COURSEWORK 2

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Machine Learning

Group 38:
Vinamra Agrawal (va1215)
Rishabh Jain (rj2315)
Alexandru Toma (ait15)
Maurizio Zen (mz4715)

Date: March 7, 2018

1 Introduction

Artificial Neural Networks, or simply Neural Networks, offer a practical method for learning from examples. Neural networks learning methods provide a robust approach to approximating real-valued, discrete-valued and vector-valued target functions. The architecture of a neural network can be summarised as follows. It consists of an input layer and an output layer, each layer having a number of units called neurons. Usually, between the input and output layers there are a number of hidden layers where intermediate computations are performed. As part of the architecture of the neural network, activation functions are used to transfer the output of one neuron to the input of another neuron. This connection between neurons is defined by the weight attributed to each neuron.
There are many types of neural networks, and this report covers a Feedforward Neural Network in which connections between the neurons do not form a cycle. That is, the information flows in one direction only: from the input neurons through the hidden neurons and, finally, to the output units.
The aim of this project is to train a neural network which addresses facial emotion recognition. We document our approaches to training the neural network and adjusting various parameters, as well as providing results and commenting on the changes generated by these attempts.

2 Part I: Data description

We are given two pieces of data: a cell array called `datasetInputs` containing input images, and another cell array called `datasetTargets` consisting of the targets corresponding to the input images. The images have been downscaled to 30 by 30 and are also greyscaled thus, each of them correspond to 900 pixels. The target data represents the emotion corresponding to each image via a 7 element binary array with the index of the emotion set to 1 (and rest set to 0). The 7 emotions in order are - angry, disgust, fear, happy, sad, surprise and neutral. Both cell arrays are divided into three different cells containing training data, test data, and validation data, respectively.

3 Part II: Creating and Training Neural Network

3.1 Keras

We used `Keras` as our API to build the neural network. The Python library is built on top of `Tensorflow` which we used for numerical computations to compare different architectures of the network. It also includes `TensorBoard` which enabled us to visualize metrics (like accuracy and loss) of networks while they are trained and compare their performance with other architectures simlutaneously. Being in python, `Keras` helped us with fast prototyping. It is also optimised to run seamlessly on CPUs, thus making the training/testing cycle simpler and quicker for us.

3.2 Normalisation

Image preprocessing techniques can have a major impact on the performance and robustness of the facial recognition system. The main objective of these techniques is to enhance the discriminative information contained in the facial images.

Firstly, we employ histogram equalization one of the most commonly used normalisation approach in face recognition. Here, the pixel intensity values are mapped from their original distribution to a uniform one, thus improving contrast and simultaneously compensating for the illumination-introduced variations in the appearance of the facial images.

Then we further mean normalised the images. We find the normalised value by subtracting the mean, and then scale each image down by dividing it with its mean.

4 Part III: Parameter Optimisation

4.1 Initial Architecture

Activation Function: Activation functions are essential for a Neural Network to learn. The most important attribution of activation functions is to calculate the weighted sum and to decide whether to activate (fire) a particular neuron. There are several examples of activation functions used for the hidden layer, as well as various combinations of these functions. For our initial architecture we decided to go with `ReLU` since it is one of the most widely used activation functions. `ReLU` is also faster than `sigmoid` and `tanh` since it carries out simpler mathematical operations. The main advantage of ReLU over the other activation functions is that it does not activate all the neurons at the same time, leading to a sparse network and more efficient computations. Another positive aspect is that when applying `ReLU` on positive input, the gradient is larger than the gradient for `sigmoid` or `tanh`. Considering all this, we opted for ReLU to be the hidden activation function.

Number of Neurons/Layer: We decided to go with 3 hidden layers with 900, 300 and 100 neurons respectively. Starting with 900 neurons (in the first hidden layer) we decrease the number of neurons in consecutive layers by a factor of 3. This seemed to be a good standard approach to start with when using feed forward networks.

Weight Initialisation: We went with the default values: `glorot_uniform` It draws samples from a uniform distribution within `[-limit, limit]` where limit is $sqrt\frac{6}{fan\_in+fan\_out}$ where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.

Stopping Criteria: Early stopping with 80 maximum epochs. Where the validation accuracy should increase by at least 1% over at most 10 epochs.

Learning Rate: Default recommended rate: 0.01 (constant)

Momentum: Default recommended momentum: 0.5 (constant). We will discuss more about variable momentum in a later part when we introduce a learning rate update schedule.

Loss Function We use Categorical Cross Entropy as our loss function. Cross entropy is the average number of bits we'll need if we encode one symbol to another.

With these initial parameters, we got the following graph for the neural network (the graphs have been smoothed by a factor of 0.6 on tensorboard).

