Chapter 1

Stacked Autoencoder Implementations

5. Introduction:

This chapter will cover the performance of the first experiment. As mentioned in the previous chapter, the source code of the Python program from the 'name' repository on GitHub was used as a base for this thesis. The baseline algorithm, stacked-autoencoder (SAE) is implemented for this experiment and the code is modified to answer the following research questions.

Research Question 1- Can optimising the hyperparameter structures influence the modified stacked-autoencoder models and performs better than the baseline model?

Research Question 2- Can the regularization techniques influence the modified stacked-autoencoder model and performs better than the baseline model?

The first objective of this experiment is to try to improve the SAE model based on collaborative filtering (CF). Tuning the hyper-parameter is very crucial for this experiment. The updated model was trained with different hyper-parameter structures and optimise the model performance. The aim of adjusting the hypermeters structure is to find out whether the hyper-parameter can significantly affect the updated model performance and performs better than the standard model. The experiment is loosely based on **Hussain et al. (2018**) paper. The author implemented a deep autoencoder (DAE) model named FlexEncoder. He tuned the hyperparameters structures and other features to study the parameters performances on the prediction accuracy of the model (**Hussain et al. 2018**). Therefore, the investigation follows precisely the same structure from the paper, but the implementation of this work is different because the paper uses different ways of implementing the algorithm, and the results may be different.

The second objective of this experiment is to improve the SAE mode using regularization techniques. Batch normalization and dropout are the techniques implemented for this experiment. The purpose of using those methods is to make the model more stable during training and prevent overfitting. Implementing batch normalization is beneficial as it boosts deep learning by reducing the internal covariate shift (**Sergey et al. 2015**). Research has been done using batch normalization on recommender systems (RS) (**Hyeungrill et al. 2018)**. According to the research paper, batch normalization was applied to a supervised neural network to prevent overfitting and performed better than other neural networks based on collaborative filtering (CF) and traditional CF algorithms (**Hyeungrill et al., 2018)**. No research has been done applying batch normalization on the autoencoders (AE) model based on RS. Therefore, batch normalization was used for each layer on the SAE model and check if the performance is improving and suitable for the SAE model.

On the other hand, dropout is a technique to alleviate overfitting. The method was inspired by **Oleksii et al. (2017) paper**. The author optimized the deep autoencoder (DAE) model for the rating prediction task and implemented different dropout values on each model layer. The task for this investigation follows the similar techniques from Oleksii's work and implement dropout for each layer on the SAE model.

After training the model, the experiment uses a specific metric for testing the model. As mentioned earlier in the previous chapter. The baseline experiment computed the training loss and test loss values for Mean Squared Error (MSE). But for this experiment, the updated model implements RMSE (Root Mean Square Error) for evaluating the model's performance. RMSE is known to put more weight on prediction with large errors.

.

5.1 Implementations of the experiments

This section presents the implementation details of the experiment. The baseline SAE model and updated SAE model were implemented for the experiment. The updated model consists of different methods used for the experiment. Each method uses different hyperparameter strictures and unique features to boost the model performance. The performance of an updated SAE model using different methods were evaluated against the baseline experiment that is using the default hyperparameters.

5.1.1 Data Preparation:

MovieLens 100,000 ratings (100K) dataset was used to evaluate the SAE model. The data was collected through the MovieLens website. The dataset consists of 100,000 ratings for 1682 movies assigned by 943 users. Each rating is an integer between 1 and 5.

Dataset	User	Items	Rating
MovieLens 100K	943	1682	100,000

Table 5.1: Number of users, movies and ratings in the u.dat dataset

5.1.1.2 Preparing Training/Test Set

There were some changes to this experiment for preparing the sets. The original program does not have the columns header's name for 'u1.base' (Training set) and 'u1.test' (Test set).

The figure can be seen back in the previous chapter in section… For code modification, two Python lists were created for adding the values for each column header's name. Both indexes were then added to the data-frame for 'training_set' and 'test_set' variables.

```
training = ['user_id', 'movie_id', 'rating', 'timestamp' ] #Create each column
test = ['user_id', 'movie_id', 'rating', 'timestamp' ]     #Create each column
```

```
# Preparing the training set and the test set
training_set = pd.read_csv('u1.base', names=training, delimiter = '\t') # Read the file
test_set = pd.read_csv('u1.test', names=test, delimiter = '\t') #Read the file
```

Figure 5.1: Reading the training and test sets

After preparing for both sets, each column header's names are visualized as shown in below:

	user_id	movie_id	rating	timestamp
0	1	1	5	874965758
1	1	2	3	876893171
2	1	3	4	878542960
3	1	4	3	876893119
4	1	5	3	889751712

Figure 5.2: Training Dataset

The first three columns of the training_set represent the 'user_id', the 'movie_id', the 'rating' and the 'timestamp' respectively. The same thing is applicable for the test_set.

