FREE eBook

LEARNING tensorflow

Free unaffiliated eBook created from
Stack Overflow contributors.

#tensorflow

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: tensorflow

It is an unofficial and free tensorflow ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official tensorflow.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with tensorflow

Remarks

This section provides an overview of what tensorflow is, and why a developer might want to use it.

It should also mention any large subjects within tensorflow, and link out to the related topics. Since the Documentation for tensorflow is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

As of Tensorflow version 1.0 installation has become much easier to perform. At minimum to install TensorFlow one needs pip installed on their machine with a python version of at least 2.7 or 3.3+.

```
pip install --upgrade tensorflow      # for Python 2.7
pip3 install --upgrade tensorflow     # for Python 3.n
```

For tensorflow on a GPU machine (as of 1.0 requires CUDA 8.0 and cudnn 5.1, AMD GPU not supported)

```
pip install --upgrade tensorflow-gpu  # for Python 2.7 and GPU
pip3 install --upgrade tensorflow-gpu # for Python 3.n and GPU
```

To test if it worked open up the correct version of python 2 or 3 and run

```
import tensorflow
```

If that succeeded without error then you have tensorflow installed on your machine.

---

*Be aware this references the master branch one can change this on the link above to reference the current stable release.)

Basic Example

Tensorflow is more than just a deep learning framework. It is a general computation framework to perform general mathematical operations in a parallel and distributed manner. An example of such is described below.

Linear Regression

A basic statistical example that is commonly utilized and is rather simple to compute is fitting a line to a dataset. The method to do so in tensorflow is described below in code and comments.

The main steps of the (TensorFlow) script are:

1. Declare placeholders (`x_ph`, `y_ph`) and variables (`W`, `b`)
2. Define the initialization operator (`init`)
3. Declare operations on the placeholders and variables (`y_pred`, `loss`, `train_op`)
4. Create a session (`sess`)
5. Run the initialization operator (`sess.run(init)`)
6. Run some graph operations (e.g. `sess.run([train_op, loss], feed_dict={x_ph: x, y_ph: y})`)

The graph construction is done using the Python TensorFlow API (could also be done using the C++ TensorFlow API). Running the graph will call low-level C++ routines.

```
'''
function: create a linear model which try to fit the line
          y = x + 2 using SGD optimizer to minimize
          root-mean-square(RMS) loss function

'''
import tensorflow as tf
import numpy as np

# number of epoch
num_epoch = 100

# training data x and label y
x = np.array([0., 1., 2., 3.], dtype=np.float32)
y = np.array([2., 3., 4., 5.], dtype=np.float32)

# convert x and y to 4x1 matrix
x = np.reshape(x, [4, 1])
y = np.reshape(y, [4, 1])

# test set(using a little trick)
x_test = x + 0.5
y_test = y + 0.5

# This part of the script builds the TensorFlow graph using the Python API

# First declare placeholders for input x and label y
# Placeholders are TensorFlow variables requiring to be explicitly fed by some
# input data
x_ph = tf.placeholder(tf.float32, shape=[None, 1])
y_ph = tf.placeholder(tf.float32, shape=[None, 1])

# Variables (if not specified) will be learnt as the GradientDescentOptimizer
# is run
# Declare weight variable initialized using a truncated_normal law
W = tf.Variable(tf.truncated_normal([1, 1], stddev=0.1))
# Declare bias variable initialized to a constant 0.1
b = tf.Variable(tf.constant(0.1, shape=[1]))
```

```
# Initialize variables just declared
init = tf.initialize_all_variables()

# In this part of the script, we build operators storing operations
# on the previous variables and placeholders.
# model: y = w * x + b
y_pred = x_ph * W + b

# loss function
loss = tf.mul(tf.reduce_mean(tf.square(tf.sub(y_pred, y_ph))), 1. / 2)
# create training graph
train_op = tf.train.GradientDescentOptimizer(0.1).minimize(loss)

# This part of the script runs the TensorFlow graph (variables and operations
# operators) just built.
with tf.Session() as sess:
    # initialize all the variables by running the initializer operator
    sess.run(init)
    for epoch in xrange(num_epoch):
        # Run sequentially the train_op and loss operators with
        # x_ph and y_ph placeholders fed by variables x and y
        _, loss_val = sess.run([train_op, loss], feed_dict={x_ph: x, y_ph: y})
        print('epoch %d: loss is %.4f' % (epoch, loss_val))

    # see what model do in the test set
    # by evaluating the y_pred operator using the x_test data
    test_val = sess.run(y_pred, feed_dict={x_ph: x_test})
    print('ground truth y is: %s' % y_test.flatten())
    print('predict y is     : %s' % test_val.flatten())
```

Tensorflow Basics

Tensorflow works on principle of dataflow graphs. To perform some computation there are two steps:

1. Represent the computation as a graph.
2. Execute the graph.

Representation: Like any directed graph a Tensorflow graph consists of nodes and directional edges.

Node: A Node is also called an Op(stands for operation). An node can have multiple incoming edges but single outgoing edge.

Edge: Indicate incoming or outgoing data from a Node. In this case input(s) and output of some Node(Op).

Whenever we say data we mean an n-dimensional vector known as Tensor. A Tensor has three properties: *Rank, Shape and Type*

- *Rank* means number of dimensions of the Tensor(a cube or box has rank 3).
- *Shape* means values of those dimensions(box can have shape 1x1x1 or 2x5x7).
- *Type* means datatype in each coordinate of Tensor.

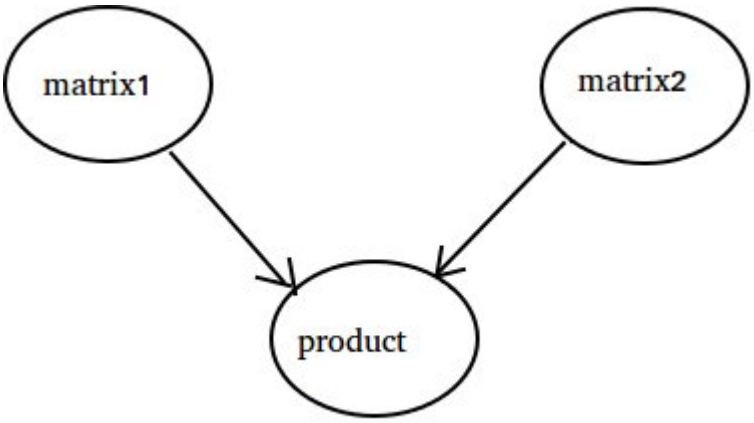
Execution: Even though a graph is constructed it is still an abstract entity. No computation actually occurs until we run it. To run a graph, we need to allocate CPU resource to Ops inside the graph. This is done using Tensorflow Sessions. Steps are:

1. Create a new session.
2. Run any Op inside the Graph. Usually we run the final Op where we expect the output of our computation.

An incoming edge on an Op is like a dependency for data on another Op. Thus when we run any Op, all incoming edges on it are traced and the ops on other side are also run.

Note: Special nodes called playing role of data source or sink are also possible. For example you can have an Op which gives a constant value thus no incoming edges(refer value 'matrix1' in the example below) and similarly Op with no outgoing edges where results are collected(refer value 'product' in the example below).

Example:




```
import tensorflow as tf

# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)

# Launch the default graph.
sess = tf.Session()

# To run the matmul op we call the session 'run()' method, passing 'product'
# which represents the output of the matmul op.  This indicates to the call
# that we want to get the output of the matmul op back.
```

```
#
# All inputs needed by the op are run automatically by the session.  They
# typically are run in parallel.
#
# The call 'run(product)' thus causes the execution of three ops in the
# graph: the two constants and matmul.
#
# The output of the op is returned in 'result' as a numpy `ndarray` object.
result = sess.run(product)
print(result)
# ==> [[ 12.]]

# Close the Session when we're done.
sess.close()
```

Counting to 10

In this example we use Tensorflow to count to 10. **Yes** this is total overkill, but it is a nice example to show an absolute minimal setup needed to use Tensorflow

```
import tensorflow as tf

# create a variable, refer to it as 'state' and set it to 0
state = tf.Variable(0)

# set one to a constant set to 1
one = tf.constant(1)

# update phase adds state and one and then assigns to state
addition = tf.add(state, one)
update = tf.assign(state, addition )

# create a session
with tf.Session() as sess:
  # initialize session variables
  sess.run( tf.global_variables_initializer() )

  print "The starting state is",sess.run(state)

  print "Run the update 10 times..."
  for count in range(10):
    # execute the update
    sess.run(update)

  print "The end state is",sess.run(state)
```

The important thing to realize here is that **state, one, addition, and update** don't actually contain values. Instead they are references to Tensorflow objects. The final result is not **state**, but instead is retrieved by using a Tensorflow to evaluate it using **sess.run(state)**

This example is from https://github.com/panchishin/learn-to-tensorflow . There are several other examples there and a nice graduated learning plan to get acquainted with manipulating the Tensorflow graph in python.

Read Getting started with tensorflow online: https://riptutorial.com/tensorflow/topic/856/getting-started-with-tensorflow

Chapter 2: Creating a custom operation with tf.py_func (CPU only)

Parameters

Parameter	Details
func	python function, which takes **numpy arrays** as its inputs and returns **numpy arrays** as its outputs
inp	list of Tensors (inputs)
Tout	list of tensorflow data types for the outputs of `func`

Examples

Basic example

The `tf.py_func(func, inp, Tout)` operator creates a TensorFlow operation that calls a Python function, `func` on a list of tensors `inp`.

See the documentation for `tf.py_func(func, inp, Tout)`.

Warning: The `tf.py_func()` operation will only run on CPU. If you are using distributed TensorFlow, the `tf.py_func()` operation must be placed on a CPU device **in the same process** as the client.

```
def func(x):
    return 2*x

x = tf.constant(1.)
res = tf.py_func(func, [x], [tf.float32])
# res is a list of length 1
```

Why to use tf.py_func

The `tf.py_func()` operator enables you to run arbitrary Python code in the middle of a TensorFlow graph. It is particularly convenient for wrapping custom NumPy operators for which no equivalent TensorFlow operator (yet) exists. Adding `tf.py_func()` is an alternative to using `sess.run()` calls inside the graph.

Another way of doing that is to cut the graph in two parts:

```
# Part 1 of the graph
inputs = ...  # in the TF graph
```

```
# Get the numpy array and apply func
val = sess.run(inputs)  # get the value of inputs
output_val = func(val)  # numpy array

# Part 2 of the graph
output = tf.placeholder(tf.float32, shape=...)
train_op = ...

# We feed the output_val to the tensor output
sess.run(train_op, feed_dict={output: output_val})
```

With `tf.py_func` this is much easier:

```
# Part 1 of the graph
inputs = ...

# call to tf.py_func
output = tf.py_func(func, [inputs], [tf.float32])[0]

# Part 2 of the graph
train_op = ...

# Only one call to sess.run, no need of a intermediate placeholder
sess.run(train_op)
```

Read Creating a custom operation with tf.py_func (CPU only) online:
https://riptutorial.com/tensorflow/topic/3856/creating-a-custom-operation-with-tf-py-func--cpu-only-

Chapter 3: Creating RNN, LSTM and bidirectional RNN/LSTMs with TensorFlow

Examples

Creating a bidirectional LSTM

```
import tensorflow as tf

dims, layers = 32, 2
# Creating the forward and backwards cells
lstm_fw_cell = tf.nn.rnn_cell.BasicLSTMCell(dims, forget_bias=1.0)
lstm_bw_cell = tf.nn.rnn_cell.BasicLSTMCell(dims, forget_bias=1.0)
# Pass lstm_fw_cell / lstm_bw_cell directly to tf.nn.bidirectional_rnn
# if only a single layer is needed
lstm_fw_multicell = tf.nn.rnn_cell.MultiRNNCell([lstm_fw_cell]*layers)
lstm_bw_multicell = tf.nn.rnn_cell.MultiRNNCell([lstm_bw_cell]*layers)

# tf.nn.bidirectional_rnn takes a list of tensors with shape
# [batch_size x cell_fw.state_size], so separate the input into discrete
# timesteps.
_X = tf.unpack(state_below, axis=1)
# state_fw and state_bw are the final states of the forwards/backwards LSTM, respectively
outputs, state_fw, state_bw = tf.nn.bidirectional_rnn(lstm_fw_multicell, lstm_bw_multicell,
_X, dtype='float32')
```

Parameters

- `state_below` is a 3D tensor of with the following dimensions: [`batch_size`, maximum sequence index, `dims`]. This comes from a previous operation, such as looking up a word embedding.
- `dims` is the number of hidden units.
- `layers` can be adjusted above 1 to create a *stacked LSTM network*.

Notes

- `tf.unpack` may not be able to determine the size of a given axis (use the `nums` argument if this is the case).
- It may be helpful to add an additional weight + bias multiplication beneath the LSTM (e.g. `tf.matmul(state_below, U) + b`.