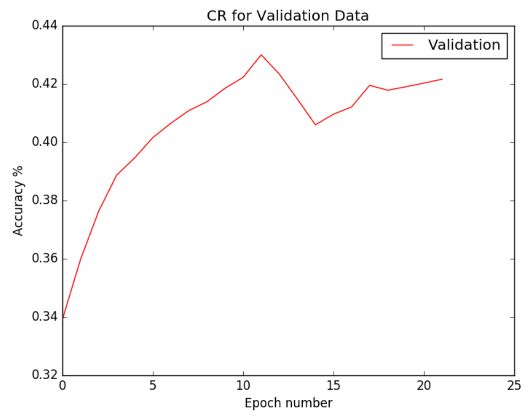

Figure 1: Validation Accuracy

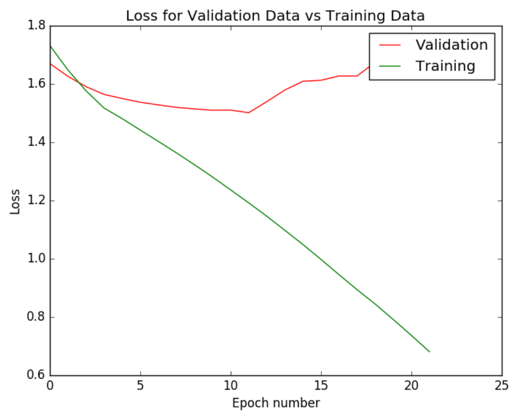

Figure 2: Validation Loss Comparison

4.2 Learning Rate

Learning rate determines the step size in the direction of steepest decrease towards minimising training error rate. Choosing extremely high learning rate will train the model quickly but on the other hand, might lead to over-train on the training set and have high validation loss error. On the other hand if the learning rate is too small, it might take too long to train the model and we might stop before the optimal value is reached because of the stopping criteria.

The following graph and table show the examples of learning rate we tried and their outcomes. Note: Accuracy = 1 - classification error.

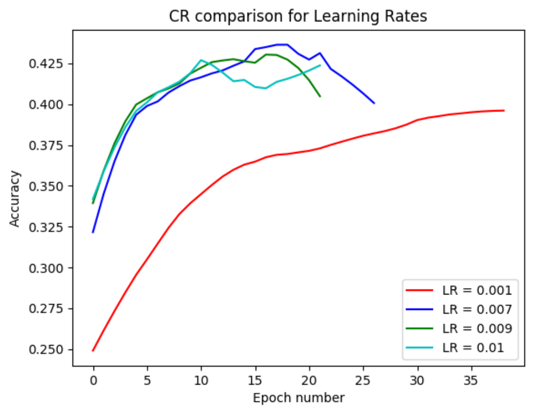

Figure 3: Validation Accuracy

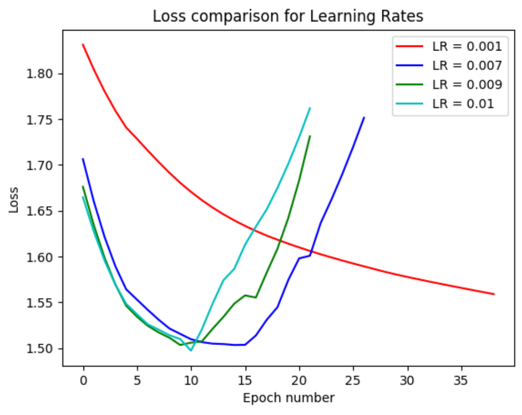

Figure 4: Validation Loss Comparison

Number	Learning Rate	Accuracy	Loss
1	0.001	39.3%	1.56
2	**0.007**	**43.3%**	**1.50**
3	**0.009**	**43.3%**	**1.54**
4	0.01	41.9%	1.52

From the table we observe that we get good results with both initial learning rate of **0.007** and **0.009** in terms of accuracy. However we have a better validation loss value on 0.007.

4.3 Learning Rate Update Schedule

Learning rate update provides us with the advantages of both higher learning rate by initially training the model at a higher rate and lower learning rate by preventing over-training of the network and maintaining low validation loss rate. Since we reduce the learning rate, we can afford to choose higher initial learning rate therefore we choose **0.009**.

4.3.1 Strategy 1: Decay by percentage

Formulae: $curr\_lr = \dfrac{inital\_lr}{1 + decay\_perc * iter}$ The first strategy to reduce the learning rate by some percentage at each iteration the following table shows the result of experiments.

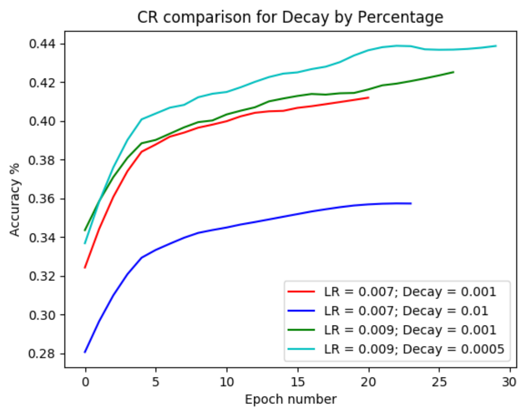

Figure 5: Validation Accuracy

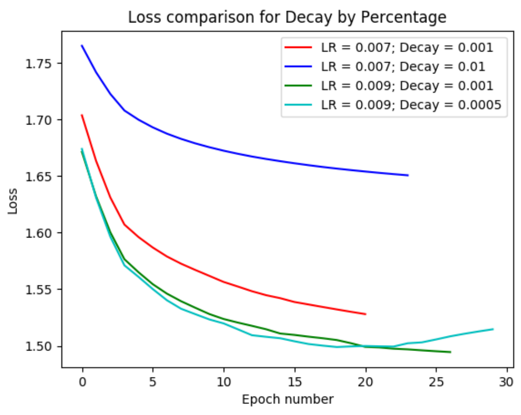

Figure 6: Validation Loss Comparison

Number	Learning Rate	Decay	Accuracy	Loss
1	0.007	0.1%	40.2%	1.54
2	0.007	1%	35.3%	1.66
3	**0.009**	**0.05%**	**43.8%**	**1.52**
4	0.009	0.1%	42.1%	1.49

We can compare that the best result is obtained with initial learning rate to be **0.009 and decay of 0.0005** for this strategy. This is a complement of high initial learning rate and a decay at each iteration. As we discussed above we get better results for 0.009(higher initial learning rate) as compared to 0.007.

4.3.2 Strategy 2: Decay after constant iterations

Formulae: $curr\_lr = \dfrac{inital\_lr * cont\_iter}{max\{curr\_iter, cont\_iter\}}$ Second strategy was to reduce the learning rate after some constant iteration by a factor. The following table shows the results of the experiment.

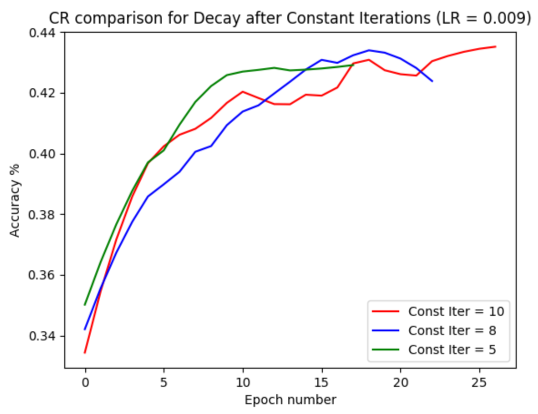

Figure 7: Validation Accuracy

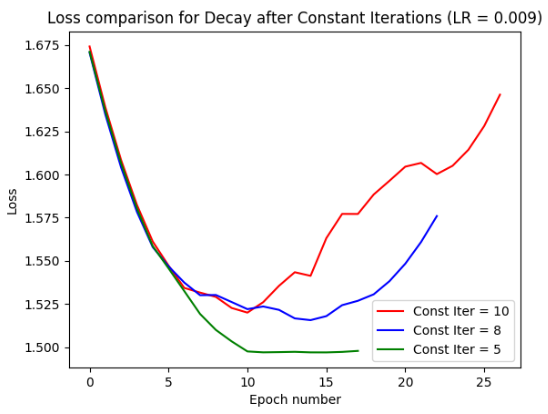

Figure 8: Validation Loss Comparison

Number	Learning Rate	Decay iteration	Accuracy	Loss
1	0.009	5	42.6%	1.50
2	**0.009**	**8**	**43.7%**	**1.53**
3	0.009	10	43.7%	1.62

The best result for this strategy is for **learning rate 0.009 and delay iterations of 8**. This has a accuracy of 43.7% and loss function of 1.53.

4.3.3 Strategy 3: Decay by a factor

Formulae: $curr\_lr = prev\_lr * factor$ In this strategy we reduce the learning rate by a factor at each iteration. The results are below in the table.

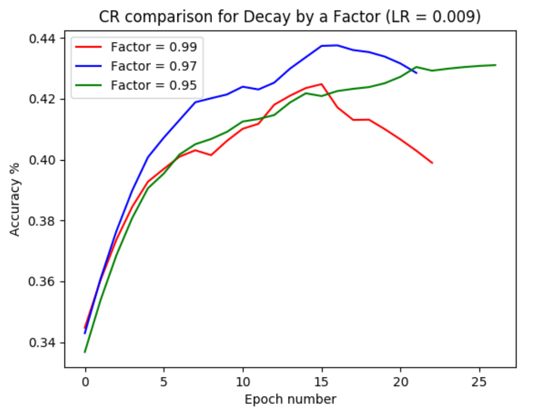

Figure 9: Validation Accuracy

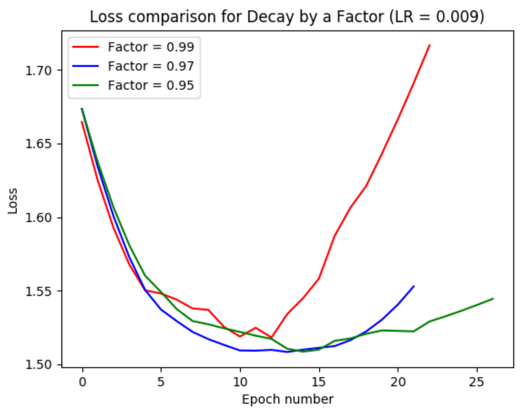

Figure 10: Validation Loss Comparison

Number	Learning Rate	Decay Factor	Accuracy	Loss
1	0.009	0.95	42.6%	1.53
2	**0.009**	**0.97**	**43.9%**	**1.52**
3	0.009	0.99	42.3%	1.56

