In [2]:
```python
from keras.layers import Dense , Flatten
from keras.models import Sequential
from keras.utils import to_categorical
from keras.datasets import mnist
from keras import utils
import matplotlib.pyplot as plt
from keras.utils import plot_model
import pydot
import graphviz
import numpy as np
```

In [3]:
```python
#load the dataset in keras
(x_train,y_train),(x_test,y_test)=mnist.load_data()
```

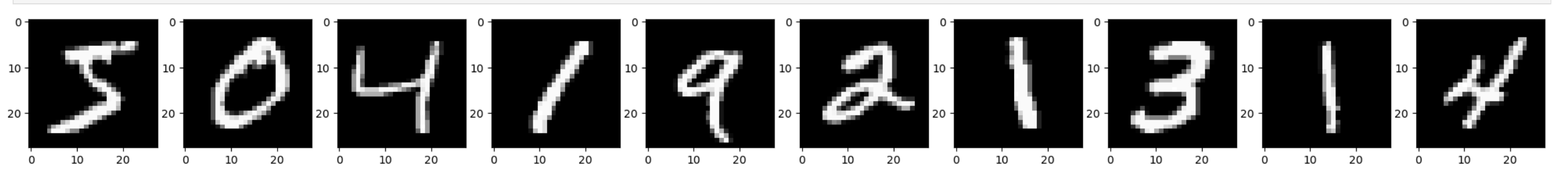
In [4]:
```python
# Get the unique labels
unique_labels = np.unique(y_train)
print("Unique labels in y_train:", unique_labels)
```

```
Unique labels in y_train: [0 1 2 3 4 5 6 7 8 9]
```

In [5]:
```python
y_train.shape
```

Out[5]: (60000,)

In [6]:
```python
fig,axis=plt.subplots(ncols=10,figsize=(25,5))

for i in range(10):
    axis[i].imshow(x_train[i],cmap='gray')
plt.show()
```



In [12]:
```python
y_test.shape
```

Out[12]: (10000,)

you're working with a multi-class classification problem with target labels like 0, 1, and 2, converting them to categorical data would transform them into one-hot encoded vectors:

0 -> [1, 0, 0]

1 -> [0, 1, 0]

2 -> [0, 0, 1]

In [8]:
```python
#convert to categorical value
y_test=to_categorical(y_test)
y_train=to_categorical(y_train)
```

In [9]:
```python
y_train.shape
```

Out[9]: (60000, 10)

In [10]:
```python
# WHy Do we need Flatten Layer
# Flattening converts these multidimensional outputs into a 1D vector, making it compatible with fully connected (dense) layers that follow.
# Fully connected layers require a 1D input, and the Flatten layer provides this.
# Convolutional and pooling layers work on 2D spatial data.
# Flatten layer converts the 2D outputs from the last pooling layer into a 1D vector.
# This 1D vector is then fed into fully connected layers for further processing.
# Without the Flatten layer, it would be challenging to transition from the convolutional layers to the fully connected layers,
# as their input formats are incompatible.

# Dense layer ?
# Dense layers are fundamental building blocks in neural networks, allowing the network to learn and make decisions based on the data.
```

In [11]:
```python
model=Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(10,activation='sigmoid'))
model.add(Dense(5,activation='sigmoid'))
model.add(Dense(10,activation='softmax'))
```

```
C:\Users\patil\anaconda3\Lib\site-packages\keras\src\layers\reshaping\flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

In [12]:
```python
model.summary()
```

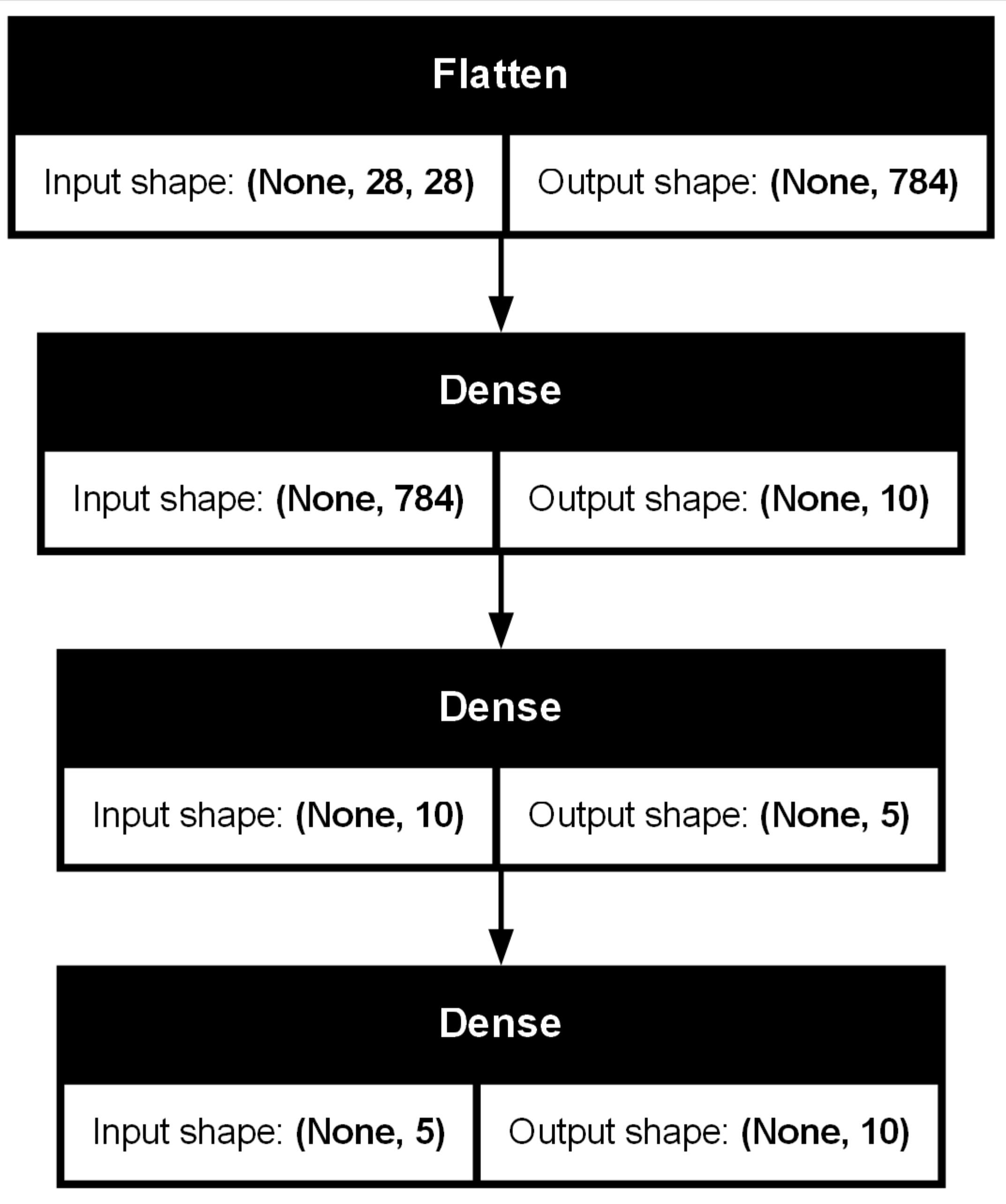
Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 10)	7,850
dense_1 (Dense)	(None, 5)	55
dense_2 (Dense)	(None, 10)	60

Total params: 7,965 (31.11 KB)
Trainable params: 7,965 (31.11 KB)
Non-trainable params: 0 (0.00 B)

In [13]:
```python
utils.plot_model(model,show_shapes=True)
```

Out[13]:



In [14]:
```python
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

In [15]:
```python
history=model.fit(x_train,y_train,epochs=10,validation_data=(x_test,y_test))
```

```
Epoch 1/10
1875/1875 ──────────── 4s 1ms/step - accuracy: 0.4089 - loss: 2.0113 - val_accuracy: 0.6467 - val_loss: 1.2520
Epoch 2/10
1875/1875 ──────────── 3s 1ms/step - accuracy: 0.6608 - loss: 1.1647 - val_accuracy: 0.7241 - val_loss: 0.9412
Epoch 3/10
1875/1875 ──────────── 2s 1ms/step - accuracy: 0.7132 - loss: 0.9385 - val_accuracy: 0.7398 - val_loss: 0.8250
Epoch 4/10
1875/1875 ──────────── 2s 1ms/step - accuracy: 0.7490 - loss: 0.8268 - val_accuracy: 0.7412 - val_loss: 0.8020
Epoch 5/10
1875/1875 ──────────── 2s 1ms/step - accuracy: 0.7549 - loss: 0.7933 - val_accuracy: 0.7737 - val_loss: 0.7595
Epoch 6/10
1875/1875 ──────────── 2s 1ms/step - accuracy: 0.7787 - loss: 0.7454 - val_accuracy: 0.7649 - val_loss: 0.7426
Epoch 7/10
1875/1875 ──────────── 2s 1ms/step - accuracy: 0.7670 - loss: 0.7364 - val_accuracy: 0.7791 - val_loss: 0.7257
Epoch 8/10
1875/1875 ──────────── 2s 1ms/step - accuracy: 0.7777 - loss: 0.7197 - val_accuracy: 0.7606 - val_loss: 0.7131
Epoch 9/10
1875/1875 ──────────── 2s 1ms/step - accuracy: 0.7734 - loss: 0.7166 - val_accuracy: 0.7884 - val_loss: 0.6709
Epoch 10/10
1875/1875 ──────────── 2s 1ms/step - accuracy: 0.7878 - loss: 0.6786 - val_accuracy: 0.7938 - val_loss: 0.6831
```

Understanding Steps Per Epoch:

An epoch refers to one complete pass through the entire training dataset.

When your dataset is too large to fit into memory, you typically use a data generator that yields batches of data.

Steps Per Epoch:

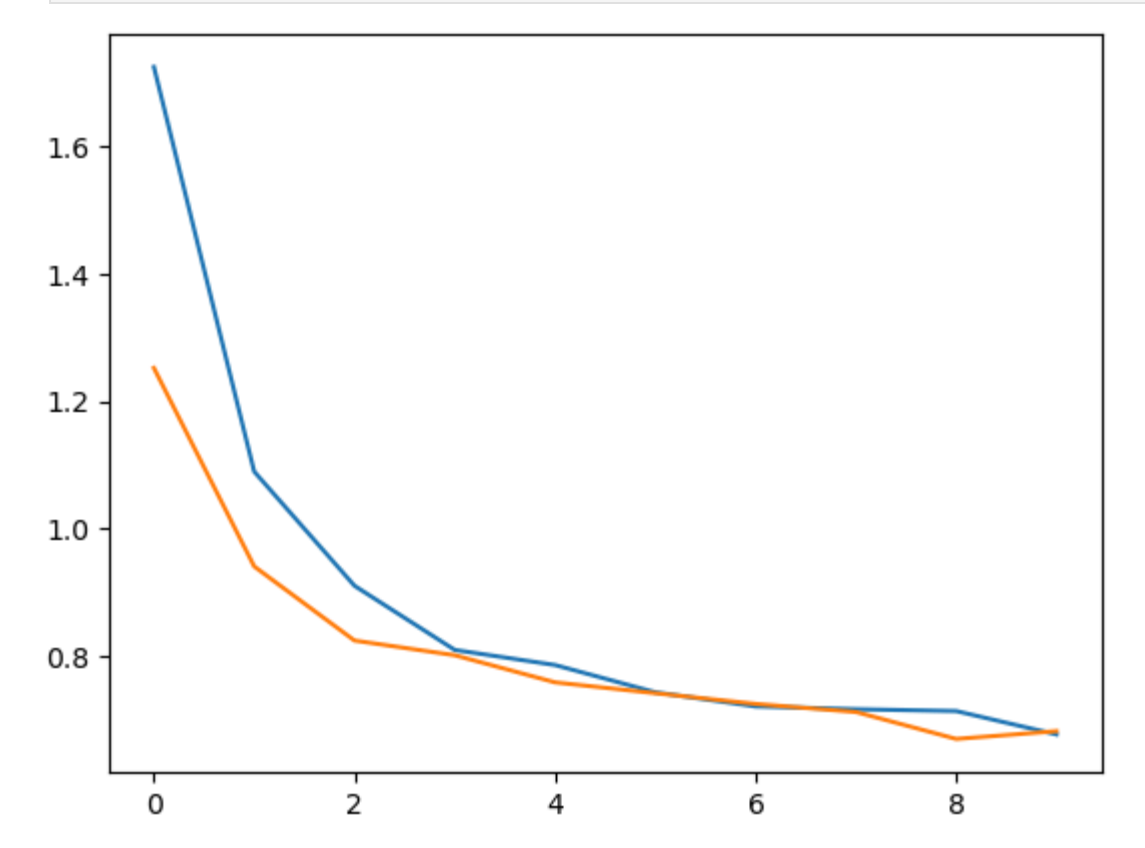
This is the number of batches of samples that the model will see during one epoch.

steps_per_epoch ensures that each epoch processes all the data in a dataset,

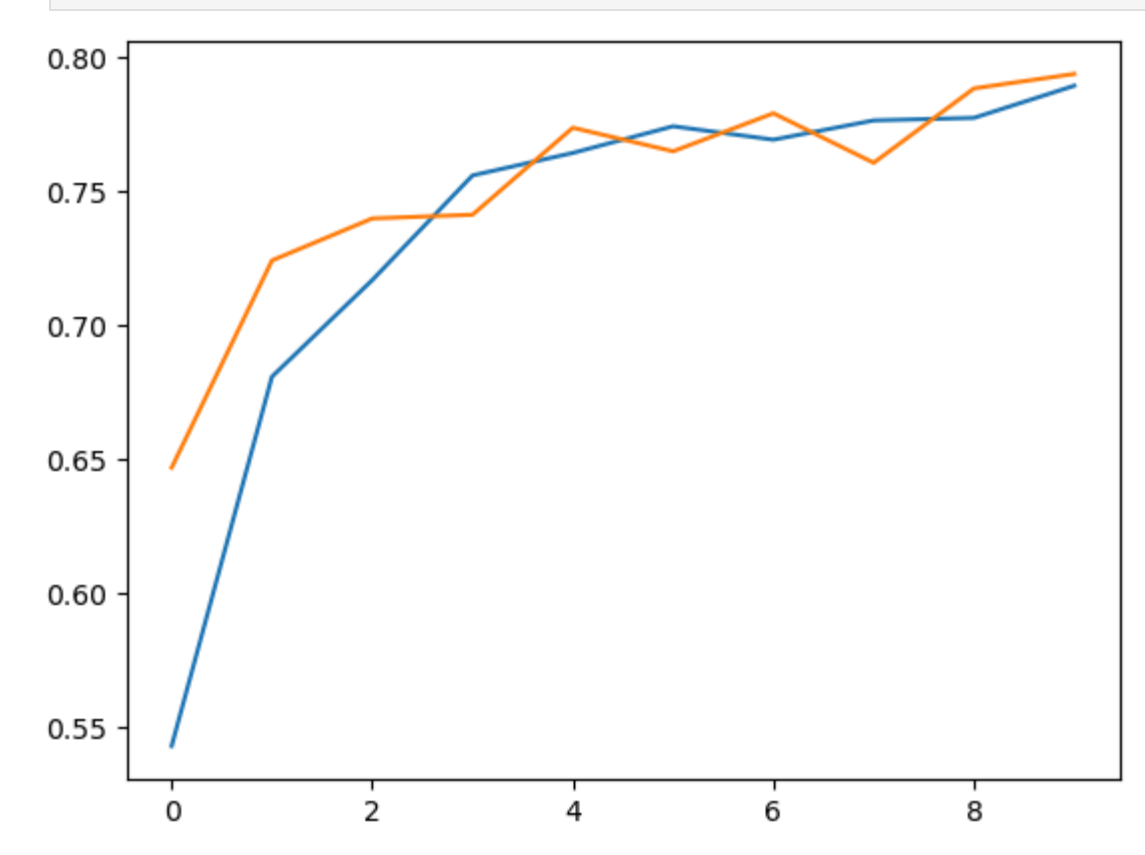
In [17]:
```python
import pandas as pd

# Assuming history is the History object returned by model.fit
history_dict = history.history

# Convert the dictionary to a DataFrame
history_df = pd.DataFrame(history_dict)

# Display the DataFrame
print(history_df)
```

```
   accuracy      loss  val_accuracy  val_loss
0  0.542700  1.723597        0.6467  1.252000
1  0.680667  1.089792        0.7241  0.941152
2  0.716783  0.910841        0.7398  0.825042
3  0.755883  0.810309        0.7412  0.802021
4  0.764283  0.786692        0.7737  0.759537
5  0.774250  0.743969        0.7649  0.742552
6  0.769233  0.721826        0.7791  0.725750
7  0.776433  0.717488        0.7606  0.713133
8  0.777383  0.714564        0.7884  0.670897
9  0.789400  0.677953        0.7938  0.683104
```

