

Hindi Vidya Prachar Samiti's
**Ramniranjan Jhunjhunwala College of Arts, Science &
Commerce**
(Empowered Autonomous College)



**Affiliated to
UNIVERSITY OF MUMBAI**

**DEPARTMENT OF INFORMATION TECHNOLOGY
2025 - 2026**

M.Sc. (IT) PART 2 - SEM IV

RJSPITE302P – Deep Learning

Name: Vishwakarma Shivam Suresh Sushila

Roll No: 6709

Hindi Vidya Prachar Samiti's
Ramniranjan Jhunjhunwala College of Arts, Science &
Commerce
(Empowered Autonomous College)

Certificate



This is to certify that Mr. Vishwakarma Shivam Suresh Suhila Roll No 6709 of M.Sc.(I.T.) Part-1 class has completed the required number of experiments in the subject of Deep Learning in the Department of Information Technology during the academic year 2025 - 2026 .

Professor In-Charge
Prof. Bharati Bhole

Co-ordinator of IT Department
Prof. Bharati Bhole

College Seal & Date

Examiner

INDEX

Sr No.	Title	Date	Remarks
1	Implement Matrix Multiplication, Eigen Vectors, Eigenvalue Computation Using Tensorflow.	Jul 17, 2025	
2	Implement Deep Feedforward Network For Xor Using Keras	Jul 17, 2025	
3	Implement Gradient-Descent Using Numpy	Jul 24, 2025	
4	Implement Image Recognition Using Mnist Dataset.	Jul 31, 2025	
5	Implement Backpropagation Using Pandas	Aug 14, 2025	
6	Implement Ensemble Learning Technique - Bagging, Decision Tree, Random Forest.	Sep 5, 2025	
7	Implement Regularization On Sonar Dataset Using Dropout Technique.	Sep 5, 2025	
8	Implement Image Classification On Cifar Dataset Using CNN	Sep 11, 2025	
9	Data Augmentation a. Implement on a single image b. Implement on a dataset	Sep 18, 2025 & Sep 20, 2025	
10	Implement RNN (Recurrent Neural Network)	Sep 20, 2025	

All Practicals Google Colab file:

https://colab.research.google.com/drive/1IYsmU9_XAHEyjbH4xWkrgsCb-4gbM_C#scrollTo=OC0rjIGdh98G

Practical 1: Implement Matrix Multiplication, Eigen Vectors, Eigenvalue Computation Using Tensorflow.

Jul 17, 2025

Matrix multiplication involves taking two matrices (rectangular arrays of numbers) and combining them to produce a new matrix. It is used in various applications, including transforming data, solving systems of linear equations, and in computer graphics.

Eigenvectors are special vectors that, when a matrix is applied to them, only get scaled (not rotated). They help in understanding the fundamental properties of matrices and are used in various fields like physics, statistics, and machine learning for tasks like dimensionality reduction.

Eigenvalues are the scaling factors associated with eigenvectors. They tell us how much the eigenvector is stretched or compressed. They give insights into the behavior of the matrix, such as its stability and the directions in which it stretches or compresses space.

Step 1: Import tensorflow library

```
# Import the TensorFlow library,  
import tensorflow as tf |
```

Step 2: Creating a matrix

```
# Define two constant tensors (2x3 matrices) using TensorFlow  
mat_X = tf.constant([[1, 2, 3, 4, 5, 6]], shape=(2, 3))  
mat_Y = tf.constant([[7, 8, 9, 10, 11, 12]], shape=(3, 2))  
  
# Print the matrices  
print("Matrix X \n{}".format(mat_X))  
print(f"\nMatrix Y \n{mat_Y}")
```

```
Matrix X  
[[1 2 3]  
 [4 5 6]]  
  
Matrix Y  
[[ 7 8]  
 [ 9 10]  
 [11 12]]
```

Step 3: Finding the product

```
# Perform matrix multiplication of mat_X and mat_Y using TensorFlow's matmul function
multiply = tf.matmul(mat_X, mat_Y)
# Print the result of the matrix multiplication
print("Matrix Multiplication result\n{}".format(multiply))
```

⇒ Matrix Multiplication result

```
[[ 58  64]
 [139 154]]
```

Step 4: Creating a random matrix to find eigenvalues and eigenvectors

```
# Create two random tensors (matrices) with specified shapes and data types
mat1 = tf.random.uniform(minval=3, maxval=11, shape=(2, 3), dtype=tf.float32)
mat2 = tf.random.uniform(minval=3, maxval=11, shape=(3, 2), dtype=tf.float32)

# Print the random matrices
print(f"Mat1 = \n{mat1}")
print(f"\nMat2 = \n{mat2}")
```

⇒ Mat1 =

```
[[6.39958  5.2628336 9.026324 ]
 [5.060322 7.7474756 3.4743643]]
```

Mat2 =

```
[[ 4.6396713  9.919684 ]
 [10.489911  7.933982 ]
 [10.435202  6.509447 ]]
```

Step 5: Calculate the Eigenvectors and eigenvalues.

```
# Create a random 2x2 matrix with values drawn from a normal distribution
random_matrix = tf.random.normal([2, 2], dtype=tf.float32)

# Compute the eigenvalues and eigenvectors of the symmetric matrix using TensorFlow's linalg.eigh function
# Eigenvectors are special vectors that are only scaled by a linear transformation (like matrix multiplication)
# The factor by which they are scaled is the eigenvalue.
# tf.linalg.eigh is used for symmetric matrices. For general matrices, tf.linalg.eig can be used.
eigenvalues, eigenvectors = tf.linalg.eigh(random_matrix)

# Print the original matrix, its eigenvalues, and its eigenvectors
print(f"\nMatrix:\n{random_matrix}")
print(f"\nEigenvalues:\n{eigenvalues}")
print(f"\nEigenvectors:\n{eigenvectors}")
```

⇒ Matrix:

```
[[ -2.0252132  1.2027283 ]
 [ 0.94582206 -0.6015368 ]]
```

Eigenvalues:

```
[-2.4971375 -0.12961282]
```

Eigenvectors:

```
[[ -0.8948004 -0.44646645]
 [ 0.44646645 -0.8948004 ]]
```

Practical 2: Implement Deep Feedforward Network For Xor Using Keras

Jul 17, 2025

FeedForward Network: This is a type of artificial neural network where information moves in one direction—from input to output. There are no loops or cycles in the connections.

Deep: A network is considered deep if it has more than one hidden layer between the input and output layers. This allows it to learn more complex patterns.

A **Deep FeedForward Network** for **XOR** is essentially a tool to learn complex patterns that a simple model can't capture. By using multiple layers, it can effectively solve the XOR problem by learning to transform inputs into a form where the outputs (0 or 1) can be predicted accurately.

Step 1: Import libraries

```
#importing numpy
import numpy as np

#importing dense layer and sequential model type from keras
from keras.layers import Dense
from keras.models import Sequential

model=Sequential() # Initialize a Sequential model

# Adding first Dense layer with 2 units, ReLU activation, and an input dimension of 2
model.add(Dense(units=2,activation='relu',input_dim=2))

# Adding second Dense layer with 1 unit and sigmoid activation
model.add(Dense(units=1,activation='sigmoid'))

# Compile the model using binary crossentropy loss function, Adam optimizer, and track accuracy
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Print the summary of the model architecture
print(model.summary())
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 2)	6
dense_3 (Dense)	(None, 1)	3

Total params: 9 (36.00 B)
 Trainable params: 9 (36.00 B)
 Non-trainable params: 0 (0.00 B)
 None

Step 2: Check the weights

```
# Print the initial weights of the model before training
# model.get_weights() returns a list of numpy arrays representing the weights and biases of each layer in the model
print(model.get_weights())
```

```
[array([[ -0.83377707,  0.862183  ],
        [ -0.3071341 , -0.95004785]], dtype=float32), array([0., 0.], dtype=float32), array([[ 0.763214  ],
        [-1.3536491]], dtype=float32), array([0.], dtype=float32)]
```

Step 3: Defining input data and training the model

```
# Define the input data 'X' and corresponding labels 'Y' for the XOR problem
# X: A numpy array containing the input pairs for the XOR function.
# Y: A numpy array containing the expected output (labels) for each input pair.
X = np.array([[0., 0.], [0, 1], [1., 0.], [1., 1.]])
Y = np.array([0., 1., 1., 0.])
# Train the model on the data for 1000 epochs with a batch size of 4
# model.fit(): Trains the model for a fixed number of epochs (iterations on the dataset).
# epochs: number of times to iterate over the entire training dataset. More epochs can lead to better learning but also overfitting.
# batch_size: number of samples per gradient update. A batch size of 4 means the model updates its weights after processing every 4 samples.
model.fit(X, Y, epochs=500, batch_size = 4)
```

```
Epoch 490/500
1/1 ----- 0s 63ms/step - accuracy: 1.0000 - loss: 0.4393
Epoch 491/500
1/1 ----- 0s 67ms/step - accuracy: 1.0000 - loss: 0.4392
Epoch 492/500
1/1 ----- 0s 136ms/step - accuracy: 1.0000 - loss: 0.4390
Epoch 493/500
1/1 ----- 0s 61ms/step - accuracy: 1.0000 - loss: 0.4387
Epoch 494/500
1/1 ----- 0s 54ms/step - accuracy: 1.0000 - loss: 0.4385
Epoch 495/500
1/1 ----- 0s 59ms/step - accuracy: 1.0000 - loss: 0.4382
Epoch 496/500
1/1 ----- 0s 62ms/step - accuracy: 1.0000 - loss: 0.4379
Epoch 497/500
1/1 ----- 0s 74ms/step - accuracy: 1.0000 - loss: 0.4378
Epoch 498/500
1/1 ----- 0s 55ms/step - accuracy: 1.0000 - loss: 0.4376
Epoch 499/500
1/1 ----- 0s 58ms/step - accuracy: 1.0000 - loss: 0.4373
Epoch 500/500
1/1 ----- 0s 51ms/step - accuracy: 1.0000 - loss: 0.4371
<keras.src.callbacks.history.History at 0x7b7e3e765cd0>
```

Step 4: Print

```
# Make predictions on the input data 'X' using the trained model
# model.predict(X) generates the output predictions for the given input data X.
print(model.predict(X, batch_size=0)) # batch_size=0 means the entire dataset is processed as one batch.
```

```
1/1 ----- 0s 38ms/step
[[0.40814525]
 [0.74890375]
 [0.53338516]
 [0.26312843]]
```

```
# Print the weights of the model after training
# Comparing the weights before and after training shows how the model has learned to map inputs to outputs.
print(model.get_weights())
```

```
[array([[ -0.3164954 , -1.1974336 ],
        [ 0.72987926,  1.1991282 ]], dtype=float32), array([ 0.3174494 , -0.00080344], dtype=float32), array([[ -1.5967938 ],
        [ 2.1946297]], dtype=float32), array([0.13526312], dtype=float32)]
```


Practical 3: Implement Gradient-Descent Using Numpy

Jul 24, 2025

Gradient Descent is an optimization technique used to find the best solution (like minimizing errors or costs) in machine learning and other mathematical problems. It is a method to find the best solution by repeatedly moving in the direction that reduces the function's value. It's like finding the lowest point on a hill by taking steps downhill, adjusting your path as you go. This technique is fundamental in training machine learning models, helping them to minimize errors and improve performance.

Step 1: Import numpy

```
[36] # Import the NumPy library
import numpy as np
```

Step2: write the below code of gradient-descent

```
def gradient_descent(x,y):
    # Initialize current m and b to 0
    m_curr=b_curr=0
    # Set the number of iterations for gradient descent
    iterations=100
    # Get the number of data points
    n=len(x) # n is the number of samples in the input data.
    # Set the learning rate
    learning_rate=0.001

    # Iterate through the specified number of iterations
    for i in range(iterations):
        # Predict y values using the current m and b
        y_predicted=m_curr*x+b_curr # This is the linear equation used to predict the target var.
        # Calculate the cost (mean squared error)
        cost=(1/n)*sum([val**2 for val in (y-y_predicted)])

        # Calculate the partial derivative of the cost function with respect to m
        md = -(2/n)*sum(x*(y-y_predicted))
        # Calculate the partial derivative of the cost function with respect to b
        bd = -(2/n)*sum(y-y_predicted)

        # Update m and b using the learning rate and partial derivatives
        m_curr = m_curr - learning_rate * md
        b_curr = b_curr - learning_rate * bd

        # Print the current m, b, cost, and iteration number
        print(f"m {m_curr}, b {b_curr}, cost {cost} iteration {i}")
```

Step 3: Display the output.