The best result for this strategy is for **learning rate 0.009 and delay factor of 0.97**. This has a accuracy of 43.8% and loss function of 1.52.

4.3.4 Strategy 4: Decay by a constant

Formulae: $curr\_lr = \frac{inital\_lr}{1 + \frac{iter}{decay\_rate}}$ In this approach we reduce the learning rate using a predefined constant value using the above formulae. The results are given below.

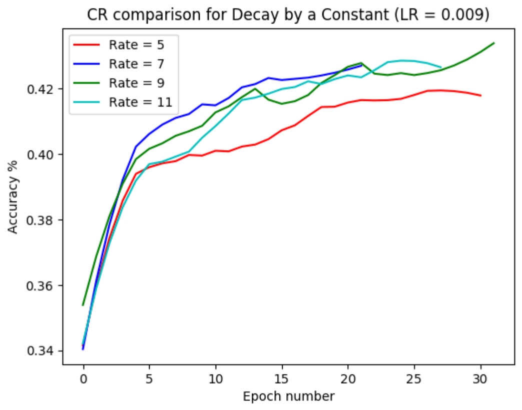

Figure 11: Validation Accuracy

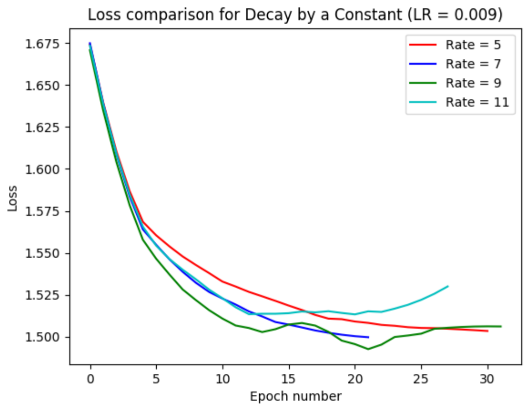

Figure 12: Validation Loss Comparison

Number	Learning Rate	Decay Rate	Accuracy	Loss
1	0.009	5	41.7%	1.51
2	0.009	7	42.1%	1.50
3	**0.009**	**9**	**43.6%**	**1.52**
4	0.009	11	42.1%	1.53

The best result for this strategy is for **learning rate 0.009 and delay iterations of 9**. This has a accuracy of 43.6% and loss function of 1.52.

Variable Momentum: Nesterov Nesterov Momentum update is a methord to update momentum that has recently been gaining popularity. It enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistenly works slightly better standard momentum.

Considering all the above strategies we have tested, we can conclude that we get the best results from **Decay by a factor, with inital learning rate 0.009 and decay factor 0.97**. The accuracy we achieved with this configuration was 43.9% and loss value of 1.52.

4.4 Dropout rate

We now analyse various ways to prevent overfitting. The first technique is called Dropout which modifies the neural network itself rather than the error function. During training, neurons are randomly dropped out with probability p. Dropout prevents overfitting because it stops the neurons from co-adapting too much. Each neuron should create useful features on its own, without relying on other hidden units to correct its mistakes. In other words, using dropout we add noise to the network during traning so that the network cannot memorise the training examples.

In our implementation, we tried several values for the probability with which neurons would be dropped out. We found that the best strategy was to use a dropout rate of `0.1` for hidden neurons. The configuration we used was the best result we got from the last part which is all three hidden layers are `ReLU` with `900`, `300`, and `100`, they have the same initial learning rate and use the same learning rate update schedule - Decay by a factor.

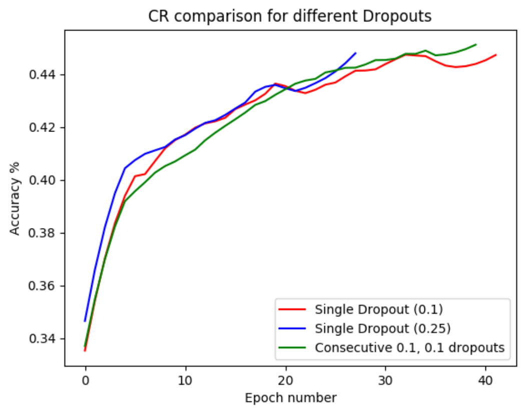

Figure 13: Validation Accuracy

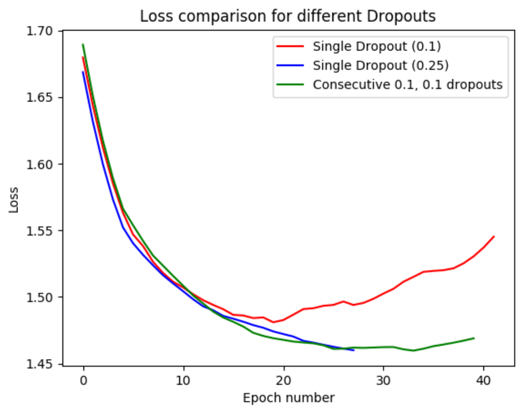

Figure 14: Validation Loss Comparison

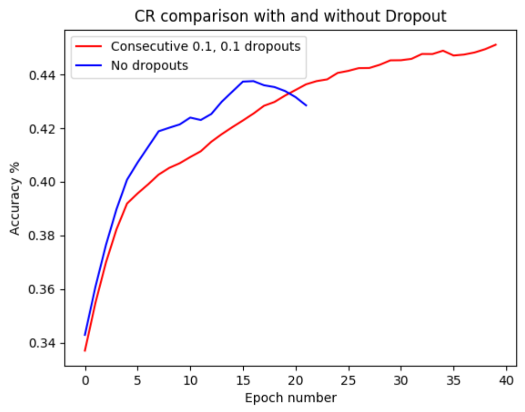

Figure 15: Validation Accuracy

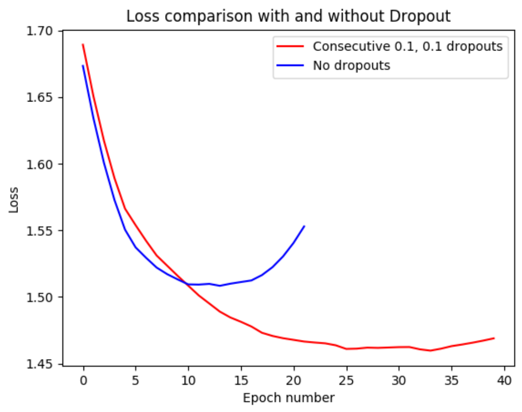

Figure 16: Validation Loss Comparison

Number	Dropout Configuration	Accuracy	Loss
1	No Dropout	43.9%	1.52
2	[0.0, 0.05, 0.0]	42.6%	1.44
3	[0.0, 0.1, 0.0]	44.1%	1.55
4	[0.0, 0.15, 0.0]	43.4%	1.57
5	[0.0, 0.25, 0.0]	44.2%	1.47
6	[0.0, 0.5, 0.0]	41.5%	1.59
7	[0.0, 0.05, 0.05]	43.5%	1.48
8	**[0.0, 0.1, 0.1]**	**44.4%**	**1.46**
9	[0.0, 0.3, 0.3]	43.6%	1.46
10	[0.1, 0.2, 0.2]	43.2%	1.47

The difference between these configurations is the dropout rate, and the results are shown in the graph above. Moreover, we observe that we obtain better results when using dropout in comparison to the previous implementations without dropout. The best result is obtained with dropout rate of [0, 0.1, 0.1] for the corresponding first, second and third layer. The **validation accuracy we achieved was 44.4% and loss function value was 1.46**.

4.5 Other regularisation

For this part, we tried different approaches for regularisation. Regularisation is one way to fight the tendency of neural networks to overfit the training data. Regularisers are useful because they allow to apply penalties on layer parameters or layer activity during optimisation. These penalties are incorporated in the loss function that the network optimises. We start by disabling the dropout rate and keeping the rest of the architecture unchanged. Then, we tried various combinations for regularisation and observed the outputs.

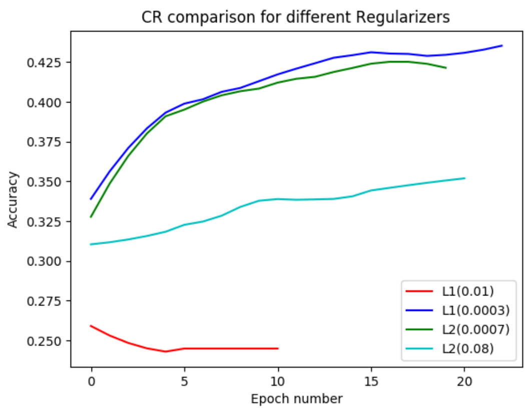

Figure 17: Validation Accuracy

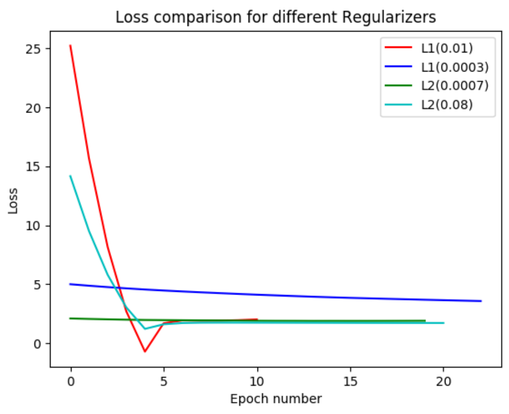

Figure 18: Loss Comparison

Number	Regularisation Type	Regularisation Rate	Accuracy	Loss
1	L1	0.0001	42.3%	2.73
2	**L1**	**0.0003**	**42.5%**	**3.61**
3	L1	0.0005	42.1%	2.46
4	L1	0.01	24.9%	1.95
5	L2	0.0003	41.7%	2.31
6	L2	0.0007	42.1%	2.16
7	L2	0.001	40.6%	1.89
8	L2	0.08	34.8%	1.71

