**ECBE Department**
*SIUC*
May 2, 2022

# ECE 572 - Neural Networks
## *Project*

*Andreas Karatzas - 856523415*

# Contents

1

# Feed Forward Neural Network in C++ 17 and OpenMP for performance optimization

## 1.1 Abstract

In the project for course ECE 572, I will implement a feed forward neural network with sigmoid activation function.

## 1.2 Introduction

Artificial Neural Networks (ANNs) are used in wide range of applications including system modeling and identification, signal processing, image processing, control systems and time series forecasting. The baseline of those models is a special class called Feed Forward Neural Network[1, 2]. In this subclass of models, data is propagated forward only, and the model parameters are grouped in layers with no intra-communication. There is theoretically an infinite number of possible architecture regarding this type of models, hence they can have a lot of layers. In fact, Feed Forward Neural Networks were the beginning of a new Artificial Intelligence (AI) sub-field called Deep Learning, where the models used are designed to capture complicated patters. For that purpose, the model architecture as well as the dataset used to train them is very large. Consequently, in such cases the model performance poses a substantial challenge. There are several attempts on efficient ANN implementation using techniques, such as parallelization, to exploit the computational capabilities of the system architecture running a model[3]. Using low-level programming languages, such as C++[4], and frameworks that enable advanced parallelization and efficient data handling techniques, such as OpenMP[5], the programmer can achieve better results in terms of performance compared to most state-of-the-art Deep Learning frameworks[6].

For the project of course ECE 572, I will try to implement a feed forward model with sigmoid activation function. The user will have to make little to no configurations before the execution of the driver. It will be fully re-configurable regarding the architecture of the model. For example, the user will be able to define the layers and number of neurons using command line arguments. Moreover, the software architecture will follow that of well known deep learning frameworks, such as PyTorch[7]. That way the user will be able to easily navigate around the project if there is some prior experience with such frameworks. Furthermore, the user will be kept well informed throughout the data loading, the training and the inference process with *progress bars*. Finally, the advantages of the proposed project will be demonstrated using a well known dataset, Fashion MNIST[8].

## 1.3 Challenges

The implementation of a neural network in theory is an easy process. However, when it comes to putting together those formulas using software, the engineer is challenged to compile an efficient code implementation. The challenge becomes even greater when the project is carried out in a low-level programming language, such as C++, where the engineer has to solve substantial numerical and challenges since the project is based on scientific computation. After solving those challenges, the engineer has to structure the code in order to implement parallel software architectures and data pipelining for efficient execution. In this project, the optimization part will utilize the OpenMP framework for software parallelization and extreme device utilization.

## 1.4 Comparison

To actually realize the magnitude of optimization achieved by the parallel version of the project, I've implemented 2 more versions. The first is the *Python* version using *PyTorch*, which is the fastest deep learning framework in Python. The second is a `C++` implementation of the proposed feed forward neural network model running in serial mode. The *Python* version was implemented to capture the performance gap compared to the `C++` implementation. The results after training using the *Python* implementation can be viewed at figure 1.1. The results after training using the `C++` serial implementation can be viewed at figure 1.2. It is already clear that the `C++` implementation of the Feed Forward Neural Network is dominant. In fact the performance boost is above $1,000$ %.

