enpm690-hw2-120172243

February 15, 2024

SAI JAGADEESH MURALIKRISHNAN

120172243

ENPM690 - HOMEWORK 2

1 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**

2 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.

2.1 1.0.0 Array manipulation using numpy

Q1 - Matrix multiplication

[ ]:
```
import numpy as np
```

[ ]:
```
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.array([ 2,  4,  6,  8, 10, 12]).reshape(3,2)
### Starting at 7, stepping by 3
vector2 =np.array([ 7, 10, 13, 16, 19, 22]).reshape(2,3)

### Print vec
# print(vector1, vector2)
print(vector1,vector2)
```

```
### Take product of the two matricies (Matrix product)
prod = np.matmul(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(2,302 , 3).reshape(10,10)
     ### Starting at 35, stepping by 9
     vector2 = np.arange(35,935, 9).reshape(10,10)

     ### Print vec

     print(vector1,"\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.diagflat(vector1[3:10, :7].diagonal())



     ### 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 = vector2[0:7, 3:10]

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


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

     ### Print
     print("\n\n",prod)
```

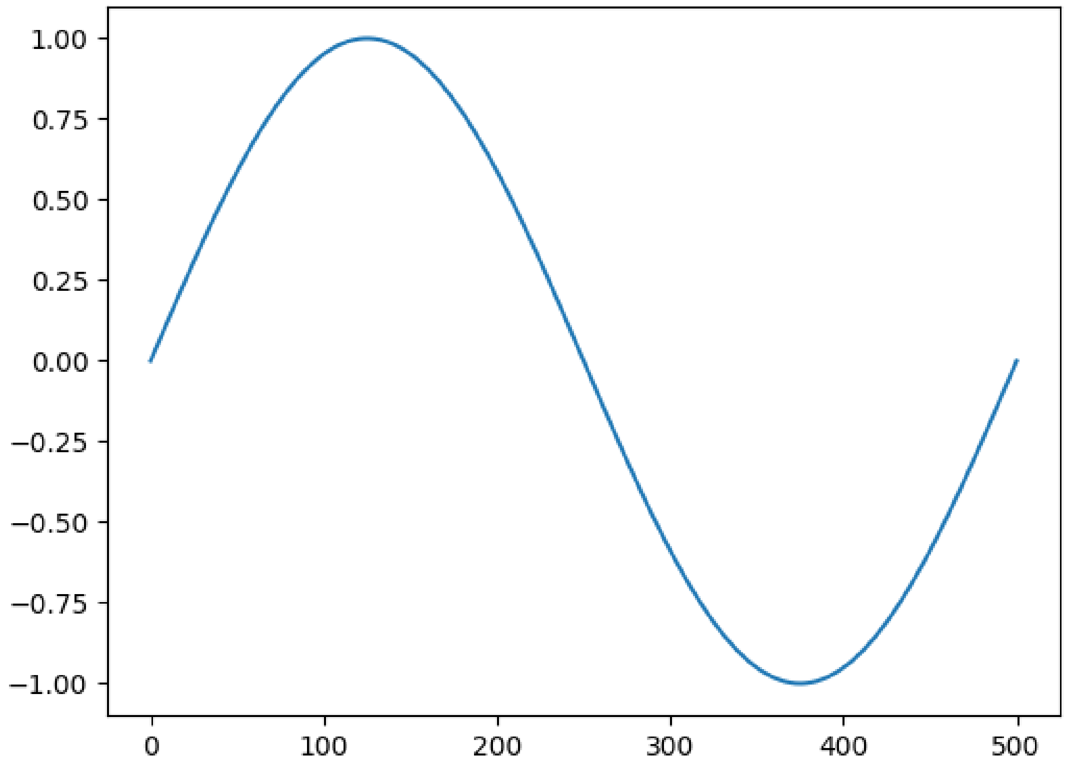
```
[[  2   5   8  11  14  17  20  23  26  29]
 [ 32  35  38  41  44  47  50  53  56  59]
 [ 62  65  68  71  74  77  80  83  86  89]
 [ 92  95  98 101 104 107 110 113 116 119]
 [122 125 128 131 134 137 140 143 146 149]
 [152 155 158 161 164 167 170 173 176 179]
 [182 185 188 191 194 197 200 203 206 209]
 [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  71  80  89  98 107 116]
 [152 161 170 179 188 197 206]
 [242 251 260 269 278 287 296]
 [332 341 350 359 368 377 386]
 [422 431 440 449 458 467 476]
 [512 521 530 539 548 557 566]
 [602 611 620 629 638 647 656]]
```

```
[[  5704   6532   7360   8188   9016   9844  10672]
 [ 19000  20125  21250  22375  23500  24625  25750]
 [ 38236  39658  41080  42502  43924  45346  46768]
 [ 63412  65131  66850  68569  70288  72007  73726]
 [ 94528  96544  98560 100576 102592 104608 106624]
 [131584 133897 136210 138523 140836 143149 145462]
 [174580 177190 179800 182410 185020 187630 190240]]
```

