Anime Recommendation System

This Project contains:-

1. Making recommendation system using correlation of one anime with others.

* Import the libraries required
* count the number of users in rating_df
* find the average rating
* Remove the duplicate rows
* Make a data frame and use crosstab and grouping it with rating
* Read the anime_df , rating_df file
* Describe the anime_df statistics
* Count the total number of ratings given for each rating points(0-10) in rating_df and -1 if r
* Remove the negative rating values
* Plot the missing values in anime_df
* Plot the missing values in rating_df
* merge both raing_df and anime_df on anime_id as key value
* Count the total ratings given to each anime
* Remove the ratings with <r total ratings given from above
* Finding the number of rows we removed during the preprocessing task
* Create a pivot table with user_id and name as index and column and rating as value
* Count total number of ratings given to each anime
* Save the dataframe as zip file using pickle
* Read the pickle file that we saved as df.zip
* Create a function to recommend the anime name based on correlation
* Try the function we created to recommend some anime

2. Making recommendation system using Neural Networks with ratings as predicting variable and user, anime as features

* Data Preprocessing
* Choose user, anime as X feature to predict rating as y
* Split the data into train and test and print the length of both datasets
* Model Building
* Import the required tensorflow libraries
* Create the model using embedding function provided by keras
* Make a LR Scheduler
* Train the model with epochs=1 (increase the number of epochs to increase the training accurac
* Plot the loss with each epochs for train and test dataset
* Extracting weights from model

```
    * Read the anime.csv file
    * Create a function to get anime name based on anime id
    * Sypnopsis Data
    * Create a function to recommend similar anime
    * Try the model that we created to recommend similar anime
    * Save the RecommenderNet() model as .h5 file
    * Save the RecommenderNet() model as .pkl file
```

1. Making recommendation system using correlation of one anime with others.

Import the libraries required

```
# import numpy
import numpy as np
# import pandas
import pandas as pd
# read the animelist csv file
rating_df = pd.read_csv('/content/animelist.csv', usecols=["user_id", "anime_id", "rating"
# print top 5 rows
rating_df.head(4)
```

count the number of users in rating_df

```
# User should rate atleast 400 animies
# check userid value counts in rating df
n_ratings = rating_df['user_id'].value_counts()

# remove the ratings if number of rating given<400 by extracting n_rating>=400
rating_df = rating_df[rating_df['user_id'].isin(n_ratings[n_ratings >= 400].index)].copy()

# print length of user id
len(rating_df)
```

```
2750798
```

find the average rating

```
# Scaling BTW (0 , 1.0)
# find min ratings
min_rating = min(rating_df['rating'])

# find max ratings
max_rating = max(rating_df['rating'])

# normalize the ratings in range (0,1)
rating_df['rating'] = rating_df["rating"].apply(lambda x: (x - min_rating) / (max_rating -

# find average ratings
AvgRating = np.mean(rating_df['rating'])

# print average ratings
print('Avg', AvgRating)
```

```
Avg 0.4140259299303642
```

Remove the duplicate rows

```
# Removing Duplicated Rows
duplicates = rating_df.duplicated()

# remove duplicate rows
if duplicates.sum() > 0:
    print('> {} duplicates'.format(duplicates.sum()))
    rating_df = rating_df[~duplicates]

# printing total duplicates
print('> {} duplicates'.format(rating_df.duplicated().sum()))
```

```
> 0 duplicates
```

Make a data frame and use crosstab and grouping it with rating

```
# Group user id with rating as key
g = rating_df.groupby('user_id')['rating'].count()

# drop bottom 20 rows from g
top_users = g.dropna().sort_values(ascending=False)[:20]

# join the rating_df with top_users on user id as key(inner join)
top_r = rating_df.join(top_users, rsuffix='_r', how='inner', on='user_id')
```

```
# Group anime id with rating as key
g = rating_df.groupby('anime_id')['rating'].count()

# drop bottom 20 rows from g
top_animes = g.dropna().sort_values(ascending=False)[:20]

# join the top_r with top_anime on anime id as key(inner join)
top_r = top_r.join(top_animes, rsuffix='_r', how='inner', on='anime_id')

# display the crosstab as user_id, anme_id and rating as value
pd.crosstab(top_r.user_id, top_r.anime_id, top_r.rating, aggfunc=np.sum)
```

