**ELL409 Assignment 3 Report**

**Kshitij Alwadhi (2019EE10577)**

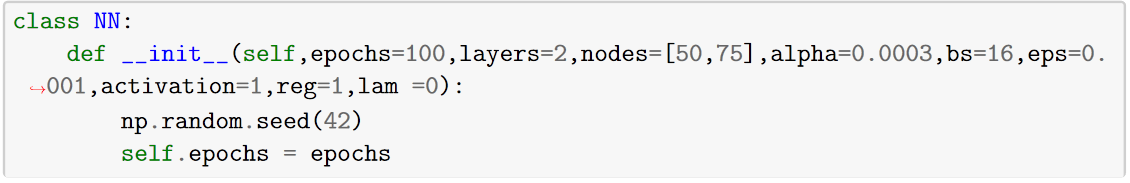
**6th November 2021**

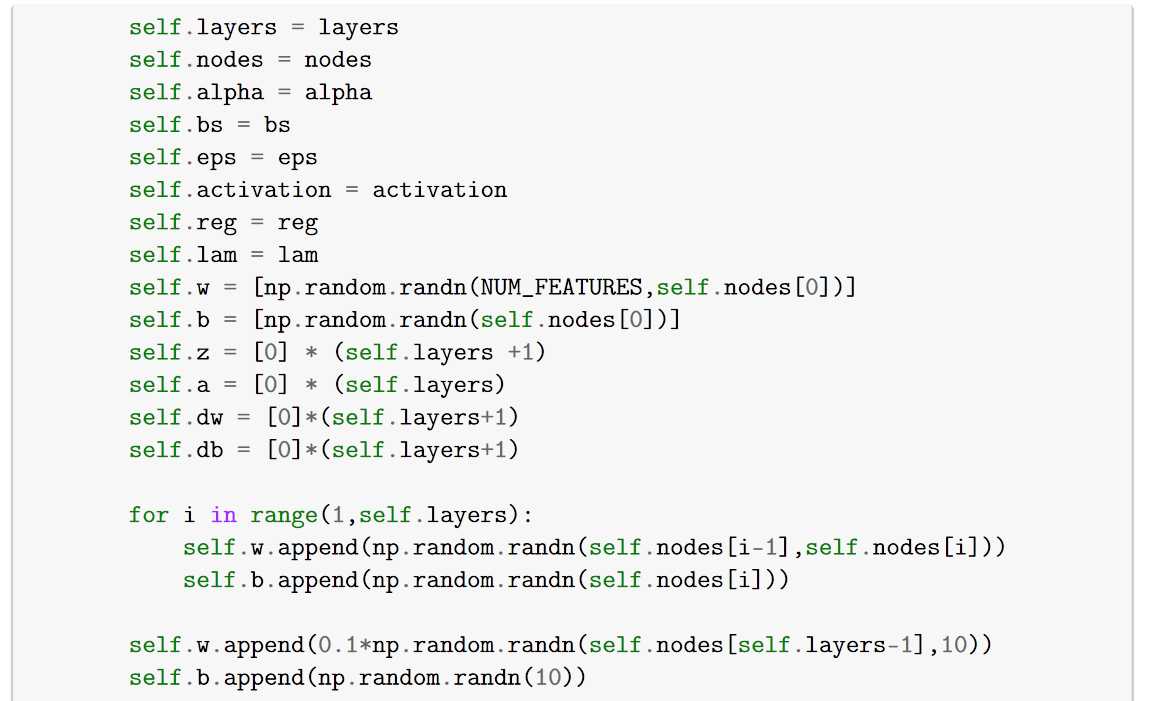
**Part 1A.**

In this part of the assignment, we experiment with our own implementation of Neural Networks, compare its performance to a standard library (Tensorflow) as well as try to understand the representations learnt by the neural network.

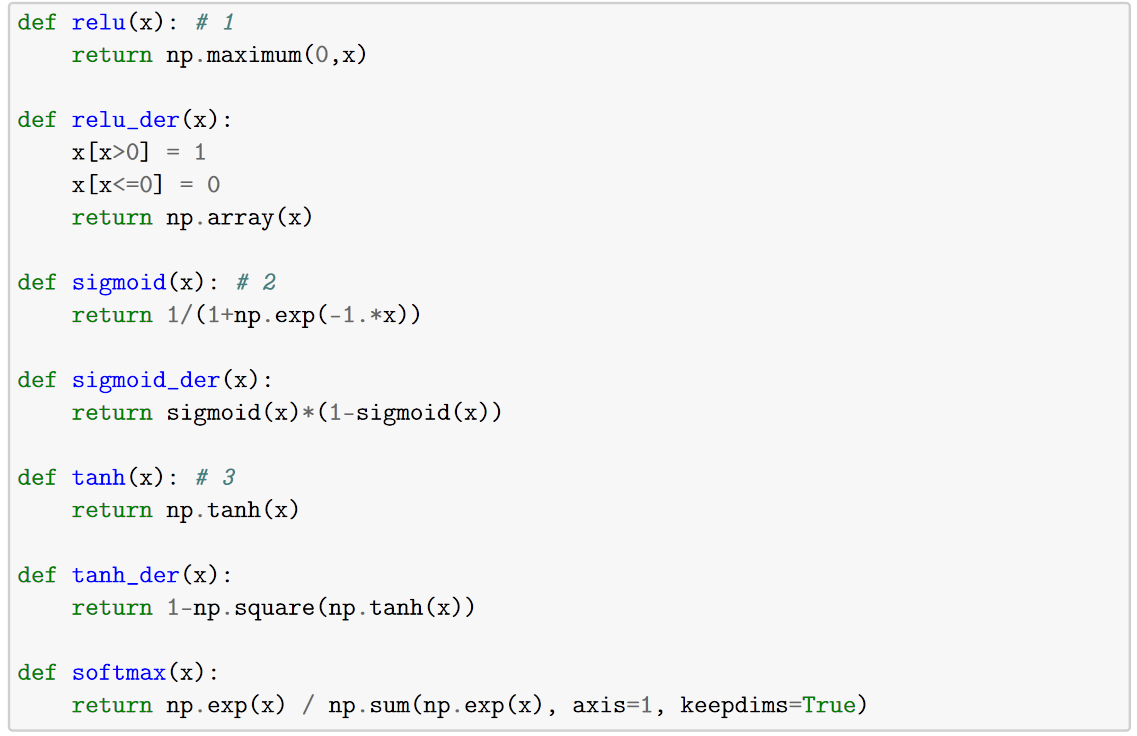
First we implement our own class of Neural Network using Numpy.

The following is the initialization portion of the neural network where we define the various parameters we will be using and also initialize the weights and the biases randomly.

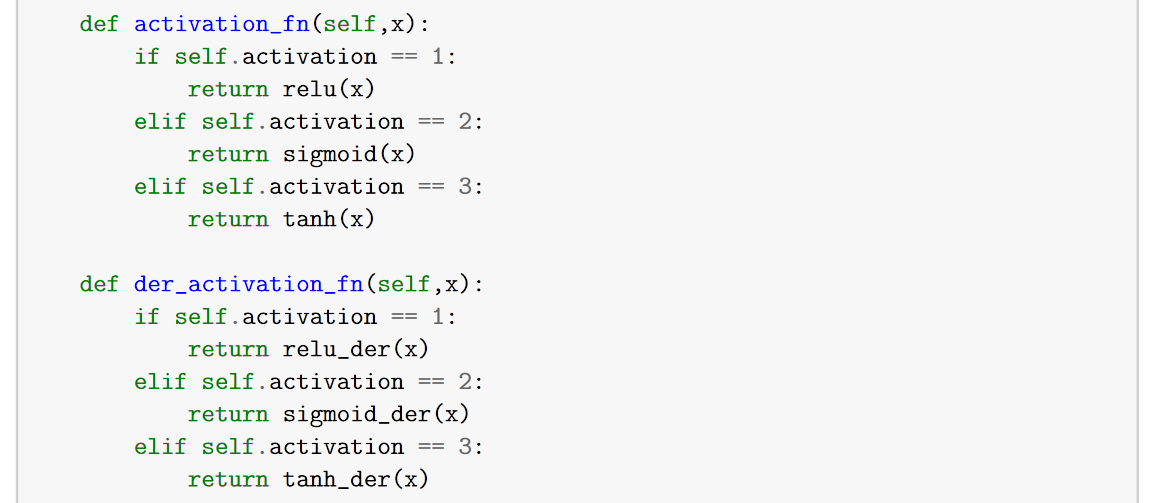




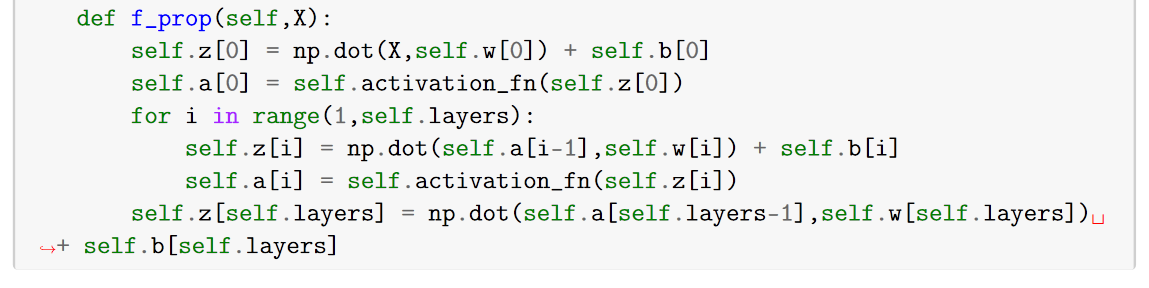
Next, we define our activation functions which we will be using in the hidden layers as well as the softmax layer for the final output.



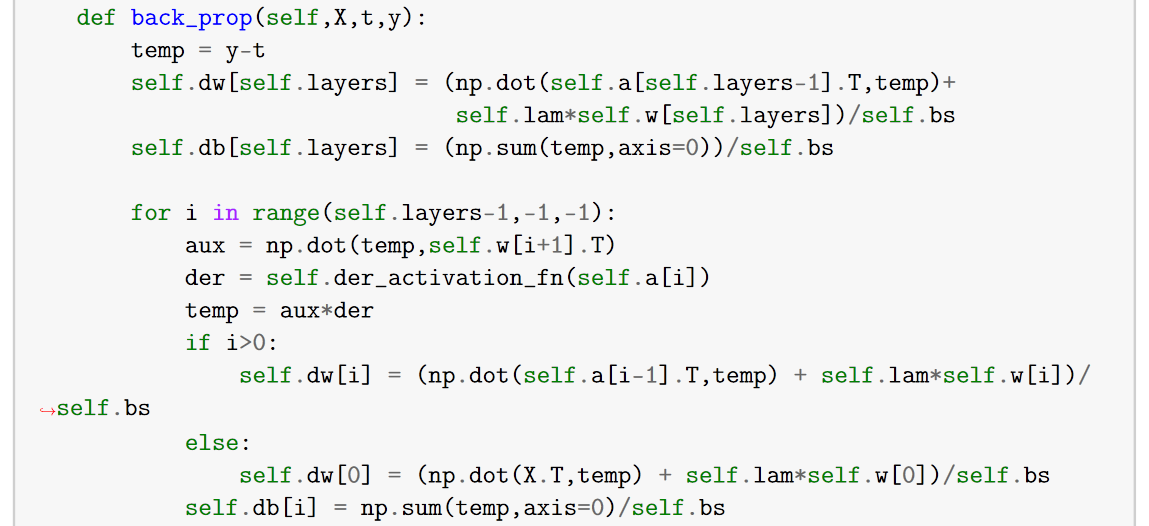
These functions are called using the following helper functions defined in our Neural Network class:



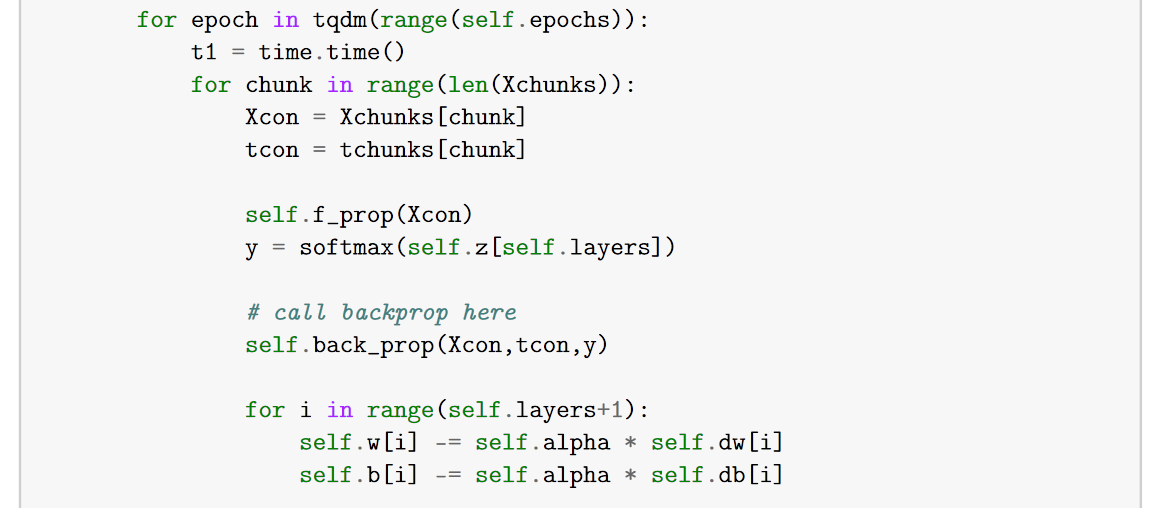
Next we move to the forward propagation step of our neural network:



For computing the gradients we perform the following backpropagation:



Now for updating our weights, we perform the following steps in our train function of the Neural Network class:



Here, the chunks are used for implementing the batch gradient descent.

