CS 6910 Fundamentals of Deep Learning

Assignment 2 Analysis Report

June 9, 2020

Report by **Team 19**

ME17B084 Nishant Prabhu, **ME17B068** Soumyadeep Mondal, **ME16B177** Saye Sharan

---

Contents

1 Submission Overview

The submission consists of the following files. Since each code file is a Jupyter IPython notebook, relevant markdown text has been added along with in-line comments to support the code.

PyTorch has been used to design and train networks in all files for this assignment.

File	**Description**
`Assignment2_Task1.ipynb`	Jupyter notebook with code for task 1, dimensionality reduction with PCA and Autoencoders for dataset 1.
`Assignment2_Task2.ipynb`	Jupyter notebook with code for task 2, stacked autoencoder based pre-training of DNN for dataset 1.
`Assignment2_Task3.ipynb`	Jupyter notebook with code for task 3, stacked autoencoder based pre-training of DNN for dataset 2.
`Assignment2_Task4.ipynb`	Jupyter notebook with code for task 4, stacked BB-RBM based pre-training of DNN for dataset 2.
`Assignment2_Task5.ipynb`	Jupyter notebook with code for task 5, stacked GB-RBM based pre-training of DNN for dataset 1.
`RBM_Builder.py`	Custom library designed to generate and train binary-binary and Gaussian-binary RBMs. Classes and functions from this script are called in `Assignment2_Task4.ipynb` and `Assignment2_Task5.ipynb`.

2 Dimensionality reduction with PCA and AANNs

In this task, we were given a dataset with data points falling under one of five classes. Features for each data points were pre-extracted and provided as arrays of size (36, 23). Our task is to train an MLFFNN in three situations: (1) with unprocessed data, (2) with data reduced to a lower dimension using Principal Component Analysis (PCA), and (3) with data reduced to a lower dimension using Auto-Associative Neural Networks (AANNs) such as stacked Autoencoders from trained Encoder-Decoder networks.

2.1 Data Preprocessing

The data was imported as space separated files using `pandas` and flattened to NumPy arrays, such that every data point is now a vector of size (1, 828). We decided to ignore spatial dependencies between features (like those between pixels of images) as the features had already been extracted, perhaps by another network. Each data point and its corresponding label is then stored in two separate lists, which will serve as inputs to our networks.

2.2 Implementation

In this section, we have described the networks used and their performance for each sub task in task 1. The performance metrics stated are the best obtained overall; the numbers can change slightly over runs.

2.2.1 Unprocessed data

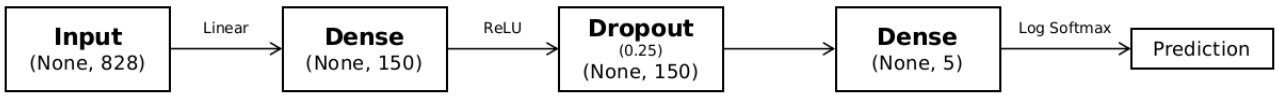

Figure 1: Model trained for unprocessed data

The model trained with unprocessed data is shown in **Figure 1**. Below are the results of evaluation at the end of training: the best obtained over several runs.

Parameter	Value
Average training logloss	0.8934
Average training accuracy	64.54%
Average testing logloss	0.9812
Average testing accuracy	65.45%

In **Figure 2**, we have the evolution of average logloss and accuracy across epochs as training progressed. As the training and testing metrics are close, we can infer that chances of the model having overfit the data are small.

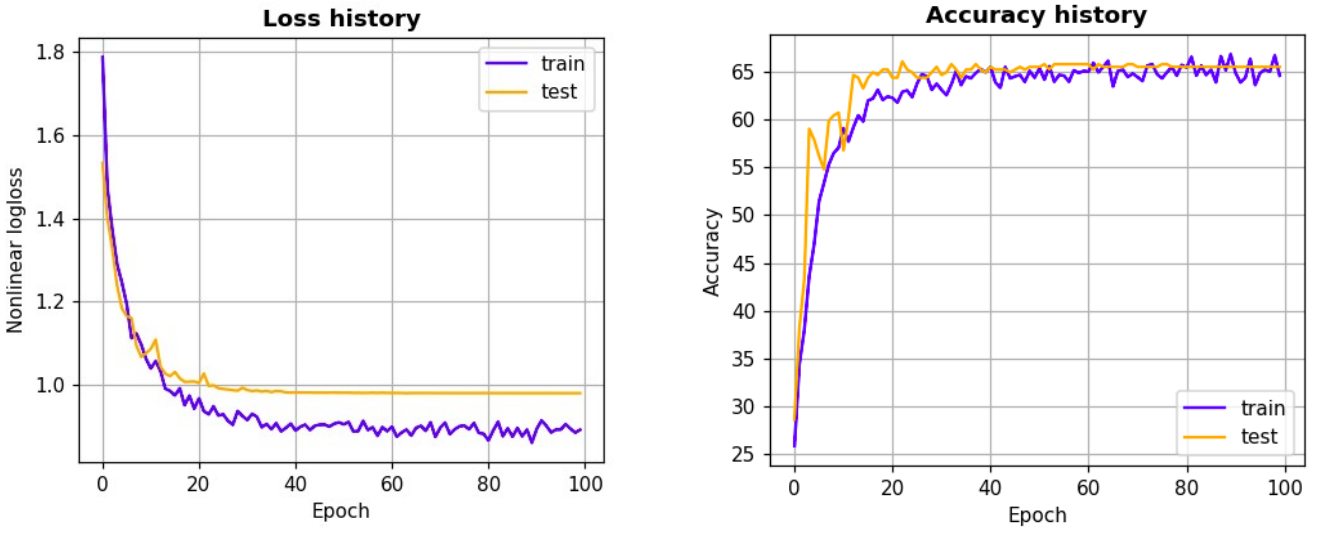

Figure 2: Loss and accuracy trends for unprocessed data

2.2.2 Data reduced in dimensions using PCA

Before we could perform any dimensionality reduction, we had to decide on an appropriate reduced dimensionality for the data. To do so, we fit a PCA object from Scikit-Learn on the data and observed the cumulative variance explained by the data from 1 to all 828 dimensions. The sum plateaus off at about 200 dimensions, meaning that most of the variance is captured by the 200 eigenvectors with highest 200 eigenvalues. The plot for the same is shown in **Figure 3**.

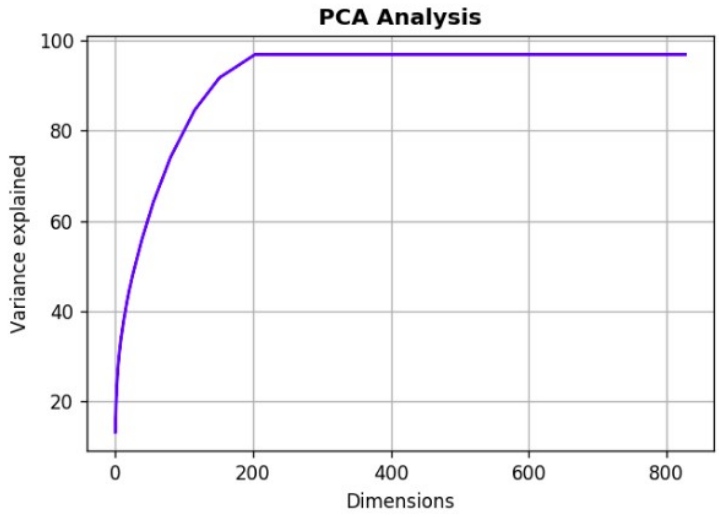

Figure 3: Cumulative sum of explained variance by reduced dimensions

The data is then reduced to 200 dimensions using the PCA object. Accordingly, the Input layer of our neural network has been modified; to reduce the number of parameters to be trained, the number of units in subsequent fully-connected layers were also reduced. The new network configuration for this task is shown in **Figure 4**.

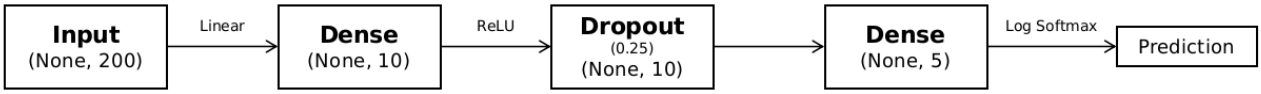

Figure 4: Model trained for data reduced in dimensions using PCA

Below are the results at the end of training (numbers shown are best obtained overall).

Parameter	Value
Average training logloss	0.5682
Average training accuracy	78.23%
Average testing logloss	0.8132
Average testing accuracy	72.19%

In **Figure 5**, we have the evolution of average logloss and accuracy as training progressed.

