Tijmen Verhulsdonck

Follow 78 Followers About

An Advanced Example of Tensorflow Estimators Part (2/3)

Note: this story used to be part of a series however all parts are now combined a single post.

Tijmen Verhulsdonck Jul 30, 2018 · 7 min read

This is the second part of a 3 part series showcasing a slightly advanced example of how to use Estimators in Tensorflow. In the first part the idea behind Estimators was introduced and the advantages of using them. A small example of how to setup a training and evaluation roster using the Tensorflow Estimator class was introduced and explained.

This part will go more in to depth on how to create an input function (“input_fn”) and the basics setup and implementation of a model function (“model_fn”). The input function will also introduce the use of the Dataset class to optimize the input pipeline, and achieve a greater throughput.

Data Loading with Estimators and Datasets

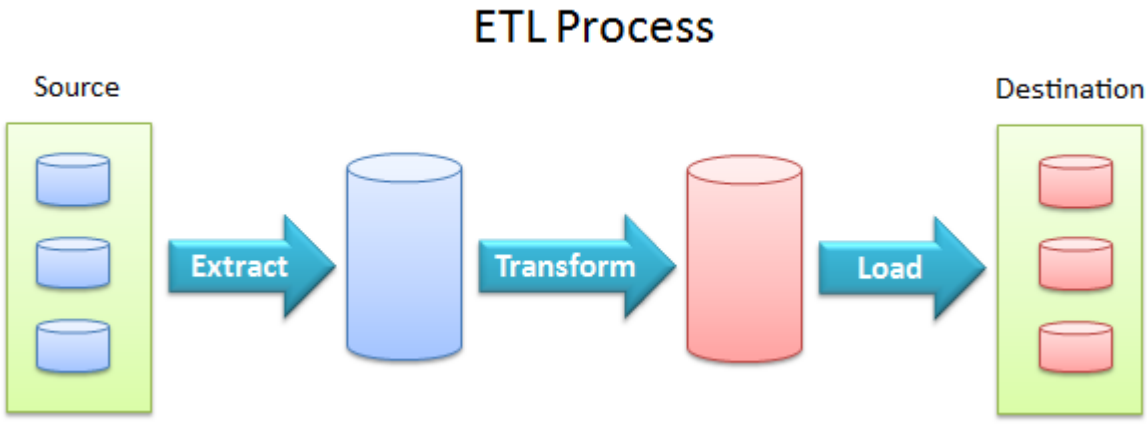
The last part glossed over the input function used during training and evaluation to generate the features and labels for the model function. However in my opinion this is one of the best things about Estimators, as you can combine them easily with the Dataset class.

Before the Estimator Dataset 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 then 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 que 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.

```
with tf.name_scope("tf_record_reader"):
            # generate file list
            files = tf.data.Dataset.list_files(glob_pattern, shuffle=training)

            # parallel fetch tfrecords dataset using the file list in parallel
            dataset = files.apply(tf.contrib.data.parallel_interleave(
                lambda filename: tf.data.TFRecordDataset(filename), cycle_length=threads))

            # shuffle and repeat examples for better randomness and allow training beyond one ep
            dataset = dataset.apply(tf.contrib.data.shuffle_and_repeat(32*self.batch_size))

            # map the parse  function to each example individually in threads*2 parallel calls
            dataset = dataset.map(map_func=lambda example: _parse_function(example, self.image_s
                                  num_parallel_calls=threads)

            # batch the examples
            dataset = dataset.batch(batch_size=self.batch_size)

            #prefetch batch
            dataset = dataset.prefetch(buffer_size=self.batch_size)

            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.

```
# Schema of fields to parse
    schema = {
```

```
        'image/encoded': tf.FixedLenFeature([], dtype=tf.string,
                                            default_value=''),
        'image/class/label': tf.FixedLenFeature([1], dtype=tf.int64,
                                                default_value=-1),
    }


    image_size = tf.cast(image_size,tf.int32)
    mean_value = tf.cast(tf.stack(mean_value),tf.float32)

    # Parse example using schema
    parsed_features = tf.parse_single_example(example_proto, schema)
    jpeg_image = parsed_features["image/encoded"]
    # generate correctly sized image using one of 2 methods
    if method == "crop":
        image = caffe_center_crop(jpeg_image,image_size,training)
    elif method == "resize":
        image = tf.image.decode_jpeg(jpeg_image)
        image = tf.image.resize_images(image, [image_size, image_size])
    else:
        raise("unknown image process method")
    # subtract mean
    image = image - mean_value

    # subtract 1 from class index as background class 0 is not used
    label_idx = tf.cast(parsed_features['image/class/label'], dtype=tf.int32)-1

    # create one hot vector
    label_vec = tf.one_hot(label_idx, num_classes)

    return {"image": tf.reshape(image,[image_size,image_size,3])}, {"class_idx": label_idx, "cla
```

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.

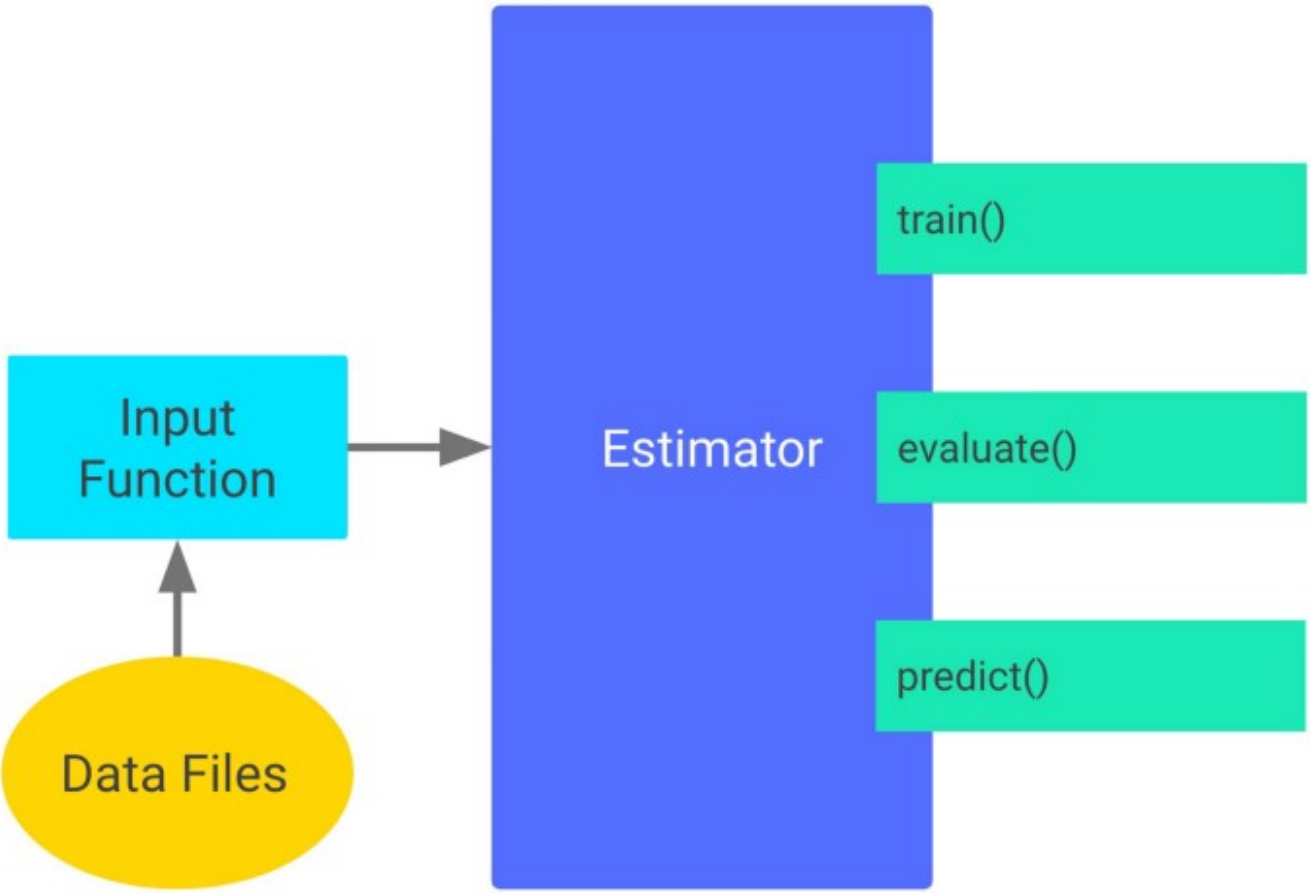
Defining Prediction,Training and Evaluation modes

The model function the estimator invokes during training, evaluation and prediction, should accept the following arguments as explained in part 1:

```
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.

```
        # output predictions
        if mode == tf.estimator.ModeKeys.PREDICT:
            _,top_5 =  tf.nn.top_k(predictions,k=5)
            predictions = {
                'top_1': tf.argmax(predictions, -1),
                'top_5': top_5,
                 'probabilities': tf.nn.softmax(predictions),
                'logits': predictions,
            }
            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.

```
        # Define train spec
        if mode == tf.estimator.ModeKeys.TRAIN:
            # init poly optimizer
            optimizer = PolyOptimizer(params)
            # define train op
            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_weights("squ
12              # create estimator training spec, which also outputs the model_stats of the model to
13              return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op, training_hooks
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 explained in Part 3 of this series 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["class_i
5                   'Recall@5': metrics.streaming_sparse_recall_at_k(predictions, tf.cast(labels["cl
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”.

Final Remarks

Implementing the model function and input function accounts for the largest part in setting up a Tensorflow Estimator. Using the Dataset class in the input function ensures maximum throughput by using an E.T.L. process, ensuring optimal accelerator usage. To fully support the prediction, training and evaluation in the Estimator the model function should contain code paths specific to each mode. All code paths share the code to setup the network but each returns a specific construction of an “EstimatorSpec” for that specific mode to function. In part 3 of this series the “SessionRunHooks” and “Scaffold” will be further explained together with some other miscellaneous things to complete the Estimator picture. Stay Tuned!

Part 3 can now be found here.

Machine Learning Dataset TensorFlow Software Engineering Etl