Read the anime_df , rating_df file

```
# Read the anime_dataset.csv file
anime_df = pd.read_csv('/content/anime_dataset.csv')

# Read the rating_dataset.csv file
rating_df = pd.read_csv('/content/rating_dataset.csv')
```

```
# Top 5 rows in anime_df dataframe
anime_df.head()
```

```
# Top 5 rows in rating_df dataframe
rating_df.head()
```

Describe the anime_df statistics

```
# Describe the anime_df
anime_df.describe()
```

Describe the rating_df statistics

```
# Describe the rating_df
rating_df.describe()
```

Count the total number of ratings given for each rating points(0-10) in rating_df and -1 if not rated

```
# Lets have a look the distribution of ratings, because those "-1" are suspicious
rating_df.rating.value_counts()
```

```
 8     1646019
-1     1476496
 7     1375287
 9     1254096
 10     955715
 6      637775
 5      282806
 4      104291
 3       41453
 2       23150
 1       16649
Name: rating, dtype: int64
```

Remove the negative rating values

```
# remove the negative rating value
rating_df = rating_df[rating_df["rating"] != -1]
```

Plot the missing values in anime_df

```
# import seaborn
import seaborn as sns

# import matplotlib
import matplotlib.pyplot as plt

#plot the figure of size(8,6)
plt.figure(figsize=(8,6))

# plot the heatmap using seaborn
sns.heatmap(anime_df.isnull())

# plot the title of graph
plt.title("Missing values in anime ?", fontsize = 15)

# plot the graph
plt.show()
```

Plot the missing values in missing_df

```
# plot the figure of size(8,6)
plt.figure(figsize=(8,6))

# plot the heatmap using seaborn
```

```
sns.heatmap(rating_df.isnull())

# plot the title of graph
plt.title("Missing values in rating?", fontsize = 15)

# plot the graph
plt.show()
```

merge both raing_df and anime_df on anime_id as key value

```
# Merge anime and rating using "anime_id" as reference
# Keep only the columns we will use
cos_df = pd.merge(rating_df,anime_df[["anime_id","name"]], left_on = "anime_id",
                  right_on = "anime_id").drop("anime_id", axis = 1)

# display top 5 rows
cos_df.head()
```

Count the total ratings given to each anime

```
# Count the number of ratings for each anime(groupby name)
count_rating = cos_df.groupby("name")["rating"].count().sort_values(ascending = False)

# print count_rating
count_rating
```

```
name
Death Note                          34226
Sword Art Online                    26310
Shingeki no Kyojin                  25290
Code Geass: Hangyaku no Lelouch     24126
Angel Beats!                        23565
                                    ...
La Primavera                            1
Chou Zenmairobo: Patrasche              1
Ushi Atama                              1
Gun-dou Musashi Recap                   1
Futago no Ookami Daibouken              1
Name: rating, Length: 9926, dtype: int64
```

Remove the ratings with <r total ratings given from above

```
# Some animes have only 1 rating, therefore it is better for the recommender system to ign
# We will keep only the animes with at least r ratings
r = 5000

# extract indexes by the use of apply method on count_rating and keep only rating >= r
more_than_r_ratings = count_rating[count_rating.apply(lambda x: x >= r)].index

# Keep only the animes with at least r ratings which are in more_than_r_ratings in the Dat
df_r = cos_df[cos_df['name'].apply(lambda x: x in more_than_r_ratings)]
```

Finding the number of rows we removed during the preprocessing task

```
# length of unique anime before preprocessing in cos_df
before = len(cos_df.name.unique())

# length of unique anime after preprocessing in df_r
after = len(df_r.name.unique())

# length of rows before preprocessing
rows_before = cos_df.shape[0]

# length of rows before preprocessing
rows_after = df_r.shape[0]

# print the above informations
print(f'''There are {before} animes in the dataset before filtering and {after} animes aft
{before} animes => {after} animes
{rows_before} rows before filtering => {rows_after} rows after filtering''')
```

```
There are 9926 animes in the dataset before filtering and 279 animes after the filter
9926 animes => 279 animes
6337239 rows before filtering => 2517097 rows after filtering
```

Create a pivot table with user_id and name as index and column and rating as value

```
# Create a matrix with userId as rows and the titles of the movies as column.
# Each cell will have the rating given by the user to the animes.
# There will be a lot of NaN values, because each user hasn't watched most of the animes
df_recom = df_r.pivot_table(index='user_id',columns='name',values='rating')

# print first 5 row, first 5 col
df_recom.iloc[:5,:5]
```

Count total number of ratings given to each anime

```
# each anime counts of rating
df_r.name.value_counts().head(10)
```

```
Death Note                          34226
Sword Art Online                    26310
Shingeki no Kyojin                  25290
Code Geass: Hangyaku no Lelouch     24126
Angel Beats!                        23565
Elfen Lied                          23528
Naruto                              22071
Fullmetal Alchemist: Brotherhood    21494
Fullmetal Alchemist                 21332
Code Geass: Hangyaku no Lelouch R2  21124
Name: name, dtype: int64
```

Save the dataframe as zip file using pickle

```
# import pickle
```

```
import pickle

# pickle the recommendor dataframe using to_pickle as "df.zip"
df_recom.to_pickle("df.zip", compression='zip')
```

Read the pickle file that we saved as df.zip

```
# read 'df.zip'
new_df= pd.read_pickle("df.zip")

# print head of new_df
new_df.head()
```

Create a function to recommend the anime name based on correlation

```
# Find corr function

def find_corr(df, name):
    '''
    Get the correlation of one anime with the others

    Args
```

```
        df (DataFrame):  with user_id as rows and movie titles as column and ratings as va
        name (str): Name of the anime

    Return
        DataFrame with the correlation of the anime with all others
    '''
    # apply corrwith on df and pass df[name]
    similar_to_movie = df.corrwith(df[name])

    # create dataframe using similar_to_movie with columns as 'Correlation'
    similar_to_movie = pd.DataFrame(similar_to_movie,columns=['Correlation'])

    # sort by values by correlation in ascending order
    similar_to_movie = similar_to_movie.sort_values(by = 'Correlation', ascending = False)

    # return similar_to_movie
    return similar_to_movie
```

Try the function we created to recommend some anime

```
# Let's try with 'Death Note' recommendations
find_corr(df_recom, 'Death Note').head(5)
```

```
# Let's try with 'Cowboy Bebop' recommendations
find_corr(df_recom, 'Cowboy Bebop').head(5)
```

```
# Let's choose an anime for recommendations
find_corr(df_recom, 'Angel Beats!').head(5)
```

