はじめてのニューラルネットワーク：分類問題の初歩

View on TensorFlow.org Run in Google Colab View source on GitHub Download notebook

Note: これらのドキュメントは私たちTensorFlowコミュニティが翻訳したものです。コミュニティによる 翻訳は**ベストエフォート**であるため、この翻訳が正確であることや英語の公式ドキュメントの 最新の状態を反映したものであることを保証することはできません。 この翻訳の品質を向上させるためのご意見をお持ちの方は、GitHubリポジトリtensorflow/docsにプルリクエストをお送りください。 コミュニティによる翻訳やレビューに参加していただける方は、docs-ja@tensorflow.org メーリングリストにご連絡ください。

このガイドでは、スニーカーやシャツなど、身に着けるものの写真を分類するニューラルネットワークのモデルを訓練します。すべての詳細を理解できなくても問題ありません。TensorFlowの全体を早足で掴むためのもので、詳細についてはあとから見ていくことになります。

このガイドでは、TensorFlowのモデルを構築し訓練するためのハイレベルのAPIである tf.keras を使用します。

1. ライブラリのロード

```
# TensorFlow と tf.keras のインポート
import tensorflow as tf
from tensorflow import keras

# ヘルパーライブラリのインポート
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

2.8.2

2. ファッションMNISTデータセットのロード

このガイドでは、Fashion MNISTを使用します。Fashion MNISTには10カテゴリーの白黒画像70,000枚が含まれています。それぞれは下図のような1枚に付き1種類の衣料品が写っている低解像度（28×28ピクセル）の画像です。

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

Fashion MNISTは、画像処理のための機械学習での"Hello, World"としてしばしば登場するMNIST データセットの代替として開発されたものです。MNISTデータセットは手書きの数字（0, 1, 2 など）から構成されており、そのフォーマットはこれから使うFashion MNISTと全く同じです。

Fashion MNISTを使うのは、目先を変える意味もありますが、普通のMNISTよりも少しだけ手応えがあるからでもあります。どちらのデータセットも比較的小さく、アルゴリズムが期待したとおりに機能するかどうかを確かめるために使われます。プログラムのテストやデバッグのためには、よい出発点になります。

ここでは、60,000枚の画像を訓練に、10,000枚の画像を、ネットワークが学習した画像分類の正確性を評価するのに使います。TensorFlowを使うと、下記のようにFashion MNISTのデータを簡単にインポートし、ロードすることが出来ます。

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

ロードしたデータセットは、NumPy配列になります。

- `train_images` と `train_labels` の2つの配列は、モデルの訓練に使用される**訓練用データセット**です。
- 訓練されたモデルは、 `test_images` と `test_labels` 配列からなる**テスト用データセット**を使ってテストします。

画像は28×28のNumPy配列から構成されています。それぞれのピクセルの値は0から255の間の整数です。**ラベル**（label）は、0から9までの整数の配列です。それぞれの数字が下表のように、衣料品の**クラス**（class）に対応しています。

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

画像はそれぞれ単一のラベルに分類されます。データセットには上記の**クラス名**が含まれていないため、後ほど画像を出力するときのために、クラス名を保存しておきます。

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

3. データの観察

モデルの訓練を行う前に、データセットのフォーマットを見てみましょう。下記のように、訓練用データセットには28×28ピクセルの画像が60,000枚含まれています。

```
train_images.shape
```

```
(60000, 28, 28)
```

同様に、訓練用データセットには60,000個のラベルが含まれます。

編集するにはダブルクリックするか Enter キーを押してください

