In [ ]:
```
#!pip install tensorflow==2.0.0
```

In [ ]:
```
import tensorflow as tf
print(tf.__version__)
```

```
2.3.0
```

Part 1: TensorFlow basics

Tensors

This is a constant tensor:

In [ ]:
```
x = tf.constant([[5, 2], [1, 3]])
print(x)
```

```
tf.Tensor(
[[5 2]
 [1 3]], shape=(2, 2), dtype=int32)
```

You can get its value as a Numpy array by calling `.numpy()`:

In [ ]:
```
x.numpy()
```

Out[ ]:
```
array([[5, 2],
       [1, 3]], dtype=int32)
```

Much like a Numpy array, it features the attributes `dtype` and `shape`:

In [ ]:
```
print('dtype:', x.dtype)
print('shape:', x.shape)
```

```
dtype: <dtype: 'int32'>
shape: (2, 2)
```

A common way to create constant tensors is via `tf.ones` and `tf.zeros` (just like `np.ones` and `np.zeros`):

In [ ]:
```
print(tf.ones(shape=(2, 1)))
print(tf.zeros(shape=(2, 1)))
```

```
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

Random constant tensors

This is all pretty normal:

In [ ]:
```
tf.random.normal(shape=(2, 2), mean=0., stddev=1.)
```

Out[ ]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[-0.5293256 , -1.0482206 ],
       [ 0.61942387, -0.6819246 ]], dtype=float32)>
```

And here's an integer tensor with values drawn from a random uniform distribution:

In [ ]:
```
tf.random.uniform(shape=(2, 2), minval=0, maxval=10, dtype='int32')
```

Out[ ]:
```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[4, 2],
       [9, 7]], dtype=int32)>
```

Variables

Variables are special tensors used to store mutable state (like the weights of a neural network). You create a Variable using some initial value.

In [ ]:
```
initial_value = tf.random.normal(shape=(2, 2))
a = tf.Variable(initial_value)
print(a)
```

```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[ 0.36464038,  1.8475662 ],
       [-0.5905565 , -0.16027513]], dtype=float32)>
```

You update the value of a Variable by using the methods `.assign(value)`, or `.assign_add(increment)` or `.assign_sub(decrement)`:

In [ ]:
```
new_value = tf.random.normal(shape=(2, 2))
a.assign(new_value)
for i in range(2):
  for j in range(2):
    assert a[i, j] == new_value[i, j]
```

In [ ]:
```
added_value = tf.random.normal(shape=(2, 2))
a.assign_add(added_value)
for i in range(2):
  for j in range(2):
    assert a[i, j] == new_value[i, j] + added_value[i, j]
```

Doing math in TensorFlow

You can use TensorFlow exactly like you would use Numpy. The main difference is that your TensorFlow code can run on GPU and TPU.

In [ ]:
```
a = tf.random.normal(shape=(2, 2))
b = tf.random.normal(shape=(2, 2))

c = a + b
d = tf.square(c)
e = tf.exp(d)
```

Computing gradients with `GradientTape`

Oh, and there's another big difference with Numpy: you can automatically retrieve the gradient of any differentiable expression.

Just open a `GradientTape`, start "watching" a tensor via `tape.watch()`, and compose a differentiable expression using this tensor as input:

In [ ]:
```
a = tf.random.normal(shape=(2, 2))
b = tf.random.normal(shape=(2, 2))

with tf.GradientTape() as tape:
  tape.watch(a)  # Start recording the history of operations applied to `a`
  c = tf.sqrt(tf.square(a) + tf.square(b))  # Do some math using `a`
  # What's the gradient of `c` with respect to `a`?
  dc_da = tape.gradient(c, a)
  print(dc_da)
```

```
tf.Tensor(
[[ 0.80471563 -0.98697984]
 [-0.7596037   0.15738489]], shape=(2, 2), dtype=float32)
```

By default, variables are watched automatically, so you don't need to manually `watch` them:

In [ ]:
```
a = tf.Variable(a)

with tf.GradientTape() as tape:
  c = tf.sqrt(tf.square(a) + tf.square(b))
  dc_da = tape.gradient(c, a)
  print(dc_da)
```

```
tf.Tensor(
[[ 0.80471563 -0.98697984]
 [-0.7596037   0.15738489]], shape=(2, 2), dtype=float32)
```

Note that you can compute higher-order derivatives by nesting tapes:

In [ ]:
```
with tf.GradientTape() as outer_tape:
  with tf.GradientTape() as tape:
    c = tf.sqrt(tf.square(a) + tf.square(b))
    dc_da = tape.gradient(c, a)
  d2c_da2 = outer_tape.gradient(dc_da, a)
  print(d2c_da2)
```

```
tf.Tensor(
[[0.176198   0.02943563]
 [0.3462551  1.219587  ]], shape=(2, 2), dtype=float32)
```

An end-to-end example: linear regression

So far you've learned that TensorFlow is a Numpy-like library that is GPU or TPU accelerated, with automatic differentiation. Time for an end-to-end example: let's implement a linear regression, the FizzBuzz of Machine Learning.

For the sake of demonstration, we won't use any of the higher-level Keras components like `Layer` or `MeanSquaredError`. Just basic ops.