Method-2 Using Cosine Similarity

```
# import numpy as np
import numpy as np

# import pandas as pd
import pandas as pd

# Read the anime_with_synopsis_reduced.csv file
anime_with_sypnopsis = pd.read_csv('anime_with_synopsis_reduced.csv')


# Top 5 rows in anime_df dataframe
anime_with_sypnopsis.head()
```

```
# extract Name, Genres and Synopsis from anime_with_sypnopsis
anime_with_sypnopsis = anime_with_sypnopsis[['Name' , 'Genres' , 'sypnopsis']]
```

```
# look at first 5 rows
anime_with_sypnopsis.head()
```

```
# create a column merge by adding synopsis and genres
anime_with_sypnopsis['merge'] = anime_with_sypnopsis['sypnopsis'] + anime_with_sypnopsis['

# look at first 5 rows
anime_with_sypnopsis.head(5)
```

```
# drop genres and synopsis from anime_with_sypnopsis
tag_df = anime_with_sypnopsis.drop(columns=['Genres','sypnopsis'])

# look at first 5 rows
tag_df.head()
```

```
# check shape of tag_df
tag_df.shape
```

```
(7998, 2)
```

```
# import ast, sklearn, nltk
import ast
import sklearn
import nltk

# import RegexpTokenizer from nltk.tokenize
from nltk.tokenize import RegexpTokenizer

# import WordNetLemmatizer, PorterStemmer from nltk.stem
from nltk.stem import WordNetLemmatizer,PorterStemmer

# import stopwords from nltk.corpus
from nltk.corpus import stopwords

# import re
import re

# instantiate WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

# instantiate PorterStemmer
stemmer = PorterStemmer()


# define preprocess function
def preprocess(sentence):
  # convert sentence into string
  sentence=str(sentence)

  # convert sentence into lowercase
  sentence = sentence.lower()

  # replace {html} into ""
  sentence=sentence.replace('{html}',"")

  # compile '<.*?>'
  cleanr = re.compile('<.*?>')

  # apply re.sub on sentence and convert cleanr to ''
  cleantext = re.sub(cleanr, '', sentence)

  # apply re.sub on cleantext and convert 'http\S+' to ''
  rem_url=re.sub(r'http\S+', '',cleantext)

  # apply re.sub on rem_url and convert '[0-9]+' to ''
  rem_num = re.sub('[0-9]+', '', rem_url)

  # Instantiate Regextokenizer and pass 'r\w+'
  tokenizer = RegexpTokenizer(r'\w+')
```

```
    # tokenize the  rem_num
    tokens = tokenizer.tokenize(rem_num)

    filtered_words = [w for w in tokens if len(w) > 2 if not w in stopwords.words('english')

    # stemmatize the word in filtered words using list comprehension
    stem_words=[stemmer.stem(w) for w in filtered_words]

    # lemmatize the word in stem words using list comprehension
    lemma_words=[lemmatizer.lemmatize(w) for w in stem_words]

    # return filtered words after applying join
    return " ".join(filtered_words)


# just run this to avoid any error related to it
import nltk
nltk.download('stopwords')
```

```
    [nltk_data] Downloading package stopwords to /root/nltk_data...
    [nltk_data]   Unzipping corpora/stopwords.zip.
    True
```

```
# just run this to avoid any error related to it
nltk.download('wordnet')
```

```
    [nltk_data] Downloading package wordnet to /root/nltk_data...
    [nltk_data]   Unzipping corpora/wordnet.zip.
    True
```

```
# preprocess each sentence in merge column
tag_df['merge']= tag_df['merge'].map(lambda s:preprocess(s))


# check first 5 row of tag_df
tag_df.head()
```

```
# import CountVectorizer from sklearn.feature_extraction.text
from sklearn.feature_extraction.text import CountVectorizer

# instantiate CountVectorizer with max_features=1000 and stop_words='english'
cv = CountVectorizer(max_features=1000,stop_words='english')
```

```
# call fit_transform on cv and pass tag_df['merge'] and apply toarray on it
vector = cv.fit_transform(tag_df['merge']).toarray()
```

```
# check shape of vector
vector.shape
```

```
(7998, 1000)
```

```
# import cosine_similarity on sklearn.metrics.pairwise
from sklearn.metrics.pairwise import cosine_similarity

# instantiate cosine_similarity and pass vector
similarity = cosine_similarity(vector)

# print similarity
similarity
```

```
array([[1.        , 0.28800922, 0.19258222, ..., 0.04724556, 0.13568134,
        0.06590622],
       [0.28800922, 1.        , 0.20131906, ..., 0.06350006, 0.        ,
        0.        ],
       [0.19258222, 0.20131906, 1.        , ..., 0.05661385, 0.04877565,
        0.10529963],
       ...,
       [0.04724556, 0.06350006, 0.05661385, ..., 1.        , 0.17948954,
        0.        ],
       [0.13568134, 0.        , 0.04877565, ..., 0.17948954, 1.        ,
        0.05007656],
       [0.06590622, 0.        , 0.10529963, ..., 0.        , 0.05007656,
        1.        ]])
```

```
# check the first index of the anime where name is Bleach
tag_df[tag_df['Name']=='Bleach'].index[0]
```

```
237
```

```
# define recommend function
def recommend(movie):

    # first index of the anime where name is movie
    index = tag_df[tag_df['Name'] == movie].index[0]

    # apply sorted on the list of enumerate(similarity[index]) with reverse=True and key =
    distances = sorted(list(enumerate(similarity[index])),reverse=True,key = lambda x: x[1

    for i in distances[1:6]:
        print(tag_df.iloc[i[0]].Name)
```

```
# check recommend on 'One Piece'
recommend('One Piece')
```

```
One Piece Film: Z
```

```
      One Piece Film: Strong World Episode 0
      One Piece Film: Strong World
      Magi: Sinbad no Bouken
      One Piece: Romance Dawn Story
```

```
# check recommend on 'Bleach'
recommend('Bleach')
```

```
      Bleach Movie 1: Memories of Nobody
      Bleach Movie 3: Fade to Black - Kimi no Na wo Yobu
      Atashin'chi 3D Movie: Jounetsu no Chou Chounouryoku Haha Dai Bousou
      Bleach: The Sealed Sword Frenzy
      Towa no Quon 1: Utakata no Kaben
```

```
# dumping the file
import pickle

# dump tag_df as 'final_movie_list.pkl' in 'wb' mode
pickle.dump(tag_df,open('final_movie_list.pkl','wb'))

# dump similarity as 'final_similarity.pkl' in 'wb' mode
pickle.dump(similarity,open('final_similarity.pkl','wb'))
```

Method-3 Using Deep learning Models

Data Preprocessing

```
# Encoding categorical data

# Print number of unique users and convert to list
user_ids = rating_df["user_id"].unique().tolist()

# map user id
user2user_encoded = {x: i for i, x in enumerate(user_ids)}
user_encoded2user = {i: x for i, x in enumerate(user_ids)}

# create a col user by mapping user_id using user2user_encoded
rating_df["user"] = rating_df["user_id"].map(user2user_encoded)

# find number of unique users in user2user_encoded
n_users = len(user2user_encoded)

# Print number of unique anime_id and convert it into list
anime_ids = rating_df["anime_id"].unique().tolist()

# map anime id
anime2anime_encoded = {x: i for i, x in enumerate(anime_ids)}
anime_encoded2anime = {i: x for i, x in enumerate(anime_ids)}
```

```
# create a col anime by mapping anime_id using anime2anime_encoded
rating_df["anime"] = rating_df["anime_id"].map(anime2anime_encoded)

# find number of unique animes in anime2anime_encoded
n_animes = len(anime2anime_encoded)

# print number of users and number of anime
print("Num of users: {}, Num of animes: {}".format(n_users, n_animes))

# print minimum rating and maximum rating
print("Min rating: {}, Max rating: {}".format(min(rating_df['rating']), max(rating_df['rat
```

```
Num of users: 69600, Num of animes: 9927
Min rating: 1, Max rating: 10
```

Choose user, anime as X feature to predict rating as y

```
# Shuffle the rating dataframe with frac=1, random_state=7
rating_df = rating_df.sample(frac=1, random_state=73)

# Choose X with anime, user as feature
X = rating_df[['user', 'anime']].values

# Choose y with rating as feature
y = rating_df["rating"]
```

Split the data into train and test and print the length of both datasets

```
# Split the dataset as choose test set size
test_set_size = 10000 #10k for test set
train_indices = rating_df.shape[0] - test_set_size

# split into train, test datasets
X_train, X_test, y_train, y_test = (
    X[:train_indices],
    X[train_indices:],
    y[:train_indices],
    y[train_indices:],
)

# print y_train set length
print('> Train set ratings: {}'.format(len(y_train)))
# print y_test set length
print('> Test set ratings: {}'.format(len(y_test)))
```

```
> Train set ratings: 6327241
> Test set ratings: 10000
```

```
# extract first and 2nd col of all rows in X_train in the list
X_train_array = [X_train[:, 0], X_train[:, 1]]

# extract first and 2nd col of all rows in X_test in the list
X_test_array = [X_test[:, 0], X_test[:, 1]]
```

Make TPU_INIT=true if you have GPU and Make TPU_INIT=false if you don't have GPU

```
# Accelerator check
import tensorflow as tf
# TPU_INIT = False if not using TPu
TPU_INIT=False
# Print nvidia-smi version used
if TPU_INIT:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver.connect()
    tpu_strategy = tf.distribute.experimental.TPUStrategy(tpu)
else:
    !nvidia-smi
# print tensorflow version
print(tf.__version__)
```

```
NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver. Make su

2.8.0
```

Model Building

Import the required tensorflow libraries

```
# import keras
import keras

# import layers from keras
from keras import layers

# import tensorflow as tf
import tensorflow as tf

# import Model from keras.models
from keras.models import Model

# import Adam optimizers from tensorflow
from tensorflow.keras.optimizers import Adam
```

Create the model using embedding function provided by keras

```
# Embedding layers
# import Add, Activation, Lambda, BatchNormalization,
# Concatenate, Dropout, Input, Embedding, Dot, Reshape, Dense, Flatten

from keras.layers import Add, Activation, Lambda, BatchNormalization, Concatenate, Dropout

# create the model as RecommenderNet()
def RecommenderNet():

    # Use embedding size as 128
    embedding_size = 128

# input_dim: Integer. Size of the vocabulary, i.e. maximum integer index + 1.
# output_dim: Integer. Dimension of the dense embedding.
# embeddings_initializer: Initializer for the embeddings matrix (see keras.initializers).
# embeddings_regularizer: Regularizer function applied to the embeddings matrix (see keras
# embeddings_constraint: Constraint function applied to the embeddings matrix (see keras.c
# mask_zero: Boolean, whether or not the input value 0 is a special "padding" value that s
#This is useful when using recurrent layers which may take variable length input.
#If this is True, then all subsequent layers in the model need to support masking or an ex
#If mask_zero is set to True, as a consequence, index 0 cannot be used in the vocabulary
#(input_dim should equal size of vocabulary + 1).
# input_length: Length of input sequences, when it is constant.
#This argument is required if you are going to connect Flatten then Dense layers upstream
    user = Input(name = 'user', shape = [1])

    # instantiate Embedding with input dim as n_users and output dim as embedding_size
    user_embedding = Embedding(name = 'user_embedding',
                       input_dim = n_users,
                       output_dim = embedding_size)(user)

    anime = Input(name = 'anime', shape = [1])

    # instantiate Embedding with input dim as n_animes and output dim as embedding_size
    anime_embedding = Embedding(name = 'anime_embedding',
                       input_dim = n_animes,
                       output_dim = embedding_size)(anime)

    #x = Concatenate()([user_embedding, anime_embedding]) using Dot(keep normalize=True an
    x = Dot(name = 'dot_product', normalize = True, axes = 2)([user_embedding, anime_embed

    # Use flatten layer to
    x = Flatten()(x)

    # Add a dense layer  with   kernel_initializer='he_normal' and 1 Neuron
    x = Dense(1, kernel_initializer='he_normal')(x)

    # add a Batch Normalization layer
    x = BatchNormalization()(x)
```

```
    # add a sigmoid activation layer
    x = Activation("sigmoid")(x)

    model = Model(inputs=[user, anime], outputs=x)
    # compile the model based on mean absolute error and mean square error
    model.compile(loss='binary_crossentropy', metrics=["mae", "mse"], optimizer='Adam')

    # return model as output
    return model

# Check if TPU_INIT=true then run RecommenderNet() as model

if TPU_INIT:
    with tpu_strategy.scope():
        model = RecommenderNet()
else:
    model = RecommenderNet()

# Print the summary of model
model.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 user (InputLayer)              [(None, 1)]          0           []

 anime (InputLayer)             [(None, 1)]          0           []

 user_embedding (Embedding)     (None, 1, 128)       8908800     ['user[0][0]']

 anime_embedding (Embedding)    (None, 1, 128)       1270656     ['anime[0][0]']

 dot_product (Dot)              (None, 1, 1)         0           ['user_embedding[0][
                                                                  'anime_embedding[0]

 flatten_1 (Flatten)            (None, 1)            0           ['dot_product[0][0]

 dense_1 (Dense)                (None, 1)            2           ['flatten_1[0][0]']

 batch_normalization_1 (BatchNo  (None, 1)           4           ['dense_1[0][0]']
 rmalization)

 activation_1 (Activation)      (None, 1)            0           ['batch_normalizatic

==================================================================================================
Total params: 10,179,462
Trainable params: 10,179,460
Non-trainable params: 2
__________________________________________________________________________________________________
```

Save the RecommenderNet() model as .h5 file

```
# save the model
```

```
model.save('./ver3_anime_model.h5')
# import FileLink library
from IPython.display import FileLink
FileLink(r'./ver3_anime_model.h5')
```

Make a LR Scheduler

```
# import Callback, ModelCheckpoint, LearningRateScheduler, TensorBoard, EarlyStopping, Red
from tensorflow.keras.callbacks import Callback, ModelCheckpoint, LearningRateScheduler, T

# Pick hyperparameters for LR Scheduling
start_lr = 0.00001
min_lr = 0.00001
max_lr = 0.00005
batch_size = 10000

# Change batch size if you are using TPUs
if TPU_INIT:
    max_lr = max_lr * tpu_strategy.num_replicas_in_sync
    batch_size = batch_size * tpu_strategy.num_replicas_in_sync

rampup_epochs = 5
sustain_epochs = 0
exp_decay = .8

# Create a function to perform LR Scheduling and change rate
def lrfn(epoch):
    if epoch < rampup_epochs:
        return (max_lr - start_lr)/rampup_epochs * epoch + start_lr
    elif epoch < rampup_epochs + sustain_epochs:
        return max_lr
    else:
        return (max_lr - min_lr) * exp_decay**(epoch-rampup_epochs-sustain_epochs) + min_l

# Used LR Scheduling with (lambda epoch: call lnfrn function)
lr_callback = LearningRateScheduler(lambda epoch: lrfn(epoch), verbose=0)

# create a checkpoint path
checkpoint_filepath = './ver3_anime_model.h5'

# Instantiate model check point with  save_weights_only=True,monitor='val_loss',mode='min'
model_checkpoints = ModelCheckpoint(filepath=checkpoint_filepath,
                                    save_weights_only=True,
                                    monitor='val_loss',
                                    mode='min',
                                    save_best_only=True)

# Use early stopping to stop training when a monitored metric has stopped improving with p
early_stopping = EarlyStopping(patience = 3, monitor='val_loss',
                               mode='min', restore_best_weights=True)
```

```
# You can use callbacks to:
# Write TensorBoard logs after every batch of training to monitor your metrics
# Periodically save your model to disk
# Do early stopping
# Get a view on internal states and statistics of a model during training

# add callbacks in a list
my_callbacks = [
    model_checkpoints,
    lr_callback,
    early_stopping,
]
```