The column ‘timestamp’ is not very useful, as discussed in the previous chapter. Therefore the ‘timestamp’ column was removed. New updates for each column name headers are visualized as shown in below:

	user_id	movie_id	rating
0	1	1	5
1	1	2	3
2	1	3	4
3	1	4	3
4	1	5	3

Figure 5. 3: Updated Training Dataset

5.2 Experiment 1: Implementation of Stacked-autoencoder using Hyperparameter Optimization Experiment

The hyper-parameter has been adjusted for most of the structures: the layers or nodes of the stacked autoencoder model, Activation functions and the optimizers. The baseline algorithm was also implemented for the experiment. The hyper-parameter structures remained the same as in the original program of the source code. The only change for the baseline model is using the RMSE metric. The RMSE value for the baseline model was compared with the value for the updated model. The results for the baseline model help to methodically compare the results of the various methods of an updated model in a rationale way.

5.2.1 Forming user-item matrix

The process in forming a user-item matrix using Python was discussed in the previous chapter. Before training the model, the user ratings and the movies were integrated into a matrix where each row contains the user ratings for all the movies, he/she watched and the columns representing the movies. Training and test set must be converted into torch tensors for fast computation.

5.2.2 Applying Neural Network

SAE models were implemented for this experiment. As mentioned in the previous chapter, the model includes two encoding and two decoding layers.

The number of units for different layers has been adjusted for this experiment. The number of units remained the same for the baseline model. But for the updated model, the number of layers and units were modified. Different methods were implemented for the SAE architecture. The hidden layers for baseline model are 20, 10, 20 and the final layer containing the nodes same as the number of movies. For testing purposes, one of the methods includes increasing the neurons or nodes for the input features. The experiment tested different input features and check if the model performance has improved and compared with the baseline model. Another method of the updated model features only one fully connected layer. The methods remove the second encoding layer and first decoding layer. Therefore, the method employed one layer for encoding and decoding respectively. The hidden layer for the updated model is only 20 and the output, which is the same number of the input features. As mentioned before, SAE model features multiple encoding and decoding layers but now, the model has transitioned into a standard autoencoder (AE) as AE is dealing with one encoding and one decoding layers.

5.2.3 Activation Function:

The activation functions were explored for the SAE model. The Baseline model used sigmoid as it gives a suitable result for the model. The method of an updated model was tested using different activation functions such as Tanh, RELU, and SELU.

5.2.4 Loss Function:

Because this experiment is dealing with a regression problem, the loss function used had to be the MSE.

5.2.5 Optimizer Function

Optimizers were analysed for this experiment. The baseline model employed RMSprop. The updated model also employed RMSprop as it a popular optimizer for training the model and assist in fitting the parameters of the structure of neural network to the dataset. Adaptive momentum estimation (ADAM) was also applied for the updated model. Other Optimizers were used such as Adagrad, AdaDelta and SGD for comparison.

5.2.6 Other Hyperparameter structure:

The baseline model was trained for 200 epochs.

The mean corrector constant has been used for identifying training and testing losses. This constant is used to take into consideration the amount movies where user didn't rate the movies.

Other hyper-parameter structure such as learning rate and weight decay remained unchanged in terms of the values for the updated model.

5.2.7 Evaluation

After training the model, the calculation of loss was done by erasing all the new movie ratings and applying the ratings from the test data. The test loss should be quite comparable to the training loss. If the test loss is much higher than the training loss, then the model indicates overfitting. If the test loss is significantly lower than the training loss, then the model suggests underfitting. The trained model can then be used to predict a new user's ratings if he/she has predictions for at least a few movies among the available dataset.

In this section, RMSE is the primary metric used for the evaluation of the SAE model. RMSE computes the difference between the predicted and real ratings. So, the lower value of RMSE indicates a high accuracy. The baseline model implemented MSE for calculating the test loss. For the updated model, the experiment continued using the MSE to calculate the test

loss and then used RMSE, which was experimentally found to be the square root of the test loss value. The figure of the process is shown below.

```
test_loss: tensor(0.9547)

[ ] from math import sqrt

[ ] RMSE = sqrt(0.9547)

    RMSE

0.9770875088752287
```

Figure 5.4: Calculating the RMSE metric

The figure above shows the Python method returning the square root of the test loss value.

5.3 Experiment 2: Implementation of Stacked-Autoencoder Using Regularization Techniques

This section presents the implementation details of the second experiment. The updated SAE model was implemented using regularization techniques for the investigation. Batch normalization and dropout techniques were all applied for each layer on the SAE model. An updated SAE model's performance was evaluated against the baseline experiment that is using the default hyperparameters. The updated model used the same hyperparameter default setting from the baseline model. Unlike the previous experiment, there is no need to adjust the layers or tuning the SAE model's nodes. The activation functions and the optimizers remained unchanged for this task.

5.3.1 Passing number of batches through network

Unlike the previous experiment, a new hyperparameter structure has been introduced to this task. The neural network is dealing with the number of batch size. Adjusting the batch sizes helps the neural network to increase the accuracy of the model's performance and the batch normalization, having a better result. To handle the number of batch sizes, PyTorch features a custom Dataset which can be used with the built-on Dataloader to feed data when training the

model. In this case, the custom Dataset Class was implemented for the MovieLens dataset. The class provides an interface for accessing all the training or testing samples in the dataset. The figure of the custom Dataset is shown below:

```
batch_size  = 200

''' Dataset Class'''
class DatasetR(Dataset):


    def __init__(self, training_set, nb_users, transform=None):
        super(DatasetR, self).__init__()

        self.training_set = training_set
        self.nb_users = nb_users
    def __len__(self):
        return self.nb_users

    def __getitem__(self, idx):
        sample = self.training_set[idx]




        return sample
```

Figure 5.5: Custom Dataset Class

The figure above shows the batch size is set to 200. Custom Dataset Class implements the following methods:

- '___len__' is the method or function that returns the size of the dataset.
- '___getitem__' is the method or function which returns a sample from the dataset given an index. In this case, the method is returning the training set in the dataset.