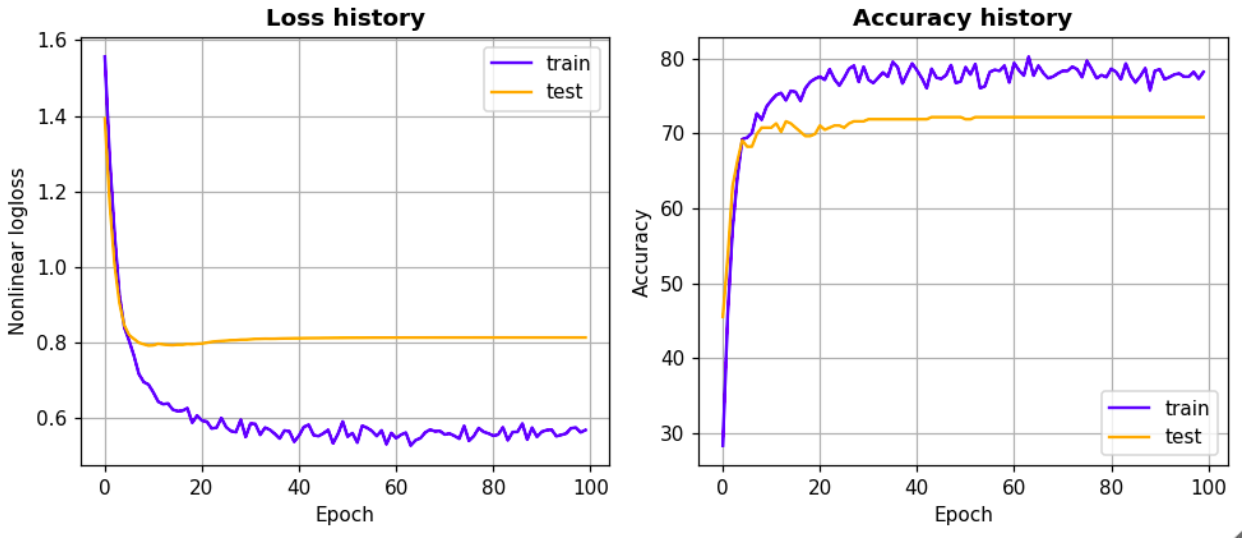

Figure 5: Loss and accuracy trends for data reduced in dimensions with PCA

Although this method has given a slight boost in testing accuracy (and significant raise in training accuracy), the gap between training and testing loss indicates a good chance of overfitting.

The model used for unprocessed data had 124,950 trainable parameters with 828 dimensions to train on. The thinned model for this case has 4,000 (number of parameters of outermost encoder-decoder network) trainable parameters with 200 dimensions to train on. The ratio of number of trainable parameters to number of features for the previous case is about 150, and for this case is 20. It might be possible that there are too many dimensions for the model to handle compared to the number of trainable parameters in the second case, resulting in overfitting.

Increasing the number of trainable parameters (50 units instead of 10 in the hidden layer) worsens the situation, as shown in **Figure 6**. So we are not very sure of the hypothesis we stated above.

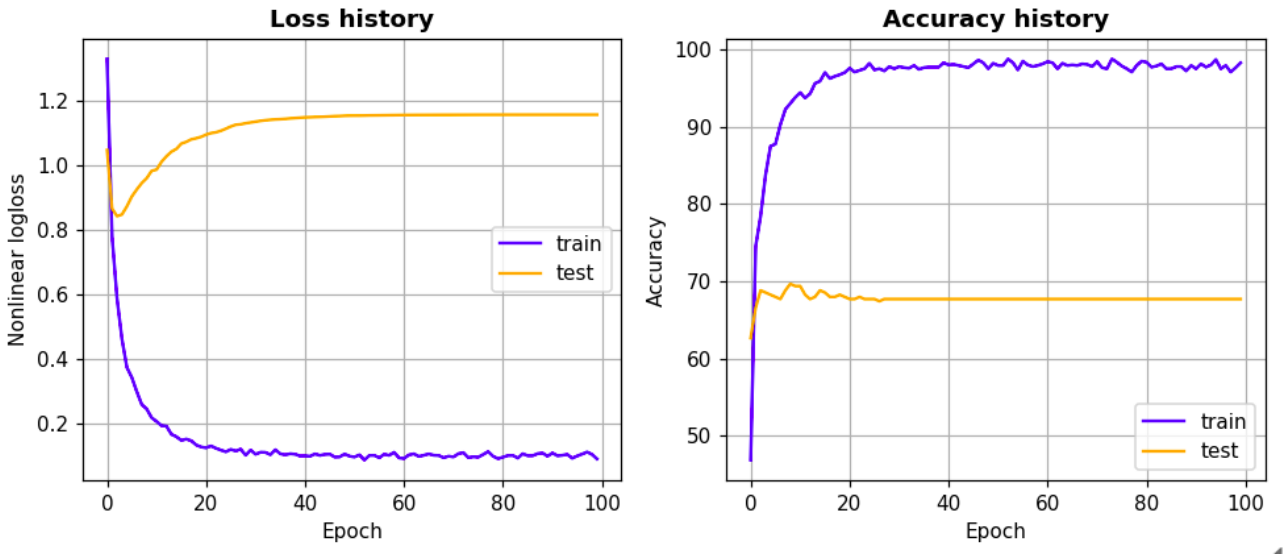

Figure 6: Loss and accuracy trends with more trainable parameters

2.2.3 Data reduced in dimensions using AANNs

In the first autoencoder, the dimensions are reduced from 828 to 300. In the second, it is reduced from 300 to 200. Mean squared error loss is used to tune the model parameters. **Figure 7** and **Figure 8** show plots of reconstruction errors of the networks post training. After training these, the encoders were stacked to generate the autoencoder and a dense layer with log softmax activation was appended for classification. Here are the results at the end of training. In **Figure 9**, we have loss and accuracy trends of the classifier trained with the autoencoder.

Parameter	Value
Average training logloss	0.6091
Average training accuracy	78.37%
Average testing logloss	0.9174
Average testing accuracy	66.29%

