Multi Layer Perceptron

Marcus Lee
marcustutorial@hotmail.com

December 4, 2023

1 Introduction

The aim of this paper is to implement a multi layer perceptron (aka Neural Network) without using any external neural network or machine learning libraries. The language I chose is C++, specifically **C++20**. Hence, there is a some manual effort required to compile the program since it is a static language.

1.1 Compilation instructions

To compile the project following tools has to be installed on the system:

- **CMake** - build tool for the project
- **libfmt** - a replacement for `std::cout`

These can be installed by downloading the executable on their official website or if you're on Ubuntu or MacOS:

```
# Ubuntu
sudo apt update && sudo apt install cmake libfmt-dev

# MacOS
brew install cmake && brew install libfmt
```

After that you can run the following commands to compile the executables

```
# Build the library
mkdir build && cd build && cmake .. && cmake --build . && make all

# Run the executables
./examples/xor/perceptron_example_xor
./examples/xor/perceptron_example_sin
./examples/xor/perceptron_example_letter_recognition
```

1.2 Code structure

The source code can be found in the attached zip file and it contains the tests, plots, data, etc. The multi layer perceptron is implemented as a C++ library and all tasks accomplished below have their own driver code importing the `MultiLayerPerceptron` class. The source code mainly resides in these few folders:

```
perceptron
├── include
├── src
├── examples
└── tests
```

Written code for the neural network can be found in `include/perceptron` and `src` whereas `tests` and `examples` houses some unit tests and the executable for the tasks.

2 Completed Tasks

2.1 Training a XOR network

The code for this can be found at `examples/xor/xor.cpp`. The code block below is the actual code that construct the network and Figure 1 below shows a visualisation of the network.

```
#define IN_FEATURES 2
#define OUT_FEATURES 1
#define HIDDEN_FEATURES 4
#define MAX_EPOCHS 10000
#define LEARNING_RATE 3.0
#define TARGET_ERROR 0.001
#define BATCH_SIZE 2

auto randomizer = perceptron::random::Xavier<Scalar>(IN_FEATURES, OUT_FEATURES);
auto activation = perceptron::activation::Sigmoid<Scalar>();
auto mlp = perceptron::MultiLayerPerceptron(
        std::vector<perceptron::Layer>{
                perceptron::Layer(IN_FEATURES, HIDDEN_FEATURES, activation),
                perceptron::Layer(HIDDEN_FEATURES, OUT_FEATURES, activation)
        },
        randomizer
);
```

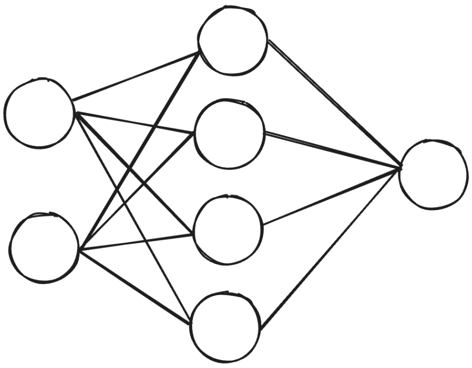

Figure 1: XOR Network

The results is as follows:

```
Epoch 534: 0.001006418985527181 (error)
Epoch 535: 0.001003637289454922 (error)
Epoch 536: 0.0010008703262003582 (error)
Error is less than target error (0.001). Stopping...
[0, 0]: [[0.027154, ]]
[0, 1]: [[0.968392, ]]
[1, 0]: [[0.967761, ]]
[1, 1]: [[0.0348811, ]]
```

As can be seen although the max epochs is 10000, the network converges much earlier than that with only 536 epochs. In fact, I have also ran it with 2 to 3 hidden units, the network also converges only that it needs more epochs. Note that the `BATCH_SIZE` here is for the **Stochastic Gradient Descent (SGD)**. The network can also be run without SGD by using the `.train()` method instead of `.SGD()`.

2.2 Learning $\sin(x)$ function

The code for this can be found at `examples/sin/sin.cpp` and the configuration and execution result is as follows:

```
#define IN_FEATURES 4
#define OUT_FEATURES 1
#define HIDDEN_FEATURES 8
#define MAX_EPOCHS 10000
#define LEARNING_RATE 0.001
#define TARGET_ERROR 0.001
#define BATCH_SIZE 100

auto randomizer = perceptron::random::Xavier<Scalar>(IN_FEATURES, OUT_FEATURES);
auto activation = perceptron::activation::TanH<Scalar>();
auto mlp = perceptron::MultiLayerPerceptron(
        std::vector<perceptron::Layer>{
                perceptron::Layer(IN_FEATURES, HIDDEN_FEATURES, activation),
                perceptron::Layer(HIDDEN_FEATURES, OUT_FEATURES, activation)
        },
        randomizer
);
```

