Getting Started for ML Beginners

This document explains how to use machine learning to classify (categorize) Iris flowers by species. This document dives deeply into the TensorFlow code to do exactly that, explaining ML fundamentals along the way.

If the following list describes you, then you are in the right place:

- You know little to nothing about machine learning.
- You want to learn how to write TensorFlow programs.
- You can code (at least a little) in Python.

If you are already familiar with basic machine learning concepts but are new to TensorFlow, read Getting Started with TensorFlow: for ML Experts.

The Iris classification problem

Imagine you are a botanist seeking an automated way to classify each Iris flower you find. Machine learning provides many ways to classify flowers. 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 solely on the length and width of their sepals and petals.

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

- Iris setosa
- Iris virginica
- Iris versicolor

From left to right, 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 data set of 120 Iris flowers with the sepal and petal measurements. This data set has become one of the canonical introductions to machine learning classification problems. (The MNIST database, which contains handwritten digits, is another popular classification problem.) The first 5 entries of the Iris data set look as follows:

Sepal length	sepal width	petal length	petal width	species
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
5.7	3.8	1.7	0.3	0

Let's introduce some terms:

- The last column (species) is called the **label**; the first four columns are called **features**. Features are characteristics of an example, while the label is the thing we're trying to predict.
- An **example** consists of the set of features and the label for one sample flower. The preceding table shows 5 examples from a data set of 120 examples.

Each label is naturally a string (for example, "setosa"), but machine learning typically relies on numeric values. Therefore, someone mapped each string to a number. Here's the representation scheme:

- 0 represents setosa
- 1 represents versicolor
- 2 represents virginica

Models and training

A **model** is the relationship between features and the label. For the Iris problem, the model defines the relationship between the sepal and petal measurements and the Iris species. Some simple models can be described with a few lines of algebra; more complex machine learning models contain such a large number of interlacing mathematical functions and parameters that they become hard to summarize mathematically.

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? Maybe. You could play with the data set long enough to determine the right relationships of petal and sepal measurements to particular species. However, a good machine learning approach *determines the model for you*. That is, if you feed enough representative examples into the right machine learning model type, the program will determine the relationship between sepals, petals, and species.

Training is the stage of machine learning in which the model is gradually optimized (learned). The Iris problem is an example of **supervised machine learning** in which a 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.)

Get the sample program

Prior to playing with the sample code in this document, do the following:

1. Install TensorFlow.
2. If you installed TensorFlow with virtualenv or Anaconda, activate your TensorFlow environment.
3. Install or upgrade pandas by issuing the following command:

   ```
   pip install pandas
   ```

Take the following steps to get the sample program:

1. Clone the TensorFlow Models repository from github by entering the following command:

   ```
   git clone https://github.com/tensorflow/models
   ```

2. Change directory within that branch to the location containing the examples used in this document:

   ```
   cd models/samples/core/get_started/
   ```

In that `get_started` directory, you'll find a program named `premade_estimator.py`.

Run the sample program

You run TensorFlow programs as you would run any Python program. Therefore, issue the following command from a command line to run `premade_estimators.py`:

```
python premade_estimator.py
```

Running the program should output a whole bunch of information ending with three prediction lines like the following:

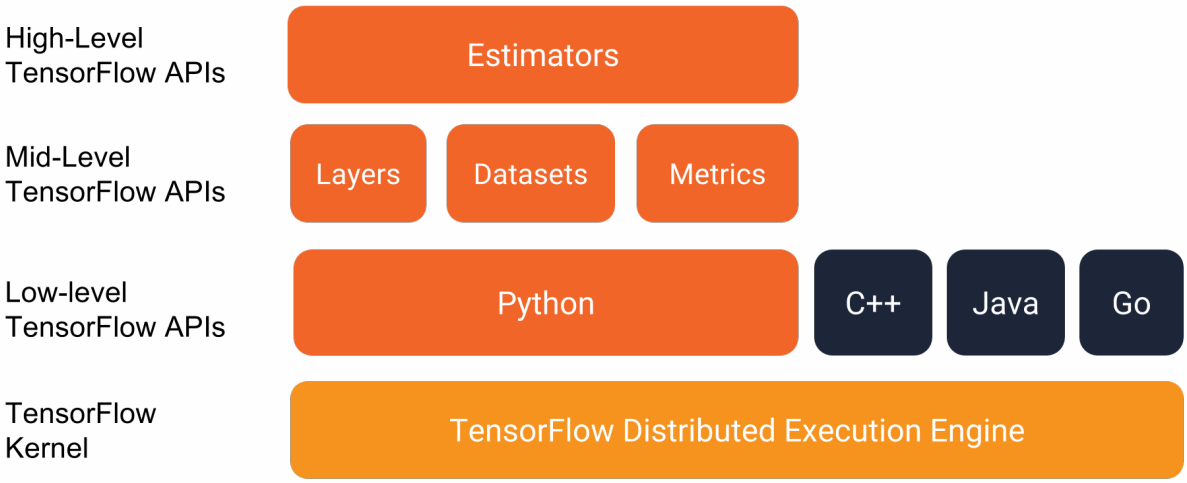
```
...
Prediction is "Setosa" (99.6%), expected "Setosa"

Prediction is "Versicolor" (99.8%), expected "Versicolor"

Prediction is "Virginica" (97.9%), expected "Virginica"
```

If the program generates errors instead of predictions, ask yourself the following questions:

- Did you install TensorFlow properly?
- Are you using the correct version of TensorFlow? The `premade_estimators.py` program requires at least TensorFlow v1.4.
- If you installed TensorFlow with virtualenv or Anaconda, did you activate the environment?

The TensorFlow programming stack

As the following illustration shows, TensorFlow provides a programming stack consisting of multiple API layers:



The TensorFlow Programming Environment.

As you start writing TensorFlow programs, we strongly recommend focusing on the following two high-level APIs:

- Estimators
- Datasets

Although we'll grab an occasional convenience function from other APIs, this document focuses on the preceding two APIs.

The program itself

Thanks for your patience; let's dig into the code. The general outline of `premade_estimator.py`--and many other TensorFlow programs--is as follows:

- Import and parse the data sets.
- Create feature columns to describe the data.
- Select the type of model
- Train the model.

- Evaluate the model's effectiveness.
- Let the trained model make predictions.

The following subsections detail each part.

Import and parse the data sets

The Iris program requires the data from the following two .csv files:

- `http://download.tensorflow.org/data/iris_training.csv`, which contains the training set.
- `http://download.tensorflow.org/data/iris_test.csv`, which contains the the test set.

The **training set** contains the examples that we'll use to train the model; the **test set** contains the examples that we'll use to evaluate the trained model's effectiveness.

The training set and test set started out as a single data set. Then, someone split the examples, with the majority going into the training set and the remainder going into the test set. Adding examples to the training set usually builds a better model; however, adding more examples to the test set enables us to better gauge the model's effectiveness. Regardless of the split, the examples in the test set must be separate from the examples in the training set. Otherwise, you can't accurately determine the model's effectiveness.

The `premade_estimators.py` program relies on the `load_data` function in the adjacent `iris_data.py` file to read in and parse the training set and test set. Here is a heavily commented version of the function:

```
TRAIN_URL = "http://download.tensorflow.org/data/iris_training.csv"
TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"

CSV_COLUMN_NAMES = ['SepalLength', 'SepalWidth',
                    'PetalLength', 'PetalWidth', 'Species']

...

def load_data(label_name='Species'):
    """Parses the csv file in TRAIN_URL and TEST_URL."""

    # Create a local copy of the training set.
    train_path = tf.keras.utils.get_file(fname=TRAIN_URL.split('/')[-1],
                                         origin=TRAIN_URL)
    # train_path now holds the pathname:
~/.keras/datasets/iris_training.csv

    # Parse the local CSV file.
    train = pd.read_csv(filepath_or_buffer=train_path,
                        names=CSV_COLUMN_NAMES,  # list of column names
                        header=0  # ignore the first row of the CSV file.
                       )
    # train now holds a pandas DataFrame, which is data structure
    # analogous to a table.

    # 1. Assign the DataFrame's labels (the right-most column) to
train_label.
    # 2. Delete (pop) the labels from the DataFrame.
    # 3. Assign the remainder of the DataFrame to train_features
    train_features, train_label = train, train.pop(label_name)

    # Apply the preceding logic to the test set.
    test_path = tf.keras.utils.get_file(TEST_URL.split('/')[-1], TEST_URL)
    test = pd.read_csv(test_path, names=CSV_COLUMN_NAMES, header=0)
    test_features, test_label = test, test.pop(label_name)

    # Return four DataFrames.
    return (train_features, train_label), (test_features, test_label)
```

Keras is an open-sourced machine learning library; `tf.keras` is a TensorFlow implementation of Keras. The `premade_estimator.py` program only accesses one `tf.keras` function; namely, the `tf.keras.utils.get_file` convenience function, which copies a remote CSV file to a local file system.

The call to `load_data` returns two `(feature,label)` pairs, for the training and test sets respectively:

```
    # Call load_data() to parse the CSV file.
    (train_feature, train_label), (test_feature, test_label) = load_data()
```

Pandas is an open-source Python library leveraged by several TensorFlow functions. A pandas **DataFrame** is a table with named columns headers and numbered rows. The features returned by `load_data` are packed in `DataFrames`. For example, the `test_feature` DataFrame looks as follows:

```
    SepalLength  SepalWidth  PetalLength  PetalWidth
0           5.9         3.0          4.2         1.5
1           6.9         3.1          5.4         2.1
2           5.1         3.3          1.7         0.5
...
27          6.7         3.1          4.7         1.5
28          6.7         3.3          5.7         2.5
29          6.4         2.9          4.3         1.3
```

Describe the data

A **feature column** is a data structure that tells your model how to interpret the data in each feature. In the Iris problem, we want the model to interpret the data in each feature as its literal floating-point value; that is, we want the model to interpret an input value like 5.4 as, well, 5.4. However, in other machine learning problems, it is often desirable to interpret data less literally. Using feature columns to interpret data is such a rich topic that we devote an entire document to it.

From a code perspective, you build a list of `feature_column` objects by calling functions from the `tf.feature_column` module. Each object describes an input to the model. To tell the model to interpret data as a floating-point value, call @{tf.feature_column.numeric_column). In `premade_estimator.py`, all four features should be interpreted as literal floating-point values, so the code to create a feature column looks as follows:

```
# Create feature columns for all features.
my_feature_columns = []
for key in train_x.keys():
    my_feature_columns.append(tf.feature_column.numeric_column(key=key))
```

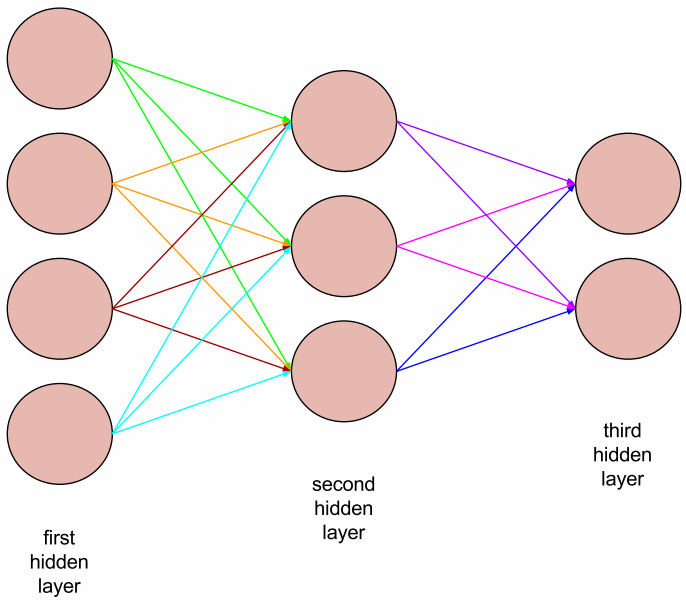
Here is a less elegant, but possibly clearer, alternative way to encode the preceding block:

```
my_feature_columns = [
    tf.feature_column.numeric_column(key='SepalLength'),
    tf.feature_column.numeric_column(key='SepalWidth'),
    tf.feature_column.numeric_column(key='PetalLength'),
    tf.feature_column.numeric_column(key='PetalWidth')
]
```

Select the type of model

We need the select the kind of model that will be trained. Lots of model types exist; picking the ideal type takes experience. We've selected a neural network to solve the Iris problem. **Neural networks** can find complex relationships between features and the label. A neural network 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. We'll be using a **fully connected neural network**, which means that the neurons in one layer take inputs from *every* neuron in the previous layer. For example, the following figure illustrates a fully connected neural network consisting of three hidden layers:

- The first hidden layer contains four neurons.
- The second hidden layer contains three neurons.
- The third hidden layer contains two neurons.



A neural network with three hidden layers.

To specify a model type, instantiate an **Estimator** class. TensorFlow provides two categories of Estimators:

- **pre-made Estimators**, which someone else has already written for you.
- **custom Estimators**, which you must code yourself, at least partially.

To implement a neural network, the `premade_estimators.py` program uses a pre-made Estimator named `tf.estimator.DNNClassifier`. This Estimator builds a neural network that classifies examples. The following call instantiates `DNNClassifier`:

```
    classifier = tf.estimator.DNNClassifier(
        feature_columns=my_feature_columns,
        hidden_units=[10, 10],
        n_classes=3)
```

Use the `hidden_units` parameter to define the number of neurons in each hidden layer of the neural network. Assign this parameter a list. For example:

```
        hidden_units=[10, 10],
```

The length of the list assigned to `hidden_units` identifies the number of hidden layers (2, in this case). Each value in the list represents the number of neurons in a particular hidden layer (10 in the first hidden layer and 10 in the second hidden layer). To change the number of hidden layers or neurons, simply assign a different list to the `hidden_units` parameter.

The ideal number of hidden layers and neurons depends on the problem and the data set. Like many aspects of machine learning, picking the ideal shape of the neural network requires some 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.

The `n_classes` parameter specifies the number of possible values that the neural network can predict. Since the Iris problem classifies 3 Iris species, we set `n_classes` to 3.

The constructor for `tf.Estimator.DNNClassifier` takes an optional argument named `optimizer`, which our sample code chose not to specify. The **optimizer** controls how the model will train. As you develop more expertise in machine learning, optimizers and **learning rate** will become very important.

Train the model

Instantiating a `tf.Estimator.DNNClassifier` creates a framework for learning the model. Basically, we've wired a network but haven't yet let data flow through it. To train the neural network, call the Estimator object's `train` method. For example:

```
    classifier.train(
        input_fn=lambda:train_input_fn(train_feature, train_label,
args.batch_size),
        steps=args.train_steps)
```

The `steps` argument tells `train` to stop training after the specified number of iterations. Increasing `steps`increases the amount of time the model will train. Counter-intuitively, training a model longer does not guarantee a better model. The default value of `args.train_steps` is 1000. The number of steps to train is a**hyperparameter** you can tune. Choosing the right number of steps usually requires both experience and experimentation.

The `input_fn` parameter identifies the function that supplies the training data. The call to the `train` method indicates that the `train_input_fn` function will supply the training data. Here's that method's signature:

```
def train_input_fn(features, labels, batch_size):

```

We're passing the following arguments to `train_input_fn`:

- `train_feature`

  is a Python dictionary in which:

  - Each key is the name of a feature.
  - Each value is an array containing the values for each example in the training set.
- `train_label` is an array containing the values of the label for every example in the training set.
- `args.batch_size` is an integer defining the **batch size**.

The `train_input_fn` function relies on the **Dataset API**. This is a high-level TensorFlow API for reading data and transforming it into a form that the `train` method requires. The following call converts the input features and labels into a `tf.data.Dataset` object, which is the base class of the Dataset API:

```
    dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))

```

The `tf.dataset` class provides many useful functions for preparing examples for training. The following line calls three of those functions:

```
    dataset =
dataset.shuffle(buffer_size=1000).repeat(count=None).batch(batch_size)

```

Training works best if the training examples are in random order. To randomize the examples, call`tf.data.Dataset.shuffle`. Setting the `buffer_size` to a value larger than the number of examples (120) ensures that the data will be well shuffled.

During training, the `train` method typically processes the examples multiple times. Calling the`tf.data.Dataset.repeat` method without any arguments ensures that the `train` method has an infinite supply of (now shuffled) training set examples.

The `train` method processes a **batch** of examples at a time. The `tf.data.Dataset.batch` method creates a batch by concatenating multiple examples. This program sets the default **batch size** to 100, meaning that the `batch` method will concatenate groups of 100 examples. The ideal batch size depends on the problem. As a rule of thumb, smaller batch sizes usually enable the `train` method to train the model faster at the expense (sometimes) of accuracy.

The following `return` statement passes a batch of examples back to the caller (the `train` method).

```
    return dataset.make_one_shot_iterator().get_next()
```

Evaluate the model

Evaluating means determining how effectively the model makes predictions. To determine the Iris classification model's effectiveness, 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 prediction against the actual label. For example, a model that picked the correct species on half the input examples would have an accuracy of 0.5. The following suggests a more effective model:

Test Set

Features	Label	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

A model that is 80% accurate.

To evaluate a model's effectiveness, each Estimator provides an `evaluate` method. The `premade_estimator.py` program calls `evaluate` as follows:

```
# Evaluate the model.
eval_result = classifier.evaluate(
    input_fn=lambda:eval_input_fn(test_x, test_y, args.batch_size))

print('\nTest set accuracy: {accuracy:0.3f}\n'.format(**eval_result))
```

The call to `classifier.evaluate` is similar to the call to `classifier.train`. The biggest difference is that `classifier.evaluate` must get its examples from the test set rather than the training set. In other words, 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 `eval_input_fn` function serves a batch of examples from the test set. Here's the `eval_input_fn` method:

```
def eval_input_fn(features, labels=None, batch_size=None):
    """An input function for evaluation or prediction"""
    if labels is None:
        # No labels, use only features.
        inputs = features
    else:
        inputs = (features, labels)

    # Convert inputs to a tf.dataset object.
    dataset = tf.data.Dataset.from_tensor_slices(inputs)

    # Batch the examples
    assert batch_size is not None, "batch_size must not be None"
    dataset = dataset.batch(batch_size)

    # Return the read end of the pipeline.
    return dataset.make_one_shot_iterator().get_next()
```

In brief, `eval_input_fn` does the following when called by `classifier.evaluate`:

1. Converts the features and labels from the test set to a `tf.dataset` object.
2. Creates a batch of test set examples. (There's no need to shuffle or repeat the test set examples.)
3. Returns that batch of test set examples to `classifier.evaluate`.

Running this code yields the following output (or something close to it):

```
Test set accuracy: 0.967
```

An accuracy of 0.967 implies that our trained model correctly classified 29 out of the 30 Iris species in the test set.

Predicting

We've now trained a model and "proven" that it is 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 simply going to manually provide the following three unlabeled examples:

```
    predict_x = {
        'SepalLength': [5.1, 5.9, 6.9],
        'SepalWidth': [3.3, 3.0, 3.1],
        'PetalLength': [1.7, 4.2, 5.4],
        'PetalWidth': [0.5, 1.5, 2.1],
    }
```

Every Estimator provides a `predict` method, which `premade_estimator.py` calls as follows:

```
predictions = classifier.predict(
    input_fn=lambda:eval_input_fn(predict_x, batch_size=args.batch_size))
```

As with the `evaluate` method, our `predict` method also gathers examples from the `eval_input_fn`method.

When doing predictions, we're *not* passing labels to `eval_input_fn`. Therefore, `eval_input_fn` does the following:

1. Converts the features from the 3-element manual set we just created.
2. Creates a batch of 3 examples from that manual set.
3. Returns that batch of examples to `classifier.predict`.

The `predict` method returns a python iterable, yielding a dictionary of prediction results for each example. This dictionary contains several keys. The `probabilities` key holds a list of three floating-point values, each representing the probability that the input example is a particular Iris species. For example, consider the following `probabilities` list:

```
'probabilities': array([  1.19127117e-08,   3.97069454e-02,   9.60292995e-
01])
```

The preceding list indicates:

- A negligible chance of the Iris being Setosa.
- A 3.97% chance of the Iris being Versicolor.
- A 96.0% chance of the Iris being Virginica.

The `class_ids` key holds a one-element array that identifies the most probable species. For example:

```
'class_ids': array([2])
```

The number `2` corresponds to Virginica. The following code iterates through the returned `predictions` to report on each prediction:

```
for pred_dict, expec in zip(predictions, expected):
    template = ('\nPrediction is "{}" ({:.1f}%), expected "{}"')

    class_id = pred_dict['class_ids'][0]
    probability = pred_dict['probabilities'][class_id]
    print(template.format(SPECIES[class_id], 100 * probability, expec))
```

Running the program yields the following output:

```
...
Prediction is "Setosa" (99.6%), expected "Setosa"

Prediction is "Versicolor" (99.8%), expected "Versicolor"

Prediction is "Virginica" (97.9%), expected "Virginica"
```

Summary

This document provides a short introduction to machine learning.

Because `premade_estimators.py` relies on high-level APIs, much of the mathematical complexity in machine learning is hidden. If you intend to become more proficient in machine learning, we recommend ultimately learning more about **gradient descent**, batching, and neural networks.

We recommend reading the Feature Columns document next, which explains how to represent different kinds of data in machine learning.