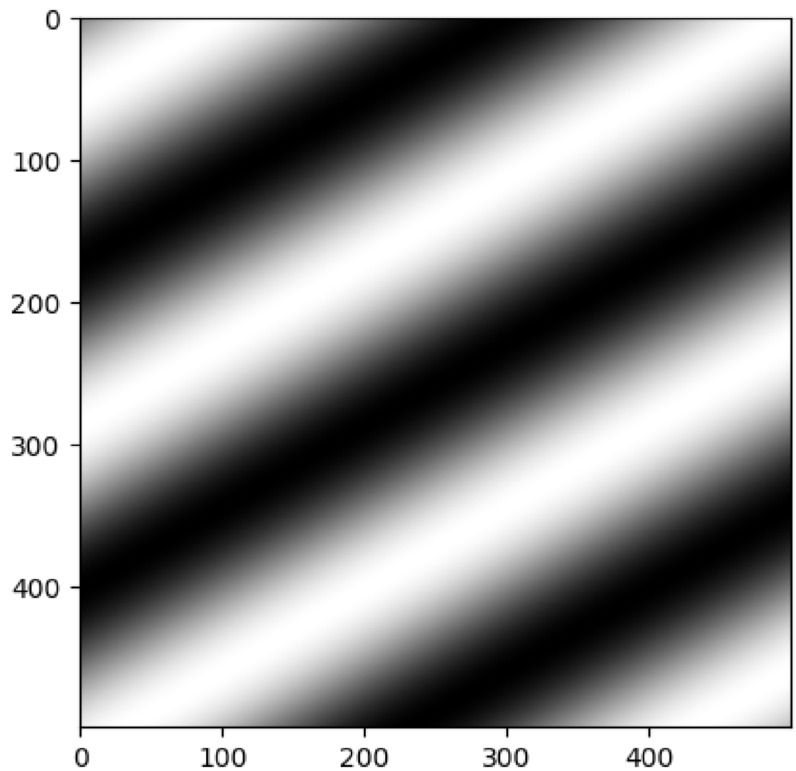
Q3 - Sin wave Sample outputs,

[ ]:
```
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.linspace(0,500,500)

### 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)
x =X*np.cos(phi) + Y*np.sin(phi)
grating =np.sin(2*np.pi*(x/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)
```

```python
#### 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.ravel()
#### Flatten the bottom layer
cars_bottom_flat = cars_bottom_layer.ravel()

#### Print layers
print("\nTop Flattened : ",cars_top_flat,"\nBottom Flattened :␣
↪",cars_bottom_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']
```

[ ]:
```python
import numpy as np

d = np.arange(16).reshape(4, 4)
print(d)
# Horizontal split
d1, d2 = np.hsplit(d, 2)  # Split into two equal parts column-wise
print(d1)
print(d2)

# Vertical split
d3,d4= np.vsplit(d, 2)  # Split into two equal parts row-wise
print(d3)
```

```
print(d4)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

[ ]:
```
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.ravel()
#### Flatten the bottom layer
cars_bottom_flat = cars_bottom_layer.ravel()

#### 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] = cars_top_flat
new_car_list[1::2] = cars_bottom_flat


#### 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.concatenate((cars_top_flat,cars_bottom_flat))

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

2.2 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, x_test = x_train / 255.0, x_test / 255.0
```

```
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([ tf.keras.layers.Flatten(input_shape=(28,␣
  ↪28)), tf.keras.layers.Dense(128, activation='relu'), tf.keras.layers.
  ↪Dense(64, activation='relu'), tf.keras.layers.Dense(10)])

