190004947
Computer Science
University of St. Andrews

CS3105 Practical 2 Report

Neural Networks

Part 1:

I finished implementing the feed-forward Neural Network using MINET and passed all three stacscheck tests.

The output dimensions were set to 10 to represent each of the digits that the network will need to recognise. The parameters for the linear layer are found in the Linear.java file in the constructor; these parameters are input dimensions, output dimensions, and a weight initialisation object (WeightInitXavier in this scenario). After the second linear layer in the network is the softmax object. The softmax function turns a vector of N real values into a vector of N real values that sum to 1. The appropriate loss function is the cross entropy loss function which is good for classification problems because it minimises the distance between two probability distributions; predicted and actual.

To count how many predictions are correct in the eval function, I looped through each element in the DoubleMatrix of the actual values and counted how many values in the prediction matrix matched those in the actual matrix. To compute accuracy simply divide the resulting number of correct predictions by the number of predictions there are.

In training the network, the loss value is stored in a double object and is calculated by calling loss.forward() on the matrix of input vectors. The early stopping criteria for training is when the network is fitting the training data too well and losing generalisation. This is determined as when the accuracy on a separate set of data goes down or stays the same a certain number of epochs in a row. This number is passed in as a variable called ‘patience’. The test accuracy is calculated by calling eval on the test set.

Part 2:

I finished implementing the two additional activation functions, ReLU and TanH, and passed all stacscheck tests.

Towards the start of the main function I wrote a simple if else statement to check whether the extra argument was 'TanH' or 'ReLU'. This would then set the actFunc variable to a new TanH or ReLU object respectively. As a starting place for building out these two classes for activation functions, I duplicated the Sigmoid class and changed the logic in the forward and backward propagation functions. In forward for TanH, I simply returned the output of MatrixFunctions.tanh(X) and set the attribute Y to this new value. Then in backward I return the product of gY and the derivative of the TanH function using the Y attribute. In the ReLU class, for the forward function I set Y as a matrix of zeroes then iterated through every element in the X matrix and, if the element was greater than zero then it would be updated in Y to match that value. The backwards function is similar, it creates a matrix of zeros of the same dimensions as the Y matrix. Then at each space in the Y matrix, if the value is greater than 0 then the gradient is 1 (linear) and the output matrix is updated to represent this. The matrix product of the output matrix and gY is returned.

When the Sequential object is instantiated, the second layer is set as the actFunc variable which will be whichever activation function that was passed in through the terminal.

Part 3:

I investigated the impact of the hyper-parameters: activation functions, mini-batch sizes, learning rates, and as an extra hyper-parameter I chose the loss function.

To see how each of these would affect the performance of the neural network I created four arrays: activationFunctions, batchSizes, learningRates, and lossFunctions. Each of them were filled with some options for each. I then wrote four for loops nested within each other and called train() with parameters that are determined by which iteration the program is in for each hyper-parameter. At each iteration, the parameters and the test accuracy will be printed out. I left this loop to run overnight and woke up to test accuracies for the network as trained with every permutation of the hyper-parameters defined in the arrays. This data has been put into a table below.