```
len(train_labels)
```

```
60000
```

ラベルはそれぞれ、0から9までの間の整数です。

```
train_labels
```

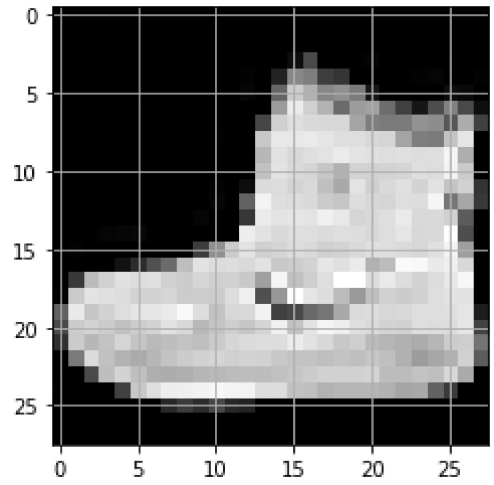
```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

matplotlibライブラリをインポートして訓練用の画像とラベルを確認してみましょう。

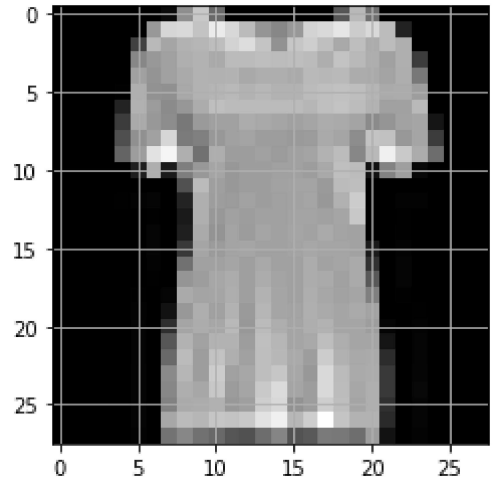
```
import matplotlib
import matplotlib.pyplot as plt

for i in [0,10,100]:
    print("train_labels", "(i="+str(i)+"): ", train_labels[i])
    print("train_images", "(i="+str(i)+"): ")
```

```
plt.imshow(train_images[i], cmap='gray')
plt.grid(True)
plt.show()
```

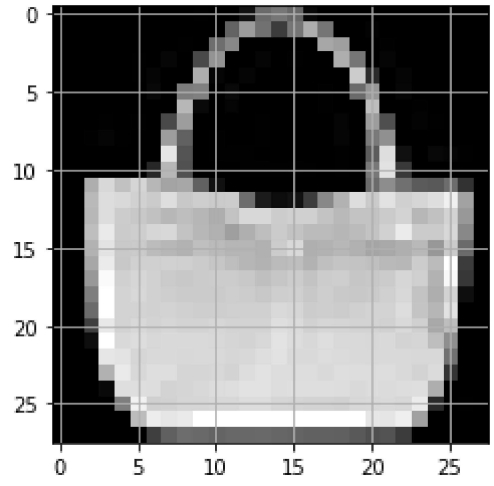
```
train_labels (i=0):  9
train_images (i=0):
```



```
train_labels (i=10):  0
train_images (i=10):
```



```
train_labels (i=100):  8
train_images (i=100):
```



テスト用データセットには、10,000枚の画像が含まれます。画像は28×28ピクセルで構成されています。

編集するにはダブルクリックするか Enter キーを押してください

```
test_images.shape
```

```
(10000, 28, 28)
```

テスト用データセットには10,000個のラベルが含まれます。

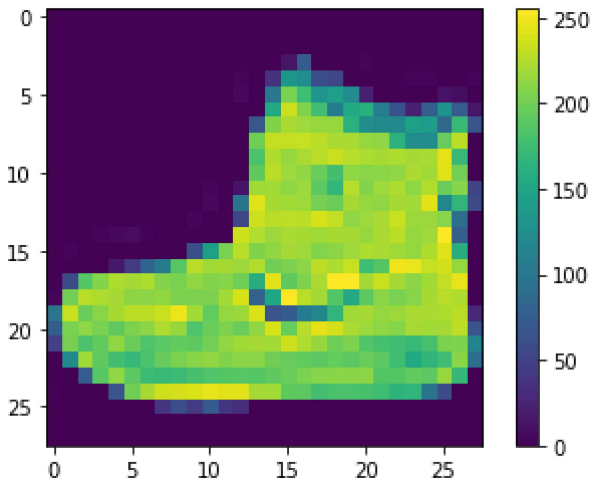
```
len(test_labels)
```

```
10000
```

4. データの前処理

ネットワークを訓練する前に、データを前処理する必要があります。最初の画像を調べてみればわかるように、ピクセルの値は0から255の間の数値です。

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



ニューラルネットワークにデータを投入する前に、これらの値を0から1までの範囲にスケールします。そのためには、画素の値を255で割ります。

訓練用データセットと**テスト用データセット**は、同じように前処理することが重要です。

```
train_images = train_images / 255.0
```

```
test_images = test_images / 255.0
```

