In this lab, you will implementat a neural network using the Keras library for the Fashion MNIST dataset.

These are the steps:

1. Import necessary libraries
2. Load the dataset
3. Print the shape
4. Data Preprocessing:

- Reshape the input data from 28x28 images to a flat vector of size 784.
- Normalize the pixel values to the range [0, 1].
- Convert class labels to one-hot encoded vectors.

5. Build the Neural Network Model

- Create a sequential model using Keras.
- Add a dense layer with 64 neurons and a sigmoid activation function.
- Add an output layer with 10 neurons (for 10 classes) and a softmax activation function.

6. Compile the model using mean squared error as the loss function and stochastic gradient descent (SGD) as the optimizer.
7. Train the model

- Use 100 epochs and a batch size of 128.

8. Display a summary of the model architecture.
9. Evaluate the model on the validation set and print the accuracy.
10. Prot the Confusion Matrix
11. Visualize the Predictions
12. Make predictions on the validation set and display the predicted class probabilities for a specific example.

Import Libraries: numpy, matplotlib

- from keras.datasets import fashion_mnist
- from keras.models import Sequential
- from keras.layers import Dense
- from tensorflow.keras.optimizers import SGD
- from keras.utils import to_categorical

In [4]:
```
#from keras.datasets import fashion_mnist
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import fashion_mnist
from keras.models import Sequential
```

```
from keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from keras.utils import to_categorical
```

Load the data into these variables: (X_train, y_train), (X_valid, y_valid)

In [6]:
```
(X_train, y_train), (X_valid, y_valid) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Print a random image from the dataset

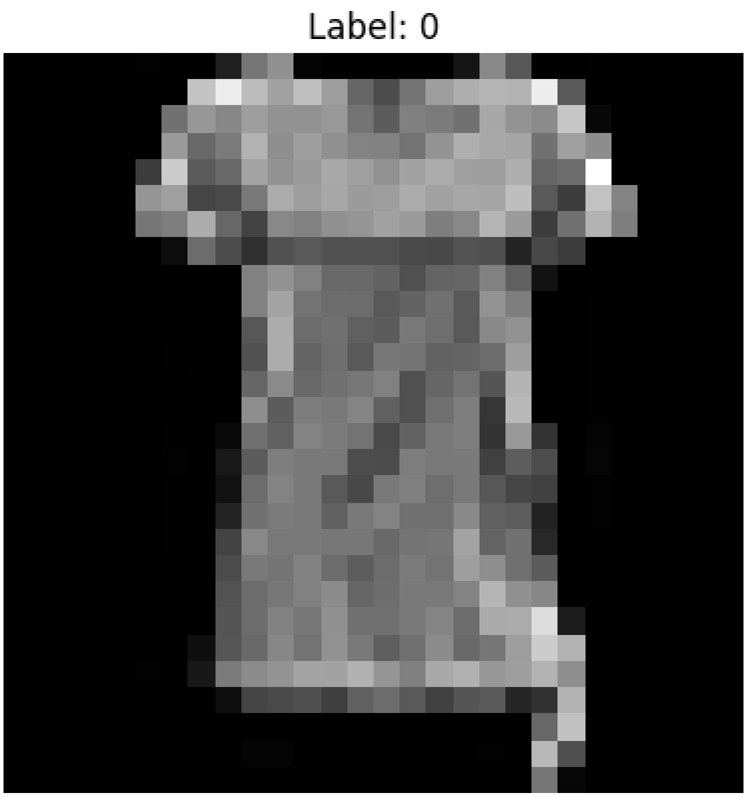
feel free to use: np.random.randint (0, X_train.shape[0])

In [24]:
```
random_index = np.random.randint(0, X_train.shape[0])


random_image = X_train[random_index].reshape(28, 28)
random_label = np.argmax(y_train[random_index])
```

Plot the image

See sample below for sample 39235. Most likely, your image will be different.

In [27]:
```
plt.imshow(random_image, cmap='gray')
plt.title(f"Label: {random_label}")
plt.axis('off')
plt.show()
```



In [ ]:

Confirm image label

you will get a number from (0 to 9).

In [31]:
```
print("Confirmed Label:", random_label)
```

```
Confirmed Label: 0
```

Plot the image in a matrix format

Use precision, suppress, and linewidth

In [35]:
```
print("Image Matrix (28x28):")
print(random_image)
```

```
Image Matrix (28x28):
[[  0   0   0   0   0   2   0   0  33 118 144   4   0   0   0   0   0  21
  136  87   0   0   1   0   0   0   0   0]
 [  0   0   0   0   0   0   0 195 236 188 161 191 157 102  76 115 157 174
  181 180 237  90   0   0   0   0   0   0]
 [  0   0   0   0   0   0 113 151 135 157 146 145 152 116  92 128 123 113
  167 148 139 198   8   0   0   0   0   0]
 [  0   0   0   0   0   0 154 105 123 178 142 158 144 132 129 115 148 175
  168 165 113 159 141   0   0   0   0   0]
 [  0   0   0   0   0  61 203  92 105 164 146 155 168 159 145 162 171 159
  157 174 102 108 255   0   0   0   0   0]
 [  0   0   0   0   0 149 159  70  73 122 171 158 167 155 158 172 161 167
  165 190  90  61 194 132   0   0   0   0]
 [  0   0   0   0   0 118 126 172 103  67 136 129 144 149 161 155 126 136
  181 155  61 113 178 126   0   0   0   0]
 [  0   0   0   0   0   0  12 108  76  47  82  90  82  82  80  73  72  83
   79  37  72  60   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 129 146 129 106 106  99  80 100 102
  129  97  18   0   0   1   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 129 164 115 109 108  90  99 112  90
  148 126   0   0   1   0   0   0   0   0]
 [  0   0   0   0   0   0   0   1   0  87 172 108 113  96  92 121 110  90
  136 145   0   0   1   0   0   0   0   0]
 [  0   0   0   0   0   0   1   1   0  82 172 100 110  90 119 116  99 100
  109 158   0   0   1   1   0   0   0   0]
 [  0   0   0   0   0   0   0   1   0 102 138 106 113 119 129  80 102 115
   85 180   0   0   1   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 142  92 125 121 133  97  80 112 126
   54 184   0   0   1   0   0   0   0   0]
 [  0   0   0   0   0   0   1   0  10 112  97 131 123 115  73  99 121 126
   51 152  51   0   4   0   0   0   0   0]
 [  0   0   0   0   0   0   2   0  25  92 126 121 122  77  76 126 121 122
   66  93  77   0   5   0   0   0   0   0]
 [  0   0   0   0   0   0   1   0  18 108 131 122  89  72 122 128 110 122
   87  70  66   0   2   0   0   0   0   0]
 [  0   0   0   0   0   0   1   0  36 113 123 121  97 122 132 113 112 136
   96  93  34   0   2   0   0   0   0   0]
 [  0   0   0   0   0   0   1   0  67 136 121 121 119 119 106 116 118 164
  100 113  40   0   0   0   0   0   0   0]
 [  0   0   0   0   0   1   0   0  76 121 116 129 108  93 108 123 115 158
  142 112  92   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  83 108 119 129 139 102 109 122 122 131
  181 144 135   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  85 108 129 119 144 112 112 122 133 109
  170 171 220  28   0   1   0   0   0   0]
 [  0   0   0   0   0   1   0  15  86 105 135 115 145 122  95 112 139 106
  118 157 204 180   0   0   0   0   0   0]
 [  0   0   0   0   0   2   0  24 123 139 148 164 162 177 149 128 168 177
  151 158 178 142   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  14  69  73  79  63  97 109  89  66  85
   90  38  50 178   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0 103 193   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   1   4   4   0   0   0   0   0   0   1
    2   0 184  79   0   0   0   0   0   0]
```

```
[   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0 118   7   0   0   0   0   0   0]]
```

In [7]:

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   5   4   1   1   2   3   0   0
0   0  30  63 112 111   0   0]
 [  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0  26 20
9 242 255 255 211 255  30   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  85  66  44  31  49  50   0 109 25
5 223 223 210 197 214 109   0]
 [  0   0   0   0   0   0   0   0   0   5   0  13 191 208 207 204 245 252  30 112 22
9 216 215 209 218 225 194   0]
 [  0   0   0   0   0   0   0   0   0   2   0  63 190 188 193 208 206 227 167 176 23
0 211 202 210 215 211 224  33]
 [  0   0   0   0   0   0   0   0   0   4   0 127 207 185 191 212 211 194 196 199 22
7 225 203 203 215 210 222 133]
 [  0   0   0   0   0   0   0   0   1   0   0 197 193 185 195 196 200 198 194 215 22
2 204 191 189 217 212 211 172]
 [  0   0   0   0   0   1   0   0   1   0  45 225 177 188 193 190 199 214 142 191 23
7 204 208 220 228 213 205 177]
 [  0   0   0   1   1   2   1   2   1   0 126 206 174 191 192 195 198 220 150 177 24
6 227 245 238 232 203 216 165]
 [  1   0   0   0   2   0   0   2   0   0 181 186 177 196 200 199 203 213 120 173 25
5 229 212 198 199 196 208 108]
 [  0   0   0   0   0   0   0   2   0  54 213 172 190 196 200 203 201 221 202 136 23
3 184 190 193 189 172 188  23]
 [  1   0   0   0   0   0   0   0   0 166 190 179 191 197 202 209 205 222 232 157 19
1 241 200 187 169 178 178   0]
 [  1   0   1   3   4   5   0   0  96 210 174 189 191 197 202 210 213 217 212 218 25
1 100   0 146 192 186 132   0]
 [  1   5   0   0   0   0   0  68 198 181 184 191 199 197 208 207 216 203 239 219   1
9   0   0 149 189 185  77   0]
 [  0   0   0   0   4  46 127 189 190 184 190 194 200 201 200 213 198 235 211   0
0   0   0 163 194 183  19   0]
 [  0  55 157 167 170 180 197 181 185 189 191 196 202 209 203 203 217 251   0   0
4   0   0 166 203 170   0   0]
 [ 63 191 184 177 166 174 177 191 184 189 192 195 209 206 209 201 252  79   0   0
3   4   0 159 214 150   0   0]
 [ 79 213 173 180 184 188 192 189 193 197 201 210 214 212 203 228 149   0   0   0
0   0   0 143 214 125   0   0]
 [  0 187 219 215 210 203 198 193 199 194 196 201 195 200 226 233   0   0   0   0
1   0   0 154 225 113   0   0]
 [  0   0  28 119 200 236 248 249 245 243 243 244 243 252 206   2   0   5   0   0
1   0   0 171 185 119   0   0]
 [  0   0   0   0   0   0  15  53  66  88 106 107  86  49   0   0   1   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   0   0   0   0   0   0
0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0]
 [  0   0   0   0   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 [ ]:

```

```

Rename the labels (class_names)

from (0,1,2,3...,9) to ('T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot')

In [43]:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Sh

random_index = np.random.randint(0, X_train.shape[0])


random_image = X_train[random_index].reshape(28, 28)
random_label_index = np.argmax(y_train[random_index])
random_label_name = class_names[random_label_index]
```

Print the shape of the train and test images and labes

i.e. X_train.shape

In [46]:

```
(X_train, y_train), (X_valid, y_valid) = fashion_mnist.load_data()


print("train_images.shape:", X_train.shape)
print("len(train_labels):", len(y_train))
print("test_images.shape:", X_valid.shape)
print("len(test_labels):", len(y_valid))
```

```
train_images.shape: (60000, 28, 28)
len(train_labels): 60000
test_images.shape: (10000, 28, 28)
len(test_labels): 10000
```

In [9]:

```
# It should look like this:
```

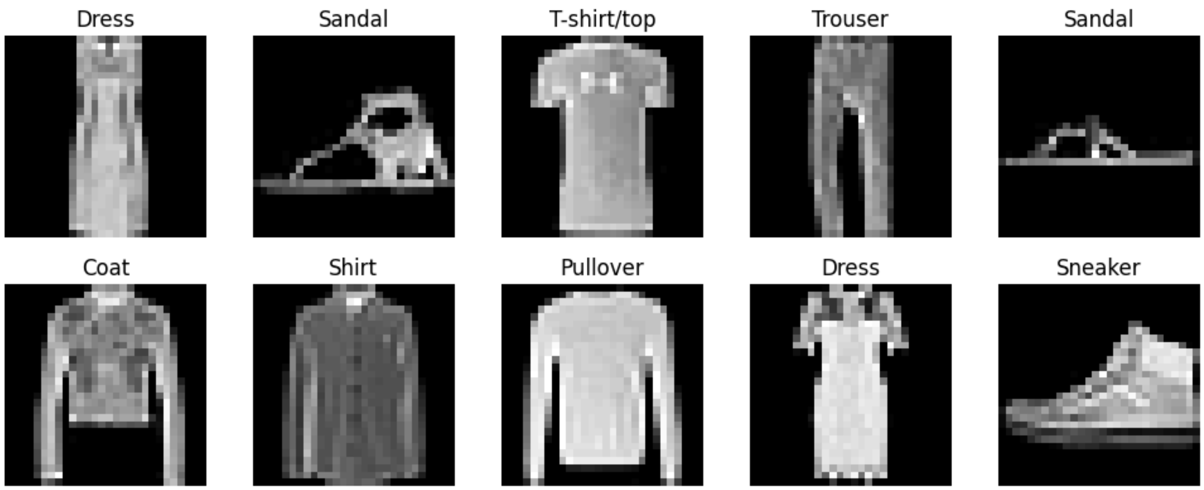
```
train_images.shape: (60000, 28, 28)
len(train_labels: 60000
test_images.shape: (10000, 28, 28)
len(test_labels): 10000
```

