CS5011 A1 Report

Jessica Cooper
170021928
jmc31@st-andrews.ac.uk

Parts Implemented/Running/Usage:

To build and train the network, then print the inputs, actual outputs and target outputs:

```
java -jar Learning1.jar
```

I also implemented a brute force optimiser to find the most effective learning rate, momentum and number of hidden nodes. To see this in action:

```
java -jar Learning1.jar o
```

To build and train the network with your own learning rate, momentum and number of hidden nodes, and print the outcome:

```
java -jar Learning1.jar [int hiddenNodes] [double learningRate]
[double momentum]
```

To play the game:

```
java -jar Learning2.jar
```

To play the game with Early Guess enabled:

```
java -jar Learning2.jar g
```

To play the game with Maybe implemented:

```
java -jar Learning3.jar
```

To play the game with Maybe implemented and Early Guess enabled:

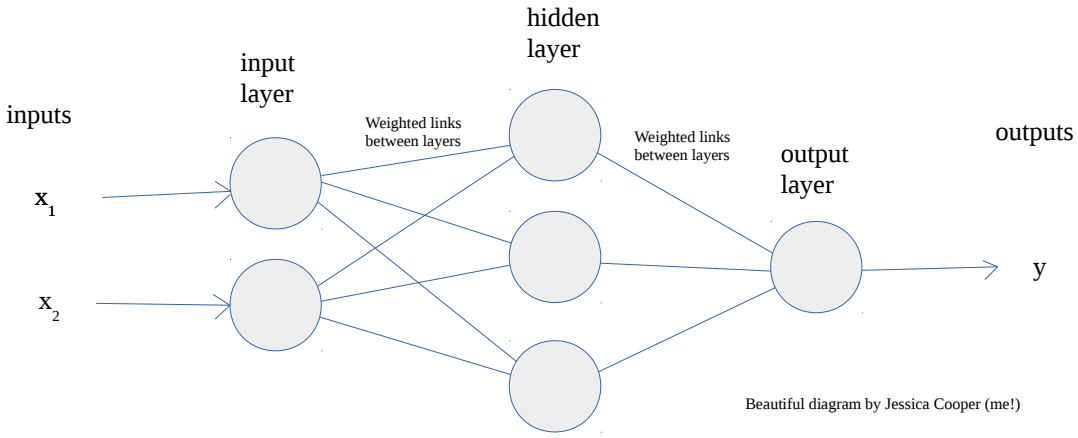
```
java -jar Learning3.jar g
```

Neural Network Review

What is a feed-forward neural network?

A neural network consists of layers of nodes or units, typically an input layer, an output layer, and a number of hidden layers between. Each of these layers consists of a number of nodes, which are linked by weighted connections. Neural networks work by taking a number of inputs (one for each input node), and passing them through the network, through the hidden layer(s), to the output nodes. Nodes in the hidden and output layers contain an activation function – in this task I have used the logistic function, but there are others – which takes the weighted sum of the outputs of the nodes in the previous layer and feeds that value through to the next layer (or the output).

A simple neural network



Neural networks like this are trained by using a training table containing a range of input data and the known correct output of that data. The input data is passed through the network and the actual output is compared with the known correct output. The difference between the two is known as the error. The training can be accomplished in different ways, but in this assignment I have used backpropagation, a method by which the error is passed backwards through the network, and the weights of the links between the nodes are adjusted relative to this error. By repeating this process until the error reaches an acceptably low percentage, the network becomes good at producing the expected output.

We can adjust this training process by changing some parameters – namely the learning rate, the momentum and the number of hidden nodes. The learning rate is simply the rate at which the link weights are adjusted per epoch, the momentum adjusts the learning rate to avoid local minimums, and the number of hidden nodes is self explanatory, and important to make sure that there are enough nodes to learn the data, but not too many that there isn't enough data to for them all to learn.