Read Creating RNN, LSTM and bidirectional RNN/LSTMs with TensorFlow online: https://riptutorial.com/tensorflow/topic/4827/creating-rnn--lstm-and-bidirectional-rnn-lstms-with-tensorflow

Chapter 4: How to debug a memory leak in TensorFlow

Examples

Use Graph.finalize() to catch nodes being added to the graph

The most common mode of using TensorFlow involves first **building** a dataflow graph of TensorFlow operators (like `tf.constant()` and `tf.matmul()`, then **running steps** by calling the `tf.Session.run()` method in a loop (e.g. a training loop).

A common source of memory leaks is where the training loop contains calls that add nodes to the graph, and these run in every iteration, causing the graph to grow. These may be obvious (e.g. a call to a TensorFlow operator like `tf.square()`), implicit (e.g. a call to a TensorFlow library function that creates operators like `tf.train.Saver()`), or subtle (e.g. a call to an overloaded operator on a `tf.Tensor` and a NumPy array, which implicitly calls `tf.convert_to_tensor()` and adds a new `tf.constant()` to the graph).

The `tf.Graph.finalize()` method can help to catch leaks like this: it marks a graph as read-only, and raises an exception if anything is added to the graph. For example:

```
loss = ...
train_op = tf.train.GradientDescentOptimizer(0.01).minimize(loss)
init = tf.initialize_all_variables()

with tf.Session() as sess:
    sess.run(init)
    sess.graph.finalize()  # Graph is read-only after this statement.

    for _ in range(1000000):
        sess.run(train_op)
        loss_sq = tf.square(loss)  # Exception will be thrown here.
        sess.run(loss_sq)
```

In this case, the overloaded `*` operator attempts to add new nodes to the graph:

```
loss = ...
# ...
with tf.Session() as sess:
    # ...
    sess.graph.finalize()  # Graph is read-only after this statement.
    # ...
    dbl_loss = loss * 2.0  # Exception will be thrown here.
```

Use the tcmalloc allocator

To improve memory allocation performance, many TensorFlow users often use `tcmalloc` instead of the default `malloc()` implementation, as `tcmalloc` suffers less from fragmentation when allocating

and deallocating large objects (such as many tensors). Some memory-intensive TensorFlow programs have been known to leak **heap address space** (while freeing all of the individual objects they use) with the default `malloc()`, but performed just fine after switching to `tcmalloc`. In addition, `tcmalloc` includes a heap profiler, which makes it possible to track down where any remaining leaks might have occurred.

The installation for `tcmalloc` will depend on your operating system, but the following works on Ubuntu 14.04 (trusty) (where `script.py` is the name of your TensorFlow Python program):

```
$ sudo apt-get install google-perftools4
$ LD_PRELOAD=/usr/lib/libtcmalloc.so.4 python script.py ...
```

As noted above, simply switching to `tcmalloc` can fix a lot of apparent leaks. However, if the memory usage is still growing, you can use the heap profiler as follows:

```
$ LD_PRELOAD=/usr/lib/libtcmalloc.so.4 HEAPPROFILE=/tmp/profile python script.py ...
```

After you run the above command, the program will periodically write profiles to the filesystem. The sequence of profiles will be named:

- `/tmp/profile.0000.heap`
- `/tmp/profile.0001.heap`
- `/tmp/profile.0002.heap`
- ...

You can read the profiles using the `google-pprof` tool, which (for example, on Ubuntu 14.04) can be installed as part of the `google-perftools` package. For example, to look at the third snapshot collected above:

```
$ google-pprof --gv `which python` /tmp/profile.0002.heap
```

Running the above command will pop up a GraphViz window, showing the profile information as a directed graph.

Read How to debug a memory leak in TensorFlow online:
https://riptutorial.com/tensorflow/topic/3883/how-to-debug-a-memory-leak-in-tensorflow

Chapter 5: How to use TensorFlow Graph Collections?

Remarks

When you have huge model, it is useful to form some groups of tensors in your computational graph, that are connected with each other. For example tf.GraphKeys class contains such standart collections as:

```
tf.GraphKeys.VARIABLES
tf.GraphKeys.TRAINABLE_VARIABLES
tf.GraphKeys.SUMMARIES
```

Examples

Create your own collection and use it to collect all your losses.

Here we will create collection for losses of Neural Network's computational graph.

First create a computational graph like so:

```
with tf.variable_scope("Layer"):
    W = tf.get_variable("weights", [m, k],
        initializer=tf.zeros_initializer([m, k], dtype=tf.float32))
    b1 = tf.get_variable("bias", [k],
        initializer = tf.zeros_initializer([k], dtype=tf.float32))
    z = tf.sigmoid((tf.matmul(input, W) + b1))

    with tf.variable_scope("Softmax"):
        U = tf.get_variable("weights", [k, r],
            initializer=tf.zeros_initializer([k,r], dtype=tf.float32))
        b2 = tf.get_variable("bias", [r],
            initializer=tf.zeros_initializer([r], dtype=tf.float32))
    out = tf.matmul(z, U) + b2
cross_entropy = tf.reduce_mean(
    tf.nn.sparse_softmax_cross_entropy_with_logits(out, labels))
```

To create a new collection, you can simply start calling `tf.add_to_collection()` - the first call will create the collection.

```
tf.add_to_collection("my_losses",
    self.config.l2 * (tf.add_n([tf.reduce_sum(U ** 2), tf.reduce_sum(W ** 2)])))
tf.add_to_collection("my_losses", cross_entropy)
```

And finally you can get tensors from your collection:

```
loss = sum(tf.get_collection("my_losses"))
```

Note that `tf.get_collection()` returns a copy of the collection or an empty list if the collection does not exist. Also, it does NOT create the collection if it does not exist. To do so, you could use `tf.get_collection_ref()` which returns a reference to the collection and actually creates an empty one if it does not exist yet.

Collect variables from nested scopes

Below is a single hidden layer Multilayer Perceptron (MLP) using nested scoping of variables.

```
def weight_variable(shape):
    return tf.get_variable(name="weights", shape=shape,
                           initializer=tf.zeros_initializer(dtype=tf.float32))

def bias_variable(shape):
    return tf.get_variable(name="biases", shape=shape,
                           initializer=tf.zeros_initializer(dtype=tf.float32))

def fc_layer(input, in_dim, out_dim, layer_name):
    with tf.variable_scope(layer_name):
        W = weight_variable([in_dim, out_dim])
        b = bias_variable([out_dim])
        linear = tf.matmul(input, W) + b
        output = tf.sigmoid(linear)

with tf.variable_scope("MLP"):
    x = tf.placeholder(dtype=tf.float32, shape=[None, 1], name="x")
    y = tf.placeholder(dtype=tf.float32, shape=[None, 1], name="y")
    fc1 = fc_layer(x, 1, 8, "fc1")
    fc2 = fc_layer(fc1, 8, 1, "fc2")

mse_loss = tf.reduce_mean(tf.reduce_sum(tf.square(fc2 - y), axis=1))
```

The MLP uses the the top level scope name `MLP` and it has two layers with their respective scope names `fc1` and `fc2`. Each layer also has its own `weights` and `biases` variables.

The variables can be collected like so:

```
trainable_var_key = tf.GraphKeys.TRAINABLE_VARIABLES
all_vars = tf.get_collection(key=trainable_var_key, scope="MLP")
fc1_vars = tf.get_collection(key=trainable_var_key, scope="MLP/fc1")
fc2_vars = tf.get_collection(key=trainable_var_key, scope="MLP/fc2")
fc1_weight_vars = tf.get_collection(key=trainable_var_key, scope="MLP/fc1/weights")
fc1_bias_vars = tf.get_collection(key=trainable_var_key, scope="MLP/fc1/biases")
```

The values of the variables can be collected using the `sess.run()` command. For example if we would like to collect the values of the `fc1_weight_vars` after training, we could do the following:

```
sess = tf.Session()
# add code to initialize variables
# add code to train the network
# add code to create test data x_test and y_test

fc1_weight_vals = sess.run(fc1, feed_dict={x: x_test, y: y_test})
print(fc1_weight_vals)  # This should be an ndarray with ndim=2 and shape=[1, 8]
```

Read How to use TensorFlow Graph Collections? online:
https://riptutorial.com/tensorflow/topic/6902/how-to-use-tensorflow-graph-collections-

Chapter 6: Math behind 2D convolution with advanced examples in TF

Introduction

2D convolution is computed in a similar way one would calculate 1D convolution: you slide your kernel over the input, calculate the element-wise multiplications and sum them up. But instead of your kernel/input being an array, here they are matrices.

Examples

No padding, strides=1

This is the most basic example, with the easiest calculations. Let's assume your `input` and `kernel` are:

$$input = \begin{pmatrix} 4 & 3 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 1 & 2 & 4 & 1 \\ 3 & 1 & 0 & 2 \end{pmatrix} \qquad kernel = \begin{pmatrix} 1 & 0 & 1 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

When you your kernel you will receive the following output: $\begin{pmatrix} 14 & 6 \\ 6 & 12 \end{pmatrix}$, which is calculated in the following way:

- 14 = 4 * 1 + 3 * 0 + 1 * 1 + 2 * 2 + 1 * 1 + 0 * 0 + 1 * 0 + 2 * 0 + 4 * 1
- 6 = 3 * 1 + 1 * 0 + 0 * 1 + 1 * 2 + 0 * 1 + 1 * 0 + 2 * 0 + 4 * 0 + 1 * 1
- 6 = 2 * 1 + 1 * 0 + 0 * 1 + 1 * 2 + 2 * 1 + 4 * 0 + 3 * 0 + 1 * 0 + 0 * 1
- 12 = 1 * 1 + 0 * 0 + 1 * 1 + 2 * 2 + 4 * 1 + 1 * 0 + 1 * 0 + 0 * 0 + 2 * 1

TF's conv2d function calculates convolutions in batches and uses a slightly different format. For an input it is `[batch, in_height, in_width, in_channels]` for the kernel it is `[filter_height, filter_width, in_channels, out_channels]`. So we need to provide the data in the correct format:

```
import tensorflow as tf
k = tf.constant([
    [1, 0, 1],
    [2, 1, 0],
    [0, 0, 1]
], dtype=tf.float32, name='k')
i = tf.constant([
    [4, 3, 1, 0],
    [2, 1, 0, 1],
    [1, 2, 4, 1],
    [3, 1, 0, 2]
], dtype=tf.float32, name='i')
kernel = tf.reshape(k, [3, 3, 1, 1], name='kernel')
```

```
image  = tf.reshape(i, [1, 4, 4, 1], name='image')
```

Afterwards the convolution is computed with:

```
res = tf.squeeze(tf.nn.conv2d(image, kernel, [1, 1, 1, 1], "VALID"))
# VALID means no padding
with tf.Session() as sess:
   print sess.run(res)
```

And will be equivalent to the one we calculated by hand.

Some padding, strides=1

Padding is just a fancy name of telling: surround your input matrix with some constant. In most of the cases the constant is zero and this is why people call it zero padding. So if you want to use a padding of 1 in our original input (check the first example with `padding=0, strides=1`), the matrix will look like this:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 3 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 0 & 1 & 2 & 4 & 1 & 0 \\ 0 & 3 & 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

To calculate the values of the convolution you do the same sliding. Notice that in our case many values in the middle do not need to be recalculated (they will be the same as in previous example. I also will not show all the calculations here, because the idea is straight-forward. The result is:

$$\begin{pmatrix} 5 & 11 & 8 & 2 \\ 7 & 14 & 6 & 2 \\ 3 & 6 & 12 & 9 \\ 5 & 12 & 5 & 6 \end{pmatrix}$$

where

- 5 = 0 * 1 + 0 * 0 + 0 * 1 + 0 * 2 + 4 * 1 + 3 * 0 + 0 * 0 + 0 * 1 + 1 * 1
- ...
- 6 = 4 * 1 + 1 * 0 + 0 * 1 + 0 * 2 + 2 * 1 + 0 * 0 + 0 * 0 + 0 * 0 + 0 * 1

TF does not support an arbitrary padding in conv2d function, so if you need some padding that is not supported, use tf.pad(). Luckily for our input the padding 'SAME' will be equal to padding = 1. So we need to change almost nothing in our previous example:

```
res = tf.squeeze(tf.nn.conv2d(image, kernel, [1, 1, 1, 1], "SAME"))
# 'SAME' makes sure that our output has the same size as input and
# uses appropriate padding. In our case it is 1.
with tf.Session() as sess:
```

```
   print sess.run(res)
```

You can verify that the answer will be the same as calculated by hand.

Padding and strides (the most general case)

Now we will apply a strided convolution to our previously described padded example and calculate the convolution where `p = 1, s = 2`

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 3 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 0 & 1 & 2 & 4 & 1 & 0 \\ 0 & 3 & 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Previously when we used `strides = 1`, our slided window moved by 1 position, with `strides = s` it moves by `s` positions (you need to calculate `s^2` elements less. But in our case we can take a shortcut and do not perform any computations at all. Because we already computed the values for `s = 1`, in our case we can just grab each second element.

