VISHNU INSTITUTE OF TECHNOLOGY

(AUTONOMUS)

VISHNUPUR, BHIMAVARAM – 534202, W.G.District, A.P.

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

CERTIFICATE

Certified that this is a bonafide record of practical work done by Mr. / Ms. ..... **Register Number** ..... **of I / II Semester** ..... **B. Tech in the** ..... **Laboratory of** ..... **department during the academic year 2022-2023.**

Date **Head of the Department** **In-charge Staff Member**

Submitted for the Practical Examination held on .....

INTERNAL EXAMINER **EXTERNAL EXAMINER**

INDEX

Signature of the faculty:

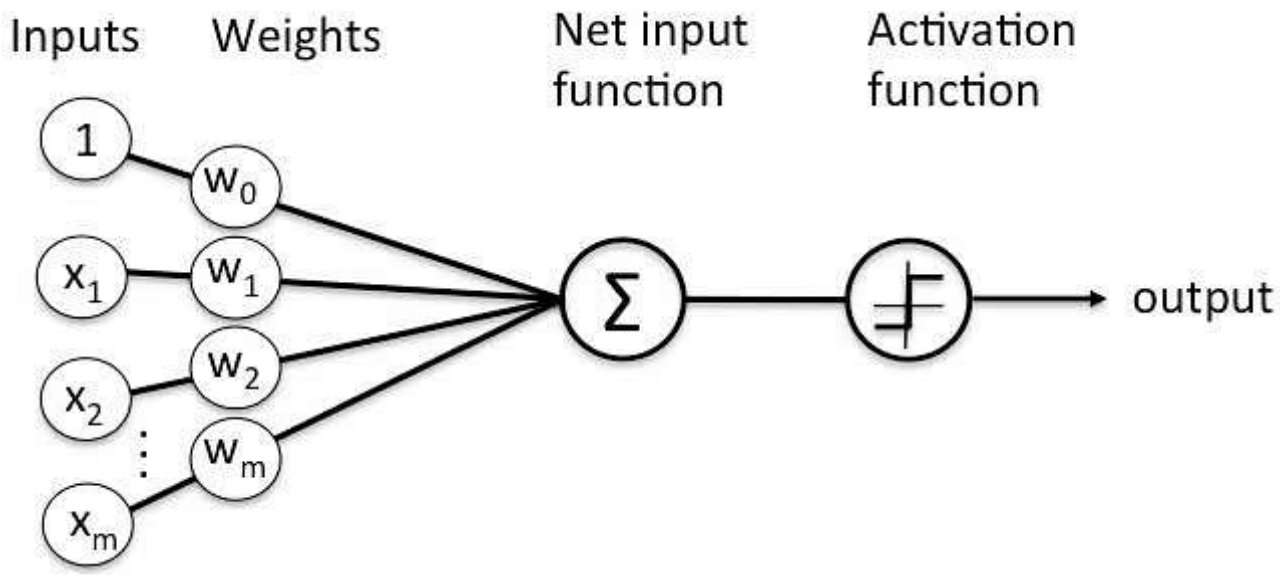
Record Marks:

Experiment-1

Perceptron

A Perceptron is a neural network unit that does certain computations to detect features or business intelligence in the input data. It is a function that maps its input “x,” which is multiplied by the learned weight coefficient, and generates an output value ”f(x).

A multi-layered perceptron model can be used to solve complex non-linear problems. It works well with both small and large input data. It helps us to obtain quick predictions after the training. It helps to obtain the same accuracy ratio with large as well as small data.




```
import numpy as np
# Setting the random seed, feel free to change it and see different solutions.
np.random.seed(42)

def stepFunction(t):
    if t >= 0:
        return 1
    return 0

def prediction(X, W, b):
    return stepFunction((np.matmul(X,W)+b)[0])

# The function should receive as inputs the data X, the labels y,
# the weights W (as an array), and the bias b,
```

```
# update the weights and bias W, b, according to the perceptron algorithm,
# and return W and b.
def perceptronStep(X, y, W, b, learn_rate = 0.01):

    for i in range(len(X)):
        yhat = prediction(X[i], W, b)
        if y[i]-yhat == 1:
            W[0] += X[i][0]*learn_rate
            W[1] += X[i][1]*learn_rate
            b += learn_rate
        elif y[i]-yhat == -1:
            W[0] -= X[i][0]*learn_rate
            W[1] -= X[i][1]*learn_rate
            b -= learn_rate
    return W, b

# This function runs the perceptron algorithm repeatedly on the dataset,
# and returns a few of the boundary lines obtained in the iterations,
# for plotting purposes.
# Feel free to play with the learning rate and the num_epochs,
# and see your results plotted below.
def trainPerceptronAlgorithm(X, y, learn_rate = 0.01, num_epochs = 25):
    x_min, x_max = min(X.T[0]), max(X.T[0])
    y_min, y_max = min(X.T[1]), max(X.T[1])
    W = np.array(np.random.rand(2,1))
    b = np.random.rand(1)[0] + x_max
    # These are the solution lines that get plotted below.
    boundary_lines = []
    for i in range(num_epochs):
        # In each epoch, we apply the perceptron step.
        W, b = perceptronStep(X, y, W, b, learn_rate)
        boundary_lines.append((-W[0]/W[1], -b/W[1]))
    return boundary_lines
```

Experiment-2

Implementation of AND OR gates using Perception

```
import numpy as np
import random
import sys


and_gate = [
# [(inputs), expected output]
    [(1, 1), 1],
    [(1, -1), -1],
    [(-1, 1), -1],
    [(-1, -1), -1]
]


or_gate = [
    [(1, 1), 1],
    [(1, -1), 1],
    [(-1, 1), 1],
    [(-1, -1), -1]
]


def activation_function(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1


def run_perceptron(gate):
    bias = (1,) # the bias is always one
    learning_constant = 0.1
    n = 50 # how many times the machine learns

    weights = []

    # initialize with 3 random weights between -1 and 1, one for each input and one for the bias
    for i in range(3):
        weights.append(random.uniform(-1, 1))

    for i in range(n):
        inputs, expected_output = random.choice(gate)
        inputs = inputs + bias # add the bias here
        weighted_sum = np.dot(inputs, weights)
        guess = activation_function(weighted_sum) # find the sign of the weighted sum
        error = expected_output - guess
        weights += learning_constant * error * np.asarray(inputs) # change the weights to include the error times input, won't change if ther


    inputs, expected_output = random.choice(gate)
    print("inputs: " + str(inputs))
    inputs = inputs + bias
    weighted_sum = np.dot(inputs, weights)
    print("weighted sum: " + str(weighted_sum))
    print("correct answer: " + str(expected_output))
    print("perceptron guess: " + str(activation_function(weighted_sum)) + '\n')


tests=2
for i in range(tests):
    print("// AND //")
    run_perceptron(and_gate)

    print("// OR //")
    run_perceptron(or_gate)
```

```
 // AND //
 inputs: (-1, 1)
 weighted sum: -0.4862978699325041
```

```
correct answer: -1
perceptron guess: -1

// OR //
inputs: (1, 1)
weighted sum: 0.4430646039214156
correct answer: 1
perceptron guess: 1

// AND //
inputs: (1, -1)
weighted sum: -0.5596067952541874
correct answer: -1
perceptron guess: -1

// OR //
inputs: (-1, -1)
weighted sum: -0.28196229758686475
correct answer: -1
perceptron guess: -1
```

Experiment-3

Simple Neural Network

The simplest type of neural network is a single-layer perceptron. It consists of one input layer and one output layer, with no hidden layers. The perceptron takes a set of input features and produces a binary output based on a weighted sum of the inputs. The weights are adjusted during training to optimize the output.

```
import numpy as np
```

+ Code + Text

```
class NeuralNetwork():

    def __init__(self):
        # seeding for random number generation
        np.random.seed(1)

        #converting weights to a 3 by 1 matrix with values from -1 to 1 and mean of 0
        self.synaptic_weights = 2 * np.random.random((3, 1)) - 1

    def sigmoid(self, x):
        #applying the sigmoid function
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        #computing derivative to the Sigmoid function
        return x * (1 - x)

    def train(self, training_inputs, training_outputs, training_iterations):

        #training the model to make accurate predictions while adjusting weights continually
        for iteration in range(training_iterations):
            #siphon the training data via  the neuron
            output = self.think(training_inputs)

            #computing error rate for back-propagation
            error = training_outputs - output

            #performing weight adjustments
            adjustments = np.dot(training_inputs.T, error * self.sigmoid_derivative(output))

            self.synaptic_weights += adjustments

    def think(self, inputs):
        #passing the inputs via the neuron to get output
        #converting values to floats

        inputs = inputs.astype(float)
        output = self.sigmoid(np.dot(inputs, self.synaptic_weights))
        return output



if __name__ == "__main__":
  #Intialise a single neuron neural network.
  neural_network = NeuralNetwork()
  print("Random starting synaptic weights: ")
  print(neural_network.synaptic_weights)

    # The training set. We have 4 examples, each consisting of 3 input values
    # and 1 output value.
  training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
  training_set_outputs = array([[0, 1, 1, 0]]).T

    # Train the neural network using a training set.
    # Do it 10,000 times and make small adjustments each time.
  neural_network.train(training_set_inputs, training_set_outputs, 10000)

  print("New synaptic weights after training: ")
```

```
print(neural_network.synaptic_weights)

  # Test the neural network with a new situation.
print("Considering new situation [1, 0, 0] -> ?: ")
print(neural_network.think(array([1, 0, 0])))
```

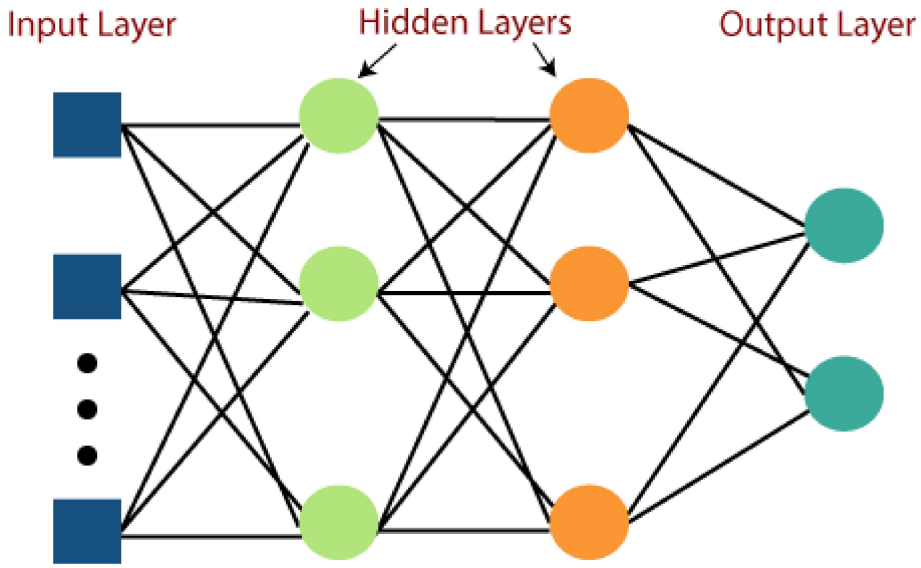
```
   Random starting synaptic weights:
   [[-0.16595599]
    [ 0.44064899]
    [-0.99977125]]
   New synaptic weights after training:
   [[ 9.67299303]
    [-0.2078435 ]
    [-4.62963669]]
   Considering new situation [1, 0, 0] -> ?:
   [0.99993704]
```

Experiment-4

Multi-layered neural network

A multi-layered neural network is a type of neural network that consists of one or more hidden layers between the input and output layers. These hidden layers perform the computation required to transform the input data into the output prediction.