The following are the tuneable parameters in the Neural Network class:

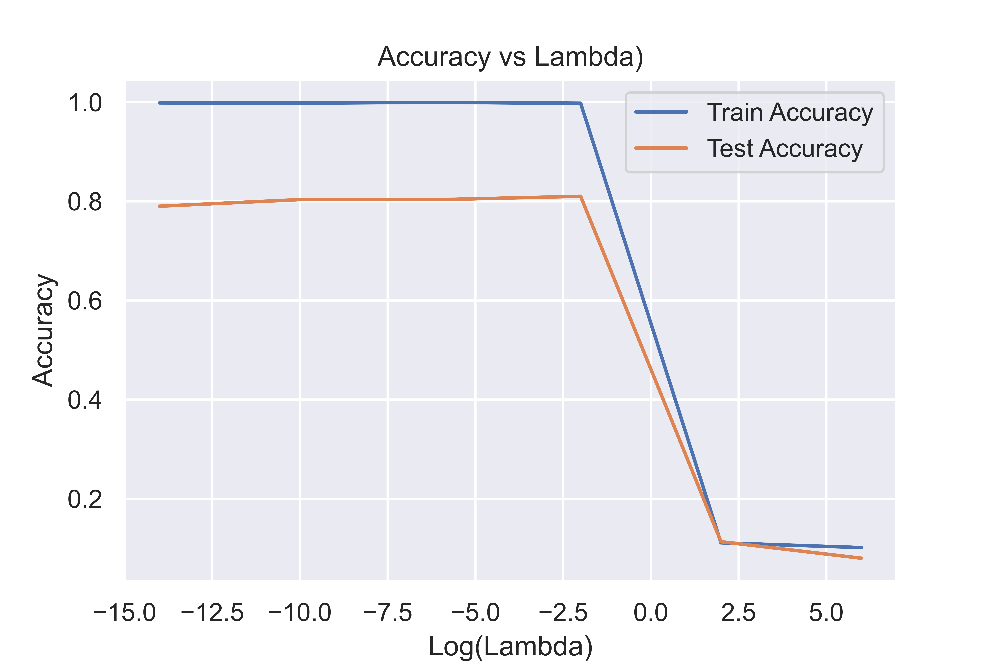
* Number of hidden layers
* Number of nodes (neurons) in each hidden layers
* Learning rate
* Batch Size
* Number of Epochs
* Regularization parameters
* Type of activation function to be used in the hidden layers
* Epsilon for early stopping

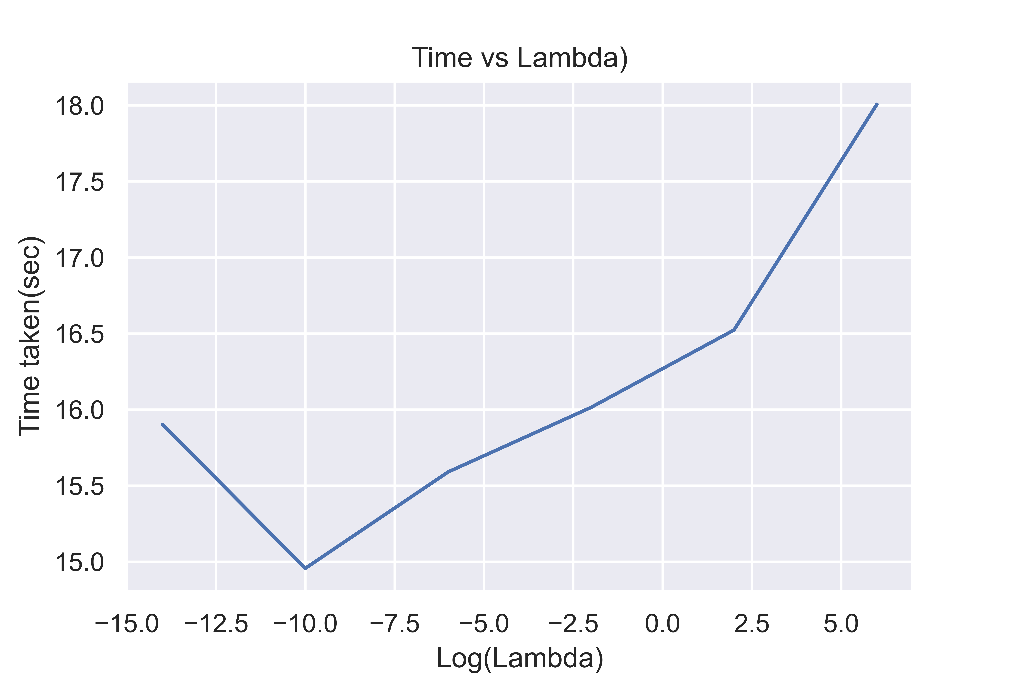
We also use early stopping in which we see if the train accuracy has been within a threshold for 3 iterations, we break the loop there.

In the next section, we see how the change in hyperparameters affect our train/test accuracies.

**Variation with lambda (# of layers = 2)**

First we fix the value of learning rate to be 0.004, number of layers to be 2, batch size of 16 and run for 100 epochs. Now by running a sweep over the values of lambda we get the following results:

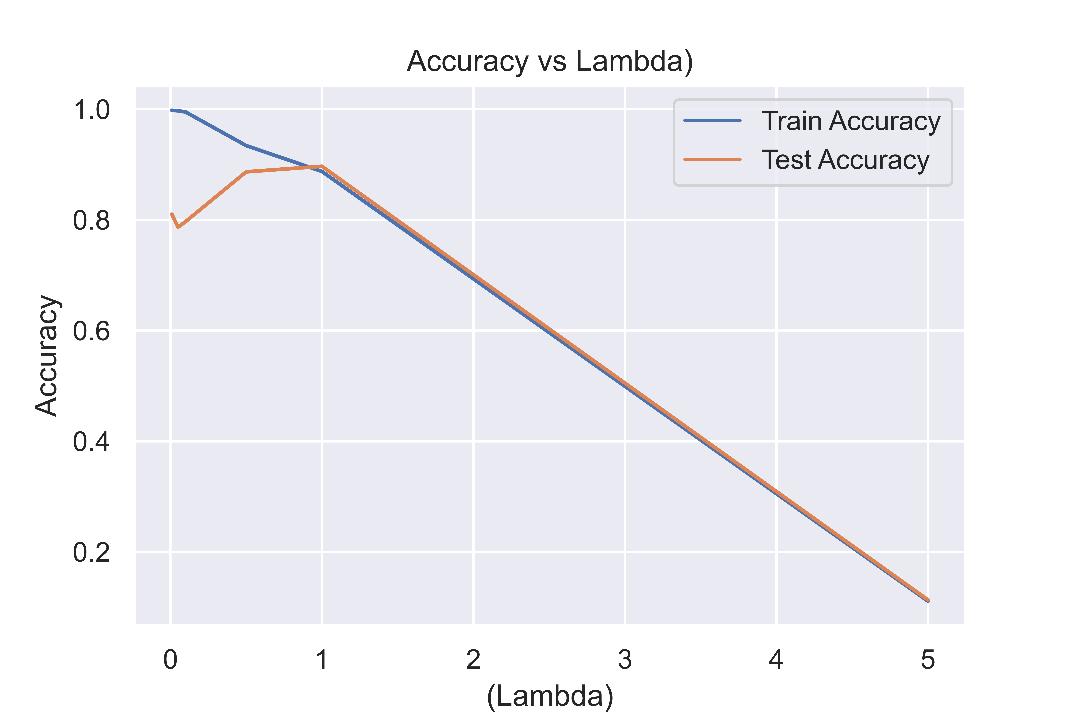
****

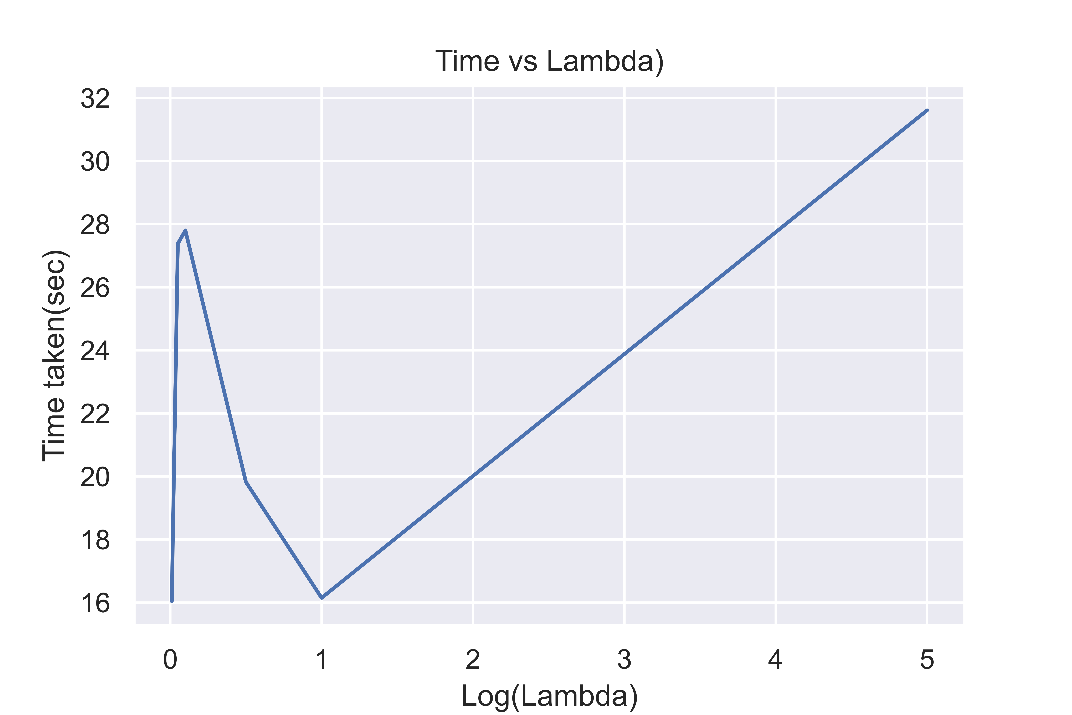
****

|  | **lam** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | -14 | 0.998519 | 0.790000 | 15.902141 |
| **1** | -10 | 0.998148 | 0.803333 | 14.956310 |
| **2** | -6 | 0.999259 | 0.803333 | 15.590128 |
| **3** | -2 | 0.997778 | 0.810000 | 16.014454 |
| **4** | 2 | 0.111111 | 0.113333 | 16.522977 |
| **5** | 6 | 0.101481 | 0.080000 | 18.006861 |

We can clearly see **overfitting** in the initial values of lambda and **underfitting** when the lambda becomes larger.

