MLP Design Project
Connectionist Computing

Zachary Yahn

Introduction

The purpose of this project is to design, test, and evaluate a Multi-Layer Perceptron (MLP) on several tasks of increasing difficulty. The perceptron is designed "from scratch" without any help from existing machine learning libraries. The number of input units, output units, and hidden units in a single layer must be configurable to accommodate the various tasks of the project.

Design and Architecture

The MLP is implemented in Python with an object-oriented approach, such that other scripts can import and run the code. Its design is based on the formulas provided in the lecture slides, including efficient gradient updates by backpropagation. Training occurs in epochs, where one epoch consists of feeding individual instances of training data in the training set through the model and then updating the weights from the error. The model sees the entire training set for each epoch.

Upon initialising a new instance of the model, the user specifies a number of input parameters, beginning with the number of input, hidden, and output units. The constructor then initialises the weights of the three layers by sampling from a uniform random distribution over [-0.25, 0.25]. The user also specifies the desired activation function for the hidden layer and output layer, which can both be different options from either linear, sigmoidal, or tanh. Finally, the user indicates whether or not to use one-hot encoding on the output. Later on in the project, options to use mini-batching and dropout were also added.

Because the same code will be run over and over for up to thousands of epochs, optimising for speed is essential. Efficient matrix multiplication via Numpy is used throughout, including vectorized versions of the activation functions. The user has the option to run training in visualisation mode, which is significantly slower because it calculates training and testing errors along the way and saves them to an internal array for graphing.

Experiments and Results

XOR

As a nod to the history of machine learning, the first task of the assignment is to model the XOR function. The MLP is first tested with default parameters for 1000 epochs with a learning rate of 1, 3 hidden units, and a sigmoid activation function. The Mean Squared Error (MSE) function is used for evaluation. The error does not converge, and the graph of the training and test error shows that the model probably did not have enough time to learn. This

model also only correctly predicts half of the examples correctly. The same experiment is repeated but with 3000 epochs. Both tests are shown below in Figure 1.

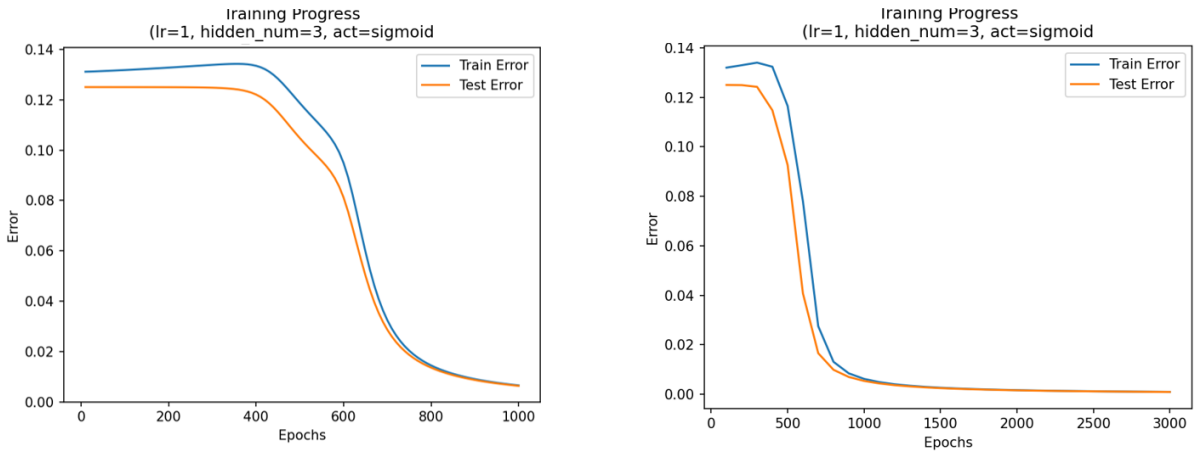

Figure 1: XOR Training for 1000 (left) and 3000 (right) epochs.

The results show that 3000 epochs is enough for the training and testing error to converge. The MLP trained for 3000 epochs also correctly predicts all four XOR cases, meaning it has successfully learned the function.