Plot several images showing the new label

In [50]:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Sh


plt.figure(figsize=(10, 10))
for i in range(10):
    random_index = np.random.randint(0, X_train.shape[0])
    random_image = X_train[random_index]
    random_label_index = y_train[random_index]
    random_label_name = class_names[random_label_index]

    plt.subplot(5, 5, i + 1)
```

```
    plt.imshow(random_image, cmap='gray')
    plt.title(random_label_name)
    plt.axis('off')

plt.tight_layout()
plt.show()
```



In [10]:
```
# sample
```

Shallow Neural Network in Keras

Plot some images without the label

In [56]:
```
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from keras.utils import to_categorical
from keras.datasets import fashion_mnist
import numpy as np
import matplotlib.pyplot as plt

(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
```

```
X_train = X_train.reshape(-1, 784).astype('float32') / 255
X_test = X_test.reshape(-1, 784).astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)


model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(784,)),
    Dense(10, activation='softmax')
])


model.compile(optimizer=SGD(), loss='mean_squared_error', metrics=['accuracy'])


model.fit(X_train, y_train, epochs=10, batch_size=128, validation_data=(X_test, y_t




plt.figure(figsize=(10, 10))
for i in range(10):
    random_index = np.random.randint(0, X_test.shape[0])
    random_image = X_test[random_index].reshape(28, 28)

    plt.subplot(2, 5, i + 1)
    plt.imshow(random_image, cmap='gray')
    plt.axis('off')

plt.tight_layout()
plt.show()
```

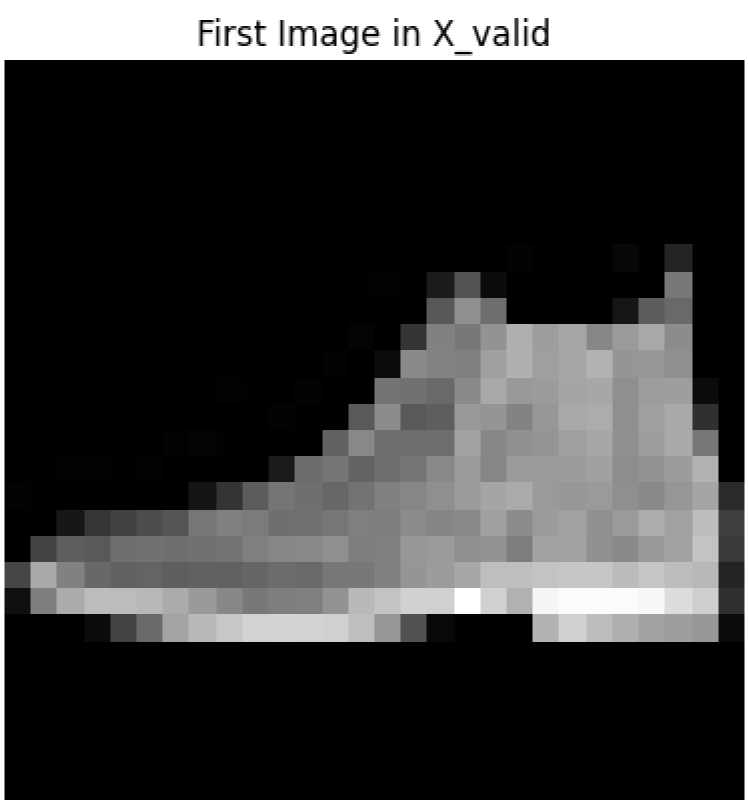
```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.0995 - loss: 0.0969 - val_acc
uracy: 0.1040 - val_loss: 0.0936
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 918us/step - accuracy: 0.1116 - loss: 0.0931 - val_a
ccuracy: 0.1420 - val_loss: 0.0918
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 968us/step - accuracy: 0.1492 - loss: 0.0915 - val_a
ccuracy: 0.1695 - val_loss: 0.0906
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 914us/step - accuracy: 0.1829 - loss: 0.0905 - val_a
ccuracy: 0.2384 - val_loss: 0.0897
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 949us/step - accuracy: 0.2570 - loss: 0.0895 - val_a
ccuracy: 0.3021 - val_loss: 0.0889
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 931us/step - accuracy: 0.3076 - loss: 0.0887 - val_a
ccuracy: 0.3280 - val_loss: 0.0881
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 914us/step - accuracy: 0.3333 - loss: 0.0879 - val_a
ccuracy: 0.3442 - val_loss: 0.0874
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 905us/step - accuracy: 0.3476 - loss: 0.0871 - val_a
ccuracy: 0.3529 - val_loss: 0.0867
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 940us/step - accuracy: 0.3569 - loss: 0.0864 - val_a
ccuracy: 0.3591 - val_loss: 0.0859
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 918us/step - accuracy: 0.3566 - loss: 0.0858 - val_a
ccuracy: 0.3631 - val_loss: 0.0853
```

In [12]:
```
#sample
```

Plot the first image

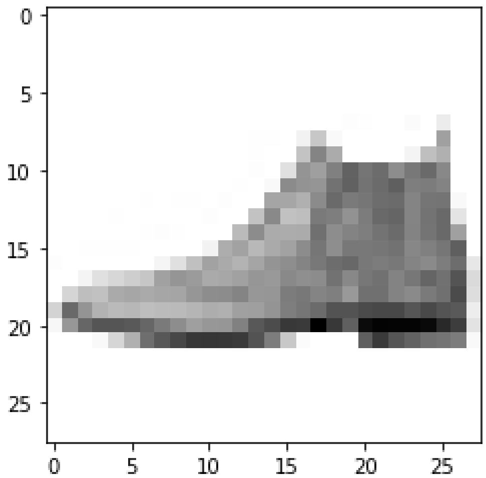
X_valid[0]

In [59]:
```
(_, _), (X_valid, y_valid) = fashion_mnist.load_data()