The result we observe was accuracy of 42.5% with L1 and regularisation rate of 0.0003. Comparing these results with the ones from dropout we notice that while L1 and L2 regularization are implemented with a clearly-defined penalty term, dropout requires a random process of switching off some units, which cannot be coherently expressed as a penalty term and therefore cannot be analyzed other than experimentally. Analysing the outputs for both L1 and L2 regularisation, we observe that the accuracy does not match the one we obtained for dropout. Consequently, we opted to include dropout in our neural network architecture.

4.6 Topology of the network

There are two ways to change the topology of the network.

4.6.1 Number of layers

In this experiment we change number of layers on our standard model to analyse its affect. The following table and graph shows the effect. Graphh

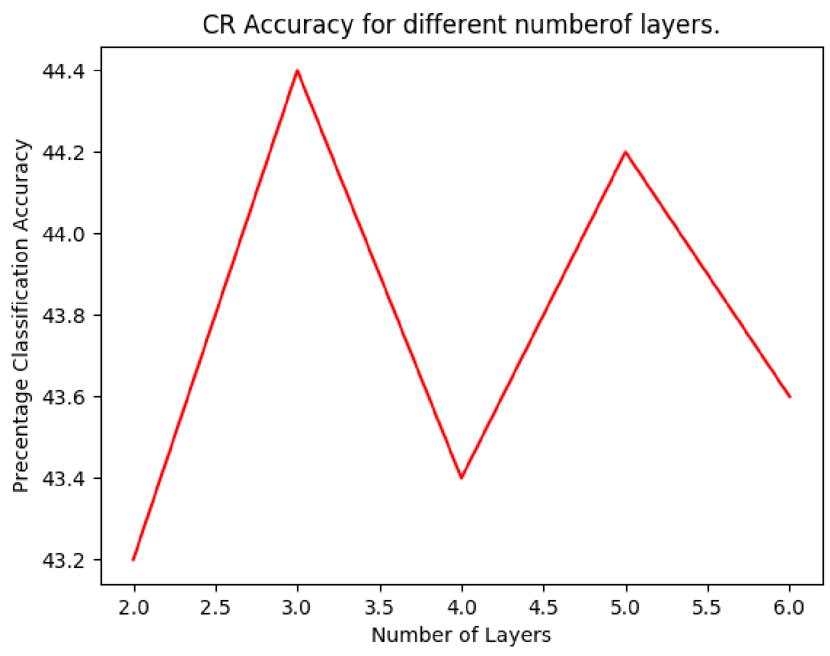

Figure 19: Validation Accuracy

Number	Number of Layers	Neurons per Layer	Accuracy
1	2	[900, 100]	43.2%
2	**3**	**[900, 300, 100]**	**44.4%**
3	4	[900, 300, 300, 100]	43.4%
4	5	[900, 300, 300, 300, 100]	44.2%
5	6	[900, 300, 300, 300, 300, 100]	43.6%

We observe that we observe little effect after we increase the number of layers more than 3 in validation accuracy. However, the training time and validation loss is much higher therefore we can conclude **number of layers 3** as the best result.

4.6.2 Number of neurons

In this experiment we change number neurons in each layer of our standard model to analyse its affect. The following table and graph shows the affect.

