

## CPSC 552 – Assignment 12

### Deep AutoEncoder based Recommender System

In one form of recommender system, we are given a sparse user-product rating matrix and our goal is to predict the empty fields of the matrix so that we can recommend new products to the user. The general problem can be formulated as set of  $m$  users and  $n$  products where some of the users may rank a few products as shown below.

	Item 1	Item 2	Item 3	...	Item n
User 1	2	3	?	...	5
User 2	?	4	3	...	?
User 3	3	2	?	...	3
...	...	...	...	...	...
User m	1	?	5	...	4

In the above picture, the columns represent the products. For example, user 2 has rated product 2 (item 2) with a rating of 4 and product 3 with a rating of 3, but the rest of the products or items have not been rated by the user 2 in the above user-item rating matrix. The ? indicates, no rating for the product by the given user. In the actual code, we can use a value of 0 for the ? indicating no rating in the beginning.

As explained in the lecture, we can use SVD to decompose the user-product rating matrix, into the three components, use a few of the top Eigen values, and then reconstruct the user-item rating matrix from the SVD components. The reconstructed user-item rating matrix will have most of the values converted to non-zero values.

Create a Python application called SVDRecommender. Then type the following code in the SVDRecommender.py file. We are assuming 18 users and 10 products in a simulated example.

```
import sys
import numpy as np

def main():
    UP_Rating = np.array([[0,2,0,0,3,0,0,0,0,0],
                          [4,0,0,0,3,0,0,0,5,0],
                          [0,0,0,0,0,3,0,0,0,0],
                          [0,5,0,0,0,0,0,0,0,2],
                          [0,0,0,0,0,0,0,5,0,0],
                          [5,2,0,0,0,0,0,0,0,0],
                          [0,0,0,3,0,0,4,0,5,2],
                          [0,3,0,4,0,0,0,0,0,0],
                          [0,0,4,0,0,2,0,0,0,1],
                          [0,0,0,0,5,0,3,0,2,0],
                          [1,5,0,0,0,4,0,0,0,0],
                          [0,0,4,0,5,0,0,3,0,2],
                          [1,0,0,4,0,0,0,1,0,0],
                          [0,5,0,3,0,0,0,0,4,0],
                          [4,0,0,0,4,0,0,0,0,0],
                          [0,0,0,5,0,2,0,0,0,0],
                          [0,3,0,0,0,0,4,0,0,0],
                          [3,3,0,0,0,0,0,0,0,5]])
    u, s, vh = np.linalg.svd(UP_Rating, full_matrices=False)
```

```

print(u.shape)
print(s.shape)
print(vh.shape)
num_components = 3
low_rank_reconstruction = u[:, :num_components] @ np.diag(s[:num_components]) @
vh[:num_components, :]
print('-----reconstructed user-product rating matrix')
print(np.round(low_rank_reconstruction))

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

The above program uses 3 Eigen values to reconstruct the user-product rating matrix. The reconstructed matrix is less sparse and can be used to recommend a product to the user that the user has not rated previously. For example, the second user will be recommended product number 7 (shown in bold below) as that column has the highest reconstructed rating (not including the products that were already rated by user 2)

[4, 0, 0, 0, 3, 0, 0, 0, 5, 0]	original recommendations by user 2.
[2. 0. 1. 1. 4. -0. <b>2.</b> 1. 3. 1.]	reconstructed data for user 2