For this task, `TanH` activation function is used intead of `Sigmoid` because there are negative values in the target and $\tanh(x)$ is in the interval $[-1, 1]$ but $\text{sigmoid}(x)$ is only in $[0, 1]$.

```
Epoch 9700: error is 7.881205768825114
Epoch 9800: error is 7.873048266608942
Epoch 9900: error is 7.864983282357822
Epoch 10000: error is 7.857008192932907
Input: [-0.20771906768934767, -0.28155039081825717, 0.4261937947634129, -0.6844652395547632]:
Expected [0.9263071737385499], Got [[0.925052, ]]
```

The result above shows that although this time the network does not achieve the `TARGET_ERROR` within 10000 epochs, it still did quite well a plot is provided below by using the expected values and the predictions from the network, the code for this plot is in `plot.py`.

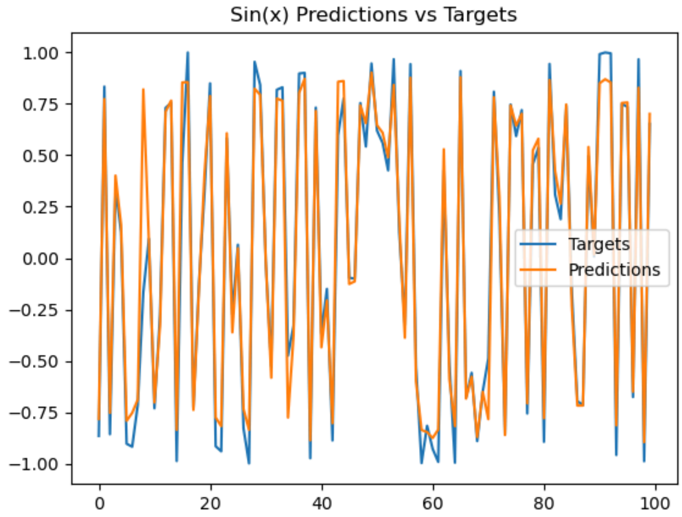

Figure 2: Predictions vs targets of sin(x) values

Looking at the figure above, we can see that the network is able to learn the patterns quite well, it almost resembles the original data points. In my opinion, the results is quite satisfactory given how quick the network arrives at this result. From my testing, using **SGD** allows the network to arrive at a local minima surprisingly faster than normal training. The results tabulated below shows a comparison in the number of epoch and corresponding error between the two methods. Note that the actual time taken to arrive at a certain epoch is very similar for the two and the difference is almost negligible.

Normal	**SGD**
565.70 (100)	6.99 (100)
249.13 (1100)	5.37 (1100)
32.03 (2100)	5.15 (2100)

Table 1: Sum of Squared Residuals (SSR) for SGD vs Normal Training

Note that the number shown above is calculated using *Sum of Squared Residuals (SSR)* as a loss function and the number inside the parenthesis is the number of epoch. It can be seen that SGD converges much faster and in fact this phenomenon is heavily affected by the `BATCH_SIZE` variable, I find that smaller batch sizes leads to faster convergence on average.

2.3 Letter recognition

The code for this part can be found in `examples/letter_recognition/letter_recognition.cpp`. The configuration is as follows:

```
#define IN_FEATURES 16
#define OUT_FEATURES 26
#define HIDDEN_FEATURES 16
#define MAX_EPOCHS 10000
#define LEARNING_RATE 0.001
#define TARGET_ERROR 0.001
#define BATCH_SIZE 16

#define DATASET_REGEX "([A-Z]),(\\d+),(\\d+),(\\d+),(\\d+),(\\d+),(\\d+),(\\d+),(\\d+),(\\d+),(\
\d+),(\\d+),(\\d+),(\\d+),(\\d+),(\\d+),(\\d+)"
#define TRAIN_DATA_PCT 0.8
```

The first feature of the dataset is a capital letter which is the expected label hence the network only takes the rest of the 16 features as inputs. As for outputs, it has 26 neurons because there are 26 capital letters from `A-Z`.

The code for data loading, processing and train test split is also included in the executable file. The gist of it is that it reads the file as a vector of strings and match over it using a regular expression. All the 16 features are used to construct the input vector and the first feature is encoded using a vector of 25 zeros and 1 one corresponding to the letter indices. For example, the letter A would be encoded as `[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]`.

Despite trying muliple combinations of activation functions, loss functions, adjusting the number of hidden units and the hyperparameters, the network is not producing a sensible result for the output. I have tried **Sigmoid** at the hidden layer and **Softmax** at the output layer which in theory should work but does not. The outputs that I get are just a vector of values between 0 and 1, not exactly 0 or 1 which can not exactly be classified as a letter.