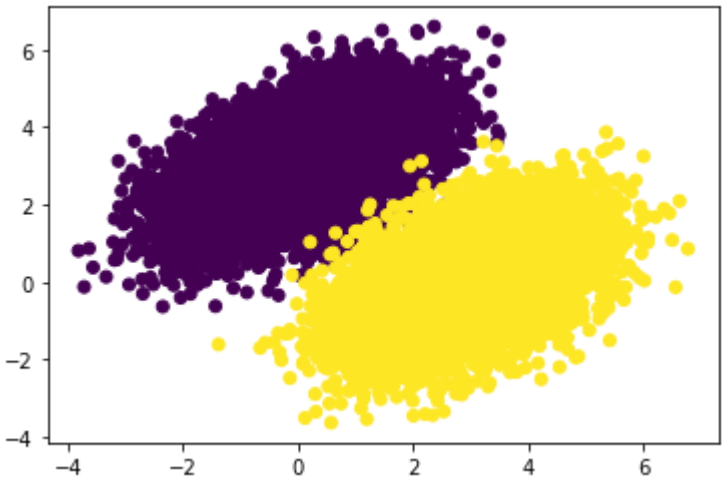
In [ ]:
```
input_dim = 2
output_dim = 1
learning_rate = 0.01

# This is our weight matrix
w = tf.Variable(tf.random.uniform(shape=(input_dim, output_dim)))
# This is our bias vector
b = tf.Variable(tf.zeros(shape=(output_dim,)))

def compute_predictions(features):
  return tf.matmul(features, w) + b

def compute_loss(labels, predictions):
  return tf.reduce_mean(tf.square(labels - predictions))

def train_on_batch(x, y):
  with tf.GradientTape() as tape:
    predictions = compute_predictions(x)
    loss = compute_loss(y, predictions)
    dloss_dw, dloss_db = tape.gradient(loss, [w, b])
  w.assign_sub(learning_rate * dloss_dw)
  b.assign_sub(learning_rate * dloss_db)
  return loss
```

Let's generate some artificial data to demonstrate our model:

In [ ]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
%matplotlib inline

# Prepare a dataset.
num_samples = 10000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3], cov=[[1, 0.5],[0.5, 1]], size=num_samples)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0], cov=[[1, 0.5],[0.5, 1]], size=num_samples)
features = np.vstack((negative_samples, positive_samples)).astype(np.float32
labels = np.vstack((np.zeros((num_samples, 1), dtype='float32'),
                    s((num_samples, 1), dtype='float32')))
```

```
plt.scatter(features[:, 0], features[:, 1], c=labels[:, 0])
```

Out[ ]: <matplotlib.collections.PathCollection at 0x7f29d00f5908>



Now let's train our linear regression by iterating over batch-by-batch over the data and repeatedly calling `train_on_batch`:

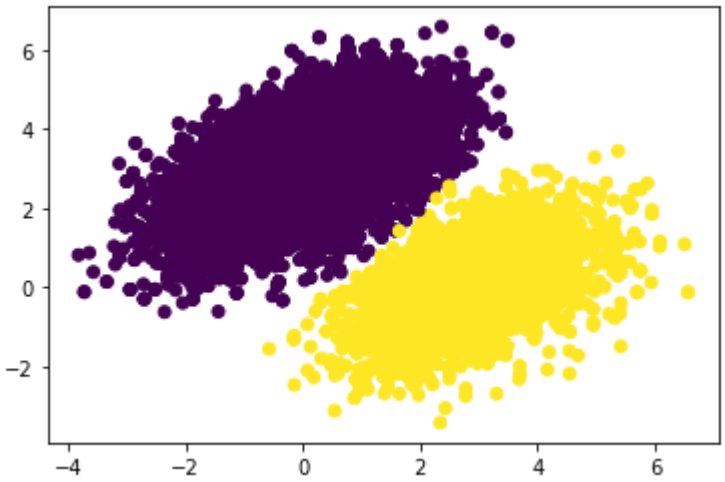
In [ ]:
```
# Shuffle the data.
random.Random(1337).shuffle(features)
random.Random(1337).shuffle(labels)

# Create a tf.data.Dataset object for easy batched iteration
dataset = tf.data.Dataset.from_tensor_slices((features, labels))
dataset = dataset.shuffle(buffer_size=1024).batch(256)

for epoch in range(10):
  for step, (x, y) in enumerate(dataset):
    loss = train_on_batch(x, y)
  print('Epoch %d: last batch loss = %.4f' % (epoch, float(loss)))
```

```
Epoch 0: last batch loss = 0.1011
Epoch 1: last batch loss = 0.0772
Epoch 2: last batch loss = 0.0601
Epoch 3: last batch loss = 0.0427
Epoch 4: last batch loss = 0.0352
Epoch 5: last batch loss = 0.0388
Epoch 6: last batch loss = 0.0369
Epoch 7: last batch loss = 0.0300
Epoch 8: last batch loss = 0.0234
Epoch 9: last batch loss = 0.0266
```

Here's how our model performs:

In [ ]:
```
predictions = compute_predictions(features)
plt.scatter(features[:, 0], features[:, 1], c=predictions[:, 0] > 0.5)
```

Out[ ]: <matplotlib.collections.PathCollection at 0x7f298803fe80>



Making it fast with `tf.function`

But how fast is our current code running?

In [ ]:
```
import time

t0 = time.time()
for epoch in range(20):
  for step, (x, y) in enumerate(dataset):
    loss = train_on_batch(x, y)
t_end = time.time() - t0
print('Time per epoch: %.3f s' % (t_end / 20,))
```

