TensorFlow variables, saving/restore”

Mar 8, 2017

Variables

*# Rank 0 tensor (scalar)*

fruit **=** tf**.**Variable("Orange", tf**.**string)

quantity **=** tf**.**Variable(2, tf**.**int16)

price **=** tf**.**Variable(3.23, tf**.**float32)

*# Rank 1 tensor*

strings **=** tf**.**Variable(["Fruit", "orange"], tf**.**string)

prices **=** tf**.**Variable([3.23, 4.02], tf**.**float64)

*# Rank 2 tensor*

answers **=** tf**.**Variable([[False, True],[False, False]], tf**.**bool)

When you train a model, we use variables to store training parameters like weight and bias, hyper parameters like learning rate, or state information like global step.

However, the best way to create a variable is using *tf.get\_variable*. It allows deep net to share parameters.

Initialize variables:

import tensorflow **as** tf

import numpy **as** np

v1 **=** tf**.**get\_variable("v1", [5, 5, 3]) *# A tensor with shape (5, 5, 3) filled with random values*

v2 **=** tf**.**get\_variable("v2", [5, 5, 3], dtype**=**tf**.**int32, trainable**=**True)

v3 **=** tf**.**get\_variable("v3", [3, 2], initializer**=**tf**.**zeros\_initializer) *# Set to 0*

v4 **=** tf**.**get\_variable("v4", [3, 2], initializer**=**tf**.**ones\_initializer) *# Set to 1*

v5 **=** tf**.**get\_variable("v5", initializer**=**tf**.**constant(2)) *# scalar: 2. float32.*

v6 **=** tf**.**get\_variable("v6", initializer**=**tf**.**constant([2])) *# [2]*

v7 **=** tf**.**get\_variable("v7", initializer**=**tf**.**constant([[2, 3], [4, 5]])) *# [[2, 3], [4, 5]]*

v8 **=** tf**.**get\_variable("v8", initializer**=**tf**.**constant(0.1, shape**=**[3, 2]))

*# [[ 1. 2.], [ 3. 4.], [ 5. 6.]]*

v9 **=** tf**.**get\_variable("v3", [3, 2], initializer**=**tf**.**constant\_initializer([1, 2, 3, 4, 5, 6]))

*# [[ 1. 2.], [ 2. 2.], [ 2. 2.]]*

v10 **=** tf**.**get\_variable("v4", [3, 2], initializer**=**tf**.**constant\_initializer([1, 2]))

Note: when we use *tf.constant* in *tf.get\_variable*, we do not need to specify the tensor shape unless we want to change the shape of the Tensor from the constant data. By default, variable is of type float32. *tf.get\_variable*assumes the variable is trainable.

Randomized the value of variables:

import tensorflow **as** tf

import numpy **as** np

W **=** tf**.**get\_variable("W", [784, 256], initializer**=**tf**.**truncated\_normal\_initializer(stddev**=**np**.**sqrt(2.0 **/** 784)))

Z **=** tf**.**get\_variable("z", [4, 5], initializer**=**tf**.**random\_uniform\_initializer(**-**1, 1))

The following program:

* Define variables and the initializers.
* Create op to update variable.
* Explicitly initialize the variables. (Always required)
* Retrieve a variable value.

import tensorflow **as** tf

*### Using variables*

*# Define variables and its initializer*

weights **=** tf**.**get\_variable("W", [784, 256], initializer**=**tf**.**truncated\_normal\_initializer(stddev**=**np**.**sqrt(2.0 **/** 784)))

biases **=** tf**.**get\_variable("z", [256], initializer**=**tf**.**zeros\_initializer)

counter **=** tf**.**get\_variable("counter", initializer**=**tf**.**constant(0))

*# Add an Op to increment a counter*

increment **=** tf**.**assign(counter , counter **+** 1)

init\_op **=** tf**.**global\_variables\_initializer()

**with** tf**.**Session() **as** sess:

*# Execute the init\_op to initialize all variables*

sess**.**run(init\_op)

*# Retrieve the value of a variable*

b **=** sess**.**run(biases)

**print**(b)

Save a Checkpoint

Variables can be saved to a disk during training. It can be reloaded to continue the training or to make inferences.

*# Create some variables*

v1 **=** tf**.**get\_variable("v1", shape**=**[3], initializer **=** tf**.**zeros\_initializer)

v2 **=** tf**.**get\_variable("v2", shape**=**[5], initializer **=** tf**.**zeros\_initializer)

*# Create the op*

inc\_v1 **=** v1**.**assign(v1**+**1)

dec\_v2 **=** v2**.**assign(v2**-**1)

init\_op **=** tf**.**global\_variables\_initializer()

*# Add ops to save and restore all the variables.*

saver **=** tf**.**train**.**Saver()

**with** tf**.**Session() **as** sess:

sess**.**run(init\_op)

inc\_v1**.**op**.**run()

dec\_v2**.**op**.**run()

*# Save the variables to disk.*

save\_path **=** saver**.**save(sess, "/tmp/model.ckpt")

Restore a checkpoint

*# Create some variables.*

*# We do not need to provide initializer or init\_op if it is restored from a checkpoint.*

v1 **=** tf**.**get\_variable("v1", shape**=**[3])

v2 **=** tf**.**get\_variable("v2", shape**=**[5])

saver **=** tf**.**train**.**Saver()

**with** tf**.**Session() **as** sess:

*# Restore variables from disk.*

saver**.**restore(sess, "/tmp/model.ckpt")

*# Check the values of the variables*

**print**("v1 : %s" **%** v1**.**eval())

**print**("v2 : %s" **%** v2**.**eval())

To save a subset of variables only.

v1 **=** tf**.**get\_variable("v1", [3], initializer **=** tf**.**zeros\_initializer)

v2 **=** tf**.**get\_variable("v2", [5], initializer **=** tf**.**zeros\_initializer)

*# Save only v2*

saver **=** tf**.**train**.**Saver({"v2": v2})

**with** tf**.**Session() **as** sess:

*# Initialize v1 since the saver will not.*

v1**.**initializer**.**run()

saver**.**restore(sess, "/tmp/model.ckpt")

Load a model and saving checkpoints regularly

This is the sample code in loading the model at the beginning and saves it occasionally during training.

import tensorflow **as** tf

import os

**def** **loadmodel**(session, saver, checkpoint\_dir):

session**.**run(tf**.**global\_variables\_initializer())

ckpt **=** tf**.**train**.**get\_checkpoint\_state(checkpoint\_dir)

**if** ckpt **and** ckpt**.**model\_checkpoint\_path:

ckpt\_name **=** os**.**path**.**basename(ckpt**.**model\_checkpoint\_path)

saver**.**restore(session, os**.**path**.**join(checkpoint\_dir, ckpt\_name))

**return** True

**else**:

**return** False

**def** **save**(session, saver, checkpoint\_dir, step):

dir **=** os**.**path**.**join(checkpoint\_dir, "model")

saver**.**save(session, dir, global\_step**=**step)

**with** tf**.**Session() **as** session:

saver **=** tf**.**train**.**Saver()

**...**

loadmodel(session, saver, "./checkpoint")

**...**

**for** i **in** range(10000):

**...**

**if** (i **%** 1000 **==** 0):

save(session, saver, "./checkpoint", i)

Trainable/Non-trainable parameters

In transfer learning, we may load a model from a checkpoint but freeze some of the layers during training by setting “trainable=False”.

freezed\_W = tf.get\_variable('CNN\_W!', [5, 5, 3, 32], trainable=False,

initializer=tf.truncated\_normal\_initializer(stddev=0.02))

...

loadmodel(session, saver, "./checkpoint")

In some problems, we may have multiple deep nets to be trained together. To have two different optimizers with different cost functions for different trainable parameters.

import tensorflow as tf

def scope\_variables(name):

with tf.variable\_scope(name):

return tf.get\_collection(tf.GraphKeys.GLOBAL\_VARIABLES,

scope=tf.get\_variable\_scope().name)

# Model parameters for the discriminator network

with tf.variable\_scope("discriminator"):

v1 = tf.get\_variable("v1", [3], initializer=tf.zeros\_initializer)

...

# Model parameters for the generator network

with tf.variable\_scope("generator"):

v2 = tf.get\_variable("v2", [2], initializer=tf.zeros\_initializer)

...

# Get all the trainable parameters for the discriminator

discriminator\_variables = scope\_variables("discriminator")

# Get all the trainable parameters for the generator

generator\_variables = scope\_variables("generator")

# 2 optimizers each for different networks

train\_discriminator = discriminator\_optimizer.minimize(d\_loss,

var\_list=discriminator\_variables)

train\_generator = generator\_optimizer.minimize(g\_loss,

var\_list=generator\_variables)

Scoping

We can use scoping such that we can create 2 different layers that have their own parameters from the same method. For example, *cnn1* and *cnn2* have their own ww and bb.

import tensorflow as tf

def conv2d(input, output\_dim, filter\_h=5, filter\_w=5, stride\_h=2, stride\_w=2, stddev=0.02):

w = tf.get\_variable('w', [filter\_h, filter\_w, input.get\_shape()[-1], output\_dim],

initializer=tf.truncated\_normal\_initializer(stddev=stddev))

conv = tf.nn.conv2d(input, w, strides=[1, stride\_h, stride\_w, 1], padding='SAME')

biases = tf.get\_variable('biases', [output\_dim], initializer=tf.constant\_initializer(0.0))

conv = tf.reshape(tf.nn.bias\_add(conv, biases), conv.get\_shape())

return conv

input1 = tf.random\_normal([1,10,10,32])

input2 = tf.random\_normal([1,20,20,32])

with tf.variable\_scope("conv1"):

cnn1 = conv2d(input1, 16)

with tf.variable\_scope("conv2"):

cnn1 = conv2d(input2, 16)

Variable sharing

Before looking into variable sharing, we first describe how *tf.Varaible* works. *tf.Variable* always create a new variable even given the same name.

*# tf.Variable always create new variable even given the same name.*

v1 **=** tf**.**Variable(10, name**=**"name1")

v2 **=** tf**.**Variable(10, name**=**"name1")

**assert**(v1 **is** **not** v2)

**print**(v1**.**name) *# name1:0*

**print**(v2**.**name) *# name1\_1:0*

If an operation named “name1” exist, the TensorFlow append “\_1”, “\_2” etc.. to the name to make it unique.

So calling the affine method twice below, we create 2 sets of W and b, i.e. 2 affine layers with their own set of W & b.

**def** **affine**(x, shape):

W **=** tf**.**Variable(tf**.**truncated\_normal(shape))

b **=** tf**.**Variable(tf**.**zeros([shape[1]]))

model **=** tf**.**nn**.**relu(tf**.**matmul(x, W) **+** b)

**return** model

x **=** tf**.**placeholder(tf**.**float32, [None, 784])

**with** tf**.**variable\_scope("n1"):

n1 **=** affine(x, [784, 500])

**with** tf**.**variable\_scope("n1"):

n2 **=** affine(x, [784, 500])

Sometimes, in a complex model, we want to share a common layer or parameters. How can we have a affine method similar to the code above but share the same W & b.

**def** **affine\_reuseable**(x, shape):

W **=** tf**.**get\_variable("W", shape,

initializer**=**tf**.**random\_normal\_initializer())

b **=** tf**.**get\_variable("b", [shape[1]],

initializer**=**tf**.**constant\_initializer(0.0))

model **=** tf**.**nn**.**relu(tf**.**matmul(x, W) **+** b)

**return** model

nx **=** tf**.**placeholder(tf**.**float32, [None, 784])

**with** tf**.**variable\_scope("n2"):

nn1 **=** affine\_reuseable(x, [784, 500])

**with** tf**.**variable\_scope("n2", reuse**=**True):

nn2 **=** affine\_reuseable(x, [784, 500])

If a variable with the give “scope/name” exists, *tf.get\_variable* returns the existing variable instead of creating one.

W **=** tf**.**get\_variable("W", shape, initializer**=**tf**.**random\_normal\_initializer())

So for the second affine\_reuseable call below, *tf.get\_variable* reuses the W & b variables created before.

**with** tf**.**variable\_scope("n2", reuse**=**True):

nn2 **=** affine\_reuseable(x, [784, 500])

Reuse

However, TensorFlow wants the developer to be self-aware of whether the variable exists or not. Developers need to have the correct setting for the “reuse” flag before calling *tf.get\_variable*. Both scenarios below will throw an **exception** when calling *tf.get\_variable*:

* if the reuse flag is False or None (default) and the variable already exists.
* if the reuse flag is True and the variable does not exist.

Do **NOT** do this

**with** tf**.**variable\_scope("foo"):

v **=** tf**.**get\_variable("v", [1])

v1 **=** tf**.**get\_variable("v")

*# Raises ValueError("... v already exists ...").*

**with** tf**.**variable\_scope("foo", reuse**=**True):

v **=** tf**.**get\_variable("v")

*# Raises ValueError("... v does not exists ...").*

Instead set the reuse flag probably.

**with** tf**.**variable\_scope("foo"):

v **=** tf**.**get\_variable("v2", [1]) *# Create a new variable.*

**with** tf**.**variable\_scope("foo", reuse**=**True):

v1 **=** tf**.**get\_variable("v2") *# reuse/share the variable "foo/v2".*

**assert** v1 **==** v

**with** tf**.**variable\_scope("foo") **as** scope:

v **=** tf**.**get\_variable("v3", [1])

scope**.**reuse\_variables()

v1 **=** tf**.**get\_variable("v3")

**assert** v1 **==** v

We can reuse *scope* instead of supplying the scope name again:

**with** tf**.**variable\_scope("model") **as** scope:

output1 **=** my\_image\_filter(input1)

**with** tf**.**variable\_scope(scope, reuse**=**True): *# Can use scope instead of "model"*

output2 **=** my\_image\_filter(input2)

Nested scope

**with** tf**.**variable\_scope("foo"):

**with** tf**.**variable\_scope("bar"):

v **=** tf**.**get\_variable("v", [1])

**assert** v**.**name **==** "foo/bar/v:0"

Caveat of variable sharing

Most developers are familiar with *tf.name\_scope* and *tf.Variables* methods. However, these APIs are NOT for shared variables. For example, *tf.get\_variable* below does not pick up the name scope created from *tf.name\_scope*.

**with** tf**.**name\_scope("foo1"):

v1 **=** tf**.**get\_variable("v", [1])

v2 **=** tf**.**Variable(1, name**=**"v2")

**with** tf**.**variable\_scope("foo2"):

v3 **=** tf**.**get\_variable("v", [1])

v4 **=** tf**.**Variable(1, name**=**"v2")

**print**(v1**.**name) *# v:0 (Unexpected!)*

**print**(v2**.**name) *# foo1/v2:0*

**print**(v3**.**name) *# foo2/v:0*

**print**(v4**.**name) *# foo2/v2:0*

The best way to avoid nasty issues with shared variables are

* Do **NOT** use *tf.name\_scope* and *tf.Variables* with shareable variables.
* Always use *tf.variable\_scope* to define the scope of a shared variable.
* Use *tf.get\_varaible* to create or retrieve a shared variable.

**with** tf**.**variable\_scope("foo"):

v **=** tf**.**get\_variable("v2", [1]) *# Create a new variable*

**with** tf**.**variable\_scope("foo", reuse**=**True):

v1 **=** tf**.**get\_variable("v2") *# Reuse a variable created before.*

Assignment

v **=** tf**.**get\_variable("v", shape**=**(), initializer**=**tf**.**zeros\_initializer())

v1 **=** v**.**assign\_add(1) *# 1.0*

v**.**assign(v1) *# 1.0*

**with** tf**.**Session() **as** session:

tf**.**global\_variables\_initializer()**.**run()

value, value1 **=** session**.**run([v, v1])

**print**(value, value1)

*# 1.0 1.0*