**Assignment 2**

Dean Magee (357496)

Introduction

This assignment centers on the construction of a multi layered neural network to classify handwritten digits. The network was trained and evaluated on the famous MNIST dataset with each label being a handwritten digit between zero and nine and each attribute capturing the intensity of a pixel of an image of a digit. For each label there were 25 attributes (25 pixel intensities) so the neural networks presented here have 25 input nodes. There were 10 output nodes that each took on a value between zero and one and each referred to a specific digit. The nodes in the final layer outputted a probability that the inputs represented a specific digit and classification was made by selecting the node with the highest probability. There are a number of problems with neural networks, some of which include overfitting, selecting the correct parameters for the model, training time and failing to reach a global minimum during gradient descent. This report will detail the construction of the neural networks required as well as go into some detail as to how these problems were overcome.

Stage 1:

The first stage involved the construction of a neural network with a single hidden layer with 100 nodes. There were a few restrictions placed on this network.

They were:

* A single layer of 100 nodes must be used
* The output layer needs to have 10 nodes
* The hyperbolic tangent function is to be used as the activation function

My background is in economics and as such my programming ability is significantly lower than the rest of the COMP 900051 cohort. I started by learning how the gradient descent algorithm worked by implementing it with a logistic regression. From here I decided to hard code a non-vectorised version of a simple neural network with 20 nodes in the single hidden layer. This implementation had a few problems. Firstly, the use of for-loops within for-loops resulted in substantial training times. Secondly, the use of regular stochastic gradient descent (gradient descent performed after passing a single example through the network) resulted in slow and imprecise convergence towards the minimum.

Using the lessons learnt here I built the required neural network where forward propagation and gradient descent were vectorised. I also implemented mini-batch gradient descent. This solved two problems. Firstly, the time taken to converge to a reasonable solution decreased, and secondly, convergence was smoother when using regular stochastic gradient descent.

Out of the 60,000 training examples that were available 10,000 were set aside as an evaluation set and 50,000 were used to train the algorithm. The file ‘neuralnet.m’ contained a function that took ‘train\_data, test\_data and structure’ as inputs and outputted predictions on the evaluation set. Train\_data and test\_data were created by running the script ‘samplesplit.m’ on the data provided. After each feed-forward iteration the outputs of the neural network were checked on the mini batch that had been just used as inputs (neuralnet.m line 54-70). The loss between the two was stored in a vector and plotted when all iterations had completed. Also, after every 100 updates to the weights and biases, the neural network was run on the evaluation set and the loss was also recorded and plotted. By looking at these two graphs I could determine whether the algorithm was converging and whether overfitting was occurring. I noticed that, even with mini batch gradient descent, convergence was not smooth at higher numbers of iterations and as a result I added a simple decay to the learning rate that changed with every iteration (see neuralnet.m - line 93). This made convergence much smoother at higher the iterations. This proved to be unnecessary later on however, as I changed the initial random values for the weights to be between -0.01 and 0.01 and this seemed to solve the problem while not requiring more iterations.

Interestingly a neural network with 100 nodes and one hidden layer did not show signs of overfitting, even at 15,000 iterations (see image 1A). If anything it looked like more iterations would lead to a better solution. It was decided that 20,000 iterations would be tried. At this level of iterations it did not look like convergence would be reached so instead of increasing the iterations it was decided that the learning rate would be increased. This, however, decreased the accuracy. In order to fix this the mini batch size was increased in the hopes that more accurate steps would allow the network to converge to the global minimum.

Best result: 81.71 % (0.0001 LR, 2000 section size, 15000 iterations).

Stage 2:

The use of a genetic algorithm to find an optimal structure for a neural network was the goal of stage 2. Here the requirements were relaxed. Up to 10 layers were allowed with each layer allowed a maximum of 100 nodes. At this point I was running short on time and a simple but effective solution needed to be developed. My genetic algorithm was designed as follows:

1. Generate an initial population. The number of different neural networks (from here referred to as individuals) can be chosen as can the number of hidden layers present in the individuals.
2. Loop:
   1. Train all the individuals on the train set.
   2. Check the ‘fitness’ of the individuals by testing them on the evaluation set.
   3. Rank the individuals according to fitness.
   4. Remove the individuals with the lowest level of fitness.
   5. Replace the removed individuals with ‘children’. Each child is a combination of two randomly chosen survivors using single point crossover.

Single point crossover is a simple process. For example: we take two individuals a=[25, 30, 40, 50, 60, 10] and b=[25, 50, 90, 10, 60, 10], where 25 are the input layer nodes and 10 are the output layer nodes, and apply single point crossover at the 3rd layer we will get c=[25, 30, 40, 10, 60, 10]. The location of the single point crossover is chosen randomly. This process is used to produce the children. Also, part 2a is parallelized in the algorithm (see genetic\_algo.m – line 42-49) using the farfor command. This significantly speeds up the process.

A few adjustments needed to be made to the neural network algorithm in order to reach an optimal solution. The use of multiple layers brings about the problem of overfitting. To combat this a regularization term was added to the loss function that penalizes large weights. This means that the updates to the weights of each node will be adjusted as follows:

Where

Due to a lack of time and computing resources the neural networks that were used in the genetic algorithm were limited to 5000 mini batch iterations. This forced an assumption to be made in that the algorithms that performed well over this limited range of iterations were likely to perform well over much more iterations.

Conclusion

This assignment took an incredibly long time to complete. As a result a full range of initial input parameters could not be tried out but this further reinforced one of the problems with neural networks. It is very difficult to find the optimal parameters to input into the algorithm. I had to choose the mini batch size, how the initial weights and biases were distributed, the step size, the step size decay, the lambda (for weight regularization) and the number of iterations to perform before stopping the back-propagation algorithm. The other problem was the lack of high performance computing equipment such as GPU’s capable of implementing neural networks (most laptop GPU’s are not suitable). This negatively affected my ability to quickly discover optimal parameters and algorithm designs. In the end the best prediction rate that could be achieved on the evaluation set for stage 1 was 81.71 %.