Neural networks are used today for a wide variety of tasks such as financial market prediction and image recognition and have proven to be a very useful tool, especially when there is a large amount of cheap and reliable training data available. However, neural networks are not suitable for all learning tasks – their black box nature makes them vulnerable to unknown biases in the input data, and of course if the data is sparse, expensive, too noisy or difficult to encode they are not the right choice. [NN L3 Lecture slides https://studres.cs.st-andrews.ac.uk/CS5010/Lectures/NN-L03-slides.pdf Accessed 3/10/17]

Part 1

Input & Output Coding

One-hot encoding the seemed like an obvious choice given the format of the data. It was fairly trivial for me to replace the ‘Yes’s and ‘No’s in the training data with ‘1.0’ and ‘0.0’ with a few simple search and replaces within the CSV file. I also added the requisite commas and braces to format the code correctly at this stage. I was then able to copy and paste the data directly into my code from the CSV file. Likewise for the outputs, I simply separated that column and replaced every instance of ‘Alex’ with { 1.0 , 0.0 , 0.0 , 0.0 , 0.0 }, then every instance of ‘Alfred’ with { 0.0 , 1.0 , 0.0 , 0.0 , 0.0 } and so on.

This would not be practical with larger and more complex datasets, and it’s not the most elegant way of doing things – a neater alternative would be to create a method in java to read in a CSV file and do the encoding within the program. However, given the small size of the dataset and the focus of this assignment on neural networks (rather than file I/O in Java) I felt this way was more efficient as it took only a couple of minutes to do manually and enabled me to get started on the neural network code straight away.

Training Table

I used two 2d arrays to contain my training data, thus:

Training input:
(Corresponding to "Curly Hair" , "Blonde" , "Red Cheeks" , "Moustache" , "Beard" , "Earrings" , "Female",)

```
{
{ 0.0 , 1.0 , 0.0 , 1.0 , 0.0 , 0.0 , 0.0 ,},
{ 0.0 , 0.0 , 0.0 , 0.0 , 1.0 , 0.0 , 0.0 ,},
{ 0.0 , 1.0 , 1.0 , 0.0 , 0.0 , 0.0 , 1.0 ,},
...
```

Expected outputs:
(One-hot encoding corresponding to "Alex" , "Alfred" , "Anita" , "Anne" , "Bernard")

```
{
{ 1.0 , 0.0 , 0.0 , 0.0 , 0.0 },
{ 0.0 , 1.0 , 0.0 , 0.0 , 0.0 },
{ 0.0 , 0.0 , 1.0 , 0.0 , 0.0 },
...
```

Constructing the Network

Using encog I constructed a simple neural network with an input layer, a hidden layer and an output layer. The input layer has 7 nodes (one for each feature) and the output layer has 5 (one for each character).

Initially I set the number of hidden nodes to 6, using the rules of thumb covered in class. (A good number of hidden nodes to start with could be either: 2x input, between input and output, or 2/3 of input+output)

The hidden and output layers use a sigmoidal function (specifically the logistic function) y = 1/(1+e^-x) which outputs a value between 0 and 1.

Training the Network

Using encog's methods I finalised and trained the network, looping through the training process multiple times until the error (the difference between the target/expected output and the actual output) was less than 1%. I initially set the learning rate to 0.5 and the momentum to 0 – these were arbitrary choices as I intended to optimise them later.

Computing and Comparing

To see what the network was actually achieving, I compared the inputs, target outputs and actual outputs by looping through them for each training pair and displaying them in the console thus:

```
Input=[BasicMLData:0.0,0.0,1.0,0.0,0.0,1.0,0.0]
Actual Output=0.012543867544628419 0.09172469475678152
0.029913408564846 0.024743259206445826 0.9091745631777496
Target Output=[BasicMLData:0.0,0.0,0.0,0.0,1.0]
```

You can see here that the index of the largest output (which will become the character guess) matches the index of the 1.0 in the target output (which corresponds to the correct guess for the inputs given), which shows that the network is working as expected.

Optimisation

With the initial arbitrary settings of learning rate: 0.5, momentum: 0.0 and number of hidden nodes: 6, the network was converging (reaching an acceptably low error) in between 30 and 50 epochs. I wanted to see if I could improve on this. To make it easier to try lots of different options, I added functionality to take the 3 arguments (learning rate, momentum and no. of hidden nodes) at the command line, and to print out the number of epochs once the training iterations had finished.

