Python Tutorial and Homework 2

This Colab Notebook contains 2 sections. The first section is a Python Tutorial intended to make you familiar with Numpy and Colab functionalities. **This section will not be graded**.

The second section is a coding assignment (Homework 2). **This section will be graded**

1.0 Python Tutorial

This assignment section won't be graded but is intended as a tutorial to refresh the basics of python and its dependencies. It also allows one to get familiarized with Google Colab.

1.0.0 Array manipulation using numpy

Q1 - Matrix multiplication

```
import numpy as np

### Create two numpy arrays with the dimensions 3x2 and 2x3
respectively using  np.arange().
### The elements of the vector are
### Vector 1 elements = [ 2,  4,  6,  8, 10, 12];
### Vector 2 elements = [ 7, 10, 13, 16, 19, 22]

### Starting at 2, stepping by 2
vector1 = np.arange(2,13,2).reshape(3,2)
### Starting at 7, stepping by 3
vector2 = np.arange(7,23,3).reshape(2,3)

### Print vec
print(vector1, vector2)

# ### Take product of the two matricies (Matrix product)
prod =np.dot(vector1, vector2)

# ### Print
print(prod)
```

```
[[ 2  4]
 [ 6  8]
 [10 12]] [[ 7 10 13]
 [16 19 22]]
[[ 78  96 114]
 [170 212 254]
 [262 328 394]]
```

Q2 - Diagonals

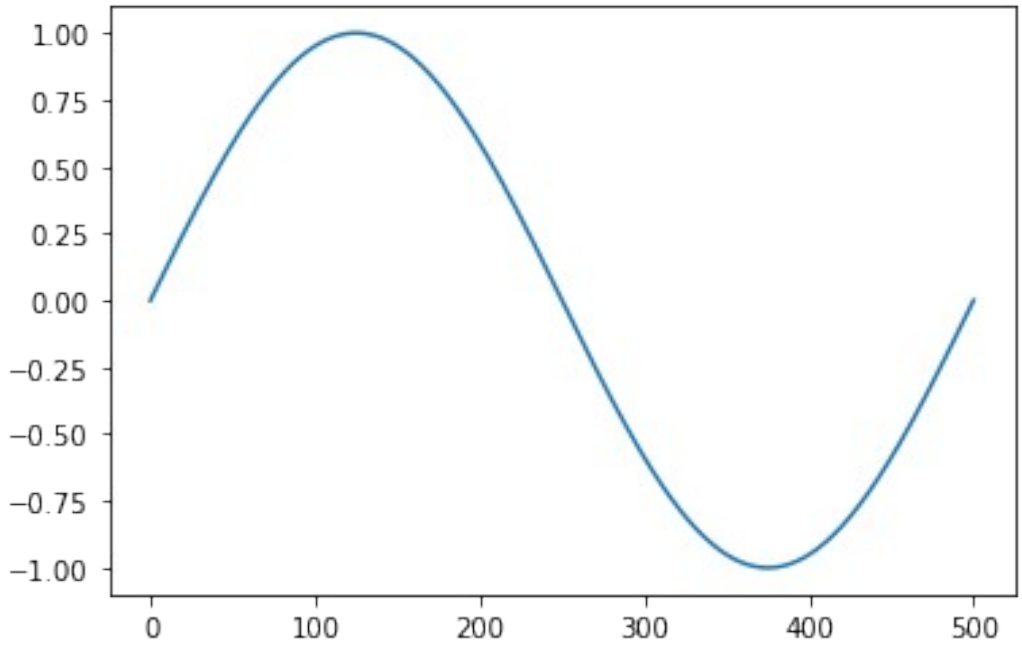
```
### Create two numpy arrays with the dimensions 10x10 using the
function np.arange().
### Starting at 2, stepping by 3
vector1 =np.arange(2,(3*10*10)+2,3).reshape(10,10)
### Starting at 35, stepping by 9
vector2 =np.arange(35,(9*10*10)+35,9).reshape(10,10)

### Print vec
print(vector1,'\n\n\n',vector2)

### Obtain the diagonal matrix of each vector1 such that the start of
the diagonal is from (3,0) and the end is (9,6)
### Reshape the the matrix such that it form a diagonal maritix of
shape(7,7)
vector1_offset_diagonal =np.diag(np.diag(vector1[3:10,0:7]))



### Obtain a 7x7 matrix from the vector 2
### starting from (left top element) = (0,3)
### ending at (right bottom element) = (6,9)
vector2_offset_diagonal =(np.diag(np.diag(vector2[0:7,3:10])))

### Print diagonal matrix
print(vector1_offset_diagonal,'\n\n\n', vector2_offset_diagonal)


### Take product of the two diagonal matricies (Matrix product)
prod =np.dot(vector1_offset_diagonal,vector2_offset_diagonal)

### Print
print(prod)
```

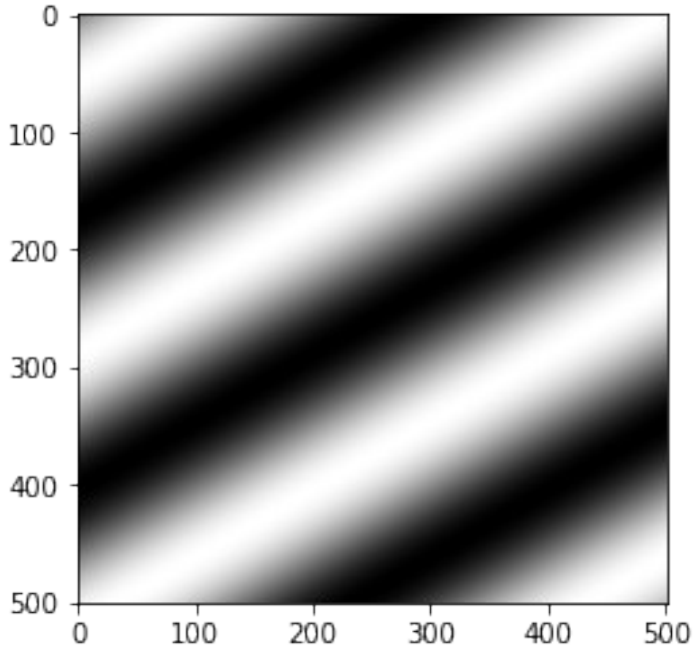
```
[[  2   5   8  11  14  17  20  23  26  29]
 [ 32  35  38  41  44  47  50  53  56  59]
 [ 62  65  68  71  74  77  80  83  86  89]
 [ 92  95  98 101 104 107 110 113 116 119]
 [122 125 128 131 134 137 140 143 146 149]
 [152 155 158 161 164 167 170 173 176 179]
 [182 185 188 191 194 197 200 203 206 209]
 [212 215 218 221 224 227 230 233 236 239]
 [242 245 248 251 254 257 260 263 266 269]
 [272 275 278 281 284 287 290 293 296 299]]


 [[ 35  44  53  62  71  80  89  98 107 116]
 [125 134 143 152 161 170 179 188 197 206]
 [215 224 233 242 251 260 269 278 287 296]
 [305 314 323 332 341 350 359 368 377 386]
```

```
 [395 404 413 422 431 440 449 458 467 476]
 [485 494 503 512 521 530 539 548 557 566]
 [575 584 593 602 611 620 629 638 647 656]
 [665 674 683 692 701 710 719 728 737 746]
 [755 764 773 782 791 800 809 818 827 836]
 [845 854 863 872 881 890 899 908 917 926]]
[[ 92   0   0   0   0   0   0]
 [  0 125   0   0   0   0   0]
 [  0   0 158   0   0   0   0]
 [  0   0   0 191   0   0   0]
 [  0   0   0   0 224   0   0]
 [  0   0   0   0   0 257   0]
 [  0   0   0   0   0   0 290]]


 [[ 62   0   0   0   0   0   0]
 [  0 161   0   0   0   0   0]
 [  0   0 260   0   0   0   0]
 [  0   0   0 359   0   0   0]
 [  0   0   0   0 458   0   0]
 [  0   0   0   0   0 557   0]
 [  0   0   0   0   0   0 656]]
[[  5704      0      0      0      0      0      0]
 [     0  20125      0      0      0      0      0]
 [     0      0  41080      0      0      0      0]
 [     0      0      0  68569      0      0      0]
 [     0      0      0      0 102592      0      0]
 [     0      0      0      0      0 143149      0]
 [     0      0      0      0      0      0 190240]]
```