Activation Function	**Batch Size**	**Learning Rate**	**Loss Function**	**Test Accuracy**
ReLU	500	0.25	Cross Entropy	0.9800
ReLU	500	0.25	MeanSquareError	0.9802
ReLU	500	0.5	CrossEntropy	0.9808
ReLU	500	0.5	MeanSquareError	0.9811
ReLU	500	0.75	CrossEntropy	0.9813
ReLU	500	0.75	Mean Square Error	0.9814
ReLU	500	1.0	Cross Entropy	0.9815
ReLU	500	1.0	Mean Square Error	0.9817
ReLU	1000	0.25	Cross Entropy	0.9812
ReLU	1000	0.25	Mean Square Error	0.9814
ReLU	1000	0.5	Cross Entropy	0.9811
ReLU	1000	0.5	Mean Square Error	0.9814
ReLU	1000	0.75	Cross Entropy	0.9808
ReLU	1000	0.75	Mean Square Error	0.9814
ReLU	1000	1.0	Cross Entropy	0.9813
ReLU	1000	1.0	Mean Square Error	0.9815
ReLU	1500	0.25	Cross Entropy	0.9811
ReLU	1500	0.25	Mean Square Error	0.9812
ReLU	1500	0.5	Cross Entropy	0.9811
ReLU	1500	0.5	Mean Square Error	0.9814
ReLU	1500	0.75	Cross Entropy	0.9814
ReLU	1500	0.75	Mean Square Error	0.9814
ReLU	1500	1.0	Cross Entropy	0.9811
ReLU	1500	1.0	Mean Square Error	0.9813
TanH	500	0.25	Cross Entropy	0.9728
TanH	500	0.25	Mean Square Error	0.9741
TanH	500	0.5	Cross Entropy	0.9776

TanH	500	0.5	Mean Square Error	0.9787
TanH	500	0.75	Cross Entropy	0.9792
TanH	500	0.75	Mean Square Error	0.9791
TanH	500	1.0	Cross Entropy	0.2973
TanH	500	1.0	Mean Square Error	0.2967
TanH	1000	0.25	Cross Entropy	0.2976
TanH	1000	0.25	Mean Square Error	0.2868
TanH	1000	0.5	Cross Entropy	0.4897
TanH	1000	0.5	Mean Square Error	0.4807
TanH	1000	0.75	Cross Entropy	0.8868
TanH	1000	0.75	Mean Square Error	0.8854
TanH	1000	1.0	Cross Entropy	0.8868
TanH	1000	1.0	Mean Square Error	0.8841
TanH	1500	0.25	Cross Entropy	0.8878
TanH	1500	0.25	Mean Square Error	0.8864
TanH	1500	0.5	Cross Entropy	0.8875
TanH	1500	0.5	Mean Square Error	0.8861
TanH	1500	0.75	Cross Entropy	0.8875
TanH	1500	0.75	Mean Square Error	0.8868
TanH	1500	1.0	Cross Entropy	0.8881
TanH	1500	1.0	Mean Square Error	0.8788

Average accuracy using ReLU activation function:	0.981171
Average accuracy using TanH activation function:	0.736233
Average accuracy using 500 batch size:	0.893969
Average accuracy using 1000 batch size:	0.809250
Average accuracy using 1500 batch size:	0.933688
Average accuracy using 0.25 learning rate:	0.849217
Average accuracy using 0.5 learning rate:	0.882232
Average accuracy using 0.75 learning rate:	0.949375
Average accuracy using 1.0 learning rate:	0.835017
Average accuracy using cross entropy loss function:	0.879600
Average accuracy using mean square error loss function:	0.878308

The highest recorded accuracy was 0.9817 and was attained using ReLU as activation function, 500 as batch size, 1.0 as learning rate, and mean square error as the loss function.

Using ReLU as the activation function resulted in a much higher average accuracy than using TanH as the activation function. However, the ReLU function can 'kill' neurones. This is when weights in the network lead to negative inputs and the ReLU function gets stuck outputting zeroes. As much as 40% of the network could be 'dead' neurones at the ned of training.

The mini-batch size with the highest average accuracy was the highest tested size, 1500, which is interesting because when the highest accuracy was recorded, a mini-batch size of 500 was used.

I tested 4 different learning rates and the highest average turned out to be at 0.75. The accuracy increased with the learning rate up until 0.75 at which point the average accuracy started to decline. The learning rate controls how quickly the model ups adapted to the problem. Lower learning rates will lead to a longer but more careful training process. A learning rate that is too high can cause the model to converge too quickly to a suboptimal solution whereas the training process can simply get stuck if the learning rate is too lower.

There was very little difference between using the cross entropy loss function and the mean squared error loss function. However, the highest recorded accuracy used the mean square error loss function.