Using this functionality I tried a few different combinations – increasing the learning rate to 0.8 saw a great improvement, with the error threshold being reached mostly within 30 epochs. A learning rate of 1 was even better at around 20 epochs, but with occasional wild deviations up to 100 epochs or more. I also tried adjusting the number of hidden nodes and the momentum using heuristics, but it struck me as a very inefficient way of trying to find the best parameters for the most efficient training set up. So, I wrote a method to try by brute force every possible combination between reasonable parameters and print out the best possible ones:

```
New best epoch count (averaged over 10 trainings): 22.9 with
hiddenNodes: 4 learningRate: 1.0 momentum: 0.3
.....................................................................
..
New best epoch count (averaged over 10 trainings): 22.5 with
hiddenNodes: 5 learningRate: 0.6 momentum: 0.5
.....................................................................
.................................
New best epoch count (averaged over 10 trainings): 21.8 with
hiddenNodes: 6 learningRate: 0.5 momentum: 0.6
```

As the exact number of epochs required will vary each time depending on the random starting weights, I decided to train the network 100 times with each possible set of parameters, and then take the average of the epoch. This produced fairly consistent results:

```
Optimum inputs are: hiddenNodes: 7 learningRate: 0.4 momentum: 0.6
with an average of 24.06 epochs to train.

Optimum inputs are: hiddenNodes: 6 learningRate: 0.5 momentum: 0.5
with an average of 25.2 epochs to train.

Optimum inputs are: hiddenNodes: 7 learningRate: 0.4 momentum: 0.6
with an average of 24.3 epochs to train.
```

I updated the parameters in the code to reflect these more efficient parameters. I wondered if 100 training episodes might not be overkill – could we get some useful parameters more quickly? Using an average of just 10 epochs produced these results:

```
Optimum inputs are: hiddenNodes: 5 learningRate: 0.6 momentum: 0.6
with an average of 22.1 epochs to train.

Optimum inputs are: hiddenNodes: 7 learningRate: 0.4 momentum: 0.7
with an average of 20.7 epochs to train.

Optimum inputs are: hiddenNodes: 5 learningRate: 0.7 momentum: 0.5
with an average of 22.1 epochs to train.
```

Which, while less consistent, are really not so different and much quicker to generate. For this reason the optimiser is implemented to average over 10 training sessions rather than 100.

Part 2

```
Think of a character:

Alex is blonde and has a beard and moustache.
Alfred has a beard, earrings and a hat.
```

```
Anita is blonde, female and has red cheeks.
Anne is female and has curly hair, earrings and a hat.
Bernard has red cheeks and earrings.
Tom has a beard and a hat.

Does your character have curly hair? Please answer y or n: n

Does your character have blonde hair? Please answer y or n: y

Does your character have red cheeks? Please answer y or n: n

Does your character have a moustache? Please answer y or n: y

I think your character is Alex

Answer y to play again or any character to quit:
```

Designing the Game System

I used a scanner to get input from the user, which I stored in an array. I then used the network to compute using that array as input. The network produces an output array, each element of which corresponds to the likelihood that the character associated with that index is the one that the user is thinking of. So, all I needed to do there was find the largest element in the output array and use the index of that to get the character's name from the associated array of names. If the user enters a pattern that does not exist in the training set, the network will simply provide its closest guess based on the input provided.

Early Guess

In order to guess early, we'll have to use the network to compute an output after each user input, using only the partial data available. To do this, I declared an array of doubles to store the user's input, initialising each element to 0.5. I chose 0.5 as it's mid-way between 0 and 1, and thus would not give incorrect probabilities to the wrong characters when the network computes the guess after each user input. (Because up until the last user input, the network is computing with only partial information, so the rest of the array is filled with these initial values. By setting them to 0.5 I am attempting to keep them neutral and yet allow the network to update in the presence of new information form the user.)

Next to be decided was at what threshold of certainty to allow the network to guess early. I made this a bit easier to implement by adding some print statements so I could see what was happening after each input, thus:

```
Does your character have curly hair? Please answer y or n: y
current guess: Alex: 0.010255429423329946 Alfred:
0.005907644294439974 Anita: 0.360270141370462 Anne:
0.5829334467859403 Bernard: 0.006115453747348067
```

After some trial and error, I settled on a value of 0.85 as a fairly good threshold for certainty. That is, as soon as the network outputs a value of above 0.85 in any position in the output array, the program will make a guess for the associated character. I ran into a problem with the network being slightly too enthusiastic after the first user input and guessing too soon, often incorrectly – I have resolved this by adding the caveat that the network must take at least 2 user inputs before guessing. This seems to offer a good balance, although the accuracy of the guess can be a bit erratic – the network is not really trained to give correct outputs with this kind of input, although on the whole it seems to be successful. Each time the program is run, you might see different levels of success with the Early Guess feature, as the network is created and trained anew. It could be made more accurate and reliable by including partial data in the training table.