Sine

The second task is to learn a sine function of the sum of four input variables. 500 examples are randomly generated, with 400 of these used for training and 100 held out for evaluation. Building off of the success of the XOR model, the first iteration is attempted with a learning rate of 1, 7 hidden units, and sigmoid activation. Once again, MSE is used for evaluation. The same test is repeated with learning rates of 0.1 and 0.01. The results for training these on 1000 epochs are shown in Figure 2, below.

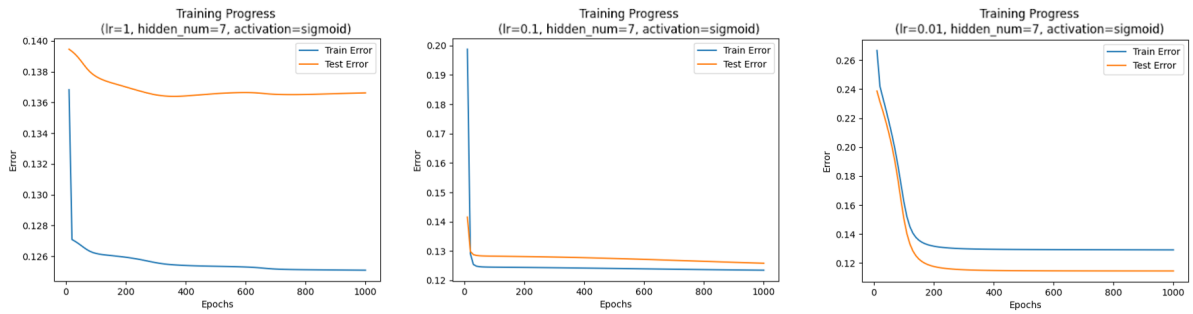

Figure 2: Sine Training for lr=1 (left), lr=0.1 (center), lr=0.01 (right).

None of these approaches achieve an error less than 0.12, so instead the same experiment is repeated but with a linear activation on the output layer. In hindsight this makes sense, because sigmoid is bounded to [0, 1] and sine from [-1, 1]. A sigmoidal activation is still kept on the hidden layer. This time, a learning rate of 1 causes the gradients to explode on the output layer, so the graph for that experiment is not shown. The results for the other two learning rates are shown in Figure 3, below.

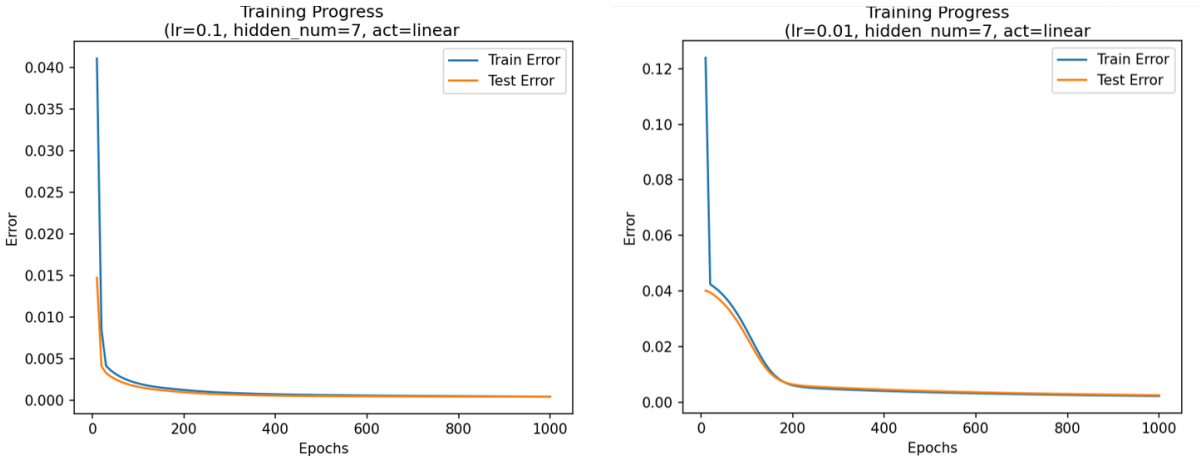

Figure 2: Sine Training with linear activation for lr=0.1 (left), lr=0.01 (right).

