Department of Data Science & Computer Applications,

Manipal Institute of Technology, Manipal

**DSE 3159 DEEP LEARNING LAB**

OBSERVATION BOOK

**Name: Sagar Kumar**

**Registration No: 210968002**

**Batch No: B1**

**Table of Contents**

[WEEK 1- Tensorflow & Keras Tutorial 3](#_Toc150108751)

[WEEK 2- EXPERIMENTING WITH DEEP NEURAL NETWORKS 7](#_Toc150108752)

[WEEK 3- CONVOLUTIONAL NEURAL NETWORKS VS FULLY CONNECTED NEURAL NETWORKS 14](#_Toc150108753)

[WEEK 4- IMPLEMENTING CONVENTIONAL CNN ARCHITECTURES AND TRANSFER LEARNING 23](#_Toc150108754)

[WEEK 5- IMPLEMENTING RECURRENT NEURAL NETWORKS FOR TIME SERIES 32](#_Toc150108755)

[FORECASTING AND STOCK MARK PREDICTION 32](#_Toc150108756)

[WEEK 7 – LSTM and GRU 40](#_Toc150108757)

[WEEK 8 - Neural Machine Translation (NMT) using Encoder-Decoder Architecture 55](#_Toc150108758)

[WEEK 9 - Neural Machine Translation (NMT) using Encoder-Decoder with attention and transformer mechanism. 65](#_Toc150108759)

[WEEK 10 – Autoencoders 82](#_Toc150108760)

[WEEK 11 – GAN’s 89](#_Toc150108761)

WEEK 1- Tensorflow & Keras Tutorial

**Q1)** Accurate measurement of body fat is inconvenient/costly and it is desirable to have easy methods of predicting Body Fat. Using the Body Fat dataset, write a Neural Network to predict body fat:

a. Number of Hidden layers = 3 and number of units are 128,64,32

b. Use RELU activation function, let learning rate be 0.1

Split the data into (80,20) split and tabulate the performance in terms of RMSE for100 epochs and comment on performance.

The attributes of the dataset are :

1. Density determined from underwater weighing

2. Percent body fat from Siri's (1956) equation

3. Age (years)

4. Weight (lbs)

5. Height (inches)

6. Neck circumference (cm)

7. Chest circumference (cm)

8. Abdomen 2 circumference (cm)

9. Hip circumference (cm)

10. Thigh circumference (cm)

11. Knee circumference (cm)

12. Ankle circumference (cm)

13. Biceps (extended) circumference (cm)

14. Forearm circumference (cm)

15. Wrist circumference (cm)

**Code:**

# Imports

import pandas as pd

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

keras = tf.keras

from keras.models import Sequential

from keras.layers import Dense, Input

from keras.utils import split\_dataset

from sklearn.model\_selection import train\_test\_split

# Read csv file and split to train and test datasets

df = pd.read\_csv('./bodyfat.csv')

X = df.drop(['BodyFat'], axis=1)

y = df.BodyFat

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, shuffle=True)

# Create, compile and train FCNN model on data

# Qa)

model = Sequential([

Input(shape=(14,)),

Dense(128, activation='relu'),

Dense(64, activation='relu'),

Dense(32, activation='relu'),

Dense(1, activation='linear'),

])

# Qb)

model.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=0.1), loss='mse', metrics=['accuracy'])

epochs = 100

batch\_size = 32

obj = model.fit(tf.convert\_to\_tensor(X\_train), tf.convert\_to\_tensor(y\_train), epochs=epochs, batch\_size=batch\_size, validation\_split=0.1)

# Performance tabulation

loss, acc = model.evaluate(tf.convert\_to\_tensor(X\_test), tf.convert\_to\_tensor(y\_test))

rmse = np.sqrt(obj.history['loss'])

pd.DataFrame({"epochs": range(1, epochs + 1), "rmse": rmse})

# Plots

fig = plt.figure()

plt.xlabel("Epochs")

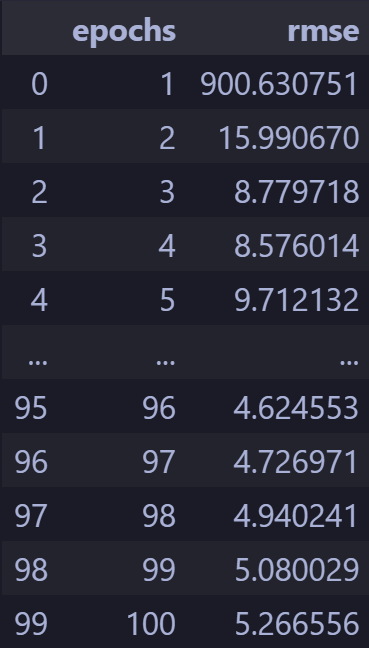
plt.ylabel("RMSE")

plt.title("RMSE vs Epochs")

plt.plot(range(1, epochs + 1), rmse)

**Results & Discussion:**

# Performance tabulation



# RMSE vs Epochs plot

A graph with a line

Description automatically generated

# RMSE vs Epochs plot w/o first 15 trials

A graph of a graph

Description automatically generated

# WEEK 2- EXPERIMENTING WITH DEEP NEURAL NETWORKS

**Q1)** Consider the following dataset ‘Churn\_Modelling.csv’

https://www.kaggle.com/datasets/aakash50897/churn-modellingcsv

The data set has 14 features which are as follows:-

1. RowNumber:- Represents the number of rows

2. CustomerId:- Represents customerId

3. Surname:- Represents surname of the customer

4. CreditScore:- Represents credit score of the customer

5. Geography:- Represents the city to which customers belongs to

6. Gender:- Represents Gender of the customer

7. Age:- Represents age of the customer

8. Tenure:- Represents tenure of the customer with a bank

9. Balance:- Represents balance hold by the customer

10. NumOfProducts:- Represents the number of bank services used by the customer

11. HasCrCard:- Represents if a customer has a credit card or not

12. IsActiveMember:- Represents if a customer is an active member or not

13. EstimatedSalary:- Represents estimated salary of the customer

14. Exited:- Represents if a customer is going to exit the bank or not.

1. Perform the required pre-processing (attribute removal, encoding, feature scaling) and write

comment lines to explain the pre-processing steps.

2. Perform experiments using (70,15,15) split and tabulate the performance in terms of Accuracy,

Precision & Recall for the following experimental setup :

1. Number of Hidden Layers and Number of Units per Layer

Number of Hidden

Layers

Number of Units Activation Function

1 128, 0 ,0 relu

2 128, 64, 0 relu

3 128, 64, 32 relu

2. Epochs (10,20,30)

3. Activation function in output layer (Sigmoid )

4. Learning rate ( 0.1, 0.01,0.001)

5. Visualize the training and validation loss against the epochs using appropriate plots.

6. Comment on performance.

**Code:**

# Imports

import tensorflow as tf

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

keras = tf.keras

from keras.layers import Dense

from keras import Sequential

from keras.optimizers import Adam

from keras.metrics import Recall, Precision

from sklearn.preprocessing import StandardScaler

import itertools as it

from sklearn.metrics import precision\_score, accuracy\_score, recall\_score

# Q1) Read dataset and preprocess

df = pd.read\_csv("churn.csv")

df.isna().any()

df.duplicated().any()

df = df.drop(['RowNumber', 'CustomerId', 'Surname'], axis=1)

df = pd.get\_dummies(df, dtype=np.int32)

df['Balance'] = df['Balance'].mask(df['Balance'] == 0).fillna(df['Balance'].mean())

df[['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary']] = StandardScaler().fit\_transform(df[['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary']])

# Split dataset into train and test

X = df.drop(['Exited'], axis=1)

y = df['Exited']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.15, random\_state=1)

# Q2) Create, compile and train multiple models

# Repeat 3 models 9 times

models = list(it.repeat([

Sequential([

Dense(128, activation='relu', input\_shape=(dims,)),

Dense(1, activation='sigmoid', input\_shape=(dims,)),

]),

Sequential([

Dense(128, activation='relu', input\_shape=(dims,)),

Dense(64, activation='relu', input\_shape=(dims,)),

Dense(1, activation='sigmoid', input\_shape=(dims,)),

]),

Sequential([

Dense(128, activation='relu', input\_shape=(dims,)),

Dense(64, activation='relu', input\_shape=(dims,)),

Dense(32, activation='relu', input\_shape=(dims,)),

Dense(1, activation='sigmoid', input\_shape=(dims,)),

]),

], 9))

batch\_size = 32

epochs = [10, 20, 30]

learning\_rate = [0.1, 0.01, 0.001]

# Train and compile models in a loop

for i in models:

for j in range(len(i)):

i[j].compile(optimizer=Adam(learning\_rate=learning\_rate[j]), loss='binary\_crossentropy', metrics=['accuracy', Recall(), Precision()])

trains = [

i[j].fit(X\_train, y\_train, epochs=epochs[j], batch\_size=batch\_size, validation\_split=0.15) for j in range(len(i)) for i in models

]

# Plots

epoch\_index = 0

fig, axs = plt.subplots(3, 3)

for j in range(3):

for k in range(3):

axs[j, k].plot(range(epochs[epoch\_index]), tlosses[1 + j + k])

axs[j, k].plot(range(epochs[epoch\_index]), vlosses[1 + j + k])

# Performance tabulation

acc = []

rec = []

prec = []

for i in range(9):

for model in models[i]:

y\_pred = model.predict(X\_test)

labels = np.where(y\_pred > 0.5, 1, 0)

acc.append(accuracy\_score(y\_test, labels))

rec.append(recall\_score(y\_test, labels))

prec.append(precision\_score(y\_test, labels))

**Results & Discussion:**

# Performance tabulation

A screenshot of a computer screen

Description automatically generated

# Q2.5) Plots

# Losses for 10 epochs



# Losses for 20 epochs

A screenshot of a graph

Description automatically generated

# Losses for 30 epochs

A screenshot of a graph

Description automatically generated

# Comments of performance

We notice that with more epochs, the validation loss increases and training loss decreases further. This is a clear sign of overfitting that can be fixed using regularization.

WEEK 3- CONVOLUTIONAL NEURAL NETWORKS VS FULLY CONNECTED NEURAL NETWORKS

Consider the Fashion MNIST dataset [Fashion MNIST dataset, an alternative to MNIST

(keras.io)] and do the following:

**Q1)** Understanding the Dataset and Pre-processing: Implement the following:

a. Compute and display the number of classes.

b. Compute and display the dimensions of each image.

c. Display one image from each class.

d. Perform normalization.

**Q2)** Performing experiments on Fully Connected Neural Networks (FCNN):

a. Design a FCNN which is most suitable for the given dataset:

Experimentally choose the best network (the intuitions and learnings from the

experiments you have performed in Week-1 and Week-2 will help you choose the

hyperparameters for the network).

b. Train and test the network (choose the best epoch size so that there is no overfitting).

c. Plot the performance curves.

**Q3)** Performing experiments on a Convolutional Neural Networks (CNNs):

a. Design CNN-1 which contains:

• One Convolution layer which uses 32 kernels each of size 5x5, stride = 1 and,

padding =0.

• One Pooling layer which uses MAXPOOLING with stride =2.

• One hidden layer having number of neurons = 100

b. Design CNN-2 which contains:

• Two back-to-back Convolution layers which uses 32 kernels each of size 3x3, stride

= 1, and padding =0.

• One Pooling layer which uses MAXPOOLING with stride =2.

• One hidden layer having number of neurons = 100

Note: use ReLU activation function after each convolution layer.

c. Train and test the networks (choose the best epoch size so that there is no overfitting).

d. Plot the performance curves for CNN-1 and CNN-2.

e. Compare the performances of CNN-1 and CNN-2.

**Q4)** Compare the performances of FCNN and CNN.

**Q5)** Compare the number of parameters in the FCNN and the CNN.

**Q6)** Discuss the computational efficiency of both networks. Which one took longer to train and

why?

