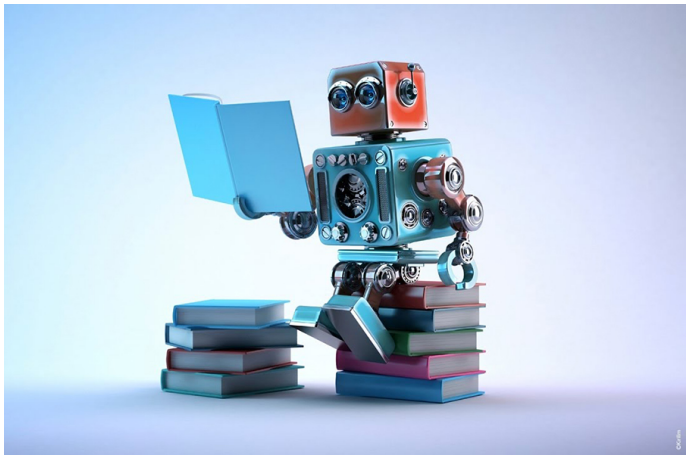
IIT BOMBAY

MACHINE LEARNING

Term Project

GPU ACCELERATION OF ARTIFICIAL NEURAL NETWORKS

-USING CUDA PYTHON



Ushasi Chaudhuri - 174310003

Megh Shukla - 173310008

May 5, 2018

GPU Acceleration of Artificial Neural Networks using CUDA Python

Ushasi Chaudhuri
Center of Studies in
Resources Engineering, IIT Bombay

Megh Shukla
Center of Studies in
Resources Engineering, IIT Bombay

***Abstract*—Artificial Neural Network model is an algorithm that mimics the functions of biological neurons in the brain. Biological neurons recognize and learn tasks through complex processes which rewire the interconnections between them. Till 1980's no such algorithm was present which would help artificial neural networks "learn" features from the sample provided. Only after Geoffrey Hinton's [1] ground-breaking work on Backpropagation algorithm was published, a wave of neural networks for machine learning was formed. Research was stalled at one point given the lack of computational power. With the advent of GPUs, Neural networks and in specific Deep Learning has made a comeback. In this project, we fuse 2 aspects of modern day computing: Python for Machine Learning and GPU acceleration on CUDA cores. We chose Python for GPU acceleration given all modern-day libraries employ C/C++ codes for backend GPU programming, hence lack of GPU programming explicitly in Python.**

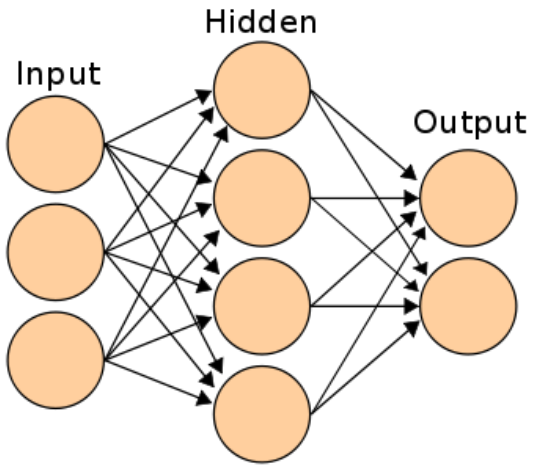
***Index Terms*—Python, GPU, Numba, CUDA, Neural Networks, GUI**

I. Introduction

We implemented the complete structure of an artificial neural network in Python, from scratch. To accelerate the time efficiency of the model, we parallelized the model using GPU. We used the CUDA API, created by Nvidia, for a parallel computing platform. The working of the model was testing on a stock market dataset. For the demonstration of the model, we build a GUI for making the interface user-friendly. The GUI was made using PyQt. Also, an executable file (.exe) of the submission file was created using PyInstaller. In our model, we have solved the problem for regression.

II. Artificial Neural Network

Human central nervous system is a complex network of approximately 86 billion nerve cells, each containing neurons, dendrites, synapses, etc. These neurons are connected to each other by Axons which creates electrical impulses to send messages throughout the network. It is one of the most complicated and naturally occurring design for learning process known to us. The core idea behind artificial neural network (ANN) is to use these kind of an inspired framework for developing computerized learning systems. Similar to this,the ANNs are composed of multiple layers, each consisting of multiple nodes. Each node is connected by links, which have associated weights with them. Fig. 6 shows an example of a single hidden layer artificial neural network. The most important part of learning in a neural network is the assignment of the weights to the links. This is carried by a back propagation algorithm.