Q3 - Sin wave

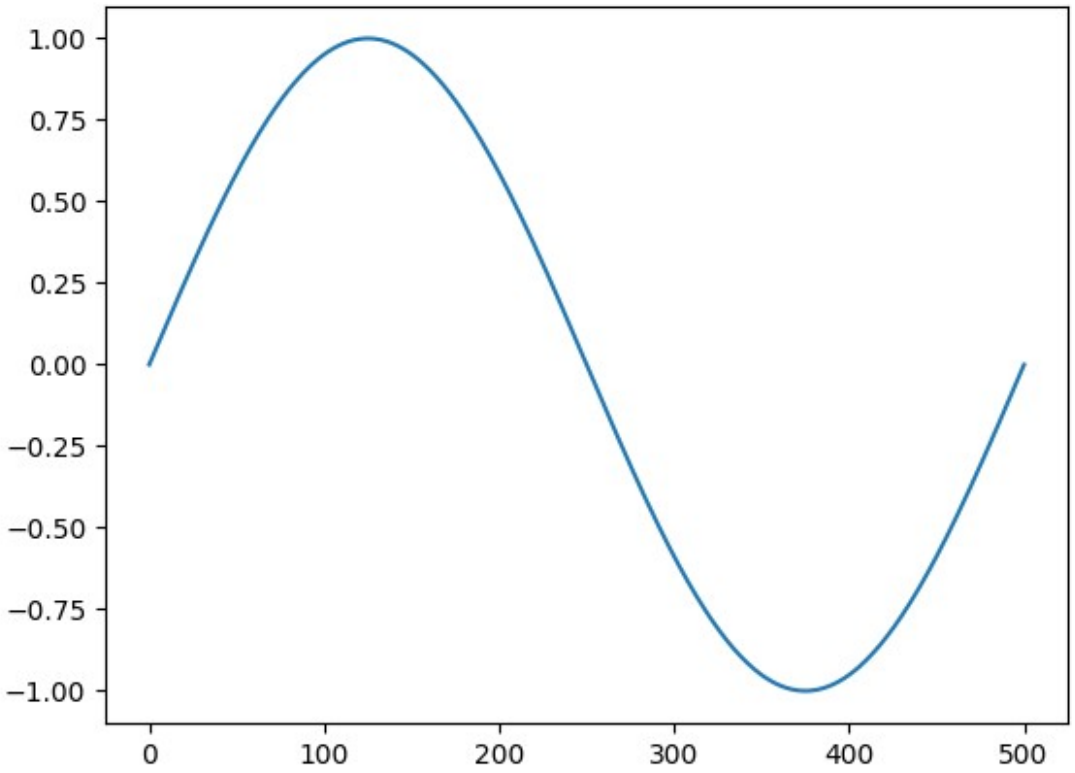
Sample outputs,





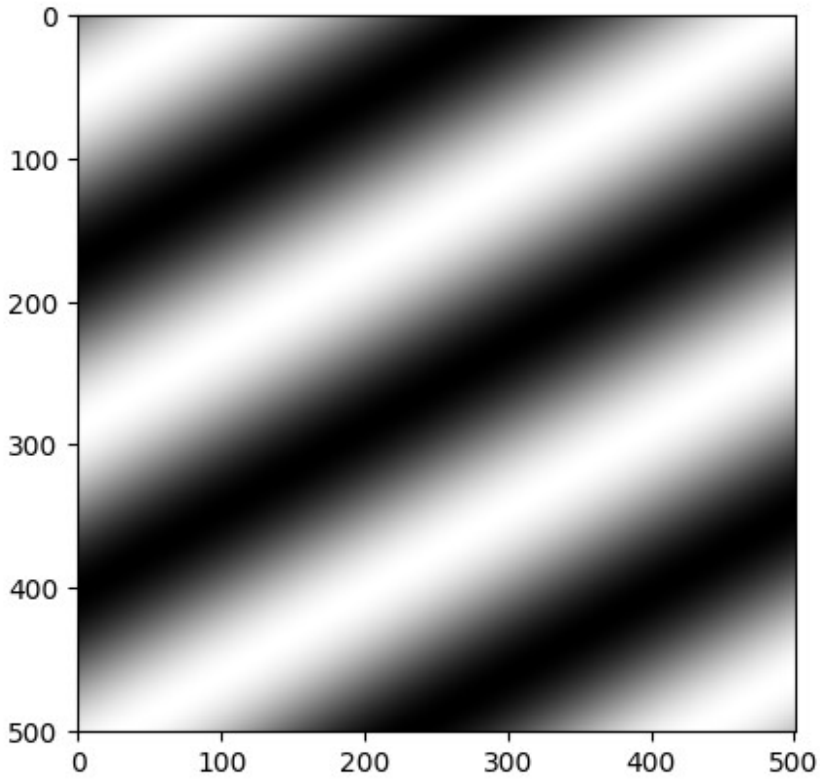

```
import matplotlib.pyplot as plt
### Create a time matrix that evenly samples a sine wave at a
frequency of 1Hz
### Starting at time step T = 0
### End at time step T = 500
time =np.arange(0,501,1)
```

```
### Given wavelength of
wavelength = 500

### Construct a sin wave using the formula sin(2*pi*(time/wavelength))
y =np.sin(2*np.pi*(time/wavelength))

#### Plot the wave
plt.plot(time, y)
plt.show()
```




```
#### Given a 2D mesh grid
X, Y = np.meshgrid(time, time)
#### wavelength and angle of rotation(phi) of the sin wave in 2d.
Imagine a 2D sine wave is being rotating about the Z axes
wavelength = 200
phi = np.pi / 3

#### Calculate the sin wave in 2d space using the formula
sin(2*pi*(x'/wavelength)) where x'=Xcos(phi) + Ysin(phi)
xprime=(X*np.cos(phi)) + (Y*np.sin(phi))
grating = np.sin(2*np.pi*(xprime/wavelength))

#### Plot the wave
#### Intuition, think of the white area as hills and the black areas
```

```
as valleys
plt.set_cmap("gray")
plt.imshow(grating)
plt.show()
```



Q4 Car Brands

```
cars = ['Civic', 'Insight', 'Fit', 'Accord', 'Ridgeline',
'Avancier','Pilot', 'Legend', 'Beat', 'FR-V', 'HR-V', 'Shuttle']

#### Create a 3D array of cars of shape 2,3,2
cars_3d =np.array(cars).reshape(2,3,2)

#### Extract the top layer of the matrix. Top layer of a matrix A of
shape(2,3,2) will have the following structue A_top = [[A[0,0,0],
A[0,0,1]],[A[0,1,0],A[0,1,1]],[A[0,2,0],A[0,2,1]]]
#### HINT - Array slicing or splitting
cars_top_layer =cars_3d[0,:,:]

#### Similarly extract the bottom layer
#### HINT - Array slicing or splitting
cars_bottom_layer =cars_3d[-1,:,:]

#### Print layers
```

```
print("\nTop Layer \n ",cars_top_layer,"\nBottom Layer\n",
cars_bottom_layer)

# #### Flatten the top layer
cars_top_flat =cars_top_layer.flatten()
# #### Flatten the bottom layer
cars_bottom_flat =cars_bottom_layer.flatten()

# #### Print layers
print("\nTop Flattened : ",cars_top_flat,"\nBottom Flattened :
",cars_bottom_flat)


new_car_list = np.empty((cars_top_layer.size +
cars_bottom_layer.size,), dtype=object)
#### Interweave the to flattened lists and insert into new_car_list
such that new_car_list=['Civic' 'Pilot' 'Fit' 'Beat' 'Ridgeline' 'HR-
V' 'Insight' 'Legend' 'Accord' 'FR-V' 'Avancier' 'Shuttle']
#### Using only array slicing
new_car_list[0::2] =
np.hstack((cars_top_flat[0::2],cars_top_flat[1::2]))
new_car_list[1::2] =
np.hstack((cars_bottom_flat[0::2],cars_bottom_flat[1::2]))

#### Concatenate and flatten the top and bottom layer such that the
final list is of the form cat_flat = ['Civic' 'Insight' 'Pilot'
'Legend' 'Fit' 'Accord' 'Beat' 'FR-V' 'Ridgeline' 'Avancier' 'HR-V'
'Shuttle']
cat_flat = np.hstack((cars_top_layer,cars_bottom_layer)).flatten()

#### Print layers
print("\n\nInterwoven - ", new_car_list,"\nConcatenate and flatten -
", cat_flat)


Top Layer
  [['Civic' 'Insight']
 ['Fit' 'Accord']
 ['Ridgeline' 'Avancier']]
Bottom Layer
 [['Pilot' 'Legend']
 ['Beat' 'FR-V']
 ['HR-V' 'Shuttle']]

Top Flattened :  ['Civic' 'Insight' 'Fit' 'Accord' 'Ridgeline'
'Avancier']
Bottom Flattened :  ['Pilot' 'Legend' 'Beat' 'FR-V' 'HR-V' 'Shuttle']


Interwoven -  ['Civic' 'Pilot' 'Fit' 'Beat' 'Ridgeline' 'HR-V'
```

```
'Insight' 'Legend'
 'Accord' 'FR-V' 'Avancier' 'Shuttle']
Concatenate and flatten -  ['Civic' 'Insight' 'Pilot' 'Legend' 'Fit'
'Accord' 'Beat' 'FR-V'
 'Ridgeline' 'Avancier' 'HR-V' 'Shuttle']
```

1.0.1 Basics tensorflow

Helper functions

```
import tensorflow as tf
from keras.utils import to_categorical

def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

Q1 MNIST Classifier

```
## Import the MNIST dataset from keras
mnist = tf.keras.datasets.mnist
### Load the data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

### Normalize the 8bit images with values in the range [0,255]
x_train = tf.keras.utils.normalize(x_train, axis=1)
x_test  = tf.keras.utils.normalize(x_test, axis=1)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

```
## Create a model with the following architecture Flatten ->
Dense(128, relu) -> Dense(64,relu) -> outpuLayer(size=10)
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

```
### Complie the model with the adam optimizer and crossentropy loss
### HINT - No One hot encoding
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

```
### Train the model on the train data for 5 epochs
model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 20s 9ms/step - loss:
0.2764 - accuracy: 0.9196
Epoch 2/5
1875/1875 [==============================] - 11s 6ms/step - loss:
0.1133 - accuracy: 0.9647
Epoch 3/5
1875/1875 [==============================] - 8s 4ms/step - loss:
0.0766 - accuracy: 0.9758
Epoch 4/5
1875/1875 [==============================] - 8s 4ms/step - loss:
0.0576 - accuracy: 0.9815
Epoch 5/5
1875/1875 [==============================] - 9s 5ms/step - loss:
0.0433 - accuracy: 0.9855
```

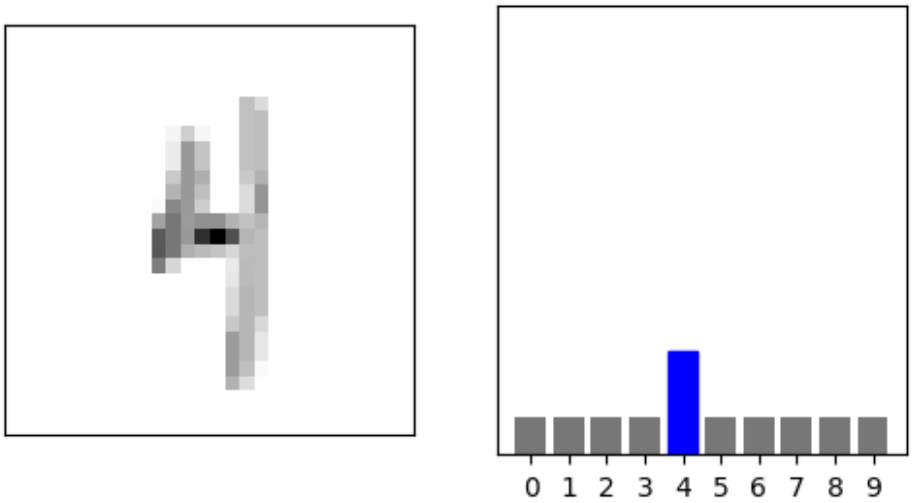
```
<keras.src.callbacks.History at 0x7e1465c6d3f0>
```

```
### Check the accuracy of the trained model
test_loss, test_acc = model.evaluate(x_test,  y_test, verbose=2)
print('\nTest accuracy:', test_acc)
```

```
313/313 - 1s - loss: 0.1051 - accuracy: 0.9697 - 655ms/epoch -
2ms/step

Test accuracy: 0.9696999788284302
```

```
### Convert the above model to a probabilistic model with a softmax as
the output layer
```

```
probability_model =
tf.keras.Sequential([model,tf.keras.layers.Softmax()])

### Run the test data through the new model and get predictions
predictions = probability_model.predict(x_test)

### Plot a test output
i = 65 ### <---- Change this to some random number to see different
predictions
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], y_test, x_test)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  y_test)
plt.show()
### Blue bars mean correct guess red bar means wrong guess!!

313/313 [==============================] - 1s 3ms/step
```



1.0.2 Basic Pytorch Tutorial

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Set the random seed for reproducibility
torch.manual_seed(42)

# Define a simple feedforward neural network
class Net(nn.Module):
    def __init__(self):
```

```
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 128)  # 28x28 input size (MNIST
images are 28x28 pixels)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)   # 10 output classes (digits 0-
9)

    def forward(self, x):
        x = x.view(-1, 784)  # Flatten the input image
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Load the MNIST dataset and apply transformations
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
shuffle=False)

# Initialize the neural network and optimizer
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        optimizer.zero_grad()

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

        running_loss += loss.item()

    print(f'Epoch {epoch+1}, Loss: {running_loss / len(trainloader)}')
```

```
print('Finished Training')

# Evaluate the model on the test set
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy on test set: {100 * correct / total}%')
```