Just few hours before submission only that I found out about **Cross Entropy Logits**, a loss function that incorporate Softmax and makes the model output “logits”, in other words the unnormalised output from the model. Because of that, although the model is not outputting exact values of 0 and 1, the output neuron that has the highest value will be considered the predicted output hence we just have to re-normalise the results outside of the model using an `argmax` filter and we will have the results we wanted. Therefore, an additional python script `evaluate_letter_recognition.py` is used to re-normalise the output logits and produce an evaluation. Reason being in Python we can use `numpy`'s `argmax` function for this task, the code snippet below shows how evaluation is done in the script:

```
# Code Snippet
targets = []
predictions = []
get_letter = lambda index: chr(index + 65)

for i in range(0, len(_targets), chunk_size):
  targets.append(get_letter(np.argmax(_targets[i:i + chunk_size])))
  predictions.append(get_letter(np.argmax(_predictions[i:i + chunk_size])))

accuracy = np.sum(targets == predictions)
```

Finally, I ran the model again with the following setup:

```
// Network setup
auto randomizer = perceptron::random::Xavier<Scalar>(IN_FEATURES, OUT_FEATURES);
auto sigmoid = perceptron::activation::Sigmoid<Scalar>();
auto linear = perceptron::activation::Linear<Scalar>();
auto mlp = perceptron::MultiLayerPerceptron(
        std::vector<perceptron::Layer>{
                perceptron::Layer(IN_FEATURES, HIDDEN_FEATURES, sigmoid),
                perceptron::Layer(HIDDEN_FEATURES, OUT_FEATURES, linear)
        },
        randomizer
);
auto loss = perceptron::loss::CrossEntropyLogits<Scalar>();
mlp.SGD(train_data, loss, MAX_EPOCHS, LEARNING_RATE, BATCH_SIZE, on_epoch_handler);
```

Sigmoid activation at the hidden layer and only used a linear activation function which does nothing at the output layer because we are going to evaluate the output outside of the model using the Python script. The output of the model is then written into CSV files and evaluated from Python, the result is as follows:

```
# Output
There are 4000 targets and 4000 predictions
Accuracy: 79.675 %
```

In the end the model is able to predict correctly $\approx 80\%$ of the letters in the dataset which is a big improvement comparing to before where it can't even get any sensible results at all. One thing to note here is that the effect of **SGD** is also quite significant for this task where playing around with the batch size does help the model to arrive at local minima faster (in fewer epochs).

Batch Size	1	4	16	48	100	1000
Error	12.16	11.16	12.66	15.26	16.83	17.60

Table 2: Cross Entropy Logits Loss with different batch size at Epoch 100

The results above are all recorded at epoch 100, we can see that as batch size increases, the model steps slower and slower towards the minimum, hence running with the smaller batch size trains faster.

3 Conclusion

I chose C++ specifically for this assignment because I know it will be the most challenging to implement a working neural network as it is much lower level compared to others eg. Python, Julia, etc. It was considerably easy to write the feed-forward part of the network however I struggled a lot in implementing backpropagation. The resources online are mostly Python and the mathematic expressions have a lot of notations which makes it not that straightforward to understand. For a few days I was stuck having a feed-forward only network until I sit down and figure out the math behind backpropagation, after I figure out all the intermediate steps needed for calculating the gradients for the entire network only that I was able to correctly implement it in code.

That said, there's still challenges faced as it involves a lot of matrix multiplication and I often get runtime error due to incorrect matrix dimensions. That said, having it completed now and looking back I really appreciate the people behind libraries such as PyTorch, Tensorflow, numpy, etc. as their work is really no easy tasks and without their work in the open source community, writing a neural network without any point of reference would be much more difficult. Note that I tried writing my own `Matrix` abstraction in C++ but gave up in the end as I was not able to get it right, there are too many edge cases to cover such as broadcast operation and in the end I opted for `NumCpp` instead which is an amazing library that replicates `numpy`.

I would like to highlight that the moment where everything clicked for me is when it comes to my realisation that neural network is nothing more than just an abstract data structure, like a tree or graph which can be represented in memory using adjacency lists (a.k.a matrix and vectors). Ever since then, it made much more sense while writing the code for it myself. In terms of discovery, it is also the first time I ever wrote **Stochastic Gradient Descent (SGD)** and only then I realise how amazing it is that such big improvements only require such less code. As for the letter recognition task, writing the code for it is not the challenging part but understanding what the model is doing and why it was not able to produce desired results is the key to deriving the final solution.

All in all, I really enjoyed doing the assignment despite the challenges and effort that I have put in, it helped me learn a lot and along the way I picked up knowledge not only in connectionist computing but also the ins and outs of solvinig linear algebra numerically in code.