Project: tensorflow

2017-02-28T22:23:10Z mrry 134comments

[Enhancement] Redesigning TensorFlow's input pipelines

Web url: https://github.com/tensorflow/tensorflow/issues/7951

API url: https://api.github.com/repos/tensorflow/tensorflow/issues/7951

[\*\*TL;DR:\*\* We're designing a new input pipeline API for TensorFlow, and we'd like to collect your feature requests on this issue.]

We've noticed that one of the biggest challenges in getting started with TensorFlow is how to load your own data into your programs. While TensorFlow has several methods that can be used to build complex input pipelines (such as [`tf.train.string\_input\_producer()`](https://www.tensorflow.org/api\_docs/python/tf/train/string\_input\_producer), [`tf.train.batch()`](https://www.tensorflow.org/api\_docs/python/tf/train/batch), etc.), they were designed for a particular use case (processing a static set of files repeatedly), and the average user experience with these methods is not great. For example:

\* Once you reach the end of a pipeline, it becomes closed and you can never use it again in the same session. This requires users to use unnatural workarounds&mdash;with control flow or multiple sessions&mdash;to get a signal after processing an entire epoch, or switch between processing two datasets (e.g. training and validation data) in the same program.

\* See #2514 and #4535 for feature requests about handling multiple epochs.

\* See #7902 and numerous Stack Overflow questions for examples of processing different datasets in the same program.

\* The current pipelines use TensorFlow queues and multiple Python threads, which can lead to poor performance (lock contention in the queues and the Python GIL) and hard-to-understand exceptions (`tf.errors.OutOfRangeError`).

\* See #6845 for a discussion of input pipeline performance.

\* See #7525 and [many more Stack Overflow questions](http://stackoverflow.com/search?q=%5Btensorflow%5D+outofrangeerror) for an example of the confusing error.

\* The pipelines behave poorly if you forget to call `tf.train.start\_queue\_runners(sess)`: in fact, they hang indefinitely and deadlock the user program.

\* See #7945 and [many Stack Overflow questions](http://stackoverflow.com/search?q=%5Btensorflow%5D+hang) for some examples of users who have been bitten by this problem.

We're decided to start from a clean slate and redesign the input pipeline API. The existing methods will remain until TF 2.0 (at least), but we are planning to add a new set of methods for loading and manipulating datasets. We're still preparing a detailed design, which we plan to share soon, but we anticipate that there will be two new APIs:

\* A `Dataset` represents a collection of data elements. Each element can be a tuple of one or more tensors (e.g. an image and its label). We will provide methods for creating datasets from tensors, and deriving them from another dataset (e.g. by slicing its elements, repeating its elements, shuffling its elements, batching its elements, mapping a function over its elements, etc.).

\* An `Iterator` can be created from a `Dataset`. An iterator represents the current position within a dataset, and exposes an operation (like `tf.QueueBase.dequeue()`) that can be run to get the next element. There will be explicit operations for initializing an iterator, so that it can be reused after you have processed all of the elements in a dataset.

A similar pattern turns up in many different settings, including [Java's Stream API](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html), [Scala's collections](http://docs.scala-lang.org/overviews/collections/overview.html) (and hence Spark's RDDs), and [.NET's Language Integrated Query](https://msdn.microsoft.com/en-us/library/bb308959.aspx).

We're announcing this plan early because we want to collect feedback on what features you&mdash;as TensorFlow users&mdash;would like to see in an input pipeline API. What other pain points have we missed? What features do you miss from other systems? What other suggestions do you have?

We look forward to hearing from you!

-------------------------------------------------------------------------

2017-02-28T22:48:15Z kmhofmann

A must-have for one of our use cases is ad-hoc creation of data elements via a callback function (which creates tensors on the fly, e.g. using py\_func() or through some other means).

More specifically, we currently have a use case where we employ two queues; an outer one, using a string\_input\_producer (with shuffling), where each string denotes/points to a "data set", and the inner queue is then produced by generating a variable amount of samples from each "data set". Which and how many samples are generated differs per epoch (potentially conditional on past training behavior). Actually, we don't even use the nomenclature of an epoch anymore, since the same data is never seen twice, and above mentioned generation/sampling goes beyond the usual data augmentation.

Long story short: With a slightly out-of-the-ordinary use case, we've been hit by pretty much all of the problems you have mentioned above, and our workarounds have not been pretty. We'd be extremely happy to see a very flexible mechanism, where such cases are supported, and data generation doesn't have to be shoehorned into forced-upon concepts like epochs, finitely repeating queues, etc. (although they can be modeled by its primitives).

I am not sure how well the planned Dataset/Iterator API would support this.

Edit: Things we still need of course, include multi-threaded data generation, and multi-threaded random shuffle producer-consumer queues. But without the bane of GIL -- maybe via easy C++/Eigen hooks and thread control on the native side? Back and forth, via pybind?

Edit2: The new input pipeline should also take support for variable-sized tensors (i.e. different per example) into account, for both training and inference, e.g. in a fully-convolutional setting.

-------------------------------------------------------------------------

2017-02-28T23:37:57Z mrry

@kmhofmann We'll certainly support `tf.py\_func()` inside a new-style input pipeline (as well as, in general, compositions of any other TensorFlow ops). I'd like to understand more about your use case, though. How frequently do you move from one outer "data set" to the next? Are there any specific operations that you perform at the end of a "data set" or can your training loop handle the concatenation of records from different "data sets"?

We're planning to have a few nested iteration primitives, so that you can write a function mapping an element (e.g. a string representing your outer "data set") to a `Dataset` (representing the records in that "data set") and then flattening them down to a single `Dataset`. (Think `SelectMany()` in C#, `flatMap()` in Java and Scala.) So I think you could implement your logic for sampling from a "data set" in one of these `flatMap()` functions.

Let me know if any of this is unclear!

-------------------------------------------------------------------------

2017-03-01T00:04:45Z 22csnyder

Oh good timing! Now I can stop writing my own (horrible) dataset class. Many of the things said already resonate with my experience.

To the extent possible, I would like to code dataset-independent tensorflow computations. I don't want to have 3 different gan classes: each with their own create graph and fit methods, simply because one dataset doesn't fit in memory, the other is an np.array, and the other is generated on the fly.

The use case that affects me the most is [do n times: train for k iter/epoch, validate model, repeat]. There are clear problems with queues like you said. A minor issue for which I offer no solution is that while the scheduling(how long to train for before validate) is done perhaps by some method of a model class that I would like to be dataset-independent, whether it makes sense to talk in terms of iter or epoch is determined by the dataset--ruining some of the independence.

Some other ideas I jotted down while brainstorming my own class:

- The dataset class (and not the model) should probably the one to have batch\_size passed to it. It's awkward to ask for batch\_size as a parameter during fitting and during dataset/queue creation, and ideally the compute graph doesn't have batch\_size baked in.

- A "verbose" dataset class should keep track of it's own statistics. It should maintain its own counters(tensors) that keep track of iterations, samples, and epochs. In most use cases I imagine these being restored the same time model parameters are restored.

-------------------------------------------------------------------------

2017-03-01T12:17:05Z TimZaman

- Most importantly we need to address the dequeueing overhead. I've seen dozens and dozens of cases where (in the profiler; iirc MEMCpyWhatever) was really slow. This was mostly an issue where the GPU would get the data from the CPU.

- It would be great if there is still a way to have an \*\*input feed that comes from multi-threaded or multi-processing python\*\*. The following is a great and reliable hack to do currently do that:

`session.run(enqueue\_op, feed\_dict=$some\_numpy\_batch\_input)` Where you can asynchronously feed the queue from python.

- It would also be wow to have \*\*GPU resident queues\*\*.

-------------------------------------------------------------------------

2017-03-01T14:57:19Z inflation

Good point. One thing is that currently the queue operations are "baked in" the computation graph, so it's hard to modify anything on the go. A higher abstraction can make it much easier without considering using control flows or other hacks.

-------------------------------------------------------------------------

2017-03-02T11:06:23Z ppwwyyxx

For a lot of my use cases, my input data is either 1. not on the file system, or 2. require complex preprocessing unavailable in TensorFlow. For both cases the existing input pipeline cannot help, so I use an input thread with enqueue/feed\_dict + a training thread with dequeue a lot.

Let's \*\*assume\*\* that in most cases, you don't need to use the model itself to produce data (though sometimes it's not true). Then a solution I really like to see, is to be able to receive(similar to dequeue) tensors from a different process. (Like #4836)

The benefits are:

1. Can use whatever tools/languages to produce data from any sources, as long as they're finally sent with certain message protocol.

2. (theoretically) doesn't require an extra python thread in the training process.

3. If the message protocol supports pub/sub, then (1) multiple training sessions can subcribe and reuse the same input data, which is very useful when trying new models. (2) data can be generated from different machines if the pre-processing is too heavy for a single CPU.

These are the features I really missed from a private system I've been using.

One disadvantage is that IPC/socket has smaller bandwidth than RAM but usually it's not a bottleneck.

I know this feature may be too far away, but I hope the new design could allow such possible future feature.

-------------------------------------------------------------------------

2017-03-02T21:24:28Z kmhofmann

@mrry One "data set" can be composed of anything between ~500-30,000 dynamically generated samples. At the moment, we don't perform specific operations at the end of each data set, i.e. everything gets put into the same (large) random shuffle queue, to mix samples between data sets. But I could also imagine cases where separation of sets might be helpful.

-------------------------------------------------------------------------

2017-03-03T11:00:40Z yunjey

Please support reading hdf5 file directly.

-------------------------------------------------------------------------

2017-03-03T12:48:58Z untom

Personally, I'm a very big fan of the `feed\_dict` method of feeding data into the graph. It is by far the most flexible, makes debugging way easier and makes for much simpler code. Thus my biggest wish would be to make that method more performant. Right now, this method starves my GPU all the time, which is a shame because most other DL frameworks (even those based on computational graphs) manage to make this much more performantly. I assume there is more copying/handling going on in the background than would be necessary.

-------------------------------------------------------------------------

2017-03-08T08:48:49Z nicolasdespres

I am glad to see this initiative. The input pipeline is definitely the

steepest part of the learning curve.

I'd like:

\* A unified API to manage both in-memory (`feed\_dict`) dataset and large one so that the same code scale and your model only have to talk to one API. Although, I have not use it yet, I liked what I read in the input pipeline documentation.

\* More iterators! They are great. Asynchronous iterators would be even better (see [PEP492](https://www.python.org/dev/peps/pep-0492/)). Iterators implementing `\_\_len\_\_` are great for progress report.

\* multiprocessing rather than threading

\* Please no `Dataset` class because, IMHO, the "dataset" concept is ill-defined. The `Dataset` class described in the original post already exists in Python: it is a list of tuples. And what is a "dataset", anyway? A collection of train/valid/test data or simply a collection of data? Is it just a file? directory? generator? Are each data item (input/target) couple? Is that always true? Is the dictionary part of the text dataset?

The choice of the data container is driven by a lot of constrains depending on its size and the execution environment. Instead of a `Dataset` container, I would prefer to have a rich set of containers offering different trade-off with respect to memory/time complexity. In addition, I would like to have a rich set of iterators, splitters, loaders, dumpers, slicers, repeaters, servers, generators to actually work with data coming from various source.

\* The epoch concept does not have a clear semantic either. In my experience, it is best defined by `epoch = global\_step / steps\_per\_epoch` and `steps\_per\_epoch = dataset\_size / batch\_size`.

Here my attempt to translate to small in-memory dataset some of the routines available in the TF's input pipeline for large dataset. Here some examples of what I would like to see available in TensorFlow:

\* [cycle\_range](https://gist.github.com/nicolasdespres/fed915518cfcdb5c4dad73409eec77ea)

\* [shuffle\_iter](https://gist.github.com/nicolasdespres/e07aa951ba62aeaea60e740238571664)

\* [batch\_iter](https://gist.github.com/nicolasdespres/bbc62cb43f1ffe9b81d971cb957650c2)

\* [iter\_shuffle\_batch\_range](https://gist.github.com/nicolasdespres/d380cc43ebb1eb2d780b816fde294869)

\* [iter\_tensors\_slice](https://gist.github.com/nicolasdespres/418279e152dca606a78584fc99630738)

\* [iter\_shuffle\_batch\_tensors](https://gist.github.com/nicolasdespres/04dd05d0a8ebf66205a687f1706748dc)

\* [iter\_shuffle\_batch\_window](https://gist.github.com/nicolasdespres/81689421f56b86a315a81f19d301508a)

These routines demonstrate how far you can go with just simple iterators over list of indices.

-------------------------------------------------------------------------

2017-03-08T17:14:06Z yaroslavvb

+1 to something like feed\_dict. That's the only way to learn by interacting with external world (training robot arms, Atari games, [Universe](https://github.com/openai/universe) ).

It could be made more efficient by avoiding copies. Like PyTorch, whose Tensors share memory buffers with underlying numpy arrays

-------------------------------------------------------------------------

2017-03-08T17:51:45Z MicaelCarvalho

I don't know TF as well as others here, so please take my comments with some skepticism:

- With `tf.py\_func` I was able to solve most of my input-related problems, like loading .mat files in a symbolic-ish manner. The one I'm currently struggling with is the integration of `tf.train.batch` with the ability of picking the source from which the input should come, for having train/val data in the same symbolic variable. #8168

I understand these functions were initially thought for simple use cases, but it would be nice to have more control of the pipeline without the burden of managing \_everything\_ (e.g. using `tf.QueueBase.from\_list` but being forced to feed queues and manage threads kind of manually).

- I'm not sure if TensorFlow optimizes the dequeue operation under the hood but, if not, I think we could greatly benefit from a parallel dequeue operation that charges data (i.e. next batch) into the GPU memory while it processes the previous data (i.e. current batch).

- I think `feed\_dict`-like solutions are not optimal for passing big chunks of data to the train function, like a batch of images, since they're basically a pause in the execution graph to force TF to interact with \_python's runtime\_. An in-graph solution sounds better, with pointers to guide the graph execution, like `feed\_dict={is\_training = True}` to indicate the input should come from the training pipeline, the model should set batchnorm and dropout (et al) to train mode etc. This way, TF could better optimize/parallelize the execution, and all solutions would scale.

- The standard functions for creating batches apparently do not provide an index to indicate which batch we are processing. For example, a `slice\_input\_producer` receives the number of epochs to be generated but there seems to be no way of knowing the epoch of one sample without counting how many we have already evaluated.

-------------------------------------------------------------------------

2017-03-08T20:40:52Z ErikGoldman

right now there are two very divergent paths to getting data into Tensorflow: feed\_dict and queues. queues are wonderful until you don't have a way to manipulate your data natively -- for example, if you want to load a .wav file, chop it into parts, and convert it to a spectrogram. at that point, you have to write a C++ op (doable, but a context switch + it makes a very inflexible pipeline) or pop back into Python land (slower, but very easy and flexible).

it seems like the best compromise between speed and flexibility is to create a TF queue and then make a bunch of Python threads that feed it with data. this allows you to do flexible data processing in Python (roughly parallelized on the CPU, apart from GIL issues) while maintaining some amount of speed benefit.

what if you just formalized that? the interface would be: push\_data, end\_of\_data (for signaling the end of an epoch), and a dequeue\_batch function that feeds the model. then your code could just load data in Python and stuff it onto the queue in parallel, while the model sits totally separate from all of that.

-------------------------------------------------------------------------

2017-03-09T20:35:25Z jhseu

We should make feed\_dict faster (likely by not copying the numpy.arrays like @yaroslavvb mentioned), but that's orthogonal to this change. No matter how much we optimize it, feed\_dict will never be the fastest way to feed data into a training job.

-------------------------------------------------------------------------

2017-03-09T21:33:57Z yaroslavvb

`feed\_dict` specifically may not be essential. To be more precise, we need support for pipelines where learning is done in an online fashion, and training data is generated by a system responding to actions of a TensorFlow network (learning Atari simulator, robotics simulator, robot interacting with real world, etc). This is necessary for most of the applications at OpenAI, here's one example -- https://github.com/openai/universe-starter-agent

-------------------------------------------------------------------------

2017-03-09T22:03:39Z jhseu

The fastest option would be to create a TensorFlow op that maintains state, takes actions as input, and generates the training data. Then add a placeholder to specify the action.

My guess is that you're looking for something that can be done completely in Python, though. There may be some mid-point between the two.

-------------------------------------------------------------------------

2017-03-09T22:55:01Z Mazecreator

I am not sure it this concept has been brought up yet, but I will at least put the problem in my own terms.

In dealing with RL problems and the training replay buffer, I couldn't find an easy way to use the Queues to speed up this feeding of samples through the feed\_dict. Also, when randomly creating a sample set, it seemed like the samples were consumed when I wanted them left in the buffer.

What I was hoping to do is feed (possibly through feed\_dict, or file) a Queue with a new sample and once the size of the buffer is exceeded, the oldest sample is removed from the buffer. So some concept of "sample age" would be nice. I am sure using a circular buffer will work to fix to a number of samples, but "age" might be of interest as well, maybe passed as part of the tuple, but in the RL case, simply the sequence of the sample being added might cover the age (FIFO).

Again, it may have just not been clear to me how to use the queues, but being able to randomly pull a mini-batch from this sample buffer and not remove the samples so a new set of samples can be collected (possibly with prior sampled examples) would be nice.

-------------------------------------------------------------------------

2017-03-10T01:51:56Z lming

I may not understand the distributed settings that TF data input pipeline API is targeting to solve. Is it possible to have a simple API design as Pytorch does: only three simple classes. I can pick up pytorch's dataset API in 5 minutes and it's good enough for all the popular academic datasets. http://pytorch.org/docs/data.html

It's great to see new efforts to solve the pain points in TF dataset API. Looking forward to a simple/beautiful/flexible API with minimum number of classes/concepts introduced. Thanks.

-------------------------------------------------------------------------

2017-03-10T02:04:28Z jhseu

@lming Yeah, the first two comments here cover that: by making a Dataset implementation that uses [py\_func](https://www.tensorflow.org/api\_docs/python/tf/py\_func), it'd be equivalent to the PyTorch implementation.

-------------------------------------------------------------------------

2017-03-10T02:06:20Z taion

I second @lming's sentiment above.

Our biggest issue with the current data loading scheme is just that it's very complicated and involves a lot of new concepts.

We don't find it spectacularly difficult to write a multithreaded data loader ourselves in Python, and generally we don't find it overly difficult to ensure that our data loading and preprocessing runs sufficiently quickly that it doesn't actually bottleneck training.

Where we're stuck is that to optimally follow recommendations, we end up in an awkward situation, one of:

- Using `feed\_dict` and suffering any relevant performance hits

- Feeding from a separate thread and dealing with some one-off queue boilerplate (except this didn't speed things up at all when we tried it)

- Reimplementing our data loading and transformation pipeline with TF primitives, perhaps with `py\_func`, but still using the TF API for managing queue runners

The Python threading API isn't perfect, but in general when we're doing mostly non-GIL-taking tasks in NumPy or whatever, the TF queue API seems more of a burden than a help.

-------------------------------------------------------------------------

2017-03-10T02:18:43Z taion

A couple of concrete use cases that come up for us:

- One of our models is a localization model. We use scikit-image `AffineTransform` objects to apply cropping and resizing operations for this model, because those objects let us easily translate our model output back to the original input coordinate space. It seems tricky to square this with a fully tensor-based API.

- We have data that comes from a large number of imbalanced segments, and in training we use some custom stratified sampling logic to ensure we present examples from each segment evenly. It's relatively straightforward for us to, in Python, generate a new draw from our original data set for every epoch, but it seems like in the API proposed above, we'd have to figure out how to implement this behavior as an `Iterator`, which seems less straightforward.

-------------------------------------------------------------------------

2017-03-14T15:03:01Z lw394

We primarily deal with time series data, and prefer to not have to batch preprocess the whole dataset prior to training every unique model input architecture. In fact, the preprocessed dataset size for one input architecture variant can be easily an order of magnitude larger than the unprocessed file set size.

We've been able to deal with this using the current system of queues,TF ops, batch\_join/etc to enable multiple on-the-fly preprocessing threads with across-file example mixing. I have to say it's really nice and flexible for hyperparameter tuning the input architecture, and I like that the entire pipeline lives in the graph, feeds data in response to sess.run(train\_op) calls, and can be restored from a common checkpoint with the model.

If you're planning to deprecate the current queues paradigm, I would like to know that the `Dataset` and `Iterator` would enable the same flexibility. For my use case it seems like `Dataset` could represent a collection of time series, and the `Iterator` would behave like a python iterator/generator and could handle any preprocessing to form batches of examples?

-------------------------------------------------------------------------

2017-03-15T11:09:54Z krisjobs

feature request: control mechanism for queues, especially in combination with TFRecordReader's/TextFileReader's read() method, which automatically dequeues. Reason: the automatic preevaluation of pending enqueues.

-------------------------------------------------------------------------

2017-03-20T16:21:29Z futurely

MXNet [IIterator](https://github.com/dmlc/mxnet/search?utf8=%E2%9C%93&q=IIterator&type=Code) is a more relevant example than Java's Stream API, Scala's collections (and hence Spark's RDDs), and .NET's Language Integrated Query. The design enables flexible composition of various components of the input pipeline such as [ImageRecordIter](https://github.com/dmlc/mxnet/blob/d8e64a67f1b5b8eff287a5305e7a86714d3b1aeb/src/io/iter\_image\_recordio.cc), [ImageNormalizeIter](https://github.com/dmlc/mxnet/search?utf8=%E2%9C%93&q=ImageNormalizeIter), [BatchLoader](https://github.com/dmlc/mxnet/search?utf8=%E2%9C%93&q=BatchLoader&type=Code), [PrefetcherIter](https://github.com/dmlc/mxnet/search?utf8=%E2%9C%93&q=PrefetcherIter&type=Code) and [ImageAugmenter](https://github.com/dmlc/mxnet/search?utf8=%E2%9C%93&q=aug\_default&type=Code).

```cpp

MXNET\_REGISTER\_IO\_ITER(ImageRecordIter)

.describe("Create iterator for dataset packed in recordio.")

.add\_arguments(ImageRecParserParam::\_\_FIELDS\_\_())

.add\_arguments(ImageRecordParam::\_\_FIELDS\_\_())

.add\_arguments(BatchParam::\_\_FIELDS\_\_())

.add\_arguments(PrefetcherParam::\_\_FIELDS\_\_())

.add\_arguments(ListDefaultAugParams())

.add\_arguments(ImageNormalizeParam::\_\_FIELDS\_\_())

.set\_body([]() {

return new PrefetcherIter(

new BatchLoader(

new ImageNormalizeIter(

new ImageRecordIter<real\_t>())));

});

```

Caffe [DB](https://github.com/BVLC/caffe/search?l=C%2B%2B&p=1&q=db&utf8=%E2%9C%93) is simpler but still usable.

Ideally, the newly designed API should be able to load existing datasets of Caffe & MXNet with easy to implement [plugins](https://github.com/dmlc/mxnet/search?utf8=%E2%9C%93&q=Caffe+Iter&type=Code).

-------------------------------------------------------------------------

2017-04-12T10:06:59Z petrux

Just my 2 cents. Happy for this decision. I think that a \*huge\* effort should be placed in tutorials: the hugest difficulty I am having -- and some colleagues with me -- is that the documentation that you can find is quite lousy and not very self-contained. I would be happy to help, of course.

-------------------------------------------------------------------------

2017-04-14T02:33:27Z danijar

> A Dataset represents a collection of data elements.

I'm not sure if this is implied but a requirement would be to stream large datasets from disk. A `Dataset` explicitly containing all data elements doesn't seem to support that.

-------------------------------------------------------------------------

2017-04-20T14:40:26Z kratzert

In the last release notes I see that you have added a new RecordInput class, which seems to be the new class intended to use as input provider? Unfortunately the documentation is still lacking further explanations. I can only find some basic infos in the C++ API docs.

Would be really interested to read something for the Python API + some example code. If you need any help, feel free to contact me or e.g. @petrux also offered help. I think he is right, that extending the documentation and providing better tutorials is highly important. Because otherwise the people will stick with feed\_dict inputs until TensorFlow 3.0 and moan about bad performance of TF

-------------------------------------------------------------------------

2017-04-20T16:30:59Z taion

I ended up doing some benchmarking for other reasons, and observed comparable performance between `feed\_dict` and using queues: https://github.com/tensorflow/tensorflow/issues/9322#issuecomment-295775991

-------------------------------------------------------------------------

2017-04-20T17:25:21Z yaroslavvb

Feed dict overhead is essentially the cost of doing an extra memcpy (Python->TensorFlow CPU->TensorFlow GPU) vs. using native ops like queues which do (TensorFlow CPU->TensorFlow GPU). So if this memcpy is small, there should be negligible.

-------------------------------------------------------------------------

2017-04-20T17:50:35Z taion

That makes sense, and it's what I was assuming.

I think that makes the advice against `feed\_dict` a bit overblown, though ‚Äì really the issue seems more like inefficient data feeding that starves the GPUs, rather than the use of `feed\_dict` itself.

-------------------------------------------------------------------------

2017-04-21T13:16:38Z kratzert

@yaroslavvb correct me if I'm wrong but this isn't entirely right. Unless you haven't implemented some input pipeline using Pythons Queue library or something similar it will be additionally the time of loading data from disk into memory and eventually preprocess them.

For Images and especially larger batch size, this might take quite a while. Here is where you can really speed things up using TFs input queues because they will load e.g. images into disk ( + preprocess ) on CPU while you are training/evaluating your network on GPU. When computations are done, you can directly grab the next batch on copy the data on the GPU, without waiting for native Python to load new data into memory.

-------------------------------------------------------------------------

2017-04-21T14:27:29Z carlthome

@kratzert, that's precisely what @taion means by

> the issue seems more like inefficient data feeding that starves the GPUs

It's hard to do asynchronous preprocessing well, so most users benefit from `tf.train` doing it for them.

-------------------------------------------------------------------------

2017-04-21T14:41:43Z kratzert

Is it? By no means I want to defend TFs input queues, but as I read the post of @yaroslavvb he states that the additional time comes (only) from passing memory between native Python and TF ( + GPU). The only thing I wanted to add is, that if you can't store all your trainings data in memory, also loading memory from disk into memory adds up time to one training cycle. I can't find any of this statement in the post of @taion, but could be a misunderstanding of my side as I'm not a English native speaker.

And I know, that for many cases asynchronous preprocessing is not possible, but for the cases it is (simple training of image classification CNN) TFs input queues help quite a lot.

-------------------------------------------------------------------------

2017-04-21T15:00:34Z carlthome

> The only thing I wanted to add is, that if you can't store all your trainings data in memory, also loading memory from disk into memory adds up time to one training cycle.

That also applies to `tf.train`. Data doesn't just appear in the graph without reading it from disk, regardless of whether you do it via Python or TensorFlow's execution engine.

Anyway, I find memapped NumPy arrays via [joblib.cache.Memory](https://pythonhosted.org/joblib/memory.html) and feed\_dict to be quite performant (GPU load over 90% throughout a training session), despite the extra copy.

-------------------------------------------------------------------------

2017-04-21T15:13:08Z kratzert

Ah okay so maybe a misunderstanding. Thought I made myself clearer.

I know that data does not appear magically in memory if I use TF. But TF makes it quite easy to place e.g. image loading + preprocessing explicitly on CPU and graph computation on GPU and both is done in parallel. So while GPU calculates ops on some data, CPU is already loading the next data into memory. And since this is done in parallel, we are effectively reducing the computational time by the amount of loading data from disk (since this takes usually less time then one forward + backward pass through the networks graph). But yes this only applies for working with CPU + GPU and has no effect if you use CPU only.

edit: The maybe only thing I like of TFs input queues is that I the state of the queues (and batch producers) can be observed in Tensorboard. For the rest I fought quite some time to get them running with all the preprocessing I wanted and with queues for testing and validation in the same run.

-------------------------------------------------------------------------

2017-04-24T09:15:13Z thejonan

I would say that aside from the steep learning curve of input pipeline which can be overcome with documentation too, the key missing points are:

1. A decent way to make custom preprocessing of the data, whether it be based on queue's or not, the idea of being able to foresee all possible data input needs is doomed.

2. An easy and established way to change the input pipeline of a graph, after it has been created, because it is the most typical usage pattern. Current `input\_map` provides such functionality, but it is rather hacky. And documented too (when it comes to `import\_meta\_graph`).

3. A way to control and monitor the epochs - currently they are rather deeply hidden and unaccessible even simple checks.

I'd support @nicolasdespres for "no Dataset" pledge, mainly because all-in-one bundle is not flexible, not future proof and also - not consistent with the TF's paradigm of providing small, stable, well-defined and assemble-able blocks for building custom models. Having some bundles, predefined "easy-starter" wrappers should be welcomed.

-------------------------------------------------------------------------

2017-04-24T09:52:40Z PatWie

I think TF does not really require another attempt to unify the data pre-processing to put it directly \*into\* the graph. Things get worse if one need custom stuff and on-the-fly generation/modification of data. Typically these modifications are not part of the forward model for a good reason: these operations do not require any backpropagation. Hence, they should be only loosely coupled.

So the ideal input pipeline (everything without backprop) should be quite simple and slim: It should consist of a queue operation which receives data (list of tensors) from \*\*some\*\* source (sockets). #8728 is a good step in this direction with the pros:

- you can use any library for preprocessing (opencv, nltk, ...)

- prefetching is totally parallel and can be done at any/multiple machine

- the sender-code can be put into any place -- even directly in game-engines or render-engines

- data generation can be done in any programming language (without custom ops)

I am not sure, if you really need something else and I do not understand why you really need `tf.image.random\_\*` \*in\* the graph.

-------------------------------------------------------------------------

2017-04-24T15:20:50Z taion

I don't think the proposal here is to get rid of queues entirely, is it?

Dataset-style abstractions are pretty common in this space, and they're quite useful. The existence of a higher-level abstraction doesn't preclude the lower-level API from also existing.

In fact for these kinds of higher-level abstractions, sooner seems better than later ‚Äì one of the greatest frustrations of reading published TF research code is that the vast majority of codebases use their own idiosyncratic layers library, as opposed to e.g. the ones in `tf.layers` or `tf.contrib.layers`, and these libraries are all different, which makes it more difficult than it should be to share work.

-------------------------------------------------------------------------

2017-04-27T20:28:20Z eamartin

I often try to use TensorFlow on very large inputs (potentially >1GB minibatch) with relatively light computation on each minibatch. These inputs are in a HDF5 file or a Numpy array either on disk or in memory, so I typically feed with ``feed\_dict``, potentially asynchronously into a queue. When running with multiple GPUs, TensorFlow is not able to even saturate the PCI-e bandwidth to the GPUs because of the memcpy from the feed\_dict to the CPU tensor.

As @yaroslavvb mentioned, the ``feed\_dict`` memcpy (on a single CPU core?) can be a huge performance bottleneck, and I'd like to see this addressed in any refactor of TensorFlow's input handling.

@jhseu You mentioned that you consider removing the ``feed\_dict`` copy as orthogonal to this issue. Do you know if there's any issue or work being done on removing the copy (at least in some cases, like row-major Numpy arrays with nice strides)?

-------------------------------------------------------------------------

2017-04-28T23:36:28Z vrv

@eamartin there have been a number of changes since the beginning of March by @alextp to speed up feed\_dict; when the memory is aligned with 16 bytes, I think we share buffers with numpy, so nightly releases may be faster for you.

The 16 byte alignment issue comes from Eigen, unfortunately, which requires the beginning of the memory addresses to be aligned with 16 bytes. I'm not sure why Eigen was not written to handle unaligned first and last "packets" so it wouldn't matter. :/

-------------------------------------------------------------------------

2017-04-29T07:23:43Z sjperkins

Would it be possible for numpy to share buffers with tensorflow variables

when they are returned from a session run?

I realise this will probably raise all sorts of mutability and state

issues, but these should be avoidable by setting the WRITABLE flag on the

returned numpy arrays to false.

On 29 Apr 2017 1:37 am, "Vijay Vasudevan" <notifications@github.com> wrote:

@eamartin <https://github.com/eamartin> there have been a number of changes

since the beginning of March by @alextp <https://github.com/alextp> to

speed up feed\_dict; when the memory is aligned with 16 bytes, I think we

share buffers with numpy, so nightly releases may be faster for you.

The 16 byte alignment issue comes from Eigen, unfortunately, which requires

the beginning of the memory addresses to be aligned with 16 bytes. I'm not

sure why Eigen was not written to pad the first and last "packets" so it

wouldn't matter. :/

‚Äî

You are receiving this because you are subscribed to this thread.

Reply to this email directly, view it on GitHub

<https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-298129805>,

or mute the thread

<https://github.com/notifications/unsubscribe-auth/ADXd5AD4yuTBPXo8pp-uvVbr9QyYnJhDks5r0nhMgaJpZM4MO9tN>

.

-------------------------------------------------------------------------

2017-04-29T19:32:13Z eamartin

Thanks for the info @vrv (and for the features @alextp ). I did a little looking around, and it looks like https://github.com/tensorflow/tensorflow/commits?author=alextp&since=2017-03-01T06:00:00Z&until=2017-04-01T05:00:00Z are the relevant commits. From my checking, these didn't make it into TF 1.1 but hopefully will be in 1.2.

-------------------------------------------------------------------------

2017-04-30T16:36:23Z sjperkins

Another thing that would be cool would be the ability for Session's to return Futures that could then be used as input to other Session runs. Said future could then be passed through the graph until the Tensor it represents requires evaluation.

```python

with tf.Session() as S:

# Note executor semantics

future = S.submit(op\_1, feed\_dict={'input\_1': 1.0, 'input\_2':2.0})

result = S.run(op\_2, feed\_dict={'data': future})

```

This concept is inspired by [dask distributed](https://github.com/dask/distributed) and other executor frameworks -- I think the flexibility offered by this abstraction is great!

-------------------------------------------------------------------------

2017-05-01T17:25:54Z alextp

@sjperkins sadly the way our current unit tests are written makes them break if variables are returned without extra copies (because many tests do a = session.run(variable); session.run(update\_variable); b = session.run(variable); assertDifferent(a, b) (which fails if they share buffers).

I considered making a ConfigProto option to share buffers even when they are not exclusively owned by the C-python bridge but didn't. Should be easy to do from the commits listed above, if you're interested.

-------------------------------------------------------------------------

2017-05-01T17:26:17Z alextp

@sjperkins for the futures, stay tuned, as we're prototyping something in this direction.

-------------------------------------------------------------------------

2017-05-01T21:52:45Z drcrook1

My primary request is that however you build the new input pipeline system that it should be completely separate from the rest of the graph. I'm giving a shot at migrating to tensorflow for our deep learning models for devices; but the input pipeline is so tightly bound to the rest of the compute graph that its like performing surgery to run inference against it.

Example: I trained using TFRecords and input queues; I got my weights/model. I want to perform inference by running my prediction operation; but because the input queue runners etc are part of the graph before that; I am stuck with that mechanism for performing inference.

See issue here: http://stackoverflow.com/questions/43708616/tensorflow-inference

I like the tf record and queue runner thing now that I'm used to it; the issue is the tight binding to the graph....

-------------------------------------------------------------------------

2017-05-01T21:56:26Z alextp

This is why tools like tf.estimator.Estimator were developed, to allow for

easier separation of concerns between training and inference and to allow

for swapping of input pipelines. Can you use Estimator to write your model?

On Mon, May 1, 2017 at 2:54 PM, David Crook <notifications@github.com>

wrote:

> My primary request is that however you build the new input pipeline system

> that it should be completely separate from the rest of the graph. I'm

> giving a shot at migrating to tensorflow for our deep learning models for

> devices; but the input pipeline is so tightly bound to the rest of the

> compute graph that its like performing surgery to run inference against it.

>

> Example: I trained using TFRecords and input queues; I got my

> weights/model. I want to perform inference by running my prediction

> operation; but because the input queue runners etc are part of the graph

> before that; I am stuck with that mechanism for performing inference.

>

> See issue here: http://stackoverflow.com/questions/43708616/tensorflow-

> inference

>

> I like the tf record and queue runner thing now that I'm used to it; the

> issue is the tight binding to the graph....

>

> ‚Äî

> You are receiving this because you were mentioned.

> Reply to this email directly, view it on GitHub

> <https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-298442742>,

> or mute the thread

> <https://github.com/notifications/unsubscribe-auth/AAATxXYK7comPG84UqyKek2mbzhnifD4ks5r1lSWgaJpZM4MO9tN>

> .

>

--

- Alex

-------------------------------------------------------------------------

2017-05-10T13:25:52Z mirosval

Currently I am worried that my training/prediction preprocessing will diverge over time.

I would be interested in a pipeline where:

\* Preprocessing and post processing can be serialized with the inference in a single model and then used from another language (no `tf.py\_func`, but able to provide implementation at runtime)

\* There is a clearer distinction between input shapes, for training you usually want batches, but for prediction you often care only about single examples

The preprocessing and post processing do not require backprop, but they sill need to carry some values with them (normalization divisors or one hot mappings). It would be ideal to have some `Op` that can `train` some values like this during training, carry them over to production within the serialized graph and then I could provide an implementation for this pre/post processing `Op` in the target production language.

-------------------------------------------------------------------------

2017-05-10T14:02:34Z carlthome

@mirosval, it seems to me that this might be what `tf.Estimator` is intended for eventually.

-------------------------------------------------------------------------

2017-05-13T02:25:11Z neighthan

Not sure if this was mentioned above and I missed it, but I would appreciate a much easier way to switch between train and validation data sets. When using `feed\_dict`, this is very simple, and this is what I'm used to. I've recently been trying to make the switch from `feed\_dict`, but this has been (at least in my limited experience) a major difficulty. The tutorial page [here](https://www.tensorflow.org/programmers\_guide/reading\_data) mainly just suggests using separate processes, but this can be a pain, especially if I want to do early stopping based on the validation data. If I could create some sort of input method (Queue, Dataset, whatever) where I can cleanly swap between training and validation inputs, that would be much nicer (again, `feed\_dict` is great for this, but if it will always be slower, it'd be nice to have a more performant alternative).

-------------------------------------------------------------------------

2017-05-13T02:29:03Z vrv

@neighthan: yes this is something that @mrry has planned as well :)

-------------------------------------------------------------------------

2017-05-13T06:55:40Z kdavis-mozilla

@neighthan Using current constructs, we've constructed a [`SwitchableDataSet`](https://github.com/mozilla/DeepSpeech/blob/master/util/data\_set\_helpers.py#L116) that allows you to at run time switch between train, validation, and test.

-------------------------------------------------------------------------

2017-05-13T07:13:39Z neighthan

@kdavis-mozilla that sounds interesting, but the link appears to be dead. Is there another reference?

-------------------------------------------------------------------------

2017-05-13T07:58:48Z kdavis-mozilla

@neighthan Sorry, fixed.

-------------------------------------------------------------------------

2017-05-13T10:45:31Z kmhofmann

Very promising! That is one of my biggest issues with the current API.

That said, looking at SwitchableDataSet, it seems like implementing new kinds of data feeding use cases will be mostly done by implementing use-case specific classes. Will the new programming model also feature an API to, say, implement what the SwitchableDataSet offers and beyond from more generic, lower-level primitives? I'm just wondering about what things users will come up with (w.r.t. data generation and usage) that would otherwise require (specific) additions to the API...

-------------------------------------------------------------------------

2017-05-13T11:45:22Z vonclites

Am I correct to assume that https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/learn/python/learn/dataframe is part of the new input pipeline initiative? Is there any code out there that uses this, even if that code is undocumented? I'd like to see an example of it in use.

I also assume that the dataframes and transforms are intended to be closely integrated with estimators? At the moment, I don't see how estimators fit (nicely) with the dataframes and transforms. I see where you have ways to generate feed\_fns from dataframes for estimators, but it seems more like an adapter to another approach, instead of part of the pipeline design.

I understand this is all very new and under development. I really like what I see! Keep up the good work.

-------------------------------------------------------------------------

2017-05-13T16:30:15Z jimfleming

@vonclites That's for interacting with Pandas DataFrame/Series, not the new Dataset construct.

-------------------------------------------------------------------------

2017-05-13T16:44:58Z vonclites

@jimfleming It seems to be more general than that. There are methods to create TensorFlowDataFrames from csv files, dicts, numpy arrays, TFRecords, as well as from pandas. It follows the nomenclature of pandas, but it also resembles Spark's pipeline, as @mrry mentioned, and has many of the features he described in the original post.

-------------------------------------------------------------------------

2017-05-13T17:01:32Z jimfleming

I agree that it's fairly generalized and it may be the basis for future

work but this has been available for quite a while and most of the files

haven't been updated in months.

On Sat, May 13, 2017 at 9:46 AM vonclites <notifications@github.com> wrote:

> @jimfleming <https://github.com/jimfleming> It seems to be more general

> than that. There are methods to create TensorFlowDataFrames from csv files,

> dicts, numpy arrays, TFRecords, as well as from pandas. It follows the

> nomenclature of pandas, but it also resembles Spark's pipeline, as @mrry

> <https://github.com/mrry> mentioned, and has many of the features he

> described in the original post.

>

> ‚Äî

> You are receiving this because you were mentioned.

>

>

> Reply to this email directly, view it on GitHub

> <https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-301259803>,

> or mute the thread

> <https://github.com/notifications/unsubscribe-auth/AAN0-Uztf2Y5vav67n\_-YYSpISvvTEOBks5r5d5ngaJpZM4MO9tN>

> .

>

-------------------------------------------------------------------------

2017-05-13T17:04:06Z vonclites

@jimfleming Good point

-------------------------------------------------------------------------

2017-05-19T07:11:06Z omoindrot

First documentation on master under `tf.contrib.data`: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/data

All this seems pretty amazing !

-------------------------------------------------------------------------

2017-05-19T14:31:23Z sjperkins

Very nice indeed. The following iterators would be great:

- [chain](https://docs.python.org/2/library/itertools.html#itertools.chain)

- [product](https://docs.python.org/2/library/itertools.html#itertools.product)

-------------------------------------------------------------------------

2017-05-19T14:56:00Z snnn

Any planing for supporting a seek-able file format, which can be started to read from an arbitrary offset?

Below is why we need it:

Assume there is a large training data set which is in text format, and we need to convert it into tfrecord format. Then we started a map-reduce job, converted it into 10 tfrecord files, started 10 workers to read them, Perfect! Then we want to run it faster, we would change the worker count to 20, 30, 40, ... it would be great if we could do this without re-generate the training data.

Solution:

First, we need to pass an offset and length together with the filename into Dataset 's constructor. Only filename is not enough.

Second, the file format it self must be seekable(aka. splitable). it should be one of the following:

1. Text format

2. Blocked binary format with paddings.

3. Has an index file.

-------------------------------------------------------------------------

2017-05-19T15:00:44Z snnn

This new API is faster than the old one. I've integrated this new dataset API into my factoration machine trainer, and it saved me 20% training time. Thanks @mrry

-------------------------------------------------------------------------

2017-05-20T03:19:36Z omoindrot

I feel like the default code for looping over a dataset is a bit ugly with an exception breaking out of a `while True` loop:

```python

sess.run(iterator.initializer)

while True:

try:

sess.run(train\_op)

except tf.errors.OutOfRangeError:

break

```

Wouldn't there be a way to have an `is\_empty` tensor indicating when the iterator is empty? Like:

```python

sess.run(iterator.initializer)

is\_empty = False

while not is\_empty:

\_, is\_empty = sess.run([train\_op, iterator.is\_empty])

```

-------------------------------------------------------------------------

2017-05-20T17:27:54Z jimfleming

If you're using a MonitoredSession and its variants you should still be able to do this:

```

with tf.train.SingularMonitoredSession() as sess:

while not sess.should\_stop():

sess.run(train\_op)

```

-------------------------------------------------------------------------

2017-05-20T23:43:04Z vonclites

Edit: Sorry, it totally works. I just realized it exits the session context entirely, but the program keeps running. The .end() calls on hooks work as well. @jimfleming you're right again ;)

~~I'm not having success with the monitored session suggestion. The program simply exits without even reporting the exception.~~

```

with tf.train.MonitoredTrainingSession() as sess:

while not sess.should\_stop():

data = sess.run(next\_batch)

print('stopped')

```

the above code prints nothing, just exits after attempting to run past the last batch. However, the following will print "hit exception" forever, unless `break` is used.

```

with tf.train.MonitoredTrainingSession() as sess:

while not sess.should\_stop():

try:

data = sess.run(next\_batch)

except tf.errors.OutOfRangeError:

print('hit exception')

print('stopped')

```

~~Not sure if I should make a feature request or bug report?~~

-------------------------------------------------------------------------

2017-05-23T19:57:12Z ahundt

Do the [High Performance Benchmarks example](https://www.tensorflow.org/performance/performance\_models) recommendations still apply now that the new [Dataset API](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/data) exists?

For instance in the benchmark [RecordInput](https://github.com/tensorflow/tensorflow/blob/e4296aefff97e6edd3d7cee9a09b9dd77da4c034/tensorflow/python/ops/data\_flow\_ops.py#L2074) is [split into minibatches](https://github.com/tensorflow/benchmarks/blob/master/scripts/tf\_cnn\_benchmarks/preprocessing.py#L356), which I tried to incorporate via #10143 just before I found this. If those recommendations are all good ones, the datasets API might benefit from the addition of those very same recommendations.

Might I suggest updating the benchmarks in accordance with this API and vice versa?

-------------------------------------------------------------------------

2017-05-23T22:05:26Z mrry

I've had my head down for a while, so there's lots to respond to here:

\* @sjperkins: Adding `Dataset.chain()` and `Dataset.product()` iterators shouldn't be too hard. I'd like to understand your use case a little better. Do you imagine that most uses will combine exactly two datasets (and hence we might use method chaining to combine them, e.g. `ds1.chain(ds2)`, `ds1.product(ds2)`) or will it be more common to combine more datasets (and hence we'd take a similar approach to `Dataset.zip()`, e.g. `Dataset.chain([ds1, ds2])`, `Dataset.product([ds1, ds2])`)? Also note that, if you need `product()` in the short term, I think you can write `ds1.flat\_map(lambda x: tf.contrib.data.Dataset.zip((tf.contrib.data.Dataset.from\_tensors(x).repeat(), ds2)))`. You could also fake out `chain()` with `Dataset.flat\_map()` and `tf.cond()` but that would be quite ugly :).

\* @snnn: Thanks for kicking the tires! It's great to hear that the new API brought a speedup for your code... we've definitely favored flexibility over performance with the initial version of the API, but look out for improvements over the coming versions.

The idea of supporting seekable file formats is very appealing, and we're figuring out a good API for that. Right now you can use `Dataset.skip()` and `Dataset.take()` to select a sub-dataset from the files, but it is not very efficient, because they materialize the skipped-over inputs before discarding them. I could imagine adding an `Iterator::Seek(size\_t n)` method internally, which would allow iterators to specialize their behavior in this case (or fall back to using `GetNext()` in a loop). This also seems like it would be important for checkpointing iterators, which might be useful for fault tolerance.

\* @omoindrot: I agree that the `try-except` construction is pretty ugly, and we should try to find ways to improve it. The current version is designed to be a drop-in replacement for the queues, which use `tf.errors.OutOfRangeError` to signal completion, and various other classes are designed to catch that exception. Exposing an `iterator.is\_empty` property would be possible, but it would be tricky to make it work in the way you suggest, because (to avoid an exception being raised) you'd need to guard the training subgraph with a `tf.cond(iterator.is\_empty, make\_train\_op, lambda: tf.no\_op())`. Another possibility would be to change the Python API for iterators to use two ops: `Iterator.move\_next()` and `Iterator.get\_current()` (e.g. like the C++ `IEnumerator` protocol), but that would introduce an additional `sess.run()` call, and make it harder to share the iterator between threads.

One possibility I've considered is to create a wrapper that turns an `Iterator`-consuming step into a Python iterator. e.g. Some straw-man code:

```python

def iterate\_step(sess, fetches):

cached\_step = sess.make\_callable(fetches)

try:

while True:

yield cached\_step.run()

except tf.errors.OutOfRangeError:

pass

# ...

for \_, step, loss in iterate\_step(sess, [train\_op, global\_step, loss]):

if step % 100 == 0:

# Run periodic eval, e.g.

```

\* @jimfleming and @vonclites: Thanks for looking into the `MonitoredSession` integration. It's great to hear that it "just works"! I think we'll still need to do something better for the more advanced cases where we might want to reinitialize an iterator in the same session.

\* @ahundt: From our initial experiments, the \*peak\* performance of the benchmark input pipeline is still slightly higher than what you get from using the `tf.contrib.data` API. However, the peak performance is much higher than the throughput of actually using the data to train a model like Inception or ResNet, so you might not notice the difference in regular use. We're investigating how to close the gap, and it's very likely that we'll incorporate some of the ideas from the benchmark code into the `Dataset` implementation.

In particular, one current limitation of the `Dataset` and `Iterator` implementation is that the entire pipeline runs on a single device, whereas the more explicit code in the benchmarks is able to split the processing across multiple CPU and GPU devices. For the best performance, it's going to be important to integrate optimizations like the `StagingArea` for prefetching data to the GPU before it is needed, and we're working on a way to do that more transparently. For now, you can manually pipeline the output of an `Iterator.get\_next()` op with the `StagingArea.put()` op in a similar manner to the benchmark code.

-------------------------------------------------------------------------

2017-05-24T02:43:05Z jasonkriss

This is fantastic! I'm using it now and loving it. @vrv @mrry is the ability to swap between train and validation datasets already there or is that still coming? I see the reinitializable iterators give us the ability to use the same iterator with multiple datasets but each time you run an init op, it essentially starts over on that dataset. I'm currently using the new `Dataset` and `Iterator` apis along with `tf.cond` to accomplish this but is there a more direct/natural way in the works?

-------------------------------------------------------------------------

2017-05-24T03:03:33Z snnn

Hi @mrry

The Iterator API doesn't have a read\_up\_to/enqueue\_many like interface, Will you add it?

-------------------------------------------------------------------------

2017-05-24T07:44:03Z sjperkins

> @sjperkins: Adding Dataset.chain() and Dataset.product() iterators shouldn't be too hard. I'd like to understand your use case a little better. Do you imagine that most uses will combine exactly two datasets (and hence we might use method chaining to combine them, e.g. ds1.chain(ds2), ds1.product(ds2)) or will it be more common to combine more datasets (and hence we'd take a similar approach to Dataset.zip(), e.g. Dataset.chain([ds1, ds2]), Dataset.product([ds1, ds2]))?

@mrry The approach similar to Dataset.zip() please, for the flexiblity. Thanks for the workarounds. :-)

-------------------------------------------------------------------------

2017-05-24T14:39:52Z EdeMeijer

Wanted to let you know I migrated from queues to Dataset and it's been great. Definitely going in the right direction. There's been a few things that are currently missing and I had to work around:

- Being able to use dictionaries for Tensor structs, in addition to tuples and lists (this would be very natural when parsing Examples, which have named features and give you dictionaries)

- Support for SparseTensors. `tf.batch` supports SparseTensor and automatic batching of SparseTensor would make my life a whole lot easier

Additionally, I had this idea where you could maybe implement a random test/train split functionality right into Dataset. This could make things easier too. Something like

```python

dataset\_train, dataset\_test = dataset.split\_train\_test(test\_ratio=0.2, seed=1234)

```

Keep up the good work!

-------------------------------------------------------------------------

2017-05-25T03:00:01Z jasonkriss

@lhlmgr The way i understand that example is that iterator needs to be reinitialized every time you switch between train and validation. This isn't the end of the world but for large datasets where we are shuffling minibatches, we want a reasonable `buffer\_size` which means each initialization is quite slow. I find the `tf.cond` approach with two separate datasets/iterators to work better/faster in that case. That way we can periodically run through validation data without losing our place int the training set.

-------------------------------------------------------------------------

2017-05-29T08:19:59Z albertz

Maybe here is a good place to refer to my Dataset related questions:

https://stackoverflow.com/questions/44132579/feed-data-into-a-tf-contrib-data-dataset-like-a-queue

https://stackoverflow.com/questions/44132307/tf-contrib-data-dataset-repeat-with-shuffle-notice-epoch-end-mixed-epochs

-------------------------------------------------------------------------

2017-05-30T18:00:59Z drasmuss

I really like the new Dataset/Iterator API! Here is a feature that would help my use-case:

I would like to be able to create iterators that share part of the data pipeline. As a simple example, something like this:

``` python

a = np.arange(12)

data = tf.contrib.data.Dataset.from\_tensor\_slices(a)

data = data.shuffle(12)

data0 = data.map(my\_func0)

data1 = data.map(my\_func1)

iter0 = data0.make\_one\_shot\_iterator()

iter1 = data1.make\_one\_shot\_iterator()

op0 = iter0.get\_next()

op1 = iter1.get\_next()

```

What I'd like is for `op0` and `op1` there to output elements in the same order (because they share the `shuffle` step), but with different functions (`my\_func0`/`my\_func1`) applied. That is, I would like to create input pipelines that share some processing, and then diverge at some point for additional processing.

-------------------------------------------------------------------------

2017-05-31T01:09:46Z ahundt

What @drasmuss suggests is very useful for segmentation tasks where both labels and images need to be augmented. For example the images could very reasonably use bilinear interpolation, but interpolating label values is not okay because a label pixel boundary of 0 and 2 should not be interpolated to the completely different label of 1.

-------------------------------------------------------------------------

2017-05-31T14:24:59Z maxfiedler

I am only starting to read into the new API, but I want to share two problems that I had with the old Input Queues in concurrence with using MonitoredSession with SessionRunHooks.

We also used two separate queues, one handling input data\_files as string names and the other one the resulting input data with preprocessing being done in between those two.

We needed to make sure that the enqueue operations fills at least a certain multiple of the batch size into the first queue for our code to run without problems (otherwise the second input queue stalled)

Looking something like this

```

queue\_size\_train = sess.run([ipc.train.rsq\_pre\_size)

while queue\_size\_train <= batch\_size \* 5:

sess.run([ipc.train.rsq\_pre\_enq],

feed\_dict={ipc.train.ph\_in: dataset.train.inputs,

ipc.train.ph\_tgt: dataset.train.targets})

queue\_size\_train = sess.run([ipc.train.rsq\_pre\_size])[0]

```

Now when I switched from a normal Session to using MonitoredSession and added a logging hook and told it to log the 'accuracy' tensor, it tried in vain to evaluate the first session run call as the hook had added that tensor to the fetch list, but with the queue being still empty there was no way to evaluate accuracy yet.

Problematic was that the program just stopped and waited for some process to begin filling the queue, but there was non, so it just did nothing, but also didn't throw an exception or give any kind of warning, which made understanding what was happening a bit difficult.

The problem was eventually easily solved just executing one enqueue operation as the very first sess.run() call within the `with MonitoredSession(...) as sess:` block, but it would be nice if e.g. the input queues could be pre-filled with some initial values upon creation, so that this issue doesn't arise.

Aside from that, we use two different input pipelines for training and validation data that we connect to the network part of our graph alternating through a switch implemented through `tf.QueueBase.from\_list`.

Now in the `with ... as sess:` block you can easily implement an if block which pipeline to choose based on `global\_step % interval == 0` but that means copy pasting the same code (sess.run() and add\_summary() calls) times the number of different input pipelines you use (e.g. train, validation\_1, validation\_2, ...)

It would be nice to integrate this directly into MonitoredSession using Hooks somehow (i.e. every 10th step create and save summaries using validation\_1 input pipeline, every 25th step ...)

-------------------------------------------------------------------------

2017-05-31T16:28:10Z vonclites

1. Anything that helps me analyze if and where the bottleneck lies in the input pipeline would be great. For example, a way to monitor the number of examples in the buffers along the input pipeline would be helpful. Or the number of items processed per thread of a .map() operation Basically, something along the lines of how the queues create summaries for the number of images they are holding.

2. I think others have mentioned something like this, but a way to create a Dataset from a streaming source of data. Something that provides similar functionality as the generator feed function in https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/estimator/inputs/queues/feeding\_functions.py (but it would be cool if the source of the data could be from an arbitrary source, maybe using a kind of publisher/subscriber model?)

Thanks for the hard work. I'm digging the new API.

-------------------------------------------------------------------------

2017-06-01T09:07:12Z kratzert

In the last couple of days I was playing around quite a bit with the new Input API and I think the `Dataset` and `Iterator` classes improve highly the clearness and readability of code (compared to the old input queues). Also switching between e.g. training and validation dataset within one session works quite effortless.

But I have also some questions and suggestions.

Maybe first a question: Is the `Dataset` class implemented based on queues? Because from the post here it doesn't get clear to me if or if not. In Tensorboard there is no additional information added with the new API about the status of any queue (how many objects are currently queued). Also observing my CPU/GPU resources/workload I can see, that the GPU workload drops to zero often (I guess in between batches).

Then a suggestion:

I think the `dataset.shuffle()` could be improved, if shuffling is not done only on the \_n\_ ( = buffer\_size) samples in memory, but somehow on the whole list of inputs. For example: I'm storing path to images and labels in a text file. If shuffling is not done already in the text file you can often have thousands of lines after each other of the same class. If I now only work with `dataset.shuffle()` it can easily happen (depending of the buffer\_size) that all elements that get shuffled are anyhow of the same class. The only. Maybe some toy example (ignoring labels and only working with image paths) to make my point clearer. For reasons of readability I work with a very small `buffer\_size` and list of file names. But I guess everyone can imagine the same just with thousands of filenames in the list and a buffer\_size e.g. of 5000.

```

import tensorflow as tf

image\_paths = tf.constant(['train/class1/img1.png',

'train/class1/img2.png',

'train/class1/img3.png',

'train/class1/img4.png',

'train/class1/img5.png',

'train/class1/img6.png',

'train/class1/img7.png',

'train/class2/img1.png',

'train/class2/img2.png',

'train/class2/img3.png',

'train/class2/img4.png',

'train/class2/img5.png',

'train/class1/img6.png',

'train/class1/img7.png'])

dataset = tf.contrib.data.Dataset.from\_tensor\_slices(image\_paths)

dataset = dataset.shuffle(3)

iterator = dataset.make\_one\_shot\_iterator()

next\_element = iterator.get\_next()

with tf.Session() as sess:

while True:

try:

path = sess.run(next\_element)

print(path)

except tf.errors.OutOfRangeError:

break

```

This would print something like:

```

'train/class1/img1.png'

'train/class1/img2.png'

'train/class1/img4.png'

'train/class1/img3.png'

'train/class1/img5.png'

'train/class1/img7.png'

'train/class2/img1.png'

'train/class1/img6.png'

'train/class2/img3.png'

'train/class2/img4.png'

'train/class1/img6.png'

'train/class2/img5.png'

'train/class1/img7.png'

'train/class2/img2.png'

```

So since there is only a shuffling between the 3 examples in the buffer, the first samples (same for batches) will all have samples only of one class. So unless the shuffling isn't done already in the list of filenames you'll have troubles training any network. And if the available dataset of images is huge, increasing the buffer\_size is often not a solution.

Another problem I see, is that like shuffling currently is implemented, there is no true shuffling of the entire dataset possible. The only workaround I found was pre-shuffling the filelist I read from the text file before creating the dataset. But once the dataset is created, it's only possible to shuffle in the range of the buffer\_size.

-------------------------------------------------------------------------

2017-06-02T12:10:51Z kmhofmann

@mrry Thanks for the a preview of the new API; I think this is a good starting point!

One function that still seems to be missing, but would be essential for one of our primary use cases (see comment above: https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-283186552) is a

`Dataset.map\_to\_multiple()`

used as in

`dataset2 = dataset1.map\_to\_multiple(func)`

function, where one element of `dataset1` is mapped to one \*\*or more\*\* elements for `dataset2`; i.e. #`dataset2` >= #`dataset1`.

As far as I understand it, `Dataset.map()` preserves a 1:1 mapping, which is not sufficient for our use case.

One concrete example of why this would be useful:

Assume `dataset1` is a list of large images (e.g. 8192x8192 each) [with corresponding labels]. Then, `dataset2` is created by (randomly) iterating over each element `d1` of `dataset1`, and for each of these elements the function `func` samples a (variable) number of sub-images [and sub-labels] (e.g. 256x256 each) from `d1`, taken from various regions of `d1`. For example, in one instance, `func` might return 142 new {image, label} pairs that will be added to `dataset2`. In another instance, it might return 389 new pairs, etc. The number of elements generated each time is variable and conditional on the properties of element `d1`.

-------------------------------------------------------------------------

2017-06-02T13:53:47Z EdeMeijer

@kmhofmann I think you can map one example to multiple examples with `Dataset.flat\_map()`. Inside your flatmap function, create a new Dataset object with one or multiple examples for every input example.

-------------------------------------------------------------------------

2017-06-02T14:02:32Z kmhofmann

@EdeMeijer That could be -- it's really hard to tell, as the documentation is quite sparse, with no examples. (There seem to be two places containing some amount of separate documentation: either on [GitHub](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/data) or on [tensorflow.org](https://www.tensorflow.org/versions/r1.2/api\_docs/python/tf/contrib/data/Dataset#flat\_map))

One thing I notice is that the arguments `num\_threads` and `output\_buffer\_size` from `map()` are missing in `flat\_map()`; does that mean no parallel processing is possible? Or is this a TODO? Hard to scope both functionality and feature set...

-------------------------------------------------------------------------

2017-06-02T14:21:26Z EdeMeijer

I based my suggestion on the one flat\_map code example on the github page you linked. There, single string tensors come in (file names) and whole Datasets are emitted in the map function, so that seems pretty clear. I guess the parallel processing from `map()` is a TODO, I'm sure they'd love a PR :)

-------------------------------------------------------------------------

2017-06-02T14:26:04Z kmhofmann

Ah, thanks, I missed that, as it was in an example about text processing. Still, the function description in the documentation seems a bit sparse, consisting of `Maps map\_func across this dataset and flattens the result`.

-------------------------------------------------------------------------

2017-06-04T09:13:21Z agniszczotka

input\_fn has to return features and labels only. What about extra params which can be used while training progress to customise loss for given input?

-------------------------------------------------------------------------

2017-06-05T15:33:20Z drasmuss

Another feature request: it'd be great if there was an `iterator.peek()` operator, which would return the current iterator value (like `iterator.get\_next()`), but not advance the iterator. This would make it easier to coordinate multiple elements of a model that all want to read from the iterator before advancing it one step.

-------------------------------------------------------------------------

2017-06-09T19:17:47Z nirmalthacker

Hi, firstly thanks for this API, Im very keen on using it. Primarily I am interested in using it to switch between training and validation datasets in the same process.

However I'm confused how one does that in this new paradigm. For instance I see no way to "get an iterator" in the middle of a dataset. As an example here is a piece of code that demonstrates what I'd like to do. Every few steps in an epoch, I'd like to run a validation op, but the output of this code shows that the iterator never advances ahead of item 0 in either dataset. How does one do that?

```

training\_dataset = tf.contrib.data.Dataset.range(100)

validation\_dataset = tf.contrib.data.Dataset.range(900,950)

iterator = tf.contrib.data.Iterator.from\_structure(training\_dataset.output\_types,

training\_dataset.output\_shapes)

next\_element = iterator.get\_next()

training\_init\_op = iterator.make\_initializer(training\_dataset)

validation\_init\_op = iterator.make\_initializer(validation\_dataset)

# Run 3 epochs with 10 steps each

for e in range(3):

print("Epoch: %d" %e)

for step in range(11):

sess.run(training\_init\_op)

ne = sess.run(next\_element)

print("Step: %d Training set: next\_element: %d " %(step, ne))

# Run validation every 5 steps only

if step % 5 == 0:

sess.run(validation\_init\_op)

ne = sess.run(next\_element)

print("Step: %d Validation set: next\_element: %d " %(step, ne))

```

In the above, since we run an init each time to get an iterator pointing to its required dataset, we end up running a training and validation on the first item of each dataset, always. How does one modify this to get the updated position of the iterator in each dataset?

-------------------------------------------------------------------------

2017-06-09T20:26:24Z jasonkriss

@nirmalthacker Every time you run an iterator init op, it restarts the iterator. Take a look at the documentation here [here](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/data#creating-an-iterator). This approach has the limitations i discussed [here](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-303909270).

-------------------------------------------------------------------------

2017-06-09T22:48:37Z nirmalthacker

@jasonkriss , I see - thanks! This is what I have now, and it handles what I wanted. Is this what you meant?

```

training\_dataset = tf.contrib.data.Dataset.range(100)

validation\_dataset = tf.contrib.data.Dataset.range(900,950)

t\_iterator = tf.contrib.data.Iterator.from\_structure(training\_dataset.output\_types,

training\_dataset.output\_shapes)

v\_iterator = tf.contrib.data.Iterator.from\_structure(validation\_dataset.output\_types,

validation\_dataset.output\_shapes)

is\_validating = tf.placeholder(dtype=bool,shape=())

next\_element = tf.cond(is\_validating, lambda:v\_iterator.get\_next(), lambda:t\_iterator.get\_next())

training\_init\_op = t\_iterator.make\_initializer(training\_dataset)

validation\_init\_op = v\_iterator.make\_initializer(validation\_dataset)

sess.run([training\_init\_op, validation\_init\_op])

# Run 3 epochs with 10 steps each

for e in range(3):

print("Epoch: %d" %e)

for step in range(11):

ne = sess.run(next\_element, feed\_dict={is\_validating: False})

print("Step: %d Training set: next\_element: %d " %(step, ne))

if step % 5 == 0:

ne = sess.run(next\_element, feed\_dict={is\_validating: True})

print("Step: %d Validation set: next\_element: %d " %(step, ne))

```

-------------------------------------------------------------------------

2017-06-09T23:47:04Z jasonkriss

@nirmalthacker Yep, that's essentially what I meant. Although, at each validation step, I will reinit the validation iterator and run through the full validation dataset. But that's certainly the gist of it.

-------------------------------------------------------------------------

2017-06-14T08:49:31Z winston-li

I'm trying migrating input pipeline from tf.train.string\_input\_producer & tf.train.shuffle\_batch to Dataset APIs. The parameter "allow\_smaller\_final\_batch" in tf.train.shuffle\_batch(...) is useful when I'd like to assure all batches are evenly divisible by number of gpus. (I'm doing data parallelization on multiple gpus, and the batch\_size is multiple of num\_gpus). Is there any setting for Dataset APIs to drop the final smaller batch if any?

-------------------------------------------------------------------------

2017-06-14T10:32:30Z EdeMeijer

@winston-li I think you could just use `.filter()` for that. If you know your batch size is for example 32, then something like

```python

dataset = dataset.filter(lambda batch: tf.shape(batch)[0] == 32)

```

should do the trick.

-------------------------------------------------------------------------

2017-06-14T10:45:01Z byronyi

@ppwwyyxx I believe a custom pipeline could be designed using TF server with distributed runtime, i.e. IPC send/recv ops. It's included in C API so it won't be too hard to port it to other languages. A down side is that you do need package the whole TF runtime wherever you need this pipeline.

We are actually working on a out-of-band data plane for TF, but it is still a great deal of ongoing work. The design would be similar to the [ExternalShuffleService](https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/deploy/ExternalShuffleService.scala) in Spark, using an in-memory storage such as LevelDB or LMDB, and a reader client in TF. If performance is a primary concern then it should be tightly integrated with hardware, i.e. GPU/NVMe/RNIC, etc.

-------------------------------------------------------------------------

2017-06-15T04:53:53Z winston-li

@EdeMeijer Thanks. I thought it should work, but after some experiments, I can't make it work as expected. I followed the guidelines of dataset README.md, with pseudo code like following:

def \_parse\_function(example\_proto):

features = {"image": tf.FixedLenFeature((), tf.string, default\_value=""),

"label": tf.FixedLenFeature((), tf.int32, default\_value=0)}

parsed\_features = tf.parse\_single\_example(example\_proto, features)

return parsed\_features["image"], parsed\_features["label"]

BATCH\_SIZE = 256

filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]

dataset = tf.contrib.data.TFRecordDataset(filenames)

dataset = dataset.map(\_parse\_function)

dataset = dataset.batch(BATCH\_SIZE)

dataset = dataset.filter(lambda imgs, lbls: tf.shape(imgs)[0] == BATCH\_SIZE)

iterator = dataset.make\_initializable\_iterator()

next\_element = iterator.get\_next()

images, labels = next\_element

# Training cycles for 100 epochs.

for \_ in range(100):

sess.run(iterator.initializer)

while True:

try:

images\_r, labels\_r = sess.run([images, labels])

print(images\_r.shape)

except tf.errors.OutOfRangeError:

break

After applied the filter, no data available in training cycles. I found the dataset (after batch, prior to filter) was in this form:

(<tf.Tensor 'arg0:0' shape=(?, 43200) dtype=float32>, <tf.Tensor 'arg1:0' shape=(?, 36) dtype=float32>)

Looks like the batch dimension is "?" (None?), so the predicate always fails... or I did something wrong?

-------------------------------------------------------------------------

2017-06-15T08:23:22Z EdeMeijer

@winston-li seeing "?" as shape is because at that point you're looking at the 'static' shape of the tensor, which isn't always defined (the graph doesn't know in advance how many examples there will be). However, `tf.shape()` evaluates the dynamic, real-time shape of a tensor, so I thought this should work.

Anyway, I tried creating a minimal example, but I'm getting internal kernel errors when my filter excludes any element, so for starters I opened https://github.com/tensorflow/tensorflow/issues/10725. However, what I \*\*did\*\* find was that, obviously, we should use Tensorflow ops instead of standard comparisons since the result of `tf.shape()` is a tensor and not a normal array.

My tf version's filter is broken, but if yours works could you try this instead?

```python

dataset = dataset.filter(lambda imgs, lbls: tf.equal(tf.shape(imgs)[0], BATCH\_SIZE))

```

-------------------------------------------------------------------------

2017-06-15T13:27:57Z winston-li

@EdeMeijer Thank you very much, it works in my case :-)

-------------------------------------------------------------------------

2017-06-15T16:11:03Z guillaumekln

For sequence data, it is common to batch sequences with same (or similar) lengths. This is usually achieved by simply sorting the dataset based on sequence length. However, this does not seem possible to achieve with the current API (if yes, please let me know how to do it).

It would be nice so see something in this flavor:

```python

dataset = dataset.sort(lambda a, b: tf.shape(a)[0] < tf.shape(b)[0], buffer\_size=10000)

```

which will sort items by chunk of 10000 (similar to `dataset.shuffle`) using the given comparison function.

-------------------------------------------------------------------------

2017-06-15T19:39:35Z kratzert

I would like to ask again (since there was no reply to my [comment](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-305435143) above: Am I right, that the new input pipeline isn't implemented using Queues? I did some tests using the new input pipeline to load an preprocess images, but it seems everything is done sequentially and their is only a negligible performance improvement over using e.g. OpenCV to load and preprocess images. I was hoping the new input pipeline would be build on top of queues, since they provide major performance boosts but make it quite hard to work with (e.g. having seperate input pipelines with queues to switch between training and validation datasets). This is quite easy with the new API but it seems there is no real performance boost. Anybody observed the same or opposite?

-------------------------------------------------------------------------

2017-06-20T06:53:21Z vvekic

@kratzert I, too, have experienced issues with getting GPUs to 100% usage and keeping them there. The Dataset API, it seems, is implemented less efficiently and though it is a welcome change regarding code clarity and simplicity as well as a more natural way of doing training and validation, it cannot (yet) substitute queues for high data rate usecases, such as computer vision.

-------------------------------------------------------------------------

2017-06-20T07:22:02Z snnn

@kratzert You are right. The new input pipeline do not have queues. If you aren't satisfied with the shuffling it provides, you can do it outside TF: You can load all the filenames into memory then shuffle them in any way you like.

-------------------------------------------------------------------------

2017-06-20T07:30:55Z kratzert

@snnn Yes I know and I do exactly this. But by doing this I can't find a way to shuffle the entire data in e.g. my training data every epoch. I can shuffle e.g. the list of filenames before creating a dataset, but once I start a session to my knowledge I can only shuffle the data from the dataset that I have in memory using dataset.shuffle(buffer\_size). But with images this can be hardly done for the entire dataset in memory. And I can't shuffle the filenames again and create a new dataset from them, once inside the session or am I wrong?

@vvekic Thanks for your reply, so I know it's not only me having performance issues. Of course the code clarity and simplicity of working with the new dataset class is a huge step forward and very welcome. But it seems that for training computer vision networks queues are still the way to go (unfortunately) as the performance boost is immense.

-------------------------------------------------------------------------

2017-06-20T07:39:29Z ppwwyyxx

@kratzert Using queues increases your performance by overlapping data loading latency, and this is independent of what you use to load your data. You can always insert queues or StagingArea in the input pipeline, regardless of whether the actual data loading is done by dataset API, the old input operators, or Python.

@byronyi What I meant is to receive tensors from non-TF processes. Because as @PatWie pointed out above, data processing doesn't really need to happen in the graph.

-------------------------------------------------------------------------

2017-06-20T07:44:44Z kratzert

@ppwwyyxx do you have any example code for combining queues and the new dataset api? Sounds awesome and I will definitely try this out later.

-------------------------------------------------------------------------

2017-06-20T13:10:23Z kratzert

@snnn Well I'll look into it, but since my knowledge in C++ isn't so profound we'll see how successfully I'll be. Anyway, I think this could be a feature that more people than I might be interested in and should/could possibly be integrated into master.

-------------------------------------------------------------------------

2017-06-20T16:52:23Z ahundt

@kratzert https://www.tensorflow.org/performance/performance\_models and the associated code shows how to use StagingArea.

-------------------------------------------------------------------------

2017-06-21T08:42:49Z EdeMeijer

@kratzert it is certainly possible to re-shuffle the filenames for every epoch, I'm doing exactly that for my own training. You can use an initializable iterator together with a placeholder to achieve this. I'm doing something like this:

```python

filenames\_op = tf.placeholder(tf.string, shape=[None])

dataset = tf.contrib.data.TFRecordDataset(filenames)

iterator = dataset.make\_initializable\_iterator()

next\_elem = iterator.get\_next()

filenames = ['file1', 'file2', 'file3']

# For every epoch, shuffle the file names and initialize the iterator

random.shuffle(filenames)

sess.run(iterator.initializer, {filenames\_op: filenames})

```

-------------------------------------------------------------------------

2017-06-21T09:31:54Z kratzert

@EdeMeijer That's smart. Thank you very much. I should have come to this on my own! Here is a complete working code snippet for anybody interested:

```python

import tensorflow as tf

import numpy as np

num\_epochs = 2

image\_paths = ['train/class1/img1.png',

'train/class1/img2.png',

'train/class1/img3.png',

'train/class1/img4.png',

'train/class1/img5.png',

'train/class1/img6.png',

'train/class1/img7.png',

'train/class2/img1.png',

'train/class2/img2.png',

'train/class2/img3.png',

'train/class2/img4.png',

'train/class2/img5.png',

'train/class1/img6.png',

'train/class1/img7.png']

num\_samples = len(image\_paths)

filenames\_op = tf.placeholder(tf.string, shape=[None])

dataset = tf.contrib.data.Dataset.from\_tensor\_slices(filenames\_op)

iterator = dataset.make\_initializable\_iterator()

next\_element = iterator.get\_next()

with tf.Session() as sess:

for epoch in range(num\_epochs):

print("Starting epoch %i" % (epoch))

np.random.shuffle(image\_paths)

sess.run(iterator.initializer, {filenames\_op: image\_paths})

for i in range(num\_samples):

path = sess.run(next\_element)

print(path)

```

This shuffles as desired on every epoch the entire dataset and gives an output e.g. like this:

```

Starting epoch 0

'train/class2/img4.png'

'train/class2/img1.png'

'train/class1/img2.png'

'train/class1/img7.png'

'train/class1/img1.png'

'train/class1/img6.png'

'train/class1/img6.png'

'train/class1/img5.png'

'train/class1/img4.png'

'train/class1/img7.png'

'train/class1/img3.png'

'train/class2/img2.png'

'train/class2/img3.png'

'train/class2/img5.png'

Starting epoch 1

'train/class1/img7.png'

'train/class1/img4.png'

'train/class2/img1.png'

'train/class1/img6.png'

'train/class1/img2.png'

'train/class1/img3.png'

'train/class1/img6.png'

'train/class2/img5.png'

'train/class2/img2.png'

'train/class1/img1.png'

'train/class2/img3.png'

'train/class2/img4.png'

'train/class1/img5.png'

'train/class1/img7.png'

```

-------------------------------------------------------------------------

2017-06-24T04:43:38Z c810armyHuan

Currently the tutorial says that we can use

```python

dataset = dataset.repeat()

dataset = dataset.shuffle(buffer\_size=10000)

```

to get shuffled data. The pattern is also used in [`tf.contrib.data.read\_batch\_features`](https://github.com/tensorflow/tensorflow/blob/1efd7f171ba30421b5d8369a93526395a721c0d9/tensorflow/contrib/data/python/ops/dataset\_ops.py#L595)

However calling `repeat` before `shuffle` could lead to the shuffle across multiple epochs.

For example, the following code

```python

import tensorflow as tf

data = tf.contrib.data

dataset = data.Dataset.from\_tensor\_slices(['file\_0', 'file\_1'])

repeat\_count = 5

shuffle\_buffer\_size = 100

repeat\_then\_shuffle = dataset.repeat(count=repeat\_count)

repeat\_then\_shuffle = repeat\_then\_shuffle.shuffle(

buffer\_size=shuffle\_buffer\_size)

repeat\_then\_shuffle\_iter = repeat\_then\_shuffle.make\_one\_shot\_iterator()

get\_next = repeat\_then\_shuffle\_iter.get\_next()

with tf.Session() as sess:

result = []

try:

while True:

result.append(sess.run(get\_next))

except tf.errors.OutOfRangeError:

pass

print(result) # [b'file\_0', b'file\_0', b'file\_0', b'file\_1', b'file\_0', b'file\_1', b'file\_0', b'file\_1', b'file\_1', b'file\_1']

```

gets 3 `file\_0` before getting a `file\_1`.

Are there any concerns about calling `shuffle` before `repeat`?

-------------------------------------------------------------------------

2017-07-09T20:33:53Z hanxiao

Just want to add something here, I implemented a \*\*multiprocess-based\*\* data feeding pipeline for multi-task learning. It can achieve avg. GPU utilization >90% and quad-core CPU utilization >95%. Less prone to memory leak and particularly good for days-long training. Not saying it's perfect, but at least works much better than current TF queue API in my case. If anyone interested: https://hanxiao.github.io/2017/07/07/Get-10x-Speedup-in-Tensorflow-Multi-Task-Learning-using-Python-Multiprocessing/

-------------------------------------------------------------------------

2017-07-10T05:34:03Z PatWie

That was already done in [TensorPack](https://github.com/ppwwyyxx/tensorpack) for a while now by @ppwwyyxx. There you also get further speedup using ZMQ -- plus it has a nice interface using Python generators. For me, the way tensorpack handles input data, is the most elegant way. I hope to see something like this in a future TF.

-------------------------------------------------------------------------

2017-07-10T13:38:08Z hanxiao

@PatWie thanks for pointing this out! I just quickly checked @ppwwyyxx repo really awesome! Thanks again

-------------------------------------------------------------------------

2017-07-14T02:28:16Z xieqihui

It would be great to have GPU resident queues.

-------------------------------------------------------------------------

2017-07-14T07:22:49Z sjperkins

> It would be great to have GPU resident queues.

@xieqihuiPG See [StagingArea](https://www.tensorflow.org/api\_docs/python/tf/contrib/staging/StagingArea) and [MapStagingArea](https://github.com/tensorflow/tensorflow/pull/9686)

-------------------------------------------------------------------------

2017-07-18T06:10:32Z Kismuz

Would greatly appreciate:

1. Efficient random sampling:

`Dataset.sample\_random()`

2. Dynamical changing and resizing methods:

`Dataset.update(), Dataset.pop()` etc., e.g. for creating streaming buffers, replay memory objects...

3. Meta- and descriptive statistic integration into dataset object and supportive methods like `Dataset.describe()`

4. Closer integration with HDF5 anyway

-------------------------------------------------------------------------

2017-07-19T18:32:23Z jrbtaylor

#11591 We need efficient sampling/shuffling for large datasets

-------------------------------------------------------------------------

2017-08-08T21:47:41Z sirfz

What about supporting custom ops to create a Dataset? For example, let's say I have a Python function which returns a new batch on each call (a generator). I want to wrap this function using `tf.py\_func` and use it to build a `Dataset`. This doesn't seem to be supported?

I currently use this method with `tf.train.\*batch\*` ops and it works nicely but I'd like to find a way to do this for evaluation as well (and figured maybe Dataset is a good way to do this with the "reintializable" iterator).

-------------------------------------------------------------------------

2017-08-11T14:48:01Z eaplatanios

@mrry This is great work and definitely very useful for creating nice learning APIs on top of TensorFlow. However, I have a couple main concerns:

- I cannot see a way currently to "unzip" a dataset. Let's say we have a trainable model that has both a `train`/`fit` method and a `infer`/`predict` method. Let's call the type of the (potentially) nested structure of inputs to our model `I` and the type of training inputs, which are only needed when training (e.g., supervision labels), `TI`. In this case, we want the `train` method to accept datasets with elements of type `(I, TI)` (i.e., a tuple of `I` and `TI`) and the `predict` method to accept datasets with elements of type `I` or `(I, TI)` (in which case it would ignore the labels). We also want the model to only have one underlying graph, supporting all these types of input. The way I could see doing that was for the underlying model to construct two iterators (one with elements type `I` and one with type `TI`) and initialize them according to the provided datasets. However, if somebody provides a dataset with elements of type `(I, TI)` to the `train` method, there is no way to unzip this dataset and initialize both iterators. One has to use `Dataset.map` twice, which is not efficient (I think but please correct me if I'm wrong) and which may also not pull matching elements from the datasets (if each pull advances the current index in the original first dataset -- I'm not sure if that happens).

- It would be nice to support iterators over tensors defined in other languages as @sirfz mentioned. I cannot see an efficient way to do that with the current API. Please correct me if I'm wrong but currently one would have to create a new `TensorDataset` per batch and re-initialize an existing iterator.

I think @fchollet may be able to comment on my first point as currently my understanding is that they are thinking or creating an entirely new graph for training only, for such cases (third step described [here](https://github.com/fchollet/keras/issues/7503)).

Also, if my description is terribly unclear, please let me know and I'll try to clarify.

-------------------------------------------------------------------------

2017-08-12T11:50:12Z tillahoffmann

The new input pipelines are great! But unfortunately, we are unable to use them for large-scale training because our data preprocessing is quite costly and needs to be distributed across multiple machines--or we just haven't figured out the right way to do it.

We have thus been using the old `FIFOQueue` interface in the following manner (pseudocode):

```python

# Set up queues

kwargs = {'capacity': ..., 'dtypes': ..., 'names': ..., 'shapes': ...}

train\_queue = tf.FIFOQueue(\*\*kwargs)

valid\_queue = tf.FIFOQueue(\*\*kwargs)

queue\_index = tf.Variable(0, trainable=False)

queue = tf.QueueBase.from\_list(queue\_index, [train\_queue, valid\_queue])

batch\_size = ...

batch = queue.dequeue\_many(batch\_size)

# Build model

output = build\_model(batch['X'])

loss = evaluate\_loss(output, batch['y'])

# Fill queues

train\_data\_stream = some\_iterable\_for\_training\_data()

validation\_data\_stream = some\_iterable\_for\_training\_data()

start\_filling\_queues\_in\_background\_thread(train\_queue, train\_data\_stream)

start\_filling\_queues\_in\_background\_thread(validation\_queue, validation\_data\_stream)

```

Having two different queues with `from\_list` allows us to switch between the training and validation queue by either setting the `queue\_index` or feeding it in the `feed\_dict`.

The `some\_interable\_for\_xxx\_data` are usually generators that get data from a bunch of workers sitting behind a load balancer (e.g. using ZeroMQ, RabbitMQ, or PubSub). This approach works well (because the queues provide a buffer) but we don't have any way of telling when the iterator is exhausted. Some workarounds are

\* closing the queue in the background thread such that a `tf.errors.OutOfRangeError` is raised when the queue is exhausted (but then we can't reopen it again #4535)

\* setting a timeout on the `session.run` of the training op and assuming that a timeout is due to the queue being exhausted (but the network connection might be down or our workers might be too slow)

\* counting the number of items we've processed and comparing with the expected number of items in the iterator (but that's fiddly and sometimes we don't even know how long the iterator is)

\* adding an `exhausted` field to the queue `names` and letting the background thread enqueue an item with `exhausted=True` together with an assertion around the dequeue operation (but using `dequeue\_many` will dequeue elements from the next epoch if the number of items per epoch is not an integer multiple of the batch size, see also #2514)

None of these are satisfactory and it would be great to see either the ability to construct `Dataset`s from python iterator \*with a queue for buffering\* or fix #4535 (which will automatically fix #2514).

Looking forward to hear whether we've just not been using the datasets API right.

-------------------------------------------------------------------------

2017-08-15T08:17:39Z rasmusbergpalm

I think the queues are nice enough. I'd like to see two things improved though:

An easier way of inputting data from native python other than using placeholders, and managing threads.

Maybe a class `InputQueue(delegate, fn, n\_filler\_threads)` that takes a tensorflow queue delegate and a python function `fn`. `fn` returns a (possibly nested) tuple of np.array or lists. The InputQueue starts `n\_filler\_threads` that calls `fn` and puts these on the `delegate`. The threads are daemons so shuts down when the main process does.

Anyway, that's just my thoughts. It's probably a lot harder than this due to the static requirements of tensorflow. Maybe you just have to provide the sizes when you create the `delegate`.

-------------------------------------------------------------------------

2017-08-21T07:17:47Z tengerye

I am using the new api `Dataset` now. But still find the problem that how to dynamically feed data to the Dataset. There are two similar questions in [here](https://stackoverflow.com/questions/45788433/how-to-enable-dataset-pipeline-has-distributed-reading-and-consuming) and [here](https://stackoverflow.com/questions/44132579/feed-data-into-a-tf-contrib-data-dataset-like-a-queue)@albertz.

As you can see, the real-world problems are more than just feeding into a series of images or texts. So I would really appreciate if you could let me to feed the data \*\*freely\*\* in terms of \*\*when\*\* and \*\*how\*\*.

I can image two options. One is efficient distributed reading through `feed\_dict`. Although it is slow, but with multi-processing, it is just a matter of machine. The other one is to \_wrap\_ some mature and widely accepted implementation.

-------------------------------------------------------------------------

2017-08-21T10:21:59Z suiyuan2009

use placeholder as input to a queue, and the model reads inputs from the queue, then use a session run thread to feed inputs(maybe produced by hadoop mapreduce) to the queue. use staging area you can even hide all preprocessing and input time.

-------------------------------------------------------------------------

2017-08-22T23:14:05Z bhack

I'm trying to test the example in the [doc](https://www.tensorflow.org/versions/r1.3/programmers\_guide/datasets#preprocessing\_data\_with\_datasetmap)

But seems that this call is passing only 1 argument to the function:

> dataset = dataset.map(\_parse\_function)

Instead the function is defined with two parameter

-------------------------------------------------------------------------

2017-08-23T00:08:41Z ahundt

@eaplatanios one relevant PR for zip/unzip is https://github.com/tensorflow/tensorflow/issues/10837

-------------------------------------------------------------------------

2017-08-24T21:35:10Z bhack

@mrry Have you tested this [section of the documentation](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/docs\_src/programmers\_guide/datasets.md#applying-arbitrary-python-logic-with-tfpy\_func) with python3?

-------------------------------------------------------------------------

2017-08-25T08:47:29Z AMairesse

@bhack I haven't been able to make it work with more than one parameter in return from the function given to py\_func. I'm using python3 and didn't tried with python2.

Is your problem similar ?

-------------------------------------------------------------------------

2017-08-25T08:53:45Z bhack

@AMairesse The first problem was solved with https://github.com/tensorflow/tensorflow/commit/2139e7d8b10764f2245f34548f6fbfc25d29bff8

-------------------------------------------------------------------------

2017-08-26T07:59:22Z AMairesse

@bhack Thanks, will try that soon, I was using a workaround which I'm not proud of :-)

The fix in the documentation is one month old and prior to v1.3 release, the tensorflow.org website is not updated when there is a new release ? [Official doc does not have the fix](https://www.tensorflow.org/programmers\_guide/datasets#applying\_arbitrary\_python\_logic\_with\_tfpy\_func)

-------------------------------------------------------------------------

2017-08-26T09:01:32Z bhack

@AMairesse I suggest you to notify this in https://github.com/tensorflow/tensorflow/issues/11786

-------------------------------------------------------------------------

2017-08-28T10:39:13Z biolee

need more operator for image process, like [map\_coordinates](https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.ndimage.interpolation.map\_coordinates.html), so We can build image augmentation pipe line only use tensorflow

And Dataset do not stably init variable defined in the map function as https://github.com/tensorflow/tensorflow/issues/12648

-------------------------------------------------------------------------

2017-08-28T13:45:18Z vvekic

I'd like to re-raise an earlier performance-related question by @kratzert that seems to have fallen out of focus. The performance gain of using the new Dataset API is negligible.

@ppwwyyxx stated that queues and StagingArea can still be used with the Dataset API, but I still haven't seen a working example of this. Do we have one?

What purpose does the new API serve if one must still include queues, data\_flow\_ops or StagingArea complexities?

-------------------------------------------------------------------------

2017-08-29T15:20:13Z GPhilo

@vvekic, I experimented a bit with queues and the Dataset API after realising in horror that of the 0.8s/step in my inference loop, 0.2s is data fetching (with GPU at 0% utilization), raising to almost 2 seconds if the HDD is being used by something else at the same time.

My pipeline looks as follows:

```

def preprocess\_image(fn):

im\_s = tf.read\_file(fn)

im = tf.image.decode\_jpeg(im\_s, channels=3)

im = inception\_preprocessing.preprocess\_for\_eval(im, width=299, height=299)

return fn, im

dataset = tf.contrib.data.Dataset.list\_files('{}/\*/\*.jpg'.format(FLAGS.dataset\_dir))

dataset.map(preprocess\_image, num\_threads=FLAGS.num\_threads)

iterator = dataset.make\_one\_shot\_iterator()

input\_queue = tf.FIFOQueue(capacity=100\*FLAGS.batch\_size,

dtypes = iterator.output\_types,

shapes=iterator.output\_shapes)

enqueue\_sample = input\_queue.enqueue(iterator.get\_next())

tf.train.add\_queue\_runner(tf.train.QueueRunner(input\_queue, [enqueue\_sample]\*FLAGS.num\_threads))

filenames, images = input\_queue.dequeue\_up\_to(FLAGS.batch\_size)

```

I still have to run this on a big dataset and check if there's any performance improvement, but at least it seems to execute correctly. The catch is, I couldn't find a way to iterate over the data more than once (which luckily enough is not my use-case), because the only iterator that won't raise an error when the `QueueRunner`s spawn the threads is the `one\_shot\_iterator`.

-------------------------------------------------------------------------

2017-08-30T06:44:07Z tocab

I don't know if I'm right here, but I have a question about the dataset API. My dataset contains one column with sequences and one with sequence length which i want treat different, because i want to pad the sequences. Is it possible to address a single column in the dataset so that it is treated different from the other column? E.g.:

```python

two\_column\_dataset = ... # This contains the column sequence and sequence length

first\_column\_dataset = two\_column\_dataset[0].padded\_batch(64, ...) # Pad only first column

second\_column\_dataset = two\_column\_dataset[1].batch(64) # Get corresponding sequence length for sequences

two\_column\_dataset = Dataset.zip((first\_column\_dataset, second\_column\_dataset))

```

Edit: After writing this, i found it out:

```python

def flat\_map\_func(sequence, sequence\_length):

first\_column\_dataset = Dataset.from\_tensors(sequence).padded\_batch(64, ...)

second\_column\_dataset = Dataset.from\_tensors(sequence\_length).padded\_batch(64)

zipped\_dataset = Dataset.zip((first\_column\_dataset, second\_column\_dataset))

return zipped\_dataset

two\_column\_dataset = two\_column\_dataset.flat\_map(flat\_map\_func)

```

-------------------------------------------------------------------------

2017-08-30T19:45:09Z mrry

This issue thread is becoming a bit unwieldy and it's getting hard to keep track of the individual discussions, so I'm going to lock it after responding to a few of the recent comments. Please feel free to open a new issue about any specific topics of feature requests related to `tf.contrib.data` and we can continue the discussion there.

In response to a few recent questions:

\* @GPhilo ([link](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-325698349)) and @kratzert ([link](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-308845375)): The Dataset API includes methods for prefetching, so it shouldn't be necessary to add a queue here, and you can retain the other advantages of Datasets (like reinitialization etc.). Passing `output\_buffer\_size=100 \* FLAGS.batch\_size` to the `dataset.map()` call, and following that with `dataset.batch(FLAGS.batch\_size)` will run your `preprocess\_image` function in parallel and should decently increase the performance.

```python

dataset = tf.contrib.data.Dataset.list\_files('{}/\*/\*.jpg'.format(FLAGS.dataset\_dir))

dataset = dataset.map(preprocess\_image, num\_threads=FLAGS.num\_threads,

output\_buffer\_size=100\*FLAGS.batch\_size)

dataset = datsaet.batch(FLAGS.batch\_size)

iterator = dataset.make\_one\_shot\_iterator()

filenames, images = iterator.get\_next()

```

Note that in TF 1.4 there will be a `Dataset.prefetch()` method that makes it easier to add prefetching at any point in the pipeline, not just after a `map()`. (You can try it by downloading the current nightly build.)

In reponse to @kratzert's [specific question](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-308845375) about the implementation, the `Dataset` and `Iterator` classes don't use TensorFlow's previous producer/consumer queues (such as `tf.FIFOQueue` or `tf.RandomShuffleQueue`), but they do include simpler (and more efficient) implementations of the core ideas. For example, `Dataset.prefetch()` will start a background thread to populate a ordered buffer that \*acts like\* a `tf.FIFOQueue`, so that downstream pipeline stages need not block. However, the [`prefetch()` implementation](https://github.com/tensorflow/tensorflow/blob/d236d19f7753ae23f91d794701efa70ace1629da/tensorflow/core/kernels/prefetch\_dataset\_op.cc) is much simpler, because it doesn't need to support as many different concurrent operations as a `tf.FIFOQueue`.

\* @vvekic ([link](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-325357396)): I'd be curious to see your code before and after trying the Dataset API, and perhaps you could follow up by opening an issue describing the performance bottleneck. Compared to feeding or a (non-`StagingArea`) queue-based pipeline, the new API should be more efficient, and I'd be curious to know which parts aren't!

At present, you're correct that the `StagingArea` functionality is not included in the Dataset API, and for peak performance in GPU workloads you will need to add a staging area manually. However, we are actively working on implementing Datasets that can span devices (see 19a55725af8102d72d4e081c5139f0e4bd5a4bb7 for some of the work in progress) and one of the first use cases for that is to support prefetching into GPU memory.

\* @tengerye ([link](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-323665406)): For dynamically feeding data into a Dataset, I'd suggest you try out the `Dataset.from\_generator()` method that we're adding to TF 1.4 (and which is available in nightly builds already). I answered @albertz's Stack Overflow question about doing this [here](https://stackoverflow.com/a/45928467/3574081). (Supporting distributed pipelines will depend on the cross-device Dataset support that I mentioned in the last answer, and we'll be implementing that soon.) I think this will also work for @rasmusbergpalm's [request](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-322407451), because you can create concurrent generators, and for @tillahoffmann's [request](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-321976369) and @sirfz's [request](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-321091444) as well. This API is very new though, so if you have any feedback, please let us know!

\* @jasonkriss ([link](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-303602263)) We've implemented something called "feedable" iterators, which let you switch the input for single graph between multiple iterators (e.g. one for training and one for testing). The programmers' guide has [more details](https://www.tensorflow.org/programmers\_guide/datasets#creating\_an\_iterator) about how to use this feature.

\* @guillaumekln ([link](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-308789560)) If you want to batch sequences with different lengths, you can use the `Dataset.group\_by\_window()` transformation. Have a look at [how this is used in the NMT model code](https://github.com/tensorflow/nmt/blob/04c8c04a8b4e805f3d0a9c42b4d17c85f1324c55/nmt/utils/iterator\_utils.py#L194) for an example.

Thanks again to all of you for your continued interest in this part of TensorFlow!

-------------------------------------------------------------------------

ple iterators (e.g. one for training and one for testing). The programmers' guide has [more details](https://www.tensorflow.org/programmers\_guide/datasets#creating\_an\_iterator) about how to use this feature.

\* @guillaumekln ([link](https://github.com/tensorflow/tensorflow/issues/7951#issuecomment-308789560)) If you want to batch sequences with different lengths, you can use the `Dataset.group\_by\_window()` transformation. Have a look at [how this is used in the NMT model code](https://github.com/tensorflow/nmt/blob/04c8c04a8b4e805f3d0a9c42b4d17c85f1324c55/nmt/utils/iterator\_utils.py#L194) for an example.

Thanks again to all of you for your continued interest in this part of TensorFlow!

-------------------------------------------------------------------------