Getting Started with TensorFlow

Part II: Monitoring Training and Validation

Nick Winovich
Department of Mathematics
Purdue University
July 2018
1  Monitored Training Sessions
   - Monitored Sessions and Hooks
   - Flags and General Configuration
   - Checkpoints and Frozen Models

2  TFRecord Files and Validation
   - Working with TFRecord Datasets
   - Dataset Handles and Validation
   - Early Stopping and Custom Hooks
1. Monitored Training Sessions
   - Monitored Sessions and Hooks
   - Flags and General Configuration
   - Checkpoints and Frozen Models

2. TFRecord Files and Validation
   - Working with TFRecord Datasets
   - Dataset Handles and Validation
   - Early Stopping and Custom Hooks
Monitored Sessions in TensorFlow

“Session-like object that handles initialization, recovery and hooks.” (TensorFlow API r1.8)

tf.MonitoredSession’s provide convenient ways for handling:

- Variable initialization
- The use of hooks
- Session recovery after errors are raised

tf.MonitoredTrainingSession’s define training sessions that:

- Automate the process of saving checkpoints and summaries
- Facilitate training TensorFlow graphs on distributed devices
Basic TensorFlow Hooks

Hooks are used to execute various operations during training when the state of a monitored session satisfies certain conditions, e.g.:

- `tf.train.CheckpointSaverHook` saves a checkpoint after specified number of steps or seconds
- `tf.train.StopAtStepHook` stops training after specified number of steps
- `tf.train.NanTensorHook` stops training in the event that an NaN value is encountered
- `tf.train.FinalOpsHook` evaluates specified tensors at the end of the training session
Before initializing a monitored training session, a ‘global step tensor’ (to track the step count) must be added to the graph:

- A global step tensor can be added in `__init__` by setting:
  ```python
  self.step = tf.train.get_or_create_global_step()
  ```

- The step can be accessed in the `train` method using:
  ```python
  step = tf.train.global_step(self.sess, self.step)
  ```

- The step count is incremented by passing it to `minimize`:
  ```python
  tf.train.AdamOptimizer(self.learning_rate)
  .minimize(self.loss, global_step=self.step)
  ```
The `tf.train.MonitoredTrainingSession` object serves as a replacement for the older `tf.train.Supervisor` wrapper.

```python
# Initialize TensorFlow monitored training session
with tf.train.MonitoredTrainingSession(
    checkpoint_dir = './Checkpoints/',
    hooks = [tf.train.StopAtStepHook(last_step=1000)],
    save_checkpoint_steps = 100) as sess:
```

- This creates a monitored session which will run for 1000 steps, saving checkpoints in "./Checkpoints/" every 100 steps
- This is used to replace: "with tf.Session() as sess:"
- Once the monitored session is initialized, the TensorFlow graph is frozen and cannot be modified; in particular, we must run `model.build_model()` and define the global step beforehand
# Initialize model and build graph
model = Model(FLAGS)
model.build_model()

# Initialize TensorFlow monitored training session
with tf.train.MonitoredTrainingSession(
    checkpoint_dir = "./Checkpoints/",
    hooks = [tf.train.StopAtStepHook(last_step=1000)],
    save_checkpoint_steps = 100) as sess:

    # Set model session and train
    model.set_session(sess)
    model.train()

- model.build_model() is run before initializing the session
- The global step can be defined in the Model __init__ method
- The set_session method simply sets "self.sess = sess"
# Define training method
def train(self):

    # Iterate through training steps
    while not self.sess.should_stop():

        # Update global step
        step = tf.train.global_step(self.sess, self.step)

        # Run optimization ops, display progress, etc.

The "while not self.sess.should_stop():" loop is used to continue the training procedure until the monitored training session indicates it should stop (e.g. final step or NaN values)

Hooks are used to determine the state of sess.should_stop by calling run_context.request_stop() after a run() call
Once `request_stop()` is called, later calls to `run()` will raise errors when attempting to use the monitored training session (for example, after the final training step has been completed).

A `tf.Session()` can be used after training and the model can be restored as described in “Checkpoints and Frozen Models”

It is also possible to use a `tf.train.FinalOpsHook`
The value of the learning rate is specified completely by the initial options and current global step; this allows the value to be restored (as opposed to values passed using a feed_dict).

The hyperparameters initial_val, decay_step, and decay_rate are typically passed as flags for tuning.

With staircase=True, decay is applied only after the specified decay step; otherwise it is applied incrementally every step.

```python
lr = tf.train.exponential_decay(self.initial_val, self.step, self.decay_step, self.decay_rate, staircase=True)
```
Note on Saving Summaries*

By default, summaries are saved at global step 0 and may raise an error if a feed dictionary is required to compute a summary.

These errors can be avoided by passing "None" to the summary related options of the monitored training session.

Summaries can then be saved manually as described in Part I.

* It should be possible to redefine tf.train.SummarySaverHook.
Outline

1 Monitored Training Sessions
   - Monitored Sessions and Hooks
   - Flags and General Configuration
   - Checkpoints and Frozen Models

2 TFRecord Files and Validation
   - Working with TFRecord Datasets
   - Dataset Handles and Validation
   - Early Stopping and Custom Hooks
Command line options, or ‘flags’, are used to provide an easy way for specifying training/model hyperparameters at runtime.

- Flags can be passed to Python programs using, for example:

  ```
  $ python train.py --batch_size 64 --use_gpu
  ```

- These flags need to be ‘parsed’ by Python using e.g. `argparse`

- Flags may require arguments (e.g. `--batch_size 64`) or may simply serve as toggles for boolean options (e.g. `--use_gpu`)

- Flags are often useful for running the same code on machines with different types of hardware (e.g. with and without GPUs)
from argparse import ArgumentParser

# Create argument parser for command line flags
parser = ArgumentParser(description="Argument Parser")

# Add arguments to argument parser
parser.add_argument("--training_steps", default=1000,
    type=int, help="Number of training steps")

parser.add_argument("--batch_size", default=64,
    type=int, help="Training batch size")

# Parse arguments from command line
FLAGS = parser.parse_args()

- Example usage: python train.py --batch_size 128

- Argument values are accessed via e.g. FLAGS.batch_size
Unpacking Flags into a Model

# Retrieve a single argument
self.batch_size = FLAGS.batch_size

# Unpack all flags to an object’s dictionary
for key, val in FLAGS.__dict__.items():
    if key not in self.__dict__.keys():
        self.__dict__[key] = val

- Unpacking flags assigns properties e.g. self.batch_size

- All model parameters can typically be passed as flags:
  
    e.g. model = Model(FLAGS)

  and assigned using the second method described above

- This also avoids overriding properties that are already set
from argparse import ArgumentParser

def parse_args():
    parser = ArgumentParser(description="Argument Parser")

    # Add boolean with default value "False"
    parser.add_argument("--use_gpu", default=False,
                        action="store_true", help="Use GPU")

    FLAGS = parser.parse_args()
    return FLAGS

Setting "--use_gpu False" results in "False" $\neq 0 \equiv False$

Instead we can select a default, e.g. False, and automatically store the boolean value True whenever the flag is passed

Now "python train.py --use_gpu" will set use_gpu=True
Note on using GPUs

Using GPUs requires additional steps which are outlined in the API:

- https://www.tensorflow.org/install/
- https://www.tensorflow.org/programmers_guide/using_gpu

- Install the CUDA Toolkit 9.0 and update `LD_LIBRARY_PATH` to include the CUDA library e.g. `/usr/local/cuda-9.0/lib64`
- Install NVIDIA command line tools and GPU drivers
- Install cuDNN SDK v7 (CUDA Deep Neural Network library)
- Install GPU version: `tensorflow-gpu` (also available w/ `pip`)
- Specify GPU to use; e.g. `export CUDA_VISIBLE_DEVICES=0`
Scaffolds can be used to pass custom savers, initialization ops, and summary ops to the training session

tf.ConfigProto is used to help configure hardware/device settings, such as the number of GPUs available
Outline

1. Monitored Training Sessions
   - Monitored Sessions and Hooks
   - Flags and General Configuration
   - Checkpoints and Frozen Models

2. TFRecord Files and Validation
   - Working with TFRecord Datasets
   - Dataset Handles and Validation
   - Early Stopping and Custom Hooks
# Create new session for model evaluation
with tf.Session() as sess:

    # Restore network parameters from latest checkpoint
    saver = tf.train.Saver()
    saver.restore(sess,
                  tf.train.latest_checkpoint("./Checkpoints/"))

    # Set model session using restored sess
    model.set_session(sess)

    # Evaluate model
    eval_loss = model.evaluate()
# Initialize TensorFlow monitored training session

```
with tf.train.MonitoredTrainingSession(
    checkpoint_dir = './Checkpoints/',
    save_checkpoint_steps = 100) as sess:
```

- When the "checkpoint_dir" option of a monitored training session is set, the session will automatically restore from the latest checkpoint in the directory if any are available.

- An error occurs if any parts of the graph have been modified since the previous checkpoint (in particular variable shapes):

  ```
  ... InvalidArgumentError (see above for traceback):
  Assign requires shapes of both tensors to match. ...
  ```
“Protocol buffers are Google’s language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.”
(https://developers.google.com/protocol-buffers)

- Frozen models are used to combine graph definitions specified in graph.pbtxt files with the variables saved in checkpoints.

- Frozen models can be ‘optimized for inference’ by removing the nodes in a graph which are unnecessary for making predictions.

- The protobuf format allows TensorFlow models to be deployed on devices which do not have Python and TensorFlow installed.
Freezing Models from Checkpoints

```python
import tensorflow as tf
from tensorflow.python.tools import freeze_graph

# Freeze model from checkpoint file
def freeze_from_checkpoint():
    checkpoint_dir = "./Checkpoints/
path = tf.train.latest_checkpoint("./Checkpoints/
input_graph_path = "./Checkpoints/graph.pbtxt"
output_nodes = "prediction"
restore_op = "save/restore_all"
filename_tensor = "save/Const:0"
output_name = "/Checkpoints/frozen_model.pb"
freeze_graph.freeze_graph(input_graph_path, "", False, path, output_nodes, restore_op, filename_tensor, output_name, True, "")
```

- This script saves a ‘frozen model’ in the checkpoint directory
- More details can be found in the `freeze_graph` source code
from tensorflow.python.tools import optimize_for_inference_lib

# Optimize frozen .pb file for inference
def optimize_frozen_file():
    frozen_graph_filename = "./Checkpoints/frozen_model.pb"

    with tf.gfile.GFile(frozen_graph_filename, "rb") as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())

    input_node_list = ["x"]
    output_node_list = ["prediction"]
    output_name = "./Checkpoints/optimized.pb"

    output_graph_def = optimize_for_inference_lib.optimize_for_inference(
        graph_def, input_node_list,
        output_node_list, tf.float32.as_datatype_enum)

    f = tf.gfile.FastGFile(output_name, "w")
    f.write(output_graph_def.SerializeToString())
# Load graph from .pb file

def load_graph():
    frozen_filename = "./Checkpoints/optimized.pb"
    with tf.gfile.GFile(frozen_filename, "rb") as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
    with tf.Graph().as_default() as graph:
        tf.import_graph_def(graph_def, name="prefix")
    return graph

# Compute network prediction
graph = load_graph()
x = graph.get_tensor_by_name("prefix/x:0")
pred = graph.get_tensor_by_name("prefix/prediction:0")
with tf.Session(graph=graph) as sess:
    input_data = np.load("input_filename.npy")
    y = sess.run(pred, feed_dict={x: input_data})

"prefix/" is added by import_graph_def (also note the ":0")
Outline

1. Monitored Training Sessions
   - Monitored Sessions and Hooks
   - Flags and General Configuration
   - Checkpoints and Frozen Models

2. TFRecord Files and Validation
   - Working with TFRecord Datasets
   - Dataset Handles and Validation
   - Early Stopping and Custom Hooks
# Import tools for creating features and float/byte lists
from tensorflow.train import Feature, FloatList, BytesList

# Define feature converter for flattened np.uint8 arrays
def _bytes_feature(vals):
    return Feature(bytes_list=BytesList(value=[[vals]]))

# Define feature converter for flattened np.float32 arrays
def _floats_feature(vals):
    return Feature(float_list=FloatList(value=[float(x) for x in vals]))

- A BytesList is typically used to store arrays of small unsigned integers or strings, while a FloatList is used to store floats.

- A set of features is then used to create a tf.train.Features object, converted into an example, and written to a protobuf file.
```python
def write_tfrecords():
    writer = tf.python_io.TFRecordWriter("training.tfrecords")
    for i in range(0,1000):

        # Load uint8 (# float32) arrays from file
        data = np.load("data_" + str(i) + ".npy")
           .flatten().astype(np.uint8)
        # .flatten().astype(np.float32)

        # Define a bytes (# float) feature with label "data"
        feature = {"data": _bytes_feature(data.tolist())}
           # _floats_feature(data.tolist())}

        # Define an example containing the features
        example = Example(features=
                           tf.train.Features(feature=feature))

        # Serialize the protocol buffer to string and write
        writer.write(example.SerializeToString())

    writer.close()
```
# Parse "example_proto" for dataset

def _parse_data(example_proto, res=28):
    
    # Define expected features with shapes and datatypes
    features = {
        "data": tf.FixedLenFeature([res, res, 1], tf.uint8),
        # tf.float32)
    }

    # Parse example from "example_proto"
    parsed = tf.parse_single_example(example_proto, features)

    # Extract and decode array from parsed feature dict
    data = tf.decode_raw(parsed["data"], tf.uint8)
        # parsed["data"]

    return data

Parse functions provide a convenient way to preprocess examples and can also be used for data augmentation.
# Define dataset from single file
dataset = tf.data.TFRecordDataset("training.tfrecords")

# Define dataset from multiple files with parallel reads
filenames = "training-*.tfrecords"
files = tf.data.Dataset.list_files(filenames)

def tfrecord_dataset(fname):
    bs = 4 * 1024 * 1024  # Add 4 mebibyte buffers
    return tf.data.TFRecordDataset(fname, buffer_size=bs)

dataset = files.apply(tf.contrib.data.parallel_interleave(
    tfrecord_dataset, cycle_length=8, sloppy=True))

- Datasets can be constructed from a single tfrecords file or from a list of files (possibly distributed over a network)
- `parallel_interleave` produces a nested collection of datasets and retrieves/interleaves elements in parallel
Fused operations provide optimized alternatives for applying a sequence of operations (as opposed to sequential composition).

\[
\text{tf.contrib.data} \text{ has fused ops for shuffling/repeating as well as mapping/batching tf.data.Dataset objects}
\]
Outline

1. Monitored Training Sessions
   - Monitored Sessions and Hooks
   - Flags and General Configuration
   - Checkpoints and Frozen Models

2. TFRecord Files and Validation
   - Working with TFRecord Datasets
   - Dataset Handles and Validation
   - Early Stopping and Custom Hooks
When working with multiple datasets, it is inconvenient to place separate iterators in the graph for each one. A more natural approach is to define a single iterator which can be instructed to retrieve elements from a specified dataset for each `run()` call.

- **String handles** provide a way of referring to a specific dataset:

  ```python
  train_dh = sess.run(dataset.string_handle())
  ```

- This results in a string which can be easily passed to the model to specify which dataset to use for a particular evaluation.
Defining Feedable Iterators

# Define placeholder for dataset handle
self.d_handle = tf.placeholder(tf.string, shape=[], name="dh")

# Define feedable iterator from dataset string handles
iterator = tf.data.Iterator
        .from_string_handle(self.d_handle,
                             self.dataset.output_types,
                             self.dataset.output_shapes)

# Define operation for getting next batch
data = iterator.get_next()

# Compute network prediction on current batch of data
self.prediction = self.dense_network(data, training=self.train)

- All datasets passed to the iterator must have elements with the same data types and shapes; this information is used to initialize the iterator and determine the structure of the rest of the graph.
Training and validation steps can now be carried out by feeding the model the corresponding dataset handle and training status:

""" Train """
    # Specify feed dictionary for training
    fd = {self.d_handle: train_dh, self.train: True}

    # Update model and save summaries
    _, summary = self.sess.run([self.optim, self.sum_op],
                                feed_dict=fd)
    writer.add_summary(summary, step)

""" Validate """
    # Specify feed dictionary for validation
    fd = {self.d_handle: val_dh, self.train: False}

    # Save validation summaries
    vsummary = self.sess.run(self.sum_op, feed_dict=fd)
    vwriter.add_summary(vsummary, step)
Training and Validation Summaries

Training and validation summaries are automatically handled by TensorBoard by writing to separate subdirectories of "./logs/":

```python
# Define summary writer for saving "training" logs
writer = tf.summary.FileWriter("./logs/training/",
                                    graph=tf.get_default_graph())

# Define summary writer for saving "validation" logs
vwriter = tf.summary.FileWriter("./logs/validation/",
                                    graph=tf.get_default_graph())
```

- The subdirectory names are used as labels for each summary
- Each summary is assigned a distinct color and plotted on the same graph for comparison (helpful for identifying over-fitting)
Training loss shown in orange and validation loss shown in blue; this is a clear example of over-fitting (i.e. poor generalization).
Outline

1. Monitored Training Sessions
   - Monitored Sessions and Hooks
   - Flags and General Configuration
   - Checkpoints and Frozen Models

2. TFRecord Files and Validation
   - Working with TFRecord Datasets
   - Dataset Handles and Validation
   - Early Stopping and Custom Hooks
The model achieves an acceptable accuracy early in the training process; to avoid performing unnecessary training steps (and possibly end up with a less accurate model) it is often helpful to stop the model early when a certain level of accuracy is reached (referred to as *early stopping*).
Session run hooks are used to extend `run()` calls by performing additional operations before and after each call (as well as before and after session initialization and at the end of the session).

`begin()`
- Called once prior to session initialization (e.g. get the global step tensor from the graph)

`before_run()`
- Called before every `sess.run()` call (typically used to specify fetches and feed_dict)

`after_run()`
- Called after every `sess.run()` call (e.g. check loss and call `request_stop()` if below tolerance)
Session Run Hooks: Order of Execution

The source code for SessionRunHook on GitHub provides a basic overview of how to configure custom hooks; this can be found at:

tensorflow/python/training/session_run_hook.py

The pseudocode detailing the execution order is as follows:

call hooks.begin()
sess = tf.Session()
call hooks.after_create_session()
while not stop is requested:
    call hooks.before_run()
    try:
        results = sess.run(merged_fetches, feed_dict=merged_feeds)
    except (errors.OutOfRangeError, StopIteration):
        break
    call hooks.after_run()
call hooks.end()
sess.close()
# Import base model for defining early stopping hook
from tensorflow.python.training.session_run_hook \
    import SessionRunHook, SessionRunArgs

# Define early stopping hook
class EarlyStoppingHook(SessionRunHook):
    def __init__(self, loss_name, feed_dict={},
                 tolerance=0.001, stopping_step=1000):
        self.loss_name = loss_name
        self.feed_dict = feed_dict
        self.tolerance = tolerance
        self.stopping_step = stopping_step

The custom hook inherits from SessionRunHook so that the
monitored training session will automatically handle calls to
begin(), before_run(), and after_run().
# Initialize global and internal step counts

def begin(self):
    self._global_step = tf.train.get_global_step()
    if self._global_step is None:
        raise RuntimeError("Global step must be defined."")
    self._step = 0

- A predefined function for retrieving the global step tensor is provided in the `tf.train` module

- The session and graph are accessible through the `run_context` argument passed to hook methods from the monitored session

- Additional tensors in the graph can be retrieved using:

  ```python
  graph = run_context.session.graph
tensor = graph.get_tensor_by_name("name")
  ```
# Specify feed_dict and tensors to be evaluated

def before_run(self, run_context):
    if self._step % self.stopping_step == 0:

        # Get graph from run_context and loss from graph
        graph = run_context.session.graph
        loss = graph.get_tensor_by_name(self.loss_name)

        # Populate feed dictionary with placeholders/values
        fd = {}  
        for key, value in self.feed_dict.items():
            placeholder = graph.get_tensor_by_name(key)
            fd[placeholder] = value

        return SessionRunArgs({"step": self._global_step, 
                               "loss": loss}, feed_dict=fd)

    else:
        return SessionRunArgs({"step": self._global_step})
# Check if current loss is below tolerance

def after_run(self, run_context, run_values):
    if self._step % self.stopping_step == 0:

        global_step = run_values.results["step"]
        current_loss = run_values.results["loss"]

        if current_loss < self.tolerance:
            run_context.request_stop()

        else:
            global_step = run_values.results["step"]

            self._step = global_step

- The monitored training session passes the values of fetches to the `run_values` argument of `after_run()` in a dictionary
- Stop requests are sent using: `run_context.request_stop()`
Creating an Early Stopping Loss*

Early stopping can also be performed without feed dictionaries by defining a separate validation dataset iterator for stopping checks:

```python
# Create early stopping batch from validation dataset
self.edataset = tf.data.TFRecordDataset("validation.tfrecords")
self.edataset = self.edataset.apply(
    tf.contrib.data.shuffle_and_repeat(10000))
self.edataset = self.edataset.apply(
    tf.contrib.data.map_and_batch(_parse_data, 20000)
self.eiterator = self.edataset.make_one_shot_iterator()

# Compute loss for early stopping checks
eloss = self.compute_loss(self.eiterator.get_next(),
    reuse=True, training=False,
    name="loss_stopping")
```

and adding a stopping loss operation to the graph:

```python
# Compute loss for early stopping checks
eloss = self.compute_loss(self.eiterator.get_next(),
    reuse=True, training=False,
    name="loss_stopping")
```

* It should be possible to use a dataset string handle instead
The EarlyStoppingHook can now be used in the same way as the predefined StopAtStepHook; in particular, it can be passed to the hooks list of the monitored training session once the loss name, stopping tolerance, and stopping step settings are specified:

```python
# Specify setting for EarlyStoppingHook
loss_name = "loss_stopping:0"
step = FLAGS.early_stopping_step
tol = FLAGS.early_stopping_tol

# Initialize TensorFlow monitored training session
with tf.train.MonitoredTrainingSession(
    hooks=[EarlyStoppingHook(loss_name, tolerance=tol, stopping_step=step)]) as sess:
```

A more complete version of EarlyStoppingHook is provided at: 
github.com/nw2190/TensorFlow_Examples/tree/master/Models
Additional examples can be found in the Models folder on GitHub:

https://github.com/nw2190/TensorFlow_Examples

Explanations of the code provided above are also available at:

https://www.math.purdue.edu/~nwinovic/tensorflow_sessions.html