Experiment 1: Design Single unit perceptron for classification of iris dataset without using predefined models.

Aim: Design Single unit perceptron for classification of iris dataset without using predefined models.

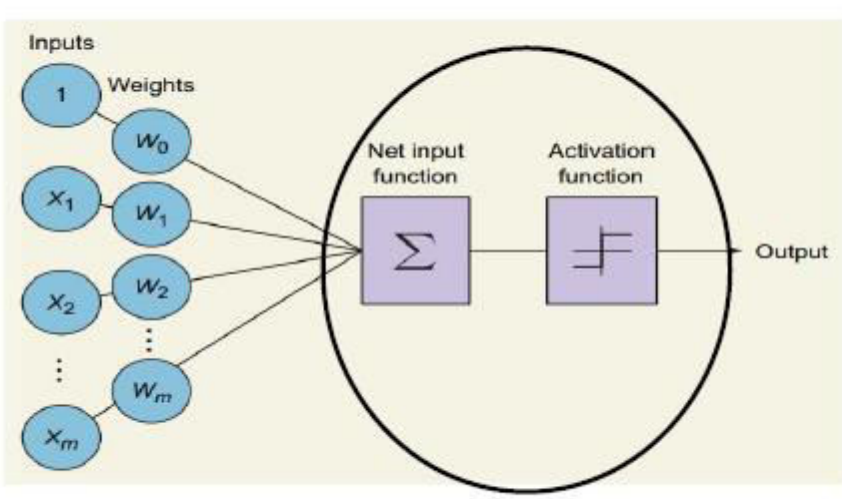
Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:

The Perceptron consists of:
Input vector denoted by uppercase X (x1, x2, . . ., xn) fed to the neuron .
bias (b) is an extra weight used while learning and adjusting the neuron to minimize the cost function
Weights vector—Each x1 is assigned a weight value w1 that represents its importance to distinguish between different input data points.
Neuron functions—The calculations performed within the neuron to modulate the input signals: the weighted sum and step activation function.

Output—controlled by the type of activation function you choose for your network. There are different activation functions eg. a step function, the output is either 0 or 1. Other activation functions produce probability output or float numbers. The output node represents the perceptron prediction.

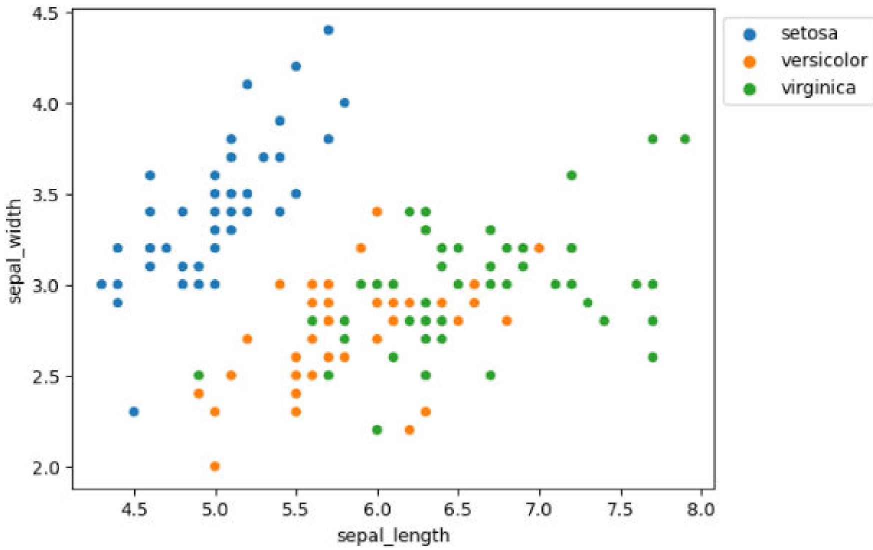


Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
iris=pd.read_csv("iris.csv")
```

```python
import seaborn as sns
import matplotlib.pyplot as plt


sns.scatterplot(x='sepal_length', y='sepal_width', hue='species', data=iris, )

# Placing Legend outside the Figure
plt.legend(bbox_to_anchor=(1, 1), loc=2)
```

Output:




```python
iris['species'].unique()
```

Output: array(['setosa', 'versicolor', 'virginica'], dtype=object)

```python
iris.groupby('species').size()
```

Output:
species
setosa 50
versicolor 50
virginica 50
dtype: int64

```python
#iris = load_iris()
iris = load_iris()
X = iris.data[:100, :2]  # Use only two features and two classes (Setosa and Versicolor)
```

```
y = iris.target[:100]
# Convert labels to -1 and 1
y = np.where(y == 0, -1, 1)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Initialize weights and bias
weights = np.zeros(X_train.shape[1])
bias = 0
learning_rate = 0.1
epochs = 10
# Perceptron training
for epoch in range(epochs):
    for i in range(X_train.shape[0]):
        linear_output = np.dot(X_train[i], weights) + bias
        y_pred = np.where(linear_output > 0, 1, -1)

        # Update weights and bias
        if y_train[i] != y_pred:
            weights += learning_rate * y_train[i] * X_train[i]
            bias += learning_rate * y_train[i]
# Testing the perceptron
correct_predictions = 0
for i in range(X_test.shape[0]):
    linear_output = np.dot(X_test[i], weights) + bias
    y_pred = np.where(linear_output > 0, 1, -1)
    if y_pred == y_test[i]:
        correct_predictions += 1
accuracy = correct_predictions / X_test.shape[0]
print(f"Accuracy: {accuracy * 100:.2f}%")
```

Output:1.0

Result:

Experiment 2: Design, train and test the MLP for tabular data and verify various activation functions and optimizers tensorflow.

Aim: Design, train and test the MLP for tabular data and verify various activation functions and optimizers tensorflow.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

A very common neural network architecture is to stack the neurons in layers on top of each other, called hidden layers. Each layer has n number of neurons. Layers are connected to each other by weight connections.

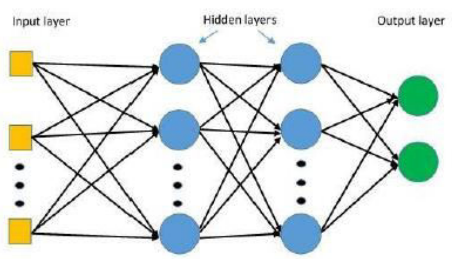


Fig 7.b. Multi Layer Perceptron(ANN)

The main components of the neural network architecture are

Input Layer

It is the initial or starting layer of the Multilayer perceptron. It takes input from the training data set and forwards it to the hidden layer. There are n input nodes in the input layer. The number of input nodes depends on the number of dataset features. Each input vector variable is distributed to each of the nodes of the hidden layer.

Hidden Layer

It is the heart of all Artificial neural networks. This layer comprises all computations of the neural network. The edges of the hidden layer have weights multiplied by the node values. This layer uses the activation function. There can be one or two hidden layers in the model. Several hidden layer nodes should be accurate as few nodes in the hidden layer make the model unable to work efficiently with complex data. More nodes will result in an overfitting problem.

Output Layer

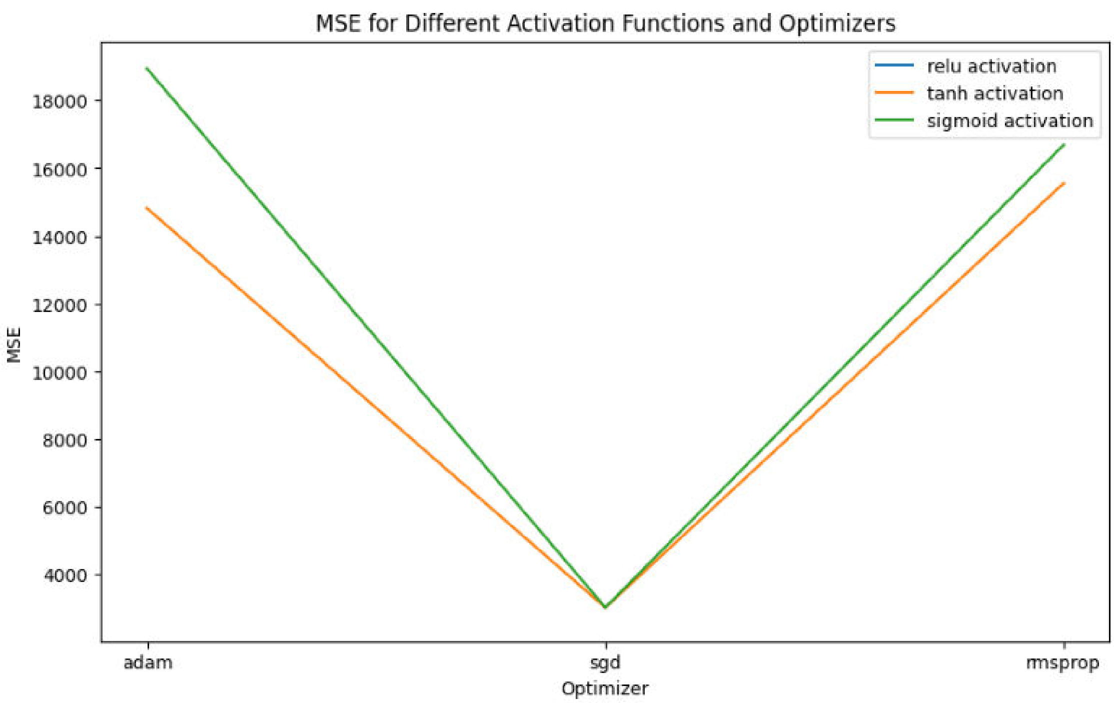
This layer gives the estimated output of the Neural Network as shown in fig 7.d. The number of nodes in the output layer depends on the type of problem. For a single targeted variable, use one node. N classification problem, ANN uses N nodes in the output layer.

Code:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
data=pd.read_csv("/kaggle/input/diabetes-dataset/diabetes.csv")
data
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Load and prepare the data
data = load_diabetes()
X = data.data
y = data.target

# Standardize the features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define the model
def create_model(activation='relu', optimizer='adam'):
    model = Sequential()
    model.add(Dense(64, input_shape=(X_train.shape[1],), activation=activation))
    model.add(Dense(32, activation=activation))
    model.add(Dense(1))  # No activation for regression output

    # Compile the model
    model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['mse'])
    return model

# Define a list of activation functions and optimizers to test
activation_functions = ['relu', 'tanh', 'sigmoid']
optimizers = ['adam', 'sgd', 'rmsprop']

results = {}

# Train and test the model with different configurations
for activation in activation_functions:
    for optimizer in optimizers:
        model = create_model(activation=activation, optimizer=optimizer)
```

```
        model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)

        # Evaluate the model
        loss, mse = model.evaluate(X_test, y_test, verbose=0)
        results[(activation, optimizer)] = mse
        print(f"Activation: {activation}, Optimizer: {optimizer}, MSE: {mse:.4f}")

# Plotting
plt.figure(figsize=(10, 6))
for activation in activation_functions:
    plt.plot([optimizer for (act, optimizer) in results.keys() if act == activation],
             [results[(activation, optimizer)] for optimizer in optimizers],
             label=f'{activation} activation')

plt.title('MSE for Different Activation Functions and Optimizers')
plt.xlabel('Optimizer')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

Output:




```
from tensorflow.keras.optimizers import SGD

# Use a smaller learning rate
optimizer = SGD(learning_rate=0.0001)

# Recreate and compile the model with the adjusted optimizer
```

```
model = create_model(activation='relu', optimizer=optimizer)

# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)