While both results converge to acceptable errors, using a learning rate of 0.1 appears to be especially effective. The results from these five trials are summarised below in Table 1.

Trial	lr	Hidden Units	Output Activation	Epochs	Test Error
1	1	7	Sigmoid	1000	0.125
2	0.1	7	Sigmoid	1000	0.126
3	0.01	7	Sigmoid	1000	0.110
4	0.1	7	Linear	1000	0.0004
5	0.01	7	Linear	1000	0.0024

Table 1: Sine training results.

Note that one can see from the graphs of Trial 4 and Trial 5 that the model is not overfitting; the test error is consistent with the training error. If the model were overfitting, one would expect to see a testing error that is much higher than the training error. This means that the MLP is successfully learning to approximate the sine function.

Character Recognition

The third and final task is to learn handwritten character recognition from a set of features. Once again, a training set and testing set are created from the provided data. This task was significantly more complicated than XOR or sine, and thus required multiple iterations of grid search with various network modifications to achieve a suitable accuracy score.

To begin with, 36 models were trained, one for each combination of hyperparameters. The model could use either a softmax or sigmoid activation function on the output layer, hidden layer sizes of 10, 15, or 20 units, and learning rates of 1, 0.5, 0.1, 0.05, 0.01, 0.001, and 0.0001. Most of these models did not converge, and instead either learned too slowly (learning rate too small) or oscillated between high and low error (learning rate too large).

The best model from this group used a learning rate of 0.01 with a hidden size of 20 units and a softmax activation function. Although sigmoid and softmax activation functions had similar performances when the other hyperparameters were held constant, it was clear from these trials that the hidden layer size was the most important factor, and that the learning rate could be tuned to work well for a given hidden layer size.

The next trial attempted to explore the limits of how large the hidden layer could be before serious overfitting occurred during training. This trial trained 32 models, with either sigmoid or softmax output activation function, hidden size of 75, 100, 125, or 150, and learning rate of 0.05, 0.02, 0.01, or 0.005. Most of the results from this trial were fairly similar, ranging from an accuracy of around 0.86 to 0.9. The most successful combination of hyperparameters was a hidden size of 100 with a learning rate of 0.005 using softmax.

Knowing that a lower learning rate with a softmax activation function would be most effective, a third grid search trial was run with all learning rates of 0.01, 0.001, or 0.0001 and hidden sizes of 10 or 100. This time, mini-batching was also implemented to evaluate whether this would help smooth out the training curve and push the testing accuracy higher. Models were trained with mini-batch sizes of 1, 8, 16, 32, and 64 for a total of 30 trials. All models used a softmax activation function. Sometimes the test accuracy did not converge, in which case it is marked with a "DNC." The results for this grid search are shown below in Table 2.

Trial	**LR**	**Batch Size**	**Hidden Dim**	**Test Acc**	**Trial**	**LR**	**Batch Size**	**Hidden Dim**	**Test Acc**
1	0.01	1	10	0.63	16	0.001	16	100	0.77
2	0.01	1	100	0.9	17	0.001	32	10	DNC
3	0.01	8	10	0.54	18	0.001	32	100	0.75
4	0.01	8	100	0.55	19	0.001	64	10	0.31
5	0.01	16	10	0.45	20	0.001	64	100	0.65
6	0.01	16	100	DNC	21	0.0001	1	10	0.59
7	0.01	32	10	0.44	22	0.0001	1	100	0.92
8	0.01	32	100	DNC	23	0.0001	8	10	0.54
9	0.01	64	10	0.44	24	0.0001	8	100	0.76
10	0.01	64	100	DNC	25	0.0001	16	10	0.41
11	0.001	1	10	0.57	26	0.0001	16	100	0.71
12	0.001	1	100	0.91	27	0.0001	32	10	0.35
13	0.001	8	10	0.5	28	0.0001	32	100	0.7

14	0.001	8	100	0.82	29	0.0001	64	10	DNC
15	0.001	16	10	0.37	30	0.0001	64	100	0.75

Table 2: Digit training results.