plt.imshow(X_valid[0], cmap='gray')
plt.title("First Image in X_valid")
plt.axis('off')
plt.show()
```



In [13]:
```
# sample
```

Out[13]: <matplotlib.image.AxesImage at 0x17dd1244bb0>

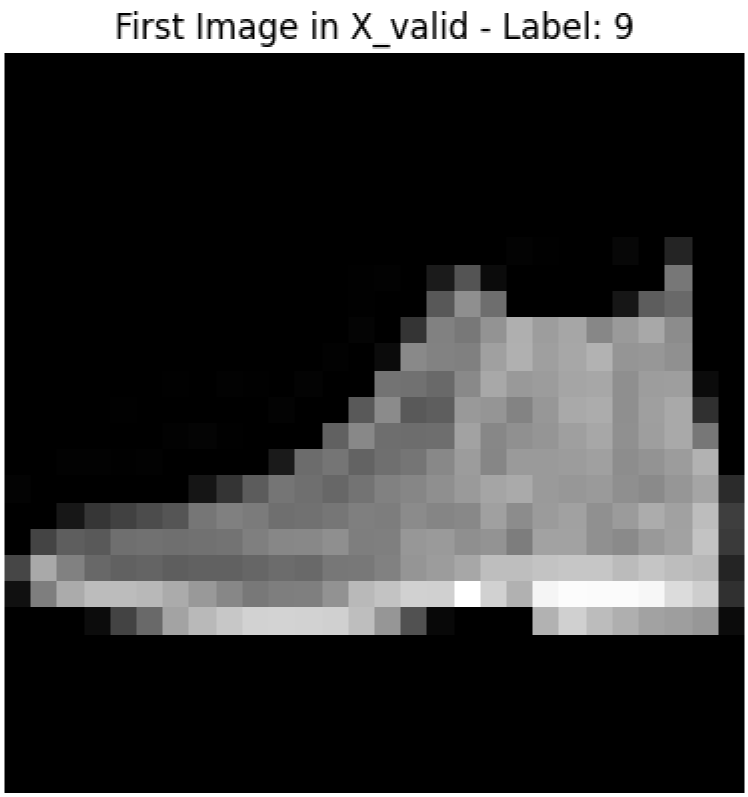


Validate that the label is a 9

y_valid[0]

In [61]:
```
(_, _), (X_valid, y_valid) = fashion_mnist.load_data()


print("Label for y_valid[0]:", y_valid[0])


plt.imshow(X_valid[0], cmap='gray')
plt.title(f"First Image in X_valid - Label: {y_valid[0]}")
plt.axis('off')
plt.show()
```

```
Label for y_valid[0]: 9
```



Preprocess data

- After you load in the images, reshape them from a two-dimensional 28x28 shape to a one-dimensional array of 784 elements (28 x 28 = 784)
- Use the as type ('float32') to convert the pixel darknesses from integers into single-precision float values.

In [15]:

```

```

Converting pixel intergers to floats

devide by 255.0

In [65]:

```
from keras.datasets import fashion_mnist
import numpy as np

# Load the dataset
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

# Preprocess the data
# Reshape the images to (num_samples, 784) and convert to float32
X_train = X_train.reshape(X_train.shape[0], 28 * 28).astype('float32') / 255.0
X_test = X_test.reshape(X_test.shape[0], 28 * 28).astype('float32') / 255.0

# Print the shapes to verify
```

```
print("X_train shape:", X_train.shape)  # Should be (60000, 784)
print("X_test shape:", X_test.shape)    # Should be (10000, 784)
```

```
X_train shape: (60000, 784)
X_test shape: (10000, 784)
```

convert the label y (y_train,y_valid) from integers into one-hot encodings

n_classes = 10

In [68]:
```
from keras.datasets import fashion_mnist
import numpy as np


(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()


X_train = X_train.reshape(X_train.shape[0], 28 * 28).astype('float32') / 255.0
X_test = X_test.reshape(X_test.shape[0], 28 * 28).astype('float32') / 255.0


print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
```

```
X_train shape: (60000, 784)
X_test shape: (10000, 784)
```

display the values of y_valid

In [80]:
```
(_, _), (X_valid, y_valid) = fashion_mnist.load_data()


y_valid_one_hot = to_categorical(y_valid, num_classes=10)


print("One-hot encoded values of y_valid:")
print(y_valid_one_hot)


print("Unique labels in one-hot encoded y_valid:", np.unique(y_valid_one_hot, axis=
```

```
One-hot encoded values of y_valid:
[[0. 0. 0. ... 0. 0. 1.]
 [0. 0. 1. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 1. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
Unique labels in one-hot encoded y_valid: [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

In [18]:

```
# sample
```

Out[18]:

```
array([[0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Design neural network architecture

- Create a Sequential model and call it "model"
- For the hidden layer, use the add() method with 64 neurons and activation = 'sigmoid' with an input_shape=(784,0)
- For the output layer, use the add() method with 10 neurons and the softmax activation function

In [ ]:

Display the summary of the model and explain what the numbers mean

In [83]:

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()

model.add(Dense(64, activation='sigmoid', input_shape=(784,)))


model.add(Dense(10, activation='softmax'))
```

```
model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 64)	50,240
dense_5 (Dense)	(None, 10)	650

Total params: 50,890 (198.79 KB)

Trainable params: 50,890 (198.79 KB)

Non-trainable params: 0 (0.00 B)

In [20]:
```
# sample
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                50240

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

Compile the model using:

- loss="categorical_crossentropy"
- Set the cost-minimizing method to stochastic gradient descent by using optimizer = SGD
- Specify the SGD learning rate hyperparameter equals to 0.01
- Set the metrics to 'accurancy' to recieve feedbak on model accurancy

In [91]:
```
from tensorflow.keras.optimizers import SGD


model.compile(
    loss="categorical_crossentropy",
    optimizer=SGD(learning_rate=0.01),
    metrics=['accuracy']
)


print("Model compiled successfully.")
```

```
Model compiled successfully.
```

Training the data

Fit the model using a batch_size = 128 and 100 epochs

- model.fit()
- set batch_size = 128
- set verbove = 1
- set epochs = 100
- validate the data