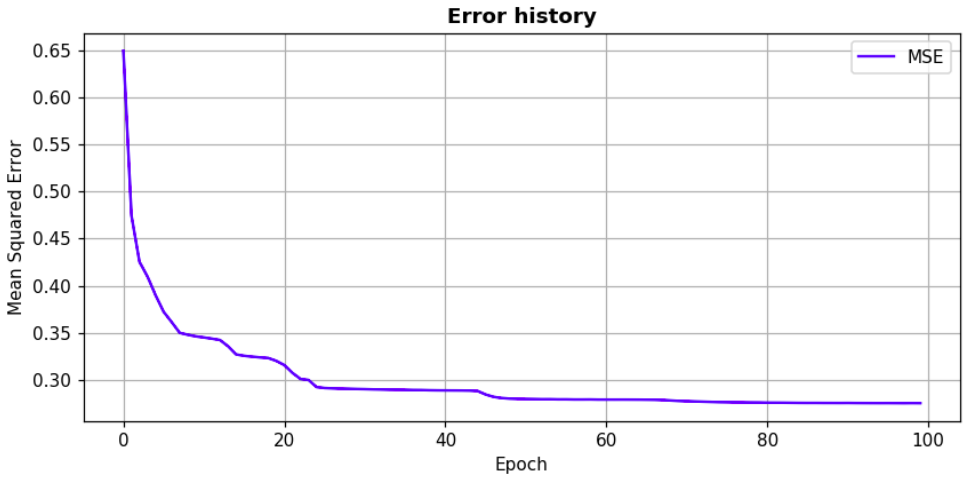

Figure 7: Reconstruction error of first encoder-decoder network

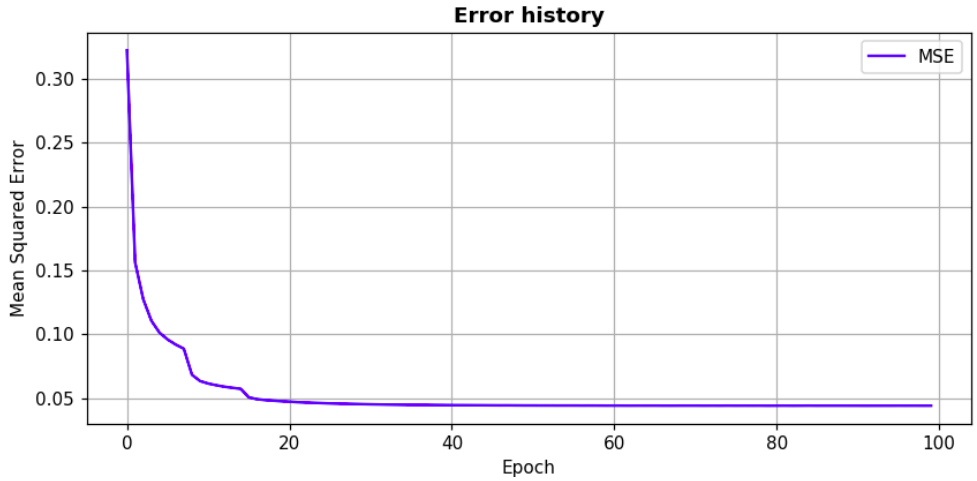

Figure 8: Reconstruction error of second encoder-decoder network

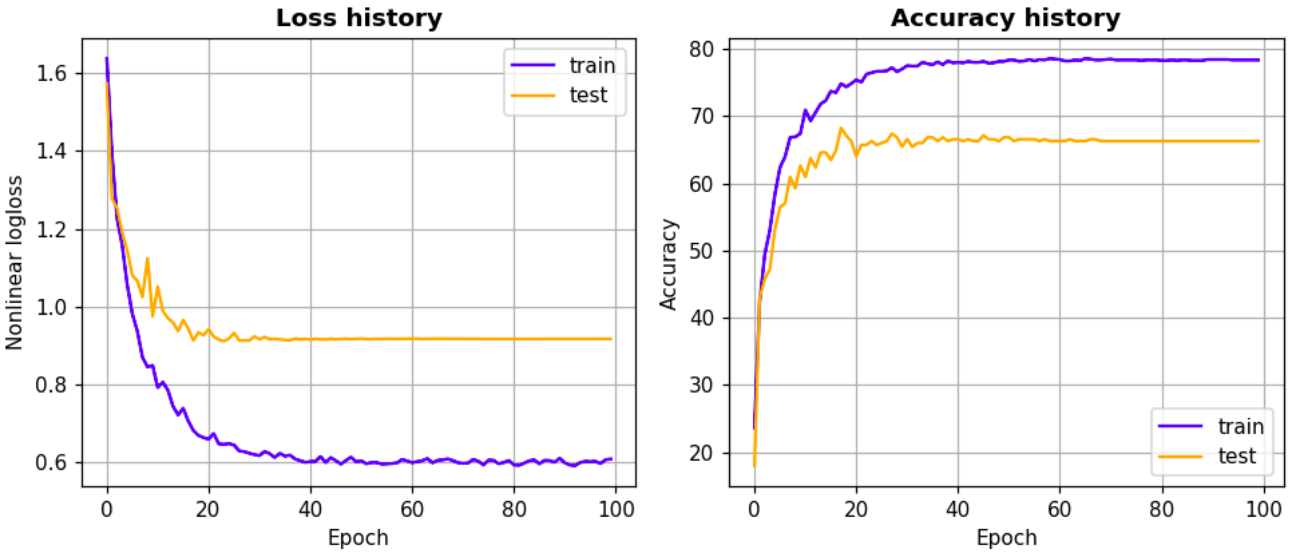

Figure 9: Loss and accuracy trends for data reduced in dimension using autoencoder

Compared to the experiment with PCA, we have slightly lesser validation accuracy while the training accuracy remains the same. Also, the gap between training and validation loss is higher in this case, indicating towards a higher chance of overfitting. This is perhaps due to the increased number of parameters that have to be tuned with the same data (encoder-decoder parameters + model parameters). One way to deal with this problem would be to have a steeper reduction in dimensions at every step, thus reducing the number of parameters to be trained in the encoder-decoder networks. This, however, did not help the model much in our trials.

2.3 Conclusion

Of the three methods, we believe unprocessed data gave the best trade-off between model performance and generalization. Although PCA and AANN based data reduction provided better model performance, they might have overfit the data resulting in poor generalization. Among the two, PCA worked better providing higher testing accuracy and better generalization than AANN.

A quick look at the scatter plot (**Figure 10**) of the data points after they were reduced to 3 dimensions by t-SNE reveals that Dataset 1 isn't very well separated into classes (at least in 3 dimensions).

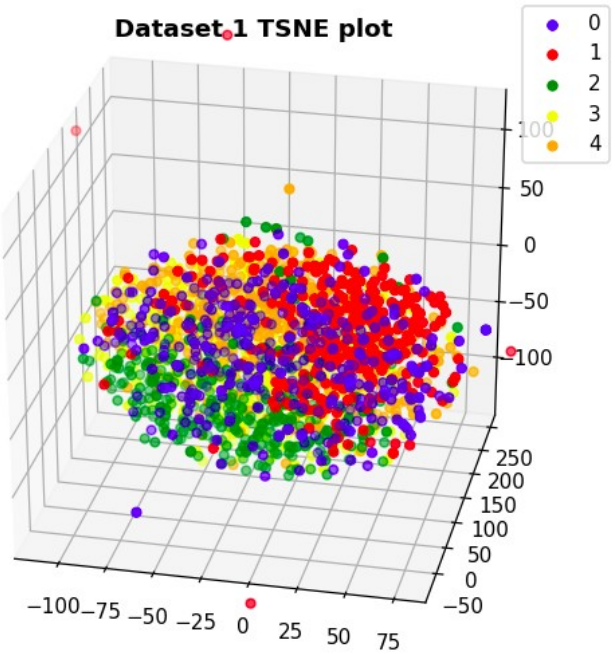

Figure 10: t-SNE transformed dataset 1 rendered in a 3D scatter plot

3 Stacked autoencoder based pre-training of DNN 1

For this example, we were asked to train a multi-layer feed-forward neural network using stacked autoencoder. To train the stacked autoencoder, we trained three encoder-decoder networks separately which reduced the dimensionality of the data first from 828 to 500, then from 500 to 300 and finally from 300 to 200. Each encoder decoder consists only of one hidden layer (bottleneck) apart from the input and output layer. The hidden layers are ReLU activated while input and output layers are linearly activated.

Here are the final reconstruction errors (MSE) of each of the encoder-decoder networks at the end of their training.

Network	Average MSE
Encoder-Decoder 1 (828 > 500)	0.0043
Encoder-Decoder 2 (500 > 300)	0.0001
Encoder-Decoder 3 (300 > 200)	0.0012

Figures 10, 11 and **12** show trends of reconstruction error for each network over 100 epochs.

