Ammaar Iftikhar

21901257

EEE 443-1 Fall 2022-23

Autoencoders

Fig 1. 200 grayscale and normalized images

Preprocessing Data:

To preprocess the data, I applied

$$Y = 0.2126 * R + 0.7152 * G + 0.0722 * B$$

to grayscale the images in the data set as was required by the question description. Following the grayscaling, I normalized and mapped them to the range [0.1, 0.9]. Before the mapping, the images are normalized as described in the question description. Both normalization and mapping are implemented in the *normalize_images* function. *normalize _images* function calls the *map_values* function to accomplish the mapping. The idea behind mapping is as follows:

*scaled data =0.4 * (data + max_value) / max_value + 1*

This will map 3 * std_dev to 0.9, -3*std_dev to 0.1, and all other numbers to the numbers (0.1, 0.9).

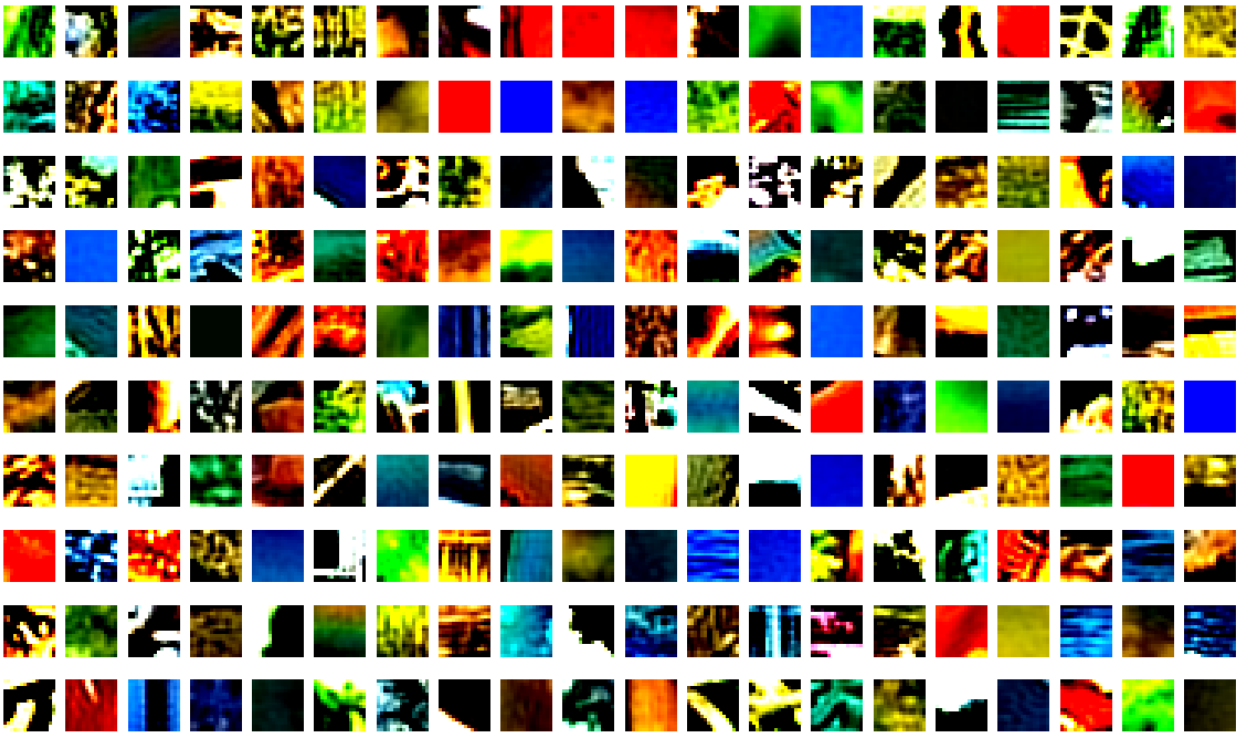


Fig 2. Original unmanipulated 200 images

Discussion:

If we compare the RGB images with the grayscale version of them we can see that the regions of the images with green color appear dark or black-ish in the converted images; this is in line with the definition proportion of contribution that green color gives in the formula above, which is appx. 72%. Our model is highly sensitive to the green color channel in the RGB and is least sensitive to the blue color channel. We can also see that the borders and change in color darkness is retained to a great extent in our grayscale images. The conversion to grayscale will help us decrease the computational demands of the Neural Network. Instead of operating on three separate channels of an image, we will be able to operate on a single channel to process the entire image.

1st Layer Weight Connection

Implementation

- Sigmoid was chosen as the activation function for both the encoder and decoder.
- Wh → weight of the hidden layer

  Wd → weight of decoder

  Wd = Wh.T
- For taking the gradients of Wh and Wd, I take the average of dWh and dWd and add them to Wh and then take transpose of it.

  dW = (dWh + dWd) / 2

  Wd = (Wh + dW).T

Discussion:

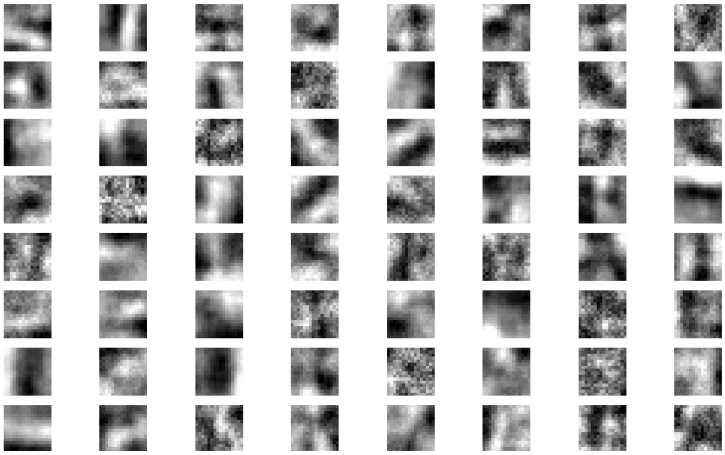


Fig 3. Weights visualization for n = 0.1, **β** *= 0.02,* ρ *= 0.02*

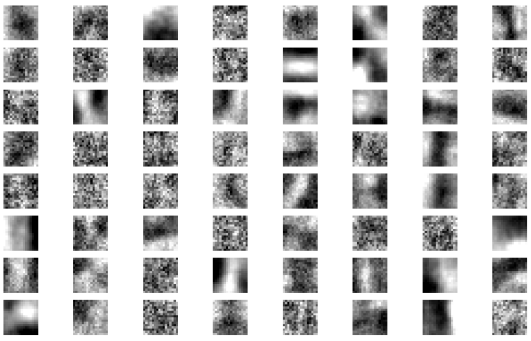


Fig 4. Weights visualization for n = 0.1, **β** *= 0.02,* ρ *= 0.01, lambda = 5*10^-5*

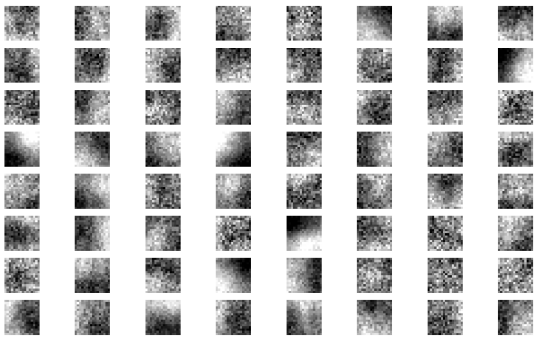


Fig 5. Weights visualization for n = 0.01, **β** *= 0.02,* ρ *= 0.05, lambda = 5*10^-4*

The extraction of different shapes and change in darkness over images shows that the network is able to extract some features from the images.

I conducted around three experiments, for hidden layer size of 64, with hyperparameters to analyze the impact of them on the learning and cross entropy loss of the autoencoder. We can see from the third image that having a lower learning rate directly leads to less sharper and more noisy images. So, the learning might note be completed in 100 epochs due to the small learning rate. When we compare this image with n = 0.01 to n = 0.1, we see that the overall features reconstructed from the weight matrix are sharper.

The other hyperparameter ρ *or the sparsity parameter* directly impacts the weights due to its contribution in the loss function gradient. We saw during testing that a decrease in the sparsity parameter and an increase in the beta hyperparameter while also having around 0.1 learning rate can lead to rapid increase/decrease in the weights which in turn leads to explosion of the network output. Both the features extracted with **sigma = 0.01 and sigma = 0.02 that the features extracted while having lower sigma values contribute in more sharper and less noisy**. Restricting the activation values of the hidden layer might contribute to better feature extraction relative to feature extraction at higher sigma values.

β also contributes to the feature extraction by determining the degree of influence that sparsity. The formula below represents the impact of beta on the change in weights.

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)}).$$

Fig 6. From sparse encoder notes of Andrew Ng.

Lambda hyperparameter impacts the total loss as well by determining the impact that larger weights will have on the loss function. With higher values of lambda, during training we see that the loss function starts to increase after 50-60 epochs of decrease in the loss function; this shows that the increasing weights start to have a higher impact on the loss value. When the lambda value is fixed to 0, there is an overall gradual decline in the loss function value.

Hidden Layer Size = 25

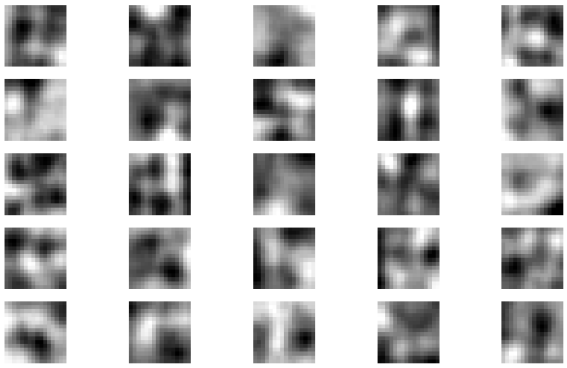


Fig 7. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.01, lambda = 0.001*

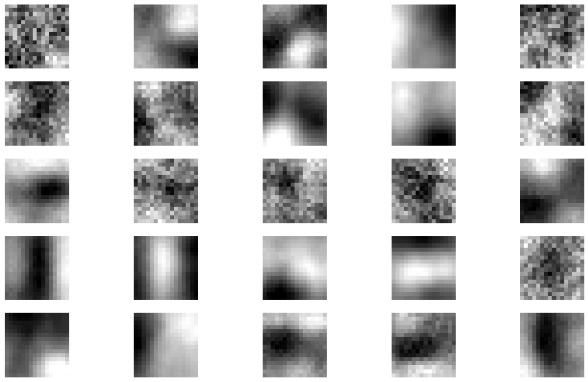


Fig 8. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.01, lambda = 5*10^-4*