Each layer in a multi-layered neural network is composed of multiple nodes, or neurons, which are connected to the neurons in the previous and next layers through weighted connections. The output of each neuron is computed as a function of the weighted sum of the inputs to the neuron, followed by an activation function. The activation function introduces non-linearity into the network and enables it to learn complex patterns in the data.




```
from numpy import exp, array, random, dot
```

```
class NeuronLayer():
    def __init__(self, number_of_neurons, number_of_inputs_per_neuron):
        self.synaptic_weights = 2 * random.random((number_of_inputs_per_neuron, number_of_neu
```

```
class NeuralNetwork():
    def __init__(self, layer1, layer2):
        self.layer1 = layer1
```

```
    self.layer2 = layer2

# The Sigmoid function, which describes an S shaped curve.
# We pass the weighted sum of the inputs through this function to
# normalise them between 0 and 1.
def __sigmoid(self, x):
    return 1 / (1 + exp(-x))

# The derivative of the Sigmoid function.
# This is the gradient of the Sigmoid curve.
# It indicates how confident we are about the existing weight.
def __sigmoid_derivative(self, x):
    return x * (1 - x)

# We train the neural network through a process of trial and error.
# Adjusting the synaptic weights each time.
def train(self, training_set_inputs, training_set_outputs, number_of_training_iterations)
    for iteration in range(number_of_training_iterations):
        # Pass the training set through our neural network
        output_from_layer_1, output_from_layer_2 = self.think(training_set_inputs)

        # Calculate the error for layer 2 (The difference between the desired output
        # and the predicted output).
        layer2_error = training_set_outputs - output_from_layer_2
        layer2_delta = layer2_error * self.__sigmoid_derivative(output_from_layer_2)

        # Calculate the error for layer 1 (By looking at the weights in layer 1,
        # we can determine by how much layer 1 contributed to the error in layer 2).
        layer1_error = layer2_delta.dot(self.layer2.synaptic_weights.T)
        layer1_delta = layer1_error * self.__sigmoid_derivative(output_from_layer_1)

        # Calculate how much to adjust the weights by
        layer1_adjustment = training_set_inputs.T.dot(layer1_delta)
        layer2_adjustment = output_from_layer_1.T.dot(layer2_delta)

        # Adjust the weights.
        self.layer1.synaptic_weights += layer1_adjustment
        self.layer2.synaptic_weights += layer2_adjustment

# The neural network thinks.
def think(self, inputs):
    output_from_layer1 = self.__sigmoid(dot(inputs, self.layer1.synaptic_weights))
    output_from_layer2 = self.__sigmoid(dot(output_from_layer1, self.layer2.synaptic_weig
    return output_from_layer1, output_from_layer2

# The neural network prints its weights
def print_weights(self):
    print("    Layer 1 (4 neurons, each with 3 inputs): ")
    print(self.layer1.synaptic_weights)
    print("    Layer 2 (1 neuron, with 4 inputs):")
    print(self.layer2.synaptic_weights)
```

```
if __name__ == "__main__":

    #Seed the random number generator
    random.seed(1)

    # Create layer 1 (4 neurons, each with 3 inputs)
    layer1 = NeuronLayer(4, 3)

    # Create layer 2 (a single neuron with 4 inputs)
    layer2 = NeuronLayer(1, 4)

    # Combine the layers to create a neural network
    neural_network = NeuralNetwork(layer1, layer2)

    print ("Stage 1) Random starting synaptic weights: ")
    neural_network.print_weights()

    # The training set. We have 7 examples, each consisting of 3 input values
    # and 1 output value.
    training_set_inputs = array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [0, 1, 0], [1, 0, 0], [1, 1
    training_set_outputs = array([[0, 1, 1, 1, 1, 0, 0]]).T

    # Train the neural network using the training set.
    # Do it 60,000 times and make small adjustments each time.
    neural_network.train(training_set_inputs, training_set_outputs, 60000)

    print("Stage 2- New synaptic weights after training: ")
    neural_network.print_weights()

    # Test the neural network with a new situation.
    print("Stage 3- Considering a new situation [1, 1, 0] -> ?: ")
    hidden_state, output = neural_network.think(array([1, 1, 0]))
    print(output)

     Stage 1) Random starting synaptic weights:
         Layer 1 (4 neurons, each with 3 inputs):
     [[-0.16595599  0.44064899 -0.99977125 -0.39533485]
      [-0.70648822 -0.81532281 -0.62747958 -0.30887855]
      [-0.20646505  0.07763347 -0.16161097  0.370439  ]]
         Layer 2 (1 neuron, with 4 inputs):
     [[-0.5910955 ]
      [ 0.75623487]
      [-0.94522481]
      [ 0.34093502]]
     Stage 2- New synaptic weights after training:
         Layer 1 (4 neurons, each with 3 inputs):
     [[ 0.3122465   4.57704063 -6.15329916 -8.75834924]
      [ 0.19676933 -8.74975548 -6.1638187   4.40720501]
      [-0.03327074 -0.58272995  0.08319184 -0.39787635]]
         Layer 2 (1 neuron, with 4 inputs):
     [[ -8.18850925]
      [ 10.13210706]
```

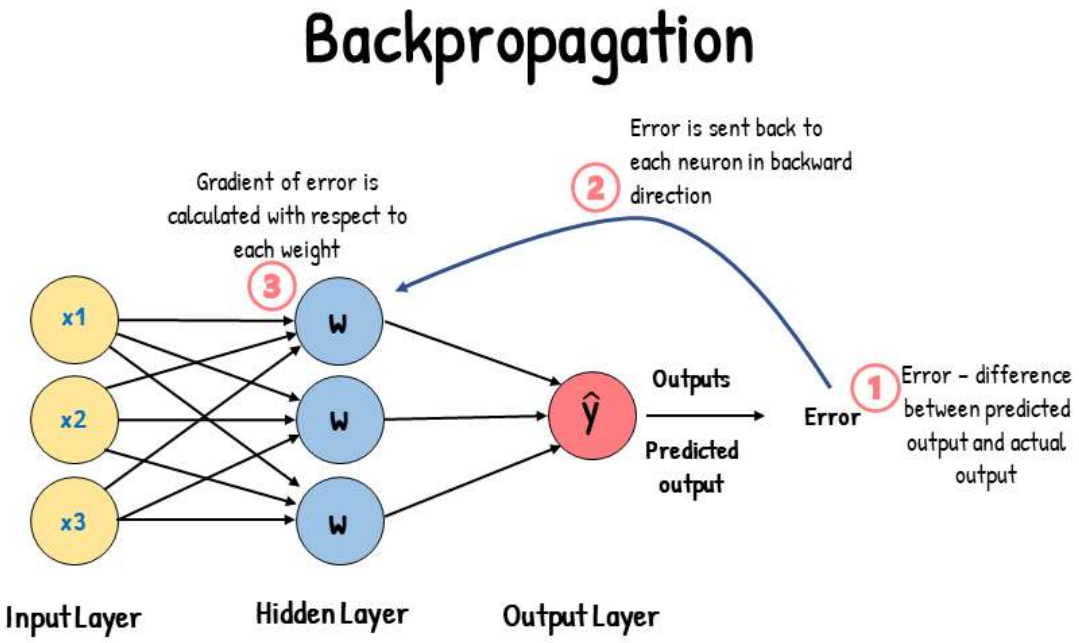
```
 [-21.33532796]
 [  9.90935111]]
Stage 3- Considering a new situation [1, 1, 0] -> ?:
[0.0078876]
```

Experiment-5

Implementing the Back propagation algorithm

Backpropagation is a widely used algorithm for training feedforward neural networks. It computes the gradient of the loss function with respect to the network weights. It is very efficient, rather than naively directly computing the gradient concerning each weight. This efficiency makes it possible to use gradient methods to train multi-layer networks and update weights to minimize loss; variants such as gradient descent or stochastic gradient descent are often used.

The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight via the chain rule, computing the gradient layer by layer, and iterating backward from the last layer to avoid redundant computation of intermediate terms in the chain rule.




```
from math import exp
from random import seed
from random import random
```

```
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
 network = list()
 hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)
 network.append(hidden_layer)
 output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs
```

```
 network.append(output_layer)
 return network


# Calculate neuron activation for an input
def activate(weights, inputs):
 activation = weights[-1]
 for i in range(len(weights)-1):
   activation += weights[i] * inputs[i]
 return activation


# Transfer neuron activation
def transfer(activation):
 return 1.0 / (1.0 + exp(-activation))


# Forward propagate input to a network output
def forward_propagate(network, row):
 inputs = row
 for layer in network:
  new_inputs = []
  for neuron in layer:
    activation = activate(neuron['weights'], inputs)
    neuron['output'] = transfer(activation)
    new_inputs.append(neuron['output'])
  inputs = new_inputs
 return inputs


# Calculate the derivative of an neuron output
def transfer_derivative(output):
 return output * (1.0 - output)


# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
 for i in reversed(range(len(network))):
  layer = network[i]
  errors = list()

  if i != len(network)-1:
    for j in range(len(layer)):
      error = 0.0
    for neuron in network[i + 1]:
      error += (neuron['weights'][j] * neuron['delta'])
      errors.append(error)
  else:
    for j in range(len(layer)):
      neuron = layer[j]
      errors.append(neuron['output'] - expected[j])
  for j in range(len(layer)):
```

```
    neuron = layer[j]
    neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])


# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']


# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):

    print("\n Network Training Begins:\n")

    for epoch in range(n_epoch):
        sum_error = 0
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))

    print("\n Network Training Ends:\n")


#Test training backprop algorithm
seed(2)
dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
    [7.673756466,3.508563011,1]]

print("\n The input Data Set :\n",dataset)
n_inputs = len(dataset[0]) - 1
print("\n Number of Inputs :\n",n_inputs)
```

```
n_outputs = len(set([row[-1] for row in dataset]))
print("\n Number of Outputs :\n",n_outputs)
```

```
 The input Data Set :
 [[2.7810836, 2.550537003, 0], [1.465489372, 2.362125076, 0], [3.396561688, 4.400293529,

 Number of Inputs :
 2

 Number of Outputs :
 2
```

```
#Network Initialization
network = initialize_network(n_inputs, 2, n_outputs)

# Training the Network
train_network(network, dataset, 0.5, 20, n_outputs)
print("\n Final Neural Network :")

i= 1
for layer in network:
    j=1
    for sub in layer:
        print("\n Layer[%d] Node[%d]:\n" %(i,j),sub)
        j=j+1
    i=i+1
```

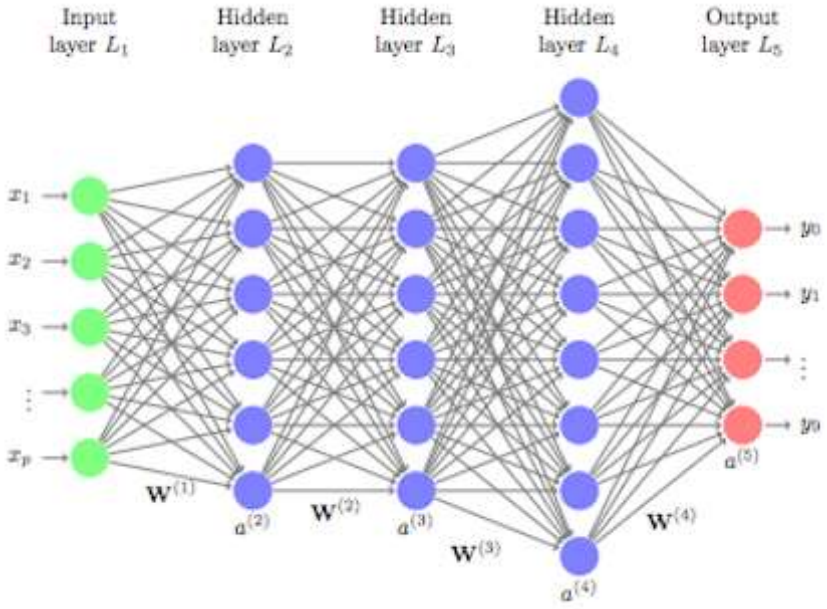
```
  Network Training Begins:

 >epoch=0, lrate=0.500, error=7.317
 >epoch=1, lrate=0.500, error=8.334
 >epoch=2, lrate=0.500, error=8.867
 >epoch=3, lrate=0.500, error=9.161
 >epoch=4, lrate=0.500, error=9.341
 >epoch=5, lrate=0.500, error=9.460
 >epoch=6, lrate=0.500, error=9.544
 >epoch=7, lrate=0.500, error=9.606
 >epoch=8, lrate=0.500, error=9.654
 >epoch=9, lrate=0.500, error=9.691
 >epoch=10, lrate=0.500, error=9.722
 >epoch=11, lrate=0.500, error=9.747
 >epoch=12, lrate=0.500, error=9.768
 >epoch=13, lrate=0.500, error=9.785
 >epoch=14, lrate=0.500, error=9.801
 >epoch=15, lrate=0.500, error=9.814
 >epoch=16, lrate=0.500, error=9.826
 >epoch=17, lrate=0.500, error=9.836
 >epoch=18, lrate=0.500, error=9.845
 >epoch=19, lrate=0.500, error=9.854
```

```
Network Training Ends:


Final Neural Network :

Layer[1] Node[1]:
{'weights': [0.9564917906946954, 0.9464316011967685, 0.055967744595817596], 'output': 0

Layer[1] Node[2]:
{'weights': [0.3358180770489934, 0.9478071890631982, 0.7995504556318416], 'output': 0.9

Layer[2] Node[1]:
{'weights': [1.8081707168764891, 1.4081193650989117, 1.7427866030736947], 'output': 0.9

Layer[2] Node[2]:
{'weights': [1.8208684089609368, 1.7761865808784896, 1.392402828706596], 'output': 0.99
```

Experiment-6

Predicting car purchase amount from car sales datasets using Feed Forward Neural Network