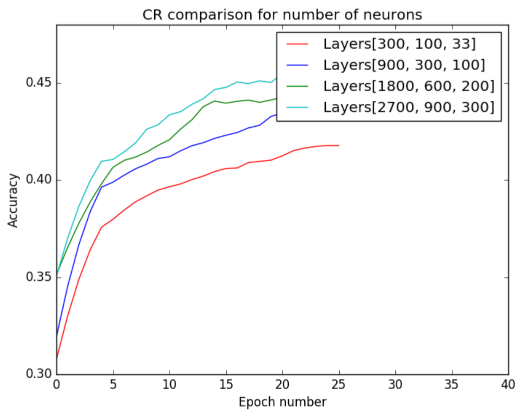

Figure 20: Validation Accuracy

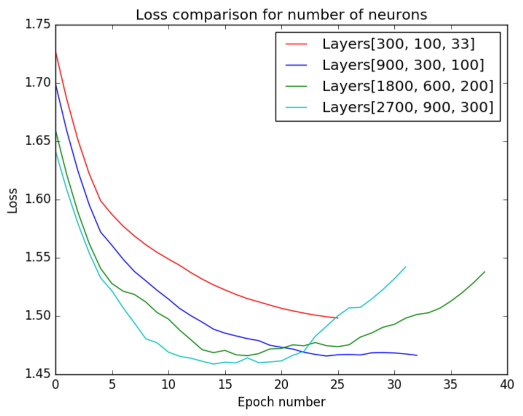

Figure 21: Loss Comparison

Number	Neurons per Layer	Accuracy	Loss
1	[300, 100, 33]	41.19%	1.49
2	[600, 200, 66]	42.4%	1.49
3	[900, 300, 100]	44.4%	1.46
4	[1350, 450, 150]	44.8%	1.50
5	[1800, 600, 200]	45.5%	1.52
6	[2700, 900, 300]	46.2%	1.54

We observe that as we increase the number of neurons in the ration, we tend to get better accuracy. However, the training time and the loss function value also increases as the number of connections in the model increase exponentially.

4.7 Trying other Activation Functions

4.7.1 Sigmoid

The main reason to use sigmoid is because it maps values between 0 and 1. This is particularly useful since it allows us to predict the probability of an output. Another advantage is that the derivative of the sigmoid allows us to easily apply gradien t descent during back propagation. However, the principal drawback of this activation function is the problem of vanishing gradients. That is, sigmoid activation function responds poorly to changes in gradient, especially when considering the end of the sigmoid function where changes are small. Thus, we reach a stage in which the neural network gets stuck during training, that is, it cannot make any progress, cannot learn further or is very slow to be feasible.

4.7.2 Leaky ReLU

For ReLU the network trains faster on positive input. However, the 0 gradient on the left end leads to the so-called dead neurons problem.
To address this problem, a variation of the ReLu activation function called leaky ReLU can be used. To some extent leaky ReLU eliminates the dying neurons problem present in ReLU, but results are not always consistent. Another disadvantage is that ReLU is not bounded, since it can take values from 0 to infinity, which in turn means that it can blow up the activation.

4.7.3 Hyperbolic Tangent (tanh)

Tanh maps values between `-1` and `1`. It actually is a scaled sigmoid function: `tanh(x) = 2*sigmoid(2x)` `1`, thus it has similar characteristics to the ones discussed above for the sigmoid function and the problem of the vanishing gradient persists to a lesser extent, although there are some differences. The main advantage of tanh over sigmoid is that it maps negative values to negative outputs. Hence, in comparison to sigmoid, outputs are zero-centred, but it does not cram values around `0`. This lowers the likelihood of the network being stuck during training.

4.7.4 Overall Results

Experimentally, we notice that ReLu also tends to overfit. Some methods to overcome this kind of shortage are batch normalisation and dropout. Adding to this, introducing tanh in the first hidden layer combined with two ReLU's in the nexttwo hidden layers prevented the network from overfitting to a better extent than having three ReLU's as hidden activation functions. Consequently, we replaced the activation function in the first hidden layer with tanh. The results generated by this new setup are shown below:

Number	Hidden Activation Functions	Accuracy	Loss
1	[ReLU, ReLU, ReLU]	43.7%	1.57
2	**[tanh, ReLU, ReLU]**	**44.0%**	**1.52**
3	[tanh, tanh, ReLU]	43.0%	1.50
4	[sigmoid, ReLu, ReLU]	36.6%	1.64
5	[softplus, ReLU, ReLU]	40.6%	1.58
6	[leaky ReLU, ReLU, ReLU]	43.0%	1.53

4.8 Optimal parameters

By observing the patterns and the results from the above sections, we decided on optimal set of parameters to maximize validation accuracy and minimise the loss as follows:

Number of layers: 3

Activation Function: ['tanh', 'relu', 'relu']

Number of neurons in Layers: [1800, 600, 100]

Dropout rate for corresponding layers: [0.1, 0.2, 0.2]

Initial Momentum: 0.7

Initial Learning rate: 0.009

Learning rate update schedule : Decay by a factor (0.97 decay rate)

As we can observe from the graphs below, we have observed a major increase from **41.9% at start of step 2 to 46.2%**. We also observe an improvement in loss function **from 1.52 to 1.42**. We also now prevent over fitting observed in part 2 while training on training set.