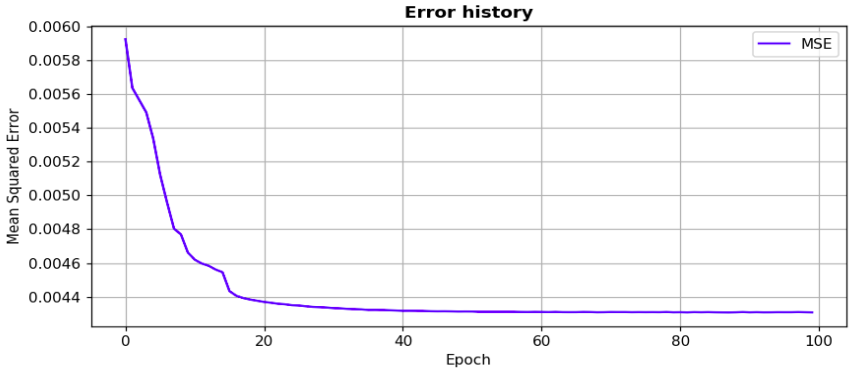

Figure 10: Reconstruction error for encoder-decoder 1

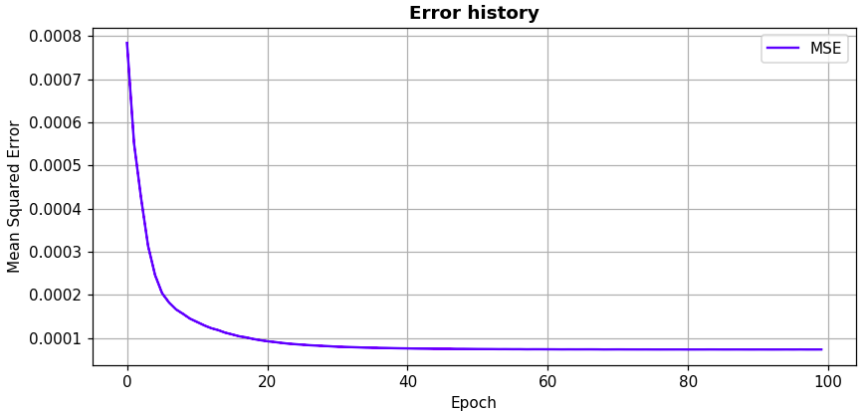

Figure 11: Reconstruction error for encoder-decoder 2

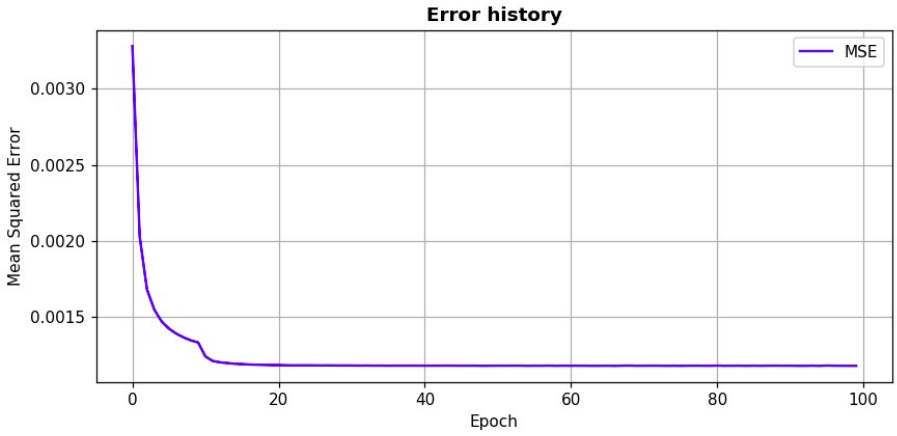

Figure 12: Reconstruction error for encoder-decoder 3

After each network was trained, we stacked their encoders in correct sequence to obtain the network architecture shown in **Figure 13**.

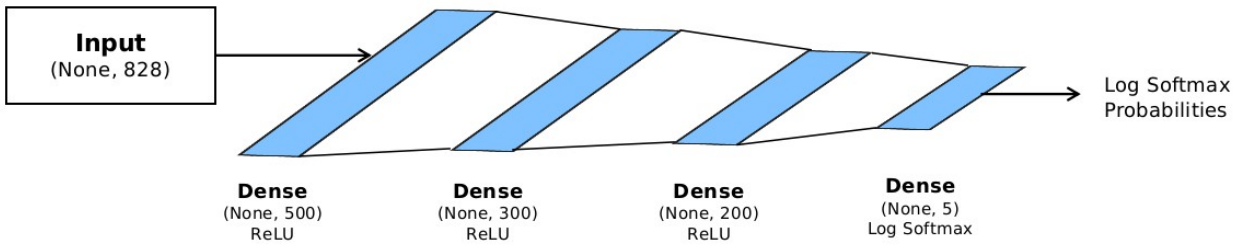

Figure 13: Stacked autoencoder architecture

Here are the results of training the model for 100 epochs.

Parameter	Value
Average training logloss	0.9700
Average training accuracy	55.55%
Average testing logloss	1.1919
Average testing accuracy	48.88%

Compared to networks trained in the previous task, this network has achieved much lower performance. This is likely due to the large increase in number of trainable parameters (equal to the number of parameters of the outermost encoder-decoder network, which is 828,000) compared to the AANN pre-trained network in task 1 (496,800 parameters). This resulted in some degree of overfitting, resulting in poor generalization and low performance on test data. Loss and accuracy trends of the network are displayed in **Figure 14**.

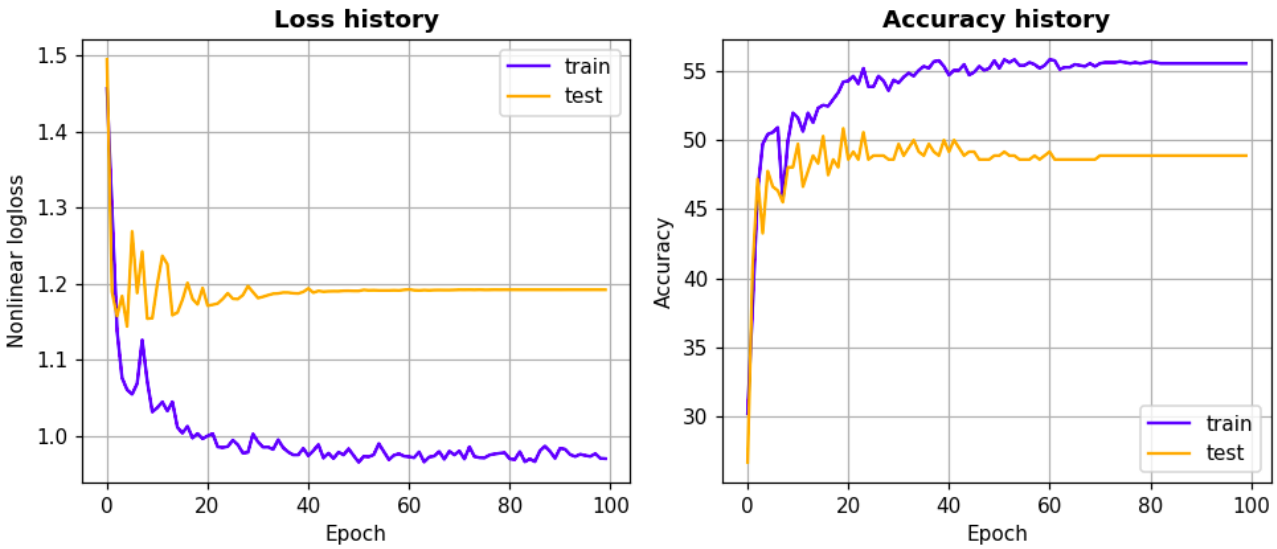

Figure 14: Loss and accuracy trends for stacked autoencoder based pre-trained DNN

We also noticed extreme instability of this network across training runs. There were instances when the model accuracy nosedived to low values (like 20%) starting from high values (like 60%). In few other (yet rare) cases, testing logloss would oscillate and settle at a high value. While random initialization can be a cause, the small size of training data is also one for sure.

P.T.O.

4 Stacked autoencoder based pre-training of DNN 2

Similar to the previous task, we had to train a stacked autoencoder based DNN for dataset 2. Dataset 2 consists of 30,000 images flattened into (1, 784) long arrays. The encoder architecture we used is shown in **Figure 15**.

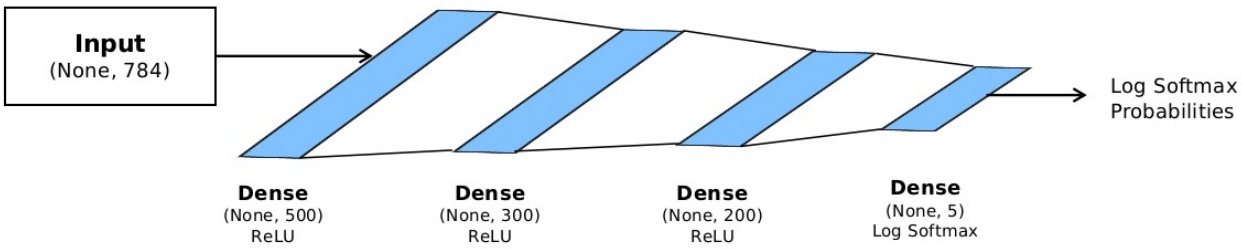

Figure 15: Stacked autoencoder architecture for dataset 2

Figures 16, 17 and **18** show reconstruction error trends of the three encoder-decoder networks.