```
# Define the input features (x) and target variable (y) as numpy arrays for the gradient descent function
x = np.array([1, 2, 3, 4, 5])
y = np.array([5, 7, 9, 11, 13])

# Call the gradient_descent function with the defined x and y
gradient_descent(x,y) # This starts the gradient descent process to find the best-fit line for the given data
```

m 0.062, b 0.018000000000000002, cost 89.0 iteration 0
m 0.122528, b 0.035592000000000006, cost 84.881304 iteration 1
m 0.181618832, b 0.052785648000000004, cost 80.955185108544 iteration 2
m 0.239306503808, b 0.069590363712, cost 77.21263768455901 iteration 3
m 0.29562421854195203, b 0.086015343961728, cost 73.64507722605434 iteration 4
m 0.35060439367025875, b 0.10206956796255283, cost 70.2443206760065 iteration 5
m 0.40427867960173774, b 0.11776180246460617, cost 67.00256764921804 iteration 6
m 0.4566779778357119, b 0.13310060678206653, cost 63.912382537082294 iteration 7
m 0.5078324586826338, b 0.14809433770148814, cost 60.966677449199324 iteration 8
m 0.5577715785654069, b 0.16275115427398937, cost 58.15869595270883 iteration 9
m 0.606524096911324, b 0.17707902249404894, cost 55.481997572035766 iteration 10
m 0.6541180926443106, b 0.1910857198675929, cost 52.9304430134884 iteration 11
m 0.7005809802869303, b 0.20477883987199186, cost 50.49818008081245 iteration 12
m 0.7459395256813859, b 0.2181657963105263, cost 48.1796302493888 iteration 13
m 0.7902198613385323, b 0.23125382756381693, cost 45.96947586827455 iteration 14
m 0.8334475014237017, b 0.2440500007406581, cost 43.862647960726896 iteration 15

Practical 4: Implement Image Recognition Using Mnist Dataset.

Jul 31, 2025

Image recognition in Deep Learning involves using advanced algorithms to identify objects or features in images. It uses neural networks to analyze and classify images. It works by extracting features from images through multiple layers, learning from labeled examples, and then applying this knowledge to identify objects or patterns in new images. This technology is used in various applications, such as facial recognition, autonomous vehicles, and medical imaging.

Step 1: Import the required libraries

```
# Import necessary libraries
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
# Enable inline plotting for matplotlib
%matplotlib inline
```

Step 2: Import mnist dataset.

```
[4] # import dataset and split into train and test data
# Load the MNIST dataset of handwritten digits
mnist = tf.keras.datasets.mnist

# Split data into training and testing sets
(X_train, y_train), (x_test, y_test) = mnist.load_data()
```

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>
11490434/11490434 ————— 0s 0us/step

Step 3: Perform following analysis.

```
[5] # Print the number of samples in the training and testing datasets
len(X_train), len(y_train), len(x_test), len(y_test)
```

(60000, 60000, 10000, 10000)

```
[6] # Print the shapes of the training and testing datasets
X_train.shape, y_train.shape, x_test.shape, y_test.shape
```

((60000, 28, 28), (60000,), (10000, 28, 28), (10000,))

Step 4: Click on “Show Data”

```
[7] # Display the first training image as a numpy array
X_train[0]
```

```
ndarray (28, 28) show data
```



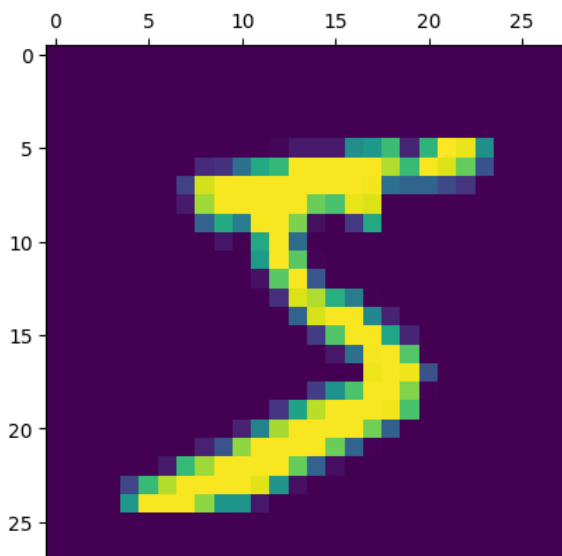
```
# Display the first training image as a numpy array
X_train[0]
```

```
[ 0,  0,  0,  0,  0,  0,  0,  0, 80, 156, 107, 253, 253,
 205, 11,  0, 43, 154,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0],
 [ 0,  0,  0,  0,  0,  0,  0,  0,  0, 14,  1, 154, 253,
 90,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0],
 [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 139, 253,
 190, 2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0],
 [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 11, 190,
 253, 70,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0],
 [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 35,
 241, 225, 160, 108,  1,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0],
 [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 81, 240, 253, 253, 119, 25,  0,  0,  0,  0,  0,  0,  0,
  0,  0],
```

Image

```
# Display the first and the 10000th training images using matplotlib
plt.matshow(X_train[0]), plt.matshow(X_train[9999])
```

```
(<matplotlib.image.AxesImage at 0x7ad1d82d67d0>,
 <matplotlib.image.AxesImage at 0x7ad1cef06350>)
```



Step 5: Normalize

```
# Normalize the pixel values of the training and test images to be between 0 and 1
X_train = X_train / 255
x_test = x_test / 255
```

```
# Display the first training image after normalization
```

```
X_train[0]
```

[illegible]

Step 6: Check the summary

(“Flatten” is used to convert 2D into 1D array)

```
# Create a Sequential model with three layers: Flatten, Dense, and Dense
model = keras.Sequential ([
    # Flatten the 28x28 pixel input images into a 1D vector
    keras.layers.Flatten (input_shape=(28, 28)),
    # Add a fully connected Dense layer with 128 neurons and ReLU activation
    keras.layers.Dense (128, activation='relu'),
    # Add a Dense layer with 10 neurons and softmax activation for output
    keras.layers.Dense (10, activation='softmax')
])
```

```
# Display a summary of the model's architecture
```

```
model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning:
super(). init (**kwargs)
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 10)	1,290

Total params: 101,770 (397.54 KB)

Trainable params: 101,770 (397.54 KB)

Non-trainable params: 0 (0.00 B)

```
# Compile the model with Stochastic Gradient Descent optimizer, sparse categorical crossentropy loss, and accuracy metric
model.compile(
    optimizer='sgd',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Step 7: History

```
# Train the model on the training data for 10 epochs, using the test data for validation
history = model.fit(X_train, y_train, epochs=10, validation_data=(x_test, y_test))
```

```
Epoch 1/10
1875/1875 ————— 6s 3ms/step - accuracy: 0.7264 - loss: 1.0413 - val_accuracy: 0.9011 - val_loss: 0.3624
Epoch 2/10
1875/1875 ————— 5s 3ms/step - accuracy: 0.9024 - loss: 0.3540 - val_accuracy: 0.9186 - val_loss: 0.2955
Epoch 3/10
1875/1875 ————— 10s 3ms/step - accuracy: 0.9155 - loss: 0.2979 - val_accuracy: 0.9266 - val_loss: 0.2628
Epoch 4/10
1875/1875 ————— 6s 3ms/step - accuracy: 0.9272 - loss: 0.2618 - val_accuracy: 0.9322 - val_loss: 0.2390
Epoch 5/10
1875/1875 ————— 11s 4ms/step - accuracy: 0.9332 - loss: 0.2412 - val_accuracy: 0.9366 - val_loss: 0.2214
Epoch 6/10
1875/1875 ————— 8s 3ms/step - accuracy: 0.9378 - loss: 0.2212 - val_accuracy: 0.9390 - val_loss: 0.2083
Epoch 7/10
1875/1875 ————— 6s 3ms/step - accuracy: 0.9425 - loss: 0.2066 - val_accuracy: 0.9437 - val_loss: 0.1940
Epoch 8/10
1875/1875 ————— 5s 3ms/step - accuracy: 0.9462 - loss: 0.1917 - val_accuracy: 0.9464 - val_loss: 0.1827
Epoch 9/10
1875/1875 ————— 10s 3ms/step - accuracy: 0.9501 - loss: 0.1798 - val_accuracy: 0.9484 - val_loss: 0.1743
Epoch 10/10
1875/1875 ————— 7s 3ms/step - accuracy: 0.9532 - loss: 0.1681 - val_accuracy: 0.9504 - val_loss: 0.1658
```

Step 8: Evaluate the model.

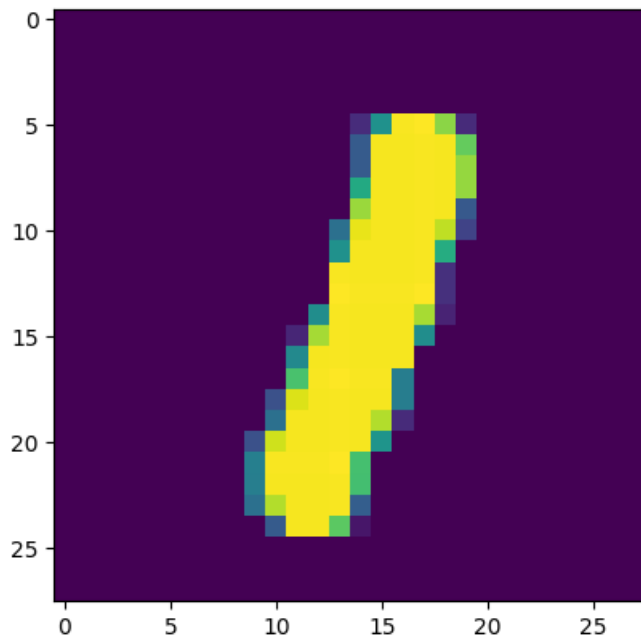
```
# evaluating the model
# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(x_test, y_test)
# Print the loss on the test data
print("Loss=%.3f" % test_loss)
# Print the accuracy on the test data
print("Accuracy=%.3f" % test_acc)
```

```
313/313 ————— 1s 2ms/step - accuracy: 0.9419 - loss: 0.1932
Loss=0.166
Accuracy=0.950
```

Step 9: Make a new prediction for new data

```
# Generate a random integer between 0 and 9999 (inclusive)
n= random.randint(0,9999)
# Display the test image at the randomly selected index
plt.imshow(x_test[n])
```

<matplotlib.image.AxesImage at 0x7ad1ac2a6090>



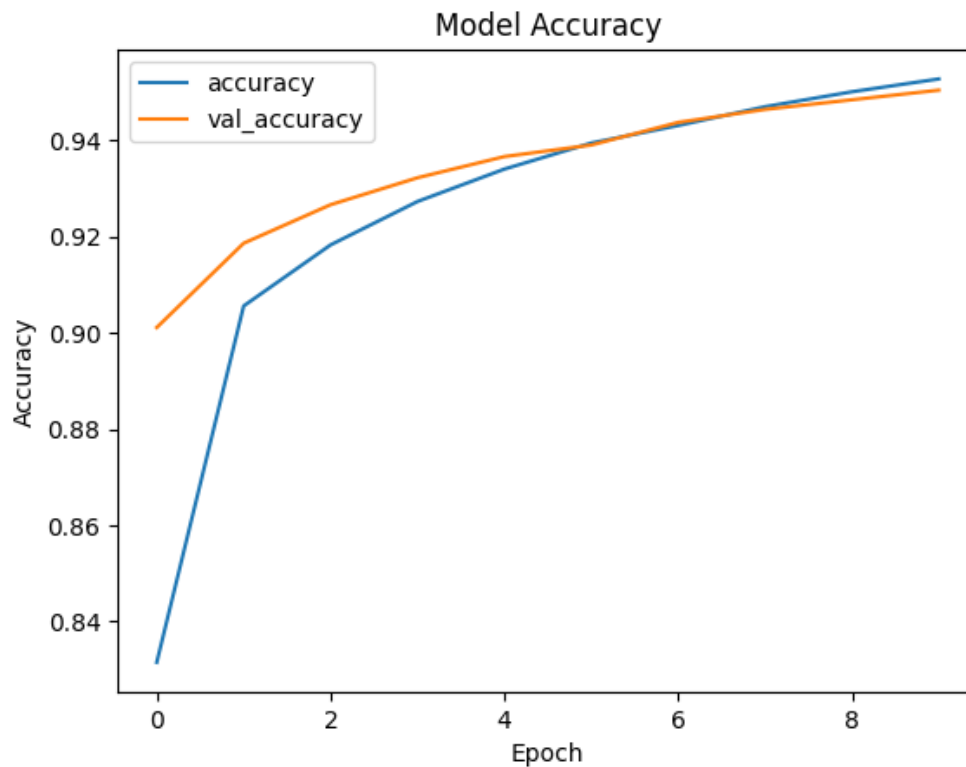
Step 10: To find the handwritten data

