In [1]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.11.0
```

In [2]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_
```

Loading the dataset returns four NumPy arrays:

- The `train_images` and `train_labels` arrays are the *training set*—the data the model uses to learn.
- The model is tested against the *test set*, the `test_images` , and `test_labels` arrays.

The images are 28x28 NumPy arrays, with pixel values ranging from 0 to 255. The *labels* are an array of integers, ranging from 0 to 9. These correspond to the *class* of clothing the image represents:

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Each image is mapped to a single label. Since the *class names* are not included with the dataset, store them here to use later when plotting the images:

In [3]:
```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

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

```
Out[4]: (60000, 28, 28)
```

Likewise, there are 60,000 labels in the training set:

```
In [5]: len(train_labels)
```

```
Out[5]: 60000
```

Each label is an integer between 0 and 9:

```
In [6]: train_labels
```

```
Out[6]: array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

There are 10,000 images in the test set. Again, each image is represented as 28 x 28 pixels:

```
In [7]: test_images.shape
```

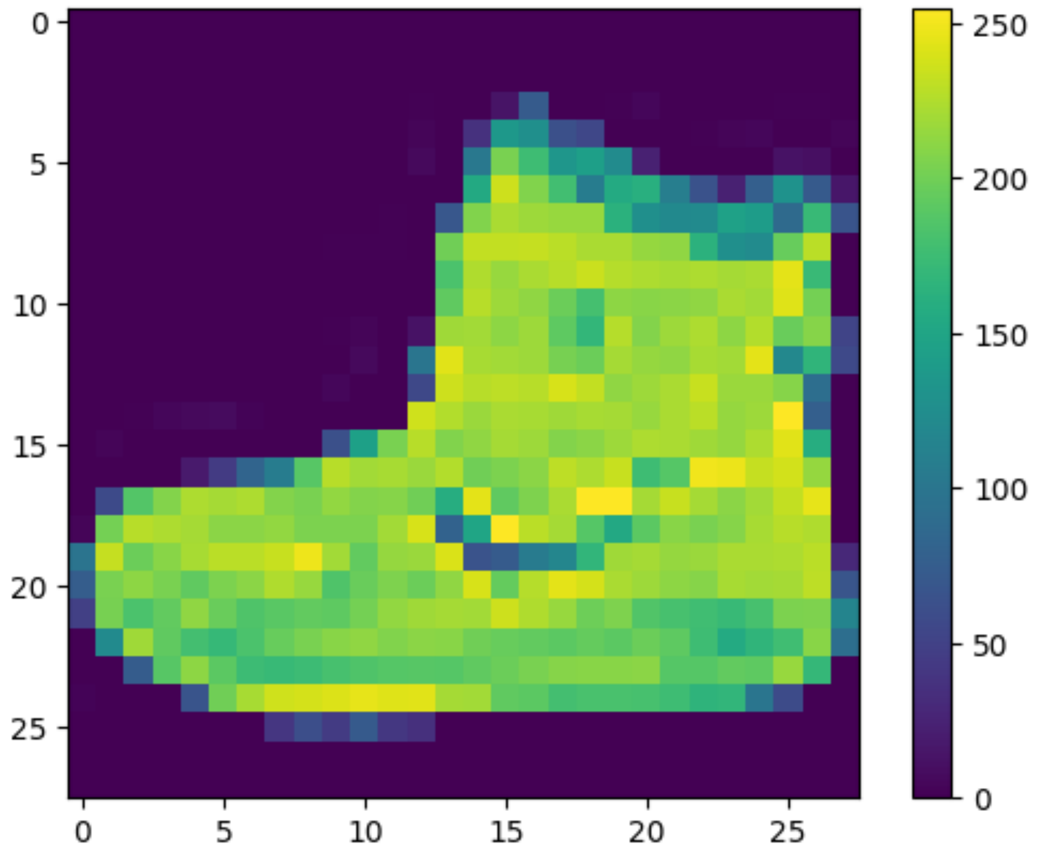
```
Out[7]: (10000, 28, 28)
```

And the test set contains 10,000 images labels:

```
In [8]: len(test_labels)
```

```
Out[8]: 10000
```

In [9]:
```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



Scale these values to a range of 0 to 1 before feeding them to the neural network model. To do so, divide the values by 255. It's important that the *training set* and the *testing set* be preprocessed in the same way:

In [10]:
```
train_images = train_images / 255.0

test_images = test_images / 255.0
```

To verify that the data is in the correct format and that you're ready to build and train the network, let's display the first 25 images from the *training set* and display the class name below each image.

In [11]:
```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Build the model

Building the neural network requires configuring the layers of the model, then compiling the model.