In [18]:
```python
plt.plot(history.history['loss'],label="Training Loss")
plt.plot(history.history['val_loss'],label="Validation Loss")
plt.show()
```



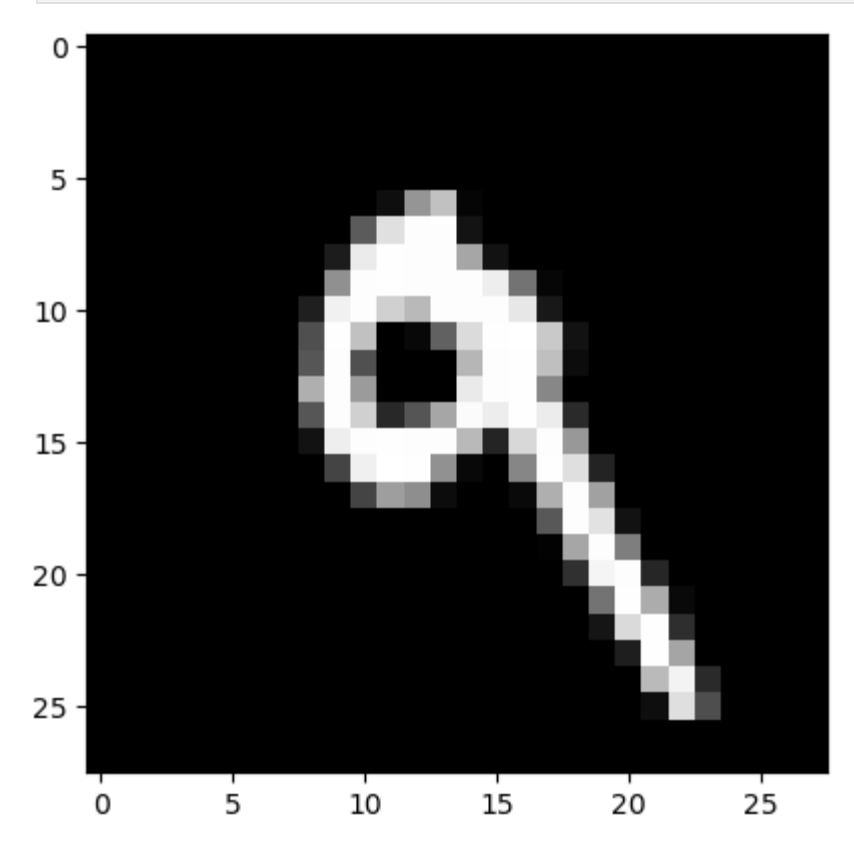
In [19]:
```python
plt.plot(history.history['accuracy'],label='Train acc')
plt.plot(history.history['val_accuracy'],label='val_acc')
plt.show()
```



In [20]:
```python
from keras.models import load_model
model_h1= load_model('mnist_model.h5')
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
```

