Train your first neural network: Basic classification

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

This guide trains a neural network model to classify images of Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck. It's okay if you don't understand all the details; this is a fast-paced overview of a complete TensorFlow program with the details explained as we go.

This guide uses tf.keras, a high-level API to build and train models in TensorFlow.

```
!pip install tensorflow==2.0.0-beta1
```

```
Collecting tensorflow==2.0.0-beta1
  Downloading https://files.pythonhosted.org/packages/29/6c/2c9a5c4d095c63c2fb37d20de
     |████████████████████████████████| 87.9MB 35kB/s
Collecting tb-nightly<1.14.0a20190604,>=1.14.0a20190603
  Downloading https://files.pythonhosted.org/packages/a4/96/571b875cd81dda9d5dfa1422a
     |████████████████████████████████| 3.1MB 39.7MB/s
Requirement already satisfied: wheel>=0.26 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: wrapt>=1.11.1 in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: google-pasta>=0.1.6 in /usr/local/lib/python3.6/dist-p
Requirement already satisfied: protobuf>=3.6.1 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: keras-applications>=1.0.6 in /usr/local/lib/python3.6/
Requirement already satisfied: absl-py>=0.7.0 in /usr/local/lib/python3.6/dist-packag
Collecting tf-estimator-nightly<1.14.0.dev2019060502,>=1.14.0.dev2019060501
  Downloading https://files.pythonhosted.org/packages/32/dd/99c47dd007dcf10d63fd89561
     |████████████████████████████████| 501kB 40.5MB/s
Requirement already satisfied: grpcio>=1.8.6 in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: keras-preprocessing>=1.0.5 in /usr/local/lib/python3.6
Requirement already satisfied: astor>=0.6.0 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: numpy<2.0,>=1.14.5 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: gast>=0.2.0 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: setuptools>=41.0.0 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: h5py in /usr/local/lib/python3.6/dist-packages (from k
Requirement already satisfied: importlib-metadata; python_version < "3.8" in /usr/loc
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.6/dist-packages (f
Installing collected packages: tb-nightly, tf-estimator-nightly, tensorflow
  Found existing installation: tensorflow 2.2.0
    Uninstalling tensorflow-2.2.0:
      Successfully uninstalled tensorflow-2.2.0
Successfully installed tb-nightly-1.14.0a20190603 tensorflow-2.0.0b1 tf-estimator-nig
```

```
from __future__ import absolute_import, division, print_function, unicode_literals

# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras
```

```
from tensorflow import keras

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/dtypes.py:516: Fut
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/dtypes.py:517: Fut
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/dtypes.py:518: Fut
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/dtypes.py:519: Fut
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/dtypes.py:520: Fut
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/dtypes.py:525: Fut
  np_resource = np.dtype([("resource", np.ubyte, 1)])
2.0.0-beta1
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:5
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:5
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:5
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:5
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:5
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:5
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

Import the CIFAR-10 dataset

The CIFAR-10 dataset (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz) consists of 60000 32x32 colour images in 10 classes (airplanes, cars, birds, cats, deers, dogs, frogs, horses, ships, and trucks), with 6000 images per class. There are 50000 training images and 10000 test images.

The classes are completely mutually exclusive. For example, there is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each:

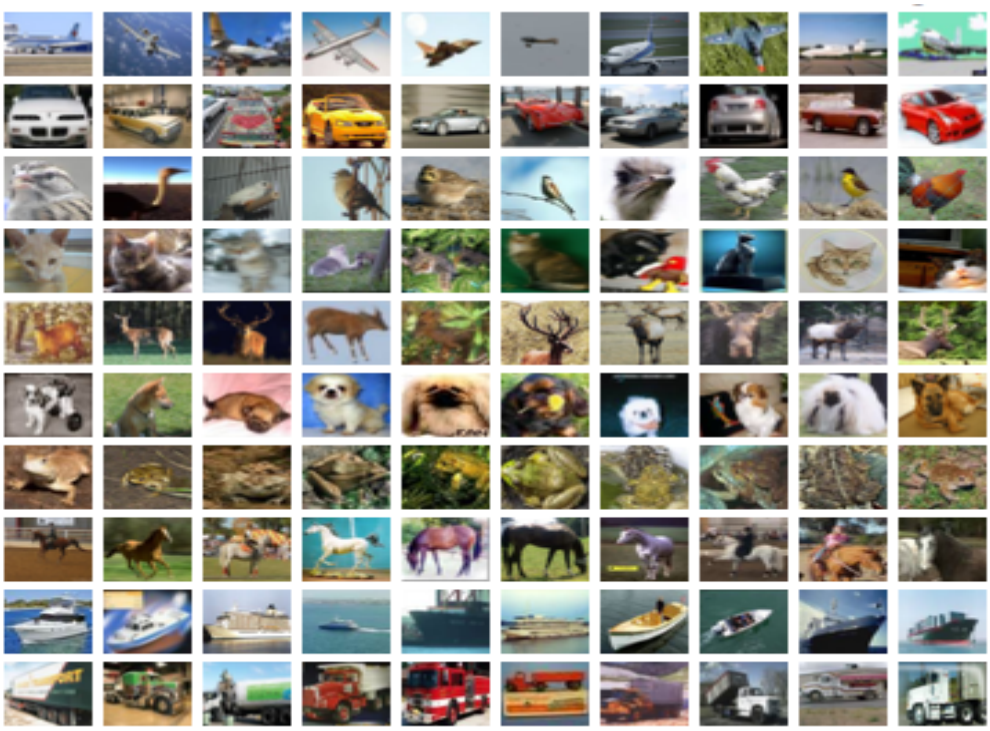


Figure 1. (CIFAR-10).