```
Time per epoch: 0.149 s
```

Let's compile the training function into a static graph. Literally all we need to do is add the `tf.function` decorator on it:

In [ ]:
```
@tf.function
def train_on_batch(x, y):
  with tf.GradientTape() as tape:
    predictions = compute_predictions(x)
    loss = compute_loss(y, predictions)
    dloss_dw, dloss_db = tape.gradient(loss, [w, b])
  w.assign_sub(learning_rate * dloss_dw)
  b.assign_sub(learning_rate * dloss_db)
  return loss
```

Let's try this again:

In [ ]:
```
t0 = time.time()
for epoch in range(20):
  for step, (x, y) in enumerate(dataset):
    loss = train_on_batch(x, y)
t_end = time.time() - t0
print('Time per epoch: %.3f s' % (t_end / 20,))
```

```
Time per epoch: 0.084 s
```

40% reduction, neat. In this case we used a trivially simple model; in general the bigger the model the greater the speedup you can get by leveraging static graphs.

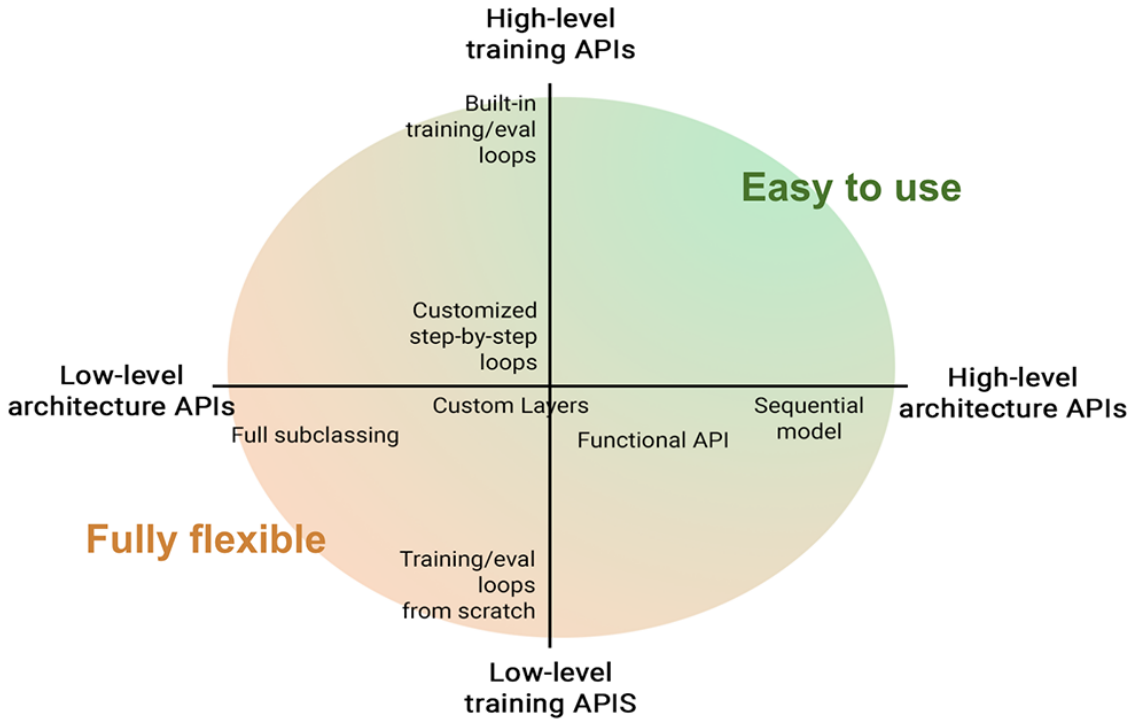
Remember: eager execution is great for debugging and printing results line-by-line, but when it's time to scale, static graphs are a researcher's best friends.

Part 2: The Keras API

Keras is a Python API for deep learning. It has something for everyone:

- If you're an engineer, Keras provides you with reusable blocks such as layers, metrics, training loops, to support common use cases. It provides a high-level user experience that's accessible and productive.
- If you're a researcher, you may prefer not to use these built-in blocks such as layers and training loops, and instead create your own. Of course, Keras allows you to do this. In this case, Keras provides you with templates for the blocks you write, it provides you with structure, with an API standard for things like Layers and Metrics. This structure makes your code easy to share with others and easy to integrate in production workflows.
- The same is true for library developers: TensorFlow is a large ecosystem. It has many different libraries. In order for different libraries to be able to talk to each other and share components, they need to follow an API standard. That's what Keras provides.

Crucially, Keras brings high-level UX and low-level flexibility together fluently: you no longer have on one hand, a high-level API that's easy to use but inflexible, and on the other hand a low-level API that's flexible but only approachable by experts. Instead, you have a spectrum of workflows, from the very high-level to the very low-level. Workflows that are all compatible because they're built on top of the same concepts and objects.



The base `Layer` class

The first class you need to know is `Layer`. Pretty much everything in Keras derives from it.

A Layer encapsulates a state (weights) and some computation (defined in the `call` method).

In [ ]:
```
from tensorflow.keras.layers import Layer

class Linear(Layer):
  """y = w.x + b"""

  def __init__(self, units=32, input_dim=32):
      super(Linear, self).__init__()
      w_init = tf.random_normal_initializer()
      self.w = tf.Variable(
          initial_value=w_init(shape=(input_dim, units), dtype='float32'),
          trainable=True)
      b_init = tf.zeros_initializer()
      self.b = tf.Variable(
          initial_value=b_init(shape=(units,), dtype='float32'),
          trainable=True)

  def call(self, inputs):
      return tf.matmul(inputs, self.w) + self.b

# Instantiate our layer.
linear_layer = Linear(4, 2)
```

A layer instance works like a function. Let's call it on some data:

In [ ]:
```
y = linear_layer(tf.ones((2, 2)))
assert y.shape == (2, 4)
```

The `Layer` class takes care of tracking the weights assigned to it as attributes:

In [ ]:
```
# Weights are automatically tracked under the `weights` property.
assert linear_layer.weights == [linear_layer.w, linear_layer.b]
```

Note that's also a shortcut method for creating weights: `add_weight`. Instead of doing