We experience these changes because we have optimised the parameters and structure of neural network to give us better results. Therefore, we can conclude the optimisation parameters and choosing the right architecture can have significant impact on the performance of a neural network. Note: Validation Classification error = 1 - Validation Accuracy

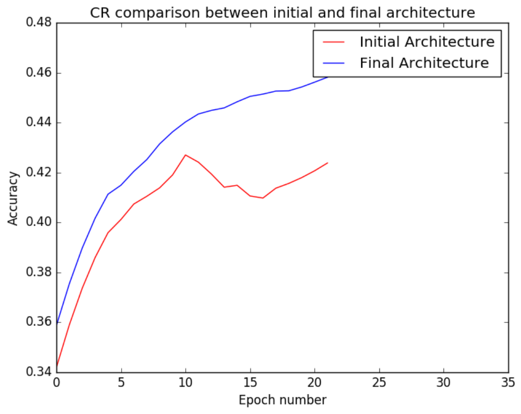

Figure 22: Validation Accuracy

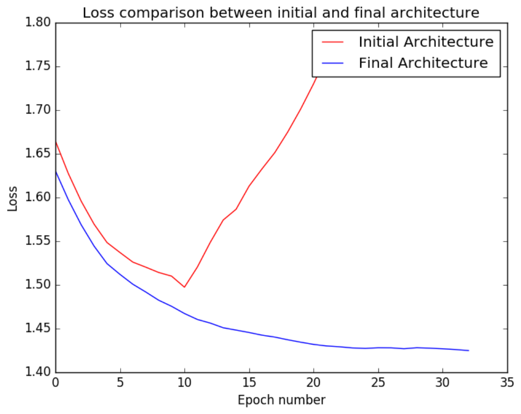

Figure 23: Loss Comparison

5 Part IV: Performance Estimation

The following are the results with the parameters in the last section on the test set.

Test Accuracy: **44.8036%**

Test Classification error: **55.1964%**

Test Loss function value: **1.4627**

Classification report:

Number	Precision	Recall	F1 Measure
angry	0.36	0.23	0.28
disgust	0.60	0.16	0.25
fear	0.34	0.23	0.27
happy	0.53	0.72	0.61
sad	0.37	0.34	0.36
surprise	0.58	0.59	0.59
neutral	0.38	0.44	0.40
avg / total	0.43	0.45	0.43

Confusion Matrix, without normalisation (same order of emotions as in the table above):

$$\begin{bmatrix} 106 & 1 & 39 & 122 & 90 & 30 & 79 \\ 13 & 9 & 3 & 12 & 9 & 3 & 7 \\ 45 & 2 & 114 & 95 & 98 & 52 & 90 \\ 36 & 1 & 39 & 644 & 67 & 24 & 84 \\ 45 & 1 & 372 & 140 & 224 & 33 & 138 \\ 17 & 0 & 31 & 53 & 29 & 246 & 39 \\ 34 & 1 & 38 & 146 & 90 & 33 & 265 \end{bmatrix}$$

Normalised Confusion Matrix (same order of emotions as in the table above):

$$\begin{bmatrix} 0.226 & 0.002 & 0.0835 & 0.261 & 0.192 & 0.064 & 0.169 \\ 0.232 & 0.160 & 0.053 & 0.214 & 0.160 & 0.053 & 0.125 \\ 0.090 & 0.004 & 0.229 & 0.191 & 0.197 & 0.104 & 0.181 \\ 0.040 & 0.001 & 0.043 & 0.719 & 0.075 & 0.026 & 0.093 \\ 0.069 & 0.001 & 0.110 & 0.214 & 0.343 & 0.050 & 0.211 \\ 0.041 & 0.0 & 0.075 & 0.128 & 0.070 & 0.594 & 0.094 \\ 0.056 & 0.001 & 0.062 & 0.240 & 0.148 & 0.0543 & 0.437 \end{bmatrix}$$

6 Part V: Questions

6.1 Best choice between Decision Trees and Neural Networks

In terms of evaluating the input performance, the decision tree algorithm has similar performance to neural networks, because both of the algorithms are eager.

Therefore, we shall focus on the training efficiency and accuracy of both algorithms. In terms of the training duration, we can say that the decision tree algorithm is faster than the neural network algorithm with respect to the number of epochs associated with the neural network. The decision tree algorithm passes through the data only once while the neural network algorithm passes the dataset as many times as the epoch number.

Having this in mind, because we are doing cross validation and the number of entries in the data set is small, a neural network with a large epoch count will most certainly over-fit. On the other hand, the decision tree algorithm should also take into account the possibility of over-fitting, but by implementing pruning we can avoid this problem.

If one algorithm is giving a better statistical error than the other one, we cannot assume that the algorithm is better than the other one. We have to take into account the results of all the folds. If one fold is substantially better than the others, this might lead to choosing an over-fit algorithm over a better one (due to the low number of entries in the dataset). The only way to discriminate between the algorithms is to have a very large dataset.

Moreover, we have to take into account the problem of adding new data to our dataset. In conclusion, the choice of adopting one algorithm over the other is data dependent and also application specific. We should choose the algorithm that gives good evaluation accuracy and good performance in the training phase.

6.2 Addition of one emotion to the dataset

For decision trees, we would have to change just the number of trees that are generated as we will have 1 more tree in this case.

For neural networks we would have to change the number of output layers, but by making this change we change the architecture of our neural network which might lead to the need of changing other parameters such as the number of neurons per hidden layer, drop-out rate, number of hidden layers and so on. Therefore the process of adapting a neural network is a bit more involved than the decision tree one due to the large number of parameters.