```
from keras.datasets import cifar10
cifar10 = keras.datasets.cifar10

(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()
```

```
Using TensorFlow backend.
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 6s 0us/step
```

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 32x32x3 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* the image represents:

Label	Class
0	Airplane
1	Automobile
2	Bird
3	Cat
4	Deer
5	Dog
6	Frog
7	Horse
8	Ship

9 Truck

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:

```
class_names = ['Airplane','Automobile','Bird','Cat','Deer','Dog','Frog','Horse','Ship','Tr
```

Explore the data

Let's explore the format of the dataset before training the model. The following shows there are 50,000 images in the training set, with each image represented as 32 x 32 x 3 pixels:

```
train_images.shape
```

```
(50000, 32, 32, 3)
```

Likewise, there are 50,000 labels in the training set:

```
len(train_labels)
```

```
50000
```

Each label is an integer between 0 and 9:

```
train_labels = train_labels.flatten()
```

There are 10,000 images in the test set. Again, each image is represented as 32 x 32 x 3 pixels:

```
test_images.shape
```

```
(10000, 32, 32, 3)
```

And the test set contains 10,000 images labels:

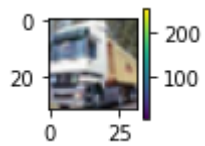
```
len(test_labels)
```

```
10000
```

Preprocess the data

The data must be preprocessed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255:

```
plt.figure(figsize=(1,1))
# plt.figure()
plt.imshow(train_images[1])
plt.colorbar()
plt.grid(False)
plt.show()
```



This is a frog.