Source:https://www.analyticsvidhya.com/blog/2014/10/ann-work-simplified/

Figure 1: A single hidden layer artificial neural network.

A. Back Propagation Algorithm

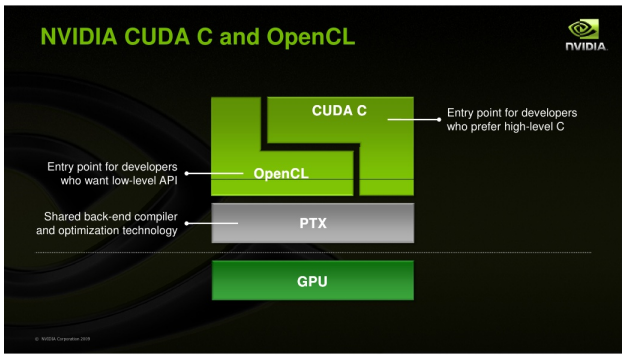
The back propagation algorithms mainly includes two main parts. The forward propagation of the incoming signal towards the output layer [2]. This part involves the calculation of the cost part, or commonly known as the error-term. Also, the propagation of the feedback from the output layer towards the input layer in order to train the model properly.

The second part involves the weight updation mechanism. The gradient of the weights are found by the output delta of the weights and input activation function. The derivative of the error term E with respect to the weights $w_{i,j}$ is as follows:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}} \tag{1}$$

Evaluating all the conditions we get,

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j} = \begin{cases} (o_j - t_j)o_j(1-o_j) & \text{if } j \text{ is an output neuron,} \\ (\sum_{\ell \in L} w_{j\ell}\delta_\ell)o_j(1-o_j) & \text{if } j \text{ is an inner neuron.} \end{cases}$$



Source:https://www.developer.nvidia.com

Figure 2: CUDA and GPU.

where, o_j is the output at the j^{th} node, and net_j is the sum. To update the weights, we use a gradient descent algorithm with learning rate η, momentum factor α and a regularization factor λ. The weight decay equation is given as:

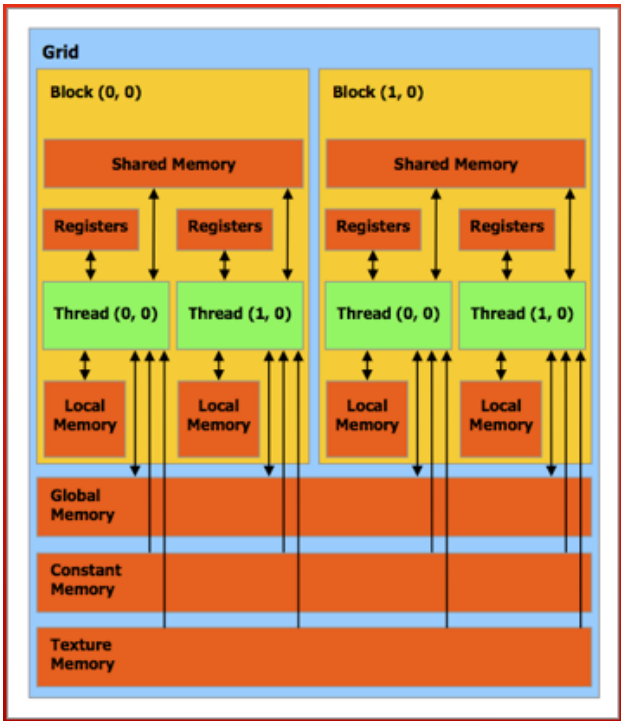
$$\Delta w_i(t+1) = -\eta \frac{\partial E}{\partial w_i} + \alpha \Delta w_i(t) - \eta \lambda w_i \tag{2}$$