In [98]:
```
from keras.utils import to_categorical
from keras.datasets import fashion_mnist

(X_train, y_train), (X_valid, y_valid) = fashion_mnist.load_data()

X_train = X_train.reshape(X_train.shape[0], 28 * 28).astype('float32') / 255.0
X_valid = X_valid.reshape(X_valid.shape[0], 28 * 28).astype('float32') / 255.0

y_train_one_hot = to_categorical(y_train, num_classes=10)
y_valid_one_hot = to_categorical(y_valid, num_classes=10)

history = model.fit(
    X_train,
    y_train_one_hot,
    batch_size=128,
    epochs=100,
    verbose=1,
    validation_data=(X_valid, y_valid_one_hot)
)

print("Model training completed.")
```

```
Epoch 1/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 976us/step - accuracy: 0.8560 - loss: 0.4020 - val_a
ccuracy: 0.8417 - val_loss: 0.4391
Epoch 2/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 902us/step - accuracy: 0.8606 - loss: 0.3957 - val_a
ccuracy: 0.8424 - val_loss: 0.4385
Epoch 3/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 906us/step - accuracy: 0.8560 - loss: 0.4045 - val_a
ccuracy: 0.8428 - val_loss: 0.4379
Epoch 4/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 903us/step - accuracy: 0.8556 - loss: 0.4046 - val_a
ccuracy: 0.8432 - val_loss: 0.4370
Epoch 5/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 857us/step - accuracy: 0.8581 - loss: 0.3997 - val_a
ccuracy: 0.8432 - val_loss: 0.4364
Epoch 6/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 851us/step - accuracy: 0.8592 - loss: 0.3997 - val_a
ccuracy: 0.8435 - val_loss: 0.4358
Epoch 7/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 908us/step - accuracy: 0.8595 - loss: 0.3966 - val_a
ccuracy: 0.8433 - val_loss: 0.4358
Epoch 8/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 858us/step - accuracy: 0.8586 - loss: 0.3996 - val_a
ccuracy: 0.8430 - val_loss: 0.4350
Epoch 9/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 851us/step - accuracy: 0.8603 - loss: 0.3966 - val_a
ccuracy: 0.8444 - val_loss: 0.4340
Epoch 10/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 858us/step - accuracy: 0.8588 - loss: 0.3963 - val_a
ccuracy: 0.8438 - val_loss: 0.4339
Epoch 11/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 909us/step - accuracy: 0.8574 - loss: 0.3971 - val_a
ccuracy: 0.8437 - val_loss: 0.4333
Epoch 12/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 860us/step - accuracy: 0.8595 - loss: 0.3952 - val_a
ccuracy: 0.8445 - val_loss: 0.4331
Epoch 13/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 940us/step - accuracy: 0.8596 - loss: 0.3965 - val_a
ccuracy: 0.8442 - val_loss: 0.4322
Epoch 14/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8595 - loss: 0.3956 - val_acc
uracy: 0.8447 - val_loss: 0.4314
Epoch 15/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8615 - loss: 0.3920 - val_acc
uracy: 0.8442 - val_loss: 0.4310
Epoch 16/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 983us/step - accuracy: 0.8625 - loss: 0.3875 - val_a
ccuracy: 0.8442 - val_loss: 0.4309
Epoch 17/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8605 - loss: 0.3928 - val_acc
uracy: 0.8463 - val_loss: 0.4299
Epoch 18/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8632 - loss: 0.3900 - val_acc
uracy: 0.8462 - val_loss: 0.4295
Epoch 19/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8629 - loss: 0.3892 - val_acc
```

```
uracy: 0.8455 - val_loss: 0.4292
Epoch 20/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8640 - loss: 0.3883 - val_acc
uracy: 0.8462 - val_loss: 0.4284
Epoch 21/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8623 - loss: 0.3883 - val_acc
uracy: 0.8453 - val_loss: 0.4279
Epoch 22/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8610 - loss: 0.3922 - val_acc
uracy: 0.8471 - val_loss: 0.4275
Epoch 23/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8642 - loss: 0.3878 - val_acc
uracy: 0.8464 - val_loss: 0.4273
Epoch 24/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 883us/step - accuracy: 0.8640 - loss: 0.3847 - val_a
ccuracy: 0.8473 - val_loss: 0.4268
Epoch 25/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 923us/step - accuracy: 0.8645 - loss: 0.3800 - val_a
ccuracy: 0.8458 - val_loss: 0.4264
Epoch 26/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 897us/step - accuracy: 0.8630 - loss: 0.3851 - val_a
ccuracy: 0.8481 - val_loss: 0.4256
Epoch 27/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 975us/step - accuracy: 0.8624 - loss: 0.3887 - val_a
ccuracy: 0.8478 - val_loss: 0.4254
Epoch 28/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 891us/step - accuracy: 0.8657 - loss: 0.3802 - val_a
ccuracy: 0.8476 - val_loss: 0.4246
Epoch 29/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 886us/step - accuracy: 0.8668 - loss: 0.3774 - val_a
ccuracy: 0.8478 - val_loss: 0.4242
Epoch 30/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 862us/step - accuracy: 0.8634 - loss: 0.3845 - val_a
ccuracy: 0.8477 - val_loss: 0.4241
Epoch 31/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 860us/step - accuracy: 0.8614 - loss: 0.3876 - val_a
ccuracy: 0.8482 - val_loss: 0.4232
Epoch 32/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 894us/step - accuracy: 0.8612 - loss: 0.3862 - val_a
ccuracy: 0.8485 - val_loss: 0.4230
Epoch 33/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 834us/step - accuracy: 0.8630 - loss: 0.3854 - val_a
ccuracy: 0.8486 - val_loss: 0.4224
Epoch 34/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 859us/step - accuracy: 0.8654 - loss: 0.3780 - val_a
ccuracy: 0.8487 - val_loss: 0.4222
Epoch 35/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 881us/step - accuracy: 0.8628 - loss: 0.3872 - val_a
ccuracy: 0.8489 - val_loss: 0.4216
Epoch 36/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 853us/step - accuracy: 0.8633 - loss: 0.3839 - val_a
ccuracy: 0.8489 - val_loss: 0.4212
Epoch 37/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 852us/step - accuracy: 0.8640 - loss: 0.3809 - val_a
ccuracy: 0.8493 - val_loss: 0.4208
Epoch 38/100
```

```
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 932us/step - accuracy: 0.8627 - loss: 0.3801 - val_a
ccuracy: 0.8483 - val_loss: 0.4203
Epoch 39/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 892us/step - accuracy: 0.8634 - loss: 0.3833 - val_a
ccuracy: 0.8495 - val_loss: 0.4202
Epoch 40/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 889us/step - accuracy: 0.8676 - loss: 0.3749 - val_a
ccuracy: 0.8494 - val_loss: 0.4201
Epoch 41/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 952us/step - accuracy: 0.8671 - loss: 0.3756 - val_a
ccuracy: 0.8497 - val_loss: 0.4192
Epoch 42/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 901us/step - accuracy: 0.8673 - loss: 0.3772 - val_a
ccuracy: 0.8500 - val_loss: 0.4187
Epoch 43/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 929us/step - accuracy: 0.8668 - loss: 0.3771 - val_a
ccuracy: 0.8496 - val_loss: 0.4182
Epoch 44/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 870us/step - accuracy: 0.8656 - loss: 0.3756 - val_a
ccuracy: 0.8499 - val_loss: 0.4181
Epoch 45/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 946us/step - accuracy: 0.8674 - loss: 0.3760 - val_a
ccuracy: 0.8510 - val_loss: 0.4176
Epoch 46/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 925us/step - accuracy: 0.8651 - loss: 0.3770 - val_a
ccuracy: 0.8499 - val_loss: 0.4174
Epoch 47/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 865us/step - accuracy: 0.8675 - loss: 0.3752 - val_a
ccuracy: 0.8515 - val_loss: 0.4168
Epoch 48/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 944us/step - accuracy: 0.8680 - loss: 0.3718 - val_a