Tuning this further for values between 0 and 5 we see the following curve:

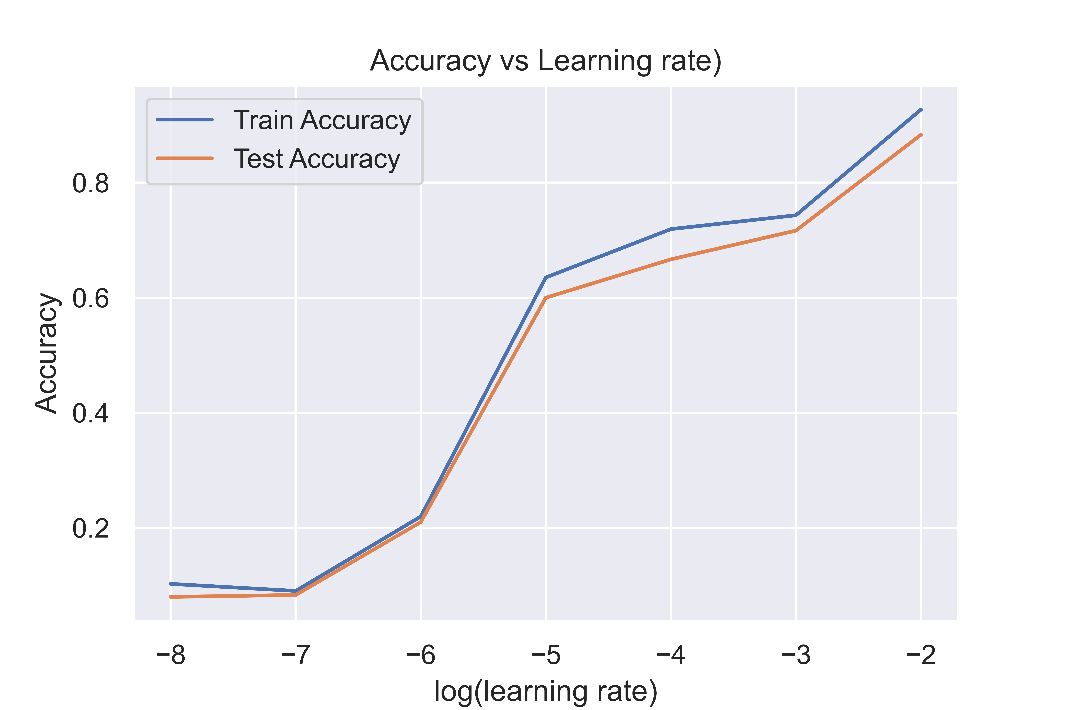
****

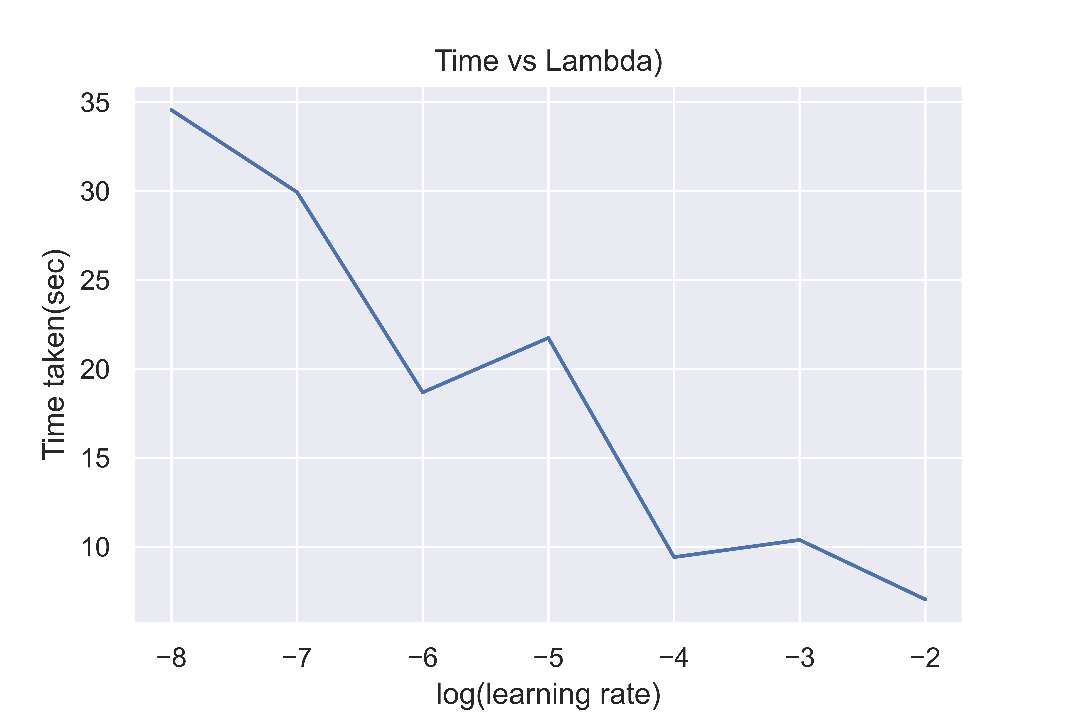
****

|  | **lam** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | 0.01 | 0.997778 | 0.810000 | 16.045384 |
| **1** | 0.05 | 0.997037 | 0.786667 | 27.378482 |
| **2** | 0.10 | 0.994815 | 0.796667 | 27.799001 |
| **3** | 0.50 | 0.934074 | 0.886667 | 19.820700 |
| **4** | 1.00 | 0.887407 | 0.896667 | 16.145968 |
| **5** | 5.00 | 0.111111 | 0.113333 | 31.607286 |

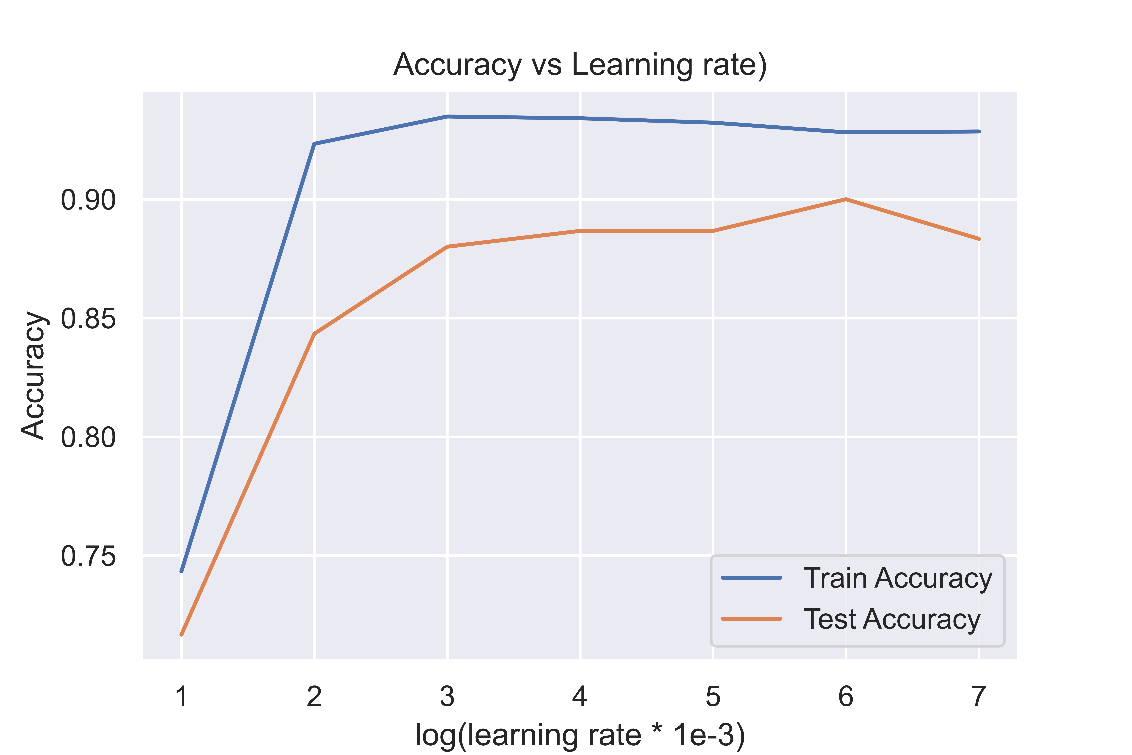
**Variation with learning rate**

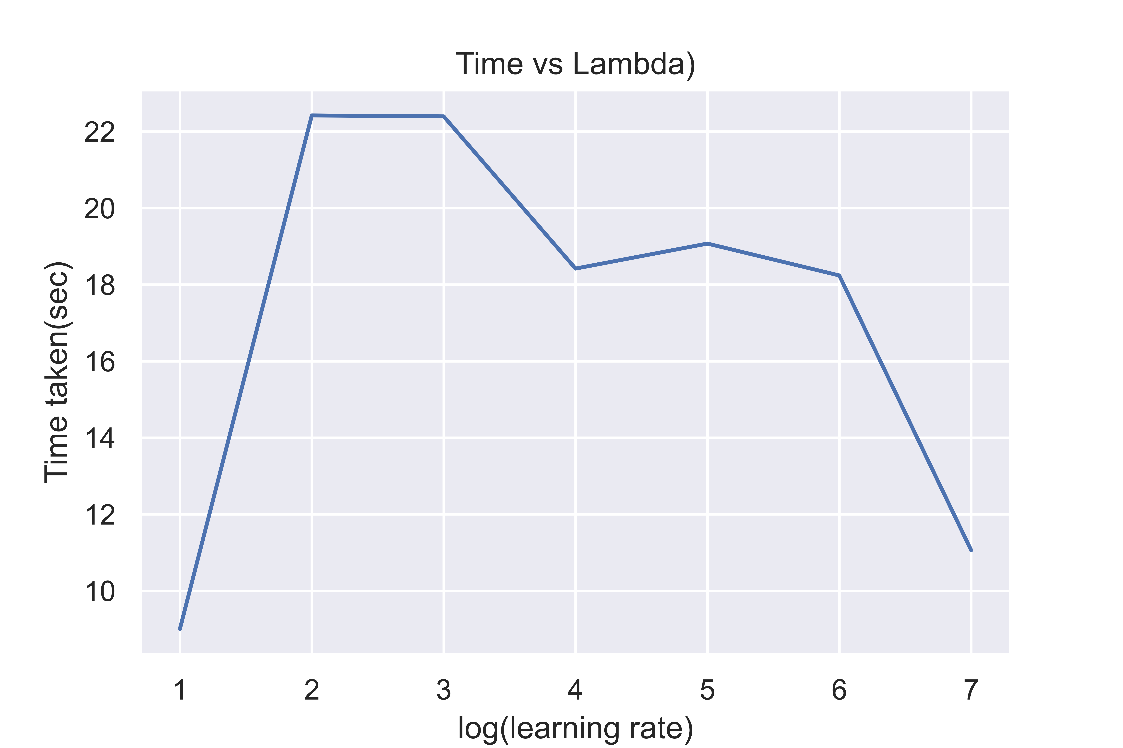
Here we fix the value of regularization lambda to be 0.5 and vary the learning rate. We get the following results:





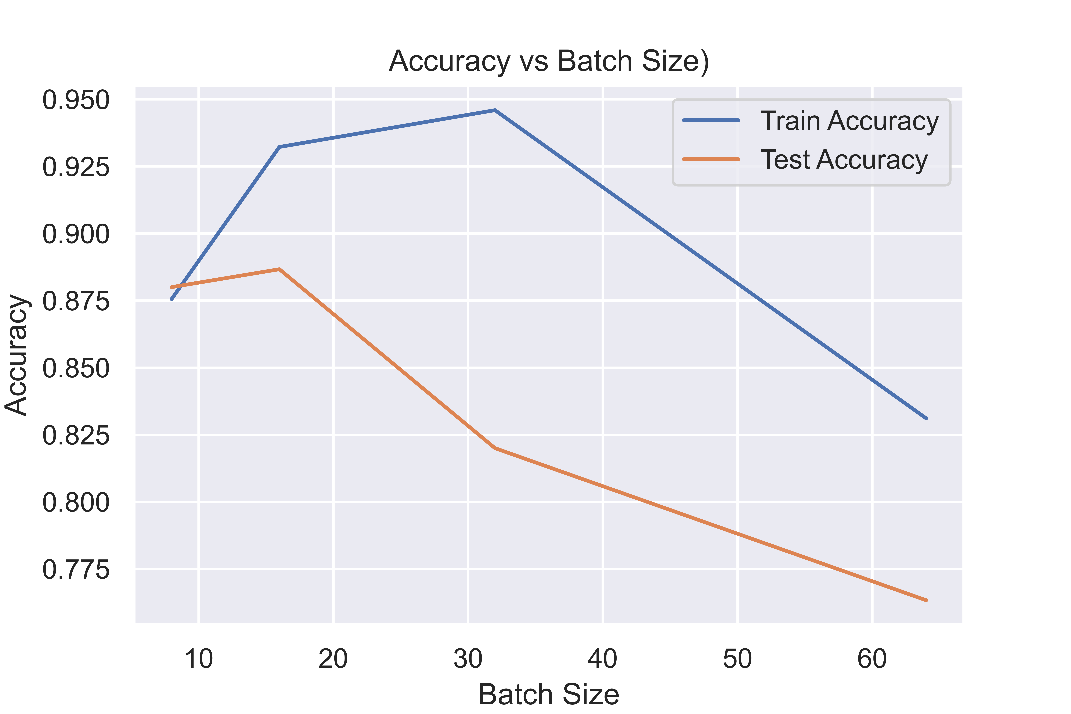
|  | **LR** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | 1.000000e-08 | 0.102593 | 0.080000 | 34.551819 |
| **1** | 1.000000e-07 | 0.090370 | 0.083333 | 29.931592 |
| **2** | 1.000000e-06 | 0.219630 | 0.210000 | 18.679262 |
| **3** | 1.000000e-05 | 0.635185 | 0.600000 | 21.738911 |
| **4** | 1.000000e-04 | 0.719259 | 0.666667 | 9.415470 |
| **5** | 1.000000e-03 | 0.743333 | 0.716667 | 10.381021 |
| **6** | 1.000000e-02 | 0.927037 | 0.883333 | 7.037894 |