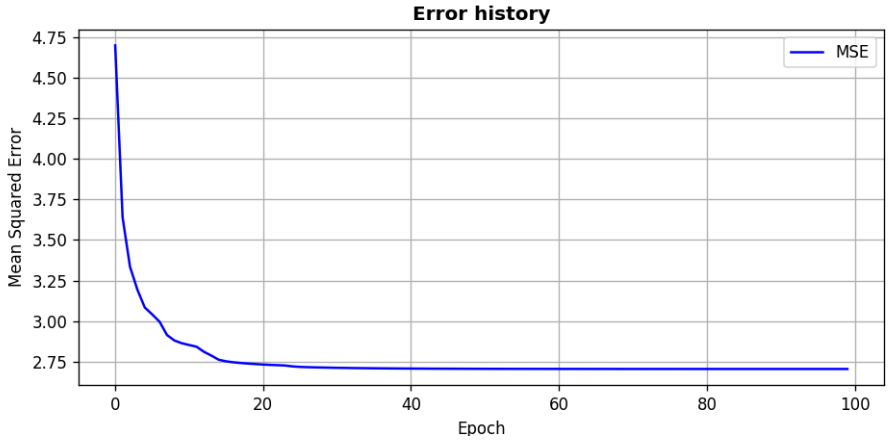

Figure 16: Reconstruction error for encoder-decoder 1

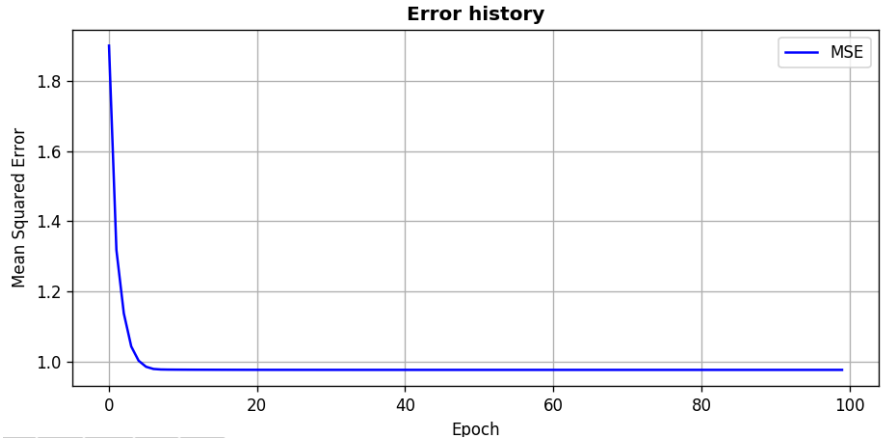

Figure 17: Reconstruction error for encoder-decoder 2

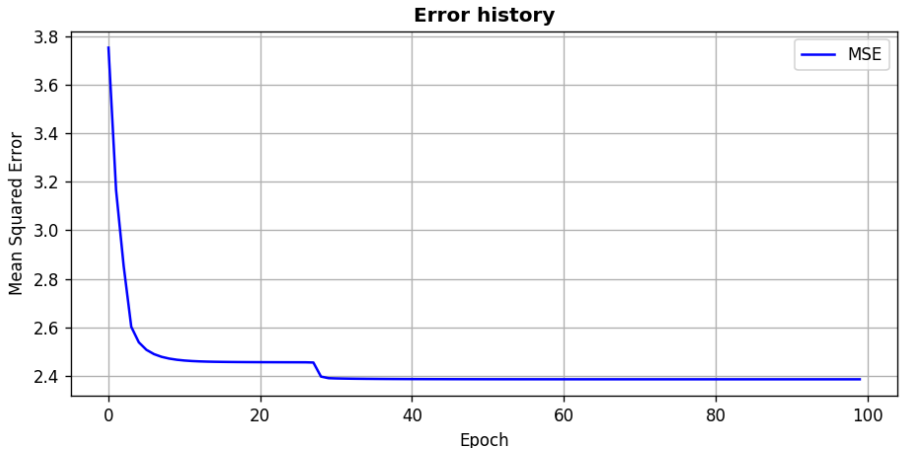

Figure 18: Reconstruction error for encoder-decoder 3

In the table on the next page, we have the results of the training the classifier (with the stacked autoencoder for 100 epochs).

Parameter	Value
Average training logloss	0.8203
Average training accuracy	52.94%
Average testing logloss	0.8295
Average testing accuracy	51.78%

In **Figure 19**, we have the loss and accuracy trends of the network as training progressed.

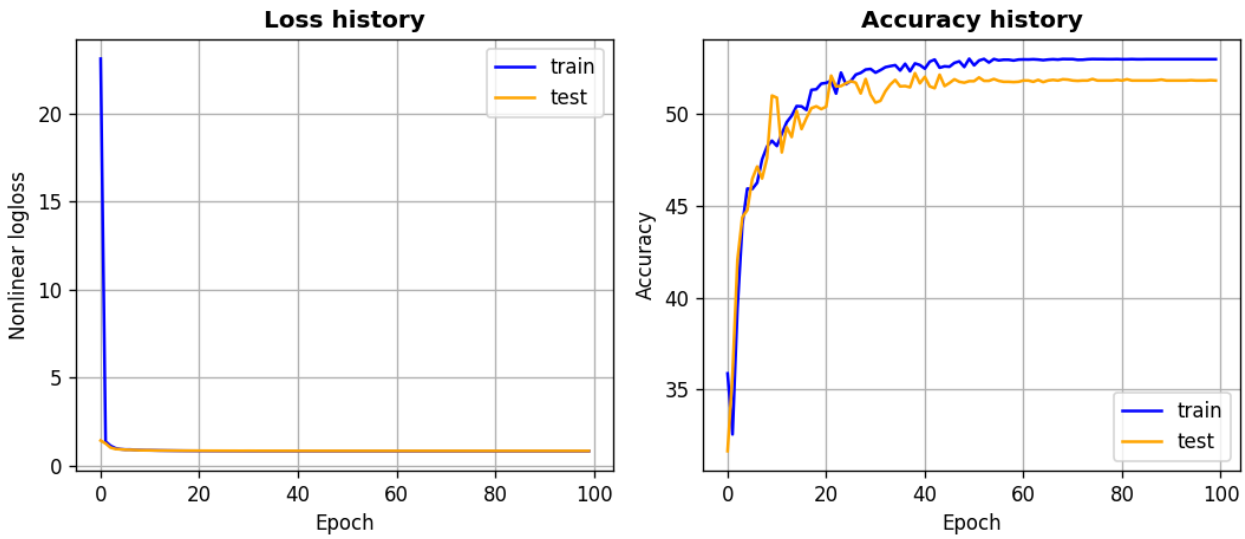

Figure 19: Loss and accuracy trends for stacked autoencoder based pre-trained DNN

Although the performance of this network is not very remarkable, what is worth noticing is the minuscule gap between the training and testing loss (also between respective accuracy). Compared to 1780 data points in the previous task, here we had 30,000 data points and less number of parameters to train. Thus the chance of overfitting was very low, which is apparent from the trends above. Even so, this network does not adhere to the rule of thumb for training DNNs: number of data points must be at least 10 times the number of parameters to train (for 393,000 parameters, we still only have 30,000 data points, which is less than 1% of the requirement).

Please also note that in this task and the previous, we did not render the autoencoders non-trainable (despite being pre-trained). This is to facilitate fine-tuning of their parameters for classification. In a way, this process informs the autoencoder which features extracted by it are important for classification and which are not. The model performance, indeed, is seen to improve with this setup.

Before we move on to the next task, **Figure 20** shows a plot of data points in dataset 2, reduced to 3 dimensions using t-SNE. Out of the five classes, we can see that 3 classes (yellow, pink and violet) have decent separability while the remaining two classes (orange and blue here) are difficult to separate. Visually estimating the sizes of the point clouds tells us that about half of the data can be separated easily; it is difficult to do so for the other half. This is somewhat similar to the model performance we have observed for this task.

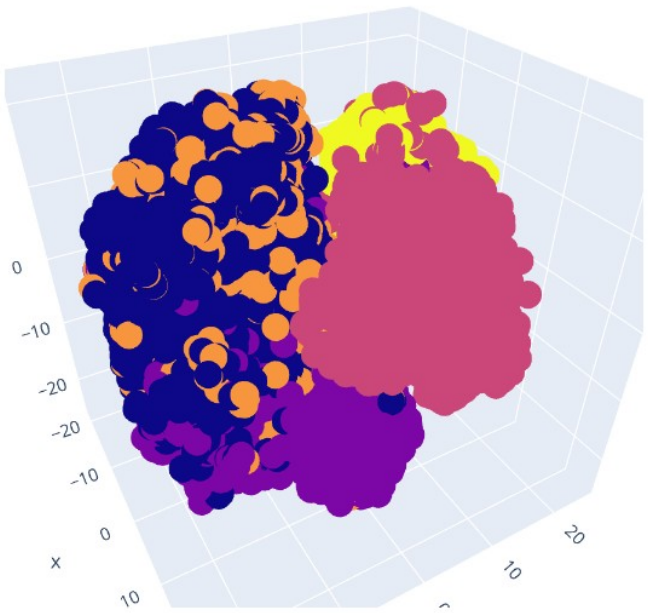

Figure 20: Scatter plot of dataset 2 reduced to 3 dimensions by t-SNE

For a better view, please refer to interactive plot in `Assignment2_Task3.ipynb`.

