In [1]:
```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.optimizers import Adam


from tensorflow.keras.utils import to_categorical
```

Importing Necessary Libraries and Loading the Dataset: TensorFlow and Keras libraries are imported for deep learning. The Fashion MNIST dataset is loaded, which contains grayscale images of clothing items. Image pixel values are normalized to a range between 0 and 1. Labels are converted to categorical one-hot encoding.

Import the Fashion MNIST dataset

This guide uses the Fashion MNIST dataset which contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels), as seen here:

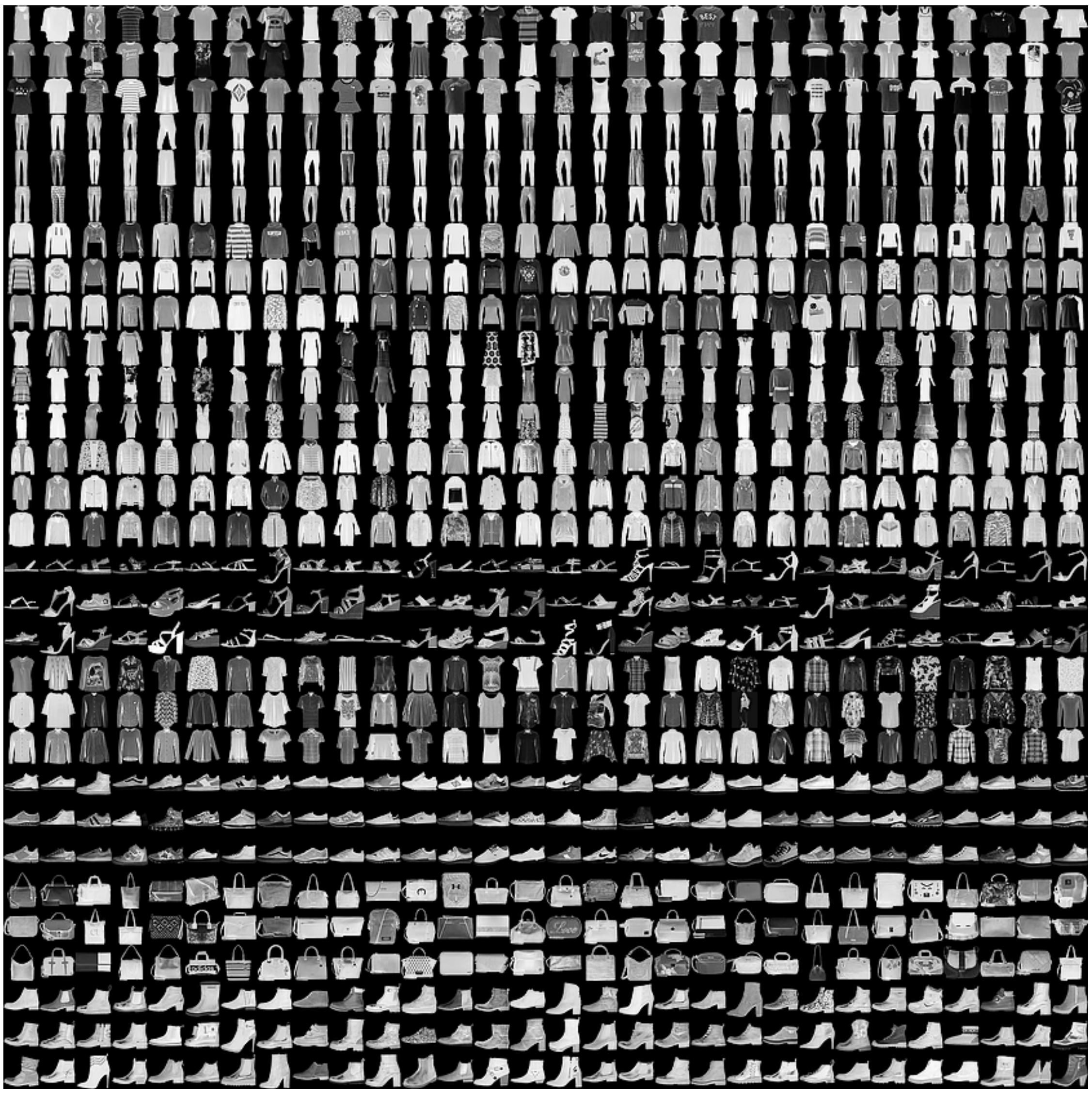


Figure 1. Fashion-MNIST samples (by Zalando, MIT License).