**Code:**

# Imports

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

keras = tf.keras

from keras import Sequential, Input

from keras.layers import Flatten, Dense, MaxPooling2D, Conv2D

from keras.optimizers import Adam

from datetime import datetime

from tensorflow.math import confusion\_matrix

import seaborn as sns

# Load dataset

(x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.fashion\_mnist.load\_data()

# Q1a)

set(y\_train)

# Q1b)

x\_train[0].shape

# Q1c)

c = 0

y = list(y\_train)

for i in range(y\_train.shape[0]):

if c == 10: break

plt.imshow(x\_train[y.index(c)], cmap="gray")

c += 1

plt.show()

# Q1d)

x\_train = x\_train / 255.0

x\_test = x\_test / 255.0

# Q2a)

fcnn\_model = Sequential([

Flatten(input\_shape=[28, 28], name="IL"),

Dense(300, activation="relu", name="HL1"),

Dense(100, activation="relu", name="HL2"),

Dense(10, activation="softmax", name="OL"),

])

fcnn\_model.compile(optimizer=Adam(learning\_rate=0.01), loss="sparse\_categorical\_crossentropy", metrics=["accuracy"])

# Q2b)

start = datetime.now()

fcnn\_hist = fcnn\_model.fit(x\_train, y\_train, epochs=50, batch\_size=50, validation\_split=0.1)

end = datetime.now()

fcnn\_time = end – start

# Q2c)

plt.plot(fcnn\_hist.history["loss"], label="loss")

plt.plot(fcnn\_hist.history["val\_loss"], label="val loss")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

plt.show()

# Predictions

fcnn\_pred = np.argmax(fcnn\_model.predict(x\_test), axis=-1)

fcnn\_pred

# Q3a)

cnn1\_model = Sequential([

Input((x\_train.shape[1], x\_train.shape[2], 1)),

Conv2D(32, (5, 5), strides=(1, 1), padding="valid", activation="relu"),

MaxPooling2D(pool\_size=(2, 2), strides=2, padding="valid"),

Flatten(),

Dense(100, activation="relu"),

Dense(10, activation="sigmoid"),

])

cnn1\_model.compile(optimizer=Adam(learning\_rate=0.01), loss="sparse\_categorical\_crossentropy", metrics=["accuracy"])

start = datetime.now()

cnn1\_hist = cnn1\_model.fit(x\_train, y\_train, epochs=50, batch\_size=50, validation\_split=0.1)

end = datetime.now()

cnn1\_time = end – start

# Plots

plt.plot(cnn1\_hist.history["loss"], label="loss")

plt.plot(cnn1\_hist.history["val\_loss"], label="val loss")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

plt.show()

# Q3b)

cnn2\_model = Sequential([

Input((x\_train.shape[1], x\_train.shape[2], 1)),

Conv2D(32, (3, 3), strides=(1, 1), padding="valid", activation="relu"),

Conv2D(32, (3, 3), strides=(1, 1), padding="valid", activation="relu"),

MaxPooling2D(pool\_size=(2, 2), strides=2, padding="valid"),

Flatten(),

Dense(100, activation="relu"),

Dense(10, activation="sigmoid"),

])

cnn2\_model.compile(optimizer=Adam(learning\_rate=0.01), loss="sparse\_categorical\_crossentropy", metrics=["accuracy"])

start = datetime.now()

cnn2\_hist = cnn2\_model.fit(x\_train, y\_train, epochs=25, batch\_size=50, validation\_split=0.1)

end = datetime.now()

cnn2\_time = end – start

cnn2\_loss, cnn2\_acc = cnn2\_model.evaluate(x\_test, y\_test, batch\_size=50)

print(f"Loss: {cnn2\_loss}")

print(f"Accuracy: {cnn2\_acc}")

# Plots

plt.plot(cnn2\_hist.history["loss"], label="loss")

plt.plot(cnn2\_hist.history["val\_loss"], label="val loss")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

plt.show()

**Results & Discussion:**

# Confusion matrix for FCNN

A screenshot of a computer screen

Description automatically generated

# Q3d) Plots

# FCNN

A graph with lines and numbers

Description automatically generated

A graph with lines and text

Description automatically generated

# CNN 1

A graph with a line and a black background

Description automatically generated

A graph with lines and numbers

Description automatically generated

# CNN 2

A graph with lines and numbers on it

Description automatically generated

A graph with lines and numbers

Description automatically generated

# Q3e)

The second CNN with 2 convolutional layers seems to perform better than the CNN with 1 layer but tends to overfit much faster. Hence, I had to reduce the epochs to 25 to avoid overfitting.

# Q4)

Both FCNN and CNN achieved high accuracy with the fashion mnist dataset. However, CNN was more efficient and performed better due to its ability to extract relevant features through convolutions.

# Q5)

FCNN has lesser parameters than CNN, 266610 v/s 462742 in the above models because of the architecture that I have chosen as depicted by `model.summary()`

# Q6)

The second CNN with 2 convolutional layers takes more time than the first CNN with 1 convolutional layer due to it having more convolutional layers which implies there is more computation involved.

The FCNN techincally should take more time to train than a CNN due to it having larger number of weights to be trained due to lack of convolutions and that CNN's are designed to extract the most relevant features from the images passed to it resulting in fewer parameters

The FCNN here shows a 7 second advantage which might be due to external factors like differing GPU loads during training, processes taking variable amounts of computer power randomly, etc.

# WEEK 4- IMPLEMENTING CONVENTIONAL CNN ARCHITECTURES AND TRANSFER LEARNING

**Q1)**

A) Implement the LeNet-5, AlexNet architecture.

B) Train, test and compare the performances of these two models on the

Cats\_and\_Dogs\_Dataset, Horse2Zebra Dataset

Note:

• The Cats\_and\_Dogs\_Dataset can downloaded from:

https://storage.googleapis.com/mledu-

datasets/cats\_and\_dogs\_filtered.zip

• Horse2Zebra Dataset can be downloaded from:

https://www.kaggle.com/datasets/balraj98/horse2zebra-dataset

**Q2)**

Train, test and report the performances using the following models on the

Cats\_and\_Dogs\_Dataset and Horse2Zebra dataset.

A) VGG-16

B) GoogleNet

C) ResNet50

D) EfficientNetB0

E) MobileNetV2

**Code:**

# Imports

import tensorflow as tf

keras = tf.keras

from keras import Sequential, Input

from keras.layers import Conv2D, Dense, AveragePooling2D, Flatten, MaxPooling2D

from keras.preprocessing.image import ImageDataGenerator

import matplotlib.pyplot as plt

from keras.preprocessing.image import img\_to\_array, load\_img

from keras.optimizers import Adam

import os

# Load data

train\_datagen = ImageDataGenerator(rescale=1./255)

val\_datagen = ImageDataGenerator(rescale=1./255)

test\_datagen = ImageDataGenerator(rescale=1./255)

base\_dir = './catsdogs'

train\_dir = os.path.join(base\_dir, 'train')

validation\_dir = os.path.join(base\_dir, 'validation')

test\_dir = os.path.join(base\_dir, 'test')

# CatsDogs

train\_cats\_dir = os.path.join(train\_dir, 'cats')

train\_dogs\_dir = os.path.join(train\_dir, 'dogs')

validation\_cats\_dir = os.path.join(validation\_dir, 'cats')

validation\_dogs\_dir = os.path.join(validation\_dir, 'dogs')

test\_cats\_dir = os.path.join(test\_dir, 'cats')

test\_dogs\_dir = os.path.join(test\_dir, 'dogs')

train\_generator = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=(28, 28),

batch\_size=20,

class\_mode='categorical',

)

val\_generator = val\_datagen.flow\_from\_directory(

validation\_dir,

target\_size=(28, 28),

batch\_size=20,

class\_mode='categorical',

)

test\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=(28, 28),

batch\_size=20,

class\_mode='categorical',

)

# HorseZebra

base\_dir = './horse2zebra'

train\_dir = os.path.join(base\_dir, 'train')

validation\_dir = os.path.join(base\_dir, 'validation')

test\_dir = os.path.join(base\_dir, 'test')

train\_horse\_dir = os.path.join(train\_dir, 'trainA')

train\_zebra\_dir = os.path.join(train\_dir, 'trainB')

validation\_horse\_dir = os.path.join(validation\_dir, 'valA')

validation\_zebra\_dir = os.path.join(validation\_dir, 'valB')

test\_horse\_dir = os.path.join(test\_dir, 'testA')

test\_zebra\_dir = os.path.join(test\_dir, 'testB')

train\_generator = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=(28, 28),

batch\_size=20,

class\_mode='categorical',

)

test\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=(28, 28),

batch\_size=20,

class\_mode='categorical',

)

val\_generator = val\_datagen.flow\_from\_directory(

validation\_dir,

target\_size=(28, 28),

batch\_size=20,

class\_mode='categorical',

)

# Q1A,B) Create models

lenet\_model = Sequential([

Input(shape=(28, 28, 3)),

Conv2D(6, (5, 5), padding="same", activation="tanh"),

AveragePooling2D(strides=2),

Conv2D(16, (5, 5), padding="valid", activation="tanh"),

AveragePooling2D(strides=2),

Flatten(),

Dense(120, activation="sigmoid"),

Dense(84, activation="sigmoid"),

Dense(2, activation="sigmoid"),

], name="LeNet")

lenet\_model.compile(loss='binary\_crossentropy', optimizer=Adam(learning\_rate=0.001), metrics=['acc'])

lenet\_hist = lenet\_model.fit(train\_generator, epochs=20, validation\_data=val\_generator)

lenet\_model.evaluate(test\_generator, batch\_size=32)

alexnet\_model = Sequential([

Input(shape=(227, 227, 3)),

Conv2D(96, (11, 11), strides=(4, 4), padding="valid", activation="tanh"),

MaxPooling2D(pool\_size=(3, 3), strides=2),

Conv2D(256, (5, 5), padding="same", activation="tanh"),

MaxPooling2D(pool\_size=(3, 3), strides=2),

Conv2D(384, (3, 3), padding="same", activation="tanh"),

Conv2D(384, (3, 3), padding="same", activation="tanh"),

Conv2D(256, (3, 3), padding="same", activation="tanh"),

MaxPooling2D(pool\_size=(3, 3), strides=2),

Flatten(),

Dense(4096, activation="sigmoid"),

Dense(4096, activation="sigmoid"),

Dense(2, activation="sigmoid"),

], name="AlexNet")

alexnet\_model.compile(loss='binary\_crossentropy', optimizer=Adam(learning\_rate=0.001), metrics=['acc'])

alexnet\_hist = alexnet\_model.fit(train\_generator, epochs=20, validation\_data=val\_generator)

alexnet\_model.evaluate(test\_generator)

# Q2A) VGG-16

vgg\_model = VGG16(input\_shape=(224, 224, 3), weights='imagenet', include\_top=False)

vgg\_model.trainable = False

custom\_vgg\_model = Sequential([

vgg\_model,

Flatten(),

Dense(2, activation='softmax')

])

custom\_vgg\_model.compile(loss='categorical\_crossentropy', optimizer=Adam(learning\_rate = 0.0001), metrics=['acc'])

vgg\_history = custom\_vgg\_model.fit(train\_generator, epochs=20, validation\_data=val\_generator)

custom\_vgg\_model.evaluate(test\_generator)

# Q2B) GoogleNet

google\_model = InceptionV3(input\_shape=(224, 224, 3), weights='imagenet', include\_top=False)

google\_model.trainable = False

custom\_google\_model = Sequential([

google\_model,

Flatten(),

Dense(2, activation='softmax')

])

custom\_google\_model.compile(loss='categorical\_crossentropy', optimizer=Adam(learning\_rate = 0.0001), metrics=['acc'])

google\_history = custom\_google\_model.fit(train\_generator, epochs=20, validation\_data=val\_generator)

custom\_google\_model.evaluate(test\_generator)

# Q2C) ResNet50

res\_model = ResNet50(input\_shape=(224, 224, 3), weights='imagenet', include\_top=False)

res\_model.trainable = False

custom\_res\_model = Sequential([

res\_model,

Flatten(),

Dense(2, activation='softmax')