# Evaluate the model
loss, mse = model.evaluate(X_test, y_test, verbose=0)
print(f"Activation: relu, Optimizer: sgd, MSE: {mse:.4f}")
```

Output:

```
Activation: relu, Optimizer: sgd, MSE: 2831.2041
```

To eliminate NaN (Not a Number) values in the MSE when using the ReLU activation function with the SGD optimizer, the following changes can be made:

Reduce Learning Rate: A high learning rate can cause issues with convergence, especially with the ReLU activation function, leading to NaN values. Lowering the learning rate for the SGD optimizer can help stabilize training.

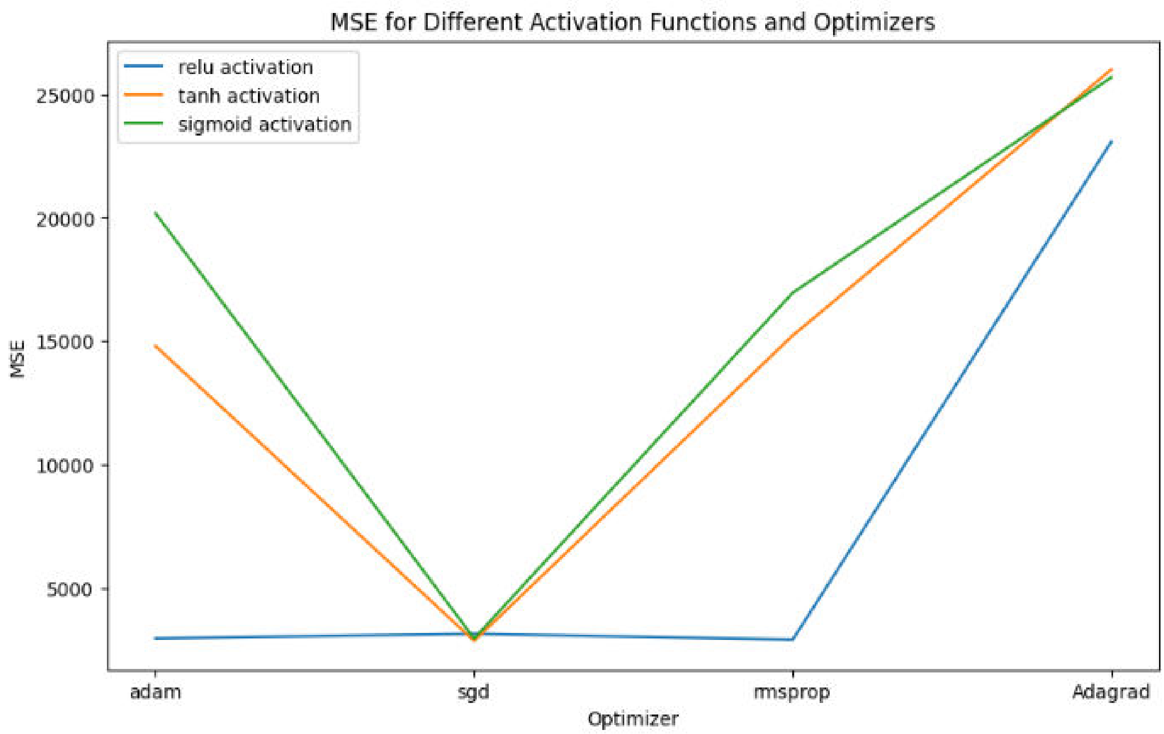
Initialize Weights Properly: Proper weight initialization (like He initialization) can also help prevent issues with ReLU activation

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization
from tensorflow.keras.optimizers import Adam, SGD, RMSprop, Adagrad
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Load and prepare the data
data = load_diabetes()
X = data.data
y = data.target

# Standardize the features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define the model
def create_model(activation='relu', optimizer='adam'):
```

```
    model = Sequential()
    model.add(Dense(64, input_shape=(X_train.shape[1],), activation=activation))
    #model.add(BatchNormalization())  # Add Batch Normalization
    model.add(Dense(32, activation=activation))
    #model.add(BatchNormalization())  # Add Batch Normalization
    model.add(Dense(1))  # No activation for regression output

    # Compile the model with gradient clipping
    optimizer = tf.keras.optimizers.get(optimizer)
    if isinstance(optimizer, SGD):
        optimizer.learning_rate = 0.0001  # Reduced learning rate

    model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['mse'])
    return model

# Define a list of activation functions and optimizers to test
activation_functions = ['relu', 'tanh', 'sigmoid']
optimizers = ['adam', 'sgd', 'rmsprop', 'Adagrad']

results = {}

# Train and test the model with different configurations
for activation in activation_functions:
    for optimizer in optimizers:
        model = create_model(activation=activation, optimizer=optimizer)
        model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)

        # Evaluate the model
        loss, mse = model.evaluate(X_test, y_test, verbose=0)
        results[(activation, optimizer)] = mse
        print(f"Activation: {activation}, Optimizer: {optimizer}, MSE: {mse:.4f}")

# Plotting
plt.figure(figsize=(10, 6))
for activation in activation_functions:
    plt.plot([optimizer for (act, optimizer) in results.keys() if act == activation],
             [results[(activation, optimizer)] for optimizer in optimizers],
             label=f'{activation} activation')

plt.title('MSE for Different Activation Functions and Optimizers')
plt.xlabel('Optimizer')
plt.ylabel('MSE')
# Option 1: Manually set the y-axis limits
#plt.ylim(0, 30000)  # Set lower and upper bounds for the y-axis

# Option 2: Use a logarithmic scale for the y-axis
#plt.yscale('log')
plt.legend()
plt.show()
```

Output

```
Activation: relu, Optimizer: adam, MSE: 2956.9028
Activation: relu, Optimizer: sgd, MSE: 3147.7812
Activation: relu, Optimizer: rmsprop, MSE: 2907.7595
Activation: relu, Optimizer: Adagrad, MSE: 23059.0273
Activation: tanh, Optimizer: adam, MSE: 14791.1758
Activation: tanh, Optimizer: sgd, MSE: 2860.1355
Activation: tanh, Optimizer: rmsprop, MSE: 15218.6572
Activation: tanh, Optimizer: Adagrad, MSE: 25984.4414
Activation: sigmoid, Optimizer: adam, MSE: 20165.0508
Activation: sigmoid, Optimizer: sgd, MSE: 2957.6797
Activation: sigmoid, Optimizer: rmsprop, MSE: 16949.9629
Activation: sigmoid, Optimizer: Adagrad, MSE: 25664.1484
```



Result:

Experiment 3: Design and implement to classify 32x32 images using MLP using tensorflow/keras and check the accuracy.

Aim: Design and implement to classify 32x32 images using MLP using tensorflow/keras and check the accuracy.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

A very common neural network architecture is to stack the neurons in layers on top of each other, called hidden layers. Each layer has n number of neurons. Layers are connected to each other by weight connections.

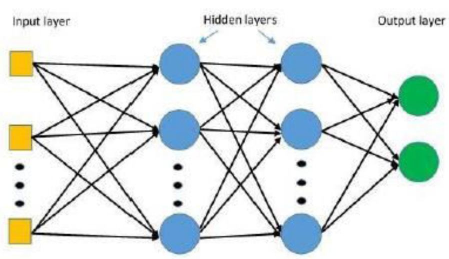


Fig 7.b. Multi Layer Perceptron(ANN)

The main components of the neural network architecture are

Input Layer

It is the initial or starting layer of the Multilayer perceptron. It takes input from the training data set and forwards it to the hidden layer. There are n input nodes in the input layer. The number of input nodes depends on the number of dataset features. Each input vector variable is distributed to each of the nodes of the hidden layer.

Hidden Layer

It is the heart of all Artificial neural networks. This layer comprises all computations of the neural network. The edges of the hidden layer have weights multiplied by the node values. This layer uses the activation function. There can be one or two hidden layers in the model. Several hidden layer nodes should be accurate as few nodes in the hidden layer make the model unable to work efficiently with complex data. More nodes will result in an overfitting problem.

Output Layer

This layer gives the estimated output of the Neural Network as shown in fig 7.d. The number of nodes in the output layer depends on the type of problem. For a single targeted variable, use one node. N classification problem, ANN uses N nodes in the output layer.

Code:

```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
# Normalize the images to a range of 0 to 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert class vectors to binary class matrices (one-hot encoding)
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build the MLP model
model = Sequential()
model.add(Flatten(input_shape=(32, 32, 3)))  # Flatten the input
model.add(Dense(512, activation='relu'))      # First hidden layer
model.add(Dense(256, activation='relu'))      # Second hidden layer
model.add(Dense(10, activation='softmax'))    # Output layer

model.summary()

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

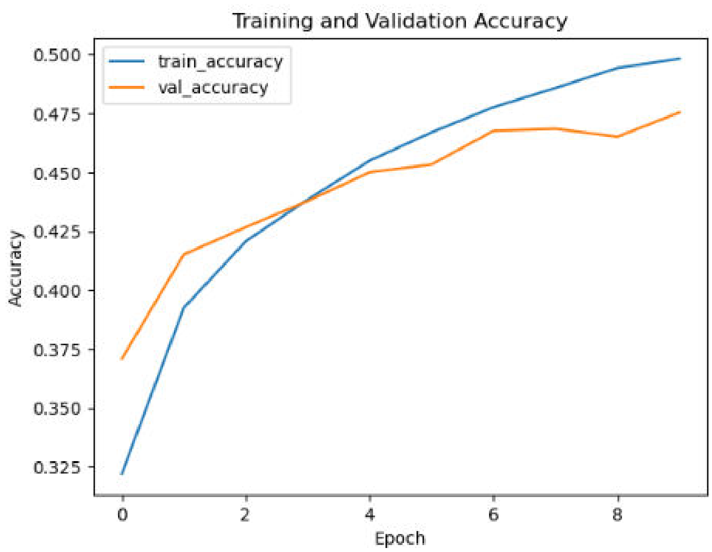
```python
# Train the model
history=model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

OUTPUT-

```
Epoch 1/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 43s 31ms/step - accuracy: 0.2744 - loss: 2.0646 - val_accuracy: 0.3708 - val_loss: 1.7652
Epoch 2/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 37s 30ms/step - accuracy: 0.3909 - loss: 1.7071 - val_accuracy: 0.4151 - val_loss: 1.6501
Epoch 3/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 41s 30ms/step - accuracy: 0.4163 - loss: 1.6210 - val_accuracy: 0.4267 - val_loss: 1.6177
Epoch 4/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 38s 30ms/step - accuracy: 0.4338 - loss: 1.5774 - val_accuracy: 0.4378 - val_loss: 1.5954
Epoch 5/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 37s 30ms/step - accuracy: 0.4525 - loss: 1.5229 - val_accuracy: 0.4500 - val_loss: 1.5634
Epoch 6/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 37s 30ms/step - accuracy: 0.4722 - loss: 1.4776 - val_accuracy: 0.4533 - val_loss: 1.5585
Epoch 7/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 41s 29ms/step - accuracy: 0.4780 - loss: 1.4534 - val_accuracy: 0.4676 - val_loss: 1.5214
Epoch 8/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 41s 30ms/step - accuracy: 0.4875 - loss: 1.4335 - val_accuracy: 0.4685 - val_loss: 1.5086
Epoch 9/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 39s 31ms/step - accuracy: 0.4979 - loss: 1.4162 - val_accuracy: 0.4651 - val_loss: 1.5241
Epoch 10/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 36s 28ms/step - accuracy: 0.5005 - loss: 1.3886 - val_accuracy: 0.4754 - val_loss: 1.4988
```

```python
# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
# Plot training & validation accuracy
import matplotlib.pyplot as plt
plt.plot(history.history['accuracy'], label='train_accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Output:



Result:

Experiment 4: Design and implement a CNN model to classify multi category JPG images with tensorflow /keras and check accuracy. Predict labels for new images.

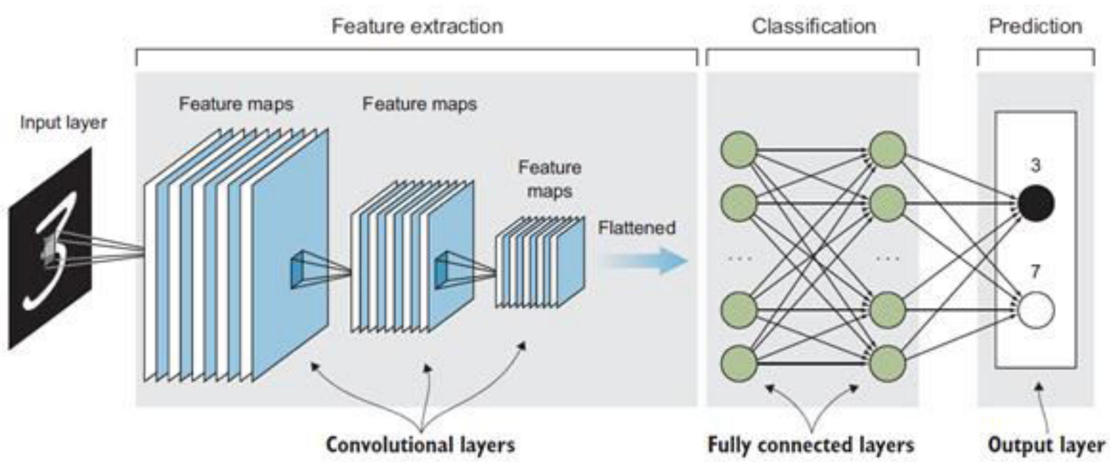
Aim: Design and implement a CNN model to classify multi category JPG images with tensorflow /keras and check accuracy. Predict labels for new images.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

Convolutional Neural Network consists of multiple layers like

- The input layer
- Convolutional layer
- Pooling layer and
- fully connected layers.



The CNN architecture consists of the following: input layer, convolutional layers, fully connected layers, and output prediction.

Code:

```
import tensorflow as tf
# CIFAR-10 Image Classification using CNN
# Step 1: Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
(x_train, y_train), (x_test, y_test) = datasets.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}")
```

Output:

```
x_train shape: (50000, 32, 32, 3)
y_train shape: (50000, 1)
x_test shape: (10000, 32, 32, 3)
y_test shape: (10000, 1)
```

```
# Reduce pixel values
# Normalize pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0

# flatten the label values
y_train, y_test = y_train.flatten(), y_test.flatten()
# number of classes
K = len(set(y_train))

# calculate total number of classes
# for output layer
print("number of classes:", K)
```

Output: `number of classes: 10`

```
# 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 = (10,10))

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)
plt.figure()
plt.imshow(x_train[12])
plt.colorbar()
# Step 4: Build the CNN Model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 output units for the 10 classes
])
# View the model summary
model.summary()
```

Output:

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_3 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_4 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_4 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_5 (Conv2D)	(None, 4, 4, 64)	36,928
max_pooling2d_5 (MaxPooling2D)	(None, 2, 2, 64)	0
flatten_1 (Flatten)	(None, 256)	0
dense_2 (Dense)	(None, 64)	16,448
dense_3 (Dense)	(None, 10)	650

Total params: 73,418 (286.79 KB)
Trainable params: 73,418 (286.79 KB)
Non-trainable params: 0 (0.00 B)

```python
# Step 5: Compile the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

```python
# Step 6: Train the Model
history = model.fit(x_train, y_train, epochs=10,validation_data=(x_test, y_test))
```

Output:

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21s 12ms/step - accuracy: 0.2870 - loss: 2.7930 - val_accuracy: 0.4809 - val_loss: 1.4280
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.4915 - loss: 1.4182 - val_accuracy: 0.5390 - val_loss: 1.2972
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.5559 - loss: 1.2550 - val_accuracy: 0.5484 - val_loss: 1.3016
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21s 14ms/step - accuracy: 0.5900 - loss: 1.1668 - val_accuracy: 0.5921 - val_loss: 1.1709
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6137 - loss: 1.0946 - val_accuracy: 0.6058 - val_loss: 1.1391
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6403 - loss: 1.0351 - val_accuracy: 0.6244 - val_loss: 1.0790
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6619 - loss: 0.9652 - val_accuracy: 0.6369 - val_loss: 1.0687
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6786 - loss: 0.9334 - val_accuracy: 0.6302 - val_loss: 1.0774
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21s 13ms/step - accuracy: 0.6882 - loss: 0.8946 - val_accuracy: 0.6575 - val_loss: 1.0149
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6974 - loss: 0.8531 - val_accuracy: 0.6379 - val_loss: 1.0576
```

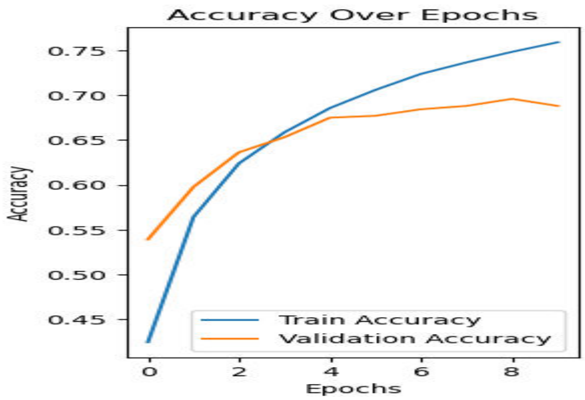
```python
# Step 7: Evaluate the Model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"\nTest accuracy: {test_acc}")
```

OUTPUT:

```
313/313 - 1s - 4ms/step - accuracy: 0.6379 - loss: 1.0576

Test accuracy: 0.6378999948501587
```

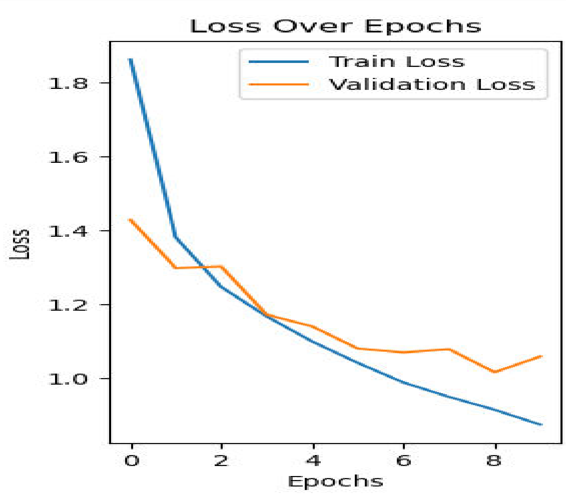
```python
# Step 8: Visualize Training and Validation Accuracy and Loss
plt.figure(figsize=(12, 4))
# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy Over Epochs')
```

OUTPUT:




```python
# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Over Epochs')

plt.show()
```

OUTPUT:

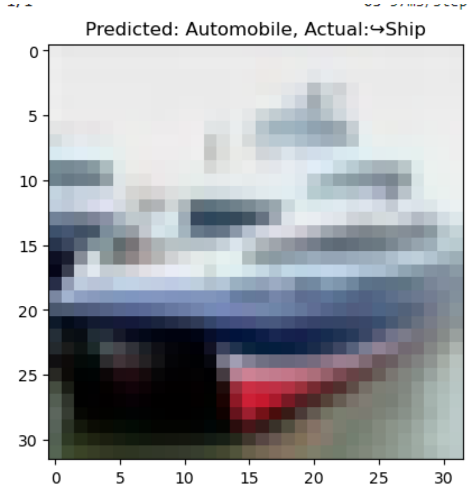



```python
# Step 9: Make Predictions on the Test Data
```

```python
predictions = model.predict(x_test)
# Visualize some predictions
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_test[i])
    plt.xlabel(f"True: {class_names[y_test[i][0]]}\nPred:
{class_names[np.argmax(predictions[i])]}")
plt.show()
```

```python
import numpy as np
# Function to predict and display an image from test data
def predict_image(index):
    img = x_test[index]
    prediction = model.predict(np.expand_dims(img, axis=0))
    predicted_class = np.argmax(prediction)
    plt.imshow(img)
    plt.title(f"Predicted: {class_names[predicted_class]},
Actual:↪{class_names[y_test[index][0]]}")
    plt.show()
# Predict an example image from test set
predict_image(1)
```

OUTPUT:




```python
To increase the accuracy from 63 %, we Use a Deeper Model
Increase the depth of your CNN by adding more convolutional layers or increase the number
of filters in existing layers to capture more complex features.
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
```

```
    layers.Conv2D(32, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.Conv2D(128, (3, 3), activation='relu'),
   # layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dense(10, activation='softmax')
])
# View the model summary
model.summary()
```

OUTPUT:

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
conv2d_1 (Conv2D)	(None, 28, 28, 32)	9,248
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 12, 12, 64)	18,496
conv2d_3 (Conv2D)	(None, 10, 10, 64)	36,928
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_4 (Conv2D)	(None, 3, 3, 128)	73,856
conv2d_5 (Conv2D)	(None, 1, 1, 128)	147,584
flatten (Flatten)	(None, 128)	0
dense (Dense)	(None, 256)	33,024
dense_1 (Dense)	(None, 10)	2,570

Total params: 322,602 (1.23 MB)
Trainable params: 322,602 (1.23 MB)
Non-trainable params: 0 (0.00 B)

```
# Step 5: Compile the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',metrics=['accuracy'])
# Step 6: Train the Model
history = model.fit(x_train, y_train, batch_size=64, epochs=10,validation_data=(x_test,
y_test))

# Step 7: Evaluate the Model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"\nTest accuracy: {test_acc}")
# Step 8: Visualize Training and Validation Accuracy and Loss
plt.figure(figsize=(12, 4))
```

```python
# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy Over Epochs')
# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Over Epochs')

plt.show()
```

Result:

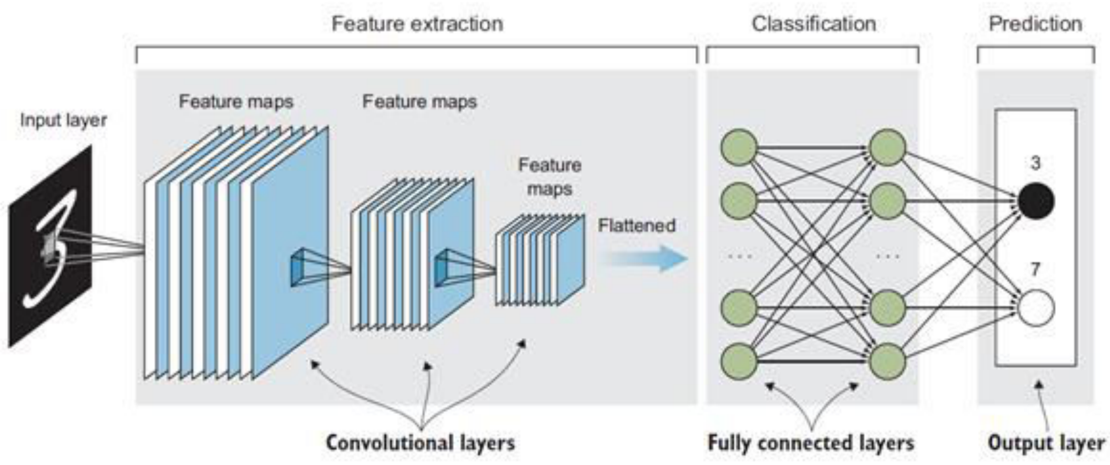
Experiment 5: : Design and implement a CNN model to classify multi category tiff images with tensorflow /keras

Aim: Design and implement a CNN model to classify multi category tiff images with tensorflow /keras and check the accuracy. Check whether your model is overfit / underfit / perfect fit and apply the techniques to avoid overfit and underfit like regulizers, dropouts etc.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory: Convolutional Neural Network consists of multiple layers like

- The input layer
- Convolutional layer
- Pooling layer and
- fully connected layers.



The CNN architecture consists of the following: input layer, convolutional layers, fully connected layers, and output prediction.

Code:

```
import os
import numpy as np
from tensorflow.keras.datasets import cifar10
from PIL import Image

# Create directories to save the images
train_dir = 'cifar10_tiff/train'
test_dir = 'cifar10_tiff/test'
os.makedirs(train_dir, exist_ok=True)
```

```
os.makedirs(test_dir, exist_ok=True)

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Define class labels for CIFAR-10
class_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']

# Helper function to save images in .tiff format
def save_images(images, labels, directory):
    for i, (image_array, label) in enumerate(zip(images, labels)):
        # Convert numpy array to PIL image
        image = Image.fromarray(image_array)

        # Define the label name
        label_name = class_labels[int(label)]

        # Create a subdirectory for each class
        label_dir = os.path.join(directory, label_name)
        os.makedirs(label_dir, exist_ok=True)

        # Save the image in .tiff format
        image_path = os.path.join(label_dir, f"{label_name}_{i}.tiff")
        image.save(image_path, format='TIFF')

# Save training and test images as .tiff
save_images(x_train, y_train, train_dir)
save_images(x_test, y_test, test_dir)

print("Images have been successfully saved as .tiff files.")
```

Step 1: Set Up Libraries and Import Modules

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os
```

Step 2: Define Data Directories and Data Generator Assuming that the .tiff images are saved in folders like cifar10_tiff/train and cifar10_tiff/test, with subdirectories for each class (e.g., airplane, automobile, etc.).

```
# Set paths
train_dir = 'cifar10_tiff/train'
test_dir = 'cifar10_tiff/test'

# Define ImageDataGenerator with data augmentation for the training set
train_datagen = ImageDataGenerator(
    rescale=1.0/255.0,
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
```

```
    validation_split=0.2  # Split 20% of training data for validation
)

# Define ImageDataGenerator for the test set (only rescaling)
test_datagen = ImageDataGenerator(rescale=1.0/255.0)

# Load training data
train_data = train_datagen.flow_from_directory(
    directory=train_dir,
    target_size=(32, 32),  # CIFAR-10 images are 32x32 pixels
    batch_size=32,
    class_mode='categorical',
    subset='training'
)

# Load validation data
validation_data = train_datagen.flow_from_directory(
    directory=train_dir,
    target_size=(32, 32),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)

# Load test data
test_data = test_datagen.flow_from_directory(
    directory=test_dir,
    target_size=(32, 32),
    batch_size=32,
    class_mode='categorical',
    shuffle=False
)
```

Output: `Found 40000 images belonging to 10 classes.`

```
Found 10000 images belonging to 10 classes.
Found 10000 images belonging to 10 classes.
```

Step 3: Build the CNN Model Define a CNN model suitable for classifying the CIFAR-10 images.

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu'),
```

```
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')  # 10 classes for CIFAR-10
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

OUTPUT

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
dropout (Dropout)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
dropout_1 (Dropout)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
dropout_2 (Dropout)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 256)	131,328
dropout_3 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2,570

Total params: 227,146 (887.29 KB)
Trainable params: 227,146 (887.29 KB)
Non-trainable params: 0 (0.00 B)

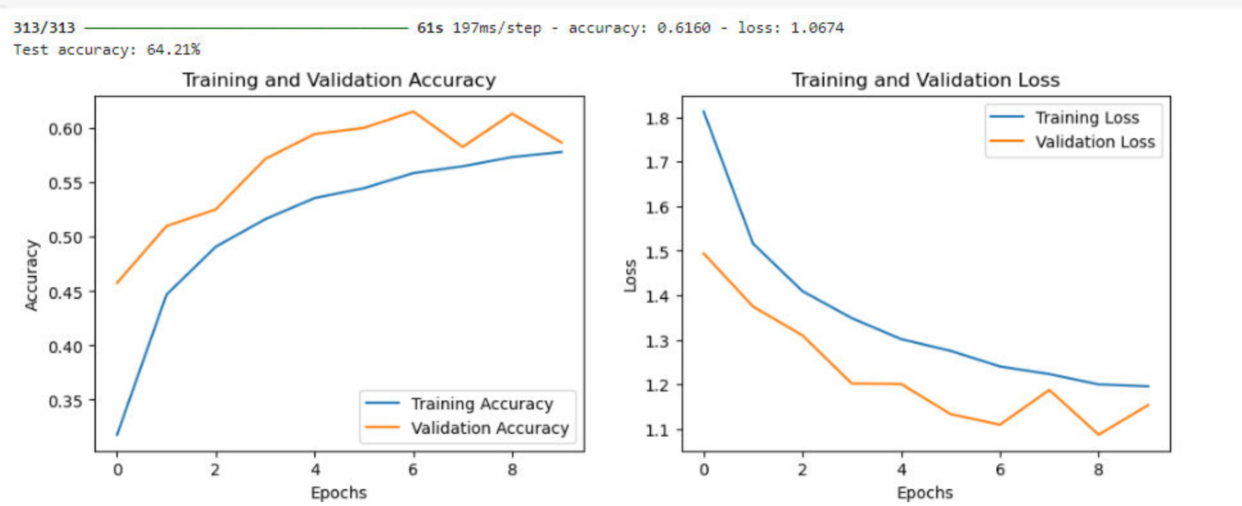
```
from tensorflow.keras.callbacks import EarlyStopping

# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

history =
model.fit(train_data,validation_data=validation_data,epochs=10,callbacks=[early_stopping])
```

Step 5: Evaluate the Model Check model performance on the test dataset and plot accuracy and loss.

```
# Evaluate on test data
test_loss, test_accuracy = model.evaluate(test_data)
print(f"Test accuracy: {test_accuracy * 100:.2f}%")

# Plot training history
import matplotlib.pyplot as plt
```

```python
# Plot accuracy
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')

# Plot loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')

plt.show()
```

OUTPUT:



Step 6: Make Predictions (Optional) You can use the trained model to make predictions on individual images or batches of images.

```python
# Get predictions
predictions = model.predict(test_data)
predicted_classes = tf.argmax(predictions, axis=1)

# Actual classes from the test data generator
```

```python
true_classes = test_data.classes

# Accuracy by comparing predicted and actual classes
accuracy = np.mean(predicted_classes == true_classes)
print(f"Prediction accuracy on test set: {accuracy * 100:.2f}%")
```

OUTPUT:

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 7s 23ms/step

Prediction accuracy on test set: 64.21%
```

Result:

Experiment 6: Implement a CNN architecture (LeNet, Alexnet, VGG, etc) model to classify multi category Satellite images with tensorflow / keras and check the accuracy

Aim: Implement a CNN architecture (LeNet, Alexnet, VGG, etc) model to classify multi category Satellite images with tensorflow / keras and check the accuracy. Check whether your model isoverfit / underfit / perfect fit and apply the techniques to avoid overfit and underfit.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

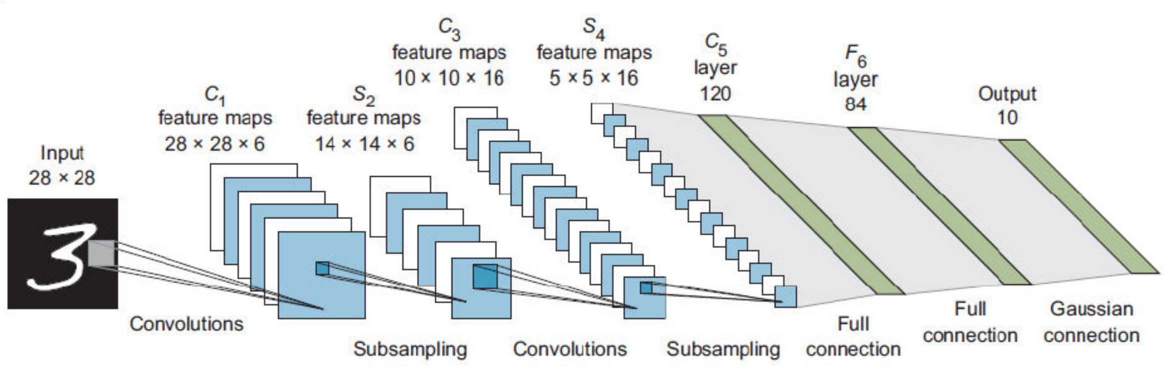
Theory:

In 1998, Lecun et al. introduced a pioneering CNN called *LeNet-5*. The architecture is composed of five weight layers, and hence the name LeNet-5: three convolutional layers and two fully connected layers. We refer to the convolutional and fully connected layers as *weight layers* because they contain trainable weights as opposed to pooling layers that don't contain any weights.

- LeNet architecture:

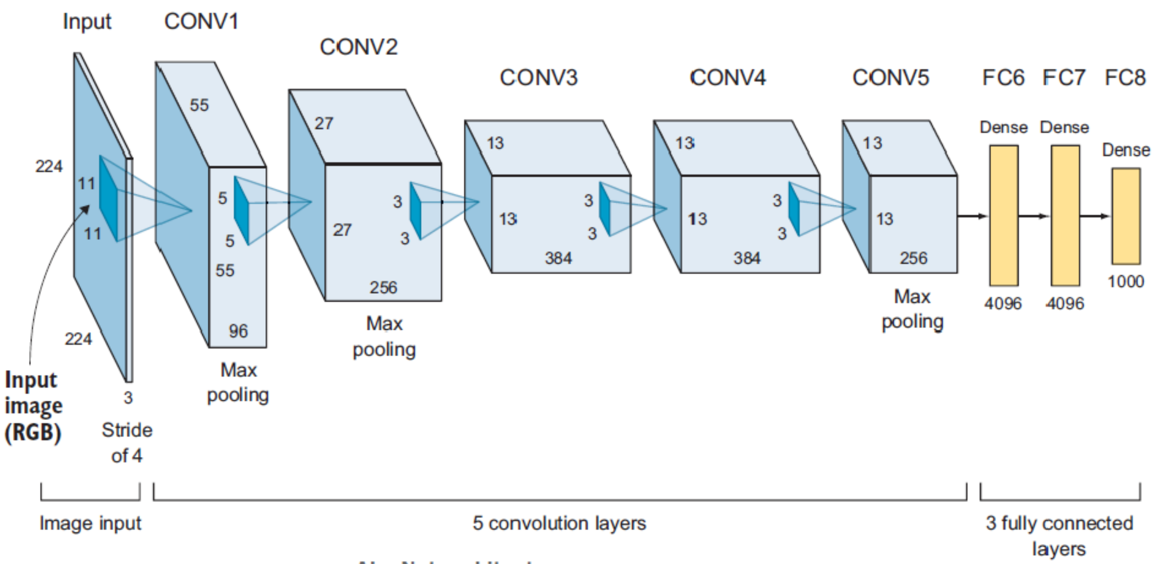
The architecture of LeNet-5 is shown in figure below:
INPUT IMAGE ⇒ C1 ⇒ TANH ⇒ S2 ⇒ C3 ⇒ TANH ⇒ S4 ⇒ C5 ⇒ TANH ⇒FC6 ⇒ SOFTMAX7

where C is a convolutional layer, S is a subsampling or pooling layer, and FC is a fully connected layer.



LeNet architecture

AlexNet was the winner of the ILSVRC image classification competition in 2012. Krizhevsky et al. created the neural network architecture and trained it on 1.2 million high-resolution images into 1,000 different classes of the ImageNet dataset. AlexNet has a lot of similarities to LeNet but is much deeper (more hidden layers) and bigger (more filters per layer). They have similar building blocks: a series of convolutional and pooling layers stacked on top of each other followed by fully connected layers and a softmax. We've seen that LeNet has around 61,000 parameters, whereas AlexNet has about 60 million parameters and 650,000 neurons, which gives it a larger learning capacity to understand more complex features.



AlexNet architecture

Code:

```
import os
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
# Define paths
data_folder_path = 'C:\\Users\\Gove\\Desktop\\archive (1) (1)\\data'
categories = ['cloudy', 'desert', 'green_area', 'water']

# Check the number of images in each category
for category in categories:
    folder_path = os.path.join(data_folder_path, category)
    print(f"{category}: {len(os.listdir(folder_path))} images")
```

output:

```
cloudy: 1500 images
desert: 1131 images
green_area: 1500 images
water: 1500 images
```

```python
# Display a few images from each category
fig, axs = plt.subplots(1, 4, figsize=(15, 5))
for i, category in enumerate(categories):
    folder_path = os.path.join(data_folder_path, category)
    img_path = os.path.join(folder_path, os.listdir(folder_path)[0])
    img = plt.imread(img_path)
    axs[i].imshow(img)
    axs[i].set_title(category)
    axs[i].axis('off')
plt.show()
```

output:


```python
from tensorflow.keras.preprocessing.image import img_to_array, load_img

# Resize images to a standard size
IMG_SIZE = (64, 64)  # For LeNet and AlexNet
data = []
labels = []

for category in categories:
    folder_path = os.path.join(data_folder_path, category)
```

```
    for img_file in os.listdir(folder_path):
        img_path = os.path.join(folder_path, img_file)
        img = load_img(img_path, target_size=IMG_SIZE)
        img_array = img_to_array(img)
        data.append(img_array)
        labels.append(categories.index(category))

# Convert to numpy arrays and normalize
data = np.array(data, dtype='float32') / 255.0
labels = np.array(labels)

# Split data into training, validation, and test sets
X_train, X_test, y_train, y_test = train_test_split(data, labels,
test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.2, random_state=42)

from tensorflow.keras import Input

def create_lenet():
    model = models.Sequential()
    # Define the input shape explicitly using Input
    model.add(Input(shape=(IMG_SIZE[0], IMG_SIZE[1], 3)))

    # First convolutional layer
    model.add(layers.Conv2D(6, (5, 5), activation='tanh'))
    model.add(layers.AveragePooling2D(pool_size=(2, 2)))  # Add
pool_size=(2, 2)

    # Second convolutional layer
    model.add(layers.Conv2D(16, (5, 5), activation='tanh'))
    model.add(layers.AveragePooling2D(pool_size=(2, 2)))  # Add
pool_size=(2, 2)

    # Fully connected layers
    model.add(layers.Flatten())
    model.add(layers.Dense(120, activation='tanh'))
    model.add(layers.Dense(84, activation='tanh'))
    model.add(layers.Dense(len(categories), activation='softmax'))
```

```
    return model

# Create and compile the model
lenet_model = create_lenet()
lenet_model.compile(optimizer=Adam(),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

def create_alexnet():
    model = models.Sequential()
    # First convolutional layer
    model.add(layers.Conv2D(96, (11, 11), strides=4, activation='relu',
input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3)))
    model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2))  #
Reduced pool size

    # Second convolutional layer
    model.add(layers.Conv2D(256, (5, 5), padding='same',
activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2))  #
Adjusted pooling layer

    # Third convolutional layer
    model.add(layers.Conv2D(384, (3, 3), padding='same',
activation='relu'))

    # Fourth convolutional layer
    model.add(layers.Conv2D(384, (3, 3), padding='same',
activation='relu'))

    # Fifth convolutional layer
    model.add(layers.Conv2D(256, (3, 3), padding='same',
activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(2, 2), strides=2))  #
Adjusted pooling layer

    # Flatten and fully connected layers
    model.add(layers.Flatten())
    model.add(layers.Dense(4096, activation='relu'))
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(4096, activation='relu'))
```

```python
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(len(categories), activation='softmax'))

    return model

# Create and compile the AlexNet model
alexnet_model = create_alexnet()
alexnet_model.compile(optimizer=Adam(),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Training LeNet model
lenet_history = lenet_model.fit(X_train, y_train,
validation_data=(X_val, y_val), epochs=10, batch_size=32)

# Training AlexNet model
alexnet_history = alexnet_model.fit(X_train, y_train,
validation_data=(X_val, y_val), epochs=10, batch_size=32)
```

output:

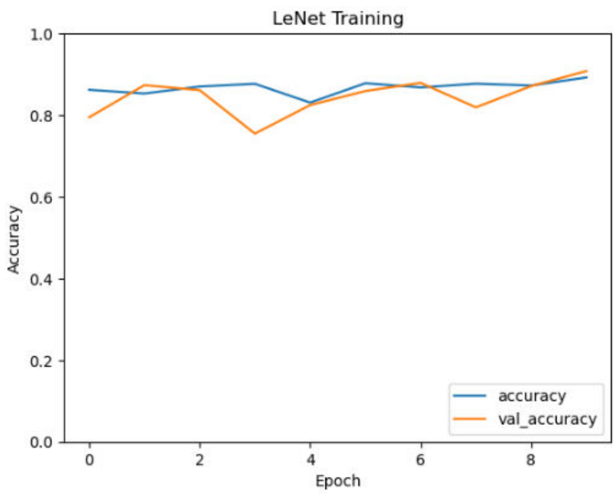
```
Epoch 10/10

113/113 ━━━━━━━━━━━━━━━━━━━━ 33s 293ms/step - accuracy:
0.8432 - loss: 0.3794 - val_accuracy: 0.7270 - val_loss: 0.5328.
```

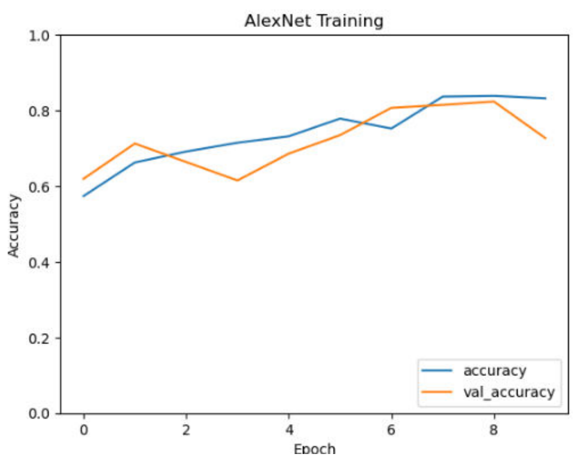
```python
# Evaluate LeNet
lenet_test_loss, lenet_test_acc = lenet_model.evaluate(X_test, y_test)
print(f"LeNet Test Accuracy: {lenet_test_acc:.4f}")
# Evaluate AlexNet
alexnet_test_loss, alexnet_test_acc = alexnet_model.evaluate(X_test,
y_test)
print(f"AlexNet Test Accuracy: {alexnet_test_acc:.4f}")
```

output:

```
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9108 - loss: 0.2518
LeNet Test Accuracy: 0.9219
36/36 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.7283 - loss: 0.5087
AlexNet Test Accuracy: 0.7178
```

```
def plot_history(history, title):
    plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label='val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0, 1])
    plt.title(title)
    plt.legend(loc='lower right')
    plt.show()
plot_history(lenet_history, "LeNet Training")
plot_history(alexnet_history, "AlexNet Training")
```





Result:

Experiment 7: Implement ResNet model to classify multi category medical images with tensorflow / keras and check the accuracy

Aim: Implement ResNet model to classify multi category medical images with tensorflow / keras and check the accuracy. Check whether your model is overfit / underfit / perfect fit and apply the techniques to avoid overfit and underfit.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

The Residual Neural Network (ResNet) was developed in 2015 by a group from the Microsoft Research team. They introduced a novel residual module architecture with skip connections. The network also features heavy batch normalization for the hidden layers. This technique allowed the team to train very deep neural networks with 50, 101, and 152 weight layers while still having lower complexity than smaller networks like VGGNet (19 layers)

Code:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
# Install necessary libraries
!pip install tensorflow matplotlib

# Import libraries
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.layers import GlobalAveragePooling2D, Dropout, Dense
import matplotlib.pyplot as plt

# Define ResNet-18
def ResNet18(input_shape=(224, 224, 3), num_classes=2):
    base_model = tf.keras.applications.ResNet50(
        include_top=False,
        weights='imagenet',
        input_shape=input_shape
    )
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dropout(0.5)(x)
```

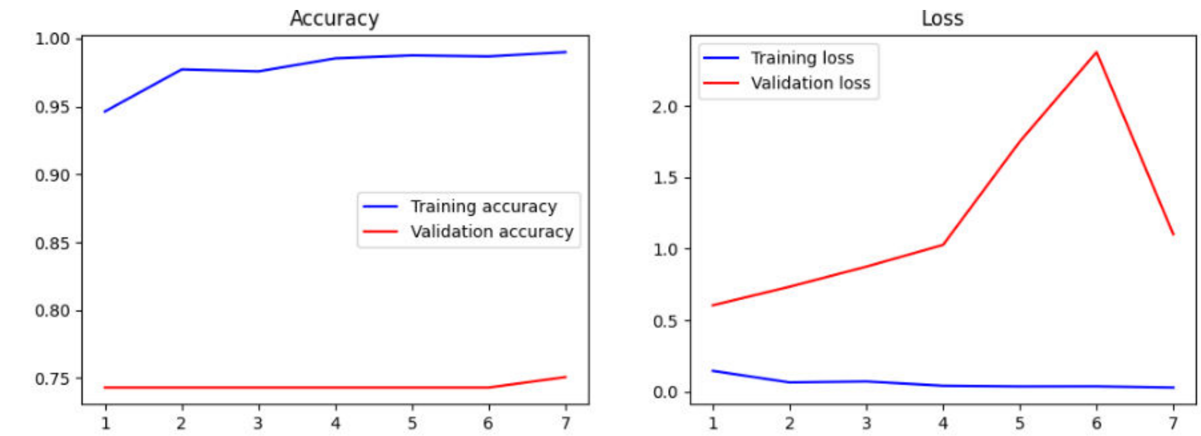
```python
    outputs = Dense(num_classes, activation='softmax')(x)
    model = models.Model(inputs=base_model.input, outputs=outputs)
    return model

# Load dataset
data_gen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1,
    shear_range=0.1,
    horizontal_flip=True,
    validation_split=0.2
)

train_gen = data_gen.flow_from_directory(
    '/kaggle/input/chest-xray-pneumonia/chest_xray/train',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)

val_gen = data_gen.flow_from_directory(
    '/kaggle/input/chest-xray-pneumonia/chest_xray/train',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)

test_gen = data_gen.flow_from_directory(
    '/kaggle/input/chest-xray-pneumonia/chest_xray/test',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical'
)

# Define and compile the model
input_shape = (224, 224, 3)
num_classes = len(train_gen.class_indices)
model = ResNet18(input_shape=input_shape, num_classes=num_classes)

model.compile(
    optimizer=Adam(learning_rate=1e-4),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Train the model
callbacks = [
    EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True),
    ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5)
```

```
]

history = model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=7,
    callbacks=callbacks
)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_gen)
print(f"Test Accuracy: {test_acc * 100:.2f}%")

# Visualize training performance
def plot_history(history):
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs = range(1, len(acc) + 1)
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(epochs, acc, 'b', label='Training accuracy')
    plt.plot(epochs, val_acc, 'r', label='Validation accuracy')
    plt.title('Accuracy')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(epochs, loss, 'b', label='Training loss')
    plt.plot(epochs, val_loss, 'r', label='Validation loss')
    plt.title('Loss')
    plt.legend()
    plt.show()

plot_history(history)
```

Output:

```
# Save the model
model.save('/kaggle/working/resnet18_pneumonia_model.h5')
```

```
Epoch 1/7
131/131 ──────────────────── 172s 942ms/step - accuracy: 0.9007 - loss: 0.2349 - val_accuracy: 0.7430 - val_loss: 0.6043 - learning_rate: 1.0000e-04
Epoch 2/7
131/131 ──────────────────── 88s 640ms/step - accuracy: 0.9735 - loss: 0.0695 - val_accuracy: 0.7430 - val_loss: 0.7342 - learning_rate: 1.0000e-04
Epoch 3/7
131/131 ──────────────────── 87s 634ms/step - accuracy: 0.9804 - loss: 0.0548 - val_accuracy: 0.7430 - val_loss: 0.8751 - learning_rate: 1.0000e-04
Epoch 4/7
131/131 ──────────────────── 87s 629ms/step - accuracy: 0.9877 - loss: 0.0355 - val_accuracy: 0.7430 - val_loss: 1.0275 - learning_rate: 1.0000e-04
Epoch 5/7
131/131 ──────────────────── 88s 643ms/step - accuracy: 0.9873 - loss: 0.0384 - val_accuracy: 0.7430 - val_loss: 1.7520 - learning_rate: 1.0000e-04
Epoch 6/7
131/131 ──────────────────── 88s 641ms/step - accuracy: 0.9885 - loss: 0.0308 - val_accuracy: 0.7430 - val_loss: 2.3789 - learning_rate: 1.0000e-04
Epoch 7/7
131/131 ──────────────────── 88s 638ms/step - accuracy: 0.9871 - loss: 0.0326 - val_accuracy: 0.7507 - val_loss: 1.1022 - learning_rate: 1.0000e-05
20/20 ──────────────────── 14s 696ms/step - accuracy: 0.6100 - loss: 0.7730
Test Accuracy: 62.50%
```



Output:

Result:

Experiment 8: Implement an image classification model using transfer learning techniques and check accuracy

Aim: Implement an image classification model using transfer learning techniques and check accuracy.Tune the required hyperparameters.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:
Transfer learning is the transfer of the knowledge (feature maps) that the network has acquired from one task, where we have a large amount of data, to a new task where data is not abundantly available.

- It is generally used where a neural network model is first trained on a problem similar to the problem that is being solved.
- One or more layers from the trained model are then used in a new model trained on the problem of interest.
- In transfer learning, we first train a base network on a base dataset and task, and then we repurpose the learned features, or transfer them to a second target network to be trained on a target dataset and task. This process will tend to work if the features are general, meaning suitable to both base and target tasks, instead of specific to the base task.

Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.applications import VGG16
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.preprocessing import image
from tensorflow.keras.utils import to_categorical
import numpy as np
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
# Convert labels to one-hot encoding
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=(32, 32,3))
# Freeze the layers of the pre-trained model to avoid retraining them
base_model.trainable = False
model = models.Sequential([
base_model, # Add the pre-trained base VGG16 model
```

```
layers.Flatten(),
layers.Dense(256, activation='relu'),
layers.Dropout(0.5), # Dropout to avoid overfitting
layers.Dense(10, activation='softmax') # 10 output classes for CIFAR-10
])
model.compile(optimizer=optimizers.Adam(learning_rate=0.0001), #
Small␣learning rate
loss='categorical_crossentropy',
metrics=['accuracy'])
```

Output:

Epoch 25/25
391/391 ━━━━━━━━━━━━━━━━━━━━ **10s** 17ms/step - accuracy: 0.5940 - loss: 1.1730 - val_accuracy: 0.5885 - val_loss: 1.1709

```
history = model.fit(x_train, y_train, batch_size=128,
epochs=25,validation_data=(x_test, y_test), verbose=1)
```

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"Test accuracy: {test_acc:.4f}")
```

output

```
313/313 - 4s - 12ms/step - accuracy: 0.5885 - loss: 1.1709
Test accuracy: 0.5885
```

```
import matplotlib.pyplot as plt
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
base_model.summary()
```


```
model.summary()
```

output:

Model: "sequential"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 1, 1, 512)	14,714,688
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 256)	131,328
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2,570

Total params: 15,116,384 (57.66 MB)
Trainable params: 133,898 (523.04 KB)
Non-trainable params: 14,714,688 (56.13 MB)
Optimizer params: 267,798 (1.02 MB)

Result:

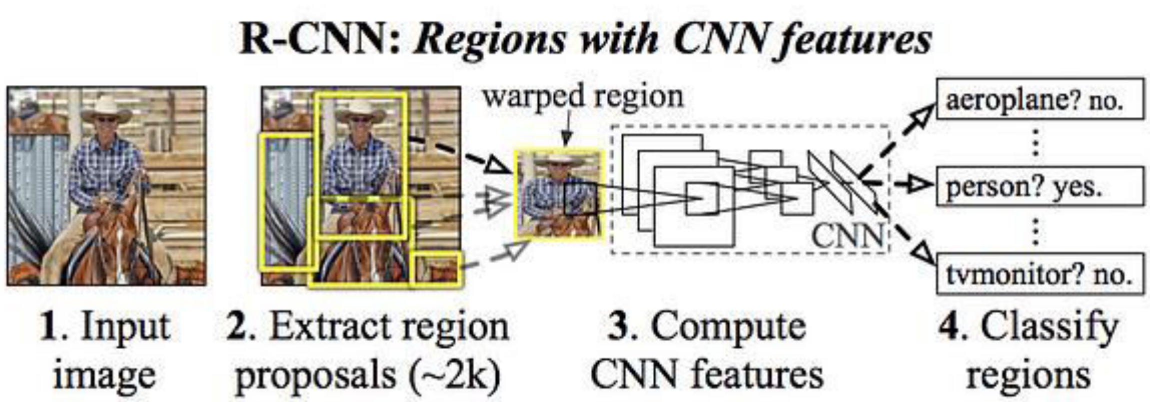
Experiment 9: Implement R-CNN model for object detection. Check with Fast and Faster R-CNN models.

Aim: Implement R-CNN model for object detection. Check with Fast and Faster R-CNN models.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

R-CNN (Region-based Convolutional Neural Network) is a deep learning framework for object detection that combines region proposal methods with convolutional neural networks. It works by first using selective search to generate potential object regions (region proposals) from an input image. Each proposal is then resized and passed through a CNN to extract features, which are subsequently fed into a classifier (such as SVM) to identify the object category. Additionally, a regression model refines the bounding box coordinates for more accurate localization. While effective, R-CNN is computationally expensive due to its multi-step process, requiring separate training for the CNN, classifier, and regression model, as well as extensive computation for every region proposal.



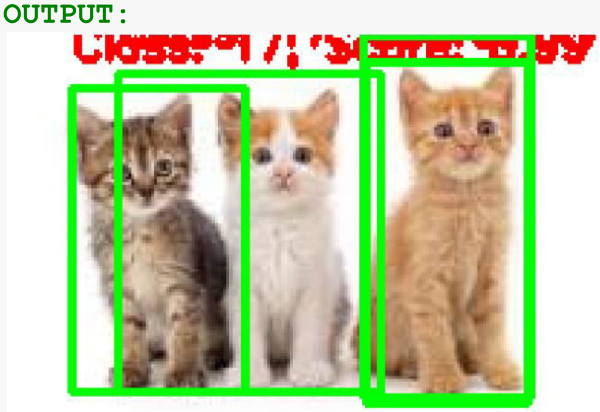
Code:

```
# Install the necessary libraries
!pip install torch torchvision

# Import required libraries
import torch
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.transforms import functional as F
```

```python
from PIL import Image
import matplotlib.pyplot as plt
import torchvision.transforms as T
import cv2
import numpy as np
from google.colab import files

# Load the Faster R-CNN model pretrained on COCO dataset
model = fasterrcnn_resnet50_fpn(pretrained=True)
model.eval()  # Set the model to evaluation mode

# Upload an image to perform detection
print("Please upload an image for detection:")
uploaded = files.upload()  # User uploads an image
image_path = list(uploaded.keys())[0]  # Get the uploaded image path

# Open the uploaded image
image = Image.open(image_path).convert("RGB")

# Preprocess the image: Convert to tensor as the model requires
transform = T.Compose([
    T.ToTensor()  # Converts PIL image to PyTorch tensor
])
image_tensor = transform(image)

# Perform object detection
with torch.no_grad():  # Disable gradient calculations for faster
processing
    predictions = model([image_tensor])  # Model inference

# Extract predictions
boxes = predictions[0]['boxes']  # Bounding boxes for detected objects
labels = predictions[0]['labels']  # Class labels for detected objects
scores = predictions[0]['scores']  # Confidence scores for detected
objects

# Convert the image to a NumPy array for visualization
image_np = np.array(image)

# Set a confidence threshold for displaying detections
```

```
confidence_threshold = 0.5

# Draw bounding boxes and class labels on the image
for i, box in enumerate(boxes):
    if scores[i] > confidence_threshold:  # Filter results based on
confidence score
        x1, y1, x2, y2 = map(int, box)  # Extract coordinates of the
bounding box
        label = labels[i].item()  # Extract the label index
        score = scores[i].item()  # Extract the confidence score

        # Draw a rectangle around the detected object
        cv2.rectangle(image_np, (x1, y1), (x2, y2), (0, 255, 0), 2)

        # Add a label with the confidence score
        text = f"Class: {label}, Score: {score:.2f}"
        cv2.putText(image_np, text, (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)

# Display the resulting image with detections
plt.figure(figsize=(12, 8))
plt.imshow(image_np)
plt.axis("off")
plt.title("Object Detection Results")
plt.show()
```

OUTPUT:

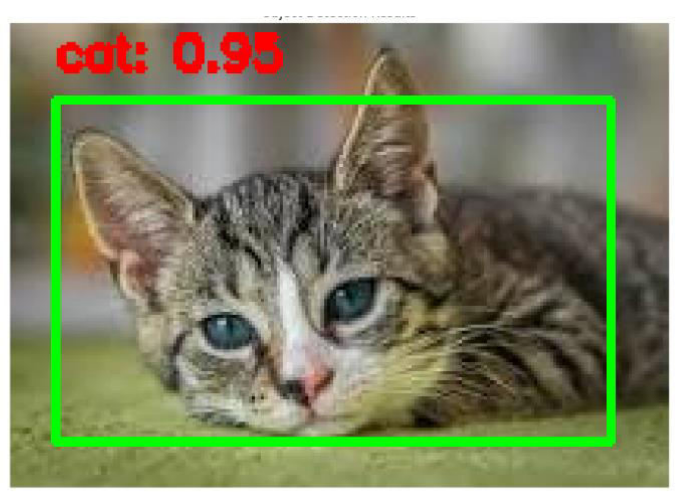


```python
# Install the necessary libraries
!pip install torch torchvision

# Import required libraries
import torch
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.transforms import functional as F
from PIL import Image
import matplotlib.pyplot as plt
import torchvision.transforms as T
import cv2
import numpy as np
from google.colab import files

# COCO class labels (index 0 is reserved for background, COCO classes
start from index 1)
COCO_CLASSES = [
    "__background__", "person", "bicycle", "car", "motorcycle",
"airplane", "bus",
    "train", "truck", "boat", "traffic light", "fire hydrant", "N/A",
"stop sign",
    "parking meter", "bench", "bird", "cat", "dog", "horse", "sheep",
"cow", "elephant",
    "bear", "zebra", "giraffe", "N/A", "backpack", "umbrella", "N/A",
"N/A", "handbag",
    "tie", "suitcase", "frisbee", "skis", "snowboard", "sports ball",
"kite", "baseball bat",
    "baseball glove", "skateboard", "surfboard", "tennis racket",
"bottle", "N/A", "wine glass",
    "cup", "fork", "knife", "spoon", "bowl", "banana", "apple",
"sandwich", "orange", "broccoli",
    "carrot", "hot dog", "pizza", "donut", "cake", "chair", "couch",
"potted plant", "bed", "N/A",
    "dining table", "N/A", "N/A", "toilet", "N/A", "TV", "laptop",
"mouse", "remote", "keyboard",
    "cell phone", "microwave", "oven", "toaster", "sink",
"refrigerator", "N/A", "book", "clock",
    "vase", "scissors", "teddy bear", "hair drier", "toothbrush"
]
```

```
# Load the Faster R-CNN model pretrained on COCO dataset
model = fasterrcnn_resnet50_fpn(pretrained=True)
model.eval()  # Set the model to evaluation mode

# Upload an image to perform detection
print("Please upload an image for detection:")
uploaded = files.upload()  # User uploads an image
image_path = list(uploaded.keys())[0]  # Get the uploaded image path

# Open the uploaded image
image = Image.open(image_path).convert("RGB")

# Preprocess the image: Convert to tensor as the model requires
transform = T.Compose([
    T.ToTensor()  # Converts PIL image to PyTorch tensor
])
image_tensor = transform(image)

# Perform object detection
with torch.no_grad():  # Disable gradient calculations for faster
processing
    predictions = model([image_tensor])  # Model inference

# Extract predictions
boxes = predictions[0]['boxes']  # Bounding boxes for detected objects
labels = predictions[0]['labels']  # Class labels for detected objects
scores = predictions[0]['scores']  # Confidence scores for detected
objects

# Convert the image to a NumPy array for visualization
image_np = np.array(image)

# Set a confidence threshold for displaying detections
confidence_threshold = 0.5

# Draw bounding boxes and class labels on the image
for i, box in enumerate(boxes):
    if scores[i] > confidence_threshold:  # Filter results based on
confidence score
```

```
        x1, y1, x2, y2 = map(int, box)  # Extract coordinates of the
bounding box
        label_index = labels[i].item()  # Extract the label index
        label_name = COCO_CLASSES[label_index]  # Get the label name
from COCO classes
        score = scores[i].item()  # Extract the confidence score
        # Draw a rectangle around the detected object
        cv2.rectangle(image_np, (x1, y1), (x2, y2), (0, 255, 0), 2)

        # Add a label with the class name and confidence score
        text = f"{label_name}: {score:.2f}"
        cv2.putText(image_np, text, (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)

# Display the resulting image with detections
plt.figure(figsize=(12, 8))
plt.imshow(image_np)
plt.axis("off")
plt.title("Object Detection Results")
plt.show()
```

Output:



Result:

Experiment 10: Implement a model to mask various categories with Semantic Segmentation

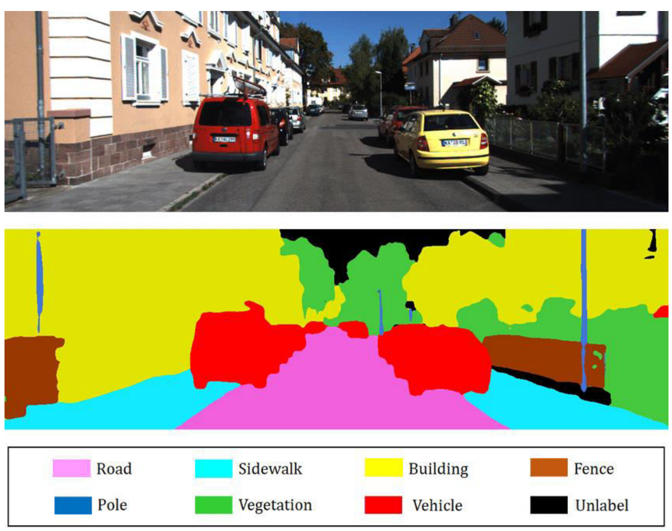
Aim: Implement a model to mask various categories with Semantic Segmentation

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

Semantic segmentation is a computer vision technique that uses deep learning to label each pixel in an image with a class:

How it works: Semantic segmentation uses image classification models to label pixels based on their semantic features, such as color, placement, or contrast. The result is a colorized map of the image, where each pixel color represents a different class label.



Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

x_train = x_train / 255.0
x_test = x_test / 255.0
```

```python
y_train_seg = (x_train.mean(axis=-1) > 0.5).astype(int)
y_test_seg = (x_test.mean(axis=-1) > 0.5).astype(int)

y_train_seg = y_train_seg[:, :, :, np.newaxis]
y_test_seg = y_test_seg[:, :, :, np.newaxis]
from tensorflow.keras import Model, Input
from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D,
concatenate

def unet_model(input_size=(32, 32, 3)):
    inputs = Input(input_size)

    # Downsampling
    c1 = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(64, (3, 3), activation='relu', padding='same')(p1)
    p2 = MaxPooling2D((2, 2))(c2)

    # Bottleneck
    c3 = Conv2D(128, (3, 3), activation='relu', padding='same')(p2)

    # Upsampling
    u1 = UpSampling2D((2, 2))(c3)
    m1 = concatenate([u1, c2])
    c4 = Conv2D(64, (3, 3), activation='relu', padding='same')(m1)

    u2 = UpSampling2D((2, 2))(c4)
    m2 = concatenate([u2, c1])
    c5 = Conv2D(32, (3, 3), activation='relu', padding='same')(m2)

    outputs = Conv2D(1, (1, 1), activation='sigmoid')(c5)

    return Model(inputs, outputs)

# Compile the model
model = unet_model()
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
```

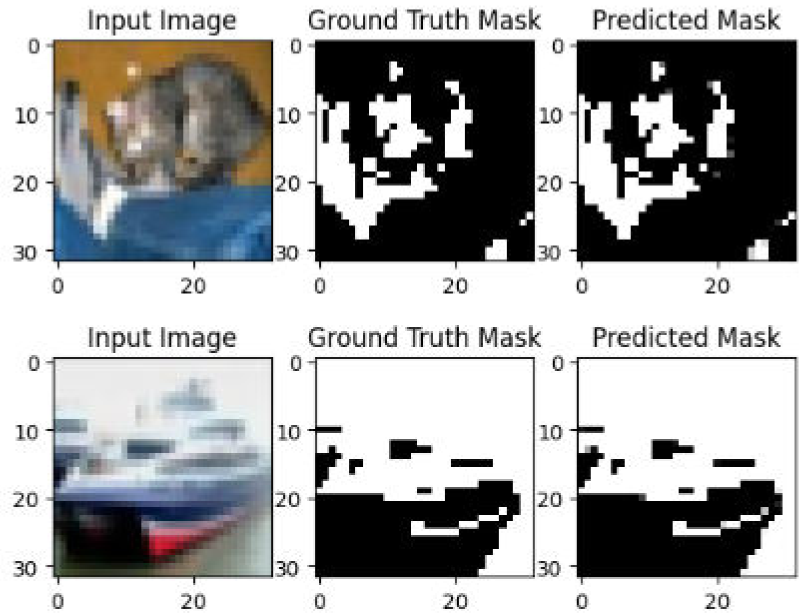
```python
# Train the model
model.fit(x_train, y_train_seg, validation_data=(x_test, y_test_seg),
epochs=10, batch_size=32)
```

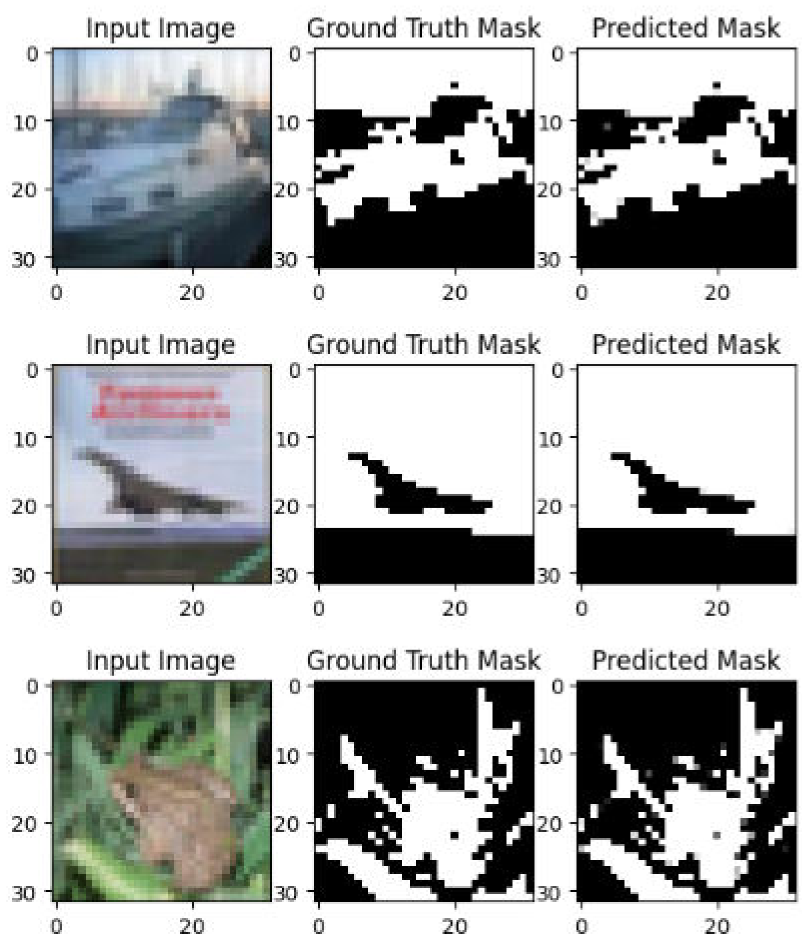
OUTPUT:

```
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy:
0.9981 - loss: 0.0060 - val_accuracy: 0.9990 - val_loss: 0.0049
```

```python
import matplotlib.pyplot as plt

pred = model.predict(x_test[:5])

# Display images and masks
for i in range(5):
    plt.subplot(1, 3, 1)
    plt.title("Input Image")
    plt.imshow(x_test[i])

    plt.subplot(1, 3, 2)
    plt.title("Ground Truth Mask")
    plt.imshow(y_test_seg[i].squeeze(), cmap='gray')

    plt.subplot(1, 3, 3)
    plt.title("Predicted Mask")
    plt.imshow(pred[i].squeeze(), cmap='gray')
    plt.show()
```

OUTPUT:


Input Image
Ground Truth Mask
Predicted Mask
Input Image
Ground Truth Mask
Predicted Mask


Input Image
Ground Truth Mask
Predicted Mask
Input Image
Ground Truth Mask
Predicted Mask
Input Image
Ground Truth Mask
Predicted Mask

Result:

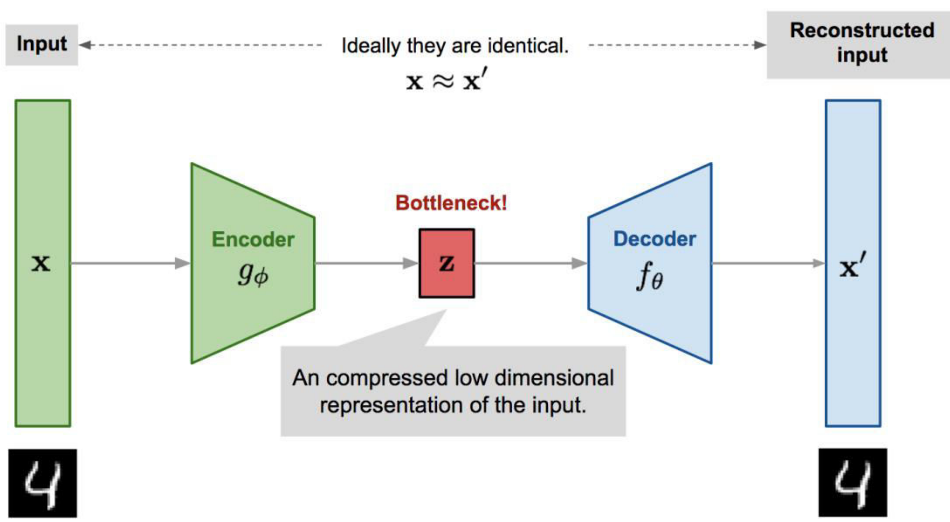
Experiment 11: Implement an Autoencoder to de-noise image.

Aim: Implement an Autoencoder to de-noise image.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

Autoencoders are a specialized class of algorithms that can learn efficient representations of input data with no need for labels. It is a class of artificial neural networks designed for unsupervised learning. Learning to compress and effectively represent input data without specific labels is the essential principle of an automatic decoder. This is accomplished using a two-fold structure that consists of an encoder and a decoder. The encoder transforms the input data into a reduced-dimensional representation, which is often referred to as "latent space" or "encoding". From that representation, a decoder rebuilds the initial input. For the network to gain meaningful patterns in data, a process of encoding and decoding facilitates the definition of essential features.



Code:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, Conv2DTranspose,
MaxPooling2D, UpSampling2D
from tensorflow.keras.optimizers import Adam
```

```
from tensorflow.keras.callbacks import EarlyStopping

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize images to [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Add random noise to the images
def add_noise(images, noise_factor=0.5):
    noisy_images = images + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=images.shape)
    noisy_images = np.clip(noisy_images, 0.0, 1.0)  # Ensure pixel
values stay between 0 and 1
    return noisy_images

# Add noise to the training and test data
x_train_noisy = add_noise(x_train)
x_test_noisy = add_noise(x_test)

# Reshape the data to add a channel dimension (for grayscale images, it
will be 1)
x_train_noisy = x_train_noisy.reshape((-1, 28, 28, 1))
x_test_noisy = x_test_noisy.reshape((-1, 28, 28, 1))
x_train = x_train.reshape((-1, 28, 28, 1))
x_test = x_test.reshape((-1, 28, 28, 1))

# Define the Autoencoder model
def build_autoencoder():
    # Encoder
    input_img = Input(shape=(28, 28, 1))
    x = Conv2D(32, (3, 3), activation='relu',
padding='same')(input_img)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    encoded = MaxPooling2D((2, 2), padding='same')(x)

    # Decoder
```

```
    x = Conv2DTranspose(128, (3, 3), activation='relu',
padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)
    x = Conv2DTranspose(64, (3, 3), activation='relu',
padding='same')(x)
    x = UpSampling2D((2, 2))(x)
    decoded = Conv2DTranspose(1, (3, 3), activation='sigmoid',
padding='same')(x)

    autoencoder = Model(input_img, decoded)
    autoencoder.compile(optimizer=Adam(), loss='binary_crossentropy')

    return autoencoder

# Build the autoencoder
autoencoder = build_autoencoder()

# Train the model
early_stop = EarlyStopping(monitor='val_loss', patience=3,
restore_best_weights=True)
autoencoder.fit(x_train_noisy, x_train, epochs=2, batch_size=256,
validation_data=(x_test_noisy, x_test), callbacks=[early_stop])

# Predict denoised images
denoised_images = autoencoder.predict(x_test_noisy)

# Visualize results
def visualize_denoising_results(noisy_images, denoised_images,
clean_images, num_images=10):
    plt.figure(figsize=(20, 4))
    for i in range(num_images):
        # Noisy image
        ax = plt.subplot(3, num_images, i + 1)
        plt.imshow(noisy_images[i].reshape(28, 28), cmap="gray")
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == 0:
            ax.set_title('Noisy Images')

        # Denoised image
```

```
        ax = plt.subplot(3, num_images, i + 1 + num_images)
        plt.imshow(denoised_images[i].reshape(28, 28), cmap="gray")
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == 0:
            ax.set_title('Denoised Images')

        # Original image
        ax = plt.subplot(3, num_images, i + 1 + num_images * 2)
        plt.imshow(clean_images[i].reshape(28, 28), cmap="gray")
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == 0:
            ax.set_title('Original Images')

    plt.show()

# Display the results
visualize_denoising_results(x_test_noisy, denoised_images, x_test)
```

Output:

Result:

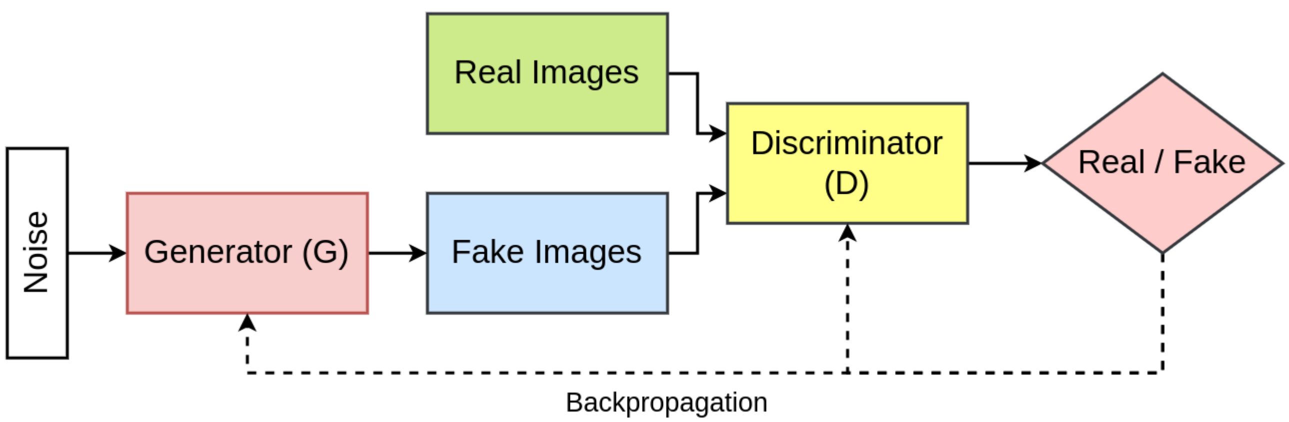
Experiment 12: Implement a GAN application to convert images.

Aim: Implement a GAN application to convert images.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory: Generative Adversarial Networks (GANs) are a powerful class of neural networks that are used for an [unsupervised learning](). GANs are made up of two [neural networks](), **a discriminator and a generator.** Generative Adversarial Networks (GANs) can be broken down into three parts:

- **Generative:** To learn a generative model, which describes how data is generated in terms of a probabilistic model.
- **Adversarial:** The word adversarial refers to setting one thing up against another. This means that, in the context of GANs, the generative result is compared with the actual images in the data set. A mechanism known as a discriminator is used to apply a model that attempts to distinguish between real and fake images.
- **Networks:** Use deep neural networks as artificial intelligence (AI) algorithms for training purposes.



Code:

```
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt

# Build Generator
def build_generator():
    model = tf.keras.Sequential([
```

```
        layers.Input(shape=(32, 32, 1)),
        layers.Conv2D(64, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(),
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Conv2DTranspose(128, kernel_size=4, strides=2,
padding="same"),
        layers.BatchNormalization(),
        layers.ReLU(),
        layers.Conv2DTranspose(3, kernel_size=4, strides=2,
padding="same", activation='tanh')
    ])
    return model

# Build Discriminator
def build_discriminator():
    model = tf.keras.Sequential([
        layers.Input(shape=(32, 32, 3)),
        layers.Conv2D(64, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(),
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Flatten(),
        layers.Dense(1, activation='sigmoid')
    ])
    return model

# Loss Functions
def discriminator_loss(real_output, fake_output):
    real_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True)(
        tf.ones_like(real_output), real_output)
    fake_loss = tf.keras.losses.BinaryCrossentropy(from_logits=True)(
        tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss

def generator_loss(fake_output):
    return tf.keras.losses.BinaryCrossentropy(from_logits=True)(
        tf.ones_like(fake_output), fake_output)
```

```python
# Optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Models
generator = build_generator()
discriminator = build_discriminator()

# Training Step
@tf.function
def train_step(real_images, gray_images):
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(gray_images, training=True)
        real_output = discriminator(real_images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
    gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discrimina
tor, discriminator.trainable_variables))
    return gen_loss, disc_loss

# Training Function
def train(dataset, epochs):
    for epoch in range(epochs):
        for real_images, gray_images in dataset:
            gen_loss, disc_loss = train_step(real_images, gray_images)
        print(f"Epoch {epoch+1}, Gen Loss: {gen_loss.numpy()}, Disc
Loss: {disc_loss.numpy()}")

# Dataset Preparation
(x_train, _), (_, _) = tf.keras.datasets.cifar10.load_data()
x_train = x_train.astype('float32') / 127.5 - 1
x_train_gray = tf.image.rgb_to_grayscale(x_train)
```

```
BATCH_SIZE = 64
BUFFER_SIZE = 10000
dataset = tf.data.Dataset.from_tensor_slices((x_train, x_train_gray))
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

# Train the Model
EPOCHS = 15
train(dataset, EPOCHS)

# Visualization
def generate_and_show(generator, gray_images, real_images):
    generated_images = generator(gray_images, training=False)
    plt.figure(figsize=(10, 5))
    for i in range(5):
        # Grayscale input
        plt.subplot(3, 5, i+1)
        plt.imshow(tf.squeeze(gray_images[i]) * 0.5 + 0.5, cmap='gray')
        plt.axis('off')
        # Generated image
        plt.subplot(3, 5, i+6)
        plt.imshow((generated_images[i] * 0.5 + 0.5).numpy())
        plt.axis('off')
        # Real image
        plt.subplot(3, 5, i+11)
        plt.imshow((real_images[i] * 0.5 + 0.5).numpy())
        plt.axis('off')
    plt.show()

# Take a sample of grayscale and real images
sample_gray = tf.expand_dims(x_train_gray[:5], axis=-1)  # Ensure shape
is (batch_size, 32, 32, 1)
sample_real = x_train[:5]  # Shape (batch_size, 32, 32, 3)

# Convert to float32 and ensure normalization
sample_gray = tf.convert_to_tensor(sample_gray, dtype=tf.float32)
sample_real = tf.convert_to_tensor(sample_real, dtype=tf.float32)

# Generate and show images
generate_and_show(generator, sample_gray, sample_real)
```

Output:

Result: