690tutorial-and-hw2

February 16, 2024

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

[130]:
```
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)
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)
```

```
[[ 78  96 114]
 [170 212 254]
 [262 328 394]]
```

Q2 - Diagonals

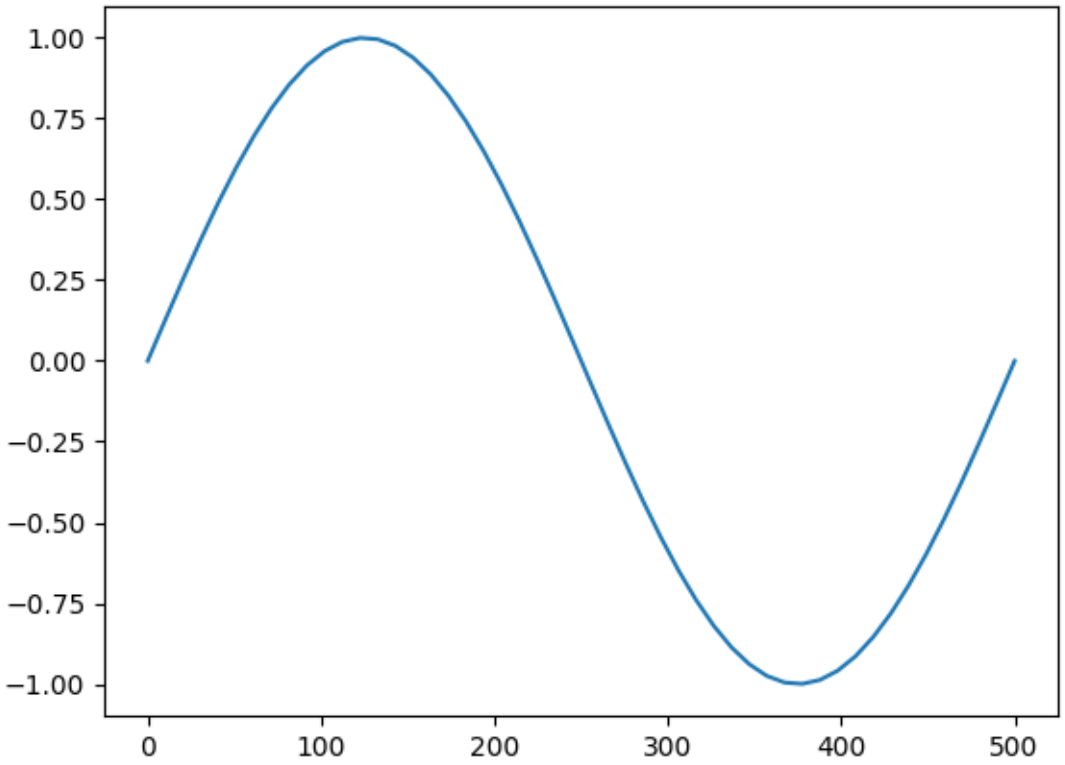
[131]:
```python
import numpy as np

# Create numpy arrays
vector1 = np.arange(2, 302, 3).reshape(10, 10)
vector2 = np.arange(35, 935, 9).reshape(10, 10)

# Obtain the diagonal matrix of vector1
vector1_offset_diagonal = np.diag(vector1[3:, :7])

# Obtain a 7x7 matrix from vector2
vector2_offset_diagonal = vector2[:7, 3:]

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

# Print diagonal matrices
print("Diagonal Matrix of vector1:")
print(vector1_offset_diagonal)
print("7x7 Matrix from vector2:")
print(vector2_offset_diagonal)

# Print product
print("Product of the two diagonal matrices:")
print(prod)
```

```
Diagonal Matrix of vector1:
[ 92 125 158 191 224 257 290]
7x7 Matrix from vector2:
[[ 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]]
Product of the two diagonal matrices:
[527044 539077 551110 563143 575176 587209 599242]
```

Q3 - Sin wave Sample outputs,

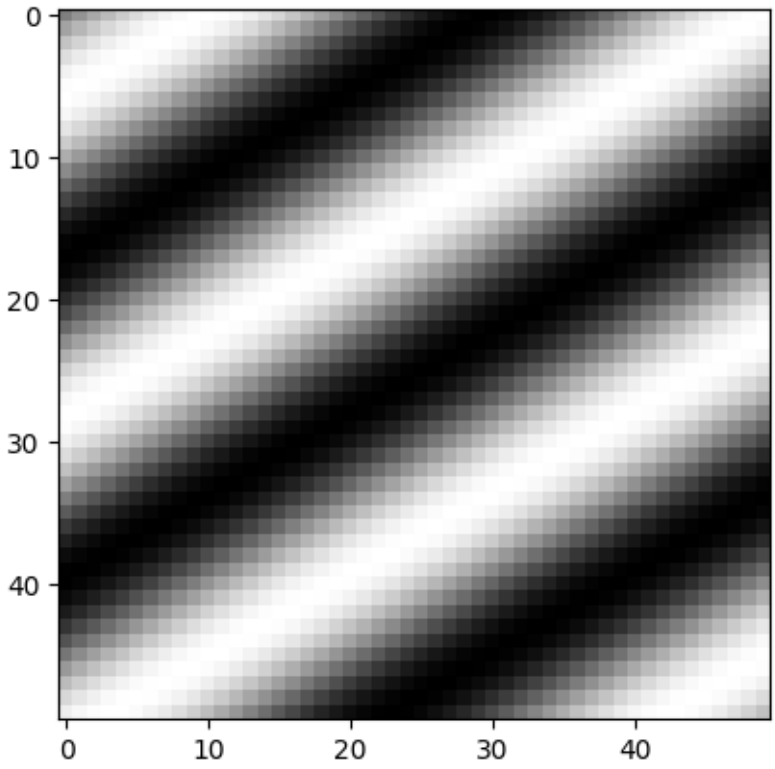
[132]:
```python
import matplotlib.pyplot as plt
### Create a time matrix that evenly samples a sine wave at a frequency of 1Hz
```

```
### Starting at time step T = 0
### End at time step T = 500
time = np.linspace(0, 500, 50)

### 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()
```



