Batch normalization

Issues with training Deep Neural Networks

- There are 2 major issues 1) Internal Covariate shift, 2) Vanishing Gradient

Internal Covariate shift

- The concept of covariate shift pertains to the change that occurs in the distribution of the input to a learning system. In deep networks, this distribution can be influenced by parameters across all input layers. Consequently, even minor changes in the network can have a significant impact on its output. This effect gets magnified as the signal propagates through the network, which can result in a shift in the distribution of the inputs to internal layers. This phenomenon is known as internal covariate shift.
- When inputs are whitened (i.e., have zero mean and unit variance) and are uncorrelated, they tend to converge faster during training. However, internal covariate shift can have the opposite effect, as it introduces changes to the distribution of inputs that can slow down convergence. Therefore, to mitigate this effect, techniques like batch normalization have been developed to normalize the inputs to each layer in the network based on statistics of the current mini-batch.

Vanishing Gradient

- Saturating non-linearities such as sigmoid or tanh are not suitable for deep networks, as the signal tends to get trapped in the saturation region as the network grows deeper. This makes it difficult for the network to learn and can result in slow convergence during training. To overcome this problem we can use the following.
- Non-linearities like ReLU which do not saturate.
- Smaller learning rates
-

Careful initializations

What is Normalization?

-

Normalization in deep learning refers to the process of transforming the input or output of a layer in a neural network to improve its performance during training. The most common type of normalization used in deep learning is batch normalization, which normalizes the activations of a layer for each mini-batch during training.

What is batch normalization?

- Batch normalization is a technique in deep learning that helps to standardize and normalize the input to each layer of a neural network by adjusting and scaling the activations. The idea behind batch normalization is to normalize the inputs to a layer to have zero mean and unit variance across each mini-batch of the training data.

Steps involved in batch normalization

1) During training, for each mini-batch of data, compute the mean and variance of the activations of each layer. This can be done using the following formulas:

- Mean: $\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i$
- Variance: $\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$
- Here, m is the size of the mini-batch, and x_i is the activation of the i-th neuron in the layer.

2) Normalize the activations of each layer in the mini-batch using the following formula:

- $\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ Here, ϵ is a small constant added for numerical stability.

3) Scale and shift the normalized activations using the learned parameters γ and β, respectively:

- $y_i = \gamma \hat{x_i} + \beta$
- The parameters γ and β are learned during training using backpropagation.

4) During inference, the running mean and variance of each layer are used for normalization instead of the mini-batch statistics. These running statistics are updated using a moving average of the mini-batch statistics during training.

The benefits of batch normalization include:

- Improved training performance: Batch normalization reduces the internal covariate shift, which is the change in the distribution of the activations of each layer due to changes in the distribution of the inputs. This allows the network to converge faster and with more stable gradients.
- Regularization: Batch normalization acts as a form of regularization by adding noise to the activations of each layer, which can help prevent overfitting.
- **Increased robustness: Batch normalization makes the network more robust to changes in the input distribution, which can help improve its generalization performance.**

  Code example for batch normalization

```
import tensorflow as tf

# Define a fully connected layer
fc_layer = tf.keras.layers.Dense(units=128, activation='relu')

# Add batch normalization to the layer
bn_layer = tf.keras.layers.BatchNormalization()

# Define the model with the layer and batch normalization
model = tf.keras.models.Sequential([fc_layer, bn_layer])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy')

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

- In the above code,the tf.keras.layers.BatchNormalization() layer is added after the fully connected layer to normalize the output before passing it to the activation function. The model.fit() function is then used to train the model using batch normalization.

Practical discussion of Callback functions:

A callback is a powerful tool to customize the behavior of a Keras model during training, evaluation, or inference.TensorBoard to visualize training progress and results with TensorBoard, or tf.keras.callbacks. ModelCheckpoint to periodically save your model during training. This callback reduces the learning rate when a metric you've mentioned during training eg. accuracy or loss has stopped improving. Models often benefit from reducing the learning rate.There are many callback functions,

- Early Stopping callback
- Model checkpointing callback
- Tensorboard callback Functions

We will also see how to save a model & load as well.

In [1]:
```
# Importing Libraries
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os
```

In [2]:
```
# Checking version of Tensorflow and Keras
print(f"Tensorflow Version {tf.__version__}")
print(f"Keras Version {tf.keras.__version__}")
```

```
Tensorflow Version 2.10.0
Keras Version 2.10.0
```

GPU / CPU Check

In [3]:
```
tf.config.list_physical_devices("GPU")
```

Out[3]:
```
[]
```

In [4]:
```
tf.config.list_physical_devices("CPU")
```

Out[4]:
```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

In [5]:
```
check_list = ['GPU','CPU']

for device in check_list:
  out = tf.config.list_physical_devices(device)
  if len(out) > 0:
    print(f"{device} is available!")
    print(f"Details >> {out}")
  else:
    print(f"{device} isn't available!")
```

```
GPU isn't available!
CPU is available!
Details >> [PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

Creating a simple classifier using keras on MNIST data

In [6]:
```
mnist = tf.keras.datasets.mnist
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
```

In [7]:
```
print(f"data type of X_train_full: {X_train_full.dtype},\n shape of X_train_full: {X_tra
```

```
data type of X_train_full: uint8,
 shape of X_train_full: (60000, 28, 28)
```

In [8]:
```
X_test.shape
```

Out[8]:
```
(10000, 28, 28)
```

In [9]:
```
len(X_test[1][0])
```

Out[9]:
```
28
```

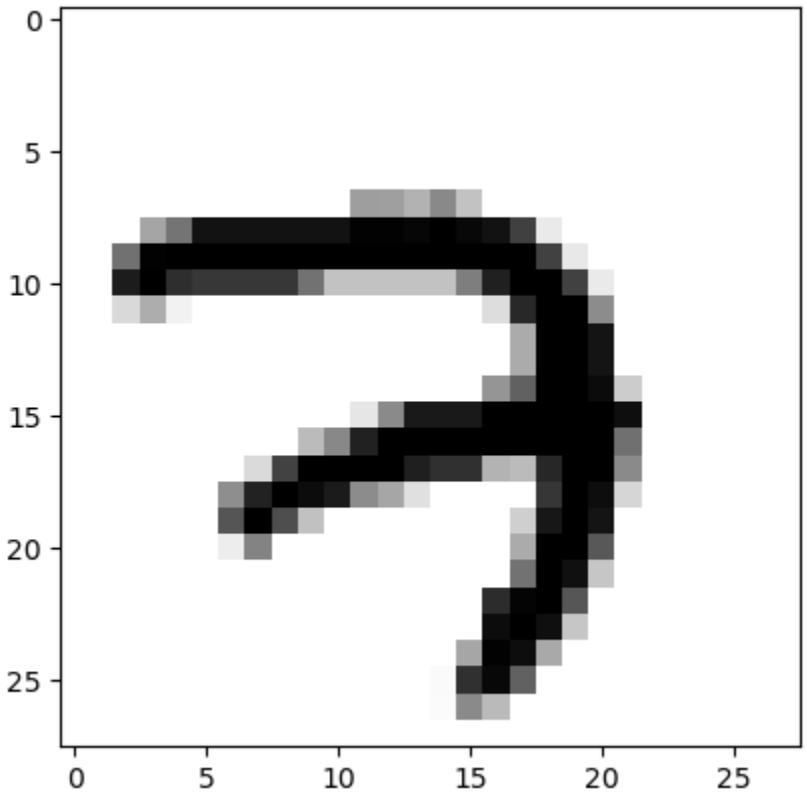
In [10]:
```
# create a validation data set from the full training data
# Scale the data between 0 to 1 by dividing it by 255. as its an unsigned data between 0
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

# scale the test set as well
X_test = X_test / 255.
```

In [11]:
```
len(X_train_full[5000:] )
```

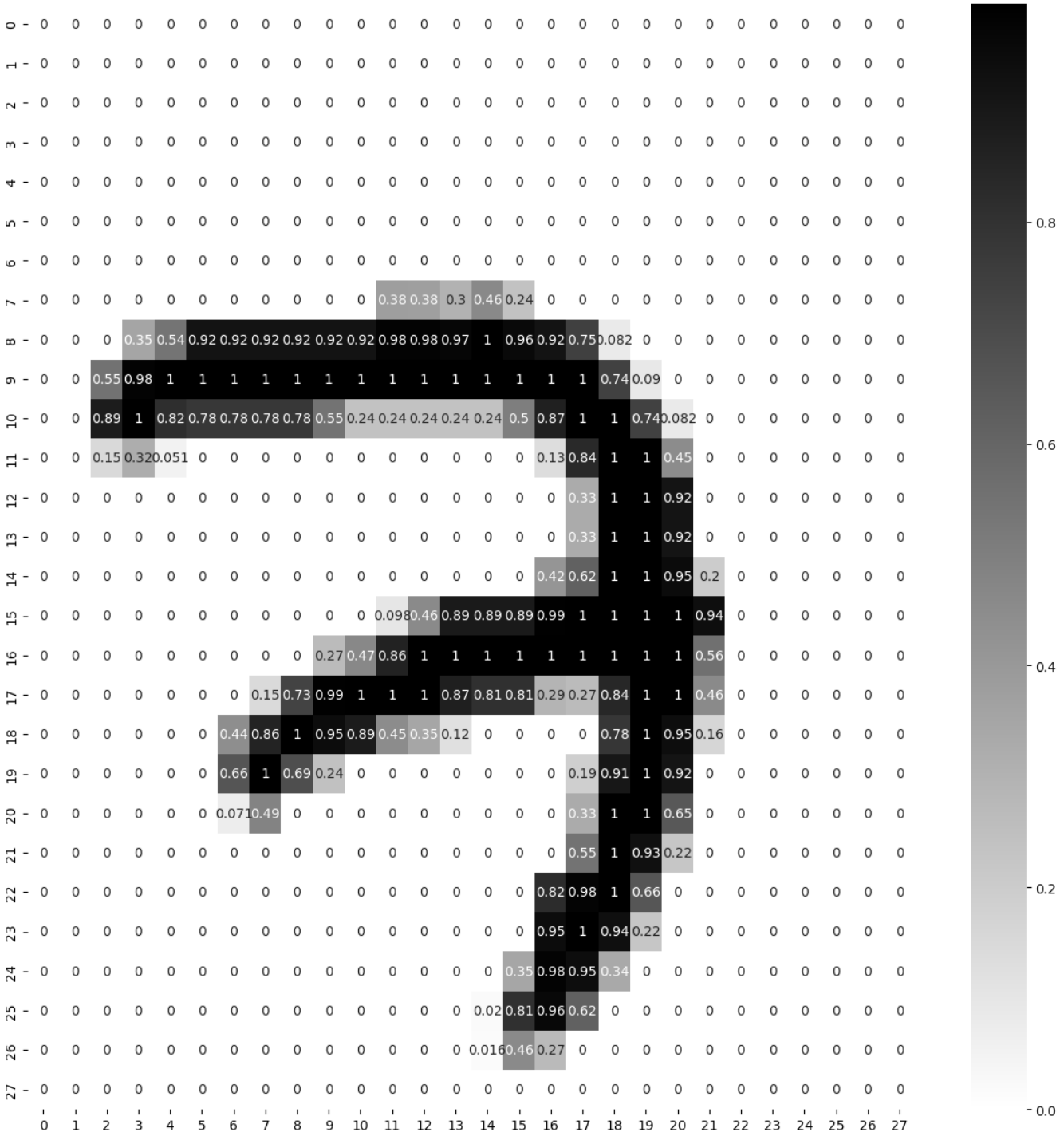
Out[11]:
```
55000
```

In [12]:
```
# Lets view some data
plt.imshow(X_train[0], cmap="binary")
plt.show()
```



In [13]:
```
plt.figure(figsize=(15,15))
sns.heatmap(X_train[0], annot=True, cmap="binary")
```

Out[13]: <Axes: >



In [14]:
```
LAYERS = [tf.keras.layers.Flatten(input_shape=[28, 28], name="inputLayer"),
          tf.keras.layers.Dense(300, activation="relu", name="hiddenLayer1"),
          tf.keras.layers.Dense(100, activation="relu", name="hiddenLayer2"),
          tf.keras.layers.Dense(10, activation="softmax", name="outputLayer")]