```

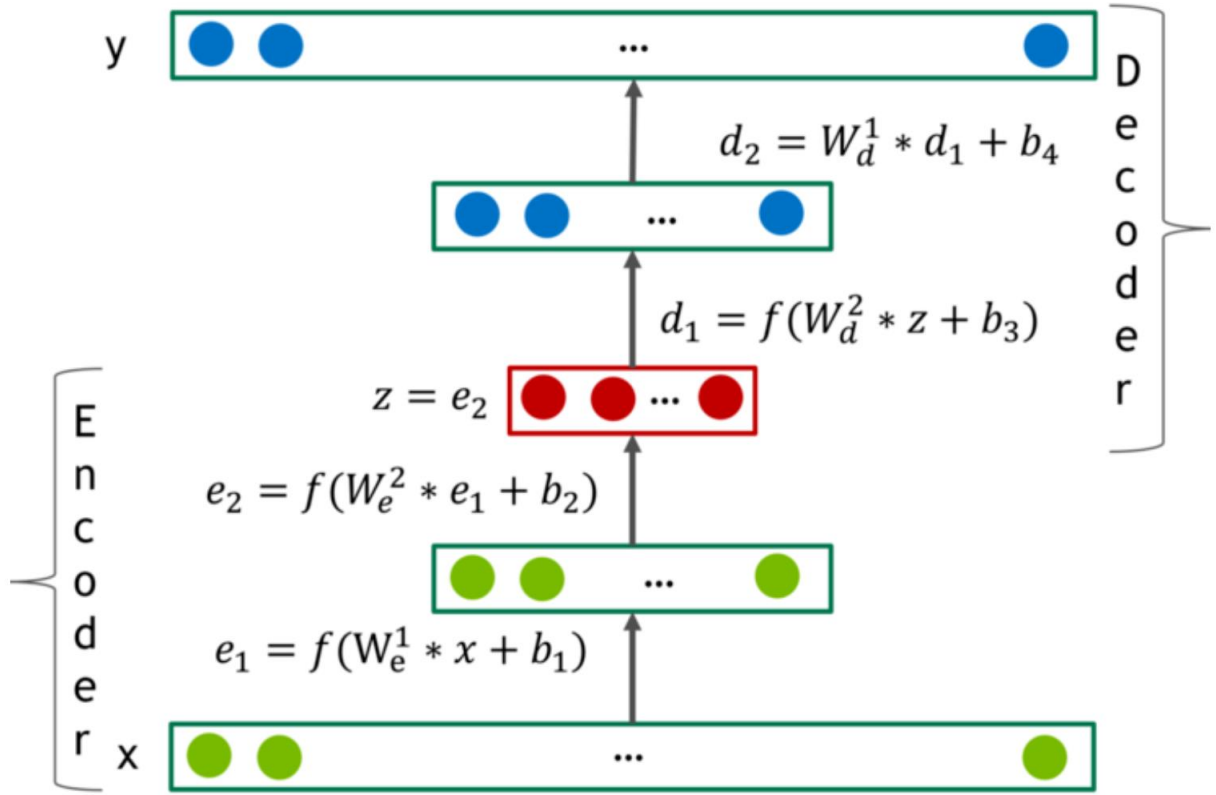
C:\WINDOWS\system32\cmd. x + v
(18, 10)
(10,)
(10, 10)
-----reconstructed user-product rating matrix
[[ 1.  1.  1. -0.  2.  0.  0.  0.  1.  1.]
 [ 2.  0.  1.  1.  4. -0.  2.  1.  3.  1.]
 [ 0.  1. -0.  0. -0.  0. -0. -0. -0.  0.]
 [ 1.  4.  0.  1. -0.  1. -0. -0. -0.  1.]
 [ 0. -0.  0. -0.  1. -0.  0.  0.  0.  0.]
 [ 2.  3.  1. -0.  1.  1. -0.  0. -0.  1.]
 [-0.  0. -0.  4.  1. -0.  3.  0.  4.  0.]
 [ 0.  3. -0.  2. -1.  1.  1. -0.  1.  1.]
 [ 1.  1.  0. -0.  1.  0. -0.  0. -0.  0.]
 [ 1. -1.  1.  1.  4. -0.  1.  1.  3.  1.]
 [ 2.  5.  0.  1. -0.  1. -0. -0. -0.  1.]
 [ 2. -0.  1. -1.  4. -0.  0.  1.  1.  1.]
 [-0.  1. -0.  2. -0.  0.  1. -0.  2.  0.]
 [ 1.  4. -0.  4. -0.  1.  1. -0.  3.  1.]
 [ 2.  0.  1. -1.  4. -0.  0.  1.  1.  1.]
 [-1.  1. -0.  3. -1.  0.  1. -0.  2. -0.]
 [ 1.  2. -0.  1.  0.  1.  0. -0.  1.  1.]
 [ 2.  4.  1. -0.  1.  1. -0.  0. -0.  2.]]
Press any key to continue . . .

```

As explained in the lecture, deep learning based systems usually perform better if the amount of data available is large. One of the popular deep learning approaches for recommender systems is based on AutoEncoders, where we use the ratings provided by a user to train the AutoEncoder by using the masked mean square error as the loss function. For example, if the user ratings by a particular user are:  $[0, 3, 0, 4, 0, 0, 0, 0, 0, 0]$ , then the MMSE loss function will only focus on the second and the fourth output from the autoencoder. Suppose the 10 actual outputs of the autoencoder (for a 10 product system) during the training phase are  $[0.1, \mathbf{0.25}, -0.2, \mathbf{0.7}, 1.2, 0.02, 0.8, 0.15, 0.33]$ . Then the loss in MMSE will only use the second and fourth outputs so for the above example, it will be calculated as:

MMSE loss =  $(3-0.25)^2 + 4-0.7)^2$  All the other outputs will not be used in computing the loss.

The architecture of the deep autoencoder is shown below.



Mean Squared Error loss:

$$MMSE = \frac{m_i * (r_i - y_i)^2}{\sum_{i=0}^n m_i} \quad (1)$$

where  $r_i$  is actual rating,  $y_i$  is reconstructed, or predicted rating, and  $m_i$  is a mask function such that  $m_i = 1$  if  $r_i \neq 0$  else  $m_i = 0$ . Note that there is a straightforward relation between RMSE score and MMSE score:  $RMSE = \sqrt{MMSE}$ .

We will program the above architecture for the Movielens dataset that contains ratings for movies.

This data set consists of:

- \* 100,000 ratings (1-5) from 943 users on 1682 movies.
- \* Each user has rated at least 20 movies.
- \* Simple demographic info for the users (age, gender, occupation, zip)

The data was collected through the MovieLens web site ([movielens.umn.edu](http://movielens.umn.edu)) during the seven-month period from September 19th, 1997 through April 22nd, 1998. This data has been cleaned up – users who had less than 20 ratings or did not have complete demographic information were removed from this

data set. The four csv files needed in the dataset will be provided on the kiwi web site for the data mining course (links.csv, movies.csv, rating.csv, tags.csv).

Create a new Python application called DeepAERecommender. Add a data folder to the project, then by right clicking on the project name, choose add existing item, add the four csv dataset files to the project that you had downloaded and unzipped to a folder on your computer.

Add a file called Utils to the project with the following code in it.

```
import pandas as pd
import numpy as np
import torch
import torchvision
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils
import matplotlib.pyplot as plt
from TrainDataset import TrainDataset
from TestDataset import TestDataset

def prepare_train_validation_movielens_step1():
    rat = pd.read_csv('data/ratings.csv')
    mov = pd.read_csv('data/movies.csv')
    df_combined = pd.merge(rat, mov, on = 'movieId')
    print(rat.describe())
    ts = rat['timestamp'].quantile(0.98)
    train_ratings = pd.DataFrame(columns=['userId', 'movieId', 'rating'])
    validation_ratings = pd.DataFrame(columns=['userId', 'movieId', 'rating'])
    for i in range(len(rat)):
        if rat['timestamp'].iloc[i] <= ts:
            train_ratings =
pd.concat([train_ratings, pd.DataFrame([{'userId': rat['userId'].iloc[i], 'movieId': rat
['movieId'].iloc[i], 'rating': rat['rating'].iloc[i]}])])
            validation_ratings =
pd.concat([validation_ratings, pd.DataFrame([{'userId': rat['userId'].iloc[i], 'movieId
': rat['movieId'].iloc[i], 'rating': rat['rating'].iloc[i]}])])
        else:
            validation_ratings =
pd.concat([validation_ratings, pd.DataFrame([{'userId': rat['userId'].iloc[i], 'movieId
': rat['movieId'].iloc[i], 'rating': rat['rating'].iloc[i]}])])

    if i%10000 == 0:
        print(i, "Completed")
    print(len(train_ratings))
    print(len(validation_ratings))
    # Remove users in validation set those are not present in Training Set
    train_users = train_ratings['userId'].unique()
    users_not_in_train_set = []

    for i in range(1,611):
        if i in train_users:
            continue
        else:
            users_not_in_train_set.append(i)
    for i in users_not_in_train_set:
        validation_ratings = validation_ratings[validation_ratings['userId']!=i]
    validation_ratings.reset_index(drop=True)
```

```

print(len(train_ratings['movieId'].unique()))
print(len(validation_ratings['movieId'].unique()))
# Remove Movies that are not in the Train Set
validation_movies = validation_ratings['movieId'].unique()
train_movies = train_ratings['movieId'].unique()
movies_not_in_train_set = []

for i in validation_movies:
    if i in train_movies:
        continue
    else:
        movies_not_in_train_set.append(i)
for i in movies_not_in_train_set:
    validation_ratings = validation_ratings[validation_ratings['movieId']!=i]
validation_ratings.reset_index(drop=True)
print('Train Users: ', train_ratings['userId'].nunique())
print('Validation Users: ', validation_ratings['userId'].nunique())
print('Train Movies: ', train_ratings['movieId'].nunique())
print('Validation Movies: ', validation_ratings['movieId'].nunique())
train_ratings.to_csv("data/train_ratings.csv")
validation_ratings.to_csv("data/validation_ratings.csv")

def prepare_train_test_movielens_step2():
    tr_ratings = pd.read_csv('data/train_ratings.csv')
    val_ratings = pd.read_csv('data/validation_ratings.csv')
    train_dataset = tr_ratings.pivot_table(index = 'userId', columns = 'movieId',
values = 'rating')
    train_dataset.fillna(0, inplace=True)
    print(train_dataset.head(10))
    test_dataset = val_ratings.pivot_table(index='userId', columns='movieId',
values='rating')
    test_dataset.fillna(0, inplace=True)
    print(test_dataset.head(10))
    train_dataset.to_csv("data/train.csv")
    test_dataset.to_csv('data/test.csv')

def get_train_test_loaders():
    transformations = transforms.Compose([transforms.ToTensor()])
    train_dat = TrainDataset('data/train.csv', transformations)
    test_dat = TestDataset('data/test.csv', transformations)
    train_loader = DataLoader(dataset=train_dat, batch_size = 128, shuffle=True,
num_workers = 1)
    test_loader = DataLoader(dataset=test_dat, batch_size=128, shuffle=True,
num_workers=1)
    return train_loader, test_loader

```

The above set of functions prepare the datasets and the train test dataloaders so that we can training our deep autoencoder. Add a class called TrainDataset.py with the following code in it.

```

import torch
from torch.utils.data import Dataset
import pandas as pd
import numpy as np

class TrainDataset(Dataset):
    def __init__(self, train_file, transform=None):

```

```

self.data = pd.read_csv(train_file)
self.data = self.data.iloc[:,1:]
self.transform = transform

if transform is not None:
    self.data = self.transform(np.array(self.data))

def __len__(self):
    return len(self.data[0])

def __getitem__(self, ind):
    user_vector = self.data.data[0][ind]

    return user_vector

```

Add a class called TestDataset with the following code in it.

```

import torch
from torch.utils.data import Dataset
import pandas as pd
import numpy as np

class TestDataset(Dataset):
    def __init__(self, test_file, transform=None):
        self.data = pd.read_csv(test_file)
        self.data = self.data.iloc[:,1:]
        self.transform = transform

        if transform is not None:
            self.data = self.transform(np.array(self.data))

    def __len__(self):
        return len(self.data[0])

    def __getitem__(self, ind):
        user_vector = self.data.data[0][ind]

        return user_vector

```

Add a class called MSELoss\_with\_Mask with the following code in it.

```

import torch
import torch.nn as nn
from torch.autograd import Variable

class MSELoss_with_Mask(nn.Module):
    def __init__(self):
        super(MSELoss_with_Mask, self).__init__()

    def forward(self, inputs, targets):
        # Masking into a vector of 1's and 0's.
        mask = (targets!=0)

```

```

mask = mask.float()

# Actual number of ratings.
# Take max to avoid division by zero while calculating loss.
other = torch.Tensor([1.0])
other = other.cuda()
number_ratings = torch.max(torch.sum(mask), other)
error = torch.sum(torch.mul(mask, torch.mul((targets-inputs), (targets-inputs))))
loss = error.div(number_ratings)
return loss[0]

```

Add a class called AEModel.py with the following code in it

```

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init
from MSELoss_with_Mask import MSELoss_with_Mask

def activation(input, type):

    if type.lower()=='selu':
        return F.selu(input)
    elif type.lower()=='elu':
        return F.elu(input)
    elif type.lower()=='relu':
        return F.relu(input)
    elif type.lower()=='relu6':
        return F.relu6(input)
    elif type.lower()=='lrelu':
        return F.leaky_relu(input)
    elif type.lower()=='tanh':
        return F.tanh(input)
    elif type.lower()=='sigmoid':
        return F.sigmoid(input)
    elif type.lower()=='swish':
        return F.sigmoid(input)*input
    elif type.lower()=='identity':
        return input
    else:
        raise ValueError("Unknown non-Linearity Type")

class AutoEncoder(nn.Module):
    def __init__(self, layer_sizes, nl_type='selu', is_constrained=True,
dp_drop_prob=0.0, last_layer_activations=True):
        """
        layer_sizes = size of each layer in the autoencoder model
        For example: [10000, 1024, 512] will result in:
            - encoder 2 layers: 10000x1024 and 1024x512. Representation layer (z)
will be 512
            - decoder 2 layers: 512x1024 and 1024x10000.

        nl_type = non-Linearity type (default: 'selu').
        is_constrained = If true then the weights of encoder and decoder are tied.
        dp_drop_prob = Dropout probability.
        last_layer_activations = Whether to apply activation on last decoder layer.

```

```

"""

super(AutoEncoder, self).__init__()

self.layer_sizes = layer_sizes
self.nl_type = nl_type
self.is_constrained = is_constrained
self.dp_drop_prob = dp_drop_prob
self.last_layer_activations = last_layer_activations

if dp_drop_prob>0:
    self.drop = nn.Dropout(dp_drop_prob)

self._last = len(layer_sizes) - 2

# Inititalize Weights
self.encoder_weights = nn.ParameterList(
[nn.Parameter(torch.rand(layer_sizes[i+1], layer_sizes[i])) for i in
range(len(layer_sizes) - 1) ] )

    # "Xavier Initialization" ( Understanding the Difficulty in training deep
feed forward neural networks - by Glorot, X. & Bengio, Y. )
    # ( Values are sampled from uniform distribution )
    for weights in self.encoder_weights:
        init.xavier_uniform_(weights)

# Encoder Bias
self.encoder_bias = nn.ParameterList(
[nn.Parameter(torch.zeros(layer_sizes[i+1])) for i in range(len(layer_sizes) - 1) ]
)

reverse_layer_sizes = list(reversed(layer_sizes))
# reversed returns iterator

# Decoder Weights
if is_constrained == False:
    self.decoder_weights = nn.ParameterList(
[nn.Parameter(torch.rand(reverse_layer_sizes[i+1], reverse_layer_sizes[i])) for i in
range(len(reverse_layer_sizes) - 1) ] )

    for weights in self.decoder_weights:
        init.xavier_uniform_(weights)

    self.decoder_bias = nn.ParameterList(
[nn.Parameter(torch.zeros(reverse_layer_sizes[i+1])) for i in
range(len(reverse_layer_sizes) - 1) ] )

def encode(self,x):
    for i,w in enumerate(self.encoder_weights):
        x = F.linear(input=x, weight = w, bias = self.encoder_bias[i] )
        x = activation(input=x, type=self.nl_type)

# Apply Dropout on the last layer
if self.dp_drop_prob > 0:
    x = self.drop(x)

```



```

        return x

    def decode(self,x):
        if self.is_constrained == True:
            # Weights are tied
            for i,w in
zip(range(len(self.encoder_weights)),list(reversed(self.encoder_weights))):
                x = F.linear(input=x, weight=w.t(), bias = self.decoder_bias[i] )
                x = activation(input=x, type=self.nl_type if i != self._last or
self.last_layer_activations else 'identity')

            else:

                for i,w in enumerate(self.decoder_weights):
                    x = F.linear(input=x, weight = w, bias = self.decoder_weights[i])
                    x = activation(input=x, type=self.nl_type if i != self._last or
self.last_layer_activations else 'identity')

        return x

    def forward(self,x):
        # Forward Pass
        return self.decode(self.encode(x))

```

Type the following code in DeepAERecommender.py file.

```

import sys
import Utils
import numpy as np
from AEModel import AutoEncoder
from MSELoss_with_Mask import MSELoss_with_Mask
import torch.optim as optim

def train(model, criterion, optimizer, train_dl, test_dl, num_epochs=50):
    for epoch in range(num_epochs):
        train_loss, valid_loss = [], []
        logs = {}
        prefix = ''

        # Training Part
        model.train()
        for i, data in enumerate(train_dl, 0):
            inputs = labels = data
            inputs = inputs.cuda()
            labels = labels.cuda()

            inputs = inputs.float()
            labels = labels.float()

            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = model(inputs)
            outputs = outputs.cuda()

```

```

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Zero the gradients
        optimizer.zero_grad()

        outputs = model(outputs.detach())
        outputs = outputs.cuda()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss.append(loss.item())
        print('MMSE loss ', loss.item())

    for i, data in enumerate(test_dl, 0):
        model.eval()
        inputs = labels = data
        inputs = inputs.cuda()
        labels = labels.cuda()

        inputs = inputs.float()
        labels = labels.float()

        outputs = model(inputs)
        outputs = outputs.cuda()
        loss = criterion(outputs, labels)

        valid_loss.append(loss.item())
        prefix = 'val_'
        print('MMSE loss', loss.item())

    print("-----output-----")
    print(outputs[0])
    print()
    print("Epoch:", epoch+1, " Training Loss: ", np.mean(train_loss), " Valid Loss: ", np.mean(valid_loss))

def main():
    #Utils.prepare_train_validation_movielenstep1()
    #Utils.prepare_trainvalidation_movielenstep2()
    train_loader, test_loader = Utils.get_trainvalidationloaders()
    layer_sizes = [9559, 512, 512, 1024]
    model = AutoEncoder(layer_sizes=layer_sizes, nl_type='relu',
is_constrained=True, dp_drop_prob=0.0, last_layer_activations=False)
    model = model.cuda()
    criterion = MSELoss_with_Mask()
    criterion = criterion.cuda()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    out = train(model, criterion, optimizer, train_loader, test_loader, 40)

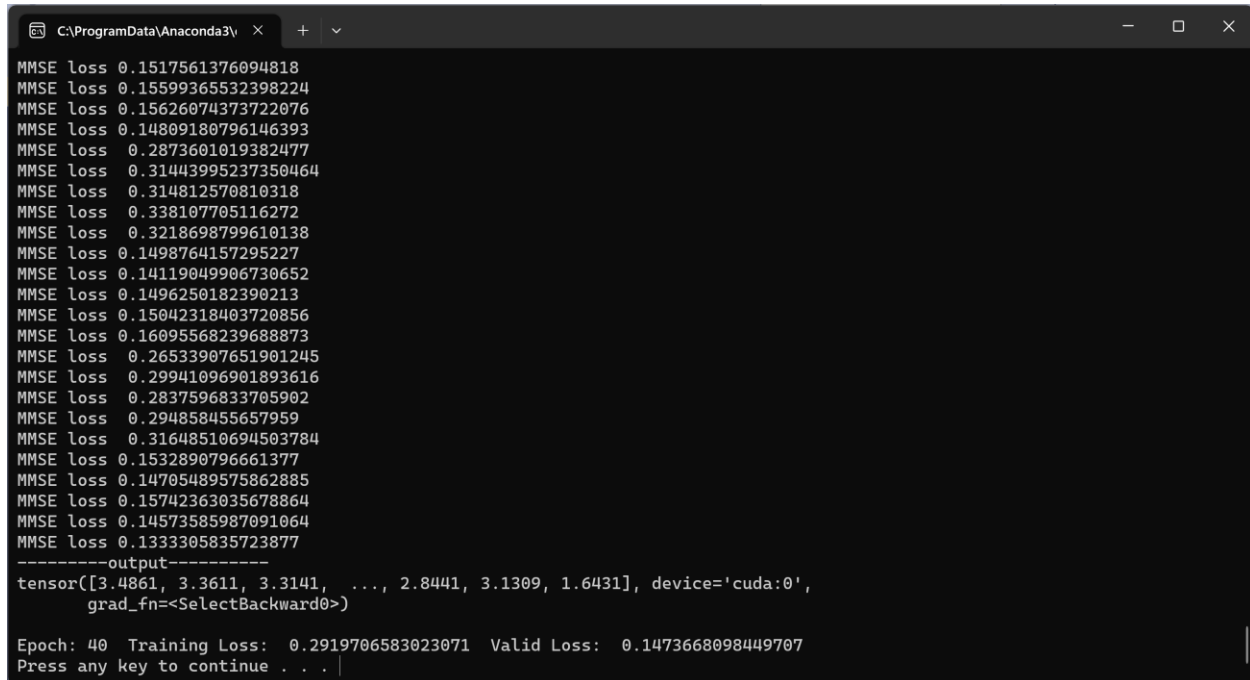
if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

First time you run the above main program, make sure to uncomment the following two lines:

```
#Utils.prepare_train_validation_movielens_step1()
#Utils.prepare_traintest_movielens_step2()
```

After you have run the program once, comment the above lines as the intermediate data needed has already been prepared by the above two functions and stored in additional csv files in the data folder. If you run the program, it will train and you can see the recommended values for the first user



```
C:\ProgramData\Anaconda3\
MMSE loss 0.1517561376094818
MMSE loss 0.15599365532398224
MMSE loss 0.15626074373722076
MMSE loss 0.14809180796146393
MMSE loss 0.2873601019382477
MMSE loss 0.31443995237350464
MMSE loss 0.314812570810318
MMSE loss 0.338107705116272
MMSE loss 0.3218698799610138
MMSE loss 0.1498764157295227
MMSE loss 0.14119049906730652
MMSE loss 0.1496250182390213
MMSE loss 0.15042318403720856
MMSE loss 0.16095568239688873
MMSE loss 0.26533907651901245
MMSE loss 0.29941096901893616
MMSE loss 0.2837596833705902
MMSE loss 0.294858455657959
MMSE loss 0.31648510694503784
MMSE loss 0.1532890796661377
MMSE loss 0.14705489575862885
MMSE loss 0.15742363035678864
MMSE loss 0.14573585987091064
MMSE loss 0.1333305835723877
-----output-----
tensor([3.4861, 3.3611, 3.3141, ..., 2.8441, 3.1309, 1.6431], device='cuda:0',
grad_fn=<SelectBackward0>)

Epoch: 40 Training Loss: 0.2919706583023071 Valid Loss: 0.1473668098449707
Press any key to continue . . . |
```

After the Autoencoder is trained, to determine the recommendation for an incoming user, we will feed the existing ratings for that user to the autoencoder, the outputs from the autoencoder will be used to determine the recommendation, i.e., the position of the highest rating in the output from the autoencoder (not including the positions where the input to the autoencoder already had a rating).