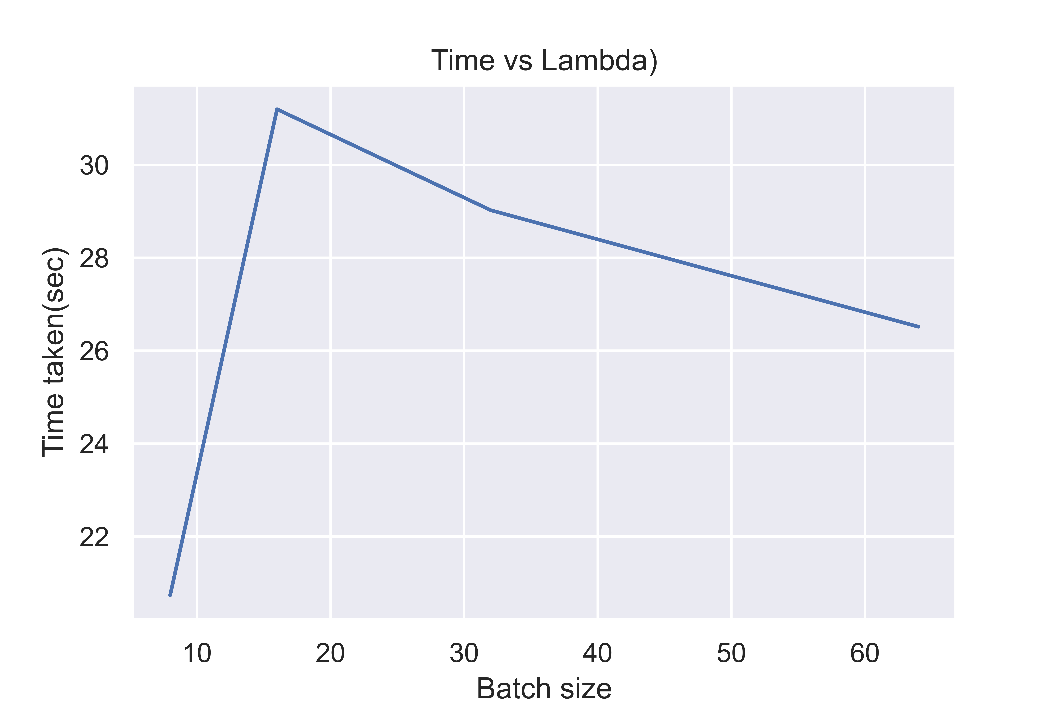




|  | **LR** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | 0.001 | 0.743333 | 0.716667 | 9.007805 |
| **1** | 0.002 | 0.923333 | 0.843333 | 22.419587 |
| **2** | 0.003 | 0.934815 | 0.880000 | 22.394939 |
| **3** | 0.004 | 0.934074 | 0.886667 | 18.418861 |
| **4** | 0.005 | 0.932222 | 0.886667 | 19.071711 |
| **5** | 0.006 | 0.928148 | 0.900000 | 18.239354 |
| **6** | 0.007 | 0.928519 | 0.883333 | 11.061211 |

**Variation of Batch size**

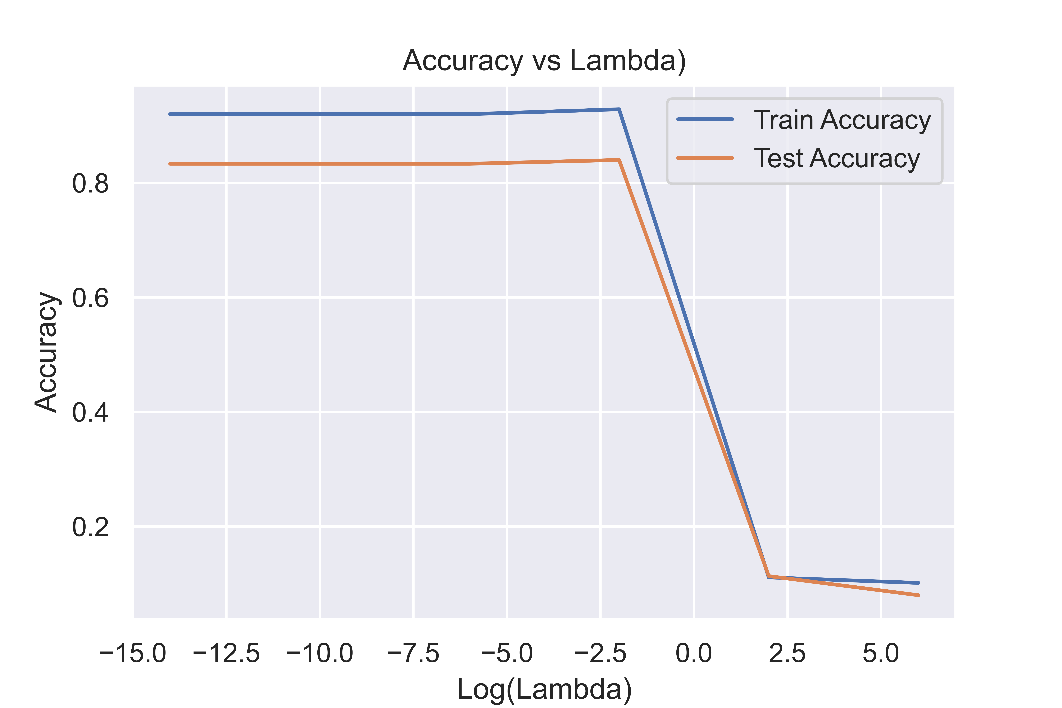
****

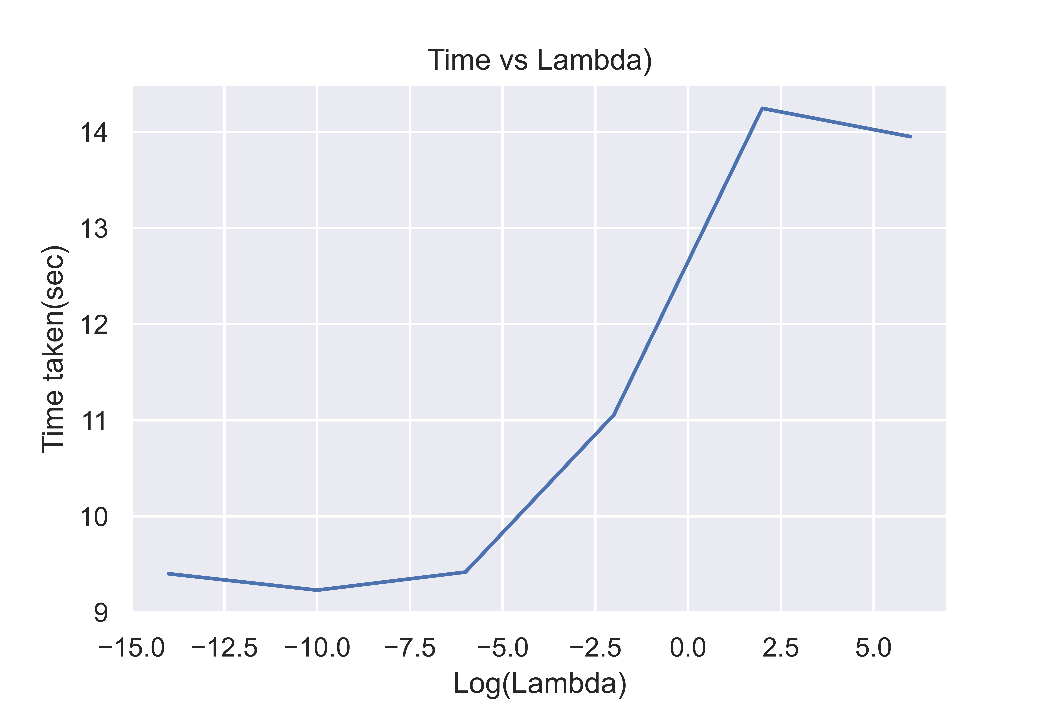
****

|  | **BS** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | 8 | 0.875556 | 0.880000 | 20.737143 |
| **1** | 16 | 0.932222 | 0.886667 | 31.194967 |
| **2** | 32 | 0.945926 | 0.820000 | 29.021666 |
| **3** | 64 | 0.831111 | 0.763333 | 26.515240 |

**Variation with lambda (# of layers = 1)**

First we fix the value of learning rate to be 0.004, number of layers to be 1, batch size of 16 and run for 100 epochs. Now by running a sweep over the values of lambda we get the following results:

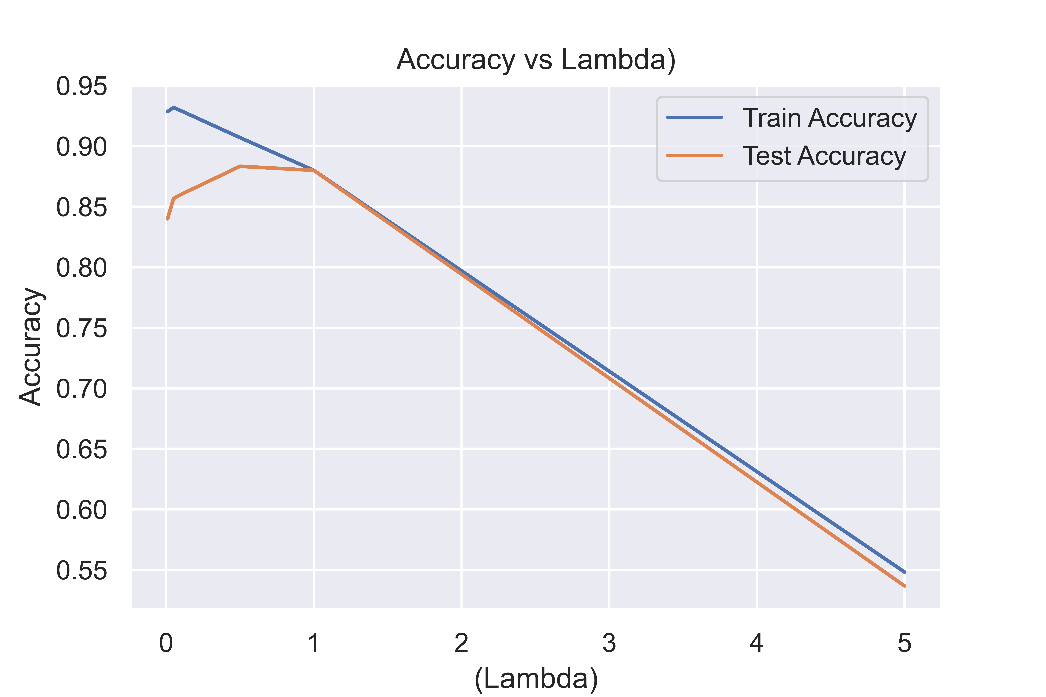
****

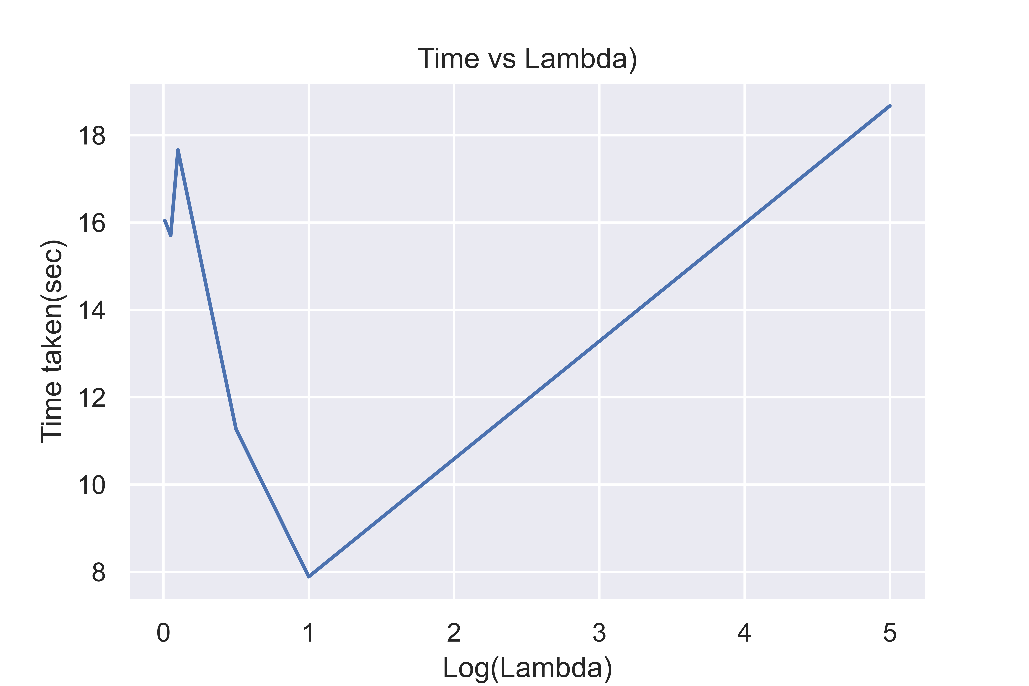
****

|  | **lam** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | -14 | 0.920000 | 0.833333 | 9.401528 |
| **1** | -10 | 0.920000 | 0.833333 | 9.230801 |
| **2** | -6 | 0.920000 | 0.833333 | 9.419060 |
| **3** | -2 | 0.928519 | 0.840000 | 11.055267 |
| **4** | 2 | 0.111111 | 0.113333 | 14.246521 |
| **5** | 6 | 0.101481 | 0.080000 | 13.951764 |

We can clearly see **overfitting** in the initial values of lambda and **underfitting** when the lambda becomes larger.