5 Stacked Binary-Binary RBM based pre-training of DNN

In this task, we were asked to represent data points in a latent space generated by stacked binary-binary Restricted Boltzmann Machines. Three binary-binary RBMs were individually trained using 10-step contrastive divergence to reconstruct data points. Logistic sigmoid function was used for computing all activations and conversion of real to binary data was performed according to this stochastic update rule.

$$P(v=1;a)=\frac{1}{1+e^{-a}}$$

5.1 Overview of RBM Builder script

This section aims to introduce the reader to major functionalities available in `RBM_Builder.py`. This file serves as a helper script for construction and training of RBMs of specified configuration. Although this script can build both binary-binary and Gaussian-binary RBMs, our discussion will be limited to the case of binary-binary RBMs (the other can be constructed simply by changing the value of an argument). The table below describes major functions in this script.

Function	Inputs	Description
`convert_to_binary`	Layer activations	Generate a matrix of uniformly distributed random numbers between 0 and 1, the same shape as the matrix of activations. For every element, if the random number is less than the activation value *a,* (which happens with probability *a*) then the state of that neuron is set to 1. Else, it is set to 0.
`calculate_h_from_v`	Visible layer activations	Generates hidden layer values using visible layer activations and current weight matrix.
`calculate_v_from_h`	Hidden layer activations	Generate visible layer values using hidden layer activations and current weight matrix.
`get_negative_sample`	Visible layer activations	Performs k-step contrastive divergence to generate a negative sample. Value of k should be specified during `RBM()` object initialization.
`calculate_grads`	Data points (positive samples) and negative samples	Calculates gradients for weights, visible biases and hidden biases based on the following rules. Weights : $<v_i h_i>_{data} - <v_i h_i>_{negative}$ Visible biases : $(v_i)_{data} - (v_i)_{negative}$ Hidden biases: $(h_i)_{data} - (h_i)_{negative}$

`update_params`	Gradients for weights and biases	Updates weights, visible biases and hidden biases of the network with given gradients.
`get_state_energy`	Data sample	Performs a forward a forward pass in the current state and calculates state energy using the Gibbs energy function.
`get_reconstruction_error`	Data sample	Generates one reconstruction of the data point (forward pass + reverse pass) and computes MSE between data point and its reconstruction.
`train`	`DataLoader()` object, number of epochs, verbosity level (optional)	Accepts data packaged by the customized `DataLoader()` class in this script. Runs through each batch of data samples for number of epochs specified and calls the above function in appropriate order to train the network. Verbosity level specifies the level of detail in which the model's learning progress will be output to `stderr`.
`get_hidden`	Data points (NumPy array-like)	Returns the output of hidden layer for every data point in the input matrix.

A few other functions have been defined for saving parameters and plotting, which are not crucial for our discussion. For the same reason, we have skipped the description of `DataLoader()` class included in the script. The reader may refer to `RBM_Builder.py` for more insight on these functions or explore other functions.

5.2 Data preprocessing

For the binary conversion method to work (described for `convert_to_binary` function in the previous section), it is necessary that input values lie between 0 and 1. For performing this, we used `MinMaxScaler()` from Scikit-Learn.

5.3 Model training

Three binary-binary RBMs were trained separately for learning latent representation of the data by reconstruction of data points. Dimensionality was first reduced from 784 to 500, then to 300 and finally down to 200. In **Figures 21, 22** and **23** we have displayed trends of model state energy and reconstruction error as training progressed.

Please note that 1-step contrastive divergence was used for generating negative samples in all 3 networks. We observed increase in model overfitting when the RBMs were trained for higher values of k.

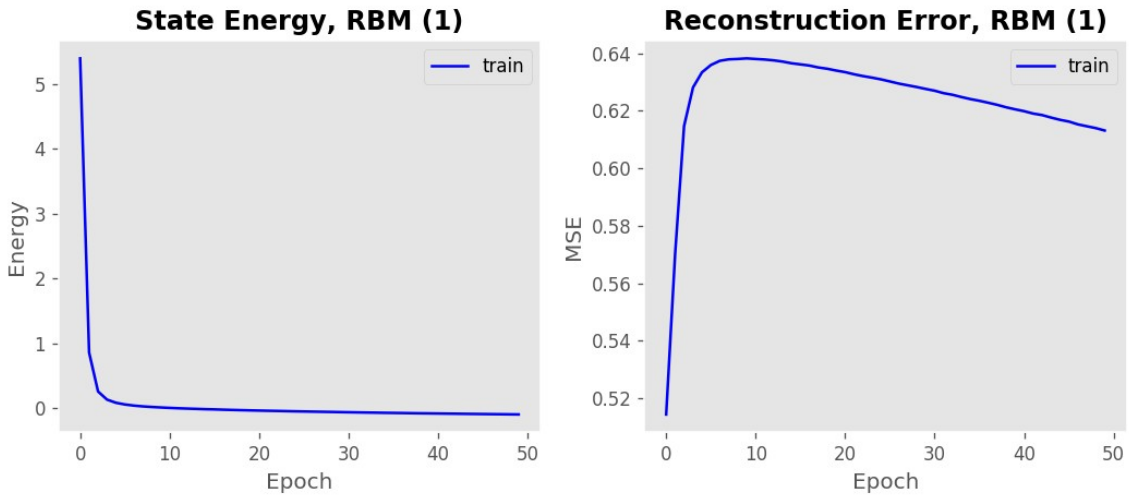

Figure 21: State energy and reconstruction error of RBM 1 over 50 epochs

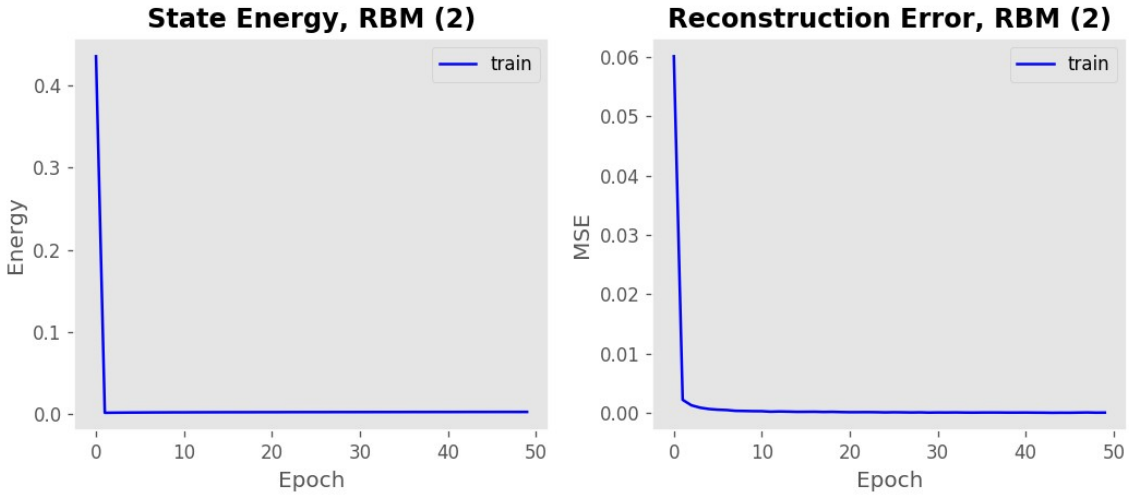

Figure 22: State energy and reconstruction error of RBM 2 over 50 epochs

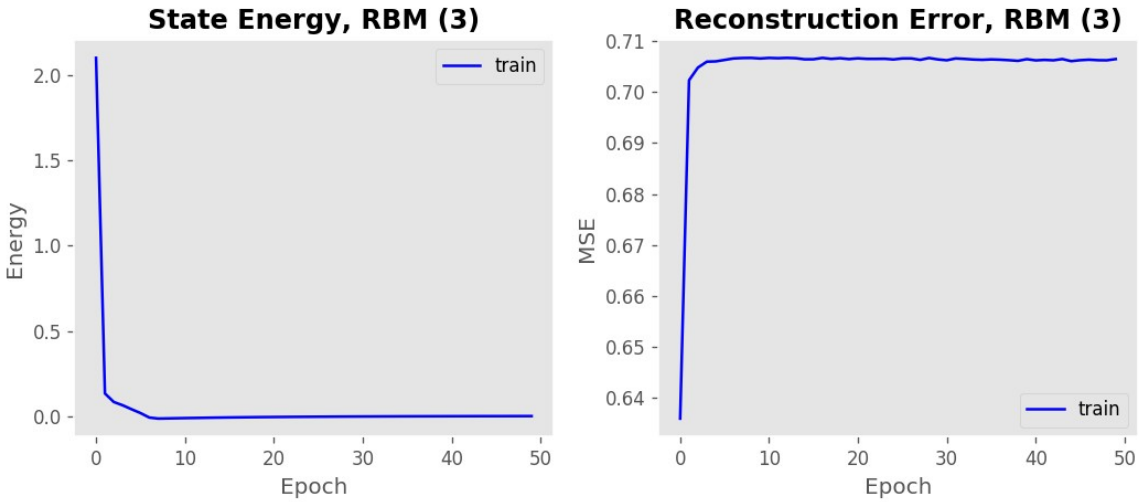

Figure 23: State energy and reconstruction error of RBM 3 over 50 epochs

The plots above somehow indicate that the models haven't been successful in reconstructing the data well. In **Figure 24**, we present a heatmap of the final latent representation (output of RBM 3) which will be passed to the classification layer.