model_clf = tf.keras.models.Sequential(LAYERS)
```

In [15]:
```
model_clf.layers
```

Out[15]:
```
[<keras.layers.reshaping.flatten.Flatten at 0x19e30cc3460>,
 <keras.layers.core.dense.Dense at 0x19e30cc2bf0>,
 <keras.layers.core.dense.Dense at 0x19e30cc2c80>,
 <keras.layers.core.dense.Dense at 0x19e30cc2cb0>]
```

In [16]:
```
model_clf.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inputLayer (Flatten)        (None, 784)               0

 hiddenLayer1 (Dense)        (None, 300)               235500

 hiddenLayer2 (Dense)        (None, 100)               30100

 outputLayer (Dense)         (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [17]:
```
# first Layer * second Layer + bias
784*300 + 300, 300*100+100, 100*10+10
```

Out[17]:
```
(235500, 30100, 1010)
```

In [18]:
```
# Total parameters to be trained -
sum((235500, 30100, 1010))
```

Out[18]:
```
266610
```

In [19]:
```
hidden1 = model_clf.layers[1]
hidden1.name
```

Out[19]:
```
'hiddenLayer1'
```

In [20]:
```
model_clf.get_layer(hidden1.name) is hidden1
```

Out[20]:
```
True
```

In [21]:
```
len(hidden1.get_weights()[1])
```

Out[21]:
```
300
```

In [22]:
```python
hidden1.get_weights()
```

Out[22]:
```
[array([[-0.00863542,  0.07263291, -0.03879336, ...,  0.00046017,
          0.04086754, -0.00439219],
        [ 0.00946854, -0.05513434, -0.01372449, ...,  0.01021195,
         -0.04177602,  0.05911638],
        [-0.02636152, -0.06728062, -0.04932404, ..., -0.017939  ,
          0.01761802, -0.04399509],
        ...,
        [ 0.01695059, -0.00667475,  0.06896693, ..., -0.02042392,
         -0.02046927,  0.01085705],
        [-0.05604234, -0.04010153,  0.00630041, ..., -0.00471536,
         -0.06079594, -0.07128973],
        [-0.02956304,  0.06041798, -0.05449025, ..., -0.06903378,
          0.0262394 , -0.02772664]], dtype=float32),
 array([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., 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., 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.], dtype=float32)]
```

In [23]:
```python
weights, biases = hidden1.get_weights()
```

In [24]:
```python
print("shape\n",weights.shape, "\n")
weights
```

```
shape
 (784, 300)
```

Out[24]:
```
array([[-0.00863542,  0.07263291, -0.03879336, ...,  0.00046017,
         0.04086754, -0.00439219],
       [ 0.00946854, -0.05513434, -0.01372449, ...,  0.01021195,
        -0.04177602,  0.05911638],
       [-0.02636152, -0.06728062, -0.04932404, ..., -0.017939  ,
         0.01761802, -0.04399509],
       ...,
       [ 0.01695059, -0.00667475,  0.06896693, ..., -0.02042392,
        -0.02046927,  0.01085705],
       [-0.05604234, -0.04010153,  0.00630041, ..., -0.00471536,
        -0.06079594, -0.07128973],
       [-0.02956304,  0.06041798, -0.05449025, ..., -0.06903378,
         0.0262394 , -0.02772664]], dtype=float32)
```

In [25]:
```
print("shape\n", biases.shape)
```

```
shape
 (300,)
```

In [26]:
```
LOSS_FUNCTION = "sparse_categorical_crossentropy" # use => tf.losses.sparse_categorical_
OPTIMIZER = "SGD" # or use with custom learning rate=> tf.keras.optimizers.SGD(0.02)
METRICS = ["accuracy"]

model_clf.compile(loss=LOSS_FUNCTION,
              optimizer=OPTIMIZER,
              metrics=METRICS)
```

Tensorboard callback Functions

In [27]:
```
# Logging

import time

def get_log_path(log_dir="logs/fit"):
  fileName = time.strftime("log_%Y_%m_%d_%H_%M_%S")
  logs_path = os.path.join(log_dir, fileName)
  print(f"Saving logs at {logs_path}")
  return logs_path

log_dir = get_log_path()
tb_cb = tf.keras.callbacks.TensorBoard(log_dir=log_dir)
```

```
Saving logs at logs/fit\log_2023_07_26_00_44_20
```

Early Stopping callback

In [28]:
```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patCKPT_path = "Model_ckpt.h5"
checkpointing_cb = tf.keras.callbacks.ModelCheckpoint(CKPT_path, save_best_only=True)ien
```

Model checkpointing callback

In [29]:
```
CKPT_path = "Model_ckpt.h5"
checkpointing_cb = tf.keras.callbacks.ModelCheckpoint(CKPT_path, save_best_only=True)
```

In [30]:
```
# Orginal train

EPOCHS = 30
VALIDATION_SET = (X_valid, y_valid)

history = model_clf.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=32, callbacks=[tb_cb, ear
```

```
Epoch 1/30
1719/1719 [==============================] - 9s 4ms/step - loss: 0.5981 - accuracy: 0.8
476 - val_loss: 0.3059 - val_accuracy: 0.9170
Epoch 2/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.2917 - accuracy: 0.9
172 - val_loss: 0.2423 - val_accuracy: 0.9336
Epoch 3/30
1719/1719 [==============================] - 8s 5ms/step - loss: 0.2407 - accuracy: 0.9
321 - val_loss: 0.2069 - val_accuracy: 0.9420
Epoch 4/30
1719/1719 [==============================] - 8s 5ms/step - loss: 0.2053 - accuracy: 0.9
412 - val_loss: 0.1821 - val_accuracy: 0.9506
Epoch 5/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1792 - accuracy: 0.
9499 - val_loss: 0.1622 - val_accuracy: 0.9564
Epoch 6/30
1719/1719 [==============================] - 14s 8ms/step - loss: 0.1592 - accuracy: 0.
9545 - val_loss: 0.1480 - val_accuracy: 0.9598
Epoch 7/30
1719/1719 [==============================] - 13s 7ms/step - loss: 0.1424 - accuracy: 0.
9602 - val_loss: 0.1381 - val_accuracy: 0.9626
Epoch 8/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1285 - accuracy: 0.
9631 - val_loss: 0.1270 - val_accuracy: 0.9644
Epoch 9/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.1169 - accuracy: 0.9
667 - val_loss: 0.1200 - val_accuracy: 0.9672
Epoch 10/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1070 - accuracy: 0.
9699 - val_loss: 0.1096 - val_accuracy: 0.9714
Epoch 11/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0982 - accuracy: 0.9
720 - val_loss: 0.1061 - val_accuracy: 0.9718
Epoch 12/30
1719/1719 [==============================] - 6s 4ms/step - loss: 0.0906 - accuracy: 0.9
746 - val_loss: 0.1002 - val_accuracy: 0.9704
Epoch 13/30
1719/1719 [==============================] - 6s 4ms/step - loss: 0.0838 - accuracy: 0.9
763 - val_loss: 0.0960 - val_accuracy: 0.9742
Epoch 14/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.0776 - accuracy: 0.9
783 - val_loss: 0.0928 - val_accuracy: 0.9732
Epoch 15/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.0721 - accuracy: 0.9
799 - val_loss: 0.0907 - val_accuracy: 0.9738
Epoch 16/30
1719/1719 [==============================] - 8s 4ms/step - loss: 0.0674 - accuracy: 0.9
820 - val_loss: 0.0865 - val_accuracy: 0.9754
Epoch 17/30
1719/1719 [==============================] - 8s 5ms/step - loss: 0.0628 - accuracy: 0.9
828 - val_loss: 0.0818 - val_accuracy: 0.9760
Epoch 18/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0587 - accuracy: 0.9
841 - val_loss: 0.0797 - val_accuracy: 0.9774
Epoch 19/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0550 - accuracy: 0.9
850 - val_loss: 0.0782 - val_accuracy: 0.9778
Epoch 20/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.0516 - accuracy: 0.
```

```
9865 - val_loss: 0.0761 - val_accuracy: 0.9776
Epoch 21/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0480 - accuracy: 0.9
874 - val_loss: 0.0754 - val_accuracy: 0.9764
Epoch 22/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0453 - accuracy: 0.9
883 - val_loss: 0.0746 - val_accuracy: 0.9782
Epoch 23/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0425 - accuracy: 0.9
889 - val_loss: 0.0727 - val_accuracy: 0.9790
Epoch 24/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0400 - accuracy: 0.9
899 - val_loss: 0.0730 - val_accuracy: 0.9784
Epoch 25/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0377 - accuracy: 0.9
910 - val_loss: 0.0728 - val_accuracy: 0.9774
Epoch 26/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0355 - accuracy: 0.9
913 - val_loss: 0.0709 - val_accuracy: 0.9798
Epoch 27/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.0333 - accuracy: 0.
9920 - val_loss: 0.0702 - val_accuracy: 0.9786
Epoch 28/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0315 - accuracy: 0.9
928 - val_loss: 0.0688 - val_accuracy: 0.9786
Epoch 29/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0294 - accuracy: 0.9
933 - val_loss: 0.0675 - val_accuracy: 0.9794
Epoch 30/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0279 - accuracy: 0.9
940 - val_loss: 0.0689 - val_accuracy: 0.9806
```