We can see from this trial that a hidden size of 100 and a batch size of 1 is by far the most effective approach, regardless of learning rate. Although batching did work, it required too much tuning of the learning rate to be an effective strategy.

Finally, another trial was conducted for 2000 epochs, where all models used a batch size of 1 with 100 hidden units. Several learning rates were tested, and the most effective one was found to be 0.0005, which achieved a final test accuracy of just over 0.94. The graph of this training is shown below in Figure 3.

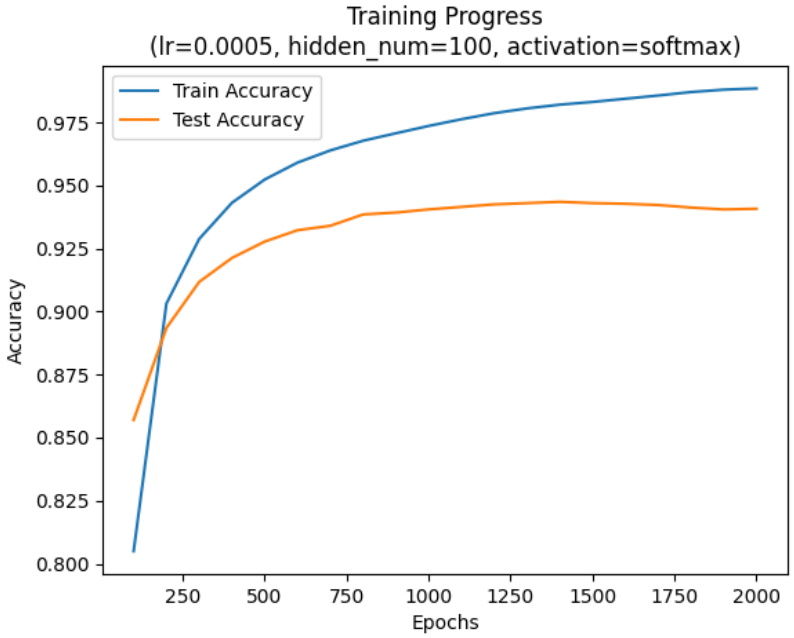

Table 2: Most successful training run.

Noticeably, the model achieves a training accuracy of nearly 0.99 and experiences overfitting after around 500 epochs. To attempt to combat this, a dropout hyperparameter was added to the model. However, for dropout levels of 0.2, 0.1, 0.05, and 0.005, dropout only worsened the performance of the model. For these reasons, the final performance of the MLP on the handwritten digit task rests at an accuracy 0.94.

It is also worth noting that doing such extensive tuning of hyperparameters might accidentally overfit the model to the test data. If this project were repeated in the future, a third validation set should also be used to validate the hyperparameters that were found to be most optimal.

Conclusions

This project, in particular the handwritten digit recognition task, was an excellent way to gain a better understanding of how an MLP functions. While backpropagation seemed like a rather opaque algorithm, implementing it forced me to go step by step and understand exactly what it was doing. The same goes for softmax, which was hard to grasp just by looking at the equation. Even the basic matrix multiplication behind forward propagation now makes much more sense on a deeper level after programming such operations.

Perhaps even more useful is the intuition I developed in the many iterations of training my MLP. I understand how batch size can be effective for smoothing out training, but only if an appropriately small learning rate is used for increasing batch sizes. I also witnessed how using more hidden layers is very useful for developing more complex representations of the inputs, at the cost of overfitting if the model is allowed to train for too long. Another important lesson was choosing the right activation function for the right problem, especially if the problem has restrictions on the domain of the output. Furthermore, I noticed that some of the “fancy” techniques like dropout and mini-batching have their uses, but that does not mean they are appropriate for every problem uniformly. In this case, the best model was one that used simple online learning without any dropout.

If I were to repeat this project in the future, I would be sure to use a validation set to evaluate hyperparameter choices. I would also try automated grid search sooner; for the first few days of working on the project I was running models one at a time. Despite these shortcomings and minor hangups, the results from training this MLP show that it successfully learned all three tasks to a high degree of accuracy.