Here, 60,000 images are used to train the network and 10,000 images to evaluate how accurately the network learned to classify images. You can access the Fashion MNIST directly from TensorFlow. Import and load the Fashion MNIST data directly from TensorFlow:

Visualizing a Sample Image from the Dataset: A random image and its corresponding label from the training dataset are displayed. This step helps understand what the data looks like.

In [7]:
```
# Load the dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Normalize pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0

# Convert labels to categorical one-hot encoding
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

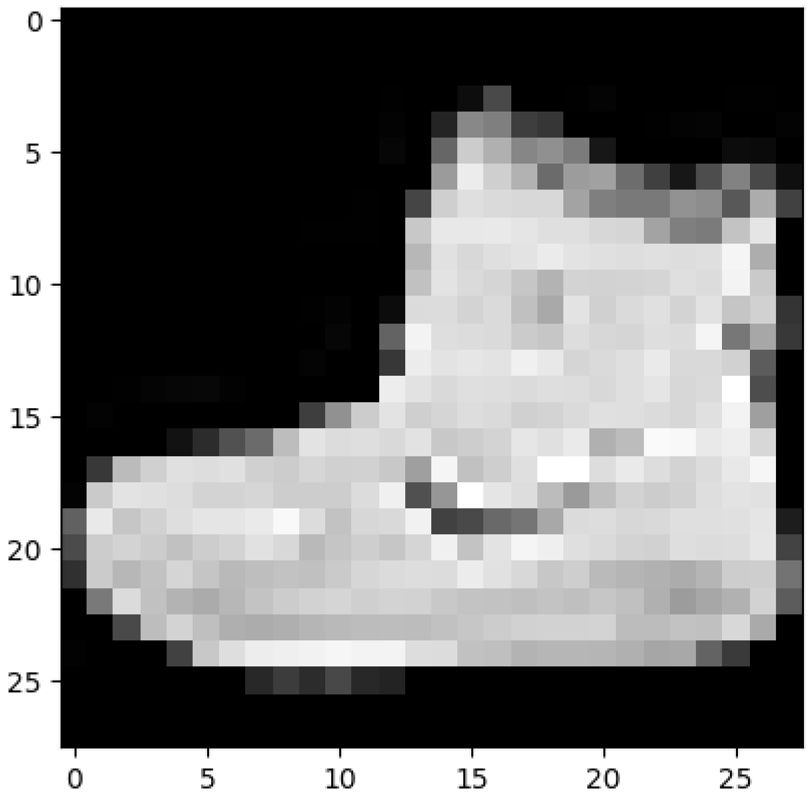
In [12]:
```
y_train[0]
```

Out[12]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1.], dtype=float32)
```