The next step is to implement the DataLoader class so that it can accept the batch size for batching, shuffling, etc. The program iterates over the DataLoader to obtain batches of training data and train the model. The figure is shown below:

```
dataset = DatasetR(training_set = training_set, nb_users = nb_users)
train_loader = torch.utils.data.DataLoader(dataset, batch_size = batch_size, shuffle=True, num_workers=4, drop_last=True)


datasetTest = DatasetR(training_set = test_set, nb_users = nb_users)
test_loader = torch.utils.data.DataLoader(datasetTest, batch_size = batch_size, shuffle=True, num_workers=4, drop_last=True)
```

Figure 5.6: DataLoader class

5.3.2 Applying Neural Network

SAE models were implemented again for this experiment. The model features two encoding and two decoding layers. This time, batch normalization and dropout were applied for each layer on the model. The method for using the batch normalization is shown below:

```
class SAE(nn.Module):
    # Initializing the class
    def __init__(self, ):
        # making the class get all the functions from the parent class nn.Module
        super(SAE, self).__init__()
        # Creating the first encoding layer. The number of input corresponds to the number of movies
        #  Decide to encode it into 20 outputs
        self.fc1 = nn.Linear(nb_movies, 20)
        # Batch Normalization.
        self.bn1 = nn.BatchNorm1d(20)
        # Creating the second encoding layer. From 20 inputs to 10 outputs
        self.fc2 = nn.Linear(20, 10)
        # Batch Normalization.
        self.bn2 = nn.BatchNorm1d(10)
        # Creating the first decoding layer. From 10 inputs to 20 outputs
        self.fc3 = nn.Linear(10, 20)
        # Batch Normalization
        self.bn3 = nn.BatchNorm1d(20)
        # Creating the second hidden layer. From 20 inputs to nb_movies outputs
        self.fc4 = nn.Linear(20, nb_movies)
        # Creating the activation fucntion which will fire up specific neurons
        self.activation = nn.Sigmoid()
```

Figure 5.7: Applying batch normalization for each layer on the SAE model

Since this is an AE model, there was no need to apply the batch normalization technique for the last layer. The method for using dropout for each layer on the SAE model is shown below:

```
class SAE(nn.Module):
    # Initializing the class
    def __init__(self, ):
        # making the class get all the functions from the parent class nn.Module
        super(SAE, self).__init__()
        # Creating the first encoding layer. The number of input corresponds to the number of movies
        #  Decide to encode it into 20 outputs
        self.fc1 = nn.Linear(nb_movies, 20)
        # Dropout
        self.do1 = nn.Dropout(0.2)
        # Creating the second encoding layer. From 20 inputs to 10 outputs
        self.fc2 = nn.Linear(20, 10)
        # Dropout
        self.do2 = nn.Dropout(0.2)
        # Creating the first decoding layer. From 10 inputs to 20 outputs
        self.fc3 = nn.Linear(10, 20)
        # Dropout
        self.do3 = nn.Dropout(0.2)
        # Creating the second hidden layer. From 20 inputs to nb_movies outputs
        self.fc4 = nn.Linear(20, nb_movies)
        # Creating the activation fucntion which will fire up specific neurons
        self.activation = nn.Sigmoid()
```

Figure 5.8: Applying dropout for each layer on the SAE model

The experiment tested different dropout probabilities values such as 0.2 and 0.4. Dropout was not included in the final layer as shown the figure above.

The number of units for different layers has not been adjusted for this experiment.

5.3.3 Other Hyperparameter structure:

The baseline model was trained for different epochs. The number of epochs were tuned for the updated model based on the finding on the loss on training and test set.

Other hyper-parameter structure such as learning rate and weight decay remained unchanged in terms of the values for the updated model.

5.3.4 Evaluation

The principal metric for the evaluation of the SAE model is RMSE. RMSE has applied again, like the last experiment, from chapter 5. The metric computes the difference between the predicted and real ratings. The process for calculating the RMSE metric was mentioned in the previous chapter, so there is no need to explain it again.

5.4 Summary of the chapter

The SAE model implementation was discussed for each experiment. The results for the RMSE were obtained for both experiments and will be explained in Chapter 2.

Chapter 2

7. Experimental Results and Discussion

In this chapter, the details of the experiments and the corresponding results were discussed. As mentioned earlier in the thesis, the MovieLens dataset, the neural network model, stacked-autoencoder (SAE), and Root Mean Squared Error (RMSE) for evaluation metric were all implemented for the research project. The experiments fall into three categories. The investigation explored tuning different hyperparameter structures in the first category, and the performance was demonstrated between the baseline model and the updated model from chapter 5. The second category experiment follows a similar procedure from the previous investigation, but this time, exploring the effectiveness of the regularization techniques to boost the model accuracy. The updated model with the regularization methods was compared with the baseline model.

