Multi-Layer Perceptron software development

Ga Jun Young
ga.young@ucdconnect.ie
16440714

Abstract

Connectionist computing is an approach of the field of cognitive science. It can tackle a variety of interesting problems, including vision and machine learning, search and modelling of complex social systems, and all the more playful games involving microcontrollers such as robots, games, and networks. The experiment conducted within this report produces a connectionist computing solution in the form of a multi-layer perceptron (MLP). It applies materials from COMP30230/COMP41390 to build a Python constructed software. The MLP was tested against the XOR function, a sin function, and the letter recognition dataset from UCI in order to examine the learning capabilities of the software. The results presented in the report yield high accuracy and produced promising predictions.

1 Introduction

1.1 Project Background

Multi-layer perceptrons are multiple layers of neurons processing information. It allows a perceptron to be trained in multiple ways, by varying the strength of the input, weight, biases, activations, and layers. It is based on the idea that neurons have multiple sets of input cells.

The construction of a neural network are commonly built with Python. Tools like TensorFlow and PyTorch are available API libraries that help users to construct complicated models at ease. However, since this project is restricted, these libraries will not be utilized. Instead, they act as a reason for approaching a Python approach.

1.2 Problem Statement

The construction of a multi-layer perceptron requires the ability to take any number of inputs, any number of outputs, and any number of hidden units in a single layer. It is also a requirement to only utilize the sigmoid and tanh function for the activation of a hidden layer, while the output layer is confined to the sigmoid and linear function. The multi-layer perceptron learns by back propagation. As a result, corresponding derivative functions are required (sigmoid, tanh, linear).

Three tests are applied to evaluate the performance of the software. (i) The xor function takes the form of Table 1.2 with 2 inputs, 1 output, and a minimum requirement of 3-4 hidden units.

X1	X2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: XOR function

Preprint. Under review.

(ii) The sin function is a constructed dataset of 200 vectors with 4 components each. Each component is randomly initialized between -1 and 1. The resulting output is a single component of the form sin(x1 - x2 + x3 - x4).

(iii) The UCI letter recognition dataset[1] with 20,000 rows of data. Each row consists of 1 letter symbol followed by 16 numeric values that have been scaled between 0 and 15. The numeric value identify characteristics of the letter image. As a result, the 16 numeric attributes will be considered the input for the MLP model. The output will be a vector of 26 components representing the alphabet.

1.3 Outline of Report

The rest of the report will be structured as follows; In Section 2 an understanding of specific components of the MLP will be outlined. In Section 3 an overall setup instruction will help reproduce the MLP model. Section 4 will overview the predicted outcomes and evaluate the model's accuracy. Lastly, Section 5 will outline the learning outcome and possible improvement to the model.

2 Related Work

2.1 Feedforward Network

A single layer feedforward neural network with *n* hidden units can be described as Figure 1. It describes the flow of information in a forward fashion. Figure 1 is an example of the structure of the feedforward neural network when built for an xor function. It consumes two inputs *x1* and *x2* with *n* hidden units. The logistic is that we apply a tanh activation function to the hidden units and a further sigmoid activation function to the output layer.

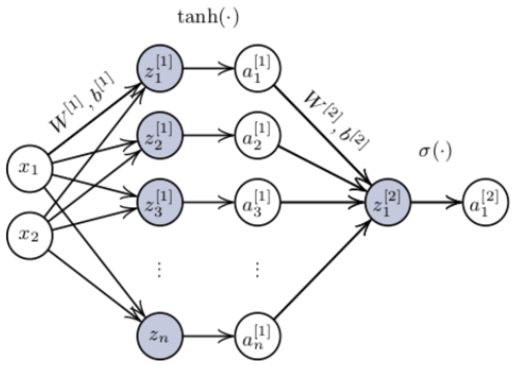

Figure 1: A single layer feedforward NN model [1]

The equations can be summarized as follows from the tutorial [1]:

$$z^{[1]} = x\,W^{[1]} + b^{[1]} \tag{1}$$

$$a^{[1]} = activationFunction(z^{[1]}) \tag{2}$$

$$z^{[2]} = a^{[1]}\,W^{[2]} + b^{[2]} \tag{3}$$

$$a^{[2]} = outputActivationFunction(z^{[2]}) \tag{4}$$

The *W* and *b* components are the weight and bias respectively. The weight reflects on how important an input (*x*) is to the overall model. While the bias is utilized to offset the generated results. Both the weight and the biases are updated in a gradient descent operation.

Within a feedforward network there are a multitude of activation functions to choose from. The activation functions determine the output of a neural network. Figure 1 should expect a binary output for the xor function.

[1]UCI letter recognition - `http://archive.ics.uci.edu/ml/datasets/Letter+Recognition`

2.2 Activation Functions

Sharma [3] explains that activation functions can be split between linear and non-linear types. A linear function is not confined to its range, whereas a non-linear function resides in the form of curves and ranges. For the constructed MLP software, 4 core activation functions are approached and thus, the following are presented.

Linear is a function that is visualized as a straight line. It is not confined to a range. Sharma [3] states that the linear function does not help with complexity or various parameters of usual data.

Sigmoid is a function that is visualized as a S-shape. Figure 2 visualizes the sigmoid function with an output range between 0 and 1. As a result, it is useful to predict probabilities that lie within this range, as well as multi-class classification [3].

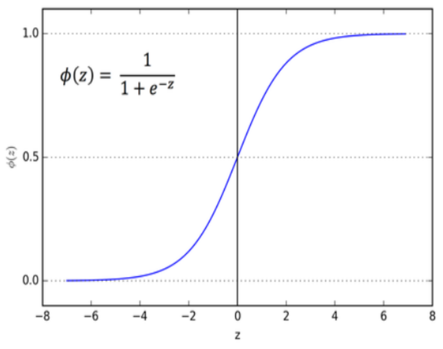

Figure 2: A sigmoid function [3].

Tanh is a function that is similar to the sigmoid function however, it is restricted to the -1 and 1 range. It has the capabilities of mapping outputs to the negative range which sigmoid was not capable of.

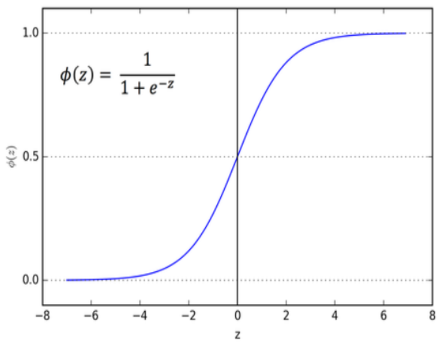

Figure 3: A tanh function [3].

Softmax is an activation function that is utilized to predict the letter recognition dataset. Like tanh, it has the same output range. However, it is capable of turning logits into probabilities using the exponential of the outputs [3]. It normalizes the numeric values and sums the exponents to produce a single value.

2.3 Backpropagation

The MLP is trained by applying gradient descent to update the weights and biases through backpropagation. The following summarizes the equations required to implement backpropagation of a single layer feedforward neural network [1]:

$$dZ^{[2]} = A^{[2]} - Y \tag{1}$$
$$dW^{[2]} = A^{[1]T}\, dZ^{[2]} \tag{2}$$
$$dZ^{[1]} = \left(dZ^{[2]}\, W^{[2]T}\right) \odot activationFunction'(Z^{[1]}) \tag{3}$$
$$dW^{[1]} = X^{T}\, dZ^{[1]} \tag{4}$$

The weight gradient is added onto the weight with a relatively small learning rate for gradient descent. Since the backpropagation uses the derivative of the activation function to update the curve and memorize the change, it is important to explore their mathematical formulas.

$$Linear' = f'(x) = 1 \tag{1}$$
$$Sigmoid' = f'(x) = sigmoid(x)(1 - sigmoid(x)) \tag{2}$$
$$Tanh' = f'(x) = 1 - tanh(x)^2 \tag{3}$$
$$Softmax' = f'(x) = softmax(x)(1 - softmax(x)) \tag{4}$$

2.4 Loss Functions

A loss function or cost function evaluates the performance of the MLP to the actual results. If the performance of the model deviates from the actual results then a large loss is presented. Parmar [2] introduces two categories of loss functions; regression losses and classification losses. The regression function deals with continuous valued predictions, whereas the classification deals with categorical values. Since the the three tests encountered deals with continuous outputs and multi-class classification both algorithms are required to be implemented.

For the regression function, we implement the mean squared error. It is the measure of average of squared difference between predictions and actual outputs. According to the article by Parmar [2], the squaring causes the predictions to heavily distance from the actual values when wrongly guessed. This allows us to evaluate a model and reduce the error as much as possible.

$$SquaredError = \frac{\sum_{i=1}^{n}(y^i - \hat{y}^i)^2}{n} \tag{1}$$

For the classification function, we implement the cross entropy function. The cross entropy loss increases as the predicted output differs from the actual output. Parmar [2] states that the cross entropy loss penalizes heavily on the predictions when they are confident but incorrect.

$$CrossEntropy = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y}) \tag{1}$$

3 Experimental Setup

Figure 4 is a similar representation to the MLP model constructed for the purpose of this project. It follows the design outlined by Lawlor et al. [1]. The initial layer is an input layer consisting of *N* inputs followed by *K - 1* hidden layer. After each forward propagation, a back propagation is carried out that calculates the loss at each epoch. An epoch is a complete presentation of training data. The resulting learning of the model presented by Lawlor et al. [1] yields a prediction output.