訓練用データセットの最初の25枚の画像を、クラス名付きで表示してみましょう。ネットワークを構築・訓練する前に、データが正しいフォーマットになっていることを確認します。

```
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()
```

5. モデルの構築

ニューラルネットワークを構築するには、まずモデルの階層を定義し、その後モデルをコンパイルします。

層の設定

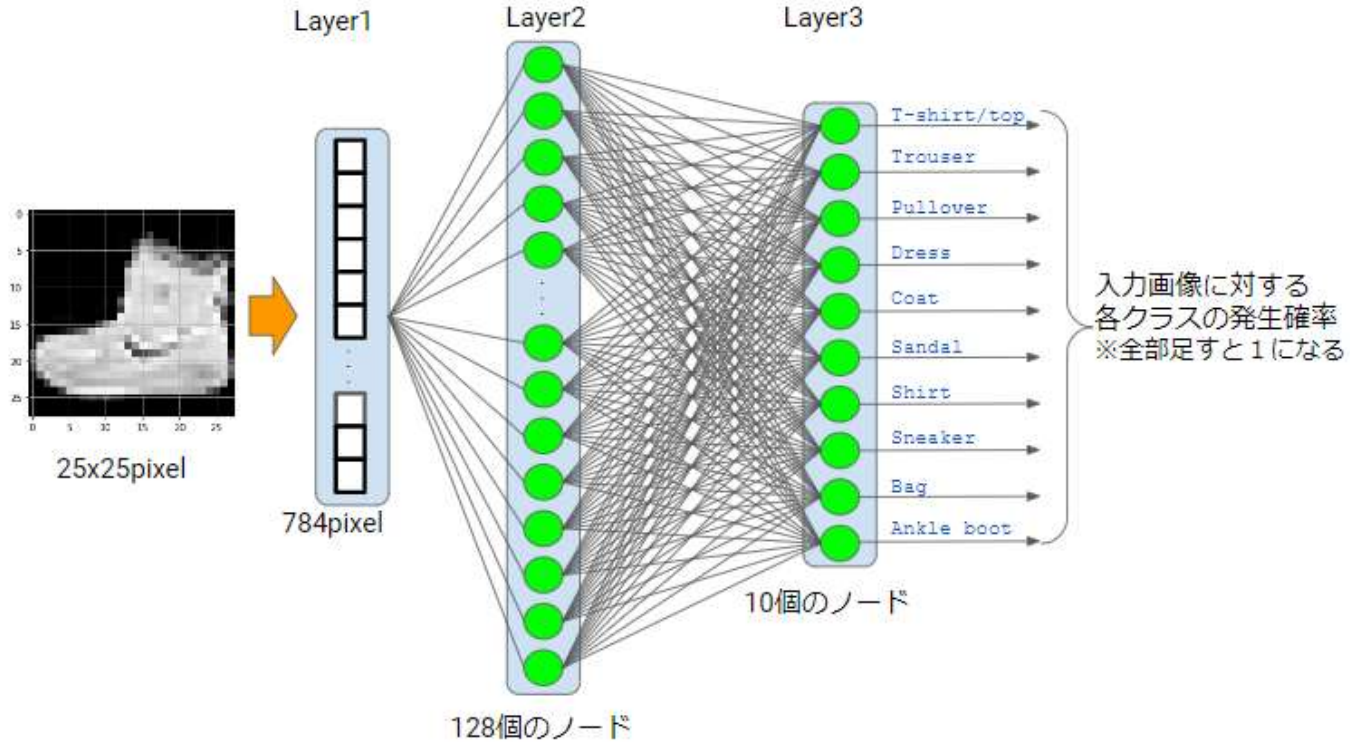
ニューラルネットワークを形作る基本的な構成要素は**層**（layer）です。層は、入力されたデータから「表現」を抽出します。それらの「表現」は、今取り組もうとしている問題に対して、より「意味のある」ものであることが期待されます。

ディープラーニングモデルのほとんどは、単純な層の積み重ねで構成されています。
`tf.keras.layers.Dense` のような層のほとんどには、訓練中に学習されるパラメータが存在します。

```
model = keras.Sequential([
    # (None, 28, 28) -> (None, 784)
    keras.layers.Flatten(input_shape=(28, 28)),
    # Layer1: Linear mapping: (None, 784) -> (None, 128)
    # Activation function: ReLU
    keras.layers.Dense(128, activation='relu'),
    # Layer3: Linear mapping: (None, 128) -> (None, 10)
    # Activation function: Softmax
    keras.layers.Dense(10, activation='softmax')
])
```

このネットワークの最初の層は、 `tf.keras.layers.Flatten` です。この層は、画像を（28×28ピクセルの）2次元配列から、28×28＝784ピクセルの、1次元配列に変換します。この層が、画像の中に積まれているピクセルの行を取り崩し、横に並べると考えてください。この層には学習すべきパラメータはなく、ただデータのフォーマット変換を行うだけです。

ピクセルが１次元化されたあと、ネットワークは2つの `tf.keras.layers.Dense` 層となります。これらの層は、密結合あるいは全結合されたニューロンの層となります。最初の `Dense` 層には、128個のノード（あるはニューロン）があります。最後の層でもある2番めの層は、10ノードの**softmax**層です。この層は、合計が1になる10個の確率の配列を返します。それぞれのノードは、今見ている画像が10個のクラスのひとつひとつに属する確率を出力します。




```
# View model architecture
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_2 (Flatten)          (None, 784)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 128)               100480    
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290      
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

6. モデルのコンパイル

モデルが訓練できるようになるには、いくつかの設定を追加する必要があります。それらの設定は、モデルの**コンパイル**(compile）時に追加されます。

- **損失関数**（loss function） ー訓練中のモデルが不正確であるほど大きな値となる関数です。この関数の値を最小化することにより、訓練中のモデルを正しい方向に向かわせようというわけです。分類問題では交差エントロピー(Cross entropy)が用いられる。
- **オプティマイザ**（optimizer）ーモデルが見ているデータと、損失関数の値から、どのようにモデルを更新するかを決定します。

- **メトリクス**（metrics） ーー訓練とテストのステップを監視するのに使用します。下記の例では*accuracy* （正解率）、つまり、画像が正しく分類された比率を使用しています。

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

7. モデルの訓練

ニューラルネットワークの訓練には次のようなステップが必要です。

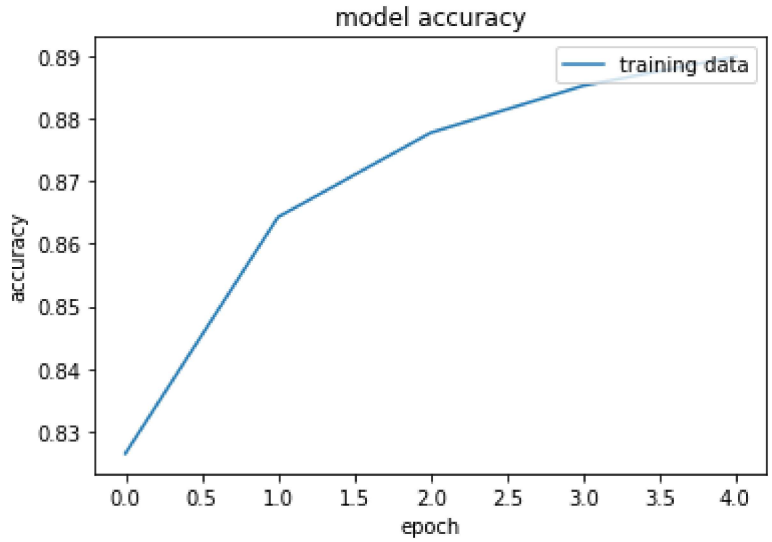
1. モデルに訓練用データを投入しますーこの例では train_images と train_labels の２つの配列です。
2. モデルは、画像とラベルの対応関係を学習します。
3. モデルにテスト用データセットの予測（分類）を行わせますーこの例では test_images 配列です。その後、予測結果と test_labels 配列を照合します。

訓練を開始するには、 model.fit メソッドを呼び出します。モデルを訓練用データに "fit"（適合）させるという意味です。