The code for those implementations can be found at `https://github.com/AndreasKaratzas/nn-optimization`. The repository is currently private. If you would like to gain access please email at andreas.karatzas@siu.edu with your GitHub account, and I will add you as a collaborator.

```
Device utilized: cpu.

model(
  (input_l): Linear(in_features=784, out_features=150, bias=True)
  (hidden_1): Linear(in_features=150, out_features=100, bias=True)
  (hidden_2): Linear(in_features=100, out_features=50, bias=True)
  (output_l): Linear(in_features=50, out_features=10, bias=True)
)
[EPOCH    0] [LOSS 0.09133] [ACCURACY   5943 out of  60000] Work took   72.8 seconds
[EPOCH    1] [LOSS 0.09140] [ACCURACY   6549 out of  60000] Work took   72.5 seconds
[EPOCH    2] [LOSS 0.09221] [ACCURACY  13070 out of  60000] Work took   78.6 seconds
[EPOCH    3] [LOSS 0.07209] [ACCURACY  37415 out of  60000] Work took   80.9 seconds
[EPOCH    4] [LOSS 0.01148] [ACCURACY  51379 out of  60000] Work took   80.0 seconds
[EPOCH    5] [LOSS 0.00217] [ACCURACY  53666 out of  60000] Work took   80.8 seconds
[EPOCH    6] [LOSS 0.00079] [ACCURACY  54865 out of  60000] Work took   76.3 seconds
[EPOCH    7] [LOSS 0.00040] [ACCURACY  55687 out of  60000] Work took   72.9 seconds
[EPOCH    8] [LOSS 0.00020] [ACCURACY  56302 out of  60000] Work took   72.3 seconds
[EPOCH    9] [LOSS 0.00012] [ACCURACY  56811 out of  60000] Work took   73.5 seconds
C:\Users\andreas\anaconda3\envs\mnist-fcn\lib\site-packages\torch\nn\_reduction.py:42: UserWarning: size_average and red
uce args will be deprecated, please use reduction='sum' instead.
  warnings.warn(warning.format(ret))
[EVALUATION] [LOSS 0.08758] [ACCURACY   9430 out of  60000] Work took    4.9 seconds
```

Figure 1.1: Results from the Python implementation.

```
Loading training dataset        |**************************************************|  100%

Loading evaluation dataset      |**************************************************|  100%

Neural Network Summary:         [f := Sigmoid]
------------------------------------------------------------
Layer [1]        785 neurons
Layer [2]        151 neurons
Layer [3]        101 neurons
Layer [4]         51 neurons
Layer [5]         10 neurons

[EPOCH    1] [LOSS 0.15253] [ACCURACY  47246 out of 60000] Work took    7 seconds
[EPOCH    2] [LOSS 0.11542] [ACCURACY  50498 out of 60000] Work took    7 seconds
[EPOCH    3] [LOSS 0.10392] [ACCURACY  51505 out of 60000] Work took    7 seconds
[EPOCH    4] [LOSS 0.09715] [ACCURACY  52041 out of 60000] Work took    7 seconds
[EPOCH    5] [LOSS 0.09407] [ACCURACY  52363 out of 60000] Work took    7 seconds
[EPOCH    6] [LOSS 0.09024] [ACCURACY  52704 out of 60000] Work took    7 seconds
[EPOCH    7] [LOSS 0.08922] [ACCURACY  52668 out of 60000] Work took    7 seconds
[EPOCH    8] [LOSS 0.08407] [ACCURACY  53162 out of 60000] Work took    7 seconds
[EPOCH    9] [LOSS 0.08109] [ACCURACY  53381 out of 60000] Work took    7 seconds
[EPOCH   10] [LOSS 0.08000] [ACCURACY  53488 out of 60000] Work took    7 seconds

[EVALUATION] [LOSS 0.09077] [ACCURACY   8767 out of 10000] Work took    0 seconds

Benchmark results: 106.89459 seconds
```

Figure 1.2: Results from the serial C++ implementation.

## 1.5 Parallel Implementation

After having reviewed the benchmarks from the *Python* version and the serial `C++` implementation, we are ready to proceed with the parallel `C++` implementation. The effective core utilization percentage can be reviewed in figure 1.3.

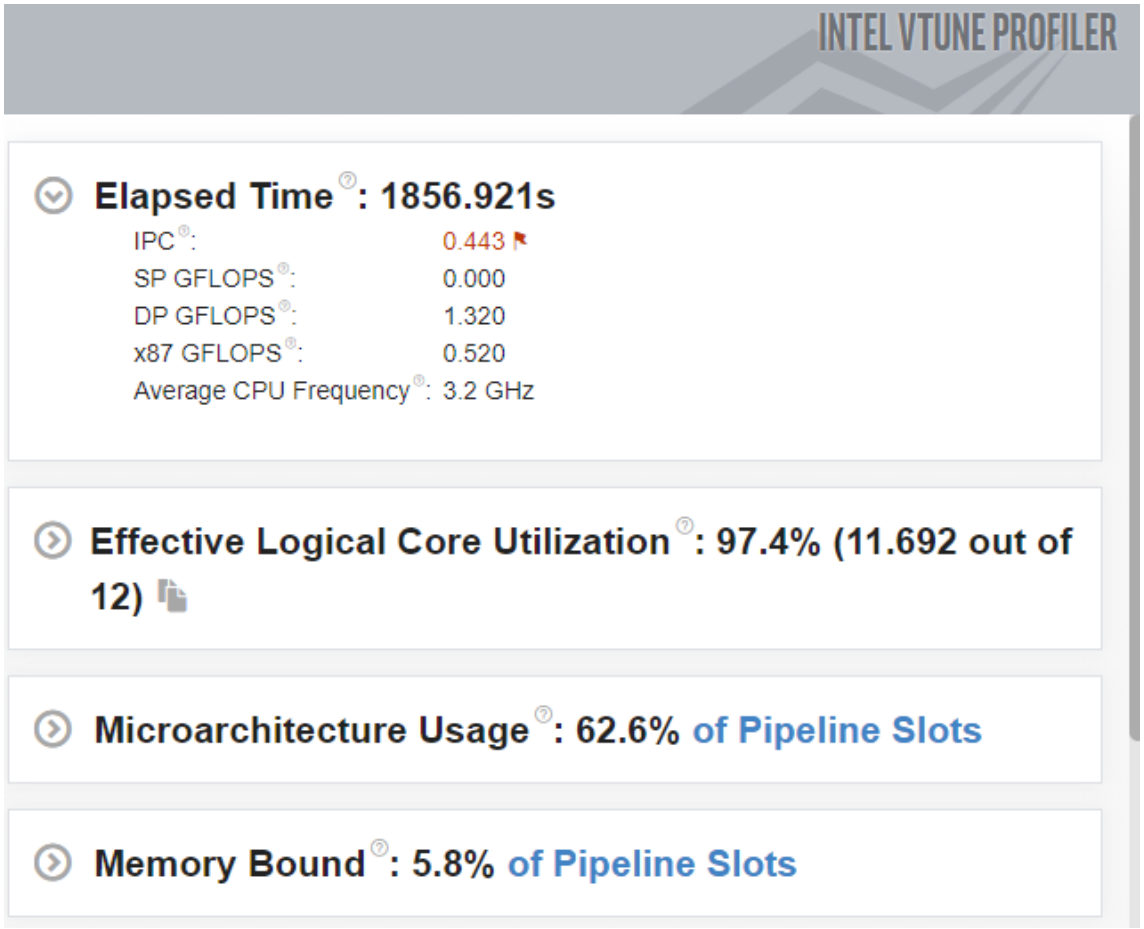


Figure 1.3: Effective Core Utilization using Intel VTUNE.

Finally, in figure 1.4 we can review a sample execution of our project. It is clear that we achieved more than 200 % performance boost.

Therefore, we can safely conclude that the project was a success since we gradually dropped the training time from 73 seconds to down to just 3 seconds per epoch.

## 1.6 Important Project Classes

The main project classes are:

- The *dataset* class
- The *nn* class

```
Loading training dataset        |************************************************|  100%

Loading evaluation dataset      |************************************************|  100%

Neural Network Summary:         [f := Sigmoid]
------------------------------------------------------------
Layer [1]        785 neurons
Layer [2]        151 neurons
Layer [3]        101 neurons
Layer [4]         51 neurons
Layer [5]         10 neurons

[EPOCH    1] [LOSS 0.14809] [ACCURACY  47720 out of 60000] Work took    4 seconds
[EPOCH    2] [LOSS 0.11222] [ACCURACY  50811 out of 60000] Work took    3 seconds
[EPOCH    3] [LOSS 0.10256] [ACCURACY  51589 out of 60000] Work took    3 seconds
[EPOCH    4] [LOSS 0.09621] [ACCURACY  52240 out of 60000] Work took    4 seconds
[EPOCH    5] [LOSS 0.09143] [ACCURACY  52575 out of 60000] Work took    3 seconds
[EPOCH    6] [LOSS 0.08790] [ACCURACY  52935 out of 60000] Work took    3 seconds
[EPOCH    7] [LOSS 0.08612] [ACCURACY  53087 out of 60000] Work took    4 seconds
[EPOCH    8] [LOSS 0.08290] [ACCURACY  53347 out of 60000] Work took    3 seconds
[EPOCH    9] [LOSS 0.08133] [ACCURACY  53419 out of 60000] Work took    3 seconds
[EPOCH   10] [LOSS 0.07926] [ACCURACY  53631 out of 60000] Work took    3 seconds

[EVALUATION] [LOSS 0.08861] [ACCURACY   8806 out of 10000] Work took    0 seconds

Benchmark results: 65.22168 seconds
```

Figure 1.4: Results from the parallel C++ implementation.

### 1.6.1 The dataset class

The dataset class handles the data that is being fed to the neural network. It includes the following features:

- Initialization of the a dataset instance using a CSV file.
- Print dataset.

A normalization process is also carried out during the data parsing phase. The normalization is based on a user defined value. This feature helps to reduce computational errors and achieve better results. If data is not properly normalized before training, then there is a good chance of neuron overflow, due to the sigmoid function. To initialize a dataset, the user must know the number of classes that exist in the dataset. THe user is required to also know the number of dataset samples. The type of data found in the dataset must be integers for the purpose of this project. Finally, the first column has to contain the class label of the corresponding sample, and the first row of the CSV must be a verbal description of the corresponding column.

### 1.6.2 The nn class

The *nn* class handles the neural network defined by the user. The user has the option to change the structure of the neural network. The user is asked to give the proper number of input and output neurons with respect to the number of input vector elements and the number of classes of the dataset that will be used to train the model. Moreover, the model structure is dynamic. In other words, the user

can define custom number of hidden layers and neurons. The structure is defined by the user using command line arguments. There are 3 types of command line arguments that are used to shape a model:

- `-i`: this argument corresponds to the number of input neurons. For example, `-i 784` corresponds to 785 (784 + 1 bias) input neurons[1].
- `-h`: this argument defines the number of neurons in a hidden layer. The user has the option to create a lot of hidden layers. For example, `-h 100 -h 50` tells the executable to create a model with 2 hidden layers, the first will have 101 (100 + 1 bias) neurons, and the second 51 (50 + 1 bias) neurons.
- `-o`: this argument corresponds to the number of output neurons. For example, `-o 10` corresponds to an output layer of 10 neurons. Let it be noted that there is no bias neuron in the output layer.

All neurons utilize the sigmoid function to determine active state. The aforementioned arguments must be given in that same order to the executable. An example model defined for the purpose of this project would be:

- For a Windows OS system: `.\nn.exe -i 784 -h 100 -o 10`
- For a UNIX system: `.nn -i 784 -h 100 -o 10`

Another useful hyper-parameter is the learning rate of the neural network. That rate is also defined at `Common.hpp` header and defaults to 0.1. This feature helps the model to avoid falling into some kind of local minima.

## 1.7 Implementation Analysis

In this paragraph, we go through analyzing the code for this project. To install the project, there instructions in paragraph **??**. To better understand the software architecture, we will describe the key-points of this project:

- The synapses of each layer are saved in the `weights` variable that lives in the `nn` class.
- The state of every neuron, i.e. the sum accumulated from neurons in the previous layer, are saved in the `z` variable that lives in the `nn` class.
- The result for each neuron after its activation is saved in the `a` variable that lives in the `nn` class.
- The `forward` routine that is defined at `nn` class implements forward propagation.

[1] The *bias* is added automatically, and therefore is not needed to be defined by the user.

- The `back_propagation` routine computes the error for each neuron, whereas the `optimize` routine backward-propagates each neuron's error. Both of those routines are defined at the `nn` class.

### 1.7.1 Driver.hpp and Driver.cpp

These files are the driver files responsible for the project execution. In the header file, we define the auxiliary header files. Those extra header files will be used throughout the project for various tasks.

### 1.7.2 neural.hpp

In the `neural.hpp` header file, we have defined the neural network class. The object of this class are instances of type `nn`. Every function of this header file is defined at:

- `fit.cpp`. In this file, we define the functions responsible for model training and evaluation.
- `forward.cpp`. In this file, we define the forward propagation algorithm.
- `optimize.cpp`. In this file, we define the backwards propagation algorithm and optimization process.
- `loss.cpp`. In this file, we define the error function (criterion) used for error computation.
- `utilities.cpp`. In this file, we define functions used to initialize the model parameters. We also define a function called `zero_grad`, which resets the state of the training algorithm before each epoch, in order to avoid processing garbage data. There is also a function named `predict` which calls the neural network using user-defined data. However, the user is responsible for the normalization of that data. Finally, we have implemented `summary` function that prints the neural network structure after compilation.
- `accuracy.cpp`. In this file, we define the function that evaluates the model accuracy.
- `export.cpp`. In this file, we define a function that exports all model parameters in order to be used at a later time for inference.

### 1.7.3 activation.hpp and activation.cpp

In these files, we have created some modules that are used for neuron activation. There are 2 implementation of the sigmoid function. For better precision, we utilized the `std::pow()` function of the `math.h` (or `cmath`) library. We also tested

the `std::exp()` function of the same library. However, due to numerical issues, this function returned inaccurate or sometimes false results. That motivated us to create a custom implementation of the aforementioned function using Taylor series approximation. We also implemented the *ReLU* activation function. In the code however, we solo-utilized the safe version of sigmoid function.

### 1.7.4 interface.hpp and interface.cpp

In these files, we have defined functions that implement a basic GUI. The GUI is responsible to keep the user informed on:

- The epoch error and accuracy.
- The epoch expected latency.
- Model summary.

There are a lot of extra function implemented, but they are currently not cross-platform and therefore are not utilized for the purpose of this project[2].

### 1.7.5 parser.hpp and parser.cpp

In these files, we define functions that help in the command line argument parsing process. There are some safety-check routines implemented in case the user doesn't use the command line arguments in a proper way. However, there are not that extensive and therefore the user is remind to use the command line arguments with caution as described at 1.6.2.

### 1.7.6 common.hpp

In this header file, we have defined all program hyper-parameters, such as the number of epochs and the learning rate (`EPOCHS` and `LEARNING_RATE`). We also define a *filepath* for the validation subset. Moreover, we also define the number of training samples and the number of classes found in the dataset. Finally, we also define the number of threads used to make the most of project run in a parallel fashion. We recommend the number of threads to be equal to the number of logical cores found in the system.

[2] Let this be considered as future work.

# Bibliography


[1] G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," *IEEE Potentials*, vol. 13, no. 4, pp. 27–31, 1994.

[2] M. Sazli, "A brief review of feed-forward neural networks," *Communications, Faculty Of Science, University of Ankara*, vol. 50, pp. 11–17, 01 2006.

[3] A. A. Huqqani, E. Schikuta, S. Ye, and P. Chen, "Multicore and gpu parallelization of neural networks for face recognition," *Procedia Computer Science*, vol. 18, pp. 349–358, 2013. 2013 International Conference on Computational Science.

[4] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley Professional, 4th ed., 2013.

[5] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, jan 1998.

[6] H. Jang, A. Park, and K. Jung, "Neural network implementation using cuda and openmp," *2008 Digital Image Computing: Techniques and Applications*, pp. 155–161, 2008.

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[8] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017.