F21BC Stage 1 Report – Group

Anthony Gabini
H00356910

Jiancheng Zhang
H00341619

Introduction

The purpose of this document is to record our efforts in the creation of an MLP-based ANN with the intent to of classify a provided dataset with the best performance achievable. We have been able to fulfil the requirements and will discuss our results as well as compare them to results found in literature.

Program Development Rationale

Creating efficient ANNs is important in the field of data mining and machine learning, as it allows for users to process great volumes of data at small performance costs whilst being as highly accurate as possible. This allowed us to form the two foundational principles of our program:

- Ensuring hyperparameters are implemented (and thus accessible) as arguments, making experimentation easy when varying values.
- Creating an easy-to-understand code structure to help with debugging and understanding for viewers.

We initially chose batch learning as the algorithm for our ANN program, however we moved to mini-batch learning as a means of compromising between batch learning and SGD.

Methods

Our program is split into two python files

1. Mlp.py: This houses the mini-batch algorithm implementation. This is composed of a class of two functions (as well as defined activation functions), shown below:

```
def __init__(self, number_of_layers,
number_of_neurons,
activation_functions, learning_rate,
loss_function):
```

```
def fit_and_transform(self,
data_original, test_data, outcome,
epochs):
```

2. Main.py: This initialises the MLP, and includes a result printing mechanism, printing directly to a file called “results.csv”.

```
myMLP = newmlp.MLP(2, [8, 1], ['tanh', 'sigmoid'], 0.001, 'cross_entropy')
result = np.array(myMLP.fit_and_transform(X_Train, X_test, y_train_outcome, 10))
```

We chose to loop through each experiment 20 times, as we believed this would provide a substantial range of values to determine average performance values.

The number of experiments done was chosen by identifying extreme points for each parameter to define a range, and then going through the range systematically. Implied behaviour on the edge of the ranges gave us reason to extend the range itself for further experimentation.

We note the difference between the implementations of epochs and the other hyperparameters. Epochs are implemented using for-loops, as seen in the code extract below:

```
# the number of epochs
for j in range(epochs):
```

This results in a far more time needed to go through each epoch as compared to increasing neuron count. This is because neurons are implemented via the use of matrices, as shown here:

```
self.weights[0] = np.random.rand(self.number_of_neurons[0],
data_original.shape[1])
self.biases[0] = np.random.rand(self.number_of_neurons[0], 1)
for i in range(self.number_of_layers):
    if i == 0:
        continue
    else:
        # initial hidden layer and output layer
        self.weights[i] = np.random.rand(self.number_of_neurons[i],
self.number_of_neurons[i - 1])
        self.biases[i] = np.random.rand(self.number_of_neurons[i], 1)
```

which are computationally quicker to process than a series of loops.

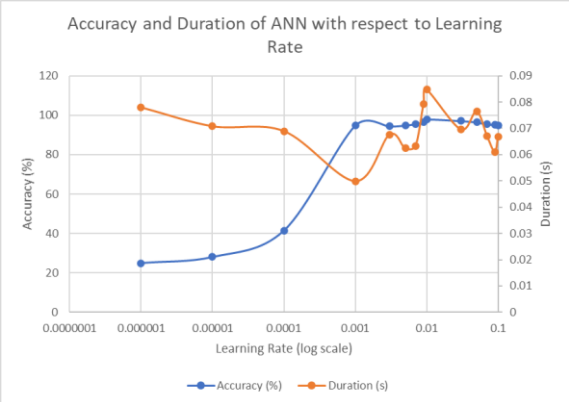
To the number of for-loops, we update variables layer by layer. Regardless of the epoch and mini-batch for-loop, we only have a few for-loop layers, and since the number of layers is quite small in this case, our MLP can run quickly. In each layer's for-loop, we use a formula to update variables (such as z and delta). So, it is important to make sure the matrix shapes can match (something that is important to keep in mind when debugging). In the formulas, we have dot- and Hadamard- product, so the matrix order cannot be changed. Usually, the shape of variables is unique in the runtime; like z, a and delta matrices have only one column (and thus are vectors), but for weights and biases, matrices can be very large. Therefore, we can predict the result of a formula before running it.