[133]:
```
#### 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_prime = X * np.cos(phi) + Y * np.sin(phi)
grating = np.sin(2 * np.pi * (x_prime / 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

[134]:
```
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_arr = np.array(cars)[np.newaxis, np.newaxis, :]
cars_3d = np.repeat(cars_arr, 2, axis=0)
```

```
#### 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.reshape(-1)
#### Flatten the bottom layer
cars_bottom_flat = cars_bottom_layer.reshape(-1)

#### 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' 'Pilot'
  'Legend' 'Beat' 'FR-V' 'HR-V' 'Shuttle']]
Bottom Layer
```

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

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


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

2.2 1.0.1 Basics tensorflow

Helper functions

[135]:
```
import tensorflow as tf
from keras.utils import to_categorical
```

[136]:
```
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

[137]:
```
## 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 / 245.0, x_test / 225.0
```

[138]:
```
## Create a model with the following architecture Flatten -> Dense(128, relu)␣
 ↪-> Dense(64,relu) -> outpuLayer(size=10)

### Complie the model with the adam optimizer and crossentropy loss
### HINT - No One hot encoding
# Import necessary modules
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense

model = Sequential()
# Create the model
model = Sequential()
model.add(Flatten(input_shape=(28, 28))) # Flatten input layer to suit our data
model.add(Dense(128, activation='relu')) # Add dense layer with 128 neurons and␣
 ↪relu activation
model.add(Dense(64, activation='relu')) # Add dense layer with 64 neurons and␣
 ↪relu activation
model.add(Dense(10)) # Output layer with 10 neurons (for 10 classes)

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

[127]:
```
### Train the model on the train data for 5 epochs
model.fit(x_train, y_train, epochs=5, batch_size=4)
```

```
Epoch 1/5
15000/15000 [==============================] - 46s 3ms/step - loss: 2.3091 -
accuracy: 0.1282
Epoch 2/5
15000/15000 [==============================] - 45s 3ms/step - loss: 2.3026 -
accuracy: 0.1217
Epoch 3/5
```

```
15000/15000 [==============================] - 43s 3ms/step - loss: 2.3026 -
accuracy: 0.1217
Epoch 4/5
15000/15000 [==============================] - 42s 3ms/step - loss: 2.3026 -
accuracy: 0.1217
Epoch 5/5
15000/15000 [==============================] - 42s 3ms/step - loss: 2.3026 -
accuracy: 0.1217
```

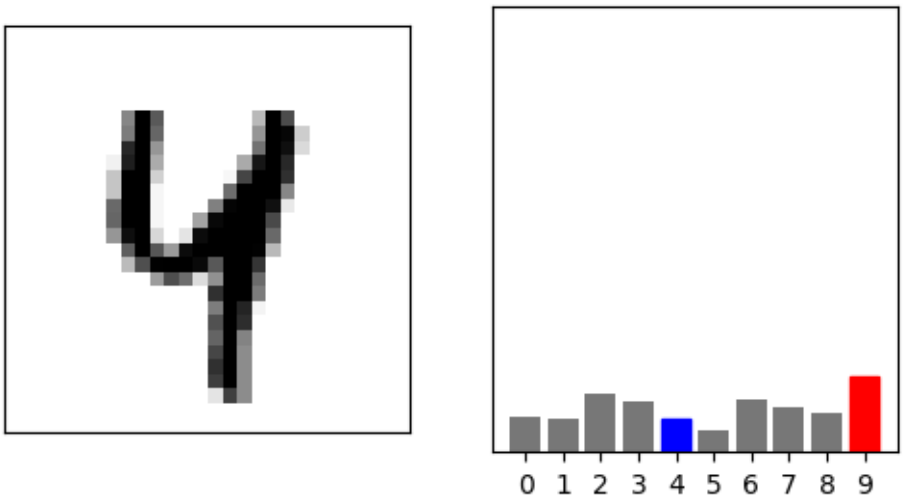
[127]:
```
<keras.src.callbacks.History at 0x7e7442fa4dc0>
```

[140]:
```
### 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: 9.5955 - accuracy: 0.1052 - 754ms/epoch - 2ms/step

Test accuracy: 0.10520000010728836
```

[148]:
```
### Convert the above model to a probabilistic model with a softmax as the␣
 ↪output layer
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense

# Assuming you have already defined your model as 'model'
# Access the layers of the model
layers = model.layers

# Create the probability model with softmax as the output layer
probability_model = Model(inputs=layers[0].input, outputs=Dense(units=10,␣
 ↪activation='softmax')(layers[-2].output))