Train the model with epochs=1 (increase the number of epochs to increase the training accuracy)

```
# Model training for epochs=10, verbose = 1 and callbacks=my_callbacks
history = model.fit(
    x=X_train_array,
    y=y_train,
    batch_size=batch_size,
    epochs=10,
    verbose=1,
    validation_data=(X_test_array, y_test),
    callbacks=my_callbacks
)

model.load_weights(checkpoint_filepath)
```

```
Epoch 1/10
633/633 [==============================] - 78s 122ms/step - loss: 0.7775 - mae: 7.307
Epoch 2/10
633/633 [==============================] - 86s 135ms/step - loss: 0.6953 - mae: 7.306
Epoch 3/10
633/633 [==============================] - 78s 123ms/step - loss: 0.5630 - mae: 7.303
Epoch 4/10
633/633 [==============================] - 74s 117ms/step - loss: 0.3802 - mae: 7.299
Epoch 5/10
633/633 [==============================] - 75s 118ms/step - loss: 0.1437 - mae: 7.293
Epoch 6/10
633/633 [==============================] - 75s 119ms/step - loss: -0.1613 - mae: 7.28
Epoch 7/10
633/633 [==============================] - 74s 116ms/step - loss: -0.5220 - mae: 7.28
Epoch 8/10
633/633 [==============================] - 74s 117ms/step - loss: -0.8724 - mae: 7.27
Epoch 9/10
633/633 [==============================] - 74s 117ms/step - loss: -1.1803 - mae: 7.27
Epoch 10/10
633/633 [==============================] - 74s 117ms/step - loss: -1.4321 - mae: 7.27
```

Plot the loss with each epochs for train and test dataset

```
#Training results

# Import matplotlib library
import matplotlib.pyplot as plt
%matplotlib inline

# plot loss and val_loss for [0:-2]
plt.plot(history.history["loss"][0:-2])
plt.plot(history.history["val_loss"][0:-2])

# plot title of graph
plt.title("model loss")

# plot y label
plt.ylabel("loss")

# plot x label
plt.xlabel("epoch")

# label legends as train, test
plt.legend(["train", "test"], loc="upper left")

# display the graph
plt.show()
```

Extracting weights from model

```
# Create a function to extract weights from layers
def extract_weights(name, model):

    # get layers from model
    weight_layer = model.get_layer(name)

    # get weights from weight layer
    weights = weight_layer.get_weights()[0]
```

```
    # normalize weights from anime embedding / user embedding layer
    weights = weights / np.linalg.norm(weights, axis = 1).reshape((-1, 1))

    # return weights
    return weights

# call extract weights function and pass model as 2nd parameter for anime_embedding
anime_weights = extract_weights('anime_embedding', model)

# call extract weights function and pass model as 2nd parameter for user_embedding
user_weights = extract_weights('user_embedding', model)
```

Read the anime.csv file

```
# read anime.csv file with low_memory=True
df = pd.read_csv('/content/anime.csv', low_memory=True)

# replace 'unknown' with np.nan
df = df.replace("Unknown", np.nan)


df.head(1)
```