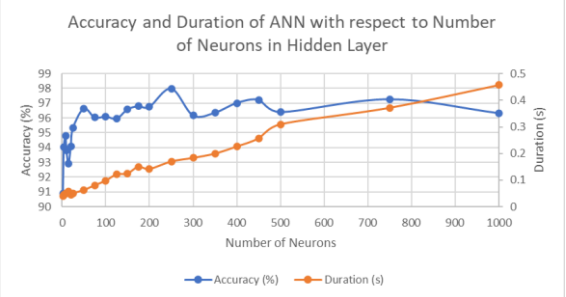
Before starting the experiments, we had to decide on benchmark values for the hyperparameters, leading to a default ANN defined as 2 hidden layers, 8 neurons, activation function of the hidden layers set as tanh, learning rate of 0.001, 10 epochs, and cross entropy for the loss function. When performing the experiments, we would alter one hyperparameter, whilst keeping the others as defined above, whilst splitting the test- and training-data by a ratio of 6:4.

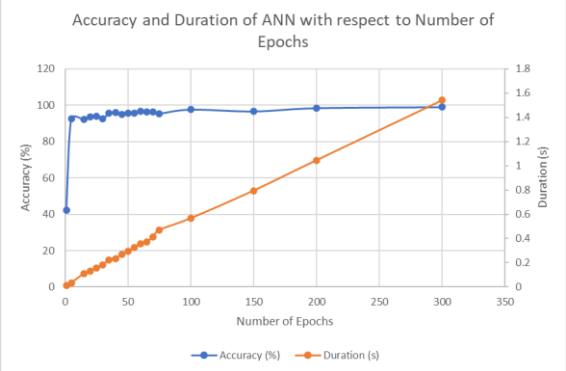
Results

Hyperparameter experimentation has led us to discover the following effects:

- Increasing Layers led to a steady decrease in accuracy, and a slight increase in duration.
- Neuron addition leads to a logarithmic-like increase in accuracy, and a linear increase in duration.
- Learning rate increase leads to a logarithmic-like increase in accuracy, and a fluctuation of duration around 0.7s (+/- 0.1s).







- Epoch increase leads to a sudden, rectifier-like increase in accuracy, whereupon it stays at a near-uniform level. Duration increases linearly.
- Leaky-ReLU provides the highest accuracy when applied as the activation function to the hidden layer, and sigmoid has the lowest. The quickest is ReLU, and Tanh is the slowest.
- Cross-Entropy has the highest accuracy and is the fastest, whilst MSE has the lowest accuracy, with a duration twice as much as CE.

We decided that further experimentation was necessary to find the combination of hyperparameters to give us the best accuracy with lowest duration. The hyperparameters chosen for further experimentation are those exhibiting log- or rectifier-like accuracy distributions, as they suggest an upper boundary that we could try reaching at relatively low performance settings.

The rest of the hyperparameters we chose to treat categorically, with the highest value exhibited used for further experimentation. Thus, we chose to experiment with 2-layered, Leaky ReLU activated, Cross-Entropy error MLPs, grid-searching around the rest of the hyperparameters. What resulted was a table with accuracy values alongside a table of corresponding duration values, with an excerpt shown below.

number of neurons		5	5	5			
learning rate		0.005	0.01	0.05			
epochs	4	28.9891	31.5301	46.9854	0.0315	0.03238	0.0284
	8	32.3953	33.6066	87.6503	0.05223	0.05438	0.09002
	12	32.3406	37.8597	89.2168	0.07828	0.06938	0.06954
	16	35.9745	44.5264	91.102	0.10203	0.09005	0.09053
	20	36.102	46.184	91.6029	0.1234	0.11848	0.11245
	24	37.1129	55.2641	92.3133	0.13475	0.1336	0.3464
	28	40.173	65.6557	93.051	0.21025	0.1939	0.15838
	32	42.0856	71.2022	92.8051	0.18657	0.17193	0.17243
	36	45.5373	80.2095	93.0419	0.20755	0.19385	0.19315
	40	51.1658	85.2459	93.4426	0.22682	0.2116	0.21245
	44	52.8597	86.694	93.4517	0.24525	0.235	0.23272
	48	53.133	90.929	93.2058	0.2639	0.2539	0.25915

This was in fact the general pattern when varying neuron count and learning rate; learning rate was most influential. We therefore chose to increase the learning rate to find a maximal accuracy value.

