs to do is go to the header file *matrix.h* and set the class attribute *gpu* to 1 and GPU capabilities will be enabled for the framework (please note that one actually needs to have GPU resources to use the framework's GPU capabilities).

## 3 Experiments

To assess the validity of the framework, experiments were conducted for both correctness and speed

### 3.1 Correctness

To assess whether the model was able to converge to a loss minima, *Scikit-learn* was used to generate 2 blobs,each containing 500 2D points clustered with standard deviation 1, essentially creating a binary classification problem (Figure 1 next page). Using a model with 2 layers, of sizes 30 and 20 respectively, and updating with a learning rate of 0.1, it was observed that both the CPU and GPU models converged within 1000 epochs. Their convergence can be seen in the graphs below (Figures 2-5 next page).

For those that are wondering why it took the models so long to converge (they both first start converging around the $200^{th}$ epoch) this may be attributed to the fact that the model is optimized using Vanilla gradient descent. It may be discerned that using a more sophisticated optimization technique would result in faster convergence.

### 3.2 Speed

To time how long it took each version to iterate over the same dataset over the same number of epochs, the same model as before was run for 20 epochs. Without a surprise, for small dataset sizes, the CPU version was ≈50x times faster than the GPU version, accounting for the Host to Device transfers, but for larger sizes the GPU was $6x$ faster. While those numbers seem disproportionately in favor of the CPU, one must note that for a dataset size of 10, the CPU took 2.8 milliseconds while the GPU took 119 milliseconds, so even though the ratio is huge, the numbers are both pretty small. When compared to speeds for large dataset sizes such as 100,000, even though the GPU is a meagre $6x$ faster than the CPU, the time taken by the CPU was 41990 milliseconds while the GPU took 6578 milliseconds. So even though the ratio is smaller when the GPU is

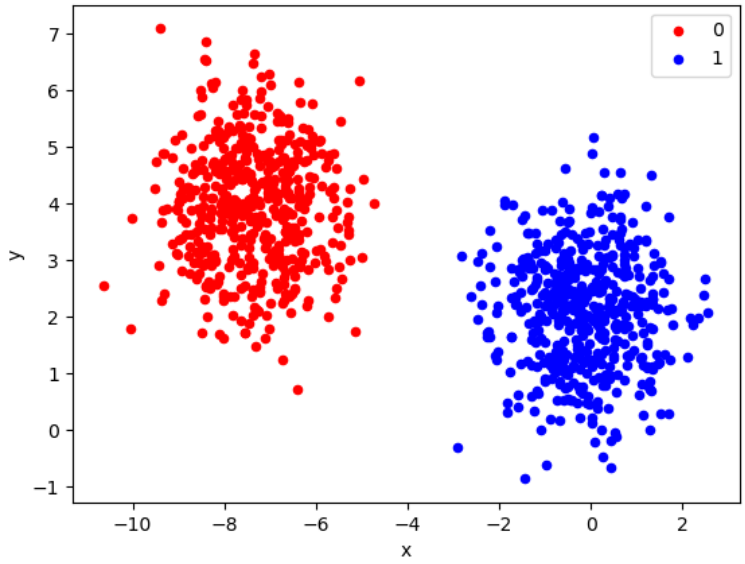


**Fig. 1.** Each blob contains 500 two dimensional points, and is clustered with a standard deviation of 1

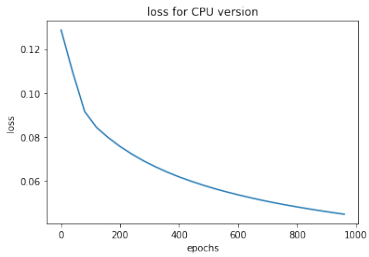


**Fig. 2.** CPU Loss

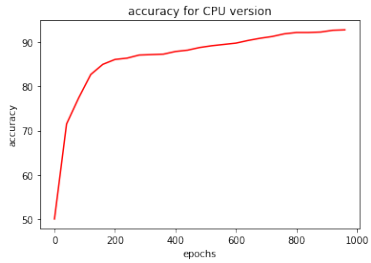


**Fig. 3.** CPU Accuracy

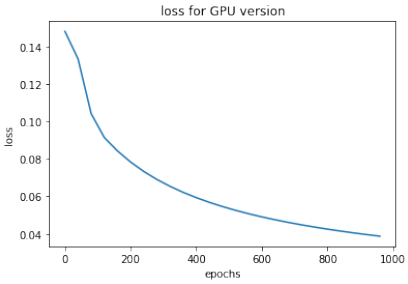


**Fig. 4.** GPU Loss

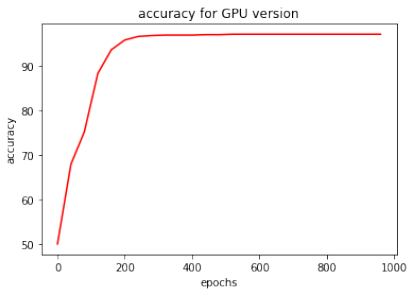


**Fig. 5.** GPU Accuracy

faster, the numbers are vast compared to the time taken for a small dataset size. The following graphs and table elucidate this point

| Dataset Size | 1 | 10 | 100 | 500 | 1000 | 5000 | 10,000 | 50,000 | 100,000 |
|---|---|---|---|---|---|---|---|---|---|
| CPU | 0.69 | 2.8 | 24 | 123 | 247 | 1280 | 2887 | 14568 | 41990 |
| GPU | 115 | 119 | 123 | 159 | 205 | 523 | 928 | 3789 | 6578 |

**Table 1.** Time taken in milliseconds for different dataset sizes by the CPU and GPU

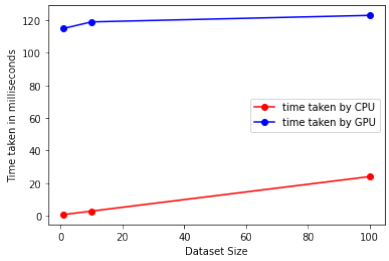


**Fig. 6.** For smaller sizes it seems as if the CPU performs exceptionally well

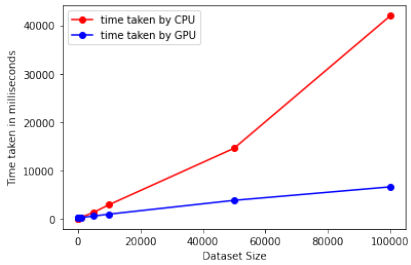


**Fig. 7.** But observing the scale of the y-axes clearly shows that this is not the case

## 4 Conclusion and Further Work

One major setback with the framework is that it is not memory efficient. Hence the code breaks down for large networks if the RAM is small. Apart from this, the following points need to be kept in mind for further work:

- Fix the allocation and deallocation of memory
- Implement batch processing and switch from 2D Matrix to multi-dimensional Tensor calculations
- Implement a few common Loss functions such as Cross Entropy Loss
- Implement common layers and activations such as Softmax layer
- Implement various optimization techniques such as AdAM[5] and Stochastic Gradient Descent

I heartily acknowledge the instructor for all his help during the course. Thanks Steven. You're the best!
The Github repo can be found here: NN_CUDA

## References


1. Andreas Holzinger. Interactive machine learning for health informatics: when do we need the human-in-the-loop? *Brain Informatics*, 3(2):119–131, Jun 2016.
2. A. Singh, N. Thakur, and A. Sharma. A review of supervised machine learning algorithms. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1310–1315, 2016.
3. François Chollet. Keras. `https://github.com/keras-team/keras`, 2015.
4. Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.
5. Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.