For the activation function, we use the standard most commonly used Sigmoid function. The Sigmoid function is given as follows:

$$s(x) = \frac{\exp(x)}{1 + \exp(x)} \tag{3}$$

III. GPU and CUDA Python

"Compute Unified Device Architecture or CUDA was designed by NVIDIA incorporation to allow for General-Purpose computing on Graphics Processing Units [3]." GPUs were designed to facilitate computer graphics related computation, but the arrival of GPGPU augmented the capacities of the Graphic Processing Unit. GPGPU allows a programmer to execute computation intensive tasks on the GPU, which were traditionally meant to be run on the CPU. It exploits the large number of cores present in the GPU, thereby allowing massive parallel computations. The launch of CUDA in 2006 brought the GPGPU capabilities to NVIDIA Graphics Processing Units. NVIDIA designated each of the cores in their GPU as CUDA cores [4], which could be programmed in popular low-level languages such as C/C++ and Fortran. Low level languages are faster to execute since they are closer to machine level code compared to other high level languages such as Python. C/C++ also have the advantage that they are compiled languages, which means that code execution is performed by converting it into machine level language, performing all the linkages and then executing the end product. Languages such as python on the other hand are interpreted languages, where code execution is done on the fly. Python converts a given line of code into machine interpretable language, which is



Source:https://www.evl.uic.edu/aej/525/pics/cuda_memory.jpg

Figure 3: GPU architecture.

then executed on the processor. While interpreted languages are easier to debug, they fall short in execution speed when compared to their complied counter parts. However, the ease of development and faster development cycles have lead to the Machine Learning industry adopt Python as the de facto language for algorithm development. Speed of algorithms is increased by using C/C++ defined kernels for GPU acceleration complementing the Python code. This is practised by many Machine / Deep Learning libraries which employ C/C++ as their back end for GPU programming.

The CUDA software model involves the use of 3 basic tools: Threads, Blocks and Grids. A thread is a sequence of instruction that is being executed by a Processing core [5]. In the case of GPUs, specialized GPU functions called kernels invoke these threads to be executed on the Graphics card. Threads can be one, two or three dimensional. While the internal representation of the threads on the GPU is the same, a programmer could model naturally occurring 2 dimensional objects such as images with 2 dimensional threads, which helps in intuitively programming the chip. All the threads launched by a kernel executed the same sequence of code. A logical arrangement of threads is given in the form of blocks, which can in turn be multidimensional like the threads. Blocks contain threads, of all which will be executed on the same multiprocessing unit. Hence from the programming view, multiple blocks share the same multiprocessor in the GPU, implying all the threads on the block use the same multiprocessor for execution. Block execution is done asynchronous,

meaning Block 4 can get placed before Block 1 and 2 in the execution sequence. Block execution is defined by the PTX compiled code which optimizes the execution sequence of the code. Threads can use certain memory resources of the GPU, such as Global memory, Shared memory and local memory. Global memory is accessible by all threads launched by the kernel, shared memory is restricted to threads in a given block, with local memory unique to each thread. Shared and local memory occupy a high bandwidth, but come at the expense of limited size. Another type of memory is constant memory. As the name suggests, all constants are defined in the constant GPU memory which is specialized to reduce access time hence increasing the throughput of the device. A common scenario arising in GPU computing is the out of sync executions of the blocks in a Grid (Grid is a logical arrangement of blocks in 2 dimensions). To achieve some extent of control in execution, we could synchronize threads at block level and at grid level. A block level synchronization of threads makes use of *cuda.synchronize()* which halts executions of threads in the block till all the threads in the block execute the sync command. It is vital for all threads to execute the sync command, failing to do so will result in unforeseen behaviour. A Grid level synchronization requires indirect methods to achieve the objective. Consequent kernel calls will force all the threads launched by the kernel to finish executing before GPU memory resources are freed up for the next series of operations. Through the years, GPU architectures have evolved allowing faster memory accesses and caching, multiple cores and greater floating point arithmetic support. The generation of CUDA GPUs is given by it's Compute Capability, which denotes the arithmetic operations and functions that the GPU can perform.