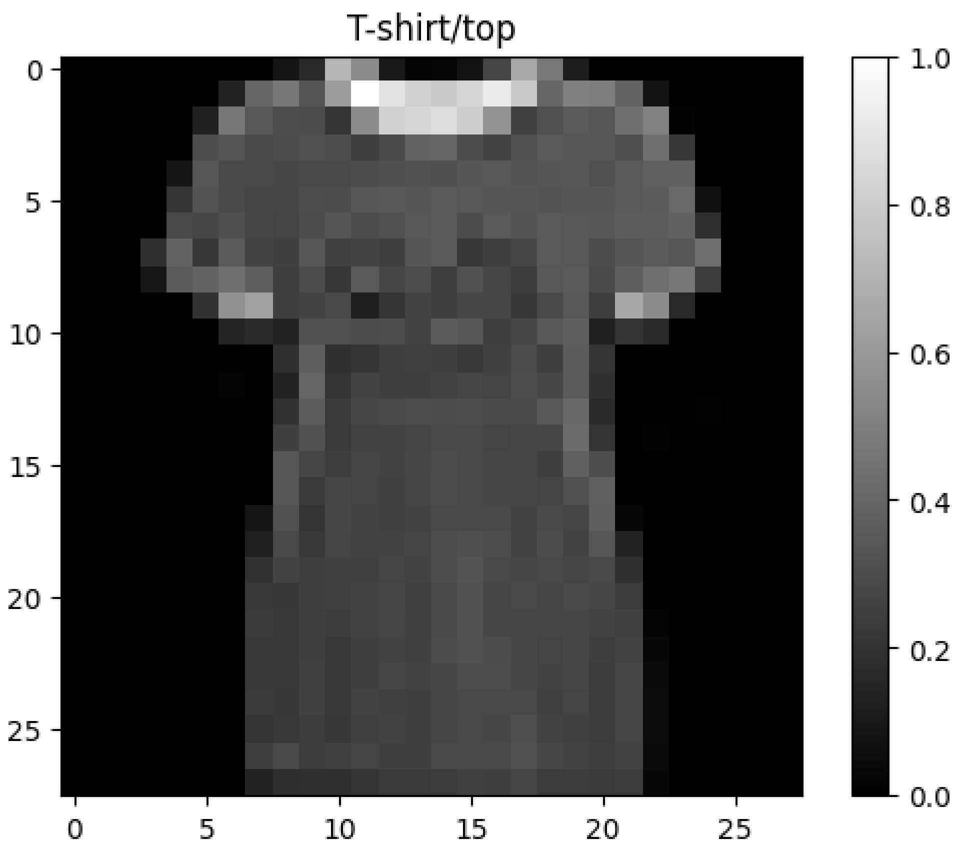
Building the Neural Network Model: A Sequential neural network model is defined. The Flatten layer prepares the image data for fully connected layers. Two Dense layers with ReLU activation introduce non-linearity. The final Dense layer with softmax activation is used for multi-class classification (10 classes).

In [15]:
```
import matplotlib.pyplot as plt
plt.imshow(x_train[0],cmap='gray')
```

Out[15]:
```
<matplotlib.image.AxesImage at 0x7d988af0d690>
```



In [20]:
```
import matplotlib.pyplot as plt
import numpy as np

# Define class names for Fashion MNIST
class_names = [
    "T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
    "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"
]

# Choose a random index from the dataset
index = np.random.randint(0, len(x_train))

# Get the image and label
image = x_train[index]
label = np.argmax(y_train[index])  # Convert one-hot encoded label to class index

# Plot the image
plt.figure()
plt.imshow(image, cmap='gray')
plt.title(class_names[label])
plt.colorbar()
plt.show()
```



In [21]:
```
# Create a Sequential model
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten layer to convert 2D images to 1D arra

    Dense(128, activation='relu'),  # Fully connected Dense layer with 128 neurons

    Dense(64, activation='relu'),   # Fully connected Dense layer with 64 neurons a

    Dense(10, activation='softmax') # Fully connected Dense layer with 10 neurons a
])
```

In [22]:
```
model.compile(optimizer=Adam(),            # Compile the model with the Adam optimi
              loss=CategoricalCrossentropy(), # Use Categorical Crossentropy loss f
              metrics=['accuracy'])           # Evaluate and display accuracy durin
```

Compiling the Model: The model is compiled with the Adam optimizer for training. Categorical Crossentropy loss is chosen as the loss function for multi-class classification. Accuracy is monitored during training.

