Basic Classification in TensorFlow 2.0

Learning Objectives

1. Build a model
2. Train this model on example data
3. Use the model to make predictions about unknown data

Introduction

In this notebook, you use machine learning to *categorize* Iris flowers by species. It uses TensorFlow to:

- Use TensorFlow's default eager execution development environment
- Import data with the Datasets API
- Build models and layers with TensorFlow's Keras API

Here firstly we will Import and parse the dataset, then select the type of model. After that Train the model.

At last we will Evaluate the model's effectiveness and then use the trained model to make predictions.

Each learning objective will correspond to a **#TODO** in the student lab notebook -- try to complete that notebook first before reviewing this solution notebook.

Configure imports

Import TensorFlow and the other required Python modules. By default, TensorFlow uses eager execution to evaluate operations immediately, returning concrete values instead of creating a computational graph that is executed later. If you are used to a REPL or the `python` interactive console, this feels familiar.

In [3]:
```
# The OS module in python provides functions for interacting with the operating system
import os
# The matplotlib module provides all the fuctionalities for visualizing model
import matplotlib.pyplot as plt
```

In [4]:
```
# Here we'll import data processing libraries like tensorflow
import tensorflow as tf
```

In [5]:
```
# Here we'll show the currently installed version of TensorFlow
print("TensorFlow version: {}".format(tf.__version__))
# Here we'll show the current status of Eager execution
print("Eager execution: {}".format(tf.executing_eagerly()))
```

```
TensorFlow version: 2.3.0
Eager execution: True
```

The Iris classification problem

Imagine you are a botanist seeking an automated way to categorize each Iris flower you find. Machine learning provides many algorithms to classify flowers statistically. For instance, a sophisticated machine learning program could classify flowers based on photographs. Our ambitions are more modest—we're going to classify Iris flowers based on the length and width measurements of their sepals and petals.

The Iris genus entails about 300 species, but our program will only classify the following three:

- Iris setosa
- Iris virginica
- Iris versicolor

Figure 1. Iris setosa (by Radomil, CC BY-SA 3.0), Iris versicolor, (by Dlanglois, CC BY-SA 3.0), and Iris virginica (by Frank Mayfield, CC BY-SA 2.0).

Fortunately, someone has already created a dataset of 120 Iris flowers with the sepal and petal measurements. This is a classic dataset that is popular for beginner machine learning classification problems.

Import and parse the training dataset

Download the dataset file and convert it into a structure that can be used by this Python program.

Download the dataset

In [6]:

```
train_dataset_url = "https://storage.googleapis.com/download.tensorflow.org/data/iris_t

# Download the training dataset file using the `tf.keras.utils.get_file` function. This
train_dataset_fp = tf.keras.utils.get_file(fname=os.path.basename(train_dataset_url),
                                           origin=train_dataset_url)

print("Local copy of the dataset file: {}".format(train_dataset_fp))
```

```
Local copy of the dataset file: /home/jupyter/.keras/datasets/iris_training.csv
```

Inspect the data

This dataset, `iris_training.csv`, is a plain text file that stores tabular data formatted as comma-separated values (CSV). Use the `head -n5` command to take a peek at the first five entries:

In [7]:
```
# Output the first five rows
!head -n5 {train_dataset_fp}
```

```
120,4,setosa,versicolor,virginica
6.4,2.8,5.6,2.2,2
5.0,2.3,3.3,1.0,1
4.9,2.5,4.5,1.7,2
4.9,3.1,1.5,0.1,0
```

From this view of the dataset, notice the following:

1. The first line is a header containing information about the dataset:
   - There are 120 total examples. Each example has four features and one of three possible label names.
2. Subsequent rows are data records, one example per line, where:
   - The first four fields are features: these are the characteristics of an example. Here, the fields hold float numbers representing flower measurements.
   - The last column is the label: this is the value we want to predict. For this dataset, it's an integer value of 0, 1, or 2 that corresponds to a flower name.

Let's write that out in code:

In [8]:
```
# Column order in CSV file
column_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species'

feature_names = column_names[:-1]
label_name = column_names[-1]

# Let's output the value of `Features` and `Label`
print("Features: {}".format(feature_names))
print("Label: {}".format(label_name))
```

```
Features: ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
Label: species
```

Each label is associated with string name (for example, "setosa"), but machine learning typically relies on numeric values. The label numbers are mapped to a named representation, such as:

- `0` : Iris setosa
- `1` : Iris versicolor
- `2` : Iris virginica

For more information about features and labels, see the ML Terminology section of the Machine Learning Crash Course.

In [9]:
```
class_names = ['Iris setosa', 'Iris versicolor', 'Iris virginica']
```

Create a `tf.data.Dataset`

TensorFlow's Dataset API handles many common cases for loading data into a model. This is a high-level API for reading data and transforming it into a form used for training.

Since the dataset is a CSV-formatted text file, use the `tf.data.experimental.make_csv_dataset` function to parse the data into a suitable format. Since this function generates data for training models, the default behavior is to shuffle the data (`shuffle=True, shuffle_buffer_size=10000`), and repeat the dataset forever (`num_epochs=None`). We also set the batch_size parameter:

In [10]:
```
batch_size = 32

# The `tf.data.experimental.make_csv_dataset()` method reads CSV files into a dataset
train_dataset = tf.data.experimental.make_csv_dataset(
    train_dataset_fp,
    batch_size,
    column_names=column_names,
    label_name=label_name,
    num_epochs=1)
```

The `make_csv_dataset` function returns a `tf.data.Dataset` of `(features, label)` pairs, where `features` is a dictionary: `{'feature_name': value}`

These `Dataset` objects are iterable. Let's look at a batch of features:

In [11]:
```
# The `next()` function returns the next item in an iterator.
features, labels = next(iter(train_dataset))

print(features)
```

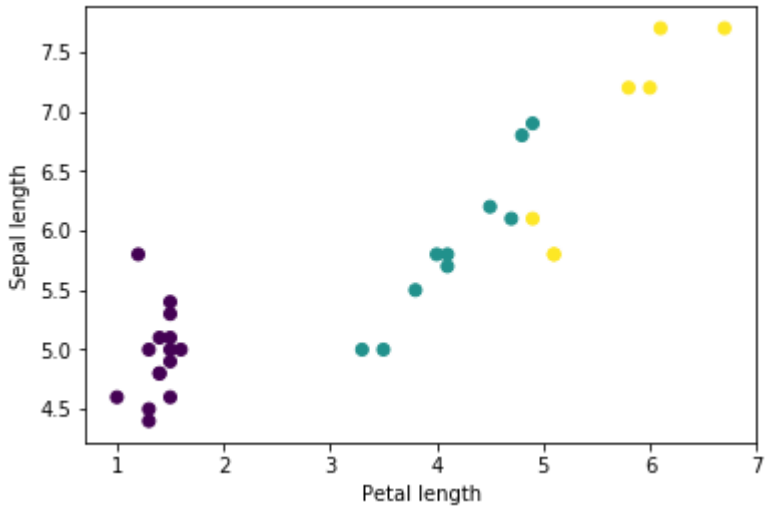
```
OrderedDict([('sepal_length', <tf.Tensor: shape=(32,), dtype=float32, numpy=
array([6.1, 6.8, 5.8, 4.6, 6.1, 4.8, 5.1, 5.3, 6.9, 5.8, 4.4, 5.5, 5.4,
       4.9, 5.1, 4.6, 5. , 4.5, 5. , 5. , 5.8, 5.7, 7.2, 7.7, 7.2, 5.8,
       4.8, 5. , 6.2, 5. , 5.8, 7.7], dtype=float32)>), ('sepal_width', <tf.Tensor: shap
e=(32,), dtype=float32, numpy=
array([3. , 2.8, 4. , 3.1, 2.8, 3. , 3.5, 3.7, 3.1, 2.7, 3. , 2.4, 3.7,
       3.1, 3.8, 3.6, 3.5, 2.3, 2. , 2.3, 2.7, 2.8, 3. , 3. , 3.2, 2.6,
       3. , 3.5, 2.2, 3.4, 2.8, 3.8], dtype=float32)>), ('petal_length', <tf.Tensor: sha
pe=(32,), dtype=float32, numpy=
array([4.9, 4.8, 1.2, 1.5, 4.7, 1.4, 1.4, 1.5, 4.9, 4.1, 1.3, 3.8, 1.5,
       1.5, 1.5, 1. , 1.3, 1.3, 3.5, 3.3, 5.1, 4.1, 5.8, 6.1, 6. , 4. ,
       1.4, 1.6, 4.5, 1.5, 5.1, 6.7], dtype=float32)>), ('petal_width', <tf.Tensor: shap
e=(32,), dtype=float32, numpy=
array([1.8, 1.4, 0.2, 0.2, 1.2, 0.3, 0.3, 0.2, 1.5, 1. , 0.2, 1.1, 0.2,
       0.1, 0.3, 0.2, 0.3, 0.3, 1. , 1. , 1.9, 1.3, 1.6, 2.3, 1.8, 1.2,
       0.1, 0.6, 1.5, 0.2, 2.4, 2.2], dtype=float32)>)])
```

Notice that like-features are grouped together, or *batched*. Each example row's fields are appended to the corresponding feature array. Change the `batch_size` to set the number of examples stored in these feature arrays.

You can start to see some clusters by plotting a few features from the batch:

In [12]:
```
# A scatter plot is a diagram where each value in the data set is represented by a dot.
# A scatter plot of y vs x with varying marker size and/or color.
plt.scatter(features['petal_length'],
            features['sepal_length'],
            c=labels,
            cmap='viridis')

plt.xlabel("Petal length")
plt.ylabel("Sepal length")
# Using show() method we can display a figure.
plt.show()
```



To simplify the model building step, create a function to repackage the features dictionary into a single array with shape: `(batch_size, num_features)` .

This function uses the `tf.stack` method which takes values from a list of tensors and creates a combined tensor at the specified dimension:

In [13]:
```
def pack_features_vector(features, labels):
  """Pack the features into a single array."""
# Using `tf.stack` we can stack a list of rank-R tensors into one rank-(R+1) tensor.
  features = tf.stack(list(features.values()), axis=1)
  return features, labels
```

Then use the `tf.data.Dataset#map` method to pack the `features` of each `(features,label)` pair into the training dataset:

In [14]:
```
# The `map()` method will pack the `features` into the training dataset:
train_dataset = train_dataset.map(pack_features_vector)
```

The features element of the `Dataset` are now arrays with shape `(batch_size, num_features)` . Let's look at the first few examples:

In [15]:
```
# The `next()` function returns the next item in an iterator.
features, labels = next(iter(train_dataset))

print(features[:5])
```

```
tf.Tensor(
[[6.9 3.1 5.1 2.3]
 [7.7 3.8 6.7 2.2]
 [6.4 3.1 5.5 1.8]
 [4.9 3.1 1.5 0.1]
 [6.5 3.2 5.1 2. ]], shape=(5, 4), dtype=float32)
```

Select the type of model

Why model?

A model is a relationship between features and the label. For the Iris classification problem, the model defines the relationship between the sepal and petal measurements and the predicted Iris species. Some simple models can be described with a few lines of algebra, but complex machine learning models have a large number of parameters that are difficult to summarize.

Could you determine the relationship between the four features and the Iris species *without* using machine learning? That is, could you use traditional programming techniques (for example, a lot of conditional statements) to create a model? Perhaps—if you analyzed the dataset long enough to determine the relationships between petal and sepal measurements to a particular species. And this becomes difficult—maybe impossible—on more complicated datasets. A good machine learning approach *determines the model for you*. If you feed enough representative examples into the right machine learning model type, the program will figure out the relationships for you.

Select the model

We need to select the kind of model to train. There are many types of models and picking a good one takes experience. This tutorial uses a neural network to solve the Iris classification problem. Neural networks can find complex relationships between features and the label. It is a highly-structured graph, organized into one or more hidden layers. Each hidden layer consists of one or more neurons. There are several categories of neural networks and this program uses a dense, or fully-connected neural network: the neurons in one layer receive input connections from *every* neuron in the previous layer. For example, Figure 2 illustrates a dense neural network consisting of an input layer, two hidden layers, and an output layer:

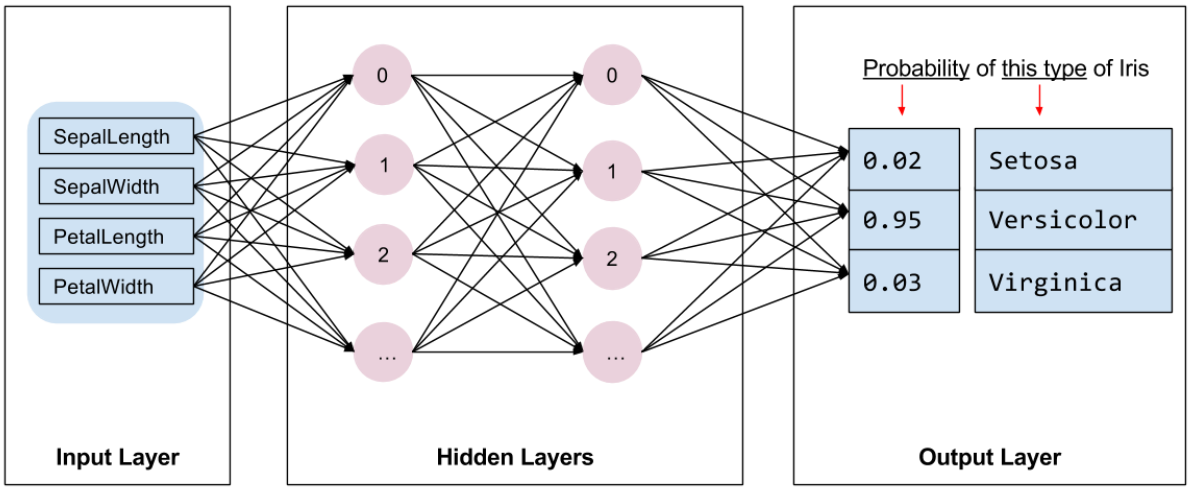


Figure 2. A neural network with features, hidden layers, and predictions.

When the model from Figure 2 is trained and fed an unlabeled example, it yields three predictions: the likelihood that this flower is the given Iris species. This prediction is called inference. For this example, the sum of the output predictions is 1.0. In Figure 2, this prediction breaks down as: `0.02` for *Iris setosa*, `0.95` for *Iris versicolor*, and `0.03` for *Iris virginica*. This means that the model predicts—with 95% probability—that an unlabeled example flower is an *Iris versicolor*.

Create a model using Keras

The TensorFlow `tf.keras` API is the preferred way to create models and layers. This makes it easy to build models and experiment while Keras handles the complexity of connecting everything together.

The `tf.keras.Sequential` model is a linear stack of layers. Its constructor takes a list of layer instances, in this case, two `tf.keras.layers.Dense` layers with 10 nodes each, and an output layer with 3 nodes representing our label predictions. The first layer's `input_shape` parameter corresponds to the number of features from the dataset, and is required:

In [16]:
```
# TODO 1
# Here `tf.keras.Sequential` used to sequentially groups a linear stack of layers into
model = tf.keras.Sequential([
# `tf.keras.layers.Dense` is inherited from: `Layer`
# `tf.keras.layers.Dense` is your regular densely-connected NN layer.
  tf.keras.layers.Dense(10, activation=tf.nn.relu, input_shape=(4,)),  # input shape re
  tf.keras.layers.Dense(10, activation=tf.nn.relu),
  tf.keras.layers.Dense(3)
])
```

The activation function determines the output shape of each node in the layer. These non-linearities are important—without them the model would be equivalent to a single layer. There are many `tf.keras.activations`, but ReLU is common for hidden layers.

The ideal number of hidden layers and neurons depends on the problem and the dataset. Like many aspects of machine learning, picking the best shape of the neural network requires a mixture of knowledge and experimentation. As a rule of thumb, increasing the number of hidden layers and neurons typically creates a more powerful model, which requires more data to train effectively.

Using the model

Let's have a quick look at what this model does to a batch of features:

In [17]:
```
predictions = model(features)
predictions[:5]
```

Out[17]:
```
<tf.Tensor: shape=(5, 3), dtype=float32, numpy=
array([[-0.09421927, -0.00249141,  0.17042825],
       [-0.07480374,  0.2615329 , -0.05820906],
       [-0.06220143,  0.21547055, -0.04997942],
```

```
       [ 0.16941598, -0.3952291 ,  0.32507014],
       [-0.07085979,  0.06307554,  0.07554132]], dtype=float32)>
```

Here, each example returns a logit for each class.

To convert these logits to a probability for each class, use the softmax function:

In [18]:
```
# `tf.nn.softmax()` will compute softmax activations.
tf.nn.softmax(predictions[:5])
```

Out[18]:
```
<tf.Tensor: shape=(5, 3), dtype=float32, numpy=
array([[0.29420087, 0.32246372, 0.38333538],
       [0.2926935 , 0.40971535, 0.29759118],
       [0.30008852, 0.3961328 , 0.3037787 ],
       [0.36536568, 0.20773302, 0.42690134],
       [0.3029404 , 0.34635746, 0.3507021 ]], dtype=float32)>
```

Taking the `tf.argmax` across classes gives us the predicted class index. But, the model hasn't been trained yet, so these aren't good predictions:

In [19]:
```
# `tf.argmax()` will returns the index with the largest value across axes of a tensor.
print("Prediction: {}".format(tf.argmax(predictions, axis=1)))
print("Labels: {}".format(labels))
```

```
Prediction: [2 1 1 2 2 2 1 2 2 1 2 1 2 1 2 1 1 2 1 2 2 1 2 2 1 1 2 1 2 1 1 1]
Labels: [2 2 2 0 2 0 1 1 0 2 0 2 0 2 0 2 1 0 2 0 0 1 0 0 2 1 0 1 0 2 2 2]
```

Train the model

Training is the stage of machine learning when the model is gradually optimized, or the model *learns* the dataset. The goal is to learn enough about the structure of the training dataset to make predictions about unseen data. If you learn *too much* about the training dataset, then the predictions only work for the data it has seen and will not be generalizable. This problem is called overfitting—it's like memorizing the answers instead of understanding how to solve a problem.

The Iris classification problem is an example of supervised machine learning: the model is trained from examples that contain labels. In unsupervised machine learning, the examples don't contain labels. Instead, the model typically finds patterns among the features.

Define the loss and gradient function

Both training and evaluation stages need to calculate the model's loss. This measures how off a model's predictions are from the desired label, in other words, how bad the model is performing. We want to minimize, or optimize, this value.

Our model will calculate its loss using the `tf.keras.losses.SparseCategoricalCrossentropy` function which takes the model's class probability predictions and the desired label, and returns the average loss across the examples.

In [20]:
```
# `tf.keras.losses.SparseCategoricalCrossentropy()` will computes the crossentropy loss
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

In [21]:
```
def loss(model, x, y, training):
  # TODO 2
  # training=training is needed only if there are layers with different
  # behavior during training versus inference (e.g. Dropout).
  y_ = model(x, training=training)

  return loss_object(y_true=y, y_pred=y_)


l = loss(model, features, labels, training=False)
print("Loss test: {}".format(l))
```

```
Loss test: 1.0735416412353516
```

Use the `tf.GradientTape` context to calculate the gradients used to optimize your model:

In [22]:
```
def grad(model, inputs, targets):
  with tf.GradientTape() as tape:
    loss_value = loss(model, inputs, targets, training=True)
  return loss_value, tape.gradient(loss_value, model.trainable_variables)
```

Create an optimizer

An optimizer applies the computed gradients to the model's variables to minimize the `loss` function. You can think of the loss function as a curved surface (see Figure 3) and we want to find its lowest point by walking around. The gradients point in the direction of steepest ascent—so we'll travel the opposite way and move down the hill. By iteratively calculating the loss and gradient for each batch, we'll adjust the model during training. Gradually, the model will find the best combination of weights and bias to minimize loss. And the lower the loss, the better the model's predictions.

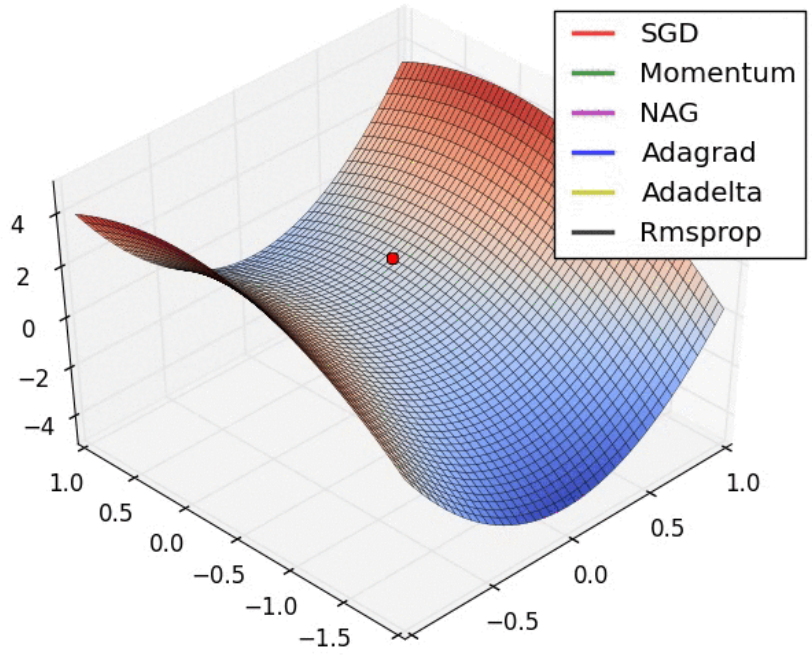


Figure 3. Optimization algorithms visualized over time in 3D space.
(Source: Stanford class CS231n, MIT License, Image credit: Alec Radford)

TensorFlow has many optimization algorithms available for training. This model uses the `tf.keras.optimizers.SGD` that implements the stochastic gradient descent (SGD) algorithm. The `learning_rate` sets the step size to take for each iteration down the hill. This is a *hyperparameter* that you'll commonly adjust to achieve better results.

Let's setup the optimizer:

In [23]:
```
# `tf.keras.optimizers.SGD()` will Gradient descent (with momentum) optimizer.
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

We'll use this to calculate a single optimization step:

In [24]:
```
loss_value, grads = grad(model, features, labels)

# Let's output the value of `Initial Loss` at `step 0`
print("Step: {}, Initial Loss: {}".format(optimizer.iterations.numpy(),
                                          loss_value.numpy()))

optimizer.apply_gradients(zip(grads, model.trainable_variables))

# Let's output the value of `Loss` at `step 1`
print("Step: {},Loss: {}".format(optimizer.iterations.numpy(),
                                 loss(model, features, labels, training=True).
```

```
Step: 0, Initial Loss: 1.0735416412353516
Step: 1,Loss: 1.0283674001693726
```

Training loop

With all the pieces in place, the model is ready for training! A training loop feeds the dataset examples into the model to help it make better predictions. The following code block sets up these training steps:

1. Iterate each *epoch*. An epoch is one pass through the dataset.
2. Within an epoch, iterate over each example in the training `Dataset` grabbing its *features* (`x`) and *label* (`y`).
3. Using the example's features, make a prediction and compare it with the label. Measure the inaccuracy of the prediction and use that to calculate the model's loss and gradients.
4. Use an `optimizer` to update the model's variables.
5. Keep track of some stats for visualization.
6. Repeat for each epoch.

The `num_epochs` variable is the number of times to loop over the dataset collection. Counter-intuitively, training a model longer does not guarantee a better model. `num_epochs` is a hyperparameter that you can tune. Choosing the right number usually requires both experience and experimentation:

In [25]:
```
## Note: Rerunning this cell uses the same model variables

# Keep results for plotting
train_loss_results = []
```

```
train_accuracy_results = []

num_epochs = 201

for epoch in range(num_epochs):
  epoch_loss_avg = tf.keras.metrics.Mean()
  epoch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

  # Training loop - using batches of 32
  for x, y in train_dataset:
    # Optimize the model
    loss_value, grads = grad(model, x, y)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

    # Track progress
    epoch_loss_avg.update_state(loss_value)  # Add current batch loss
    # Compare predicted label to actual label
    # training=True is needed only if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    epoch_accuracy.update_state(y, model(x, training=True))

  # End epoch
  train_loss_results.append(epoch_loss_avg.result())
  train_accuracy_results.append(epoch_accuracy.result())

  if epoch % 50 == 0:
    print("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch,
                                                                epoch_loss_avg.result()
                                                                epoch_accuracy.result()
```

```
Epoch 000: Loss: 1.032, Accuracy: 63.333%
Epoch 050: Loss: 0.399, Accuracy: 93.333%
Epoch 100: Loss: 0.231, Accuracy: 96.667%
Epoch 150: Loss: 0.159, Accuracy: 97.500%
Epoch 200: Loss: 0.116, Accuracy: 97.500%
```

Visualize the loss function over time

While it's helpful to print out the model's training progress, it's often *more* helpful to see this progress. TensorBoard is a nice visualization tool that is packaged with TensorFlow, but we can create basic charts using the `matplotlib` module.

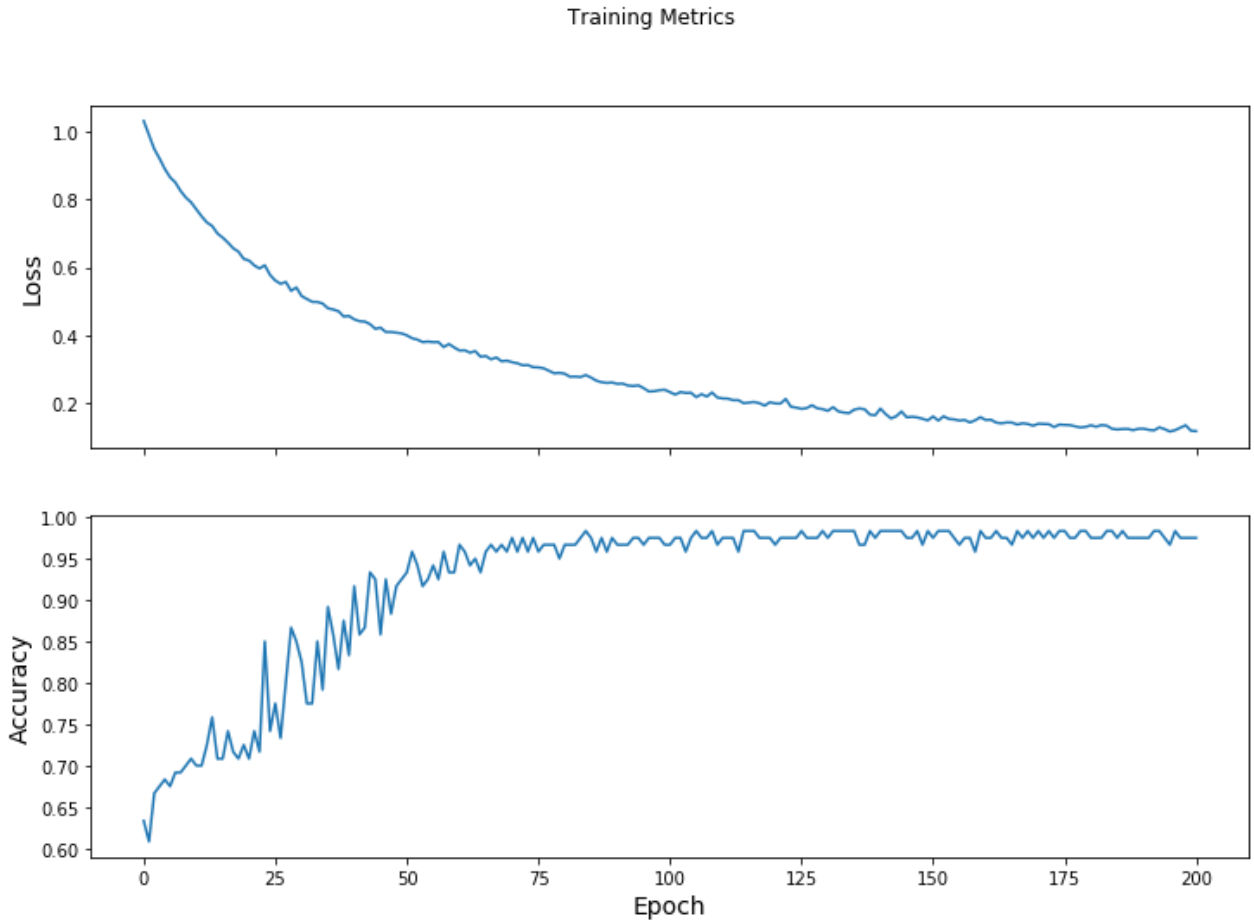
Interpreting these charts takes some experience, but you really want to see the *loss* go down and the *accuracy* go up:

In [26]:

```
fig, axes = plt.subplots(2, sharex=True, figsize=(12, 8))
fig.suptitle('Training Metrics')

axes[0].set_ylabel("Loss", fontsize=14)
axes[0].plot(train_loss_results)

axes[1].set_ylabel("Accuracy", fontsize=14)
axes[1].set_xlabel("Epoch", fontsize=14)
axes[1].plot(train_accuracy_results)
# `plt.show()` will display a figure
plt.show()
```



Evaluate the model's effectiveness

Now that the model is trained, we can get some statistics on its performance.

Evaluating means determining how effectively the model makes predictions. To determine the model's effectiveness at Iris classification, pass some sepal and petal measurements to the model and ask the model to predict what Iris species they represent. Then compare the model's predictions against the actual label. For example, a model that picked the correct species on half the input examples has an accuracy of `0.5` . Figure 4 shows a slightly more effective model, getting 4 out of 5 predictions correct at 80% accuracy:

Example features				Label	Model prediction
5.9	3.0	4.3	1.5	1	1
6.9	3.1	5.4	2.1	2	2
5.1	3.3	1.7	0.5	0	0
6.0	3.4	4.5	1.6	1	2
5.5	2.5	4.0	1.3	1	1

Figure 4. An Iris classifier that is 80% accurate.

Setup the test dataset

Evaluating the model is similar to training the model. The biggest difference is the examples come from a separate test set rather than the training set. To fairly assess a model's effectiveness, the examples used to evaluate a model must be different from the examples used to train the model.

The setup for the test `Dataset` is similar to the setup for training `Dataset`. Download the CSV text file and parse that values, then give it a little shuffle:

In [27]:
```
test_url = "https://storage.googleapis.com/download.tensorflow.org/data/iris_test.csv"

# The `tf.keras.utils.get_file` will downloads a file from a URL if it not already in t
test_fp = tf.keras.utils.get_file(fname=os.path.basename(test_url),
                                  origin=test_url)
```

In [28]:
```
# The `tf.data.experimental.make_csv_dataset()` method reads CSV files into a dataset
test_dataset = tf.data.experimental.make_csv_dataset(
    test_fp,
    batch_size,
    column_names=column_names,
    label_name='species',
    num_epochs=1,
    shuffle=False)

# The `map()` method will pack the `features` into the training dataset:
test_dataset = test_dataset.map(pack_features_vector)
```

Evaluate the model on the test dataset

Unlike the training stage, the model only evaluates a single epoch of the test data. In the following code cell, we iterate over each example in the test set and compare the model's prediction against the actual label. This is used to measure the model's accuracy across the entire test set:

In [29]:
```
test_accuracy = tf.keras.metrics.Accuracy()

for (x, y) in test_dataset:
  # training=False is needed only if there are layers with different
  # behavior during training versus inference (e.g. Dropout).
  logits = model(x, training=False)
  prediction = tf.argmax(logits, axis=1, output_type=tf.int32)
  test_accuracy(prediction, y)

print("Test set accuracy: {:.3%}".format(test_accuracy.result()))
```

```
Test set accuracy: 96.667%
```

We can see on the last batch, for example, the model is usually correct:

In [30]:
```
# Using `tf.stack` we can stack a list of rank-R tensors into one rank-(R+1) tensor.
tf.stack([y,prediction],axis=1)
```

Out[30]:
```
<tf.Tensor: shape=(30, 2), dtype=int32, numpy=
array([[1, 1],
       [2, 2],
       [0, 0],
```

```
       [1, 1],
       [1, 1],
       [1, 1],
       [0, 0],
       [2, 2],
       [1, 1],
       [2, 2],
       [2, 2],
       [0, 0],
       [2, 2],
       [1, 1],
       [1, 1],
       [0, 0],
       [1, 1],
       [0, 0],
       [0, 0],
       [2, 2],
       [0, 0],
       [1, 1],
       [2, 2],
       [1, 2],
       [1, 1],
       [1, 1],
       [0, 0],
       [1, 1],
       [2, 2],
       [1, 1]], dtype=int32)>
```

Use the trained model to make predictions

We've trained a model and "proven" that it's good—but not perfect—at classifying Iris species. Now let's use the trained model to make some predictions on unlabeled examples; that is, on examples that contain features but not a label.

In real-life, the unlabeled examples could come from lots of different sources including apps, CSV files, and data feeds. For now, we're going to manually provide three unlabeled examples to predict their labels. Recall, the label numbers are mapped to a named representation as:

- `0` : Iris setosa
- `1` : Iris versicolor
- `2` : Iris virginica

In [31]:

```
# TODO 3
predict_dataset = tf.convert_to_tensor([
    [5.1, 3.3, 1.7, 0.5,],
    [5.9, 3.0, 4.2, 1.5,],
    [6.9, 3.1, 5.4, 2.1]
])

# training=False is needed only if there are layers with different
# behavior during training versus inference (e.g. Dropout).
predictions = model(predict_dataset, training=False)

for i, logits in enumerate(predictions):
  class_idx = tf.argmax(logits).numpy()
  p = tf.nn.softmax(logits)[class_idx]
  name = class_names[class_idx]
  print("Example {} prediction: {} ({:4.1f}%)".format(i, name, 100*p))
```

```
Example 0 prediction: Iris setosa (97.6%)
Example 1 prediction: Iris versicolor (87.9%)
Example 2 prediction: Iris virginica (88.8%)
```