A. CUDA Python implementation using Numba

Numba is an open source JIT (Just In Time) compiler which is used to compile from Python to CPU and GPU machine level code [6]. Numba is a project sponsored by Anaconda Inc (erstwhile Continuum Analytics), which uses the LLVM compilation of GPU kernel code into PTX code which is executed by the Graphics Processing Unit. The following is taken from the Numba website:

"*Numba gives you the power to speed up your applications with high performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters. Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to run on either CPU or GPU hardware, and is designed to integrate with the Python scientific software stack.*"

The CUDA python implementation builds upon Numba and provides CUDA specific methods such as transferring variables stored on the RAM to the global memory of GPU, allocating memory on the system, dealing with threads blocks and grids and multiple such functionalities [7]. Additionally, there are other libraries such as PyCUDA which provides implementation of common algorithms such as Fast Fourier Transform, Random Number Generation and Sorting.

IV. Implementation

A. PyQt4 for Graphical User Interface

There are two aspects of the program, the GUI which is handled in *ANNGui.py* and the Neural Network GPU acceleration handled by the *ANN˙gpu.py*. The GUI is used to accept network parameters, hyperparameters, dataset and other miscellaneous input. It's intention is to make the program user friendly at the same time treat the Neural Network algorithm as a black box for users not well versed with programming. The GUI was designed using *QtDesigner*, an application that is bundled along with **PyQt4** library for Python. To convert the '.ui' file generated by *QtDesigner*, we use a batch file provided along with PyQt 4 called "pyuic4.bat" with a cmd command to generate a ".py" extension.

Using the python file generated for the user interface, we can make calls to various functions based on global events such as *"triggered"*, *"clicked*" which are activated when various elements of the GUI are used. The GUI is built with robust fault tolerance as it checks for valid inputs corresponding to the fields provided. Error handling is managed by try and except statements with message pop-ups explaining the type of error. The GUI is made informative by including a text field box which displays the network parameters as determined by the user. In addition it also displays the error value over the training set. The GUI elements are laid out in a grid manner to give the interface a professional look. The GUI has been designed in such a way, so as to limit the hard coding as much as possible. The following fields are required for the user to input.

1) Choosing the dataset.
2) Number of data samples.
3) The training, validation, testing dataset split.
4) Number of neurons in input, hidden(s) and output layer.
5) Sigmoidal gain
6) Threshold value
7) Learning rate
8) Momentum factor
9) Regularization factor

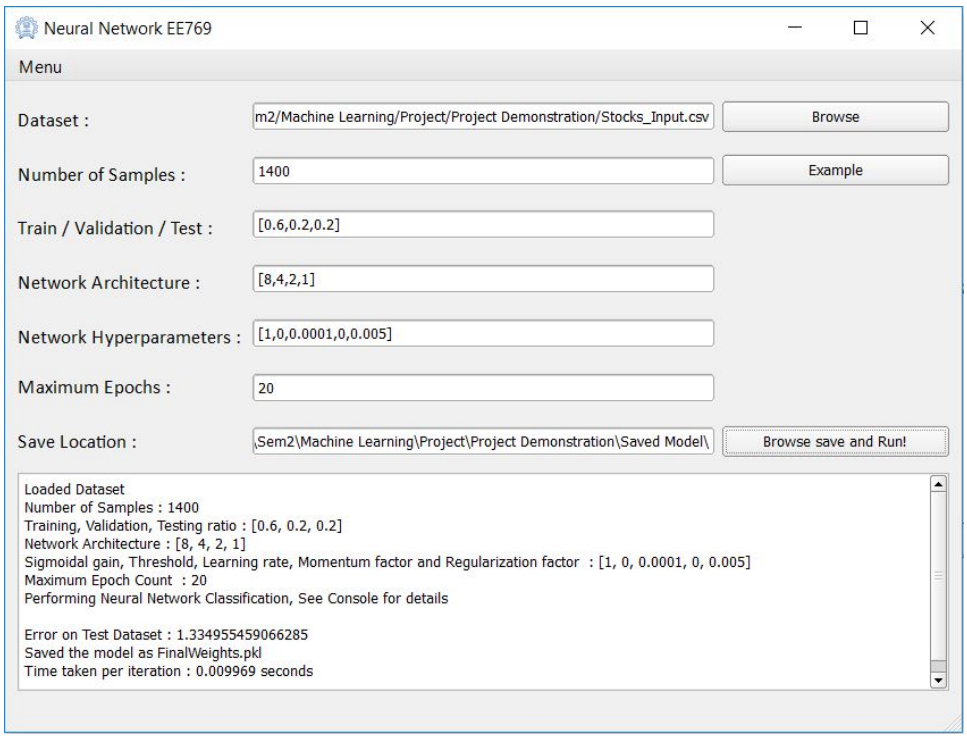


Figure 4: GUI interface for the system.

