Yaroslav Bulatov
Oct 28 · 5 min read

TensorFlow meets PyTorch with Eager execution.

One of the main user complaints about TensorFlow was the constraint imposed by having to structure your computations as a static graph.

Relaxing this requirement was one of my projects when I was at Google Brain, eventually open-sourced as imperative mode. However it relied on private/unstable APIs which became too costly to maintain over time.

Luckily, PyTorch coming out crystallized researcher needs/wants, and there has been a concerted effort to support this kind of mode as a first-class citizen.

It's still under active development but the version available in nightly release is quite usable, to try it out:

```
pip install tf-nightly-gpu
python
from tensorflow.contrib.eager.python import tfe
tfe.enable_eager_execution()
a = tf.random_uniform((10,))
b = tf.random_uniform((10,))
for i in range(100):
  a = a*a
  if a[0]>b[0]:
  break
print(i)
```

Note that there's no longer need to deal with graph or session and execution happens immediately.

To utilize GPU, copy tensors to the proper device first

```
a = a.gpu() # copies tensor to default GPU (GPU0)
a = a.gpu(0) # copies tensor to GPU0
a = a.gpu(1) # copies tensor to GPU1
a = a.cpu() # copies tensor back to CPU
```

transfers.py hosted with ❤ by GitHub view raw

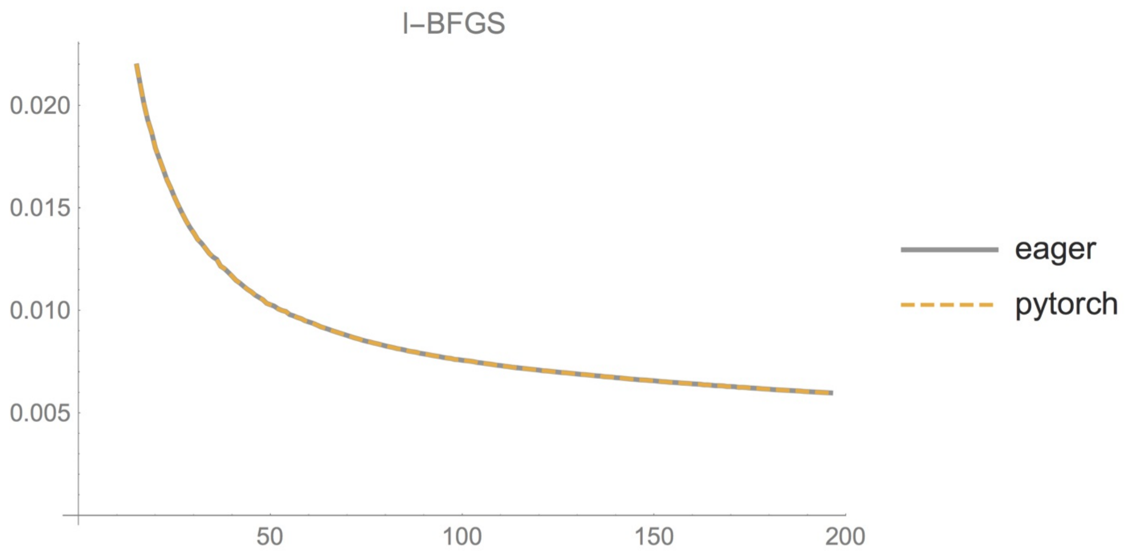
So what can you do with it?

Port imperative code

You can port an existing imperative code from numpy/pytorch/matlab by mechanically substituting correct API calls. IE

- torch.sum -> tf.reduce_sum"
- array.T -> tf.transpose(array)
- etc

I tried this as an exercise on PyTorch implementation of l-BFGS, and running two implementations side-by-side on GPU (PyTorch, Eager) gave me identical results to first 8 decimal digits on first try. This may be the most surprising thing to ever happen to me.



Use existing graph-based code

If your code doesn't rely on graph-specific API like graph_editor, you should be able to take existing code and run it with eager execution enabled.

There's also an experimental feature "graph_callable" that should enable you to use arbitrary TensorFlow subgraphs as a function that you can call. It's still in flux, but I was able to get an example working which wraps resnet_model from tensorflow/models as a graph_callable. Here's an example of training this model on a random batch.

Once this feature is ready it should also help with performance issues, see Performance section below.

Do more things with gradients.

There's new differentiation primitive tfe.gradients_function which mirrors autograd's grad. You can call "gradients_function" on an existing function "n" times to get "n"th derivative, ie

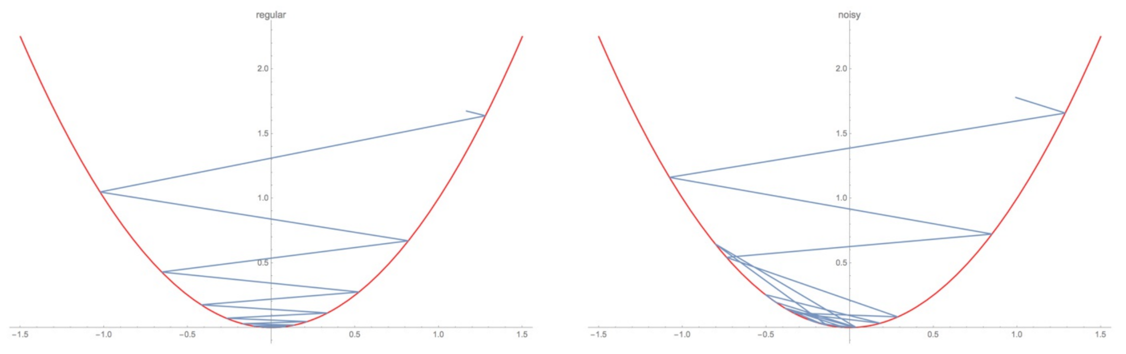
```
# expensive way to compute factorial of n
```

```python
def factorial(n):
  def f(x):
    return tf.pow(x, n)
  for i in range(n):
    f = tfe.gradients_function(f)
  return f(1.)
```

There's also a "custom_gradient" primitive which makes it much easier to create custom gradients. IE, suppose we wanted something like the square function, but which adds noise during backprop.

```python
@tfe.custom_gradient
def noisy_square(x):
   def grad(b):
       true_grad = 2*b*x
       return true_grad+tf.random_uniform(())
    return (x*x), grad
grad = tfe.gradients_function(noisy_square)
x = 2.
points = []
for i in range(20):
   x -= .9*grad(x)[0]
```

The result looks like this



You can see the second version has more trouble converging, but if it does converge, it'll generalize better!

This kind of gradient modification is useful for implementing advanced optimization algorithms like KFAC algorithm. Recall from my earlier explanation for PyTorch that KFAC for simple networks is equivalent to gradient descent where activation and backprop values are whitened.

This is equivalent to saying that gradient is transformed by multiplying it with whitening matrices on both sides

$$G \leftarrow (BB')^{-1}G(AA')^{-1}$$

$$G \to (BB\ ) \quad G(AA\ )$$

Suppose you've saved these matrices as m1, m2, your custom matmul would look like this:

```
@tfe.custom_gradient
def kfac_matmul(W, A):
    def grad(B):
        true_grad1 = B @ tf.transpose(A)
        true_grad2 = tf.transpose(W) @ B
     return [m1 @ true_grad1 @ m2, true_grad2]
 return W @ A, grad
```

kfac.py hosted with ❤ by **GitHub** **view raw**

Note, true_grad1, true_grad2 are the true backprops of matmul, see page 4 of Mike Giles "An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation"

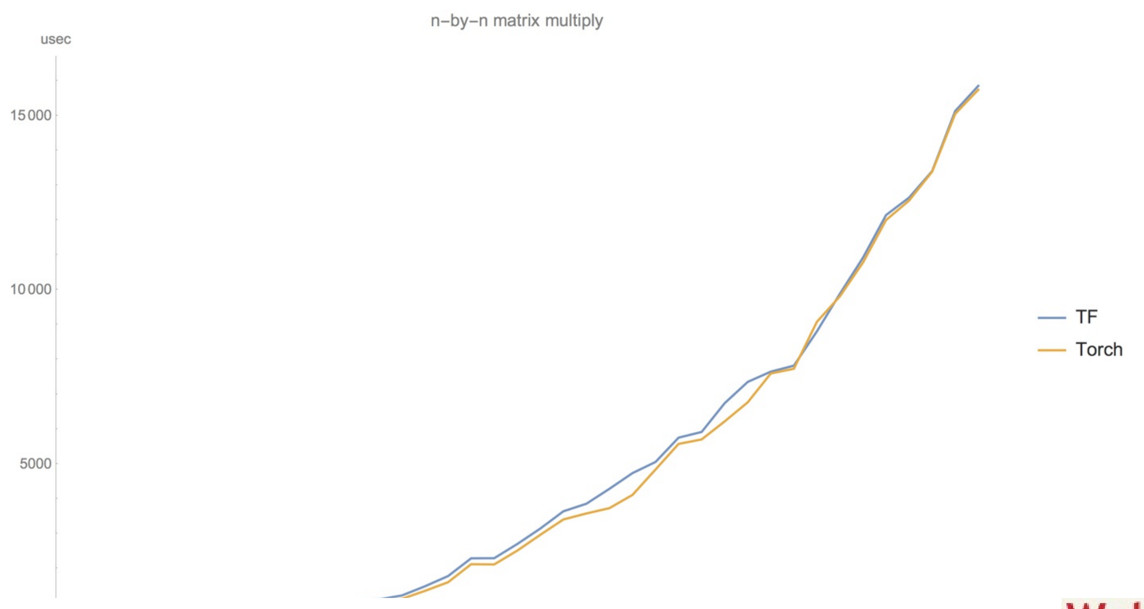
You can recover original KFAC by using kfac_matmul in place of tf.matmul and using Gradient Descent algorithm, or you could experiment with novel variations by using Momentum or Adam instead.

For an end-to-end example of KFAC that runs with Eager execution enabled , see this.

Performance

Whether eager execution makes your program is a little slower or a lot slower depends on how much of your computation is spent in high arithmetic intensity ops like conv or matmul.

IE, doing pure matrix multiplications (longer than 1 millisecond) is not much different whether you use TensorFlow eager, PyTorch or TensorFlow classic.

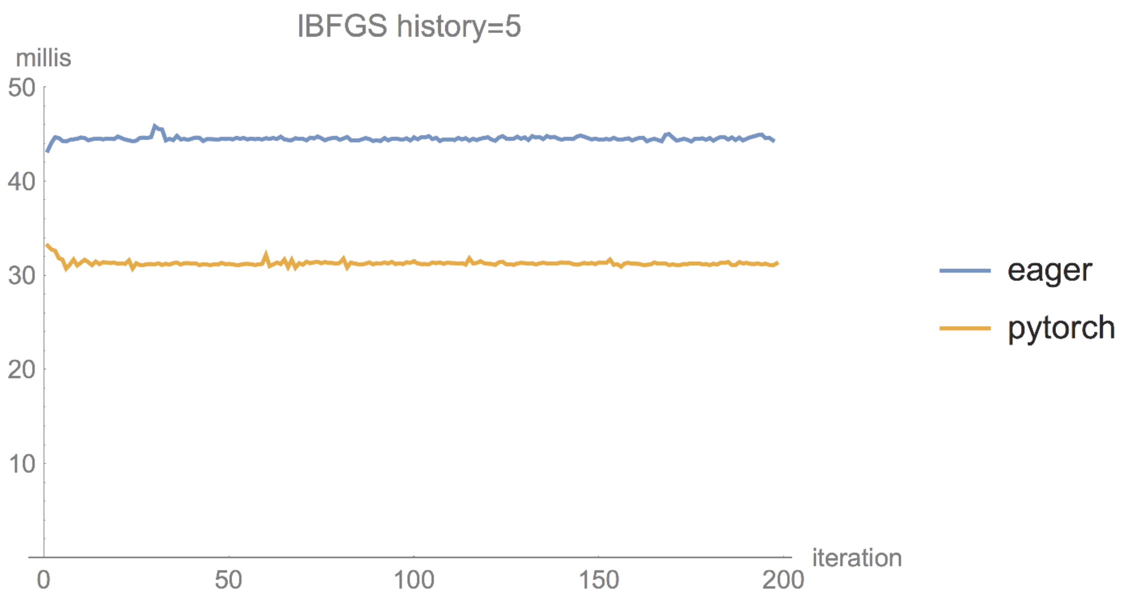


On other hand, end-to-end examples are more affected.

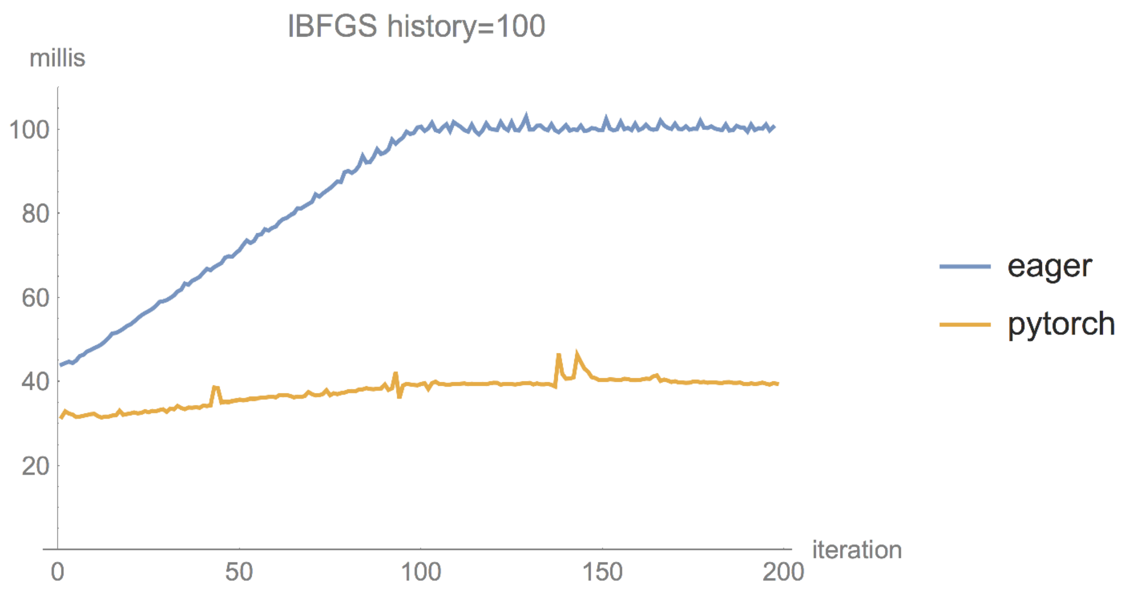
I was getting 20% slower than PyTorch in TF with eager execution when runtime was dominated by O(n^(1.5)) ops like matmul/conv ops, or 2–5 times slower on cases with a lot of O(n) ops like vector addition.

As a toy example, consider following Andrew Ng UFLDL example to train MNIST autoencoder.

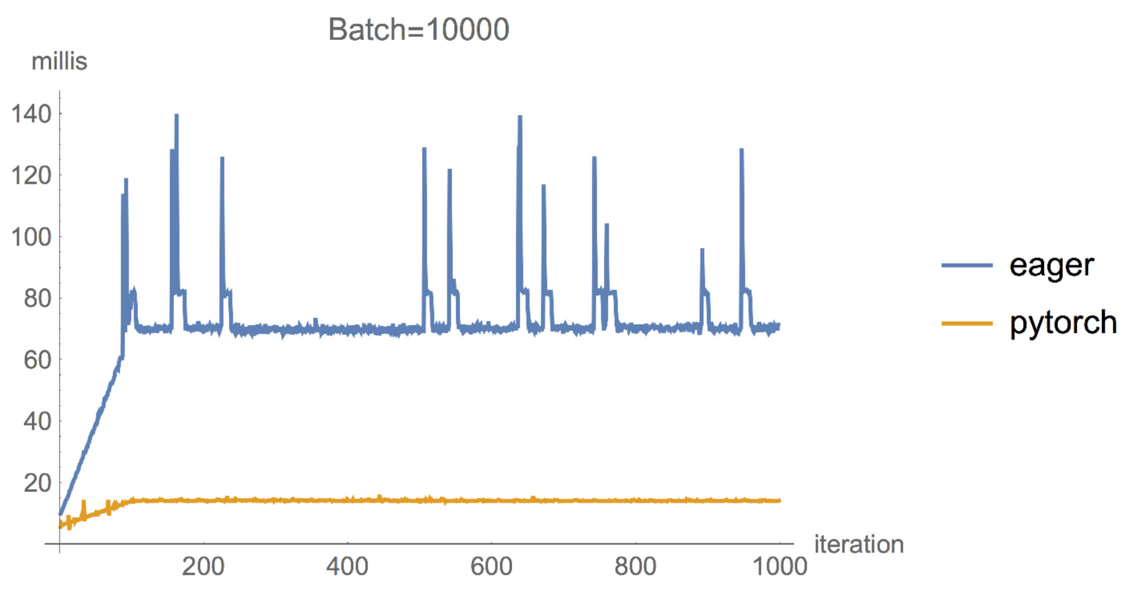
With batch-size 60k and l-BFGS history=5, the bulk of computation is spent in autoencoder forward pass, and Eager version is 1.4x slower.



With batch-size 60k and l-BFGS history=100, the two loops doing "two-step recursion" for each step of l-BFGS (dot products and vector adds) now go to 100, and Eager version now becomes 2.5x slower while PyTorch is only slightly affected.



Finally if we reduce batch size to 10k, we see that each iteration is 5x slower, occasionally spiking to 10x slower, probably due to garbage collection strategy.



Conclusion

While not as performant yet, this execution mode makes makes prototyping a lot easier. It's probably going to be the preferred starting mode for anyone building new computations in TF.