### Complie the model with the adam optimizer and crossentropy loss
### HINT - No One hot encoding
```

```
model.compile( optimizer='adam', loss=tf.keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True), 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 [==============================] - 6s 3ms/step - loss: 0.2383 -
accuracy: 0.9305
Epoch 2/5
1875/1875 [==============================] - 7s 3ms/step - loss: 0.1002 -
accuracy: 0.9690
Epoch 3/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0725 -
accuracy: 0.9776
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0526 -
accuracy: 0.9832
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0432 -
accuracy: 0.9854
```

[ ]:
```
<keras.src.callbacks.History at 0x7b8de7c5a410>
```

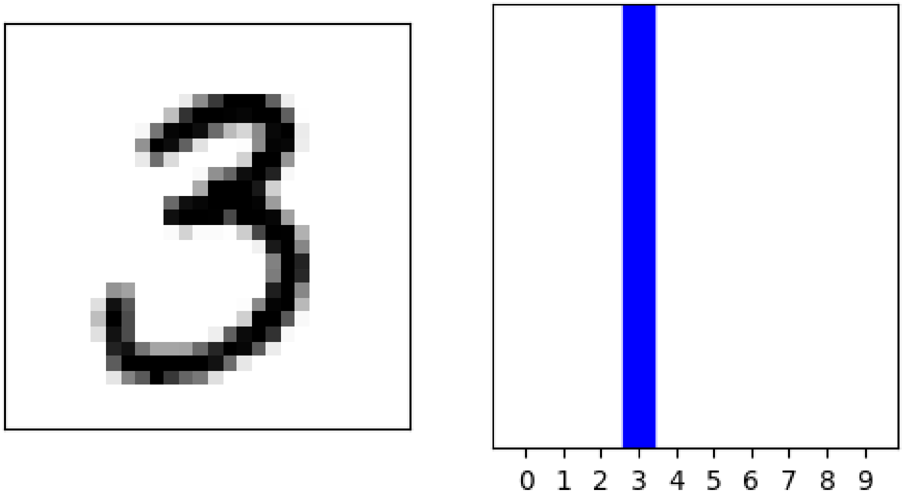
[ ]:
```
### Check the accuracy of the trained model
test_loss, test_acc = model.evaluate(x_test,  y_test, verbose=2)
print('\nTest accuracy:', test_acc)
```

```
313/313 - 1s - loss: 0.0750 - accuracy: 0.9778 - 562ms/epoch - 2ms/step

Test accuracy: 0.9778000116348267
```

[ ]:
```
### 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) ### <---- This is the output of␣
 ↪the model

### Plot a test output
i = 90 ### <---- 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 1ms/step
```



2.3 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)
```

```python
        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: 0.7497772100065817
Epoch 2, Loss: 0.36693694127965837
Epoch 3, Loss: 0.32101490559068313
Epoch 4, Loss: 0.29383779380684977
Epoch 5, Loss: 0.273217272605183
Epoch 6, Loss: 0.2538067406571623
Epoch 7, Loss: 0.2366939038077969
Epoch 8, Loss: 0.22136161107816169
Epoch 9, Loss: 0.2073850411016232
Epoch 10, Loss: 0.19490794614275128
Finished Training
Accuracy on test set: 94.47%
```

