Practical_1

TensorFlow:

TensorFlow is an open-source deep learning framework developed by Google.

It provides a flexible and efficient ecosystem for building and training machine learning models.

TensorFlow allows you to define and execute computational graphs, where nodes represent operations and edges represent data flow.

It supports various neural network architectures, including feedforward networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and more.

TensorFlow offers automatic differentiation, allowing you to compute gradients for optimizing models with gradient-based optimization algorithms.

It provides tools for distributed computing, model deployment, and productionizing machine learning applications.

tf.constant(value): Creates a constant tensor with a specified value.

tf.Variable(initial_value): Creates a mutable tensor variable.

tf.placeholder(dtype): Creates a placeholder tensor for feeding data into the graph.

tf.add(x, y): Adds two tensors element-wise.

tf.matmul(a, b): Performs matrix multiplication of two tensors.

tf.reduce_sum(input_tensor): Computes the sum of elements along specified axes.

tf.reduce_mean(input_tensor): Computes the mean of elements along specified axes.

tf.nn.relu(input_tensor): Applies the ReLU activation function element-wise.

tf.nn.softmax(logits): Computes the softmax activation function element-wise.

tf.argmax(input_tensor, axis): Returns the indices of the maximum values along a specified axis.

Tensors are multi-dimensional arrays with a uniform type (called a dtype). You can see all supported dtypes at tf.dtypes.DType.

If you're familiar with NumPy, tensors are (kind of) like np.arrays.

All tensors are immutable like Python numbers and strings: you can never update the contents of a tensor, only create a new one. Here is a "scalar" or "rank-0" tensor . A scalar contains a single value, and no "axes".

```
import tensorflow as tf
# This will be an int32 tensor by default; see "dtypes" below.
rank_0_tensor = tf.constant(4)
print(rank_0_tensor)
```

```
tf.Tensor(4, shape=(), dtype=int32)
```

:A "vector" or "rank-1" tensor is like a list of values. A vector has one axis:

```
# Let's make this a float tensor.
rank_1_tensor = tf.constant([2.0, 3.0, 4.0])
print(rank_1_tensor)
```

```
tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)
```

A "matrix" or "rank-2" tensor has two axes:

```
# If you want to be specific, you can set the dtype (see below) at creation time
rank_2_tensor = tf.constant([[1, 2],
                             [3, 4],
                             [5, 6]], dtype=tf.float16)
print(rank_2_tensor)
```

```
tf.Tensor(
[[1. 2.]
 [3. 4.]
 [5. 6.]], shape=(3, 2), dtype=float16)
```

Tensors may have more axes; here is a tensor with three axes:

```
# There can be an arbitrary number of
# axes (sometimes called "dimensions")
rank_3_tensor = tf.constant([
  [[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]],])

print(rank_3_tensor)
```

```
tf.Tensor(
[[[ 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]]], shape=(3, 2, 5), dtype=int32)
```

There are many ways you might visualize a tensor with more than two axes.

You can convert a tensor to a NumPy array either using np.array or the tensor.numpy method:

You can do basic math on tensors, including addition, element-wise multiplication, and matrix multiplication.

```
a = tf.constant([[1, 2],
                 [3, 4]])
b = tf.constant([[1, 1],
                 [1, 1]]) # Could have also said `tf.ones([2,2], dtype=tf.int32)`

print(tf.add(a, b), "\n")
print(tf.multiply(a, b), "\n")
print(tf.matmul(a, b), "\n")
print(a + b, "\n") # element-wise addition
print(a * b, "\n") # element-wise multiplication
print(a @ b, "\n") # matrix multiplication
```

```
tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[3 3]
 [7 7]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[3 3]
 [7 7]], shape=(2, 2), dtype=int32)
```

Tensors are used in all kinds of operations (or "Ops").

```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])
print(c)

# Find the largest value
```

```
print(tf.reduce_max(c))
# Find the index of the largest value
print(tf.math.argmax(c))
# Compute the softmax
print(tf.nn.softmax(c))
```

```
tf.Tensor(
[[ 4.  5.]
 [10.  1.]], shape=(2, 2), dtype=float32)
tf.Tensor(10.0, shape=(), dtype=float32)
tf.Tensor([1 0], shape=(2,), dtype=int64)
tf.Tensor(
[[2.6894143e-01 7.3105860e-01]
 [9.9987662e-01 1.2339458e-04]], shape=(2, 2), dtype=float32)
```

```
tf.convert_to_tensor([1,2,3])
```

```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([1, 2, 3], dtype=int32)>
```

```
tf.reduce_max([1,2,3])
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=3>
```

About shapes

Tensors have shapes. Some vocabulary:

Shape: The length (number of elements) of each of the axes of a tensor.

Rank: Number of tensor axes. A scalar has rank 0, a vector has rank 1, a matrix is rank 2.

Axis or Dimension: A particular dimension of a tensor.

Size: The total number of items in the tensor, the product of the shape vector's elements.

```
rank_4_tensor = tf.zeros([3, 2, 4, 5])
print(rank_4_tensor)
```

```
tf.Tensor(
[[[[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. 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. 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. 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. 0.]
   [0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0.]]]], shape=(3, 2, 4, 5), dtype=float32)
```

```
print("Type of every element:", rank_4_tensor.dtype)
print("Number of axes:", rank_4_tensor.ndim)
print("Shape of tensor:", rank_4_tensor.shape)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
print("Elements along the last axis of tensor:", rank_4_tensor.shape[-1])
print("Total number of elements (3*2*4*5): ", tf.size(rank_4_tensor).numpy())
```

```
Type of every element: <dtype: 'float32'>
Number of axes: 4
Shape of tensor: (3, 2, 4, 5)
Elements along axis 0 of tensor: 3
Elements along the last axis of tensor: 5
Total number of elements (3*2*4*5):  120
```

Indexing

Single-axis indexing TensorFlow follows standard Python indexing rules, similar to indexing a list or a string in Python, and the basic rules for NumPy indexing.

indexes start at 0

negative indices count backwards from the end colons, :, are used for slices: start:stop:step

```
rank_1_tensor = tf.constant([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
print(rank_1_tensor.numpy())
```

```
[ 0  1  1  2  3  5  8 13 21 34]
```

Indexing with a scalar removes the axis:

```
print("First:", rank_1_tensor[0].numpy())
print("Second:", rank_1_tensor[1].numpy())
print("Last:", rank_1_tensor[-1].numpy())
```

```
First: 0
Second: 1
Last: 34
```

Indexing with a : slice keeps the axis:

```
print("Everything:", rank_1_tensor[:].numpy())
print("Before 4:", rank_1_tensor[:4].numpy())
print("From 4 to the end:", rank_1_tensor[4:].numpy())
print("From 2, before 7:", rank_1_tensor[2:7].numpy())
print("Every other item:", rank_1_tensor[::2].numpy())
print("Reversed:", rank_1_tensor[::-1].numpy())
```

```
Everything: [ 0  1  1  2  3  5  8 13 21 34]
Before 4: [0 1 1 2]
From 4 to the end: [ 3  5  8 13 21 34]
From 2, before 7: [1 2 3 5 8]
Every other item: [ 0  1  3  8 21]
Reversed: [34 21 13  8  5  3  2  1  1  0]
```

Multi-axis indexing

Higher rank tensors are indexed by passing multiple indices.

The exact same rules as in the single-axis case apply to each axis independently.

```
print(rank_2_tensor.numpy())
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

```
# Pull out a single value from a 2-rank tensor
print(rank_2_tensor[1, 1].numpy())
```

```
4.0
```

You can index using any combination of integers and slices:

```
# Get row and column tensors
print("Second row:", rank_2_tensor[1, :].numpy())
print("Second column:", rank_2_tensor[:, 1].numpy())
print("Last row:", rank_2_tensor[-1, :].numpy())
```

```
print("First item in last column:", rank_2_tensor[0, -1].numpy())
print("Skip the first row:")
print(rank_2_tensor[1:, :].numpy(), "\n")
```

```
Second row: [3. 4.]
Second column: [2. 4. 6.]
Last row: [5. 6.]
First item in last column: 2.0
Skip the first row:
[[3. 4.]
 [5. 6.]]
```

Here is an example with a 3-axis tensor:

```
print(rank_3_tensor[:, :, 4])
```

```
tf.Tensor(
[[ 4  9]
 [14 19]
 [24 29]], shape=(3, 2), dtype=int32)
```

Manipulating Shapes

Reshaping a tensor is of great utility.

```
# Shape returns a `TensorShape` object that shows the size along each axis
x = tf.constant([[1], [2], [3]])
print(x.shape)
```

```
(3, 1)
```

```
# You can convert this object into a Python list, too
print(x.shape.as_list())
```

```
[3, 1]
```

You can reshape a tensor into a new shape. The tf.reshape operation is fast and cheap as the underlying data does not need to be duplicated.

```
# You can reshape a tensor to a new shape.
# Note that you're passing in a list
reshaped = tf.reshape(x, [1, 3])
print(x.shape)
print(reshaped.shape)
```

```
(3, 1)
(1, 3)
```

The data maintains its layout in memory and a new tensor is created, with the requested shape, pointing to the same data.

TensorFlow uses C-style "row-major" memory ordering, where incrementing the rightmost index corresponds to a single step in memory.

```
print(rank_3_tensor)
```

```
tf.Tensor(
[[[ 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]]], shape=(3, 2, 5), dtype=int32)
```

If you flatten a tensor you can see what order it is laid out in memory.

```
# A `-1` passed in the `shape` argument says "Whatever fits".
print(tf.reshape(rank_3_tensor, [-1]))
```

```
tf.Tensor(
[ 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], shape=(30,), dtype=int32)
```

Typically the only reasonable use of tf.reshape is to combine or split adjacent axes (or add/remove 1s).

For this 3x2x5 tensor, reshaping to (3x2)x5 or 3x(2x5) are both reasonable things to do, as the slices do not mix:

```
print(tf.reshape(rank_3_tensor, [3*2, 5]), "\n")
print(tf.reshape(rank_3_tensor, [3, -1]))
```

```
tf.Tensor(
[[ 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]], shape=(6, 5), dtype=int32)

tf.Tensor(
[[ 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]], shape=(3, 10), dtype=int32)
```

Reshaping will "work" for any new shape with the same total number of elements, but it will not do anything useful if you do not respect the order of the axes.

Swapping axes in tf.reshape does not work; you need tf.transpose for that.

```
# Bad examples: don't do this

# You can't reorder axes with reshape.
print(tf.reshape(rank_3_tensor, [2, 3, 5]), "\n")

# This is a mess
print(tf.reshape(rank_3_tensor, [5, 6]), "\n")

# This doesn't work at all
try:
  tf.reshape(rank_3_tensor, [7, -1])
except Exception as e:
  print(f"{type(e).__name__}: {e}")
```

```
tf.Tensor(
[[[ 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]]], shape=(2, 3, 5), dtype=int32)

tf.Tensor(
[[ 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]], shape=(5, 6), dtype=int32)

InvalidArgumentError: {{function_node __wrapped__Reshape_device_/job:localhost/replica:0/task:0/device:CPU:0}} Input to reshape is
```

More on DTypes To inspect a tf.Tensor's data type use the Tensor.dtype property.

When creating a tf.Tensor from a Python object you may optionally specify the datatype.

If you don't, TensorFlow chooses a datatype that can represent your data. TensorFlow converts Python integers to tf.int32 and Python floating point numbers to tf.float32.

Otherwise TensorFlow uses the same rules NumPy uses when converting to arrays.

You can cast from type to type.

```
the_f64_tensor = tf.constant([2.2, 3.3, 4.4], dtype=tf.float64)
the_f16_tensor = tf.cast(the_f64_tensor, dtype=tf.float16)
# Now, cast to an uint8 and lose the decimal precision
```

```
the_u8_tensor = tf.cast(the_f16_tensor, dtype=tf.uint8)
print(the_u8_tensor)
```

```
tf.Tensor([2 3 4], shape=(3,), dtype=uint8)
```

Broadcasting Broadcasting is a concept borrowed from the equivalent feature in NumPy.

In short, under certain conditions, smaller tensors are "stretched" automatically to fit larger tensors when running combined operations on them.

The simplest and most common case is when you attempt to multiply or add a tensor to a scalar. In that case, the scalar is broadcast to be the same shape as the other argument.

```
x = tf.constant([1, 2, 3])

y = tf.constant(2)
z = tf.constant([2, 2, 2])
# All of these are the same computation
print(tf.multiply(x, 2))
print(x * y)
print(x * z)
```

```
tf.Tensor([2 4 6], shape=(3,), dtype=int32)
tf.Tensor([2 4 6], shape=(3,), dtype=int32)
tf.Tensor([2 4 6], shape=(3,), dtype=int32)
```

Likewise, axes with length 1 can be stretched out to match the other arguments. Both arguments can be stretched in the same computation.

In this case a 3x1 matrix is element-wise multiplied by a 1x4 matrix to produce a 3x4 matrix. Note how the leading 1 is optional: The shape of y is 4

```
# These are the same computations
x = tf.reshape(x,[3,1])
y = tf.range(1, 5)
print(x, "\n")
print(y, "\n")
print(tf.multiply(x, y))
```

```
tf.Tensor(
[[1]
 [2]
 [3]], shape=(3, 1), dtype=int32)

tf.Tensor([1 2 3 4], shape=(4,), dtype=int32)

tf.Tensor(
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]], shape=(3, 4), dtype=int32)
```

Here is the same operation without broadcasting:

```
x_stretch = tf.constant([[1, 1, 1, 1],
                         [2, 2, 2, 2],
                         [3, 3, 3, 3]])

y_stretch = tf.constant([[1, 2, 3, 4],
                         [1, 2, 3, 4],
                         [1, 2, 3, 4]])

print(x_stretch * y_stretch)  # Again, operator overloading
```

```
tf.Tensor(
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]], shape=(3, 4), dtype=int32)
```

Most of the time, broadcasting is both time and space efficient, as the broadcast operation never materializes the expanded tensors in memory.

You see what broadcasting looks like using tf.broadcast_to.

```
print(tf.broadcast_to(tf.constant([1, 2, 3]), [3, 3]))
```

```
tf.Tensor(
[[1 2 3]
 [1 2 3]
 [1 2 3]], shape=(3, 3), dtype=int32)
```

String tensors tf.string is a dtype, which is to say you can represent data as strings (variable-length byte arrays) in tensors.

The strings are atomic and cannot be indexed the way Python strings are.

The length of the string is not one of the axes of the tensor. See tf.strings for functions to manipulate them.

Here is a scalar string tensor:

```
# Tensors can be strings, too here is a scalar string.
scalar_string_tensor = tf.constant("Gray wolf")
print(scalar_string_tensor)
```

```
tf.Tensor(b'Gray wolf', shape=(), dtype=string)
```

```
# If you have three string tensors of different lengths, this is OK.
tensor_of_strings = tf.constant(["Gray wolf",
                                 "Quick brown fox",
                                 "Lazy dog"])
# Note that the shape is (3,). The string length is not included.
print(tensor_of_strings)
```

```
tf.Tensor([b'Gray wolf' b'Quick brown fox' b'Lazy dog'], shape=(3,), dtype=string)
```

Practical_2

Write a program to train a sigmoid neuron using gradient descent to approximate the relationship between an input feature X and the corresponding target values Y.

Find the optimal weights (w) and bias (b) for the sigmoid neuron such that it minimizes the mean squared error (MSE) between its predictions and the true target values.

```
import numpy as np
X = [0.5,2.5]
Y = [0.2,0.9]

def f(x,w,b):
  return 1/(1+np.exp(-(w*x+b)))

def error(w,b):
  err = 0.0
  for x,y in zip(X,Y):
    fx = f(x,w,b)
    err += (fx-y)**2
  return 0.5*err

def grad_b(x,w,b,y):
  fx = f(x,w,b)
  return (fx-y)*fx*(1-fx)

def grad_w(x,w,b,y):
  fx = f(x,w,b)
  return (fx-y)*fx*(1-fx)*x

def do_gradient_descent():

  w,b,eta,max_epochs = -2,-2,1.0,1000

  for i in range(max_epochs):
    dw,db = 0,0
    for x,y in zip(X,Y):
      dw += grad_w(x,w,b,y)
      db += grad_b(x,w,b,y)

    w = w - eta*dw
    b = b - eta*db
    return w, b
updated_w, updated_b = do_gradient_descent()
print("Updated weights:", updated_w)
print("Updated bias:", updated_b)


```

```
Updated weights: -1.99450768078436
Updated bias: -1.9922888407847856
```

To implement a simple binary classifier using a sigmoid neuron with gradient descent.

```
import numpy as np

class SigmoidNeuron:
    def __init__(self, input_size):
        # Initialize weights and bias randomly
        self.weights = np.random.randn(input_size, 1)
        self.bias = np.random.randn()
```

```

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def predict(self, X):
        # Compute the linear combination of inputs and weights
        z = np.dot(X, self.weights) + self.bias

        # Apply the sigmoid activation function
        return self.sigmoid(z)

    def compute_loss(self, y_true, y_pred):
        # Binary Cross-Entropy Loss
        loss = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
        return loss

    def gradient_descent(self, X, y_true, learning_rate, num_epochs):
        for epoch in range(num_epochs):
            # Forward pass: predict the output
            y_pred = self.predict(X)

            # Compute the loss
            loss = self.compute_loss(y_true, y_pred)

            # Compute the gradients
            d_weights = np.dot(X.T, (y_pred - y_true)) / len(y_true)
            d_bias = np.sum(y_pred - y_true) / len(y_true)

            # Update the weights and bias
            self.weights -= learning_rate * d_weights
            self.bias -= learning_rate * d_bias

            # Print the loss for every 100 epochs (optional)
            if epoch % 100 == 0:
                print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Sample data for demonstration
X = np.array([[1.0, 2.0], [2.0, 3.0], [3.0, 4.0], [4.0, 5.0]])
y_true = np.array([[0], [0], [1], [1]])

# Create a SigmoidNeuron with 2 input features
sigmoid_neuron = SigmoidNeuron(input_size=2)

# Perform gradient descent to learn the weights and bias
learning_rate = 0.1
num_epochs = 1000
sigmoid_neuron.gradient_descent(X, y_true, learning_rate, num_epochs)

# Predict the output for new data
new_data = np.array([[5.0, 6.0], [1.0, 2.0]])
predictions = sigmoid_neuron.predict(new_data)
print("Predictions:", predictions)
```

```
Epoch 0, Loss: 1.4499
Epoch 100, Loss: 0.3356
Epoch 200, Loss: 0.2752
Epoch 300, Loss: 0.2366
Epoch 400, Loss: 0.2095
Epoch 500, Loss: 0.1892
Epoch 600, Loss: 0.1732
Epoch 700, Loss: 0.1603
Epoch 800, Loss: 0.1496
Epoch 900, Loss: 0.1405
Predictions: [[0.99881964]
 [0.02458607]]
```

```python
import numpy as np

class SigmoidNeuron:
    def __init__(self, input_size):
        # Initialize weights and bias randomly
        self.weights = np.random.randn(input_size, 1)
        self.bias = np.random.randn()

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def predict(self, X):
        # Compute the linear combination of inputs and weights
        z = np.dot(X, self.weights) + self.bias

        # Apply the sigmoid activation function
        return self.sigmoid(z)

    def compute_loss(self, y_true, y_pred):
        # Mean Squared Error Loss
        loss = np.mean((y_true - y_pred)**2)
        return loss

    def gradient_descent(self, X, y_true, learning_rate, num_epochs):
        for epoch in range(num_epochs):
            # Forward pass: predict the output
            y_pred = self.predict(X)

            # Compute the loss
            loss = self.compute_loss(y_true, y_pred)

            # Compute the gradients
            d_weights = np.dot(X.T, (y_pred - y_true)) / len(y_true)
            d_bias = np.sum(y_pred - y_true) / len(y_true)

            # Update the weights and bias
            self.weights -= learning_rate * d_weights
            self.bias -= learning_rate * d_bias

            # Print the loss for every 100 epochs (optional)
            if epoch % 100 == 0:
                print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Sample data for demonstration
X = np.array([[1.0, 2.0], [2.0, 3.0], [3.0, 4.0], [4.0, 5.0]])
y_true = np.array([[6.0], [7.0], [9.0], [10.0]])

# Create a SigmoidNeuron with 2 input features
sigmoid_neuron = SigmoidNeuron(input_size=2)

# Perform gradient descent to learn the weights and bias
learning_rate = 0.1
num_epochs = 1000
sigmoid_neuron.gradient_descent(X, y_true, learning_rate, num_epochs)

# Predict the output for new data
new_data = np.array([[3.0, 4.0], [4.0, 5.0]])
predictions = sigmoid_neuron.predict(new_data)
print("Predictions:", predictions)
```

```
Epoch 0, Loss: 51.5120
Epoch 100, Loss: 51.5000
Epoch 200, Loss: 51.5000
Epoch 300, Loss: 51.5000
```

```
Epoch 400, Loss: 51.5000
Epoch 500, Loss: 51.5000
Epoch 600, Loss: 51.5000
Epoch 700, Loss: 51.5000
Epoch 800, Loss: 51.5000
Epoch 900, Loss: 51.5000
Predictions: [[1.]
 [1.]]
```

```
import numpy as np

# Sample data for demonstration
X = np.array([[1.0], [2.0], [3.0], [4.0], [5.0]])
y = np.array([[5.0], [7.0], [9.0], [11.0], [13.0]])

# Add a bias term to X
X_b = np.c_[np.ones((5, 1)), X]

# Define the learning rate and number of iterations
learning_rate = 0.1
num_iterations = 1000

# Initialize the weights randomly
theta = np.random.randn(2, 1)

# Gradient Descent algorithm
for iteration in range(num_iterations):
    # Calculate the predicted values
    y_pred = X_b.dot(theta)

    # Calculate the error
    error = y_pred - y

    # Calculate the gradient
    gradient = X_b.T.dot(error) / len(y)

    # Update the weights using the gradient and learning rate
    theta -= learning_rate * gradient

# Print the final weights
print("Intercept (b):", theta[0][0])
print("Slope (m):", theta[1][0])
```

```
Intercept (b): 2.999999873216533
Slope (m): 2.000000035116987
```

```
import numpy as np

# Generate some sample data for demonstration
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Add a bias term to X
X_b = np.c_[np.ones((100, 1)), X]

# Define the learning rate and number of iterations
learning_rate = 0.1
num_iterations = 1000

# Initialize the weights randomly
theta = np.random.randn(2, 1)

# Gradient Descent algorithm
for iteration in range(num_iterations):
```

KAUSTUBH RAYKAR

```
    # Calculate the predicted values
    y_pred = X_b.dot(theta)

    # Calculate the error
    error = y_pred - y

    # Calculate the gradient
    gradient = X_b.T.dot(error) / len(y)

    # Update the weights using the gradient and learning rate
    theta -= learning_rate * gradient

# Print the final weights
print("Intercept:", theta[0][0])
print("Slope:", theta[1][0])
```

```
Intercept: 4.215096094633133
Slope: 2.7701134419877866
```

Practical_3

AND Logic Using Single Perceptron

```
import numpy as np

# Define the activation function (step function)
def step_function(x):
    return 1 if x >= 0 else 0

# Define the AND gate function
def AND_gate(x1, x2):
    # Define the weights and bias
    weights = np.array([0.5, 0.5])
    bias = -0.7

    # Calculate the weighted sum
    weighted_sum = np.dot(np.array([x1, x2]), weights) + bias

    # Apply the activation function
    output = step_function(weighted_sum)

    return output

# Test the AND gate function
inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
for x1, x2 in inputs:
    output = AND_gate(x1, x2)
    print(f"Input: ({x1}, {x2}), Output: {output}")
```

```
Input: (0, 0), Output: 0
Input: (0, 1), Output: 0
Input: (1, 0), Output: 0
Input: (1, 1), Output: 1
```

OR Logic Using Single Perceptron

```
import numpy as np

# Define the step function as the activation function
def step_function(x):
    return 1 if x >= 0 else 0

# Define the single perceptron function
def perceptron(inputs, weights, bias):
    weighted_sum = np.dot(inputs, weights) + bias
    output = step_function(weighted_sum)
    return output

# Define the OR gate function using a single perceptron
def OR_gate(x1, x2):
    inputs = np.array([x1, x2])
    weights = np.array([0.5, 0.5])
    bias = -0.2
    output = perceptron(inputs, weights, bias)
    return output

# Test the OR gate function
inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
for x1, x2 in inputs:
```

KAUSTUBH RAYKAR

```
    output = OR_gate(x1, x2)
    print(f"Input: ({x1}, {x2}), Output: {output}")
```

```
Input: (0, 0), Output: 0
Input: (0, 1), Output: 1
Input: (1, 0), Output: 1
Input: (1, 1), Output: 1
```

XOR logic using OR and NAND logic

```
def OR(x1, x2):
    # OR logic gate implementation
    weights = [0.5, 0.5]
    bias = -0.2
    summation = x1 * weights[0] + x2 * weights[1] + bias
    return 1 if summation > 0 else 0

def NAND(x1, x2):
    # NAND logic gate implementation
    weights = [-0.5, -0.5]
    bias = 0.7
    summation = x1 * weights[0] + x2 * weights[1] + bias
    return 1 if summation > 0 else 0

def XOR(x1, x2):
    # XOR logic gate implementation using OR and NAND gates
    or_result = OR(x1, x2)
    nand_result = NAND(x1, x2)
    return AND(nand_result, or_result)

def AND(x1, x2):
    # AND logic gate implementation (used for XOR implementation)
    weights = [0.5, 0.5]
    bias = -0.7
    summation = x1 * weights[0] + x2 * weights[1] + bias
    return 1 if summation > 0 else 0

# Testing the XOR gate
print("XOR Logic:")
print("0 XOR 0 =", XOR(0, 0))
print("0 XOR 1 =", XOR(0, 1))
print("1 XOR 0 =", XOR(1, 0))
print("1 XOR 1 =", XOR(1, 1))
```

```
XOR Logic:
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

XOR Logic using Two Layer Perceptrons

```

def perceptron(inputs, weights, bias):
    # Perceptron implementation
    summation = 0
    for i in range(len(inputs)):
        summation += inputs[i] * weights[i]
    summation += bias
    return 1 if summation > 0 else 0

def XOR(x1, x2):
    # XOR logic gate implementation using a two-layer perceptron
    # Define the weights and biases for the perceptrons in the first and second layers
    or_weights = [0.5, 0.5]
```

```
    or_bias = -0.2
    nand_weights = [-0.5, -0.5]
    nand_bias = 0.7
    and_weights = [0.5, 0.5]
    and_bias = -0.7
    hidden_weights = [0.5, 0.5]
    hidden_bias = -0.7

    # Calculate the outputs of the OR and NAND perceptrons
    or_output = perceptron([x1, x2], or_weights, or_bias)
    nand_output = perceptron([x1, x2], nand_weights, nand_bias)

    # Calculate the output of the hidden layer perceptron
    hidden_output = perceptron([or_output, nand_output], hidden_weights, hidden_bias)

    # Calculate the output of the XOR gate using the AND perceptron
    xor_output = perceptron([or_output, hidden_output], and_weights, and_bias)
    return xor_output

# Testing the XOR gate
print("XOR Logic:")
print("0 XOR 0 =", XOR(0, 0))
print("0 XOR 1 =", XOR(0, 1))
print("1 XOR 0 =", XOR(1, 0))
print("1 XOR 1 =", XOR(1, 1))
```

```
XOR Logic:
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

KAUSTUBH RAYKAR

Practical_4

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
from tensorflow.keras.callbacks import ModelCheckpoint, CSVLogger
import matplotlib.pyplot as plt
```

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import keras
import tensorflow as tf

from tensorflow import keras
from keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropou
from tensorflow.keras.layers import GlobalMaxPooling2D, MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers, optimizers
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import accuracy_score

import warnings
warnings.filterwarnings('ignore')

```

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

```
#preprocessing
x_train = x_train/255
x_test = x_test/255
y_train = to_categorical(y_train,10) #10 classes in  CIFAR-10
y_test = to_categorical(y_test,10)
```

```
# Building Base model
model = Sequential()
model.add(Conv2D(32,(4,4), input_shape = (32,32,3),activation = 'relu'))
model.add(MaxPooling2D(pool_size =(2,2)))
model.add(Conv2D(32,(4,4), input_shape = (32,32,3),activation = 'relu'))
model.add(MaxPooling2D(pool_size =(2,2)))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
model.compile(loss='categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
```

Filters — It refers to the number of filters to be applied in the convolution. Eg. 32 or 64.

Kernel_size — It refers to the length of the convolution window. Eg. (3,3) or (4,4).

Activation — It refers to the regularizer function. Eg. ReLU, Leaky ReLU, Tanh, Sigmoid.

Pooling or MaxPooling2D Layer: In this layer, we are scaling down the size of an image. We are keeping the size (2,2) for the pooling layer.

Flatten Layer: This layer converts the n-dimensional array to 1 dimensional array.

Dense Layer: This layer is a fully connected layer i.e.all the neurons in the current layer are connected to the next layer. For our model, we are setting the first dense layer with 128 neurons and the second dense layer with 10 neurons.

model.compile() function: This function is used to compile the model. Here, we are defining three parameters -

Loss function — It is used to evaluate how well our algorithm models the dataset. We can select options like 'Categorical cross entropy', 'Binary cross entropy', 'sparse categorical cross entropy' depending on our dataset.

Optimizer — With this we can change the attributes of a neural network like weights and learning rate. Here, we can choose from different optimizers like Adam, AdaDelta, SGD etc.

Metrics — It is used to understand the performance of our model. Eg. Accuracy, Mean Squared Error etc.

model.fit() function: This function is used to train our model which takes the training and test data to fit our model.

model.summary() function: This function is used to see all the parameters and shapes in all layers of our model.

Epochs — Number of times we are passing the complete dataset forward and backward through the neural network.

Verbose — options to view our output. Eg. verbose = 0 will print nothing, verbose = 1 will print the progress bar and one line per epoch while verbose = 2 will print one line per epoch.

```
model.summary()
history = model.fit(x_train, y_train, epochs = 20, verbose=1,validation_data=(x_test,y_test))
```

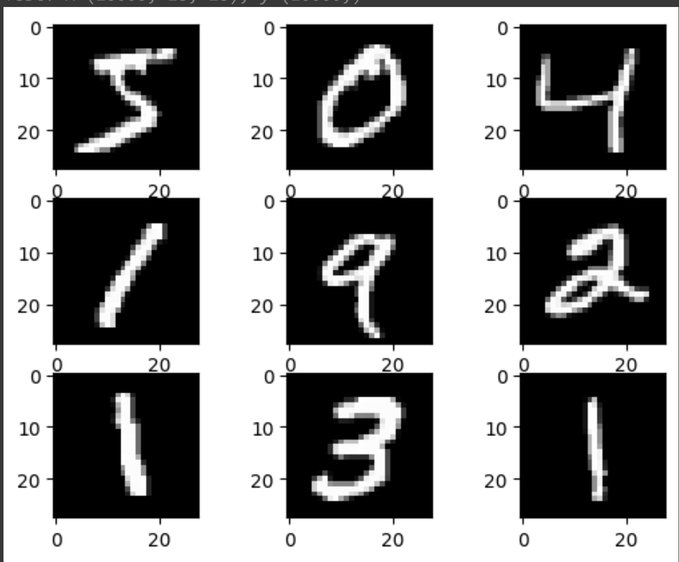
```

```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 29, 29, 32)        1568

 max_pooling2d_6 (MaxPoolin  (None, 14, 14, 32)        0
 g2D)

 conv2d_7 (Conv2D)           (None, 11, 11, 32)        16416

 max_pooling2d_7 (MaxPoolin  (None, 5, 5, 32)          0
 g2D)

 flatten_3 (Flatten)         (None, 800)               0

 dense_6 (Dense)             (None, 128)               102528
```

Practical_5

```
# example of loading the mnist dataset
from tensorflow.keras.datasets import mnist
from matplotlib import pyplot as plt
# load dataset
(trainX, trainy), (testX, testy) = mnist.load_data()
# summarize loaded dataset
print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
# plot first few images
for i in range(9):
  # define subplot
  plt.subplot(330 + 1 + i)
  # plot raw pixel data
  plt.imshow(trainX[i], cmap=plt.get_cmap('gray'))
# show the figure
plt.show()
```

```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```



```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras import layers, models

# Load and preprocess the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = tf.keras.utils.to_categorical(train_labels)
test_labels = tf.keras.utils.to_categorical(test_labels)

# Build the CNN model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

KAUSTUBH RAYKAR

```
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

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

# Training the model
history = model.fit(train_images, train_labels, epochs=10, batch_size=64,
                    validation_split=0.2)  # Use part of the training data as validation

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels)
print("Test accuracy:", test_acc)

# Make predictions on new images
#predictions = model.predict(new_images)
```

```
Epoch 1/10
750/750 [==============================] - 54s 71ms/step - loss: 0.2118 - accuracy: 0.9348 - val_loss: 0.0743 - val_accuracy: 0.977
Epoch 2/10
750/750 [==============================] - 53s 71ms/step - loss: 0.0567 - accuracy: 0.9820 - val_loss: 0.0515 - val_accuracy: 0.984
Epoch 3/10
750/750 [==============================] - 52s 70ms/step - loss: 0.0398 - accuracy: 0.9877 - val_loss: 0.0536 - val_accuracy: 0.983
Epoch 4/10
 39/750 [>.............................] - ETA: 1:01 - loss: 0.0240 - accuracy: 0.9928
```

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras import layers, models
(x_train, y_train), (x_test, y_test) = mnist.load_data()
assert x_train.shape == (60000, 28, 28)
assert x_test.shape == (10000, 28, 28)
assert y_train.shape == (60000,)
assert y_test.shape == (10000,)

```

Practical_6

NumPy (Numerical Python) is a fundamental package for scientific computing in Python.

It provides powerful tools for working with large, multi-dimensional arrays and matrices.

NumPy offers a wide range of mathematical functions to perform operations on arrays efficiently.

Key features include array manipulation, mathematical operations, linear algebra, Fourier transform, and random number generation.

NumPy is widely used as a foundational library in many scientific and data analysis packages.

```
import numpy as np

# Create a 1D array
arr_1d = np.array([1, 2, 3, 4, 5])

# Create a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Create an array of zeros
zeros_arr = np.zeros((3, 4))

# Create an array of ones
ones_arr = np.ones((2, 3))

# Create an array with a range of values
range_arr = np.arange(0, 10, 2)

# Create a random array
random_arr = np.random.random((3, 3))
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
addition = arr1 + arr2

# Element-wise multiplication
multiplication = arr1 * arr2

# Dot product
dot_product = np.dot(arr1, arr2)

# Transpose
arr = np.array([[1, 2], [3, 4]])
transpose_arr = arr.T

# Reshape
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape((2, 3))
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Accessing an element
element = arr[2]

# Slicing an array
slice_arr = arr[1:4]

# Updating array elements
arr[3] = 10

```

```
# Boolean indexing
bool_index = arr[arr > 3]

# Fancy indexing
fancy_index = arr[[1, 3, 4]]

arr = np.array([1, 2, 3, 4, 5])

# Sum of all elements
sum_arr = np.sum(arr)

# Mean of all elements
mean_arr = np.mean(arr)

# Maximum and minimum values
max_val = np.max(arr)
min_val = np.min(arr)

# Element-wise logarithm
log_arr = np.log(arr)

# Element-wise exponential
exp_arr = np.exp(arr)
arr = np.array([1, 2, 3, 4, 5])

# Sum of all elements
sum_arr = np.sum(arr)

# Mean of all elements
mean_arr = np.mean(arr)

# Maximum and minimum values
max_val = np.max(arr)
min_val = np.min(arr)

# Element-wise logarithm
log_arr = np.log(arr)

# Element-wise exponential
exp_arr = np.exp(arr)
```

```
import tensorflow as tf

tf.compat.v1.disable_eager_execution()

a = tf.constant(5)
b = tf.constant(6)
c = tf.constant(7)
d = tf.multiply(a,b)
e = tf.add(c,d)
f = tf.subtract(a,c)

with tf.compat.v1.Session() as sess:
  outs = sess.run(f)
  print(outs)
```

```
-2
```

```
import tensorflow as tf
tf.compat.v1.disable_eager_execution()
# Create a 1D tensor (vector)
a = tf.constant([1, 2, 3])
```

```

# Create a 2D tensor (matrix)
b = tf.constant([[1, 2, 3], [4, 5, 6]])
# Reshape a tensor
c = tf.reshape(b, [2, 3])
print(c)

# Perform element-wise multiplication
d =tf.multiply(a, c)

# Print the result
with tf.compat.v1.Session() as sess:
    result = sess.run(d)
    print(result)
```

```
Tensor("Reshape_11:0", shape=(2, 3), dtype=int32)
[[ 1  4  9]
 [ 4 10 18]]
```

```

```

1-D (Tensor Flow)

```
>>> import numpy as np
>>> tensor_1d = np.array([1.3, 1, 4.0, 23.99])
>>> print(tensor_1d)
```

```
[ 1.3   1.    4.   23.99]
```

The indexing of elements is same as Python lists. The first element starts with index of 0; to print the values through index, all you need to do is mention the index number.

```
print (tensor_1d[0])
print (tensor_1d[2])
```

```
1.3
4.0
```

Two dimensional Tensors Sequence of arrays are used for creating “two dimensional tensors”.

The creation of two-dimensional tensors is described below –

```
>>> import numpy as np
>>> tensor_2d = np.array([(1,2,3,4),(4,5,6,7),(8,9,10,11),(12,13,14,15)])
>>> print(tensor_2d)
```

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

The specific elements of two dimensional tensors can be tracked with the help of row number and column number specified as index numbers.

```
tensor_2d[0][2]
```

```
3
```

Tensor Handling and Manipulations we will learn about Tensor Handling and Manipulations.

To begin with, let us consider the following code

```
import tensorflow as tf
import numpy as np

matrix1 = np.array([(2,2,2),(2,2,2),(2,2,2)],dtype = 'int32')
matrix2 = np.array([(1,1,1),(1,1,1),(1,1,1)],dtype = 'int32')
```

KAUSTUBH RAYKAR

```

print (matrix1)
print (matrix2)

matrix1 = tf.constant(matrix1)
matrix2 = tf.constant(matrix2)
matrix_product = tf.matmul(matrix1, matrix2)
matrix_sum = tf.add(matrix1,matrix2)
matrix_3 = np.array([(2,7,2),(1,4,2),(9,0,2)],dtype = 'float32')
print (matrix_3)
```

```
[[2 2 2]
 [2 2 2]
 [2 2 2]]
[[1 1 1]
 [1 1 1]
 [1 1 1]]
[[2. 7. 2.]
 [1. 4. 2.]
 [9. 0. 2.]]
```

Activation Function in Tensor Flow

```
import tensorflow as tf

# ReLU activation
input_tensor = tf.constant([-2, -1, 0, 1, 2])
relu_tensor = tf.nn.relu(input_tensor)

# Softmax activation
logits = tf.constant([1.0, 2.0, 3.0])
softmax_tensor = tf.nn.softmax(logits)
```

Loss Function

```
import tensorflow as tf

# Mean squared error loss
labels = tf.constant([0.5, 0.7, 0.9])
predictions = tf.constant([0.2, 0.6, 0.8])
mse_loss = tf.losses.mean_squared_error(labels, predictions)

```

```
import tensorflow as tf

# Gradient descent optimizer
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)

# Adam optimizer
optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
```

Double-click (or enter) to edit

Practical_7

In this example, we create a feedforward neural network using the Keras Sequential API. The network consists of an input layer, a hidden layer, and an output layer. Each layer is defined using the Dense class from Keras.

We compile the model by specifying the loss function (binary cross-entropy), the optimizer (Adam), and the metrics to be evaluated during training (accuracy).

Next, we train the model using the fit() method. We provide the training data (X_train) and the corresponding labels (y_train), along with the number of epochs (1000) and verbose set to 0 for silent training.

After training, we evaluate the model's performance on the test data (X_test and y_test) and print the loss and accuracy.

Finally, we make predictions on the test data using the predict() method and print the predicted values.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Create a dataset for training the neural network
X_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_train = np.array([[0], [1], [1], [0]])
# Create a sequential model
model = Sequential()
# Add layers to the model
model.add(Dense(4, input_dim=2, activation='relu'))  # Input layer with
model.add(Dense(4, activation='relu'))  # Hidden layer with 4 neurons
model.add(Dense(1, activation='sigmoid'))  # Output layer with 1 neuron
# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc
# Train the model
model.fit(X_train, y_train, epochs=1000, verbose=0)

# Test the model
X_test = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_test = np.array([[0], [1], [1], [0]])
loss, accuracy = model.evaluate(X_test, y_test)
print("Test loss:", loss)
print("Test accuracy:", accuracy)

# Make predictions
predictions = model.predict(X_test)
print("Predictions:", predictions)
```

```
1/1 [==============================] - 0s 176ms/step - loss: 0.1024 - accuracy: 1.0000
Test loss: 0.1024220734834671
Test accuracy: 1.0
1/1 [==============================] - 0s 117ms/step
Predictions: [[0.27413878]
 [0.96845025]
 [0.96014494]
 [0.01642622]]
```

KAUSTUBH RAYKAR

Practical_8

Binary and Multi-class classification using feedforward NN classifier

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

# Load and preprocess the Fashion MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Define the model architecture
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)), #keras.layers.Input(784,),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, verbose=1)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print('Test accuracy:', test_acc)

# Make predictions
predictions = model.predict(x_test)
predicted_labels = np.argmax(predictions, axis=1)

# 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], cmap=plt.cm.binary)
    plt.xlabel(f"Predicted: {predicted_labels[i]}")
plt.show()

```

KAUSTUBH RAYKAR

```
Epoch 1/10
1875/1875 [==============================] - 11s 5ms/step - loss: 0.5021 - accurac
Epoch 2/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3812 - accuracy
Epoch 3/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3392 - accuracy
Epoch 4/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3142 - accuracy
Epoch 5/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2973 - accuracy
Epoch 6/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2821 - accuracy
Epoch 7/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2691 - accuracy
Epoch 8/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2592 - accuracy
Epoch 9/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2486 - accuracy
Epoch 10/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2398 - accuracy
Test accuracy: 0.8769000172615051
313/313 [==============================] - 1s 2ms/step
```

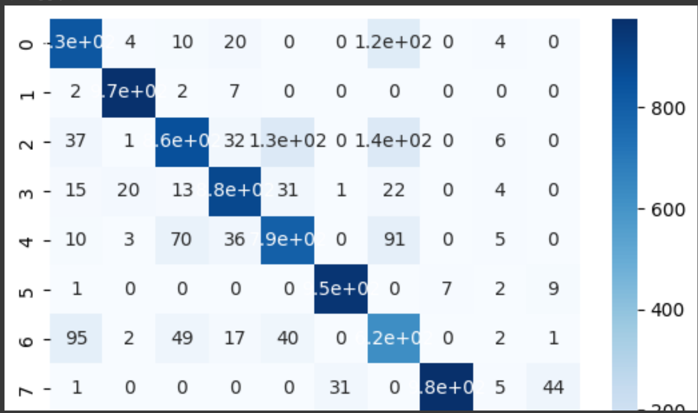

```
from sklearn.metrics import classification_report,confusion_matrix
print(classification_report(predicted_labels,y_test))
```

```
              precision    recall  f1-score   support

           0       0.83      0.84      0.84       984
           1       0.97      0.99      0.98       980
           2       0.85      0.71      0.78      1201
           3       0.89      0.89      0.89       991
           4       0.79      0.79      0.79      1009
           5       0.95      0.98      0.97       973
           6       0.62      0.75      0.68       828
           7       0.97      0.92      0.95      1056
           8       0.97      0.97      0.97      1002
           9       0.95      0.97      0.96       976

    accuracy                           0.88     10000
   macro avg       0.88      0.88      0.88     10000
weighted avg       0.88      0.88      0.88     10000
```

```
import seaborn as sns
sns.heatmap(confusion_matrix(predicted_labels,y_test),annot=True,cmap='Blues')
```

KAUSTUBH RAYKAR

<Axes: >



EEG Eye-state classification

Based on the above code template, perform binary classification for the EEG Eye-State detection.All data is from one continuous EEG measurement with the Emotiv EEG Neuroheadset. The duration of the measurement was 117 seconds. The eye state was detected via a camera during the EEG measurement and added later manually to the file after analyzing the video frames.

Target:

1 indicates the eye-closed and

0 the eye-open state.

All values are in chronological order with the first measured value at the top of the data.

Use performance metrics such as Accuracy, Precision, Recall, F1-score, Jaccard's index to address the classification framework.

Practical_9

To write a program to implement a basic feedforward neural network with backpropagation and gradient descent, covering activation functions, loss computation, weight updates, and prediction.

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

# Initialize the neural network parameters
input_size = 3
hidden_size = 3
output_size = 2
learning_rate = 0.1
epochs = 1000

# Initialize weights and biases for the network
np.random.seed(42)
weights_input_hidden1 = np.random.rand(input_size, hidden_size)
biases_hidden1 = np.zeros((1, hidden_size))
weights_hidden1_hidden2 = np.random.rand(hidden_size, hidden_size)
biases_hidden2 = np.zeros((1, hidden_size))
weights_hidden2_output = np.random.rand(hidden_size, output_size)
biases_output = np.zeros((1, output_size))

# Training data
X = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6],
              [0.7, 0.8, 0.9]])
y = np.array([[0, 1],
              [1, 0],
              [0, 1]])

# Training loop
for epoch in range(epochs):
    # Forward propagation
    hidden1_output = sigmoid(np.dot(X, weights_input_hidden1) + biases_hidden1)
    hidden2_output = sigmoid(np.dot(hidden1_output, weights_hidden1_hidden2) + biases_hidden2
    output_layer_input = np.dot(hidden2_output, weights_hidden2_output) + biases_output
    output_probabilities = softmax(output_layer_input)

    # Compute loss (cross-entropy)
    loss = -np.sum(y * np.log(output_probabilities))

    # Backpropagation
    output_error = output_probabilities - y
    hidden2_error = np.dot(output_error, weights_hidden2_output.T) * (hidden2_output * (1 - h
    hidden1_error = np.dot(hidden2_error, weights_hidden1_hidden2.T) * (hidden1_output * (1 -

    # Update weights and biases
    weights_hidden2_output -= learning_rate * np.dot(hidden2_output.T, output_error)
    biases_output -= learning_rate * np.sum(output_error, axis=0, keepdims=True)
    weights_hidden1_hidden2 -= learning_rate * np.dot(hidden1_output.T, hidden2_error)
    biases_hidden2 -= learning_rate * np.sum(hidden2_error, axis=0, keepdims=True)
    weights_input_hidden1 -= learning_rate * np.dot(X.T, hidden1_error)
    biases_hidden1 -= learning_rate * np.sum(hidden1_error, axis=0, keepdims=True)
```

KAUSTUBH RAYKAR

```
    biases_hidden1 -= learning_rate * np.sum(hidden1_error, axis=0, keepdims=True)

    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {loss}")

print("Training completed!")

# Make predictions
test_input = np.array([[0.2, 0.3, 0.4]])
hidden1_pred = sigmoid(np.dot(test_input, weights_input_hidden1) + biases_hidden1)
hidden2_pred = sigmoid(np.dot(hidden1_pred, weights_hidden1_hidden2) + biases_hidden2)
output_pred = softmax(np.dot(hidden2_pred, weights_hidden2_output) + biases_output)
print("Predicted probabilities:", output_pred)
```

```
Epoch 0, Loss: 2.2849849866820864
Epoch 100, Loss: 1.9103866552908952
Epoch 200, Loss: 1.9102424137845517
Epoch 300, Loss: 1.9101091236093417
Epoch 400, Loss: 1.9099845617024762
Epoch 500, Loss: 1.9098667829106595
Epoch 600, Loss: 1.9097540534491375
Epoch 700, Loss: 1.9096447953004272
Epoch 800, Loss: 1.9095375385160285
Epoch 900, Loss: 1.909430878984222
Training completed!
Predicted probabilities: [[0.33375329 0.66624671]]
```

Practical_10

```
import numpy as np

# Define sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define softmax activation function
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

# Define cross-entropy loss function
def cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.sum(y_true * np.log(y_pred))

# Generate a probabilistic dataset
np.random.seed(42)
num_samples = 1000
input_data = np.random.rand(num_samples, 4)
target_output = np.random.randint(2, size=(num_samples, 1))  # Binary target labels

# Add more input-output pairs (modify as needed)
additional_input_outputs = [
    (np.array([0.7, 0.4, 0.8, 0.2]), np.array([0])),
    (np.array([0.3, 0.6, 0.1, 0.9]), np.array([1])),
    (np.array([0.9, 0.1, 0.5, 0.6]), np.array([1])),
    (np.array([0.2, 0.3, 0.4, 0.5]), np.array([0]))
]

input_outputs = list(additional_input_outputs)

# Neural network architecture
input_size = 4
hidden1_size = 8
hidden2_size = 6
output_size = 1

# Initialize random weights and biases
np.random.seed(42)
weights_input_hidden1 = np.random.rand(input_size, hidden1_size)
biases_hidden1 = np.random.rand(1, hidden1_size)
weights_hidden1_hidden2 = np.random.rand(hidden1_size, hidden2_size)
biases_hidden2 = np.random.rand(1, hidden2_size)
weights_hidden2_output = np.random.rand(hidden2_size, output_size)
biases_output = np.random.rand(1, output_size)

# Hyperparameters
learning_rate = 0.1
epochs = 10000

# Training loop
for epoch in range(epochs):
    total_loss = 0
    for i in range(num_samples + len(additional_input_outputs)):
```

KAUSTUBH RAYKAR

```
    for i in range(num_samples + len(additional_input_outputs)):
        # Forward propagation
        input_data_i = input_data[i:i+1] if i < num_samples else additional_input_outputs[i -
        target_output_i = target_output[i:i+1] if i < num_samples else additional_input_outpu
        hidden1_input = np.dot(input_data_i, weights_input_hidden1) + biases_hidden1
        hidden1_output = sigmoid(hidden1_input)
        hidden2_input = np.dot(hidden1_output, weights_hidden1_hidden2) + biases_hidden2
        hidden2_output = sigmoid(hidden2_input)
        output_input = np.dot(hidden2_output, weights_hidden2_output) + biases_output
        predicted_output = softmax(output_input)

        # Convert target label to one-hot encoding
        target_onehot = np.zeros((1, 2))
        target_onehot[0, target_output_i[0, 0]] = 1

        # Calculate loss
        loss = cross_entropy(target_onehot, predicted_output)
        total_loss += loss

        # Backpropagation
        output_error = target_onehot - predicted_output
        hidden2_error = output_error.dot(weights_hidden2_output.T)
        hidden2_delta = hidden2_error * sigmoid_derivative(hidden2_output)
        hidden1_error = hidden2_delta.dot(weights_hidden1_hidden2.T)
        hidden1_delta = hidden1_error * sigmoid_derivative(hidden1_output)

        # Update weights and biases using gradient descent
        weights_hidden2_output += hidden2_output.T.dot(output_error) * learning_rate
        biases_output += np.sum(output_error, axis=0, keepdims=True) * learning_rate
        weights_hidden1_hidden2 += hidden1_output.T.dot(hidden2_delta) * learning_rate
        biases_hidden2 += np.sum(hidden2_delta, axis=0, keepdims=True) * learning_rate
        weights_input_hidden1 += input_data_i.T.dot(hidden1_delta) * learning_rate
        biases_hidden1 += np.sum(hidden1_delta, axis=0, keepdims=True) * learning_rate

    # Print average loss for the epoch
    avg_loss = total_loss / (num_samples + len(additional_input_outputs))
    print(f"Epoch {epoch+1}/{epochs} - Average Loss: {avg_loss:.4f}")

# Print final predicted output and loss
print("Final Predicted Output:", predicted_output)
print("Final Loss:", loss)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-e02cfc01f442> in <cell line: 57>()
     78         # Backpropagation
     79         output_error = target_onehot - predicted_output
---> 80         hidden2_error = output_error.dot(weights_hidden2_output.T)
     81         hidden2_delta = hidden2_error * sigmoid_derivative(hidden2_output)
     82         hidden1_error = hidden2_delta.dot(weights_hidden1_hidden2.T)

ValueError: shapes (1,2) and (1,6) not aligned: 2 (dim 1) != 1 (dim 0)
```

SEARCH STACK OVERFLOW

```
import numpy as np

# Define sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define softmax activation function
```

```
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

# Define cross-entropy loss function
def cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.sum(y_true * np.log(y_pred))

# Neural network architecture
input_size = 4
hidden1_size = 8
hidden2_size = 6
output_size = 2

# Initialize random weights and biases
np.random.seed(42)
weights_input_hidden1 = np.random.rand(input_size, hidden1_size)
biases_hidden1 = np.random.rand(1, hidden1_size)
weights_hidden1_hidden2 = np.random.rand(hidden1_size, hidden2_size)
biases_hidden2 = np.random.rand(1, hidden2_size)
weights_hidden2_output = np.random.rand(hidden2_size, output_size)
biases_output = np.random.rand(1, output_size)

# Sample input data and corresponding target output
input_data = np.array([[0.1, 0.2, 0.3, 0.4]])
target_output = np.array([[0, 1]])  # One-hot encoded target
additional_input_outputs = [
    (np.array([0.7, 0.4, 0.8, 0.2]), np.array([0])),
    (np.array([0.3, 0.6, 0.1, 0.9]), np.array([1])),
    (np.array([0.9, 0.1, 0.5, 0.6]), np.array([1])),
    (np.array([0.2, 0.3, 0.4, 0.5]), np.array([0]))
]

input_outputs.extend(additional_input_outputs)

# Hyperparameters
learning_rate = 0.1
epochs = 10000

# Training loop
for epoch in range(epochs):
    # Forward propagation
    hidden1_input = np.dot(input_data, weights_input_hidden1) + biases_hidden1
    hidden1_output = sigmoid(hidden1_input)
    hidden2_input = np.dot(hidden1_output, weights_hidden1_hidden2) + biases_hidden2
    hidden2_output = sigmoid(hidden2_input)
    output_input = np.dot(hidden2_output, weights_hidden2_output) + biases_output
    predicted_output = softmax(output_input)

    # Calculate loss
    loss = cross_entropy(target_output, predicted_output)

    # Backpropagation
    output_error = target_output - predicted_output
    hidden2_error = output_error.dot(weights_hidden2_output.T)
    hidden2_delta = hidden2_error * sigmoid_derivative(hidden2_output)
    hidden1_error = hidden2_delta.dot(weights_hidden1_hidden2.T)
    hidden1_delta = hidden1_error * sigmoid_derivative(hidden1_output)

    # Update weights and biases using gradient descent
```

```
    weights_hidden2_output += hidden2_output.T.dot(output_error) * learning_rate
    biases_output += np.sum(output_error, axis=0, keepdims=True) * learning_rate
    weights_hidden1_hidden2 += hidden1_output.T.dot(hidden2_delta) * learning_rate
    biases_hidden2 += np.sum(hidden2_delta, axis=0, keepdims=True) * learning_rate
    weights_input_hidden1 += input_data.T.dot(hidden1_delta) * learning_rate
    biases_hidden1 += np.sum(hidden1_delta, axis=0, keepdims=True) * learning_rate

# Print final predicted output and loss
print("Final Predicted Output:", predicted_output)
print("Final Loss:", loss)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-15-eee56ea7feb8> in <cell line: 46>()
     44 ]
     45 
---> 46 input_outputs.extend(additional_input_outputs)
     47 
     48 # Hyperparameters

NameError: name 'input_outputs' is not defined
```

SEARCH STACK OVERFLOW

Practical_11

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
```

```
# Load and preprocessing
(x_train,_), (x_test,_)= mnist.load_data()
x_train = x_train.astype('float32')/255.0
x_test = x_test.astype('float32')/255.0

```

```
#flatten image (28x28 to 784)
x_train = x_train.reshape((-1,784))
x_test = x_test.reshape((-1,784))
```

```
#define architechture of autoencoder
encoding_dim = 32

input_layer = Input(shape=(784,))
encoded = Dense(encoding_dim, activation = 'relu')(input_layer)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)

#Compile the autoencoder
autoencoder.compile(optimizer = 'adam', loss = 'binary_crossentropy')

#Train the autoencoder
autoencoder.fit(x_train,x_train, epochs= 50, batch_size = 256, shuffle = True, validation_dat
```

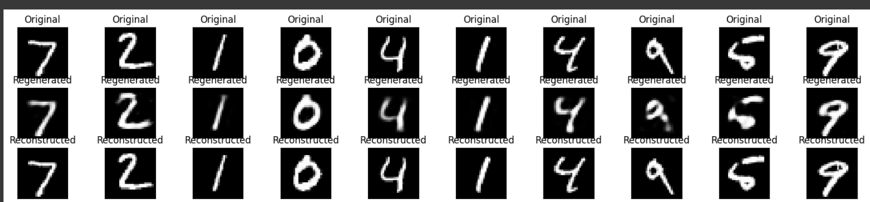
```
Epoch 1/50
235/235 [==============================] - 6s 22ms/step - loss: 0.2766 - val_loss: 0.1931
Epoch 2/50
235/235 [==============================] - 3s 12ms/step - loss: 0.1728 - val_loss: 0.1551
Epoch 3/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1458 - val_loss: 0.1356
Epoch 4/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1305 - val_loss: 0.1236
Epoch 5/50
235/235 [==============================] - 2s 10ms/step - loss: 0.1200 - val_loss: 0.1145
Epoch 6/50
235/235 [==============================] - 4s 16ms/step - loss: 0.1126 - val_loss: 0.1084
Epoch 7/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1075 - val_loss: 0.1041
Epoch 8/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1037 - val_loss: 0.1009
Epoch 9/50
235/235 [==============================] - 3s 11ms/step - loss: 0.1010 - val_loss: 0.0985
Epoch 10/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0989 - val_loss: 0.0968
Epoch 11/50
235/235 [==============================] - 3s 14ms/step - loss: 0.0975 - val_loss: 0.0956
Epoch 12/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0960 - val_loss: 0.0938
Epoch 13/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0947 - val_loss: 0.0932
Epoch 14/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0943 - val_loss: 0.0927
Epoch 15/50
235/235 [==============================] - 3s 13ms/step - loss: 0.0939 - val_loss: 0.0925
Epoch 16/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0937 - val_loss: 0.0925
Epoch 17/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0935 - val_loss: 0.0922
Epoch 18/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0934 - val_loss: 0.0922
Epoch 19/50
```

```
235/235 [==============================] - 3s 11ms/step - loss: 0.0933 - val_loss: 0.0922
Epoch 20/50
235/235 [==============================] - 3s 15ms/step - loss: 0.0932 - val_loss: 0.0919
Epoch 21/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0931 - val_loss: 0.0919
Epoch 22/50
235/235 [==============================] - 2s 11ms/step - loss: 0.0931 - val_loss: 0.0918
Epoch 23/50
235/235 [==============================] - 2s 11ms/step - loss: 0.0930 - val_loss: 0.0917
Epoch 24/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0930 - val_loss: 0.0917
Epoch 25/50
235/235 [==============================] - 4s 15ms/step - loss: 0.0929 - val_loss: 0.0917
Epoch 26/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0929 - val_loss: 0.0916
Epoch 27/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0929 - val_loss: 0.0916
Epoch 28/50
235/235 [==============================] - 2s 10ms/step - loss: 0.0928 - val_loss: 0.0916
Epoch 29/50
235/235 [==============================] - 3s 12ms/step - loss: 0.0928 - val_loss: 0.0917
```

```
#used train autoencoder to regenerate and reconstruct images
regenerated_images = autoencoder.predict(x_test)
```

```
313/313 [==============================] - 0s 1ms/step
```

```
#Visualize original, regenrated and reconstructed
n = 10
plt.figure(figsize=(20,4))
for i in range(n):
  ax = plt.subplot(3,n,i+1)
  plt.imshow(x_test[i].reshape(28,28))
  plt.title('Original')
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)


  ax= plt.subplot(3,n,i+n+1)
  plt.imshow(regenerated_images[i].reshape(28,28))
  plt.title('Regenerated')
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)

  ax= plt.subplot(3,n,i+2 * n+1)
  plt.imshow(x_test[i].reshape(28,28))
  plt.title('Reconstructed')
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
plt.show()
```



Practical_12

```
import numpy as np
import pandas as pd
from numpy import unique, argmax
from tensorflow.keras.datasets.mnist import load_data
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPool2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.utils import plot_model
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
```

```
(train_x, train_y), (test_x, test_y) = mnist.load_data()
```

```
#printing the shapes
print(train_x.shape, train_y.shape)
print(test_x.shape , test_y.shape)
```

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

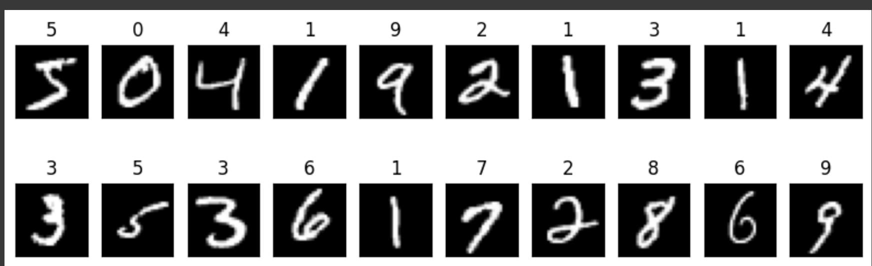
```
#reshaping train and test sets
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1],train_x.shape[2],1))
test_x = test_x .reshape((test_x.shape[0], test_x.shape[1],test_x.shape[2],1))
```

```
#printing the shapes
print(train_x.shape, train_y.shape)
print(test_x.shape , test_y.shape)
```

```
(60000, 28, 28, 1) (60000,)
(10000, 28, 28, 1) (10000,)
```

```
#normalizing the pixel values of images
train_x = train_x.astype('float32')/255.0
test_x = test_x.astype('float32')/255.0
```

```
#plotting images of dataset
fig = plt.figure(figsize = (10,3))
for i in range(20):
    ax= fig.add_subplot(2, 10, i+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(train_x[i]), cmap='gray')
    ax.set_title(train_y[i])
```




```
#Let us try to print the shape of a single image.
shape = train_x.shape[1:]
shape
```

```
(28, 28, 1)
```

```
#CNN Model
model = Sequential()
#adding convolutional layer
model.add(Conv2D(128, (3,3), activation='relu', input_shape= shape))
model.add(MaxPool2D((2,2)))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(MaxPool2D((2,2)))
model.add(Conv2D(32, (3,3), activation='relu'))
model.add(MaxPool2D((2,2)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

```
model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_3 (Conv2D)           (None, 26, 26, 128)       1280      
                                                                 
 max_pooling2d_3 (MaxPooling  (None, 13, 13, 128)      0         
 2D)                                                             
                                                                 
 conv2d_4 (Conv2D)           (None, 11, 11, 64)        73792     
                                                                 
 max_pooling2d_4 (MaxPooling  (None, 5, 5, 64)         0         
 2D)                                                             
                                                                 
 conv2d_5 (Conv2D)           (None, 3, 3, 32)          18464     
                                                                 
 max_pooling2d_5 (MaxPooling  (None, 1, 1, 32)         0         
 2D)                                                             
                                                                 
 dropout_1 (Dropout)         (None, 1, 1, 32)          0         
                                                                 
 flatten_2 (Flatten)         (None, 32)                0         
                                                                 
 dense_33 (Dense)            (None, 500)               16500     
                                                                 
 dense_34 (Dense)            (None, 10)                5010      
                                                                 
=================================================================
Total params: 115,046
Trainable params: 115,046
Non-trainable params: 0
_________________________________________________________________
```

```
#compiling model
model.compile(optimizer='adam', loss = 'sparse_categorical_crossentropy',
              metrics= ['accuracy'])
x=model.fit(train_x,train_y,epochs=10,batch_size=256, verbose= 1,
            validation_split = 0.1)
```

```
Epoch 1/10
211/211 [==============================] - 7s 24ms/step - loss: 0.9067 - accuracy: 0.6913 - val_loss: 0.2049 - val_accuracy: 0.9502
Epoch 2/10
211/211 [==============================] - 4s 18ms/step - loss: 0.4106 - accuracy: 0.8681 - val_loss: 0.1217 - val_accuracy: 0.9652
Epoch 3/10
211/211 [==============================] - 4s 17ms/step - loss: 0.3154 - accuracy: 0.8981 - val_loss: 0.1193 - val_accuracy: 0.9692
Epoch 4/10
211/211 [==============================] - 4s 17ms/step - loss: 0.2666 - accuracy: 0.9154 - val_loss: 0.0993 - val_accuracy: 0.9738
Epoch 5/10
211/211 [==============================] - 4s 17ms/step - loss: 0.2407 - accuracy: 0.9239 - val_loss: 0.0970 - val_accuracy: 0.9763
Epoch 6/10
211/211 [==============================] - 4s 18ms/step - loss: 0.2141 - accuracy: 0.9313 - val_loss: 0.0868 - val_accuracy: 0.9785
Epoch 7/10
211/211 [==============================] - 4s 17ms/step - loss: 0.1911 - accuracy: 0.9401 - val_loss: 0.0840 - val_accuracy: 0.9790
Epoch 8/10
211/211 [==============================] - 4s 17ms/step - loss: 0.1759 - accuracy: 0.9440 - val_loss: 0.0756 - val_accuracy: 0.9813
Epoch 9/10
211/211 [==============================] - 4s 18ms/step - loss: 0.1659 - accuracy: 0.9475 - val_loss: 0.0795 - val_accuracy: 0.9793
Epoch 10/10
211/211 [==============================] - 5s 23ms/step - loss: 0.1548 - accuracy: 0.9513 - val_loss: 0.0738 - val_accuracy: 0.9820
```

```
loss, accuracy= model.evaluate(test_x, test_y, verbose = 0)
print(f'Accuracy: {accuracy*100}')
```

```
Accuracy: 97.97999858856201
```

```
ypred_CNN=model.predict(test_x)
```

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

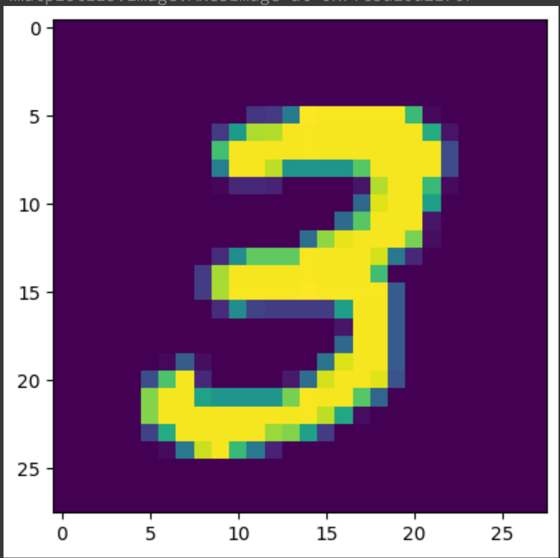
Autoencoder for Generating Images

```
from tensorflow.keras.datasets import mnist
```

```
train_x=train_x.reshape(-1,train_x.shape[1],train_x.shape[2])
```

```
plt.imshow(np.squeeze(train_x[7]))
```

```
<matplotlib.image.AxesImage at 0x7fcbd10d2170>
```




```
train_x=train_x/255
test_x=test_x/255
```

```
print(train_x.shape)
```

```
(60000, 28, 28)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Flatten,Reshape
from tensorflow.keras.optimizers import SGD,Adam
import tensorflow as tf
```

```
#Encoder
encoder=Sequential()
encoder.add(Flatten(input_shape=[train_x.shape[1],train_x.shape[2]]))
encoder.add(Dense(500,activation='relu'))
encoder.add(Dense(400,activation='relu'))
encoder.add(Dense(300,activation='relu'))
encoder.add(Dense(200,activation='relu'))
encoder.add(Dense(100,activation='relu'))
encoder.add(Dense(50,activation='relu'))
encoder.add(Dense(25,activation='relu'))
encoder.add(Dense(10,activation='relu'))
```

```
#Decoder
decoder=Sequential()
decoder.add(Dense(25,input_shape=[10],activation='relu'))
decoder.add(Dense(50,activation='relu'))
decoder.add(Dense(100,activation='relu'))
decoder.add(Dense(200,activation='relu'))
decoder.add(Dense(300,activation='relu'))
decoder.add(Dense(400,activation='relu'))
decoder.add(Dense(500,activation='relu'))
decoder.add(Dense(train_x.shape[1]*train_x.shape[2],activation='sigmoid'))
decoder.add(Reshape([28,28]))
```

```
autoencoder=Sequential([encoder,decoder])
```

```
autoencoder.compile(loss='binary_crossentropy',optimizer='Adam',
                    metrics=['accuracy'])
```

```
autoencoder.fit(train_x,train_x,batch_size=256,epochs=50,
                validation_split=0.2,verbose=True)
```

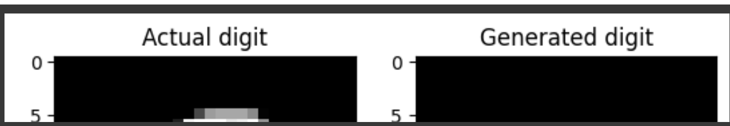
```
Epoch 1/50
188/188 [==============================] - 7s 12ms/step - loss: 0.0563 - accuracy: 0.2608 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 2/50
188/188 [==============================] - 2s 13ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 3/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 4/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 5/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 6/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 7/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 8/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 9/50
188/188 [==============================] - 3s 14ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 10/50
188/188 [==============================] - 2s 12ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 11/50
188/188 [==============================] - 2s 13ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 12/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 13/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 14/50
188/188 [==============================] - 3s 17ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 15/50
188/188 [==============================] - 3s 17ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 16/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 17/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 18/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 19/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 20/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 21/50
188/188 [==============================] - 2s 12ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 22/50
188/188 [==============================] - 2s 13ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 23/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 24/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 25/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 26/50
188/188 [==============================] - 2s 10ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 27/50
188/188 [==============================] - 2s 9ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.293
Epoch 28/50
188/188 [==============================] - 2s 12ms/step - loss: 0.0079 - accuracy: 0.2955 - val_loss: 0.0079 - val_accuracy: 0.29
Epoch 29/50
```

KAUSTUBH RAYKAR

```
#prediction
predicted=autoencoder.predict(test_x)
```

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

```
for i in range(10):
    n=np.random.randint(1,(test_x.shape[0]))
    plt.figure()
    plt.subplot(1,2,1)
    plt.imshow(test_x[n],cmap='gray')
    plt.title('Actual digit')
    plt.subplot(1,2,2)
    plt.imshow(predicted[n],cmap='gray')
    plt.title('Generated digit')
```


Actual digit
Generated digit
0
5
0
5

Practical_13

```
import numpy as np
from tensorflow import keras
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding,SimpleRNN, Dense
from tensorflow.keras.preprocessing import sequence
```

```
max_features = 10000
maxlen = 500
batch_size = 32
```

```
(x_train,y_train),(x_test,y_test) = imdb.load_data(num_words=max_features)

#pad sequence so tht they have thee same length
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 [==============================] - 0s 0us/step
```

```
#define the model
model = Sequential()
model.add(Embedding(max_features,32))
model.add(SimpleRNN(32))
model.add(Dense(1,activation = 'sigmoid'))
```

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

#train the model
model.fit(x_train,y_train, epochs = 10, batch_size=batch_size, validation_split= 0.2)

#Evaluate the model on the test set
loss, accuracy = model.evaluate(x_test,y_test,batch_size=batch_size)
print(f'Test Loss : {loss}, Test Accuracy:{accuracy}')
```

```
Epoch 1/10
365/625 [================>............] - ETA: 2:08 - loss: 0.6205 - accuracy: 0.6482
```