In [31]:
```
# Checkpoint training

#Loading Checkpoint model
ckpt_model = tf.keras.models.load_model(CKPT_path)

history = ckpt_model.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=32, callbacks=[tb_cb, ear
```

```
Epoch 1/30
1719/1719 [==============================] - 8s 4ms/step - loss: 0.0278 - accuracy: 0.9
939 - val_loss: 0.0670 - val_accuracy: 0.9794
Epoch 2/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.0264 - accuracy: 0.9
944 - val_loss: 0.0673 - val_accuracy: 0.9794
Epoch 3/30
1719/1719 [==============================] - 6s 4ms/step - loss: 0.0248 - accuracy: 0.9
947 - val_loss: 0.0669 - val_accuracy: 0.9804
Epoch 4/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.0233 - accuracy: 0.9
952 - val_loss: 0.0694 - val_accuracy: 0.9798
Epoch 5/30
1719/1719 [==============================] - 7s 4ms/step - loss: 0.0221 - accuracy: 0.9
960 - val_loss: 0.0681 - val_accuracy: 0.9798
Epoch 6/30
1719/1719 [==============================] - 9s 5ms/step - loss: 0.0208 - accuracy: 0.9
962 - val_loss: 0.0675 - val_accuracy: 0.9808
Epoch 7/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.0198 - accuracy: 0.9
967 - val_loss: 0.0664 - val_accuracy: 0.9802
Epoch 8/30
1719/1719 [==============================] - 11s 6ms/step - loss: 0.0186 - accuracy: 0.
9972 - val_loss: 0.0662 - val_accuracy: 0.9802
Epoch 9/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.0177 - accuracy: 0.
9972 - val_loss: 0.0667 - val_accuracy: 0.9798
Epoch 10/30
1719/1719 [==============================] - 11s 7ms/step - loss: 0.0169 - accuracy: 0.
9973 - val_loss: 0.0698 - val_accuracy: 0.9798
Epoch 11/30
1719/1719 [==============================] - 10s 6ms/step - loss: 0.0160 - accuracy: 0.
9977 - val_loss: 0.0660 - val_accuracy: 0.9800
Epoch 12/30
1719/1719 [==============================] - 13s 7ms/step - loss: 0.0150 - accuracy: 0.
9981 - val_loss: 0.0661 - val_accuracy: 0.9806
Epoch 13/30
1719/1719 [==============================] - 12s 7ms/step - loss: 0.0143 - accuracy: 0.
9982 - val_loss: 0.0676 - val_accuracy: 0.9804
Epoch 14/30
1719/1719 [==============================] - 11s 6ms/step - loss: 0.0136 - accuracy: 0.
9983 - val_loss: 0.0666 - val_accuracy: 0.9810
Epoch 15/30
1719/1719 [==============================] - 11s 6ms/step - loss: 0.0130 - accuracy: 0.
9985 - val_loss: 0.0660 - val_accuracy: 0.9812
Epoch 16/30
1719/1719 [==============================] - 11s 6ms/step - loss: 0.0123 - accuracy: 0.
9987 - val_loss: 0.0666 - val_accuracy: 0.9810
```

Saving the Model

In [32]:
```
import time
import os

def save_model_path(MODEL_dir = "TRAINED_MODEL"):
  os.makedirs(MODEL_dir, exist_ok= True)
  fileName = time.strftime("Model_%Y_%m_%d_%H_%M_%S_.h5")
  model_path = os.path.join(MODEL_dir, fileName)
  print(f"Model {fileName} will be saved at {model_path}")
  return model_path
```

In [33]:
```
UNIQUE_PATH = save_model_path()
UNIQUE_PATH
```

```
Model Model_2023_07_26_00_53_52_.h5 will be saved at TRAINED_MODEL\Model_2023_07_26_00_
53_52_.h5
```

Out[33]:
```
'TRAINED_MODEL\\Model_2023_07_26_00_53_52_.h5'
```

In [34]:
```
tf.keras.models.save_model(model_clf, UNIQUE_PATH)
```

In [35]:
```
history.params
```

Out[35]:
```
{'verbose': 1, 'epochs': 30, 'steps': 1719}
```

In [36]:
```
# history.history
```

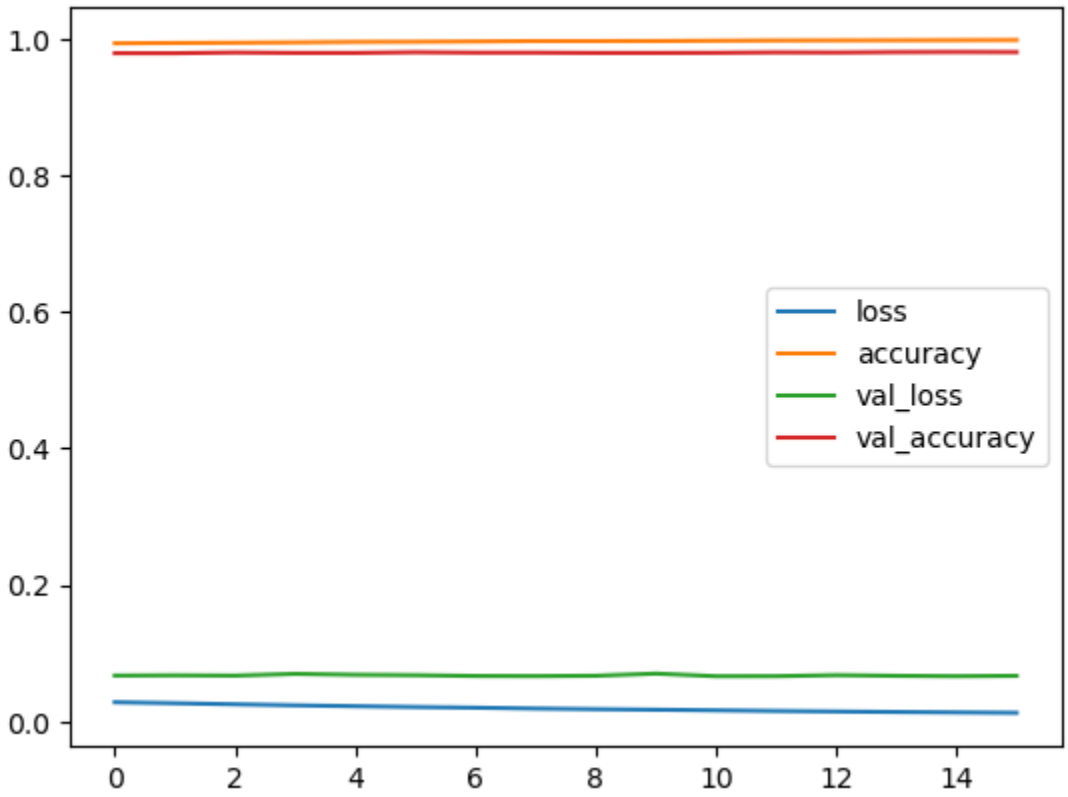
In [37]:
```
pd.DataFrame(history.history)
```

Out[37]:

	loss	accuracy	val_loss	val_accuracy
0	0.027838	0.993945	0.066974	0.9794
1	0.026399	0.994364	0.067307	0.9794
2	0.024844	0.994709	0.066889	0.9804
3	0.023314	0.995218	0.069436	0.9798
4	0.022058	0.995982	0.068107	0.9798
5	0.020838	0.996182	0.067498	0.9808
6	0.019833	0.996655	0.066364	0.9802
7	0.018600	0.997200	0.066153	0.9802
8	0.017650	0.997182	0.066678	0.9798
9	0.016868	0.997273	0.069837	0.9798
10	0.015999	0.997709	0.066010	0.9800
11	0.015030	0.998073	0.066082	0.9806
12	0.014267	0.998218	0.067568	0.9804
13	0.013561	0.998345	0.066585	0.9810
14	0.012970	0.998509	0.066026	0.9812
15	0.012334	0.998727	0.066566	0.9810

In [38]:
```
pd.DataFrame(history.history).plot()
```

Out[38]: <Axes: >



In [39]:
```
model_clf.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0693 - accuracy: 0.977
7
```

Out[39]:
```
[0.06929262727499008, 0.9776999950408936]
```

In [40]:
```
x_new = X_test[:3]
# x_new
```

In [41]:
```
actual = y_test[:3]
actual
```

Out[41]:
```
array([7, 2, 1], dtype=uint8)
```

In [42]:
```
y_prob = model_clf.predict(x_new)
y_prob.round(3)
```

```
1/1 [==============================] - 0s 302ms/step
```

Out[42]:
```
array([[0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 1.   , 0.   ,
        0.   ],
       [0.   , 0.   , 0.999, 0.001, 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.   , 0.998, 0.   , 0.   , 0.   , 0.   , 0.   , 0.001, 0.001,
        0.   ]], dtype=float32)
```

In [43]:
```
y_prob
```

Out[43]:
```
array([[1.5794069e-06, 7.6853473e-07, 1.0801427e-04, 1.3993130e-04,
        5.5980118e-09, 2.3529131e-07, 2.5731752e-12, 9.9971408e-01,
        9.0249375e-07, 3.4513163e-05],
       [2.5457732e-06, 1.9585187e-04, 9.9901319e-01, 7.7849202e-04,
        1.9284004e-12, 2.7663469e-07, 1.9941704e-06, 1.1564657e-09,
        7.6111187e-06, 3.1541395e-13],
       [2.6645846e-06, 9.9758554e-01, 6.5624081e-05, 3.2423293e-05,
        2.2758909e-04, 8.4465260e-05, 8.3897270e-05, 8.2147971e-04,
        1.0715602e-03, 2.4860985e-05]], dtype=float32)
```

In [44]:
```
y_pred = np.argmax(y_prob, axis = -1)
```

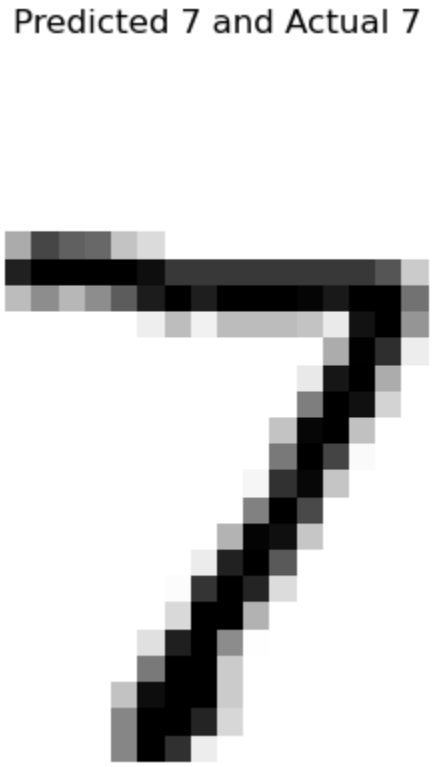
In [45]:
```
y_pred
```

Out[45]:
```
array([7, 2, 1], dtype=int64)
```

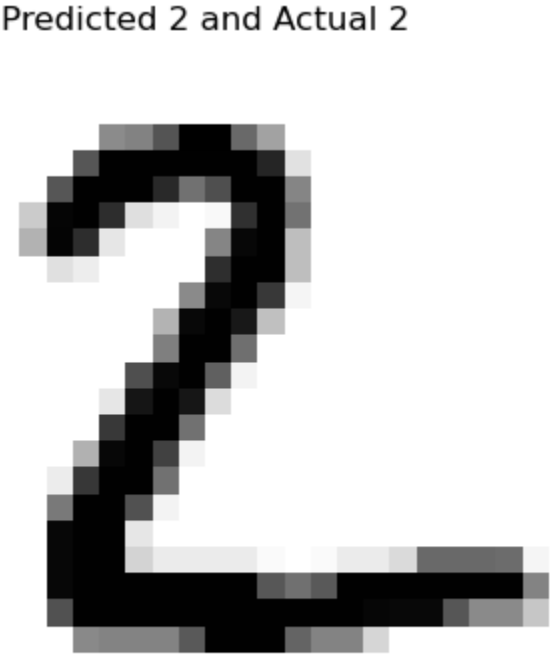
In [46]:
```
actual
```

Out[46]:
```
array([7, 2, 1], dtype=uint8)
```

In [47]:
```
# plot
for data, pred, actual_data in zip(x_new, y_pred, actual):
  plt.imshow(data, cmap="binary")
  plt.title(f"Predicted {pred} and Actual {actual_data}")
  plt.axis("off")
  plt.show()
  print("######################")
```

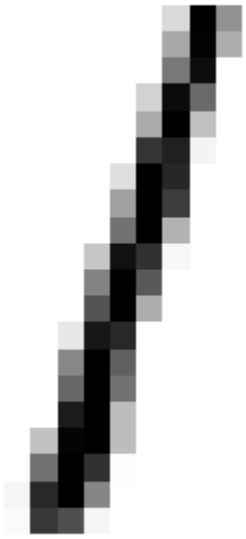


######################



######################

Predicted 1 and Actual 1



######################

In [ ]:

Assignment Questions

Q1. Explain the concept of batch normalization in the context of Artificial Neural Networks.

Batch normalization is a technique used in Artificial Neural Networks to normalize the inputs of each layer during training. It aims to stabilize and speed up the training process by reducing internal covariate shift. Internal covariate shift refers to the change in the distribution of each layer's inputs during training, which can slow down learning and require more careful tuning of hyperparameters.

Batch normalization addresses this issue by normalizing the inputs of each layer to have zero mean and unit variance. It does this by computing the mean and variance of the inputs within a mini-batch (a small subset of the training data) and then applying a normalization transformation. Additionally, batch normalization introduces learnable parameters, scale, and shift, which allow the network to learn the optimal mean and variance for each layer. These parameters give the model more flexibility to adapt to the data distribution.

Q2. Describe the benefits of using batch normalization during training.

Using batch normalization during training offers several benefits:

- Improved training stability: Batch normalization helps to mitigate the vanishing and exploding gradient problems, making it easier for deep neural networks to converge during training.
- Faster convergence: By reducing internal covariate shift, batch normalization accelerates the training process, leading to faster convergence and fewer training iterations required.
- Reduced sensitivity to initialization: Batch normalization makes neural networks less sensitive to the choice of weight initialization, allowing for a wider range of initialization strategies.
- Regularization effect: Batch normalization acts as a form of regularization, reducing the need for other regularization techniques like dropout.
- Allows for larger learning rates: The improved stability provided by batch normalization allows for the use of larger learning rates, which can speed up training further.

Q3. Discuss the working principle of batch normalization, including the normalization step and the learnable parameters.

The working principle of batch normalization involves two main steps: normalization and learnable parameters.

Normalization Step: For each mini-batch of input data during training, batch normalization computes the mean and variance of the data within that mini-batch. It then normalizes the data by subtracting the mean and dividing by the standard deviation, resulting in inputs with zero mean and unit variance:

x_normalized = (x - mean) / sqrt(variance + epsilon)

where x is the input data, mean and variance are the batch-wise statistics, and epsilon is a small constant added to avoid division by zero.

Learnable Parameters: Batch normalization introduces two learnable parameters for each layer: scale (gamma) and shift (beta). These parameters are applied after normalization and allow the model to learn the optimal scaling and shifting of the normalized inputs. The transformed output is given by:

y = gamma * x_normalized + beta

Impementation

In [1]:
```
# Importing Libraries
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os
```

Creating a simple classifier using keras on MNIST data

In [2]:
```
mnist = tf.keras.datasets.mnist
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
```

In [3]:
```
print(f"data type of X_train_full: {X_train_full.dtype},\n shape of X_train_full: {X_tra
```

```
data type of X_train_full: uint8,
 shape of X_train_full: (60000, 28, 28)
```

In [4]:
```
X_test.shape
```

Out[4]:
```
(10000, 28, 28)
```

In [5]:
```
len(X_test[1][0])
```

Out[5]:
```
28
```

In [6]:
```
X_train_full[0]
```

```
Out[6]: array([[  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,   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,   3,
         18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
        253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
        253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253,
        253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253,
        205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,
         90,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253,
        190,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190,
        253,  70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35,
        241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,  45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,  16,  93, 252, 253, 187,   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, 249, 253, 249,  64,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,  46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39,
        148, 229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,
```

```
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221,
 253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253,
 253, 253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253,
 195,  80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,
  11,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   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]], dtype=uint8)
```

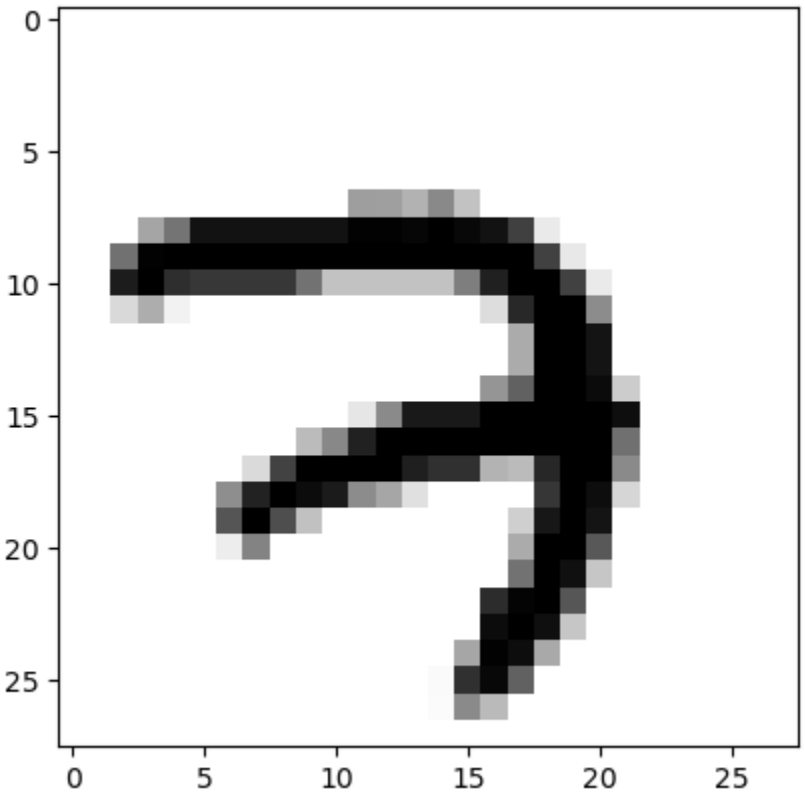
In [7]:
```
# create a validation data set from the full training data
# Scale the data between 0 to 1 by dividing it by 255. as its an unsigned data between 0
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

# scale the test set as well
X_test = X_test / 255.
```

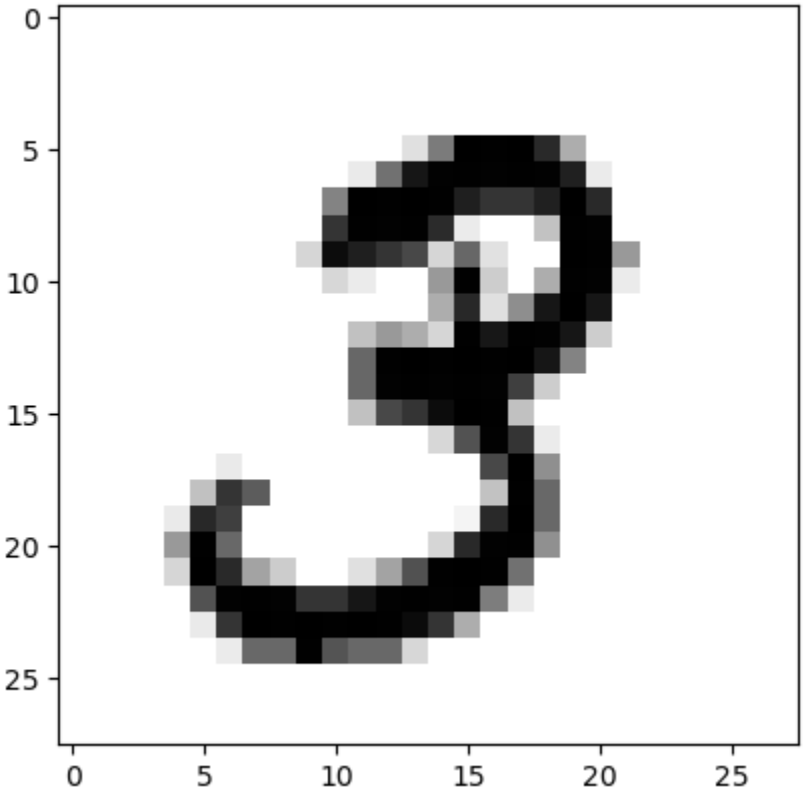
In [8]:
```
len(X_train_full[5000:] )
```

Out[8]:
```
55000
```

In [9]:
```
# Lets view some data
plt.imshow(X_train[0], cmap="binary")
plt.show()
```

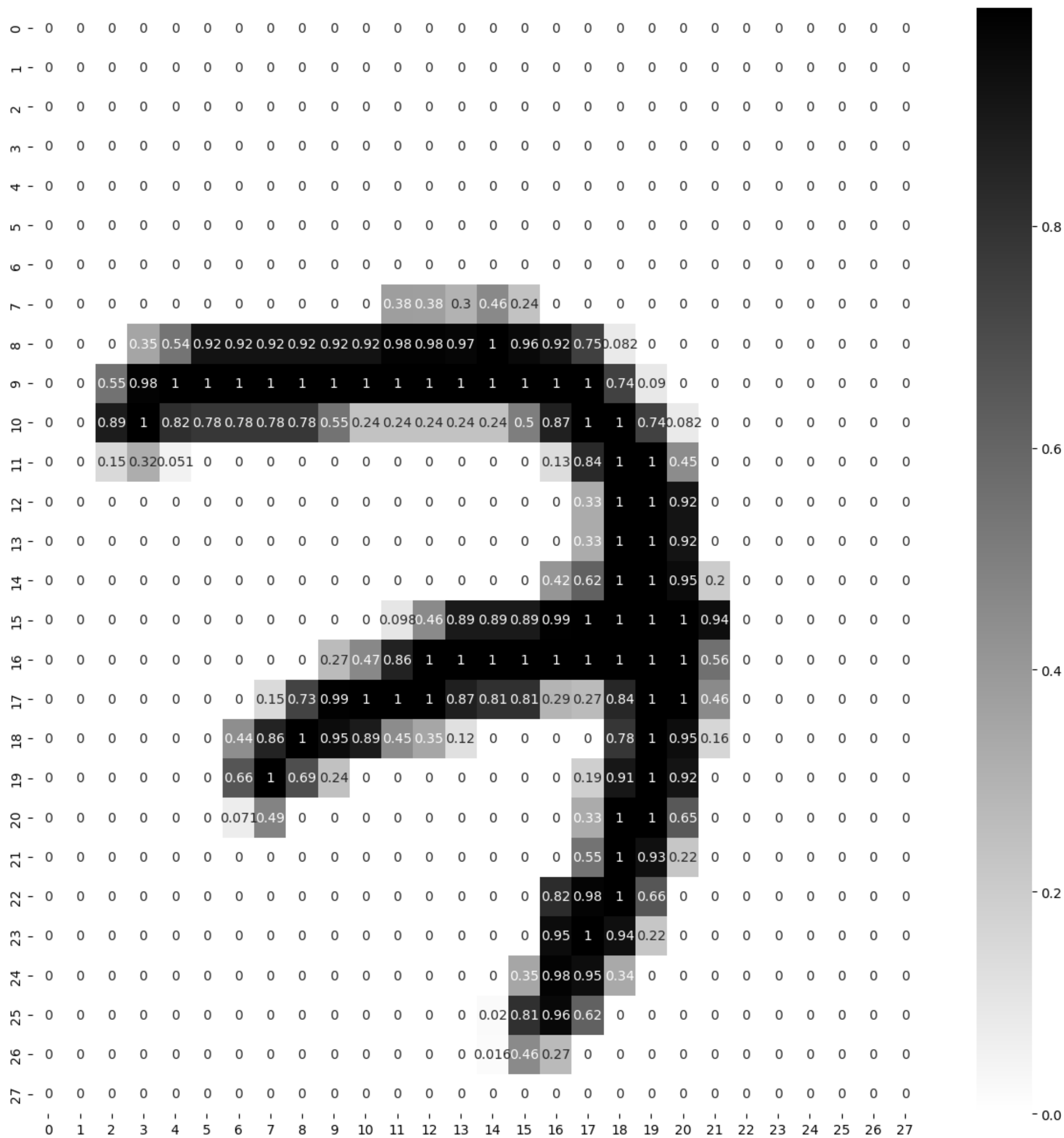


In [10]:
```
# Lets view some data
plt.imshow(X_train[1], cmap="binary")
plt.show()
```



In [11]:
```
plt.figure(figsize=(15,15))
sns.heatmap(X_train[0], annot=True, cmap="binary")
```

Out[11]: <Axes: >



In [ ]:
```
SO Input is : 28 * 28 :784 : flattern the input layeers

And we have 10 Output:[0,1,2,3,4,5,6,7,8,9]  ## Softmax is used their because it tell pr
```

Without Batch Normalization

In [12]:
```
# Creating Layers of ANN
LAYERS = [tf.keras.layers.Flatten(input_shape=[28, 28], name="inputLayer"),
          tf.keras.layers.Dense(300, activation="relu", name="hiddenLayer1"),
          tf.keras.layers.Dense(100, activation="relu", name="hiddenLayer2"),
          tf.keras.layers.Dense(10, activation="softmax", name="outputLayer")]

model_clf_without_bn = tf.keras.models.Sequential(LAYERS)
```

In [13]:
```
model_clf_without_bn.layers
```

Out[13]:
```
[<keras.layers.reshaping.flatten.Flatten at 0x21939ca6da0>,
 <keras.layers.core.dense.Dense at 0x21939ca6740>,
 <keras.layers.core.dense.Dense at 0x21939ca6500>,
 <keras.layers.core.dense.Dense at 0x21939ca43a0>]
```

In [14]:
```
model_clf_without_bn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inputLayer (Flatten)        (None, 784)               0

 hiddenLayer1 (Dense)        (None, 300)               235500

 hiddenLayer2 (Dense)        (None, 100)               30100

 outputLayer (Dense)         (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

In [15]:
```
# first Layer * second Layer + bias
784*300 + 300, 300*100+100, 100*10+100
```

Out[15]:
```
(235500, 30100, 1100)
```

In [16]:
```
# Total parameters to be trained
sum((235500, 30100, 1010))
```

Out[16]:
```
266610
```

In [17]:

```
LOSS_FUNCTION = "sparse_categorical_crossentropy" # use => tf.losses.sparse_categorical_
OPTIMIZER = "ADAM" # or use with custom learning rate=> tf.keras.optimizers.SGD(0.02)
METRICS = ["accuracy"]


# Record starting time
start_time = time.time()

model_clf_without_bn.compile(loss=LOSS_FUNCTION,
              optimizer=OPTIMIZER,
              metrics=METRICS)
```

In [18]:
```
# training

EPOCHS = 10
VALIDATION_SET = (X_valid, y_valid)

# Record starting time
start_time = time.time()

history = model_clf_without_bn.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=64)

# Record ending time
end_time = time.time()
 # Calculate training time
training_time = end_time - start_time
print(f"Total training time: {training_time:.2f} seconds")
```

```
Epoch 1/10
860/860 [==============================] - 6s 5ms/step - loss: 0.2387 - accuracy: 0.930
3 - val_loss: 0.1005 - val_accuracy: 0.9700
Epoch 2/10
860/860 [==============================] - 4s 4ms/step - loss: 0.0893 - accuracy: 0.972
8 - val_loss: 0.0895 - val_accuracy: 0.9714
Epoch 3/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0595 - accuracy: 0.981
7 - val_loss: 0.0751 - val_accuracy: 0.9762
Epoch 4/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0426 - accuracy: 0.986
1 - val_loss: 0.0703 - val_accuracy: 0.9814
Epoch 5/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0313 - accuracy: 0.989
2 - val_loss: 0.0843 - val_accuracy: 0.9780
Epoch 6/10
860/860 [==============================] - 5s 6ms/step - loss: 0.0278 - accuracy: 0.990
8 - val_loss: 0.0726 - val_accuracy: 0.9818
Epoch 7/10
860/860 [==============================] - 5s 6ms/step - loss: 0.0204 - accuracy: 0.993
3 - val_loss: 0.0752 - val_accuracy: 0.9812
Epoch 8/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0159 - accuracy: 0.994
8 - val_loss: 0.0866 - val_accuracy: 0.9808
Epoch 9/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0155 - accuracy: 0.994
6 - val_loss: 0.0835 - val_accuracy: 0.9776
Epoch 10/10
860/860 [==============================] - 4s 4ms/step - loss: 0.0135 - accuracy: 0.995
4 - val_loss: 0.0837 - val_accuracy: 0.9814
```

In [19]:
```
history.params
```

Out[19]:
```
{'verbose': 1, 'epochs': 10, 'steps': 860}
```

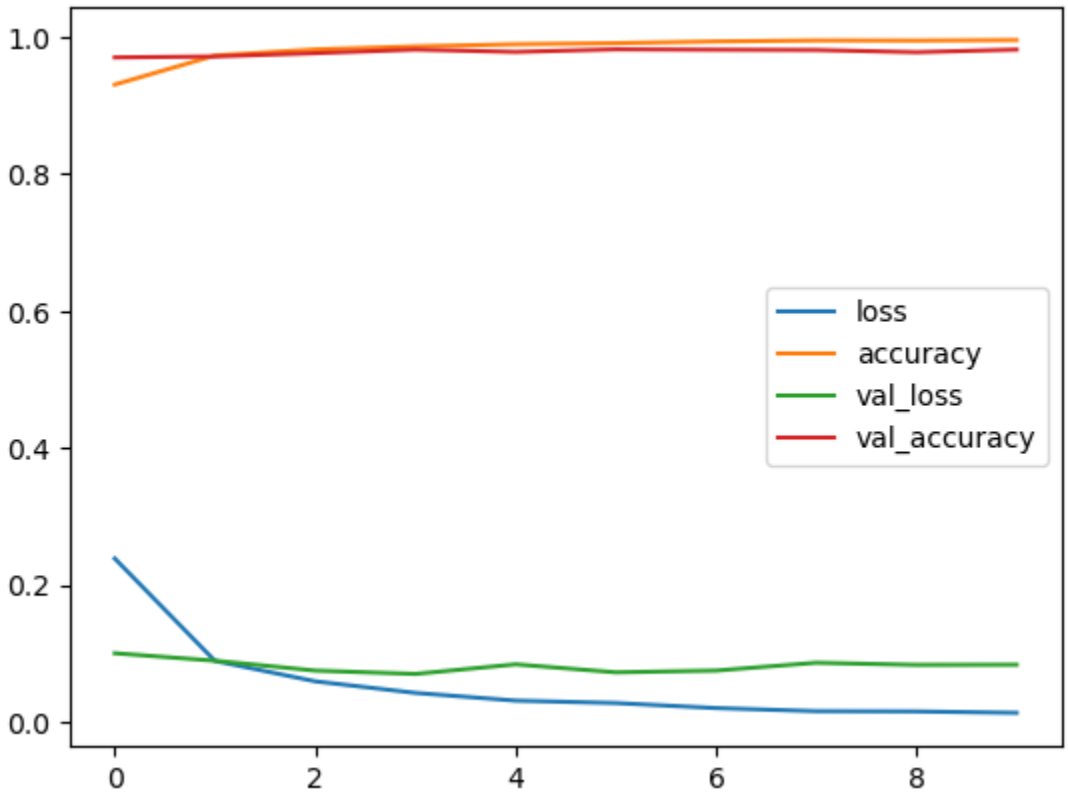
In [20]:
```
pd.DataFrame(history.history)
```

Out[20]:

	loss	accuracy	val_loss	val_accuracy
0	0.238738	0.930345	0.100475	0.9700
1	0.089264	0.972782	0.089536	0.9714
2	0.059478	0.981673	0.075137	0.9762
3	0.042577	0.986127	0.070268	0.9814
4	0.031306	0.989236	0.084280	0.9780
5	0.027780	0.990764	0.072608	0.9818
6	0.020448	0.993327	0.075167	0.9812
7	0.015920	0.994818	0.086569	0.9808
8	0.015546	0.994618	0.083470	0.9776
9	0.013514	0.995400	0.083705	0.9814

In [21]:
```
pd.DataFrame(history.history).plot()
```

Out[21]:
```
<Axes: >
```



In [23]:
```
# Evaluate both models on the test dataset
loss_without_bn, accuracy_without_bn = model_clf_without_bn.evaluate(X_test, y_test, ver

print("Model without Batch Normalization:")
print(f"Accuracy: {accuracy_without_bn*100:.2f}%, Loss: {loss_without_bn:.4f}")
```

```
Model without Batch Normalization:
Accuracy: 97.92%, Loss: 0.0877
```

With Batch Normalization

In [24]:
```
# Define the layers of the model with batch normalization
LAYERS1 = [tf.keras.layers.Flatten(input_shape=[28, 28], name="inputLayer"),
           tf.keras.layers.Dense(300, activation="relu", name="hiddenLayer1"),
           tf.keras.layers.BatchNormalization(name="batchNorm1"),
           tf.keras.layers.Dense(100, activation="relu", name="hiddenLayer2"),
           tf.keras.layers.BatchNormalization(name="batchNorm2"),
           tf.keras.layers.Dense(10, activation="softmax", name="outputLayer")]

model_clf_with_bn = tf.keras.models.Sequential(LAYERS1)
```

In [25]:
```
model_clf_with_bn.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inputLayer (Flatten)        (None, 784)               0

 hiddenLayer1 (Dense)        (None, 300)               235500

 batchNorm1 (BatchNormalizat  (None, 300)              1200
 ion)

 hiddenLayer2 (Dense)        (None, 100)               30100

 batchNorm2 (BatchNormalizat  (None, 100)              400
 ion)

 outputLayer (Dense)         (None, 10)                1010

=================================================================
Total params: 268,210
Trainable params: 267,410
Non-trainable params: 800
_________________________________________________________________
```

In [26]:
```
LOSS_FUNCTION = "sparse_categorical_crossentropy" # use => tf.losses.sparse_categorical_
OPTIMIZER = "ADAM" # or use with custom learning rate=> tf.keras.optimizers.SGD(0.02)
METRICS = ["accuracy"]



model_clf_with_bn.compile(loss=LOSS_FUNCTION,
              optimizer=OPTIMIZER,
              metrics=METRICS)
```

In [28]:
```
# training
import time

EPOCHS = 10
VALIDATION_SET = (X_valid, y_valid)

# Record starting time
start_time = time.time()

history = model_clf_with_bn.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=64)

# Record ending time
end_time = time.time()
# Calculate training time
training_time = end_time - start_time
print(f"Total training time: {training_time:.2f} seconds")
```

```
Epoch 1/10
860/860 [==============================] - 8s 7ms/step - loss: 0.2105 - accuracy: 0.937
2 - val_loss: 0.0989 - val_accuracy: 0.9702
Epoch 2/10
860/860 [==============================] - 6s 7ms/step - loss: 0.0910 - accuracy: 0.972
2 - val_loss: 0.0835 - val_accuracy: 0.9756
Epoch 3/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0654 - accuracy: 0.979
1 - val_loss: 0.0852 - val_accuracy: 0.9726
Epoch 4/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0488 - accuracy: 0.984
3 - val_loss: 0.0738 - val_accuracy: 0.9774
Epoch 5/10
860/860 [==============================] - 4s 5ms/step - loss: 0.0426 - accuracy: 0.985
8 - val_loss: 0.0756 - val_accuracy: 0.9786
Epoch 6/10
860/860 [==============================] - 5s 6ms/step - loss: 0.0369 - accuracy: 0.988
0 - val_loss: 0.0636 - val_accuracy: 0.9800
Epoch 7/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0288 - accuracy: 0.990
2 - val_loss: 0.0733 - val_accuracy: 0.9802
Epoch 8/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0281 - accuracy: 0.990
2 - val_loss: 0.0687 - val_accuracy: 0.9802
Epoch 9/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0224 - accuracy: 0.992
2 - val_loss: 0.0764 - val_accuracy: 0.9810
Epoch 10/10
860/860 [==============================] - 5s 5ms/step - loss: 0.0203 - accuracy: 0.993
0 - val_loss: 0.0688 - val_accuracy: 0.9802
Total training time: 49.95 seconds
```

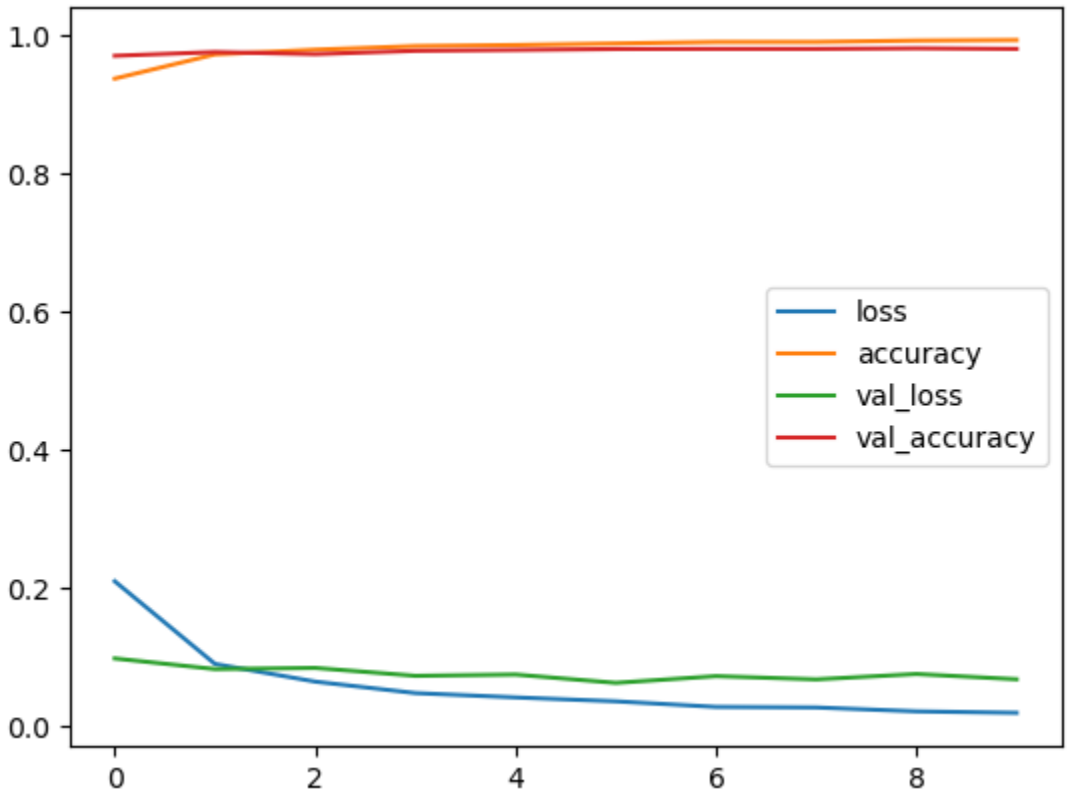
In [31]:
```
pd.DataFrame(history.history)
```

Out[31]:

	loss	accuracy	val_loss	val_accuracy
0	0.210480	0.937218	0.098913	0.9702
1	0.090966	0.972236	0.083534	0.9756
2	0.065417	0.979055	0.085216	0.9726
3	0.048829	0.984273	0.073797	0.9774
4	0.042590	0.985800	0.075644	0.9786
5	0.036925	0.988018	0.063586	0.9800
6	0.028844	0.990164	0.073334	0.9802
7	0.028057	0.990236	0.068650	0.9802
8	0.022416	0.992218	0.076421	0.9810
9	0.020310	0.993018	0.068799	0.9802

In [32]:
```
pd.DataFrame(history.history).plot()
```

Out[32]: <Axes: >



In [30]:
```
loss_with_bn, accuracy_with_bn = model_clf_with_bn.evaluate(X_test, y_test, verbose=0)

print("Model with Batch Normalization:")
print(f"Accuracy: {accuracy_with_bn*100:.2f}%, Loss: {loss_with_bn:.4f}")
```

```
Model with Batch Normalization:
Accuracy: 97.96%, Loss: 0.0748
```

SO the Difference is :

In [37]:
```
# Evaluate both models on the test dataset
loss_without_bn, accuracy_without_bn = model_clf_without_bn.evaluate(X_test, y_test, ver
loss_with_bn, accuracy_with_bn = model_clf_with_bn.evaluate(X_test, y_test, verbose=0)

print("Model without Batch Normalization:")
print(f"Accuracy: {accuracy_without_bn*100:.2f}%, Loss: {loss_without_bn:.4f}")

print("Model with Batch Normalization:")
print(f"Accuracy: {accuracy_with_bn*100:.2f}%, Loss: {loss_with_bn:.4f}")
```

```
Model without Batch Normalization:
Accuracy: 97.92%, Loss: 0.0877
Model with Batch Normalization:
Accuracy: 97.96%, Loss: 0.0748
```

The provided experiment results show the comparison of two models trained on the same dataset using different batch sizes. The models were trained with and without batch normalization. Let's analyze the effects of different batch sizes on training dynamics and model performance.

Experiment Results:

- Model without Batch Normalization:
- Accuracy: 97.92%
- Loss: 0.0877
- Model with Batch Normalization:
- Accuracy: 97.96%
- Loss: 0.0748

Observations:

Both models achieve high accuracy, indicating that they are able to effectively learn from the data and make accurate predictions on unseen samples. The model with batch normalization slightly outperforms the model without batch normalization in terms of accuracy and loss. This suggests that batch normalization has provided some improvement in the model's performance.

Testing The Model

In [39]:
```
x_new = X_test[:3]
x_new
```

Out[39]:
```
array([[[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.]]])
```

In [40]:
```
actual = y_test[:3]
actual
```

Out[40]:
```
array([7, 2, 1], dtype=uint8)
```

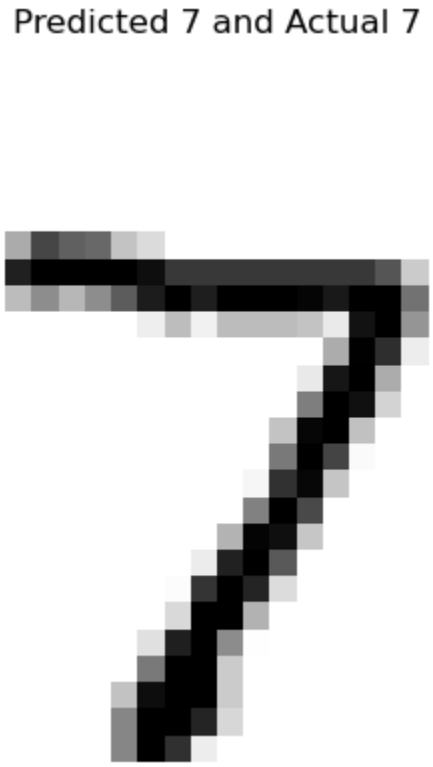
In [41]:
```
y_prob = model_clf_with_bn.predict(x_new)
y_prob.round(3)
```

```
1/1 [==============================] - 0s 307ms/step
```

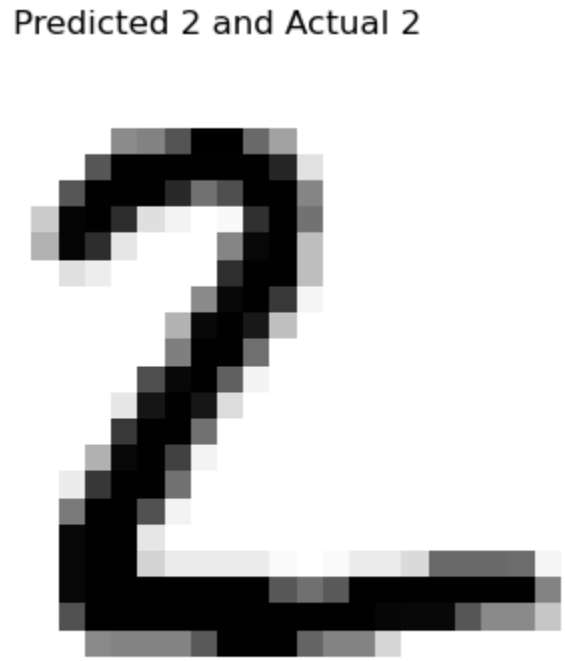
Out[41]:
```
array([[0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 1.   , 0.   ,
        0.   ],
       [0.   , 0.   , 1.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.   , 0.998, 0.   , 0.   , 0.   , 0.001, 0.   , 0.   , 0.   ,
        0.   ]], dtype=float32)
```

In [43]:
```
y_pred = np.argmax(y_prob, axis = -1)
```

In [44]:
```
# plot
for data, pred, actual_data in zip(x_new, y_pred, actual):
  plt.imshow(data, cmap="binary")
  plt.title(f"Predicted {pred} and Actual {actual_data}")
  plt.axis("off")
  plt.show()
  print("######################")
```

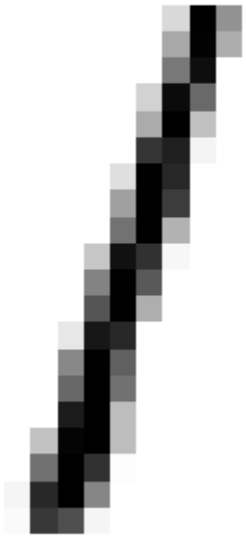



```
######################
```



######################

Predicted 1 and Actual 1



######################

----------**Done**----------

Assignment Questions

In [34]:
```
# Importing Libraries
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [35]:
```
# Checking version of Tensorflow and Keras
print(f"Tensorflow Version {tf.__version__}")
print(f"Keras Version {tf.keras.__version__}")
```

```
Tensorflow Version 2.10.0
Keras Version 2.10.0
```

In [36]:
```
df=pd.read_csv("wine.csv")
```

In [37]:
```
df.head()
```

Out[37]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	bad
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	bad
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	bad
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	good
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	bad

In [38]:
```
df.shape
```

Out[38]:
```
(1599, 12)
```

In [39]:
```
df.columns
```

Out[39]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

In [40]:
```
# Calculate class distribution
class_distribution = df['quality'].value_counts()

# Display the class distribution
print(class_distribution)

# Check if the target variable is imbalanced
if len(class_distribution) == 2:
    majority_class_count = max(class_distribution)
    minority_class_count = min(class_distribution)
    class_ratio = majority_class_count / minority_class_count

    if class_ratio > 2:
        print("The target variable is imbalanced.")
    else:
        print("The target variable is not imbalanced.")
else:
    print("The target variable is not binary.")
```

```
good    855
bad     744
Name: quality, dtype: int64
The target variable is not imbalanced.
```

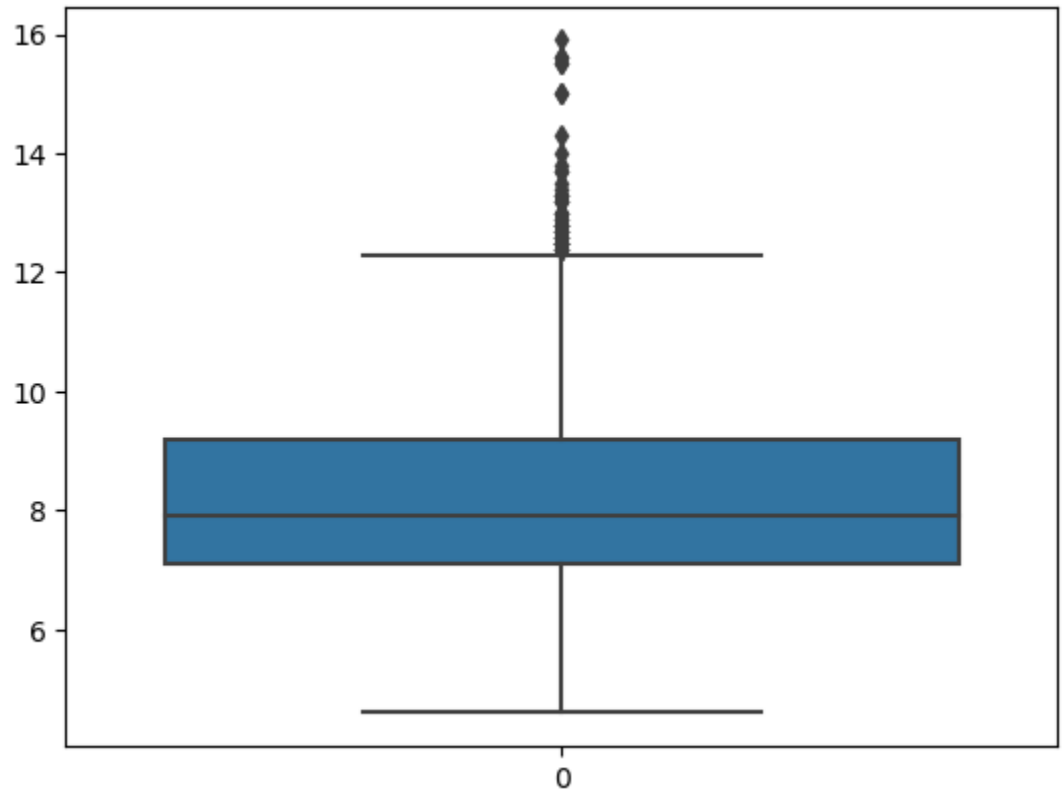
As Our target vaiable is not imbalanced

In [41]:
```
print('Checking for Null values in the dataframe:','\n',df.isnull().sum(),'\n')
```

```
Checking for Null values in the dataframe:
 fixed acidity          0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

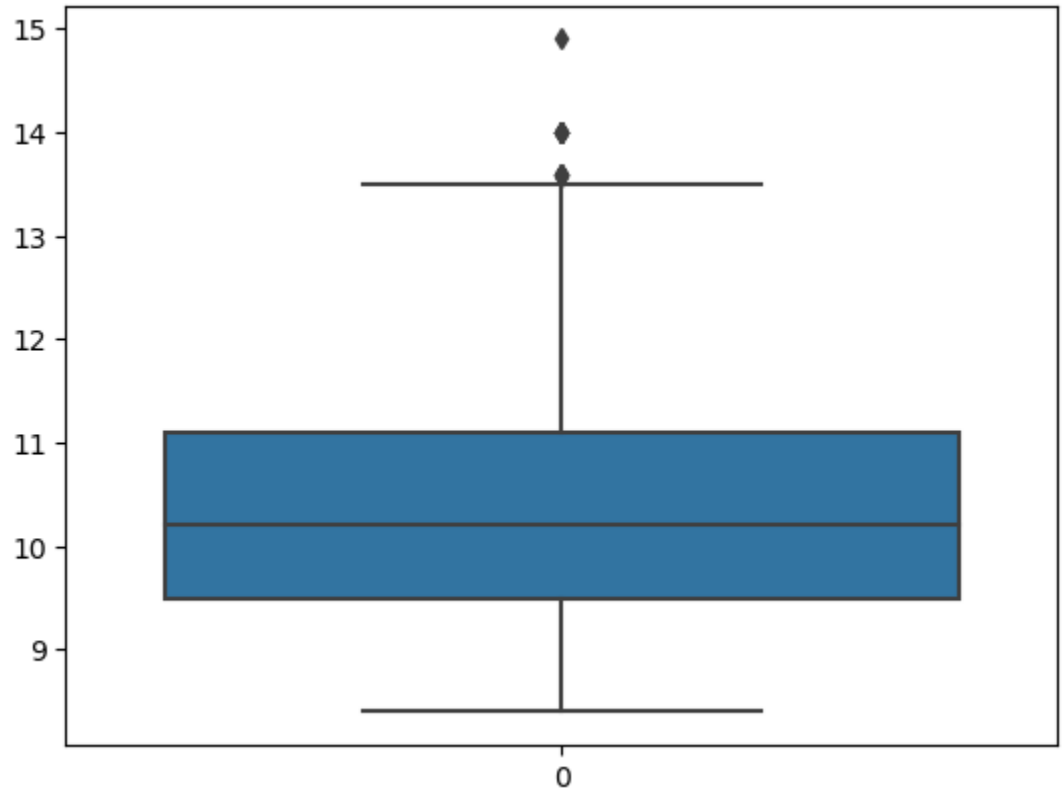
In [42]:
```
sns.boxplot(df['fixed acidity'])
```

Out[42]: <Axes: >



In [43]:
```
sns.boxplot(df['alcohol'])
```

Out[43]: <Axes: >



As we have some Outliers in data , But we are using Deep Learning Model so dont worry

In [44]:
```
y = df.quality
X = df.drop(columns = ['quality'])
```

In [45]:
```
X.shape
```

Out[45]:
```
(1599, 11)
```

In [46]:
```
X.head()
```

Out[46]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4

In [47]:
```
y.head()
```

Out[47]:
```
0     bad
1     bad
2     bad
3    good
4     bad
Name: quality, dtype: object
```

As we have our Target Feature in "Good" or "Bad" so we use Label encoder

In [48]:
```
# Let's perform categorical features encoding:
from sklearn.preprocessing import LabelEncoder
LE = LabelEncoder()
```

In [49]:
```
# Target_feature Encoding:
y = LE.fit_transform(y)
```

In [50]:
```
y  # 0= bad 1=Good
```

Out[50]:
```
array([0, 0, 0, ..., 1, 0, 1])
```

In [51]:
```
X_train_full, X_test, y_train_full, y_test = train_test_split(X,y, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full,y_train_full, random_
```

In [52]:
```
print(X_train_full.shape)
print(X_test.shape)
print(X_train.shape)
print(X_valid.shape)
```

```
(1199, 11)
(400, 11)
(899, 11)
(300, 11)
```

In [53]:
```
X_train.shape[1:]
```

Out[53]: (11,)

In [54]:
```
X_train.shape[1:]
```

Out[54]: (11,)

In [55]:
```
y_train.shape[:1]
```

Out[55]: (899,)

In [56]:
```
# Creating layers of ANN
LAYERS = [
    tf.keras.layers.Dense(30, activation="relu", name="HiddenLayer1", input_shape=X_trai
    tf.keras.layers.Dense(10, activation="relu", name="HiddenLayer2"),
    tf.keras.layers.Dense(5, activation='relu', name="HiddenLayer3"),
    tf.keras.layers.Dense(1, activation="sigmoid", name="OutputLayer")
]

# Create the model using Sequential API
model = tf.keras.models.Sequential(LAYERS)
```

In [57]:
```
# Logging

import time

def get_log_path(log_dir="logs/fit"):
  fileName = time.strftime("log_%Y_%m_%d_%H_%M_%S")
  logs_path = os.path.join(log_dir, fileName)
  print(f"Saving logs at {logs_path}")
  return logs_path

log_dir = get_log_path()
tb_cb = tf.keras.callbacks.TensorBoard(log_dir=log_dir)
```

```
Saving logs at logs/fit\log_2023_07_31_21_39_54
```

In [58]:
```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=5, restore_best_weights=Tr
```

In [59]:
```
CKPT_path = "Model_ckpt.h5"
checkpointing_cb = tf.keras.callbacks.ModelCheckpoint(CKPT_path, save_best_only=True)
```

In [60]:
```
# Q13. Use binary cross-entropy as the loss function, Adam optimizer, and ['accuracy'] a
loss_function = 'binary_crossentropy'
optimizer = 'adam'
metrics = ['accuracy']

# Q14. Compile the model
model.compile(optimizer=optimizer, loss=loss_function, metrics=metrics)
```

In [61]:
```
# Q12. Print the model summary
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 HiddenLayer1 (Dense)        (None, 30)                360

 HiddenLayer2 (Dense)        (None, 10)                310

 HiddenLayer3 (Dense)        (None, 5)                 55

 OutputLayer (Dense)         (None, 1)                 6

=================================================================
Total params: 731
Trainable params: 731
Non-trainable params: 0
_________________________________________________________________
```

In [62]:
```
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

In [68]:
```
# Orginal train

EPOCHS = 40
VALIDATION_SET = (X_valid, y_valid)

history = model.fit(X_train, y_train, epochs=EPOCHS,
                    validation_data=VALIDATION_SET, batch_size=64, callbacks=[tb_cb, ear
```

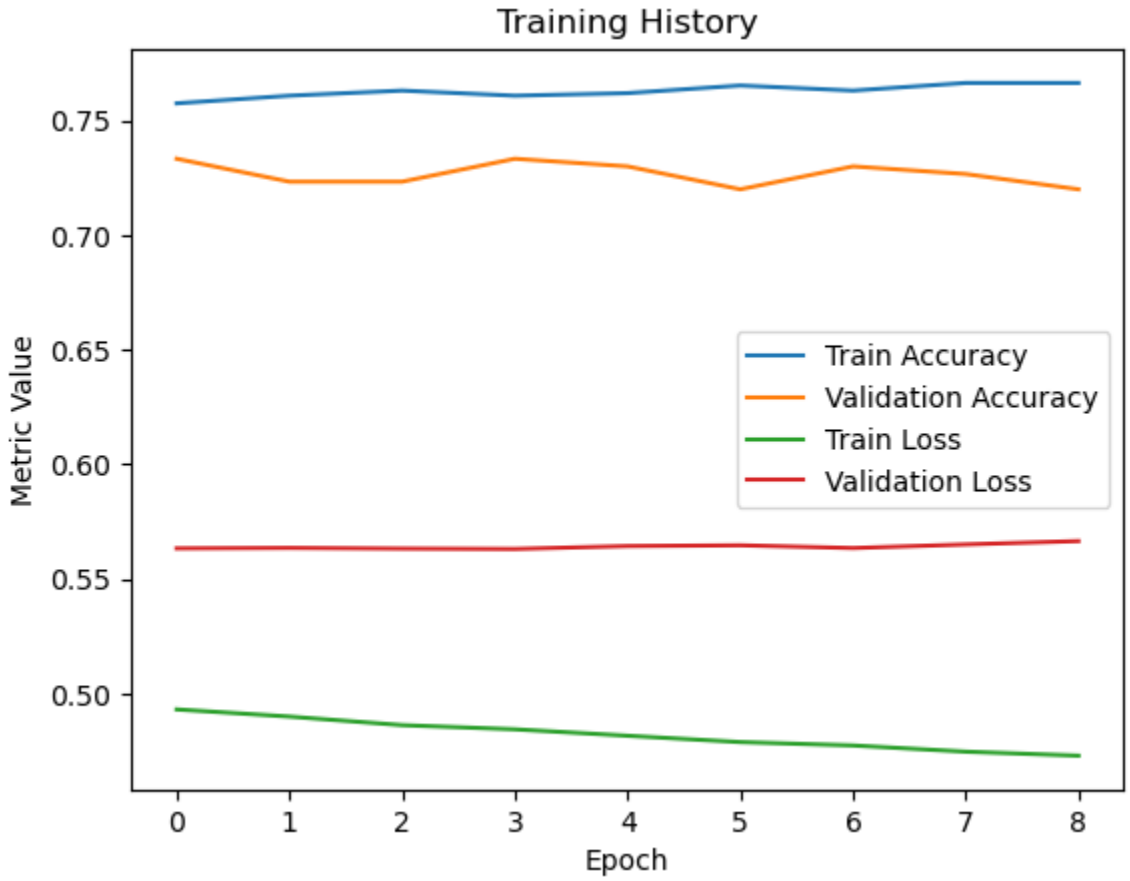
```
Epoch 1/40
15/15 [==============================] - 1s 20ms/step - loss: 0.4931 - accuracy: 0.7575
- val_loss: 0.5633 - val_accuracy: 0.7333
Epoch 2/40
15/15 [==============================] - 0s 11ms/step - loss: 0.4900 - accuracy: 0.7608
- val_loss: 0.5635 - val_accuracy: 0.7233
Epoch 3/40
15/15 [==============================] - 0s 12ms/step - loss: 0.4862 - accuracy: 0.7631
- val_loss: 0.5632 - val_accuracy: 0.7233
Epoch 4/40
15/15 [==============================] - 0s 17ms/step - loss: 0.4844 - accuracy: 0.7608
- val_loss: 0.5631 - val_accuracy: 0.7333
Epoch 5/40
15/15 [==============================] - 0s 11ms/step - loss: 0.4816 - accuracy: 0.7620
- val_loss: 0.5643 - val_accuracy: 0.7300
Epoch 6/40
15/15 [==============================] - 0s 12ms/step - loss: 0.4788 - accuracy: 0.7653
- val_loss: 0.5647 - val_accuracy: 0.7200
Epoch 7/40
15/15 [==============================] - 0s 12ms/step - loss: 0.4773 - accuracy: 0.7631
- val_loss: 0.5635 - val_accuracy: 0.7300
Epoch 8/40
15/15 [==============================] - 0s 12ms/step - loss: 0.4746 - accuracy: 0.7664
- val_loss: 0.5651 - val_accuracy: 0.7267
Epoch 9/40
15/15 [==============================] - 0s 14ms/step - loss: 0.4729 - accuracy: 0.7664
- val_loss: 0.5665 - val_accuracy: 0.7200
```

In [69]:
```
# Q16. Get the model's parameters
model_params = model.get_weights()

# Q17. Store the model's training history as a Pandas DataFrame
history_df = pd.DataFrame(history.history)
```

In [70]:
```
# Q18. Plot the model's training history
plt.plot(history_df['accuracy'], label='Train Accuracy')
plt.plot(history_df['val_accuracy'], label='Validation Accuracy')
plt.plot(history_df['loss'], label='Train Loss')
plt.plot(history_df['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Metric Value')
plt.title('Training History')
plt.show()

# Q19. Evaluate the model's performance using the test data
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}')
```




```
13/13 [==============================] - 0s 4ms/step - loss: 0.5381 - accuracy: 0.7400
Test Loss: 0.5381, Test Accuracy: 0.7400
```

In [71]:
```
X_test.shape
```

Out[71]:
```
(400, 11)
```

In [72]:
```
new = X_test[0]
```

In [74]:
```
new.reshape((1,11))
```

Out[74]:
```
array([[-0.34408707,  0.15940242, -0.98437473, -0.02598979,  0.46615703,
        -0.18979812, -0.0378264 ,  0.21710754, -0.46498919, -0.02247738,
        -0.78903789]])
```

In [81]:
```
model.predict(new.reshape((1,11)))
```

```
1/1 [==============================] - 0s 64ms/step
```

Out[81]:
```
array([[0.39237672]], dtype=float32)
```

In [83]:
```
# Assuming you have already trained the model and loaded the test data (X_test, y_test)

# Make predictions on the test data using the trained model
y_pred_probs = model.predict(X_test)

# Convert the predicted probabilities to class labels (0 or 1)
y_pred_labels = (y_pred_probs > 0.5).astype(int)

# If you have used the sigmoid activation function in the output layer
# and want to predict the class with the highest probability directly:
# y_pred_labels = np.argmax(y_pred_probs, axis=1)

# Evaluate the model's performance
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred_labels)

# Create a confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_labels)

# Print the classification report
class_report = classification_report(y_test, y_pred_labels)

print("Accuracy:", accuracy)
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(class_report)
```

```
13/13 [==============================] - 0s 4ms/step
Accuracy: 0.74
Confusion Matrix:
[[137  41]
 [ 63 159]]
Classification Report:
              precision    recall  f1-score   support

           0       0.69      0.77      0.72       178
           1       0.80      0.72      0.75       222

    accuracy                           0.74       400
   macro avg       0.74      0.74      0.74       400
weighted avg       0.75      0.74      0.74       400
```

In [87]:
```
# Assuming you have already trained the model and loaded the test data (X_test)

# Make predictions on the test data using the trained model
y_pred_probs = model.predict(X_test)

# Verify the shape of the y_pred_probs array
print("Shape of y_pred_probs:", y_pred_probs.shape)

# Assuming '0' represents 'Bad' and '1' represents 'Good',
# you can access the probability of 'Good' wine for the first sample (index 0) as follow
prob_good = y_pred_probs[0][0]

print("Probability of Good Wine:", prob_good)
```

```
13/13 [==============================] - 0s 4ms/step
Shape of y_pred_probs: (400, 1)
Probability of Good Wine: 0.39237672
```

In [ ]:
```

```

Assignment Question

Q1. What is the purpose of forward propagation in a neural network?

Answer: The purpose of forward propagation in a neural network is to compute the output of the network based on given input data. It involves passing the input through the network's layers, applying weights and biases to the data, and activating neurons using specific functions until the final output is generated.

Q2. How is forward propagation implemented mathematically in a single-layer feedforward neural network?

Answer: In a single-layer feedforward neural network, the forward propagation process can be mathematically represented as follows:

Input: The input features are denoted as a vector x = [x_1, x_2, ..., xn]. Weighted Sum: Each input feature is multiplied by its corresponding weight, and the biases are added to the weighted sum. Activation Function: The weighted sum is then passed through an activation function, producing the output of the neuron.

Q3. How are activation functions used during forward propagation?

Answer: Activation functions are applied during forward propagation to introduce non-linearity in the neural network, allowing it to learn complex patterns and make predictions for more diverse datasets. The activation function takes the output of a neuron and transforms it into a new value, which becomes the input for the next layer in the network.

Q4. What is the role of weights and biases in forward propagation?

Answer: The weights and biases are crucial parameters in forward propagation. The weights determine the strength of connections between neurons, controlling how much influence each input has on the neuron's output. Biases, on the other hand, shift the output of the activation function and allow the network to learn from different parts of the data distribution.

Q5. What is the purpose of applying a softmax function in the output layer during forward propagation?

Answer: The softmax function is typically applied in the output layer of a neural network when dealing with multi-class classification problems. It converts the raw output scores (logits) of the network into probabilities. The softmax function ensures that the output probabilities sum up to 1, making it easier to interpret the model's certainty about each class.

Q6. What is the purpose of backward propagation in a neural network?

Answer: The purpose of backward propagation, also known as backpropagation, is to adjust the network's weights and biases based on the computed error during forward propagation. By propagating the error backward through the network, it allows the model to learn and improve its performance through the process of gradient descent.

Q7. How is backward propagation mathematically calculated in a single-layer feedforward neural network?

Answer: In a single-layer feedforward neural network, backward propagation involves calculating the gradients of the loss function with respect to the weights and biases. These gradients indicate how the weights and biases should be adjusted to minimize the error. The chain rule is used to compute these gradients layer-by-layer, starting from the output layer and moving backward towards the input layer.

Q8. Can you explain the concept of the chain rule and its application in backward propagation?

Answer: The chain rule is a fundamental concept in calculus that allows us to find the derivative of a composite function. In the context of neural networks, it enables us to compute the gradients of the loss function with respect to the weights and biases of each layer. During backward propagation, the chain rule is applied to calculate how the changes in the output of a layer affect the error, and these gradients are used to update the parameters of the network through gradient descent.

Q9. What are some common challenges or issues that can occur during backward propagation, and how can they be addressed?

Answer: Some common challenges or issues during backward propagation are:

- Vanishing gradients: When gradients become very small, hindering the learning process. This can be mitigated using activation functions that preserve gradients, like ReLU.
- Exploding gradients: When gradients become extremely large, causing instability during learning. Techniques like gradient clipping can be applied to control the magnitude of gradients.
- Overfitting: When the model becomes too complex and performs well on training data but poorly on unseen data. Regularization techniques, such as L1 or L2 regularization, can be used to address this problem.
- Learning rate selection: Choosing an appropriate learning rate is essential for stable and efficient learning. Techniques like learning rate scheduling or adaptive learning rate methods (e.g., Adam) can be used.
- Local minima: The optimization process may get stuck in local minima, leading to suboptimal solutions