So if the solution is case of `s = 1` was

$$\begin{pmatrix} 5 & 11 & 8 & 2 \\ 7 & 14 & 6 & 2 \\ 3 & 6 & 12 & 9 \\ 5 & 12 & 5 & 6 \end{pmatrix}$$

in case of `s = 2` it will be:

$$\begin{pmatrix} 14 & 2 \\ 12 & 6 \end{pmatrix}$$

Check the positions of values 14, 2, 12, 6 in the previous matrix. The only change we need to perform in our code is to change the strides from 1 to 2 for width and height dimension (2-nd, 3-rd).

```
res = tf.squeeze(tf.nn.conv2d(image, kernel, [1, 2, 2, 1], "SAME"))
with tf.Session() as sess:
   print sess.run(res)
```

By the way, there is nothing that stops us from using different strides for different dimensions.

Read Math behind 2D convolution with advanced examples in TF online: https://riptutorial.com/tensorflow/topic/10015/math-behind-2d-convolution-with-advanced-examples-in-tf

Chapter 7: Matrix and Vector Arithmetic

Examples

Elementwise Multiplication

To perform elementwise multiplication on tensors, you can use either of the following:

- `a*b`
- `tf.multiply(a, b)`

Here is a full example of elementwise multiplication using both methods.

```
import tensorflow as tf
import numpy as np

# Build a graph
graph = tf.Graph()
with graph.as_default():
    # A 2x3 matrix
    a = tf.constant(np.array([[ 1, 2, 3],
                              [10,20,30]]),
                    dtype=tf.float32)
    # Another 2x3 matrix
    b = tf.constant(np.array([[2, 2, 2],
                              [3, 3, 3]]),
                    dtype=tf.float32)

    # Elementwise multiplication
    c =  a * b
    d = tf.multiply(a, b)

# Run a Session
with tf.Session(graph=graph) as session:
    (output_c, output_d) = session.run([c, d])
    print("output_c")
    print(output_c)
    print("\noutput_d")
    print(output_d)
```

Prints out the following:

```
output_c
[[  2.   4.   6.]
 [ 30.  60.  90.]]

output_d
[[  2.   4.   6.]
 [ 30.  60.  90.]]
```

Scalar Times a Tensor

In the following example a 2 by 3 tensor is multiplied by a scalar value (2).

```
# Build a graph
graph = tf.Graph()
with graph.as_default():
    # A 2x3 matrix
    a = tf.constant(np.array([[ 1, 2, 3],
                              [10,20,30]]),
                    dtype=tf.float32)

    # Scalar times Matrix
    c =  2 * a

# Run a Session
with tf.Session(graph=graph) as session:
    output = session.run(c)
    print(output)
```

This prints out

```
[[  2.   4.   6.]
 [ 20.  40.  60.]]
```

Dot Product

The dot product between two tensors can be performed using:

```
tf.matmul(a, b)
```

A full example is given below:

```
# Build a graph
graph = tf.Graph()
with graph.as_default():
    # A 2x3 matrix
    a = tf.constant(np.array([[1, 2, 3],
                              [2, 4, 6]]),
                            dtype=tf.float32)
    # A 3x2 matrix
    b = tf.constant(np.array([[1, 10],
                              [2, 20],
                              [3, 30]]),
                    dtype=tf.float32)

    # Perform dot product
    c = tf.matmul(a, b)

# Run a Session
with tf.Session(graph=graph) as session:
    output = session.run(c)
    print(output)
```

prints out

```
[[  14.  140.]
 [  28.  280.]]
```

Read Matrix and Vector Arithmetic online: https://riptutorial.com/tensorflow/topic/2953/matrix-and-vector-arithmetic

Chapter 8: Measure the execution time of individual operations

Examples

Basic example with TensorFlow's Timeline object

The `Timeline` object allows you to get the execution time for each node in the graph:

- you use a classic `sess.run()` but also specify the optional arguments `options` and `run_metadata`
- you then create a `Timeline` object with the `run_metadata.step_stats` data

Here is an example program that measures the performance of a matrix multiplication:

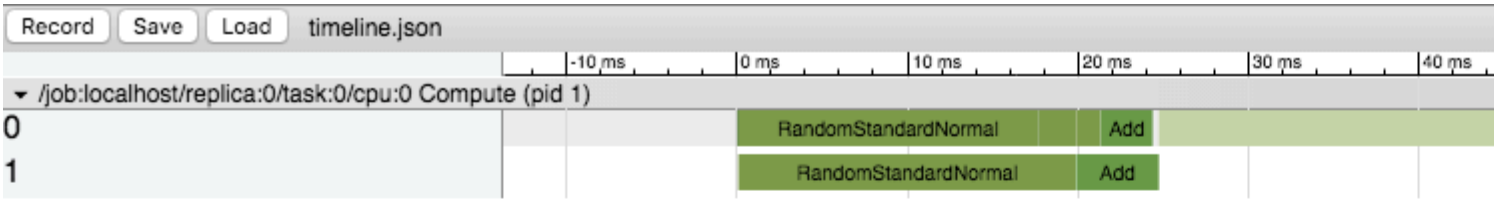
```
import tensorflow as tf
from tensorflow.python.client import timeline

x = tf.random_normal([1000, 1000])
y = tf.random_normal([1000, 1000])
res = tf.matmul(x, y)

# Run the graph with full trace option
with tf.Session() as sess:
    run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
    run_metadata = tf.RunMetadata()
    sess.run(res, options=run_options, run_metadata=run_metadata)

    # Create the Timeline object, and write it to a json
    tl = timeline.Timeline(run_metadata.step_stats)
    ctf = tl.generate_chrome_trace_format()
    with open('timeline.json', 'w') as f:
        f.write(ctf)
```

You can then open Google Chrome, go to the page `chrome://tracing` and load the `timeline.json` file. You should see something like:



Read Measure the execution time of individual operations online:
https://riptutorial.com/tensorflow/topic/3850/measure-the-execution-time-of-individual-operations

Chapter 9: Minimalist example code for distributed Tensorflow.

Introduction

This document shows how to create a cluster of TensorFlow servers, and how to distribute a computation graph across that cluster.

Examples

Distributed training example

```
import tensorflow as tf

FLAGS = None

def main(_):
    ps_hosts = FLAGS.ps_hosts.split(",")
    worker_hosts = FLAGS.worker_hosts.split(",")

    # Create a cluster from the parameter server and worker hosts.
    cluster = tf.train.ClusterSpec({"ps": ps_hosts, "worker": worker_hosts})

    # Create and start a server for the local task.
    server = tf.train.Server(cluster, job_name=FLAGS.job_name, task_index=FLAGS.task_index)

    if FLAGS.job_name == "ps":
        server.join()
    elif FLAGS.job_name == "worker":

        # Assigns ops to the local worker by default.
        with tf.device(tf.train.replica_device_setter(worker_device="/job:worker/task:%d" %
FLAGS.task_index, cluster=cluster)):

            # Build model...
            loss = ...
            global_step = tf.contrib.framework.get_or_create_global_step()

            train_op = tf.train.AdagradOptimizer(0.01).minimize(loss, global_step=global_step)

        # The StopAtStepHook handles stopping after running given steps.
        hooks=[tf.train.StopAtStepHook(last_step=1000000)]

        # The MonitoredTrainingSession takes care of session initialization,
        # restoring from a checkpoint, saving to a checkpoint, and closing when done
        # or an error occurs.
        with tf.train.MonitoredTrainingSession(master=server.target,
                                               is_chief=(FLAGS.task_index == 0),
                                               checkpoint_dir="/tmp/train_logs",
                                               hooks=hooks) as mon_sess:
            while not mon_sess.should_stop():
                # Run a training step asynchronously.
                # See `tf.train.SyncReplicasOptimizer` for additional details on how to
```

```
perform *synchronous* training.
               # mon_sess.run handles AbortedError in case of preempted PS.
               mon_sess.run(train_op)
```

Read Minimalist example code for distributed Tensorflow. online: https://riptutorial.com/tensorflow/topic/10950/minimalist-example-code-for-distributed-tensorflow-

Chapter 10: Multidimensional softmax

Examples

Creating a Softmax Output Layer

When `state_below` is a 2D Tensor, `U` is a 2D weights matrix, `b` is a `class_size`-length vector:

```
logits = tf.matmul(state_below, U) + b
return tf.nn.softmax(logits)
```

When `state_below` is a 3D tensor, `U`, `b` as before:

```
def softmax_fn(current_input):
    logits = tf.matmul(current_input, U) + b
    return tf.nn.softmax(logits)

raw_preds = tf.map_fn(softmax_fn, state_below)
```

Computing Costs on a Softmax Output Layer

Use `tf.nn.sparse_softmax_cross_entropy_with_logits`, but beware that it can't accept the output of `tf.nn.softmax`. Instead, calculate the unscaled activations, and then the cost:

```
logits = tf.matmul(state_below, U) + b
cost = tf.nn.sparse_softmax_cross_entropy_with_logits(logits, labels)
```

In this case: `state_below` and `U` should be 2D matrices, `b` should be a vector of a size equal to the number of classes, and `labels` should be a 2D matrix of `int32` or `int64`. This function also supports activation tensors with more than two dimensions.

Read Multidimensional softmax online:
https://riptutorial.com/tensorflow/topic/4999/multidimensional-softmax

Chapter 11: Placeholders

Parameters

Parameter	Details
data type (dtype)	specifically one of the data types provided by the tensorflow package. E.g. `tensorflow.float32`
data shape (shape)	Dimensions of placeholder as list or tuple. `None` can be used for dimensions that are unknown. E.g. (None,30) would define a (? x 30) dimension placeholder
name (name)	A name for the operation (optional).

Examples

Basics of Placeholders

Placeholders allow you to feed values into a tensorflow graph. Aditionally They allow you to specify constraints regarding the dimensions and data type of the values being fed in. As such they are useful when creating a neural network to feed new training examples.

The following example declares a placeholder for a 3 by 4 tensor with elements that are (or can be typecasted to) 32 bit floats.

```
a = tf.placeholder(tf.float32, shape=[3,4], name='a')
```

Placeholders will not contain any values on their own, so it is important to feed them with values when running a session otherwise you will get an error message. This can be done using the `feed_dict` argument when calling `session.run()`, eg:

```
# run the graph up to node b, feeding the placeholder `a` with values in my_array
session.run(b, feed_dict={a: my_array})
```

Here is a simple example showing the entire process of declaring and feeding a placeholer.

```
import tensorflow as tf
import numpy as np

# Build a graph
graph = tf.Graph()
with graph.as_default():
    # declare a placeholder that is 3 by 4 of type float32
    a = tf.placeholder(tf.float32, shape=(3, 4), name='a')

    # Perform some operation on the placeholder
```

```
    b = a * 2

# Create an array to be fed to `a`
input_array = np.ones((3,4))

# Create a session, and run the graph
with tf.Session(graph=graph) as session:
    # run the session up to node b, feeding an array of values into a
    output = session.run(b, feed_dict={a: input_array})
    print(output)
```

The placeholder takes a 3 by 4 array of ones, and that tensor is then multiplied by 2 at node b, wich then returns and prints out the following:

```
[[ 2.  2.  2.  2.]
 [ 2.  2.  2.  2.]
 [ 2.  2.  2.  2.]]
```

Placeholder with Default

Often one wants to intermittently run one or more validation batches during the course of training a deep network. Typically the training data are fed by a queue while the validation data might be passed through the `feed_dict` parameter in `sess.run()`. `tf.placeholder_with_default()` is designed to work well in this situation:

```
import numpy as np
import tensorflow as tf

IMG_SIZE = [3, 3]
BATCH_SIZE_TRAIN = 2
BATCH_SIZE_VAL = 1

def get_training_batch(batch_size):
    ''' training data pipeline '''
    image = tf.random_uniform(shape=IMG_SIZE)
    label = tf.random_uniform(shape=[])
    min_after_dequeue = 100
    capacity = min_after_dequeue + 3 * batch_size
    images, labels = tf.train.shuffle_batch(
        [image, label], batch_size=batch_size, capacity=capacity,
        min_after_dequeue=min_after_dequeue)
    return images, labels

# define the graph
images_train, labels_train = get_training_batch(BATCH_SIZE_TRAIN)
image_batch = tf.placeholder_with_default(images_train, shape=None)
label_batch = tf.placeholder_with_default(labels_train, shape=None)
new_images = tf.mul(image_batch, -1)
new_labels = tf.mul(label_batch, -1)

# start a session
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
```

```
    # typical training step where batch data are drawn from the training queue
    py_images, py_labels = sess.run([new_images, new_labels])
    print('Data from queue:')
    print('Images: ', py_images)  # returned values in range [-1.0, 0.0]
    print('\nLabels: ', py_labels) # returned values [-1, 0.0]

    # typical validation step where batch data are supplied through feed_dict
    images_val = np.random.randint(0, 100, size=np.hstack((BATCH_SIZE_VAL, IMG_SIZE)))
    labels_val = np.ones(BATCH_SIZE_VAL)
    py_images, py_labels = sess.run([new_images, new_labels],
                      feed_dict={image_batch:images_val, label_batch:labels_val})
    print('\n\nData from feed_dict:')
    print('Images: ', py_images) # returned values are integers in range [-100.0, 0.0]
    print('\nLabels: ', py_labels) # returned values are -1.0

    coord.request_stop()
    coord.join(threads)
```

In this example `image_batch` and `label_batch` are generated by `get_training_batch()` unless the corresponding values are passed as the `feed_dict` parameter during a call to `sess.run()`.