```
# Make predictions on the test data
predicted_value = model.predict(x_test)
# Print the predicted digit for the randomly selected image
print("Handwritten number in the image is = %d" % np.argmax(predicted_value[n]))
```

313/313 ————— 1s 1ms/step
Handwritten number in the image is = 1

Step 11: Plot the graph for Accuracy and Loss

```
# Plot the training accuracy over epochs
plt.plot(history.history['accuracy'])
# Plot the validation accuracy over epochs
plt.plot(history.history['val_accuracy'])
# Set the title of the plot
plt.title('Model Accuracy')
#Label the y-axis
plt.ylabel('Accuracy')
#Label the x-axis
plt.xlabel('Epoch')
# Add a legend to distinguish between training and validation accuracy
plt.legend(['accuracy', 'val_accuracy'])
# Display the plot
plt.show()
```




```
[17] # Display the array of predicted probabilities for the randomly selected image  
predicted_value[n]
```

```
→ array([3.78493096e-05, 8.89086187e-01, 3.88961961e-03, 2.45250314e-02,  
        7.99321351e-05, 4.32658489e-05, 5.07519726e-05, 1.55665595e-02,  
        6.54206574e-02, 1.30015169e-03], dtype=float32)
```

```
[18] # Print the maximum predicted probability for the randomly selected image  
np.max(predicted_value[n])
```

```
→ np.float32(0.8890862)
```

```
[19] # Print the shape of the predicted values array  
predicted_value.shape
```

```
→ (10000, 10)
```

Practical 5: Implement Backpropagation Using Pandas

Aug 14, 2025

Backpropagation is a key concept in training neural networks. The goal is to minimize the difference between the model's predictions and the actual outcomes. This difference is called the "error" or "loss." It is a method for training neural networks by adjusting the weights to minimize errors through a process of forward and backward passes. This helps the network learn from its mistakes and improve over time.

Step 1: Import the Libraries

```
[33] import numpy as np # Import NumPy for numerical operations.
      import matplotlib.pyplot as plt # Import Matplotlib for plotting.
```

Step 2: Sigmoid Function

```
▶ # Define the sigmoid activation function and its derivative.
def nlinear(x, deriv=False):
    # If deriv is True, return the derivative of the sigmoid function.
    if(deriv==True):
        return x*(1-x) # Derivative of sigmoid is x * (1-x) where x is the output of the sigmoid.
    # If deriv is False, return the sigmoid function output.
    return 1/(1+np.exp(-x))
```

Step 3: Creating datasets

```
▶ # Define the input data 'x' for the neural network.
# Each row is a training example, and each column is an input feature.
x = np.array([[0,0,1],
              [0,1,1],
              [1,0,1],
              [1,1,1]])

# Define the output data 'y' (the ground truth labels).
# Each row corresponds to the output for the respective input row in 'x'.
y = np.array([[0],
              [0],
              [1],
              [1]])

# The output is a column vector.
# or
# y = np.array([[0, 0, 1, 1]]).T

# Print the input and output data to verify.
x, y
```

```

→ array([[0, 0, 1],
        [0, 1, 1],
        [1, 0, 1],
        [1, 1, 1]]),
       array([[0],
              [0],
              [1],
              [1]]))

```

```

▶ # Seed the random number generator for reproducibility.
# This ensures that the initial weights are the same every time the code is run.
np.random.seed(1)

# Initialize the weights for the synapse (connection between input and output layers).
# The weights are initialized randomly with values between -1 and 1.
# The shape (3, 1) indicates 3 input features and 1 output neuron.
synapse0 = 2*np.random.random((3,1)) - 1
# Print the initial weights.
synapse0

```

```

→ array([[ -0.16595599],
        [ 0.44064899],
        [-0.99977125]])

```

```

▶ # Start the training process using gradient descent and backpropagation.
# Iterate over a specified number of epochs (training iterations).
for i in range(1000):
    # Forward propagation: Calculate the output of the neural network.
    # layer0 is the input layer.
    layer0 = x
    # layer1 is the output layer, calculated by taking the dot product of the input and weights,
    # and then applying the nonlinear activation function (sigmoid).
    layer1 = nlinear(np.dot(layer0, synapse0))

    # Calculate the error: The difference between the actual output (y) and the predicted output (layer1).
    layer1_error = y - layer1

    # Backpropagation: Calculate the delta (change) for updating the weights.
    # The delta is calculated by multiplying the error with the derivative of the activation function
    # at the current output values. This determines how much the weights need to change to reduce the error.
    layer1_delta = layer1_error * nlinear(layer1, deriv=True)

    # Update the weights: Adjust the weights based on the calculated delta.
    # The dot product of the transpose of layer0 and layer1_delta gives the weight updates.
    # The weights are updated by adding this change.
    synapse0 += np.dot(layer0.T, layer1_delta)

```

Step 4: Print the Output

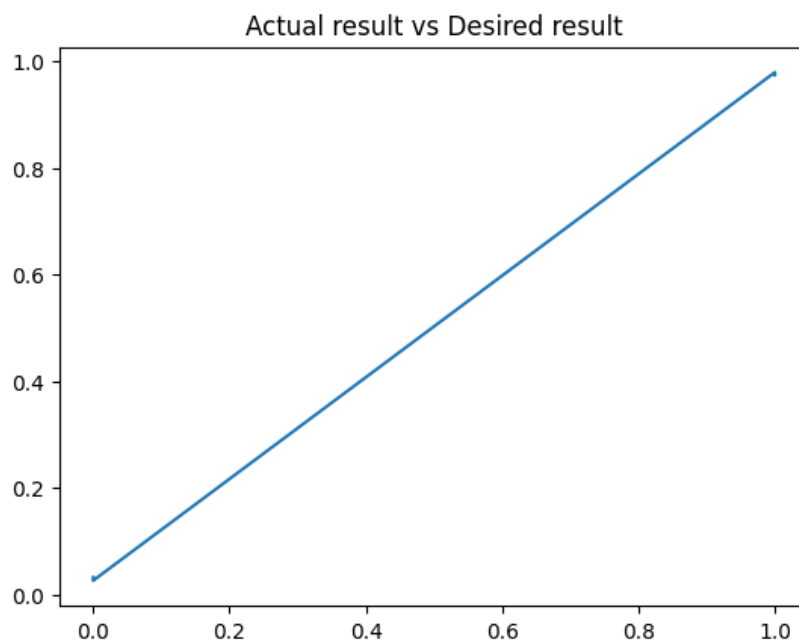
```
# Print the output of the neural network after training.  
print("Output after training\n", layer1)  
# Print the actual desired output.  
print("\nActual Output\n", y)
```

```
⇒ Output after training  
[[0.03178421]  
 [0.02576499]  
 [0.97906682]  
 [0.97414645]]
```

```
Actual Output  
[[0]  
 [0]  
 [1]  
 [1]]
```

Step 5: Plot the Graph

```
# Plot the actual output (y) against the predicted output (layer1).  
# A perfect model would show a straight line with a slope of 1.  
plt.title("Actual result vs Desired result")  
plt.plot(y, layer1)  
# Display the plot.  
plt.show()
```



Practical 6: Implement Ensemble Learning Technique - Bagging, Decision Tree, Random Forest.

Sep 5, 2025

Dataset link → <https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database?resource=download>

A **decision tree** is a flowchart-like structure used to make decisions or predictions. It splits the data into branches based on certain criteria until it reaches a decision or prediction. It is a single model that makes predictions based on a series of decision rules.

Bagging is an ensemble method that improves the accuracy and robustness of machine learning models by combining predictions from multiple models. It combines multiple models (e.g., decision trees) trained on different subsets of data to improve performance and stability.

Random forest is an extension of bagging specifically for decision trees. It builds multiple decision trees and combines their predictions to improve accuracy and control overfitting. It is a type of bagging that specifically uses multiple decision trees, each trained on random subsets of data and features, and combines their predictions for better accuracy.

Step 1: Import the pandas library to load and display the dataset

```
import pandas as pd # For data manipulation
from sklearn.preprocessing import StandardScaler # For feature scaling
from sklearn.model_selection import train_test_split, cross_val_score # For splitting data and model evaluation
from sklearn.tree import DecisionTreeClassifier # For building the Decision Tree model
from sklearn.ensemble import BaggingClassifier # For ensemble learning (bagging)
```

```
df = pd.read_csv("/content/sample_data/diabetes.csv") # Replace with your dataset's path
display(df.head()) # Display the first few rows of the dataset
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Step 2: Describe Dataset

```
# Get descriptive statistics
df.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Step 3: Write the below code

```
# Count the number of instances for each outcome
df.Outcome.value_counts() # Check the distribution of the target variable (Outcome)
```

	count
Outcome	
0	500
1	268

Step 4: Splitting the dataset into features and target

```
# define input feature(X) and target(y) / Outcome
X = df.drop("Outcome", axis="columns") # drop the outcome columns
y = df.Outcome # setting Outcome as target var
```

Step 5: Standardizing the features

```
scaler = StandardScaler() # Initialize the StandardScaler
X_scaled = scaler.fit_transform(X) # Scale the input features
X_scaled[:3] # Display the first three rows of the scaled features
```

```
array([[ 0.63994726,  0.84832379,  0.14964075,  0.90726993, -0.69289057,
         0.20401277,  0.46849198,  1.4259954 ],
       [-0.84488505, -1.12339636, -0.16054575,  0.53090156, -0.69289057,
        -0.68442195, -0.36506078, -0.19067191],
       [ 1.23388019,  1.94372388, -0.26394125, -1.28821221, -0.69289057,
        -1.10325546,  0.60439732, -0.10558415]])
```

Step 6: Perform training and testing dataset

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled,
    y,
    stratify=y,
    random_state=10) # Use stratified sampling
X_train.shape, X_test.shape # Check the shape
```

```
((576, 8), (192, 8))
```

Step 7: Write the below code:

```
# Check the distribution of the target variable in the training set
y_train.value_counts()
```

```
count
Outcome
0      375
1      201
```

```
# Class distribution ratio in training set
201/375 #Example ratio for training set
```

```
0.536
```

```
# Check the distribution of the target variable in the testing set
y_test.value_counts()
```

```
count
Outcome
0      125
1       67
```

```
# Class distribution ratio in testing set
67/125 # Example ratio for testing set
```

```
0.536
```

Step 8: Train using stand-alone model

A "**standalone model**" refers to a machine learning model that is used on its own, without any additional models or techniques combined with it

```
# Perform cross-validation using a Decision Tree classifier
scores = cross_val_score(DecisionTreeClassifier(), X, y, cv=5) # 5-fold cross-validation
scores # Display individual cross-validation
```

```
array([0.707779221, 0.68831169, 0.69480519, 0.77777778, 0.70588235])
```

Step 9: Find out the mean of scores

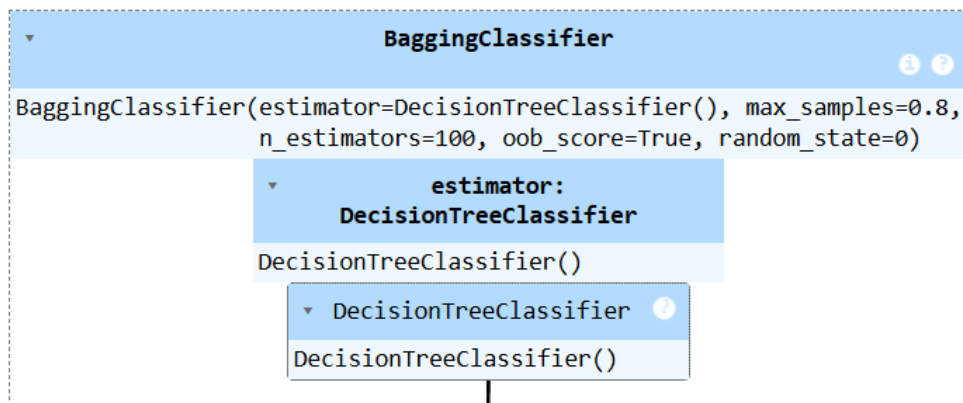
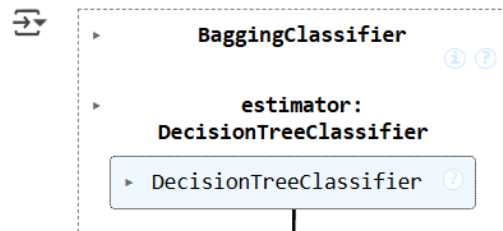
```
# Calculate the mean of the cross-validation scores
scores.mean()
```

```
np.float64(0.714913844325609)
```

Step 10: Train using bagging

```
# Create a Bagging Classifier using a Decision Tree as the base estimator
bag_model = BaggingClassifier(
    estimator = DecisionTreeClassifier(), #Base model is a Decision Tree
    n_estimators=100, # Number of base estimators (trees)
    max_samples=0.8, # Max samples for training each tree
    oob_score=True, # Out-of-bag score for evaluation
    random_state=0
)
```

```
# Train the Bagging Classifier on the training data
bag_model.fit(X_train, y_train)
```



```
# Out-of-bag score
bag_model.oob_score_ # Display the OOB score
```

```
0.7534722222222222
```

```
# reinitialize and retainf the bagginh model using cross validation
bag_model = BaggingClassifier(
    estimator = DecisionTreeClassifier(), #Base model is a Decision Tree
    n_estimators=100, # Number of base estimators (trees)
    max_samples=0.8, # Max samples for training each tree
    oob_score=True, # Out-of-bag score for evaluation
    random_state=0
)

scores = cross_val_score(bag_model, X, y, cv=5) # 5-fold cross-validation
scores # Display individual cross-validation
```

```
array([0.75324675, 0.72727273, 0.74675325, 0.82352941, 0.73856209])
```



```
▶ scores.mean() # Calculate the mean of the cross-validation scores  
↔ np.float64(0.7578728461081402)
```

Step 11: Train using random forest model.

A **Random Forest** is an ensemble learning technique used for classification and regression tasks that operates by constructing a multitude of decision trees during training. It combines the predictions of these trees to produce a more accurate and robust output. The final prediction is made by aggregating the results from all the individual trees, typically using majority voting for classification or averaging for regression

```
# Train using random forest model  
from sklearn.ensemble import RandomForestClassifier  
  
# perform 5 fold cross validation with RandomForestClassifier (50 trees)  
scores = cross_val_score(RandomForestClassifier(n_estimators=50), X, y, cv=5) # 5-fold cross-validation  
scores # Display individual cross-validation  
  
↔ array([0.74025974, 0.72727273, 0.76623377, 0.83006536, 0.77124183])  
  
scores.mean()  
  
↔ np.float64(0.7670146846617436)
```

Practical 7: Implement Regularization On Sonar Dataset Using Dropout Technique.

Sep 5, 2025

Data set link →

<https://www.kaggle.com/datasets/rupakroy/sonarcsv>

Regularization is used to prevent overfitting, where a model becomes too complex and starts to capture noise in the training data instead of the underlying pattern. Dropout is one of the most popular regularization methods, particularly in the context of training deep neural networks.

Dropout is a regularization technique that involves randomly "dropping out" (i.e., setting to zero) a proportion of the neurons in a neural network during training. This prevents the network from becoming too dependent on any particular neuron, thereby encouraging it to learn more robust and generalizable features.

Note: M and R means Mines and Rocks

Step 1: Import the necessary libraries and load the dataset

```
# Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
```

```
# Load dataset
df = pd.read_csv("/content/sample_data/sonar.csv", header=None)
df.sample(5) # show random sample of 5 rows
```

	0	1	2	3	4	5	6	7	8	9	...	51	52	53	54	55	56	57
118	0.0363	0.0478	0.0298	0.0210	0.1409	0.1916	0.1349	0.1613	0.1703	0.1444	...	0.0115	0.0190	0.0055	0.0096	0.0050	0.0066	0.0114
106	0.0331	0.0423	0.0474	0.0818	0.0835	0.0756	0.0374	0.0961	0.0548	0.0193	...	0.0078	0.0174	0.0176	0.0038	0.0129	0.0066	0.0044
62	0.0086	0.0215	0.0242	0.0445	0.0667	0.0771	0.0499	0.0906	0.1229	0.1185	...	0.0072	0.0054	0.0022	0.0016	0.0029	0.0058	0.0050
46	0.0308	0.0339	0.0202	0.0889	0.1570	0.1750	0.0920	0.1353	0.1593	0.2795	...	0.0167	0.0127	0.0138	0.0090	0.0051	0.0029	0.0122
155	0.0211	0.0128	0.0015	0.0450	0.0711	0.1563	0.1518	0.1206	0.1666	0.1345	...	0.0117	0.0023	0.0047	0.0049	0.0031	0.0024	0.0039

5 rows × 61 columns

Step 2: Do the analysis

```
df.shape # get the shape (rows, columns) for the dataset
```

```
(208, 61)
```

Step 3: Calculate the missing data

```
print(df.isna().sum()) # chekc for missing data
```

```
0    0
1    0
2    0
3    0
4    0
..
56   0
57   0
58   0
59   0
60   0
Length: 61, dtype: int64
```

Step 4: Print the Columns names

```
df.columns # Display columns name(in this case, just indices ther are no headers)
```

```
Index([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
       36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
       54, 55, 56, 57, 58, 59, 60],
      dtype='int64')
```

Step 5: Count Frequency of values

```
# Checking unique values in target column
df[60].value_counts()
```

```
count
60
M    111
R     97
dtype: int64
```

M and R means Mines and Rocks

Step 6: Create feature matrix and target variable

```
# Create feature matrix and target var
X = df.drop(60, axis='columns') # drop the target colkumns index 60
y = df[60] # Assign target column to y
```

Step 7: Splitting the dataset into train and test ad checking shape

```

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.25,
    random_state=1) # Split the data (25% test)
X_train.shape, X_test.shape, y_train.shape, y_test.shape

```

⇒ ((156, 60), (52, 60), (156, 1), (52, 1))

Step 8 : Import libraries □ Define, compile and train the model

```

# build DL model
import tensorflow as tf
from tensorflow import keras

# define model architecture
model = keras.Sequential([
    keras.layers.Dense(60, input_dim=60, activation='relu'), # first input layer with 60 neuron
    keras.layers.Dropout(0.5), # Dropout to prevent overfitting
    keras.layers.Dense(30, activation='relu'), # second input layer with 30 neuron
    keras.layers.Dropout(0.5), # Dropout to prevent overfitting
    keras.layers.Dense(15, activation='relu'), # third input layer with 30 neuron
    keras.layers.Dense(1, activation='sigmoid') # output layer with sigmoid activation for binary classification
])

# compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']) # use adam optimizer and binary cross entropy

# train the model
model.fit(X_train, y_train, epochs=100, batch_size=8) # train the model for 100 epochs with batch size 8

```

20/20 ————— 0s 3ms/step - accuracy: 0.8880 - loss: 0.2560
Epoch 95/100
20/20 ————— 0s 3ms/step - accuracy: 0.8977 - loss: 0.2628
Epoch 96/100
20/20 ————— 0s 3ms/step - accuracy: 0.8616 - loss: 0.3166
Epoch 97/100
20/20 ————— 0s 3ms/step - accuracy: 0.8679 - loss: 0.3000
Epoch 98/100
20/20 ————— 0s 3ms/step - accuracy: 0.9096 - loss: 0.2674
Epoch 99/100
20/20 ————— 0s 3ms/step - accuracy: 0.9071 - loss: 0.1997
Epoch 100/100
20/20 ————— 0s 3ms/step - accuracy: 0.8723 - loss: 0.2628
<keras.src.callbacks.history.History at 0x7949a42b6d20>

Step 9: Evaluate the model

```

# Evaluate the model
model.evaluate(X_test, y_test) # evaluate the model's performance on test data

```

⇒ 2/2 ————— 0s 24ms/step - accuracy: 0.7652 - loss: 0.5282
[0.4754674732685089, 0.7884615659713745]

Step 10: Predict the output and display the binary values

```
# generate predication
y_pred = model.predict(X_test).reshape(-1)
y_pred = np.round(y_pred) # round prediaction 0 to 1 binary output

y_pred
```

```
2/2 0s 57ms/step
array([0., 1., 1., 0., 1., 1., 1., 1., 0., 1., 0., 0., 0., 0., 0., 1.,
       0., 0., 1., 0., 1., 0., 1., 0., 0., 0., 1., 0., 0., 0., 1.,
       1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 1., 1., 0., 0.,
       0.], dtype=float32)
```

Step 11: Print the Classification report and confusion matrix

```
# print classifciation report
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
precision    recall  f1-score   support

 False       0.75     0.89     0.81         27
  True       0.85     0.68     0.76         25

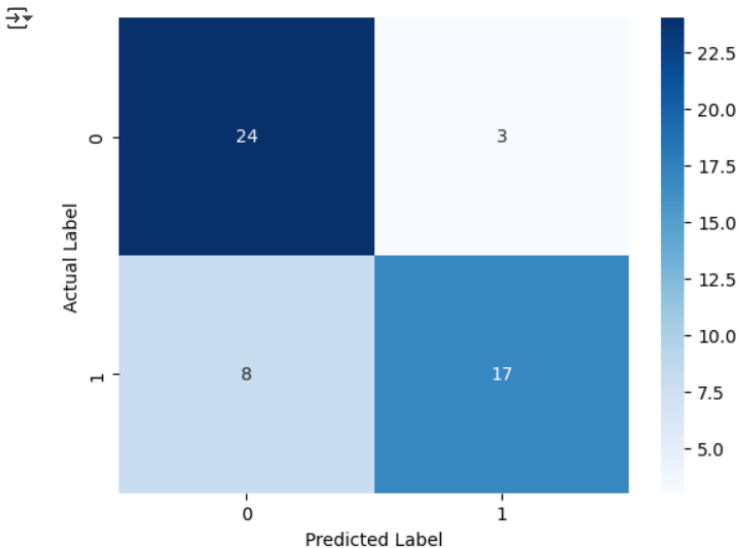
 accuracy          0.79         52
 macro avg       0.80     0.78     0.78         52
 weighted avg    0.80     0.79     0.79         52
```

```
# plot confusion matrix
from sklearn.metrics import confusion_matrix

conf_matrix = confusion_matrix(y_test, y_pred)
conf_matrix
```

```
array([[24,  3],
       [ 8, 17]])
```

```
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap="Blues")
plt.xlabel('Predicted Label')
plt.ylabel('Actual Label')
plt.show()
```



Step 12: trying without Dropout - it will overfit the model

```
# test without DropOut - it will overfit the model

# define model arcghitecture
model = keras.Sequential([
    keras.layers.Dense(60, input_dim=60, activation='relu'), # firstinput layer with 60 neuron
    keras.layers.Dense(30, activation='relu'), # second input layer with 30 neuron
    keras.layers.Dense(15, activation='relu'), # third input layer with 30 neuron
    keras.layers.Dense(1, activation='sigmoid') # output layer with sigmoid activation for binaty classification
])

# comile the mode
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']) # use adam optimizer and binarycross entropy

# train the model
model.fit(X_train, y_train, epochs=100, batch_size=8) # train the model for 100 ephocs with batch size pf 8
```

```
20/20 ————— 1s 14ms/step - accuracy: 1.0000 - loss: 0.0069
Epoch 96/100
20/20 ————— 0s 3ms/step - accuracy: 1.0000 - loss: 0.0050
Epoch 97/100
20/20 ————— 0s 13ms/step - accuracy: 1.0000 - loss: 0.0051
Epoch 98/100
20/20 ————— 1s 9ms/step - accuracy: 1.0000 - loss: 0.0047
Epoch 99/100
20/20 ————— 0s 8ms/step - accuracy: 1.0000 - loss: 0.0058
Epoch 100/100
20/20 ————— 0s 5ms/step - accuracy: 1.0000 - loss: 0.0052
<keras.src.callbacks.history.History at 0x79498c180770>
```

```
# Evaluate the model
print(model.evaluate(x_test, y_test)) # evaluate the model's performance on test data
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

2/2 ————— 1s 82ms/step - accuracy: 0.8093 - loss: 0.8398
[0.7527222037315369, 0.807692289352417]

	precision	recall	f1-score	support
False	0.75	0.89	0.81	27
True	0.85	0.68	0.76	25
accuracy			0.79	52
macro avg	0.80	0.78	0.78	52
weighted avg	0.80	0.79	0.79	52

[[24 3]
[8 17]]

Practical 8: Implement Image Classification On Cifar Dataset Using CNN

Sep 11, 2025

Data set →

<https://www.kaggle.com/c/cifar-10/>

Image classification using a Convolutional Neural Network (CNN) is a process where a computer model is trained to recognize and categorize images into different classes or labels. Convolutional Neural Network (CNN) is a type of deep learning model specifically designed for working with images. Unlike traditional neural networks, CNNs can automatically detect and learn features from images, such as edges, textures, and patterns, through their layers.

Step 1: Import libraries and load dataset into train and test

```
# Import necessary libraries for building and training a CNN model
import tensorflow as tf # Import TensorFlow, the core library
from tensorflow.keras.models import Sequential # Import Sequential model for building a linear stack of layers
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout # Import different types of layers
from tensorflow.keras.optimizers import Adam # Import the Adam optimizer for training
from tensorflow.keras.preprocessing.image import ImageDataGenerator # Import ImageDataGenerator for data augmentation
from tensorflow.keras.utils import to_categorical # Import to_categorical for one-hot encoding labels
import matplotlib.pyplot as plt # Import Matplotlib for plotting images and training history
from sklearn.metrics import classification_report, confusion_matrix # Import metrics for evaluating model performance
import seaborn as sns # Import Seaborn for enhanced visualizations, especially for the confusion matrix
import numpy as np # Import NumPy for numerical operations
```

```
# Load the CIFAR-10 dataset, a widely used dataset for image classification
# The dataset contains 60,000 32x32 color images in 10 classes, with 6,000 images per class.
# It is split into 50,000 training images and 10,000 test images.
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
```

Step 2: Display the shape

```
# Display the shape
x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

```
→ ((50000, 32, 32, 3), (50000, 10), (10000, 32, 32, 3), (10000, 10))
```

Step 3: Checking the Training name


```
# Define the names of the 10 classes in the CIFAR-10 dataset
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(12, 6)) # set figure size
# Loop through the first 10 training images and display them with their class names
for i in range(10):
    plt.subplot(2, 5, i+1) # Create a subplot for each image
    plt.imshow(X_train[i]) # Display the image
    label_index = y_train[i][0] # Get the class label index for the image
    plt.title(class_names[label_index]) # Set the title the class name
    plt.axis('off') # Turn off the axis
```



Step 4: Normalizing the training and test features

```
# Normalize the pixel values of the training and test images
# Image pixel values are typically in the range of 0 to 255.
# Dividing by 255 scales these values to the range of 0 to 1.
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

Step 5: Encoding the training and test targets

```
# Convert the integer class labels to one-hot encoded vectors
# One-hot encoding converts each integer label into a binary vector
# of length equal to the number of classes.
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

Step 6: Building the sequential model

```
model = Sequential() # Build the Convolutional Neural Network (CNN) model using Keras Sequential API

# Add the first Convolutional Layer
model.add(
    Conv2D(
        32, # Conv2D: Learns features by convolving filters over the input image.
        (3, 3), # 32: Number of filters.
        activation='relu', # (3, 3): Size of the convolutional kernel (filter).
        input_shape=(32, 32, 3) # Rectified Linear Unit activation function, introduces non-linearity.
    ) # shape of the input images (height, width, channels).
)

# Add the first MaxPooling Layer
# MaxPooling2D: Downsamples the input representation, reducing spatial dimensions and number of parameters.
model.add(MaxPooling2D((2, 2))) # Reduce the spatial dimensions

# Add the second Convolutional Layer with 64 filters
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2))) # Add the second MaxPooling Layer

# Add the third Convolutional Layer with 128 filters
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2))) # Add the third MaxPooling Layer

model.add(Flatten()) # Flatten the output of the convolutional layers

# Add a Fully Connected (Dense) Layer
model.add(Dense(128, activation='relu')) # first fully connected layer

# Add a Dropout Layer
model.add(Dropout(0.5)) # Dropout layer to prevent overfitting

# Add Dense: Output layer with 10 neurons, one for each class.
model.add(Dense(10, activation='softmax')) # Softmax activation for multi-class classification
```

Step 7: Compiling the model

```
# Compile the CNN model for training
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Step 8: Data augmentation

```
# Configure data augmentation using ImageDataGenerator

datagen = ImageDataGenerator(
    width_shift_range=0.1, # Randomly shift images horizontally by up to 10% of the width
    height_shift_range=0.1, # Randomly shift images vertically by up to 10% of the height
    horizontal_flip=True # Randomly flip images horizontally
)
datagen.fit(X_train) # Compute quantities required for featurewise normalization
```

Step 9: Fitting the model

```
history = model.fit(
    datagen.flow(X_train, y_train, batch_size=64),
    epochs=5,
    validation_data=(X_test, y_test),
    verbose=1
)
```

```
Epoch 1/5
/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDa
self._warn_if_super_not_called()
782/782 ————— 50s 48ms/step - accuracy: 0.2675 - loss: 1.9522 - val_accuracy: 0.4931 - val_loss: 1.3903
Epoch 2/5
782/782 ————— 30s 39ms/step - accuracy: 0.4630 - loss: 1.4853 - val_accuracy: 0.5564 - val_loss: 1.2146
Epoch 3/5
782/782 ————— 53s 54ms/step - accuracy: 0.5290 - loss: 1.3167 - val_accuracy: 0.6125 - val_loss: 1.1015
Epoch 4/5
782/782 ————— 69s 37ms/step - accuracy: 0.5734 - loss: 1.2137 - val_accuracy: 0.6317 - val_loss: 1.0415
Epoch 5/5
782/782 ————— 37s 47ms/step - accuracy: 0.5967 - loss: 1.1456 - val_accuracy: 0.6607 - val_loss: 0.9597
```

Step 10: Model Evaluation

```
loss, accuracy = model.evaluate(X_test, y_test)
# Print the test accuracy formatted to 4 decimal places
print(f"Test Accuracy: {accuracy:.4f}")
```

```
313/313 ————— 1s 4ms/step - accuracy: 0.6625 - loss: 0.9501
Test Accuracy: 0.6607
```

Step 11: Predication of test images and plot

```
y_pred = model.predict(X_test)
# Convert the predicted probabilities to class labels
# np.argmax() returns the index of the maximum probability for each image, which is the predicted class.
y_pred_classes = np.argmax(y_pred, axis=1) # Convert predictions to class labels
# Convert the true one-hot encoded labels to class labels
y_true = np.argmax(y_test, axis=1) # Convert true labels to class labels
```

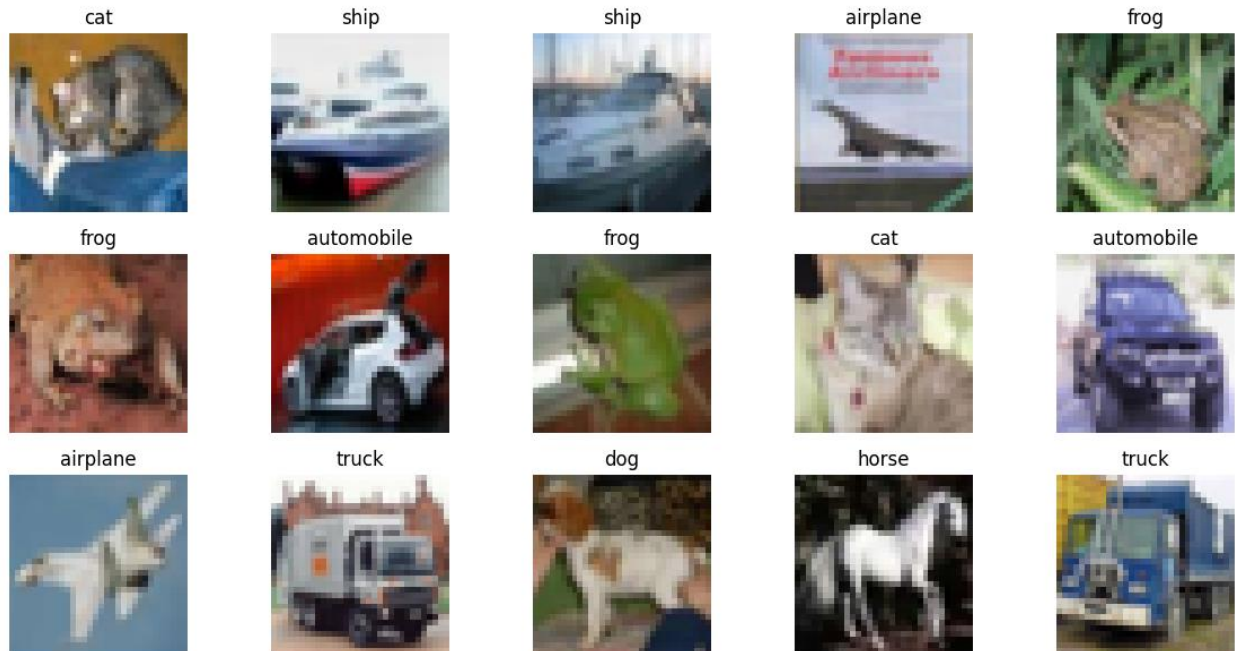
```
313/313 ————— 3s 5ms/step
```

```
# Define the names of the 10 classes in the CIFAR-10 dataset
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

# Create a figure to display the images
plt.figure(figsize=(12, 6))

# Loop through the first 15 test images
for i in range(15):
    plt.subplot(3, 5, i+1) # bottom row: test images
    plt.imshow(X_test[i]) # Display the image
    label_index = np.argmax(y_test[i]) # Get the true class label index for the image
    plt.title(class_names[label_index]) # Set the title of the subplot to the class name
    plt.axis('off') # Turn off the axis

plt.tight_layout() # Adjust the layout to prevent titles and labels from overlapping
plt.show() # Display the figure
```



Step 12: Other plot and metrics

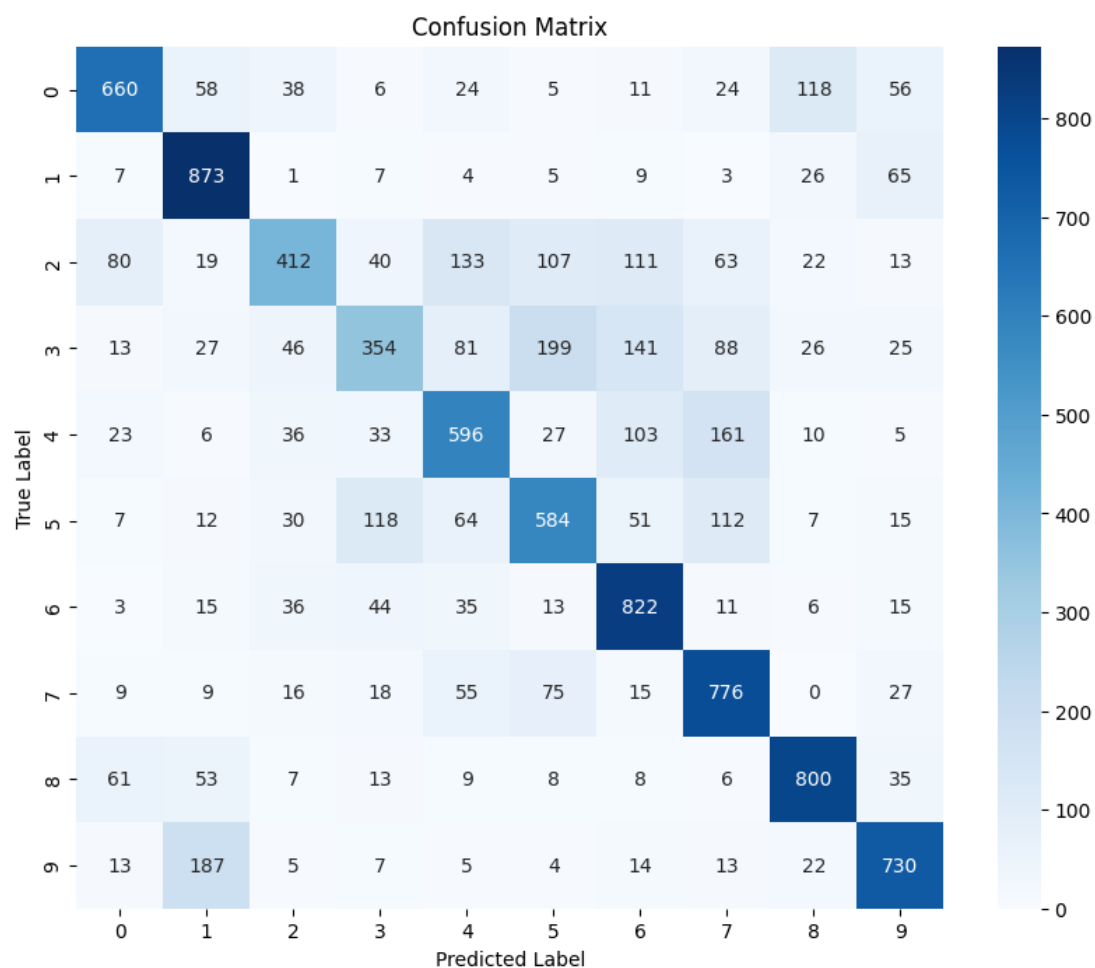
```
# Print the classification report to evaluate the model's performance in detail
# classification_report: Provides precision, recall, f1-score, and support for each class.
print("Classification Report:")
print(classification_report(y_true, y_pred_classes))

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred_classes)
# Create a figure for the confusion matrix heatmap
plt.figure(figsize=(10, 8))

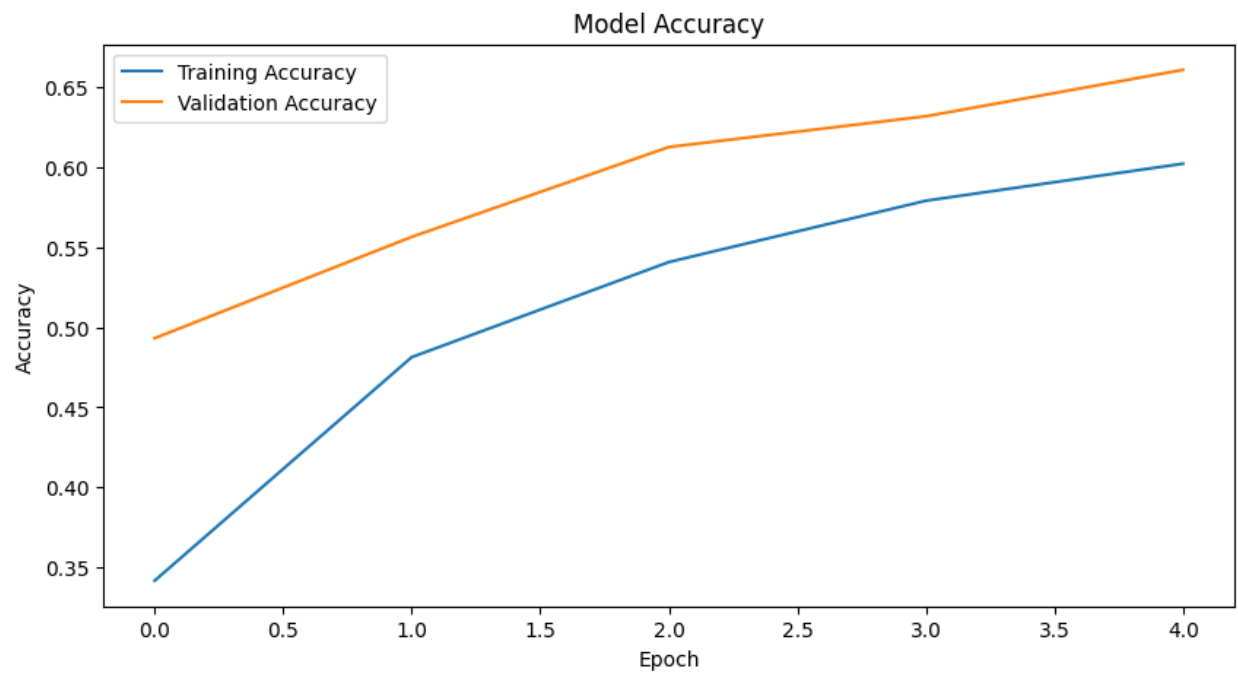
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=range(10), yticklabels=range(10))
# Set the title and labels for the heatmap
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
# Display the heatmap
plt.show()
```

Classification Report:

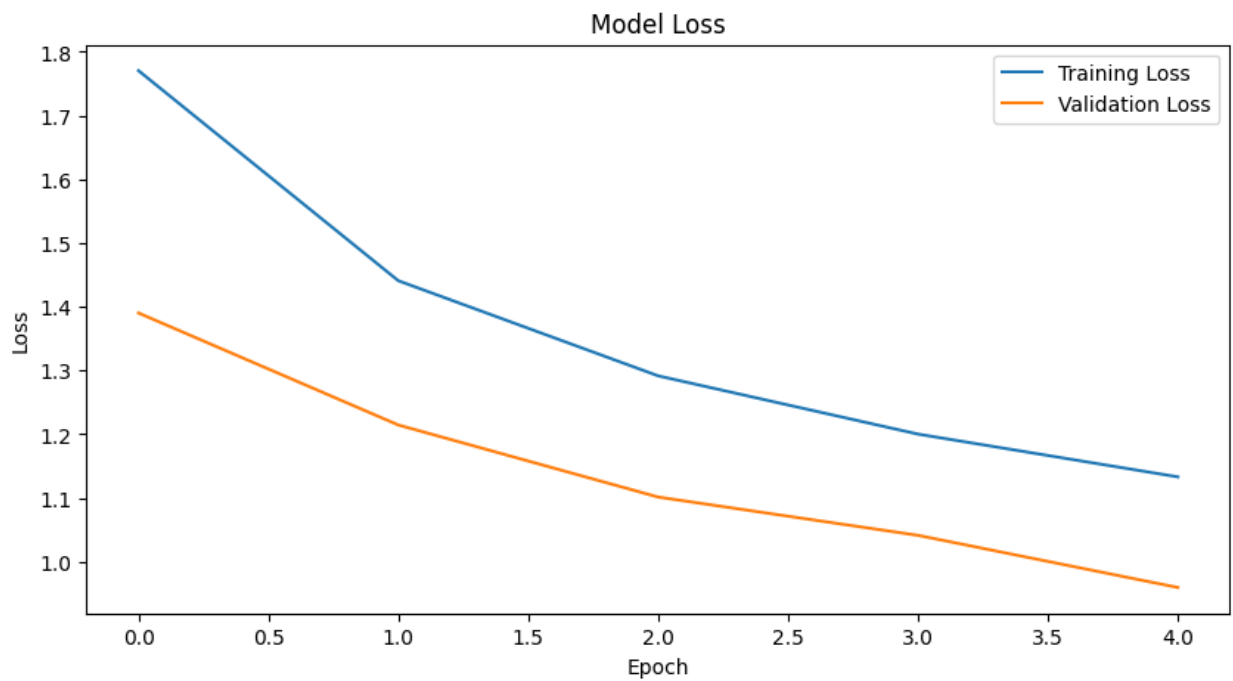
	precision	recall	f1-score	support
0	0.75	0.66	0.70	1000
1	0.69	0.87	0.77	1000
2	0.66	0.41	0.51	1000
3	0.55	0.35	0.43	1000
4	0.59	0.60	0.59	1000
5	0.57	0.58	0.58	1000
6	0.64	0.82	0.72	1000
7	0.62	0.78	0.69	1000
8	0.77	0.80	0.79	1000
9	0.74	0.73	0.74	1000
accuracy			0.66	10000
macro avg	0.66	0.66	0.65	10000
weighted avg	0.66	0.66	0.65	10000



```
# Plot the training and validation accuracy over epochs
plt.figure(figsize=(10, 5))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



```
# Plot the training and validation loss over epochs
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Practical 9: Data Augmentation

Dataset Link:

https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz

Data augmentation is a technique used to artificially expand the size of a dataset by creating modified versions of existing data. The goal is to make the model more robust and help it generalize better by learning from a variety of different but related inputs. This is especially useful when you have a small dataset but want to improve model performance.

A. Implement on a single image

Sep 18, 2025

Step 1: Import the necessary libraries

```
# importing packages
import numpy as np
import os
import PIL
import PIL.Image
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras import layers
import pathlib
import matplotlib.pyplot as plt
```

Step 2: Load the dataset & give path to access the data

```
# define dataset url
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"

# download and extract dataset
archive = tf.keras.utils.get_file(origin=dataset_url, extract=True)
# give the path to access the data
data_dir = pathlib.Path(archive).with_suffix('')
```

Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
228813984/228813984 ————— 11s 0us/step

Step 3: Count the rose images and print it

```
roses_dir = pathlib.Path('/root/.keras/datasets/flower_photos.tgz/flower_photos') / 'roses'

# List all image files in the 'roses' directory
rose_images = list(roses_dir.glob('*.jpg'))

# Count the number of images to confirm how many are there
print(f"Total number of rose images: {len(rose_images)}")
```

Total number of rose images: 641

Step 4: set all images of Roses in a list and display one


```
PIL.Image.open(str(rose_images[0]))
```



Step 5: Count the tulips images and print it

```
tulips_dir = pathlib.Path('/root/.keras/datasets/flower_photos.tgz/flower_photos') / 'tulips'

# List all image files in the 'tulips' directory
tulips_images = list(tulips_dir.glob('*.jpg'))

# Count the number of images to confirm how many are there
print(f"Total number of tulips images: {len(tulips_images)}")
```

⇒ Total number of tulips images: 799

Step 6: set all images of Tulips in a list and display one

```
PIL.Image.open(str(tulips_images[0]))
```



Step 7: Load the dataset

```

▶ # load the dataset
(train_ds, val_ds, test_ds), metadata = tfds.load(
    'tf_flowers',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
    as_supervised=True,
)

```

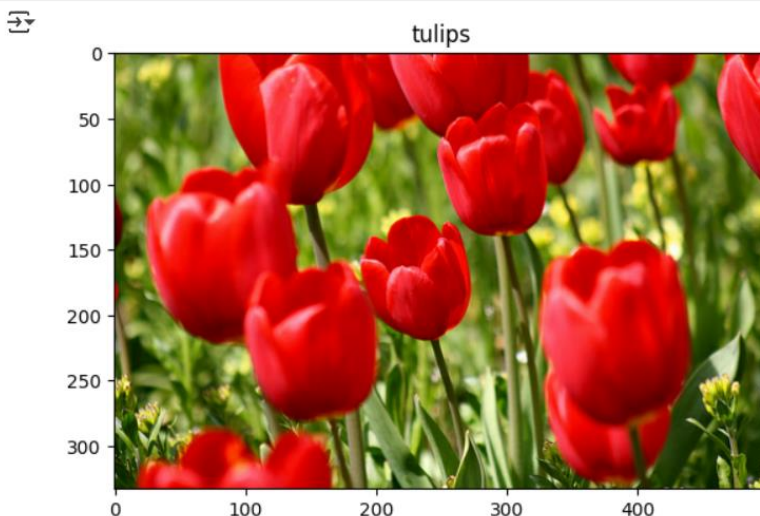
⚡ WARNING:absl:Variant folder /root/tensorflow_datasets/tf_flowers/3.0.1 has no dataset_info.json
 Downloading and preparing dataset Unknown size (download: Unknown size, generated: Unknown size,
 DI Completed...: 100% 1/1 [00:11<00:00, 11.63s/ url]
 DI Size...: 100% 218/218 [00:11<00:00, 22.39 MiB/s]
 Dataset tf_flowers downloaded and prepared to /root/tensorflow_datasets/tf_flowers/3.0.1. Subseq

Step 8: Display image

```

▶ get_label_name = metadata.features['label'].int2str # retrieve label mapping function
image, label = next(iter(train_ds)) # get an image and a label
_ = plt.imshow(image) # display the image
_ = plt.title(get_label_name(label)) # set the plot title

```



Step 9: Create sequential model for processing

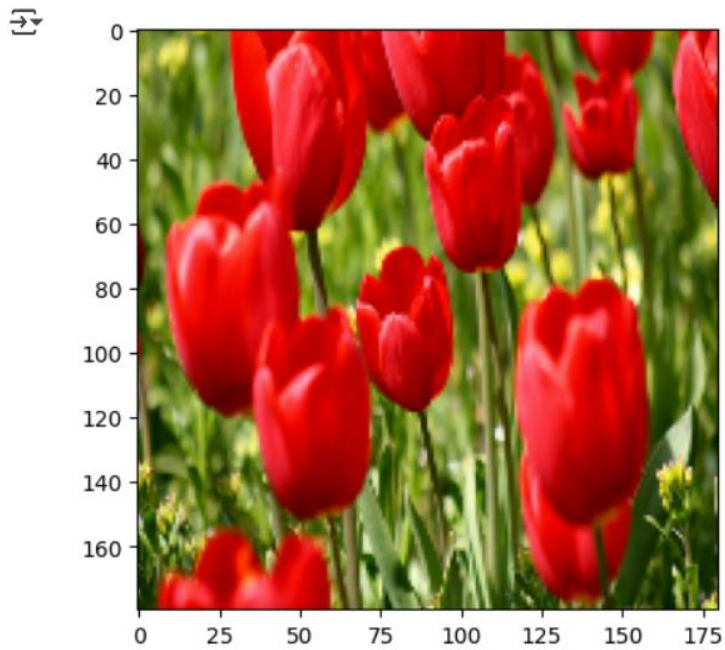
```

▶ # define image size
IMG_SIZE = 180
# create a sequential model for preprocessing
resize_and_rescale = tf.keras.Sequential([
    layers.Resizing(IMG_SIZE, IMG_SIZE),
    layers.Rescaling(1./255)
])

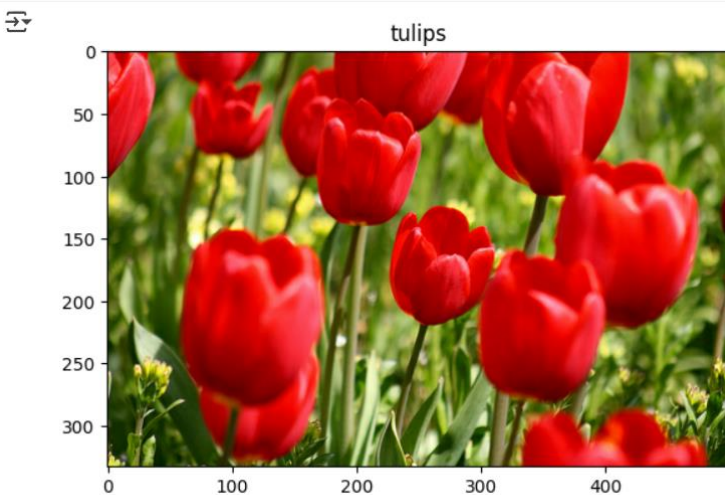
```

Step 10: Applying different image operation and display

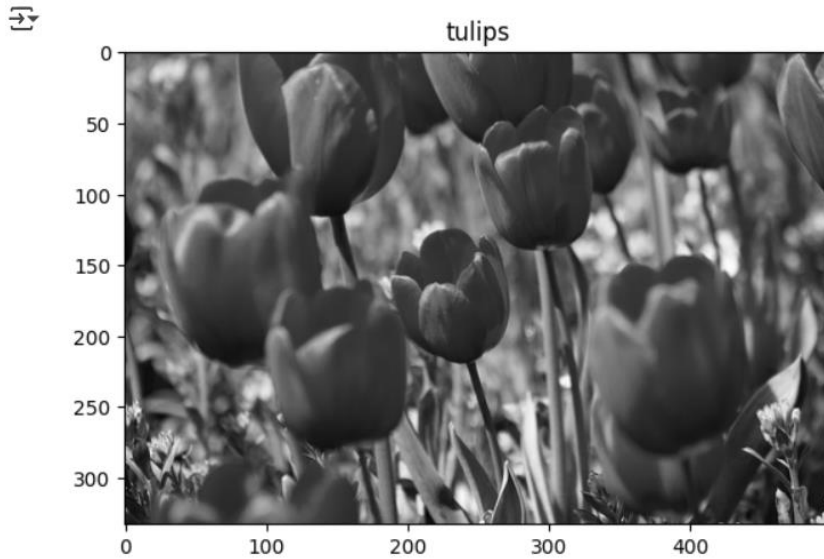
```
#apply preprocessing pipeline  
result = resize_and_rescale(image)  
_ = plt.imshow(result)
```



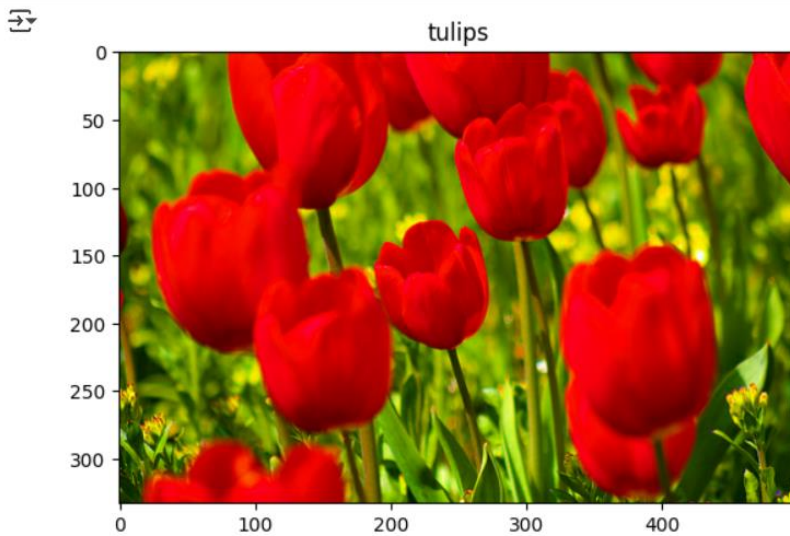
```
# flipping image from left to right  
flipped = tf.image.flip_left_right(image)  
_ = plt.imshow(flipped)  
_ = plt.title(get_label_name (label))
```



```
# convert image to grayscale
grayscale = tf.image.rgb_to_grayscale(image)
_ = plt.imshow(grayscale, cmap='gray')
_ = plt.title(get_label_name(label))
```

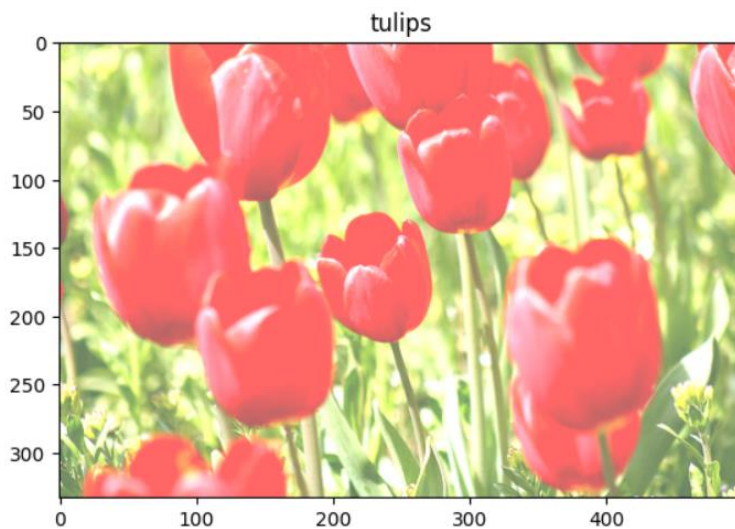


```
# adjust image saturation
saturated = tf.image.adjust_saturation(image, 3)
_ = plt.imshow(saturated)
_ = plt.title(get_label_name(label))
```



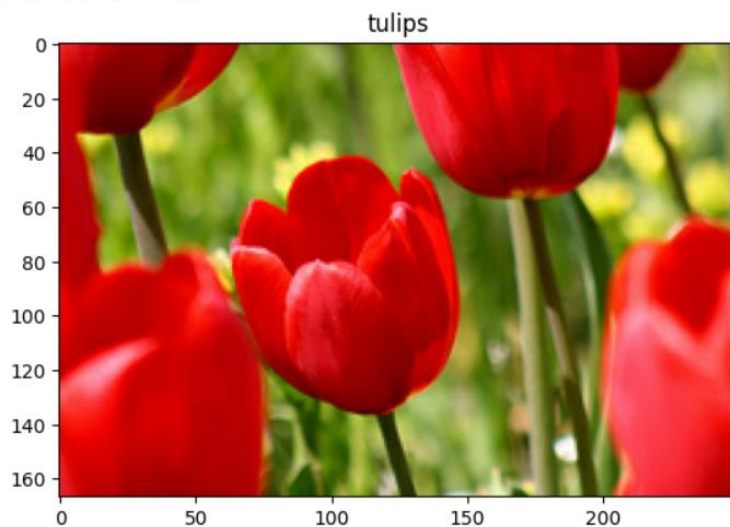

```
# adjust image brightness  
brightness = tf.image.adjust_brightness(image, 0.4)  
_ = plt.imshow(brightness)  
plt.title(get_label_name(label))
```

Text(0.5, 1.0, 'tulips')

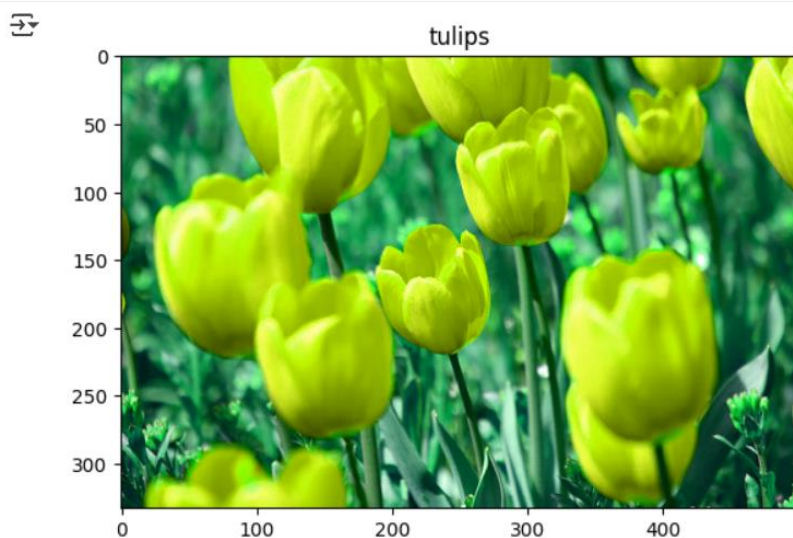


```
# crop the image  
cropped = tf.image.central_crop(image, central_fraction=0.5)  
_ = plt.imshow(cropped)  
plt.title(get_label_name(label))
```

Text(0.5, 1.0, 'tulips')



```
# adjust image hue
hue = tf.image.adjust_hue(image, 0.2)
_ = plt.imshow(hue)
_ = plt.title(get_label_name(label))
```



```
# Rotate the input image 90 degrees counterclockwise.
rotated = tf.image.rot90(image)
_ = plt.imshow(rotated)
_ = plt.title(get_label_name(label))
```

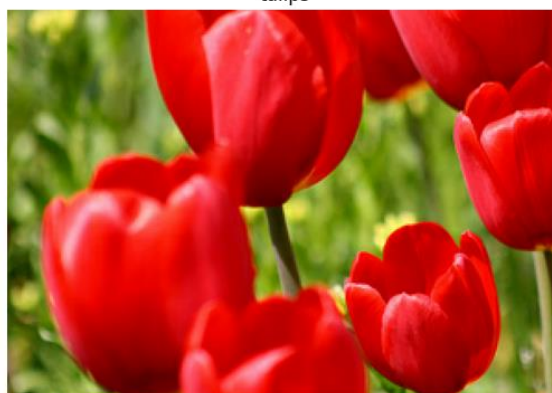


```
▶ for i in range(4):  
    seed = (i, 0) # Tuple of two integers  
    stateless_random_crop = tf.image.stateless_random_crop(  
        value=image, size=[210, 300, 3], seed=seed  
    )  
    plt.imshow(stateless_random_crop.numpy().astype("uint8")) # Convert tensor to image  
    plt.title(get_label_name(label))  
    plt.axis('off')  
    plt.show()
```

tulips



tulips



tulips



tulips



B. Implement on a dataset

Sep 20, 2025

Step 1: Importing necessary libraries.

```
# importing packages
import numpy as np
import os
import PIL
import PIL.Image
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras import layers
import pathlib
```

Step 2: Load the dataset from the URL.