ccuracy: 0.8505 - val_loss: 0.4169
Epoch 49/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 865us/step - accuracy: 0.8678 - loss: 0.3737 - val_a
ccuracy: 0.8513 - val_loss: 0.4162
Epoch 50/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 917us/step - accuracy: 0.8670 - loss: 0.3776 - val_a
ccuracy: 0.8517 - val_loss: 0.4155
Epoch 51/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 919us/step - accuracy: 0.8689 - loss: 0.3737 - val_a
ccuracy: 0.8513 - val_loss: 0.4154
Epoch 52/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 907us/step - accuracy: 0.8673 - loss: 0.3735 - val_a
ccuracy: 0.8521 - val_loss: 0.4150
Epoch 53/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 860us/step - accuracy: 0.8697 - loss: 0.3715 - val_a
ccuracy: 0.8518 - val_loss: 0.4147
Epoch 54/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 906us/step - accuracy: 0.8673 - loss: 0.3710 - val_a
ccuracy: 0.8524 - val_loss: 0.4142
Epoch 55/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 912us/step - accuracy: 0.8663 - loss: 0.3749 - val_a
ccuracy: 0.8521 - val_loss: 0.4138
Epoch 56/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 857us/step - accuracy: 0.8688 - loss: 0.3710 - val_a
ccuracy: 0.8519 - val_loss: 0.4134
```

```
Epoch 57/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 946us/step - accuracy: 0.8676 - loss: 0.3760 - val_a
ccuracy: 0.8528 - val_loss: 0.4130
Epoch 58/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 866us/step - accuracy: 0.8683 - loss: 0.3720 - val_a
ccuracy: 0.8520 - val_loss: 0.4128
Epoch 59/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 922us/step - accuracy: 0.8697 - loss: 0.3690 - val_a
ccuracy: 0.8524 - val_loss: 0.4124
Epoch 60/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 900us/step - accuracy: 0.8672 - loss: 0.3738 - val_a
ccuracy: 0.8526 - val_loss: 0.4122
Epoch 61/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 943us/step - accuracy: 0.8670 - loss: 0.3749 - val_a
ccuracy: 0.8531 - val_loss: 0.4121
Epoch 62/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 883us/step - accuracy: 0.8685 - loss: 0.3693 - val_a
ccuracy: 0.8526 - val_loss: 0.4114
Epoch 63/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 907us/step - accuracy: 0.8709 - loss: 0.3661 - val_a
ccuracy: 0.8534 - val_loss: 0.4114
Epoch 64/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 909us/step - accuracy: 0.8710 - loss: 0.3656 - val_a
ccuracy: 0.8529 - val_loss: 0.4110
Epoch 65/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 927us/step - accuracy: 0.8694 - loss: 0.3716 - val_a
ccuracy: 0.8535 - val_loss: 0.4102
Epoch 66/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 895us/step - accuracy: 0.8725 - loss: 0.3625 - val_a
ccuracy: 0.8532 - val_loss: 0.4100
Epoch 67/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 879us/step - accuracy: 0.8695 - loss: 0.3687 - val_a
ccuracy: 0.8538 - val_loss: 0.4100
Epoch 68/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 908us/step - accuracy: 0.8697 - loss: 0.3667 - val_a
ccuracy: 0.8538 - val_loss: 0.4096
Epoch 69/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 858us/step - accuracy: 0.8713 - loss: 0.3656 - val_a
ccuracy: 0.8536 - val_loss: 0.4097
Epoch 70/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 893us/step - accuracy: 0.8699 - loss: 0.3661 - val_a
ccuracy: 0.8543 - val_loss: 0.4087
Epoch 71/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 920us/step - accuracy: 0.8673 - loss: 0.3715 - val_a
ccuracy: 0.8542 - val_loss: 0.4087
Epoch 72/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 888us/step - accuracy: 0.8706 - loss: 0.3677 - val_a
ccuracy: 0.8546 - val_loss: 0.4086
Epoch 73/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 919us/step - accuracy: 0.8697 - loss: 0.3651 - val_a
ccuracy: 0.8542 - val_loss: 0.4078
Epoch 74/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 894us/step - accuracy: 0.8708 - loss: 0.3658 - val_a
ccuracy: 0.8544 - val_loss: 0.4076
Epoch 75/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 869us/step - accuracy: 0.8711 - loss: 0.3650 - val_a
```

```
ccuracy: 0.8548 - val_loss: 0.4073
Epoch 76/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 931us/step - accuracy: 0.8715 - loss: 0.3579 - val_a
ccuracy: 0.8546 - val_loss: 0.4079
Epoch 77/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 916us/step - accuracy: 0.8711 - loss: 0.3641 - val_a
ccuracy: 0.8547 - val_loss: 0.4065
Epoch 78/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 913us/step - accuracy: 0.8699 - loss: 0.3644 - val_a
ccuracy: 0.8549 - val_loss: 0.4062
Epoch 79/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 957us/step - accuracy: 0.8711 - loss: 0.3646 - val_a
ccuracy: 0.8540 - val_loss: 0.4061
Epoch 80/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8708 - loss: 0.3638 - val_acc
uracy: 0.8554 - val_loss: 0.4056
Epoch 81/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8712 - loss: 0.3629 - val_acc
uracy: 0.8553 - val_loss: 0.4054
Epoch 82/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8717 - loss: 0.3615 - val_acc
uracy: 0.8551 - val_loss: 0.4052
Epoch 83/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8725 - loss: 0.3584 - val_acc
uracy: 0.8554 - val_loss: 0.4047
Epoch 84/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8736 - loss: 0.3592 - val_acc
uracy: 0.8549 - val_loss: 0.4047
Epoch 85/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8705 - loss: 0.3635 - val_acc
uracy: 0.8550 - val_loss: 0.4042
Epoch 86/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8713 - loss: 0.3620 - val_acc
uracy: 0.8550 - val_loss: 0.4045
Epoch 87/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8701 - loss: 0.3597 - val_acc
uracy: 0.8550 - val_loss: 0.4038
Epoch 88/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8716 - loss: 0.3629 - val_acc
uracy: 0.8562 - val_loss: 0.4034
Epoch 89/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8740 - loss: 0.3584 - val_acc
uracy: 0.8557 - val_loss: 0.4038
Epoch 90/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8696 - loss: 0.3648 - val_acc
uracy: 0.8568 - val_loss: 0.4031
Epoch 91/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 950us/step - accuracy: 0.8714 - loss: 0.3617 - val_a
ccuracy: 0.8558 - val_loss: 0.4032
Epoch 92/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 889us/step - accuracy: 0.8740 - loss: 0.3598 - val_a
ccuracy: 0.8559 - val_loss: 0.4029
Epoch 93/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 949us/step - accuracy: 0.8752 - loss: 0.3532 - val_a
ccuracy: 0.8575 - val_loss: 0.4019
Epoch 94/100
```

```
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 946us/step - accuracy: 0.8734 - loss: 0.3547 - val_a
ccuracy: 0.8562 - val_loss: 0.4017
Epoch 95/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 978us/step - accuracy: 0.8727 - loss: 0.3594 - val_a
ccuracy: 0.8560 - val_loss: 0.4020
Epoch 96/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 956us/step - accuracy: 0.8731 - loss: 0.3585 - val_a
ccuracy: 0.8553 - val_loss: 0.4014
Epoch 97/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8727 - loss: 0.3601 - val_acc
uracy: 0.8563 - val_loss: 0.4010
Epoch 98/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8733 - loss: 0.3590 - val_acc
uracy: 0.8564 - val_loss: 0.4006
Epoch 99/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8716 - loss: 0.3598 - val_acc
uracy: 0.8562 - val_loss: 0.4004
Epoch 100/100
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 959us/step - accuracy: 0.8735 - loss: 0.3530 - val_a
ccuracy: 0.8574 - val_loss: 0.4000
Model training completed.
```