```
Epoch 1, Loss: 0.7497771851766084
Epoch 2, Loss: 0.3669367114395729
Epoch 3, Loss: 0.3210167052077332
Epoch 4, Loss: 0.29383885016096933
Epoch 5, Loss: 0.2732162083000707
Epoch 6, Loss: 0.2538067396285374
Epoch 7, Loss: 0.23667215858139337
Epoch 8, Loss: 0.22128436360945072
Epoch 9, Loss: 0.20731170845629054
Epoch 10, Loss: 0.1948725388272167
Finished Training
Accuracy on test set: 94.49%
```

2.0 **Homework 2**

90 points

Note : This section will be graded and must be attemped using Pytorch only

Graded Section : Deep Learning Approach

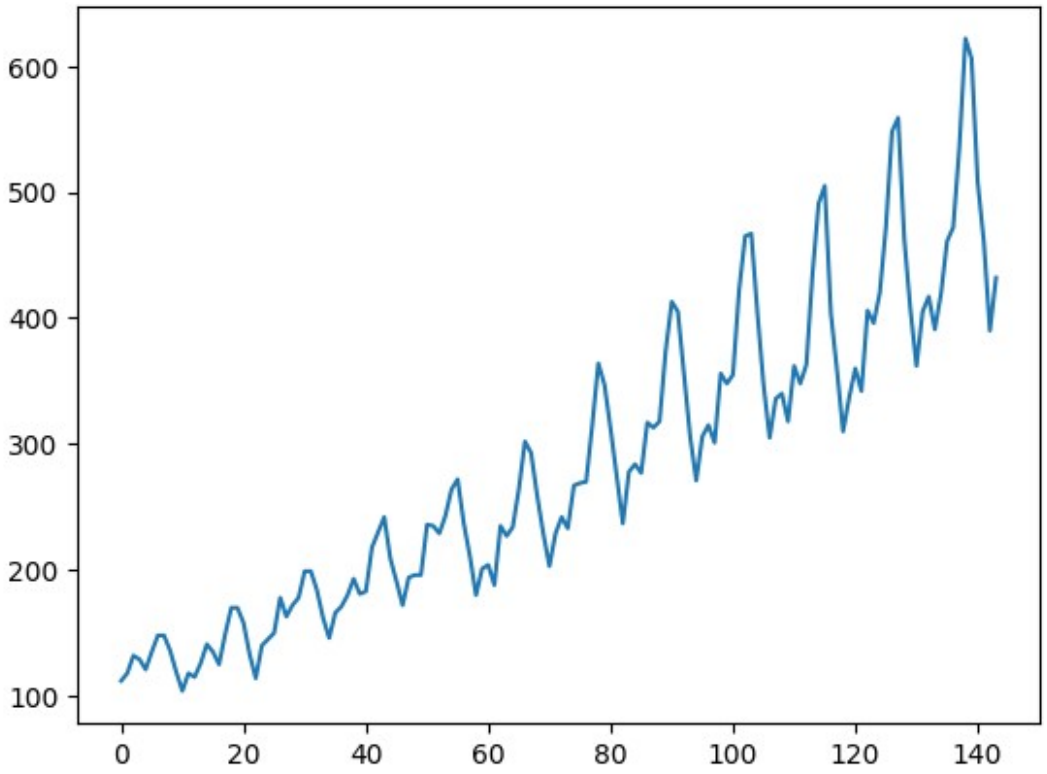
Time-Series Prediction Time series and sequence prediction could be a really amazing to predict/estimate a robot's trajectory which requires temporal data at hand. In this assignemnt we will see how this could be done using Deep Learning.

Given a dataset link for airline passengers prediction problem. Predict the number of international airline passengers in units of 1,000 given a year and a month. Here is how the data looks like.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
import math

#Importing the dataset
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-
passengers.csv"
#reading the dataset using pandas
df = pd.read_csv(url)

# Extracting time series data
timeseries = df["Passengers"].values.astype('float32')
monthseries = df["Month"].values.astype('str')


# Plotting the dataset
plt.plot(timeseries)
plt.show()
```



1. Write the dataloader code to pre-process the data for pytorch tensors using any library of your choice. Here is a good resource for the dataloader Video link

```
# Write your code here for the dataloader in Pytorch
# Using the timeseries data extracted above, separate data for
training and testing
train_size = math.ceil(int(len(timeseries) * 0.90))
test_size = len(timeseries) - train_size
train, test = timeseries[:train_size], timeseries[train_size:]

#Class to split create data for the LSTM Model
class DatasetCreator:
    def __init__(self, lookback):
        self.lookback = lookback

    def __call__(self, dataset):
        X, y = [], []
        for i in range(len(dataset) - self.lookback):
            feature = dataset[i:i + self.lookback]
            target = dataset[i + self.lookback]
            X.append(feature)
            y.append(target)

        X = np.array(X).astype(np.float32)
        y = np.array(y).astype(np.float32)
        return torch.tensor(X), torch.tensor(y)
```

```
#Defining the Lookback value
lookback = 4

#Object of DatasetCreator Class
dataset_creator = DatasetCreator(lookback)
#Extracting training datasets
X_train, y_train = dataset_creator.__call__(train)
#Extracting testing datasets
X_test, y_test = dataset_creator.__call__(test)

#Dataloading the training and testing datasets
trainloader = data.DataLoader(data.TensorDataset(X_train, y_train),
shuffle=False, batch_size=12)
testloader = data.DataLoader(data.TensorDataset(X_test, y_test),
shuffle=False, batch_size=12)
```

1. Create the model in pytorch here uinsg 1. Long-Short Term Memory (LSTM) and 2. Recurrent Neural Network (RNN). Here is a good resource for Custom model generation.

Train using the two models. Here is the resource for the same Video link

```
# write your code here for the custom model creating for both the
methods

class LstmModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.lstm = nn.LSTM(input_size=1, hidden_size=50,
num_layers=1, batch_first=True)
        self.linear = nn.Linear(50, 1)

    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.linear(x[:, -1, :])
        return x


class RnnModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.rnn = nn.RNN(input_size=1, hidden_size=50, num_layers=1,
batch_first=True)
        self.linear = nn.Linear(50, 1)

    def forward(self, x):
        x, _ = self.rnn(x)
        x = self.linear(x[:, -1, :])
```

```
        return x
```

1. Evaluate and Compare the result using proper metric. Justify the metrics used. In the model RMSE and Visual inspection is carried out to evaluate and compare the result of the model

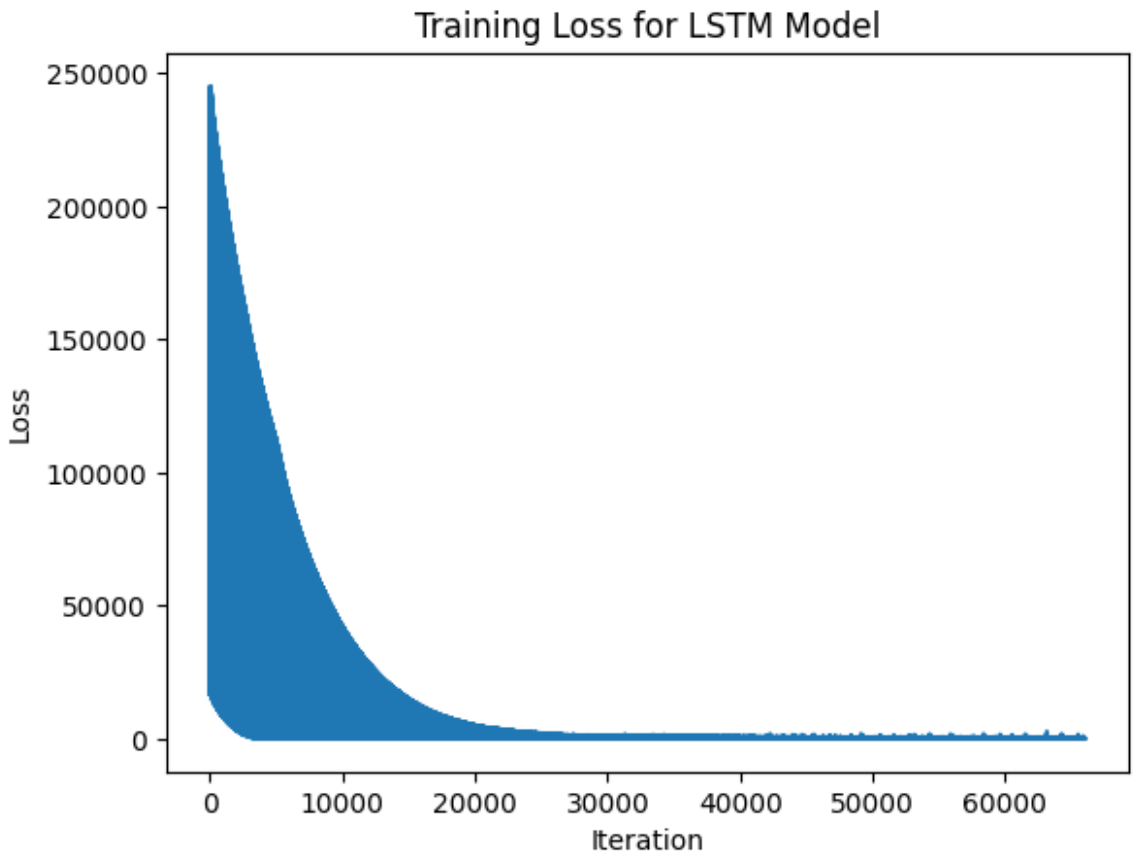
- Root Mean Squared Error(RMSE): RMSE measures the average magnitude of the error between predicted and actual values. Lower the RMSE, the better is the models performance and hence RMSE is used to evaluate the model.
- The results have been compared by plotting the time series graph of the actual input values to the training and testing inputs to the LSTM and RNN model. This allows us to quantitatively visualise how well our model is capturing and analysing trends in the data.

```
#Runnning the LSTM Model for the train and test dataset and
calcualtion of
lstmmodel = LstmModel()
#Defining optimiser and Loss function
optimizer = optim.Adam(lstmmodel.parameters(),lr=0.001)
loss_fn = nn.MSELoss(reduction='mean')
lossplot=[]
n_epochs = 6000

# Looping through epochs
for epoch in range(n_epochs):
    lstmmodel.train()
    for X_batch, y_batch in trainloader:
        y_pred = lstmmodel(X_batch.unsqueeze(-1))  #Forward pass:
Getting predictions from the model
        loss = loss_fn(y_pred, y_batch.unsqueeze(-1))  #Calculating
the Loss
        lossplot.append(loss)
        optimizer.zero_grad()  # Zeroing the gradients
        loss.backward() # Backpropagation
        optimizer.step()
    # Validation
    if epoch % 100 == 0:
        lstmmodel.eval() # Evaludating the model
        with torch.no_grad():
          y_pred_train = lstmmodel(X_train.unsqueeze(-1)).squeeze(-1)
          y_pred_test = lstmmodel(X_test.unsqueeze(-1)).squeeze(-1)

          # Calculating RMSE for train and test data
          test_rmse = np.sqrt(loss_fn(y_pred_test, y_test))
          train_rmse = np.sqrt(loss_fn(y_pred_train, y_train))

          threshold = 0.1  # Define a threshold
          correct_train = ((torch.abs(y_pred_train - y_train) /
y_train) <= threshold).sum().item()
          accuracy_train = correct_train / len(y_train) * 100
```

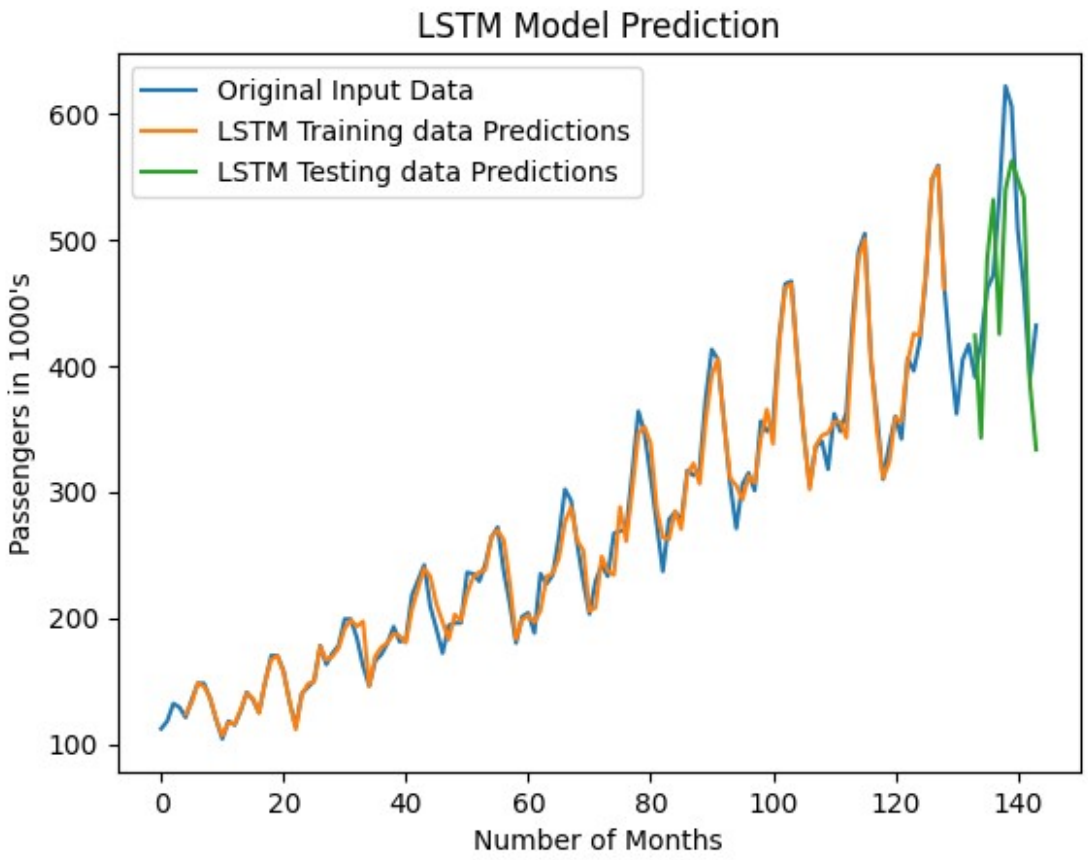
```
          correct_test = ((torch.abs(y_pred_test - y_test) / y_test)
<= threshold).sum().item()
          accuracy_test = correct_test / len(y_test) * 100
        print("Epoch %d: train RMSE: %.4f, test RMSE: %.4f, Loss:
%.4f" % (epoch, train_rmse, test_rmse,loss))

with torch.no_grad():
    # shifting the train predictions for plot
    train_plot = np.ones_like(timeseries) * np.nan
    y_pred = lstmmodel(X_train.unsqueeze(-1)).squeeze(-1)
    train_plot[lookback:train_size] = y_pred
    # shifting the test predictions for plot
    test_plot = np.ones_like(timeseries) * np.nan
    y_pred = lstmmodel(X_test.unsqueeze(-1)).squeeze(-1)
    test_plot[train_size+lookback:len(timeseries)] = y_pred


Epoch 0: train RMSE: 283.6658, test RMSE: 487.3471, Loss: 245203.3281
Epoch 100: train RMSE: 243.2734, test RMSE: 443.9193, Loss:
203828.5469
Epoch 200: train RMSE: 211.2160, test RMSE: 408.3787, Loss:
172748.3125
Epoch 300: train RMSE: 182.1785, test RMSE: 374.7237, Loss:
145645.5625
Epoch 400: train RMSE: 157.4483, test RMSE: 344.0069, Loss:
122882.5469
Epoch 500: train RMSE: 133.1683, test RMSE: 311.6363, Loss:
100940.1172
Epoch 600: train RMSE: 111.0234, test RMSE: 279.3086, Loss: 81103.7109
Epoch 700: train RMSE: 94.0255, test RMSE: 251.9253, Loss: 65933.3906
Epoch 800: train RMSE: 79.8559, test RMSE: 227.0689, Loss: 53455.9492
Epoch 900: train RMSE: 68.1757, test RMSE: 204.3927, Loss: 43144.1797
Epoch 1000: train RMSE: 58.7229, test RMSE: 183.8110, Loss: 34665.7773
Epoch 1100: train RMSE: 51.3716, test RMSE: 166.3927, Loss: 27771.1836
Epoch 1200: train RMSE: 45.6475, test RMSE: 150.1798, Loss: 22036.1035
Epoch 1300: train RMSE: 41.4311, test RMSE: 135.8278, Loss: 17571.8750
Epoch 1400: train RMSE: 37.6521, test RMSE: 121.8599, Loss: 13823.7324
Epoch 1500: train RMSE: 34.5982, test RMSE: 110.9431, Loss: 10855.3105
Epoch 1600: train RMSE: 32.2858, test RMSE: 101.0053, Loss: 8574.8154
Epoch 1700: train RMSE: 30.2896, test RMSE: 93.8418, Loss: 6748.5601
Epoch 1800: train RMSE: 28.3176, test RMSE: 88.3102, Loss: 5283.7876
Epoch 1900: train RMSE: 27.2468, test RMSE: 83.7472, Loss: 4184.1978
Epoch 2000: train RMSE: 26.2569, test RMSE: 80.1486, Loss: 3383.1245
Epoch 2100: train RMSE: 25.5569, test RMSE: 79.3338, Loss: 2859.8625
Epoch 2200: train RMSE: 23.8867, test RMSE: 79.3112, Loss: 2174.4580
Epoch 2300: train RMSE: 23.3016, test RMSE: 79.3631, Loss: 1735.2740
Epoch 2400: train RMSE: 22.4271, test RMSE: 84.6986, Loss: 1453.5647
Epoch 2500: train RMSE: 21.8876, test RMSE: 82.3410, Loss: 1181.8951
Epoch 2600: train RMSE: 21.6589, test RMSE: 81.8441, Loss: 857.1945
Epoch 2700: train RMSE: 20.7263, test RMSE: 84.6497, Loss: 773.8430
```

```
Epoch 2800: train RMSE: 20.0742, test RMSE: 86.0414, Loss: 652.7208
Epoch 2900: train RMSE: 20.1812, test RMSE: 87.5726, Loss: 637.6691
Epoch 3000: train RMSE: 20.0577, test RMSE: 85.9973, Loss: 686.0533
Epoch 3100: train RMSE: 20.2364, test RMSE: 66.9181, Loss: 682.4213
Epoch 3200: train RMSE: 19.0405, test RMSE: 66.3239, Loss: 469.5577
Epoch 3300: train RMSE: 18.4801, test RMSE: 66.0961, Loss: 324.0569
Epoch 3400: train RMSE: 18.4539, test RMSE: 67.1485, Loss: 425.8816
Epoch 3500: train RMSE: 18.2011, test RMSE: 63.0015, Loss: 118.3030
Epoch 3600: train RMSE: 17.9727, test RMSE: 59.3432, Loss: 214.6743
Epoch 3700: train RMSE: 17.6532, test RMSE: 63.2923, Loss: 214.1452
Epoch 3800: train RMSE: 16.8878, test RMSE: 64.0543, Loss: 130.3487
Epoch 3900: train RMSE: 16.5961, test RMSE: 63.4125, Loss: 75.2223
Epoch 4000: train RMSE: 18.3780, test RMSE: 59.6778, Loss: 647.7572
Epoch 4100: train RMSE: 15.9803, test RMSE: 64.0311, Loss: 44.8738
Epoch 4200: train RMSE: 15.9507, test RMSE: 65.3455, Loss: 48.1899
Epoch 4300: train RMSE: 15.5754, test RMSE: 66.3013, Loss: 43.8142
Epoch 4400: train RMSE: 17.7673, test RMSE: 63.8307, Loss: 461.3052
Epoch 4500: train RMSE: 15.6008, test RMSE: 63.9796, Loss: 90.2026
Epoch 4600: train RMSE: 17.9432, test RMSE: 62.9175, Loss: 398.2756
Epoch 4700: train RMSE: 15.0865, test RMSE: 65.7506, Loss: 151.6618
Epoch 4800: train RMSE: 15.3566, test RMSE: 62.6022, Loss: 95.0289
Epoch 4900: train RMSE: 14.4834, test RMSE: 63.6913, Loss: 53.6872
Epoch 5000: train RMSE: 13.8981, test RMSE: 64.1724, Loss: 56.1613
Epoch 5100: train RMSE: 14.2670, test RMSE: 67.6250, Loss: 199.8257
Epoch 5200: train RMSE: 13.9442, test RMSE: 62.3501, Loss: 59.9440
Epoch 5300: train RMSE: 14.1799, test RMSE: 65.3634, Loss: 111.1066
Epoch 5400: train RMSE: 13.5518, test RMSE: 65.6400, Loss: 61.4863
Epoch 5500: train RMSE: 18.3785, test RMSE: 63.7021, Loss: 442.2151
Epoch 5600: train RMSE: 15.3458, test RMSE: 64.2083, Loss: 273.9984
Epoch 5700: train RMSE: 13.0065, test RMSE: 64.9778, Loss: 29.9530
Epoch 5800: train RMSE: 12.2167, test RMSE: 66.1986, Loss: 3.3790
Epoch 5900: train RMSE: 12.0523, test RMSE: 65.4371, Loss: 2.2215
```

```python
loss_array = [loss.detach().numpy() for loss in lossplot]
plt.plot(loss_array)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Loss for LSTM Model')
plt.show()
```




```
#Visualising the model training and testing plots to check the model
accuracy
plt.plot(timeseries, label='Original Input Data')
plt.plot(train_plot, label='LSTM Training data Predictions')
plt.plot(test_plot, label='LSTM Testing data Predictions')
plt.title('LSTM Model Prediction')
plt.xlabel("Number of Months")
plt.ylabel("Passengers in 1000's")
plt.legend()
plt.show()
```




```
#Instantiating the optimizer and Loss Function

rnnmodel = RnnModel()
# Defining optimizer and Loss function
optimizer = optim.Adam(rnnmodel.parameters(), lr=0.001)
loss_fn = nn.MSELoss()

n_epochs = 6000
# Looping through epochs
for epoch in range(n_epochs):
    rnnmodel.train()
    for X_batch, y_batch in trainloader:
        rnny_pred = rnnmodel(X_batch.unsqueeze(-1))  # Adding an extra
dimension for LSTM input
        loss = loss_fn(rnny_pred, y_batch.unsqueeze(-1))  # Adding an
extra dimension for target
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # Validation
    if epoch % 100 == 0:
        rnnmodel.eval()
        with torch.no_grad():
```