Tuning this further for values between 0 and 5 we see the following curve:

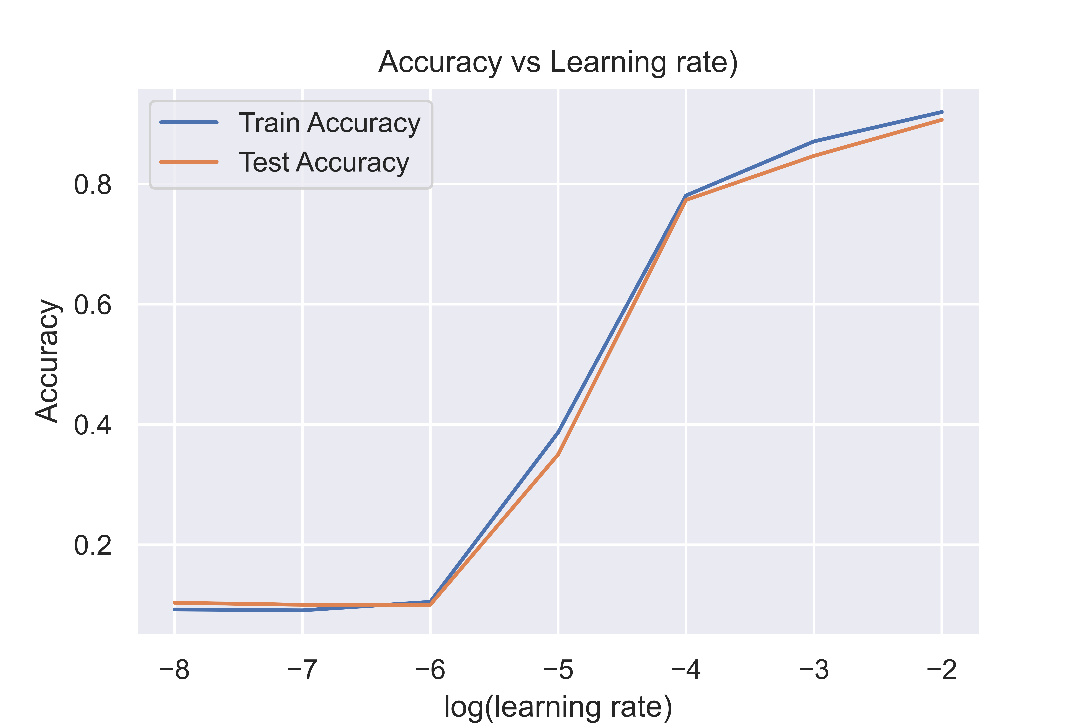
****

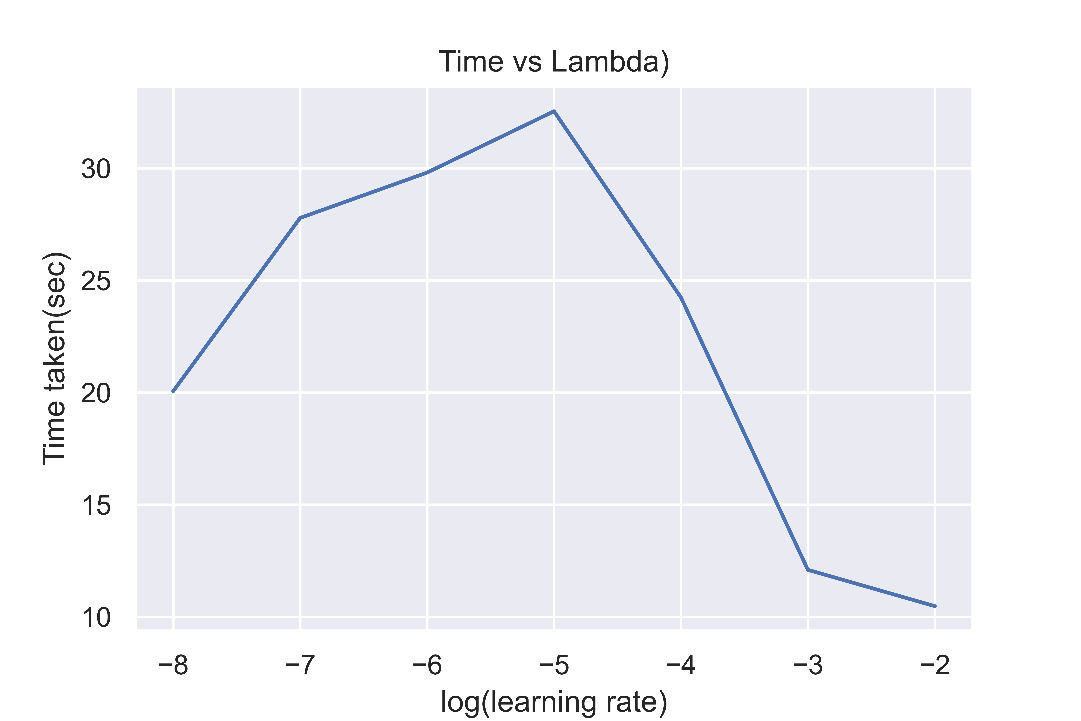
****

|  | **lam** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | 0.01 | 0.928519 | 0.840000 | 16.044469 |
| **1** | 0.05 | 0.931852 | 0.856667 | 15.705497 |
| **2** | 0.10 | 0.929259 | 0.860000 | 17.658879 |
| **3** | 0.50 | 0.907037 | 0.883333 | 11.265135 |
| **4** | 1.00 | 0.880000 | 0.880000 | 7.890082 |
| **5** | 5.00 | 0.548148 | 0.536667 | 18.670657 |

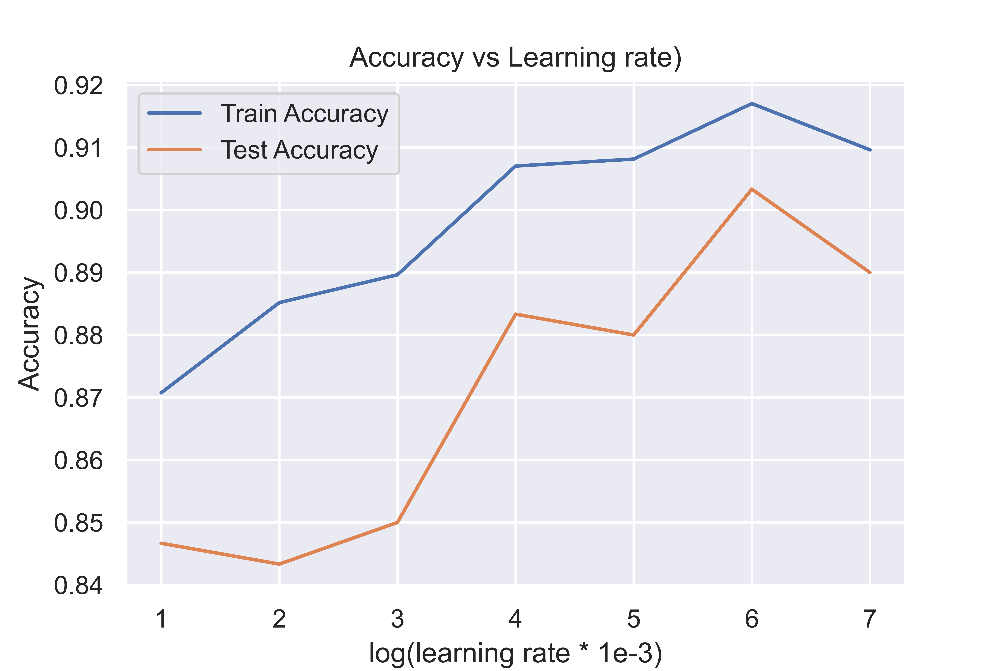
**Variation with learning rate**

Here we fix the value of regularization lambda to be 0.5 and vary the learning rate. We get the following results:





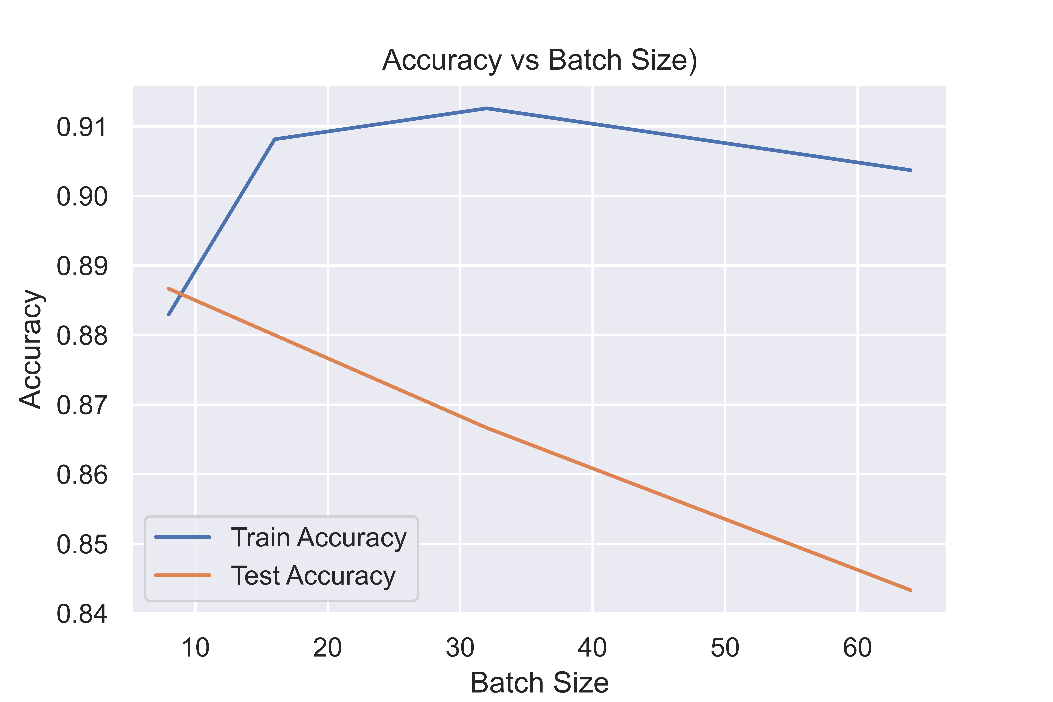
|  | **LR** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | 1.000000e-08 | 0.092222 | 0.103333 | 20.061172 |
| **1** | 1.000000e-07 | 0.090741 | 0.100000 | 27.785561 |
| **2** | 1.000000e-06 | 0.104815 | 0.100000 | 29.807645 |
| **3** | 1.000000e-05 | 0.386667 | 0.350000 | 32.541521 |
| **4** | 1.000000e-04 | 0.780741 | 0.773333 | 24.240064 |
| **5** | 1.000000e-03 | 0.870741 | 0.846667 | 12.096484 |
| **6** | 1.000000e-02 | 0.919630 | 0.906667 | 10.478140 |

