Artificial Neural Networks and Deep Architecture, DD2437

SHORT REPORT

Lab Assignment 4: Deep Network Architectures with Restricted Boltzmann Machines and Autoencoders

Author :

Putra, Ramadhani Pamapta

Rahgozar, Parastu

Setiawan, Stanley

1. Aim and Objectives

After the lab completion, students are expected to be able to know:

- Explain key ideas underlying the learning process of RBMs and autoencoders
- Apply basic algorithms for greedy training of RBM and autoencoder layers with the use of commonly available deep learning libraries
- Design multi-layer neural network architectures based on RBM and autoencoder layers for classification problems
- Study the functionality (including generative aspects) and test the performance of low-scale deep architectures developed using RBMs (deep belief networks, DBNs) and autoencoders.

2. Scope

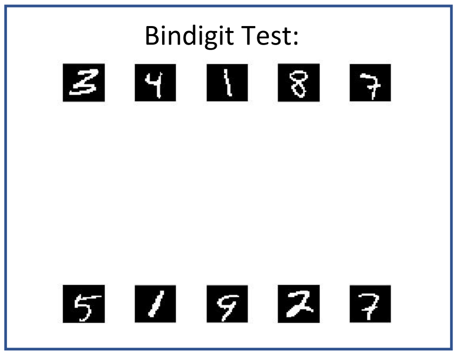
In this assignment, we used deep learning libraries for matlab or python. In Matlab, we have M. Tanaka's Deep Neural Network Toolbox; and in python we have scikit, tensorflow and keras. At first, we used matlab and M. Tanaka's library. However, because we have encountered some problems and we have difficulties to understand and trying to see the documentations, we decided to use another library. In the end, we used python with Tensorflow, Keras and Scikit Learn as we can understand more on how to use it and how the library works.

3. Tools Used

- Python with Jupyter
- Tensorflow
- Keras
- ScikitLearn

4. Results/Findings

4.1 RBM and Autoencoder features for binary-type MNIST images



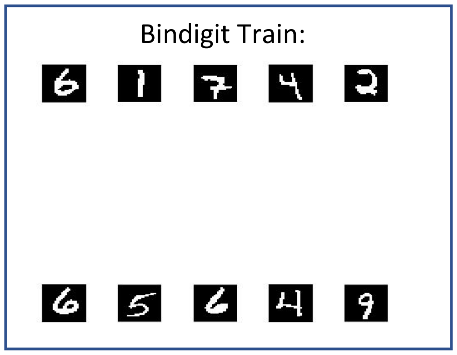


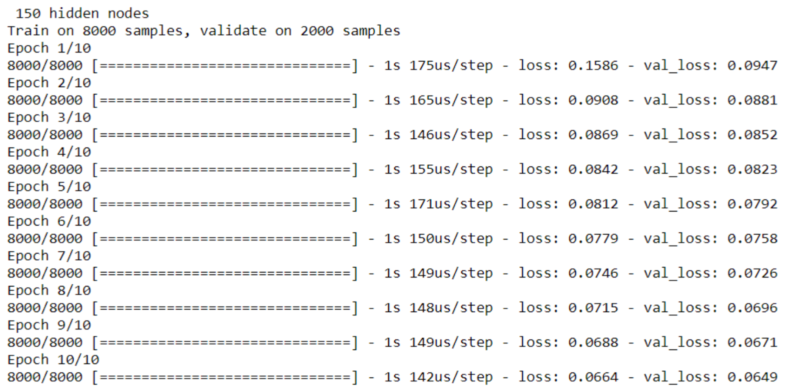
Figure 1 (Left) shows the images reconstructed from the Test MNIST data. (Right) from the Train MNIST data

From the MNIST images database, we are given the Test and Train data of the digits. It consists of digit 0 – 9 in several forms which can be seen in Figure 1. However, in Figure 1 we only showed the first 10 digits from both the test and train of MNIST data base.

After we know what are inside the MNIST data, we then proceed to build an autoencoder and a RBM network with different hidden nodes. The hidden nodes that we used were 50, 75, 100 and 150 hidden nodes. In addition, all of our learning rate was 0.01

4.1.1 Autoencoder

The suitable library that we can use for autoencoder was by using Scikit Learn for Python. First, we need to adjust the input dimensions (784 input) and place it to the encoder and decoder. After we have successfully set the parameter, we can then compile and train our network according to the desired hidden nodes. Output of the library that we used after we train the network is shown in Figure 2.



```
 150 hidden nodes
Train on 8000 samples, validate on 2000 samples
Epoch 1/10
8000/8000 [==============================] - 1s 175us/step - loss: 0.1586 - val_loss: 0.0947
Epoch 2/10
8000/8000 [==============================] - 1s 165us/step - loss: 0.0908 - val_loss: 0.0881
Epoch 3/10
8000/8000 [==============================] - 1s 146us/step - loss: 0.0869 - val_loss: 0.0852
Epoch 4/10
8000/8000 [==============================] - 1s 155us/step - loss: 0.0842 - val_loss: 0.0823
Epoch 5/10
8000/8000 [==============================] - 1s 171us/step - loss: 0.0812 - val_loss: 0.0792
Epoch 6/10
8000/8000 [==============================] - 1s 150us/step - loss: 0.0779 - val_loss: 0.0758
Epoch 7/10
8000/8000 [==============================] - 1s 149us/step - loss: 0.0746 - val_loss: 0.0726
Epoch 8/10
8000/8000 [==============================] - 1s 148us/step - loss: 0.0715 - val_loss: 0.0696
Epoch 9/10
8000/8000 [==============================] - 1s 149us/step - loss: 0.0688 - val_loss: 0.0671
Epoch 10/10
8000/8000 [==============================] - 1s 142us/step - loss: 0.0664 - val_loss: 0.0649
```

Figure 2 Output when we have trained the network. In this example, we used 150 Hidden nodes with 10 iterations

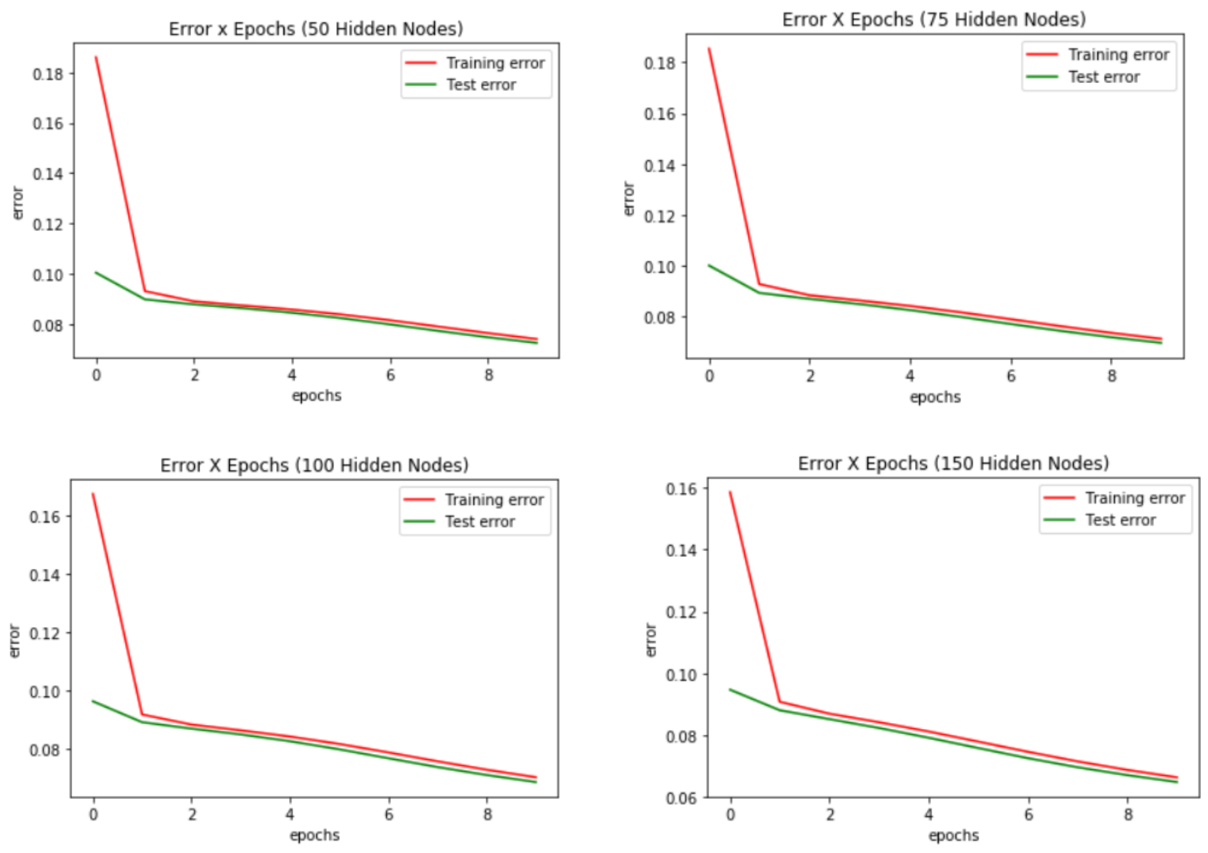
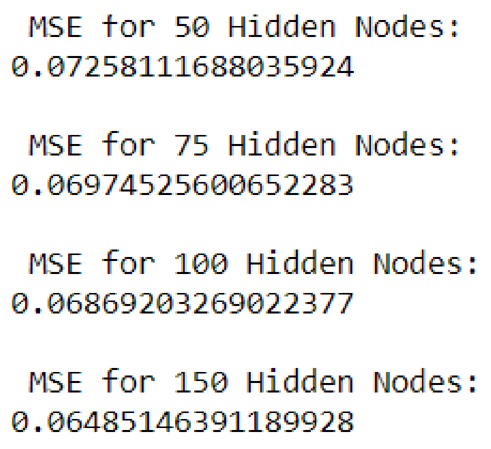


Figure 4 Training and Test Error for each iteration in different number of hidden nodes in 10 Iterations

Next, we can then plot the error for each iteration shown in Figure 3. What we can observe when looking on the graphs is that more number of hidden nodes will result in steeper error difference in each epoch. In addition, we can see the final RMSE (shown in Figure 4.) from each hidden node and we can conclude that more hidden nodes represent lower error value.



```
 MSE for 50 Hidden Nodes:
0.07258111688035924

 MSE for 75 Hidden Nodes:
0.06974525600652283

 MSE for 100 Hidden Nodes:
0.06869203269022377

 MSE for 150 Hidden Nodes:
0.06485146391189928
```

Figure 3 Mean Square error for different number of hidden nodes after 10 iterations

Next, we plot each digit from the test set of MNIST and compare it with the reconstructed images from our network. With using 10 iterations, the results from this segment are shown in Figure 5 below. We can see that more number of hidden nodes will result in a better reconstruction of the images from MNIST. In addition, the network can also detect when the image given to them are not perfect. For example, look at "8" in Figure 5. We can see that "8" is not complete but the network can detect and say that it is 8 which means that our algorithm have the capability to reconstruct a set that also have some noise.

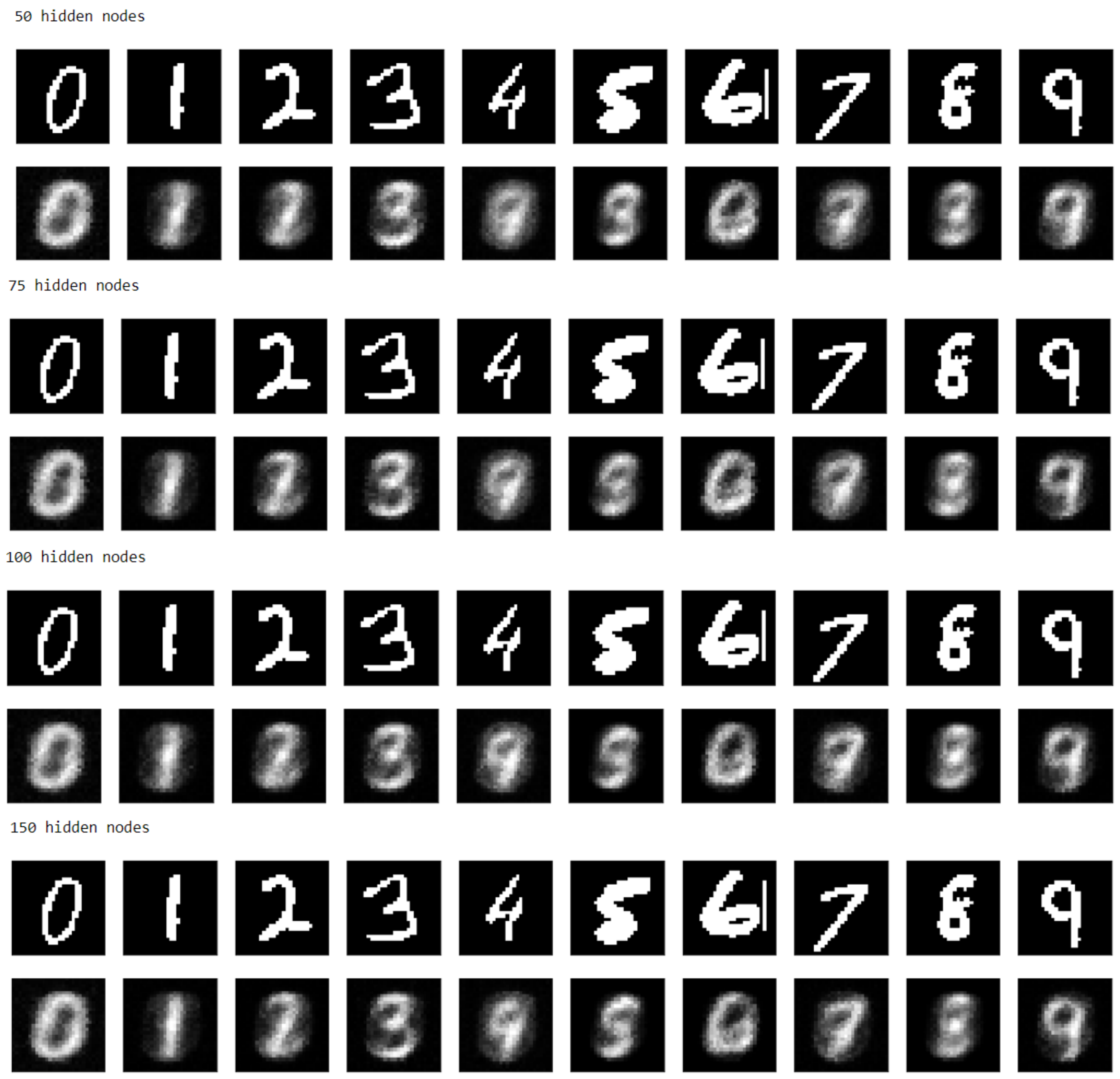
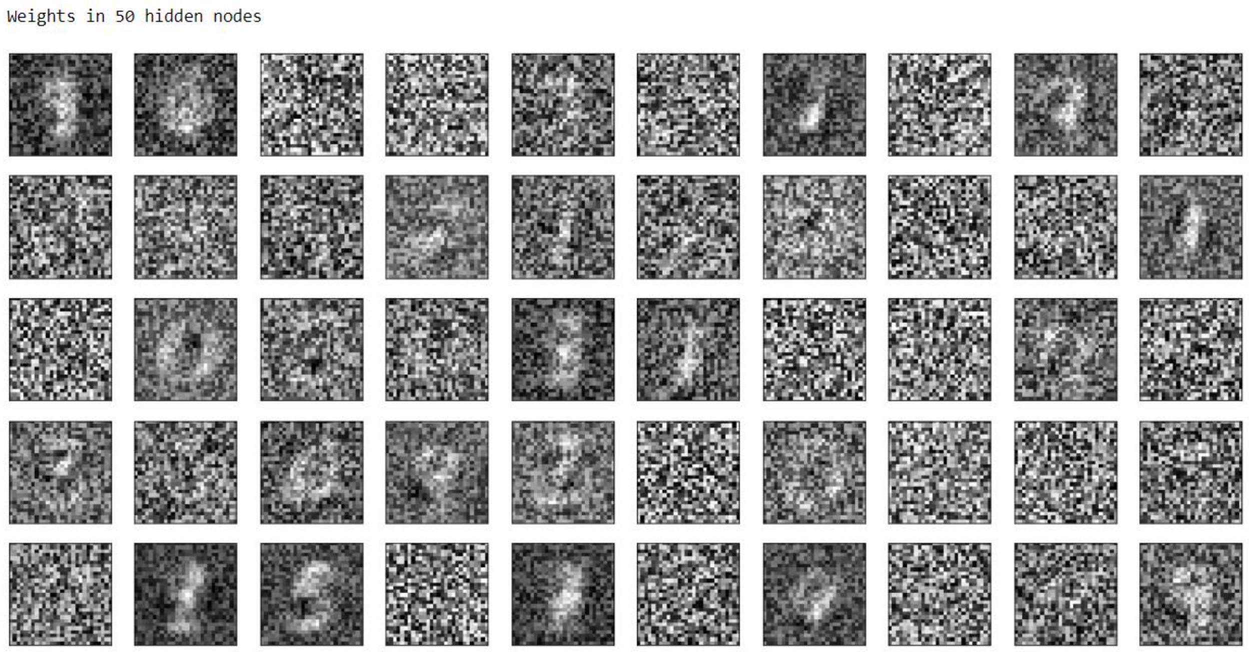


Figure 5 Comparing Test and reconstructed images from our network. All of the test for this part was using 10 iterations in each different number of hidden nodes.

Lastly, we plotted the weights which are formed after we have trained the network. Number of hidden nodes represents the number of weights present in the network. The weights from the network are shown below:



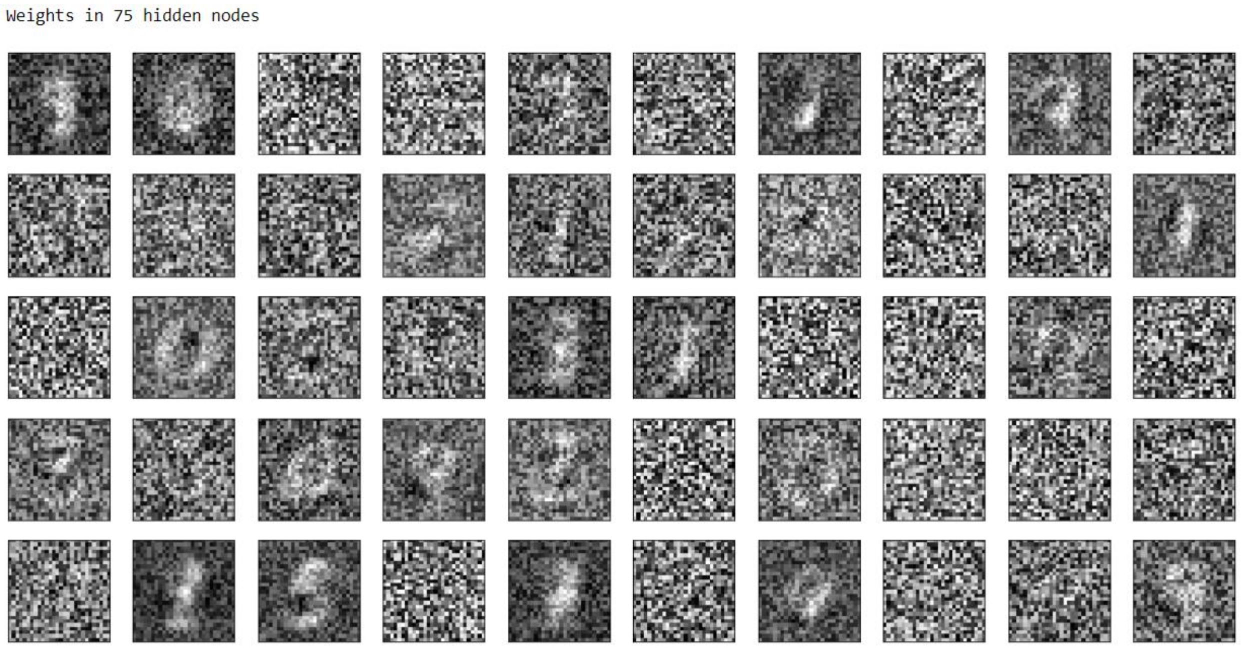


Figure 6 Weights for 50 and 75 hidden nodes in 10 iterations

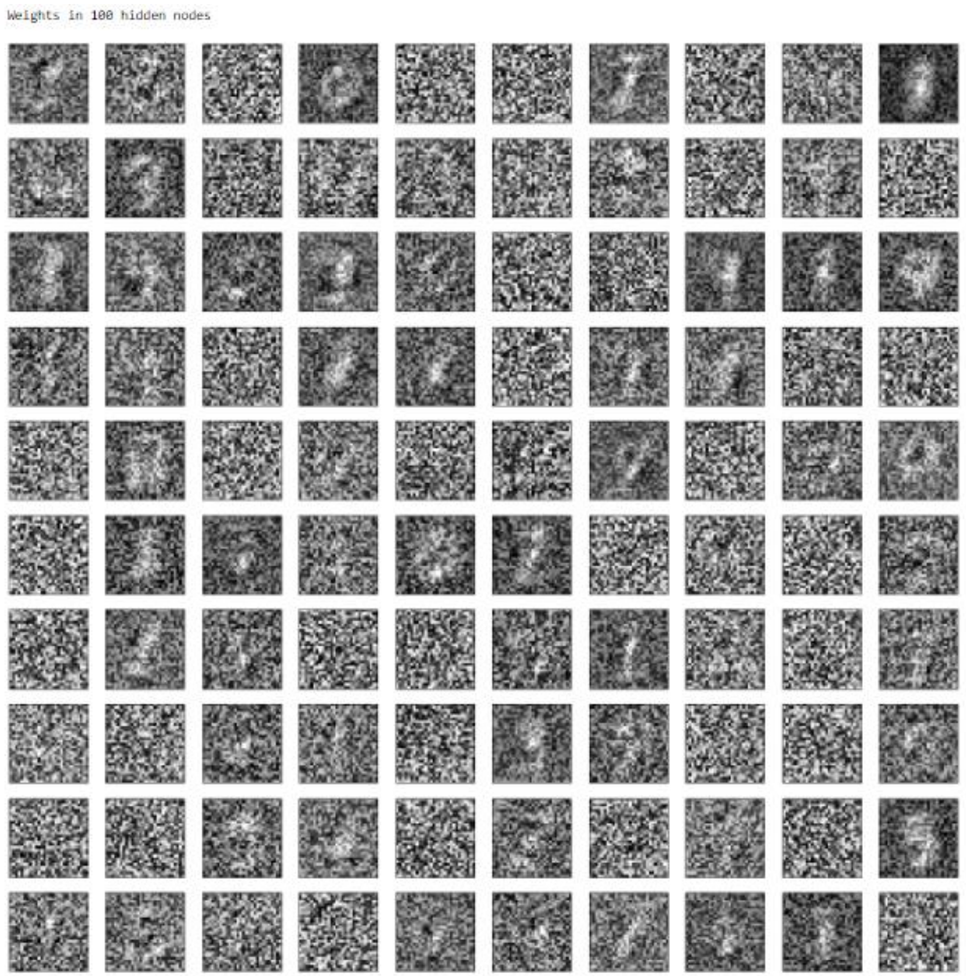
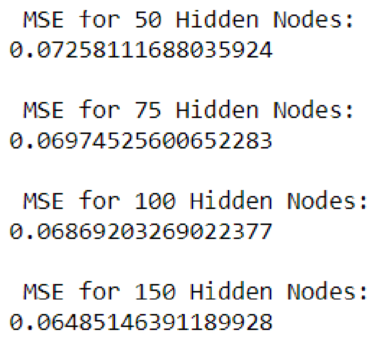
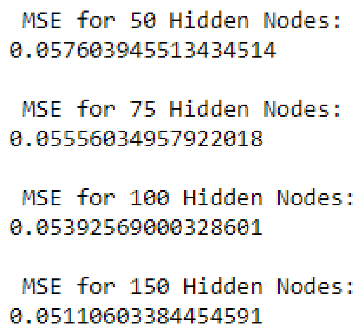


Figure 8 Weights for 100 Hidden nodes in 10 iteration



```
 MSE for 50 Hidden Nodes:
0.07258111688035924

 MSE for 75 Hidden Nodes:
0.06974525600652283

 MSE for 100 Hidden Nodes:
0.06869203269022377

 MSE for 150 Hidden Nodes:
0.06485146391189928
```



```
 MSE for 50 Hidden Nodes:
0.057603945513434514

 MSE for 75 Hidden Nodes:
0.05556034957922018

 MSE for 100 Hidden Nodes:
0.05392569000328601

 MSE for 150 Hidden Nodes:
0.05110603384454591
```

Figure 7 (left) is the MSE for 10 iterations and (right) is MSE for 20 iterations

When we are using 20 iterations to train the network, the end mean square error are less than compared to 10 iterations. The comparison can be seen in Figure 8.

Different from when we observe for 10 iterations, the error for more number of nodes relatively have similar steep in the graph (observe Figure 9 next page). However, the minimum number of MSE are obtained by using more number of nodes.

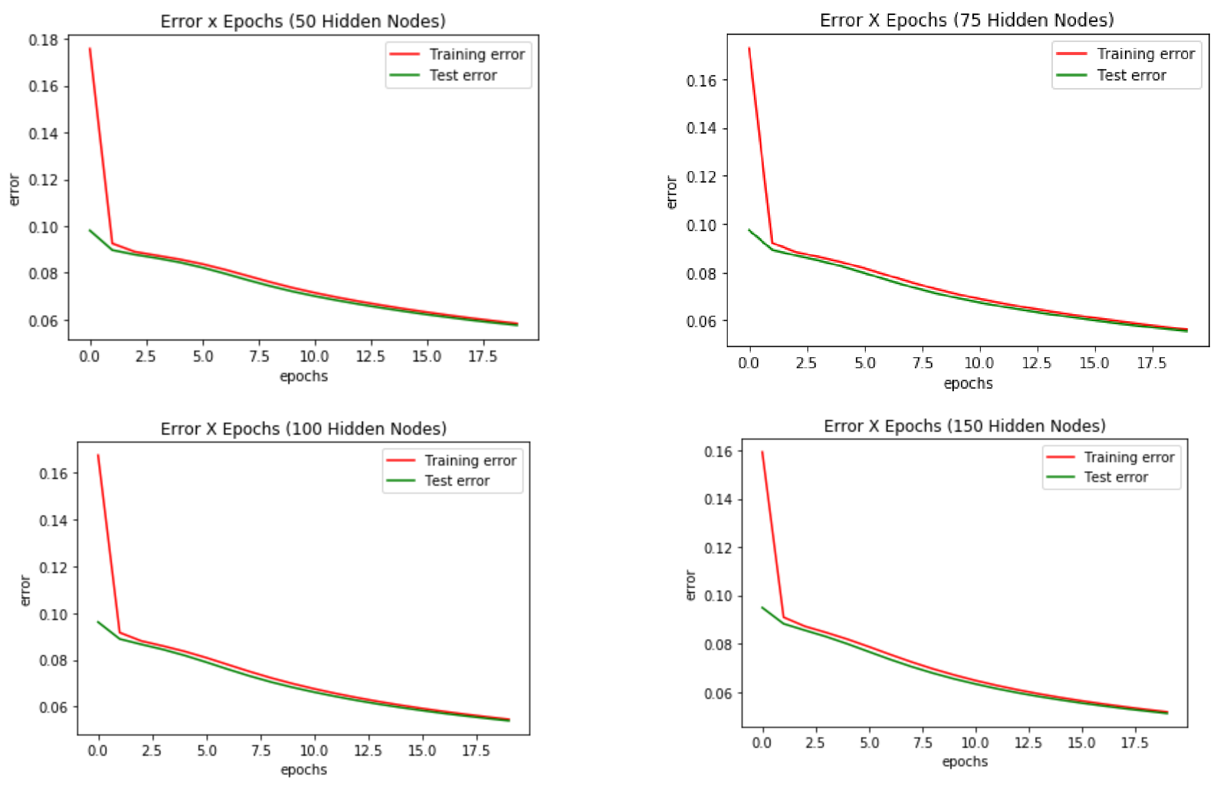


Figure 10 Comparison of Error in each epoch for 20 iterations. Each graph shows different number of hidden nodes

Next, we then plot the digit set and compare it with reconstructed set. When we compare the reconstructed image for 150 hidden nodes in 10 and 20 iterations, we conclude that 20 iterations make a clearer image. As before, more number of hidden nodes will result in a clear output.

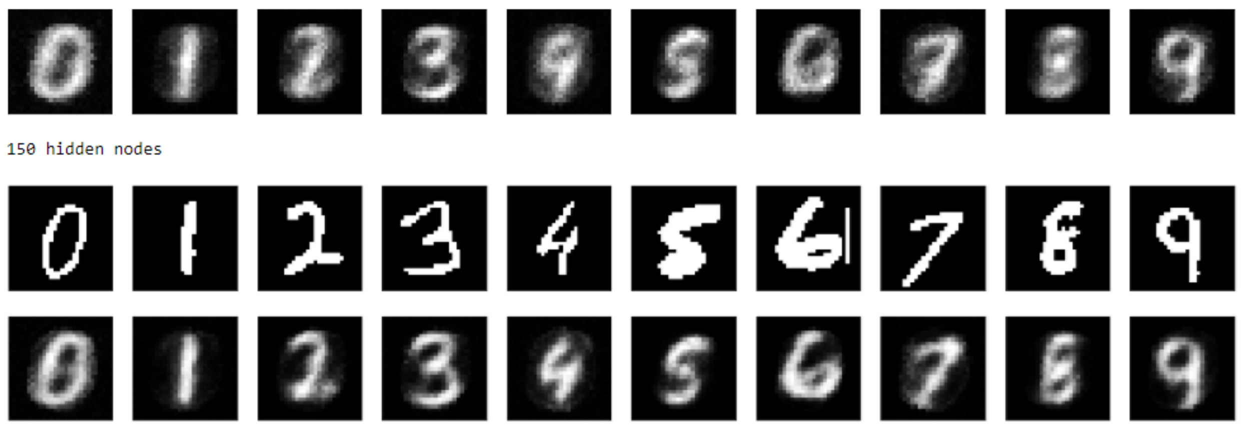


Figure 9 First row represents the reconstructed image for 150 hidden nodes in 10 iterations. Further below are the reconstructed image for 20 iterations

For the rest of the comparison between digit test set and the reconstructed set can be seen in Figure 11. Same as previous, more number of hidden nodes will result in better quality of reconstruction.

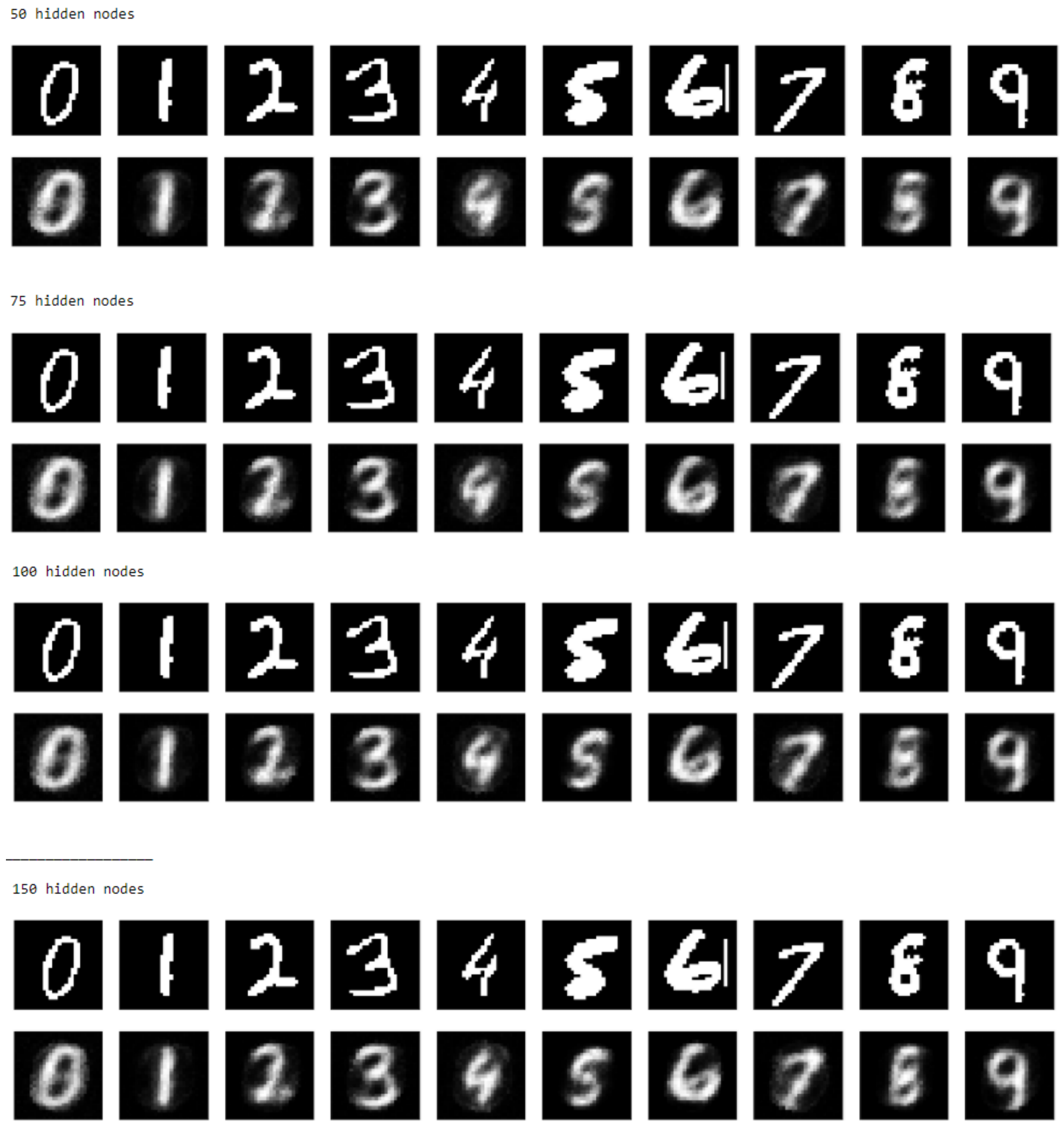
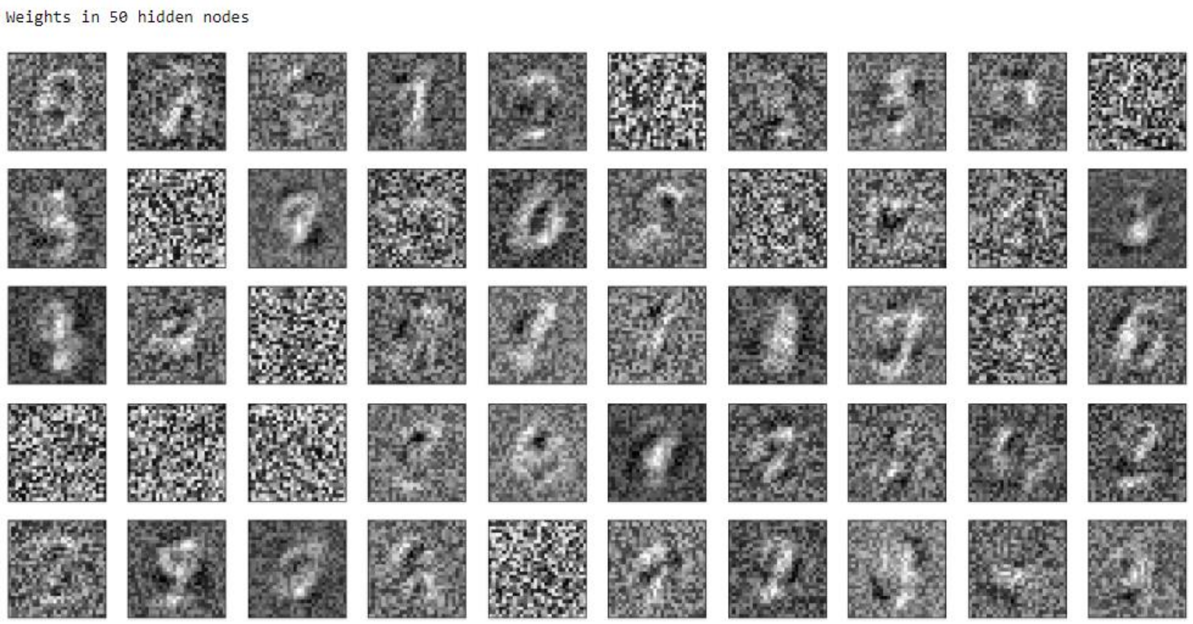


Figure 11 With 20 iterations on different number of hidden nodes

In the last section, we then reconstruct the Weights for each number of hidden nodes shown in the next pages. What we observe is that with the addition of iterations, the weights become more "filled" and there are less noise in the weights if we compare it with 10 iterations.



