l03c01_classifying_images_of_clothing

December 8, 2020

1 Classifying Images of Clothing

Run in Google Colab

View source on GitHub

In this tutorial, we'll build and train a neural network to classify images of clothing, like sneakers and shirts.

It's okay if you don't understand everything. This is a fast-paced overview of a complete TensorFlow program, with explanations along the way. The goal is to get the general sense of a TensorFlow project, not to catch every detail.

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

1.1 Install and import dependencies

We'll need TensorFlow Datasets, an API that simplifies downloading and accessing datasets, and provides several sample datasets to work with. We're also using a few helper libraries.

[ ]:
```
!pip install -U tensorflow_datasets
```

[1]:
```
import tensorflow as tf
```

[3]:
```
# Import TensorFlow Datasets
import tensorflow_datasets as tfds
tfds.disable_progress_bar()

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

[2]:
```
import logging
logger = tf.get_logger()
logger.setLevel(logging.ERROR)
```

1.2 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 $\times$ 28 pixels), as seen here:

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

Fashion MNIST is intended as a drop-in replacement for the classic MNIST dataset—often used as the "Hello, World" of machine learning programs for computer vision. The MNIST dataset contains images of handwritten digits (0, 1, 2, etc) in an identical format to the articles of clothing we'll use here.

This guide uses Fashion MNIST for variety, and because it's a slightly more challenging problem than regular MNIST. Both datasets are relatively small and are used to verify that an algorithm works as expected. They're good starting points to test and debug code.

We will use 60,000 images 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, using the Datasets API:

[17]:
```python
dataset, metadata = tfds.load('fashion_mnist', as_supervised=True,␣
 ↪with_info=True, data_dir="/content/tensorflow_datasets")
train_dataset, test_dataset = dataset['train'], dataset['test']
```

```
Downloading and preparing dataset fashion_mnist/3.0.0 (download: 29.45 MiB,
generated: Unknown size, total: 29.45 MiB) to
/content/tensorflow_datasets/fashion_mnist/3.0.0...
Shuffling and writing examples to
/content/tensorflow_datasets/fashion_mnist/3.0.0.incompleteIPOXQL/fashion_mnist-
train.tfrecord
Shuffling and writing examples to
/content/tensorflow_datasets/fashion_mnist/3.0.0.incompleteIPOXQL/fashion_mnist-
test.tfrecord
Dataset fashion_mnist downloaded and prepared to
/content/tensorflow_datasets/fashion_mnist/3.0.0. Subsequent calls will reuse
this data.
```

Loading the dataset returns metadata as well as a *training dataset* and *test dataset*.

- The model is trained using `train_dataset`.
- The model is tested against `test_dataset`.

The images are 28×28 arrays, with pixel values in the range `[0, 255]`. The *labels* are an array of integers, in the range `[0, 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:

[9]:
```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal',      'Shirt',   'Sneaker',  'Bag',   'Ankle boot']
```

1.2.1 Explore the data

Let's explore the format of the dataset before training the model. The following shows there are 60,000 images in the training set, and 10000 images in the test set:

[ ]:
```
num_train_examples = metadata.splits['train'].num_examples
num_test_examples = metadata.splits['test'].num_examples
print("Number of training examples: {}".format(num_train_examples))
print("Number of test examples:     {}".format(num_test_examples))
```

1.3 Preprocess the data

The value of each pixel in the image data is an integer in the range `[0,255]`. For the model to work properly, these values need to be normalized to the range `[0,1]`. So here we create a normalization function, and then apply it to each image in the test and train datasets.

[11]:
```
train_dataset
```

[11]:
```
<DatasetV1Adapter shapes: ((28, 28, 1), ()), types: (tf.uint8, tf.int64)>
```

[12]:
```
tf.cast(train_dataset, tf.float32)
train_dataset
```

```
       ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-12-41147b629d3c> in <module>()
   ----> 1 tf.cast(train_dataset, tf.float32)
         2 train_dataset


       /usr/local/lib/python3.6/dist-packages/tensorflow/python/util/dispatch.
↪py in wrapper(*args, **kwargs)
       199     """Call target, and fall back on dispatchers if there is a␣
↪TypeError."""
       200     try:
   --> 201       return target(*args, **kwargs)
       202     except (TypeError, ValueError):
       203       # Note: convert_to_eager_tensor currently raises a ValueError,␣
↪not a
```

```
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py␣
↪in cast(x, dtype, name)
    919       # allows some conversions that cast() can't do, e.g. casting␣
↪numbers to
    920       # strings.
--> 921       x = ops.convert_to_tensor(x, name="x")
    922       if x.dtype.base_dtype != base_type:
    923         x = gen_math_ops.cast(x, base_type, name=name)