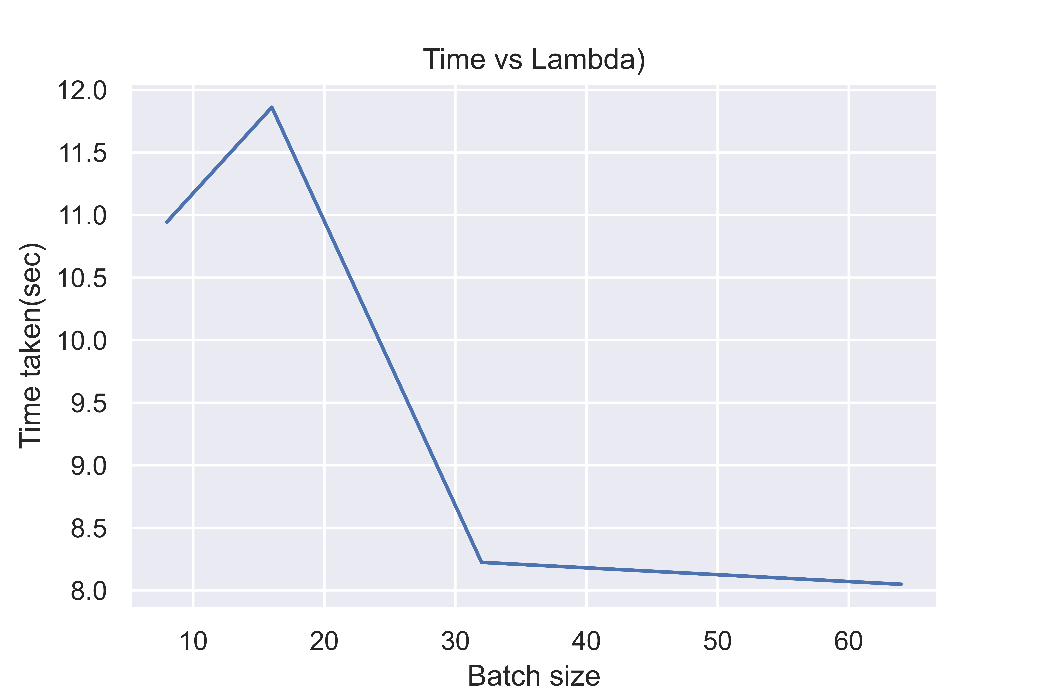




|  | **LR** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | 0.001 | 0.870741 | 0.846667 | 13.978901 |
| **1** | 0.002 | 0.885185 | 0.843333 | 10.872702 |
| **2** | 0.003 | 0.889630 | 0.850000 | 7.490244 |
| **3** | 0.004 | 0.907037 | 0.883333 | 16.960423 |
| **4** | 0.005 | 0.908148 | 0.880000 | 13.727301 |
| **5** | 0.006 | 0.917037 | 0.903333 | 16.368423 |
| **6** | 0.007 | 0.909630 | 0.890000 | 13.126486 |

**Variation with Batch size**

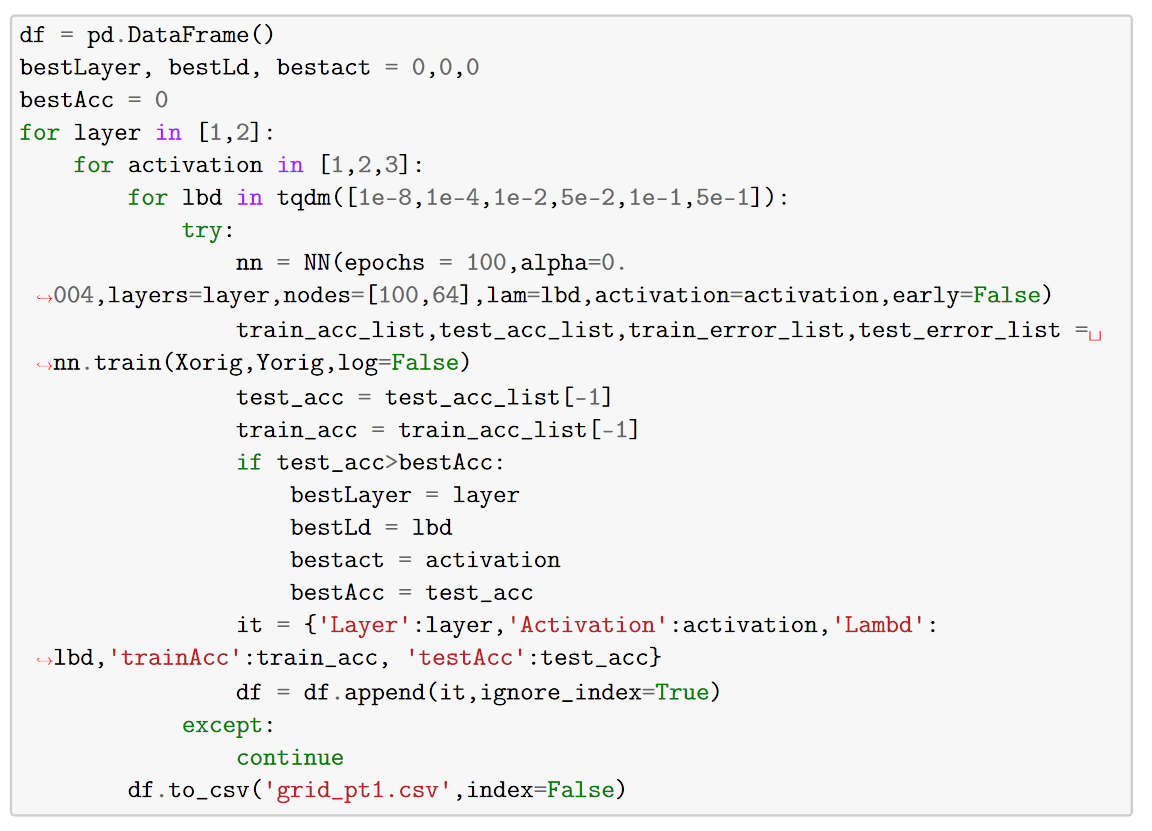
****

****

|  | **BS** | **train\_acc** | **test\_acc** | **time\_taken** |
| --- | --- | --- | --- | --- |
| **0** | 8 | 0.882963 | 0.886667 | 10.944870 |
| **1** | 16 | 0.908148 | 0.880000 | 11.861279 |
| **2** | 32 | 0.912593 | 0.866667 | 8.223766 |
| **3** | 64 | 0.903704 | 0.843333 | 8.048591 |

**Grid Search:**

We fix the value of learning rate to be 0.004 and the batch size to be 16 and then vary the other hyperparameters based on the ranges derived from previous plots.



The following are the top results that we get from this:

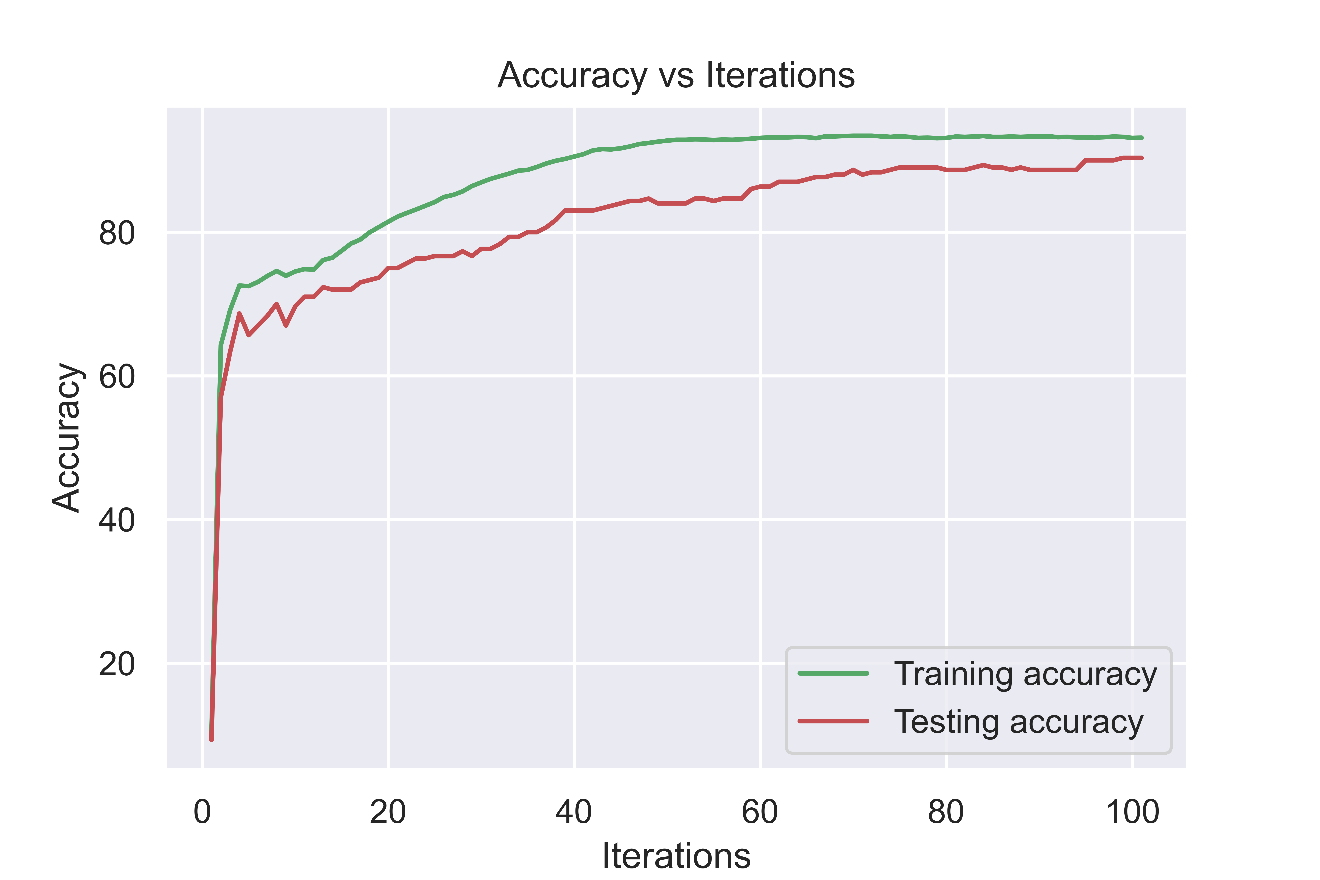
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Activation | Lambda | # Layer | Test Acc. | Train Acc. |
| Relu | 0.5 | 1 | 0.903333 | 0.918519 |
| Tanh | 0.5 | 1 | 0.903333 | 0.902593 |
| Relu | 0.5 | 2 | 0.903333 | 0.931481 |
| Tanh | 0.5 | 2 | 0.893333 | 0.914815 |

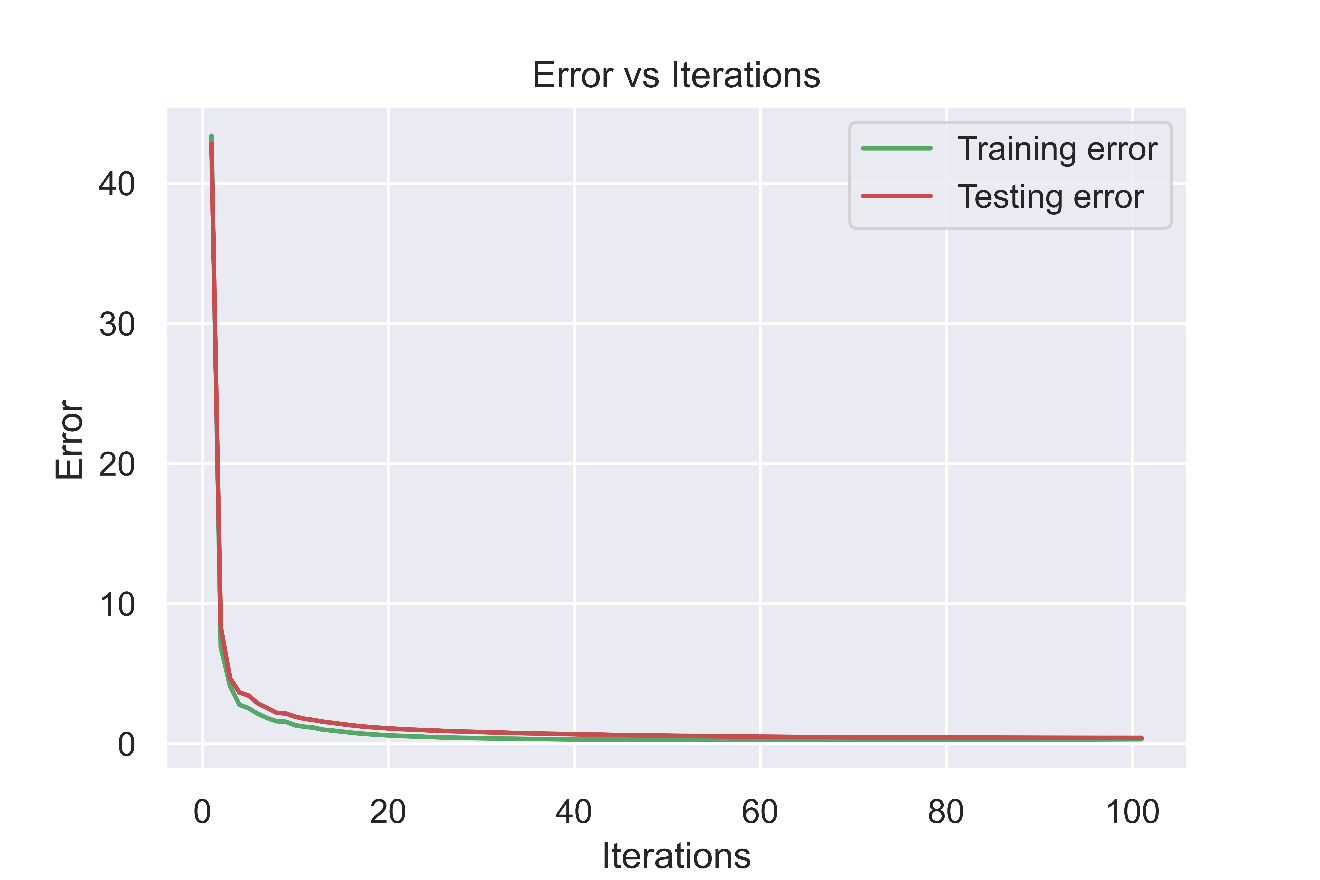
Finally, we pick the following set of hyperparameters:

|  |  |
| --- | --- |
| Hyperparameter | Value |
| Learning rate | 0.004 |
| Batch Size | 16 |
| Activation Function | Relu |
| # of layers | 2 |
| Nodes in the layers | (100,64) |
| Regularization Lambda | 0.5 |