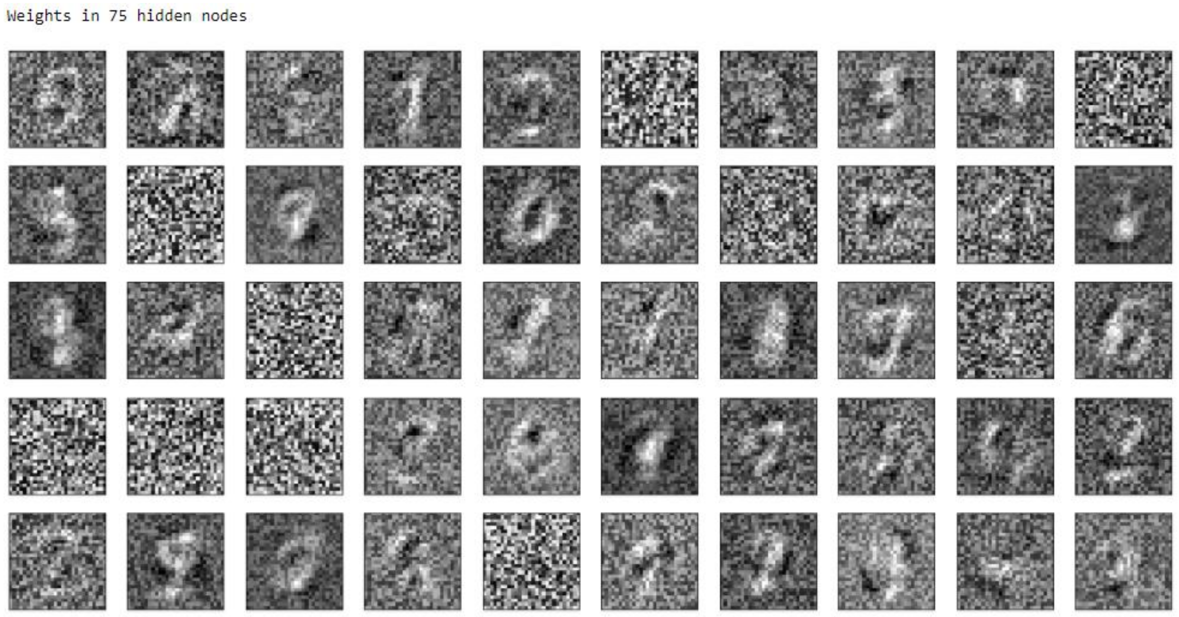


Figure 12 Weights for 50 and 75 hidden nodes in 20 iterations

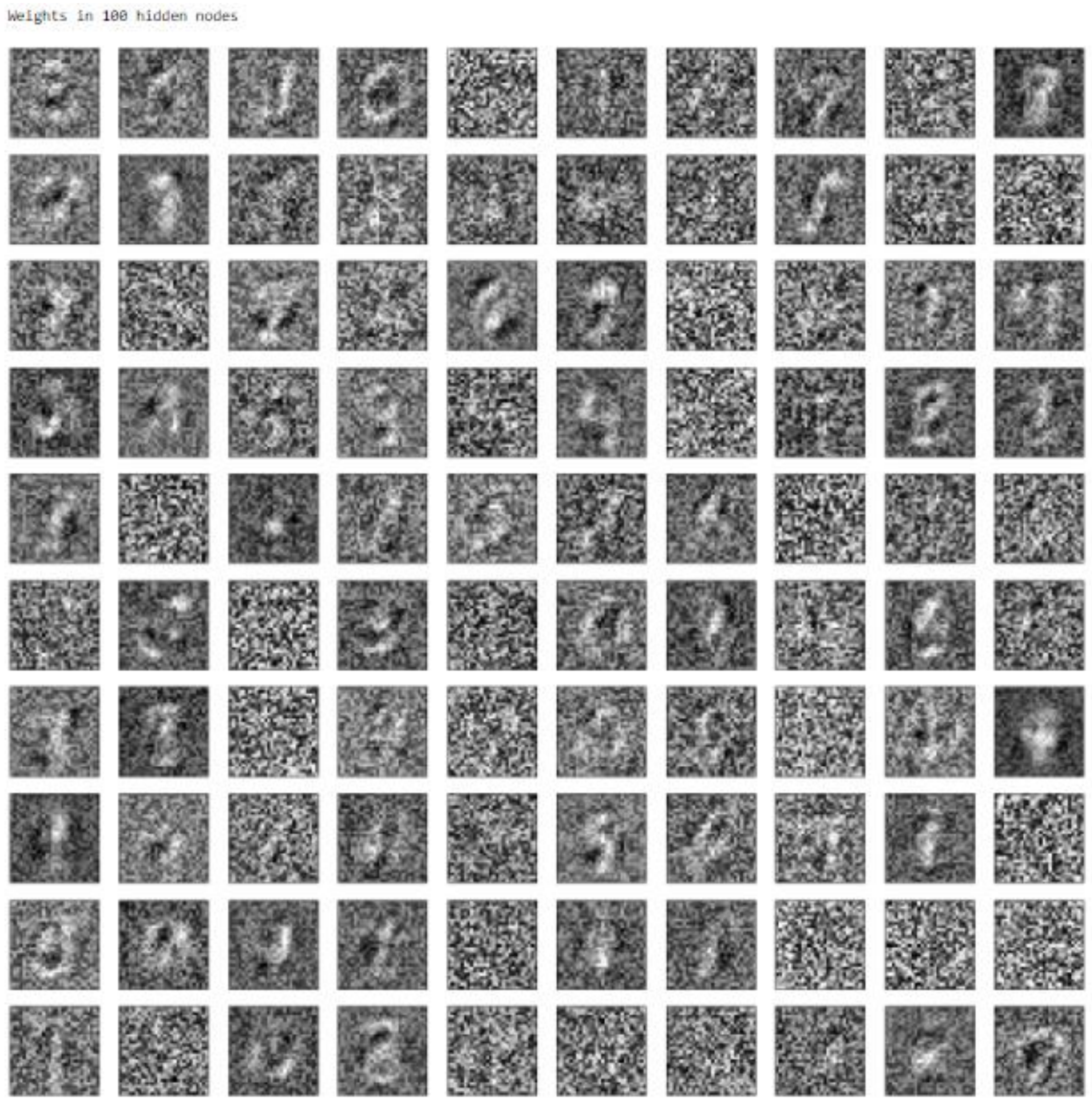


Figure 13 Weights for 100 hidden nodes in 20 iterations

4.1.2 Restricted Boltzmann Machine (RBM)

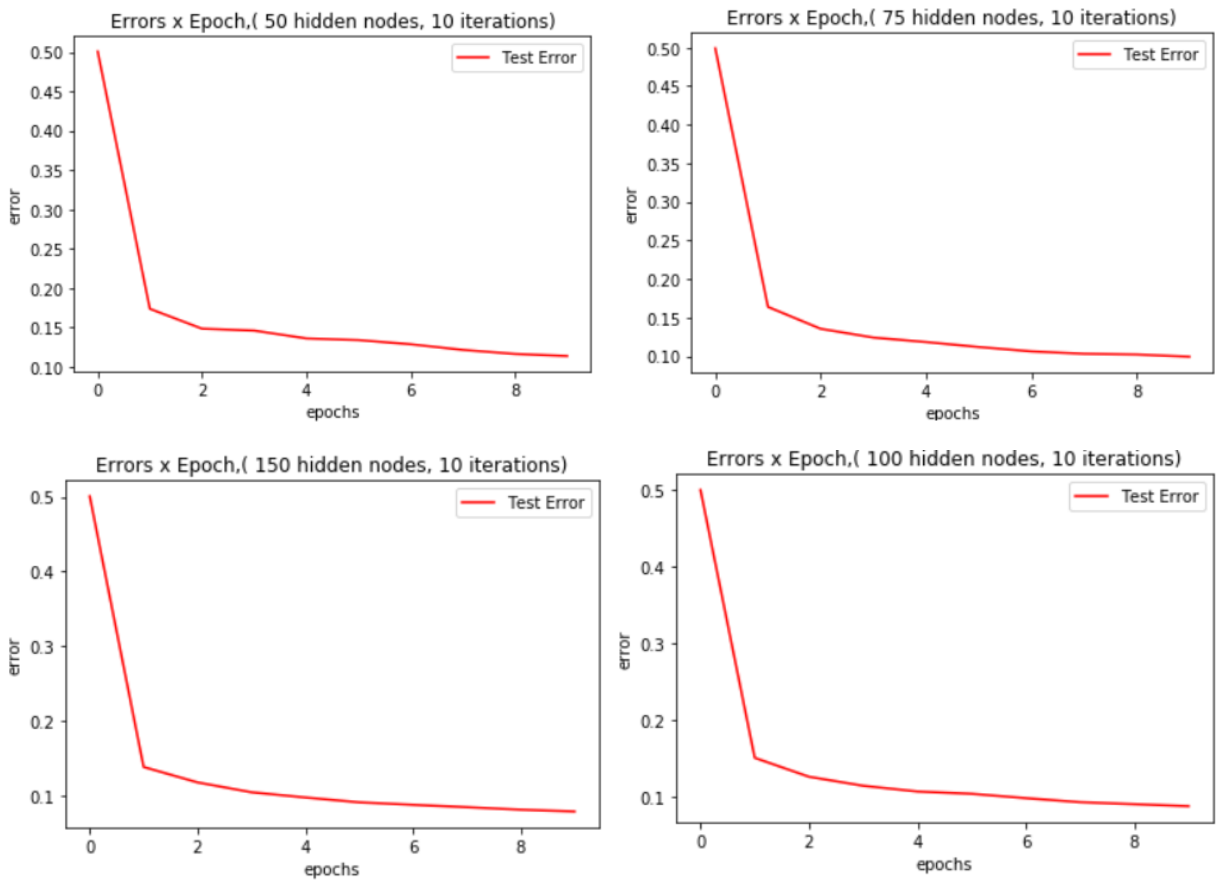
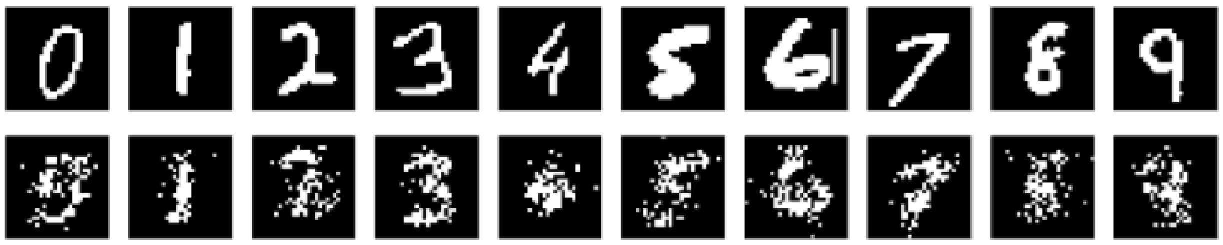


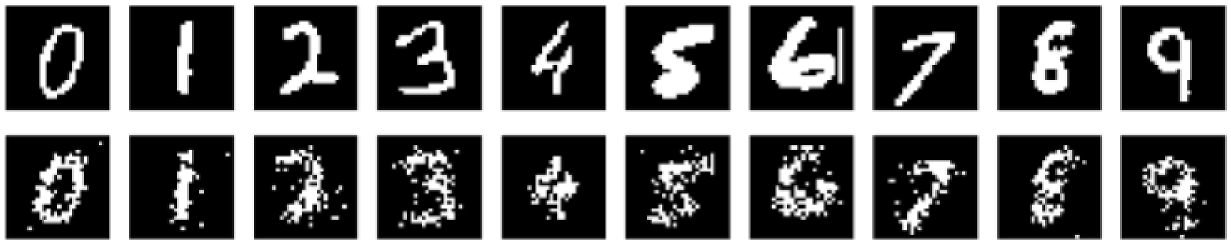
Figure 14 Error vs epoch for 50, 75, 100 and 150 hidden nodes in 10 iterations using RBM algorithm

For RBM, we also used Scikit Learn from python. However, in this task, we have difficulties to present the error graph per epoch for the training error. Hence, we tried solving it by presenting the mean square error for each iteration which then means a more heavier computation. This is due that the iteration are repeated to represent the mean square error. Another difficulty is when we tried to represent a cross-validation to see the error for the test and training set compared to the reconstructed image. After several times trying to read the literatures and tried, it took a long computational speed hence we fail to try to present the graph.

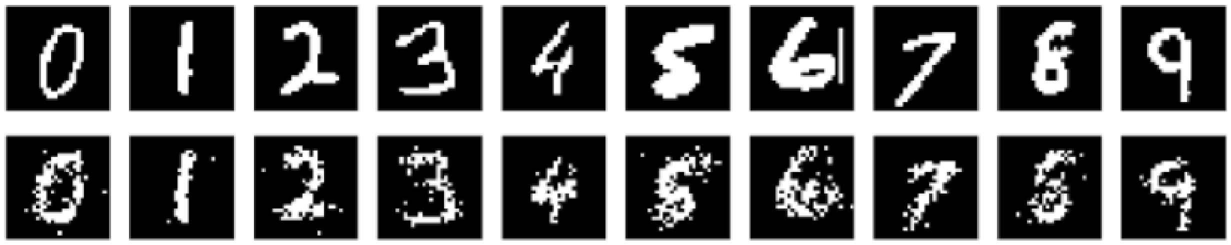
So, for figure 14, we present the graph of errors for each epoch for 10 iterations in different number of hidden nodes of the network. What we can see from the graph is that for 150 hidden nodes, we have less value of error and the error tends to fluctuate for 50 hidden nodes.



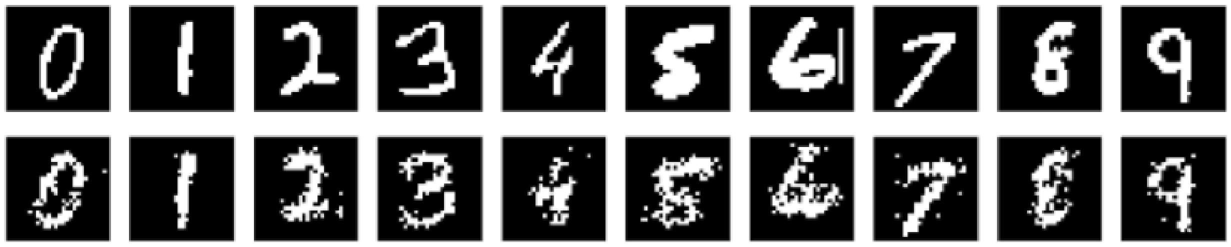
```
MSE for 50 hidden nodes, 10 iterations = 0.11347576530612245
```



```
MSE for 75 hidden nodes, 10 iterations = 0.09916836734693878
```



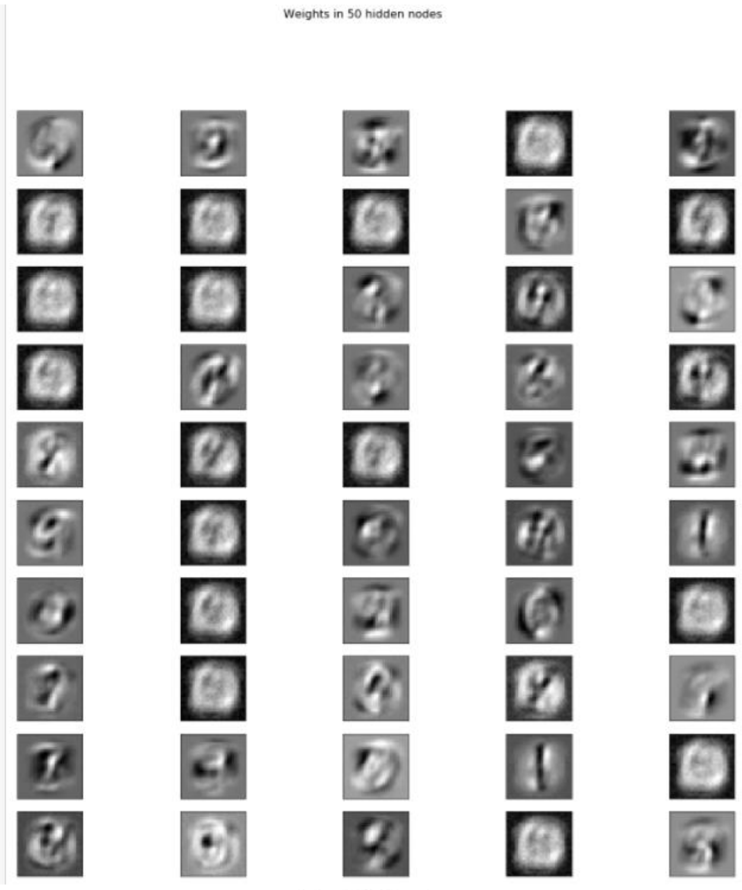
```
MSE for 100 hidden nodes, 10 iterations = 0.08801147959183672
```



```
MSE for 150 hidden nodes, 10 iterations = 0.07893941326530611
```

Figure 15 Comparing the test and reconstructed images for 50, 75, 100 and 150 hidden nodes respectively from top to bottom

Next, we wanted to show the reconstruction made by using 10 iterations for each hidden node in Figure 15. Compared to RBM, these images are more clear and solid. However, the best reconstructed images are obtained when we have more number of layers.



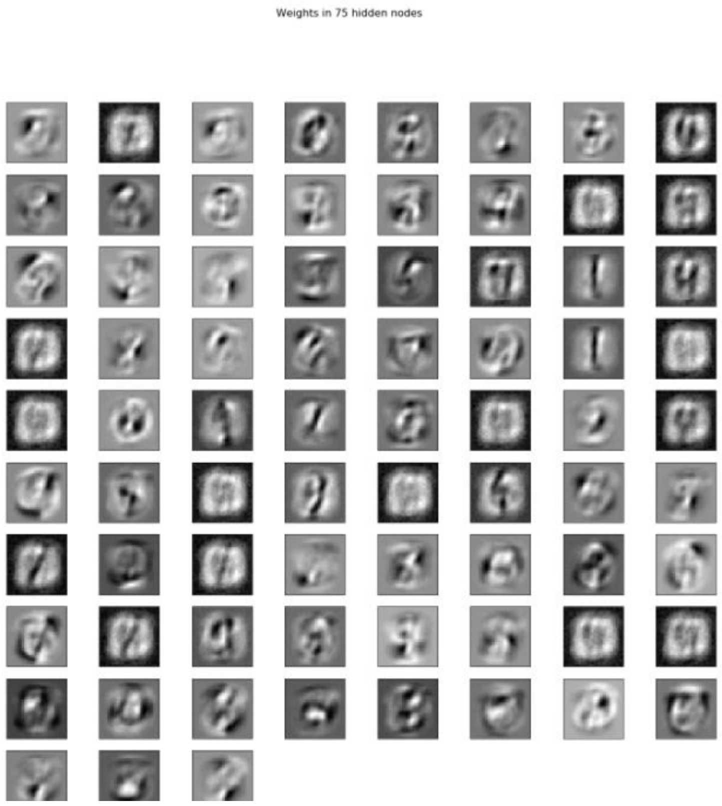


Figure 16 Weights for 50 and 75 hidden nodes in 10 iterations (RBM)

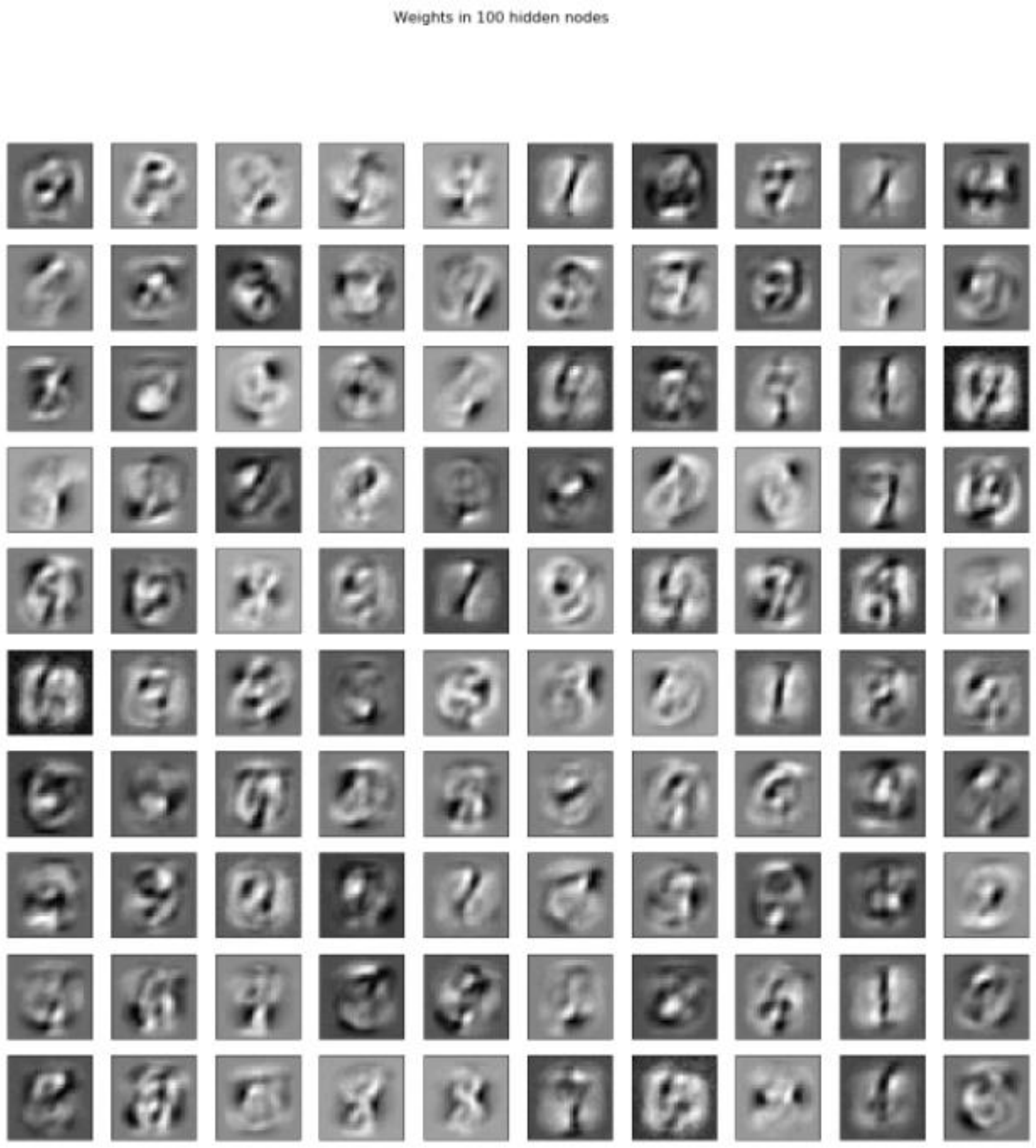


Figure 17 Weights for 100 hidden nodes in 10 iterations (RBM)

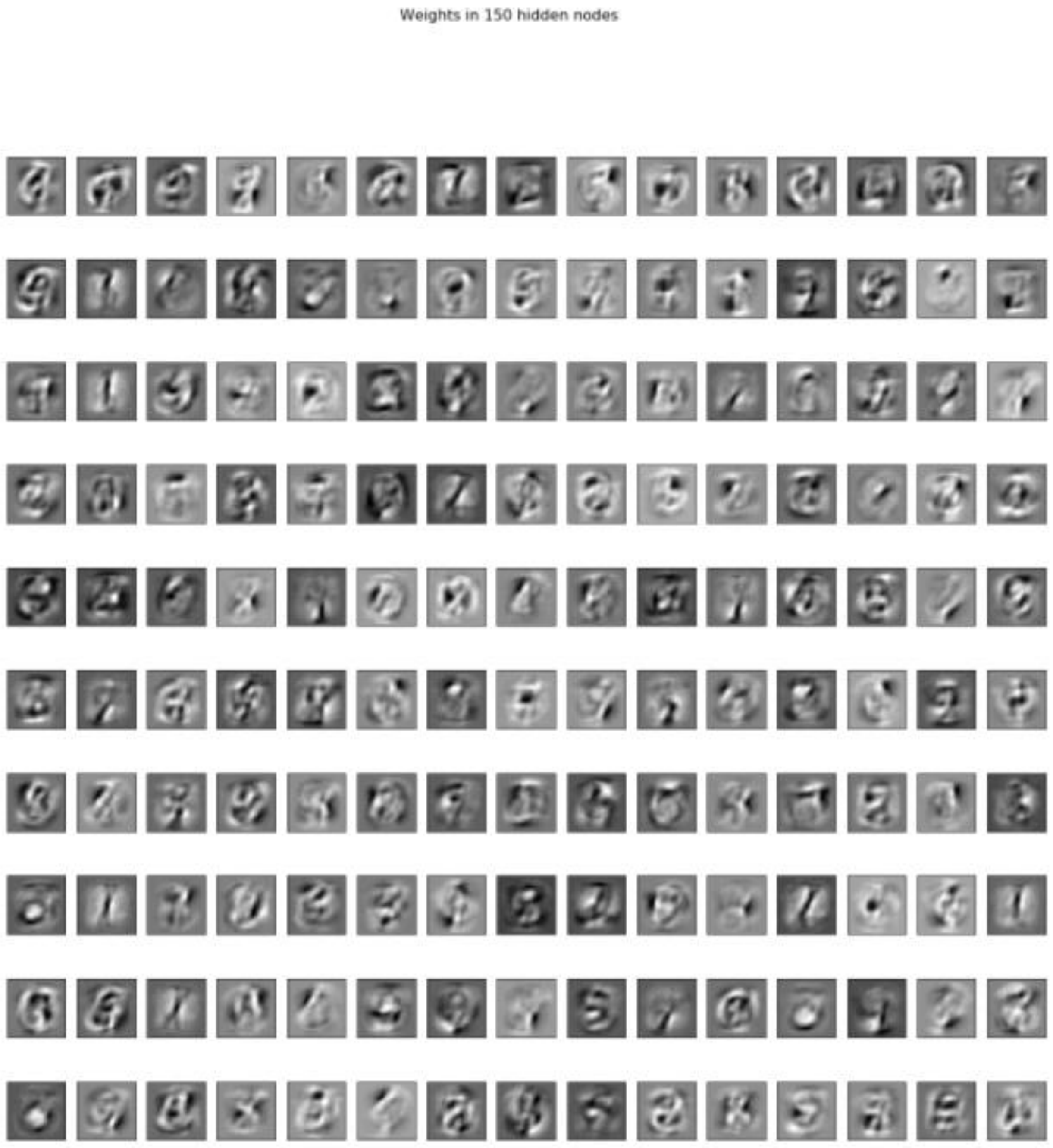


Figure 18 Weights for 150 hidden nodes in 10 iterations (RBM)

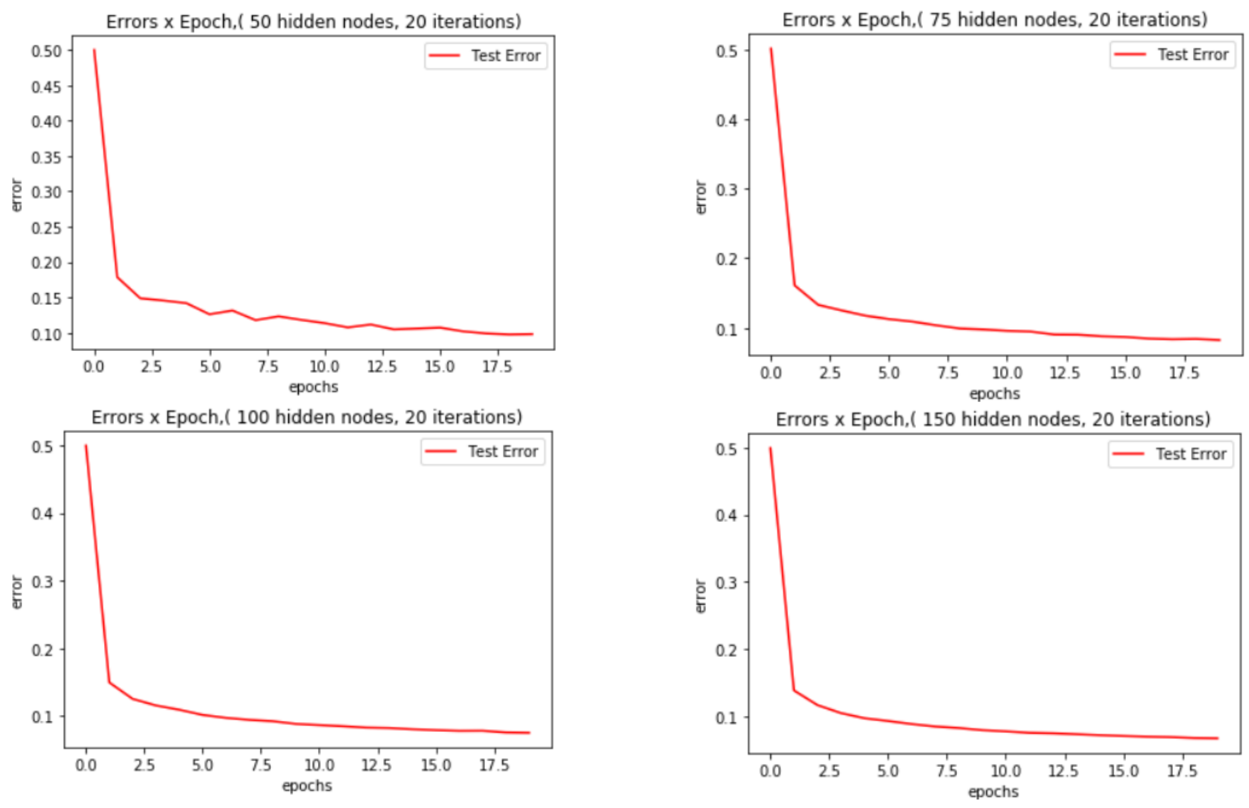
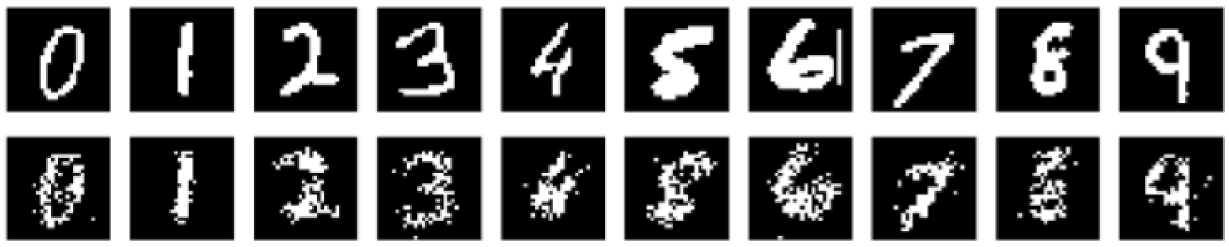


Figure 19 Graph for Error for each iteration in 20 iterations

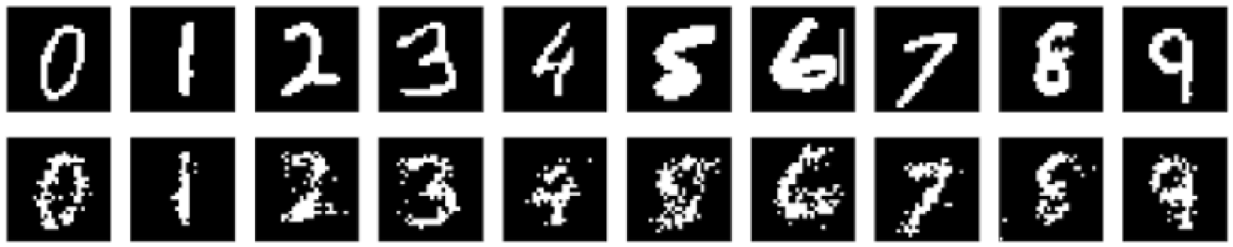
Same with 10 iterations, we observe that there are some fluctuations when we only have 50 hidden nodes. The fluctuations are also present in 75 hidden nodes but has less amplitude. From the table below, we can also see the differences which we have for 10 and 20 iterations. It is clear that more number of iteration means that we have less number of error. The comparison is shown in Table 1.