])

custom\_res\_model.compile(loss='categorical\_crossentropy', optimizer=Adam(learning\_rate = 0.0001), metrics=['acc'])

res\_history = custom\_res\_model.fit(train\_generator, epochs=20, validation\_data=val\_generator)

custom\_res\_model.evaluate(test\_generator)

# Q2D) EfficientNetB0

eff\_model = EfficientNetB0(input\_shape=(224, 224, 3), weights='imagenet', include\_top=False)

eff\_model.trainable = False

custom\_eff\_model = Sequential([

eff\_model,

Flatten(),

Dense(2, activation='softmax')

])

custom\_eff\_model.compile(loss='categorical\_crossentropy', optimizer=Adam(learning\_rate = 0.0001), metrics=['acc'])

eff\_history = custom\_eff\_model.fit(train\_generator, epochs=20, validation\_data=val\_generator)

custom\_eff\_model.evaluate(test\_generator)

# Q2E) MobileNetV2

mobile\_model = MobileNetV2(input\_shape=(224, 224, 3), weights='imagenet', include\_top=False)

mobile\_model.trainable = False

custom\_mobile\_model = Sequential([

mobile\_model,

Flatten(),

Dense(2, activation='softmax')

])

custom\_mobile\_model.compile(loss='categorical\_crossentropy', optimizer=Adam(learning\_rate = 0.0001), metrics=['acc'])

mobile\_history = custom\_mobile\_model.fit(train\_generator, epochs=20, validation\_data=val\_generator)

custom\_mobile\_model.evaluate(test\_generator)

**Results & Discussion:**

# Catsdogs

Lenet:

Loss: 0.6820

Accuracy: 0.6007

AlexNet:

Loss: 0.7056

Accuracy: 0.5

LeNet performs better than AlexNet in this instance

VGG16:

Loss: 0.2222

Accuracy: 0.9092

GoogleNet:

Loss: 0.0510

Accuracy: 0.9876

ResNet50

Loss: 0.5743

Accuracy: 0.6927

EfficientNetB0

Loss: 0.6945

Accuracy: 0.5000

MobileNetV2

Loss: 0.0871

Accuracy: 0.9789

GoogleNet (InceptionV3 which is not actually GoogleNet – InceptionV1) performs the best overall

# HorseZebra

Lenet:

Loss: 0.4199

Accuracy: 0.8038

AlexNet:

Loss: 0.6913

Accuracy: 0.5385

LeNet yet again performs better here.

VGG16:

Loss: 0.0904

Accuracy: 0.9692

GoogleNet:

Loss: 0.0956

Accuracy: 0.9846

ResNet50

Loss: 0.2821

Accuracy: 0.8885

EfficientNetB0

Loss: 1.1298

Accuracy: 0.5077

MobileNetV2

Loss: 0.0983

Accuracy: 0.9846

GoogleNet yet again performs the best overall.

# WEEK 5- IMPLEMENTING RECURRENT NEURAL NETWORKS FOR TIME SERIES

# FORECASTING AND STOCK MARK PREDICTION

**Q1)** Use the following code to generate a time series:

def generate\_time\_series(sample\_size, n\_steps):

freq1, freq2, offsets1, offsets2 = np.random.rand(4, sample\_size, 1)

time = np.linspace(0, 1, n\_steps)

series = 0.5 \* np.sin((time - offsets1) \* (freq1 \* 10 + 10)) #wave1+

series += 0.2 \* np.sin((time - offsets2) \* (freq2 \* 20 + 20)) #wave2+

series += 0.1 \* (np.random.rand(sample\_size, n\_steps) - 0.5) #noise

return series[..., np.newaxis].astype(np.float32)

The above code generates as many time series as requested, which can be specified using the

“sample\_size” argument. Each time series is of length “n\_steps” and there is just one value per time

step in each series.

Use the above code to do the following:

A) Create a dataset of 10,000 samples with 51 time steps each (Note: the 51st time step should be

used as the label)

B) Split the dataset in the ratio training: validation: testing = 70:20:10.

C) Design, train, test and compare the performances of the following on the prediction of the

value of 51st time step in the generated time series.

a. Fully connected neural network.

b. Simple RNN with one layer (output layer)

c. Simple RNN with one hidden layer and one output layer.

d. Simple RNN with one hidden layer and one output layer.

e. Simple RNN with two hidden layers and one output layer.

**Q2)** Consider the Google Stock Prediction dataset.

The 14 columns are:

symbol : - Name of the company (in this case Google).

date :- year and date

close:- closing of stock value

high:- highest value of stock at that day

low:- lowest value of stock at that day

open:- opening value of stock at that day

volume

adjClose

adjHigh

adjLow

adjOpen

adjVolume

divCash

splitFactor

A. Build a Simple RNN model with 5 layers (use dropouts if required) to predict the stock price for

the years 2020 and 2021.

B. Compare the accuracy using MAPE and MSE.

C. Comment on how many epochs (dropouts) is required for adequate learning.

D. Plot the actual vs predicted values using the test data for the year 2020 and 2021 .

**Code:**

# Imports

import tensorflow as tf

keras = tf.keras

from keras.layers import SimpleRNN, Dense, Flatten, Dropout

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_absolute\_percentage\_error, mean\_squared\_error

from keras import Sequential, Input

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# Q1A) Generating time series data

def gen\_time\_series(sample\_size: int, n\_steps: int):

freq1, freq2, offsets1, offsets2 = np.random.rand(4, sample\_size, 1)

time = np.linspace(0, 1, n\_steps)

series = 0.5 \* np.sin((time - offsets1) \* (freq1 \* 10 + 10))

series += 0.2 \* np.sin((time - offsets2) \* (freq2 \* 20 + 20))

series += 0.1 \* (np.random.rand(sample\_size, n\_steps) - 0.5)

return series[..., np.newaxis].astype(np.float32)

data = gen\_time\_series(10000, 51)

X = data[:, 0:50]

y = data[:, -1]

# Q1B) Split dataset into train, val and test

X\_train, X\_val, X\_test = X[0:7000, :], X[7000:9000, :], X[9000:10000, :]

y\_train, y\_val, y\_test = y[0:7000, :], y[7000:9000, :], y[9000:10000, :]

# Q1Ca) FCNN model

fcnn\_model = Sequential([

Flatten(input\_shape=[50, 1]),

Dense(1)

])

fcnn\_model.compile(loss="mse", optimizer="adam")

fcnn\_hist = fcnn\_model.fit(X\_train, y\_train, epochs=20, validation\_data=(X\_val, y\_val))

fcnn\_model.evaluate(X\_test, y\_test)

# Q1Cb) RNN 1 model

rnn1\_model = Sequential([

Input((50, 1)),

SimpleRNN(1)

])

rnn1\_model.compile(loss="mse", optimizer="adam", metrics=['accuracy'])

rnn1\_hist = rnn1\_model.fit(X\_train, y\_train, epochs=20, validation\_data=(X\_val, y\_val))

rnn1\_model.evaluate(X\_test, y\_test)

# Q1Cc,d) RNN 2

rnn2\_model = Sequential([

Input((50, 1)),

SimpleRNN(1, return\_sequences=True),

SimpleRNN(1)

])

rnn2\_model.compile(loss="mse", optimizer="adam")

rnn2\_hist = rnn2\_model.fit(X\_train, y\_train, epochs=20, validation\_data=(X\_val, y\_val))

rnn2\_model.evaluate(X\_test, y\_test)

# Q1Ce) RNN 3

rnn3\_model = Sequential([

Input((50, 1)),

SimpleRNN(1, return\_sequences=True),

SimpleRNN(1, return\_sequences=True),

SimpleRNN(1)

])

rnn3\_model.compile(loss="mse", optimizer="adam")

rnn3\_hist = rnn3\_model.fit(X\_train, y\_train, epochs=20, validation\_data=(X\_val, y\_val))

rnn3\_model.evaluate(X\_test, y\_test)

# Q2)

# Read data

df = pd.read\_csv('GOOG.csv')

# Preprocess

df['date'] = pd.to\_datetime(df['date'])

df\_2020\_2021 = df[(df['date'].dt.year >= 2020) & (df['date'].dt.year <= 2021)]

features = ['close', 'high', 'low', 'open', 'volume', 'adjClose', 'adjHigh', 'adjLow', 'adjOpen', 'adjVolume']

df\_selected = df\_2020\_2021[features]

scaler = MinMaxScaler()

df\_norm = scaler.fit\_transform(df\_selected)

window\_size = 30

sequences = []

target = []

for i in range(len(df\_norm) - window\_size):

sequences.append(df\_norm[i:i+window\_size])

target.append(df\_norm[i+window\_size])

sequences = np.array(sequences)

target = np.array(target)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(sequences, target, test\_size=0.2, random\_state=42)

# Q2A) RNN model

model = Sequential([

SimpleRNN(units=64, activation='relu', return\_sequences=True, input\_shape=(window\_size, len(features))),

Dropout(0.2),

SimpleRNN(units=64, activation='relu', return\_sequences=True),

Dropout(0.2),

SimpleRNN(units=64, activation='relu', return\_sequences=True),

Dropout(0.2),

SimpleRNN(units=64, activation='relu'),

Dropout(0.2),

Dense(units=len(features))

])

model.compile(optimizer='adam', loss='mean\_squared\_error')

hist = model.fit(X\_train, y\_train, epochs=50, batch\_size=32,validation\_split=0.1)

# Plots

training = hist.history['loss']

validation = hist.history['val\_loss']

plt.plot(training, label='Training')

plt.plot(validation, label='Validation')

plt.legend()

plt.show()

# Q2B) MSE and MAPE

predicted\_prices = model.predict(X\_test)

predicted\_prices\_actual = scaler.inverse\_transform(predicted\_prices)

mape = mean\_absolute\_percentage\_error(y\_test, predicted\_prices)

print(f"MAPE: {mape:.2f}%")

mse = mean\_squared\_error(y\_test, predicted\_prices)

print(f"MSE: {mse:.2f}")

# Q2D) Actual vs Predicted prices

actual\_price = scaler.inverse\_transform(y\_test)

plt.plot(actual\_price[:, 0], label='Actual')

plt.plot(predicted\_prices\_actual[:, 0], label='Predicted')

plt.xlabel('Day')

plt.ylabel('Normalized Close Price')

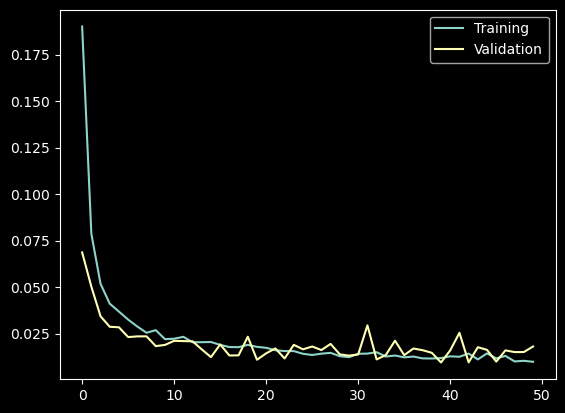
plt.title('Actual vs Predicted')

plt.legend()

plt.show()

**Results & Discussion:**

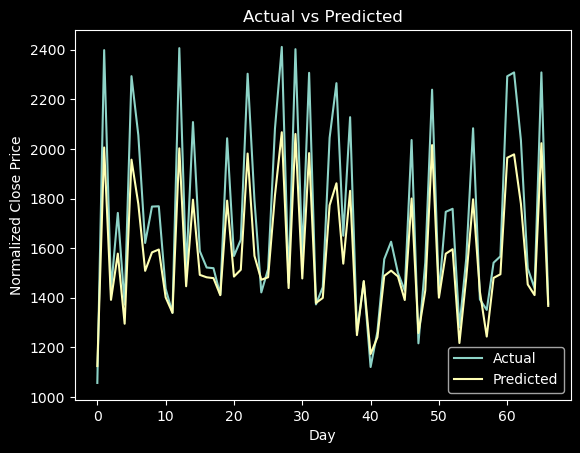
# Loss plot



# Q2C) Results

Good results were seen with 50 epochs and 4 dropouts

# Q2D) Actual vs Predicted price plot



WEEK 7 – LSTM and GRU

**Q1)** Using the following data set:

https://www.kaggle.com/code/cemalefetezcan/imdb-review-sentiment-classification

divide the data into train/validation data sets, build 2 models to perform movie review sentiment analysis

Model 1:

1. Perform required text pre-processing – lowering text, removing URLs, punctuation , stop words and

correct spelling .

2. Perform tokenization and lemmatization on cleaned data .

3. Visualize the most frequent words and bigrams

4. Visualize the practical words that represent positive and negative sentiment in the dataset.

5. Create an embedding layer and build a 15 layer LSTM/GRU and a 20 layer BidRNN for predicting

the sentiment.

6. Build your own test dataset with 10 movie reviews and tabulate accuracy.

**Code:**

# Imports

import nltk

from nltk.corpus import stopwords

from nltk.stem.porter import PorterStemmer

from nltk.stem import WordNetLemmatizer

import tensorflow as tf

keras = tf.keras

from keras import Input, Sequential

from keras.optimizers import Adam

from keras.layers import Dense, LSTM, Dropout, Embedding, GRU, Bidirectional, BatchNormalization

from wordcloud import WordCloud

from collections import Counter

import csv

import math

import numpy as np

import pandas as pd

import re

import string

from keras.preprocessing.text import Tokenizer

from nltk.tokenize import word\_tokenize

from textblob import TextBlob

from functools import lru\_cache

from sklearn.feature\_extraction.text import CountVectorizer

import matplotlib.pyplot as plt

from tensorflow.keras.preprocessing.sequence import pad\_sequences

nltk.download("stopwords")

nltk.download("punkt")

nltk.download("wordnet")

# Q1)

train\_df = pd.read\_csv("Train.csv")

test\_df = pd.read\_csv("Test.csv")

val\_df = pd.read\_csv("Valid.csv")

train\_df.head(3)

train\_df = train\_df.drop\_duplicates()

test\_df = test\_df.drop\_duplicates()

val\_df = val\_df.drop\_duplicates()

train\_df.shape

VOCAB\_SIZE = 10000

MAX\_LENGTH = 200

OUTPUT\_DIM = 128

# Create stemmer, tokenizer and lemmatizer

stemmer = PorterStemmer()

lemmatizer = WordNetLemmatizer()

tokenizer = Tokenizer(num\_words=VOCAB\_SIZE, oov\_token="<OOV>")

# Preprocessing

@lru\_cache(maxsize=None)

def preprocess(text: str) -> str:

text = text.lower()

text = re.sub('\[.\*?\]', "", text)

text = re.sub('\\W', " ", text)

text = re.sub("https?://\S+|www\.\S+", "", text)

text = re.sub('<.\*?>+', "", text)

text = re.sub('[%s]' % re.escape(string.punctuation), "", text)

text = re.sub('\n', "", text)

text = re.sub('\w\*\d\w\*', "", text)

return text

stop\_words = set(stopwords.words('english'))

stop\_words.add("br")

@lru\_cache(maxsize=None)

def tokenize\_and\_lemmatize(text: str) -> [str]:

tokens = word\_tokenize(text)

stemmed = [lemmatizer.lemmatize(token) for token in tokens if token not in stop\_words]

return " ".join(stemmed)

train\_df.iloc[:, 0] = train\_df.iloc[:, 0].apply(lambda x: tokenize\_and\_lemmatize(preprocess(x)))

train\_df.head(3)

test\_df.iloc[:, 0] = test\_df.iloc[:, 0].apply(lambda x: tokenize\_and\_lemmatize(preprocess(x)))

test\_df.head(3)

val\_df.iloc[:, 0] = val\_df.iloc[:, 0].apply(lambda x: tokenize\_and\_lemmatize(preprocess(x)))

val\_df.head(3)

# Frequent word plot

cv = CountVectorizer(ngram\_range=(1, 1))

cv\_train = cv.fit\_transform(train\_df.iloc[:, 0])

word\_freq = pd.DataFrame(cv\_train.sum(axis=0), columns=cv.get\_feature\_names\_out()).T.sort\_values(by=0, ascending=False)

plt.figure(figsize=(10, 6))

plt.bar(word\_freq.index[:20], word\_freq.iloc[:20, 0])

plt.xticks(rotation=90)

plt.title('Top 20 Most Frequent Words in Train Data')

plt.xlabel('Words and Bigrams')

plt.ylabel('Frequency')

plt.show()

# Positive and negative sentiment plot

pos\_df = train\_df[train\_df['label'] == 1]

neg\_df = train\_df[train\_df['label'] == 0]

cv\_pos = CountVectorizer(ngram\_range=(1, 1))

cv\_neg = CountVectorizer(ngram\_range=(1, 1))

cv\_train\_pos = cv\_pos.fit\_transform(pos\_df.iloc[:, 0])

cv\_train\_neg = cv\_neg.fit\_transform(neg\_df.iloc[:, 0])

word\_freq\_pos = pd.DataFrame(cv\_train\_pos.sum(axis=0), columns=cv\_pos.get\_feature\_names\_out()).T.sort\_values(by=0, ascending=False)

word\_freq\_neg = pd.DataFrame(cv\_train\_neg.sum(axis=0), columns=cv\_neg.get\_feature\_names\_out()).T.sort\_values(by=0, ascending=False)

plt.figure(figsize=(10, 6))

plt.bar(word\_freq\_pos.index[:20], word\_freq\_pos.iloc[:20, 0])

plt.xticks(rotation=90)

plt.title('Top 20 Most Frequent Words and Bigrams in Positive Sentiment')

plt.xlabel('Words and Bigrams')

plt.ylabel('Frequency')

plt.show()

plt.figure(figsize=(10, 6))

plt.bar(word\_freq\_neg.index[:20], word\_freq\_neg.iloc[:20, 0])

plt.xticks(rotation=90)

plt.title('Top 20 Most Frequent Words and Bigrams in Negative Sentiment')

plt.xlabel('Words and Bigrams')

plt.ylabel('Frequency')

plt.show()

# Sentiment wordcloud plot

positive\_text = train\_df[train\_df['label'] == 0]['text'].values

negative\_text = train\_df[train\_df['label'] == 1]['text'].values

positive\_freq = Counter(" ".join(positive\_text).split())

negative\_freq = Counter(" ".join(negative\_text).split())

positive\_wordcloud = WordCloud(width=800, height=500, random\_state=21, max\_font\_size=110).generate\_from\_frequencies(positive\_freq)

negative\_wordcloud = WordCloud(width=800, height=500, random\_state=21, max\_font\_size=110).generate\_from\_frequencies(negative\_freq)

plt.figure(figsize=(10, 7))

plt.imshow(positive\_wordcloud, interpolation="bilinear")

plt.axis('off')

plt.title('Words representing Positive Sentiment')

plt.show()

plt.figure(figsize=(10, 7))

plt.imshow(negative\_wordcloud, interpolation="bilinear")

plt.axis('off')

plt.title('Words representing Negative Sentiment')

plt.show()

# Tokenizing

tokenizer.fit\_on\_texts(train\_df.iloc[:, 0])

train\_sequences = tokenizer.texts\_to\_sequences(train\_df.iloc[:, 0])

train\_sequences = pad\_sequences(train\_sequences, maxlen=MAX\_LENGTH, padding='post', truncating='post')

tokenizer.fit\_on\_texts(test\_df.iloc[:, 0])

test\_sequences = tokenizer.texts\_to\_sequences(test\_df.iloc[:, 0])

test\_sequences = pad\_sequences(test\_sequences, maxlen=MAX\_LENGTH, padding='post', truncating='post')

train\_labels= pd.get\_dummies(train\_df['label']).values

test\_labels= pd.get\_dummies(test\_df['label']).values

val\_labels= pd.get\_dummies(val\_df['label']).values

# Q2)

# Model creation and training

lstm\_model = Sequential([

Embedding(input\_dim=VOCAB\_SIZE, output\_dim=OUTPUT\_DIM, input\_length=MAX\_LENGTH),

LSTM(64, return\_sequences=True),

BatchNormalization(),

Dropout(0.5),

LSTM(64),

BatchNormalization(),

Dropout(0.5),

Dense(2, activation="sigmoid"),

])

lstm\_model.compile(optimizer=Adam(learning\_rate=0.0001), loss="binary\_crossentropy", metrics=["accuracy"])

lstm\_model.summary()

lstm\_hist = lstm\_model.fit(train\_sequences, train\_labels, epochs=3, validation\_data=(val\_sequences, val\_labels))

gru\_model = Sequential([

Embedding(input\_dim=VOCAB\_SIZE, output\_dim=OUTPUT\_DIM, input\_length=MAX\_LENGTH),

GRU(64, return\_sequences=True),

BatchNormalization(),

Dropout(0.5),

GRU(64),

BatchNormalization(),

Dropout(0.5),

Dense(2, activation="sigmoid"),

])

gru\_model.compile(optimizer=Adam(learning\_rate=0.0001), loss="binary\_crossentropy", metrics=["accuracy"])

gru\_model.summary()

gru\_hist = gru\_model.fit(train\_sequences, train\_labels, epochs=3, validation\_data=(val\_sequences, val\_labels))

bid\_model = Sequential([

Embedding(input\_dim=VOCAB\_SIZE, output\_dim=OUTPUT\_DIM, input\_length=MAX\_LENGTH),

Bidirectional(LSTM(64, return\_sequences=True)),

BatchNormalization(),

Dropout(0.5),

Bidirectional(LSTM(64, return\_sequences=True)),

BatchNormalization(),

Dropout(0.5),

Bidirectional(LSTM(64)),

BatchNormalization(),

Dropout(0.5),

Dense(2, activation="sigmoid"),

])

bid\_model.compile(optimizer=Adam(learning\_rate=0.0001), loss="binary\_crossentropy", metrics=["accuracy"])

bid\_model.summary()

bid\_hist = bid\_model.fit(train\_sequences, train\_labels, epochs=3, validation\_data=(val\_sequences, val\_labels))

# Q3)

# Loss plots

plt.title("LSTM Train vs Validation loss curve")

plt.plot(range(3), lstm\_hist.history["loss"])

plt.plot(range(3), lstm\_hist.history["val\_loss"])

plt.title("GRU Train vs Validation loss curve")

plt.plot(range(3), gru\_hist.history["loss"])

plt.plot(range(3), gru\_hist.history["val\_loss"])

plt.title("Bidirectional Train vs Validation loss curve")

plt.plot(range(3), bid\_hist.history["loss"])

plt.plot(range(3), bid\_hist.history["val\_loss"])

# Q4)

data = {

'text': [

'An outstanding masterpiece with excellent performances.',

'A timeless classic that never fails to impress.',

'A thrilling experience with a memorable villain.',

'A unique and unconventional film that\'s not for everyone.',

'A disaster of a movie that\'s so bad it\'s good.',

'A beautiful love story that will bring you to tears.',

'A lackluster film with a weak plot and poor character development.',

'An epic conclusion to the Marvel Cinematic Universe saga.',

'A bizarre and confusing adaptation that misses the mark.',

'A brilliant social commentary with unexpected twists and turns.'