```
print(class_names[train_labels[0]])
```

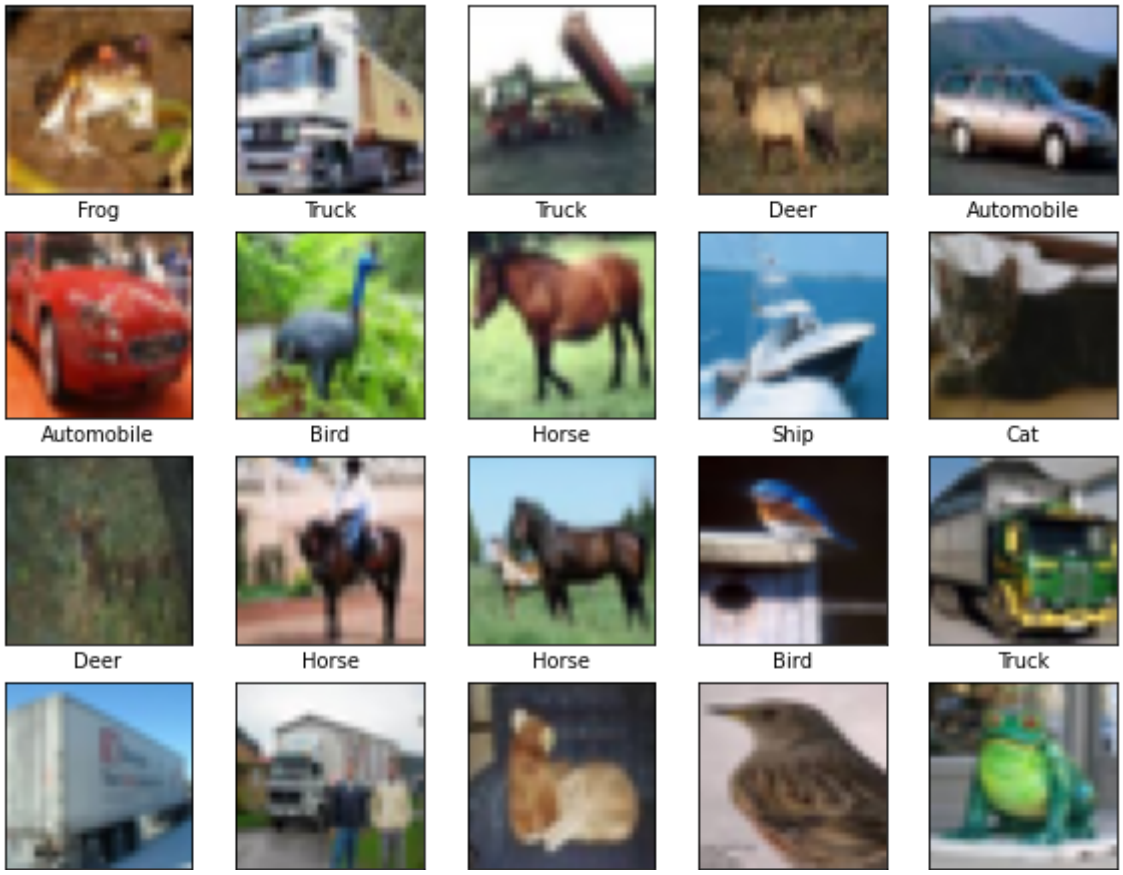
```
Frog
```

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

```
train_images = train_images / 255.0

test_images = test_images / 255.0
```

To verify that the data is in the correct format and that we'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.

```
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.

Deer Cat Frog Frog Bird

Set up the layers

The basic building block of a neural network is the *layer*. Layers extract representations from the data fed into them. Hopefully, these representations are meaningful for the problem at hand.

Most of deep learning consists of chaining together simple layers. Most layers, such as `tf.keras.layers.Dense`, have parameters that are learned during training.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(32, 32, 3)),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dense(256, activation='relu'),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
```

The first layer in this network, `tf.keras.layers.Flatten`, transforms the format of the images from a two-dimensional array (of 32 by 32 pixels) to a one-dimensional array (of 32 * 32 = 1024 pixels). Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn; it only reformats the data.

After the pixels are flattened, the network consists of a sequence of four `tf.keras.layers.Dense` layers. These are densely connected, or fully connected, neural layers. The first `Dense` layer has 512 nodes (or neurons). The second layer has 256 nodes, third layer has 128 nodes and fourth (and last) layer is a 10-node *softmax* layer that returns an array of 10 probability scores that sum to 1. Each node contains a score that indicates the probability that the current image belongs to one of the 10 classes.

Compile the model

Before the model is ready for training, it needs a few more settings. These are added during the model's *compile* step:

- *Loss function* —This measures how accurate the model is during training. We want to minimize this function to "steer" the model in the right direction.
- *Optimizer* —This is how the model is updated based on the data it sees and its loss function.
- *Metrics* —Used to monitor the training and testing steps. The following example uses *accuracy*, the fraction of the images that are correctly classified.

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

Train the model

Training the neural network model requires the following steps:

1. Feed the training data to the model. In this example, the training data is in the `train_images` and `train_labels` arrays.
2. The model learns to associate images and labels.
3. We ask the model to make predictions about a test set — in this example, the `test_images` array. We verify that the predictions match the labels from the `test_labels` array.

To start training, call the `model.fit` method — so-called because it "fits" the model to the training data:

```
history = model.fit(train_images, train_labels, validation_split = 0.3, epochs=10)
```

```
Train on 35000 samples, validate on 15000 samples
Epoch 1/10
35000/35000 [==============================] - 16s 467us/sample - loss: 1.4603 - accu
Epoch 2/10
35000/35000 [==============================] - 16s 468us/sample - loss: 1.4344 - accu
Epoch 3/10
35000/35000 [==============================] - 16s 466us/sample - loss: 1.4072 - accu
Epoch 4/10
35000/35000 [==============================] - 16s 470us/sample - loss: 1.3883 - accu
Epoch 5/10
35000/35000 [==============================] - 16s 468us/sample - loss: 1.3561 - accu
Epoch 6/10
35000/35000 [==============================] - 16s 460us/sample - loss: 1.3416 - accu
```

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.88 (or 88%) on the training data. We used 50 epochs to reach this number.

Evaluate accuracy

Next, compare how the model performs on the test dataset:

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

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

```
10000/10000 [==============================] - 1s 148us/sample - loss: 1.5064 - accur

Test accuracy: 0.4704
```

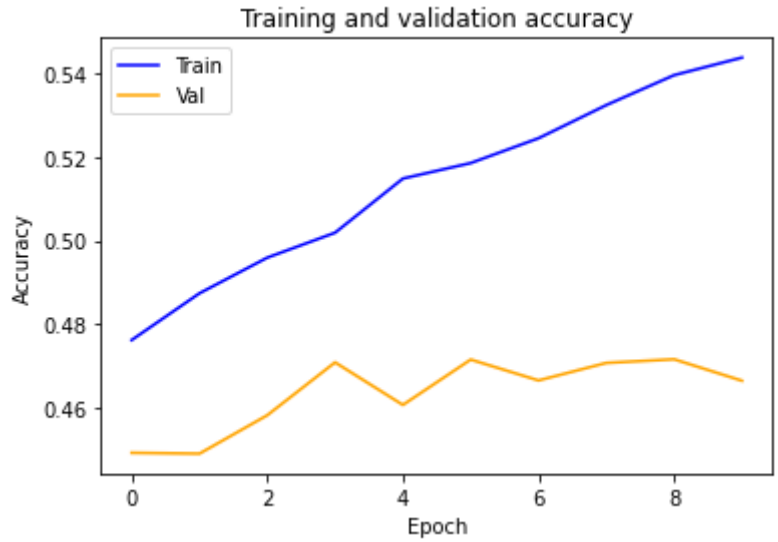
```
# list all data in history
print(history.history.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

# Get the number of epochs
epochs = range(len(acc))

plt.title('Training and validation accuracy')
plt.plot(epochs, acc, color='blue', label='Train')
plt.plot(epochs, val_acc, color='orange', label='Val')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

_ = plt.figure()
plt.title('Training and validation loss')
plt.plot(epochs, loss, color='blue', label='Train')
plt.plot(epochs, val_loss, color='orange', label='Val')
```

```
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f1c75115828>
```



It turns out that the accuracy on the test dataset is a little less than the accuracy on the training dataset. This gap between training accuracy and test accuracy represents *overfitting*. Overfitting is when a machine learning model performs worse on new, previously unseen inputs than on the training data.

Make predictions

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

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

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

```
predictions[0]
```

```
array([0.00195768, 0.00759159, 0.02175678, 0.53222513, 0.09621765,
       0.2509605 , 0.04329256, 0.02013181, 0.02266834, 0.00319788],
      dtype=float32)
```

A prediction is an array of 10 Labels (categories). They represent the model's "confidence" that the image corresponds to each of the classes. We can see which label has the highest confidence value:

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

```
'Cat'
```

The label of the element is below. Is the classification correct or not ?