7.1 Experiment one: Implementation of Stacked-autoencoder using Hyperparameter Optimization Results

This section presents the results of experiment one from chapter 5. The results give insight into how the updated model is evaluated against the baseline model performed using the default hyper-parameter values. The updated model has completed a series of experiments using different hyperparameter optimization techniques and values. The investigation runs for 200 epochs.

The updated algorithm includes the graph for training loss values through the epoch. So, the chart was added for the different experiments in a single figure for easy comparison. The metric used is RMSE for measuring the model's performance. The lower the RMSE, the better the accuracy.

The base model has three hidden layers. The default hyperparameter is as shown in table 8.1 below:

Hyper-parameter Structure	**Value**
Epoch	200
Learning rate	0.01
Number of hidden layers	3
Number of features	1 (nb_movies)
Model Architecture	nb_movies, 20,10,10,20,20, nb_movies
Optimizer	RMsProp
Activation Functions	Sigmoid

Table 7.1: Baseline Model hyperparamter settings

7.1.1 Performance Analysis

This section present different hyper-parameter techniques were used for the experiments. After preparing training/test sets and preparing the training the network and corresponding test, the model was compiled. The hyper-parameter has been adjusted for most of the structures: Activation functions, the optimizers, the layers of the stacked autoencoder model, and the epoch size. The loss function for all the experiments is Mean Squared Root (MSE).

7.1.1.2 Activation functions

The effects of using various activation functions were performed and demonstrated. Hyperbolic tangent (TANH), and RELU were all tested. Sigmoid was also tested for comparison as well. Additionally, Sigmoid and TANH were combined to speed up specific neurons and tested. The hyperparameter structures for the updated model are as shown in table 8.2 below:

Hyper-parameter Structure	Value
Epoch	200
Learning rate	0.01
Number of hidden layers	3
Number of features	1 (nb_movies)
Model Architecture	nb_movies, 20,10,10,20,20, nb_movies
Optimizer	RMSprop
Activation Functions	Sigmoid, Tanh, RELU, SELU, Sigmoid plus Tanh

Table 7.2: Updated model for the SAE model (Activation Functions)

Firstly, after the training was performed, the activations functions were plotted. This visualisation serves as indicator of a number of epochs.

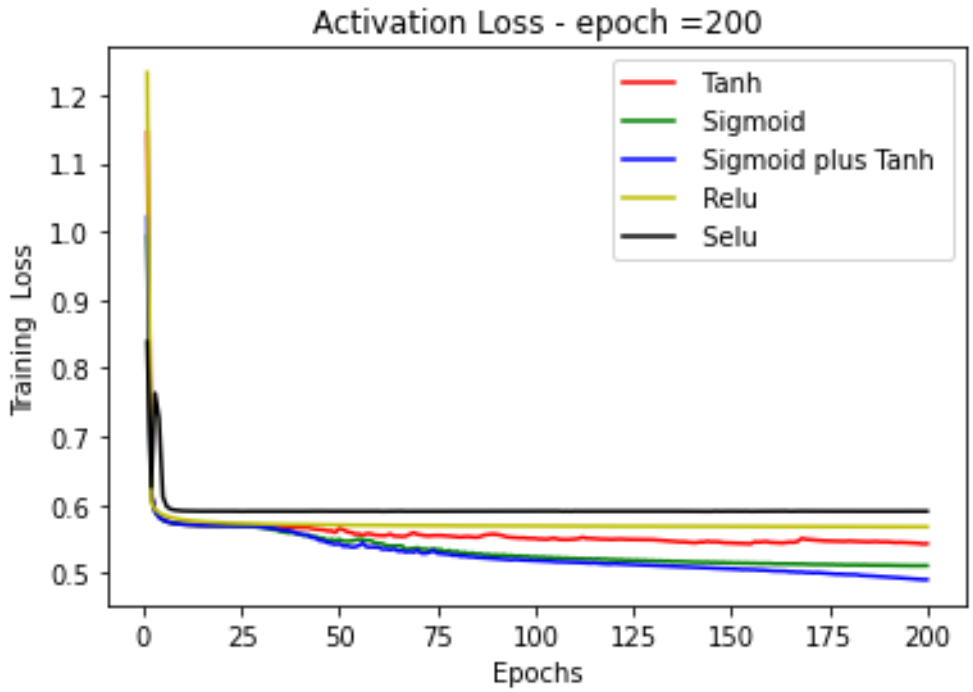

Figure 7.1: Activation Function training loss results

The figure above shows the loss's value over the epochs for the SAE updated model. In other words, it shows how the performance of the model is improving during the training. Different activation functions had plotted each other. Different line colors correspond to different activation functions. It is notable how some different functions continued to decrease after 45 epochs until the end and would probably continue falling. Sigmoid was chosen in the baseline model, and in this task, Sigmoid perform much better than Tanh, RELU, and SELU. RELU is another activation function, and it is a widely popular function but performs worse than

sigmoid in this experiment. Another task was to activate sigmoid and Tanh together, and the performance was improved and perform better than sigmoid alone. The following table contains the training loss values and computing time from each activation function for an update model:

Activation Function	**Training loss**	**Time**
Sigmoid	0.9147	5.0 minutes
Tanh	0.9692	5.0 minutes
Relu	1.0130	13.0 minutes
Selu	1.0538	5.0 minutes
Sigmoid plus Tanh	0.8753	5.0 minutes

Table 7.3: Training performances (Activation Functions)

After training the model, the test loss was calculated for each activation function. The updated model of the test loss values is shown in table 7.4.