Adding a New Character & Feature

I extended the training table by adding a hat as the new feature, and a new character named Tom, who has a beard and a hat. I added a few random mistakes in this new data, to match the quality of the existing dataset. To decide on the new parameters I ran the optimiser (as in part 1) which provided the following results:

```
Optimum settings are: hiddenNodes: 6 learningRate: 0.9 momentum:
0.1 with an average of 34.4 epochs to train.

Optimum settings are: hiddenNodes: 6 learningRate: 0.8 momentum:
0.0 with an average of 35.8 epochs to train.

Optimum settings are: hiddenNodes: 6 learningRate: 0.3 momentum:
0.7 with an average of 34.0 epochs to train.
```

Interestingly, the output of the optimiser was much more varied this time than in part 1. I ran it many times and the optimal learning rate and momentum fluctuated wildly – however, the optimum number of hidden nodes stayed constant at 6. In the end I decided to use a rough average, (6 hidden nodes, a learning rate of 0.7 and a momentum of 0.2) which appears to work well. I then removed the optimiser from part 2 to make the code more concise, since its not necessary to play the game, and its functionality for demonstration purposes is no different than in part 1.

Part 3

```
Does your character have curly hair? Please answer y, n or m
(maybe): n

Does your character have blonde hair? Please answer y, n or m
(maybe): m

Does your character have red cheeks? Please answer y, n or m
(maybe): y

Does your character have a moustache? Please answer y, n or m
(maybe): n
```

```
Does your character have a beard? Please answer y, n or m (maybe):
n

Does your character have earrings? Please answer y, n or m
(maybe): m

Does your character have female gender? Please answer y, n or m
(maybe): y

I guess your character is Anita
```

Maybe

Had I more time I would have liked to implement every addition in part 3! Unfortunately I do not, so I have chosen to implement just the 'maybe' option. This simply allows the user to input 'm' for maybe as a response, in addition to 'y' yes and 'n' no. Because no new information has really been provided, when the user inputs ‘m’ I simply move to the next index in the user input array, leaving the previous element at the default neutral setting of 0.5.

Evaluation

I am pleased with my implementation of this task. Due to my optimiser method, the network is trained extremely efficiently, and guesses very accurately when the standard version of the game is played. I am pleased to offer some alternate usage options for the user in part 1, so that they are able to set up and train the network with their own parameters, and view the number of epochs it took to complete the training. I feel that these, along with the option to run the optimiser and see the different combinations of parameters and their efficiency, are interesting additions.

The early guess and maybe additions proved difficult to get working consistently, I believe due to the small size of the training data and the fact that the network was not trained to handle uncertainty. I would solve this by increasing the amount of training data and including ‘maybe’s in it – this would enable the network to be properly trained to handle errors in input, incomplete user input or ‘maybe’ inputs, and thereby make more accurate guesses earlier.

Possible Improvements

- Read in a CSV file to get the training table. Whilst not necessary to complete this task successfully, this would be a more elegant method of getting data into the program, and provide an easier way to update the training data.
- I would like to do something along the lines of getting feedback from the user after a guess is made, and then using that feedback to retrain the network after each game. That way, I could remove the 'only guess after two inputs' constraint and allow the network to make mistakes, because those mistakes would go on to improve the network and allow it to make more accurate, early guesses in future.
- Increasing the amount of training data significantly would would be a great improvement, as it would not only increase the noise tolerance of the network, but would also make it possible to split the data into a training set and a test set for more accurate training.

Wordcount: 3013

Bibliography

NN L3 Lecture slides https://studres.cs.st-andrews.ac.uk/CS5010/Lectures/NN-L03-slides.pdf Accessed 3/10/17

L4 W2 Lecture slides: https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L4_w2.pdf Accessed 29/09/17

Programming Neural Networks with Encog3 in Java, Jeff Heaton, 2011. https://s3.amazonaws.com/heatonresearch-books/free/Encog3Java-User.pdf Accessed 29/09/17

HelloWorld.java, Jeff Heaton, https://github.com/encog/encog-sample-java/blob/master/src/main/java/HelloWorld.java Accessed 30/09/17

Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.