In [21]:
```python
plt.imshow(x_test[7],cmap='gray')
plt.show()
```



In [22]:
```python
#for predicting single value
```

In [23]:
```python
x=np.reshape(x_test[[7]],(1,28,28))
np.argmax(model.predict(x))
```

```
1/1 ──────────── 0s 58ms/step
```

Out[23]: 9

In [24]:
```python
# to predict multipla data points
```

Example in Context:

Let's assume indices = [7, 8, 9] and x_test is a list of images. The loop would work as follows:

First Iteration:

i = 0, idx = 7

image = x_test[7] (retrieves the 8th image from x_test and stores it in image)

Second Iteration:

i = 1, idx = 8

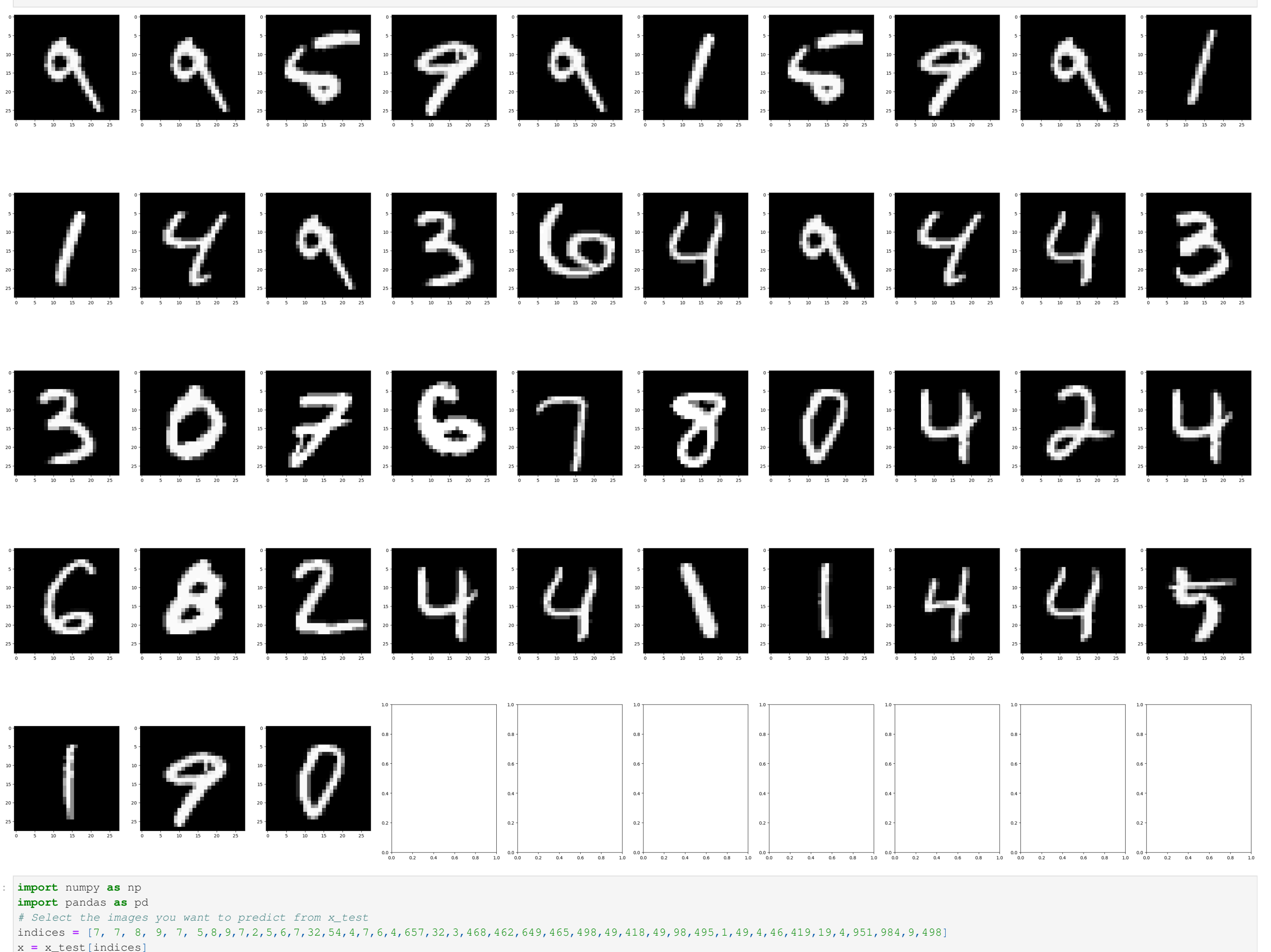
image = x_test[8] (retrieves the 9th image from x_test and stores it in image)

Third Iteration:

i = 2, idx = 9

image = x_test[9] (retrieves the 10th image from x_test and stores it in image)

In [26]:
```python
import matplotlib.pyplot as plt

# Select the indices of the images you want to plot
indices = [7, 7, 8, 9, 7, 5, 8, 9, 7, 2, 5, 6, 7, 32, 54, 4, 7, 6, 4, 657, 32, 3, 468, 462, 649, 465, 498, 49, 418, 49, 98, 495, 1, 49, 4, 46, 419, 19, 4, 951, 984, 9, 498]
# Number of images
num_images = len(indices)

# Create subplots
fig, axes = plt.subplots(nrows=5,ncols=10, figsize=(50,35))

# Flatten the axes array for easy iteration
# When axes is flattened, you can use a single loop to iterate over all subplots, without needing to manage row and column indices separately.
axes = axes.flatten()

# Loop through the images and plot them
for i, idx in enumerate(indices):
    # Get the image from x_test
    image = x_test[idx]
    # Plot the image
    axes[i].imshow(image, cmap='gray')
plt.show()
```



In [64]:
```python
import numpy as np
import pandas as pd
# Select the images you want to predict from x_test
indices = [7, 7, 8, 9, 7, 5,8,9,7,2,5,6,7,32,54,4,7,6,4,657,32,3,468,462,649,465,498,49,418,49,98,495,1,49,4,46,419,19,4,951,984,9,498]
x = x_test[indices]

# Ensure the input data type is float32
x = x.astype('float32')

# Reshape to the proper dimensions: (number_of_images, height, width, channels)
x = np.reshape(x, (len(indices), 28, 28, 1))

# Make predictions
predictions = model.predict(x)

# Print the predicted classes
predicted_classes = np.argmax(predictions, axis=1)
print(predicted_classes)

df=pd.DataFrame(predictions)
df.head()
```

```
2/2 ──────────── 0s 24ms/step
[9 9 4 9 9 1 4 9 9 1 1 4 9 3 2 4 9 4 4 3 3 0 7 6 7 1 0 4 2 4 6 2 2 4 4 3 1
 4 4 4 1 9 0]
```

Out[64]:

	0	1	2	3	4	5	6	7	8	9
0	0.000003	0.000857	0.001698	0.002834	0.232782	0.003157	0.003792	0.074334	0.012968	0.667575
1	0.000003	0.000857	0.001698	0.002834	0.232782	0.003157	0.003792	0.074334	0.012968	0.667575
2	0.011730	0.000343	0.008267	0.002604	0.629283	0.079222	0.109258	0.002508	0.044182	0.112603

```
3  0.000003  0.000857  0.001698  0.002834  0.232782  0.003157  0.003782  0.074334  0.012968  0.667575
4  0.000003  0.000857  0.001698  0.002834  0.232782  0.003157  0.003782  0.074334  0.012968  0.667575
```

Why the `1` ? `x = np.reshape(x, (len(indices), 28, 28, 1))`

Grayscale Images: If your images are grayscale, they have only one channel, which is why we use 1. Each pixel is represented by a single intensity value, unlike colored images where each pixel is represented by multiple values (e.g., Red, Green, Blue).

Example Context: If you had RGB images (colored), the shape would instead be `(len(valid_indices), 28, 28, 3)`, where 3 represents the three color channels (Red, Green, Blue).

In [ ]: