Logging and Monitoring Basics with tf.contrib.learn

When training a model, it's often valuable to track and evaluate progress in real time. In this tutorial, you'll learn how to use TensorFlow's logging capabilities and the `Monitor` API to audit the in-progress training of a neural network classifier for categorizing irises. This tutorial builds on the code developed in tf.contrib.learn Quickstart (https://www.tensorflow.org/get_started/tflearn) so if you haven't yet completed that tutorial, you may want to explore it first, especially if you're looking for an intro/refresher on tf.contrib.learn basics.

Setup

For this tutorial, you'll be building upon the following code from tf.contrib.learn Quickstart (https://www.tensorflow.org/get_started/tflearn):

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os

import numpy as np
import tensorflow as tf

# Data sets
IRIS_TRAINING = os.path.join(os.path.dirname(__file__), "iris_training.csv")
IRIS_TEST = os.path.join(os.path.dirname(__file__), "iris_test.csv")

def main(unused_argv):
    # Load datasets.
    training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
        filename=IRIS_TRAINING, target_dtype=np.int, features_dtype=np.float32
    test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
        filename=IRIS_TEST, target_dtype=np.int, features_dtype=np.float32)

    # Specify that all features have real-value data
    feature_columns = [tf.contrib.layers.real_valued_column("", dimension=4)]

    # Build 3 layer DNN with 10, 20, 10 units respectively.
    classifier = tf.contrib.learn.DNNClassifier(feature_columns=feature_columns
                                                hidden_units=[10, 20, 10],
                                                n_classes=3,
```

```
                                                    model_dir="/tmp/iris_model")

    # Fit model.
    classifier.fit(x=training_set.data,
                   y=training_set.target,
                   steps=2000)

    # Evaluate accuracy.
    accuracy_score = classifier.evaluate(x=test_set.data,
                                         y=test_set.target)["accuracy"]
    print('Accuracy: {0:f}'.format(accuracy_score))

    # Classify two new flower samples.
    new_samples = np.array(
        [[6.4, 3.2, 4.5, 1.5], [5.8, 3.1, 5.0, 1.7]], dtype=float)
    y = list(classifier.predict(new_samples, as_iterable=True))
    print('Predictions: {}'.format(str(y)))

if __name__ == "__main__":
  tf.app.run()
```

Copy the above code into a file, and download the corresponding training (http://download.tensorflow.org/data/iris_training.csv) and test (http://download.tensorflow.org/data/iris_test.csv) data sets to the same directory.

In the following sections, you'll progressively make updates to the above code to add logging and monitoring capabilities. Final code incorporating all updates is available for download here (https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/monitors/iris_monitors.py).

Overview

The tf.contrib.learn Quickstart tutorial (https://www.tensorflow.org/get_started/tflearn) walked through how to implement a neural net classifier to categorize iris examples into one of three species.

But when the code (#setup) from this tutorial is run, the output contains no logging tracking how model training is progressing—only the results of the `print` statements that were included:

```
Accuracy: 0.933333
Predictions: [1 2]
```

Without any logging, model training feels like a bit of a black box; you can't see what's happening as TensorFlow steps through gradient descent, get a sense of whether the model is converging appropriately, or audit to determine whether early stopping (https://en.wikipedia.org/wiki/Early_stopping) might be appropriate.

One way to address this problem would be to split model training into multiple `fit` calls with smaller numbers of steps in order to evaluate accuracy more progressively. However, this is not recommended practice, as it greatly slows down model training. Fortunately, tf.contrib.learn offers another solution: a Monitor API (https://www.tensorflow.org/api_docs/python/tf/contrib/learn/monitors) designed to help you log metrics and evaluate your model while training is in progress. In the following sections, you'll learn how to enable logging in TensorFlow, set up a ValidationMonitor to do streaming evaluations, and visualize your metrics using TensorBoard.

Enabling Logging with TensorFlow

TensorFlow uses five different levels for log messages. In order of ascending severity, they are **`DEBUG`**, **`INFO`**, **`WARN`**, **`ERROR`**, and **`FATAL`**. When you configure logging at any of these levels, TensorFlow will output all log messages corresponding to that level and all levels of higher severity. For example, if you set a logging level of **`ERROR`**, you'll get log output containing **`ERROR`** and **`FATAL`** messages, and if you set a level of **`DEBUG`**, you'll get log messages from all five levels.

By default, TensorFlow is configured at a logging level of **`WARN`**, but when tracking model training, you'll want to adjust the level to **`INFO`**, which will provide additional feedback as **`fit`** operations are in progress.

Add the following line to the beginning of your code (right after your **`imports`**):

```
tf.logging.set_verbosity(tf.logging.INFO)
```

Now when you run the code, you'll see additional log output like the following:

```
INFO:tensorflow:loss = 1.18812, step = 1
INFO:tensorflow:loss = 0.210323, step = 101
INFO:tensorflow:loss = 0.109025, step = 201
```

With **`INFO`**-level logging, tf.contrib.learn automatically outputs training-loss metrics (https://en.wikipedia.org/wiki/Loss_function) to stderr after every 100 steps.

Configuring a ValidationMonitor for Streaming Evaluation

Logging training loss is helpful to get a sense whether your model is converging, but what if you want further insight into what's happening during training? tf.contrib.learn provides several high-level `Monitors` you can attach to your `fit` operations to further track metrics and/or debug lower-level TensorFlow operations during model training, including:

Monitor	Description
`CaptureVariable`	Saves a specified variable's values into a collection at every *n* steps of training
`PrintTensor`	Logs a specified tensor's values at every *n* steps of training
`SummarySaver`	Saves `tf.Summary` (https://www.tensorflow.org/api_docs/python/tf/Summary) p (https://developers.google.com/protocol-buffers/) for a given tensor using a `tf.summary.FileWriter` (https://www.tensorflow.org/api_docs/python/tf/sun at every *n* steps of training
`ValidationMonitor`	Logs a specified set of evaluation metrics at every *n* steps of training, and, if desire early stopping under certain conditions

Evaluating Every *N* Steps

用ValidationMonitor每隔N步在测试集上验证一次准确率

For the iris neural network classifier, while logging training loss, you might also want to simultaneously evaluate against test data to see how well the model is generalizing. You can accomplish this by configuring a `ValidationMonitor` with the test data (`test_set.data` and `test_set.target`), and setting how often to evaluate with `every_n_steps`. The default value of `every_n_steps` is `100`; here, set `every_n_steps` to `50` to evaluate after every 50 steps of model training:

```
validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
    test_set.data,
    test_set.target,
    every_n_steps=50)
```

Place this code right before the line instantiating the `classifier`.

`ValidationMonitors` rely on saved checkpoints to perform evaluation operations, so you'll want to modify instantiation of the `classifier` to add a `tf.contrib.learn.RunConfig` (https://www.tensorflow.org/api_docs/python/tf/contrib/learn/RunConfig) that includes `save_checkpoints_secs`, which specifies how many seconds should elapse between checkpoint saves during training. Because the iris data set is quite small, and thus trains

quickly, it makes sense to set `save_checkpoints_secs` to 1 (saving a checkpoint every second) to ensure a sufficient number of checkpoints:

```
classifier = tf.contrib.learn.DNNClassifier(
    feature_columns=feature_columns,
    hidden_units=[10, 20, 10],
    n_classes=3,
    model_dir="/tmp/iris_model",
    config=tf.contrib.learn.RunConfig(save_checkpoints_secs=1))
```

NOTE: The `model_dir` parameter specifies an explicit directory (`/tmp/iris_model`) for model data to be stored; this directory path will be easier to reference later on than an autogenerated one. Each time you run the code, any existing data in `/tmp/iris_model` will be loaded, and model training will continue where it left off in the last run (e.g., running the script twice in succession will execute 4000 steps during training—2000 during each `fit` operation). To start over model training from scratch, delete `/tmp/iris_model` before running the code.

Finally, to attach your `validation_monitor`, update the `fit` call to include a `monitors` param, which takes a list of all monitors to run during model training:

```
classifier.fit(x=training_set.data,
               y=training_set.target,
               steps=2000,
               monitors=[validation_monitor])
```

Now, when you rerun the code, you should see validation metrics in your log output, e.g.:

```
INFO:tensorflow:Validation (step 50): loss = 1.71139, global_step = 0, accura
...
INFO:tensorflow:Validation (step 300): loss = 0.0714158, global_step = 268, a
...
INFO:tensorflow:Validation (step 1750): loss = 0.0574449, global_step = 1729,
```

自定义log的内容

Customizing the Evaluation Metrics with MetricSpec

By default, if no evaluation metrics are specified, `ValidationMonitor` will log both loss (https://en.wikipedia.org/wiki/Loss_function) and accuracy, but you can customize the list of metrics that will be run every 50 steps. To specify the exact metrics you'd like to run in each evaluation pass, you can add a `metrics` param to the `ValidationMonitor` constructor. `metrics` takes a dict of key/value pairs, where each key is the name you'd like logged for the metric, and the corresponding value is a `MetricSpec`

(https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/contrib/learn/python/learn/metric_spec.py)
object.

The `MetricSpec` constructor accepts four parameters:

- `metric_fn`. The function that calculates and returns the value of a metric. This can be a predefined function available in the `tf.contrib.metrics` (https://www.tensorflow.org/api_docs/python/tf/contrib/metrics) module, such as `tf.contrib.metrics.streaming_precision` (https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_precision) or `tf.contrib.metrics.streaming_recall` (https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_recall).

  Alternatively, you can define your own custom metric function, which must take `predictions` and `labels` tensors as arguments (a `weights` argument can also optionally be supplied). The function must return the value of the metric in one of two formats:

  - A single tensor
  - A pair of ops `(value_op, update_op)`, where `value_op` returns the metric value and `update_op` performs a corresponding operation to update internal model state.
- `prediction_key`. The key of the tensor containing the predictions returned by the model. This argument may be omitted if the model returns either a single tensor or a dict with a single entry. For a `DNNClassifier` model, class predictions will be returned in a tensor with the key `tf.contrib.learn.PredictionKey.CLASSES` (https://www.tensorflow.org/api_docs/python/tf/contrib/learn/PredictionKey#CLASSES).
- `label_key`. The key of the tensor containing the labels returned by the model, as specified by the model's `input_fn` (https://www.tensorflow.org/get_started/input_fn). As with `prediction_key`, this argument may be omitted if the `input_fn` returns either a single tensor or a dict with a single entry. In the iris example in this tutorial, the `DNNClassifier` does not have an `input_fn` (`x`,`y` data is passed directly to `fit`), so it's not necessary to provide a `label_key`.
- `weights_key`. *Optional*. The key of the tensor (returned by the `input_fn` (https://www.tensorflow.org/get_started/input_fn)) containing weights inputs for the `metric_fn`.

The following code creates a `validation_metrics` dict that defines three metrics to log during model evaluation:

- `"accuracy"`, using `tf.contrib.metrics.streaming_accuracy` (https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_accuracy) as the `metric_fn`
- `"precision"`, using `tf.contrib.metrics.streaming_precision` (https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_precision) as the `metric_fn`
- `"recall"`, using `tf.contrib.metrics.streaming_recall` (https://www.tensorflow.org/api_docs/python/tf/contrib/metrics/streaming_recall) as the `metric_fn`

```
validation_metrics = {
    "accuracy":
        tf.contrib.learn.MetricSpec(
            metric_fn=tf.contrib.metrics.streaming_accuracy,
            prediction_key=tf.contrib.learn.PredictionKey.CLASSES),
    "precision":
        tf.contrib.learn.MetricSpec(
            metric_fn=tf.contrib.metrics.streaming_precision,
            prediction_key=tf.contrib.learn.PredictionKey.CLASSES),
    "recall":
        tf.contrib.learn.MetricSpec(
            metric_fn=tf.contrib.metrics.streaming_recall,
            prediction_key=tf.contrib.learn.PredictionKey.CLASSES)
}
```

Add the above code before the `ValidationMonitor` constructor. Then revise the `ValidationMonitor` constructor as follows to add a `metrics` parameter to log the accuracy, precision, and recall metrics specified in `validation_metrics` (loss is always logged, and doesn't need to be explicitly specified):

```
validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
    test_set.data,
    test_set.target,
    every_n_steps=50,
    metrics=validation_metrics)
```

Rerun the code, and you should see precision and recall included in your log output, e.g.:

```
INFO:tensorflow:Validation (step 50): recall = 0.0, loss = 1.20626, global_st
...
INFO:tensorflow:Validation (step 600): recall = 1.0, loss = 0.0530696, global
...
INFO:tensorflow:Validation (step 1500): recall = 1.0, loss = 0.0617403, globa
```

Early Stopping with ValidationMonitor

Note that in the above log output, by step 600, the model has already achieved precision and recall rates of 1.0. This raises the question as to whether model training could benefit from early stopping (https://en.wikipedia.org/wiki/Early_stopping).

In addition to logging eval metrics, `ValidationMonitor`s make it easy to implement early stopping when specified conditions are met, via three params:

Param	Description
`early_stopping_metric`	Metric that triggers early stopping (e.g., loss or accuracy) under conditions specified in `early_stopping_rounds` and `early_stopping_metric_minimize`. Default is `"loss"`.
`early_stopping_metric_minimize`	`True` if desired model behavior is to minimize the value of `early_stopping_metric`; `False` if desired model behavior is to maximize the value of `early_stopping_metric`. Default is `True`.
`early_stopping_rounds`	Sets a number of steps during which if the `early_stopping_metric` does not decrease (if `early_stopping_metric_minimize` is `True`) or increase (if `early_stopping_metric_minimize` is `False`), training will be stopped. Default is `None`, which means early stopping will never occur.

Make the following revision to the `ValidationMonitor` constructor, which specifies that if loss (`early_stopping_metric="loss"`) does not decrease (`early_stopping_metric_minimize=True`) over a period of 200 steps (`early_stopping_rounds=200`), model training will stop immediately at that point, and not complete the full 2000 steps specified in `fit`:

```
validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
    test_set.data,
    test_set.target,
    every_n_steps=50,
    metrics=validation_metrics,
    early_stopping_metric="loss",
    early_stopping_metric_minimize=True,
    early_stopping_rounds=200)
```

Rerun the code to see if model training stops early:

```
...
INFO:tensorflow:Validation (step 1150): recall = 1.0, loss = 0.056436, global
INFO:tensorflow:Stopping. Best step: 800 with loss = 0.048313818872.
```

Indeed, here training stops at step 1150, indicating that for the past 200 steps, loss did not decrease, and that overall, step 800 produced the smallest loss value against the test data set. This suggests that additional calibration of hyperparameters by decreasing the step count might further improve the model.

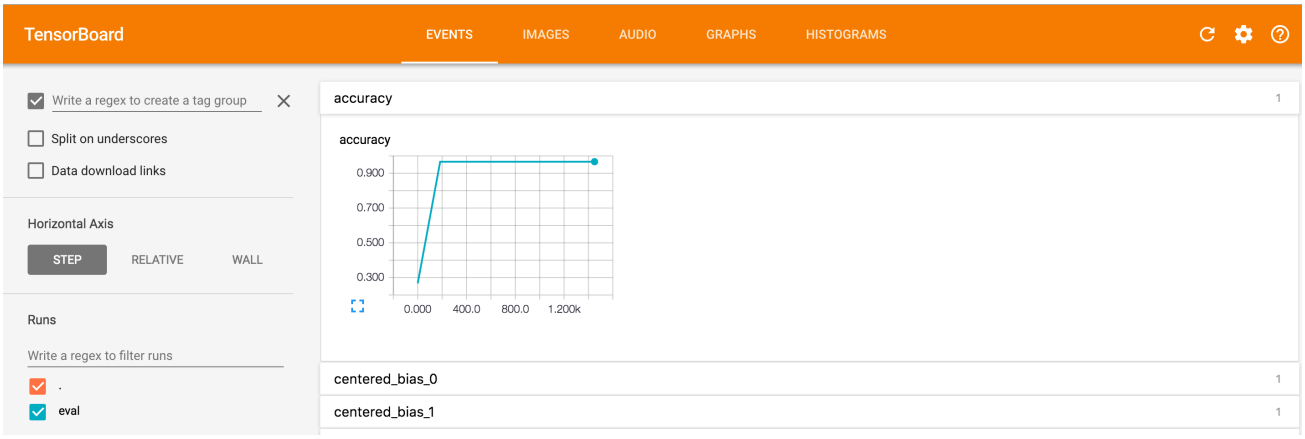
Visualizing Log Data with TensorBoard

Reading through the log produced by `ValidationMonitor` provides plenty of raw data on model performance during training, but it may also be helpful to see visualizations of this data to get further insight into trends—for example, how accuracy is changing over step count. You can use TensorBoard (a separate program packaged with TensorFlow) to plot graphs like this by setting the `logdir` command-line argument to the directory where you saved your model training data (here, `/tmp/iris_model`). Run the following on your command line:

```
$ tensorboard --logdir=/tmp/iris_model/
Starting TensorBoard 39 on port 6006
```

Then navigate to `http://0.0.0.0:<port_number>` in your browser, where `<port_number>` is the port specified in the command-line output (here, `6006`).

If you click on the accuracy field, you'll see an image like the following, which shows accuracy plotted against step count:



For more on using TensorBoard, see TensorBoard: Visualizing Learning (https://www.tensorflow.org/get_started/summaries_and_tensorboard) and TensorBoard: Graph

Visualization (https://www.tensorflow.org/get_started/graph_viz).

上次更新日期：六月 19, 2017