Table 1 Comparison of MSE in 10 and 20 iterations (RBM)

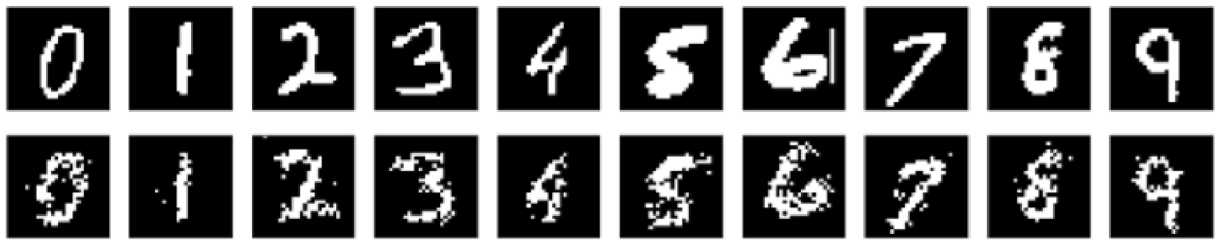
No. of Hidden Nodes	MSE for 10 Iteration	MSE for 20 Iteration
50	0.11347576530612245	0.09798979591836736
75	0.09916836734693878	0.08262563775510204
100	0.08801147959183672	0.07512946428571428
150	0.07893941326530611	0.06702742346938775



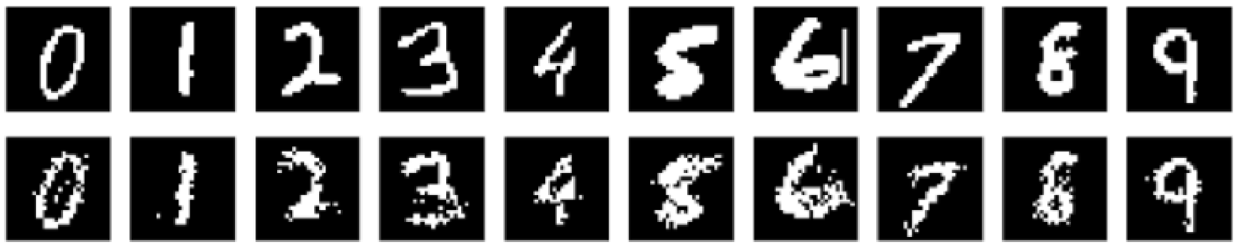
```
MSE for 50 hidden nodes, 20 iterations = 0.09798979591836736
```



```
MSE for 75 hidden nodes, 20 iterations = 0.08262563775510204
```



```
MSE for 100 hidden nodes, 20 iterations = 0.07512946428571428
```



```
MSE for 150 hidden nodes, 20 iterations = 0.06702742346938775
```

Figure 20 Comparing the test and reconstructed images for 50, 75, 100 and 150 hidden nodes respectively from top to bottom (20 iterations)

Figure 20 shows the comparison between test and reconstructed images. With 20 iterations, we have much clear images compared to 10 iterations. The comparison between using 10 and 20 iterations can be seen in Figure 21. Most striking differences can be seen for the reconstructed “3”, “6” and “8”. In our opinion, 20 iterations make the reconstruction better. Figure 21 is shown on the next page.

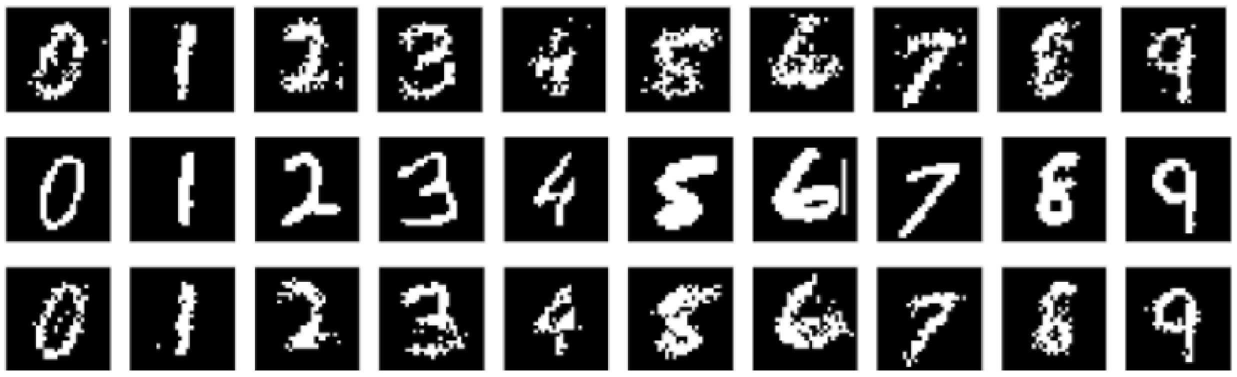


Figure 21 First row represents the reconstructed image for 150 hidden nodes in 10 iterations. Further below are the reconstructed image for 20 iterations

After this, we plotted the Weights for all the hidden nodes. Like those in RBF, with more number of iterations, the weights seemed filled and have less noise compared to the 10 iterations. The weights can be seen in figures 22 – 25.

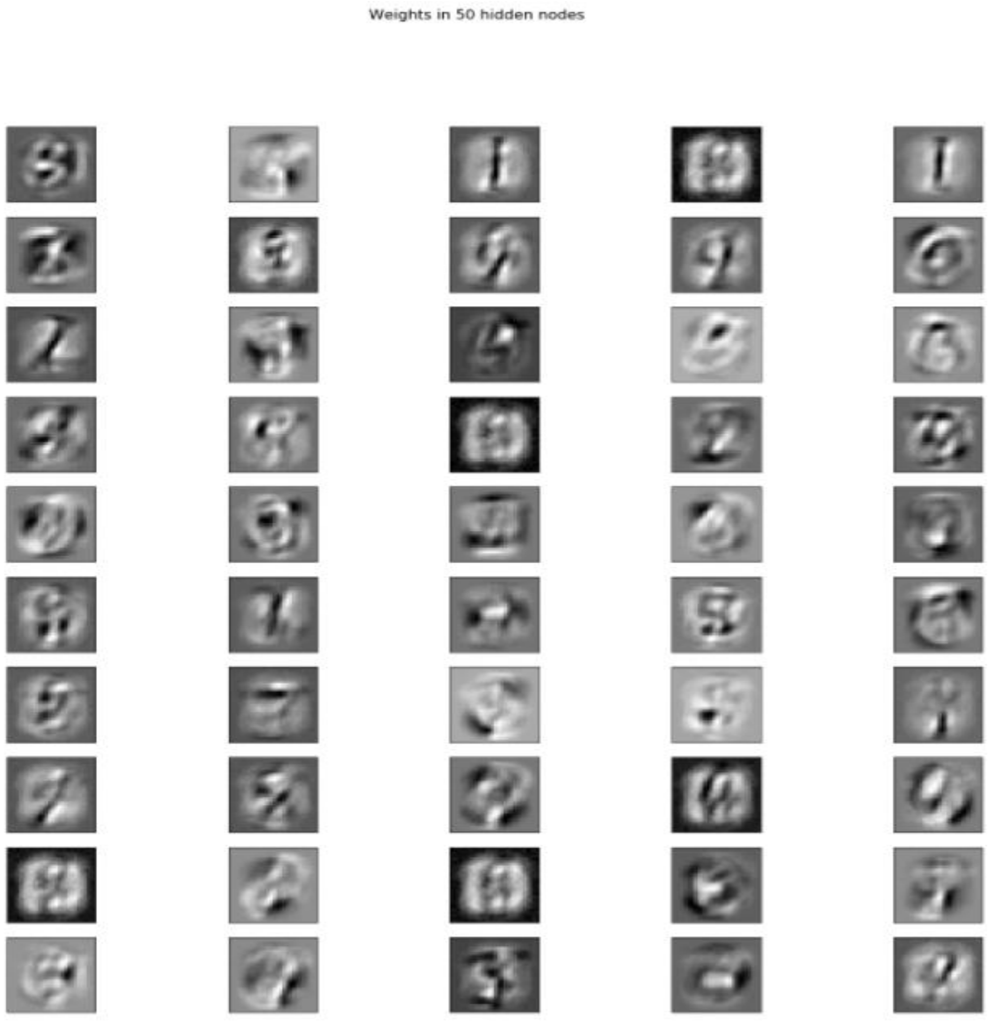


Figure 22 Weights for 50 hidden nodes 20 iterations

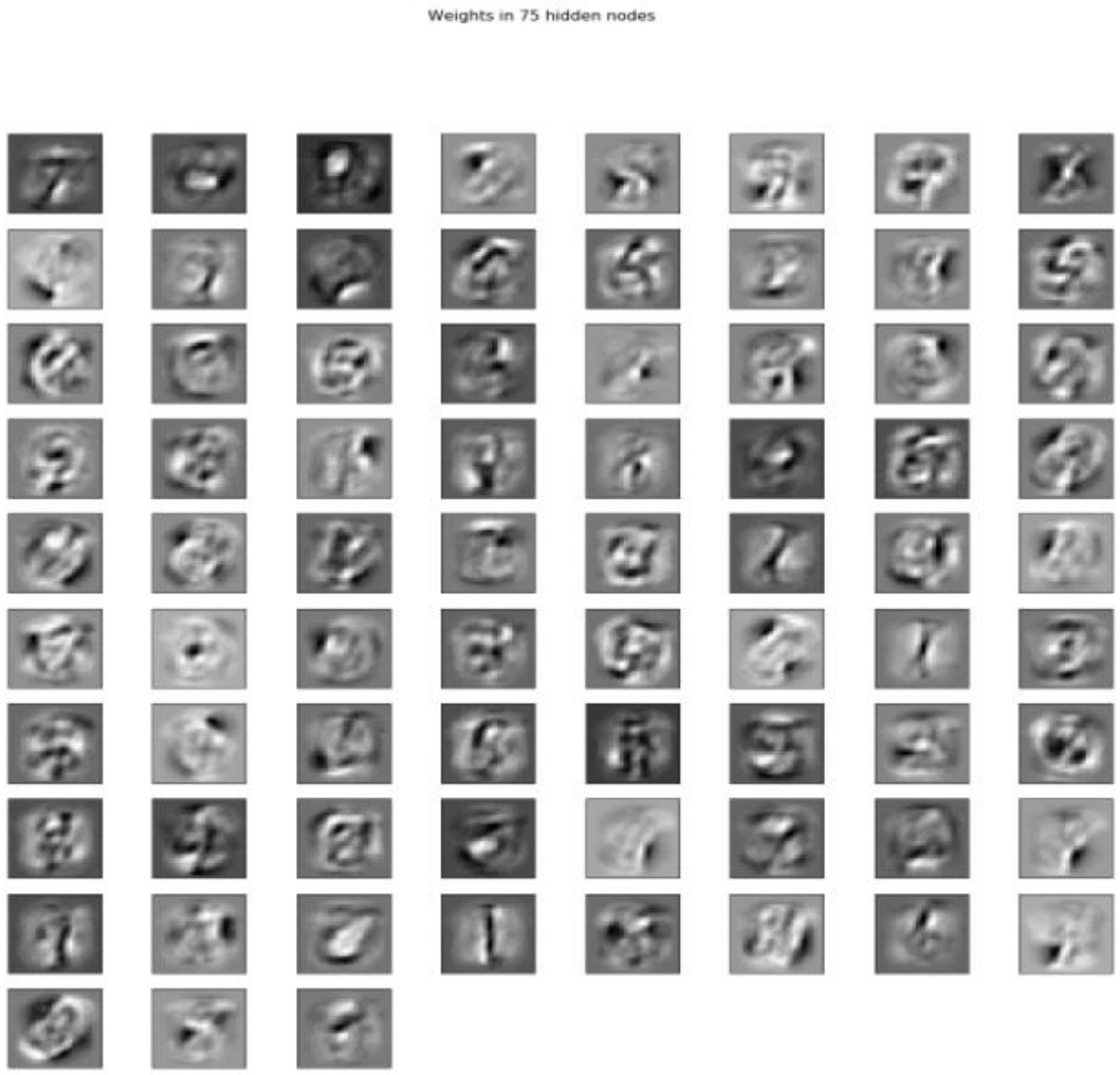


Figure 23 Weights for 75 hidden nodes 20 iterations

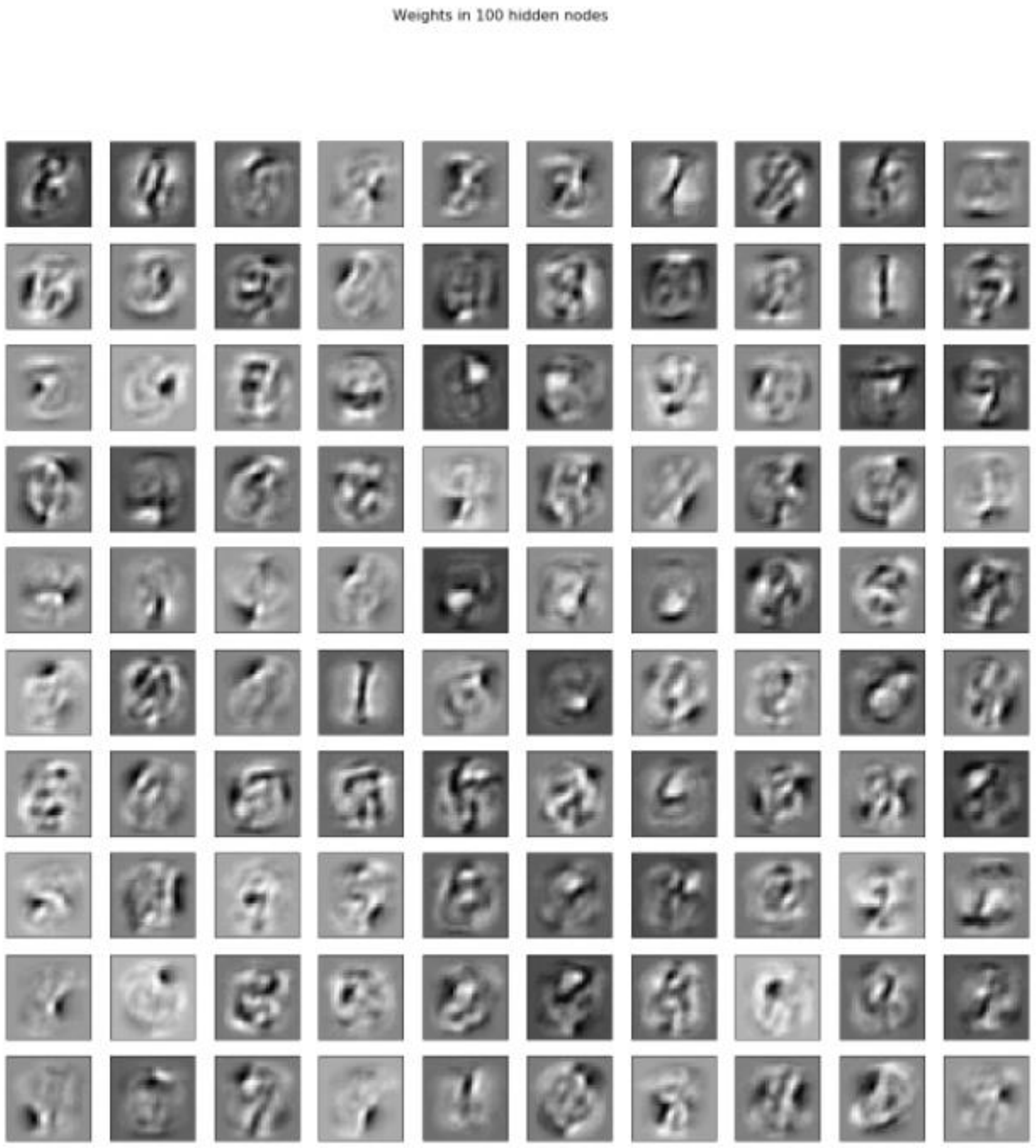


Figure 24 Weights for 100 hidden nodes 20 iterations

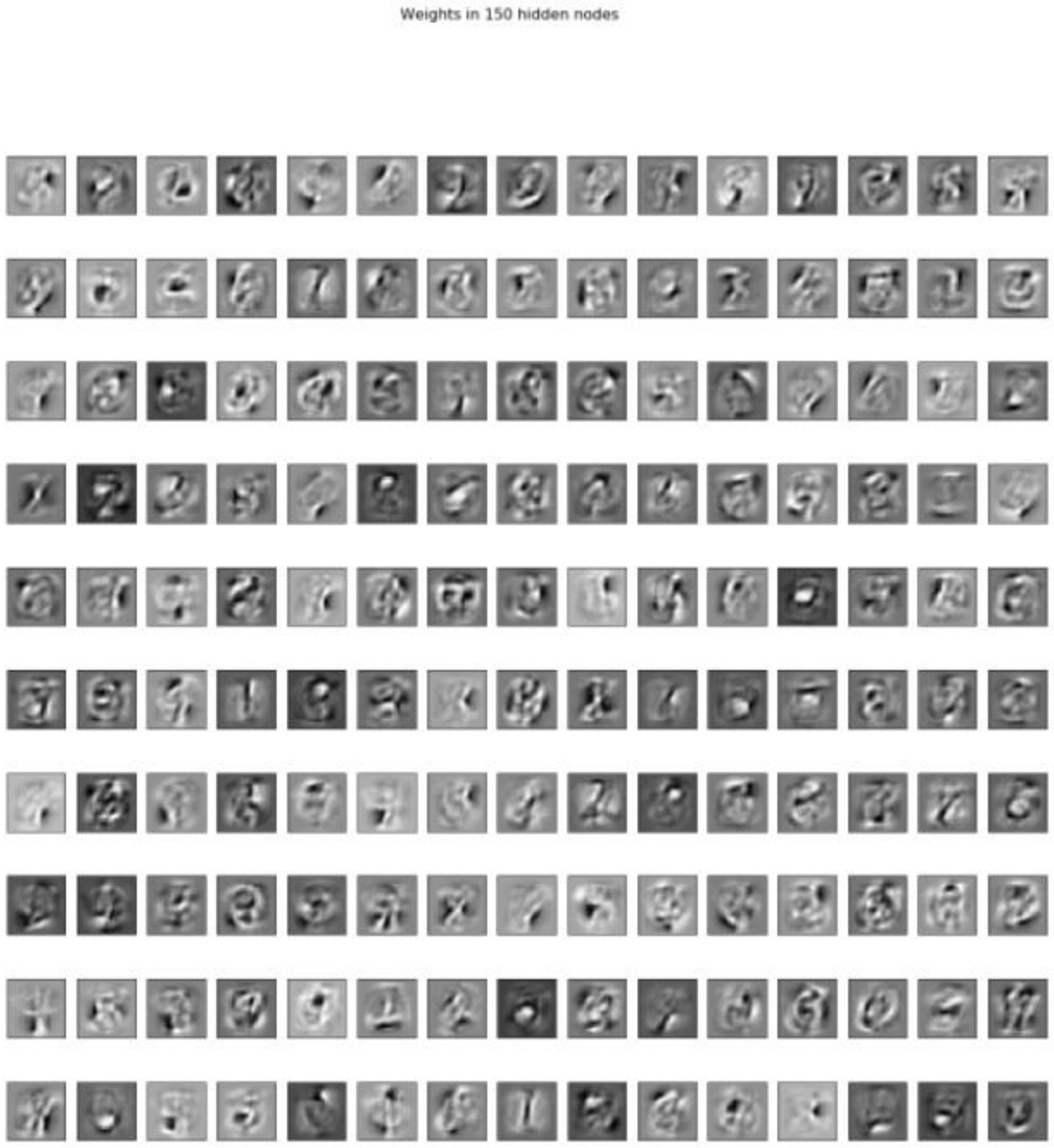


Figure 25 Weights for 150 hidden nodes 20 iterations

Lastly, to make a comparison between RBM and Autoencoder, we made a table between these two for each number of iterations and number of nodes:

Table 2 Comparison of MSE in different algorithm, iteration and number of hidden nodes

No. of Hidden Nodes	Autoencoder		RBM	
	10 Iterations	20 Iterations	10 Iterations	20 Iterations
50	0.072581116880	0.057603945513	0.113475765306	0.097989795918
75	0.069745256007	0.055560349579	0.099168367347	0.082625637755
100	0.0686920326902	0.053925690003	0.088011479592	0.075129464286
150	0.064851463912	0.051106033845	0.078939413265	0.067027423469

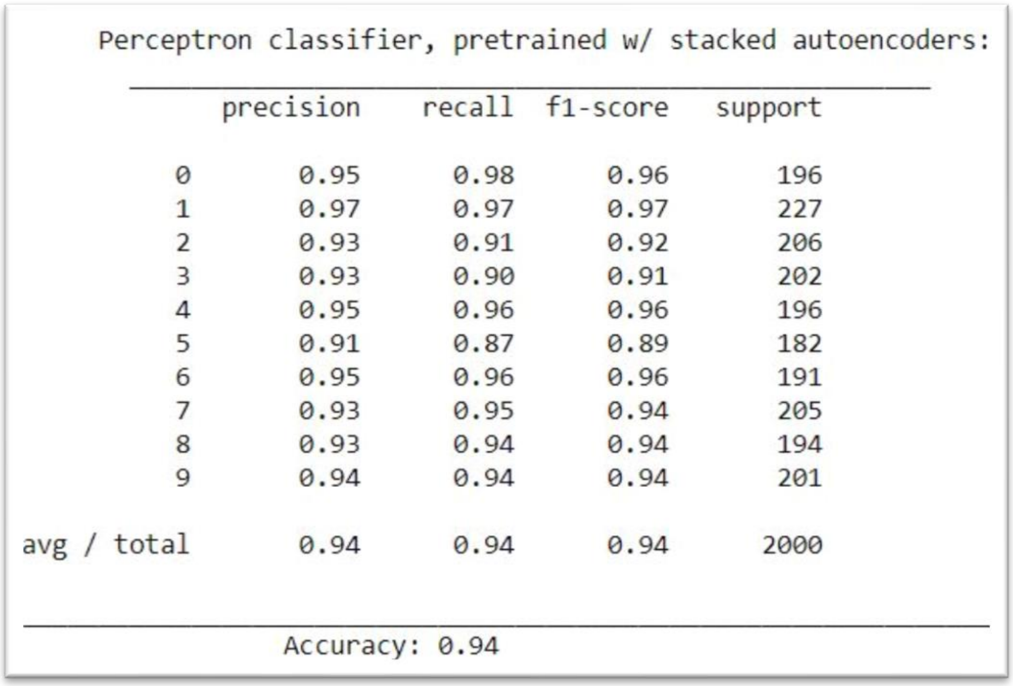
The least number of MSE that we obtained was from Autoencoder with 20 iterations and 150 hidden nodes.

4.2 **DBN and stacked autoencoders for MNIST digit classifications**

In this part of the lab, we use two different deeper architecture for MNIST digit classification : *stacked autoencoders* and *stacked RBMs (DBN)* -based digit classifier. These architectures are pre-trained in a greedy layer-wise manner and containing a classification layer trained in supervised mode. We will compare any difference in classifying performance in using 1, 2, and 3 hidden layers. Classifying performance will utilize classification report (using `sklearn.metrics.classification_report` attribute) and accuracy score (using `sklearn.metrics.accuracy_score` attribute). Both architectures are using a same set of configurations as follows:

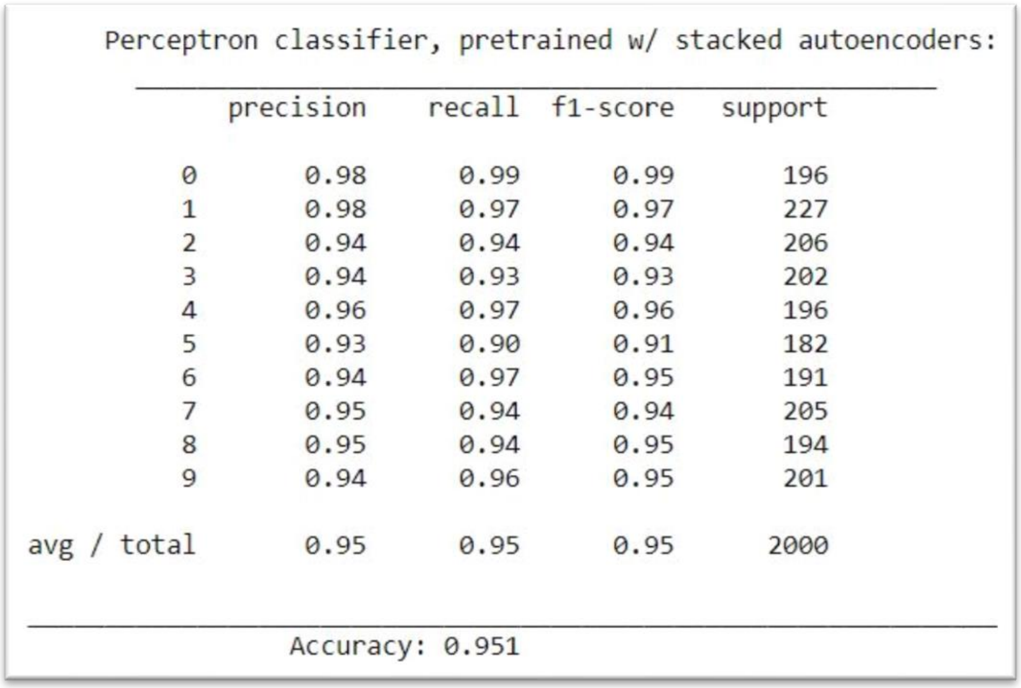
- Learning rate : 0.01
- Hidden nodes : 150 (first hidden layer), 100 (second hidden layer) , 50 (third hidden layer)
- Epoch : 50
- Activation functions : Rectified Linear Unit (relu)
- Batch size : 64

4.2.1 Digit Classification with Stacked Autoencoders



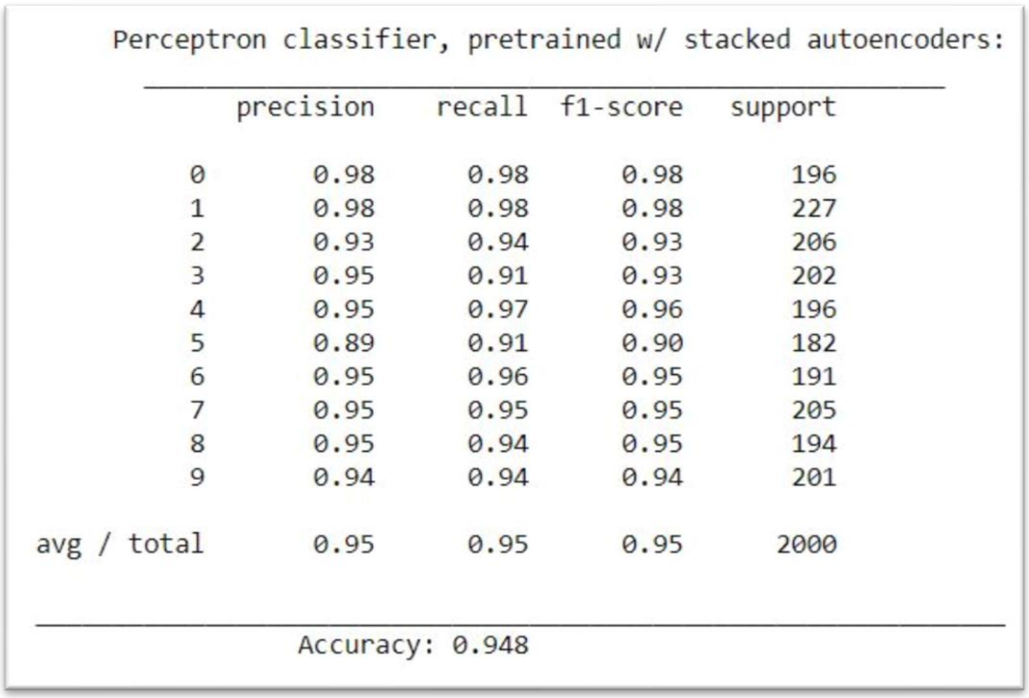
```
     Perceptron classifier, pretrained w/ stacked autoencoders:
       ______________________________________________________
             precision    recall  f1-score   support

          0       0.95      0.98      0.96       196
          1       0.97      0.97      0.97       227
          2       0.93      0.91      0.92       206
          3       0.93      0.90      0.91       202
          4       0.95      0.96      0.96       196
          5       0.91      0.87      0.89       182
          6       0.95      0.96      0.96       191
          7       0.93      0.95      0.94       205
          8       0.93      0.94      0.94       194
          9       0.94      0.94      0.94       201

avg / total       0.94      0.94      0.94      2000


_________________________________________________________________
                Accuracy: 0.94
```