Evaluate the model

In [ ]:

```

```

Create a function to prin the confusion matrix

Feel free to use this function below or do your own one

Explain the confusion Matrix with your own words

Import Libraries

- From collections input Counter
- Import confusion_matrix from sklearn
- import itertoos

In [24]:

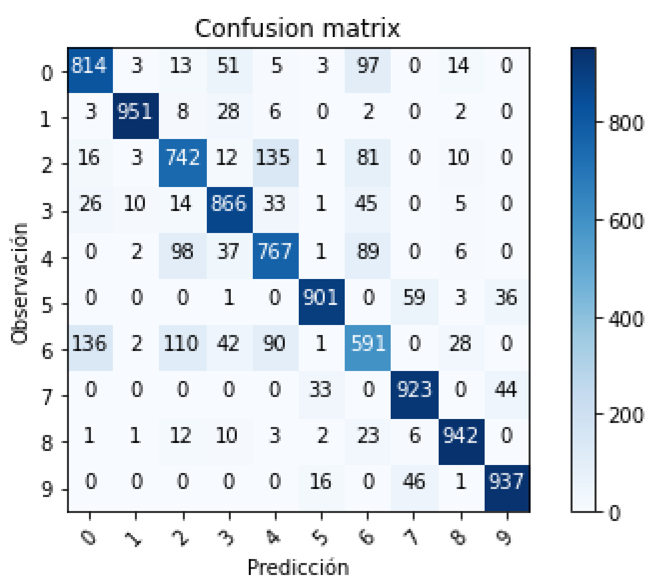
```
from collections import Counter
from sklearn.metrics import confusion_matrix
import itertools
```

In [26]:

```
# Look at confusion matrix
#Note, this code is taken straight from the SKLEARN website, an nice way of viewing
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
```

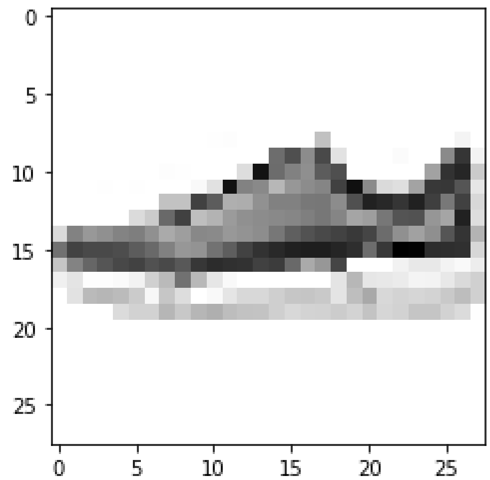
```
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('Observación')
    plt.xlabel('Predicción')
```

In [29]:
```
# Predict the values from the validation dataset
Y_pred = model.predict(X_valid)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_valid, axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(10))
```



In [44]:
```
x_test_old = X_valid.reshape(10000, 28,28)
plt.imshow(x_test_old[9], cmap=plt.cm.binary)
```

Out[44]: <matplotlib.image.AxesImage at 0x17ddea170d0>



In [45]:
```
predictions = model.predict(X_valid)
```

In [46]:
```
np.argmax(predictions[9])
```

Out[46]: 7

In [47]:
```
print(predictions[9])
```

```
[0.   0.   0.   0.   0.   0.01 0.   0.98 0.   0.  ]
```