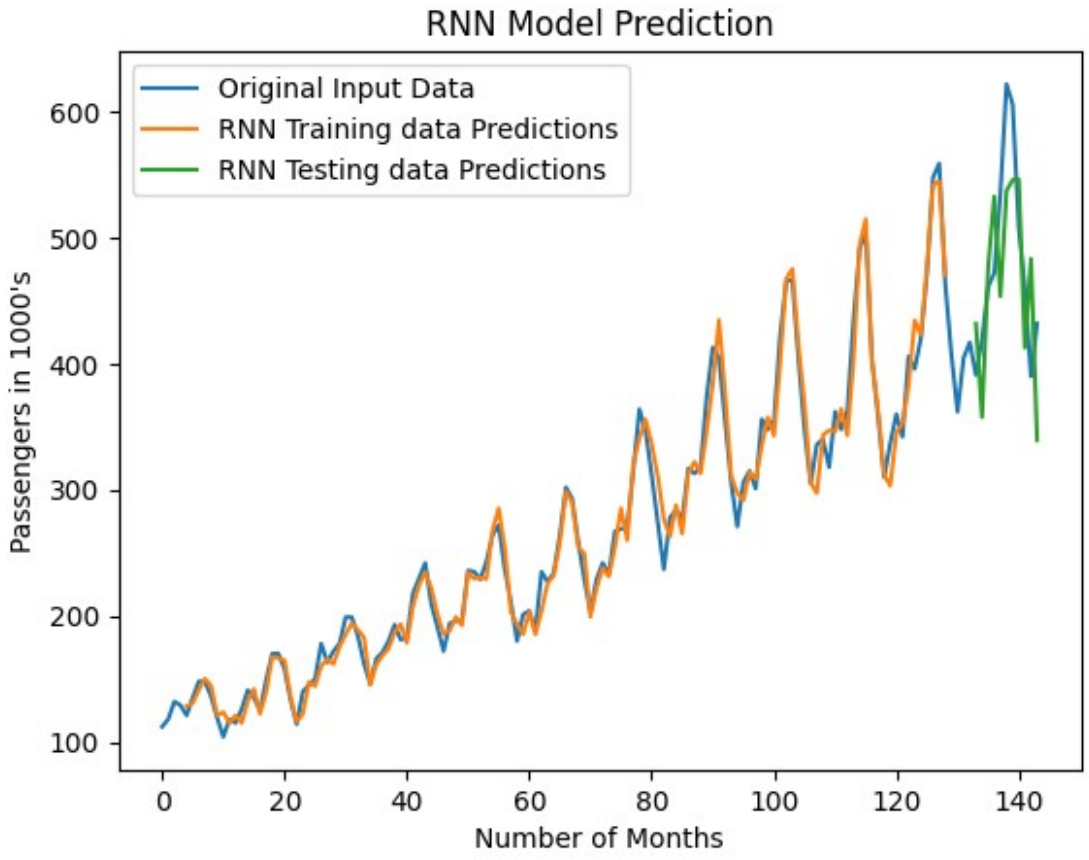
```
            rnny_pred_train = rnnmodel(X_train.unsqueeze(-
1)).squeeze(-1)
            train_rmse = np.sqrt(loss_fn(rnny_pred_train, y_train))
            rnny_pred_test = rnnmodel(X_test.unsqueeze(-1)).squeeze(-
1)
            test_rmse = np.sqrt(loss_fn(rnny_pred_test, y_test))

            threshold = 0.1  # Define a threshold
            rnncorrect_train = ((torch.abs(rnny_pred_train - y_train)
/ y_train) <= threshold).sum().item()
            rnnaccuracy_train = rnncorrect_train / len(y_train) * 100
            rnncorrect_test = ((torch.abs(rnny_pred_test - y_test) /
y_test) <= threshold).sum().item()
            rnnaccuracy_test = rnncorrect_test / len(y_test) * 100
        print("Epoch %d: train RMSE: %.4f, test RMSE: %.4f, Loss:
%.4f" % (epoch, train_rmse, test_rmse,loss))

with torch.no_grad():
    # shift train predictions for plotting
    rnntrain_plot = np.ones_like(timeseries) * np.nan
    rnny_pred = rnnmodel(X_train.unsqueeze(-1)).squeeze(-1)
    rnntrain_plot[lookback:train_size] = rnny_pred
    # shift test predictions for plotting
    rnntest_plot = np.ones_like(timeseries) * np.nan
    rnny_pred = rnnmodel(X_test.unsqueeze(-1)).squeeze(-1)
    rnntest_plot[train_size+lookback:len(timeseries)] = rnny_pred
```

```
Epoch 0: train RMSE: 282.1872, test RMSE: 485.7993, Loss: 243724.5000
Epoch 100: train RMSE: 236.3437, test RMSE: 436.3376, Loss:
196995.6250
Epoch 200: train RMSE: 197.1136, test RMSE: 392.2644, Loss:
159496.3750
Epoch 300: train RMSE: 164.2597, test RMSE: 352.7437, Loss:
129171.1406
Epoch 400: train RMSE: 137.5921, test RMSE: 317.7661, Loss:
104934.3906
Epoch 500: train RMSE: 115.1113, test RMSE: 285.5721, Loss: 84784.3906
Epoch 600: train RMSE: 96.5154, test RMSE: 256.2759, Loss: 68244.6719
Epoch 700: train RMSE: 80.9765, test RMSE: 229.4886, Loss: 54617.3828
Epoch 800: train RMSE: 68.3021, test RMSE: 205.2157, Loss: 43501.2305
Epoch 900: train RMSE: 58.1824, test RMSE: 183.3255, Loss: 34477.8086
Epoch 1000: train RMSE: 50.3498, test RMSE: 163.8373, Loss: 27240.0820
Epoch 1100: train RMSE: 44.1780, test RMSE: 146.6948, Loss: 21481.7988
Epoch 1200: train RMSE: 39.1259, test RMSE: 131.8990, Loss: 16885.3027
Epoch 1300: train RMSE: 35.3696, test RMSE: 119.1354, Loss: 13210.6064
Epoch 1400: train RMSE: 32.0399, test RMSE: 108.1314, Loss: 10263.2100
Epoch 1500: train RMSE: 30.2227, test RMSE: 98.6982, Loss: 8064.3032
Epoch 1600: train RMSE: 27.6884, test RMSE: 91.5418, Loss: 6214.1416
Epoch 1700: train RMSE: 27.1464, test RMSE: 93.3818, Loss: 6557.6436
Epoch 1800: train RMSE: 25.8435, test RMSE: 87.3162, Loss: 5056.8647
```

```
Epoch 1900: train RMSE: 25.4259, test RMSE: 85.4993, Loss: 4355.1030
Epoch 2000: train RMSE: 23.1621, test RMSE: 81.0218, Loss: 3540.7878
Epoch 2100: train RMSE: 22.1724, test RMSE: 79.7689, Loss: 3038.2781
Epoch 2200: train RMSE: 21.2177, test RMSE: 75.7229, Loss: 2474.8464
Epoch 2300: train RMSE: 20.4017, test RMSE: 75.5479, Loss: 2105.8530
Epoch 2400: train RMSE: 19.8006, test RMSE: 72.0948, Loss: 2055.2297
Epoch 2500: train RMSE: 18.8869, test RMSE: 71.6884, Loss: 1565.4844
Epoch 2600: train RMSE: 19.1389, test RMSE: 70.9878, Loss: 1268.1133
Epoch 2700: train RMSE: 19.1260, test RMSE: 71.6259, Loss: 1320.2576
Epoch 2800: train RMSE: 17.9423, test RMSE: 72.3820, Loss: 969.9998
Epoch 2900: train RMSE: 17.5874, test RMSE: 67.1378, Loss: 986.1608
Epoch 3000: train RMSE: 17.0555, test RMSE: 67.5810, Loss: 761.6896
Epoch 3100: train RMSE: 22.9555, test RMSE: 58.5589, Loss: 1135.4772
Epoch 3200: train RMSE: 20.5413, test RMSE: 66.0816, Loss: 1710.0006
Epoch 3300: train RMSE: 19.6002, test RMSE: 64.9503, Loss: 1287.5342
Epoch 3400: train RMSE: 21.1239, test RMSE: 63.4499, Loss: 1699.7233
Epoch 3500: train RMSE: 19.0506, test RMSE: 64.7407, Loss: 1265.1819
Epoch 3600: train RMSE: 19.2403, test RMSE: 65.4385, Loss: 1154.6608
Epoch 3700: train RMSE: 20.4830, test RMSE: 71.2157, Loss: 2008.4141
Epoch 3800: train RMSE: 18.2935, test RMSE: 67.2945, Loss: 1615.1718
Epoch 3900: train RMSE: 17.9690, test RMSE: 71.2326, Loss: 1551.6117
Epoch 4000: train RMSE: 20.9183, test RMSE: 64.9591, Loss: 1926.8652
Epoch 4100: train RMSE: 16.2384, test RMSE: 67.6016, Loss: 933.8312
Epoch 4200: train RMSE: 16.5064, test RMSE: 71.5516, Loss: 941.5156
Epoch 4300: train RMSE: 15.8152, test RMSE: 60.9481, Loss: 963.2586
Epoch 4400: train RMSE: 14.7469, test RMSE: 60.8394, Loss: 290.3502
Epoch 4500: train RMSE: 14.0826, test RMSE: 63.5884, Loss: 416.2492
Epoch 4600: train RMSE: 15.4595, test RMSE: 62.2370, Loss: 521.0163
Epoch 4700: train RMSE: 13.6059, test RMSE: 63.7560, Loss: 289.2528
Epoch 4800: train RMSE: 13.3668, test RMSE: 63.1970, Loss: 212.2537
Epoch 4900: train RMSE: 13.3835, test RMSE: 61.5233, Loss: 160.1872
Epoch 5000: train RMSE: 13.7180, test RMSE: 65.0818, Loss: 526.5771
Epoch 5100: train RMSE: 22.2873, test RMSE: 59.9410, Loss: 1317.8331
Epoch 5200: train RMSE: 17.9827, test RMSE: 64.7171, Loss: 297.7516
Epoch 5300: train RMSE: 17.3548, test RMSE: 61.1212, Loss: 274.8957
Epoch 5400: train RMSE: 16.1121, test RMSE: 68.5883, Loss: 101.1858
Epoch 5500: train RMSE: 14.6710, test RMSE: 68.1638, Loss: 87.0218
Epoch 5600: train RMSE: 14.4215, test RMSE: 68.4792, Loss: 73.2922
Epoch 5700: train RMSE: 14.0843, test RMSE: 71.6664, Loss: 45.6507
Epoch 5800: train RMSE: 14.0319, test RMSE: 68.9918, Loss: 41.6092
Epoch 5900: train RMSE: 13.6696, test RMSE: 73.1099, Loss: 34.1758
```

```
#Visualising the model training and testing plots to check the model
accuracy
plt.plot(timeseries, label='Original Input Data')
plt.plot(rnntrain_plot, label='RNN Training data Predictions')
plt.plot(rnntest_plot, label='RNN Testing data Predictions')
plt.title('RNN Model Prediction')
plt.xlabel("Number of Months")
plt.ylabel("Passengers in 1000's")
```

```
plt.legend()
plt.show()
```



Input the Year and month in YYYY-MM format to obtain the prediction for the number of passengers(in 1000's) from the LSTM and RNN Model for the particular year and month combination

```
from datetime import datetime

# Entered combination
entered_year = 1960
entered_month = 12

# Year and month entered by the user
year, month = map(int, input("Enter a year and month after 1960-12 for
obtaining the prediction using LSTM Model and RNN Model  (in format
'YYYY-MM'): ").split('-'))

# Convert entered year and month to datetime object
entered_date = datetime(entered_year, entered_month, 1)
user_date = datetime(year, month, 1)

# Calculate the number of months between the entered combination and
```

```
the user input
num_months = (user_date.year - entered_date.year) * 12 +
(user_date.month - entered_date.month)

print("Number of months between '1960-12' and the entered
combination:", num_months)
```

```
Enter a year and month after 1960-12 for obtaining the prediction
using LSTM Model and RNN Model  (in format 'YYYY-MM'): 1961-12
Number of months between '1960-12' and the entered combination: 12
```

```
#Prediction using LSTM Model
timeseriesX = np.array(timeseries).astype(np.float32)
timeseriesX = torch.tensor(timeseriesX)

num_forecasted_steps=num_months
sequence_to_plot=timeseriesX.squeeze().numpy()
historical_data=sequence_to_plot[-6:]

forecasted_values=[]

with torch.no_grad():
  for _ in range(num_forecasted_steps):
    historical_data_tensor=torch.as_tensor(historical_data).view(1, -
1, 1).float()
    predicted_value = lstmmodel(historical_data_tensor).squeeze(-1)
    forecasted_values.append(predicted_value[0])
    historical_data = np.roll(historical_data, shift=-1)
    historical_data[-1] = predicted_value
predicted_value = predicted_value.item()
print(f"The predicted passengers for the given input {year:04d}-
{month:02d} using LSTM predcition is: ",predicted_value)
```

```
The predicted passengers for the given input 1961-12 using LSTM
predcition is:  538.1292114257812
```

```
#Prediction using RNN Model
timeseriesX = np.array(timeseries).astype(np.float32)
timeseriesX = torch.tensor(timeseriesX)

rnnnum_forecasted_steps=num_months
rnnsequence_to_plot=timeseriesX.squeeze().numpy()
rnnhistorical_data=rnnsequence_to_plot[-6:]
rnnforecasted_values=[]

with torch.no_grad():
  for _ in range(rnnnum_forecasted_steps):

rnnhistorical_data_tensor=torch.as_tensor(rnnhistorical_data).view(1,
-1, 1).float()
    rnnpredicted_value = rnnmodel(rnnhistorical_data_tensor).squeeze(-
```

```
1)
    rnnforecasted_values.append(rnnpredicted_value.item())
    rnnhistorical_data = np.roll(rnnhistorical_data, shift=-1)
    rnnhistorical_data[-1] = rnnpredicted_value

rnnpredicted_value = rnnpredicted_value.item()
print(f"The predicted passengers for the given input {year:04d}-
{month:02d} using LSTM predcition is: ",rnnpredicted_value)
```

```
The predicted passengers for the given input 1961-12 using LSTM
predcition is:  346.64129638671875
```

[Bonus 5 points] Suggest some things that could be done to improve the results.

Solution: To imporove the results:

1. A bigger dataset is required : LSTM networks usually require lot of data so perhaps the airline data is not big enough.
2. Tuning the Hyper Parameters: Neural networks in general have a huge number of hyperparameters to deal with and small changes can produce noticeable differences.
3. The model can be over fitted and some sort of dropout or regularization can be implemented to help improve the results
4. Experimenting with different model architectures such as adding more layers and more number of units in each layer or using Bi-Directional models that capture information form both future and past time steps
5. Preprocessing of Data= Normalising or scaling the data can help the model converge faster and can avoid numerical instabilities

[Bonus 5 points] Suggest where this could be used in Robotics other than the example given in the beginning.

Solution: the RNN and LSTM models can be implemented to carry out segmentation of data, this can be used by medical robots in classification tasks like tumor detection and disease diagnosis.

Furthermore, the robot can predict its future path and navigate through a place where data is not available.

Also, the robots can learn from demonstration and can replicate complex manipulation and motion tasks.

LSTM and RNN models can also be used to predict future behaviours during interaction with humans and adapt their behavior accordingly.