B. Numba – CUDA Python

The essence of Neural Network computation lies in the ANN˙gpu.py code. We have logically split the code in four segments, the first deals with imports and defining global variables and the second segment deals with declaration of CUDA kernels which are stored in the GPU memory. These kernels include matrix multiplication, neuron sensitivity calculations and weight updation functions. The launch of CUDA kernels is performed by specifying the number of threads and blocks required in each dimension along with the variables that need to be passed to the kernel. While it is tempting to place a large number of threads per block for faster parallel computations, there is a physical restriction on this number due to limited memory capacity on the GPU. Hence care needs to be taken to conform to the specifications of the CUDA device as stated by NVIDIA.

The third segment defines class for the Neural Network model and the layers associated with it. Class Neural Network deals with initializing the network hyperparameters such as learning rate, momentum rate and regularization as provided in the GUI. It also initializes the weights between the layers, the Kroencker deltas (sensitivity) associated with each neuron as well as a matrix which stores the weight change in the previous iteration. Class Neural Network also transfers these initialized variables to the device memory using CUDA functions imported from Numba. It should be noted that during the entire training process, the transfer of variables from the CPU RAM to the GPU Global Memory takes place only once. This is done to avoid large overheads in time caused due to frequent transfers between the CPU and the GPU, which has a huge detrimental impact on the performance of the algorithm. Once transferred to the Global Memory, all computations take place on the device, hence cost of transferring memory from host to device is amortized over all the epochs.

The Layers class has input and outputs methods associated with it, which calculates and stores input and output values for each neuron in the device memory. Two conditions have been specified for the input, if the layer object is an input layer or otherwise. Input layer takes it's inputs straight from the training sample provided whereas the other layers premultiply the output of the previous layer with the weight between the two layers. A similar logic follows for the output method associated with the method class. The computation of all methods in the Layer class is performed within the GPU.

The final segment deals with functions for Feedforward and Backpropagation evaluation, along with a function to preprocess the dataset as well as a function the streamline the execution of other functions. Essentially it acts as the "main" code associated with C/C++ code. Feedforward computes the input and output values in the manner prescribed by literature on Neural networks. Each Layer object computes the input and output associated with it's neurons and stores them in the device, which is used by subsequent layers. The backpropagation algorithm uses standard gradient descent algorithm to solve the optimization problem. For each layer, it's Kroencker delta is computed (unique to each neuron),

```
@cuda.jit()
def Sigmoid(NodeInputValues,NodeOutputValues, networkParameters):
    '''
    Sigmoid activation function
    input = network hyperparameters
    returns activated value in -1 to 1

    PLEASE NOTE : Sigmoid is scaled from -1 to 1 and not the original 0 to 1.
    '''
    threadIDx=cuda.threadIdx.x
    threadIDy=cuda.threadIdx.y
    blockIDx=cuda.blockIdx.x
    blockIDy=cuda.blockIdx.y
    blockX=cuda.blockDim.x
    blockY=cuda.blockDim.y
    i=(blockX*blockIDx)+threadIDx
    j=(blockY*blockIDy)+threadIDy

    if(i<NodeOutputValues.shape[0] and j<NodeOutputValues.shape[1]):
        NodeOutputValues[i,j]=(2*(1.0/(1+pow(exp,-networkParameters[0]*(NodeInputValues[i,j]-networkParameters[1])))))-1
    else:
        return
```

Figure 5: A sample CUDA code snippet.

which is used to compute the weight change for the layer it is associated with. The weight changes are obtain by multiplying the sensitivity values with the output of the neurons and it's learning rate. In addition we use momentum and regularization while computing the final weight for the iteration as given by the Backpropagation algorithm [Simon Haykin].

The function Preprocess is used to split the dataset into training, testing and validation components. It also scales the values to zero mean and unit variance so that weight updation takes place uniformly. Script function streamlines the execution of various functions and variable creation in order. The final output of the program is a Validation/Training error vs Epochs which depicts how the learning takes place. Validation curve is plotted to provide for early stopping if desired. The total error over the Testing set is computed and displayed with the weights being stored as a Python pickle file in the desired directory.

C. *PyInstaller*

To provide the code in an easily distributable format, we made an executable using PyInstaller, an open source framework for making Operating and CPU specific executables. We have provided a Windows executable for x64 systems, given that was the only operating system we were using. To make an executable, a "spec" file is prepared which contains all the meta-data required to build the executable. All hidden imports, additional data needed are to be specified in the ".spec" file. PyInstaller creates two folders, "Build" and "dist". "Build" holds all the information and error reports generated during the formation of executable, while the "dist" folder holds the executable that is used to distribute to consumers. For ease of use, we have prepared the executable as a stand alone entity.

V. Dataset

For testing our algorithm, we used a stock market dataset. The volume of the dataset is huge. It has about 8 lakh entries. However, it has a small feature set of about 8 features. Stock market prediction modelling is considered to be one of the most challenging problems in the field of machine learning. The model prediction is challenging as the stock market usually depends on a number of possible factors, which is difficult to be included as a feature set. Designing an exhaustive feature vector is technically not possible. Also, there are many un-quantifiable factors which affect the stock market drastically, such as a sudden fall of some big company, or recession. The stock market is in some sense modelled as a partially observable Markov process. A Markov process means that a future event depends on the present, and not on the past. The dataset on which we are working, tries to estimate the "Trade value" of each inputs. We ask the user to input the **train:validation:test** split ratio through the GUI, so as to avoid any kind of hard coding.

VI. Results

We were able to successfully design a neural network regression problem. The validation error was found to be reducing with the number of epochs. The test error also became convergent after a few epochs. We tested and found that nearly 40+ epoch were sufficiently converging the error graphs. Fig. 7 shows the validation error with the number of epochs and the testing errors with the number of epoch graph. The graphs are seen to be decreasing with epochs. A significant reduction in the time efficiency is observed as opposed to a non-parallelized artificial neural network.

```
Command Prompt
73914 INFO: Graph cross-reference written to E:\IIT Bombay\IITB\Sem2\Machine Learning\Project\build\ANNGui\xref-ANNGui.h
tml
74132 INFO: Appending 'datas' from .spec
74132 INFO: checking PYZ
74132 INFO: Building PYZ because out00-PYZ.toc is non existent
74148 INFO: Building PYZ (ZlibArchive) E:\IIT Bombay\IITB\Sem2\Machine Learning\Project\build\ANNGui\out00-PYZ.pyz
77101 INFO: Building PYZ (ZlibArchive) E:\IIT Bombay\IITB\Sem2\Machine Learning\Project\build\ANNGui\out00-PYZ.pyz compl
eted successfully.
77148 INFO: checking PKG
77148 INFO: Building PKG because out00-PKG.toc is non existent
77148 INFO: Building PKG (CArchive) out00-PKG.pkg
117684 INFO: Building PKG (CArchive) out00-PKG.pkg completed successfully.
117746 INFO: Bootloader c:\users\megh shukla\appdata\local\programs\python\python36\lib\site-packages\PyInstaller\bootlo
ader\Windows-64bit\run.exe
117746 INFO: checking EXE
117746 INFO: Building EXE because out00-EXE.toc is non existent
117762 INFO: Building EXE from out00-EXE.toc
117778 INFO: SRCPATH [('CSRE.ico', None)]
117778 INFO: Updating icons from ['CSRE.ico'] to C:\Users\MEGHSH~1\AppData\Local\Temp\tmpux42g7id
117778 INFO: Writing RT_GROUP_ICON 0 resource with 90 bytes
117778 INFO: Writing RT_ICON 1 resource with 1128 bytes
117778 INFO: Writing RT_ICON 2 resource with 4264 bytes
117778 INFO: Writing RT_ICON 3 resource with 9640 bytes
117778 INFO: Writing RT_ICON 4 resource with 16936 bytes
117778 INFO: Writing RT_ICON 5 resource with 67624 bytes
117778 INFO: Writing RT_ICON 6 resource with 18184 bytes
117793 INFO: Appending archive to EXE E:\IIT Bombay\IITB\Sem2\Machine Learning\Project\dist\ANNGui.exe
117871 INFO: Building EXE from out00-EXE.toc completed successfully.

E:\IIT Bombay\IITB\Sem2\Machine Learning\Project>
```

Figure 6: Making the .exe application using PyInstaller.

