TensorFlow checkpoint save and read

I have a TensorFlow based neural network and a set of variables.

The training function is like this:

```
def train(load = True, step)
    """
    Defining the neural network is skipped here
    """

    train_step = tf.train.AdamOptimizer(1e-4).minimize(mse)
    # Saver
    saver = tf.train.Saver()
```

```
    if not load:
        # Initalizing variables
        sess.run(tf.initialize_all_variables())
    else:
        saver.restore(sess, 'Variables/map.ckpt')
        print 'Model Restored!'

    # Perform stochastic gradient descent
    for i in xrange(step):
        train_step.run(feed_dict = {x: train, y_: label})

    # Save model
    save_path = saver.save(sess, 'Variables/map.ckpt')
    print 'Model saved in file: ', save_path
    print 'Training Done!'
```

I was calling the training function like this:

```
# First train
train(False, 1)
# Following train
for i in xrange(10):
    train(True, 10)
```

I did this kind of training because I needed to feed different dataset to my model. However, if I call the train function in this way TensorFlow will generate error message indicating that it cannot read the saved model from file.

After some experiments I found that this happened because the checkpoint saving was slow. Before the file was written to the disk the next train function would start reading, thus generate the error.

I have tried to use time.sleep() function to make some delay between each call but it didn't work.

Anyone knows how to work out this kind of write/read error? Thank you very much!

python io tensorflow

asked Dec 5 '15 at 23:32

yc2986

334 4 12

1 Answer

There is a subtle issue in your code: each time you call the `train()` function, more nodes are added to the same TensorFlow graph, for all the model variables and the rest of the neural network. This means that each time you construct a `tf.train.Saver()`, it includes all of the variables for the previous calls to `train()`. Each time you recreate the model, the variables are created with an extra `_N` suffix to give them a unique name:

1. Saver constructed with variables `var_a`, `var_b`.
2. Saver constructed with variables `var_a`, `var_b`, `var_a_1`, `var_b_1`.
3. Saver constructed with variables `var_a`, `var_b`, `var_a_1`, `var_b_1`, `var_a_2`, `var_b_2`.
4. etc.

The default behavior for a `tf.train.Saver` is to associate each variable with the name of the corresponding op. This means that `var_a_1` won't be initialized from `var_a`, because they end up with different names.

The solution is to create a new graph each time you call `train()`. The simplest way to fix it is to change your main program to create a new graph for each call to `train()` as follows:

```
# First train
with tf.Graph().as_default():
    train(False, 1)

# Following train
for i in xrange(10):
    with tf.Graph().as_default():
        train(True, 10)
```

...or, equivalently, you could move the `with` block inside the `train()` function.

answered Dec 7 '15 at 5:19

mrry
46.8k 3 112 162

So the behavior of adding nodes to the graph is something similar to the class/object thing in C++? Each time the train() function finish the graph object will not be destructed. If I continue to add variables with the same name like W1, b1 it will be switched to W1_1 and b1_1, thus make the load fail. Is my understanding right? Is this problem due to some destructor not called at the end of training process? Thank you! – yc2986 Dec 7 '15 at 20:12

Essentially, unless you explicitly construct a `tf.Graph` and set it as default using the `with` construct, all nodes will be added to a global graph that is only destroyed at the end of the process. (This is not ideal, but it makes some other use cases much easier.) Using the `with` block ensures that the graph is deregistered at the end of the block, which should give you the desired behavior—and avoid a memory leak! – mrry Dec 7 '15 at 21:12