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/ops.
↪py in convert_to_tensor(value, dtype, name, as_ref, preferred_dtype,␣
↪dtype_hint, ctx, accepted_result_types)
   1497
   1498     if ret is None:
-> 1499       ret = conversion_func(value, dtype=dtype, name=name,␣
↪as_ref=as_ref)
   1500
   1501     if ret is NotImplemented:


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/
↪constant_op.py in _constant_tensor_conversion_function(v, dtype, name, as_ref)
    336                                          as_ref=False):
    337   _ = as_ref
--> 338   return constant(v, dtype=dtype, name=name)
    339
    340


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/
↪constant_op.py in constant(value, dtype, shape, name)
    262   """
    263   return _constant_impl(value, dtype, shape, name,␣
↪verify_shape=False,
--> 264                         allow_broadcast=True)
    265
    266


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/
↪constant_op.py in _constant_impl(value, dtype, shape, name, verify_shape,␣
↪allow_broadcast)
    273       with trace.Trace("tf.constant"):
    274         return _constant_eager_impl(ctx, value, dtype, shape,␣
↪verify_shape)
```

```
  --> 275     return _constant_eager_impl(ctx, value, dtype, shape,␣
↪verify_shape)
      276
      277   g = ops.get_default_graph()


      /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/
↪constant_op.py in _constant_eager_impl(ctx, value, dtype, shape, verify_shape)
      298 def _constant_eager_impl(ctx, value, dtype, shape, verify_shape):
      299   """Implementation of eager constant."""
  --> 300   t = convert_to_eager_tensor(value, ctx, dtype)
      301   if shape is None:
      302     return t


      /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/
↪constant_op.py in convert_to_eager_tensor(value, ctx, dtype)
       96       dtype = dtypes.as_dtype(dtype).as_datatype_enum
       97   ctx.ensure_initialized()
  ---> 98   return ops.EagerTensor(value, ctx.device_name, dtype)
       99
      100


      ValueError: Attempt to convert a value (<DatasetV1Adapter shapes: ((28,␣
↪28, 1), ()), types: (tf.uint8, tf.int64)>) with an unsupported type (<class␣
↪'tensorflow.python.data.ops.dataset_ops.DatasetV1Adapter'>) to a Tensor.
```

[ ]:
```
def normalize(images, labels):
  images = tf.cast(images, tf.float32)
  images /= 255
  return images, labels

# The map function applies the normalize function to each element in the train
# and test datasets
train_dataset =  train_dataset.map(normalize)
test_dataset  =  test_dataset.map(normalize)

# The first time you use the dataset, the images will be loaded from disk
# Caching will keep them in memory, making training faster
train_dataset =  train_dataset.cache()
test_dataset  =  test_dataset.cache()
```

1.3.1 Explore the processed data

Let's plot an image to see what it looks like.

[ ]:
```
# Take a single image, and remove the color dimension by reshaping
for image, label in test_dataset.take(1):
  break
image = image.numpy().reshape((28,28))

# Plot the image - voila a piece of fashion clothing
plt.figure()
plt.imshow(image, cmap=plt.cm.binary)
plt.colorbar()
plt.grid(False)
plt.show()
```

Display the first 25 images from the *training set* and display the class name below each image. Verify that the data is in the correct format and we're ready to build and train the network.

[ ]:
```
plt.figure(figsize=(10,10))
i = 0
for (image, label) in test_dataset.take(25):
    image = image.numpy().reshape((28,28))
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(image, cmap=plt.cm.binary)
    plt.xlabel(class_names[label])
    i += 1
plt.show()
```

1.4 Build the model

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

1.4.1 Setup the layers

The basic building block of a neural network is the *layer*. A layer extracts a representation from the data fed into it. Hopefully, a series of connected layers results in a representation that is meaningful for the problem at hand.

Much of deep learning consists of chaining together simple layers. Most layers, like `tf.keras.layers.Dense`, have internal parameters which are adjusted ("learned") during training.

[ ]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

This network has three layers:

- **input** `tf.keras.layers.Flatten` — This layer transforms the images from a 2d-array of 28 × 28 pixels, to a 1d-array of 784 pixels (28*28). Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn, as it only reformats the data.
- **"hidden"** `tf.keras.layers.Dense`— A densely connected layer of 128 neurons. Each neuron (or node) takes input from all 784 nodes in the previous layer, weighting that input according to hidden parameters which will be learned during training, and outputs a single value to the next layer.
- **output** `tf.keras.layers.Dense` — A 128-neuron, followed by 10-node *softmax* layer. Each node represents a class of clothing. As in the previous layer, the final layer takes input from the 128 nodes in the layer before it, and outputs a value in the range `[0, 1]`, representing the probability that the image belongs to that class. The sum of all 10 node values is 1.

  Note: Using `softmax` activation and `SparseCategoricalCrossentropy()` has issues and which are patched by the `tf.keras` model. A safer approach, in general, is to use a linear output (no activation function) with `SparseCategoricalCrossentropy(from_logits=True)`.

1.4.2 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* — An algorithm for measuring how far the model's outputs are from the desired output. The goal of training is this measures loss.
- *Optimizer* —An algorithm for adjusting the inner parameters of the model in order to minimize loss.
- *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=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
```

1.5 Train the model

First, we define the iteration behavior for the train dataset: 1. Repeat forever by specifying `dataset.repeat()` (the `epochs` parameter described below limits how long we perform training). 2. The `dataset.shuffle(60000)` randomizes the order so our model cannot learn anything from the order of the examples. 3. And `dataset.batch(32)` tells `model.fit` to use batches of 32 images and labels when updating the model variables.

Training is performed by calling the `model.fit` method: 1. Feed the training data to the model using `train_dataset`. 2. The model learns to associate images and labels. 3. The `epochs=5` parameter limits training to 5 full iterations of the training dataset, so a total of 5 * 60000 = 300000 examples.

(Don't worry about `steps_per_epoch`, the requirement to have this flag will soon be removed.)

[ ]:
```
BATCH_SIZE = 32
train_dataset = train_dataset.cache().repeat().shuffle(num_train_examples).
 ↪batch(BATCH_SIZE)
test_dataset = test_dataset.cache().batch(BATCH_SIZE)
```

[ ]:
```
model.fit(train_dataset, epochs=5, steps_per_epoch=math.ceil(num_train_examples/
 ↪BATCH_SIZE))
```

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.

1.6 Evaluate accuracy

Next, compare how the model performs on the test dataset. Use all examples we have in the test dataset to assess accuracy.

[ ]:
```
test_loss, test_accuracy = model.evaluate(test_dataset, steps=math.
 ↪ceil(num_test_examples/32))
print('Accuracy on test dataset:', test_accuracy)
```

As it turns out, the accuracy on the test dataset is smaller than the accuracy on the training dataset. This is completely normal, since the model was trained on the `train_dataset`. When the model sees images it has never seen during training, (that is, from the `test_dataset`), we can expect performance to go down.

1.7 Make predictions and explore

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

[ ]:
```
for test_images, test_labels in test_dataset.take(1):
  test_images = test_images.numpy()
  test_labels = test_labels.numpy()
  predictions = model.predict(test_images)
```

[ ]:
```
predictions.shape
```

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]
```

A prediction is an array of 10 numbers. These describe the "confidence" of the model that the image corresponds to each of the 10 different articles of clothing. We can see which label has the highest confidence value:

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

So the model is most confident that this image is a shirt, or `class_names[6]`. And we can check the test label to see this is correct:

[ ]:
```
test_labels[0]
```

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

[ ]:
```
def plot_image(i, predictions_array, true_labels, images):
  predictions_array, true_label, img = predictions_array[i], true_labels[i],␣
↪images[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img[...,0], 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')
```

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)
```

[ ]:
```
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)
```

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

[ ]:
```
# Plot the first X test images, their predicted label, and the true label
# Color correct predictions in blue, incorrect predictions in red
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)
```

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)
```

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

[ ]:
```
# Add the image to a batch where it's the only member.
img = np.array([img])

print(img.shape)
```

Now predict the image:

[ ]:
```
predictions_single = model.predict(img)

print(predictions_single)
```

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

`model.predict` returns a list of lists, one for each image in the batch of data. Grab the predictions for our (only) image in the batch:

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

And, as before, the model predicts a label of 6 (shirt).

2 Exercises

Experiment with different models and see how the accuracy results differ. In particular change the following parameters: * Set training epochs set to 1 * Number of neurons in the Dense layer following the Flatten one. For example, go really low (e.g. 10) in ranges up to 512 and see how accuracy changes * Add additional Dense layers between the Flatten and the final `Dense(10)`, experiment with different units in these layers * Don't normalize the pixel values, and see the effect that has

Remember to enable GPU to make everything run faster (Runtime -> Change runtime type -> Hardware accelerator -> GPU). Also, if you run into trouble, simply reset the entire environment and start from the beginning: * Edit -> Clear all outputs * Runtime -> Reset all runtimes