Feed forward neural networks are artificial neural networks in which nodes do not form loops. This type of neural network is also known as a multi-layer neural network as all information is only passed forward. During data flow, input nodes receive data, which travel through hidden layers, and



exit output nodes.

```
#Load Libraries & Capture Dataset
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import seaborn as sb # library that uses Matplotlib underneath to plot graphs
import matplotlib.pyplot as plt #Matlab
from warnings import filterwarnings #ignore warnings from specified module
filterwarnings("ignore")


#Reading Dataset
data = pd.read_csv("car_purchasing.csv",encoding='ISO-8859-1')


#returns the first n rows for the object based on position
data.head()
```

	customer name	JobTitle	customer e-mail	country	ge
0	Martina Avila	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	cubilia.Curae.Phasellus@quisaccumsanconvallis.edu	Bulgaria	
1	Harlan Barnes	CAPTAIN III (POLICE DEPARTMENT)	eu.dolor@diam.co.uk	Belize	
2	Naomi Rodriquez	CAPTAIN III (POLICE DEPARTMENT)	vulputate.mauris.sagittis@ametconsectetueradip...	Algeria	
3	Jade Cunningham	WIRE ROPE CABLE MAINTENANCE MECHANIC	malesuada@dignissim.com	Cook Islands	
4	Cedric Leach	DEPUTY CHIEF OF DEPARTMENT, (FIRE DEPARTMENT)	felis.ullamcorper.viverra@egetmollislectus.net	Brazil	

```
 #printing information about the DataFrame
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 15 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   customer name        500 non-null    object 
 1   JobTitle             500 non-null    object 
 2   customer e-mail      500 non-null    object 
 3   country              500 non-null    object 
 4   gender               500 non-null    int64  
 5   age                  500 non-null    int64  
 6   BasePay              500 non-null    float64
 7   OvertimePay          500 non-null    float64
 8   OtherPay             500 non-null    float64
 9   Benefits             0 non-null      float64
 10  TotalPay             500 non-null    float64
 11  TotalPayBenefits     500 non-null    float64
 12  credit card debt     500 non-null    float64
 13  net worth            500 non-null    float64
 14  car purchase amount  500 non-null    float64
dtypes: float64(9), int64(2), object(4)
memory usage: 58.7+ KB
```

```
#calculating the relationship between each column in the data set.
data.corr()[["car purchase amount"]].sort_values(["car purchase amount"])
```

	car purchase amount
gender	-0.066408
OvertimePay	-0.031566
OtherPay	-0.008536
TotalPay	0.006014
TotalPayBenefits	0.006014
credit card debt	0.028882
BasePay	0.037602
net worth	0.488580
age	0.633273
car purchase amount	1.000000
Benefits	NaN

```
# Missing data check
data.isna().sum()
```

```
customer name            0
JobTitle                 0
customer e-mail          0
country                  0
gender                   0
age                      0
BasePay                  0
OvertimePay              0
OtherPay                 0
Benefits               500
TotalPay                 0
TotalPayBenefits         0
credit card debt         0
net worth                0
car purchase amount      0
dtype: int64
```

```
# Dropping columns
car_df = data.drop(["customer name","customer e-mail","country","JobTitle","Benefits"],axis=1
```

```
 #Define X and Y
Y = car_df[["car purchase amount"]]
```

```
X = car_df.drop(["car purchase amount"],axis=1)
print(X.shape,Y.shape)
```

```
(500, 9) (500, 1)
```

```
#Pre-Processing
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_scaled = mms.fit_transform(X)
Y_scaled = mms.fit_transform(Y.values.reshape(-1,1))
```

```
print(X_scaled.shape,Y_scaled.shape)
```

```
(500, 9) (500, 1)
```

```
#Division of data in tarining and testing set
from sklearn.model_selection import train_test_split
xtrain,xtest,ytrain,ytest = train_test_split(X_scaled,Y_scaled,test_size=0.25,random_state=10
```

```
print(xtrain.shape,ytrain.shape,xtest.shape,ytest.shape)
```

```
(375, 9) (375, 1) (125, 9) (125, 1)
```

```
#accuracy
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor()
model.fit(xtrain, ytrain)
print(model.score(xtest, ytest)*100)
```

```
46.977336504651724
```

```
#ANN Model
import tensorflow.keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(25, input_dim=9, activation='relu'))
model.add(Dense(25, activation='relu'))
model.add(Dense(1, activation='linear'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 25)                250
```

```
 dense_1 (Dense)             (None, 25)                650

 dense_2 (Dense)             (None, 1)                 26

=================================================================
Total params: 926
Trainable params: 926
Non-trainable params: 0
_________________________________________________________________
```

```
#defining the loss function
#Adam optimization is a stochastic gradient descent method that is based on adaptive estimati
#epoch-number times that the learning algorithm will work through the entire training dataset
model.compile(optimizer='adam',loss='mean_squared_error')
epochs_hist = model.fit(xtrain,ytrain,epochs=10,batch_size=50,verbose=1,validation_split=0.2)
```

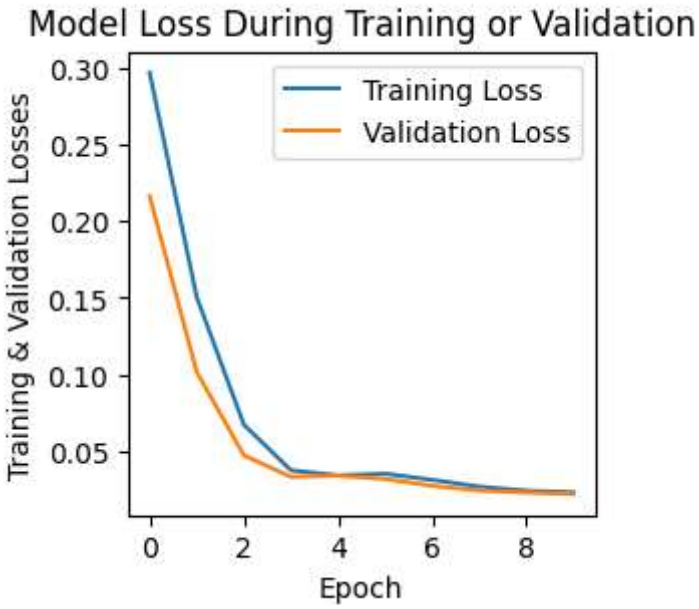
```
Epoch 1/10
6/6 [==============================] - 3s 96ms/step - loss: 0.2971 - val_loss: 0.2164
Epoch 2/10
6/6 [==============================] - 0s 11ms/step - loss: 0.1505 - val_loss: 0.1016
Epoch 3/10
6/6 [==============================] - 0s 17ms/step - loss: 0.0672 - val_loss: 0.0471
Epoch 4/10
6/6 [==============================] - 0s 18ms/step - loss: 0.0373 - val_loss: 0.0331
Epoch 5/10
6/6 [==============================] - 0s 14ms/step - loss: 0.0340 - val_loss: 0.0339
Epoch 6/10
6/6 [==============================] - 0s 14ms/step - loss: 0.0351 - val_loss: 0.0317
Epoch 7/10
6/6 [==============================] - 0s 41ms/step - loss: 0.0311 - val_loss: 0.0272
Epoch 8/10
6/6 [==============================] - 0s 14ms/step - loss: 0.0267 - val_loss: 0.0241
Epoch 9/10
6/6 [==============================] - 0s 18ms/step - loss: 0.0239 - val_loss: 0.0229
Epoch 10/10
6/6 [==============================] - 0s 15ms/step - loss: 0.0229 - val_loss: 0.0218
```

```
epochs_hist.history.keys()
```

```
dict_keys(['loss', 'val_loss'])
```

```
#Visualization
plt.figure(figsize=(3,3),dpi=100)
plt.plot(epochs_hist.history["loss"])
plt.plot(epochs_hist.history["val_loss"])

plt.title('Model Loss During Training or Validation')
plt.ylabel('Training & Validation Losses')
plt.xlabel('Epoch')
plt.legend(['Training Loss', 'Validation Loss'])
```

```
<matplotlib.legend.Legend at 0x7fe242fb5fc0>
```




```
#Predictions with some random data provided
X_random_sample = np.array([[0,42,167411.18,0,400184.25,567595.43,567595.43,11609.38091,23896
y_predict = model.predict(X_random_sample)
```

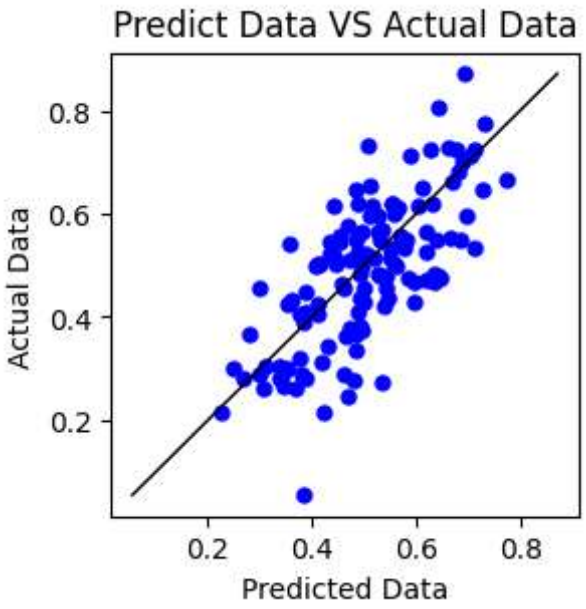
```
1/1 [==============================] - 0s 88ms/step
```

```
#Algorithms
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge

#Metrics
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error


#function
def performance(model,X_train,y_train,y_pred,y_test):
    print('Train Score:',model.score(xtrain,ytrain))
    print('Test Score:',r2_score(ytest,y_pred))
    print('')
    print('MSE:',mean_squared_error(ytest,y_pred))
    print('MAE:',mean_absolute_error(ytest,y_pred))


#Fitting the model
#Linear Regression establishes a relationship between dependent variable (Y)
#one or more independent variables (X) using a best fit straight line (also known as regressi
lr = LinearRegression()
lr.fit(xtrain,ytrain)

#The predicted data
```

```
lr_pred = lr.predict(xtest)
performance(lr,xtrain,ytrain,lr_pred,ytest)
```

```
     Train Score: 0.666373022883424
     Test Score: 0.4833359479160346

     MSE: 0.010542845566519393
     MAE: 0.08267223296769366
```

```
#Comparision & Visualisation
plt.figure(figsize=(3,3),dpi=100)
plt.scatter(lr_pred,ytest,c='blue',marker='o',s=25)
plt.plot([ytest.min(),ytest.max()],[ytest.min(),ytest.max()],c='black',lw=1)

plt.xlabel('Predicted Data')
plt.ylabel('Actual Data')
plt.title('Predict Data VS Actual Data')
plt.show()
```

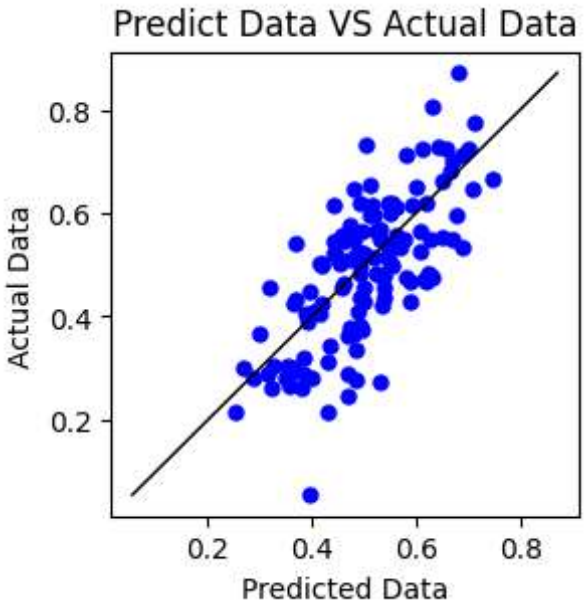



```
#Fitting the model
#Ridge Regression is a technique used when the data suffers from multicollinearity
#independent variables are highly correlated.
#amount of shrinkage (or constraint) that will be implemented in the equation.
# the larger is the alpha, the higher is the smoothness constraint.
ridge = Ridge(alpha = 1)
ridge.fit(xtrain,ytrain)

#The predicted data
ridge_pred = ridge.predict(xtest)
performance(ridge,xtrain,ytrain,ridge_pred,ytest)
```

```
     Train Score: 0.6614677775077044
     Test Score: 0.49423458296133316
```

```
MSE: 0.010320452261421083
MAE: 0.08219794340251466
```

```
#Comparision & Visualisation
plt.figure(figsize=(3,3),dpi=100)
plt.scatter(ridge_pred,ytest,c='blue',marker='o',s=25)
plt.plot([ytest.min(),ytest.max()],[ytest.min(),ytest.max()],c='black',lw=1)
plt.xlabel('Predicted Data')
plt.ylabel('Actual Data')
plt.title('Predict Data VS Actual Data')
plt.show()
```



Experiment-7

Image processing operations :

Histogram equalization, thresholding, edge detection, data augmentation, morphological operations

```
# mount the drive where we will save stats and loade our libraries from
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 pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
from tensorflow.keras.datasets import cifar10
```

```
(X_train,y_train) , (X_test, y_test) = cifar10.load_data()
```

```
X_train.shape
```

```
(50000, 32, 32, 3)
```

```
X_train[0]
```

```
array([[[ 59,  62,  63],
        [ 43,  46,  45],
        [ 50,  48,  43],
        ...,
        [158, 132, 108],
        [152, 125, 102],
        [148, 124, 103]],

       [[ 16,  20,  20],
        [  0,   0,   0],
        [ 18,   8,   0],
        ...,
        [123,  88,  55],
        [119,  83,  50],
        [122,  87,  57]],

       [[ 25,  24,  21],
        [ 16,   7,   0],
        [ 49,  27,   8],
        ...,
        [118,  84,  50],
        [120,  84,  50],
        [109,  73,  42]],

       ...,

       [[208, 170,  96],
        [201, 153,  34],
        [198, 161,  26],
        ...,
        [160, 133,  70],
        [ 56,  31,   7],
        [ 53,  34,  20]],

       [[180, 139,  96],
        [173, 123,  42],
        [186, 144,  30],
        ...,
        [184, 148,  94],
        [ 97,  62,  34],
        [ 83,  53,  34]],

       [[177, 144, 116],
        [168, 129,  94],
        [179, 142,  87],
```

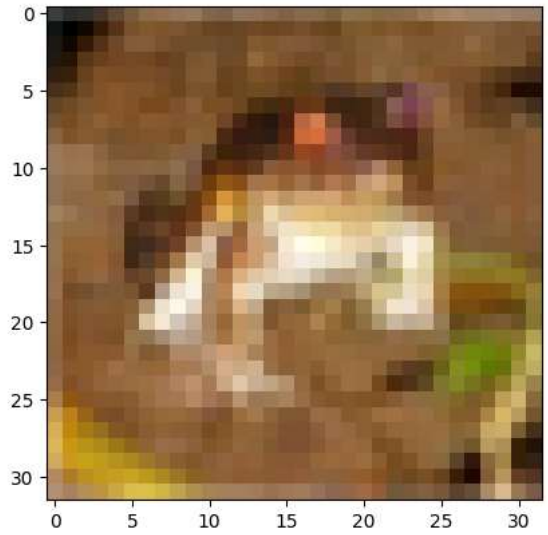
```
        ...,
        [216, 184, 140],
        [151, 118,  84],
        [123,  92,  72]]], dtype=uint8)
```

```
X_train[0].shape
```

```
(32, 32, 3)
```

```
plt.imshow(X_train[0])
```

```
<matplotlib.image.AxesImage at 0x7fc2389b1ae0>
```



Visualizing the Data

To visualize the images, we will again use the matplotlib library. We don't need to worry about the details of this visualization, but if interested, you can learn more about matplotlib at a later time.