This led us to discover the combination for a 100% accuracy rate, a setting of 35 epochs, LR of 0.5, and 52 Neurons in the hidden layer.

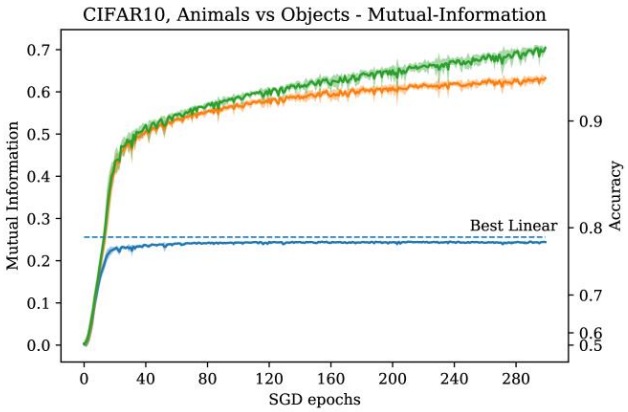
Discussion & Conclusions

We have found that finding the right balance between the hyperparameters is a game of compromise, requiring fine-tuning to achieve the near-perfect setting (Vasilev et al., 2019).

This is very dependent on the dataset itself, whose nature will dictate the structure of the MLP and the subsequent hyperparameters.

The program required a high learning to achieve a relatively high accuracy is because of the low number of features in the dataset. Generally, a learning rate of 0.0001 and 0.01 is acceptable, however too small a rate creates too minimal corrections for the algorithm to learn at a high enough rate (Bonaccorso, 2018).

The shape of the epoch experiment graph corresponds to those researched online, such as experiments on epoch variation on the CIFAR10 Animals vs Objects dataset, as shown on the right.



(NAKKIRAN ET AL., 2019)

Leaky ReLU resulted in the highest accuracy out of the 4 activation functions tested is not a surprise; it is meant to be an improvement on ReLU, which has the innate problem of potentially creating dead neurons, where there is no improvement due to no gradient (Nakkiran et al., 2019).

Cross-entropy is innately more suitable for classification problems; it is expected to have a higher accuracy rating as opposed to MSE, which is better suited for regression-related work (as it does not guarantee a minimisation of the error function over epochs) (Kumar, 2021).

The decrease of accuracy with greater number of layers is not surprising; the program was most likely suffering from overfitting, implying a simpler ANN structure is more optimal for the dataset.

In conclusion, our experimentation has provided results that are like that of other experiments found in wider literature. One thing we could not experiment on due to the nature of the coursework is the effects of changing the number of input neurons; having the opportunity to do so would potentially lead to different results, as well as lead to multiple optimal ANN configurations, since we noted that the learning rate is related to the size of the dataset worked on by the program.

References

Vasilev, I., Slater, D., Spacagna, G., Roelants, P. and Zocca, V., 2019. *Python Deep Learning - Second Edition*. 2nd ed. Birmingham: Packt Publishing, p.279.

Bonaccorso, G., 2018. *Mastering Machine Learning Algorithms*. Birmingham: Packt Publishing, p.338.

Nakkiran, P., Kaplun, G., Kalimeris, D., Yang, T., Edelman, B., Zhang, F. and Barak, B., 2019. *SGD on Neural Networks Learns Functions of Increasing Complexity*. [online] DeepAI. Available at: <https://deepai.org/publication/sgd-on-neural-networks-learns-functions-of-increasing-complexity> [Accessed 25 October 2021].

Kumar, A., 2021. *Why using Mean Squared Error(MSE) cost function for Binary Classification is a bad idea?*. [online] Data Analytics. Available at: <https://towardsdatascience.com/why-using-mean-squared-error-mse-cost-function-for-binary-classification-is-a-bad-idea-933089e90df7> [Accessed 25 October 2021].