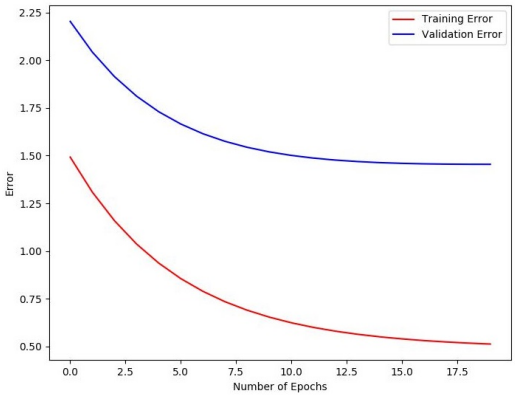


Figure 7: Validation and train error plot.

VII. Learning involved and Difficulties encountered

As a part of this project, we have learnt the architecture and usage of CUDA cores. We learnt to use the Python library for GPU programming, Numba, specifically catering to NVIDIA's GPUs. While we studied Backpropagation algorithm during EE769, we truly learnt the inner nuances while programming it. Hence we can claim that we have a thorough understanding of the subject. We have learnt to design and implement a GUI using PyQt4 which gives the project a polished appearance. We have also covered the extra mile by making a working executable that does not depend on the installation of Python on the user's machine using PyInstaller. All the above experience has been gained from scratch as a part of this project.

One of the most difficult elements while working on the Project was learning CUDA Python, due to its' steep learning curve and limited documentation and examples available. Essentially, transferring a variable data type into GPU memory obscures the contents of the memory from the user. As an example, if I declare a variable normally, I can access its value by printing out the variable. But once it's transferred to the GPU device, printing results in the display of it's datatype, which is now "cuda NDarray" and it's memory location. Hence there is no straight forward way of accessing it's contents, which is a frustrating experience while debugging. This implies most of the mathematical operations are difficult to debug, leaving any possible covert bugs extremely difficult to fix. Also, the natural representation of matrices is lost. On CPU memory, we can view the matrix as a 2 dimensional entity with numbers, whereas on the device, if I run a thread for all elements on the matrix to display the values, I will get a set of values printed serially without any logical semblance to a matrix, making a matrix interpretation of the values cumbersome. The example function provided by Numba for fast matrix multiplication did not work for us when we tried it out with a [rows x 1] and [1 x cols] matrix. Debugging this bug caused us a few days! There are multiple such issues

faced which for sake of conciseness we are not including.

While making an executable with PyInstaller, we faced frequent missing imports issues that lead to program crashes. It took us a while to figure out which imports were not being picked up by PyInstaller, such as those of Pandas and Scipy modules. A major issue we encountered is the crash of program on computers where Numba cannot detect the cudatoolkit file. **This is an error which cannot be fixed by us and depends on how swiftly Numba and NVIDIA address this issue to prevent this failure from happening.**

To summarize, the time needed to learn the use of these packages was exhausting, but in all we were successful in stringing them together to produce a working model!

REFERENCES

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Neurocomputing: Foundations of research," J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA, USA: MIT Press, 1988, ch. Learning Representations by Back-propagating Errors, pp. 696–699. [Online]. Available: http://dl.acm.org/citation.cfm?id=65669.104451

[2] S. S. Haykin, *Neural networks and learning machines*, 3rd ed. Pearson Education, 2009.

[3] " General-purpose˙computing˙on˙graphics˙processing˙units," https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units/, [Online; accessed 2-May-2018].

[4] " Accelerate Python code on GPUs," https://www.youtube.com/watch?v=jKV1m8APttU&index=10&list=PL5B692fm6vScfBaxgY89IRWFzDt0Khm, [Online; accessed 2-May-2018].

[5] "How-to-cuda-python," https://developer.nvidia.com/how-to-cuda-python, [Online; accessed 2-May-2018].

[6] "Numba," http://numba.pydata.org/numba-doc/0.37.0/user/index.html, [Online; accessed 2-May-2018].

[7] " Numba Python," https://numba.pydata.org/, [Online; accessed 2-May-2018].