],

'label': [1, 1, 1, 0, 0, 1, 0, 1, 0, 1]

}

df = pd.DataFrame(data)

df

tokenizer.fit\_on\_texts(df.iloc[:, 0])

test\_sequences = tokenizer.texts\_to\_sequences(df.iloc[:, 0])

test\_sequences = pad\_sequences(test\_sequences, maxlen=MAX\_LENGTH, padding='post', truncating='post')

test\_labels = df['label']

predictions = lstm\_model.predict(test\_sequences)

predicted\_labels = [int(round(np.argmax(x))) for x in predictions]

correct\_predictions = sum([predicted\_labels[i] == test\_labels[i] for i in range(len(test\_labels))])

lstm\_accuracy = correct\_predictions / len(test\_labels)

predictions = gru\_model.predict(test\_sequences)

predicted\_labels = [int(round(np.argmax(x))) for x in predictions]

correct\_predictions = sum([predicted\_labels[i] == test\_labels[i] for i in range(len(test\_labels))])

gru\_accuracy = correct\_predictions / len(test\_labels)

predictions = bid\_model.predict(test\_sequences)

predicted\_labels = [int(round(np.argmax(x))) for x in predictions]

correct\_predictions = sum([predicted\_labels[i] == test\_labels[i] for i in range(len(test\_labels))])

bid\_accuracy = correct\_predictions / len(test\_labels)

print(lstm\_accuracy, gru\_accuracy, bid\_accuracy)

**Results & Discussion:**

# Data analysis plots

# Frequent uni/bigrams

A graph of blue columns

Description automatically generated with medium confidence

A graph of a number of blue bars

Description automatically generated

# Sentiment graphs

A graph of blue bars

Description automatically generated

A graph of blue bars

Description automatically generated

A black background with words

Description automatically generated

A close up of words

Description automatically generated

# Train plots

# Loss plots

A line graph with a blue line

Description automatically generated

A graph of a line

Description automatically generated

A graph of a line

Description automatically generated with medium confidence

# Accuracy for madeup test dataset

LSTM – 0.6

GRU – 0.6

BidRNN – 0.5

WEEK 8 - Neural Machine Translation (NMT) using Encoder-Decoder Architecture

**Q1)** Using the following NMT repo:

Tab-delimited Bilingual Sentence Pairs from the Tatoeba Project (Good for Anki and Similar Flashcard

Applications) (manythings.org)

1. Perform required text pre-processing

2. Train, Validate, Test, and compare the two encoder-decoder models, each containing the following,

to translate English sentences to Hindi

a. LSTM cells

b. GRU

Use this documentation as a reference:

Character-level recurrent sequence-to-sequence model (keras.io)

**Code:**

# Imports

import numpy as np

import tensorflow as tf

keras = tf.keras

# Download dataset

!!curl -O http://www.manythings.org/anki/hin-eng.zip

!!unzip hin-eng.zip

batch\_size = 64

epochs = 100

latent\_dim = 256 # Latent dimensionality of the encoding space.

num\_samples = 10000

data\_path = "hin.txt"

# Q1) Preprocessing

input\_texts = []

target\_texts = []

input\_characters = set()

target\_characters = set()

with open(data\_path, "r", encoding="utf-8") as f:

lines = f.read().split("\n")

for line in lines[: min(num\_samples, len(lines) - 1)]:

input\_text, target\_text, \_ = line.split("\t")

# We use "tab" as the "start sequence" character

# for the targets, and "\n" as "end sequence" character.

target\_text = "\t" + target\_text + "\n"

input\_texts.append(input\_text)

target\_texts.append(target\_text)

for char in input\_text:

if char not in input\_characters:

input\_characters.add(char)

for char in target\_text:

if char not in target\_characters:

target\_characters.add(char)

input\_characters = sorted(list(input\_characters))

target\_characters = sorted(list(target\_characters))

num\_encoder\_tokens = len(input\_characters)

num\_decoder\_tokens = len(target\_characters)

max\_encoder\_seq\_length = max([len(txt) for txt in input\_texts])

max\_decoder\_seq\_length = max([len(txt) for txt in target\_texts])

print("Number of samples:", len(input\_texts))

print("Number of unique input tokens:", num\_encoder\_tokens)

print("Number of unique output tokens:", num\_decoder\_tokens)

print("Max sequence length for inputs:", max\_encoder\_seq\_length)

print("Max sequence length for outputs:", max\_decoder\_seq\_length)

input\_token\_index = dict([(char, i) for i, char in enumerate(input\_characters)])

target\_token\_index = dict([(char, i) for i, char in enumerate(target\_characters)])

encoder\_input\_data = np.zeros(

(len(input\_texts), max\_encoder\_seq\_length, num\_encoder\_tokens), dtype="float32"

)

decoder\_input\_data = np.zeros(

(len(input\_texts), max\_decoder\_seq\_length, num\_decoder\_tokens), dtype="float32"

)

decoder\_target\_data = np.zeros(

(len(input\_texts), max\_decoder\_seq\_length, num\_decoder\_tokens), dtype="float32"

)

for i, (input\_text, target\_text) in enumerate(zip(input\_texts, target\_texts)):

for t, char in enumerate(input\_text):

encoder\_input\_data[i, t, input\_token\_index[char]] = 1.0

encoder\_input\_data[i, t + 1 :, input\_token\_index[" "]] = 1.0

for t, char in enumerate(target\_text):

# decoder\_target\_data is ahead of decoder\_input\_data by one timestep

decoder\_input\_data[i, t, target\_token\_index[char]] = 1.0

if t > 0:

# decoder\_target\_data will be ahead by one timestep

# and will not include the start character.

decoder\_target\_data[i, t - 1, target\_token\_index[char]] = 1.0

decoder\_input\_data[i, t + 1 :, target\_token\_index[" "]] = 1.0

decoder\_target\_data[i, t:, target\_token\_index[" "]] = 1.0

# Q2a) LSTM

encoder\_inputs = keras.Input(shape=(None, num\_encoder\_tokens))

encoder = keras.layers.LSTM(latent\_dim, return\_state=True)

encoder\_outputs, state\_h, state\_c = encoder(encoder\_inputs)

encoder\_states = [state\_h, state\_c]

decoder\_inputs = keras.Input(shape=(None, num\_decoder\_tokens))

decoder\_lstm = keras.layers.LSTM(latent\_dim, return\_sequences=True, return\_state=True)

decoder\_outputs, \_, \_ = decoder\_lstm(decoder\_inputs, initial\_state=encoder\_states)

decoder\_dense = keras.layers.Dense(num\_decoder\_tokens, activation="softmax")

decoder\_outputs = decoder\_dense(decoder\_outputs)

model = keras.Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

model.compile(

optimizer="rmsprop", loss="categorical\_crossentropy", metrics=["accuracy"]

)

model.fit(

[encoder\_input\_data, decoder\_input\_data],

decoder\_target\_data,

batch\_size=batch\_size,

epochs=epochs,

validation\_split=0.2,

)

encoder\_inputs = model.input[0]

encoder\_outputs, state\_h\_enc, state\_c\_enc = model.layers[2].output

encoder\_states = [state\_h\_enc, state\_c\_enc]

encoder\_model = keras.Model(encoder\_inputs, encoder\_states)

decoder\_inputs = model.input[1]

decoder\_state\_input\_h = keras.Input(shape=(latent\_dim,))

decoder\_state\_input\_c = keras.Input(shape=(latent\_dim,))

decoder\_states\_inputs = [decoder\_state\_input\_h, decoder\_state\_input\_c]

decoder\_lstm = model.layers[3]

decoder\_outputs, state\_h\_dec, state\_c\_dec = decoder\_lstm(

decoder\_inputs, initial\_state=decoder\_states\_inputs

)

decoder\_states = [state\_h\_dec, state\_c\_dec]

decoder\_dense = model.layers[4]

decoder\_outputs = decoder\_dense(decoder\_outputs)

decoder\_model = keras.Model(

[decoder\_inputs] + decoder\_states\_inputs, [decoder\_outputs] + decoder\_states

)

reverse\_input\_char\_index = dict((i, char) for char, i in input\_token\_index.items())

reverse\_target\_char\_index = dict((i, char) for char, i in target\_token\_index.items())

def decode\_sequence(input\_seq):

states\_value = encoder\_model.predict(input\_seq)

target\_seq = np.zeros((1, 1, num\_decoder\_tokens))

target\_seq[0, 0, target\_token\_index["\t"]] = 1.0

stop\_condition = False

decoded\_sentence = ""

while not stop\_condition:

output\_tokens, h, c = decoder\_model.predict([target\_seq] + states\_value)

sampled\_token\_index = np.argmax(output\_tokens[0, -1, :])

sampled\_char = reverse\_target\_char\_index[sampled\_token\_index]

decoded\_sentence += sampled\_char

if sampled\_char == "\n" or len(decoded\_sentence) > max\_decoder\_seq\_length:

stop\_condition = True

target\_seq = np.zeros((1, 1, num\_decoder\_tokens))

target\_seq[0, 0, sampled\_token\_index] = 1.0

states\_value = [h, c]

return decoded\_sentence

for seq\_index in range(20):

input\_seq = encoder\_input\_data[seq\_index : seq\_index + 1]

decoded\_sentence = decode\_sequence(input\_seq)

print("-")

print("Input sentence:", input\_texts[seq\_index])

print("Decoded sentence:", decoded\_sentence)

# Q2b) GRU

encoder\_inputs = keras.Input(shape=(None, num\_encoder\_tokens))

encoder = keras.layers.GRU(latent\_dim, return\_state=True)

encoder\_outputs, state\_c = encoder(encoder\_inputs)

encoder\_states = [state\_c]

decoder\_inputs = keras.Input(shape=(None, num\_decoder\_tokens))

decoder\_gru = keras.layers.GRU(latent\_dim, return\_sequences=True, return\_state=True)

decoder\_outputs, \_ = decoder\_gru(decoder\_inputs, initial\_state=encoder\_states)

decoder\_dense = keras.layers.Dense(num\_decoder\_tokens, activation="softmax")

decoder\_outputs = decoder\_dense(decoder\_outputs)

model = keras.Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

model.summary()

model.compile(

optimizer="rmsprop", loss="categorical\_crossentropy", metrics=["accuracy"]

)

model.fit(

[encoder\_input\_data, decoder\_input\_data],

decoder\_target\_data,

batch\_size=batch\_size,

epochs=epochs,

validation\_split=0.2,

)

encoder\_inputs = model.input[0]

encoder\_outputs, state\_c\_enc = model.layers[2].output

encoder\_states = [state\_c\_enc]

encoder\_model = keras.Model(encoder\_inputs, encoder\_states)

decoder\_inputs = model.input[1]

decoder\_state\_input\_c = keras.Input(shape=(latent\_dim,))

decoder\_states\_inputs = [decoder\_state\_input\_c]

decoder\_gru = model.layers[3]

decoder\_outputs, state\_c\_dec = decoder\_gru(

decoder\_inputs, initial\_state=decoder\_states\_inputs

)

decoder\_states = [state\_c\_dec]

decoder\_dense = model.layers[4]

decoder\_outputs = decoder\_dense(decoder\_outputs)

decoder\_model = keras.Model(

[decoder\_inputs] + decoder\_states\_inputs, [decoder\_outputs] + decoder\_states

)

reverse\_input\_char\_index = dict((i, char) for char, i in input\_token\_index.items())

reverse\_target\_char\_index = dict((i, char) for char, i in target\_token\_index.items())

def decode\_sequence(input\_seq):

states\_value = encoder\_model.predict(input\_seq)

# states\_value = np.reshape(states\_value, (1, 1, 1, 90))

target\_seq = np.zeros((1, 1, num\_decoder\_tokens))

target\_seq[0, 0, target\_token\_index["\t"]] = 1.0

stop\_condition = False

decoded\_sentence = ""

print("hello")

while not stop\_condition:

output\_tokens, c = decoder\_model.predict([target\_seq])

sampled\_token\_index = np.argmax(output\_tokens[0, -1, :])

sampled\_char = reverse\_target\_char\_index[sampled\_token\_index]

decoded\_sentence += sampled\_char

if sampled\_char == "\n" or len(decoded\_sentence) > max\_decoder\_seq\_length:

stop\_condition = True

target\_seq = np.zeros((1, 1, num\_decoder\_tokens))

print(target\_seq.shape)

target\_seq[0, 0, sampled\_token\_index] = 1.0

target\_seq = np.reshape(target\_seq, (1, 1, num\_decoder\_tokens))

states\_value = [c]

return decoded\_sentence

for seq\_index in range(20):

input\_seq = encoder\_input\_data[seq\_index : seq\_index + 1]

decoded\_sentence = decode\_sequence(input\_seq)

print(decoded\_sentence)

print("-")

print("Input sentence:", input\_texts[seq\_index])

print("Decoded sentence:", decoded\_sentence)

**Results & Discussion:**

# Translation outputs

Input sentence: Wow!