Create a function to get anime name based on anime id

```
# Fixing Names

# return anime name given an anime id
def getAnimeName(anime_id):
    try:
        name = df[df.anime_id == anime_id].eng_version.values[0]
        if name is np.nan:
            name = df[df.anime_id == anime_id].Name.values[0]
    except:
```

```
            print('error')

        return name
    # rename df feature's names
    df['anime_id'] = df['MAL_ID']
    df["eng_version"] = df['English name']
    df['eng_version'] = df.anime_id.apply(lambda x: getAnimeName(x))

    # sort df on the basis of score
    df.sort_values(by=['Score'],
                   inplace=True,
                   ascending=False,
                   kind='quicksort',
                   na_position='last')

    # selecting features for df
    df = df[["anime_id", "eng_version",
             "Score", "Genres", "Episodes",
             "Type", "Premiered", "Members"]]


    # get anime frame from anime
    def getAnimeFrame(anime):
        if isinstance(anime, int):
            # return where anime_id == anime
            return df[df.anime_id == anime]

        if isinstance(anime, str):
            # return where eng_version == anime
            return df[df.eng_version == anime]
```

Sypnopsis Data

```
cols = ["MAL_ID", "Name", "Genres", "sypnopsis"]

# read anime_with_synopsis.csv with usecols= cols
sypnopsis_df = pd.read_csv('/content/anime_with_synopsis.csv', usecols=cols)

# create a synopsis function to return sypnopsis_df
def getSypnopsis(anime):
    if isinstance(anime, int):
        # return first snynopsis of synopsis_df where its MAL_ID == anime
        return sypnopsis_df[sypnopsis_df.MAL_ID == anime].sypnopsis.values[0]

    if isinstance(anime, str):
        # return first snynopsis of synopsis_df where its Name == anime
        return sypnopsis_df[sypnopsis_df.Name == anime].sypnopsis.values[0]
```

Create a function to recommend similar anime

```
    # define empty list
    final_df = []

    # similar anime function
    def find_similar_animes(name):
        # choose top 5 recommendations
        n=5

        return_dist=False
        neg=False

        try:
            # get anime frame using .anime_id.values[0] on getAnimeFrame
            index = getAnimeFrame(name).anime_id.values[0]

            # use get method on anime2anime_encoded
            encoded_index = anime2anime_encoded.get(index)

            # equal to anime_weights
            weights = anime_weights

            # find distance of one anime with another wrt similarity using dot product
            dists = np.dot(weights, weights[encoded_index])

            # sort dists using argsort
            sorted_dists = np.argsort(dists)

            # find closest n recommendations
            n = n + 1
            if neg:
                # up to n from sorted_dists
                closest = sorted_dists[:n]
            else:
                # from -n to 0 in sorted_dists
                closest = sorted_dists[-n:]


            rindex = df
            # create a SimilarityArr empty list
            SimilarityArr = []

            # find closest n recommendations
            for close in closest:

                # get close from anime_encoded2anime
                decoded_id = anime_encoded2anime.get(close)

                # call getSynopsis on decoded_id
                sypnopsis = getSypnopsis(decoded_id)

                # get anime frame on decoded id
                anime_frame = getAnimeFrame(decoded_id)

                # get english version of anime name
                anime_name = anime_frame.eng_version.values[0]
```

```
            genre = anime_frame.Genres.values[0]
            similarity = dists[close]

            # append the columns to SimilarityArr
            SimilarityArr.append({"anime_id": decoded_id,"name": anime_name,
                                  "similarity": similarity,"genre": genre,
                                  'sypnopsis': sypnopsis
                                 })

            # sort Frame by similarity and convert SimilarityArr to dataframe
        Frame = pd.DataFrame(SimilarityArr).sort_values(by="similarity", ascending=False)

        # drop genre, synopsis features from frame dataframe
        Frame = Frame[Frame.anime_id != index].drop(['genre', 'sypnopsis'], axis=1)

        # copy the Frame as final_df dataframe
        final_df = pd.DataFrame(Frame)

        # return final_df
        return final_df

    # print if no anime is recommended
    except:
        print('{}!, Not Found in Anime list'.format(name))
```

Try the model that we created to recommend similar anime

```
# call the recommendation function with One Piece
find_similar_animes('One Piece')
```

```
 One Piece!, Not Found in Anime list
```

```
# call the recommendation function with Armitage III: Dual-Matrix
find_similar_animes('Armitage III: Dual-Matrix')
```

Save the model as .pkl file

```
# import pickle library
import pickle
```

```
# dump pickle file final_df with name movie_dict.pkl in wb mode
pickle.dump(final_df,open('movie_dict.pkl','wb'))
```

----------The End----------