3.1 Model Architecture

The overall architecture is built with jupitar notebook through Python. A multi-layer perceptron class is constructed that holds the corresponding functions outlined in Section 2. The model design is a variant of the proposed model by Lawlor et al. [1]. The MLP constructed contains 3 layers, one

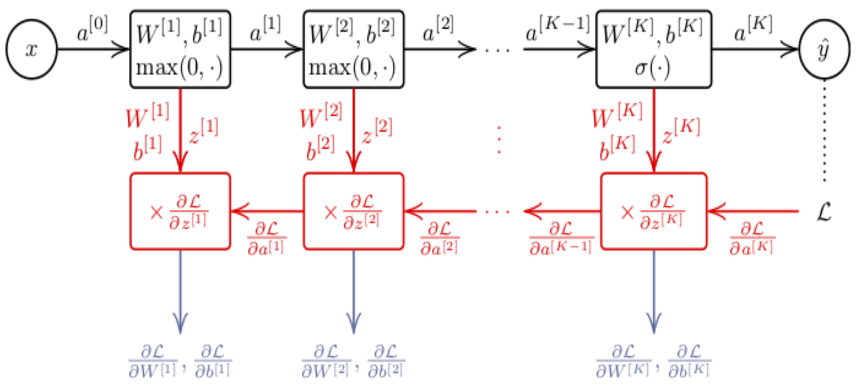
x
$a^{[0]}$
$W^{[1]}, b^{[1]}$ $\max(0,\cdot)$
$a^{[1]}$
$W^{[2]}, b^{[2]}$ $\max(0,\cdot)$
$a^{[2]}$
$a^{[K-1]}$
$W^{[K]}, b^{[K]}$ $\sigma(\cdot)$
$a^{[K]}$
$\hat{y}$
$\mathcal{L}$
$\frac{\partial \mathcal{L}}{\partial W^{[1]}}, \frac{\partial \mathcal{L}}{\partial b^{[1]}}$
$\frac{\partial \mathcal{L}}{\partial W^{[2]}}, \frac{\partial \mathcal{L}}{\partial b^{[2]}}$
$\frac{\partial \mathcal{L}}{\partial W^{[K]}}, \frac{\partial \mathcal{L}}{\partial b^{[K]}}$

Figure 4: A K layer NN model [1].

input, one hidden, and one output layer. It was not necessary to provide more hidden layers as the application became more complex, slow, and not necessary to solve the problem tasks.

The initial stage of the software requires an instantiation. Users are required to initialize an MLP object with the number of inputs, hidden units, and outputs. Followed by, the hidden layer's activation, the output layers activation, and 2 hyperparameters (learning rate and epochs). With these parameters, we were able to automatically decide on the appropriate loss function for the task based on Section 2.

The weights were randomly initialized using a method by He et al. [1]. It is a scaling of the Xavier initialisation. Lawlor et al. [1] states that by utilizing He initialization the performance is improved.

Each activation function and their corresponding derivatives were constructed outside of the class model as it did not coincide. However, the forward, backward, and update weight functions were implemented into the core class.

The methods alone will not produce a trainable network. Instead it is necessary to combine the constructed methods together. The fit method is constructed to repeatedly perform forward propagation, calculation of loss, backward propagation, and weight updates. The loss produced are saved in a log for visualization.

For the software to predict, a prediction method is created that takes in an input vector. It relies on forward propagation to produce a predicted output. Then, a relevant prediction procedure takes place based on the task at hand. A regression problem will simply append the predicted output into an array of outputs. While a classification problem will identify the class at which the predicted output belongs to.

Furthermore, the whole application is brought together under a model method that takes in the training and test set. This model can train and predict under one method. It can also generate the resulting training and test accuracy which will be used for the evaluation of the tasks performed.

4 Results

Task	*Train Accuracy*	*Test Accuracy*
XOR	99.1%	99.1%
Sin	99.8%	99.8%
UCI letter	81.9%	79.5%

Table 2: Performance of tasks

For the analysis, I considered the performance of 1 metric to evaluate the model and it's task. Accuracy is a common metric that measures all the correctly identified cases. It is generally used when all classes are important. Table 4 represents the resulting accuracy scores of the software. It shows that the difference between the training and test accuracy are rather close. This property shows that the results aren't over fitting in terms of accuracy output. In general, a large difference between train and test accuracy is a sign of over fitting. Another methodology to test for over fitting is by varying the learning rate, and the number of hidden units. A large learning rate may converge the model too quickly passing an optimal parameter value. This overshot is a sign of poor performance. Similarly, the number of hidden units can yield different accuracy results. If by increasing the hidden units, decreases the test accuracy then it is a sign that over fitting had occurred.

The xor function provides an output in the range of 0 and 1. Therefore, the output activation is sigmoid. Furthermore, the tanh activation for the hidden layer presented better results. The best test accuracy was 99.1% with an epoch of 3000, learning rate of 0.5 and the number of hidden units at 64. I have tested up up to 128 hidden units and on that specific iteration, the test accuracy dropped significantly and thus, early stopping is required. As the accuracy shows that after 64 hidden units there seemed to be a drop in learning outcome.

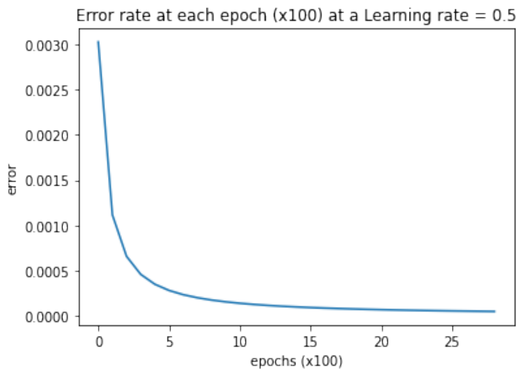

Figure 5: XOR model graph results of error against epoch.

The sin function dataset of 200 vectors were split between a training set of 150 and a test set of 50 samples. As the sin function outputs in the range of 1 to -1 a linear output is more suitable. Similarly, the tanh function acts as the activation function for the hidden layers. Figure 6 displays the best accuracy of 99.8%. It is constructed with a 5 hidden units, a learning rate of 0.1, and 1300 epochs. It is noted that after 5 hidden units, the accuracy drops which is a sign that we early stop the model's iteration. Figure 7 displays the outcome of the sin function which is visualized as a sin wave.

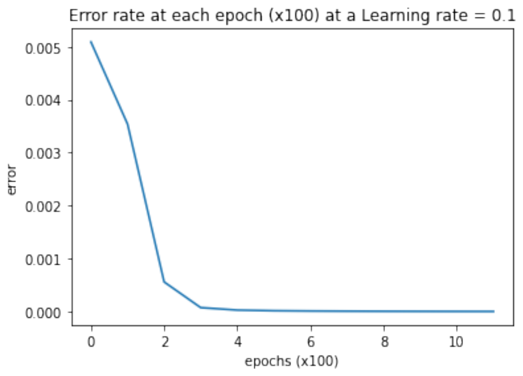

Figure 6: Sin model graph results of error against epoch.

To examine the validity of this result, Figure 7 represents the test and training set outputs. The dot representation is almost identical which shows that the 99.8% accuracy is valid and overfitting has not occurred. The 0.2 % difference is negligible and can be seen by the hints of the green triangle on the figure. Although this may still be overfitting, a lower learning rate of 0.01 was tested. The results also yielded a high test accuracy of 98.2%.

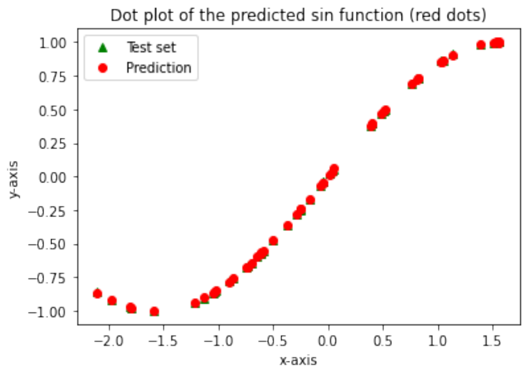

Figure 7: Dot plot of the predicted sin function.

The letter recognition dataset yields 20,000 rows of data. The dataset was converted to a csv file for python usage. The outputs were one hot encoded into a vector of 26 components of 0s and a single 1. The index at which the 1 is placed at represents the letter the vector corresponds to. Although the output, is mapped between 0 and 1, it was not possible to use sigmoid or the linear functions. The loss rate would appear extremely large and the predicted results were random. Instead, softmax was utilized with the given explanation in Section 2. In order for the model to work, a tanh activation function was used for the hidden layer.

The best sample test accuracy was 79.5%. The traininig took an extremely long time to process, thus an epoch of 1000 was used with a learning rate of 0.1, and 30 hidden units. Since the training dataset consisted of 15,000 vectors, the resulting accuracy is rather low. A better solution may involve using mini-batches, where the 15,000 would be split into 8 batches and trained. Nevertheless, the resulting model was capable of correctly predicting the majority of the sample which is better than random guessing.

Figure 8 represents an epoch graph. It is visible that there are peaks at moments where losses are great. However, the requirement was to use a minimum of 1000 epochs. The generated figure suggested that we should stop training at 400 epoch as the loss rate increases increases incredibly. However, the test accuracy did not seem to have decreased during the process therefore it was still a valid result.

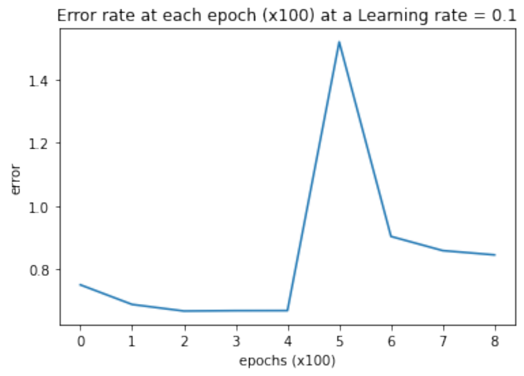

Figure 8: Letter recognition graph results of error against epoch.

5 Conclusion & Future Work

The construction of the MLP using a combination of knowledge from online resources, lecture notes and tutorials from deep learning and connectionist computing provided a handful of learning outcomes. My initial construction of the model was designed fairly similiarly to the tutorial of deep learning. It was for sure a multi-layer perceptron with K-1 hidden layers and the final layer being an output layer. However, I did not want to rely on this application built in the tutorial therefore, I designed a different model with the pseudo code provided in the assignment. As a result, the model became a 3 layer model with a single input, single output, and a single hidden layer. Although, it may be beneficial to add more hidden layers, the outcome of the accuracy did not change as much when comparing to the K-layer structure built from deep learning.

Initially, I naively inserted a learning rate, a large epoch, and a fixed number of hidden units as recommended by the assignment instruction. However, I realized that by exploring different values of each property, I can find better accuracy. As a result, it explored the context of overfitting. I realized that the learning rate could cause a missed opportunity whereby the optimal solution is missed. I also learned that the number of hidden units could heavily effect the resulting accuracy. Nevertheless, at certain scenarios having too large of a number for the hidden units caused long training periods. Similarly, a large number of epochs prolongs the training process.

By having created an MLP, I was able to interpret the reasons for various functions like the forward, backward, and loss functions as explained in Section 2. Beyond the implementation, I harnessed better Python skills. I wasn't fairly familiar with callback functions but after some Python API exploration, I figured out the process. It made the design of the application cleaner and more cohesive with less redundant code. The numpy library was also very benefitial as it allowed for dot products which were harder to implement in other languages. Therefore, I believe for the process of building an MLP, Python is a simpler approach.

To adapt the application, several ideas come into mind. The following is a list of solutions:

1. K - 1 hidden layers: by having the number of hidden units in an array format where each value in the array represents the number of hidden units for that index layer.
2. Implement more activation functions: there are many activation functions as described by Sharma [3]. i.e. ReLU, binary step, SoftPlus
3. Try out different loss functions: the application utilizes a squared error for regression problems and cross entropy for classification problems. However, there are many more loss functions that suit different activation functions too.
4. Improvements to the MLP: there are many ways to improve an MLP, dealing with large datasets may require regularisation to prevent overfitting. Dropout is a regularisation technique that trains a different model at each iteration. A mini-batch gradient descent may also be used to seperate the training into different iterations.
5. Scripts: global functions like sigmoid and its derivatives may be better off to be placed into script files. A notebook presents a step by step process with outputs of each phase, however the activation functions could have been simply loaded into the notebook instead. This encourages the reader to look at the MLP class and the results.

References

[1] A. Lawlor, B. M. Namee, G. Pollastri, and G. Silvestre. Comp47650 - deep learning tutorial. 2020.

[2] R. Parmar. Common loss functions in machine learning, Sept 2018. URL `https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23`.

[3] S. Sharma. Activation functions in neural networks, Sept 2017. URL `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`.