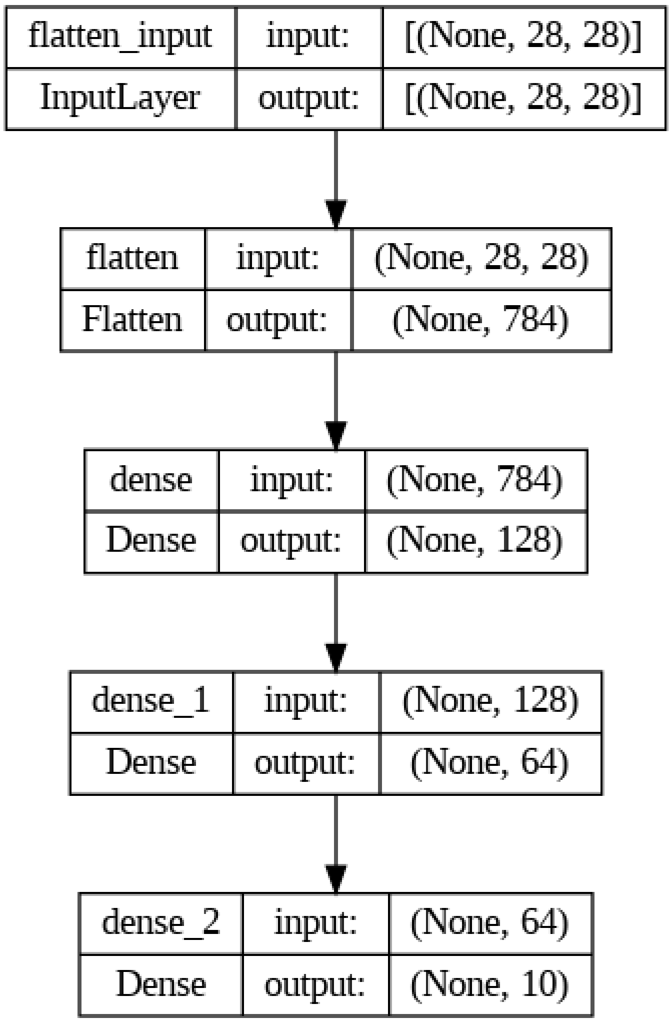
In [23]:
```
model.summary()  # Display a summary of the model's architecture, including layer t
```

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

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 109,386
Trainable params: 109,386
Non-trainable params: 0
_________________________________________________________________
```

Displaying the Model Summary You display a summary of the model architecture, showing the layers, their output shapes, and the number of trainable parameters.

In [24]:
```python
from tensorflow.keras.utils import plot_model

# Plot the model architecture
plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)
```

Out[24]:


flatten_input | input: | [(None, 28, 28)]
InputLayer | output: | [(None, 28, 28)]
flatten | input: | (None, 28, 28)
Flatten | output: | (None, 784)
dense | input: | (None, 784)
Dense | output: | (None, 128)
dense_1 | input: | (None, 128)
Dense | output: | (None, 64)
dense_2 | input: | (None, 64)
Dense | output: | (None, 10)

In [25]:
```python
num_epochs = 10          # Number of training epochs
batch_size = 64          # Number of samples in each training batch

# Train the model on the training data and validate using a portion of it
history = model.fit(x_train, y_train,             # Training data and labels
                    epochs=num_epochs,            # Number of epochs to train
                    batch_size=batch_size,       # Number of samples in each batch
                    validation_split=0.2)        # Fraction of training data to use
```