Note that we'll have to reshape the data from its current 1D shape of 784 pixels, to a 2D shape of 28x28 pixels to make sense of the image:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(40,40))

num_images = 20
for i in range(num_images):
    row = X_train[i]
    label = y_train[i]

    image = row
    plt.subplot(1, num_images, i+1)
    plt.title(label, fontdict={'fontsize': 30})
    plt.axis('off')
    plt.imshow(image, cmap='gray')
```


```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
```

```
X_train/=255
X_test/=255
```

Histogram equalization

Histogram Equalization is a computer image processing technique used to improve contrast in images. It accomplishes this by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image.

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
# read a image using imread
img = cv2.imread('flower.jpg',0)
cv2_imshow(img)
```

```
import matplotlib.pyplot as plt

# creating a Histograms Equalization
# of a image using cv2.equalizeHist()
equ = cv2.equalizeHist(img)
cv2_imshow(equ)
```

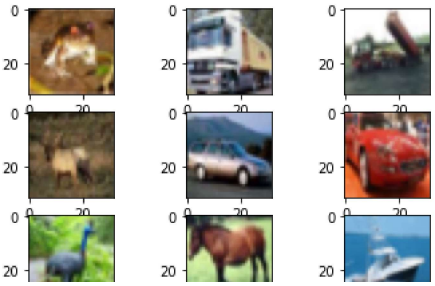
Data Augmentation

In order to teach our model to be more robust when looking at new data, we're going to programmatically increase the size and variance in our dataset. This is known as data augmentation, a useful technique for many deep learning applications.

The increase in size gives the model more images to learn from while training. The increase in variance helps the model ignore unimportant features and select only the features that are truly important in classification, allowing it to generalize better.

Keras comes with an image augmentation class called ImageDataGenerator. We recommend checking out the documentation here. It accepts a series of options for augmenting your data. Later in the course, we'll have you select a proper augmentation strategy. For now, take a look at the options we've selected below, and then execute the cell to create an instance of the class:

```
#load data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
for i in range(0,9):
    plt.subplot(330 + 1 + i)
    plt.imshow(x_train[i])
plt.show()
```



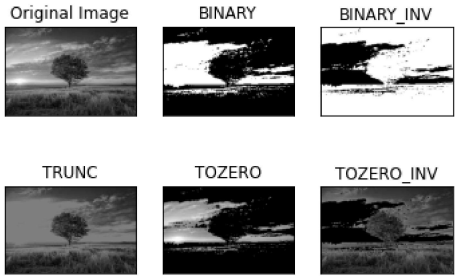
```
# set up image augmentation
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=10,  # randomly rotate images in the range (degrees, 0 to 180)
    zoom_range=0.1,  # Randomly zoom image
    width_shift_range=0.1,  # randomly shift images horizontally (fraction of total width)
    height_shift_range=0.1,  # randomly shift images vertically (fraction of total height)
    horizontal_flip=True,  # randomly flip images horizontally
    vertical_flip=False, # Don't randomly flip images vertically
)
datagen.fit(X_train)


# see example augmentation images
for X_batch, y_batch in datagen.flow(x_train, y_train, batch_size=9):
    for i in range(0, 9):
        plt.subplot(330 + 1 + i)
        plt.imshow(X_batch[i].astype(np.uint8))
    plt.show()
    break
```

Thresholding

```
import cv2
from google.colab.patches import cv2_imshow
image = cv2.imread('download.jpg')
threshold_value = 120
max_val = 255
ret, image = cv2.threshold(image, threshold_value, max_val, cv2.THRESH_BINARY)
cv2_imshow(image)
```

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
```

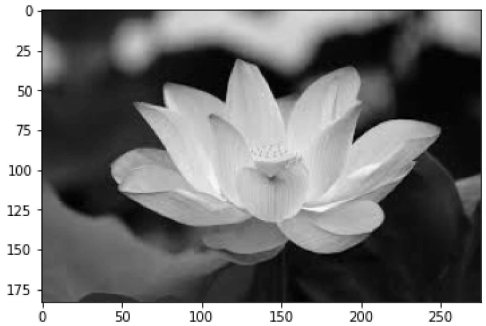
```
img = cv.imread('download.jpg',0)
ret,thresh1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
ret,thresh2 = cv.threshold(img,127,255,cv.THRESH_BINARY_INV)
ret,thresh3 = cv.threshold(img,127,255,cv.THRESH_TRUNC)
ret,thresh4 = cv.threshold(img,127,255,cv.THRESH_TOZERO)
ret,thresh5 = cv.threshold(img,127,255,cv.THRESH_TOZERO_INV)
titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
for i in range(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i],'gray',vmin=0,vmax=255)
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```



Edge detection

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
# read the image
image = cv2.imread("flower.jpg")
cv2_imshow(image)
```

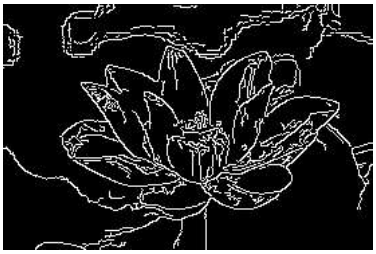
```
# convert it to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# show the grayscale image
plt.imshow(gray, cmap="gray")
plt.show()
```



All we need to do now, is to pass this image to cv2.Canny() function which finds edges in the input image and marks them in the output map edges using the Canny algorithm:

```
# perform the canny edge detector to detect image edges
edges = cv2.Canny(gray, threshold1=30, threshold2=100)
```

```
cv2_imshow(edges)
```



The smallest value between threshold1 and threshold2 is used for edge linking. The largest value is used to find initial segments of strong edges.

Experiment-8

Image classification using CNN on Cifar Dataset

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Flatten, Dropout, BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import classification_report, confusion_matrix
```

+ Code + Text

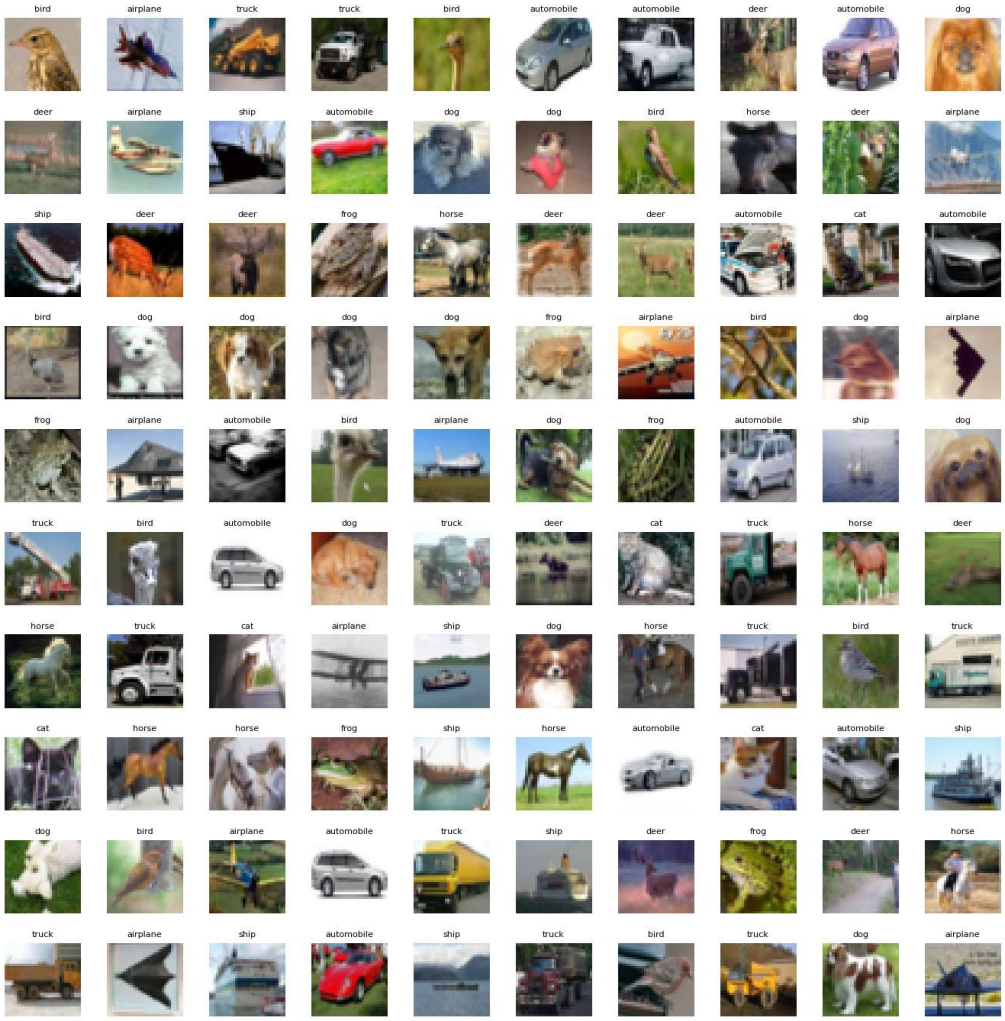
The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print(f"X_train shape: {X_train.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
X_train shape: (50000, 32, 32, 3)
y_train shape: (50000, 1)
X_test shape: (10000, 32, 32, 3)
y_test shape: (10000, 1)
```

```
# Define the labels of the dataset
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
'dog', 'frog', 'horse', 'ship', 'truck']
```

```
# Define the labels of the dataset
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
'dog', 'frog', 'horse', 'ship', 'truck']
# Let's view more images in a grid format
# Define the dimensions of the plot grid
W_grid = 10
L_grid = 10
# fig, axes = plt.subplots(L_grid, W_grid)
# subplot return the figure object and axes object
# we can use the axes object to plot specific figures at various locations
fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))
axes = axes.ravel() # flaten the 15 x 15 matrix into 225 array
n_train = len(X_train) # get the length of the train dataset
# Select a random number from 0 to n_train
for i in np.arange(0, W_grid * L_grid): # create evenly spaces variables
  # Select a random number
  index = np.random.randint(0, n_train)
  # read and display an image with the selected index
  axes[i].imshow(X_train[index,1:])
  label_index = int(y_train[index])
  axes[i].set_title(labels[label_index], fontsize = 8)
  axes[i].axis('off')
plt.subplots_adjust(hspace=0.4)
```




```
classes_name = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']
classes, counts = np.unique(y_train, return_counts=True)
plt.barh(classes_name, counts)
plt.title('Class distribution in training set')
```

```
Text(0.5, 1.0, 'Class distribution in training set')
```


```
#Data Preprocessing
# Scale the data
X_train = X_train / 255.0
X_test = X_test / 255.0
# Transform target variable into one-hotencoding
y_cat_train = to_categorical(y_train, 10)
y_cat_test = to_categorical(y_test, 10)


print(y_train)
print(y_cat_train)
```

```
[[6]
 [9]
 [9]
 ...
 [9]
 [1]
 [1]]
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 0. 0. 1.]
 ...
 [0. 0. 0. ... 0. 0. 1.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]]
```

```
#model building
INPUT_SHAPE = (32, 32, 3)
KERNEL_SIZE = (3, 3)
model = Sequential()
# Convolutional Layer
#layer-1
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
# Pooling layer
model.add(MaxPool2D(pool_size=(2, 2)))
# Dropout layers
model.add(Dropout(0.25))
#layer-2
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
#layer-3
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
#flatten
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))
```

```
METRICS = ['accuracy',tf.keras.metrics.Precision(name='precision'),tf.keras.metrics.Recall(name='recall')]
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=METRICS)


model.summary()
```

```
      conv2d (Conv2D)             (None, 32, 32, 32)        896

      batch_normalization (BatchN  (None, 32, 32, 32)       128
      ormalization)

      conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

      batch_normalization_1 (Batc  (None, 32, 32, 32)       128
      hNormalization)

      max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
      )

      dropout (Dropout)           (None, 16, 16, 32)        0

      conv2d_2 (Conv2D)           (None, 16, 16, 64)        18496

      batch_normalization_2 (Batc  (None, 16, 16, 64)       256
      hNormalization)

      conv2d_3 (Conv2D)           (None, 16, 16, 64)        36928

      batch_normalization_3 (Batc  (None, 16, 16, 64)       256
      hNormalization)

      max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
      2D)

      dropout_1 (Dropout)         (None, 8, 8, 64)          0

      conv2d_4 (Conv2D)           (None, 8, 8, 128)         73856

      batch_normalization_4 (Batc  (None, 8, 8, 128)        512
      hNormalization)

      conv2d_5 (Conv2D)           (None, 8, 8, 128)         147584

      batch_normalization_5 (Batc  (None, 8, 8, 128)        512
      hNormalization)

      max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
      2D)

      dropout_2 (Dropout)         (None, 4, 4, 128)         0

      flatten (Flatten)           (None, 2048)              0

      dense (Dense)               (None, 128)               262272

      dropout_3 (Dropout)         (None, 128)               0

      dense_1 (Dense)             (None, 10)                1290

     =================================================================
     Total params: 552,362
     Trainable params: 551,466
     Non-trainable params: 896
     _________________________________________________________________
```

```
#Early Stopping(for reduce over fitting)
early_stop = EarlyStopping(monitor='val_loss', patience=2)


#Data Augmentations
batch_size = 32
data_generator = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
train_generator = data_generator.flow(X_train, y_cat_train, batch_size)
steps_per_epoch = X_train.shape[0] // batch_size
r = model.fit(train_generator,
epochs=5,
steps_per_epoch=steps_per_epoch,
validation_data=(X_test, y_cat_test),
callbacks=[early_stop],
batch_size=batch_size,
)
```

```
Epoch 1/5
1562/1562 [==============================] - 493s 314ms/step - loss: 1.6169 - accuracy: 0.4169 - precision: 0.6305 - recall: 0.2069 - va
Epoch 2/5
 217/1562 [===>..........................] - ETA: 6:49 - loss: 1.3383 - accuracy: 0.5262 - precision: 0.6972 - recall: 0.3426
```

Loss function Evaluation the training loss is decreasing while the validation loss is increasing, it may indicate that the model is overfitting to the training data and is not able to generalize well to new data. On the other hand, if both the training loss and validation loss are decreasing, it indicates that the model is learning well from the training data and is able to generalize well to new data.

Accuracy: If the training accuracy is increasing while the validation accuracy is decreasing, it may indicate that the model is overfitting to the training data and is not able to generalize well to new data. On the other hand, if both the training accuracy and validation accuracy are increasing, it indicates that the model is learning well from the training data and is able to generalize well to new data.

```
# Model Evaluation
plt.figure(figsize=(12, 16))
#Loss Function Evalution
plt.subplot(4, 2, 1)
plt.plot(r.history['loss'], label='Loss')
plt.plot(r.history['val_loss'], label='val_Loss')
plt.title('Loss Function Evolution')
plt.legend()

#Accuracy Function Evalution
plt.subplot(4, 2, 2)
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.title('Accuracy Function Evolution')
plt.legend()

#Precision Function Evalution
plt.subplot(4, 2, 3)
plt.plot(r.history['precision'], label='precision')
plt.plot(r.history['val_precision'], label='val_precision')
plt.title('Precision Function Evolution')
plt.legend()

#Recall Function Evalution
plt.subplot(4, 2, 4)
plt.plot(r.history['recall'], label='recall')
plt.plot(r.history['val_recall'], label='val_recall')
plt.title('Recall Function Evolution')
plt.legend()
```

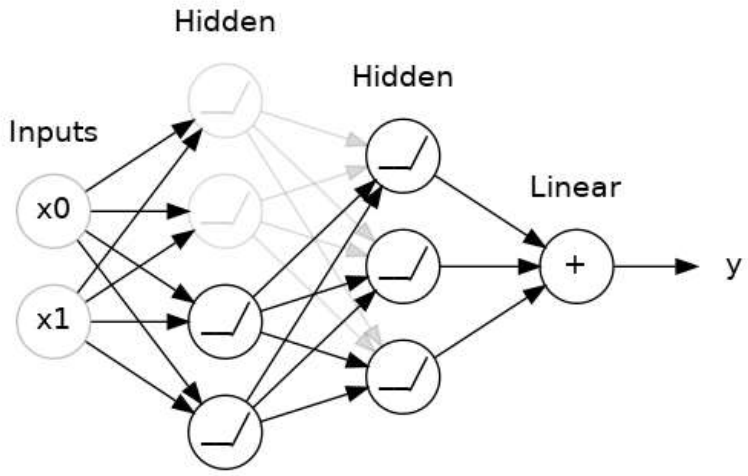
Experiment-9

Dropout

The first of these is the "dropout layer", which can help correct overfitting.

In the last lesson we talked about how overfitting is caused by the network learning spurious patterns in the training data. To recognize these spurious patterns a network will often rely on very a specific combinations of weight, a kind of "conspiracy" of weights. Being so specific, they tend to be fragile: remove one and the conspiracy falls apart.

This is the idea behind **dropout**. To break up these conspiracies, we randomly *drop out* some fraction of a layer's input units every step of training, making it much harder for the network to learn those spurious patterns in the training data. Instead, it has to search for broad, general patterns, whose weight patterns tend to be more robust.



Here, 50% dropout has been added between the two hidden layers.

You could also think about dropout as creating a kind of *ensemble* of networks. The predictions will no longer be made by one big network, but instead by a committee of smaller networks. Individuals in the committee tend to make different kinds of mistakes, but be right at the same time, making the committee as a whole better than any individual. (If you're familiar with random forests as an ensemble of decision trees, it's the same idea.)

Adding Dropout

In Keras, the dropout rate argument `rate` defines what percentage of the input units to shut off. Put the `Dropout` layer just before the layer you want the dropout applied to:

```
keras.Sequential([
    # ...
    layers.Dropout(rate=0.3), # apply 30% dropout to the next layer
    layers.Dense(16),
    # ...
])
```

Batch Normalization

The next special layer we'll look at performs "batch normalization" (or "batchnorm"), which can help correct training that is slow or unstable.

With neural networks, it's generally a good idea to put all of your data on a common scale, perhaps with something like scikit-learn's StandardScaler or MinMaxScaler. The reason is that SGD will shift the network weights in proportion to how large an activation the data produces. Features that tend to produce activations of very different sizes can make for unstable training behavior.

Now, if it's good to normalize the data before it goes into the network, maybe also normalizing inside the network would be better! In fact, we have a special kind of layer that can do this, the **batch normalization layer**. A batch normalization layer looks at each batch as it comes in, first

normalizing the batch with its own mean and standard deviation, and then also putting the data on a new scale with two trainable rescaling parameters. Batchnorm, in effect, performs a kind of coordinated rescaling of its inputs.

Most often, batchnorm is added as an aid to the optimization process (though it can sometimes also help prediction performance). Models with batchnorm tend to need fewer epochs to complete training. Moreover, batchnorm can also fix various problems that can cause the training to get "stuck". Consider adding batch normalization to your models, especially if you're having trouble during training.

Adding Batch Normalization

It seems that batch normalization can be used at almost any point in a network. You can put it after a layer...

```
layers.Dense(16, activation='relu'),
layers.BatchNormalization(),
```

... or between a layer and its activation function:

```
layers.Dense(16),
layers.BatchNormalization(),
layers.Activation('relu'),
```

And if you add it as the first layer of your network it can act as a kind of adaptive preprocessor, standing in for something like Sci-Kit Learn's `StandardScaler`.

Example - Using Dropout and Batch Normalization

Let's continue developing the *Red Wine* model. Now we'll increase the capacity even more, but add dropout to control overfitting and batch normalization to speed up optimization. This time, we'll also leave off standardizing the data, to demonstrate how batch normalization can stabalize the training.

```
# Setup plotting
import matplotlib.pyplot as plt

plt.style.use('seaborn-whitegrid')
# Set Matplotlib defaults
plt.rc('figure', autolayout=True)
plt.rc('axes', labelweight='bold', labelsize='large',
       titleweight='bold', titlesize=18, titlepad=10)


import pandas as pd
red_wine = pd.read_csv('red-wine.csv')

# Create training and validation splits
df_train = red_wine.sample(frac=0.7, random_state=0)
df_valid = red_wine.drop(df_train.index)

# Split features and target
X_train = df_train.drop('quality', axis=1)
X_valid = df_valid.drop('quality', axis=1)
y_train = df_train['quality']
y_valid = df_valid['quality']
```

```
<ipython-input-1-dc8b72943de1>:4: MatplotlibDeprecationWarning: The seaborn styles shipped by Matplotlib are deprecated since 3.6, as th
  plt.style.use('seaborn-whitegrid')
```

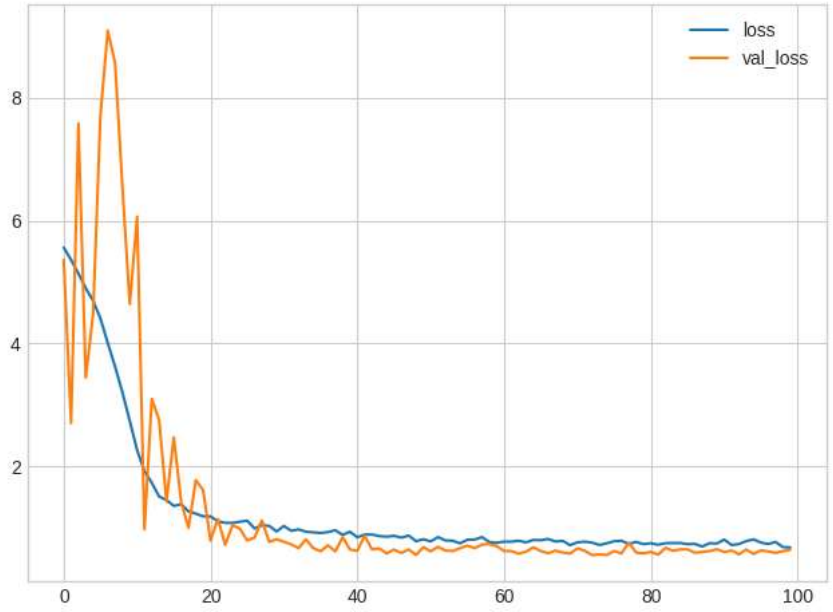
When adding dropout, you may need to increase the number of units in your `Dense` layers.

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(1024, activation='relu', input_shape=[11]),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
```

```
    layers.BatchNormalization(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1),
])
```

There's nothing to change this time in how we set up the training.

```
model.compile(
    optimizer='adam',
    loss='mae',
)

history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=100,
    verbose=0,
)


# Show the learning curves
history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot();
```



You'll typically get better performance if you standardize your data before using it for training. That we were able to use the raw data at all, however, shows how effective batch normalization can be on more difficult datasets.

Experiment-10

Plant Disease Prediction using YOLO

```
# clone YOLOv5 repository
!git clone https://github.com/ultralytics/yolov5  # clone repo
%cd yolov5
!git reset --hard 064365d8683fd002e9ad789c1e91fa3d021b44f0
```

```
Cloning into 'yolov5'...
remote: Enumerating objects: 15123, done.
remote: Counting objects: 100% (78/78), done.
remote: Compressing objects: 100% (66/66), done.
remote: Total 15123 (delta 42), reused 33 (delta 12), pack-reused 15045
Receiving objects: 100% (15123/15123), 14.19 MiB | 13.75 MiB/s, done.
Resolving deltas: 100% (10353/10353), done.
/content/yolov5
HEAD is now at 064365d Update parse_opt() in export.py to work as in train.py (#10789)
```

```
# install dependencies as necessary
!pip install -qr requirements.txt  # install dependencies (ignore errors)
import torch

from IPython.display import Image, clear_output  # to display images
from utils.downloads import attempt_download  # to download models/datasets

# clear_output()
print('Setup complete. Using torch %s %s' % (torch.__version__, torch.cuda.get_device_properties(0) if torch.cuda.is_available() else 'CPU'))
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 184.0/184.0 KB 12.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.7/62.7 KB 6.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 79.0 MB/s eta 0:00:00
Setup complete. Using torch 1.13.1+cu116 _CudaDeviceProperties(name='Tesla T4', major=7, minor=5, total_memory=15109MB, multi_processor_
```

```
#follow the link below to get your download code from from Roboflow
!pip install -q roboflow
from roboflow import Roboflow
rf = Roboflow(model_format="yolov5", notebook="roboflow-yolov5")
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 49.0/49.0 KB 3.3 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 67.8/67.8 KB 6.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 54.5/54.5 KB 4.8 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 140.6/140.6 KB 17.3 MB/s eta 0:00:00
  Building wheel for wget (setup.py) ... done
upload and label your dataset, and get an API KEY here: https://app.roboflow.com/?model=yolov5&ref=roboflow-yolov5
```

```
pwd
```

```
'/content/yolov5'
```

```
%cd /content/yolov5
#after following the link above, recieve python code with these fields filled in
#from roboflow import Roboflow
#rf = Roboflow(api_key="YOUR API KEY HERE")
#project = rf.workspace().project("YOUR PROJECT")
#dataset = project.version("YOUR VERSION").download("yolov5")
```

```
/content/yolov5
```

```
#!pip install roboflow

#from roboflow import Roboflow
rf = Roboflow(api_key="HkTO1slI7QEAkwGT6Dxq")
project = rf.workspace("joseph-nelson").project("plantdoc")
dataset = project.version(3).download("yolov5")
```

```
loading Roboflow workspace...
loading Roboflow project...
```

```
Downloading Dataset Version Zip in PlantDoc-3 to yolov5pytorch: 100% [461479587 / 461479587] bytes
Extracting Dataset Version Zip to PlantDoc-3 in yolov5pytorch:: 100%|██████████| 5147/5147 [00:03<00:00, 1685.98it/s]
```

```
%cat {dataset.location}/data.yaml
```

```
names:
- Apple Scab Leaf
- Apple leaf
- Apple rust leaf
- Bell_pepper leaf spot
- Bell_pepper leaf
- Blueberry leaf
- Cherry leaf
- Corn Gray leaf spot
- Corn leaf blight
- Corn rust leaf
- Peach leaf
- Potato leaf early blight
- Potato leaf late blight
- Potato leaf
- Raspberry leaf
- Soyabean leaf
- Soybean leaf
- Squash Powdery mildew leaf
- Strawberry leaf
- Tomato Early blight leaf
- Tomato Septoria leaf spot
- Tomato leaf bacterial spot
- Tomato leaf late blight
- Tomato leaf mosaic virus
- Tomato leaf yellow virus
- Tomato leaf
- Tomato mold leaf
- Tomato two spotted spider mites leaf
- grape leaf black rot
- grape leaf
nc: 30
train: PlantDoc-3/train/images
val: PlantDoc-3/test/images
```

Define Model Configuration and Architecture

We will write a yaml script that defines the parameters for our model like the number of classes, anchors, and each layer.

You do not need to edit these cells, but you may.

```
# define number of classes based on YAML
import yaml
with open(dataset.location + "/data.yaml", 'r') as stream:
    num_classes = str(yaml.safe_load(stream)['nc'])
```

```
#this is the model configuration we will use for our tutorial
%cat /content/yolov5/models/yolov5s.yaml
```

```
# YOLOv5 🚀 by Ultralytics, GPL-3.0 license

# Parameters
nc: 80  # number of classes
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple
anchors:
  - [10,13, 16,30, 33,23]  # P3/8
  - [30,61, 62,45, 59,119]  # P4/16
  - [116,90, 156,198, 373,326]  # P5/32

# YOLOv5 v6.0 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Conv, [64, 6, 2, 2]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, C3, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 6, C3, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, C3, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 3, C3, [1024]],
   [-1, 1, SPPF, [1024, 5]],  # 9
```

```
       ]

     # YOLOv5 v6.0 head
     head:
       [[-1, 1, Conv, [512, 1, 1]],
        [-1, 1, nn.Upsample, [None, 2, 'nearest']],
        [[-1, 6], 1, Concat, [1]],  # cat backbone P4
        [-1, 3, C3, [512, False]],  # 13

        [-1, 1, Conv, [256, 1, 1]],
        [-1, 1, nn.Upsample, [None, 2, 'nearest']],
        [[-1, 4], 1, Concat, [1]],  # cat backbone P3
        [-1, 3, C3, [256, False]],  # 17 (P3/8-small)

        [-1, 1, Conv, [256, 3, 2]],
        [[-1, 14], 1, Concat, [1]],  # cat head P4
        [-1, 3, C3, [512, False]],  # 20 (P4/16-medium)

        [-1, 1, Conv, [512, 3, 2]],
        [[-1, 10], 1, Concat, [1]],  # cat head P5
        [-1, 3, C3, [1024, False]],  # 23 (P5/32-large)

        [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)
       ]


#customize iPython writefile so we can write variables
from IPython.core.magic import register_line_cell_magic

@register_line_cell_magic
def writetemplate(line, cell):
    with open(line, 'w') as f:
        f.write(cell.format(**globals()))


%%writetemplate /content/yolov5/models/custom_yolov5s.yaml

# parameters
nc: {num_classes}  # number of classes
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23]  # P3/8
  - [30,61, 62,45, 59,119]  # P4/16
  - [116,90, 156,198, 373,326]  # P5/32

# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, BottleneckCSP, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 1, SPP, [1024, [5, 9, 13]]],
   [-1, 3, BottleneckCSP, [1024, False]],  # 9
  ]

# YOLOv5 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P4
   [-1, 3, BottleneckCSP, [512, False]],  # 13

   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 4], 1, Concat, [1]],  # cat backbone P3
   [-1, 3, BottleneckCSP, [256, False]],  # 17 (P3/8-small)

   [-1, 1, Conv, [256, 3, 2]],
   [[-1, 14], 1, Concat, [1]],  # cat head P4
   [-1, 3, BottleneckCSP, [512, False]],  # 20 (P4/16-medium)

   [-1, 1, Conv, [512, 3, 2]],
```

```
   [[-1, 10], 1, Concat, [1]],  # cat head P5
   [-1, 3, BottleneckCSP, [1024, False]],  # 23 (P5/32-large)

   [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)
  ]
```

Train Custom YOLOv5 Detector

Next, we'll fire off training!

Here, we are able to pass a number of arguments:

- **img:** define input image size
- **batch:** determine batch size
- **epochs:** define the number of training epochs. (Note: often, 3000+ are common here!)
- **data:** set the path to our yaml file
- **cfg:** specify our model configuration
- **weights:** specify a custom path to weights. (Note: you can download weights from the Ultralytics Google Drive folder)
- **name:** result names
- **nosave:** only save the final checkpoint
- **cache:** cache images for faster training

```
# train yolov5s on custom data for 100 epochs
# time its performance
%%time
%cd /content/yolov5/
!python train.py --img 416 --batch 16 --epochs 3 --data {dataset.location}/data.yaml --cfg ./models/custom_yolov5s.yaml --weights '' --name y
```

```
/content/yolov5
train: weights=, cfg=./models/custom_yolov5s.yaml, data=/content/yolov5/PlantDoc-3/data.yaml, hyp=data/hyps/hyp.scratch-low.yaml, epo
github: ⚠️ YOLOv5 is out of date by 25 commits. Use `git pull` or `git clone https://github.com/ultralytics/yolov5` to update.
YOLOv5 🚀 v7.0-72-g064365d Python-3.8.10 torch-1.13.1+cu116 CUDA:0 (Tesla T4, 15110MiB)

hyperparameters: lr0=0.01, lrf=0.01, momentum=0.937, weight_decay=0.0005, warmup_epochs=3.0, warmup_momentum=0.8, warmup_bias_lr=0.1
ClearML: run 'pip install clearml' to automatically track, visualize and remotely train YOLOv5 🚀 in ClearML
Comet: run 'pip install comet_ml' to automatically track and visualize YOLOv5 🚀 runs in Comet
TensorBoard: Start with 'tensorboard --logdir runs/train', view at http://localhost:6006/
2023-02-13 11:03:34.495527: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Dee
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-02-13 11:03:35.356410: W tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic libra
2023-02-13 11:03:35.356540: W tensorflow/compiler/xla/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic libra
2023-02-13 11:03:35.356560: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libra
Downloading https://ultralytics.com/assets/Arial.ttf to /root/.config/Ultralytics/Arial.ttf...
100% 755k/755k [00:00<00:00, 119MB/s]

                 from  n    params  module                                  arguments
  0                -1  1      3520  models.common.Focus                     [3, 32, 3]
  1                -1  1     18560  models.common.Conv                      [32, 64, 3, 2]
  2                -1  1     19904  models.common.BottleneckCSP             [64, 64, 1]
  3                -1  1     73984  models.common.Conv                      [64, 128, 3, 2]
  4                -1  3    161152  models.common.BottleneckCSP             [128, 128, 3]
  5                -1  1    295424  models.common.Conv                      [128, 256, 3, 2]
  6                -1  3    641792  models.common.BottleneckCSP             [256, 256, 3]
  7                -1  1   1180672  models.common.Conv                      [256, 512, 3, 2]
  8                -1  1    656896  models.common.SPP                       [512, 512, [5, 9, 13]]
  9                -1  1   1248768  models.common.BottleneckCSP             [512, 512, 1, False]
 10                -1  1    131584  models.common.Conv                      [512, 256, 1, 1]
 11                -1  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
 12          [-1, 6]  1         0  models.common.Concat                    [1]
 13                -1  1    378624  models.common.BottleneckCSP             [512, 256, 1, False]
 14                -1  1     33024  models.common.Conv                      [256, 128, 1, 1]
 15                -1  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
 16          [-1, 4]  1         0  models.common.Concat                    [1]
 17                -1  1     95104  models.common.BottleneckCSP             [256, 128, 1, False]
 18                -1  1    147712  models.common.Conv                      [128, 128, 3, 2]
 19         [-1, 14]  1         0  models.common.Concat                    [1]
 20                -1  1    313088  models.common.BottleneckCSP             [256, 256, 1, False]
 21                -1  1    590336  models.common.Conv                      [256, 256, 3, 2]
 22         [-1, 10]  1         0  models.common.Concat                    [1]
 23                -1  1   1248768  models.common.BottleneckCSP             [512, 512, 1, False]
 24      [17, 20, 23]  1     94395  models.yolo.Detect                      [30, [[10, 13, 16, 30, 33, 23], [30, 61, 62, 45, 59, 119
custom_YOLOv5s summary: 233 layers, 7333307 parameters, 7333307 gradients

AMP: checks passed ✅
optimizer: SGD(lr=0.01) with parameter groups 59 weight(decay=0.0), 70 weight(decay=0.0005), 62 bias
albumentations: Blur(p=0.01, blur_limit=(3, 7)), MedianBlur(p=0.01, blur_limit=(3, 7)), ToGray(p=0.01), CLAHE(p=0.01, clip_limit=(1,
```

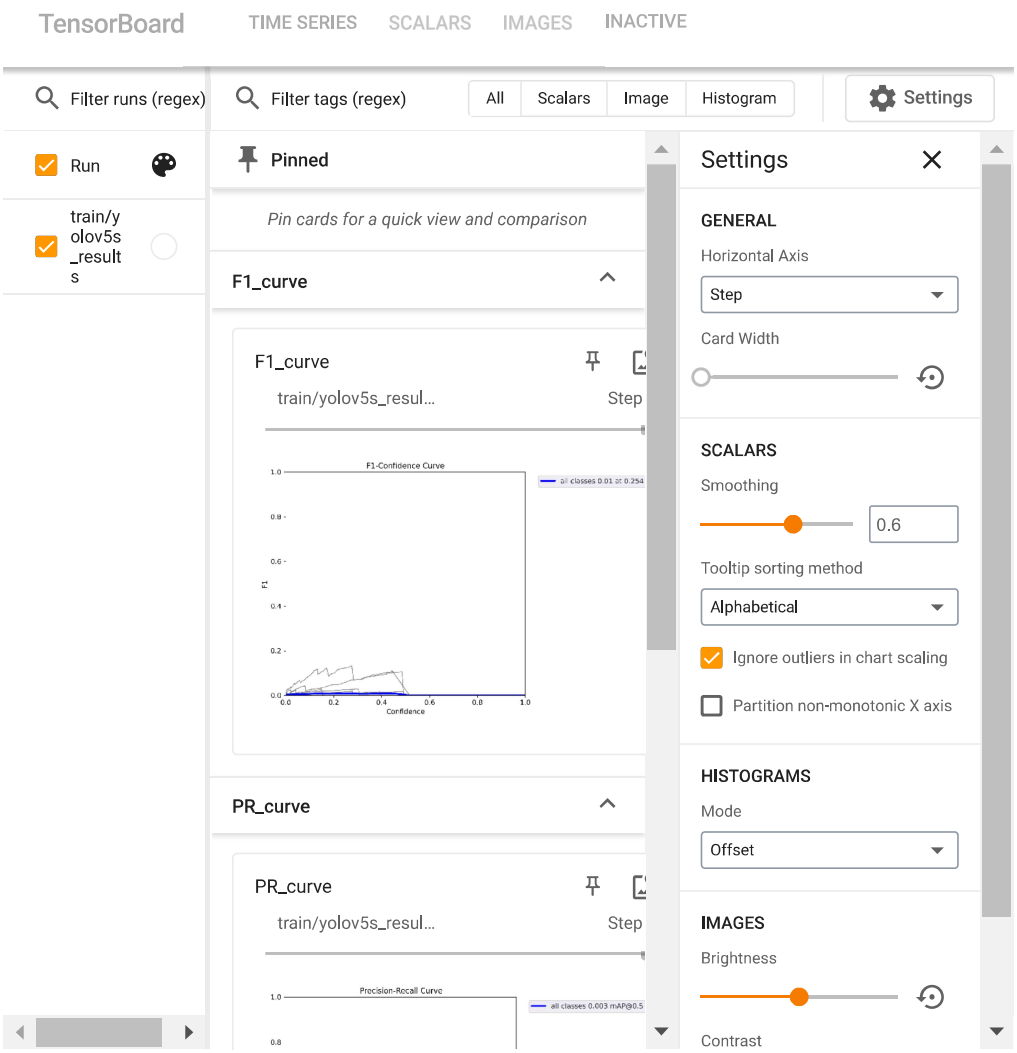
```
train: Scanning /content/yolov5/PlantDoc-3/train/labels... 2330 images, 10 backgrounds, 0 corrupt: 100% 2330/2330 [00:01<00:00, 1664
train: New cache created: /content/yolov5/PlantDoc-3/train/labels.cache
train: Caching images (0.8GB ram): 100% 2330/2330 [00:43<00:00, 53.90it/s]
val: Scanning /content/yolov5/PlantDoc-3/test/labels... 239 images, 1 backgrounds, 0 corrupt: 100% 239/239 [00:00<00:00, 680.85it/s]
val: New cache created: /content/yolov5/PlantDoc-3/test/labels.cache
val: Caching images (0.1GB ram): 100% 239/239 [00:07<00:00, 32.50it/s]

AutoAnchor: 5.22 anchors/target, 1.000 Best Possible Recall (BPR). Current anchors are a good fit to dataset ✅
Plotting labels to runs/train/yolov5s_results/labels.jpg...
```

Evaluate Custom YOLOv5 Detector Performance

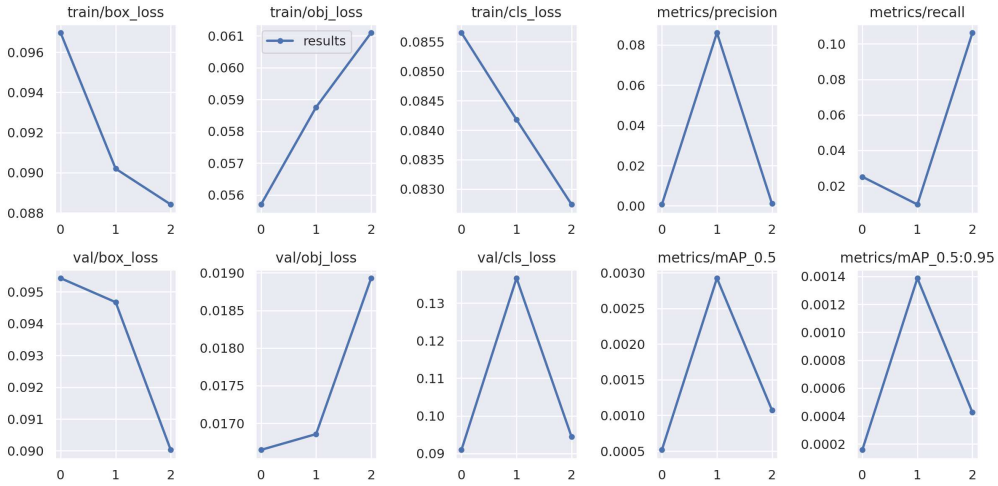
Training losses and performance metrics are saved to Tensorboard and also to a logfile defined above with the **--name** flag when we train. In our case, we named this `yolov5s_results`. (If given no name, it defaults to `results.txt`.) The results file is plotted as a png after training completes.

Note from Glenn: Partially completed `results.txt` files can be plotted with `from utils.utils import plot_results; plot_results()`.

```
# Start tensorboard
# Launch after you have started training
# logs save in the folder "runs"
%load_ext tensorboard
%tensorboard --logdir runs
```




```
# we can also output some older school graphs if the tensor board isn't working for whatever reason...
from utils.plots import plot_results  # plot results.txt as results.png
Image(filename='/content/yolov5/runs/train/yolov5s_results/results.png', width=1000)  # view results.png
```



Curious? Visualize Our Training Data with Labels

After training starts, view `train*.jpg` images to see training images, labels and augmentation effects.

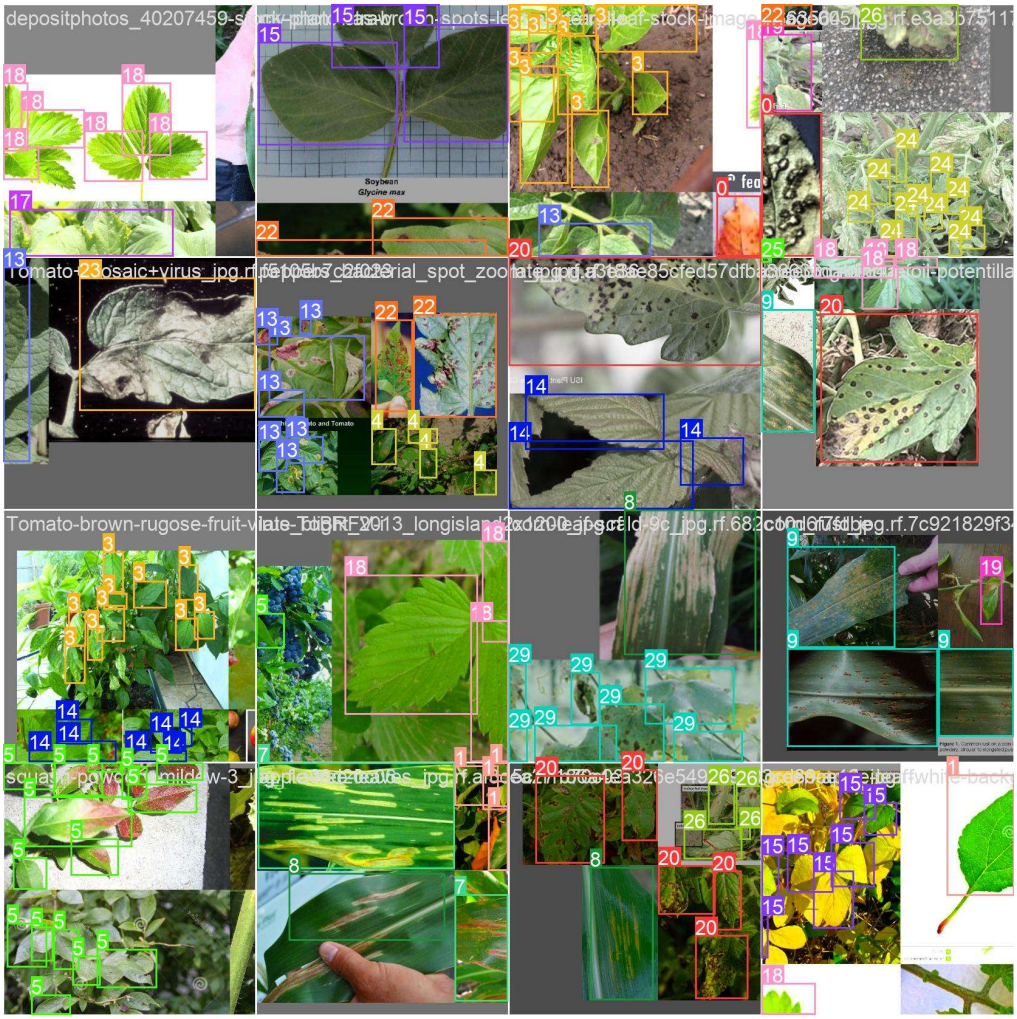
Note a mosaic dataloader is used for training (shown below), a new dataloading concept developed by Glenn Jocher and first featured in [YOLOv4](#).

```
# first, display our ground truth data
print("GROUND TRUTH TRAINING DATA:")
Image(filename='/content/yolov5/runs/train/yolov5s_results/val_batch0_labels.jpg', width=900)
```

```
GROUND TRUTH TRAINING DATA:
```


```
# print out an augmented training example
print("GROUND TRUTH AUGMENTED TRAINING DATA:")
Image(filename='/content/yolov5/runs/train/yolov5s_results/train_batch0.jpg', width=900)
```

GROUND TRUTH AUGMENTED TRAINING DATA:



Run Inference With Trained Weights

Run inference with a pretrained checkpoint on contents of `test/images` folder downloaded from Roboflow.

```
# trained weights are saved by default in our weights folder
%ls runs/
```

```
train/
```

```
%ls runs/train/yolov5s_results/weights
```

```
best.pt  last.pt
```

```
# when we ran this, we saw .007 second inference time. That is 140 FPS on a TESLA P100!
# use the best weights!
%cd /content/yolov5/
!python detect.py --weights runs/train/yolov5s_results/weights/best.pt --img 416 --conf 0.4 --source /content/yolov5/PlantDoc-3/test/images
```

```
/content/yolov5
detect: weights=['runs/train/yolov5s_results/weights/best.pt'], source=/content/yolov5/PlantDoc-3/test/images, data=data/coco128.yam
YOLOv5 🚀 v7.0-72-g064365d Python-3.8.10 torch-1.13.1+cu116 CUDA:0 (Tesla T4, 15110MiB)

Fusing layers...
custom_YOLOv5s summary: 182 layers, 7324731 parameters, 0 gradients
image 1/239 /content/yolov5/PlantDoc-3/test/images/%2320+Bacterial+Spot+and+Speck_jpg.rf.529f5b7c182e0b156c17d5acc218f6ff.jpg: 320x41
image 2/239 /content/yolov5/PlantDoc-3/test/images/0000_jpg.rf.91a736e0600590aab4fd34a5cee826df.jpg: 416x416 (no detections), 10.7ms
image 3/239 /content/yolov5/PlantDoc-3/test/images/000_jpg.rf.db0f154a8e9021ffe589c51961b95881.jpg: 320x416 (no detections), 10.1ms
image 4/239 /content/yolov5/PlantDoc-3/test/images/00_jpg.rf.8c1ce233a1d6ed77f1828dedd3e51587.jpg: 320x416 (no detections), 9.4ms
image 5/239 /content/yolov5/PlantDoc-3/test/images/00pe_jpg.rf.ba197270529fea4c38c4bb820705e257.jpg: 416x320 (no detections), 12.9ms
image 6/239 /content/yolov5/PlantDoc-3/test/images/01_jpg.rf.e6e9899cdc03b9bc6fd88dfa8165e023.jpg: 320x416 (no detections), 9.5ms
image 7/239 /content/yolov5/PlantDoc-3/test/images/02_-Rust-2017-207u24s_jpg.rf.65d6d8dde198e95d5c33acf7f4c0a7e9.jpg: 256x416 (no det
image 8/239 /content/yolov5/PlantDoc-3/test/images/02c_jpg.rf.c1f37f4ee73224dd5724c72322cbf287.jpg: 256x416 (no detections), 9.7ms
image 9/239 /content/yolov5/PlantDoc-3/test/images/039b47d574bc4bb8a14259a1cd96a741_jpg.rf.936d126db84ade65cd15a5c32985a12f.jpg: 320x
```

```
image 10/239 /content/yolov5/PlantDoc-3/test/images/03gb_jpg.rf.6df176e0e0fc5a7327c39751e0c8daf2.jpg: 288x416 (no detections), 14.4m
image 11/239 /content/yolov5/PlantDoc-3/test/images/052609%20Hartman%20Crabapple%20scab%20single%20leaf_JPG_jpg.rf.d8e544802ba0bca79
image 12/239 /content/yolov5/PlantDoc-3/test/images/0605_Rust-induced_leafspot_jpg.rf.ef434ba62c50e0a9e0ec1dad1ad7bdb1.jpg: 256x416
image 13/239 /content/yolov5/PlantDoc-3/test/images/0796_20graylssymt_jpg.rf.10da66d4b5680911525766f76b4daa21.jpg: 352x416 (no detect
image 14/239 /content/yolov5/PlantDoc-3/test/images/0796_39maizerust_jpg.rf.3848688eb6feebf09c81332dff652fed.jpg: 288x416 (no detect
image 15/239 /content/yolov5/PlantDoc-3/test/images/0796_40comrust_jpg.rf.e1c4a903881ee72b7a15cb88114b57a2.jpg: 288x416 (no detectio
image 16/239 /content/yolov5/PlantDoc-3/test/images/0796_47southrust_jpg.rf.0d2eded242b395947062ff463fd2eb1b.jpg: 256x416 (no detect
image 17/239 /content/yolov5/PlantDoc-3/test/images/0796_52srusttelia_jpg.rf.10671bdc41af58802dc6b81f8ebfcfad.jpg: 320x416 (no detect
image 18/239 /content/yolov5/PlantDoc-3/test/images/07_17_18-Common_Tomato_Diseases_Canker-258x300_jpg.rf.b3ae6ef12dbabe17f46714b8f7e
image 19/239 /content/yolov5/PlantDoc-3/test/images/07c_jpg.rf.a07b23c64861bb3b478033d358811272.jpg: 256x416 (no detections), 10.3ms
image 20/239 /content/yolov5/PlantDoc-3/test/images/07feb_ma_sbr3_JPG_jpg.rf.91a39b40cbd02c3374060f3d6b043408.jpg: 288x416 (no detect
image 21/239 /content/yolov5/PlantDoc-3/test/images/0_jpg.rf.cd543e6a413f1712097877ec9dae8268.jpg: 320x416 1 Corn leaf blight, 12.0m
image 22/239 /content/yolov5/PlantDoc-3/test/images/0c_jpg.rf.07df07e775b5a4186290271decb853f9.jpg: 192x416 (no detections), 13.1ms
image 23/239 /content/yolov5/PlantDoc-3/test/images/100983448_jpg.rf.d0ead28bf9be6b6cf77d5a66b2fdf259.jpg: 416x416 1 Peach leaf, 2 Pe
image 24/239 /content/yolov5/PlantDoc-3/test/images/10148582-green-leaf-of-pepper_jpg.rf.8be9dee647ba6c943cfc446d043a5e85.jpg: 320x41
image 25/239 /content/yolov5/PlantDoc-3/test/images/11-40580_5_jpg.rf.7cf3bd8a59b254381db18fa8db7f3bb0.jpg: 224x416 (no detections),
image 26/239 /content/yolov5/PlantDoc-3/test/images/110822-206-Tomato-blight_jpg.rf.b0b8fbc9c7f3e37bbde049053ef62a6d.jpg: 288x416 (no
image 27/239 /content/yolov5/PlantDoc-3/test/images/12-19striprustJIM_jpg.rf.047bdd7df272f87ec788d8bb54af0c7b.jpg: 416x160 (no detect
image 28/239 /content/yolov5/PlantDoc-3/test/images/1234080-Early-Blight_jpg.rf.40c98aeba920d65514d0c457cef848d8.jpg: 416x416 (no det
image 29/239 /content/yolov5/PlantDoc-3/test/images/1321189_jpg.rf.19622820196e4bc9c7a1cf81fd94f948.jpg: 160x416 (no detections), 13
image 30/239 /content/yolov5/PlantDoc-3/test/images/1355_50commonrust_jpg.rf.ac3882fd765b376b7097354a23949876.jpg: 416x288 (no detect
image 31/239 /content/yolov5/PlantDoc-3/test/images/1421_0_jpeg?itok=FMtmgePj_jpg.rf.8dbc928d09b476ff938160003d6fdc67.jpg: 320x416 1
image 32/239 /content/yolov5/PlantDoc-3/test/images/160314_web_jpg.rf.e1c6dcd50b34e41c4839a4b1e49c4cbf.jpg: 416x320 (no detections),
image 33/239 /content/yolov5/PlantDoc-3/test/images/1684_jpg.rf.49bb6aace2fd0583c10c60867449e050.jpg: 416x384 (no detections), 9.8ms
image 34/239 /content/yolov5/PlantDoc-3/test/images/17fc47_jpg.rf.d6d17a3e17035fdbdb8f4012b051779d.jpg: 416x320 (no detections), 12.6
image 35/239 /content/yolov5/PlantDoc-3/test/images/185161-004-EAF28842_jpg.rf.0cd20638224999d0ca76a2d62b5687e9.jpg: 416x416 (no dete
image 36/239 /content/yolov5/PlantDoc-3/test/images/18c_jpg.rf.ae48a4717be48a7fa4ee4a542283bb4b.jpg: 320x416 (no detections), 9.3ms
image 37/239 /content/yolov5/PlantDoc-3/test/images/1b321015-6e33-4f18-aade-888f4383fe92_jpeg_jpg.rf.8c6e7c4ebb80e2c8f36b1f37e46bc9ad
image 38/239 /content/yolov5/PlantDoc-3/test/images/20090710-lateblight_jpg.rf.d27151997b5271ea4f54deb265f95386.jpg: 320x416 (no dete
image 39/239 /content/yolov5/PlantDoc-3/test/images/2011-011_jpg.rf.bbed8cc476eb2d38b81c40fe7a493731.jpg: 320x416 (no detections), 9
image 40/239 /content/yolov5/PlantDoc-3/test/images/2013-08-20-06_jpg.rf.d93eec1a08c29cdf65eb90d48ef5b377.jpg: 320x416 (no detection
image 41/239 /content/yolov5/PlantDoc-3/test/images/20130519cedarapplerust_jpg.rf.478c755dfb97b6109b9571e1d6e707f1.jpg: 320x416 (no d
image 42/239 /content/yolov5/PlantDoc-3/test/images/20130610_110514_jpg.rf.25e7cf4b563b9a1f6417f983c2746cd9.jpg: 416x320 (no detectio
image 43/239 /content/yolov5/PlantDoc-3/test/images/20130802_111632_jpg.rf.0ee091f5ef23ae955796ec5836402584.jpg: 320x416 (no detectio
image 44/239 /content/yolov5/PlantDoc-3/test/images/2013Corn_GrayLeafSpot_0815_0003_JPG_jpg.rf.bccbb753bb16cfa931bd97b944a2e6d9.jpg:
image 45/239 /content/yolov5/PlantDoc-3/test/images/2015070295153021_jpg.rf.fbf08db1defddaa2223ed6cef1651563.jpg: 416x320 (no detect
image 46/239 /content/yolov5/PlantDoc-3/test/images/20180511_090912-14gtw8a-e1526047952754_jpg.rf.712c5b63f1ac7412f964900b3c404c85.j
image 47/239 /content/yolov5/PlantDoc-3/test/images/20180511_091133-24l1vhg-e1526047988236_jpg.rf.8045929fa5e6b257908942c9c9fedacd.j
image 48/239 /content/yolov5/PlantDoc-3/test/images/20180511_091252-1gy5xf5-e1526048000596_jpg.rf.0ca2d22b0453ef9c85dde39d36776594.j
image 49/239 /content/yolov5/PlantDoc-3/test/images/2256-body-1501555581-1_jpg.rf.7cabbe1c825c2013e21146fe1b273159.jpg: 320x416 (no d
image 50/239 /content/yolov5/PlantDoc-3/test/images/2540_600_jpg.rf.6e43636627265c1ff9b1b28777715ecf.jpg: 416x352 (no detections), 14
```

```
#display inference on ALL test images
#this looks much better with longer training above

import glob
from IPython.display import Image, display

for imageName in glob.glob('/content/yolov5/runs/detect/exp/*.jpg'): #assuming JPG
    display(Image(filename=imageName))
    print("\n")
```

Deploy Model Weights to Roboflow

Now that you have trained your custom detector, you can upload it to Roboflow to deploy you model to a Hosted API and edge containers.

```
project.version(dataset.version).deploy(model_type="yolov5", model_path=f"/content/yolov5/runs/train/yolov5s_results/")


#While your deployment is processing, checkout the deployment docs to take your model to most destinations https://docs.roboflow.com/inferenc


#Run inference on your model on a persistant, auto-scaling, cloud API

#load model
model = project.version(dataset.version).model

#choose random test set image
import os, random
test_set_loc = dataset.location + "/test/images/"
random_test_image = random.choice(os.listdir(test_set_loc))
print("running inference on " + random_test_image)

pred = model.predict(test_set_loc + random_test_image, confidence=40, overlap=30).json()
pred
```

Experiment-11

Text Classification using RNN

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

# Load the IMDB Reviews dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.imdb.load_data(num_words=10000)
```

```
print(x_train)
```

```
[list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 4
 list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119, 954, 189, 102, 5,
 list([1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13, 71, 149, 14, 22, 112, 4, 2401, 311, 12, 16, 3711, 33, 75, 43,
 ...
 list([1, 11, 6, 230, 245, 6401, 9, 6, 1225, 446, 2, 45, 2174, 84, 8322, 4007, 21, 4, 912, 84, 2, 325, 725, 134, 2, 1715, 84, 5, 36, 28,
 list([1, 1446, 7079, 69, 72, 3305, 13, 610, 930, 8, 12, 582, 23, 5, 16, 484, 685, 54, 349, 11, 4120, 2959, 45, 58, 1466, 13, 197, 12, 1
 list([1, 17, 6, 194, 337, 7, 4, 204, 22, 45, 254, 8, 106, 14, 123, 4, 2, 270, 2, 5, 2, 2, 732, 2098, 101, 405, 39, 14, 1034, 4, 1310, 9
```

Padding is a common technique used in natural language processing (NLP) to ensure that all input sequences have the same length. This is often necessary because many NLP models, such as neural networks, require fixed-length input sequences.

```
# Pad the sequences to have equal length
max_len = 500
x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train, maxlen=max_len)
x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test, maxlen=max_len)
```

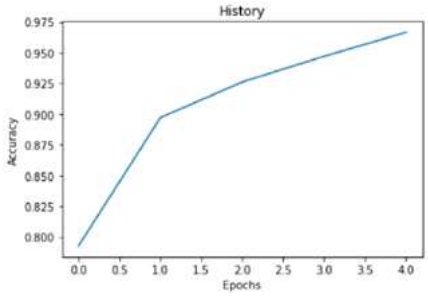
```
# Set the input and output dimensions
input_dim = 10000
output_dim = 1
# Create the input layer
inputs = tf.keras.Input(shape=(None,), dtype="int32")

# Create the model
x = tf.keras.layers.Embedding(input_dim, 128)(inputs)
x = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True))(x)
x = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64))(x)
outputs = tf.keras.layers.Dense(output_dim, activation="sigmoid")(x)
model = tf.keras.Model(inputs, outputs)
```

```
# Compile the model
model.compile("adam", "binary_crossentropy", metrics=["accuracy"])
```

```
# Train the model
batch_size = 32
epochs = 5
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test))
```

```
Epoch 1/5
782/782 [==============================] - ETA: 0s - loss: 0.4999 - accuracy: 0.7549
```

```
# Plot the accuracy
fig = plt.plot(history.history['accuracy'])
title = plt.title("History")
xlabel = plt.xlabel("Epochs")
ylabel = plt.ylabel("Accuracy")
```


History
Accuracy
Epochs
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Experiment-12

Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. Introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network.

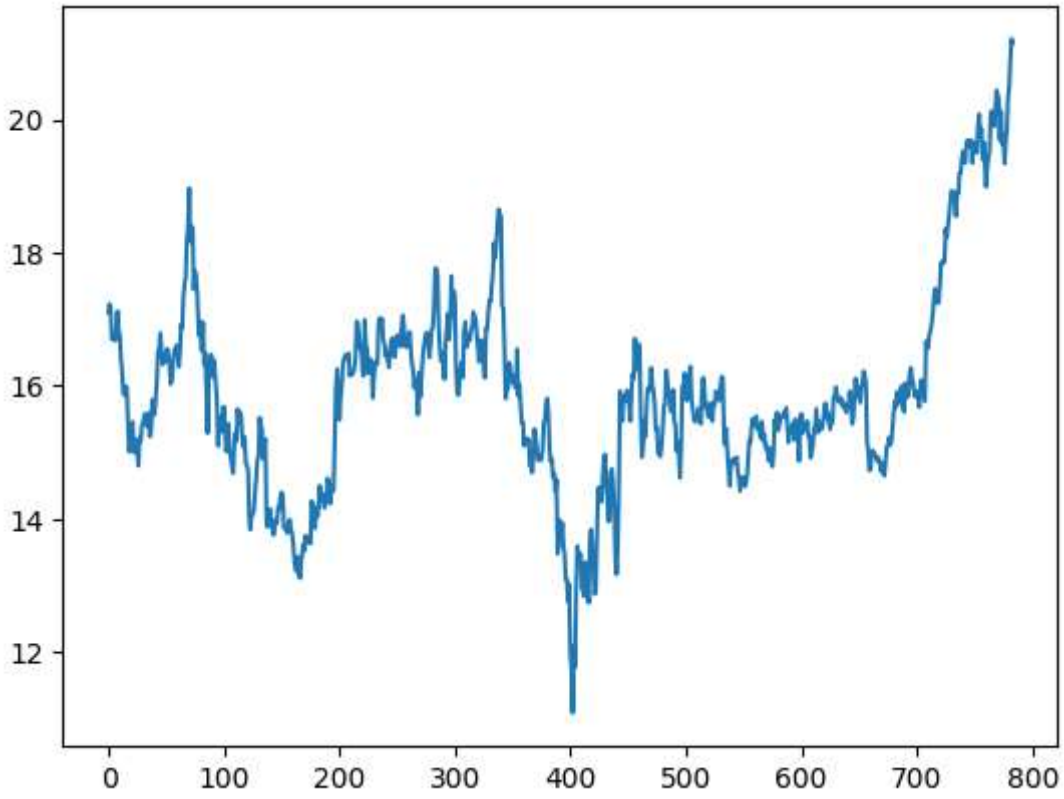
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

from subprocess import check_output
from keras.layers.core import Dense, Activation, Dropout
from keras.layers import LSTM
from keras.models import Sequential
from sklearn.model_selection import train_test_split
import time #helper libraries
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
from numpy import newaxis


prices_dataset =  pd.read_csv('prices.csv',header=0)
prices_dataset
```

	date	symbol	open	close	low	high	volu
0	2016-01-05 00:00:00	WLTW	123.430000	125.839996	122.309998	126.250000	216360
1	2016-01-06 00:00:00	WLTW	125.239998	119.980003	119.940002	125.540001	238640
2	2016-01-07 00:00:00	WLTW	116.379997	114.949997	114.930000	119.739998	248950

```
yahoo = prices_dataset[prices_dataset['symbol']=='YHOO']
yahoo_stock_prices = yahoo.close.values.astype('float32')
yahoo_stock_prices = yahoo_stock_prices.reshape(784, 1)
yahoo_stock_prices.shape
```

```
(784, 1)
```

```
plt.plot(yahoo_stock_prices)
plt.show()

scaler = MinMaxScaler(feature_range=(0, 1))
yahoo_stock_prices = scaler.fit_transform(yahoo_stock_prices)
```




```
train_size = int(len(yahoo_stock_prices) * 0.80)
test_size = len(yahoo_stock_prices) - train_size
train, test = yahoo_stock_prices[0:train_size,:], yahoo_stock_prices[train_size:len(yahoo_sto
print(len(train), len(test))
```

```
627 157
```

```
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)


# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)


trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))


#Step 2 Build Model
model = Sequential()

model.add(LSTM(units=50, return_sequences=True, input_shape=(trainX.shape[1], 1)))
model.add(Dropout(0.2))

model.add(LSTM(
    100,
    return_sequences=False))
model.add(Dropout(0.2))

model.add(Dense(units=1))
model.add(Activation('linear'))

start = time.time()
model.compile(loss='mse', optimizer='rmsprop')
print ('compilation time : ', time.time() - start)

     compilation time :  0.018996000289916992


model.fit(
    trainX,
    trainY,
    batch_size=128,
    epochs=10,
    validation_split=0.05)

     Epoch 1/10
     5/5 [==============================] - 7s 269ms/step - loss: 0.1850 - val_loss: 0.1309
     Epoch 2/10
     5/5 [==============================] - 0s 15ms/step - loss: 0.1436 - val_loss: 0.0983
     Epoch 3/10
```

```
5/5 [==============================] - 0s 17ms/step - loss: 0.1100 - val_loss: 0.0698
Epoch 4/10
5/5 [==============================] - 0s 15ms/step - loss: 0.0800 - val_loss: 0.0447
Epoch 5/10
5/5 [==============================] - 0s 14ms/step - loss: 0.0539 - val_loss: 0.0249
Epoch 6/10
5/5 [==============================] - 0s 17ms/step - loss: 0.0339 - val_loss: 0.0114
Epoch 7/10
5/5 [==============================] - 0s 15ms/step - loss: 0.0201 - val_loss: 0.0040
Epoch 8/10
5/5 [==============================] - 0s 15ms/step - loss: 0.0123 - val_loss: 8.7458e-0
Epoch 9/10
5/5 [==============================] - 0s 16ms/step - loss: 0.0095 - val_loss: 3.1441e-0
Epoch 10/10
5/5 [==============================] - 0s 15ms/step - loss: 0.0085 - val_loss: 4.1392e-0
<keras.callbacks.History at 0x7fedac4e6d70>
```

work in progress

```
def plot_results_multiple(predicted_data, true_data,length):
    plt.plot(scaler.inverse_transform(true_data.reshape(-1, 1))[length:])
    plt.plot(scaler.inverse_transform(np.array(predicted_data).reshape(-1, 1))[length:])
    plt.show()

#predict lenght consecutive values from a real one
def predict_sequences_multiple(model, firstValue,length):
    prediction_seqs = []
    curr_frame = firstValue

    for i in range(length):
        predicted = []

        print(model.predict(curr_frame[newaxis,:,:]))
        predicted.append(model.predict(curr_frame[newaxis,:,:])[0,0])

        curr_frame = curr_frame[0:]
        curr_frame = np.insert(curr_frame[0:], i+1, predicted[-1], axis=0)

        prediction_seqs.append(predicted[-1])

    return prediction_seqs

predict_length=5
predictions = predict_sequences_multiple(model, testX[0], predict_length)
print(scaler.inverse_transform(np.array(predictions).reshape(-1, 1)))
plot_results_multiple(predictions, testY, predict_length)
```

```
1/1 [==============================] - 1s 834ms/step
[[0.43460876]]
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 1s 836ms/step
[[0.87638855]]
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 28ms/step
[[1.4228654]]
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 25ms/step
[[2.09786]]
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 25ms/step
[[2.8798258]]
1/1 [==============================] - 0s 25ms/step
[[15.48824 ]
 [19.959051]
 [25.489397]
 [32.320343]
 [40.233833]]
```