Q.1: Write a program for digits recognition using tensorflow

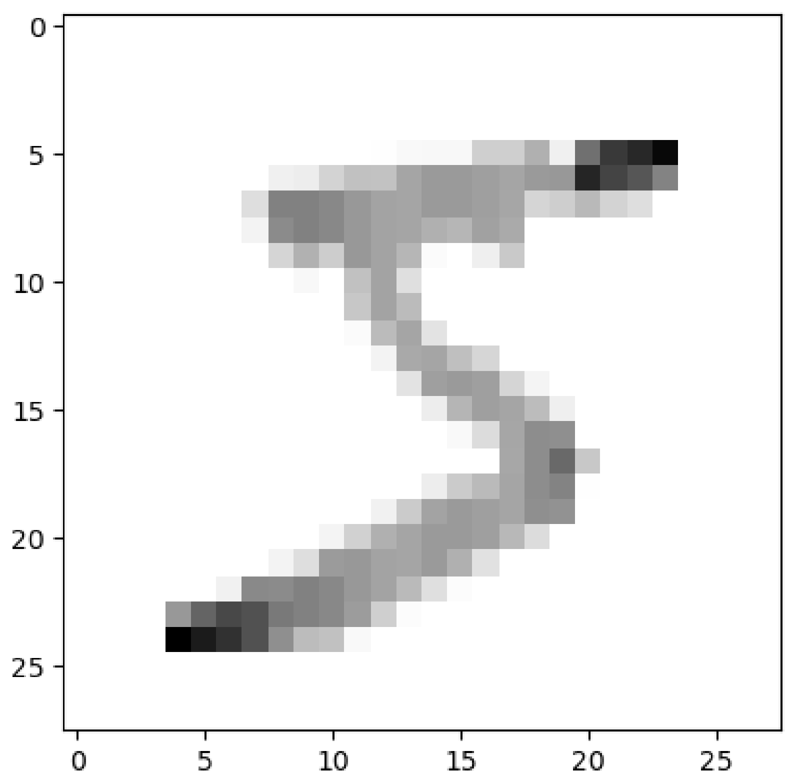
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

mnist = tf.keras.datasets.mnist
(x_train,y_train) , (x_test,y_test) = mnist.load_data()

x_train = tf.keras.utils.normalize(x_train,axis=1)
x_test = tf.keras.utils.normalize(x_test,axis=1)

def draw(n):
    plt.imshow(n,cmap=plt.cm.binary)
    plt.show()
draw(x_train[0])
```



```python
model = tf.keras.models.Sequential()

model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))


model.add(tf.keras.layers.Dense(128,activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128,activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(10,activation=tf.nn.softmax))

model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
            )
model.fit(x_train,y_train,epochs=3)
```

```
Epoch 1/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.8655 - loss:
0.4771
Epoch 2/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9653 - loss:
0.1111
Epoch 3/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.9783 - loss:
0.0706

<keras.src.callbacks.history.History at 0x1a25c8098d0>
```

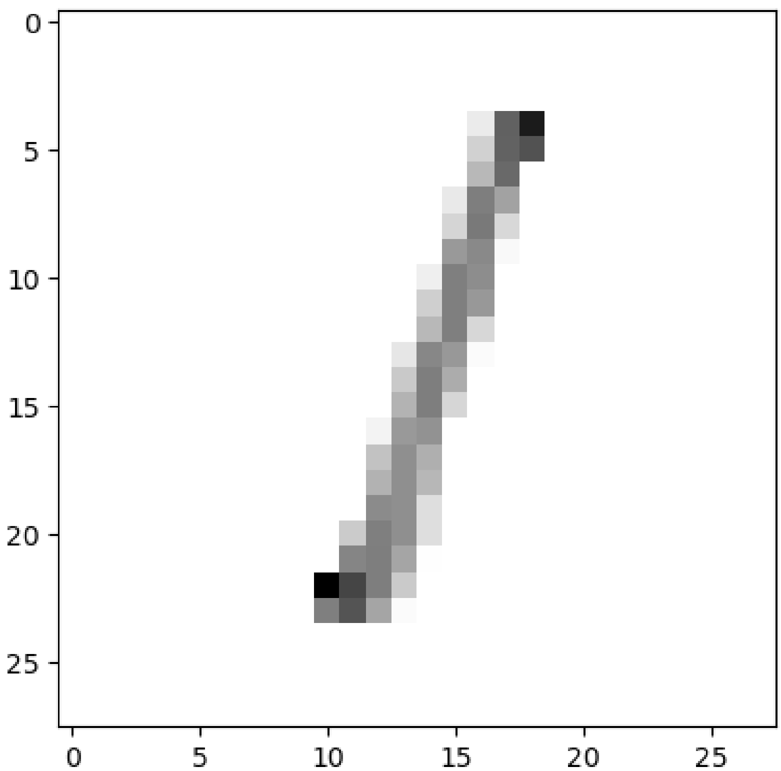
```python
val_loss,val_acc = model.evaluate(x_test,y_test)
print("loss-> ",val_loss,"\nacc-> ",val_acc)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 813us/step - accuracy: 0.9659 - loss:
0.1108
loss->  0.09320031851530075
acc->  0.9715999960899353
```

```python
predictions=model.predict([x_test])
print('label -> ',y_test[2])
print('prediction -> ',np.argmax(predictions[2]))

draw(x_test[2])
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
label ->  1
prediction ->  1
```



```
model.save('digit_recognition.keras')

model.summary()

Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ flatten (Flatten)                │ (None, 784)             │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense (Dense)                    │ (None, 128)             │
100,480 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_1 (Dense)                  │ (None, 128)             │
16,512 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ dense_2 (Dense)                  │ (None, 10)              │
1,290 │
```

```
└──────────────────────────────────┴─────────────────────────┴─────────────
──────┘

 Total params: 354,848 (1.35 MB)

 Trainable params: 118,282 (462.04 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 236,566 (924.09 KB)

model.evaluate(x_test, y_test)

313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 932us/step - accuracy: 0.9659 - loss: 0.1108

[0.09320031851530075, 0.9715999960899353]
```

Q.2: What are Activation Functions? Give Examples and Explain.

Certainly! Activation functions play a crucial role in artificial neural networks. They determine whether a neuron should be activated or not based on the weighted sum of its inputs and a bias. Let’s explore some common activation functions:

1. Step Function: The step function is one of the simplest activation functions. It uses a threshold value. If the net input (denoted as (y)) is greater than the threshold, the neuron is activated. Mathematically: [f(y) = \begin{cases} 1 & \text{if } y > \text{threshold} 0 & \text{otherwise} \end{cases}] Graphically, it looks like this: !Step Function

2. Sigmoid Function: The sigmoid function is widely used due to its smoothness and non-linearity. It maps the net input to a value between 0 and 1. Mathematically: [f(y) = \frac{1}{1 + e^{-y}}] Graphically, it has an S-shaped curve: !Sigmoid Function

3. ReLU (Rectified Linear Unit): ReLU is the most popular activation function. It replaces negative inputs with zero and leaves positive inputs unchanged. Mathematically: [f(y) = \max(0, y)] Graphically: !ReLU Function

4. Leaky ReLU: Leaky ReLU improves upon ReLU by allowing a small linear component for negative inputs. It avoids the “dying ReLU” problem where some neurons remain inactive. Mathematically: [f(y) = \begin{cases} y & \text{if } y > 0 0.01y & \text{otherwise} \end{cases}] Graphically: !Leaky ReLU Function

These activation functions help introduce non-linearity, allowing neural networks to learn complex patterns in data.

Q.3: Explain Higer order Tensor with the help of example.

Let's dive into higher-order gradients in TensorFlow.

First-Order Gradients First-order gradients, often referred to simply as gradients (∇), guide us in the ascent of a function, similar to climbing up a hill. Mathematically, for a scalar function (f(x)), the first-order gradient is given by:

[\nabla f(x) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right]]

In other words, it represents the rate of change of the function with respect to each input variable. TensorFlow provides the tf.GradientTape function to compute these gradients during the forward pass. Let's look at an example:

```python
import tensorflow as tf

x = tf.Variable(5.0)
y = tf.Variable(2.0)

def f(x, y):
    return 2 * x**3 + 5 * y**2 + 11 * x + 5

# Calculate derivative w.r.t. x
with tf.GradientTape() as tape:
    z = f(x, y)
    dx = tape.gradient(z, x)

# Calculate derivative w.r.t. y (create a new tape)
with tf.GradientTape() as tape:
    z = f(x, y)
    dy = tape.gradient(z, y)

print("Partial derivative of f with respect to x:", dx.numpy())
print("Partial derivative of f with respect to y:", dy.numpy())
```

```
Partial derivative of f with respect to x: 161.0
Partial derivative of f with respect to y: 20.0
```

Higher-Order Gradients Higher-order derivatives provide insights into the curvature of the function. TensorFlow allows us to compute not only first-order but also second, third, and nth derivatives seamlessly. The second-order gradient (Hessian matrix) for (f(x)) is given by:

[\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \vdots & \vdots & \ddots & \vdots \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}]

In practice, you can compute higher-order gradients using the same tf.GradientTape mechanism. TensorFlow makes it easy to explore the behavior of your models beyond simple gradients.