Using **5-fold cross validation**, we get an average test Accuracy of **0.9033.**

The following are the training curves that we get for this hyperparameter setting:

****

****

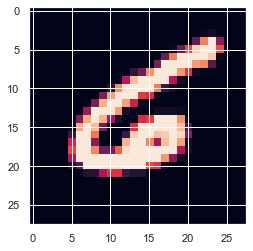
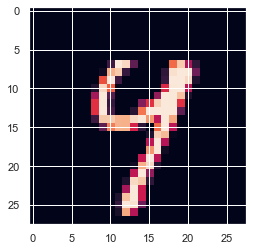
**Error Analysis**

We get the following confusion matrix for this set of hyperparameters:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Predicted | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | All |
| Actual |  |  |  |  |  |  |  |  |  |  |  |
| 0 | 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 26 |
| 1 | 0 | 24 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 27 |
| 2 | 0 | 0 | 26 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 28 |
| 3 | 2 | 0 | 1 | 38 | 0 | 0 | 0 | 0 | 0 | 0 | 41 |
| 4 | 0 | 0 | 0 | 0 | 18 | 0 | 0 | 0 | 0 | 3 | 21 |
| 5 | 1 | 0 | 0 | 1 | 0 | 26 | 0 | 0 | 2 | 0 | 30 |
| 6 | 1 | 0 | 0 | 0 | 1 | 2 | 24 | 0 | 0 | 0 | 28 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 32 | 0 | 1 | 33 |
| 8 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 32 | 0 | 36 |
| 9 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 27 | 30 |
| All | 30 | 25 | 28 | 40 | 21 | 29 | 26 | 35 | 35 | 31 | 300 |

Following are some of the data-points where the model is going wrong:

Original number: 6, Prediction: 4; Original number: 4, Prediction: 9

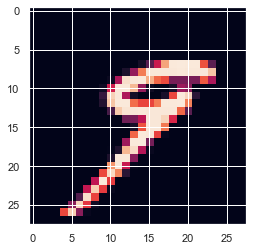
We can see that these images which the model is classifying incorrectly are indeed difficult to categorize. In the first picture, it got confused between 6 and 4 since the circular portion of 6 is not drawn perfectly and indeed would have looked more like a 4 had it been extended downwards. The second picture also can be easily confused with a 4 and a 9 as most of the features are similar to a 9 except the top being connected.

The model is able to extract the features but sometimes struggles when some features are very common.

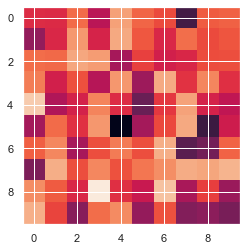
**Representations learned by the layers**

By plotting the various representations learned by the intermediate layers, we get the following:

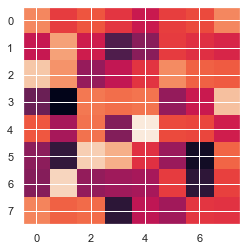
Input image:



Representation learned by layer 1 (100 nodes):



Representation learned by layer 2 (64 nodes):



**Using a standard library**

If we use a standard library with the same hyperparameters and the architecture, we get the following results:

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_3 (Dense) (None, 100) 78500

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_4 (Dense) (None, 64) 6464

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_5 (Dense) (None, 10) 650

=================================================================

Total params: 85,614

Trainable params: 85,614

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

After running for 100 epochs and early stopping with best val\_accuracy:

Epoch 37/100

2700/2700 [==============================] - 0s 123us/sample - loss: 0.0629 - accuracy: 0.9537 - val\_loss: 0.3278 - val\_accuracy: 0.9167

We get a **validation accuracy of 0.9167** and a training accuracy of 0.9537.

The validation accuracy is very close to what we got using our own implementation. The few differences that occur might be due to the following:

1. Better initialization of the random weight matrices leading to better fitting.
2. Training time is also faster (33.25 second vs 41 second) because of better implementation using tensors in Tensorflow.

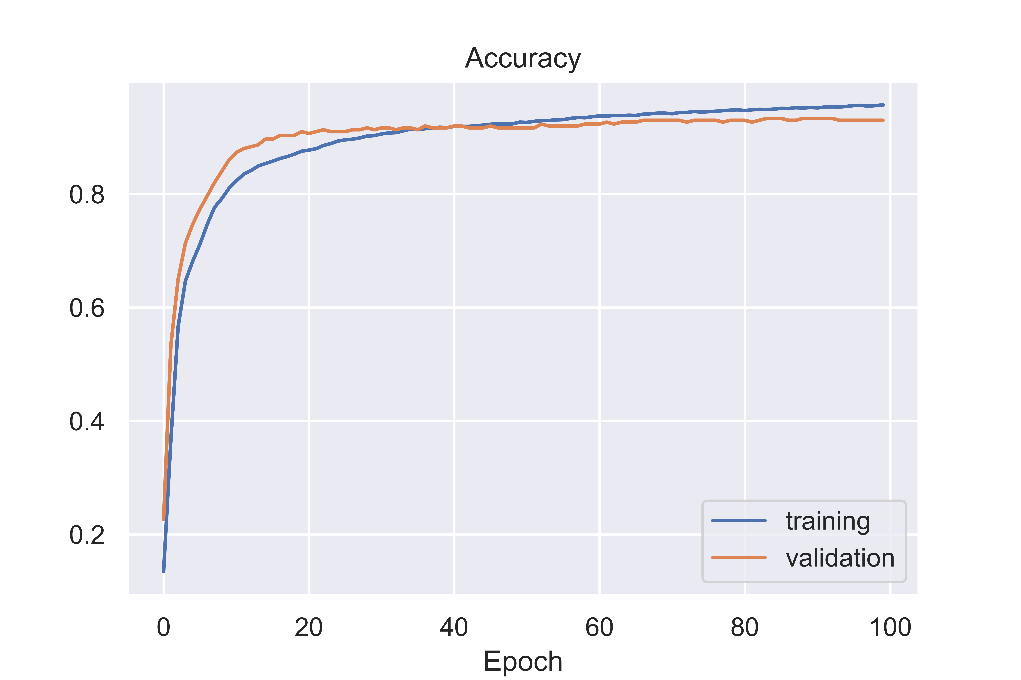
**Part 1B**

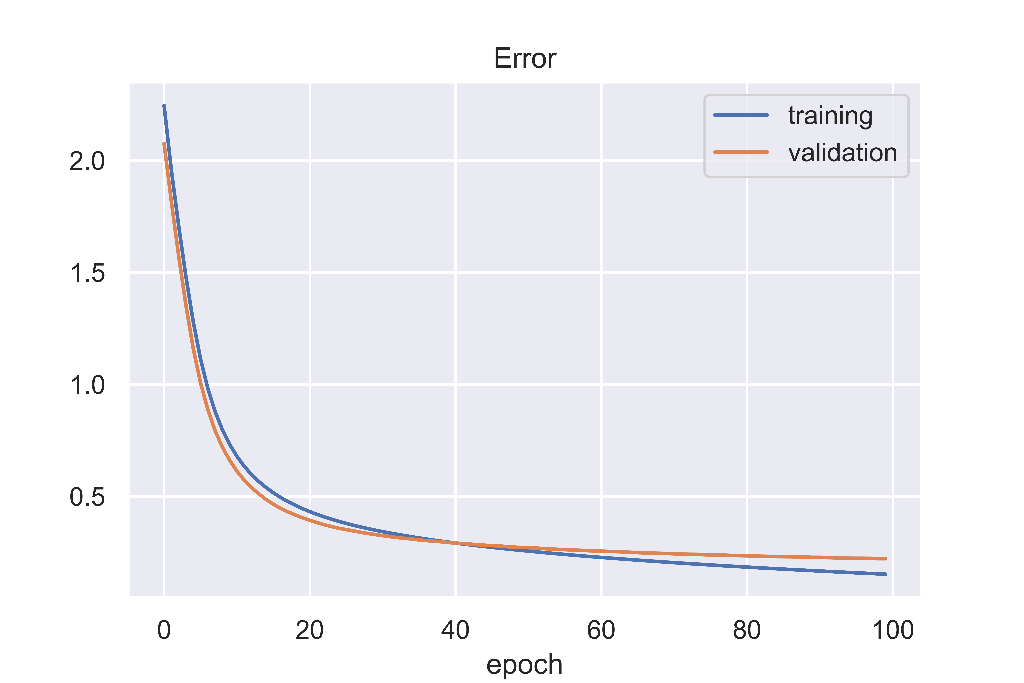
Here we use the PCA representation given to us in the second assignment.

We first tune the parameters using the same approach as followed in part 1A. After tuning, we compare the performances across various experiments as follows:

|  |  |  |
| --- | --- | --- |
| Setting | Training Accuracy | Testing Accuracy |
| Raw pixels + 2 hidden layers | 95.37% | 91.67% |
| PCA + 0 hidden layers | 87.15% | 88.67% |
| PCA + 1 hidden layer | 94.44% | 91.67% |
| PCA + 2 hidden layers | 95.70% | 93.00% |

We also get the following training plots for the last case:





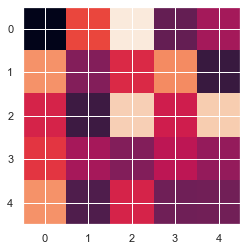
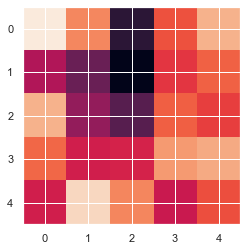
We see that adding hidden layers do help in increasing the test accuracy of the model. Simple logistic regression was not able to perform very well but once we added an additional layer, the accuracy jumped past 90%.

Moreover, we see that using the raw pixels with 2 hidden layers has similar performance as that in the case of PCA + 1 hidden layer. This shows that the PCA representation is very effective and is probably an embedding extracted from an intermediate layer of a dense neural network.

**Representations learned:**

We also try plotting the representations learned by the neural network with along with the PCA representation given to us:

**Input label: 9** **Input label: 5**

These representations are similar to what we were getting by plotting the intermediate layers of the neural network implemented in problem 1A.

**Part 2A (Using CNN)**

In this part of the assignment, we use Tensorflow library to implement Convolutional Neural Networks (CNN) for the problem. We use the full MNIST dataset available on the internet. We implement the following architecture:

Model: "sequential\_4"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_5 (Conv2D) (None, 26, 26, 32) 320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_5 (MaxPooling2 (None, 13, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_4 (Flatten) (None, 5408) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_7 (Dense) (None, 25) 135225

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_2 (Dropout) (None, 25) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_8 (Dense) (None, 10) 260

=================================================================

Total params: 135,805

Trainable params: 135,805

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

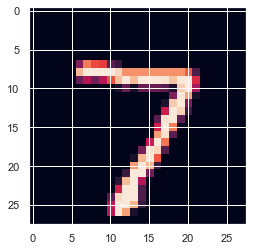
After training for 10 epochs, we get the following results:

Epoch 10/10

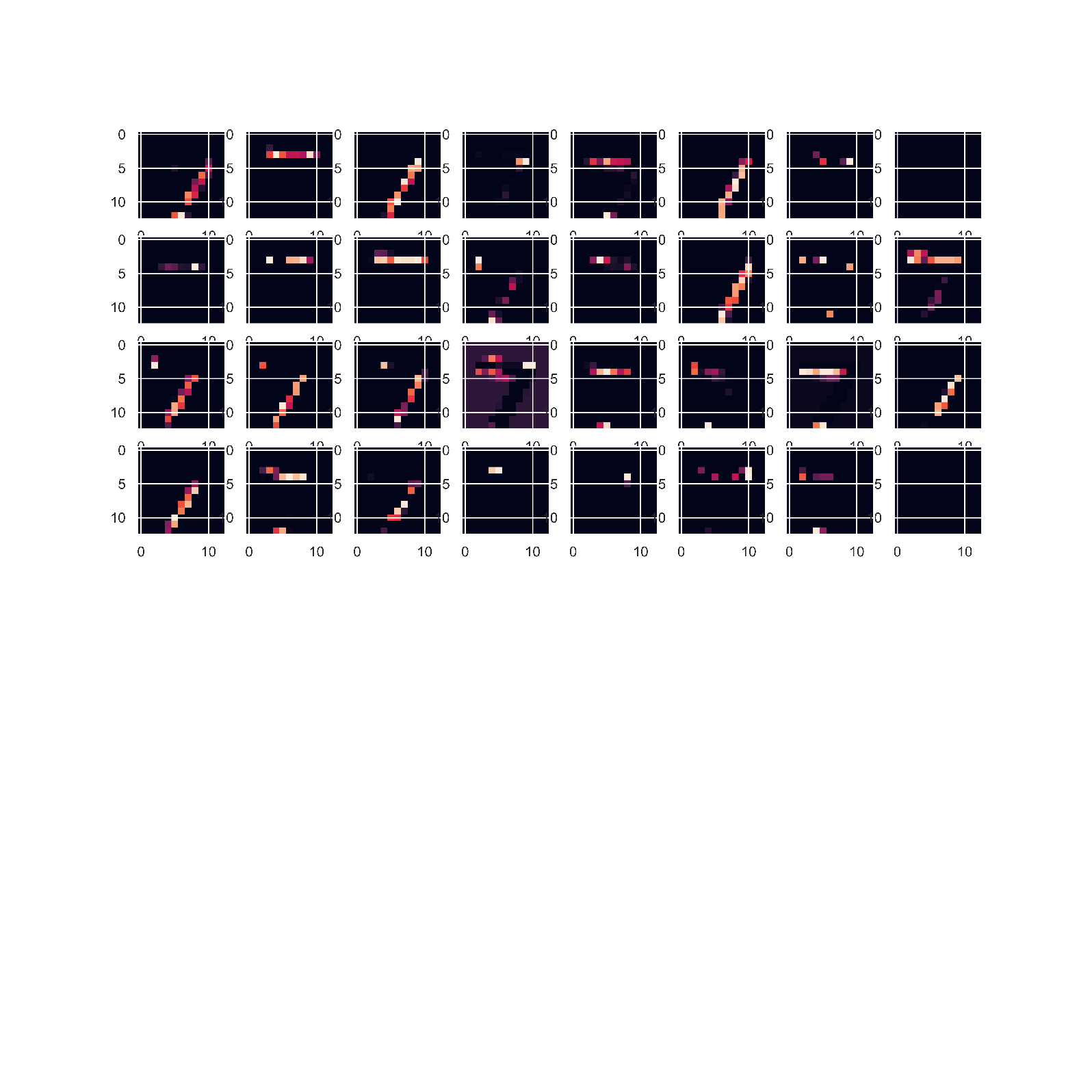
54000/54000 [==============================] - 22s 399us/sample - loss: 0.2172 - accuracy: 0.9221 - val\_loss: 0.0515 - val\_accuracy: 0.9867

Here, we get a **validation accuracy of 98.67%** and a **test accuracy of 98.35%** which is way greater than what we were getting without using a convolutional layer.

Next, we try to visualize the representations learned by the layers of the neural net especially the ones learned by the filters of the conv2D layers. We feed in the following image to the CNN model:

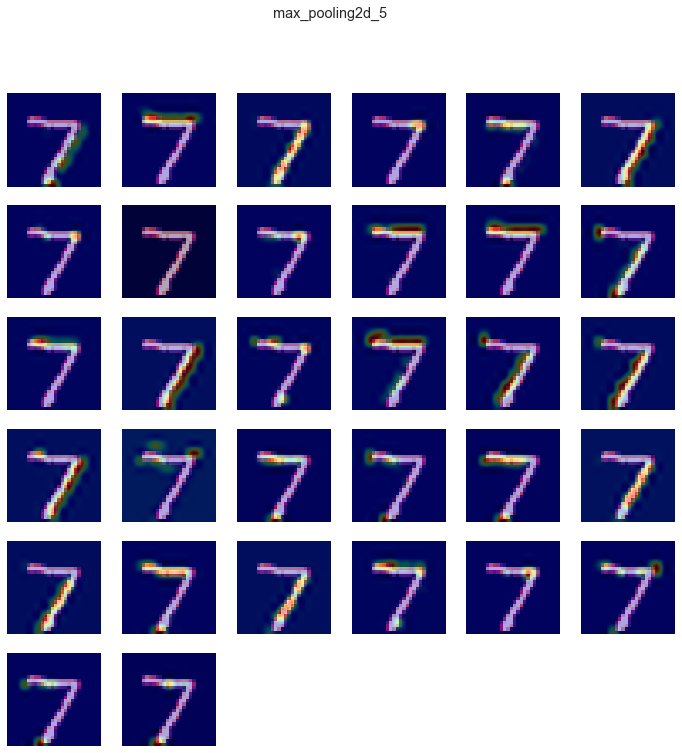


Following are the representations learned by the Max pooling layer:



We observe that the Conv2D layer is able to learn the representation of edges of the digit given in the input. This is possible because we have not flattened out the input image and there’s a filter moving on the 2D image which is capable of learning this.

We also try plotting the heat-map (trigger points) using the Keract library (reference given at the end):

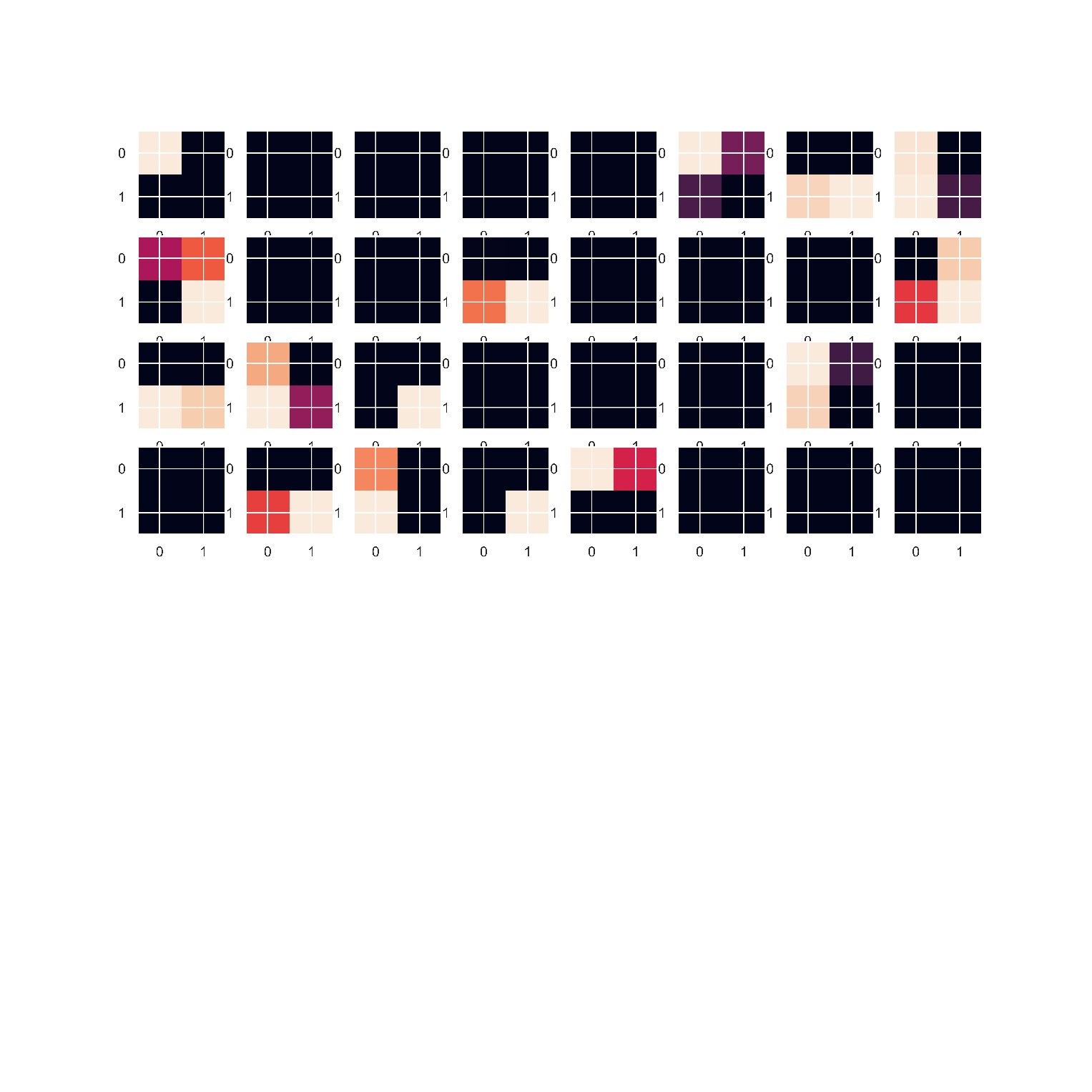


We also try varying the size of the filters (kernel size) of the conv2D layer to see the effect it has on the model performance:

|  |  |
| --- | --- |
| **Kernel size** | **Test Accuracy** |
| (2x2) | 97.68% |
| (3x3) | 98.35% |
| (4x4) | 98.48% |
| (8x8) | 98.64% |
| (24x24) | 96.90% |

We see that on increasing the kernel size, the accuracy increases but after a certain limit, it starts falling as the filters become too large and are not able to capture the minute information required for categorization. Hence, there is a sweet spot in between.

For the 24x24 case, the following are the representations learned by the max pooling layer:



We observe that as the kernel size is increased beyond a limit, the representations start making way lesser sense as before.

**Part 2B (Auto-encoders)**

In this part, we experiment using Auto-encoders in Tensorflow. Auto-encoders are used to reconstruct an input image after passing through a neural network.

We implement the following two models:

**The encoder:**

Model: "model\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) [(None, 784)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 32) 25120

=================================================================

Total params: 25,120

Trainable params: 25,120

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**The decoder:**

Model: "model\_2"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_2 (InputLayer) [(None, 32)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 784) 25872

=================================================================

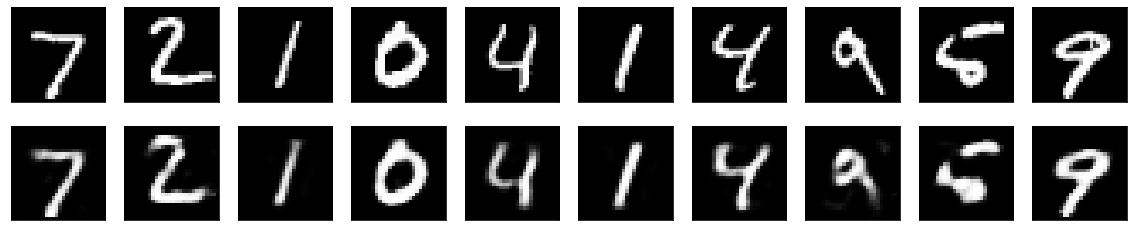
Total params: 25,872

Trainable params: 25,872

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Here, the model takes in a 784 dimensional input and encodes it (embedding) to a 32 dimensional space. Then the decoder maps it back to a 784 dimensional space (same as the input dimension). Now, we visualize the performance of this model:



On the top are the input images and the bottom row are the reconstructed images after passing through our auto-encoder model.

So what we do now is, after training this auto-encoder model, we feed this model the data we had for part 1A containing 3000 points (784 dimensional) and get the output of the intermediate layer (32 dimensional). This way we have mapped down the 784 dimensional data to 32 dimensional data by using a model trained in an unsupervised way.

Now we feed this 32 dimensional data to the same neural network we created in part 1A, except now we have the input of 32 dimensions.

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_7 (Dense) (None, 100) 3300

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_8 (Dense) (None, 64) 6464

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_9 (Dense) (None, 10) 650

=================================================================

Total params: 10,414

Trainable params: 10,414

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

After training this model, we get the following:

Epoch 65/65

2700/2700 [==============================] - 0s 74us/sample - loss: 0.0142 - accuracy: 0.9996 - val\_loss: 0.2483 - val\_accuracy: 0.9267

The **validation accuracy** we get is **92.67%**, which is even greater than what we got before. Also, it took only 11 seconds to train.

So in a nutshell, we mapped down the 784 dimensional data to 32 dimensions using sparse auto-encoders working in an unsupervised setting. This was trained on the full MNIST data. Next, we feed our n=3000 data given to us for 1A to this auto-encoder to get the 32 dimensional data from the intermediate layer. Finally, we fed this data to our dense neural network to perform the classification task.

The accuracy of this can be further improved if we use CNN in the encoder to model more complex features into this embedding.

**References**

1. <https://blog.keras.io/building-autoencoders-in-keras.html>
2. <https://towardsdatascience.com/how-to-make-an-autoencoder-2f2d99cd5103>
3. <https://www.kaggle.com/arpitjain007/guide-to-visualize-filters-and-feature-maps-in-cnn>
4. <https://keras.io/examples/vision/mnist_convnet/>
5. <https://peterroelants.github.io/posts/neural-network-implementation-part04/>
6. <http://cs229.stanford.edu/summer2020/cs229-notes-deep_learning.pdf>
7. <https://www.machinecurve.com/index.php/2019/12/02/visualize-layer-outputs-of-your-keras-classifier-with-keract/>
8. <https://pypi.org/project/keract/>
9. https://stackoverflow.com/questions/41711190/keras-how-to-get-the-output-of-each-layer