Figure 26.1 Classification reports using stacked autoencoders with 1 hidden layers



```
     Perceptron classifier, pretrained w/ stacked autoencoders:
       ______________________________________________________
             precision    recall  f1-score   support

          0       0.98      0.99      0.99       196
          1       0.98      0.97      0.97       227
          2       0.94      0.94      0.94       206
          3       0.94      0.93      0.93       202
          4       0.96      0.97      0.96       196
          5       0.93      0.90      0.91       182
          6       0.94      0.97      0.95       191
          7       0.95      0.94      0.94       205
          8       0.95      0.94      0.95       194
          9       0.94      0.96      0.95       201

avg / total       0.95      0.95      0.95      2000


_________________________________________________________________
                Accuracy: 0.951
```

Figure 26.2 Classification reports using stacked autoencoders with 2 hidden layers



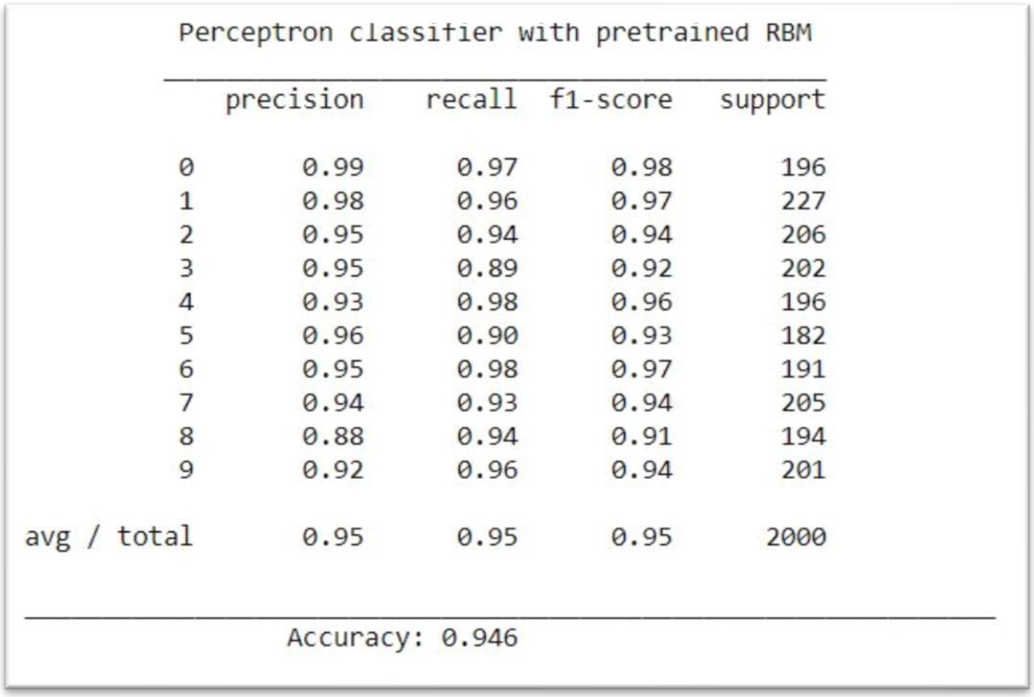
```
        Perceptron classifier, pretrained w/ stacked autoencoders:
          ____________________________________________________
                    precision    recall  f1-score   support

              0       0.98      0.98      0.98       196
              1       0.98      0.98      0.98       227
              2       0.93      0.94      0.93       206
              3       0.95      0.91      0.93       202
              4       0.95      0.97      0.96       196
              5       0.89      0.91      0.90       182
              6       0.95      0.96      0.95       191
              7       0.95      0.95      0.95       205
              8       0.95      0.94      0.95       194
              9       0.94      0.94      0.94       201

    avg / total       0.95      0.95      0.95      2000


    __________________________________________________________
                     Accuracy: 0.948
```

Figure 26.3 Classification reports using stacked autoencoders with 3 hidden layers

Figure 26 show the classification reports of each classifier with stacked autoencoders pre-training. The *precision* is the ratio **TP / (TP + FP)** where **TP** is the number of true positives and **FP is** the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The *recall* is the ratio **TP / (TP + FN)** where **TP** is the number of true positives and **FN** is the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. The *F-beta* score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0. The *support* is the amount of testing labels for each different class (or digit in this case).

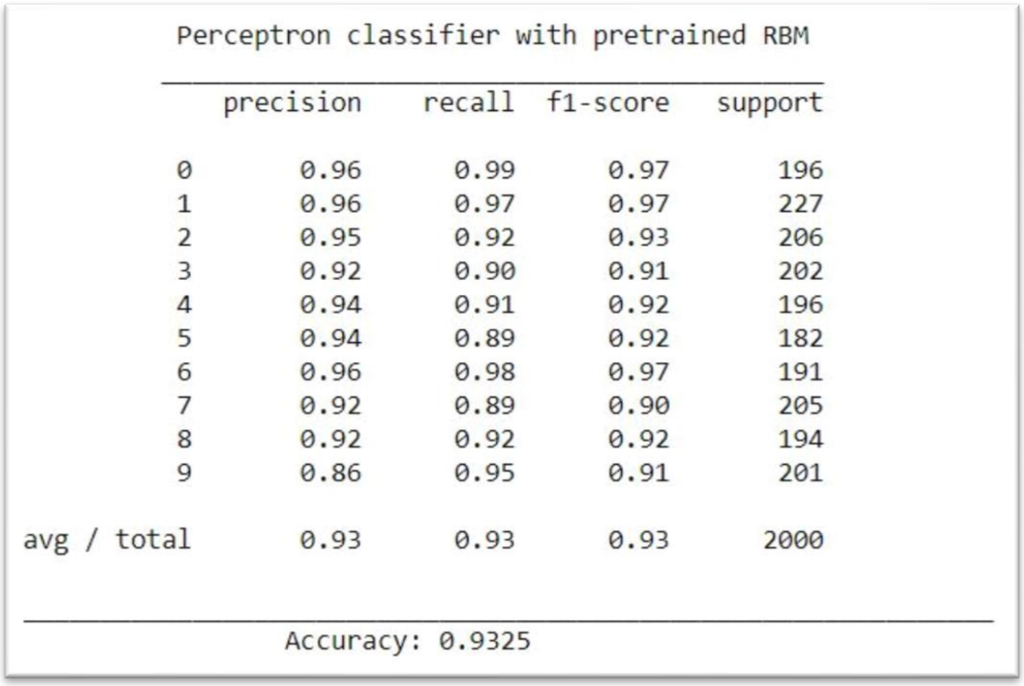
In Figure 26.1, 26.2, and 26.3, there are classifier performance reports from 1, 2, and 3 hidden layers, respectively. In terms of accuracy score, we can see that there is no significant difference between utilizing 1, 2 , and 3 hidden layers, although we can also see that using 1 hidden layers results in the lowest accuracy score. The precision, recall, and f1-score results in all classification reports confirm this. Also, in classifying digit '5' , the f1-scores show the lowest among all digits from 0-9. This means that using stacked autoencoders, the digit '5' is the hardest and the most challenging digit to classify.

4.2.2 **Digit Classification with DBN (Stacked RBMs)**



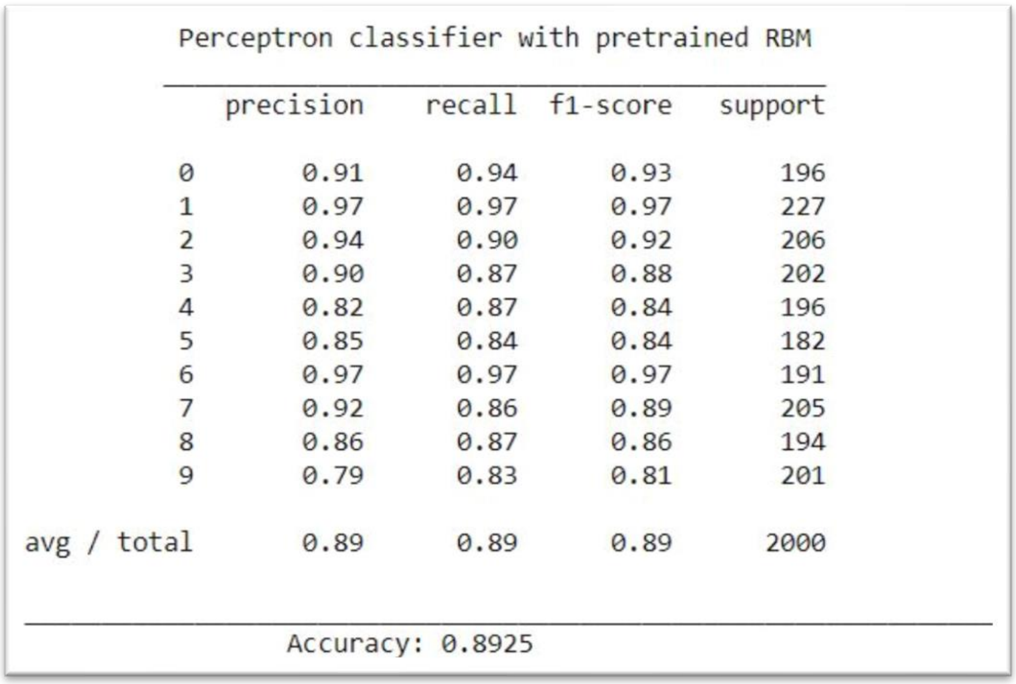
```
              Perceptron classifier with pretrained RBM
             ___________________________________________
                  precision    recall  f1-score   support

             0       0.99      0.97      0.98       196
             1       0.98      0.96      0.97       227
             2       0.95      0.94      0.94       206
             3       0.95      0.89      0.92       202
             4       0.93      0.98      0.96       196
             5       0.96      0.90      0.93       182
             6       0.95      0.98      0.97       191
             7       0.94      0.93      0.94       205
             8       0.88      0.94      0.91       194
             9       0.92      0.96      0.94       201

   avg / total       0.95      0.95      0.95      2000

 ________________________________________________________________
                  Accuracy: 0.946
```

Figure 27.1 Classification reports using DBN (stacked RBMs) with 1 hidden layer



```
              Perceptron classifier with pretrained RBM
             ___________________________________________
                  precision    recall  f1-score   support

             0       0.96      0.99      0.97       196
             1       0.96      0.97      0.97       227
             2       0.95      0.92      0.93       206
             3       0.92      0.90      0.91       202
             4       0.94      0.91      0.92       196
             5       0.94      0.89      0.92       182
             6       0.96      0.98      0.97       191
             7       0.92      0.89      0.90       205
             8       0.92      0.92      0.92       194
             9       0.86      0.95      0.91       201

   avg / total       0.93      0.93      0.93      2000

 ________________________________________________________________
                  Accuracy: 0.9325
```

Figure 27.2 Classification reports using DBN (stacked RBMs) with 2 hidden layer



```
                    Perceptron classifier with pretrained RBM
                  _____________________________________________
                      precision    recall  f1-score   support

                  0       0.91      0.94      0.93       196
                  1       0.97      0.97      0.97       227
                  2       0.94      0.90      0.92       206
                  3       0.90      0.87      0.88       202
                  4       0.82      0.87      0.84       196
                  5       0.85      0.84      0.84       182
                  6       0.97      0.97      0.97       191
                  7       0.92      0.86      0.89       205
                  8       0.86      0.87      0.86       194
                  9       0.79      0.83      0.81       201

        avg / total       0.89      0.89      0.89      2000

        __________________________________________________________________
                           Accuracy: 0.8925
```

Figure 27.3 Classification reports using DBN (stacked RBMs) with 3 hidden layer

In Figure 27 we can see the classification reports from classifiers with stacked autoencoders pre-training. The classification reports contain *precision* , *recall*, *F1*-score and *support* informations, which have the same meaning with those in previous classification reports.

In Figure 27.1, 27.2, and 27.3, there are classifier performance reports from 1, 2, and 3 hidden layers, respectively. In terms of accuracy score, we can see that there is a significant difference between 1 and 2 hidden layers versus 3 hidden layers. However, unlike the previous part of experiment (classifier with stacked autoencoders), we do not see that the digit five has the lowest f1-score among all digit(as the class). Moreover, there is no certain digit that having the lowest score in all number of hidden layers. Another difference with the stacked autoencoders case is that in stacked RBMs, using more number of hidden layers affects the classification performance negatively (lower accuracy score and F1-score).

5. Reflection

- Using Scikit Learn library from Python is quite straight forward.
- However, with libraries there are limitations on how to get the results. Especially trying to get specifically what we wanted to know.
- Computational effort is more in RBM due to the algorithm.
- In addition, it will take a long time to use cross validation in RBM.
- In MNIST digit classification using stacked autoencoders (with 1, 2 and 3 hidden layers), digit '5' seems the hardest class to classify. However, if we use DBN instead, there is no lowest digit in all number of hidden layers (lowest digit accuracy is random in all hidden layers).
- In MNIST digit classification using DBN, we saw significant difference using different hidden layers. However, in case of stacked autoencoders there is no significant difference at all.

6. Conclusion

- With an addition of iteration and number of nodes, it will result in low number of MSE error
- More number of iteration will store more better data in the weights, hence it will make a better reconstruction for the images
- Using both RBM and Autoencoders, we can reconstruct the data even if we have corrupted input
- Low number of hidden nodes in RBM will make some fluctuations. It is clearly seen with 50 hidden nodes. However, there are mild fluctuation present at 75 hidden nodes in 20 iterations as well.
- The lowest MSE error are obtained by 150 hidden nodes with 20 iterations using autoencoders. Hence, we prefer to use autoencoders more.
- In MNIST digit classification using deeper architecture, performance using stacked autoencoders is better than stacked RBMs in terms of accuracy (by using a same number of hidden layers)