Sirena Follow
Feb 5 · 4 min read

Override Tensorflow Backward-Propagation

Tensorflow is a great tool that works with deep learning. There are a lot of operations that you easily can implement and make good model that solves needed problems. But sometimes there are models that have own specific computations.

I faced with one of them. When you create neural network architecture you can find that some operations do not flow in backward-propagation. On a piece of paper you can compute gradient and derive the formulas that are participated in backward-propagation, but Tensorflow due to its complexity cannot resolve the gradient and as a consequence you cannot train neural network.

As I said Tensorflow is a great tool and it proposes the ability to override gradient and write own custom gradient that can flow in backward-propagation. But how to do it and make it really works?

Firstly, I will explain it using simple network just for understanding this process. And then I will show real case where I had to have to override the gradient.

Simple network:

```
stddev = 1e-1
```

```
labels = [0, 1]
b = [1, 2]


labels_ph = tf.placeholder(shape=(2), dtype=tf.int32)
b_ph = tf.placeholder(shape=(2),dtype=tf.float32)
```

```
c = tf.Variable(tf.truncated_normal(shape=[2],
stddev=stddev))
k = tf.Variable(tf.truncated_normal(shape=[2],
stddev=stddev))
```

```
d = tf.add(b_ph, c)
e = tf.add(c, k)
a = tf.multiply(d, e)
```

```
loss = tf.losses.softmax_cross_entropy(logits=a,
onehot_labels=labels_ph)
```

```
opt = tf.train.AdamOptimizer(learning_rate=0.001,
beta1=0.9, beta2=0.999, epsilon=0.1)
```

Imagine that this part cannot be calculated in backpropagation—Tensorflow returns *None* for such gradients:

```
e = tf.add(c, k)
```

It means that it's not so obvious for Tensorflow how to compute the gradient in backpropagation. But you know that this is a differentiated model and on a piece of paper you can derive the formula. So what you need—just tell Tensorflow how to do it.

1. **tf.RegisterGradient** allows overrides the gradient: for this operation create a method where you setup the custom gradient.

```
def add_grad(c, k, d, graph, name=None):
    with tf.name_scope(name, "AddGrad", [c, k, d]) as name:
        return py_func(forward_func,
                       [c, k, d],
                       [np.float32],
                       graph,
                       name=name,
                       grad=backprop_func)
```

In **add_grad** you return the function that overrides gradient. The **py_func** looks like this:

```
def py_func(func, inp, Tout, graph, stateful=True,
name=None, grad=None):

    # Need to generate a unique name to avoid duplicates
```

```
    # if you have more than one custom gradients:
    rnd_name = 'PyFuncGrad'+ str(np.random.randint(0,
1E+8))

    tf.RegisterGradient(rnd_name)(grad)
    with graph.gradient_override_map({"PyFunc": rnd_name}):
        return tf.py_func(func, inp, Tout,
stateful=stateful, name=name)
```

As you see Tensorflow has own wrapper for python functions—**tf.py_func**. But at the same time it means that forward propagation you have to implement without Tensorflow operations. Because if you use Tensroflow operations in function that afterwards you pass to tf.py_func —it's going to fail.

I couldn't find the way how to implement forward propagation using Tensorflow operations, but I think it should be for having better performance. Please, share with me if you know.

2. Forward and Backprop methods:

a) What should be in forward pass (forward_func):

In **forward_func** you have to implement forward propagation using python operations.

```
def forward_func(c, k, d):
    e = np.add(c, k)
    return e.astype(np.float32)
```

As you see *d* variable is redundant in this method, but you cannot avoid using it, because it will be needed in backprop. It means that forward and backprop are at the same bunch—they share variables. You can notice that in **py_func** you declare all variables that are needed in both methods.

b) What you want to get in backpropagation (backprop_func):

Finally, in **backprop_func** we can implement the custom gradient. But custom gradient you do not pass to any wrapper, so you have to do it with Tensorflow operations.

```
def backprop_func(op, grad):
    c = op.inputs[0]
    k = op.inputs[1]
    d = op.inputs[2]
    e = tf.add(c, k)
    return (e + d) * grad , d * grad, e * grad
```

Notice, that we have to return gradient for each variable that was passed to that function—it is partial derivatives. I do not show how to calculate the gradient for each variable. Please do it by yourself.

op contains all passed variables; grad—the flown gradient from the back propagation.

3. Then explicitly call compute gradients and apply them when you initialize graph:

```
grads = opt.compute_gradients(loss,
tf.trainable_variables())
grads = list(grads)
train_op = opt.apply_gradients(grads_and_vars=grads)
```

And replace

```
e = tf.add(c, k)
```

with

```
e = add_grad(c, k, d)
```

The work is done. But to be sure that computations will be without errors, you can check firstly calculating on a peace of paper and then run code. For that example we can call **tf.add** instead of **add_grad** and

remember values, and then call **add_grad** to compare the outputs. I got the same result.

. . .

Real problem

In paper "Clothing Retrieval with Visual Attention Model" they describe attention network that generates Bernoulli series has to be multiplied with another feature map. Unfortunately Bernoulli is not differentiable, hence backward propagation will not flow.

Follow the above points I implement forward propagation that generates Bernoulli series with the given shape and in backprop function I implement custom gradient—it was just multiplication between intermediate layer (another feature map) and coming gradient.

Also there is **tf.stop_gradient**. It can be helpful in some cases. This operation allows to stop flow gradient further just for the given operation, it doesn't prevent backward-propagation altogether.

P.S. *I found the solution surfing the Stackoverflow resource, so it was done by collecting almost all the answers of this topic.*