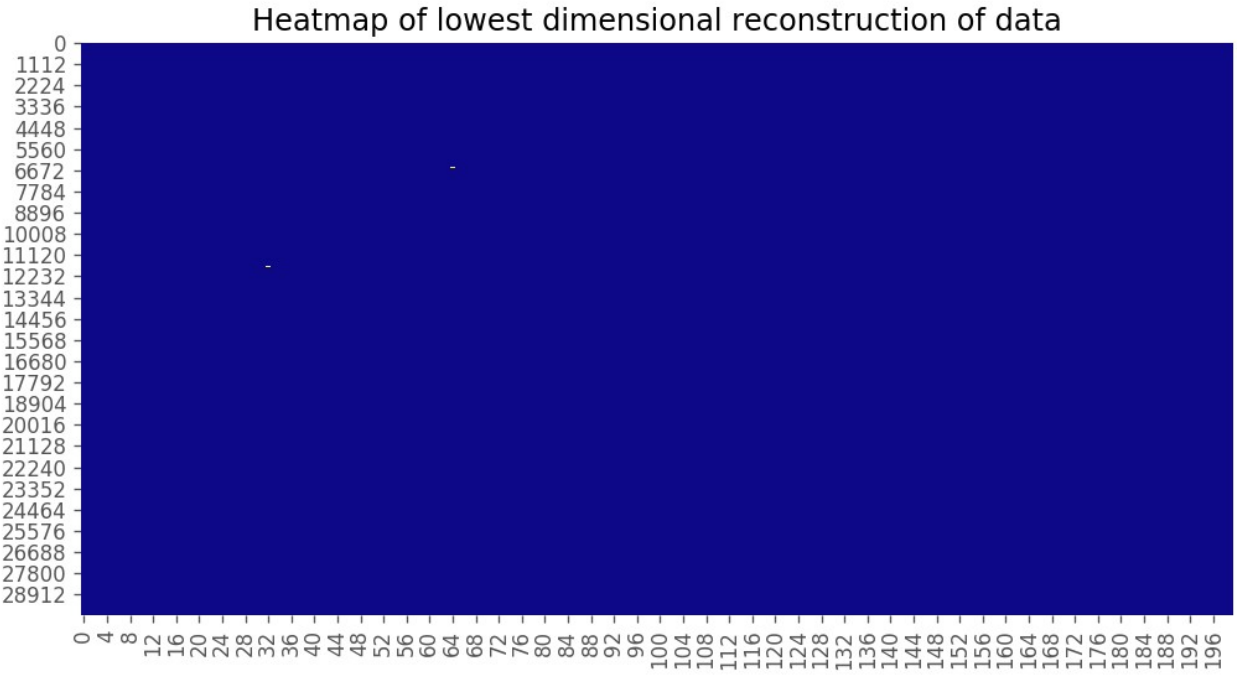

Figure 24: Heatmap of latent representation of dataset 2 in 200 dimensions

It is clear from the figure above that the networks have generate a pool of 0's (blue) with so few 1's interspersed between that they are almost invisible. With this reduced dimension representation, we cannot expect good classification results. Much to our expectation, **Figure 25** shows the loss and accuracy trends of the classifier as training progressed. The final training and testing accuracy and logloss have also been mentioned.

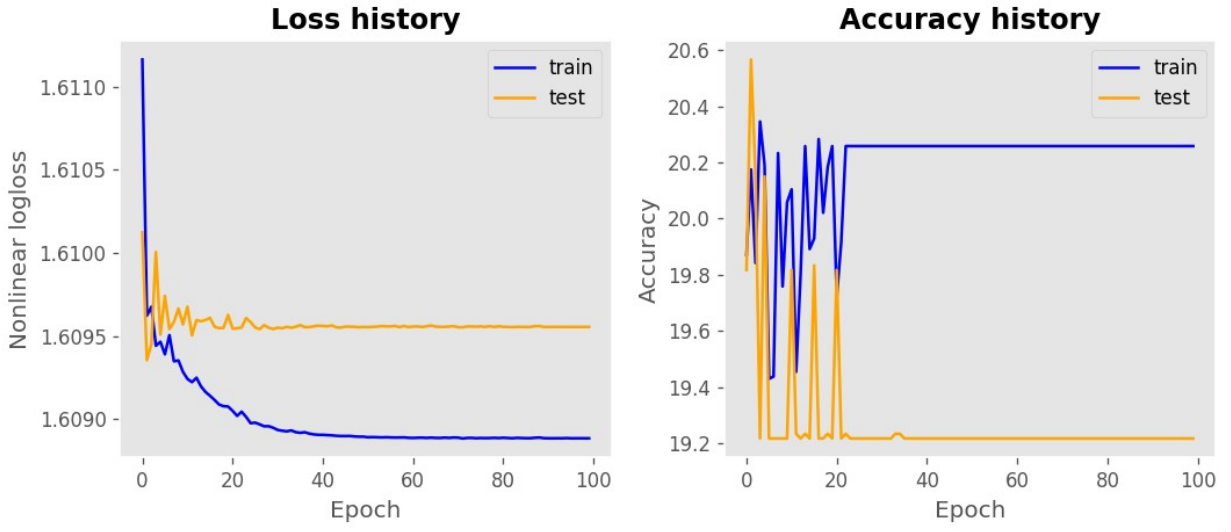

Figure 25: Loss and accuracy trends for stacked RBM based classification network

Parameter	Value
Average training logloss	1.6089
Average training accuracy	20.26%
Average testing logloss	1.6096
Average testing accuracy	19.22%

Inspecting the model's predictions revealed that it output the same class label for all data points, which gets it one-fifth of the answers correct. From the kind of data the classification layer has been provided, this is no surprise.

5.4 Possible reasons for failure

In a recent Quora post, Ian Goodfellow claimed that stacked Boltzmann Machines are obsolete. There are many reasons for this, but here are a few that are also relevant to our experiment.

- RBMs, primarily, are good at modeling data that is inherently binary (or can be represented in binary format with minimal loss of information) in nature. Pre-training networks using RBMs for real valued data often makes it perform very poorly.
- Stochastic updates and approximations made while performing contrastive divergence make the training process highly unstable and largely intractable. This is not a favorable feature when developing interpret-able machine learning systems.
- Popularity of autoencoder based networks for pre-training tasks. Although I'm confident that neural networks do not experience jealousy, the small user base of RBMs makes it difficult to find help when the network isn't generating results the way we expect it to. This is especially important for our case, since we were using RBMs to perform a task for which it generally isn't used (hence very less help from the internet).

Directly or indirectly, many other reasons have contributed to the results we obtained. In the next task, we introduce Gaussian-binary RBMs (often referred to as Gaussian RBMs) to try and preserve a little more of the data's native distribution.

6 Stacked Gaussian-Binary RBMs based pre-training of DNN

In this task, we had to train 3 Gaussian RBMs individually on dataset 1 and use the reduced dimensions representation generated by them to train a classifier. The modus operandi for this task remains the same as the previous one, with the only difference that the visible layer will not be converted to binary. **Figures 26, 27** and **28** show the model state energy and reconstruction error of each RBM as training progressed.

6.1 Data preprocessing

It is required that the visible layer inputs be normally distributed (mean variance normalized). Thus, we used `StandardScaler()` from Scikit-Learn to perform this operation.

6.2 Model training

Dimensionality was reduced from 828 to 500, then to 300 and then finally to 200. 1-step contrastive divergence was used for generating negative samples in all networks.

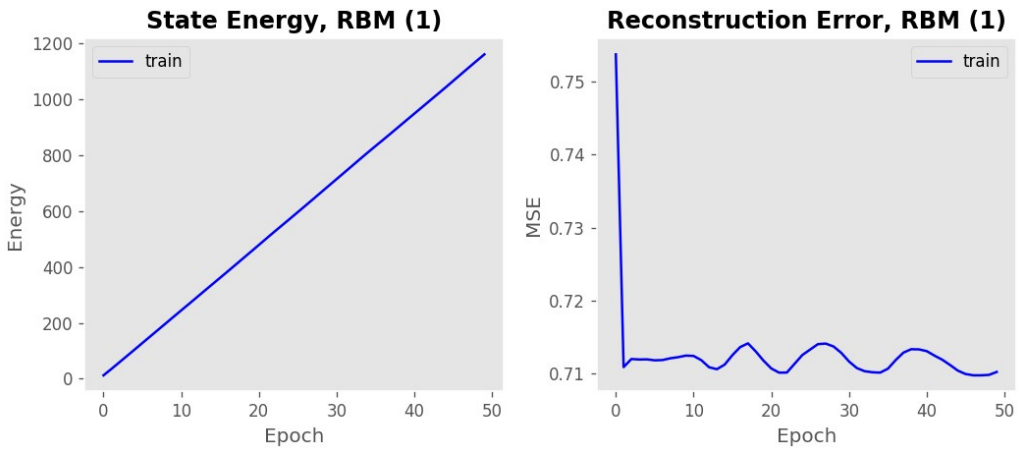

Figure 26: State energy and reconstruction error of RBM 1 over 50 epochs

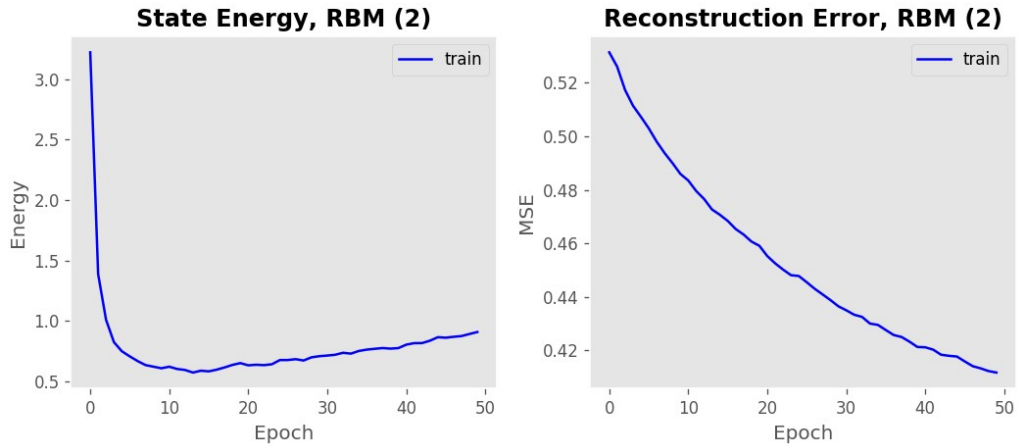

Figure 27: State energy and reconstruction error of RBM 2 over 50 epochs

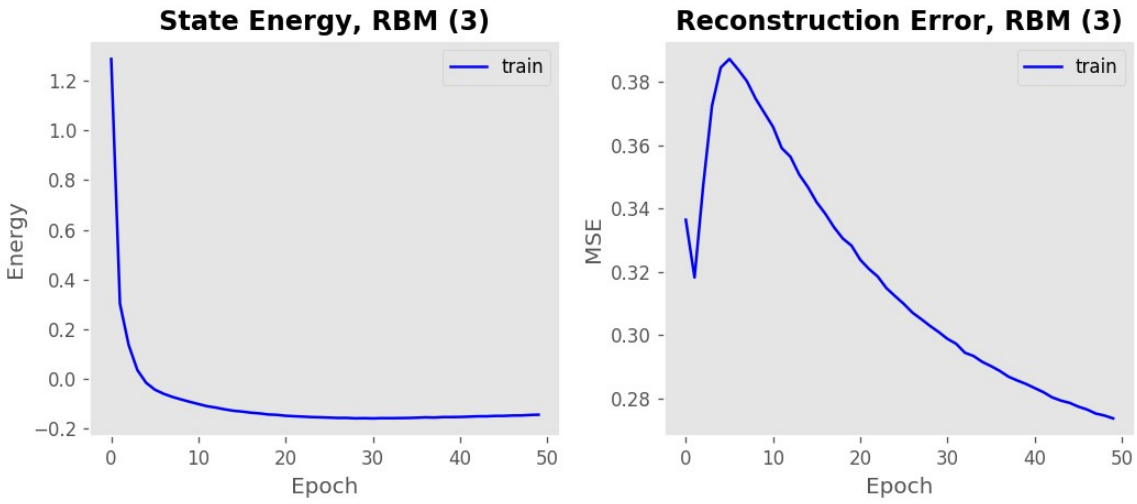

Figure 28: State energy and reconstruction error of RBM 3 over 50 epochs

The performance here looks slightly better than in the previous task; the reconstruction errors are reducing in the latter phase of training unlike the previous case. It seems as if the models did not reach convergence in 50 epochs. However, it was painstaking to train them for longer as they train very slowly. We did attempt training them for 80 epochs, but no significant increase in performance (classification) was observed by doing that.

The hidden representation of the third RBM was used for training the classifier. Here are the results of training for 100 epochs. **Figure 29** shows the loss and accuracy trends of the classifier as training progressed.

Parameter	Value
Average training logloss	1.3639
Average training accuracy	37.64%
Average testing logloss	1.7842
Average testing accuracy	26.12%

P.T.O.

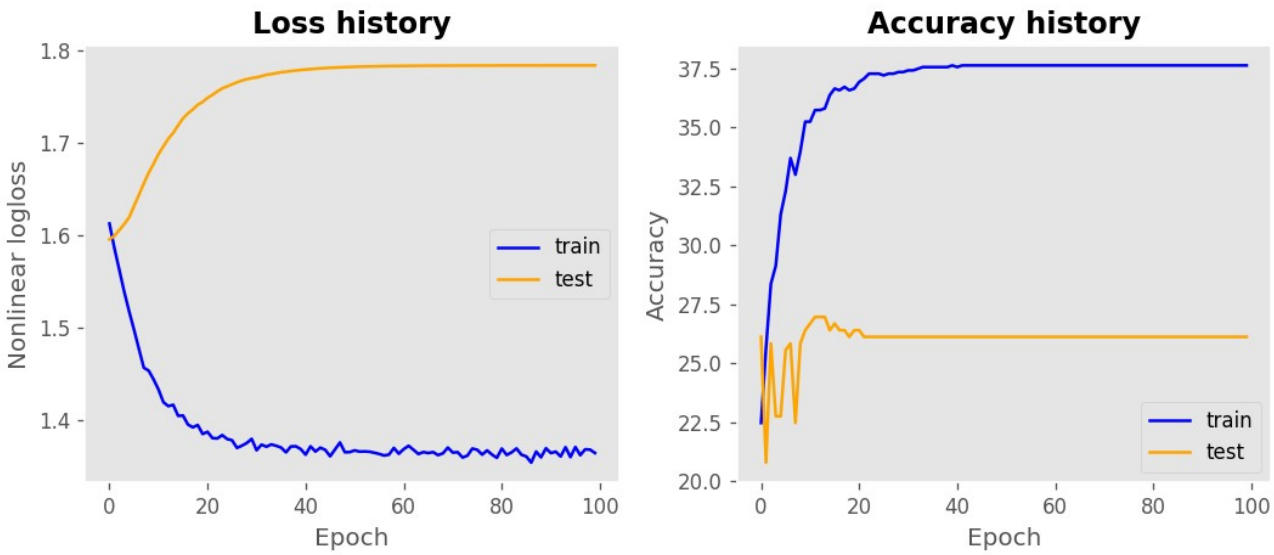

Figure 29: Loss and accuracy trends for stacked GB-RBM based pre-trained DNN

The performance of this classifier is marginally better than the previous one. This is probably due to the lesser loss of information from keeping the visible units real valued. However, as with previous networks trained on dataset 1, we see high degree of model overfitting from the loss trends. We suspect that the instability of this network's method of training adds to the small size of the dataset in causing overfitting.

7 End note

Based on the results of all tasks, we have concluded that AANNs, especially autoencoder based pre-training methods, are among the best in representing data in reduced dimensions and providing improved classifier initialization. Statistical methods like PCA are also equally well-performing, but the applicability of these methods remains limited to 2 dimensional data, to the best of our knowledge. AANNs, on the other hand, can deal with data of any number of dimensions; this becomes important in expanding use-cases of neural networks to diverse kinds of data including static images, videos, audio, etc.

8 References

- A practical guide to training Restricted Boltzmann Machines, Geoffrey Hinton, **CS Toronto**