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

### Plot a test output
i = 42
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()
```

```
313/313 [==============================] - 1s 2ms/step
```



2.3 1.0.2 Basic Pytorch Tutorial

[149]:
```
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%
```

3 2.0 Homework 2

90 points

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

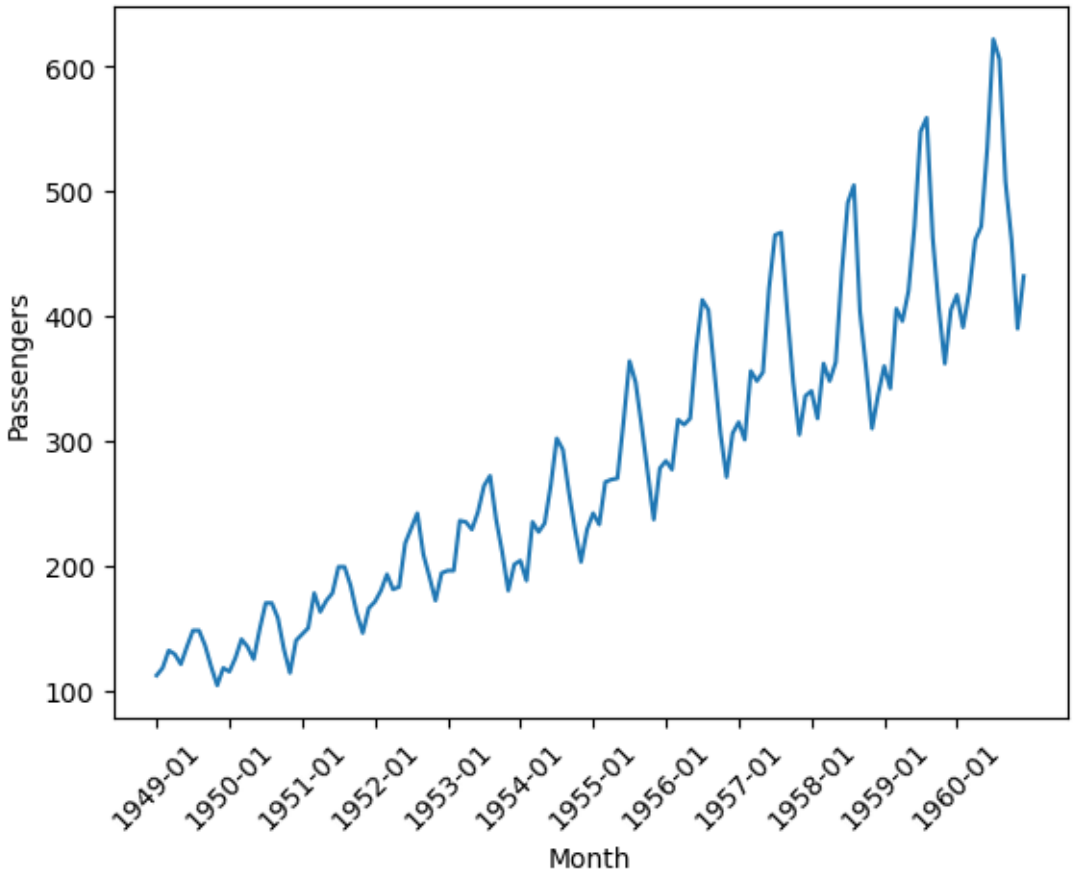
3.1 Graded Section : Deep Learning Approach

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.

[150]:
```
import matplotlib.pyplot as plt
import pandas as pd
file_name = '/airline-passengers.csv' # dataset path
# Reading data using pandas or csv
df = pd.read_csv(file_name)
```

[151]:
```
# plotting the dataset
timeseries = df[["Passengers"]].values.astype('float32')
# plotting the dataset
plt.plot(timeseries)
plt.xlabel('Month')
plt.ylabel('Passengers')
plt.xticks(range(0, len(timeseries), 12), df["Month"][::12], rotation=45)
plt.show()

import matplotlib.pyplot as plt
```



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

[152]:
```
# Write your code here for the dataloader in Pytorch
import numpy as np
import torch
import torch.nn as nn
from torch.autograd import Variable
from sklearn.preprocessing import MinMaxScaler
from torch.utils.data import Dataset, DataLoader

# Load the data from the CSV file
df = pd.read_csv('/airline-passengers.csv')

# Create the Airline Dataset class
class Airline(Dataset):
    def __init__(self):
        self.df = df
        self.number_of_samples = len(df)
```

```
        df['Month'] = df.iloc[:, 0].values
        self.x_data = torch.from_numpy(pd.to_datetime(df['Month']).dt.month.
 ↪to_numpy()).float()
        self.y_data = torch.from_numpy(df['Passengers'].to_numpy()).float()
        print(self.x_data)
        print(self.y_data, end = '\n\n')

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.number_of_samples
```

[153]:
```
airline = Airline()

# Load the data with DataLoader
train_loader = DataLoader(dataset=airline, batch_size=12, shuffle=True)

dataiter = iter(train_loader)
data = next(dataiter)
inputs, targets = data

print("Shape of inputs and targets from the first batch in the DataLoader:")

print("Inputs shape (features):", inputs.shape)
print("Targets shape (labels):", targets.shape)
```

```
tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,  1.,  2.,
         3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,  1.,  2.,  3.,  4.,
         5.,  6.,  7.,  8.,  9., 10., 11., 12.,  1.,  2.,  3.,  4.,  5.,  6.,
         7.,  8.,  9., 10., 11., 12.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,
         9., 10., 11., 12.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.,
        11., 12.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,
         1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,  1.,  2.,
         3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,  1.,  2.,  3.,  4.,
         5.,  6.,  7.,  8.,  9., 10., 11., 12.,  1.,  2.,  3.,  4.,  5.,  6.,
         7.,  8.,  9., 10., 11., 12.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,
         9., 10., 11., 12.])
tensor([112., 118., 132., 129., 121., 135., 148., 148., 136., 119., 104., 118.,
        115., 126., 141., 135., 125., 149., 170., 170., 158., 133., 114., 140.,
        145., 150., 178., 163., 172., 178., 199., 199., 184., 162., 146., 166.,
        171., 180., 193., 181., 183., 218., 230., 242., 209., 191., 172., 194.,
        196., 196., 236., 235., 229., 243., 264., 272., 237., 211., 180., 201.,
        204., 188., 235., 227., 234., 264., 302., 293., 259., 229., 203., 229.,
        242., 233., 267., 269., 270., 315., 364., 347., 312., 274., 237., 278.,
        284., 277., 317., 313., 318., 374., 413., 405., 355., 306., 271., 306.,
        315., 301., 356., 348., 355., 422., 465., 467., 404., 347., 305., 336.,
```

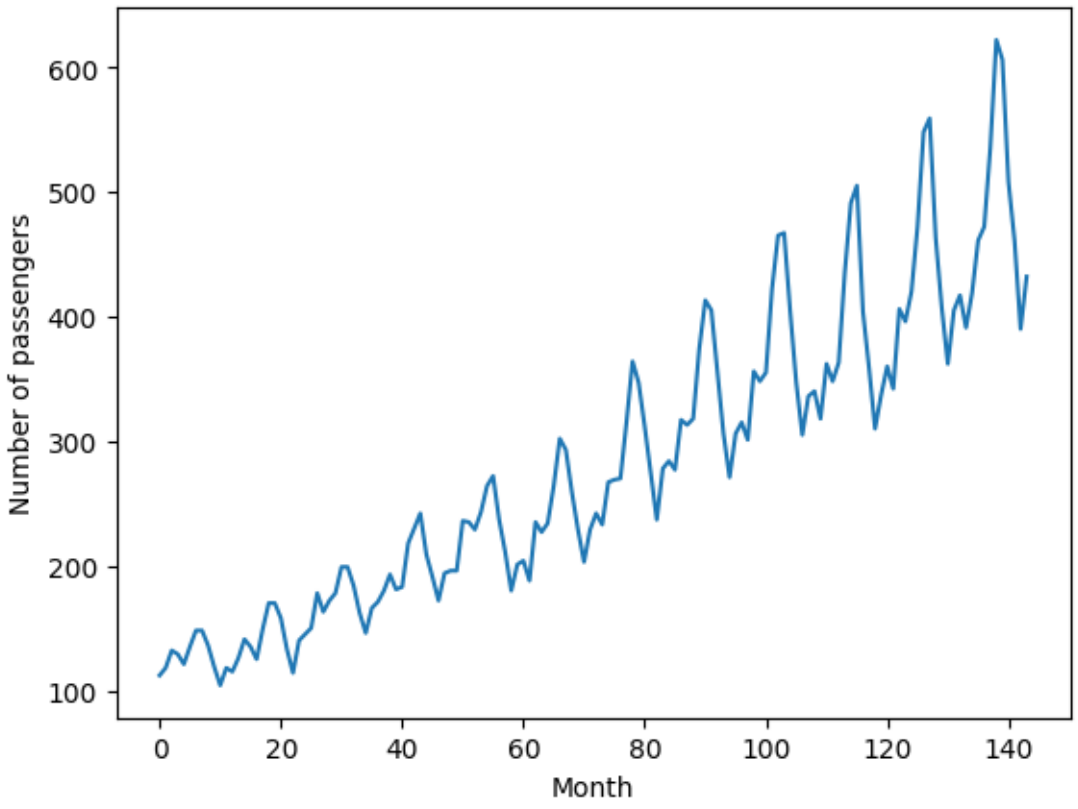
```
        340., 318., 362., 348., 363., 435., 491., 505., 404., 359., 310., 337.,
        360., 342., 406., 396., 420., 472., 548., 559., 463., 407., 362., 405.,
        417., 391., 419., 461., 472., 535., 622., 606., 508., 461., 390., 432.])

Shape of inputs and targets from the first batch in the DataLoader:
Inputs shape (features): torch.Size([12])
Targets shape (labels): torch.Size([12])
```

[154]:
```
from sklearn.model_selection import train_test_split
df.head()
train, test = train_test_split(df, test_size=0.1)
```

[155]:
```
train.shape
```

[155]:
```
(129, 2)
```

[156]:
```
training_set = pd.read_csv('/airline-passengers.csv')

training_set = training_set.iloc[:,1:2].values

plt.xlabel('Month')
plt.ylabel('Number of passengers')
plt.plot(training_set)
plt.show()
```



[157]:
```
def sequence(data, seq_length):
    x = []
    y = []

    for i in range(len(data)-seq_length-1):
        _x = data[i:(i+seq_length)]
        _y = data[i+seq_length]
        x.append(_x)
        y.append(_y)

    return np.array(x),np.array(y)

scaler = MinMaxScaler()
training_data = scaler.fit_transform(training_set)

seq_length = 12
x, y = sequence(training_data, seq_length)

train_size = int(len(y) * 0.9)
test_size = len(y) - train_size


dataX = Variable(torch.Tensor(x))
dataY = Variable(torch.Tensor(y))

trainX = dataX[:train_size]
trainY = dataY[:train_size]

testX = dataX[train_size:]
testY = dataY[train_size:]

print("Data Information:")
print("Training data:", training_data.shape)
print("Data X:", dataX.shape)
print("Data Y:", dataY.shape)
print("Train X:", trainX.shape)
print("Train Y:", trainY.shape)
```

```
Data Information:
Training data: (144, 1)
Data X: torch.Size([131, 12, 1])
Data Y: torch.Size([131, 1])
Train X: torch.Size([117, 12, 1])
Train Y: torch.Size([117, 1])
```

[158]:
```
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).
```

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

[159]:
```
# write your code here for the custom model creating for both the methods
class LSTMModel(nn.Module):

    def _init_(self, input_size, hidden_size, num_layers, num_classes):
        super(LSTMModel, self)._init_()


        self.num_layers = num_layers
        self.hidden_size = hidden_size

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
 ↪batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

        def forward(self, x):
        # Initialize hidden state with zeros
          h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.
 ↪device)
          c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.
 ↪device)

         # Forward propagate LSTM
          out, _ = self.lstm(x, (h0, c0))

        # Decode the hidden state of the last time step
          out = self.fc(out[:, -1, :])
          return out
```

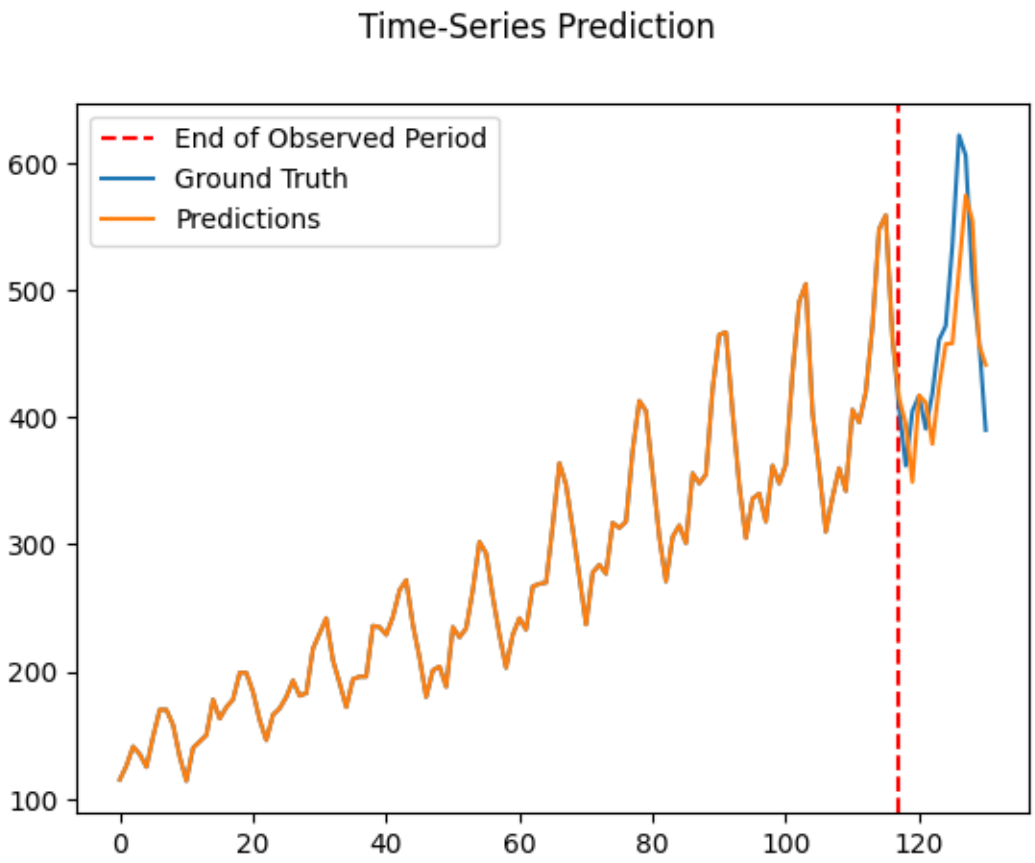
[160]:
```
num_epochs = 2000
learning_rate = 0.01

input_size = 1
hidden_size = 2
num_layers = 1
num_classes = 1

lstm = LSTM(num_classes, input_size, hidden_size, num_layers)
```

```
criterion = torch.nn.MSELoss()    # mean-squared error for regression
optimizer = torch.optim.Adam(lstm.parameters(), lr=learning_rate)

# Train the model
for epoch in range(num_epochs):
    outputs = lstm(trainX)
    optimizer.zero_grad()

    # obtain the loss function
    loss = criterion(outputs, trainY)

    loss.backward()

    optimizer.step()
    if epoch % 100 == 0:
      print("Epoch: %d, loss: %.5f" % (epoch, loss.item()))
```

```
Epoch: 0, loss: 1.33119
Epoch: 100, loss: 0.02229
Epoch: 200, loss: 0.00706
Epoch: 300, loss: 0.00539
Epoch: 400, loss: 0.00513
Epoch: 500, loss: 0.00486
Epoch: 600, loss: 0.00457
Epoch: 700, loss: 0.00429
Epoch: 800, loss: 0.00404
Epoch: 900, loss: 0.00383
Epoch: 1000, loss: 0.00366
Epoch: 1100, loss: 0.00355
Epoch: 1200, loss: 0.00347
Epoch: 1300, loss: 0.00342
Epoch: 1400, loss: 0.00338
Epoch: 1500, loss: 0.00335
Epoch: 1600, loss: 0.00332
Epoch: 1700, loss: 0.00329
Epoch: 1800, loss: 0.00326
Epoch: 1900, loss: 0.00323
```

[161]:
```
lstm.eval()

# Predict on the test data
with torch.no_grad():
    all_predict = lstm(testX)

# Convert predictions and ground truth to numpy arrays
```

```
data_predict = sc.inverse_transform(np.row_stack([trainY.numpy(), all_predict.
 ↪numpy()]))
dataY_plot = sc.inverse_transform(dataY.numpy())

# Plot the predictions and ground truth
plt.axvline(x=train_size, c='r', linestyle='--', label='End of Observed Period')
plt.plot(dataY_plot, label='Ground Truth')
plt.plot(data_predict, label='Predictions')
plt.suptitle('Time-Series Prediction')
plt.legend()
plt.show()
```

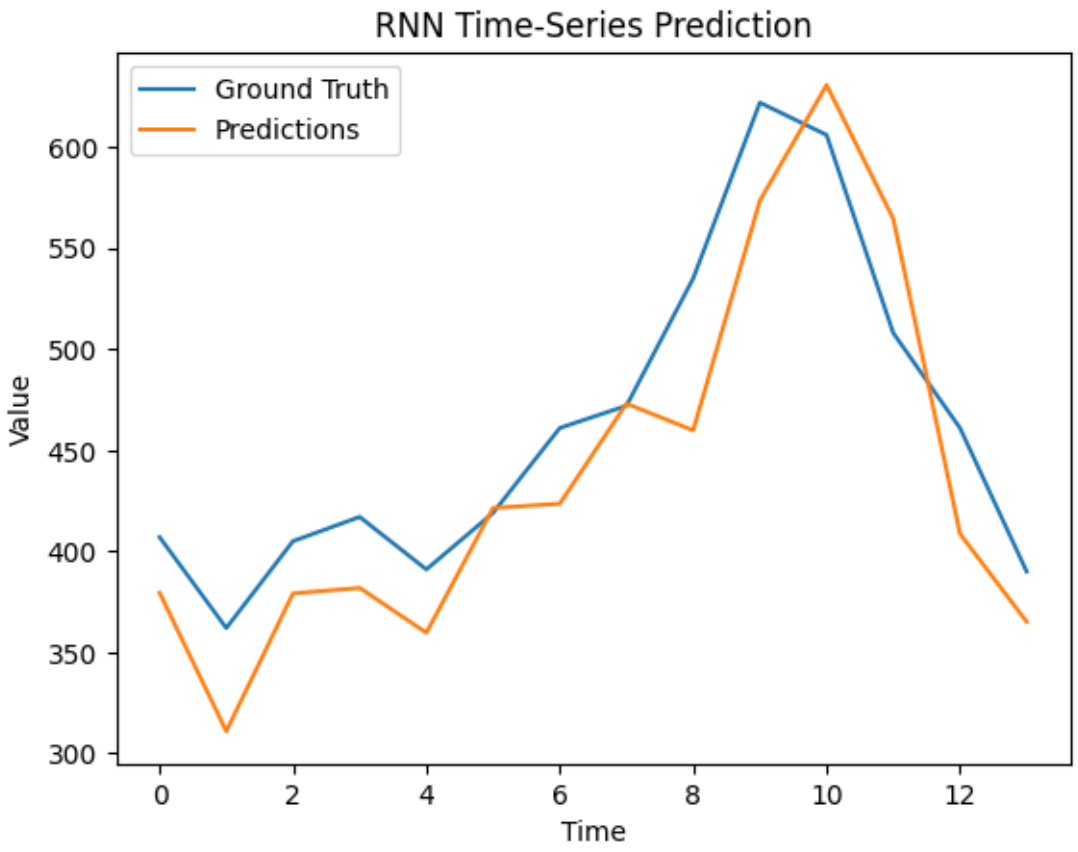


[162]:
```
import torch.optim as optim

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(SimpleRNN, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
```

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

# Set hyperparameters
input_size = 1
hidden_size = 8
output_size = 1
num_layers = 1
num_epochs = 2000
learning_rate = 0.01

# Instantiate the model
rnn = SimpleRNN(input_size, hidden_size, output_size, num_layers)

# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(rnn.parameters(), lr=learning_rate)

# Train the model
for epoch in range(num_epochs):
    outputs = rnn(trainX)
    optimizer.zero_grad()
    loss = criterion(outputs, trainY)
    loss.backward()
    optimizer.step()

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

# Evaluation
rnn.eval()
with torch.no_grad():
    test_outputs = rnn(testX)
    mse = criterion(test_outputs, testY)
    print(f'Mean Squared Error on Test Data: {mse.item():.4f}')

# Inverse transform the predictions and ground truth
predicted_data = sc.inverse_transform(test_outputs.numpy())
ground_truth_data = sc.inverse_transform(testY.numpy())

# Plot the results
#plt.axvline(x=train_size, c='r', linestyle='--', label='right limit of GT␣
 ↪used')
plt.plot(ground_truth_data, label='Ground Truth')
```

```
plt.plot(predicted_data, label='Predictions')

plt.title('RNN Time-Series Prediction')
plt.xlabel('Time')
plt.ylabel('Value')
plt.legend()
plt.show()
```

```
Epoch [1/2000], Loss: 0.0577
Epoch [101/2000], Loss: 0.0037
Epoch [201/2000], Loss: 0.0031
Epoch [301/2000], Loss: 0.0022
Epoch [401/2000], Loss: 0.0027
Epoch [501/2000], Loss: 0.0024
Epoch [601/2000], Loss: 0.0035
Epoch [701/2000], Loss: 0.0028
Epoch [801/2000], Loss: 0.0026
Epoch [901/2000], Loss: 0.0021
Epoch [1001/2000], Loss: 0.0014
Epoch [1101/2000], Loss: 0.0010
Epoch [1201/2000], Loss: 0.0008
Epoch [1301/2000], Loss: 0.0007
Epoch [1401/2000], Loss: 0.0007
Epoch [1501/2000], Loss: 0.0006
Epoch [1601/2000], Loss: 0.0005
Epoch [1701/2000], Loss: 0.0005
Epoch [1801/2000], Loss: 0.0005
Epoch [1901/2000], Loss: 0.0005
Mean Squared Error on Test Data: 0.0061
```

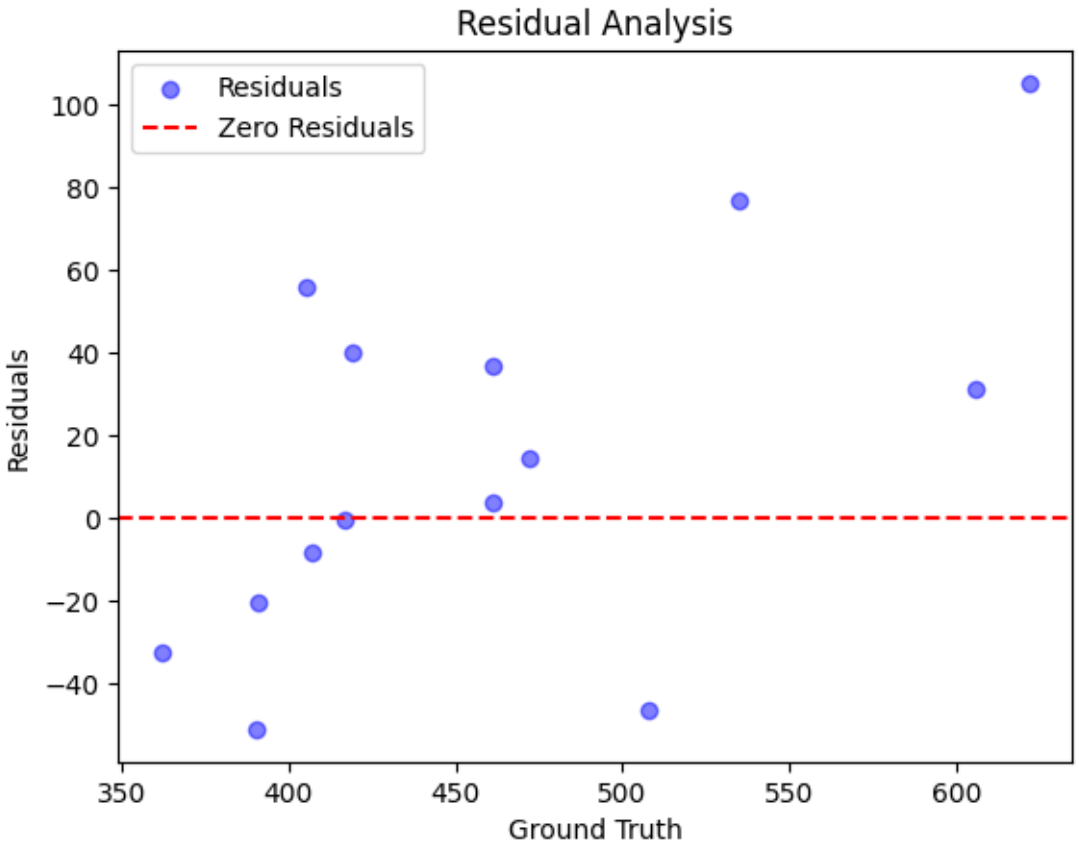


3. Evaluate and Compare the result using proper metric. Justify the metrics used.

[163]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Convert predictions to numpy array
predictions = all_predict.data.numpy()

# Inverse transform the normalized predictions
predictions = sc.inverse_transform(predictions)

# Inverse transform the normalized ground truth
ground_truth = sc.inverse_transform(testY.data.numpy())

# Calculate Mean Absolute Error (MAE)
mae = mean_absolute_error(ground_truth, predictions)
print(f"Mean Absolute Error (MAE): {mae}")

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(ground_truth, predictions)
print(f"Mean Squared Error (MSE): {mse}")
```

```
Mean Absolute Error (MAE): 37.44038772583008
Mean Squared Error (MSE): 2188.98193359375
```

[164]:
```python
from sklearn.metrics import confusion_matrix

# Calculate residuals for LSTM
residuals = ground_truth - predictions

# Create a scatter plot of ground truth vs. residuals
plt.scatter(ground_truth, residuals, alpha=0.5, color='b', label='Residuals')
plt.axhline(y=0, color='r', linestyle='--', label='Zero Residuals')
plt.xlabel('Ground Truth')
plt.ylabel('Residuals')
plt.title('Residual Analysis')
plt.legend()
plt.show()
```

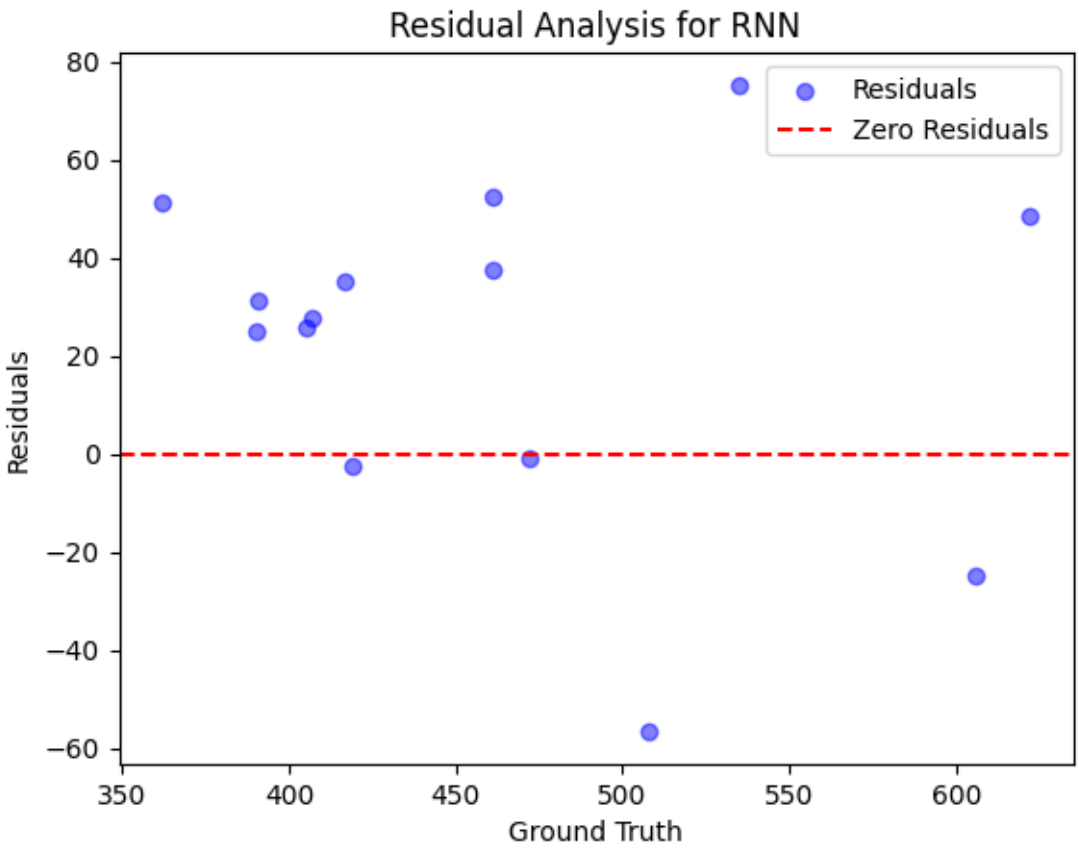


[165]:
```python
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Convert predictions to numpy array
```

```
predictions = all_predict.data.numpy()

# Inverse transform the normalized predictions
predictions = sc.inverse_transform(predictions)

# Inverse transform the normalized ground truth
ground_truth = sc.inverse_transform(testY.data.numpy())

# Calculate Mean Absolute Error (MAE)
mae = mean_absolute_error(ground_truth, predictions)
print(f"Mean Absolute Error (MAE): {mae}")

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(ground_truth, predictions)
print(f"Mean Squared Error (MSE): {mse}")
```

```
Mean Absolute Error (MAE): 37.44038772583008
Mean Squared Error (MSE): 2188.98193359375
```

[166]:
```
# Calculate residuals for RNN
residuals_rnn = ground_truth_data - predicted_data

# Create a scatter plot of ground truth vs. residuals for RNN
plt.scatter(ground_truth_data, residuals_rnn, alpha=0.5, color='b',␣
 ↪label='Residuals')
plt.axhline(y=0, color='r', linestyle='--', label='Zero Residuals')
plt.xlabel('Ground Truth')
plt.ylabel('Residuals')
plt.title('Residual Analysis for RNN')
plt.legend()
plt.show()
```



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

Various approaches and model optimization techniques are used to improve the performance of long short-term memory networks (LSTMs) and recurrent neural networks (RNNs). Here are some of them:

Increase Model Complexity: - To capture more complicated patterns, increase the number of layers in the RNN/LSTM architecture. To boost the model's capability, change the number of hidden units or neurons in each layer.

Data Augmentation: By applying changes such as random noise addition, time warping, or random sampling to create new data points or enhance the current dataset. By using data augmentation approaches, the model may be exposed to a greater variety of patterns and the training data can be made more diverse.

Regularization : - To avoid overfitting, use regularization strategies like dropout or repeated dropout. Take into account applying L1 or L2 weight regularization to the recurring weights.

Implement Bidirectional LSTM/RNN : Sequences are processed by bidirectional models in both forward and backward directions, allowing for the acquisition of more contextual data.

Scheduling at Learning Rate : To modify the learning rate during training, use learning rate scheduling. This may facilitate a more effective convergence of the model.

It is not limited to the above-mentioned methods but there are also other methods which can be used to improve the results.

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

Beyond the standard examples, recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) can be used in a variety of robotics applications. RNNs and LSTMs can be applied in the following robotics domains:

Robot Control and Path Planning: Robot Control activities like trajectory planning, motion prediction, and obstacle avoidance can be modeled using RNNs and LSTMs to account for temporal dependencies. They can predict the robot's future states and modify its control actions based on sensor data (such as lidar and camera).

Fault Diagnosis and Anomaly Detection: By identifying patterns in sensor data and system logs, RNNs and LSTMs are able to identify anomalies and departures from typical functioning in robotic systems. Because they can spot early indicators of malfunction or degradation in robot components, they can help with fault diagnosis and predictive maintenance.

Drones and autonomous cars: RNNs and LSTMs are able to anticipate how pedestrians and other cars will behave when autonomous cars or drones are in close proximity. They can aid in making decisions by forecasting future trajectories and analyzing temporal sequences of sensor data, which can help with lane changes, merging, and collision avoidance.

Healthcare Robotics: Wearable sensor time-series data can be analyzed using RNNs and LSTMs to track patients' health and identify anomalies. By identifying trends in sensor data, they can help with medication adherence, fall detection, remote patient monitoring, and other healthcare duties.

Agricultural Robotics: To improve farming techniques, RNNs and LSTMs can evaluate sensor data from agricultural systems, such as crop growth, temperature, and soil moisture. To increase agricultural management and yield, they can forecast crop yields, plan irrigation and fertilization schedules, and identify pests or illnesses in crops early on.