```
w_init = tf.random_normal_initializer()
self.w = tf.Variable(initial_value=w_init(shape=shape,
dtype='float32'))
```

You would typically do:

```
self.w = self.add_weight(shape=shape, initializer='random_normal')
```

It’s good practice to create weights in a separate `build` method, called lazily with the shape of the first inputs seen by your layer. Here, this pattern prevents us from having to specify input_dim in the constructor:

In [ ]:
```
class Linear(Layer):
  """y = w.x + b"""

  def __init__(self, units=32):
      super(Linear, self).__init__()
      self.units = units
```

```
  def build(self, input_shape):
      self.w = self.add_weight(shape=(input_shape[-1], self.units),
                               initializer='random_normal',
                               trainable=True)
      self.b = self.add_weight(shape=(self.units,),
                               initializer='random_normal',
                               trainable=True)

  def call(self, inputs):
      return tf.matmul(inputs, self.w) + self.b


# Instantiate our lazy layer.
linear_layer = Linear(4)

# This will also call `build(input_shape)` and create the weights.
y = linear_layer(tf.ones((2, 2)))
assert len(linear_layer.weights) == 2
```

Trainable and non-trainable weights

Weights created by layers can be either trainable or non-trainable. They're exposed in `trainable_weights` and `non_trainable_weights`. Here's a layer with a non-trainable weight:

In [ ]:
```
class ComputeSum(Layer):
  """Returns the sum of the inputs."""

  def __init__(self, input_dim):
      super(ComputeSum, self).__init__()
      # Create a non-trainable weight.
      self.total = tf.Variable(initial_value=tf.zeros((input_dim,)),
                               trainable=False)

  def call(self, inputs):
      self.total.assign_add(tf.reduce_sum(inputs, axis=0))
      return self.total

my_sum = ComputeSum(2)
x = tf.ones((2, 2))

y = my_sum(x)
print(y.numpy())  # [2. 2.]

y = my_sum(x)
print(y.numpy())  # [4. 4.]

assert my_sum.weights == [my_sum.total]
assert my_sum.non_trainable_weights == [my_sum.total]
assert my_sum.trainable_weights == []
```

```
[2. 2.]
[4. 4.]
```

Recursively composing layers

Layers can be recursively nested to create bigger computation blocks. Each layer will track the weights of its sublayers (both trainable and non-trainable.

In [ ]:
```python
# Let's reuse the Linear class
# with a `build` method that we defined above.

class MLP(Layer):
    """Simple stack of Linear layers."""

    def __init__(self):
        super(MLP, self).__init__()
        self.linear_1 = Linear(32)
        self.linear_2 = Linear(32)
        self.linear_3 = Linear(10)

    def call(self, inputs):
        x = self.linear_1(inputs)
        x = tf.nn.relu(x)
        x = self.linear_2(x)
        x = tf.nn.relu(x)
        return self.linear_3(x)

mlp = MLP()

# The first call to the `mlp` object will create the weights.
y = mlp(tf.ones(shape=(3, 64)))

# Weights are recursively tracked.
assert len(mlp.weights) == 6
```

Built-in layers

Keras provides you with a wide range of built-in layers, so that you don't have to implement your own layers all the time.

- Convolution layers
- Transposed convolutions
- Separateable convolutions
- Average and max pooling
- Global average and max pooling
- LSTM, GRU (with built-in cuDNN acceleration)
- BatchNormalization
- Dropout
- Attention

- ConvLSTM2D
- etc.

Keras follows the principles of exposing good default configurations, so that layers will work fine out of the box for most use cases if you leave keyword arguments to their default value. For instance, the `LSTM` layer uses an orthogonal recurrent matrix initializer by default, and initializes the forget gate bias to one by default.

The `training` argument in `call`

Some layers, in particular the `BatchNormalization` layer and the `Dropout` layer, have different behaviors during training and inference. For such layers, it is standard practice to expose a `training` (boolean) argument in the `call` method.

By exposing this argument in `call`, you enable the built-in training and evaluation loops (e.g. `fit`) to correctly use the layer in training and inference.

In [ ]:
```
class Dropout(Layer):

  def __init__(self, rate):
    super(Dropout, self).__init__()
    self.rate = rate

  def call(self, inputs, training=None):
    if training:
      return tf.nn.dropout(inputs, rate=self.rate)
    return inputs

class MLPWithDropout(Layer):

  def __init__(self):
      super(MLPWithDropout, self).__init__()
      self.linear_1 = Linear(32)
      self.dropout = Dropout(0.5)
      self.linear_3 = Linear(10)

  def call(self, inputs, training=None):
      x = self.linear_1(inputs)
      x = tf.nn.relu(x)
      x = self.dropout(x, training=training)
      return self.linear_3(x)

mlp = MLPWithDropout()
y_train = mlp(tf.ones((2, 2)), training=True)
y_test = mlp(tf.ones((2, 2)), training=False)
```

A more Functional way of defining models

To build deep learning models, you don't have to use object-oriented programming all the time. Layers can also be composed functionally, like this (we call it the "Functional API"):

In [ ]:
```
# We use an `Input` object to describe the shape and dtype of the inputs.
# This is the deep learning equivalent of *declaring a type*.
# The shape argument is per-sample; it does not include the batch size.
# The functional API focused on defining per-sample transformations.
# The model we create will automatically batch the per-sample transformation
# so that it can be called on batches of data.
inputs = tf.keras.Input(shape=(16,))

# We call layers on these "type" objects
# and they return updated types (new shapes/dtypes).
x = Linear(32)(inputs) # We are reusing the Linear layer we defined earlier.
x = Dropout(0.5)(x) # We are reusing the Dropout layer we defined earlier.
outputs = Linear(10)(x)

# A functional `Model` can be defined by specifying inputs and outputs.
# A model is itself a layer like any other.
model = tf.keras.Model(inputs, outputs)

# A functional model already has weights, before being called on any data.
# That's because we defined its input shape in advance (in `Input`).
assert len(model.weights) == 4

# Let's call our model on some data.
y = model(tf.ones((2, 16)))
assert y.shape == (2, 10)
```

The Functional API tends to be more concise than subclassing, and provides a few other advantages (generally the same advantages that functional, typed languages provide over untyped OO development). However, it can only be used to define DAGs of layers -- recursive networks should be defined as `Layer` subclasses instead.

Key differences between models defined via subclassing and Functional models are explained in this blog post.

Learn more about the Functional API here.

In your research workflows, you may often find yourself mix-and-matching OO models and Functional models.

For models that are simple stacks of layers with a single input and a single output, you can also use the `Sequential` class which turns a list of layers into a `Model`:

In [ ]:
```
from tensorflow.keras import Sequential
```

```
model = Sequential([Linear(32), Dropout(0.5), Linear(10)])

y = model(tf.ones((2, 16)))
assert y.shape == (2, 10)
```

Loss classes

Keras features a wide range of built-in loss classes, like `BinaryCrossentropy`, `CategoricalCrossentropy`, `KLDivergence`, etc. They work like this:

In [ ]:
```
bce = tf.keras.losses.BinaryCrossentropy()
y_true = [0., 0., 1., 1.]  # Targets
y_pred = [1., 1., 1., 0.]  # Predictions
loss = bce(y_true, y_pred)
print('Loss:', loss.numpy())
```

```
Loss: 11.522857
```

Note that loss classes are stateless: the output of `__call__` is only a function of the input.

Metric classes

Keras also features a wide range of built-in metric classes, such as `BinaryAccuracy`, `AUC`, `FalsePositives`, etc.

Unlike losses, metrics are stateful. You update their state using the `update_state` method, and you query the scalar metric result using `result`:

In [ ]:
```
m = tf.keras.metrics.AUC()
m.update_state([0, 1, 1, 1], [0, 1, 0, 0])
print('Intermediate result: ', m.result().numpy())

m.update_state([1, 1, 1, 1], [0, 1, 1, 0])
print('Final result: ', m.result().numpy())
```

```
Intermediate result:  0.6666667
Final result:  0.71428573
```

The internal state can be cleared with `metric.reset_states`.

You can easily roll out your own metrics by subclassing the `Metric` class:

- Create the state variables in `__init__`
- Update the variables given `y_true` and `y_pred` in `update_state`
- Return the metric result in `result`
- Clear the state in `reset_states`

Here's a quick implementation of a `BinaryTruePositives` metric as a demonstration:

In [ ]:
```
class BinaryTruePositives(tf.keras.metrics.Metric):

  def __init__(self, name='binary_true_positives', **kwargs):
    super(BinaryTruePositives, self).__init__(name=name, **kwargs)
    self.true_positives = self.add_weight(name='tp', initializer='zeros')

  def update_state(self, y_true, y_pred, sample_weight=None):
    y_true = tf.cast(y_true, tf.bool)
    y_pred = tf.cast(y_pred, tf.bool)

    values = tf.logical_and(tf.equal(y_true, True), tf.equal(y_pred, True))
    values = tf.cast(values, self.dtype)
    if sample_weight is not None:
      sample_weight = tf.cast(sample_weight, self.dtype)
      sample_weight = tf.broadcast_weights(sample_weight, values)
      values = tf.multiply(values, sample_weight)
    self.true_positives.assign_add(tf.reduce_sum(values))

  def result(self):
    return self.true_positives

  def reset_states(self):
    self.true_positive.assign(0)
```

Optimizer classes & a quick end-to-end training loop

You don't normally have to define by hand how to update your variables during gradient descent, like we did in our initial linear regression example. You would usually use one of the built-in Keras optimizer, like `SGD`, `RMSprop`, or `Adam`.

Here's a simple MNSIT example that brings together loss classes, metric classes, and optimizers.

In [ ]:
```
from tensorflow.keras import layers

# Prepare a dataset.
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train[:].reshape(60000, 784).astype('float32') / 255
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
dataset = dataset.shuffle(buffer_size=1024).batch(64)

# Instantiate a simple classification model
model = tf.keras.Sequential([
  layers.Dense(256, activation=tf.nn.relu),
  layers.Dense(256, activation=tf.nn.relu),
  layers.Dense(10)
])
```

```python
# Instantiate a logistic loss function that expects integer targets.
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Instantiate an accuracy metric.
accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

# Instantiate an optimizer.
optimizer = tf.keras.optimizers.Adam()

# Iterate over the batches of the dataset.
for step, (x, y) in enumerate(dataset):

  # Open a GradientTape.
  with tf.GradientTape() as tape:

    # Forward pass.
    logits = model(x)

    # Loss value for this batch.
    loss_value = loss(y, logits)

  # Get gradients of weights wrt the loss.
  gradients = tape.gradient(loss_value, model.trainable_weights)

  # Update the weights of our linear layer.
  optimizer.apply_gradients(zip(gradients, model.trainable_weights))

  # Update the running accuracy.
  accuracy.update_state(y, logits)

  # Logging.
  if step % 100 == 0:
    print('Step:', step)
    print('Loss from last step:', float(loss_value))
    print('Total running accuracy so far:', float(accuracy.result()))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dat
asets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
Step: 0
Loss from last step: 2.358793258666992
Total running accuracy so far: 0.09375
Step: 100
Loss from last step: 0.21707287430763245
Total running accuracy so far: 0.8310643434524536
Step: 200
Loss from last step: 0.2818300127983093
Total running accuracy so far: 0.8765547275543213
Step: 300
Loss from last step: 0.23447920382022858
Total running accuracy so far: 0.8955564498901367
Step: 400
Loss from last step: 0.11367885768413544
Total running accuracy so far: 0.9080813527107239
Step: 500
Loss from last step: 0.11368697881698608
Total running accuracy so far: 0.9158245921134949
Step: 600
Loss from last step: 0.0994415432214737
Total running accuracy so far: 0.9220309853553772
Step: 700
Loss from last step: 0.047019436955451965
Total running accuracy so far: 0.9272022247314453
Step: 800
Loss from last step: 0.07821480929851532
Total running accuracy so far: 0.9307116270065308
Step: 900
Loss from last step: 0.09753896296024323
Total running accuracy so far: 0.9345518946647644
```

We can reuse our `SparseCategoricalAccuracy` metric instance to implement a testing loop:

In [ ]:
```
x_test = x_test[:].reshape(10000, 784).astype('float32') / 255
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.batch(128)

accuracy.reset_states()  # This clears the internal state of the metric

for step, (x, y) in enumerate(test_dataset):
  logits = model(x)
  accuracy.update_state(y, logits)

print('Final test accuracy:', float(accuracy.result()))
```

```
Final test accuracy: 0.9607999920845032
```

The `add_loss` method

Sometimes you need to compute loss values on the fly during a foward pass (especially regularization losses). Keras allows you to compute loss values at any time, and to recursively keep track of them via the `add_loss` method.

Here's an example of a layer that adds a sparsity regularization loss based on the L2 norm of the inputs:

In [ ]:
```
class ActivityRegularization(Layer):
  """Layer that creates an activity sparsity regularization loss."""

  def __init__(self, rate=1e-2):
    super(ActivityRegularization, self).__init__()
    self.rate = rate

  def call(self, inputs):
    # We use `add_loss` to create a regularization loss
    # that depends on the inputs.
    self.add_loss(self.rate * tf.reduce_sum(tf.square(inputs)))
    return inputs
```

Loss values added via `add_loss` can be retrieved in the `.losses` list property of any `Layer` or `Model`:

In [ ]:
```
from tensorflow.keras import layers

class SparseMLP(Layer):
  """Stack of Linear layers with a sparsity regularization loss."""

  def __init__(self, output_dim):
      super(SparseMLP, self).__init__()
      self.dense_1 = layers.Dense(32, activation=tf.nn.relu)
      self.regularization = ActivityRegularization(1e-2)
      self.dense_2 = layers.Dense(output_dim)

  def call(self, inputs):
      x = self.dense_1(inputs)
      x = self.regularization(x)
      return self.dense_2(x)


mlp = SparseMLP(1)
y = mlp(tf.ones((10, 10)))

print(mlp.losses)  # List containing one float32 scalar
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=0.6885692>]
```

These losses are cleared by the top-level layer at the start of each forward pass -- they don't accumulate. So `layer.losses` always contain only the losses created during the last forward pass. You would typically use these losses by summing them before computing your gradients when writing a training loop.

In [ ]:
```python
# Losses correspond to the *last* forward pass.
mlp = SparseMLP(1)
mlp(tf.ones((10, 10)))
assert len(mlp.losses) == 1
mlp(tf.ones((10, 10)))
assert len(mlp.losses) == 1  # No accumulation.

# Let's demonstrate how to use these losses in a training loop.

# Prepare a dataset.
(x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
dataset = tf.data.Dataset.from_tensor_slices(
    (x_train.reshape(60000, 784).astype('float32') / 255, y_train))
dataset = dataset.shuffle(buffer_size=1024).batch(64)

# A new MLP.
mlp = SparseMLP(10)

# Loss and optimizer.
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1)

for step, (x, y) in enumerate(dataset):
  with tf.GradientTape() as tape:

    # Forward pass.
    logits = mlp(x)

    # External loss value for this batch.
    loss = loss_fn(y, logits)

    # Add the losses created during the forward pass.
    loss += sum(mlp.losses)

    # Get gradients of weights wrt the loss.
    gradients = tape.gradient(loss, mlp.trainable_weights)

  # Update the weights of our linear layer.
  optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))

  # Logging.
  if step % 100 == 0:
    print(step, float(loss))
```

```
0 4.304479598999023
100 2.2919511795043945
200 2.2856969833374023
300 2.249835968017578
400 2.154803514480591
500 2.179860830307007
600 2.0276057720184326
700 2.064443349838257
800 2.0333380699157715
900 1.828589916229248
```

A detailed end-to-end example: a Variational AutoEncoder (VAE)

If you want to take a break from the basics and look at a slightly more advanced example, check out this Variational AutoEncoder implementation that demonstrates everything you've learned so far:

- Subclassing `Layer`
- Recursive layer composition
- Loss classes and metric classes
- `add_loss`
- `GradientTape`

Using built-in training loops

It would be a bit silly if you had to write your own low-level training loops every time for simple use cases. Keras provides you with a built-in training loop on the `Model` class. If you want to use it, either subclass from `Model` or create a `Functional` or `Sequential` model.

To demonstrate it, let's reuse the MNIST setup from above:

In [ ]:
```
# Prepare a dataset.
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
dataset = dataset.shuffle(buffer_size=1024).batch(64)

# Instantiate a simple classification model
model = tf.keras.Sequential([
  layers.Dense(256, activation=tf.nn.relu),
  layers.Dense(256, activation=tf.nn.relu),
  layers.Dense(10)
])

# Instantiate a logistic loss function that expects integer targets.
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Instantiate an accuracy metric.
accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

# Instantiate an optimizer.
optimizer = tf.keras.optimizers.Adam()
```

First, call `compile` to configure the optimizer, loss, and metrics to monitor.

In [ ]:
```
model.compile(optimizer=optimizer, loss=loss, metrics=[accuracy])
```

Then we call `fit` on our model to pass it the data:

In [ ]:
```
model.fit(dataset, epochs=3)
```

```
Epoch 1/3
938/938 [==============================] - 2s 2ms/step - loss: 0.2215 - spar
se_categorical_accuracy: 0.9352
Epoch 2/3
938/938 [==============================] - 2s 2ms/step - loss: 0.0874 - spar
se_categorical_accuracy: 0.9737
Epoch 3/3
938/938 [==============================] - 2s 2ms/step - loss: 0.0582 - spar
se_categorical_accuracy: 0.9817
```

Out[ ]:
```
<tensorflow.python.keras.callbacks.History at 0x7f29713263c8>
```

Done! Now let's test it:

In [ ]:
```
x_test = x_test[:].reshape(10000, 784).astype('float32') / 255
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.batch(128)

loss, acc = model.evaluate(test_dataset)
print('loss:', loss, 'acc:', acc)
```

```
79/79 [==============================] - 0s 2ms/step - loss: 0.0848 - sparse
_categorical_accuracy: 0.9747
loss: 0.08479571342468262 acc: 0.9746999740600586
```

Note that you can also monitor your loss and metrics on some validation data during `fit`.

Also, you can call `fit` directly on Numpy arrays, so no need for the dataset conversion:

In [ ]:
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255

num_val_samples = 10000
x_val = x_train[-num_val_samples:]
y_val = y_train[-num_val_samples:]
x_train = x_train[:-num_val_samples]
y_train = y_train[:-num_val_samples]

# Instantiate a simple classification model
model = tf.keras.Sequential([
  layers.Dense(256, activation=tf.nn.relu),
  layers.Dense(256, activation=tf.nn.relu),
  layers.Dense(10)
])

# Instantiate a logistic loss function that expects integer targets.
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Instantiate an accuracy metric.
```

```
accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

# Instantiate an optimizer.
optimizer = tf.keras.optimizers.Adam()

model.compile(optimizer=optimizer,
              loss=loss,
              metrics=[accuracy])
model.fit(x_train, y_train,
          validation_data=(x_val, y_val),
          epochs=3,
          batch_size=64)
```

```
Epoch 1/3
782/782 [==============================] - 2s 2ms/step - loss: 0.2464 - spar
se_categorical_accuracy: 0.9276 - val_loss: 0.1197 - val_sparse_categorical_
accuracy: 0.9645
Epoch 2/3
782/782 [==============================] - 2s 2ms/step - loss: 0.0940 - spar
se_categorical_accuracy: 0.9714 - val_loss: 0.0982 - val_sparse_categorical_
accuracy: 0.9723
Epoch 3/3
782/782 [==============================] - 2s 2ms/step - loss: 0.0605 - spar
se_categorical_accuracy: 0.9810 - val_loss: 0.0774 - val_sparse_categorical_
accuracy: 0.9763
```

Out[ ]: `<tensorflow.python.keras.callbacks.History at 0x7f2970435860>`

Callbacks

One of the neat features of `fit` (besides built-in support for sample weighting and class weighting) is that you can easily customize what happens during training and evaluation by using callbacks.

A callback is an object that is called at different points during training (e.g. at the end of every batch or at the end of every epoch) and does stuff.

There's a bunch of built-in callback available, like `ModelCheckpoint` to save your models after each epoch during training, or `EarlyStopping`, which interrupts training when your validation metrics start stalling.

And you can easily write your own callbacks.

In [ ]:
```
# Instantiate a simple classification model
model = tf.keras.Sequential([
  layers.Dense(256, activation=tf.nn.relu),
  layers.Dense(256, activation=tf.nn.relu),
  layers.Dense(10)
])

# Instantiate a logistic loss function that expects integer targets.
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

```
# Instantiate an accuracy metric.
accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

# Instantiate an optimizer.
optimizer = tf.keras.optimizers.Adam()

model.compile(optimizer=optimizer,
              loss=loss,
              metrics=[accuracy])

# Instantiate some callbacks
callbacks = [tf.keras.callbacks.EarlyStopping(),
             tf.keras.callbacks.ModelCheckpoint(filepath='my_model.keras',
                                                save_best_only=True)]

model.fit(x_train, y_train,
          validation_data=(x_val, y_val),
          epochs=30,
          batch_size=64,
          callbacks=callbacks)
```

```
Epoch 1/30
782/782 [==============================] - 2s 2ms/step - loss: 0.2457 - spar
se_categorical_accuracy: 0.9287 - val_loss: 0.1373 - val_sparse_categorical_
accuracy: 0.9582
Epoch 2/30
782/782 [==============================] - 2s 2ms/step - loss: 0.0949 - spar
se_categorical_accuracy: 0.9706 - val_loss: 0.0981 - val_sparse_categorical_
accuracy: 0.9711
Epoch 3/30
782/782 [==============================] - 2s 2ms/step - loss: 0.0610 - spar
se_categorical_accuracy: 0.9811 - val_loss: 0.0944 - val_sparse_categorical_
accuracy: 0.9717
Epoch 4/30
782/782 [==============================] - 2s 2ms/step - loss: 0.0455 - spar
se_categorical_accuracy: 0.9851 - val_loss: 0.0802 - val_sparse_categorical_
accuracy: 0.9768
Epoch 5/30
782/782 [==============================] - 2s 2ms/step - loss: 0.0334 - spar
se_categorical_accuracy: 0.9893 - val_loss: 0.0800 - val_sparse_categorical_
accuracy: 0.9780
Epoch 6/30
782/782 [==============================] - 2s 2ms/step - loss: 0.0266 - spar
se_categorical_accuracy: 0.9912 - val_loss: 0.0993 - val_sparse_categorical_
accuracy: 0.9755
```

Out[ ]: <tensorflow.python.keras.callbacks.History at 0x7f296e2fee10>

Parting words

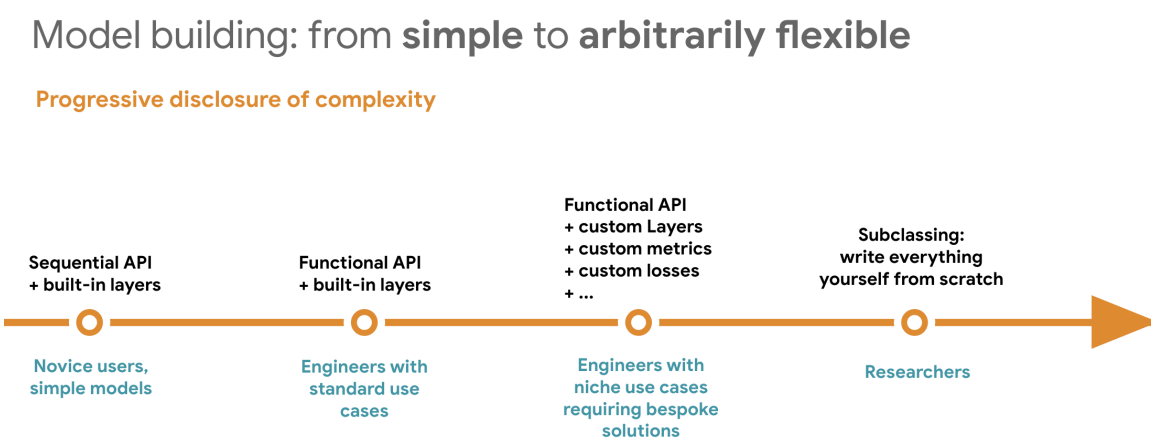
I hope this guide has given you a good overview of what's possible with TensorFlow 2.0 and Keras!

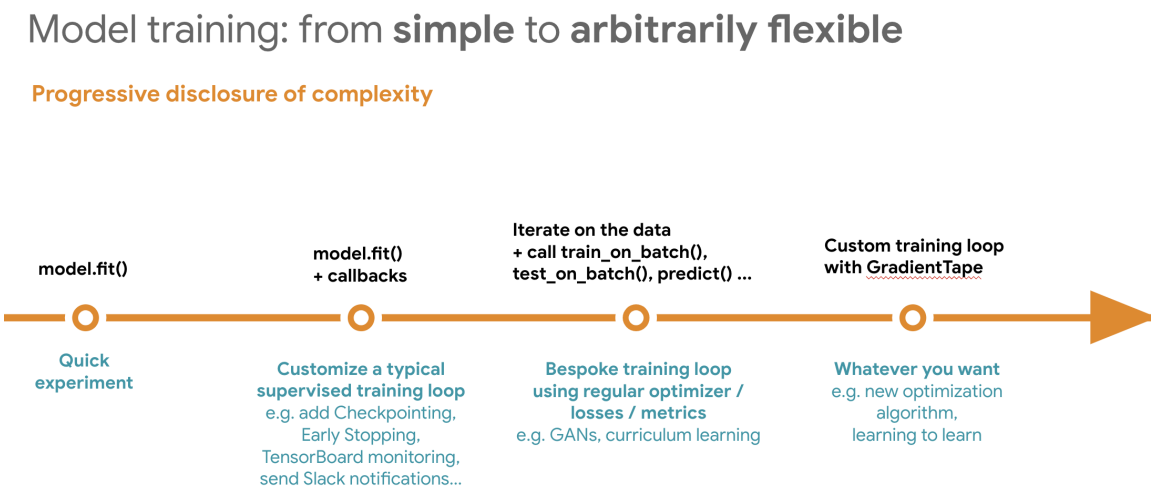
Remember that TensorFlow and Keras don't represent a single workflow. It's a spectrum of workflows, each with its own trade-off between usability and flexibility. For instance, you've noticed that it's much easier to use `fit` than to write a custom training loop, but `fit` doesn't give you the same level of granular control for research use cases.

So use the right tool for the job!

A core principle of Keras is "progressive disclosure of complexity": it's easy to get started, and you can gradually dive into workflows where you write more and more logic from scratch, providing you with complete control.

This applies to both model definition, and model training.





What to learn next

Next, there are many more topics you may be interested in:

- Saving and serialization
- Distributed training on multiple GPUS
- Exporting models to TFLite for deployment on Android or embedded systems
- Exporting models to TensorFlow.js for deployment in the browser