Read Placeholders online: https://riptutorial.com/tensorflow/topic/2952/placeholders

Chapter 12: Q-learning

Examples

Minimal Example

Q-learning is a variant of model-free reinforcement learning. In Q-learning we want the agent to estimate how good a (state, action) pair is so that it can choose good actions in each state. This is done by approximating an action-value function (Q) that fits in equation below:

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a')$$

Where *s* and *a* are state and action at current time step. *R* is the immediate reward and γ is discount factor. And, *s'* is the observed next state.

As the agent interacts with the environment, it sees a state that it is in, performs an action, gets the reward, and observes the new state that it has moved to. This cycle continues until the agent reaches a terminating state. Since Q-learning is an off-policy method, we can save each (state, action, reward, next_state) as an experience in a replay buffer. These experiences are sampled in each training iteration and used to improve our estimation of Q. Here is how:

1. From `next_state` calculate the Q value for next step by assuming that the agent greedily chooses an action in that state, hence the `np.max(next_state_value)` in the code below.
2. The Q value of next step is discounted and added to the immediate reward observed by the agent: (state, action, **reward**, state')
3. If a state-action result in termination of the episode, we use `Q = reward` instead of steps 1 and 2 above (episodic learning). So we need to also add termination flag to each experience that is being added to the buffer: (state, action, reward, next_state, terminated)
4. At this point, we have a Q value calculated from `reward` and `next_state` and also we have another Q value that is the output of the q-network function approximator. By changing the parameters of q-network function approximator using gradient descend and minimizing the difference between these two action values, the Q function approximator converges toward the true action values.

Here is an implementation of deep Q network.

```
import tensorflow as tf
import gym
import numpy as np

def fullyConnected(name, input_layer, output_dim, activation=None):
    """
    Adds a fully connected layer after the `input_layer`. `output_dim` is
    the size of next layer. `activation` is the optional activation
    function for the next layer.
    """
    initializer = tf.random_uniform_initializer(minval=-.003, maxval=.003)
```

```
    input_dims = input_layer.get_shape().as_list()[1:]
    weight = tf.get_variable(name + "_w", shape=[*input_dims, output_dim],
                             dtype=tf.float32, initializer=initializer)
    bias = tf.get_variable(name + "_b", shape=output_dim, dtype=tf.float32,
                           initializer=initializer)
    next_layer = tf.matmul(input_layer, weight) + bias

    if activation:
        next_layer = activation(next_layer, name=name + "_activated")

    return next_layer

class Memory(object):
    """
    Saves experiences as (state, action, reward, next_action,
    termination). It only supports discrete action spaces.
    """

    def __init__(self, size, state_dims):
        self.length = size

        self.states = np.empty([size, state_dims], dtype=float)
        self.actions = np.empty(size, dtype=int)
        self.rewards = np.empty((size, 1), dtype=float)
        self.states_next = np.empty([size, state_dims], dtype=float)
        self.terminations = np.zeros((size, 1), dtype=bool)

        self.memory = [self.states, self.actions,
                       self.rewards, self.states_next, self.terminations]

        self.pointer = 0
        self.count = 0

    def add(self, state, action, reward, next_state, termination):
        self.states[self.pointer] = state
        self.actions[self.pointer] = action
        self.rewards[self.pointer] = reward
        self.states_next[self.pointer] = next_state
        self.terminations[self.pointer] = termination
        self.pointer = (self.pointer + 1) % self.length
        self.count += 1

    def sample(self, batch_size):
        index = np.random.randint(
            min(self.count, self.length), size=(batch_size))
        return (self.states[index], self.actions[index],
            self.rewards[index], self.states_next[index],
            self.terminations[index])

class DQN(object):
    """
    Deep Q network agent.
    """

    def __init__(self, state_dim, action_dim, memory_size, layer_dims,
                 optimizer):

        self.action_dim = action_dim
        self.state = tf.placeholder(
            tf.float32, [None, state_dim], "states")
        self.action_ph = tf.placeholder(tf.int32, [None], "actions")
```

```
        self.action_value_ph = tf.placeholder(
            tf.float32, [None], "action_values")
        self.memory = Memory(memory_size, state_dim)

        def _make():
            flow = self.state
            for i, size in enumerate(layer_dims):
                flow = fullyConnected(
                    "layer%i" % i, flow, size, tf.nn.relu)

            return fullyConnected(
                "output_layer", flow, self.action_dim)

        # generate the learner network
        with tf.variable_scope('learner'):
            self.action_value = _make()
        # generate the target network
        with tf.variable_scope('target'):
            self.target_action_value = _make()

        # get parameters for learner and target networks
        from_list = tf.get_collection(
            tf.GraphKeys.TRAINABLE_VARIABLES, scope='learner')
        target_list = tf.get_collection(
            tf.GraphKeys.TRAINABLE_VARIABLES, scope='target')

        # create a copy operation from parameters of learner
        # to parameters of target network
        from_list = sorted(from_list, key=lambda v: v.name)
        target_list = sorted(target_list, key=lambda v: v.name)
        self.update_target_network = []
        for i in range(len(from_list)):
            self.update_target_network.append(target_list[i].assign(from_list[i]))

        # gather the action-values of the performed actions
        row = tf.range(0, tf.shape(self.action_value)[0])
        indexes = tf.stack([row, self.action_ph], axis=1)
        action_value = tf.gather_nd(self.action_value, indexes)

        # calculate loss of Q network
        self.single_loss = tf.square(action_value - self.action_value_ph)
        self._loss = tf.reduce_mean(self.single_loss)

        self.train_op = optimizer.minimize(self._loss)

    def train(self, session, batch=None, discount=.97):
        states, actions, rewards, next_states, terminals =\
            self.memory.sample(batch)
        next_state_value = session.run(
            self.target_action_value, {self.state: next_states})
        observed_value = rewards + discount * \
            np.max(next_state_value, 1, keepdims=True)
        observed_value[terminals] = rewards[terminals]

        _, batch_loss = session.run([self.train_op, self._loss], {
            self.state: states, self.action_ph: actions,
            self.action_value_ph: observed_value[:, 0]})
        return batch_loss

    def policy(self, session, state):
        return session.run(self.action_value, {self.state: [state]})[0]
```

```
    def memorize(self, state, action, reward, next_state, terminal):
        self.memory.add(state, action, reward, next_state, terminal)

    def update(self, session):
        session.run(self.update_target_network)
```

In deep Q network few mechanisms are used to improve the convergence of the agent. One is emphasis on *randomly* sampling the experiences from replay buffer to prevent any temporal relation between sampled experiences. Another mechanism is using target network in evaluation of the Q-value for `next_state`. The target network is similar the the learner network but its parameters are modified much less frequently. Also, the target network is not updated by the gradient descent, instead every once in a while its parameters are copied from the learner network.

The code below, is an example of this agent learning to perform actions in a cartpole environment.

```
ENVIRONMENT = 'CartPole-v1'  # environment name from `OpenAI`.
MEMORY_SIZE = 50000  # how many of recent time steps should be saved in agent's memory
LEARNING_RATE = .01  # learning rate for Adam optimizer
BATCH_SIZE = 8  # number of experiences to sample in each training step
EPSILON = .1  # how often an action should be chosen randomly. This encourages exploration
EPXILON_DECAY = .99  # the rate of decaying `EPSILON`
NETWORK_ARCHITECTURE = [100] # shape of the q network. Each element is one layer
TOTAL_EPISODES = 500  # number of total episodes
MAX_STEPS = 200  # maximum number of steps in each episode
REPORT_STEP = 10  # how many episodes to run before printing a summary

env = gym.make(ENVIRONMENT)  # initialize environment
state_dim = env.observation_space.shape[
    0]  # dimensions of observation space
action_dim = env.action_space.n

optimizer = tf.train.AdamOptimizer(LEARNING_RATE)
agent = DQN(state_dim, action_dim, MEMORY_SIZE,
            NETWORK_ARCHITECTURE, optimizer)

eps = [EPSILON]

def runEpisode(env, session):
    state = env.reset()
    total_l = 0.
    total_reward = 0.
    for i in range(MAX_STEPS):
        if np.random.uniform() < eps[0]:
            action = np.random.randint(action_dim)
        else:
            action_values = agent.policy(session, state)
            action = np.argmax(action_values)

        next_state, reward, terminated, _ = env.step(action)

        if terminated:
            reward = -1

        total_reward += reward

        agent.memorize(state, action, reward, next_state, terminated)
```

```
        state = next_state
        total_l += agent.train(session, BATCH_SIZE)

        if terminated:
            break

    eps[0] *= EPXILON_DECAY
    i += 1

    return i, total_reward / i, total_l / i

session = tf.InteractiveSession()
session.run(tf.global_variables_initializer())

for i in range(1, TOTAL_EPISODES + 1):
    leng, reward, loss = runEpisode(env, session)
    agent.update(session)
    if i % REPORT_STEP == 0:
        print(("Episode: %4i " +
               "| Episod Length: %3i " +
               "| Avg Reward: %+.3f " +
               "| Avg Loss: %6.3f " +
               "| Epsilon: %.3f") %
              (i, leng, reward, loss, eps[0]))
```

Read Q-learning online: https://riptutorial.com/tensorflow/topic/9967/q-learning

Chapter 13: Reading the data

Examples

Count examples in CSV file

```
import tensorflow as tf
filename_queue = tf.train.string_input_producer(["file.csv"], num_epochs=1)
reader = tf.TextLineReader()
key, value = reader.read(filename_queue)
col1, col2 = tf.decode_csv(value, record_defaults=[[0], [0]])

with tf.Session() as sess:
  sess.run(tf.initialize_local_variables())
  tf.train.start_queue_runners()
  num_examples = 0
  try:
    while True:
      c1, c2 = sess.run([col1, col2])
      num_examples += 1
  except tf.errors.OutOfRangeError:
    print "There are", num_examples, "examples"
```

`num_epochs=1` makes `string_input_producer` queue to close after processing each file on the list once. It leads to raising `OutOfRangeError` which is caught in `try:`. By default, `string_input_producer` produces the filenames infinitely.

`tf.initialize_local_variables()` is a tensorflow Op, which, when executed, initializes `num_epoch` *local* variable inside `string_input_producer`.

`tf.train.start_queue_runners()` start extra treads that handle adding data to the queues asynchronically.

Read & Parse TFRecord file

TFRecord files is the native tensorflow binary format for storing data (tensors). To read the file you can use a code similar to the CSV example:

```
import tensorflow as tf
filename_queue = tf.train.string_input_producer(["file.tfrecord"], num_epochs=1)
reader = tf.TFRecordReader()
key, serialized_example = reader.read(filename_queue)
```

Then, you need to parse the examples from `serialized_example` Queue. You can do it either using `tf.parse_example`, which requires previous batching, but is faster or `tf.parse_single_example`:

```
batch = tf.train.batch([serialized_example], batch_size=100)
parsed_batch = tf.parse_example(batch, features={
  "feature_name_1": tf.FixedLenFeature(shape=[1], tf.int64),
  "feature_name_2": tf.FixedLenFeature(shape=[1], tf.float32)
```

```
})
```

`tf.train.batch` joins consecutive values of given tensors of shape `[x, y, z]` to tensors of shape `[batch_size, x, y, z]`. `features` dict maps names of the features to tensorflow's definitions of features. You use `parse_single_example` in a similar way:

```
parsed_example = tf.parse_single_example(serialized_example, {
  "feature_name_1": tf.FixedLenFeature(shape=[1], tf.int64),
  "feature_name_2": tf.FixedLenFeature(shape=[1], tf.float32)
})
```

`tf.parse_example` and `tf.parse_single_example` return a dictionary mapping feature names to the tensor with the values.

To batch examples coming from `parse_single_example` you should extract the tensors from the dict and use `tf.train.batch` as before:

```
parsed_batch = dict(zip(parsed_example.keys(),
    tf.train.batch(parsed_example.values(), batch_size=100)
```

You read the data as before, passing the list of all the tensors to evaluate to `sess.run`:

```
with tf.Session() as sess:
  sess.run(tf.initialize_local_variables())
  tf.train.start_queue_runners()
  try:
    while True:
      data_batch = sess.run(parsed_batch.values())
      # process data
  except tf.errors.OutOfRangeError:
    pass
```

Random shuffling the examples

To randomly shuffle the examples, you can use `tf.train.shuffle_batch` function instead of `tf.train.batch`, as follows:

```
parsed_batch = tf.train.shuffle_batch([serialized_example],
    batch_size=100, capacity=1000,
    min_after_dequeue=200)
```

`tf.train.shuffle_batch` (as well as `tf.train.batch`) creates a `tf.Queue` and keeps adding `serialized_examples` to it.

`capacity` measures how many elements can be stored in Queue in one time. Bigger capacity leads to bigger memory usage, but lower latency caused by threads waiting to fill it up.

`min_after_dequeue` is the minimum number of elements present in the queue after getting elements from it. The `shuffle_batch` queue is not shuffling elements perfectly uniformly - it is designed with huge data, not-fitting-memory one, in mind. Instead, it reads between `min_after_dequeue` and `capacity`

elements, store them in memory and randomly chooses a batch of them. After that it enqueues some more elements, to keep its number between `min_after_dequeue` and `capacity`. Thus, the bigger value of `min_after_dequeue`, the more random elements are - the choice of `batch_size` elements is guaranteed to be taken from at least `min_after_dequeue` consecutive elements, but the bigger `capacity` has to be and the longer it takes to fill the queue initially.

Reading data for n epochs with batching

Assume your data examples are already read to a python's variable and you would like to read it n times, in batches of given size:

```
import numpy as np
import tensorflow as tf
data = np.array([1, 2, 3, 4, 5])
n = 4
```

To merge data in batches, possibly with random shuffling, you can use `tf.train.batch` or `tf.train.batch_shuffle`, but you need to pass to it the tensor that would produce whole data n times:

```
limited_tensor = tf.train.limit_epochs(data, n)
batch = tf.train.shuffle_batch([limited_tensor], batch_size=3, enqueue_many=True, capacity=4)
```

The `limit_epochs` converts the numpy array to tensor under the hood and returns a tensor producing it n times and throwing an OutOfRangeError afterwards. The `enqueue_many=True` argument passed to `shuffle_batch` denotes that each tensor in the tensor list `[limited_tensor]` should be interpreted as containing a number of examples. Note that capacity of the batching queue can be smaller than the number of examples in the tensor.

One can process the data as usual:

```
with tf.Session() as sess:
  sess.run(tf.initialize_local_variables())
  tf.train.start_queue_runners()
  try:
    while True:
      data_batch = sess.run(batch)
      # process data
  except tf.errors.OutOfRangeError:
    pass
```

How to load images and labels from a TXT file

It has not been explained in the Tensorflow documentation how to load images and labels directly from a TXT file. The code below illustrates how I achieved it. However, it does not mean that is the best way to do it and that this way will help in further steps.

For instance, I'm loading the labels in one single integer value {0,1} while the documentation uses a one-hot vector [0,1].

```
# Learning how to import images and labels from a TXT file
#
# TXT file format
#
# path/to/imagefile_1 label_1
# path/to/imagefile_2 label_2
# ...                 ...
#
# where label_X is either {0,1}

#Importing Libraries
import os
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.python.framework import ops
from tensorflow.python.framework import dtypes

#File containing the path to images and the labels [path/to/images label]
filename = '/path/to/List.txt'

#Lists where to store the paths and labels
filenames = []
labels = []

#Reading file and extracting paths and labels
with open(filename, 'r') as File:
    infoFile = File.readlines() #Reading all the lines from File
    for line in infoFile: #Reading line-by-line
        words = line.split() #Splitting lines in words using space character as separator
        filenames.append(words[0])
        labels.append(int(words[1]))

NumFiles = len(filenames)

#Converting filenames and labels into tensors
tfilenames = ops.convert_to_tensor(filenames, dtype=dtypes.string)
tlabels = ops.convert_to_tensor(labels, dtype=dtypes.int32)

#Creating a queue which contains the list of files to read and the value of the labels
filename_queue = tf.train.slice_input_producer([tfilenames, tlabels], num_epochs=10,
shuffle=True, capacity=NumFiles)

#Reading the image files and decoding them
rawIm= tf.read_file(filename_queue[0])
decodedIm = tf.image.decode_png(rawIm) # png or jpg decoder

#Extracting the labels queue
label_queue = filename_queue[1]

#Initializing Global and Local Variables so we avoid warnings and errors
init_op = tf.group(tf.local_variables_initializer() ,tf.global_variables_initializer())

#Creating an InteractiveSession so we can run in iPython
sess = tf.InteractiveSession()

with sess.as_default():
    sess.run(init_op)

    # Start populating the filename queue.
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord)
```

```
    for i in range(NumFiles): #length of your filenames list
        nm, image, lb = sess.run([filename_queue[0], decodedIm, label_queue])

        print image.shape
        print nm
        print lb

        #Showing the current image
        plt.imshow(image)
        plt.show()

    coord.request_stop()
    coord.join(threads)
```

Read Reading the data online: https://riptutorial.com/tensorflow/topic/6684/reading-the-data

Chapter 14: Save and Restore a Model in TensorFlow

Introduction

Tensorflow distinguishes between saving/restoring the current values of all the variables in a graph and saving/restoring the actual graph structure. To restore the graph, you are free to use either Tensorflow's functions or just call your piece of code again, that built the graph in the first place. When defining the graph, you should also think about which and how variables/ops should be retrievable once the graph has been saved and restored.

Remarks

In the restoring model section above if I understand correctly you build the model and then restore the variables. I believe rebuilding the model is not necessary so long as you add the relevant tensors/placeholders when saving using `tf.add_to_collection()`. For example:

```
tf.add_to_collection('cost_op', cost_op)
```

Then later you can restore the saved graph and get access to `cost_op` using

```
with tf.Session() as sess:
    new_saver = tf.train.import_meta_graph('model.meta')`
    new_saver.restore(sess, 'model')
    cost_op = tf.get_collection('cost_op')[0]
```

Even if you don't run `tf.add_to_collection()`, you can retrieve your tensors, but the process is a bit more cumbersome, and you may have to do some digging to find the right names for things. For example:

in a script that builds a tensorflow graph, we define some set of tensors `lab_squeeze`:

```
...
with tf.variable_scope("inputs"):
    y=tf.convert_to_tensor([[0,1],[1,0]])
    split_labels=tf.split(1,0,x,name='lab_split')
    split_labels=[tf.squeeze(i,name='lab_squeeze') for i in split_labels]
...
with tf.Session().as_default() as sess:
    saver=tf.train.Saver(sess,split_labels)
    saver.save("./checkpoint.chk")
```

we can recall them later on as follows:

```
with tf.Session() as sess:
    g=tf.get_default_graph()
```

```
    new_saver = tf.train.import_meta_graph('./checkpoint.chk.meta')`
    new_saver.restore(sess, './checkpoint.chk')
    split_labels=['inputs/lab_squeeze:0','inputs/lab_squeeze_1:0','inputs/lab_squeeze_2:0']

    split_label_0=g.get_tensor_by_name('inputs/lab_squeeze:0')
    split_label_1=g.get_tensor_by_name("inputs/lab_squeeze_1:0")
```

There are a number of ways to find the name of a tensor -- you can find it in your graph on tensor board, or you can search through for it with something like:

```
sess=tf.Session()
g=tf.get_default_graph()
...
x=g.get_collection_keys()
[i.name for j in x for i in g.get_collection(j)] # will list out most, if not all, tensors on
the graph
```

Examples

Saving the model

Saving a model in tensorflow is pretty easy.

Let's say you have a linear model with input `x` and want to predict an output `y`. The loss here is the mean square error (MSE). The batch size is 16.

```
# Define the model
x = tf.placeholder(tf.float32, [16, 10])  # input
y = tf.placeholder(tf.float32, [16, 1])   # output

w = tf.Variable(tf.zeros([10, 1]), dtype=tf.float32)

res = tf.matmul(x, w)
loss = tf.reduce_sum(tf.square(res - y))

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(loss)
```

Here comes the Saver object, which can have multiple parameters (cf. doc).

```
# Define the tf.train.Saver object
# (cf. params section for all the parameters)
saver = tf.train.Saver(max_to_keep=5, keep_checkpoint_every_n_hours=1)
```

Finally we train the model in a `tf.Session()`, for `1000` iterations. We only save the model every `100` iterations here.

```
# Start a session
max_steps = 1000
with tf.Session() as sess:
    # initialize the variables
```

```
    sess.run(tf.initialize_all_variables())

    for step in range(max_steps):
        feed_dict = {x: np.random.randn(16, 10), y: np.random.randn(16, 1)}  # dummy input
        _, loss_value = sess.run([train_op, loss], feed_dict=feed_dict)

        # Save the model every 100 iterations
        if step % 100 == 0:
            saver.save(sess, "./model", global_step=step)
```

After running this code, you should see the last 5 checkpoints in your directory:

- `model-500` and `model-500.meta`
- `model-600` and `model-600.meta`
- `model-700` and `model-700.meta`
- `model-800` and `model-800.meta`
- `model-900` and `model-900.meta`

Note that in this example, while the `saver` actually saves both the current values of the variables as a checkpoint and the structure of the graph (`*.meta`), no specific care was taken w.r.t how to retrieve e.g. the placeholders `x` and `y` once the model was restored. E.g. if the restoring is done anywhere else than this training script, it can be cumbersome to retrieve `x` and `y` from the restored graph (especially in more complicated models). To avoid that, always give names to your variables / placeholders / ops or think about using `tf.collections` as shown in one of the remarks.

Restoring the model

Restoring is also quite nice and easy.

Here's a handy helper function:

```
def restore_vars(saver, sess, chkpt_dir):
    """ Restore saved net, global score and step, and epsilons OR
    create checkpoint directory for later storage. """
    sess.run(tf.initialize_all_variables())

    checkpoint_dir = chkpt_dir

    if not os.path.exists(checkpoint_dir):
        try:
            print("making checkpoint_dir")
            os.makedirs(checkpoint_dir)
            return False
        except OSError:
            raise

    path = tf.train.get_checkpoint_state(checkpoint_dir)
    print("path = ",path)
    if path is None:
        return False
    else:
        saver.restore(sess, path.model_checkpoint_path)
        return True
```

Main code:

```
path_to_saved_model = './'
max_steps = 1

# Start a session
with tf.Session() as sess:

    ... define the model here ...

    print("define the param saver")
    saver = tf.train.Saver(max_to_keep=5, keep_checkpoint_every_n_hours=1)

    # restore session if there is a saved checkpoint
    print("restoring model")
    restored = restore_vars(saver, sess, path_to_saved_model)
    print("model restored ",restored)

    # Now continue training if you so choose

    for step in range(max_steps):

        # do an update on the model (not needed)
        loss_value = sess.run([loss])
        # Now save the model
        saver.save(sess, "./model", global_step=step)
```

Read Save and Restore a Model in TensorFlow online:
https://riptutorial.com/tensorflow/topic/5000/save-and-restore-a-model-in-tensorflow

Chapter 15: Save Tensorflow model in Python and load with Java

Introduction

Building and especially training a model may be easiest done in Python so how to you load and use the trained model in Java?

Remarks

The model can accept any number of inputs, so change the NUM_PREDICTIONS if you want to run more predictions than one. Realize that the Java is using JNI to call into the C++ tensorflow model, so you will see some info messages coming from the model when you run this.

Examples

Create and save a model with Python

```
import tensorflow as tf
# good idea
tf.reset_default_graph()

# DO MODEL STUFF
# Pretrained weighting of 2.0
W = tf.get_variable('w', shape=[], initializer=tf.constant(2.0), dtype=tf.float32)
# Model input x
x = tf.placeholder(tf.float32, name='x')
# Model output y = W*x
y = tf.multiply(W, x, name='y')

# DO SESSION STUFF
sess = tf.Session()
sess.run(tf.global_variables_initializer())

# SAVE THE MODEL
builder = tf.saved_model.builder.SavedModelBuilder("/tmp/model" )
builder.add_meta_graph_and_variables(
  sess,
  [tf.saved_model.tag_constants.SERVING]
)
builder.save()
```

Load and use the model in Java.

```
public static void main( String[] args ) throws IOException
{
    // good idea to print the version number, 1.2.0 as of this writing
    System.out.println(TensorFlow.version());
```

```
    final int NUM_PREDICTIONS = 1;

    // load the model Bundle
    try (SavedModelBundle b = SavedModelBundle.load("/tmp/model", "serve")) {

        // create the session from the Bundle
        Session sess = b.session();
        // create an input Tensor, value = 2.0f
        Tensor x = Tensor.create(
            new long[] {NUM_PREDICTIONS},
            FloatBuffer.wrap( new float[] {2.0f} )
        );

        // run the model and get the result, 4.0f.
        float[] y = sess.runner()
            .feed("x", x)
            .fetch("y")
            .run()
            .get(0)
            .copyTo(new float[NUM_PREDICTIONS]);

        // print out the result.
        System.out.println(y[0]);
    }
}
```

Read Save Tensorflow model in Python and load with Java online:
https://riptutorial.com/tensorflow/topic/10718/save-tensorflow-model-in-python-and-load-with-java

Chapter 16: Simple linear regression structure in TensorFlow with Python

Introduction

A model widely used in traditional statistics is the linear regression model. In this article, the objective is to follow the step-by-step implementation of this type of models. We are going to represent a simple linear regression structure.

For our study, we will analyze the age of the children on the **x** axis and the height of the children on the **y** axis. We will try to predict the height of the children, using their age, applying simple linear regression.[in TF finding the best W and b]

Parameters

Parameter	Description
train_X	np array with x dimension of information
train_Y	np array with y dimension of information

Remarks

I used TensorBoard sintaxis to track the behavior of some parts of the model, cost, train and activation elements.

```
with tf.name_scope("") as scope:
```

Imports used:

```
import numpy as np
import tensorflow as tf
```

Type of application and language used:

> I have used a traditional console implementation app type, developed in Python, to represent the example.

Version of TensorFlow used:

> 1.0.1

Conceptual **academic** example/reference extracted from here:

Examples

Simple regression function code structure

Function definition:

```
def run_training(train_X, train_Y):
```

Inputs variables:

```
X = tf.placeholder(tf.float32, [m, n])
Y = tf.placeholder(tf.float32, [m, 1])
```

Weight and bias representation

```
W = tf.Variable(tf.zeros([n, 1], dtype=np.float32), name="weight")
b = tf.Variable(tf.zeros([1], dtype=np.float32), name="bias")
```

Lineal Model:

```
with tf.name_scope("linear_Wx_b") as scope:
    activation = tf.add(tf.matmul(X, W), b)
```

Cost:

```
with tf.name_scope("cost") as scope:
    cost = tf.reduce_sum(tf.square(activation - Y)) / (2 * m)
    tf.summary.scalar("cost", cost)
```

Training:

```
with tf.name_scope("train") as scope:
    optimizer = tf.train.GradientDescentOptimizer(0.07).minimize(cost)
```

TensorFlow session:

```
with tf.Session() as sess:
    merged = tf.summary.merge_all()
    writer = tf.summary.FileWriter(log_file, sess.graph)
```

Note: **merged** and writer are part of the TensorBoard strategy to track the model behavior.

```
    init = tf.global_variables_initializer()
    sess.run(init)
```

Repeating 1.5k times the training loop:

```
    for step in range(1500):
       result, _ = sess.run([merged, optimizer], feed_dict={X: np.asarray(train_X), Y:
np.asarray(train_Y)})
       writer.add_summary(result, step)
```

Print Training Cost:

```
    training_cost = sess.run(cost, feed_dict={X: np.asarray(train_X), Y: np.asarray(train_Y)})
    print "Training Cost: ", training_cost, "W=", sess.run(W), "b=", sess.run(b), '\n'
```

Concrete prediction based on the model trained:

```
    print "Prediction for 3.5 years"
    predict_X = np.array([3.5], dtype=np.float32).reshape([1, 1])

    predict_X = (predict_X - mean) / std
    predict_Y = tf.add(tf.matmul(predict_X, W), b)
    print "Child height(Y) =", sess.run(predict_Y)
```

Main Routine

```
def main():
    train_X, train_Y = read_data()
    train_X = feature_normalize(train_X)
    run_training(train_X, train_Y)
```

Note: remember review functions dependencies. **read_data**, **feature_normalize** and **run_training**

Normalization Routine

```
def feature_normalize(train_X):
    global mean, std
    mean = np.mean(train_X, axis=0)
    std = np.std(train_X, axis=0)

    return np.nan_to_num((train_X - mean) / std)
```

Read Data routine

```
def read_data():
    global m, n
```

```
    m = 50
    n = 1

    train_X = np.array(
```

Internal data for the array

```
    ).astype('float32')

    train_Y = np.array(
```

Internal data for the array

```
    ).astype('float32')

    return train_X, train_Y
```

Read Simple linear regression structure in TensorFlow with Python online: https://riptutorial.com/tensorflow/topic/9954/simple-linear-regression-structure-in-tensorflow-with-python

Chapter 17: Tensor indexing

Introduction

Various examples showing how Tensorflow supports indexing into tensors, highlighting differences and similarities to numpy-like indexing where possible.

Examples

Extract a slice from a tensor

Refer to the `tf.slice(input, begin, size)` documentation for detailed information.

Arguments:

- `input`: Tensor
- `begin`: starting location for each dimension of `input`
- `size`: number of elements for each dimension of `input`, using `-1` includes all remaining elements

---

Numpy-like slicing:

```
# x has shape [2, 3, 2]
x = tf.constant([[[1., 2.], [3., 4. ], [5. , 6. ]],
                 [[7., 8.], [9., 10.], [11., 12.]]])

# Extracts x[0, 1:2, :] == [[[ 3.,  4.]]]
res = tf.slice(x, [0, 1, 0], [1, 1, -1])
```

Using negative indexing, to retrieve the last element in the third dimension:

```
# Extracts x[0, :, -1:] == [[[2.], [4.], [6.]]]
last_indice = x.get_shape().as_list()[2] - 1
res = tf.slice(x, [0, 1, last_indice], [1, -1, -1])
```

Extract non-contiguous slices from the first dimension of a tensor

Generally `tf.gather` gives you access to elements in the first dimension of a tensor (e.g. rows 1, 3 and 7 in a 2-dimensional Tensor). If you need access to any other dimension than the first one, or if you don't need the whole slice, but e.g. only the 5th entry in the 1st, 3rd and 7th row, you are better off using `tf.gather_nd` (see upcoming example for this).

`tf.gather` arguments:

- `params`: A tensor you want to extract values from.
- `indices`

: A tensor specifying the indices pointing into `params`

Refer to the tf.gather(params, indices) documentation for detailed information.

We want to extract the 1st and 4th row in a 2-dimensional tensor.

```
# data is [[0, 1, 2, 3, 4, 5],
#          [6, 7, 8, 9, 10, 11],
#          ...
#          [24, 25, 26, 27, 28, 29]]
data = np.reshape(np.arange(30), [5, 6])
params = tf.constant(data)
indices = tf.constant([0, 3])
selected = tf.gather(params, indices)
```

`selected` has shape `[2, 6]` and printing its value gives

```
[[ 0  1  2  3  4  5]
 [18 19 20 21 22 23]]
```

`indices` can also just be a scalar (but cannot contain negative indices). E.g. in the above example:

```
tf.gather(params, tf.constant(3))
```

would print

```
[18 19 20 21 22 23]
```

Note that `indices` can have any shape, but the elements stored in `indices` always only refer to the *first* dimension of `params`. E.g. if you want to retrieve both the 1st and 3rd row *and* the 2nd and 4th row at the same time, you can do this:

```
indices = tf.constant([[0, 2], [1, 3]])
selected = tf.gather(params, indices)
```

Now `selected` will have shape `[2, 2, 6]` and its content reads:

```
[[[ 0  1  2  3  4  5]
  [12 13 14 15 16 17]]

 [[ 6  7  8  9 10 11]
  [18 19 20 21 22 23]]]
```

You can use `tf.gather` to compute a permutation. E.g. the following reverses all rows of `params`:

```
indices = tf.constant(list(range(4, -1, -1)))
selected = tf.gather(params, indices)
```

`selected` is now

```
[[24 25 26 27 28 29]
 [18 19 20 21 22 23]
 [12 13 14 15 16 17]
 [ 6  7  8  9 10 11]
 [ 0  1  2  3  4  5]]
```

If you need access to any other than the first dimension, you could work around that using `tf.transpose`: E.g. to gather columns instead of rows in our example, you could do this:

```
indices = tf.constant([0, 2])
selected = tf.gather(tf.transpose(params, [1, 0]), indices)
selected_t = tf.transpose(selected, [1, 0])
```

`selected_t` is of shape `[5, 2]` and reads:

```
[[ 0  2]
 [ 6  8]
 [12 14]
 [18 20]
 [24 26]]
```

However, `tf.transpose` is rather expensive, so it might be better to use `tf.gather_nd` for this use case.

Numpy-like indexing using tensors

This example is based on this post: TensorFlow - numpy-like tensor indexing.

In Numpy you can use arrays to index into an array. E.g. in order to select the elements at `(1, 2)` and `(3, 2)` in a 2-dimensional array, you can do this:

```
# data is [[0, 1, 2, 3, 4, 5],
#          [6, 7, 8, 9, 10, 11],
#          [12 13 14 15 16 17],
#          [18 19 20 21 22 23],
#          [24, 25, 26, 27, 28, 29]]
data = np.reshape(np.arange(30), [5, 6])
a = [1, 3]
b = [2, 2]
selected = data[a, b]
print(selected)
```

This will print:

```
[ 8 20]
```

To get the same behaviour in Tensorflow, you can use `tf.gather_nd`, which is an extension of `tf.gather`. The above example can be written like this:

```
x = tf.constant(data)
idx1 = tf.constant(a)
idx2 = tf.constant(b)
result = tf.gather_nd(x, tf.stack((idx1, idx2), -1))

with tf.Session() as sess:
    print(sess.run(result))
```

This will print:

```
[ 8 20]
```

`tf.stack` is the equivalent of `np.asarray` and in this case stacks the two index vectors along the last dimension (which in this case is the 1st) to produce:

```
[[1 2]
 [3 2]]
```

How to use tf.gather_nd

`tf.gather_nd` is an extension of `tf.gather` in the sense that it allows you to not only access the 1st dimension of a tensor, but potentially all of them.

Arguments:

- `params`: a Tensor of rank `P` representing the tensor we want to index into
- `indices`: a Tensor of rank `Q` representing the indices into `params` we want to access

The output of the function depends on the shape of `indices`. If the innermost dimension of `indices` has length `P`, we are collecting single elements from `params`. If it is less than `P`, we are collecting slices, just like with `tf.gather` but without the restriction that we can only access the 1st dimension.

Collecting elements from a tensor of rank 2

To access the element at `(1, 2)` in a matrix, we can use:

```
# data is [[0, 1, 2, 3, 4, 5],
#          [6, 7, 8, 9, 10, 11],
#          [12 13 14 15 16 17],
#          [18 19 20 21 22 23],
#          [24, 25, 26, 27, 28, 29]]
data = np.reshape(np.arange(30), [5, 6])
x = tf.constant(data)
result = tf.gather_nd(x, [1, 2])
```

where `result` will just be `8` as expected. Note how this is different from `tf.gather`: the same indices passed to `tf.gather(x, [1, 2])` would have given as the 2nd and 3rd *row* from `data`.

If you want to retrieve more than one element at the same time, just pass a list of index pairs:

```
result = tf.gather_nd(x, [[1, 2], [4, 3], [2, 5]])
```

which will return `[ 8 27 17]`

Collecting rows from a tensor of rank 2

If in the above example you want to collect rows (i.e. slices) instead of elements, adjust the `indices` parameter as follows:

```
data = np.reshape(np.arange(30), [5, 6])
x = tf.constant(data)
result = tf.gather_nd(x, [[1], [3]])
```

This will give you the 2nd and 4th row of `data`, i.e.

```
[[ 6  7  8  9 10 11]
 [18 19 20 21 22 23]]
```

Collecting elements from a tensor of rank 3

The concept of how to access rank-2 tensors directly translates to higher dimensional tensors. So, to access elements in a rank-3 tensor, the innermost dimension of `indices` must have length 3.

```
# data is [[[ 0  1]
#           [ 2  3]
#           [ 4  5]]
#
#          [[ 6  7]
#           [ 8  9]
#           [10 11]]]
data = np.reshape(np.arange(12), [2, 3, 2])
x = tf.constant(data)
result = tf.gather_nd(x, [[0, 0, 0], [1, 2, 1]])
```

`result` will now look like this: `[ 0 11]`

Collecting batched rows from a tensor of rank 3

Let's think of a rank-3 tensor as a batch of matrices shaped `(batch_size, m, n)`. If you want to collect the first and second row for every element in the batch, you could use this:

```
# data is [[[ 0  1]
#           [ 2  3]
#           [ 4  5]]
#
#          [[ 6  7]
#           [ 8  9]
#           [10 11]]]
data = np.reshape(np.arange(12), [2, 3, 2])
x = tf.constant(data)
```

```
result = tf.gather_nd(x, [[[0, 0], [0, 1]], [[1, 0], [1, 1]]])
```

which will result in this:

```
[[[0 1]
  [2 3]]

 [[6 7]
  [8 9]]]
```

Note how the shape of `indices` influences the shape of the output tensor. If we would have used a rank-2 tensor for the `indices` argument:

```
result = tf.gather_nd(x, [[0, 0], [0, 1], [1, 0], [1, 1]])
```

the output would have been

```
[[0 1]
 [2 3]
 [6 7]
 [8 9]]
```

Read Tensor indexing online: https://riptutorial.com/tensorflow/topic/2511/tensor-indexing

Chapter 18: TensorFlow GPU setup

Introduction

This topic is about setting up and managing GPUs in TensorFlow.

It assumes that the GPU version of TensorFlow has been installed (see https://www.tensorflow.org/install/ for more information on the GPU installation).

You also might want to have a look to the official documentation: https://www.tensorflow.org/tutorials/using_gpu

Remarks

Main sources:

- https://www.tensorflow.org
- https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto
- https://stackoverflow.com/a/37901914
- https://github.com/tensorflow/tensorflow/issues/152
- https://github.com/tensorflow/tensorflow/issue/9201

Examples

Run TensorFlow on CPU only - using the `CUDA_VISIBLE_DEVICES` environment variable.

To ensure that a GPU version TensorFlow process only runs on CPU:

```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="-1"
import tensorflow as tf
```

For more information on the `CUDA_VISIBLE_DEVICES`, have a look to this answer or to the CUDA documentation.

Run TensorFlow Graph on CPU only - using `tf.config`

```
import tensorflow as tf
sess = tf.Session(config=tf.ConfigProto(device_count={'GPU': 0}))
```

Bear in mind that this method prevents the TensorFlow Graph from using the GPU but TensorFlow still lock the GPU device as described in this an issue opened on this method. Using the `CUDA_VISIBLE_DEVICES` seems to be the best way to ensure that TensorFlow is kept away from the GPU card (see this answer).

Use a particular set of GPU devices

To use a particular set of GPU devices, the `CUDA_VISIBLE_DEVICES` environment variable can be used:

```
import os
os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID" # see issue #152
os.environ["CUDA_VISIBLE_DEVICES"]="0" # Will use only the first GPU device


os.environ["CUDA_VISIBLE_DEVICES"]="0,3" # Will use only the first and the fourth GPU devices
```

(Quoted from this answer; more information on the CUDA environment variables here.)

List the available devices available by TensorFlow in the local process.

```
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
```

Control the GPU memory allocation

By default, TensorFlow pre-allocate the whole memory of the GPU card (which can causes `CUDA_OUT_OF_MEMORY` warning).

To change this, it is possible to

- change the percentage of memory pre-allocated, using `per_process_gpu_memory_fraction` config option,

  > A value between 0 and 1 that indicates what fraction of the available GPU memory to pre-allocate for each process. 1 means to pre-allocate all of the GPU memory, 0.5 means the process allocates ~50% of the available GPU memory.

- disable the pre-allocation, using `allow_growth` config option. Memory allocation will grow as usage grows.

  > If true, the allocator does not pre-allocate the entire specified GPU memory region, instead starting small and growing as needed.

For example:

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
sess = tf.Session(config=config) as sess:
```

or

```
config = tf.ConfigProto()
```

```
config.gpu_options.allow_growth = True
sess= tf.Session(config=config):
```

More information on the config options here.

Read TensorFlow GPU setup online: https://riptutorial.com/tensorflow/topic/10621/tensorflow-gpu-setup

Chapter 19: Using 1D convolution

Examples

Basic example

Update: TensorFlow now supports 1D convolution since version r0.11, using `tf.nn.conv1d`.

Consider a basic example with an input of length `10`, and dimension `16`. The batch size is `32`. We therefore have a placeholder with input shape `[batch_size, 10, 16]`.

```
batch_size = 32
x = tf.placeholder(tf.float32, [batch_size, 10, 16])
```

We then create a filter with width 3, and we take `16` channels as input, and output also `16` channels.

```
filter = tf.zeros([3, 16, 16])  # these should be real values, not 0
```

Finally we apply `tf.nn.conv1d` with a stride and a padding:

- **stride**: integer `s`
- **padding**: this works like in 2D, you can choose between `SAME` and `VALID`. `SAME` will output the same input length, while `VALID` will not add zero padding.

For our example we take a stride of 2, and a valid padding.

```
output = tf.nn.conv1d(x, filter, stride=2, padding="VALID")
```

The output shape should be `[batch_size, 4, 16]`.
With `padding="SAME"`, we would have had an output shape of `[batch_size, 5, 16]`.

For previous versions of TensorFlow, you can just use 2D convolutions while setting the height of the inputs and the filters to `1`.

Math behind 1D convolution with advanced examples in TF

`To calculate 1D convolution by hand, you slide your kernel over the input, calculate the element-wise multiplications and sum them up.

The easiest way is for padding=0, stride=1

So if your `input = [1, 0, 2, 3, 0, 1, 1]` and `kernel = [2, 1, 3]` the result of the convolution is `[8, 11, 7, 9, 4]`, which is calculated in the following way:

- *8 = 1 * 2 + 0 * 1 + 2 * 3*
- *11 = 0 * 2 + 2 * 1 + 3 * 3*
- *7 = 2 * 2 + 3 * 1 + 0 * 3*
- *9 = 3 * 2 + 0 * 1 + 1 * 3*
- *4 = 0 * 2 + 1 * 1 + 1 * 3*

TF's conv1d function calculates convolutions in batches, so in order to do this in TF, we need to provide the data in the correct format (doc explains that input should be in `[batch, in_width, in_channels]`, it also explains how kernel should look like). So

```
import tensorflow as tf
i = tf.constant([1, 0, 2, 3, 0, 1, 1], dtype=tf.float32, name='i')
k = tf.constant([2, 1, 3], dtype=tf.float32, name='k')

print i, '\n', k, '\n'

data   = tf.reshape(i, [1, int(i.shape[0]), 1], name='data')
kernel = tf.reshape(k, [int(k.shape[0]), 1, 1], name='kernel')

print data, '\n', kernel, '\n'

res = tf.squeeze(tf.nn.conv1d(data, kernel, 1, 'VALID'))
with tf.Session() as sess:
    print sess.run(res)
```

which will give you the same answer we calculated previously: `[ 8. 11. 7. 9. 4.]`

Convolution with padding

Padding is just a fancy way to tell append and prepend your input with some value. In most of the cases this value is 0, and this is why most of the time people name it zero-padding. TF support 'VALID' and 'SAME' zero-padding, for an arbitrary padding you need to use tf.pad(). 'VALID' padding means no padding at all, where the same means that the output will have the same size of the input. Let's calculate the convolution with `padding=1` on the same example (notice that for our kernel this is 'SAME' padding). To do this we just append our array with 1 zero at the beginning/end: `input = [0, 1, 0, 2, 3, 0, 1, 1, 0]`.

Here you can notice that you do not need to recalculate everything: all the elements stay the same except of the first/last one which are:

- *1 = 0 * 2 + 1 * 1 + 0 * 3*
- *3 = 1 * 2 + 1 * 1 + 0 * 3*

So the result is `[1, 8, 11, 7, 9, 4, 3]` which is the same as calculated with TF:

```
res = tf.squeeze(tf.nn.conv1d(data, kernel, 1, 'SAME'))
with tf.Session() as sess:
```

```
    print sess.run(res)
```

Convolution with strides

Strides allow you to skip elements while sliding. In all our previous examples we slided 1 element, now you can slide `s` elements at a time. Because we will use a previous example, there is a trick: sliding by `n` elements is equivalent to sliding by 1 element and selecting every n-th element.

So if we use our previous example with `padding=1` and change `stride` to 2, you just take the previous result `[1, 8, 11, 7, 9, 4, 3]` and leave each 2-nd element, which will result in `[1, 11, 9, 3]`. You can do this in TF in the following way:

```
res = tf.squeeze(tf.nn.conv1d(data, kernel, 2, 'SAME'))
with tf.Session() as sess:
    print sess.run(res)
```

Read Using 1D convolution online: https://riptutorial.com/tensorflow/topic/5447/using-1d-convolution

Chapter 20: Using Batch Normalization

Parameters

`contrib.layers.batch_norm` params	Remarks
`beta`	python `bool` type. Whether or not to center the `moving_mean` and `moving_variance`
------	------
`gamma`	python `bool` type. Whether or not to scale the `moving_mean` and `moving_variance`
------	------
`is_training`	Accepts python `bool` or TensorFlow `tf.palceholder(tf.bool)`
------	------
`decay`	The default setting is `decay=0.999`. A smaller value (i.e. `decay=0.9`) is better for smaller dataset and/or less training steps.

Remarks

Here is a screen shot of the result of the working example above.

```
Initialized
Epoch: 0:       Loss: 2.576616          Tra
Epoch: 100:     Loss: 0.745796          Tra
Epoch: 200:     Loss: 0.569677          Tra
Epoch: 300:     Loss: 0.608862          Tra
Epoch: 400:     Loss: 0.437363          Tra
Epoch: 500:     Loss: 0.369688          Tra
Epoch: 600:     Loss: 0.394675          Tra
Epoch: 700:     Loss: 0.442162          Tra
Epoch: 800:     Loss: 0.305682          Tra
Epoch: 900:     Loss: 0.361511          Tra
Finished training
Test accuracy: 97.210002%
```

The code and a jupyter notebook version of this working example can be found at the author's repository

Examples

A Full Working Example of 2-layer Neural Network with Batch Normalization (MNIST Dataset)

Import libraries (language dependency: python 2.7)

```
import tensorflow as tf
import numpy as np
from sklearn.datasets import fetch_mldata
from sklearn.model_selection import train_test_split
```

load data, prepare data

```
mnist = fetch_mldata('MNIST original', data_home='./')
print "MNIST data, X shape\t", mnist.data.shape
print "MNIST data, y shape\t", mnist.target.shape
```

```
print mnist.data.dtype
print mnist.target.dtype

mnist_X = mnist.data.astype(np.float32)
mnist_y = mnist.target.astype(np.float32)
print mnist_X.dtype
print mnist_y.dtype
```

One-Hot-Encode y

```
num_classes = 10
mnist_y = np.arange(num_classes)==mnist_y[:, None]
mnist_y = mnist_y.astype(np.float32)
print mnist_y.shape
```

Split training, validation, testing data

```
train_X, valid_X, train_y, valid_y = train_test_split(mnist_X, mnist_y,
test_size=10000,\
                                                      random_state=102,  stratify=mnist.target)
train_X, test_X,  train_y, test_y  = train_test_split(train_X, train_y,     test_size=10000,\
                                                     random_state=325, stratify=train_y)
print 'Dataset\t\tFeatureShape\tLabelShape'
print 'Training set:\t', train_X.shape,'\t', train_y.shape
print 'Validation set:\t', valid_X.shape,'\t', valid_y.shape
print 'Testing set:\t', test_X.shape, '\t', test_y.shape
```

Build a simple 2 layer neural network graph

```
num_features = train_X.shape[1]
batch_size = 64
hidden_layer_size = 1024
```

An initialization function

```
def initialize(scope, shape, wt_initializer, center=True, scale=True):
    with tf.variable_scope(scope, reuse=None) as sp:
        wt = tf.get_variable("weights", shape, initializer=wt_initializer)
        bi = tf.get_variable("biases", shape[-1], initializer=tf.constant_initializer(1.))
        if center:
            beta = tf.get_variable("beta", shape[-1],
initializer=tf.constant_initializer(0.0))
        if scale:
            gamma = tf.get_variable("gamma", shape[-1],
initializer=tf.constant_initializer(1.0))
        moving_avg = tf.get_variable("moving_mean", shape[-1],
initializer=tf.constant_initializer(0.0), \
```

```
                                trainable=False)
        moving_var = tf.get_variable("moving_variance", shape[-1],
initializer=tf.constant_initializer(1.0), \
                                trainable=False)
    sp.reuse_variables()
```

Build Graph

```
init_lr = 0.001
graph = tf.Graph()
with graph.as_default():
    # prepare input tensor
    tf_train_X = tf.placeholder(tf.float32, shape=[batch_size, num_features])
    tf_train_y = tf.placeholder(tf.float32, shape=[batch_size, num_classes])
    tf_valid_X, tf_valid_y = tf.constant(valid_X), tf.constant(valid_y)
    tf_test_X,  tf_test_y  = tf.constant(test_X),  tf.constant(test_y)

    # setup layers
    layers = [{'scope':'hidden_layer', 'shape':[num_features, hidden_layer_size],
           'initializer':tf.truncated_normal_initializer(stddev=0.01)},
          {'scope':'output_layer', 'shape':[hidden_layer_size, num_classes],
           'initializer':tf.truncated_normal_initializer(stddev=0.01)}]
    # initialize layers
    for layer in layers:
        initialize(layer['scope'], layer['shape'], layer['initializer'])

    # build model - for each layer: -> X -> X*wt+bi -> batch_norm -> activation -> dropout (if
not output layer) ->
    layer_scopes = [layer['scope'] for layer in layers]
    def model(X, layer_scopes, is_training, keep_prob, decay=0.9):
        output_X = X
        for scope in layer_scopes:
            # X*wt+bi
            with tf.variable_scope(scope, reuse=True):
                wt = tf.get_variable("weights")
                bi = tf.get_variable("biases")
            output_X = tf.matmul(output_X, wt) + bi
            # Insert Batch Normalization
            # set `updates_collections=None` to force updates in place however it comes with
speed penalty
            output_X = tf.contrib.layers.batch_norm(output_X, decay=decay,
is_training=is_training,
                                                    updates_collections=ops.GraphKeys.UPDATE_OPS,
scope=scope, reuse=True)
            # ReLu activation
            output_X = tf.nn.relu(output_X)
            # Dropout for all non-output layers
            if scope!=layer_scopes[-1]:
                output_X = tf.nn.dropout(output_X, keep_prob)
        return output_X

    # setup keep_prob
    keep_prob = tf.placeholder(tf.float32)

    # compute loss, make predictions
    train_logits = model(tf_train_X, layer_scopes, True, keep_prob)
    train_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(train_logits,
tf_train_y))
```

```
    train_pred = tf.nn.softmax(train_logits)
    valid_logits = model(tf_valid_X, layer_scopes, False, keep_prob)
    valid_pred = tf.nn.softmax(valid_logits)
    test_logits = model(tf_test_X, layer_scopes, False, keep_prob)
    test_pred = tf.nn.softmax(test_logits)

    # compute accuracy
    def compute_accuracy(predictions, labels):
        correct_predictions = tf.equal(tf.argmax(predictions, 1), tf.argmax(labels, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_predictions, tf.float32))
        return accuracy

    train_accuracy = compute_accuracy(train_pred, tf_train_y)
    valid_accuracy = compute_accuracy(valid_pred, tf_valid_y)
    test_accuracy  = compute_accuracy(test_pred , tf_test_y)

    # setup learning rate, optimizer
    global_step = tf.Variable(0)
    learning_rate = tf.train.exponential_decay(init_lr,global_step, decay_steps=500,
decay_rate=0.95, staircase=True)
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(train_loss,
global_step=global_step)
```

Start a session

```
num_steps = 1000
with tf.Session(graph=graph) as sess:
    tf.initialize_all_variables().run()
    print('Initialized')
    for step in range(num_steps):
        offset = (step * batch_size) % (train_y.shape[0] - batch_size)
        batch_X = train_X[offset:(offset+batch_size), :]
        batch_y = train_y[offset:(offset+batch_size), :]
        feed_dict = {tf_train_X : batch_X, tf_train_y : batch_y, keep_prob : 0.6}
        _, tloss, tacc = sess.run([optimizer, train_loss, train_accuracy],
feed_dict=feed_dict)
        if step%50==0:
            # only evaluate validation accuracy every 50 steps to speed up training
            vacc = sess.run(valid_accuracy, feed_dict={keep_prob : 1.0})
            print('Epoch: %d:\tLoss: %f\t\tTrain Acc: %.2f%%\tValid Acc: %2.f%%\tLearning
rate: %.6f' \
                %(step, tloss, (tacc*100), (vacc*100), learning_rate.eval()))
    print("Finished training")
    tacc = sess.run([test_accuracy], feed_dict={keep_prob : 1.0})
    print("Test accuracy: %4f%%" %(tacc*100))
```

Read Using Batch Normalization online: https://riptutorial.com/tensorflow/topic/7909/using-batch-normalization

Chapter 21: Using if condition inside the TensorFlow graph with tf.cond

Parameters

Parameter	Details
pred	a TensorFlow tensor of type `bool`
fn1	a callable function, **with no argument**
fn2	a callable function, **with no argument**
name	(optional) name for the operation

Remarks

- `pred` cannot be just `True` or `False`, it needs to be a Tensor
- The function `fn1` and `fn2` should return the same number of outputs, with the same types.

Examples

Basic example

```
x = tf.constant(1.)
bool = tf.constant(True)

res = tf.cond(bool, lambda: tf.add(x, 1.), lambda: tf.add(x, 10.))
# sess.run(res) will give you 2.
```

When f1 and f2 return multiple tensors

The two functions `fn1` and `fn2` can return multiple tensors, but they have to return the exact same number and types of outputs.

```
x = tf.constant(1.)
bool = tf.constant(True)

def fn1():
    return tf.add(x, 1.), x

def fn2():
    return tf.add(x, 10.), x

res1, res2 = tf.cond(bool, fn1, fn2)
# tf.cond returns a list of two tensors
```

```
# sess.run([res1, res2]) will return [2., 1.]
```

define and use functions f1 and f2 with parameters

You can pass parameters to the functions in tf.cond() using **lambda** and the code is as bellow.

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

def fn1(a, b):
  return tf.mul(a, b)

def fn2(a, b):
  return tf.add(a, b)

pred = tf.placeholder(tf.bool)
result = tf.cond(pred, lambda: fn1(x, y), lambda: fn2(y, z))
```

Then you can call it as bellowing:

```
with tf.Session() as sess:
  print sess.run(result, feed_dict={x: 1, y: 2, z: 3, pred: True})
  # The result is 2.0
  print sess.run(result, feed_dict={x: 1, y: 2, z: 3, pred: False})
  # The result is 5.0
```

Read Using if condition inside the TensorFlow graph with tf.cond online: https://riptutorial.com/tensorflow/topic/2628/using-if-condition-inside-the-tensorflow-graph-with-tf-cond

Chapter 22: Using transposed convolution layers

Examples

Using tf.nn.conv2d_transpose for arbitary batch sizes and with automatic output shape calculation.

Example of how to calculate the output shape and overcome the difficulties of using tf.nn.conv2d_transpose with unknown batch size (when input.get_shape() is (?, H, W, C) or (?, C, H, W)).

```
def upconvolution (input, output_channel_size, filter_size_h, filter_size_w,
                   stride_h, stride_w, init_w, init_b, layer_name,
                   dtype=tf.float32, data_format="NHWC", padding='VALID'):
    with tf.variable_scope(layer_name):
      #calculation of the output_shape:
      if data_format == "NHWC":
        input_channel_size = input.get_shape().as_list()[3]
        input_size_h = input.get_shape().as_list()[1]
        input_size_w = input.get_shape().as_list()[2]
        stride_shape = [1, stride_h, stride_w, 1]
        if padding == 'VALID':
          output_size_h = (input_size_h - 1)*stride_h + filter_size_h
          output_size_w = (input_size_w - 1)*stride_w + filter_size_w
        elif padding == 'SAME':
          output_size_h = (input_size_h - 1)*stride_h + 1
          output_size_w = (input_size_w - 1)*stride_w + 1
        else:
          raise ValueError("unknown padding")
        output_shape = tf.stack([tf.shape(input)[0],
                                 output_size_h, output_size_w,
                                 output_channel_size])
      elif data_format == "NCHW":
        input_channel_size = input.get_shape().as_list()[1]
        input_size_h = input.get_shape().as_list()[2]
        input_size_w = input.get_shape().as_list()[3]
        stride_shape = [1, 1, stride_h, stride_w]
        if padding == 'VALID':
          output_size_h = (input_size_h - 1)*stride_h + filter_size_h
          output_size_w = (input_size_w - 1)*stride_w + filter_size_w
        elif padding == 'SAME':
          output_size_h = (input_size_h - 1)*stride_h + 1
          output_size_w = (input_size_w - 1)*stride_w + 1
        else:
          raise ValueError("unknown padding")
        output_shape = tf.stack([tf.shape(input)[0],
                                 output_channel_size,
                                 output_size_h, output_size_w])
      else:
        raise ValueError("unknown data_format")

      #creating weights:
```

```
    shape = [filter_size_h, filter_size_w,
             output_channel_size, input_channel_size]
    W_upconv = tf.get_variable("w", shape=shape, dtype=dtype,
                               initializer=init_w)

    shape=[output_channel_size]
    b_upconv = tf.get_variable("b", shape=shape, dtype=dtype,
                               initializer=init_b)

    upconv = tf.nn.conv2d_transpose(input, W_upconv, output_shape, stride_shape,
                                    padding=padding,
                                    data_format=data_format)
    output = tf.nn.bias_add(upconv, b_upconv, data_format=data_format)

    #Now output.get_shape() is equal (?,?,?,?) which can become a problem in the
    #next layers. This can be repaired by reshaping the tensor to its shape:
    output = tf.reshape(output, output_shape)
    #now the shape is back to (?, H, W, C) or (?, C, H, W)

    return output
```

Read Using transposed convolution layers online:
https://riptutorial.com/tensorflow/topic/9640/using-transposed-convolution-layers

Chapter 23: Variables

Examples

Declaring and Initializing Variable Tensors

Variable tensors are used when the values require updating within a session. It is the type of tensor that would be used for the weights matrix when creating neural networks, since these values will be updated as the model is being trained.

Declaring a variable tensor can be done using the `tf.Variable()` or `tf.get_variable()` function. It is recommended to use `tf.get_variable`, as it offers more flexibility eg:

```
# Declare a 2 by 3 tensor populated by ones
a = tf.Variable(tf.ones([2,3], dtype=tf.float32))
a = tf.get_variable('a', shape=[2, 3], initializer=tf.constant_initializer(1))
```

Something to note is that declaring a variable tensor does not automatically initialize the values. The values need to be intialized explicitly when starting a session using one of the following:

- `tf.global_variables_initializer().run()`
- `session.run(tf.global_variables_initializer())`

The following example shows the full process of declaring and initializing a variable tensor.

```
# Build a graph
graph = tf.Graph()
with graph.as_default():
    a = tf.get_variable('a', shape=[2,3], initializer=tf.constant_initializer(1),
dtype=tf.float32))     # Create a variable tensor

# Create a session, and run the graph
with tf.Session(graph=graph) as session:
    tf.global_variables_initializer().run()  # Initialize values of all variable tensors
    output_a = session.run(a)            # Return the value of the variable tensor
    print(output_a)                      # Print this value
```

Which prints out the following:

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
```

Fetch the value of a TensorFlow variable or a Tensor

Sometimes we need to fetch and print the value of a TensorFlow variable to guarantee our program is correct.

For example, if we have the following program:

```
import tensorflow as tf
import numpy as np
a = tf.Variable(tf.random_normal([2,3])) # declare a tensorflow variable
b = tf.random_normal([2,2]) #declare a tensorflow tensor
init = tf.initialize_all_variables()
```

if we want to get the value of a or b, the following procedures can be used:

```
with tf.Session() as sess:
    sess.run(init)
    a_value = sess.run(a)
    b_value = sess.run(b)
    print a_value
    print b_value
```

or

```
with tf.Session() as sess:
    sess.run(init)
    a_value = a.eval()
    b_value = b.eval()
    print a_value
    print b_value
```

Read Variables online: https://riptutorial.com/tensorflow/topic/2954/variables

Chapter 24: Visualizing the output of a convolutional layer

Introduction

There are many ways of visualizing the convolutional layers, but they share same components: fetching the values of a part of the convolutional neural networks, and visualizing those values. Note those visualizations should not and can not display on the TensorBoard.

Examples

A basic example of 2 steps

The example assumes you have successfully run and fully understand the tutorial of MNIST(Deep MNIST for expert).

```
%matplotlib inline
import matplotlib.pyplot as plt

# con_val is a 4-d array, the first indicates the index of image, the last indicates the index
of kernel
def display(con_val, kernel):
    plt.axis('off')
    plt.imshow(np.sum(con_val[:, :, :, kernel], axis=0), cmap=plt.get_cmap('gray'))
    plt.show()
```

The above function visualizes an array (con_val) containing the values of a convolutional layer given the kernel. The function sums up the values of all examples and plot them in gray-scale.

The following codes fetch values from the first convolutional layer and call the above function to display.

```
labels = np.nonzero(mnist.test.labels)[1] # convert "one-hot vectors" to digits (0-9)

for i in range(2): # display only 0 and 1
    con_val = h_pool1.eval(feed_dict={x:mnist.test.images[labels == i, :]}) #fetch
    display(con_val, 3)
```

The codes only plot the visualizations corresponding to labels of 0 and 1. You will be able to see the results as these.


Read Visualizing the output of a convolutional layer online:
https://riptutorial.com/tensorflow/topic/9346/visualizing-the-output-of-a-convolutional-layer

Credits

S. No	Chapters	Contributors
1	Getting started with tensorflow	adn, Community, daoliker, Engineero, Ishant Mrinal, Jacob Holloway, Maciej Lipinski, Mad Matts, Nicolas, Olivier Moindrot, Steven, Swapniel
2	Creating a custom operation with tf.py_func (CPU only)	mrry, Olivier Moindrot, SherylHohman
3	Creating RNN, LSTM and bidirectional RNN/LSTMs with TensorFlow	RamenChef, struct
4	How to debug a memory leak in TensorFlow	mrry, Vladimir Bystricky
5	How to use TensorFlow Graph Collections?	Augustin, Conchylicultor, kaufmanu, NCC, Nitred, Sultan Kenjeyev, Андрей Задаянчук
6	Math behind 2D convolution with advanced examples in TF	Salvador Dali
7	Matrix and Vector Arithmetic	Ishant Mrinal, ronrest
8	Measure the execution time of individual operations	mrry, Olivier Moindrot
9	Minimalist example code for distributed Tensorflow.	Ishant Mrinal
10	Multidimensional softmax	struct
11	Placeholders	Huy Vo, Ishant Mrinal, RamenChef, RobR, ronrest, Tom

12	Q-learning	BarzinM
13	Reading the data	basuam, sygi
14	Save and Restore a Model in TensorFlow	AryanJ-NYC, BarzinM, black_puppydog, danijar, Hara Hara Mahadevaki, kaufmanu, Olivier Moindrot, Rajarshee Mitra, Steven, Steven Hutt, Tom
15	Save Tensorflow model in Python and load with Java	Ishant Mrinal, Karl Nicholas
16	Simple linear regression structure in TensorFlow with Python	ml4294, Nicolas Bortolotti
17	Tensor indexing	Androbin, kaufmanu, Olivier Moindrot
18	TensorFlow GPU setup	Nicolas
19	Using 1D convolution	Olivier Moindrot, Salvador Dali
20	Using Batch Normalization	Zhongyu Kuang
21	Using if condition inside the TensorFlow graph with tf.cond	Kongsea, Olivier Moindrot, Paulo Alves
22	Using transposed convolution layers	BlueSun
23	Variables	Ishant Mrinal, kame, Kongsea, ronrest
24	Visualizing the output of a convolutional layer	Tengerye