Activation Function	**Test loss**
Sigmoid	0.9498
Tanh	1.0002
Relu	1.0165
Selu	1.0567
Sigmoid plus Tanh	0.9698

Table 7.4: Test performances (Activation functions)

As you can see the table 7.4 above, each activation function produces good results for computing the test loss. Sigmoid, Relu, and Selu perform very well as the training loss value (from table 1.3) are relatively closer to the test loss value. Sigmoid, however, produced a lower test loss value compared to other activation functions. Relu and Selu have the training loss, and test loss values greater than 0.99. Tanh was used, and the results show a minor overfitting model. Finally, Sigmoid and Tanh were combined and despite performing very well in the training set compared with other activation functions. Their performance in the test set was not superior then Sigmoid alone. Therefore, Sigmoid was the best activation function for this experiment, as it produces better loss values for the training and test sets.

The final step is to calculate the RMSE of an SAE model. RMSE measures the performance of the model. The lowest RMSE scores for each activation function were obtained are shown in table 8.5 below.

Activation Function	RMSE
Sigmoid	0.9745
Tanh	1.0000
Relu	1.0082
Selu	1.0279
Sigmoid plus Tanh	0.9847

Table 7.5: RMSE performances (Activation functions)

Table 7.5 above shows a low RMSE value for each activation function. Sigmoid produced a better result in terms of accuracy as this function has a shallow RMSE score compared to other activation functions. Tanh, Relu and Selu have the RMSE score greater than or equal to 1. 0000. Those functions have quite a similar RMSE score. On the other hand, Sigmoid, combined with Tanh, got an RMSE score of 0.9847. The score is very close to the sigmoid result.

7.1.1.3 Optimizers

The next optimization technique was using different optimizers to compare the performance between them. The baseline model employed RMSprop and was used again for the updated model. The optimizers for the updated model include ADAM, SGD, Adagrad, and Adadelta. The hyperparameter structures for the updated model are as shown in table 7.6 below:

Hyper-parameter Structure	Value
Epoch	200
Learning rate	0.01
Number of hidden layers	3
Number of features	1 (nb_movies)
Model Architecture	nb_movies, 20,10,10,20,20, nb_movies
Optimizer	RMSprop, ADAM, SGD, adagrad and adadelta
Activation Functions	Sigmoid

Table 7.6: Updated model for SAE Model (Optimizers)

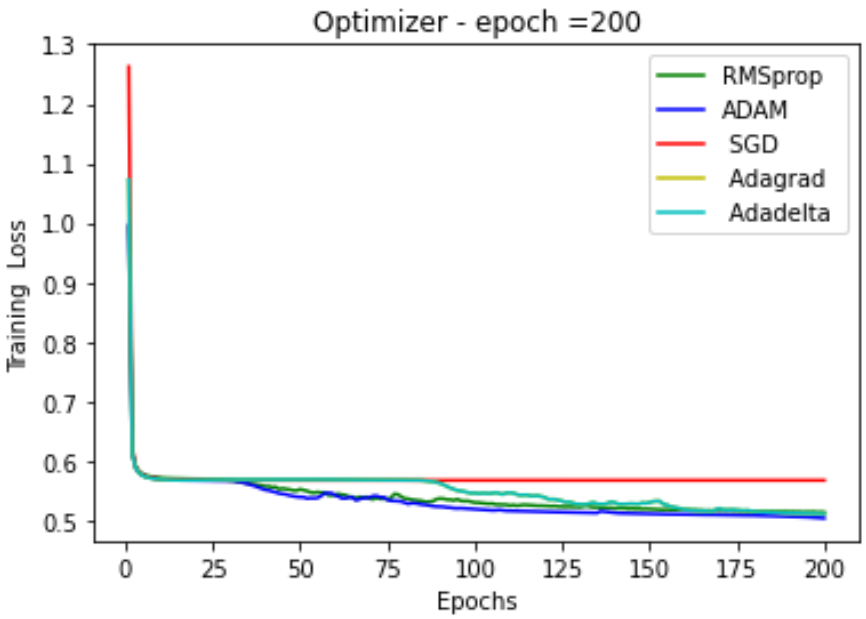

Figure 7.2: Optimizers training loss results

The figure above shows that the performance for each optimizer during the training. Different line colours represent to different optimizer. The graph shows that optimizer, Adam performed marginally better than RMSprop. RMSprop and AdaGrad lines are very similar. The other optimizers such as SGD and Adadelta performed the worst results during training. The following table contains the training loss values and computing time from each activation function for an update model:

Optimizer	**Training loss**	**Time**
RMSprop	0.9147	5.0 minutes
ADAM	0.8985	6.0 minutes
SGD	1.0119	3.0 minutes
Adagrad	0.9117	4.0 minutes
Adadelta	1.0142	6.0 minutes

Table 7.7: Training performances (Optimizers)

The test loss for each optimizer were calculated after training the model. The updated model of the test loss values is shown in table 7.8.

Optimizer	Test loss
RMSprop	0.9498
ADAM	0.9395
SGD	1.0189
Adagrad	0.9515
Adadelta	1.0216

Table 7.8: Test performances (Optimizers)

The test loss for each optimizer was obtained. RMSprop, ADAM, SGD, Adagrad, and AdaDelta test loss values are closer than their training loss values. ADAM has the lowest test loss value, followed by RMSprop and Adagrad.

Finally, the metric RMSE was calculated for the optimizers. The lowest RMSE scores for each optimizer were obtained are shown in table 7.9 below:

Optimizer	RMSE
RMSprop	0.9745
ADAM	0.9692
SGD	1.0094
Adagrad	0.9754
Adadelta	1.0107

Table 7. 9: RMSE performances (Optimizers)

Table 7.9 above shows a low RMSE value for each optimizer. The result provided by ADAM performs only marginally better than RMSprop and Adagrad. RMSprop and Adagrad show similar RMSE results. In contrast, SGD, alongside Adadelta have a score of RMSE greater than 1.00. The optimal RMSE values show that ADAM, RMSprop, and Adadelta were the best optimizers for this experiment in terms of prediction accuracy.

7.1.1.4 Adjustment of Layers and nodes.

The subsequent investigation covered adjusting the layers and the nodes of the SAE model. The first task was to increase the number of nodes for each layer, and the other part is to remove one layer each for encoding and decoding. The updated model is now a standard autoencoder since it's not dealing with multiple encoding and decoding layers. The baseline SAE model was compared with the standard AE model. The structures for the updated models are as shown in table 7.10 and table 7.11 below:

Hyper-parameter Structure	Value
Epoch	200
Learning rate	0.01
Number of hidden layers	3
Number of features	1 (nb_movies)
Model Architecture	nb_movies, 25,25,12,12,25,12, nb_movies
Optimizer	RMSprop
Activation Functions	Sigmoid

Table 7.10: Updated model for the SAE model NB 1

Hyper-parameter Structure	Value
Epoch	200
Learning rate	0.01
Number of hidden layers	1
Number of features	1 (nb_movies)
Model Architecture	nb_movies, 20, 20,nb_movies
Optimizer	RMSprop
Activation Functions	Sigmoid

Table 7.11: Updated model for the standard AE model NB 2

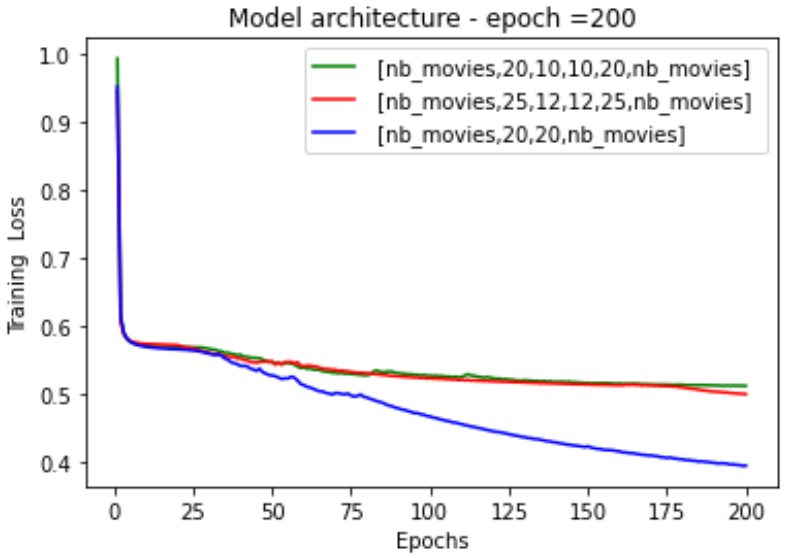

Figure 7.3: Model Architecture training loss results

The figure above shows the performance of different model architecture during the training. The colour line red represents the updated SAE model with increased nodes performed slightly better than the baseline SAE mode with the line colour green. In contrast, the blue line in the graph is the standard AE model, and the training performance was better than the SAE models in this task. The following table contains the training loss values and computing time from each model architectures for an updated model:

Model Architecture	**Training loss**	**Time**
nb_movies,20,20,10,10,20,20,nb_movies	0.9146	5.0 minutes
nb_movies,25,12,12,25,25,nb_movies	0.8931	6.0 minutes
nb_movies,20,20,nb_movies	0.7062	4.0 minutes

Table 7.12: Training performances (Model Architectures)

The training loss value for standard AE model is relatively low. The value is 0.7062. The model is easier to train and learns faster than SAE models.

The test loss for each model architectures were calculated after training the model. The updated model of the test loss values is shown in table 7.13 below.

Model Architecture	**Test loss**
nb_movies,20,10,10,20,20,nb_movies	0.9498
nb_movies,25,12,12,25,25,nb_movies	0.9407
nb_movies,20,20,nb_movies	1.0220

Table 7.13: Test performances **(**Model Architectures**)**

There is a big gap between the test loss value and training loss value from table 7.12 for standard AE. The model suffered a huge test loss value meaning the model is overfitting.