In [12]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

In [13]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(fr
              metrics=['accuracy'])
```

In [14]:
```
model.fit(train_images, train_labels, epochs=30)
```

```
Epoch 1/30
1875/1875 [==============================] - 5s 2ms/step - loss: 0
.5030 - accuracy: 0.8237
Epoch 2/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.3795 - accuracy: 0.8640
Epoch 3/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.3400 - accuracy: 0.8763
Epoch 4/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.3141 - accuracy: 0.8858
Epoch 5/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2990 - accuracy: 0.8900
Epoch 6/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2816 - accuracy: 0.8959
Epoch 7/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2704 - accuracy: 0.8999
Epoch 8/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2594 - accuracy: 0.9031
Epoch 9/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2488 - accuracy: 0.9055
Epoch 10/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2403 - accuracy: 0.9091
Epoch 11/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2317 - accuracy: 0.9128
Epoch 12/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2250 - accuracy: 0.9160
Epoch 13/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2175 - accuracy: 0.9179
Epoch 14/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2124 - accuracy: 0.9202
Epoch 15/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.2054 - accuracy: 0.9225
Epoch 16/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1993 - accuracy: 0.9245
Epoch 17/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1944 - accuracy: 0.9273
Epoch 18/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1894 - accuracy: 0.9286
Epoch 19/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1828 - accuracy: 0.9311
Epoch 20/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1809 - accuracy: 0.9317
Epoch 21/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1749 - accuracy: 0.9338
Epoch 22/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1719 - accuracy: 0.9345
Epoch 23/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1645 - accuracy: 0.9387
Epoch 24/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1624 - accuracy: 0.9388
Epoch 25/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1599 - accuracy: 0.9402
Epoch 26/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1558 - accuracy: 0.9419
Epoch 27/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1524 - accuracy: 0.9431
Epoch 28/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1472 - accuracy: 0.9444
Epoch 29/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1466 - accuracy: 0.9452
Epoch 30/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0
.1420 - accuracy: 0.9469
```

```
Out[14]: <keras.callbacks.History at 0x7f326fff3370>
```

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.91 (or 91%) on the training data.

In [15]:
```
test_loss, test_acc = model.evaluate(test_images,  test_labels, ver

print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.4365 - accuracy: 0.8796 - 481ms/epoch - 2ms
/step

Test accuracy: 0.8795999884605408
```

Make predictions

With the model trained, you can use it to make predictions about some images. Attach a softmax layer to convert the model's linear outputs—logits (https://developers.google.com/machine-learning/glossary#logits)—to probabilities, which should be easier to interpret.

In [16]:
```
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()]
```

In [17]:
```
predictions = probability_model.predict(test_images)
```

```
313/313 [==============================] - 0s 1ms/step
```

Here, the model has predicted the label for each image in the testing set. Let's take a look at the first prediction:

In [18]:
```
predictions[0]
```

Out[18]:
```
array([3.8219037e-12, 6.0263157e-12, 1.1343045e-19, 5.9180233e-16,
       1.9016950e-22, 4.5273669e-08, 5.1865828e-16, 1.9605557e-05,
       1.2793957e-13, 9.9998027e-01], dtype=float32)
```

A prediction is an array of 10 numbers. They represent the model's "confidence" that the image corresponds to each of the 10 different articles of clothing. You can see which label has the highest confidence value:

In [19]:
```
np.argmax(predictions[0])
```

Out[19]:
```
9
```

So, the model is most confident that this image is an ankle boot, or `class_names[9]` . Examining the test label shows that this classification is correct:

In [20]:
```
test_labels[0]
```

Out[20]:
```
9
```

Graph this to look at the full set of 10 class predictions.

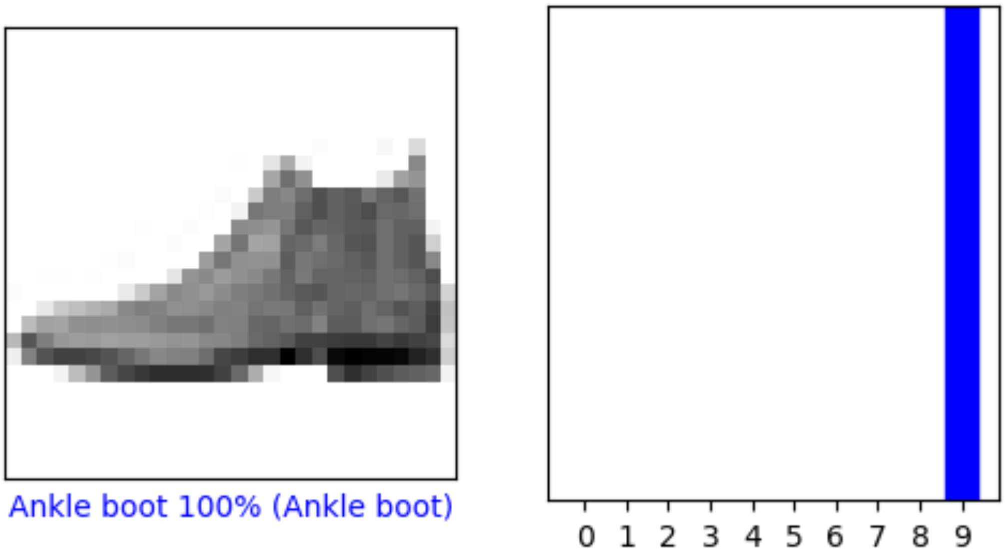
In [21]:
```
def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label]
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

Verify predictions

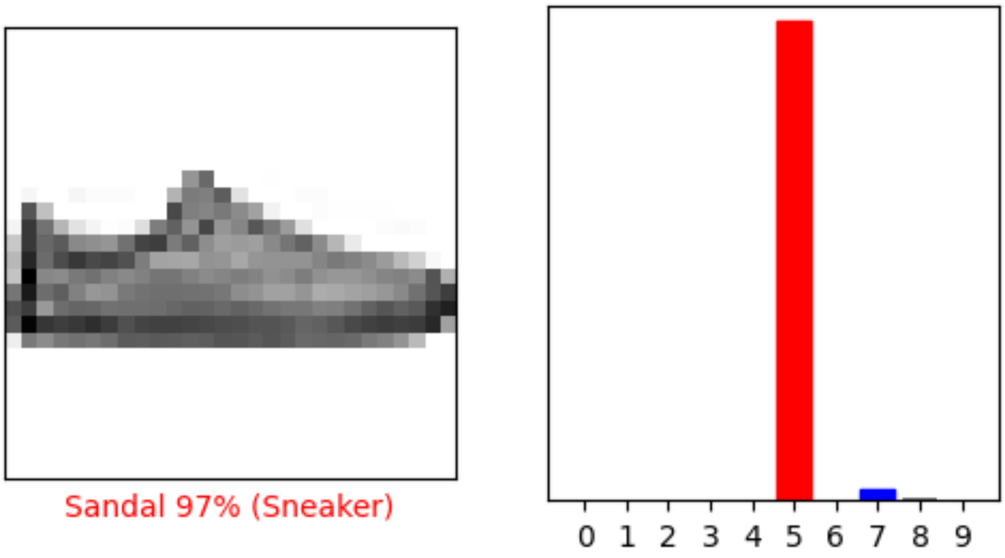
With the model trained, you can use it to make predictions about some images.

Let's look at the 0th image, predictions, and prediction array. Correct prediction labels are blue and incorrect prediction labels are red. The number gives the percentage (out of 100) for the predicted label.

In [22]:
```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```

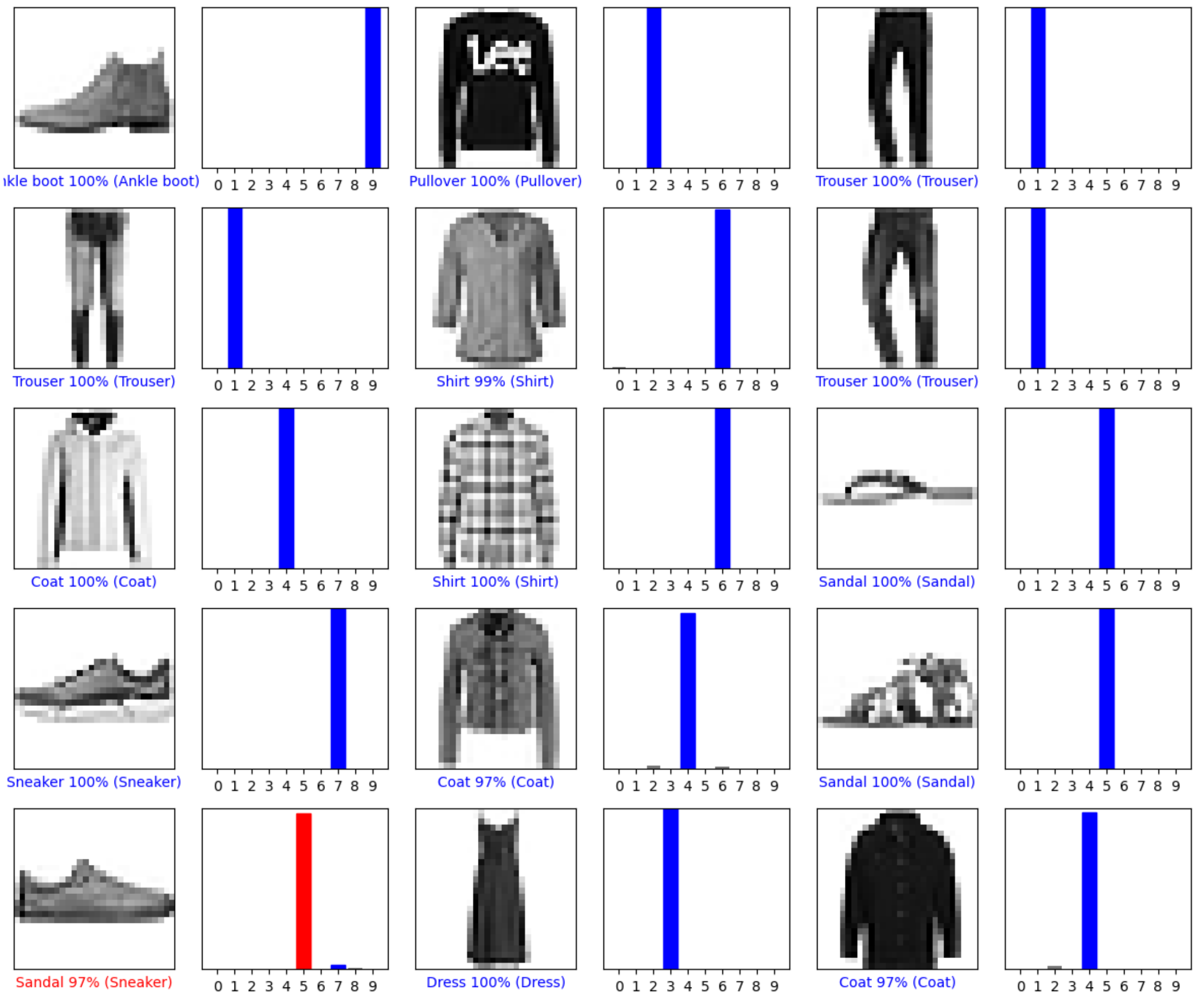


In [23]:
```
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```



Let's plot several images with their predictions. Note that the model can be wrong even when very confident.

In [24]:
```
# Plot the first X test images, their predicted labels, and the tru
# Color correct predictions in blue and incorrect predictions in re
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions[i], test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```



Use the trained model

Finally, use the trained model to make a prediction about a single image.

In [25]:
```
# Grab an image from the test dataset.
img = test_images[1]

print(img.shape)
```

```
(28, 28)
```

`tf.keras` models are optimized to make predictions on a *batch*, or collection, of examples at once. Accordingly, even though you're using a single image, you need to add it to a list:

In [26]:
```
# Add the image to a batch where it's the only member.
img = (np.expand_dims(img,0))

print(img.shape)
```

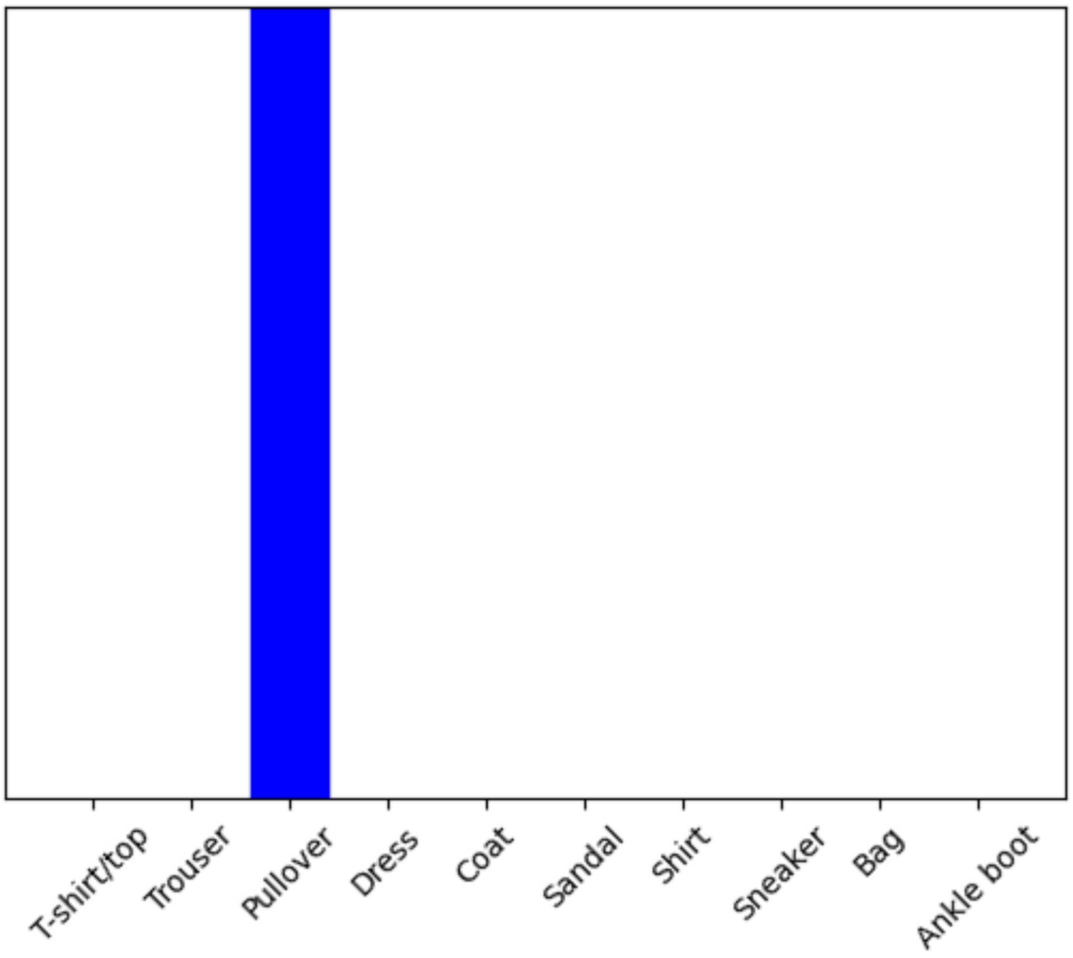
```
(1, 28, 28)
```

Now predict the correct label for this image:

In [27]:
```
predictions_single = probability_model.predict(img)

print(predictions_single)
```

```
1/1 [==============================] - 0s 19ms/step
[[2.3126481e-06 1.3403139e-21 9.9972457e-01 6.3178419e-14 2.613139
2e-04
  7.6748816e-20 1.1800360e-05 6.7637641e-20 1.6529707e-15 6.025850
4e-21]]
```

In [28]:
```
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
```



`tf.keras.Model.predict` returns a list of lists—one list for each image in the batch of data. Grab the predictions for our (only) image in the batch:

In [29]:
```
np.argmax(predictions_single[0])
```

Out[29]:
```
2
```

And the model predicts a label as expected.