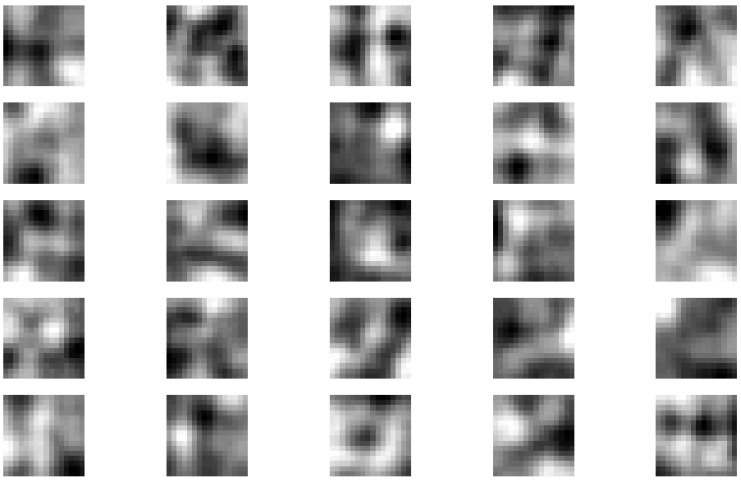


Fig 9. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.01, lambda = 0*

Hidden Layer 49

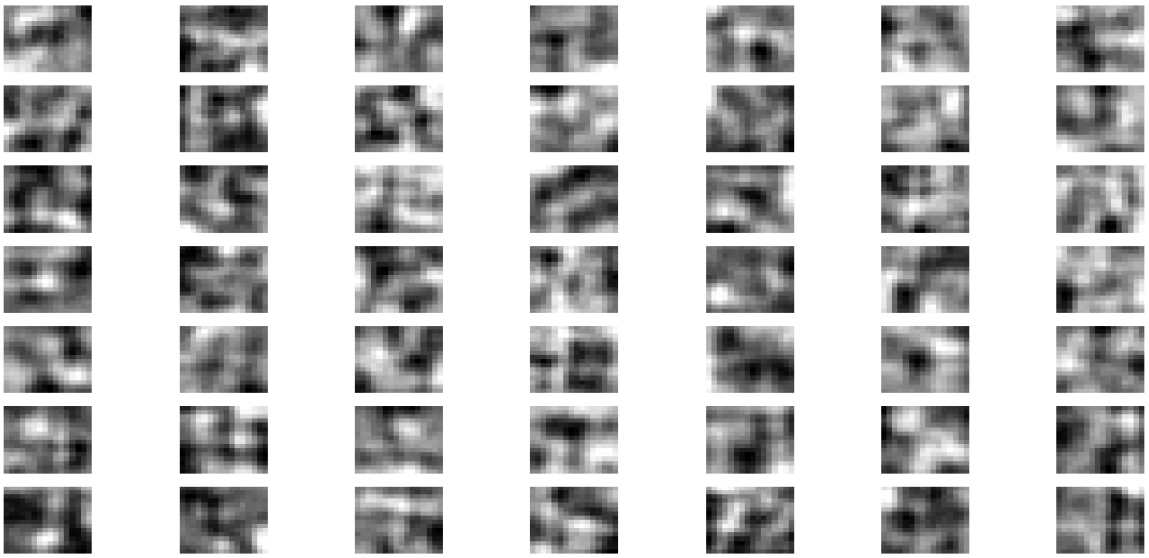


Fig 10. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.1, lambda = 0.001*

Fig 11. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.1, lambda = 0.0005*

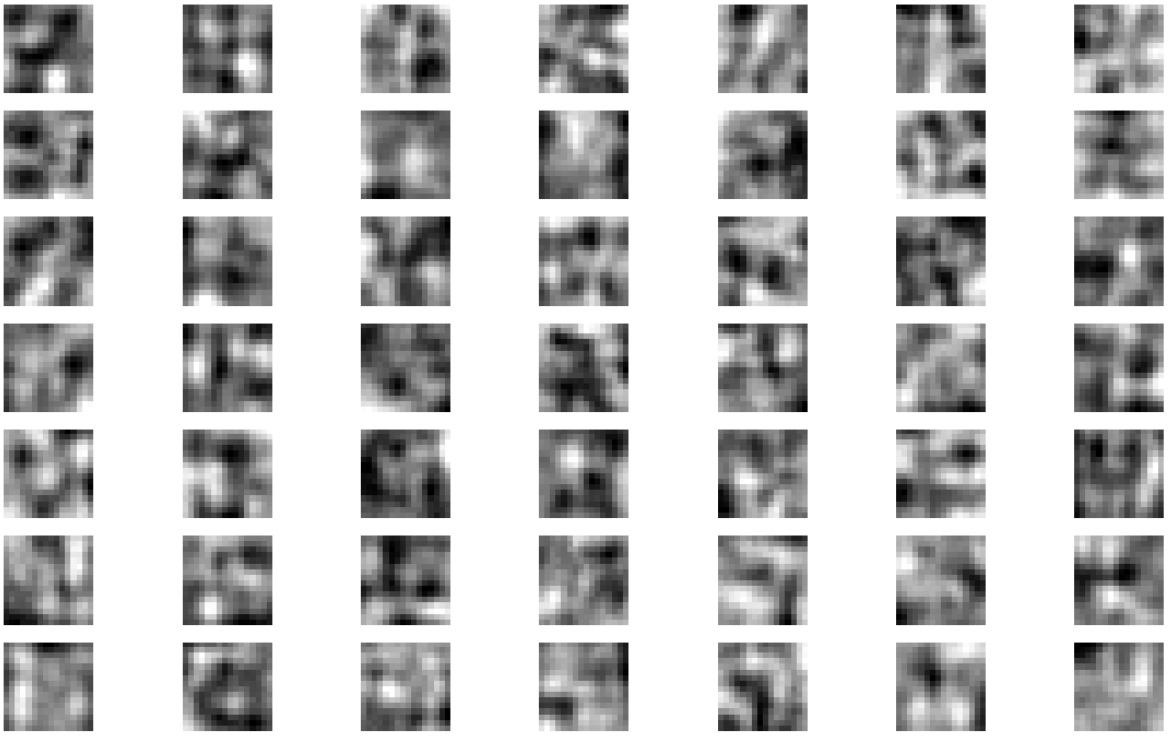


Fig 12. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.1, lambda = 0.*

We can see that the images become more dense as we decrease the value of lambda from 0.001 to 0. This is similar to the previous case.

Hidden Layer Neuron = 100

Fig 13. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.01, lambda = 0.001*

Fig 14. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.01, lambda = 0.0005*

Fig 15. Weights visualization for n = 0.1, **β** *= 0.1,* ρ *= 0.01, lambda = 0.*

Observations from training

One aspect that we can notice is that the shapes become more elaborate with decrease in the value of lambda, while with 0 or extremely low value of lambda we can see that the shapes tend to be simpler. So, having a low value might lead to overfitting while having a higher value might lead to underfitting; this is due to the fact that higher lambda will try to make the weights in the neural network close to zero.

We also noted that when we were training with 0.001 lambda, as the training moved beyond 60-70 epochs the error started to increase due to the increase in weights. In addition to this, this also led to a higher impact of weights on the weight changes.

The increase in the number of hidden units will lead to an increase in the amount of noise that the images will have. It will also lead to more redundancy in the features that the units will capture.

From the different number of different units, we can see that the system with more neurons is able to detect more elaborate features in the training set which will improve the overall capability of the decoder to reconstruct images.

In my opinion, the best set of parameters are: rho = 0.1, beta = 0.01, lambda = 0.0005 with learning rate 0.01. We achieved a final loss of 0.49 in this. This is not the lowest loss that we achieved, but these are the reasons why I think it is better:

- Elaborate features can be seen in hidden weight visualization which shows the potential of the network to learn important features.
- The features are compressed more than they are when we use 100 hidden layers
- The loss is low compared to training with 25 hidden layers.
- Weight regularization is considered in the loss, but it is not afforded a high influence.

params["Lin"], params["Lhid"], params["rho"] = 256, 49, 0.1
params["beta"], params ["lambda"] = .01, 0.0001
learning rate = 0.1

Word Prediction

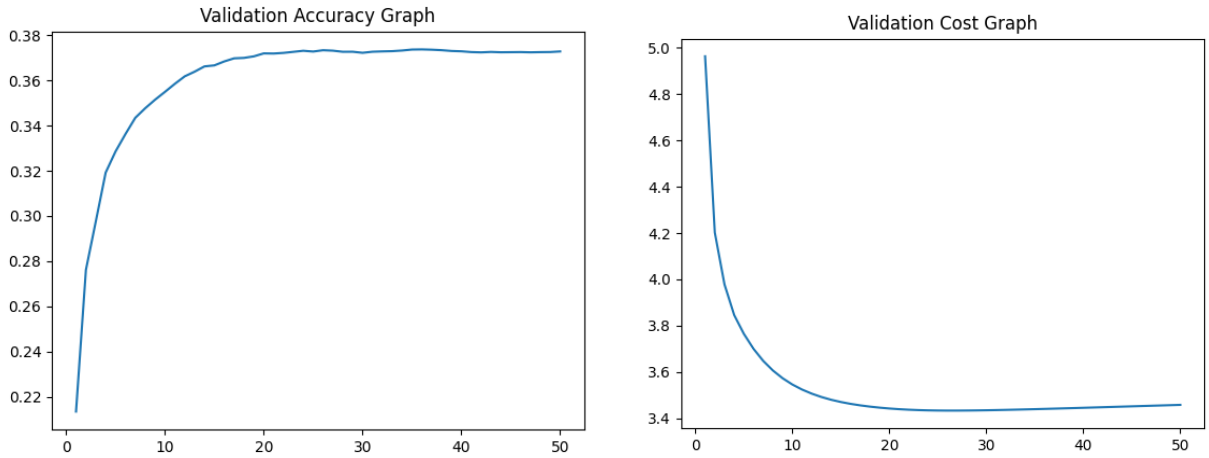


Fig 16. Validation set for D = 32 and P = 256

Test Information

TEST SET INFORMATION:

correct prediction = 17237

prediction percentage = 0.37068817204301074

validation set loss = 3.4614476306760045

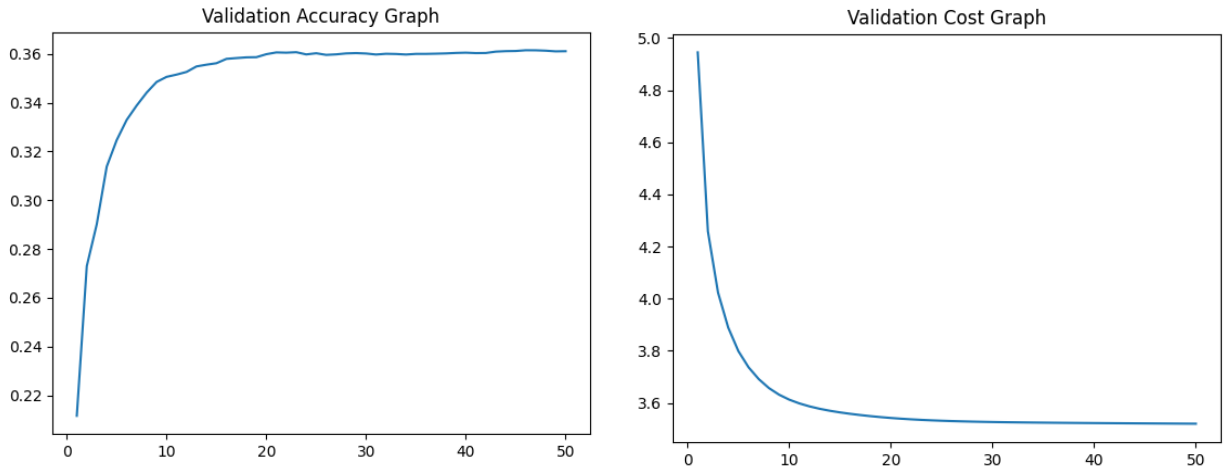


Fig. Validation set cost and accuracy for D = 16 and P = 128

Test Information

TEST SET INFORMATION:
correct prediction = 16707
prediction percentage = 0.35929032258064514
validation set loss = 3.5298858544878726

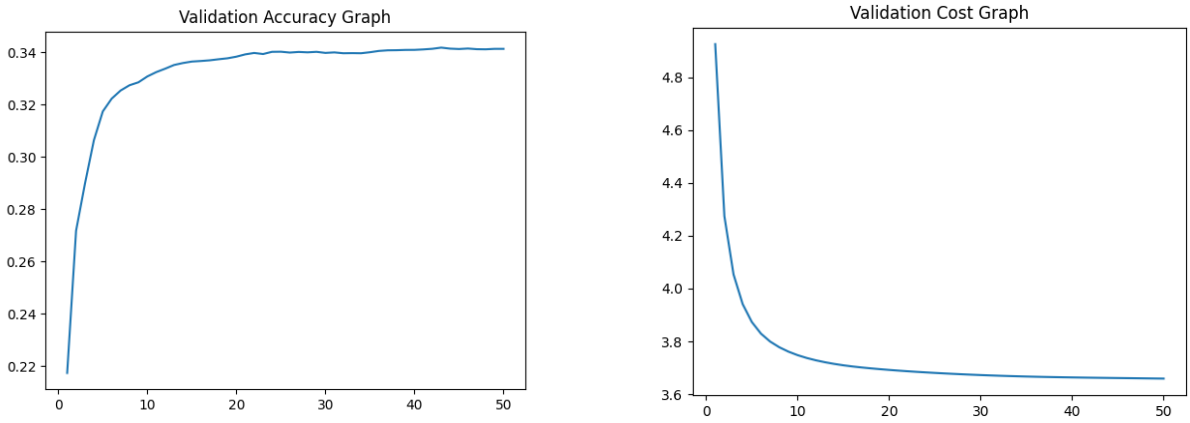


Fig 17. Validation set performance of D= 8 and P = 64

Test Set

TEST SET INFORMATION:
correct prediction = 15847
prediction percentage = 0.34079569892473116
validation set loss = 3.66735184420971

Before creating the embedding vector for the words, I am preprocessing them by creating a red hot vector representation of them. Following this, I am using the embedding matrix to get their embedding representation. For backpropagation, I am taking the sum of the weight update or delta Wembedding.

From the 3 different models that we created we can see that the best performing model is D= 32 and P = 256. Its test set cross entropy loss is 3.44 which is around .8 less than the nearest competitor. Its accuracy is also 1% higher than the accuracy achieved with D = 16 and P = 256.

From this we can see that the size of the hidden layer correlates to the performance of the network.

Another aspect is the saturation that the accuracy achieves after around 30 epochs for all three different experiments. In all three experiments if the validation error fell to 0.1 the training would stop, however the validation loss was always higher than 3.

Part B.

```
[b'were' b',' b'and'] b'they'
[b'when' b'are' b'we'] b'going'
[b'a' b'year' b'since'] b'you'
[b'but' b'not' b'for'] b'long'
[b',' b'for' b'life'] b'.'
[b'you' b'are' b'not'] b'going'
[b'nt' b'have' b'him'] b'.'
[b'him' b'any' b'good'] b'.'
[b'what' b"'s" b'in'] b'it'
[b'did' b'nt' b'want'] b'to'
[b'have' b'a' b'long'] b'way'
```

Some of the predicted sentences make complete sense, but others need more context to them. For example “you are not going” is quite predictable, but “, for life .” is not. A bit more context might make them more understandable; this issue also points to the difficulty of the problem–prediction of the 4th word from the previous 3 might not provide enough context for the task. Having more surrounding words (left or right context) will definitely improve the accuracy while simultaneously increasing computational requirements for the training.

RNNs, LSTMs, and GRUs

Implementation of the Recurrent Layer in RNN

The recurrent layer has been implemented with the help of recurrent function. The recurrent structure of the *rollout* assists in understanding the recurrence relation.

V → weight matrix for input to recurrent layer connection

W → weight matrix for previous output to recurrent layer connection

x → matrix multiplication

Y [t] = V x X + W x Y[t – 1] + b

The logic behind the implementation is given in the form of the pseudocode as follows:

Pseudocode

```
RNN(W, V, b,  X, i , t):

    if i = t
        return tanh(V x X[ 0 ] + b)
    else
        X'' = RNN(W, V, b, X,  i+1, t )
        Y   = V x X[t – i ] + W x X'' + b

        return tanh(Y)
```

- Due to the problem of vanishing gradients and saturation of sigmoid function, after testing I decided to use ReLU

layer_dims = [128, 32, 16, 6] were selected during hyperparameter tuning for RNNs and LSTMs.

For GRUs [128, 70, 36, 20, 6] were the dimensions

Discussion

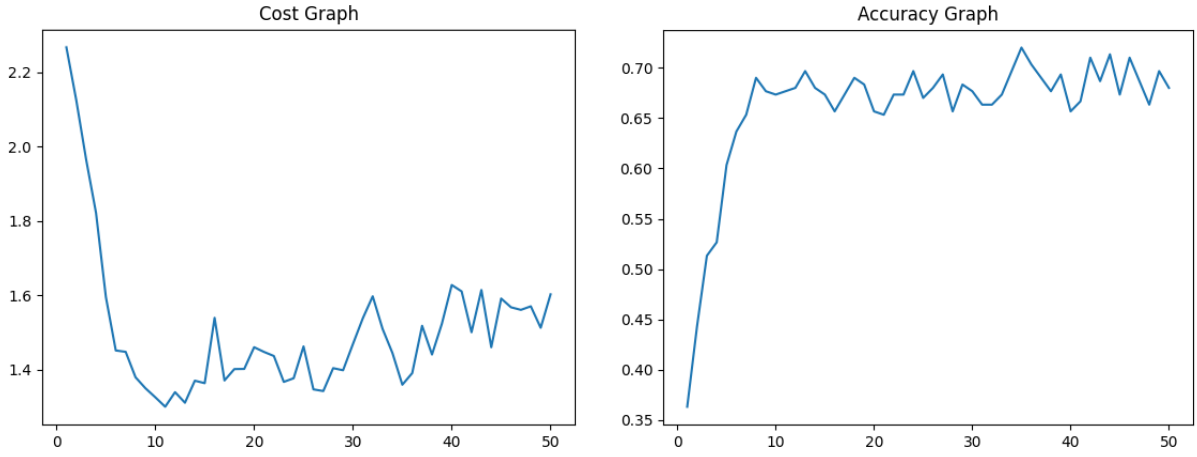


Fig 18. Validation set accuracy and cost for RNN with n = 0.01

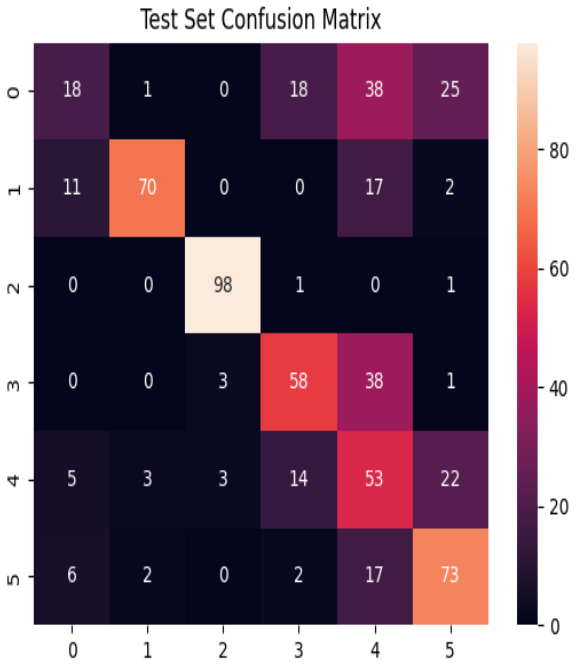


Fig 19. Test set Confusion Matrix

Test set

correct prediction count = 370

prediction percentage = 0.6166666666666667

set loss = 1.8696312407663291

[[18. 1. 0. 18. 38. 25.]
[11. 70. 0. 0. 17. 2.]
[0. 0. 98. 1. 0. 1.]
[0. 0. 3. 58. 38. 1.]
[5. 3. 3. 14. 53. 22.]
[6. 2. 0. 2. 17. 73.]]

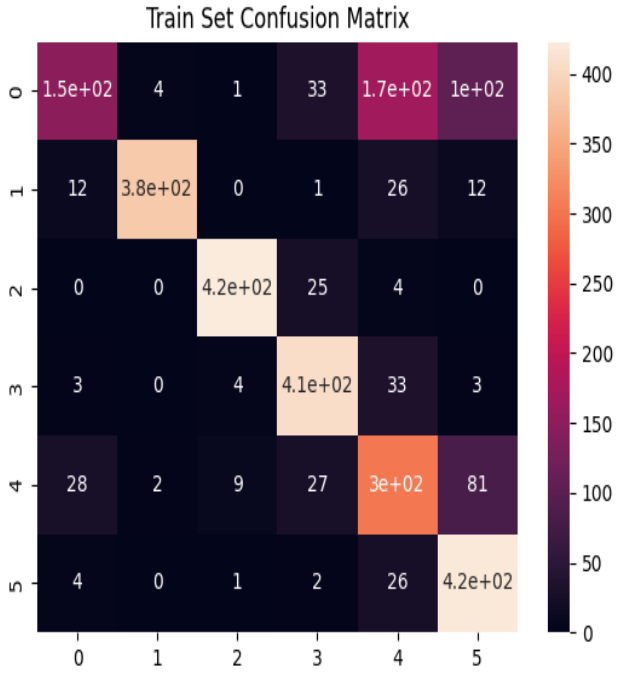


Fig 20. Training set confusion matrix

correct prediction count = 2086

prediction percentage = 0.7725925925925926

set loss = 0.9955685541478737

[[148. 4. 1. 33. 170. 103.]
[12. 385. 0. 1. 26. 12.]
[0. 0. 423. 25. 4. 0.]
[3. 0. 4. 407. 33. 3.]
[28. 2. 9. 27. 304. 81.]
[4. 0. 1. 2. 26. 419.]]

While training *Recurrent Neural Networks or RNNs* with learning rate = 0.1and sigmoid activation function in the hidden layer, I faced some issues like diminishing gradients and saturation of the network; this can be attributed to the convergence of sigmoid function to 1 with high values and 0 with low values. To fix this issue I replaced sigmoid activation function with Rectified or ReLU function. But there still were issues like exploding gradients. Due to this reason, I trained the network with a learning rate 0.01. With this learning rate I was achieving around 55-60% accuracy. To improve it, I scaled the weight and bias update in the Recurrent layer by a constant factor of 10. The idea came to me due to the fact that we add the gradient for around 150 rollbacks before updating the recurrent connection weights. I saw that the magnitude of the final delta W and delta B will be relatively higher than the rest of the network. After the scale down, I achieved ***around 77% and 67% accuracy on training and test sets respectively.***

I am also utilizing gradient momentum as required in the task specification. The highest accuracy we achieved for the validation set was around 72%. The major issue for RNN was its volatility in validation accuracy during training.

LSTMs

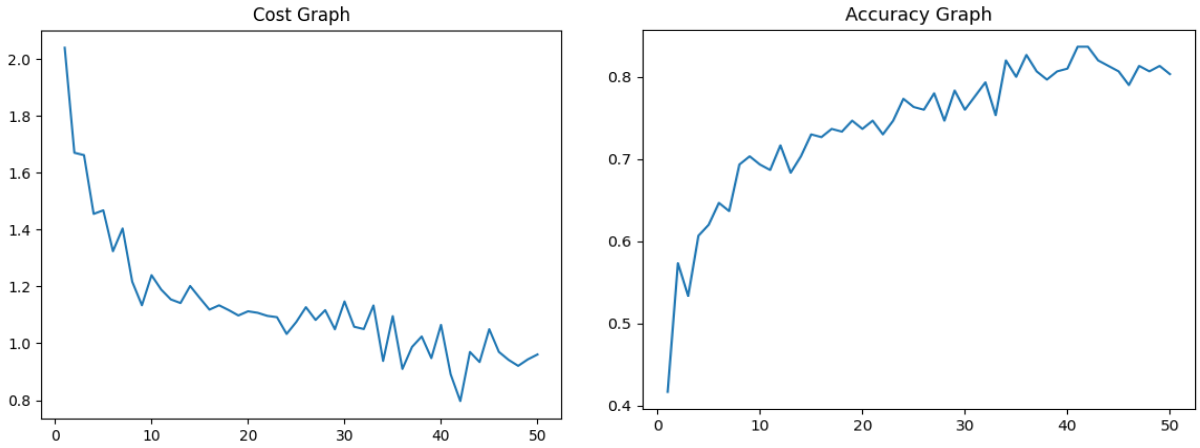
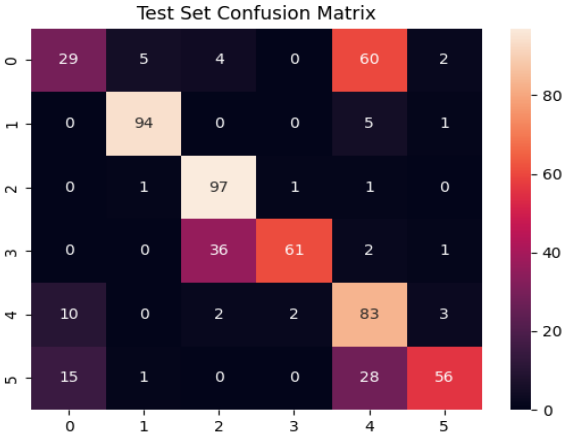


Fig 21. The cross entropy loss and accuracy of LSTM while training with n = 0.1




```
count = 420
prediction percentage = 0.7
set loss = 1.7559997652579569
[[29. 5. 4. 0. 60. 2.]
 [ 0. 94. 0. 0. 5. 1.]
 [ 0. 1. 97. 1. 1. 0.]
 [ 0. 0. 36. 61. 2. 1.]
 [10. 0. 2. 2. 83. 3.]
 [15. 1. 0. 0. 28. 56.]]
```

Fig 22. Test set confusion matrix

Training set

```
correct prediction count = 2331
prediction percentage = 0.8633333333333333
set loss = 0.7005863857897807
[[297. 9. 3. 0. 132. 3.]
 [ 4. 446. 0. 0. 1. 0.]
 [ 1. 0. 431. 4. 5. 0.]
 [ 2. 2. 9. 403. 41. 0.]
 [ 21. 31. 9. 3. 382. 2.]
 [ 40. 2. 1. 0. 44. 372.]]
```

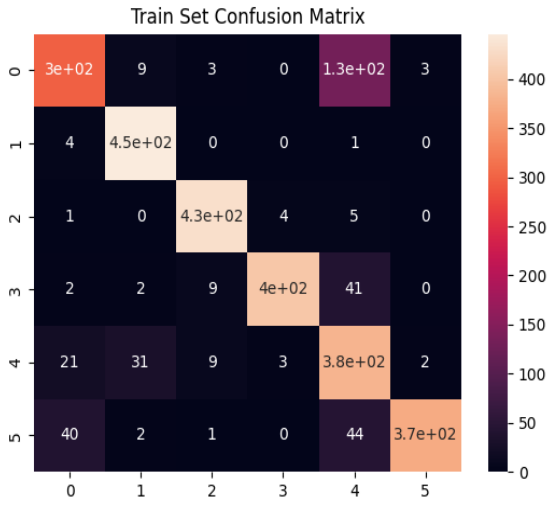


Fig 23. Training set confusion matrix

LSTM is more robust when it comes to tolerance of high gradients or overflows. I didn't face any such issue when running LSTMs. I was able to train it properly at a learning rate of 0.1; this was not possible for RNNs due to exploding gradients.

The final accuracy rate for LSTMs was 9% than that of RNNs on both the test set and training set.

Most of the misclassifications occur for the following pairs:

- Upstairs → jogging and vice versa
- Upstairs → downstairs and vice versa
- walking → downstairs and vice versa
- Standing → Upstairs and vice versa

We can speculate that there is a certain amount of similarity between these activities because both the RNN and LSTM have an overlap for some of the pairs like downstairs → upstairs might have more similarity than standing → jogging will have; this might make distinguishing such pairs of activities difficult for the Networks

The speed of LSTM was relatively slower than that of RNN, but the overall improvement of classification compensated for it. The increased robustness and lack of overflow and NaN also renders LSTM better than RNN. In my opinion, using LSTMs over RNNs is better both in terms of performance and stability of the system.

GRU

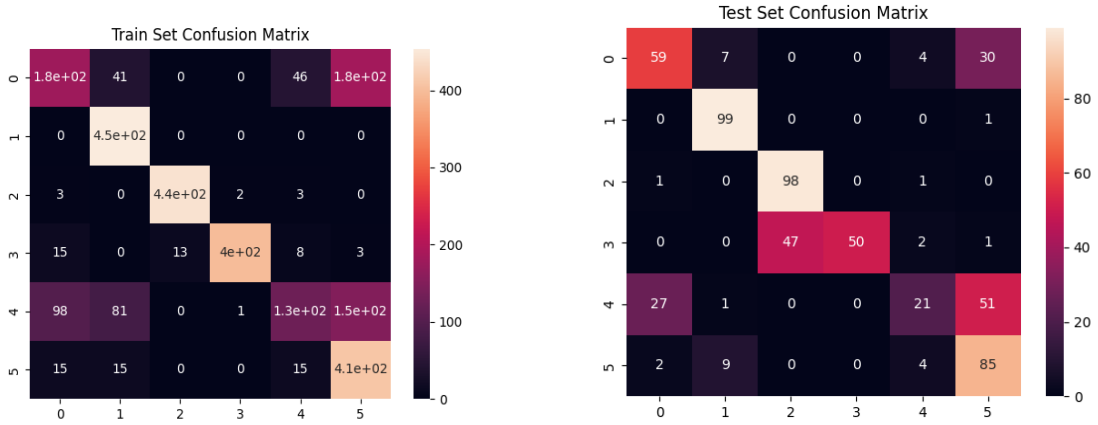


Fig 24. Training set confusion matrix

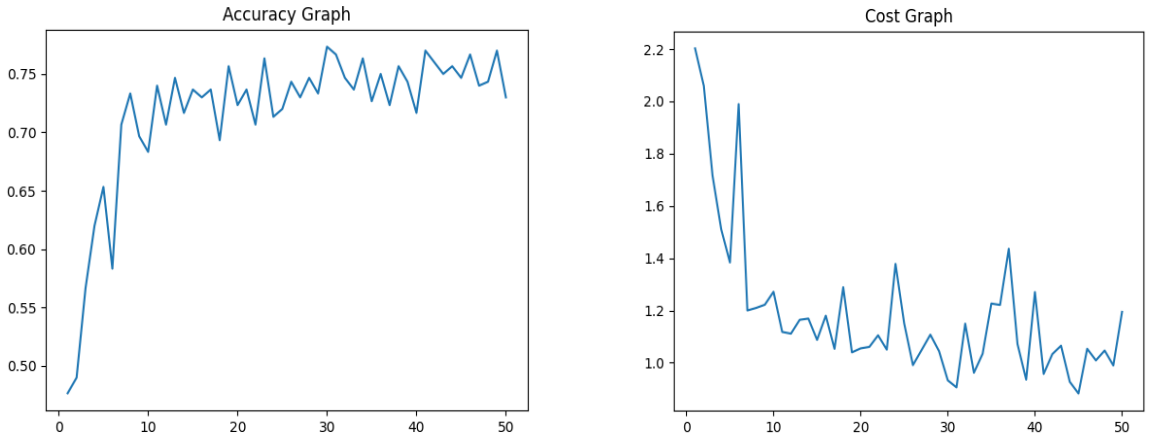


Fig 25. Training and test set performance

GRUs are less robust than LSTMs and are more prone to overflow at learning rate of 0.1 or more. At 0.1 there was high fluctuation in accuracy rate which showed that the learning rate should be decreased to achieve smoother learning.. They took similar time for training as LSTMs but didn't provide any improvement in performance. The maximum accuracy we achieved on the validation set was around 77% which is lower than the maximum achieved by LSTMs by about 3-4%; this was achieved by training the model at 0.05 learning rate.

GRUs performed better than RNNs as expected. It solved the issue of diminishing gradients. As expected its performance was around that of LSTMs.

An issue that I noticed during training GRUs was that its validation accuracy curve is filled with sharper rises and falls compared to both RNN and LSTM. LSTMs and GRUs provided similar kinds of accuracy performance. GRUs and LSTMs both achieved lower cross entropy loss than RNNs which also points to its better performance.

The misclassifications for GRUs have a similar kind of distribution as that of RNN and LSTM which shows similar kinds of difficulty with varying magnitude for all different network architectures; this, I believe, can be attributed to some sort of similarity between different activities.

Test Set:
correct prediction count = 412
prediction percentage = 0.6866666666666666
set loss = 1.6048062624982804
[[59. 7. 0. 0. 4. 30.]
[0. 99. 0. 0. 0. 1.]
[1. 0. 98. 0. 1. 0.]
[0. 0. 47. 50. 2. 1.]
[27. 1. 0. 0. 21. 51.]
[2. 9. 0. 0. 4. 85.]]
Training set
correct prediction count = 2009
prediction percentage = 0.7440740740740741
set loss = 1.171700909761799
[[179. 41. 0. 0. 46. 183.]
[0. 454. 0. 0. 0. 0.]
[3. 0. 443. 2. 3. 0.]
[15. 0. 13. 399. 8. 3.]

[98. 81.   0.   1. 126. 149.]
[15. 15.   0.   0. 15. 408.]]