The calculation for RMSE for each model architecture is shown in table 7.14 below:

Model Architecture	**RMSE**
nb_movies,20,20,10,10,20,20,nb_movies	0.9745
nb_movies,25,25,12,12,25,25,nb_movies	0.9698
nb_movies,20,20,nb_movies	1.0220

Table 7.14: RMSE performances (Model Architectures)

Table 7.14 above shows a low RMSE value for each model architecture. The updated model with increased nodes has the lowest RMSE values than other model. But slightly outperformed the baseline SAE model. The standard AE received RMSE score relatively high than the SAE models.

7.2 Experiment 2: Implementation of Stacked-Autoencoder Using Regularization Techniques Results

This section presents the results of experiment two from chapter 6. the investigation is similar to the last experiment. But, this time, the experiment is conducted to investigate the effect of batch normalization and dropout on the training SAE model. The results give insight into how the updated model is performed using the regularization technique. Different methods were performed using the updated SAE architecture. The default hyperparameter structures

were kept the same. The model has seven layers having the layer sizes [nb_movies, 20,10,10,20,20, nb_movies]. The model uses sigmoid as an activation function and RMSprop as an optimizer. The loss function used is MSE and RMSE as a metric to evaluate performance. Also, for this investigation, one hyper-parameter structure is added for dealing with batch normalization. Each number of batch sizes were implemented for different methods.. The hyperparameters for the updated model are as shown in table 1 below:

Hyper-parameter Structure	**Value**
Epoch	200
Batch size	200
Learning rate	0.01
Number of hidden layers	3
Number of features	1 (nb_movies)
Model Architecture	nb_movies, 20,10,10,20,20, nb _movies
Optimizer	RMsProp
Activation Functions	Sigmoid

Table 7.15: Updated model (Batch Normalization) hyperparameters settings

7.2.1 Performance Analysis

This section present different techniques were used for the experiments. Batch normalization and dropout were applied for each layer on the SAE model

7.2.1.2 Batch Normalization results

The effects of using Batch Normalization strategy was performed and demonstrated.

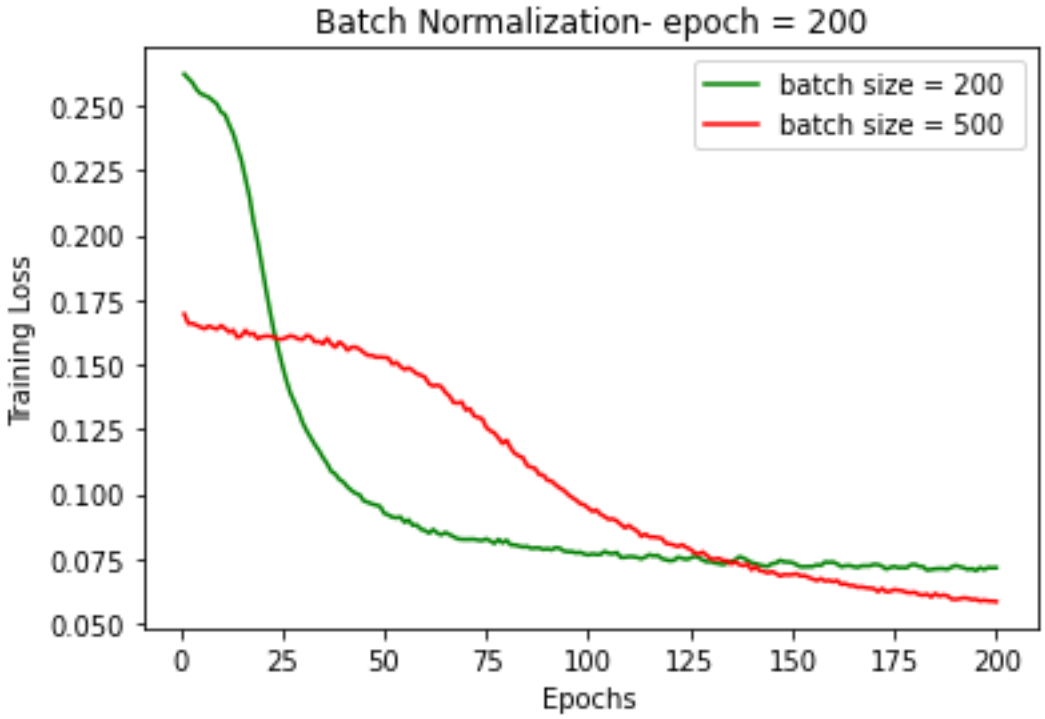

Figure 7.4: Batch Normalization effect- training loss results

The figure above shows the performance of the model improving during the training. The line colours correspond to different batch size. After adding batch normalization and setting suitable batch size and number of epochs, the training loss has improved. The figure shows how the effect of using batch normalization decreased after each epoch until the end. The results with different batch sizes were very superior than the results from the previous experiments.

The following table contains the training loss values and computing time from each batch size for an update model using the batch normalization strategy:

Batch Size	**Training loss**	**Time**
200	0.0714	1.0 minutes
500	0.0583	1.0 minutes

Table 7.16: Training performance (Batch Normalization effects).

The batch size 500 achieved better training loss value then the batch size with 200, as shown in table 7.16. Also, the training the model for each batch size was completed for only one minute. Adding the batch normalization for the SAE model helps to speed up the training and increases the generalization ability.

The test loss was then calculated for each batch size. The updated model of the test loss values is shown in table 7.17.

Batch size	Test loss
200	0.0796
500	0.0619

Table 7.17: Test performance (Batch Normalization effects)

The table 7.17 above shows both batch sizes produced good results for computing the test loss. The test loss value for each batch size is were closer to the training loss value from table 7.16.

The calculation for the RMSE was done for each batch size and achieved relatively low RMSE scores compared to the results from the previous experiments. The RMSE scored were obtained are shown in table 7.18 below

Batch Size	RMSE
200	0.2821
500	0.2487

Table 7.18: RMSE performance (Batch Normalization effects)

The effect of batch normalization helps the model achieved better RMSE performance than other models that were performed in earlier experiments. The batch size for 200 and 500 got the RMSE score of 0.2821 and 2487 respectively. Batch size was the key factor for training and testing the model

7.2.1.3 Dropout results

The effects of using dropout techniques were performed and demonstrated. Two methods were implemented for adding dropout strategy. One method includes applying dropout on every layer of the SAE model. The other method employed dropout on encoding layers. The dropout probability value is 0.2 for each method.

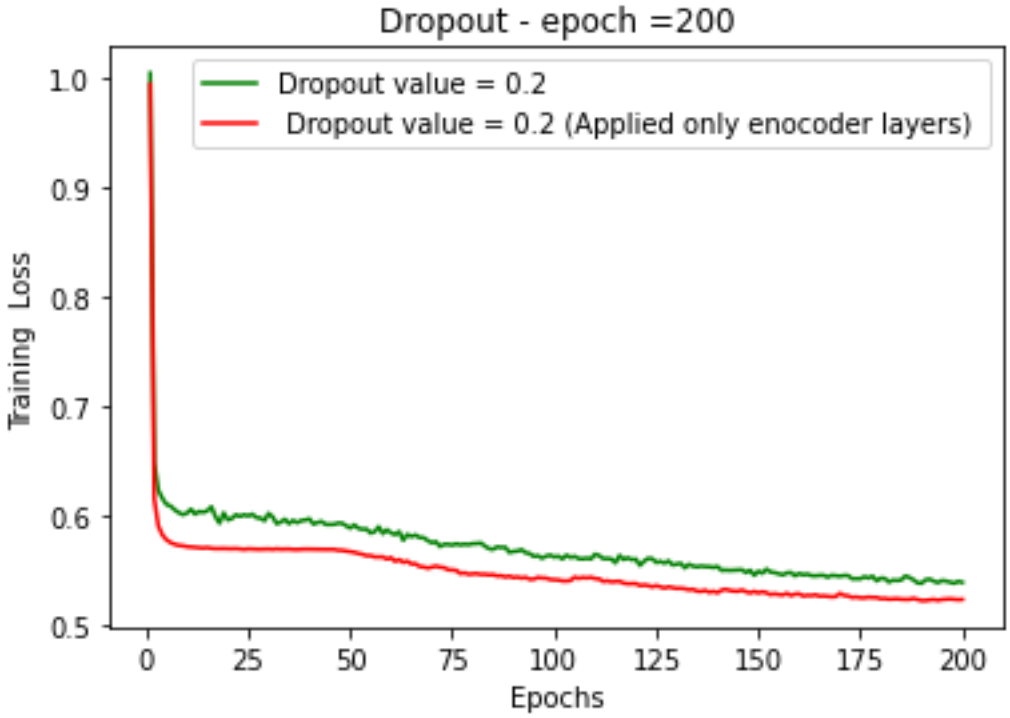

Figure 7. 5: Dropout effect- training loss results

The figure shows the performance of the model using dropout effect during the training. The green line represents dropout probability value of 0.2 while the red line represents the same value, but the value is applied only on encoding layer. The aim of using the dropout technique is to prevent the model overfitting. The training loss results is shown in the table below:

Dropout probability value	**Training loss**	**Time**
0.2	0.9615	5.0 minutes
0.2 (applied encoding layers)	0.9338	5.0 minutes

Table 7.19: Training performance (Dropout effects).

The training loss results were a bit high compared with the results from the baseline model. The method that applied dropout on encoding layers performed better than the other method.

But the main part of task is checking the test loss for each method. The test loss values should be relatively closer to the training loss value since dropout alleviates overfitting produce small gap between the losses. The test loss for each method were calculated. The test loss values are shown in the table.

Dropout probability value	**Test loss**
0.2	0.9626
0.2 (applied encoding layers)	0.9572

Table 7.20: Test performance (Dropout effects).

The result shows the first method with dropout value of 0.2 result in test loss value of 0.9626 which is relatively closer to the training loss value of 0.9615 from table. The dropout effect was the key factor for this task as this technique helped the updated prevented overfitting and produced a small gap between the training and the test loss values. Let see the performance of the RMSE shown in the table below:

Dropout probability value	**RMSE**
0.2	0.9811
0.2 (applied encoding layers)	0.9783

Table 7.21: Test performance (Dropout effects).

The RMSE performance using the dropout technique did not achieved lower score for each method. The method with the dropout applied on encoding layer showed similar RMSE score compared with the RMSE result of the baseline model from table. The other method achieved 0.9811 but overall, the dropout technique helped the updated model for each method to prevent overfitting and improved generalization.