3 2.0 Homework 2

90 points

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

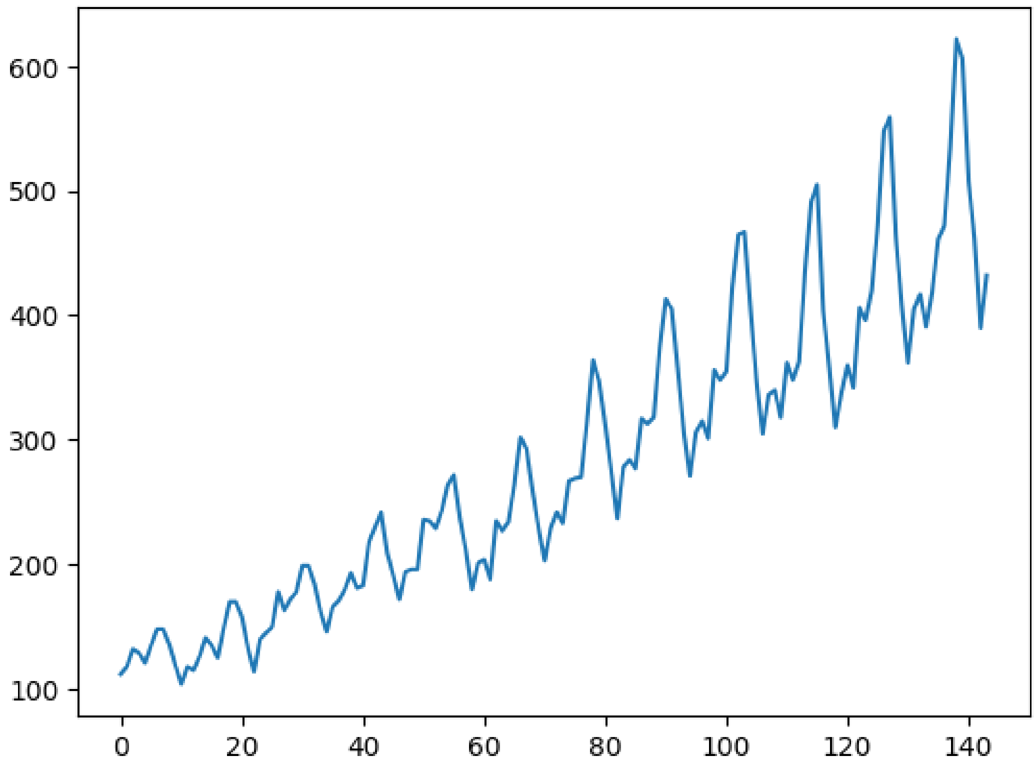
3.1 Graded Section : Deep Learning Approach

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.

[2]:
```
import matplotlib.pyplot as plt
import pandas as pd

# from google.colab import drive
# drive.mount('/content/drive')
# file_name = '/content/drive/My Drive/airline.csv'  # Adjusted dataset path␣
↪for my drive

file_name = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/
↪airline-passengers.csv'  # Adjusted dataset path from url

# Reading data using pandas or csv
df = pd.read_csv(file_name)
```

[3]:
```
# plotting the dataset
timeseries = df[["Passengers"]].values.astype('float32')
# 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

[4]:
```
import numpy as np
from torch.utils.data import TensorDataset, DataLoader
import pandas as pd
import torch



# the data is split into 75% - training and 25% -  testing sets - not normalized
train_size = int(len(timeseries) * 0.75)
test_size = len(timeseries) - train_size
train, test = timeseries[:train_size], timeseries[train_size:]

# Function to create sequences
def create_dataset(dataset, lookback):
```

```
    X, y = [], []
    for i in range(len(dataset)-lookback):
        features = dataset[i:i+lookback]
        target= dataset[i+1:i+lookback+1]
        X.append(features)
        y.append(target)
    return np.array(X), np.array(y)


# lookback period for each sequence
lookback_len = 4
X_train, y_train = create_dataset(train, lookback_len)
X_test, y_test = create_dataset(test, lookback_len)

# Convert to PyTorch tensors from Numpy array
X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.
 ↪tensor(y_train, dtype=torch.float32)
X_test, y_test = torch.tensor(X_test, dtype=torch.float32), torch.
 ↪tensor(y_test, dtype=torch.float32)

# DataLoader for training data
train_data = TensorDataset(X_train, y_train)
train_dataloader = DataLoader(train_data, shuffle=True, batch_size=8)

# DataLoader for testing data
test_dataset = TensorDataset(X_test, y_test)
test_dataloader = DataLoader(test_dataset, shuffle=True, batch_size=8)
```

2. 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

[5]:
```
import torch.optim as optim
import torch.nn as nn



# Defining LSTM model

class LSTMModel(nn.Module):
    def __init__(self, input_size , hidden_size, num_layers,output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
 ↪batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
        hidden_state = torch.zeros(self.num_layers, x.size(0), hidden_size)
        cell_state = torch.zeros(self.num_layers, x.size(0), hidden_size)
        x, _ = self.lstm(x, (hidden_state, cell_state))
        x = self.fc(x)
        return x

# Defining RNN model

class RNNModel(nn.Module):
    def __init__(self, input_size , hidden_size, num_layers, output_size):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.RNN(input_size, hidden_size, num_layers,␣
 ↪batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        hidden_state = torch.zeros(self.num_layers, x.size(0), hidden_size)
        # cell_state = torch.zeros(self.num_layers, x.size(0), hidden_size) #␣
 ↪No cell state in RNN
        x, _ = self.lstm(x, hidden_state)
        x = self.fc(x)
        return x

#Training the models
learning_rate = 0.001     # learning rate
n_iters = 2001           # epochs

#Hyperprameters
input_size = 1
hidden_size = 50
num_layers = 1
output_size = 1

# Model Intialization
lstm_model = LSTMModel(input_size, hidden_size, num_layers, output_size)
rnn_model = RNNModel(input_size, hidden_size, num_layers, output_size)

#Loss Function
Loss_fuction = nn.MSELoss()

# Model optimizers
lstm_optimizer = optim.Adam(lstm_model.parameters(), lr=learning_rate)
rnn_optimizer = optim.Adam(rnn_model.parameters(), lr=learning_rate)
```

```python
def training(n_iters, model, train_loader, loss_function, optimizer):
    for epoch in range(n_iters):
        model.train()

        for  i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = loss_function(outputs, labels)
            loss.backward()
            optimizer.step()


        if epoch % 100 == 0:
            print (f'Epoch [{epoch}/{n_iters-1}], Training Loss: {loss.item():.
 ↪4f}')


print('Started Training for LSTM Model...')
training(n_iters, lstm_model, train_dataloader, Loss_fuction, lstm_optimizer)
print('Finished Training for LSTM Model')

print('Started Training for RNN Model...')
training(n_iters, rnn_model, train_dataloader, Loss_fuction, rnn_optimizer)
print('Finished Training for RNN Model')
```

```
Started Training for LSTM Model…
Epoch [0/2000], Training Loss: 56857.5352
Epoch [100/2000], Training Loss: 44488.8789
Epoch [200/2000], Training Loss: 38635.5391
Epoch [300/2000], Training Loss: 15707.9209
Epoch [400/2000], Training Loss: 18013.9023
Epoch [500/2000], Training Loss: 15918.0059
Epoch [600/2000], Training Loss: 1383.2938
Epoch [700/2000], Training Loss: 4961.6704
Epoch [800/2000], Training Loss: 864.4319
Epoch [900/2000], Training Loss: 1391.5520
Epoch [1000/2000], Training Loss: 488.5534
Epoch [1100/2000], Training Loss: 960.0272
Epoch [1200/2000], Training Loss: 439.7687
Epoch [1300/2000], Training Loss: 451.4600
Epoch [1400/2000], Training Loss: 650.0129
Epoch [1500/2000], Training Loss: 663.6610
Epoch [1600/2000], Training Loss: 594.7778
Epoch [1700/2000], Training Loss: 496.6418
Epoch [1800/2000], Training Loss: 373.5475
Epoch [1900/2000], Training Loss: 401.3045
```

```
Epoch [2000/2000], Training Loss: 422.9029
Finished Training for LSTM Model
Started Training for RNN Model…
Epoch [0/2000], Training Loss: 88161.1328
Epoch [100/2000], Training Loss: 21637.6758
Epoch [200/2000], Training Loss: 20948.1133
Epoch [300/2000], Training Loss: 7170.3491
Epoch [400/2000], Training Loss: 6092.5171
Epoch [500/2000], Training Loss: 13496.2725
Epoch [600/2000], Training Loss: 1161.4777
Epoch [700/2000], Training Loss: 2420.1511
Epoch [800/2000], Training Loss: 402.0431
Epoch [900/2000], Training Loss: 1274.4594
Epoch [1000/2000], Training Loss: 423.2807
Epoch [1100/2000], Training Loss: 512.4075
Epoch [1200/2000], Training Loss: 653.1232
Epoch [1300/2000], Training Loss: 506.6119
Epoch [1400/2000], Training Loss: 594.9711
Epoch [1500/2000], Training Loss: 796.0073
Epoch [1600/2000], Training Loss: 307.6074
Epoch [1700/2000], Training Loss: 810.5856
Epoch [1800/2000], Training Loss: 721.8411
Epoch [1900/2000], Training Loss: 384.6929
Epoch [2000/2000], Training Loss: 656.5779
Finished Training for RNN Model
```

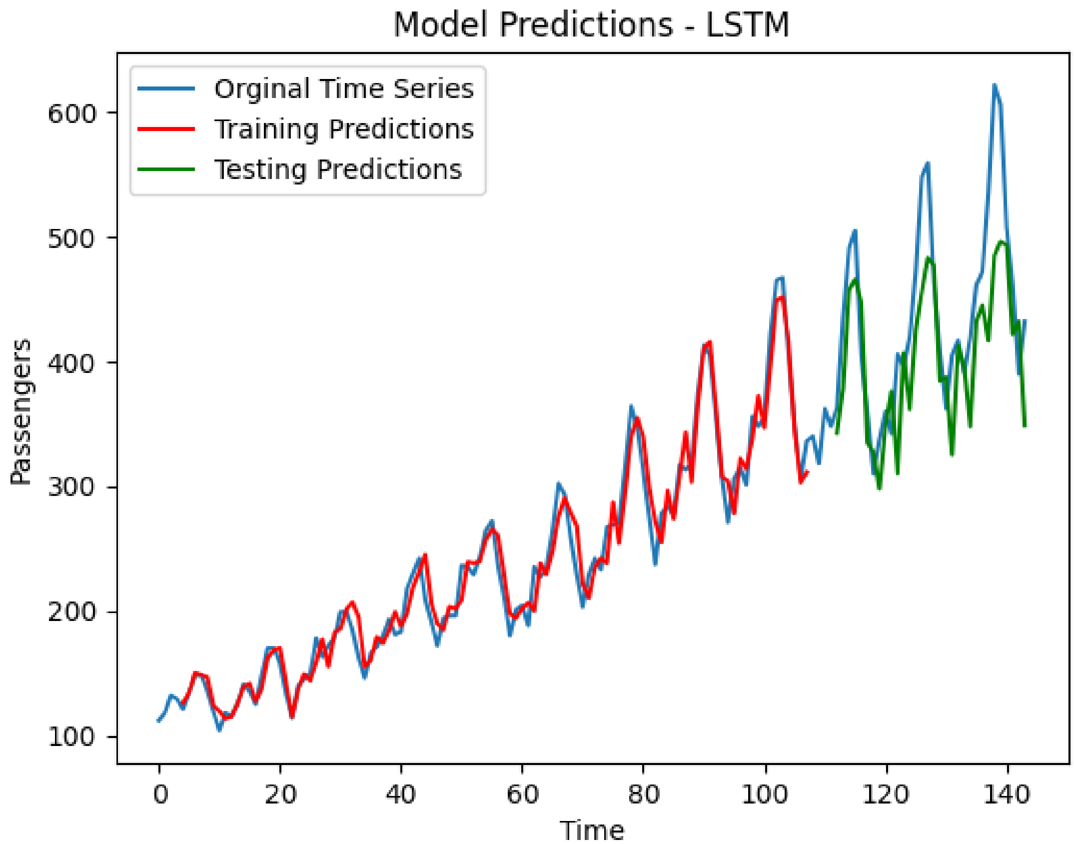
3. Evaluate and Compare the result using proper metric. Justify the metrics used.

[7]:
```python
import matplotlib.pyplot as plt
import numpy as np

#Model evaluation and comparisons using proper metrics - RMSE, MAE and MAPE
def Model_evaluation(model, dataloader, loss_func):
    model.eval()
    net_loss = 0.0
    net_mae = 0.0
    net_mape = 0.0
    net_samples = 0
    with torch.no_grad():
        for inputs, labels in dataloader:
            outputs = model(inputs)
            loss = loss_func(outputs, labels)
            mae = torch.mean(torch.abs(outputs - labels))
            mape = torch.mean(torch.abs((outputs - labels) / labels)) * 100

            batch_size = inputs.shape[0]
            net_loss += loss.item() * batch_size
            net_mae += mae.item() * batch_size
```

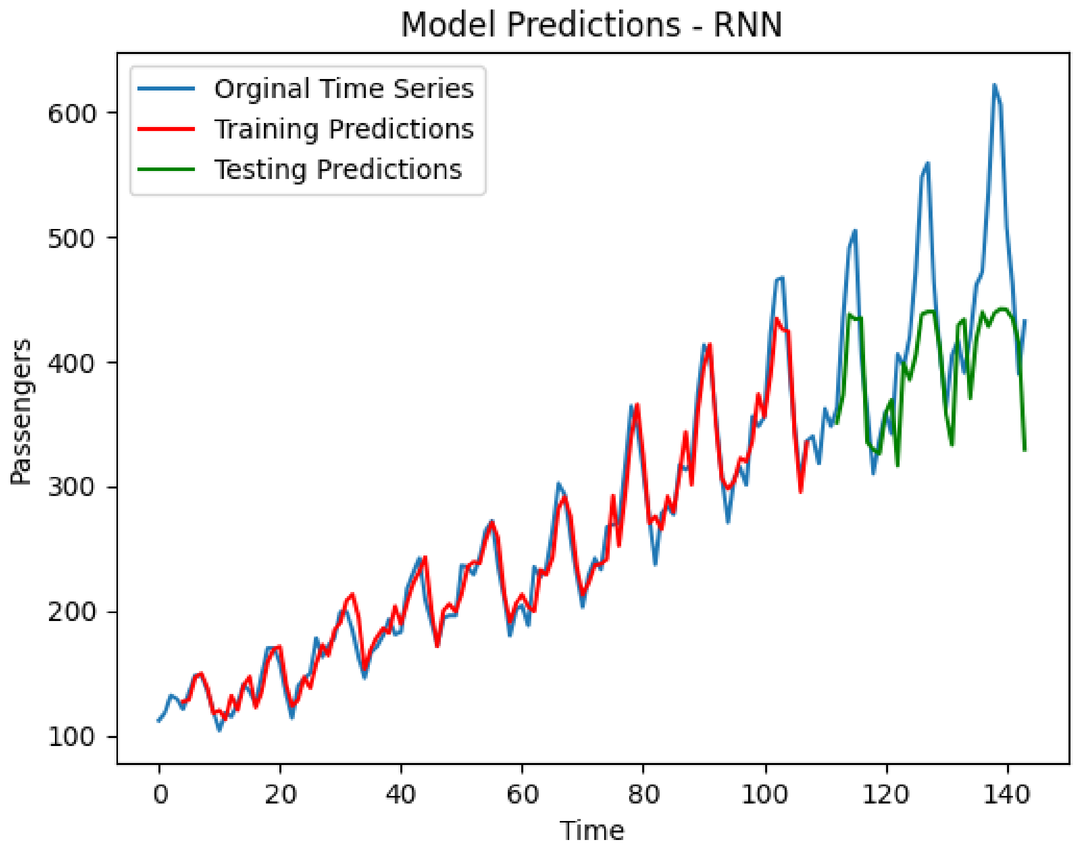
```
            net_mape += mape.item() * batch_size
            net_samples += batch_size

    MSE = net_loss / net_samples
    RMSE = np.sqrt(MSE)
    MAE = net_mae / net_samples
    MAPE = net_mape / net_samples
    return RMSE, MAE, MAPE


LSTM_metrics_training = Model_evaluation(lstm_model, train_dataloader,␣
  ↪Loss_fuction)
RNN_metrics_training = Model_evaluation(rnn_model, train_dataloader,␣
  ↪Loss_fuction)
print(f'LSTM Model - Evaluation metrics after training\nRMSE:␣
  ↪{LSTM_metrics_training[0]}\nMAE: {LSTM_metrics_training[1]}\nMAPE:␣
  ↪{LSTM_metrics_training[2]}%\n')
print(f'RNN Model - Evaluation metrics after training\nRMSE:␣
  ↪{RNN_metrics_training[0]}\nMAE: {RNN_metrics_training[1]}\nMAPE:␣
  ↪{RNN_metrics_training[2]}%\n')

LSTM_metrics_testing = Model_evaluation(lstm_model, test_dataloader,␣
  ↪Loss_fuction)
RNN_metrics_testing = Model_evaluation(rnn_model, test_dataloader, Loss_fuction)

print(f'LSTM Model - Evaluation metrics after testing\nRMSE:␣
  ↪{LSTM_metrics_testing[0]}\nMAE: {LSTM_metrics_testing[1]}\nMAPE:␣
  ↪{LSTM_metrics_testing[2]}%\n')
print(f'RNN Model - Evaluation metrics after testing\nRMSE:␣
  ↪{RNN_metrics_testing[0]}\nMAE: {RNN_metrics_testing[1]}\nMAPE:␣
  ↪{RNN_metrics_testing[2]}%')

# Plotting both training and testing data to show the comparison
def plot_predictions(model, X_train, X_test, color_scheme):
    with torch.no_grad():
        predictions_train = model(X_train)[:, -1, :]
        predictions_test = model(X_test)[:, -1, :]

        plot_data_train = np.ones_like(timeseries) * np.nan
        plot_data_test = np.empty_like(timeseries) * np.nan
        plot_data_train[lookback_len:train_size] = predictions_train.cpu().
  ↪numpy()
        plot_data_test[train_size + lookback_len:] = predictions_test.cpu().
  ↪numpy()

        plt.plot(timeseries, label='Orginal Time Series')
```

```
        plt.plot(plot_data_train, color=color_scheme[0], label='Training␣
↪Predictions')
        plt.plot(plot_data_test, color=color_scheme[1], label='Testing␣
↪Predictions')
        if model == lstm_model:
          plt.title('Model Predictions - LSTM')
        else:
          plt.title('Model Predictions - RNN')

        plt.xlabel('Time')
        plt.ylabel('Passengers')
        plt.legend()
        plt.show()




# Plotting for LSTM model
plot_predictions(lstm_model, X_train, X_test, ['red', 'green'])

# Plotting for RNN model
plot_predictions(rnn_model, X_train, X_test, ['red', 'green'])
```

```
LSTM Model - Evaluation metrics after training
RMSE: 21.047735490625907
MAE: 16.39998568021334
MAPE: 7.1823476277864895%

RNN Model - Evaluation metrics after training
RMSE: 20.23670707204464
MAE: 15.861563242398775
MAPE: 7.024677496690017%

LSTM Model - Evaluation metrics after testing
RMSE: 66.94184378412915
MAE: 51.576223373413086
MAPE: 11.1041841506958%

RNN Model - Evaluation metrics after testing
RMSE: 69.93887145033221
MAE: 54.333516120910645
MAPE: 11.584525346755981%
```



*Kindly look at the next page to see RNN model predictions plotting and justifications



Justification for the usage of metric:

Root Mean Squared Error (RMSE): * RMSE is a simple score that tells us how far off our predictions are from what actual data is.

- Here Mean squared error loss function is used and by accumulating the net loss and dividing it by the length of the dataloader we gwt the total MSE when it is used with numpy square root function (RMSE = np.sqrt(MSE))we get RSME.
- It's especially useful when we're trying to forecast things over time, like weather or stock prices, using models called LSTM and RNN.
- The higher the RMSE, the more mistakes the model is making but in our case it is considerably less.
- By looking at RMSE, we can easily compare these models to see which one is better at making predictions as it is showing that LSTM can perform better than RNN after testing.

Mean Absolute Percentage Error (MAPE): * MAPE measures the size of the error in percentage terms. It is calculated as the average of the absolute percentage errors of the predictions made.

- Because it's in percentage, it's simple to compare how different are predictions, no matter whatever the size of the data is.

- MAPE focuses on how big errors are compared to actual values, which is great for understanding the impact of mistakes in things specially like time based forecasts.

Mean Absolute Error (MAE): * This metric measures the average magnitude of the errors in a set of predictions, without considering their direction.

- It's calculated as the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight.

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

To enhance time series prediction results, consider refining the model and exploring diverse modeling techniques.

- Start by fine-tuning hyperparameters such as learning rate and batch size, and experiment with model architectures by adjusting the number of layers.
- Regularization techniques like dropout and L2 regularization can help prevent overfitting in this case.
- Furthermore, explore advanced models like GRU for efficiency, Temporal Convolutional Networks (TCN) for capturing long-range dependencies, and transformers for their ability to handle complex patterns.
- Combining methods, combining multiple models, can also enhance prediction accuracy by using the strengths in various approaches.

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

- **Human-Robot Interaction:** Social robots, healthcare assistants, and collaborative robots (cobots) can adjust their actions based on anticipated human movements which in case foreseeing the future.
- **Energy Management:** In solar-powered robots or those with limited battery capacity, accurate predictions help manage energy resources effectively again uses forecasting using time-series.
- **Autonomous Navigation:** Beyond simple trajectory prediction, deep learning models can analyze sequences of data from various sensors (LIDAR, GPS, IMU, cameras) to predict and navigate complex environments autonomously. This is essential for robots in exploration, delivery drones, and autonomous vehicles, allowing them to adapt to dynamic conditions and obstacles.
- **Predictive Maintenance:** By analyzing time-series data from sensors monitoring the condition of robotic components, deep learning models can predict when parts may fail or require maintenance. This application is crucial for industrial robots operating in manufacturing lines, ensuring minimal downtime and optimal performance.
- **Environmental Monitoring and Adaptation:** Time series forecasting aids in adapting to changing environmental conditions. Agricultural robots, weather stations, and underwater exploration robots use predictions to adjust their actions based on upcoming weather, soil moisture, or ocean currents using deep learning methods.