```
Epoch 1/10
750/750 [==============================] - 5s 5ms/step - loss: 0.5319 - accuracy:
0.8139 - val_loss: 0.4251 - val_accuracy: 0.8499
Epoch 2/10
750/750 [==============================] - 4s 6ms/step - loss: 0.3866 - accuracy:
0.8626 - val_loss: 0.3967 - val_accuracy: 0.8531
Epoch 3/10
750/750 [==============================] - 3s 5ms/step - loss: 0.3480 - accuracy:
0.8730 - val_loss: 0.3767 - val_accuracy: 0.8610
Epoch 4/10
750/750 [==============================] - 3s 5ms/step - loss: 0.3196 - accuracy:
0.8829 - val_loss: 0.3265 - val_accuracy: 0.8813
Epoch 5/10
750/750 [==============================] - 4s 5ms/step - loss: 0.2986 - accuracy:
0.8897 - val_loss: 0.3365 - val_accuracy: 0.8792
Epoch 6/10
750/750 [==============================] - 4s 6ms/step - loss: 0.2860 - accuracy:
0.8940 - val_loss: 0.3180 - val_accuracy: 0.8857
Epoch 7/10
750/750 [==============================] - 4s 5ms/step - loss: 0.2708 - accuracy:
0.8997 - val_loss: 0.3202 - val_accuracy: 0.8861
Epoch 8/10
750/750 [==============================] - 3s 4ms/step - loss: 0.2612 - accuracy:
0.9020 - val_loss: 0.3126 - val_accuracy: 0.8867
Epoch 9/10
750/750 [==============================] - 4s 6ms/step - loss: 0.2501 - accuracy:
0.9084 - val_loss: 0.3156 - val_accuracy: 0.8838
Epoch 10/10
750/750 [==============================] - 4s 5ms/step - loss: 0.2429 - accuracy:
0.9085 - val_loss: 0.3201 - val_accuracy: 0.8864
```

Plotting the Model Architecture You use plot_model to generate an architecture diagram of the model and save it as 'model.png'. This diagram visualizes the layers and connections in the neural network.

In [ ]:
```
# # Get loss values from training history
# train_loss = history.history['loss']
# val_loss = history.history['val_loss']
# epochs = range(1, num_epochs + 1)

# # Plot loss vs. epoch
# plt.figure()
# plt.plot(epochs, train_loss, 'g', label='Training loss')
# plt.plot(epochs, val_loss, 'b', label='Validation loss')
# plt.title('Training and Validation Loss')
# plt.xlabel('Epochs')
# plt.ylabel('Loss')
# plt.legend()
# plt.show()
```

In [26]:
```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print("\nTest accuracy:", test_acc)
```

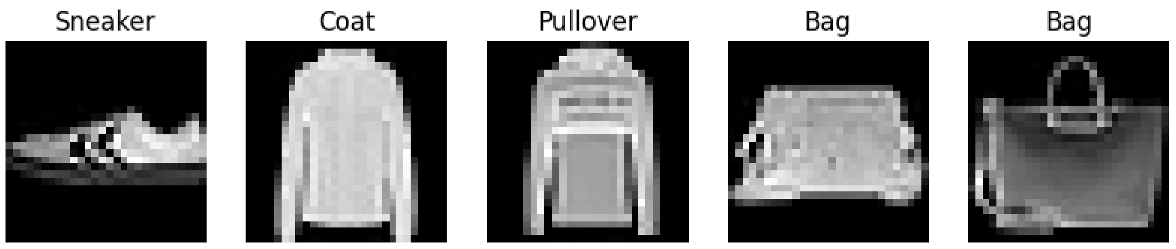
```
313/313 - 0s - loss: 0.3493 - accuracy: 0.8766 - 470ms/epoch - 2ms/step

Test accuracy: 0.8766000270843506
```

. Evaluating the Model on Test Data After training, you evaluate the model's performance on the test data (x_test, y_test) using model.evaluate(). It calculates the test loss and accuracy.

In [29]:
```
# Predict on some test images
num_images_to_predict = 5

random_indices = np.random.randint(0, len(x_test), size=num_images_to_predict)
```

```
test_images = x_test[random_indices]
predicted_labels = model.predict(test_images)
predicted_classes = np.argmax(predicted_labels, axis=1)

# Display the images along with predicted classes
plt.figure(figsize=(10, 5))
for i in range(num_images_to_predict):
    plt.subplot(1, num_images_to_predict, i + 1)
    plt.imshow(test_images[i], cmap='gray')
    plt.title(class_names[predicted_classes[i]])
    plt.axis('off')
plt.show()
```

```
1/1 [==============================] - 0s 36ms/step
```



You randomly select a few test images, use the trained model to make predictions (model.predict()), and then display the images along with their predicted class names.