```
test_labels = test_labels.flatten()
class_names[test_labels[0]]
```

```
'Cat'
```

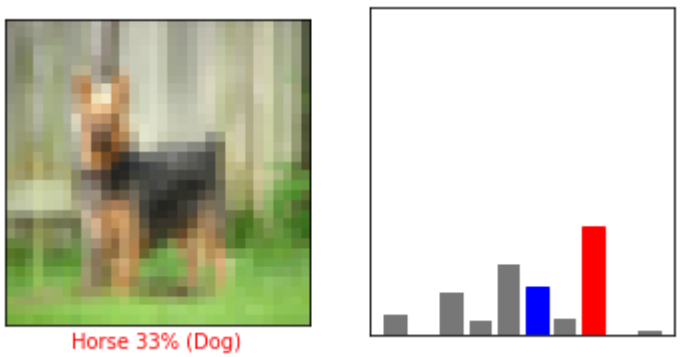
We can graph this to look at the full set of 10 class predictions.

```
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)
```

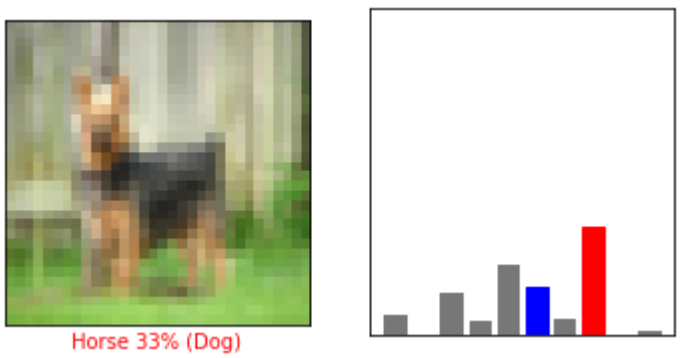
```
  predicted_label = np.argmax(predictions_array)

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

Let's look at the 0th image, predictions, and prediction array.

```
#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.show()
```



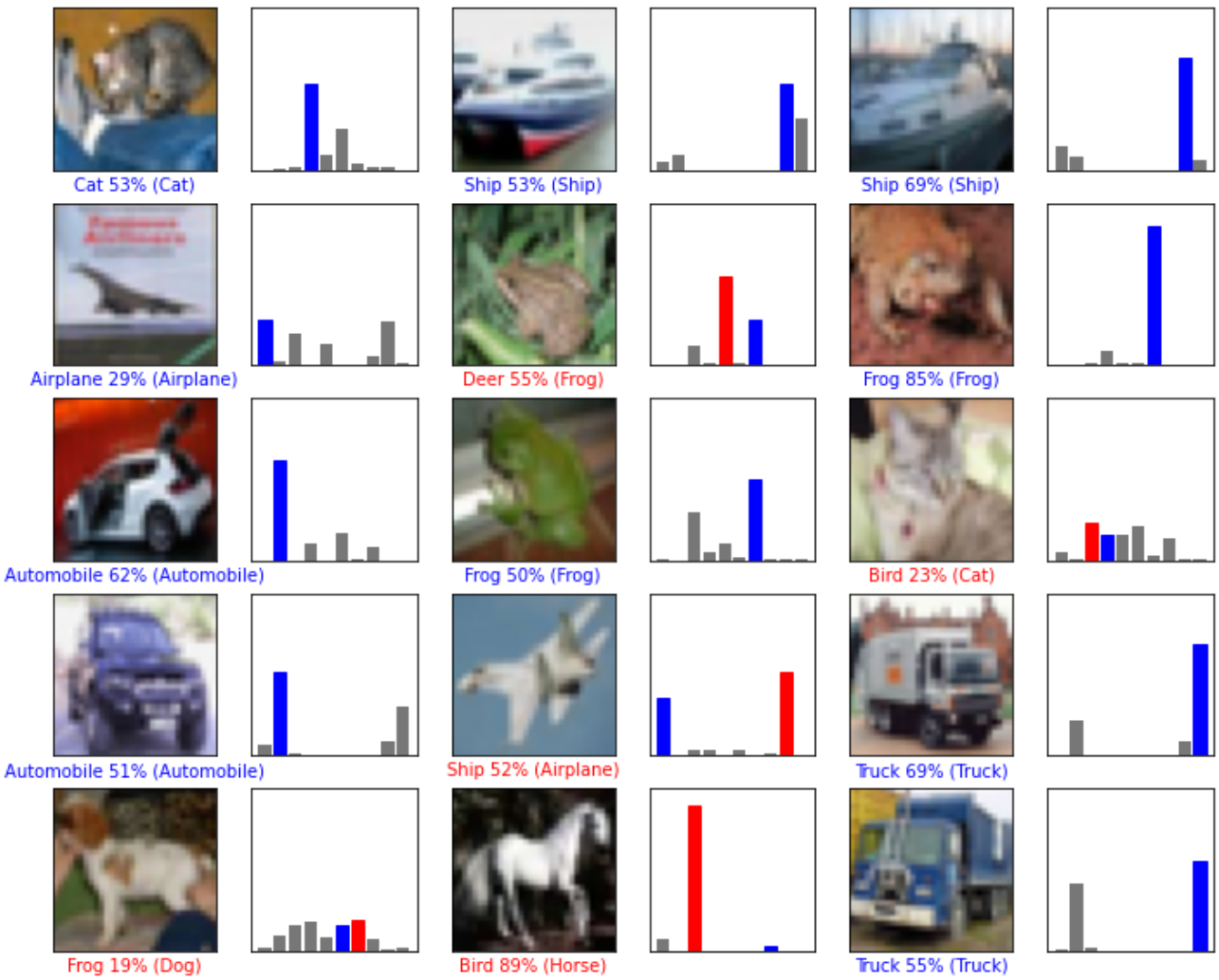

```
#i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
plt.show()
```



Let's plot several images with their predictions. Correct prediction labels are blue and incorrect prediction labels are red. The number gives the percentage (out of 100) for the predicted label. Note that the model can be wrong even when very confident.

```
# Plot the first X test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 5
```

```
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, test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions, test_labels)
plt.show()
```



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

```
# Grab an image from the test dataset.
img = test_images[0]

print(img.shape)
```

```
(32, 32, 3)
```

`tf.keras` models are optimized to make predictions on a *batch*, or collection, of examples at

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

print(img.shape)
```

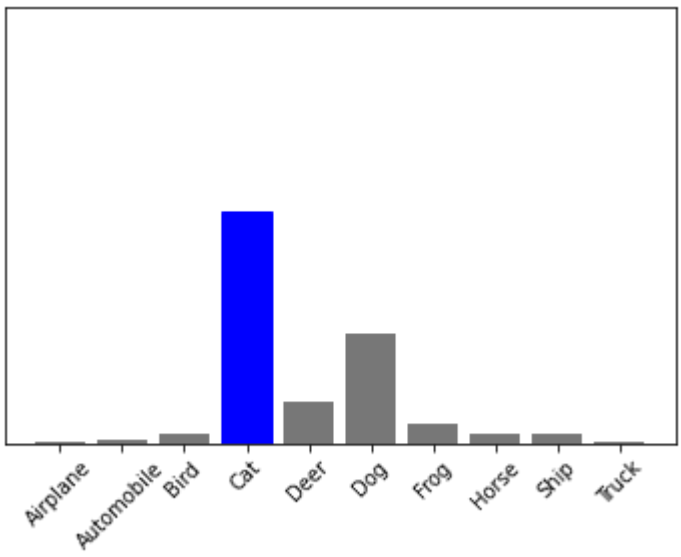
```
(1, 32, 32, 3)
```

Now predict the correct label for this image:

```
predictions_single = model.predict(img)

print(predictions_single)
```

```
[[0.00195769 0.0075916  0.02175679 0.53222495 0.09621781 0.2509604
  0.04329259 0.0201318  0.0226684  0.00319788]]
```

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



`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:

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

```
3
```

And, as before, the model predicts a label of 9.