Decoded sentence: मैं तुम्हें किता करता हूँ।

Input sentence: Duck!

Decoded sentence: मैं तुम्हें किता करता हूँ।

Input sentence: Help!

Decoded sentence: मैं तुम्हें किता करता हूँ।

Input sentence: Jump.

Decoded sentence: मैं तुम्हें किता करता हूँ।

Input sentence: We won.

Decoded sentence: मैं तुम्हें किता करता हूँ।

GRU performed better than LSTM with better BLEU scores

# WEEK 9 - Neural Machine Translation (NMT) using Encoder-Decoder with attention and transformer mechanism.

**Q1)** Using the following NMT repo:

Tab-delimited Bilingual Sentence Pairs from the Tatoeba Project (Good for Anki and Similar Flashcard

Applications) (manythings.org)

1. Perform required text pre-processing

2. Train and test the encode-decoder model with attention mechanism.

3. Use any pre-trained transformer model , fine tune it on given dataset for language translation tasks.

4. Compare the performance of model defined in (2) and (3).

**Code:**

# Imports

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.utils import shuffle

import string

from string import digits

import re

import tensorflow as tf

keras = tf.keras

from keras import Sequential

from keras.layers import Dense, LSTM, GRU, Embedding, Dropout, Input, Concatenate, TimeDistributed

from keras.models import Model

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from keras.preprocessing.text import Tokenizer

import random

# Q1) Preprocessing

uncleaned\_data\_list = data.split('\n')

uncleaned\_data\_list.pop()

len(uncleaned\_data\_list)

english\_word = []

hindi\_word = []

cleaned\_data\_list = []

for word in uncleaned\_data\_list:

split = word.split('\t')

english\_word.append(split[0])

hindi\_word.append(split[1])

language\_data = pd.DataFrame(columns=['English','Hindi'])

language\_data['English'] = english\_word

language\_data['Hindi'] = hindi\_word

language\_data.to\_csv('language\_data.csv', index=False)

english\_text = language\_data['English'].values

hindi\_text = language\_data['Hindi'].values

len(english\_text), len(hindi\_text)

english\_text\_ = [x.lower() for x in english\_text]

hindi\_text\_ = [x.lower() for x in hindi\_text]

english\_text\_ = [re.sub("'",'',x) for x in english\_text\_]

hindi\_text\_ = [re.sub("'",'',x) for x in hindi\_text\_]

def remove\_punc(text\_list):

table = str.maketrans('', '', string.punctuation)

removed\_punc\_text = []

for sent in text\_list:

sentance = [w.translate(table) for w in sent.split(' ')]

removed\_punc\_text.append(' '.join(sentance))

return removed\_punc\_text

english\_text\_ = remove\_punc(english\_text\_)

hindi\_text\_ = remove\_punc(hindi\_text\_)

remove\_digits = str.maketrans('', '', digits)

removed\_digits\_text = []

for sent in english\_text\_:

sentance = [w.translate(remove\_digits) for w in sent.split(' ')]

removed\_digits\_text.append(' '.join(sentance))

english\_text\_ = removed\_digits\_text

hindi\_text\_ = [re.sub("[२३०८१५७९४६]","",x) for x in hindi\_text\_]

hindi\_text\_ = [re.sub("[\u200d]","",x) for x in hindi\_text\_]

english\_text\_ = [x.strip() for x in english\_text\_]

hindi\_text\_ = [x.strip() for x in hindi\_text\_]

hindi\_text\_ = ["start " + x + " end" for x in hindi\_text\_]

hindi\_text\_[0], english\_text\_[0]

X = english\_text\_

Y = hindi\_text\_

X\_train, X\_test, y\_train, y\_test=train\_test\_split(X,Y,test\_size=0.1)

def max\_length(data):

max\_length\_ = max([len(x.split(' ')) for x in data])

return max\_length\_

max\_length\_english = max\_length(X\_train)

max\_length\_hindi = max\_length(y\_train)

max\_length\_english\_test = max\_length(X\_test)

max\_length\_hindi\_test = max\_length(y\_test)

max\_length\_hindi, max\_length\_english

# Tokenize

englishTokenizer = Tokenizer()

englishTokenizer.fit\_on\_texts(X\_train)

Eword2index = englishTokenizer.word\_index

vocab\_size\_source = len(Eword2index) + 1

X\_train = englishTokenizer.texts\_to\_sequences(X\_train)

X\_train = pad\_sequences(X\_train, maxlen=max\_length\_english, padding='post')

X\_test = englishTokenizer.texts\_to\_sequences(X\_test)

X\_test = pad\_sequences(X\_test, maxlen = max\_length\_english, padding='post')

hindiTokenizer = Tokenizer()

hindiTokenizer.fit\_on\_texts(y\_train)

Hword2index = hindiTokenizer.word\_index

vocab\_size\_target = len(Hword2index) + 1

y\_train = hindiTokenizer.texts\_to\_sequences(y\_train)

y\_train = pad\_sequences(y\_train, maxlen=max\_length\_hindi, padding='post')

y\_test = hindiTokenizer.texts\_to\_sequences(y\_test)

y\_test = pad\_sequences(y\_test, maxlen = max\_length\_hindi, padding='post')

vocab\_size\_source, vocab\_size\_target

X\_train = np.array(X\_train)

y\_train = np.array(y\_train)

X\_test = np.array(X\_test)

y\_test = np.array(y\_test)

# Q2) Transformer

import tensorflow as tf

from tensorflow.python.keras import backend as K

logger = tf.get\_logger()

class AttentionLayer(tf.keras.layers.Layer):

def \_\_init\_\_(self, \*\*kwargs):

super(AttentionLayer, self).\_\_init\_\_(\*\*kwargs)

def build(self, input\_shape):

assert isinstance(input\_shape, list)

# Create a trainable weight variable for this layer.

self.W\_a = self.add\_weight(

name="W\_a",

shape=tf.TensorShape((input\_shape[0][2], input\_shape[0][2])),

initializer="uniform",

trainable=True,

)

self.U\_a = self.add\_weight(

name="U\_a",

shape=tf.TensorShape((input\_shape[1][2], input\_shape[0][2])),

initializer="uniform",

trainable=True,

)

self.V\_a = self.add\_weight(

name="V\_a",

shape=tf.TensorShape((input\_shape[0][2], 1)),

initializer="uniform",

trainable=True,

)

super(AttentionLayer, self).build(

input\_shape

) # Be sure to call this at the end

def call(self, inputs):

"""

inputs: [encoder\_output\_sequence, decoder\_output\_sequence]

"""

assert type(inputs) == list

encoder\_out\_seq, decoder\_out\_seq = inputs

logger.debug(f"encoder\_out\_seq.shape = {encoder\_out\_seq.shape}")

logger.debug(f"decoder\_out\_seq.shape = {decoder\_out\_seq.shape}")

def energy\_step(inputs, states):

"""Step function for computing energy for a single decoder state

inputs: (batchsize \* 1 \* de\_in\_dim)

states: (batchsize \* 1 \* de\_latent\_dim)

"""

logger.debug("Running energy computation step")

if not isinstance(states, (list, tuple)):

raise TypeError(

f"States must be an iterable. Got {states} of type {type(states)}"

)

encoder\_full\_seq = states[-1]

""" Computing S.Wa where S=[s0, s1, ..., si]"""

# <= batch size \* en\_seq\_len \* latent\_dim

W\_a\_dot\_s = K.dot(encoder\_full\_seq, self.W\_a)

""" Computing hj.Ua """

U\_a\_dot\_h = K.expand\_dims(

K.dot(inputs, self.U\_a), 1

) # <= batch\_size, 1, latent\_dim

logger.debug(f"U\_a\_dot\_h.shape = {U\_a\_dot\_h.shape}")

""" tanh(S.Wa + hj.Ua) """

# <= batch\_size\*en\_seq\_len, latent\_dim

Ws\_plus\_Uh = K.tanh(W\_a\_dot\_s + U\_a\_dot\_h)

logger.debug(f"Ws\_plus\_Uh.shape = {Ws\_plus\_Uh.shape}")

""" softmax(va.tanh(S.Wa + hj.Ua)) """

# <= batch\_size, en\_seq\_len

e\_i = K.squeeze(K.dot(Ws\_plus\_Uh, self.V\_a), axis=-1)

# <= batch\_size, en\_seq\_len

e\_i = K.softmax(e\_i)

logger.debug(f"ei.shape = {e\_i.shape}")

return e\_i, [e\_i]

def context\_step(inputs, states):

"""Step function for computing ci using ei"""

logger.debug("Running attention vector computation step")

if not isinstance(states, (list, tuple)):

raise TypeError(

f"States must be an iterable. Got {states} of type {type(states)}"

)

encoder\_full\_seq = states[-1]

# <= batch\_size, hidden\_size

c\_i = K.sum(encoder\_full\_seq \* K.expand\_dims(inputs, -1), axis=1)

logger.debug(f"ci.shape = {c\_i.shape}")

return c\_i, [c\_i]

# we don't maintain states between steps when computing attention

# attention is stateless, so we're passing a fake state for RNN step function

fake\_state\_c = K.sum(encoder\_out\_seq, axis=1)

fake\_state\_e = K.sum(

encoder\_out\_seq, axis=2

) # <= (batch\_size, enc\_seq\_len, latent\_dim

""" Computing energy outputs """

# e\_outputs => (batch\_size, de\_seq\_len, en\_seq\_len)

last\_out, e\_outputs, \_ = K.rnn(

energy\_step, decoder\_out\_seq, [fake\_state\_e], constants=[encoder\_out\_seq]

)

""" Computing context vectors """

last\_out, c\_outputs, \_ = K.rnn(

context\_step, e\_outputs, [fake\_state\_c], constants=[encoder\_out\_seq]

)

return c\_outputs, e\_outputs

def compute\_output\_shape(self, input\_shape):

"""Outputs produced by the layer"""

return [

tf.TensorShape((input\_shape[1][0], input\_shape[1][1], input\_shape[1][2])),

tf.TensorShape((input\_shape[1][0], input\_shape[1][1], input\_shape[0][1])),

]

from keras import backend as K

K.clear\_session()

latent\_dim = 500

encoder\_inputs = Input(shape=(max\_length\_english,))

enc\_emb = Embedding(vocab\_size\_source, latent\_dim,trainable=True)(encoder\_inputs)

encoder\_lstm1 = LSTM(latent\_dim,return\_sequences=True,return\_state=True)

encoder\_output1, state\_h1, state\_c1 = encoder\_lstm1(enc\_emb)

encoder\_lstm2 = LSTM(latent\_dim,return\_sequences=True,return\_state=True)

encoder\_output2, state\_h2, state\_c2 = encoder\_lstm2(encoder\_output1)

encoder\_lstm3=LSTM(latent\_dim, return\_state=True, return\_sequences=True)

encoder\_outputs, state\_h, state\_c= encoder\_lstm3(encoder\_output2)

decoder\_inputs = Input(shape=(None,))

dec\_emb\_layer = Embedding(vocab\_size\_target, latent\_dim,trainable=True)

dec\_emb = dec\_emb\_layer(decoder\_inputs)

decoder\_lstm = LSTM(latent\_dim, return\_sequences=True, return\_state=True)

decoder\_outputs,decoder\_fwd\_state, decoder\_back\_state = decoder\_lstm(dec\_emb,initial\_state=[state\_h, state\_c])

attn\_layer = AttentionLayer()

attn\_out, attn\_states = attn\_layer([encoder\_outputs, decoder\_outputs])

decoder\_concat\_input = Concatenate(axis=-1, name='concat\_layer')([decoder\_outputs, attn\_out])

decoder\_dense = TimeDistributed(Dense(vocab\_size\_target, activation='softmax'))

decoder\_outputs = decoder\_dense(decoder\_concat\_input)

model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

model.summary()

model.compile(optimizer='rmsprop',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

history = model.fit(

[X\_train, y\_train[:, :-1]],

y\_train.reshape(y\_train.shape[0], y\_train.shape[1], 1)[:, 1:],

epochs=50,

batch\_size=512,

validation\_data=(

[X\_test, y\_test[:, :-1]],

y\_test.reshape(y\_test.shape[0], y\_test.shape[1], 1)[:, 1:],

),

)

# Plot

from matplotlib import pyplot

pyplot.plot(history.history['loss'], label='train')

pyplot.plot(history.history['val\_loss'], label='test')

pyplot.legend()

pyplot.show()

latent\_dim=500

encoder\_inputs = model.input[0]

encoder\_outputs, state\_h, state\_c = model.layers[6].output

encoder\_model = Model(inputs=encoder\_inputs,outputs=[encoder\_outputs, state\_h, state\_c])

decoder\_state\_input\_h = Input(shape=(latent\_dim,))

decoder\_state\_input\_c = Input(shape=(latent\_dim,))

decoder\_hidden\_state\_input = Input(shape=(22,latent\_dim))

decoder\_inputs = model.layers[3].output

dec\_emb\_layer = model.layers[5]

dec\_emb2= dec\_emb\_layer(decoder\_inputs)

decoder\_lstm = model.layers[7]

decoder\_outputs2, state\_h2, state\_c2 = decoder\_lstm(dec\_emb2, initial\_state=[decoder\_state\_input\_h, decoder\_state\_input\_c])

attn\_layer = model.layers[8]

attn\_out\_inf, attn\_states\_inf = attn\_layer([decoder\_hidden\_state\_input, decoder\_outputs2])

concate = model.layers[9]

decoder\_inf\_concat = concate([decoder\_outputs2, attn\_out\_inf])

decoder\_dense = model.layers[10]

decoder\_outputs2 = decoder\_dense(decoder\_inf\_concat)

decoder\_model = Model(

[decoder\_inputs] + [decoder\_hidden\_state\_input,decoder\_state\_input\_h, decoder\_state\_input\_c],

[decoder\_outputs2] + [state\_h2, state\_c2])

Eindex2word = englishTokenizer.index\_word

Hindex2word = hindiTokenizer.index\_word

def decode\_sequence(input\_seq):

e\_out, e\_h, e\_c = encoder\_model.predict(input\_seq)

target\_seq = np.zeros((1, 1))

target\_seq[0, 0] = Hword2index['start']

stop\_condition = False

decoded\_sentence = ''

while not stop\_condition:

output\_tokens, h, c = decoder\_model.predict([target\_seq] + [e\_out, e\_h, e\_c])

sampled\_token\_index = np.argmax(output\_tokens[0, -1, :])

if sampled\_token\_index == 0:

break

else:

sampled\_token = Hindex2word[sampled\_token\_index]

if sampled\_token != 'end':

decoded\_sentence += ' ' + sampled\_token

if sampled\_token == 'end' or len(decoded\_sentence.split()) >= (26-1):

stop\_condition = True

target\_seq = np.zeros((1, 1))

target\_seq[0, 0] = sampled\_token\_index

e\_h, e\_c = h, c

return decoded\_sentence

def seq2summary(input\_seq):

newString = ""

for i in input\_seq:

if (i != 0 and i != Hword2index["start"]) and i != Hword2index["end"]:

newString = newString + Hindex2word[i] + " "

return newString

def seq2text(input\_seq):

newString = ""

for i in input\_seq:

if i != 0:

newString = newString + Eindex2word[i] + " "

return newString

# Predictions

for i in range(10):

print("Review:", seq2text(X\_test[i]))

print("Original summary:", seq2summary(y\_test[i]))

print(X\_test[i].shape)

print("Predicted summary:", decode\_sequence(X\_test[i].reshape(1, 22)))

print("\n")

# Q3) Pretrained transformer

import transformers

MODEL\_CHECKPOINT = "t5-small"

from transformers import AutoTokenizer, TFAutoModelForSeq2SeqLM

from transformers import DataCollatorForSeq2Seq

model\_name = 't5-small'

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

model = TFAutoModelForSeq2SeqLM.from\_pretrained(model\_name)

train\_encodings = tokenizer(list(language\_data["English"]), text\_target=list(language\_data["Hindi"]), truncation=True, padding=True, max\_length=200)

data\_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=model\_name, return\_tensors="tf")

def postprocess\_text(preds, labels):

preds = [pred.strip() for pred in preds]

labels = [[label.strip()] for label in labels]

return preds, labels

def compute\_metrics(eval\_preds):

preds, labels = eval\_preds

if isinstance(preds, tuple):

preds = preds[0]

decoded\_preds = tokenizer.batch\_decode(preds, skip\_special\_tokens=True)

labels = np.where(labels != -100, labels, tokenizer.pad\_token\_id)

decoded\_labels = tokenizer.batch\_decode(labels, skip\_special\_tokens=True)

decoded\_preds, decoded\_labels = postprocess\_text(decoded\_preds, decoded\_labels)

result = metric.compute(predictions=decoded\_preds, references=decoded\_labels)

result = {"bleu": result["score"]}

prediction\_lens = [np.count\_nonzero(pred != tokenizer.pad\_token\_id) for pred in preds]

result["gen\_len"] = np.mean(prediction\_lens)

result = {k: round(v, 4) for k, v in result.items()}

return result

from transformers import AdamWeightDecay

optimizer = AdamWeightDecay(learning\_rate=2e-5, weight\_decay\_rate=0.01)

from transformers import TFAutoModelForSeq2SeqLM

model = TFAutoModelForSeq2SeqLM.from\_pretrained(model\_name)

tf\_train\_set = model.prepare\_tf\_dataset(

train\_encodings,

shuffle=True,

batch\_size=16,

collate\_fn=data\_collator,

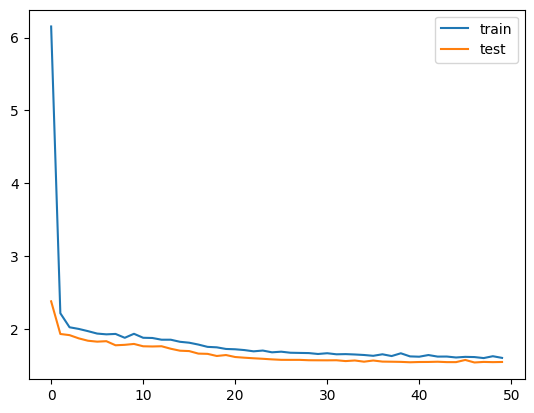
)

model.compile(optimizer=optimizer)

model.fit(x=tf\_train\_set, epochs=3)

**Results & Discussion:**

# Loss plot



# Translation predictions

Review: this box contains five apples

Original summary: इस डब्बे में पाँच सेव हैं।

Predicted summary: तुम एक में

Review: they are the problem

Original summary: वे इस समस्या के बारे में बातचीत कर रहे हैं।

Predicted summary: क्या क्या

The pretrained transformer got higher BLEU scores than the trained transformer and hence performed better.

WEEK 10 – Autoencoders

**Q1)** Download the pins-face-recognition dataset from the given link

https://www.kaggle.com/datasets/hereisburak/pins-face-recognition and perform the following

tasks.

1.Preprocess the data, ensuring it is suitable for training an autoencoder.

2.Design an autoencoder architecture for image reconstruction. Train the model on the preprocessed

dataset and evaluate the model's performance in terms of image reconstruction.

3. Visualize original images and their reconstructed counterparts to assess the quality of reconstruction.

4. Introduce noise to the images to create a noisy dataset.

5. Train the Autoencoder model using the noisy images as input and the clean images as target output.

Evaluate the model's performance in terms of noise reduction.

6. Visualize noisy images, their denoised counterparts, and the original clean images to observe the

denoising effect of the model.

**Code:**

# Imports

from keras.layers import Dense, Input, Conv2D, LSTM, MaxPool2D, UpSampling2D

from sklearn.model\_selection import train\_test\_split

from keras.callbacks import EarlyStopping

from keras.utils import to\_categorical

from numpy import argmax, array\_equal

import matplotlib.pyplot as plt

from keras.models import Model

from imgaug import augmenters

from random import randint

import pandas as pd

import numpy as np

import os

from tensorflow.keras.preprocessing.image import array\_to\_img, img\_to\_array, load\_img

import random

from sklearn.model\_selection import train\_test\_split

# Q1) Dataset preparation

images = []

i=0

for fname in os.listdir('./Images'):

img = load\_img(f'./Images/{fname}')

img=img.resize([100,100])

images.append(img\_to\_array(img))

images=np.array(images)

images=images/255.00

train\_x,val\_x=train\_test\_split(images,test\_size=0.2)

for i in range(0,5):

plt.imshow(random.choice(images))

plt.show()

train\_x.shape

val\_x.shape

train\_x=train\_x.reshape([14027,size])

val\_x=val\_x.reshape([3507,size])

# Q2) Autoencoder

input\_layer = Input(shape=(size))

## encoding architecture

encode\_layer1 = Dense(1500, activation='relu')(input\_layer)

encode\_layer2 = Dense(1000, activation='relu')(encode\_layer1)

encode\_layer3 = Dense(500, activation='relu')(encode\_layer2)

## latent view

latent\_view = Dense(10, activation='sigmoid')(encode\_layer3)

## decoding architecture

decode\_layer1 = Dense(500, activation='relu')(latent\_view)

decode\_layer2 = Dense(1000, activation='relu')(decode\_layer1)

decode\_layer3 = Dense(1500, activation='relu')(decode\_layer2)

## output layer

output\_layer = Dense(size)(decode\_layer3)

model = Model(input\_layer, output\_layer)

model.compile(optimizer='adam', loss='mse')

model.fit(train\_x, train\_x, epochs=1, batch\_size=2048,validation\_data=(val\_x,val\_x) )

preds = model.predict(val\_x)

# Q3) Visualize images

from PIL import Image

f, ax = plt.subplots(1,5)

f.set\_size\_inches(80, 40)

for i in range(5):

ax[i].imshow(val\_x[i].reshape(100, 100,3))

plt.show()

f, ax = plt.subplots(1,5)

f.set\_size\_inches(80, 40)

for i in range(5):

ax[i].imshow(preds[i].reshape(100, 100,3))

plt.show()

# Q4) Image noise

noise = augmenters.SaltAndPepper(0.1)

seq\_object = augmenters.Sequential([noise])

early\_stopping = EarlyStopping(monitor='val\_loss', min\_delta=0, patience=10, verbose=1, mode='auto')

train\_x\_n = seq\_object.augment\_images(train\_x \* 255) / 255

val\_x\_n = seq\_object.augment\_images(val\_x \* 255) / 255

f, ax = plt.subplots(1,5)

f.set\_size\_inches(80, 40)

for i in range(5,10):

ax[i-5].imshow(train\_x[i].reshape(100, 100, 3))

plt.show()

f, ax = plt.subplots(1,5)

f.set\_size\_inches(80, 40)

for i in range(5,10):

ax[i-5].imshow(train\_x\_n[i].reshape(100, 100, 3))

plt.show()

train\_x = train\_x.reshape(-1, 100, 100, 3)

train\_x\_n = train\_x\_n.reshape(-1, 100, 100, 3)

# Q5) Model

# input layer

input\_layer = Input(shape=(100, 100, 3))

# encoding architecture

encoded\_layer1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input\_layer) #shape = (28, 28, 64)

encoded\_layer1 = MaxPool2D( (2, 2), padding='same')(encoded\_layer1) #Shape: (14, 14, 64)

encoded\_layer2 = Conv2D(16, (3, 3), activation='relu', padding='same')(encoded\_layer1) #Shape: (14, 14, 32)

encoded\_layer2 = MaxPool2D( (2, 2), padding='same')(encoded\_layer2) #Shape: (7, 7, 32)

encoded\_layer3 = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded\_layer2) #Shape: (7, 7, 16)

latent\_view = MaxPool2D( (2, 2), padding='same')(encoded\_layer3) #Shape: (4, 4, 16)

# decoding architecture

decoded\_layer1 = Conv2D(8, (3, 3), activation='relu', padding='same')(latent\_view) #Shape: (4, 4, 16)

decoded\_layer1 = UpSampling2D((2, 2))(decoded\_layer1) #Shape: (8, 8, 16) (due to upsampling with a 2x2 size)

decoded\_layer2 = Conv2D(16, (3, 3), activation='relu', padding='same')(decoded\_layer1) #Shape: (8, 8, 32)

decoded\_layer2 = UpSampling2D((2, 2))(decoded\_layer2) #Shape: (16, 16, 32) (due to upsampling with a 2x2 size)

decoded\_layer3 = Conv2D(32, (3, 3), activation='relu')(decoded\_layer2) #Shape: (14, 14, 64) (due to the default padding='valid' in Conv2D)

decoded\_layer3 = UpSampling2D((2, 2))(decoded\_layer3) #Shape: (28, 28, 64) (due to upsampling with a 2x2 size)

output\_layer = Conv2D(3, (3, 3), padding='same')(decoded\_layer3) #Shape: (28, 28, 1) (output channels represent the reconstructed image)

# compile the model

model\_2 = Model(input\_layer, output\_layer)

model\_2.compile(optimizer='adam', loss='mse')

history = model\_2.fit(train\_x\_n, train\_x, epochs=15, batch\_size=32, validation\_data=(val\_x\_n, val\_x))

# Q6) Visualize

preds = model\_2.predict(train\_x\_n[:10])

f, ax = plt.subplots(1,5)

f.set\_size\_inches(80, 40)

for i in range(5,10):

ax[i-5].imshow(preds[i].reshape(100, 100, 3))

plt.show()

**Results & Discussion:**

# Reconstruction

# Original

A person with his hand on his mouth

Description automatically generated

# Reconstructed

A blurry image of a person's face

Description automatically generated

# Denoising

# Original

A close up of a person's face

Description automatically generated

# Noisy

A person with dark hair

Description automatically generated

# Reconstructed

A blurry image of a person's face

Description automatically generated

# WEEK 11 – GAN’s

**Q1)** Consider the Fashion MNIST dataset as an input perform the following tasks

1.Build the GAN model to generate a new dataset.

2. Experiment with 100, 150, 200 epochs, latent dimensions as 20, 50, 100 and visualize the impact of

changing these hyperparameters on the quality of generated image.

3. Plot the generated image and save them as a new dataset.

**Code:**

# Imports

import numpy as np

import random

import time

import cv2

import tensorflow as tf

from tensorflow.keras.datasets import fashion\_mnist

import tensorflow.keras.backend as K

from tensorflow.keras import Sequential

from tensorflow.keras import Model, callbacks

from tensorflow.keras.layers import Dense, Flatten, Reshape, UpSampling2D,BatchNormalization, Dropout, Conv2D, Conv2DTranspose, LeakyReLU

from tensorflow.keras.losses import BinaryCrossentropy

from tensorflow.keras.utils import image\_dataset\_from\_directory

from tensorflow.keras.layers.experimental.preprocessing import Rescaling

import matplotlib.pyplot as plt

import plotly.express as px

import warnings

warnings.filterwarnings('ignore')

# Q1) Preprocessing and model

(x\_train, y\_train), (x\_test, y\_test) = fashion\_mnist.load\_data()

BUFFER\_SIZE = x\_train.shape[0]

BATCH\_SIZE = 500

BUFFER\_SIZE = BUFFER\_SIZE // BATCH\_SIZE \* BATCH\_SIZE

x\_train = x\_train[:BUFFER\_SIZE]

y\_train = y\_train[:BUFFER\_SIZE]

print(f'Input data shape: {x\_train.shape}')

print(f'Number of labels: {y\_train.size}')

x\_train = x\_train / 255

x\_test = x\_test / 255

x\_train = np.reshape(x\_train, (len(x\_train), 28, 28, 1))

x\_test = np.reshape(x\_test, (len(x\_test), 28, 28, 1))

train\_dataset = tf.data.Dataset.from\_tensor\_slices(x\_train).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

# Plot images

cnt\_imgs = 32

counter = 0

IMAGE\_SIZE = (64, 64)

plt.figure(figsize=(28, 10))

for img in x\_train:

plt.subplot(4, 8, counter + 1)

draw\_img = cv2.resize(img, IMAGE\_SIZE)

plt.imshow(draw\_img, cmap='gray')

plt.axis('off')

counter += 1

if counter == cnt\_imgs:

break

# Model

cross\_entropy = BinaryCrossentropy(from\_logits=True)

def generator\_loss(fake\_output):

"""Function of finding the value of the loss function for the generator (for BATCH)"""

loss = cross\_entropy(tf.ones\_like(fake\_output), fake\_output)

return loss

def discriminator\_loss(real\_output, fake\_output):

"""Function of finding the value of the loss function for the discriminator (for BATCH)"""

real\_loss = cross\_entropy(tf.ones\_like(real\_output), real\_output)

fake\_loss = cross\_entropy(tf.zeros\_like(fake\_output), fake\_output)

total\_loss = real\_loss + fake\_loss

return total\_loss

generator\_optimizer = tf.optimizers.Adam(1e-4)

discriminator\_optimizer = tf.optimizers.Adam(1e-4)

hidden\_dim = 20

generator = Sequential(name='generator')

generator.add(Dense(7 \* 7 \* 256, activation='relu', input\_shape=(hidden\_dim, )))

generator.add(BatchNormalization())

generator.add(Reshape((7, 7, 256)))

generator.add(Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', activation='relu'))

generator.add(BatchNormalization())

generator.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', activation='relu'))

generator.add(BatchNormalization())

generator.add(Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', activation='sigmoid'))

discriminator = Sequential()

discriminator.add(Conv2D(64, (5,5), strides = (2,2), padding='same', input\_shape=(28,28,1,)))

discriminator.add(LeakyReLU(0.2))

discriminator.add(Dropout(0.3))

discriminator.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))

discriminator.add(LeakyReLU(0.2))

discriminator.add(Dropout(0.3))

discriminator.add(Flatten())

discriminator.add(Dense(1, activation='sigmoid'))

generator.compile(optimizer=generator\_optimizer, loss='binary\_crossentropy')

discriminator.compile(optimizer=discriminator\_optimizer, loss='binary\_crossentropy')

@tf.function

def train\_step(images) -> (float, float):

"""Function for updating weight coefficients at one training step (for one BATCH)"""

noise = tf.random.normal([BATCH\_SIZE, hidden\_dim])

with tf.GradientTape() as gen\_tape, tf.GradientTape() as disc\_tape:

generated\_images = generator(noise, training=True)

real\_output = discriminator(images, training=True)

fake\_output = discriminator(generated\_images, training=True)

gen\_loss = generator\_loss(fake\_output)

disc\_loss = discriminator\_loss(real\_output, fake\_output)

generator\_gradients = gen\_tape.gradient(gen\_loss, generator.trainable\_variables)

discriminator\_gradients = disc\_tape.gradient(disc\_loss, discriminator.trainable\_variables)

generator\_optimizer.apply\_gradients(zip(generator\_gradients, generator.trainable\_variables))

discriminator\_optimizer.apply\_gradients(zip(discriminator\_gradients, discriminator.trainable\_variables))

return gen\_loss, disc\_loss

def train(dataset, epochs) -> None:

"""A function to start the learning process for all epochs for the generator and discriminator

dataset: a set of real images that we store in generator"""

history = []

max\_print\_label = 10

th = BUFFER\_SIZE // (BATCH\_SIZE \* max\_print\_label)

for epoch in range(1, EPOCHS + 1):

print(f'{epoch}/{EPOCHS}: ', end='')

start = time.time()

n = 0

gen\_loss\_epoch = 0

for image\_batch in dataset:

gen\_loss, disc\_loss = train\_step(image\_batch)

gen\_loss\_epoch += K.mean(gen\_loss)

if (n % th == 0):

print('=', end='')

n += 1

print('>', end = ' ')

history += [gen\_loss\_epoch / n]

print(': loss = ' + str(history[-1]))

print(f'The time of the epoch {epoch} is: {time.time() - start} second')

return history

# 100 epochs

EPOCHS = 100

gen\_history = train(train\_dataset, EPOCHS)

plt.figure(figsize=(24, 10))

plt.title('Visualization of the generator learning process', fontsize=16)

plt.plot(np.arange(1, EPOCHS + 1), gen\_history)

plt.xlabel('Epochs', fontsize=16)

plt.ylabel('Loss', fontsize=16)

plt.xticks(fontsize=16)

plt.yticks(fontsize=16)

plt.xlim(1, EPOCHS)

plt.grid()

plt.legend(fontsize=16)

plt.show()

n = 4

total = 2 \* n + 1

cnter = 0

plt.figure(figsize=(28, 8))

num = 1

for i in range(-n, n + 1):

for j in range(-n, n + 1):

ax = plt.subplot(7, 11, num)

num += 1

ip\_data = np.array([0.5\*i/n for \_ in range(20)])

ip\_data = np.expand\_dims(ip\_data,axis=0)

img = generator.predict(ip\_data)

plt.imshow(img[0, :, :, 0], cmap='gray')

plt.axis('off')

if num == 78:

break

plt.show()

# 150 epochs

EPOCHS = 50 # 150

gen\_history = train(train\_dataset, EPOCHS)

plt.figure(figsize=(24, 10))

plt.title('Visualization of the generator learning process', fontsize=16)

plt.plot(np.arange(1, EPOCHS + 1), gen\_history)

plt.xlabel('Epochs', fontsize=16)

plt.ylabel('Loss', fontsize=16)

plt.xticks(fontsize=16)

plt.yticks(fontsize=16)

plt.xlim(1, EPOCHS)

plt.grid()

plt.legend(fontsize=16)

plt.show()

n = 4

total = 2 \* n + 1

cnter = 0

plt.figure(figsize=(28, 8))

num = 1

for i in range(-n, n + 1):

for j in range(-n, n + 1):

ax = plt.subplot(7, 11, num)

num += 1

ip\_data = np.array([0.5\*i/n for \_ in range(20)])

ip\_data = np.expand\_dims(ip\_data,axis=0)

img = generator.predict(ip\_data)

plt.imshow(img[0, :, :, 0], cmap='gray')

plt.axis('off')

if num == 78:

break

plt.show()

# 200 epochs

EPOCHS = 50 # 200

gen\_history = train(train\_dataset, EPOCHS)

plt.figure(figsize=(24, 10))

plt.title('Visualization of the generator learning process', fontsize=16)

plt.plot(np.arange(1, EPOCHS + 1), gen\_history)

plt.xlabel('Epochs', fontsize=16)

plt.ylabel('Loss', fontsize=16)

plt.xticks(fontsize=16)

plt.yticks(fontsize=16)

plt.xlim(1, EPOCHS)

plt.grid()

plt.legend(fontsize=16)

plt.show()

n = 4

total = 2 \* n + 1

cnter = 0

plt.figure(figsize=(28, 8))

num = 1

for i in range(-n, n + 1):

for j in range(-n, n + 1):

ax = plt.subplot(7, 11, num)

num += 1

ip\_data = np.array([0.5\*i/n for \_ in range(20)])

ip\_data = np.expand\_dims(ip\_data,axis=0)

img = generator.predict(ip\_data)

plt.imshow(img[0, :, :, 0], cmap='gray')

plt.axis('off')

if num == 78:

break

plt.show()

**Results & Discussion:**

# Loss plots

# 100 epochs

A graph with blue lines

Description automatically generated

# 150 epochs

A graph with blue lines

Description automatically generated

# 200 epochs

A graph with blue lines

Description automatically generated

# Generations

# 100 epochs

A group of images of a person's head

Description automatically generated

# 150 epochs

A group of images of a person's body

Description automatically generated

# 200 epochs

A group of squares with dots

Description automatically generated with medium confidence

# Result

We see that the generations become more clear and precise with higher epochs and lower latent sizes.