In [48]:
```
np.sum(predictions[9])
```

Out[48]: 1.0

Explain the results of the confusion matrix with your own words

The confusion matrix is a valuable tool for visualizing how well your classification model is performing. It breaks down the model's predictions into a grid where each row represents the actual classes of the data and each column shows the predicted classes. The diagonal cells, from the top left to the bottom right, indicate the number of correct predictions made by the model—these are your true positives (TP). For example, if the model correctly identified 500 instances of the class '7', you'd see that number in the intersection of row 7 and column 7. On the other hand, the off-diagonal values in a row represent instances where the model made incorrect predictions, known as false positives (FP). For instance, if a true '3' was incorrectly predicted as a '7', that would appear in the row for '3' and the column for '7'. Similarly, the off-diagonal values in a column show false negatives (FN), which are true instances that were misclassified. So, if a true '3' was classified as a '7', it would be reflected in the column for '7' but in the row for '3'. You can also normalize the confusion matrix to show the percentage of correct predictions, making it easier to compare performance across different classes, especially when they are imbalanced. Ideally, a strong model will have high values in the diagonal cells, showing that it's making accurate predictions, while the off-diagonal values should be low, indicating minimal misclassifications. For example, if your matrix shows that most T-shirts and trousers are classified correctly, but there are some instances where T-shirts are misclassified as trousers, it highlights that while the model is generally effective, there are specific areas—like distinguishing between T-shirts and trousers—that need improvement.

Finally, Plot the ReLU function

The Rectified Linear Unit function

The Rectified Linear Unit (ReLU) function is another one of the most commonly used activation functions. It outputs a value from o to infinity. It is basically a piecewise function and can be expressed as follows:

$$ f(x)=\left\{\begin{array}{ll}{0} & {\text { for } x<0} \\ {x} & {\text { for } x \geq 0}\end{array}\right.$$
That is, $f(x)$ returns zero when the value of x is less than zero and $f(x)$ returns x when the value of x is greater than or equal to zero. It can also be expressed as follows:

$$ f(x)=\max (0, x)$$

Create a ReLU function

In [103…

```
def ReLU(x):
    if x<0:
        return 0
    else:
        return x
```

Test the function with a positve value and a negative value

In [106…

```
#Postivie
positive_test = ReLU(5)

print(f'ReLU(5) = {positive_test}')
```

```
ReLU(5) = 5
```

In [108…

```
#Negative
negative_test = ReLU(-3)

print(f'ReLU(-3) = {negative_test}')
```

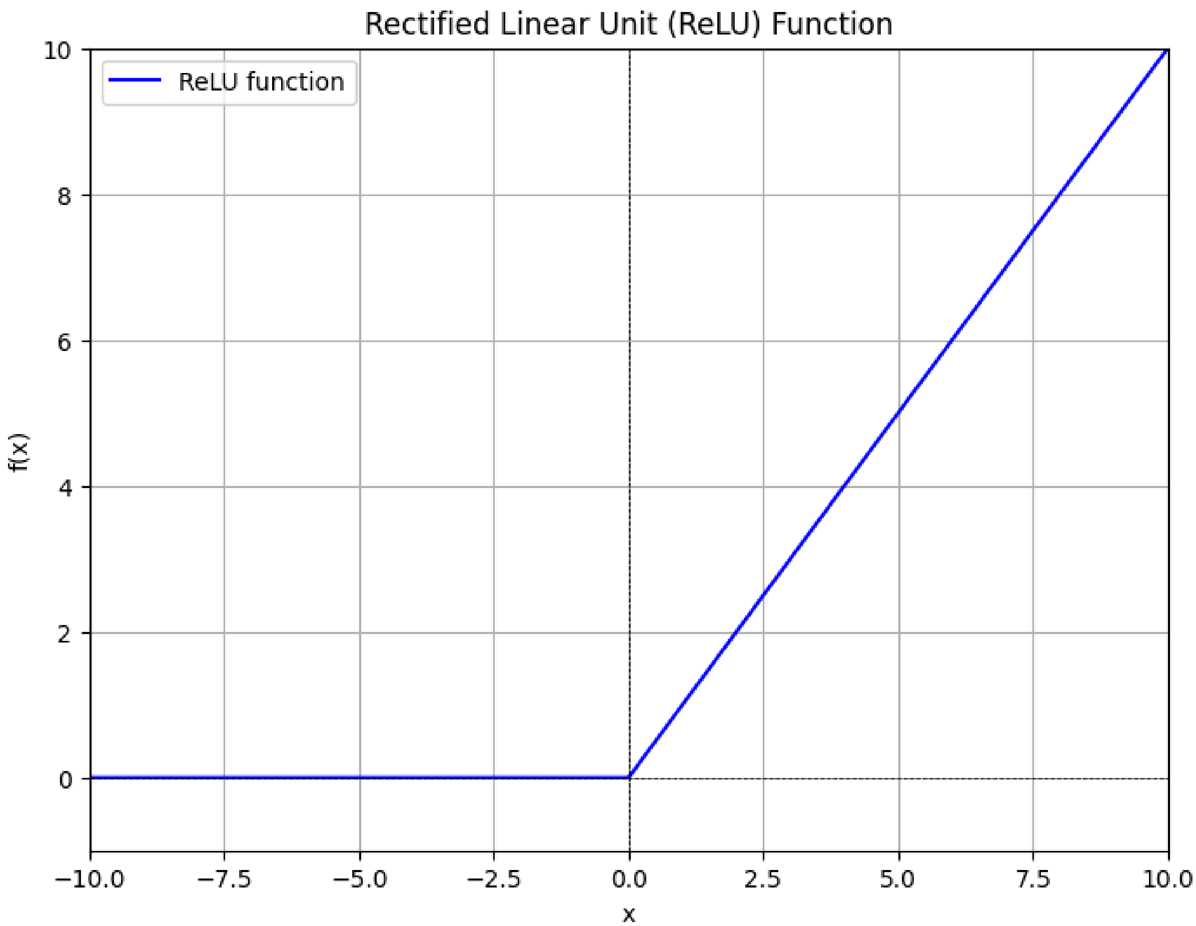
```
ReLU(-3) = 0
```

Plot the ReLU function.

In [111…

```
x_values = np.linspace(-10, 10, 400)
y_values = [ReLU(x) for x in x_values]

plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label='ReLU function', color='blue')
plt.title('Rectified Linear Unit (ReLU) Function')
```

```
plt.xlabel('x')
plt.ylabel('f(x)')
plt.axhline(0, color='black', lw=0.5, ls='--')
plt.axvline(0, color='black', lw=0.5, ls='--')
plt.grid()
plt.legend()
plt.xlim(-10, 10)
plt.ylim(-1, 10)
plt.show()
```



In [ ]:

In [ ]: