

[Get started](#)[Open in app](#)[Follow](#)

523K Followers




An Advanced Example of the Tensorflow Estimator Class

With code and an in-depth look into some of the hidden features.

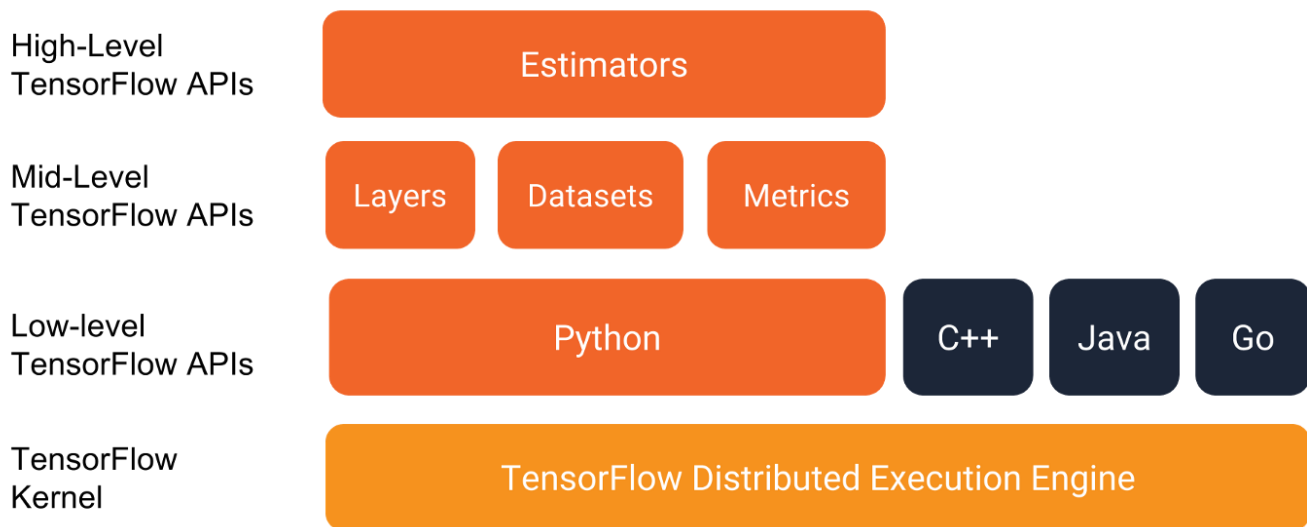


Tijmen Verhulsdonck Jul 27, 2018 · 13 min read

Estimators were introduced in version 1.3 of the Tensorflow API, and are used to abstract and simplify training, evaluation and prediction. If you haven't worked with Estimators before I suggest to start by reading this [article](#) and get some familiarity as I won't be covering all of the basics when using estimators. Instead I hope to demystify and clarify some aspects more detailed as  estimators and switching to Estimators from an existing code-base.

Why use Estimators?

Anyone who has been working with Tensorflow for a long time will know that it used to be a lot more time consuming to setup and use Tensorflow as it is today. Today there are numerous libraries that simplify development such as slim, tflearn and more. These often reduced the code needed to define a network from multiple files and classes to a single function. They also simplified managing the training and evaluation, and to a small extend data preparation and loading. The Estimator class of Tensorflow does not change anything about the network definition but it simplifies and abstracts managing training, evaluation and prediction. It stands out from the other libraries due to it's low level optimizations, useful abstractions and support from the core Tensorflow dev team.



Visualization of the different API levels

That is the long story, in short Estimators are faster to run and to implement, simpler (once you get used to them) and well supported.

Article structure

This article will break down some of the features of the Estimators, using examples from a GitHub project of mine [SqueezeNext-Tensorflow](#). The network implemented in the project is from a paper released in 2018 called “[SqueezeNext](#)”. This network is very lightweight and fast due to a novel approach to separable convolutions. The researchers released a [caffe](#) version but to facilitate experimentation using the available Tensorflow libraries I recreated the algorithm from the paper in Tensorflow.

The article is setup using the following structure:

- Setting Up The Estimator
- Data Loading with Estimators and Datasets
- Defining Prediction, Training and Evaluation modes
- Session hooks and scaffolds
- Prediction

Setting up the Estimator

In order to setup the Estimator for a certain training and evaluation roster it's good to first understand how the Estimator setup works. To construct an instance of the Estimator class you can use the following call:

```
classifier = tf.estimator.Estimator(model_dir=model_dir,
                                   model_fn=model_fn,
                                   params=params)
```

In this call “model_dir” is the path to the folder where the Estimator should store and load checkpoints and event files. The “model_fn” parameter is a function that consumes the features, labels, mode and params in the following order:

```
def model_fn(features, labels, mode, params):
```

The Estimator will always supply those parameters when it executes the model function for training, evaluation or prediction. The features parameter contains a dictionary of tensors with the features that you want to feed to your network, the labels parameter contains a dictionary of tensors with the labels you want to use for training. These two parameters are generated by the input fn which will be explained later. The third parameter mode describes whether the “model_fn” is being called for training, evaluation or prediction. The final parameter params is a simple dictionary that can contain python variables and such that can be used during network definition (think total steps for learning rate schedule etc.).

Train and evaluation example

Now that the Estimator object is initialized it can be used to start training and evaluating the network. Below is an excerpt of [train.py](#) which implements the above instructions to create an Estimator object after which it starts training and evaluating.

```
1      # calculate steps per epoch
2      steps_per_epoch = (args.num_examples_per_epoch / args.batch_size)
3
4      # setup config dictionary
5      config = configs[args.configuration]
6      config["model_dir"] = args.model_dir
```

```

7     config["output_train_images"] = args.output_train_images
8     config["total_steps"] = args.num_epochs * steps_per_epoch
9     config["model_dir"] = args.model_dir
10    config["fine_tune_ckpt"] = args.fine_tune_ckpt
11    # init model class
12    model = Model(config, args.batch_size)
13
14    # create classifier
15    classifier = tf.estimator.Estimator(
16        model_dir=args.model_dir,
17        model_fn=model.model_fn,
18        params=config)
19    tf.logging.info("Total steps = {}, num_epochs = {}, batch size = {}".format(config["
20                                                                    args.bat
21
22    # setup train spec
23    train_spec = tf.estimator.TrainSpec(input_fn=lambda: model.input_fn(args.training_fi
24                                        max_steps=config["total_steps"])
25
26    # setup eval spec evaluating ever n seconds
27    eval_spec = tf.estimator.EvalSpec(
28        input_fn=lambda: model.input_fn(args.validation_file_pattern, False),
29        steps=args.num_eval_examples / args.batch_size,
30        throttle_secs=args.eval_every_n_secs)
31
32    # run train and evaluate
33    tf.estimator.train_and_evaluate(classifier, train_spec, eval_spec)
34
35    classifier.evaluate(input_fn=lambda: model.input_fn(args.validation_file_pattern, Fa
36                      steps=args.num_eval_examples / args.batch_size)

```

train.py hosted with ❤ by GitHub

[view raw](#)

From: <https://github.com/Timen/squeezenext-tensorflow/blob/master/train.py>

This snippet of code first sets up the config dictionary for the params, and uses it together with the “model_fn” to construct an Estimator Object. It then creates a “TrainSpec” with an “input_fn” and the “max_steps” the model should train for. A similar thing is done to create the “EvalSpec” where the “steps” are the number of steps of each evaluation, and “throttle_secs” defines the interval in seconds between each evaluation. The “tf.estimator.train_and_evaluate” is used to start the training and evaluation roster

using the Estimator object, “*TrainSpec*” and “*EvalSpec*”. Finally an evaluation is explicitly called once more after the training finishes.

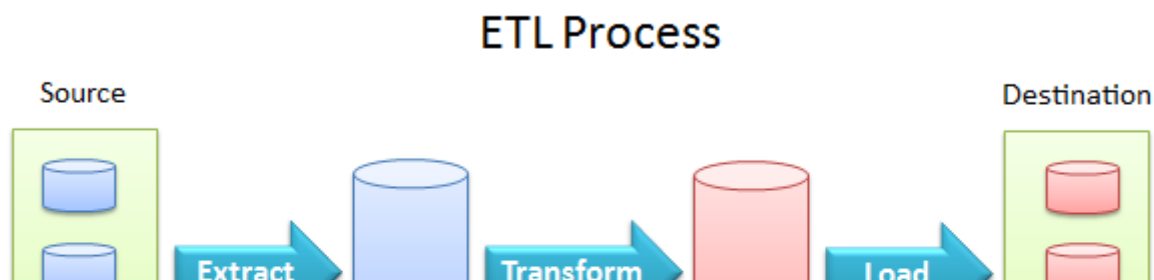
Data Loading with Estimators and Datasets

In my opinion the best things about Estimators, is that you can combine them easily with the Dataset class. Before the Estimator and Dataset class combination it was hard to prefetch and process examples on the CPU asynchronously from the GPU. Prefetching and processing on the CPU would in theory make sure there would be a batch of examples ready in memory any time the GPU was done processing the previous batch, in practice this was easier said than done. The problem with just assigning the fetch and process step to CPU is that unless it is done in parallel with the model processing on the GPU, the GPU still has to wait for the CPU to fetch the data from the storage disk and process it before it can start processing the next batch. For a long time queuerunners, asynchronous prefetching using python threads and other solutions were suggested, and in my experience none of them ever worked flawlessly and efficiently.

But using the Estimator class and combining it with the Dataset class is very easy, clean and works in parallel with the GPU. It allows the CPU to fetch, preprocess and queue batches of examples such that there is always a new batch ready for the GPU. Using this method I have seen the utilization of a GPU stay close to 100% during training and global steps per second for small models (<10MB) increase 4 fold.

Dataset Class

The Tensorflow Dataset class is designed as an E.T.L. process, which stands for Extract, Transform and Load. These steps will be defined soon, but this guide will only explain how to use tfrecords in combination with the Dataset class. For other formats (csv, numpy etc.) this page has a good write-up, however I suggest using tfrecords as they offer better performance and are easier to integrate with a Tensorflow development pipeline.





From: <http://blog.appliedinformaticsinc.com/etl-extract-transform-and-load-process-concept/>

The whole E.T.L. process can be implemented using the Dataset class in only 7 lines of code as shown below. It might look complicated at first but read on for a detailed explanation of each lines functionality.

```
1  with tf.name_scope("tf_record_reader"):
2      # generate file list
3      files = tf.data.Dataset.list_files(glob_pattern, shuffle=training)
4
5      # parallel fetch tfrecords dataset using the file list in parallel
6      dataset = files.apply(tf.contrib.data.parallel_interleave(
7          lambda filename: tf.data.TFRecordDataset(filename), cycle_length=threads
8
9      # shuffle and repeat examples for better randomness and allow training beyond
10     dataset = dataset.apply(tf.contrib.data.shuffle_and_repeat(32*self.batch_size)
11
12     # map the parse function to each example individually in threads*2 parallel
13     dataset = dataset.map(map_func=lambda example: _parse_function(example, self
14                           num_parallel_calls=threads)
15
16     # batch the examples
17     dataset = dataset.batch(batch_size=self.batch_size)
18
19     #prefetch batch
20     dataset = dataset.prefetch(buffer_size=self.batch_size)
21
22     return dataset.make_one_shot_iterator()
```

dataloader.py hosted with ❤ by GitHub

[view raw](#)

From: <https://github.com/Timen/squeezenext-tensorflow/blob/master/dataloader.py>

Extract

The first step in a Dataset input pipeline is to load the data from the tfrecords into memory. This starts with making a list of tfrecords available using a glob pattern e.g. “./Datasets/train-*.tfrecords” and the list_files function of the Dataset class. The

`parallel_interleave` function is applied to the list of files, which ensures parallel extraction of the data as explained [here](#). Finally a merged shuffle and repeat function is used to prefetch a certain number of examples from the tfrecords and shuffle them. The repeat ensures that there are always examples available by repeating from the start once the last example of every tfrecord is read.

```
files = tf.data.Dataset.list_files(glob_pattern, shuffle=True)

dataset = files.apply(tf.contrib.data.parallel_interleave(
    lambda filename:
        tf.data.TFRecordDataset(filename),
    cycle_length=threads*2
))

dataset = dataset.apply(tf.contrib.data.shuffle_and_repeat
    (32*self.batch_size))
```

Transform

Now that the data is available in memory the next step is to transform it, preferably into something that does not need any further processing in order to be fed to the neural network input. A call to the dataset's map function is required to do this as shown below, where “map_func” is the function applied to every individual example on the CPU and “num_parallel_calls” the number of parallel invocations of the “map_func” to use.

```
threads = multiprocessing.cpu_count()

dataset = dataset.map(map_func=lambda example:
    _parse_function(example, self.image_size,
        self.num_classes, training=training),
    num_parallel_calls=threads)
```

In this case the “map_func” is the parse function shown below, this function processes an example from the tfrecords (created using this [repo](#)) and outputs a tuple of dictionaries containing tensors representing the features and labels respectively. Note the use of a lambda function to pass python variables separately from the example to the parse

function, as the example is unparsed data from the tfrecord and provided by the Dataset class.

```
1  # Schema of fields to parse
2      schema = {
3          'image/encoded': tf.FixedLenFeature([], dtype=tf.string,
4                                              default_value=''),
5          'image/class/label': tf.FixedLenFeature([1], dtype=tf.int64,
6                                              default_value=-1),
7      }
8
9
10     image_size = tf.cast(image_size,tf.int32)
11     mean_value = tf.cast(tf.stack(mean_value),tf.float32)
12
13     # Parse example using schema
14     parsed_features = tf.parse_single_example(example_proto, schema)
15     jpeg_image = parsed_features["image/encoded"]
16     # generate correctly sized image using one of 2 methods
17     if method == "crop":
18         image = caffe_center_crop(jpeg_image,image_size,training)
19     elif method == "resize":
20         image = tf.image.decode_jpeg(jpeg_image)
21         image = tf.image.resize_images(image, [image_size, image_size])
22     else:
23         raise("unknown image process method")
24     # subtract mean
25     image = image - mean_value
26
27     # subtract 1 from class index as background class 0 is not used
28     label_idx = tf.cast(parsed_features['image/class/label'], dtype=tf.int32)-1
29
30     # create one hot vector
31     label_vec = tf.one_hot(label_idx, num_classes)
32
33     return {"image": tf.reshape(image,[image_size,image_size,3])}, {"class_idx": label_i
```

dataloader.py hosted with ❤ by GitHub

[view raw](#)

From: <https://github.com/Timen/squeezenext-tensorflow/blob/master/dataloader.py>

Keep in mind that this parse function only processes one example at the time as show by the `tf.parse_single_example` call, but does so a number of times in parallel. To prevent

running into any CPU bottlenecks it is important to keep the parse function fast, some tips on how to do this can be found [here](#). All the individually processed examples are then batched and ready for processing.

```
dataset = dataset.batch(batch_size=self.batch_size)
```

Load

The final step of the ETL process is loading the batched examples onto the accelerator (GPU) ready for processing. In the Dataset class this is achieved by prefetching, which is done by calling the prefetch function of the dataset.

```
dataset = dataset.prefetch(buffer_size=self.batch_size)
```

Prefetching uncouples the producer (Dataset object on CPU) from the consumer (GPU), this allows them to run in parallel for increased throughput.

Input Function

Once the whole E.T.L. process is fully defined and implemented, the “input_fn” can be created by initializing the iterator and grabbing the next example using the following line:

```
input_fn = dataset.make_one_shot_iterator().get_next()
```

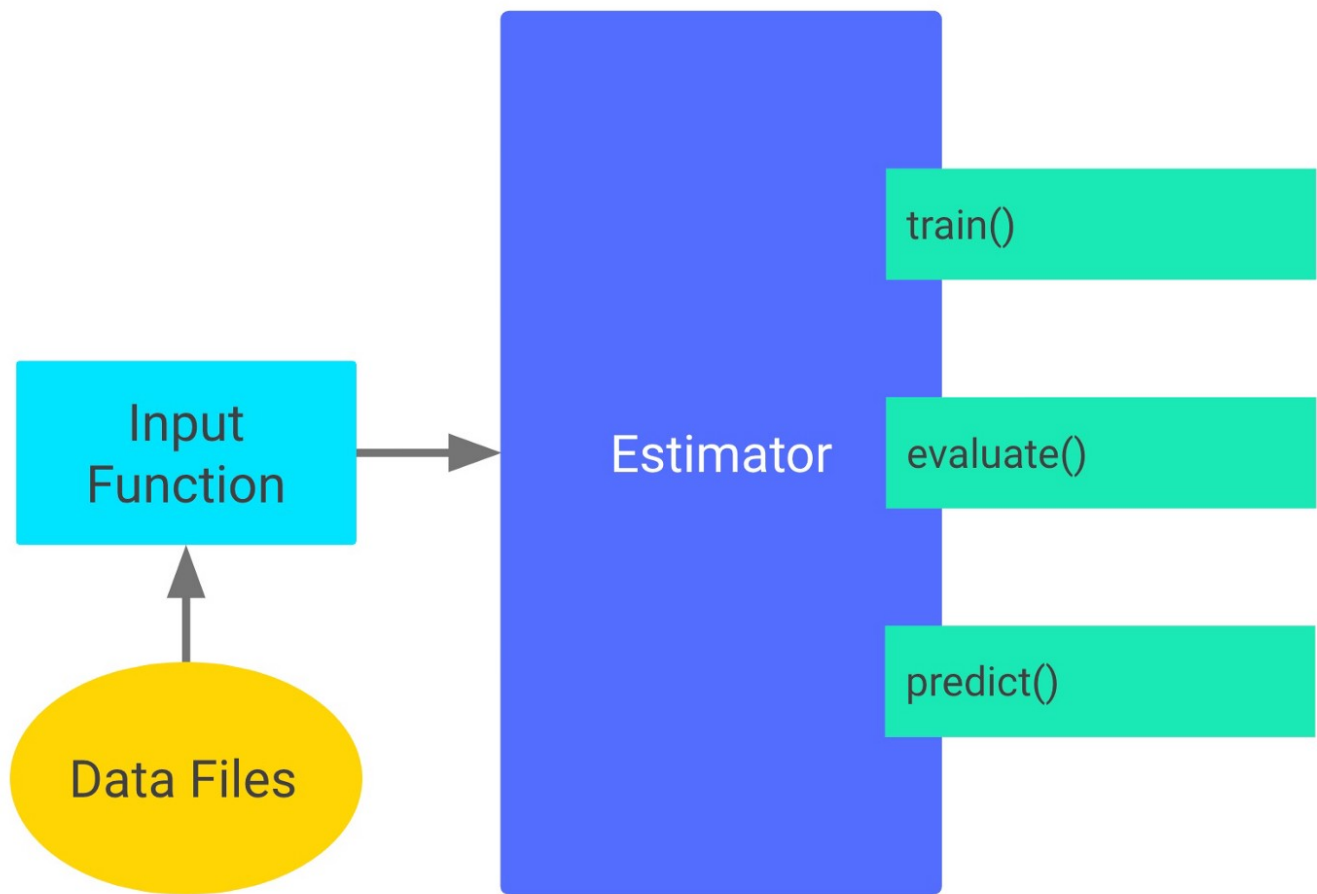
This input function is used by the Estimator as an input for the model function.

Defining Prediction, Training and Evaluation modes

A quick reminder, the model function the estimator invokes during training, evaluation and prediction, should accept the following arguments as explained earlier:

```
def model_fn(features, labels, mode, params):
```

The features and labels are in this case supplied by the Dataset class, the params are mostly for hyper parameters used during network initialization, but the mode (of type “tf.estimator.ModeKeys”) dictates what action the model is going to be performing. Each mode is used to setup the model for that specific purpose, the available modes are prediction, training and evaluation.



From: <https://k-d-w.org/blog/103/denoising-autoencoder-as-tensorflow-estimator>

The different modes can be used by calling the respective functions (shown above) of an Estimator Object namely; “predict”, “train” and “evaluate”. The code-path of every mode has to return an “EstimatorSpec” with the required fields for that mode, e.g. when the mode is predict, it has to return an “EstimatorSpec” that includes the predictions field:

```
return tf.estimator.EstimatorSpec(mode, predictions=predictions)
```

Predict

The most basic mode is the prediction mode “*tf.estimator.ModeKeys.PREDICT*”, which as the name suggests is used to do predictions on data using the Estimator object. In this mode the “EstimatorSpec” expects a dictionary of tensors which will be executed and the results of which will be made available as numpy values to python.

```
1      # output predictions
2      if mode == tf.estimator.ModeKeys.PREDICT:
3          _, top_5 = tf.nn.top_k(predictions, k=5)
4          predictions = {
5              'top_1': tf.argmax(predictions, -1),
6              'top_5': top_5,
7              'probabilities': tf.nn.softmax(predictions),
8              'logits': predictions,
9          }
10     return tf.estimator.EstimatorSpec(mode, predictions=predictions)
```

model.py hosted with ❤ by GitHub

[view raw](#)

From: https://github.com/Timen/squeezenext-tensorflow/blob/master/squeezenext_model.py

In this excerpt you can see the predictions dictionary setup to generate classical image classification results. The if statement ensures that this code path is only executed when the predict function of an Estimator object is executed. The dictionary of tensors is passed to the “EstimatorSpec” as the predictions argument together with the mode. It is smart to define the prediction code-path first as it is the simplest, and since most of the code is used for training and evaluation as-well it can show problems early on.

Train

To train a model in the “*tf.estimator.ModeKeys.TRAIN*” mode it is necessary to create a so called “train_op”, this op is a tensor that when executed performs the back propagation to update the model. Simply put it is the minimize function of an optimizer such as the AdamOptimizer. The “train_op” and the scalar loss tensor are the minimum required arguments to create an “EstimatorSpec” for training. Below you can see an example of this being done.

```
1      # Define train spec
2      if mode == tf.estimator.ModeKeys.TRAIN:
```

```

2         if mode == tf.estimator.ModeKeys.TRAIN:
3             # init poly optimizer
4             optimizer = PolyOptimizer(params)
5             # define train op
6             train_op = optimizer.optimize(loss, training, params["total_steps"])
7
8             # if params["output_train_images"] is true output images during training
9             if params["output_train_images"]:
10                 tf.summary.image("training", features["image"])
11                 scaffold = tf.train.Scaffold(init_op=None, init_fn=tools.fine_tune.init_weight)
12                 # create estimator training spec, which also outputs the model_stats of the
13                 return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op, training_input_producer=
14                     tools.stats._ModelStats("squeezenext", params["model_dir"],
15                                             features["image"].get_shape().as_list()[0])
16                     ],scaffold=scaffold)

```

model.py hosted with ❤ by GitHub

[view raw](#)

From: https://github.com/Timen/squeezenext-tensorflow/blob/master/squeezenext_model.py

Note the non-required arguments “training_hooks” and “scaffold”, these will be further explained later but in short they are used to add functionality to the setup and tear-down of a model and training session.

Evaluate

The final mode that needs a code-path in the model function is “*tf.estimator.ModeKeys.EVAL*”. The most important thing in order to perform an eval is the the metrics dictionary, this should be structured as a dictionary of tuples, where the first element of the tuple is a tensor containing the actual metric value and the second element is the tensor that updates the metric value. The update operation is necessary to ensure a reliable metric calculation over the whole validation set. Since it will often be impossible to evaluate the whole validation set in one batch, multiple batches have to be used. To prevent noise in the metric value due to per batch differences, the update operation is used to keep a running average (or gather all results) over all batches. This setup ensures the metric value is calculated over the whole validation set and not a single batch.

```

1         if mode == tf.estimator.ModeKeys.EVAL:
2             # Define the metrics:
3             metrics_dict = {

```

```

4         'Recall@1': tf.metrics.accuracy(tf.argmax(predictions, axis=-1), labels)
5         'Recall@5': metrics.streaming_sparse_recall_at_k(predictions, tf.cast(labels, tf.int64),
6                                                         5)
7     }
8     # output eval images
9     eval_summary_hook = tf.train.SummarySaverHook(
10         save_steps=100,
11         output_dir=os.path.join(params["model_dir"], "eval"),
12         summary_op=tf.summary.image("validation", features["image"]))
13
14     #return eval spec
15     return tf.estimator.EstimatorSpec(
16         mode, loss=loss, eval_metric_ops=metrics_dict,
17         evaluation_hooks=[eval_summary_hook])

```

model.py hosted with ❤ by GitHub

[view raw](#)

From: https://github.com/Timen/squeezenext-tensorflow/blob/master/squeezenext_model.py

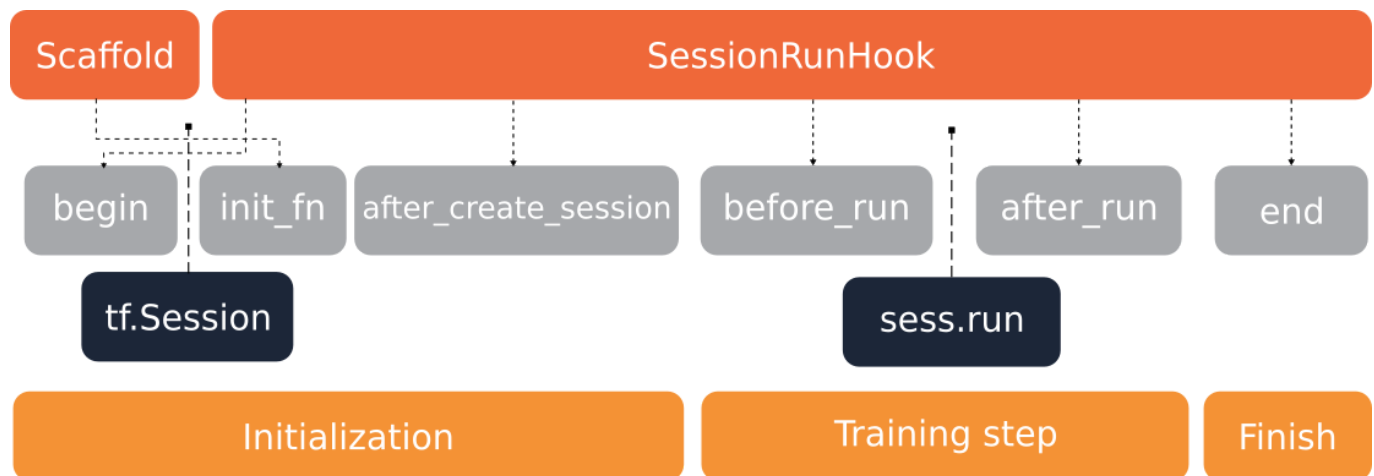
In the example above only the “loss” and “eval_metric_ops” are required arguments, the third argument “evaluation_hooks” is used to execute “tf.summary” operations as they are not automatically executed when running the Estimators evaluate function. In this example the “evaluation_hooks” are used to store images from the validation set to display using a Tensorboard. To achieve this a “SummarySaverHook” with the same output directory as the “model_dir” is initialized with a “tf.summary.image” operation and passed (encapsulated in an iterable) to the “EstimatorSpec”.

Switching to the Estimator class

Now that I explained the advantages of using the Estimator class and how to use it, I hope you are excited to start using Estimators for your Tensorflow projects. However switching to Estimators from an existing code-base is not necessarily straightforward. As the Estimator class abstracts away most of the initialization and execution during training, any custom initialization and execution loops can no longer be an implementation using “tf.Session” and “sess.run()”. This could be a reason for someone with a large code-base not to transition to the Estimator class, as there is no easy and straightforward transition path. While this article is not a transition guide, I will clarify some of the new procedures for initialization and execution. This will hopefully fill in some of the gaps left by the official [tutorials](#).

Scaffolds and SessionRunHooks

The main tools to influence the initialization and execution loop are the “Scaffold” object and the “SessionRunHook” object. The “Scaffold” is used for custom first time initialization of a model and can only be used to construct an “EstimatorSpec” in training mode. The “SessionRunHook” on the other hand can be used to construct an “EstimatorSpec” for each execution mode and is used each time train, evaluate or predict is called. Both the “Scaffold” and “SessionRunHook” provide certain functions that the Estimator class calls during use. Below you can see a timeline showing which function is called and when during the initialization and training process. This also shows that the Estimator under the hood still uses “*tf.Session*” and “*sess.run*”.



Function execution timeline executed by the Estimator of the Scaffold and SessionRunHook classes.

Scaffolds

A scaffold can be passed as the scaffold argument when constructing the “EstimatorSpec” for training. In a scaffold you can specify a number of different operations to be called during various stages of initialization. However for now we will focus on the “*init_fn*” argument as the others are more for distributed settings which this series will not cover. The “*init_fn*” is called after graph finalization but before the “*sess.run*” is called for the first time, it is only used once and thus perfect for custom variable initialization. A good example of this is if you want to finetune using a pretrained network and want to be selective about which variables to restore.

```
1 # look for checkpoint
2 model_path = tf.train.latest_checkpoint(path)
3 initializer_fn = None
4
5 if model_path:
```

```

5         if model_path:
6             # only restore variables in the scope_name scope
7             variables_to_restore = slim.get_variables_to_restore(include=[scope_name])
8             # Create the saver which will be used to restore the variables.
9             initializer_fn = slim.assign_from_checkpoint_fn(model_path, variables_to_restore)
10        else:
11            print("could not find the fine tune ckpt at {}".format(path))
12            exit()
13
14        def InitFn(scaffold, sess):
15            initializer_fn(sess)
16        return InitFn

```

`fine_tune.py` hosted with ❤ by GitHub

[view raw](#)

From: https://github.com/Timen/squeezenext-tensorflow/blob/master/tools/fine_tune.py

Above you can see an example of this implemented, the function checks for the existence of the fine-tune checkpoint and creates an initializer function using slim that filters on the “*scope_name*” variable. This function is then encapsulated in the function format that the Estimator expects which consumes the “Scaffold” object itself and a “*tf.Session*” object. Inside the “*init_fn*” the session is used to run the “*initializer_fn*”.

```

# setup fine tune scaffold
scaffold = tf.train.Scaffold(init_op=None,
                             init_fn=tools.fine_tune.init_weights(params["fine_tune_ckpt"]))

# create estimator training spec
return tf.estimator.EstimatorSpec(tf.estimator.ModeKeys.TRAIN,
                                   loss=loss,
                                   train_op=train_op, scaffold=scaffold)

```

This “*init_fn*” can then be passed as an argument to construct a “Scaffold” object, as show above. This “Scaffold” object is used to construct the “EstimatorSpec” for the train mode. While this is a relatively simple example (and can also be achieved with “WarmStartSettings”), it can be easily expanded for more advanced onetime initialization functionality.

SessionRunHooks

As shown in the timeline earlier the “SessionRunHook” class can be used to alter certain parts of the training, evaluation or prediction loop. This is done using a concept known as hooking, I will not go into too much detail on what hooking is exactly as in this case it is pretty self explanatory. But using a “SessionRunHook” is slightly different from using a scaffold. Instead of initializing the “SessionRunHook” object with one or more functions as arguments it is used as a super class. It is then possible to overwrite the methods “begin”, “after_create_session”, “before_run”, “after_run” and “end”, to extend functionality. Below an excerpt showing how to overwrite the “begin” function is shown.

```
1
2 class ModelStats(tf.train.SessionRunHook):
3     """Logs model stats to a csv."""
4
5     def __init__(self, scope_name, path, batch_size):
6         """
7         Set class variables
8         :param scope_name:
9             Used to filter for tensors which name contain that specific variable scope
10        :param path:
11            path to model dir
12        :param batch_size:
13            batch size during training
14        """
15        self.scope_name = scope_name
16        self.batch_size = batch_size
17        self.path = path
18
19    def begin(self):
20        """
21        Method to output statistics of the model to an easy to read csv, listing the
22        number of parameters, in the model dir.
23        :param session:
24            Tensorflow session
25        :param coord:
26            unused
27        """
28        # get graph and operations
29        graph = tf.get_default_graph()
30        operations = graph.get_operations()
31        # setup dictionaries
32        biases = defaultdict(lambda: None)
33        stat_dict = defaultdict(lambda: {"params":0, "maccs":0, "adds":0, "comps":0})
```


From: <https://github.com/Timen/squeezenext-tensorflow/blob/master/tools/stats.py>

Keep in mind that the “*begin*” function does not get any arguments but each hook is slightly different, please refer [here](#) to check what arguments are passed to which function. Now one can create a “SessionRunHook” with the extended begin method by calling:

```
stats_hook = tools.stats.ModelStats("squeezenext",
                                    params["model_dir"],
                                    self.batch_size)
```

This object can then be used when constructing an “EstimatorSpec” for training, evaluating and predicting by passing it to the respective arguments “training_hooks”, “evaluation_hooks” and “prediction_hooks”. Note that each arguments ends with hooks, and they expect an iterable of one or multiple hooks so always encapsulate your hooks in an iterable.

Prediction

Training and evaluation have extensively been covered, but prediction has not yet been fully explained. This is mostly because prediction is the easiest of the the 3 (train,eval,predict) Estimator modes, but there are still some pitfalls one can run into. I explained earlier how to setup an “EstimatorSpec” for prediction, but not how to use it. Below is a small example showing how to use an Estimator for prediction.

```
1  #subtract imagenet mean
2  mean_sub = image.astype(np.float32) - np.array([123, 117, 104]).astype(np.float32)
3  image = np.expand_dims(np.array(mean_sub), 0)
4  my_input_fn = tf.estimator.inputs.numpy_input_fn(
5      x={"image": image},
6      shuffle=False,
7      batch_size=1)
8
9  # setup synset lookup table for human readable labels
10 lookup_table = _build_synset_lookup(args.imagenet_metadata_file)
11 challenge_synsets = [l.strip() for l in
12                      tf.gfile.FastGFile(args.labels_file, 'r').readlines()]
```

```
--
13
14 # perform prediction
15 predictions = classifier.predict(input_fn=my_input_fn)
16
17 #use your predictions
```

predict.py hosted with ❤ by GitHub

[view raw](#)

From: <https://github.com/Timen/squeezenext-tensorflow/blob/master/predict.py>

For prediction it does not make sense to make/use tfrecords, so instead the “*numpy_input_fn*” is used. To the “*x*” argument a dictionary of numpy arrays is passed containing the features, in this case an image.

Note: Depending on whether or not you are preprocess images in the “input_fn” during training, you would need to perform the same preprocessing here, either in numpy before passing to the “numpy_input_fn” or in Tensorflow.

With the numpy input function setup, a simple call to predict is enough to create a prediction. The “*predictions*” variable in the example above will not actually contain the result yet, instead it is an object of the generator class, to get the actual result you need to iterate over the it. This iteration will start the Tensorflow execution and produce the actual result.

Final Remarks

I hope I was able to demystify some of the not so documented features of tensorflow Estimators and fill in the gaps left by the official tutorials. All the code in this post comes from this [github repo](#), so please head over there for an example of Estimators used in combination with the Datasets class. If you have any more questions please leave them here, I’ll try my best to answer them.

Sign up for The Daily Pick

By Towards Data Science

Hands-on real-world examples, research, tutorials, and cutting-edge techniques delivered Monday to Thursday. Make learning your daily ritual. [Take a look](#)

Your email

Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.

Machine Learning

TensorFlow

Estimators

Squeezenext

Software Engineering

[About](#) [Help](#) [Legal](#)

Get the Medium app