```
history = model.fit(train_images, train_labels, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.4942 - accuracy: 0.8265
Epoch 2/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3723 - accuracy: 0.8642
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3348 - accuracy: 0.8777
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3142 - accuracy: 0.8852
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2956 - accuracy: 0.8899
```

```
#print('history:', history.history)
plt.plot(history.history['accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['training data'], loc='upper right')
plt.show()
```



モデルの訓練の進行とともに、損失値と正解率が表示されます。このモデルの場合、訓練用データでは0.88（すなわち88%）の正解率に達します。

8. 正解率の評価

次に、テスト用データセットに対するモデルの性能を比較します。

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

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

```
313/313 - 1s - loss: 0.3409 - accuracy: 0.8758 - 508ms/epoch - 2ms/step

Test accuracy: 0.8758000135421753
```

ご覧の通り、テスト用データセットでの正解率は、訓練用データセットでの正解率よりも少し低くなります。この訓練時の正解率とテスト時の正解率の差は、**過学習**（over fitting）の一例です。過学習とは、新しいデータに対する機械学習モデルの性能が、訓練時と比較して低下する現象です。

9. 予測する

モデルの訓練が終わったら、そのモデルを使って画像の分類予測を行うことが出来ます。

```
predictions = model.predict(test_images)
```

これは、モデルがテスト用データセットの画像のひとつひとつを分類予測した結果です。最初の予測を見てみましょう。

```
predictions[0]
```

```
array([1.3520274e-05, 1.9566012e-06, 1.6952136e-04, 1.9968097e-06,
       1.7772116e-05, 3.4522515e-02, 4.3107175e-05, 1.9951023e-02,
       3.5278943e-05, 9.4524330e-01], dtype=float32)
```

予測結果は、10個の数字の配列です。これは、その画像が10の衣料品の種類のそれぞれに該当するかの「確信度」を表しています。どのラベルが一番確信度が高いかを見てみましょう。

```
np.argmax(predictions[0])
```

```
9
```

というわけで、このモデルは、この画像が、アンクルブーツ、 class_names[9] である可能性が最も高いと判断したことになります。これが正しいかどうか、テスト用ラベルを見てみましょう。

```
test_labels[0]
```

```
9
```

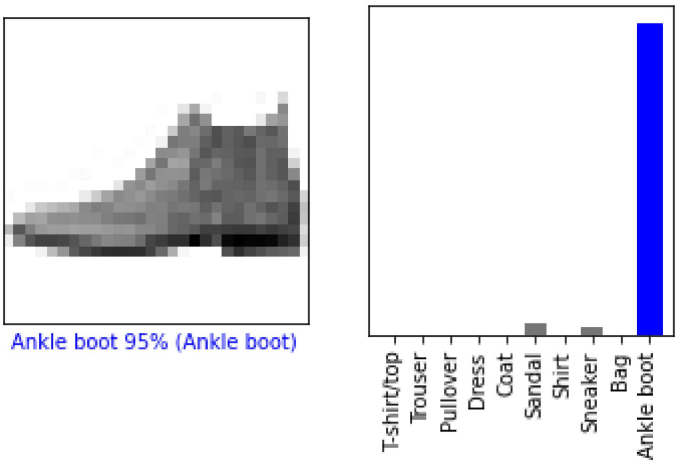
10チャンネルすべてをグラフ化してみることができます。

```
def plot_image(i, predictions_array, true_label, img):
    predictions_array, true_label, img = predictions_array[i], 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):
    predictions_array, true_label = predictions_array[i], true_label[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
```

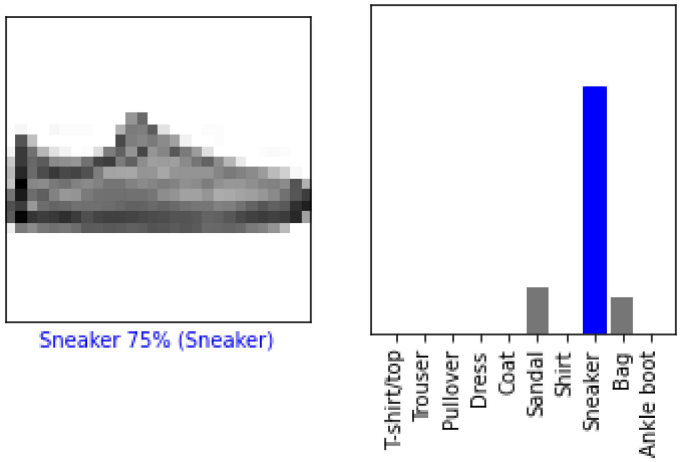
```
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)

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

0番目の画像と、予測、予測配列を見てみましょう。

```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
_ = plt.xticks(range(10), class_names, rotation=90)
plt.show()
```

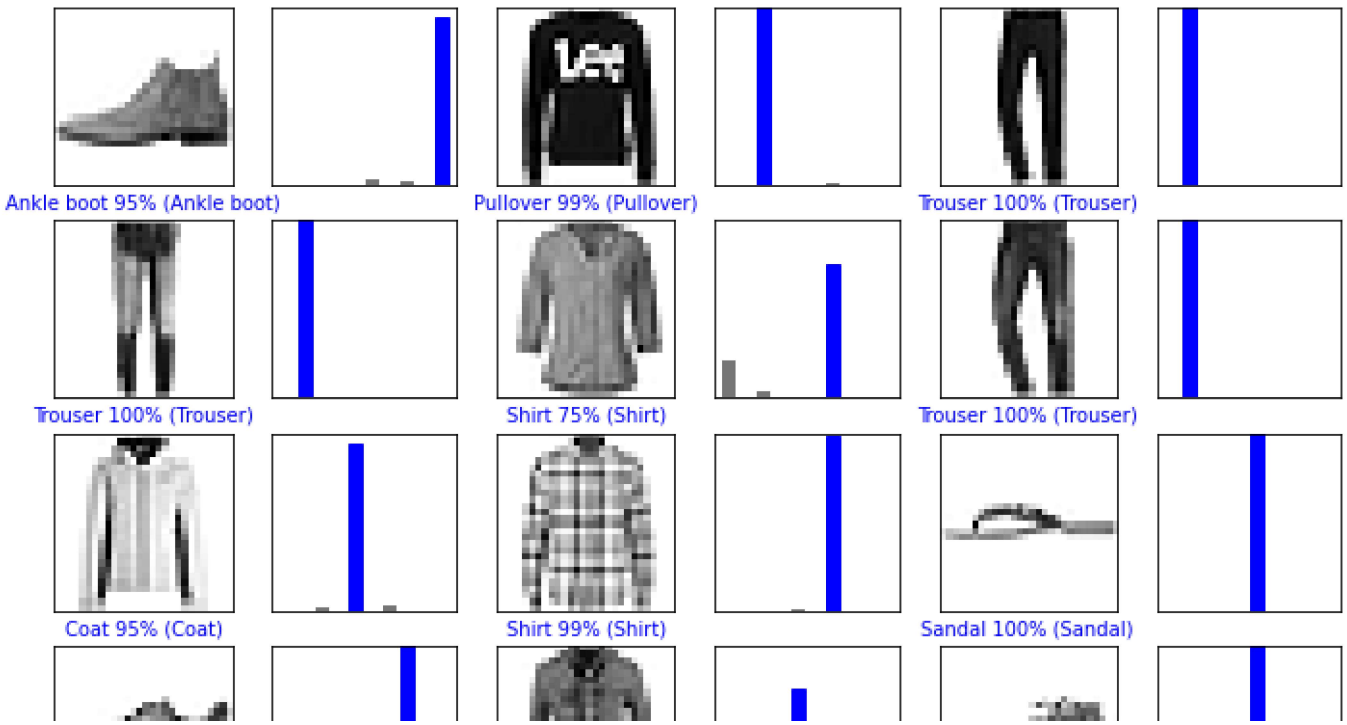



```
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
_ = plt.xticks(range(10), class_names, rotation=90)
plt.show()
```



予測の中のいくつかの画像を、予測値とともに表示してみましょう。正しい予測は青で、誤っている予測は赤でラベルを表示します。数字は予測したラベルのパーセント（100分率）を示します。自信があるように見えても間違っていることがあることに注意してください。()内が正解ラベルの値です。

```
# X個のテスト画像、予測されたラベル、正解ラベルを表示します。
# 正しい予測は青で、間違った予測は赤で表示しています。
num_rows = 10
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, test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions, test_labels)
plt.show()
```



最後に、訓練済みモデルを使って1枚の画像に対する予測を行います。

```
# テスト用データセットから画像を1枚取り出す
img = test_images[0]

print(img.shape)
```

```
(28, 28)
```

tf.keras モデルは、サンプルの中の**バッチ**（batch）あるいは「集まり」について予測を行うように作られています。そのため、1枚の画像を使う場合でも、リスト化する必要があります。

```
# 画像を1枚だけのバッチのメンバーにする
img = (np.expand_dims(img,0))

print(img.shape)
```

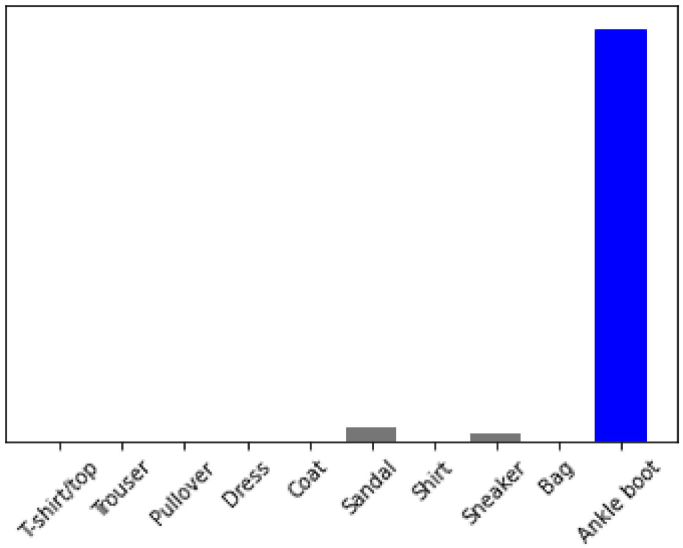
```
(1, 28, 28)
```

そして、予測を行います。

```
predictions_single = model.predict(img)

print(predictions_single)
```

```
[[1.3520288e-05 1.9566030e-06 1.6952169e-04 1.9968099e-06 1.7772083e-05
  3.4522478e-02 4.3107138e-05 1.9951025e-02 3.5278914e-05 9.4524342e-01]]
```

```
plot_value_array(0, predictions_single, test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
```



model.predict メソッドの戻り値は、リストのリストです。リストの要素のそれぞれが、バッチの中の画像に対応します。バッチの中から、（といってもバッチの中身は１つだけですが）予測を取り出します。

```
np.argmax(predictions_single[0])
```

```
9
```

というわけで、モデルは9というラベルを予測しました。