```
# URL of the dataset containing flower photos.
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
# Use TensorFlow's utility function to download the dataset from the URL.
# 'origin' specifies where to download from,
# 'untar=True' will automatically extract the archive.
# 'cache_dir' is set to the current directory ('.'), meaning the file will be stored locally here.
data_dir = tf.keras.utils.get_file('flower_photos', origin=dataset_url, untar=True, cache_dir='.')
```

Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
228813984/228813984 — 11s 0us/step

Step 3: Give the path where the dataset is downloaded

```
[3] # data_dir holds the path to the directory where the dataset is downloaded and extracted.
data_dir
```

```
PosixPath('./datasets/flower_photos')
```

Step 4: Convert the dataset into pathlib path object

```
data_dir = pathlib.Path(data_dir)
data_dir
```

```
PosixPath('datasets/flower_photos')
```

Step 5: Listing all the .jpg files within the subdirectories of data_dir.

```
image_list = list(data_dir.glob('*/**/*.jpg'))
# Display the first 5 jpg file paths from the dataset as a preview.
image_list[:5]
```

```
[PosixPath('datasets/flower_photos/flower_photos/tulips/8687675254_c93f50d8b0_m.jpg'),
PosixPath('datasets/flower_photos/flower_photos/tulips/4561670472_0451888e32_n.jpg'),
PosixPath('datasets/flower_photos/flower_photos/tulips/3396033831_bb88d93630.jpg'),
PosixPath('datasets/flower_photos/flower_photos/tulips/4290566894_c7f061583d_m.jpg'),
PosixPath('datasets/flower_photos/flower_photos/tulips/14044685976_0064faed21.jpg')]
```


Step 6: Counting the total number of .jpg images in the subdirectories of data_dir

```
# Count the total number of .jpg images in the subdirectories of data_dir.  
image_count = len(list(data_dir.glob('*/*.jpg')))  
# Print the total number of jpg images found in the dataset.  
print(image_count)
```

⇒ 3670

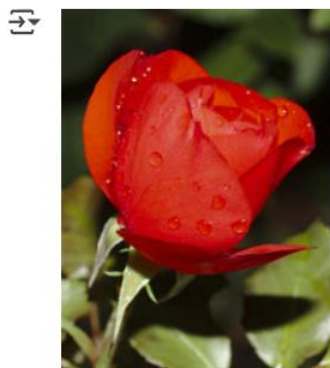
Step 7: Display paths of roses

```
#display path of rose image  
rose_dir = pathlib.Path('/root/.keras/datasets/flower_photos.tgz/flower_photos') / 'roses'  
  
# Get all .jpg files (or use * to get all files)  
rose_images = list(rose_dir.glob('*'))  
  
# Display first 5  
rose_images[:5]
```

⇒ [PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/21346056089_e6f8074e5f_m.jpg'),
PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/12406418663_af20dc225f_n.jpg'),
PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/5181899042_0a6ffe0c8a_n.jpg'),
PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/15537825851_a80b6321d7_n.jpg'),
PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/12338444334_72fcc2fc58_m.jpg')]

Step 8: Displaying Roses

```
PIL.Image.open(rose_images[0])
```



Step 9: Read the images from disk → creating dictionary for flower types to their corresponding numeric labels

```
flower_images_dict = {  
    'roses': list(data_dir.glob('roses/*')),  
    'daisy' : list(data_dir.glob('daisy/*')),  
    'dandelion': list(data_dir.glob('dandelion/*')),  
    'sunflowers': list(data_dir.glob('sunflowers/*')),  
    'tulips': list(data_dir.glob('tulips/*'))  
}  
  
flowers_labels_dict = {  
    'roses': 0,  
    'daisy': 1,  
    'dandelion': 2,  
    'sunflowers': 3,  
    'tulips': 4  
}
```

Step 10: Get the first 5 images path

```
flower_images_dict['roses'][:5]  
[PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/21346056089_e6f8074e5f_m.jpg'),  
PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/12406418663_af20dc225f_n.jpg'),  
PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/5181899042_0a6ffe0c8a_n.jpg'),  
PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/15537825851_a80b6321d7_n.jpg'),  
PosixPath('/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/12338444334_72fcc2fc58_m.jpg')]
```

Step 11: Converting the path of the first image file in the 'roses' category to a string.

```
str(flower_images_dict['roses'][0])  
'/root/.keras/datasets/flower_photos.tgz/flower_photos/roses/21346056089_e6f8074e5f_m.jpg'
```

Step 12: Getting the image

```
PIL.Image.open(flower_images_dict['roses'][0])
```



Step 13: Importing opencv library & reading the image

```
import cv2
img = cv2.imread(str(flower_images_dict['roses'][0]))
```

Step 14: Resize the image

```
cv2.resize(img, (180, 180)).shape
```

```
⇒ (180, 180, 3)
```

Step 15: Initialize and iterate

```
X, y = [], []

for flower_name, images in flower_images_dict.items():
    for image in images:
        img = cv2.imread(str(image))
        resized_img = cv2.resize(img, (180, 180))
        X.append(resized_img)
        y.append(flowers_labels_dict[flower_name])
```

Step 16: Converting list of images to x and y arrays → getting the shape

```
X = np.array(X)
y = np.array(y)
```

```
X.shape, y.shape
```

```
⇒ ((3670, 180, 180, 3), (3670,))
```

Step 18: Dividing data into training and testing sets

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Step 19: Scaling pixel values of both training and testing images

```
x_train_scaled = x_train / 255  
x_test_scaled = x_test / 255
```

Practical 10: Implement RNN (Recurrent Neural Network)

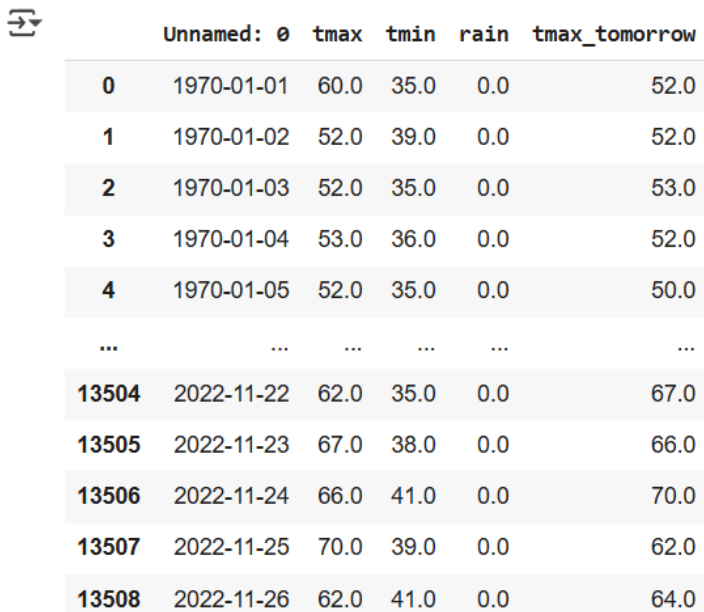
Sep 20, 2025

Date set link →

https://github.com/VikParuchuri/zero_to_gpt/blob/master/data/clean_weather.csv

```
import numpy as np
import pandas as pd
```

```
df = pd.read_csv('/content/sample_data/clean_weather.csv')
df
```



The output shows a pandas DataFrame with 13509 rows and 5 columns. The columns are 'Unnamed: 0', 'tmax', 'tmin', 'rain', and 'tmax_tomorrow'. The first few rows show data for January 1st to 5th, 1970. The last few rows shown are for November 22nd to 26th, 2022.

	Unnamed: 0	tmax	tmin	rain	tmax_tomorrow
0	1970-01-01	60.0	35.0	0.0	52.0
1	1970-01-02	52.0	39.0	0.0	52.0
2	1970-01-03	52.0	35.0	0.0	53.0
3	1970-01-04	53.0	36.0	0.0	52.0
4	1970-01-05	52.0	35.0	0.0	50.0
...
13504	2022-11-22	62.0	35.0	0.0	67.0
13505	2022-11-23	67.0	38.0	0.0	66.0
13506	2022-11-24	66.0	41.0	0.0	70.0
13507	2022-11-25	70.0	39.0	0.0	62.0
13508	2022-11-26	62.0	41.0	0.0	64.0

13509 rows × 5 columns

```
df.ffmpeg()
```



	Unnamed: 0	tmax	tmin	rain	tmax_tomorrow
0	1970-01-01	60.0	35.0	0.0	52.0
1	1970-01-02	52.0	39.0	0.0	52.0
2	1970-01-03	52.0	35.0	0.0	53.0
3	1970-01-04	53.0	36.0	0.0	52.0
4	1970-01-05	52.0	35.0	0.0	50.0
...
13504	2022-11-22	62.0	35.0	0.0	67.0
13505	2022-11-23	67.0	38.0	0.0	66.0
13506	2022-11-24	66.0	41.0	0.0	70.0
13507	2022-11-25	70.0	39.0	0.0	62.0
13508	2022-11-26	62.0	41.0	0.0	64.0

13509 rows × 5 columns



```
df["tmax"].head(10)
```



	tmax
0	60.0
1	52.0
2	52.0
3	53.0
4	52.0
5	50.0
6	52.0
7	56.0
8	54.0
9	57.0

dtype: float64

```
# 4: convert to Array
df["tmax"].head(6).to_numpy()[np.newaxis, :]
```

```
array([[60., 52., 52., 53., 52., 50.]])
```

```
df["tmax"].head(10).to_numpy()[np.newaxis, :]
```

```
array([[60., 52., 52., 53., 52., 50., 52., 56., 54., 57.]])
```

```
# set a arandom seed
np.random.seed(0)
```

```
# 6: initialize weights
# take input no and turn into 2 features
i_weight = np.random.rand(1, 2)
# hidden to hidden weight connection -2 features to 2 features
h_weight = np.random.rand(2, 2)
# output weights connection - tyurn 2 features into one predication
o_weight = np.random.rand(2, 1)
```

```
# get temerpature value
# get 3 temp value from our data
temps = df["tmax"].tail(3).to_numpy()
temps
```

```
array([66., 70., 62.]])
```

```
# 8. assign seque input at each time to step to a different variable
# x0 ,means input at time step 0
# ensure that each element is at 1x1 matrix, so we can multply it
x0 = temps[0].reshape(1,1)
x1 = temps[1].reshape(1,1)
x2 = temps[2].reshape(1,1)
```

```
# 9. calaulate hidden state adn output at each time step
xi_0 = x0 @ i_weight

# there ius no previous time step , so there is no hidden state
# pply RelU over the input to get the hidden state for times step 0 xh_0
xh_0 = np.maximum(0, xi_0)
# get the output at time step 0 xo_0
xo_0 = xh_0 @ o_weight
xo_0
```

```
array([[57.94406231]])
```

```
# we feed the input in the same way as the previous time step
xi_1 = x1 @ i_weight
# this time we do have previous time step , so we calculate xh
# this is multiplying the previous hidden state xh_0 by the hidden weights
xh = xh_0 @ h_weight
xh_1 = np.maximum(0, xh + xi_1)

xo_1 = xh_1 @ o_weight
xo_1
```

→ array([[124.54916092]])

```
▶ xi_2 = x2 @ i_weight
xh = xh_1 @ h_weight
xh_2 = np.maximum(0, xh + xi_2)

xo_2 = xh_2 @ o_weight
xo_2
```

→ array([[190.94853131]])

```
# 10. define new biases and weights
i_weight = np.random.rand(1, 5) / 5-.1
h_weight = np.random.rand(5, 5) / 5-.1
o_weight = np.random.rand(5, 1) / 5-.1
o_weight = np.random.rand(5, 1)*50
o_bias = np.random.rand(1, 1)
```

```
▶ # 11. initialize outputs and hidden states
outputs = np.zeros(3)
hiddens = np.zeros((3, 5))
```

```
▶ # 12. compute outputs and hidden state
prev_hidden = None
sequence = df["tmax"].tail(3).to_numpy()
for i in range(3):
    x = sequence[i].reshape(1, 1)
    xi = x @ i_weight
    if prev_hidden is not None:
        xh = xi + prev_hidden @ h_weight
    else:
        xh = np.tanh(xi)
        prev_hidden = xh
        hiddens[i, ] = xh

    xo = xh @ o_weight + o_bias
    outputs[i] = xo
```



```
# 13. define mean squared error and gradient function\
def mse(actual, predicated):
    return np.mean((actual - predicated) ** 2)

def mse_grad(actual, predicated):
    return (predicated - actual)
```

```
# 14. compute gradient s for backpropogation
# actual nesxt dat temperature
actual = np.array([70, 62, 65])
loss_grad = mse_grad(actual, outputs)
loss_grad
```

```
⇒ array([-59.70189656,  8.86316178, -2.69099411])
```