**Tuple** - mute-able since it isn’t modifiable and used to return a multi returned values from function to be de-tupled.

**List** - modifiable values (un muteable)

Returned from a function **when order isn’t important** i.e no random access. When order is important then return **dict**

random access example : data[12] ->where need to acces specific cell

* list = [,] a list of any : len(roidb)
* disct = {}
* any entity is an **object** by nature and has **methods** according to its type : str or int
  1. name = 'voc\_{}\_{}'.format(year, split) # : 'voc\_{}\_{}' => str format =method
* dictionary = {x:label, y:observe} ,
  + read the keys of the dictionary: data.keys()
  + data.keys()
  + dict\_keys(['batch\_label', 'data', 'labels', 'filenames'])
* data['data']
* data['filenames'][1]

====================================================

Ascending order (with step -1) starting from index=0 five elements

preds = (np.argsort(prob)[::-1])[0:5]

input.get\_shape()[-1] :Take last variable (cyclic right)

=============

L[::-1]  
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]  
kk = np.arange(1,20,0.5) = 1:0.5:20

This also works for tuples, arrays, and strings:

==============

def layer\_decorated(self, \*args, \*\*kwargs) :

\*args: elipses ; \*\*kwargs:keywords args

==========================

Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

batch[0], .. batch[1]

* tuple = (, , )
* im = data['data'][0].reshape(3,32,32).transpose(1,2,0)
* ctrl+/ = comment
* Python script 2.x to 3.x
  + 2to3 -w <Filname>
* Load and save serialization of binary
  + pickle.**load**

import cPickle

cPickle.dump([blobs,rpn\_rois],open("test.pkl","wb"))

jj = pickle.load( open( "test.pkl", "rb" ) )

np.fromfile();

### plt.imshow(roidb[:,:,0])

### plt.imshow(roidb[:,:,0],cmap=plt.cm.gray\_r,vmin=0, vmax=10)

### 

### 

### 

### the following special forms using **leading or trailing underscores** are recognize (these can generally be combined with any case convention):

### \_single\_leading\_underscore: weak "internal use" indicator. E.g. "from M import \*" does not import objects whose name starts with an underscore.

### single\_trailing\_underscore\_: used by convention to avoid conflicts with Python keyword, e.g.

### Tkinter.Toplevel(master, class\_='ClassName')

### \_\_double\_leading\_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, \_\_boo becomes \_FooBar\_\_boo; see below).

### \_\_double\_leading\_and\_trailing\_underscore\_\_: "magic" objects or attributes that live in user-controlled namespaces. E.g. \_\_init\_\_, \_\_import\_\_ or \_\_file\_\_. Never invent such names; only use them as documented.

### 

**@property** : calls method function of get and set with operation

**tf.py\_func** : convert python to TF graph

**TF constants file** : **flags.xyz** :

**Decorator(designe pattern)**: epilog, main, prolog

@layer

def conv()

def layer(op)

def layer\_decorator(self,...):

layer\_output = op(self)

return self

return layer\_decorated

numpy

The best way to think about NumPy arrays is that they consist of two parts, a *data buffer* which is just a block of raw elements, and a *view* which describes how to interpret the data buffer.

For example, if we create an array of 12 integers:

>>> a = numpy.arange(12) >>> a array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

a.shape (12,)

Here the *shape* (12,) means the array is indexed by a single index which runs from 0 to 11. Conceptually, if we label this single index i, the array a looks like this:

i= 0 1 2 3 4 5 6 7 8 9 10 11 ┌────┬────┬────┬────┬────┬────┬────┬────┬────┬────┬────┬────┐ │ 0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │ 7 │ 8 │ 9 │ 10 │ 11 │ └────┴────┴────┴────┴────┴────┴────┴────┴────┴────┴────┴────┘

If we [reshape](http://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html) an array, this doesn't change the data buffer. Instead, it creates a new view that describes a different way to interpret the data. So after:

**Broadcast : numpy (instead of sugmenting by reshape in matlab) - syntactic sugar**

The smaller array is “broadcast” across the larger array so that they have compatible shapes so that looping occurs in C instead of Python

Anchors = [1 A 4]

Kshifts = [K 1 4 ]

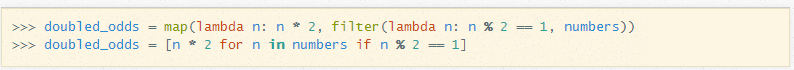
= [K A 4]

Image (3d array): 256 x 256 x 3  
Scale (1d array): 3  
Result (3d array): 256 x 256 x 3

When either of the dimensions compared is one, the other is used. In other words, dimensions with size 1 are stretched or “copied” to match the other.

**lazy loading** : only when you access the variable (roi\_db) at first it invokes the loading

**list comprehension** : syntactic sugar : map + filter



gt\_roidb = [self.\_load\_kitti\_voxel\_exemplar\_annotation(index)

for index in self.image\_index]

Will run the for statement for the above line

### 

### **Classes** :

Constructor : def \_\_init\_\_(self

### Questions:

How to calculate mAP, AP

<http://stackoverflow.com/questions/35365007/tensorflow-precision-recall-f1-score-and-confusion-matrix>

supervisor

coordinator

Tensorboard

### Tensorflow format

**images**: 4-D Tensor of shape [batch, height, width, channels]

Feeding with TF:Supply feed data through the feed\_dict argument to a run() or eval() call that initiates computation.

Tensor slice : tf.slice()

## Debugging a python script with input argument

use ipdb, first install : pip install ipdb

Invoke from the dos prompt :

python train\_model\_simple.py experiment.yaml

Code:

args = parser.parse\_args()

#added to catch the argument

import ipdb

ipdb.set\_trace()

#args parser will have the input file parameter

with open(args.config\_file) as fp:

You would get a prompt

ipdb> a

**a[rgs]**

a is one of my favorites. It prints out all the arguments the current function received.

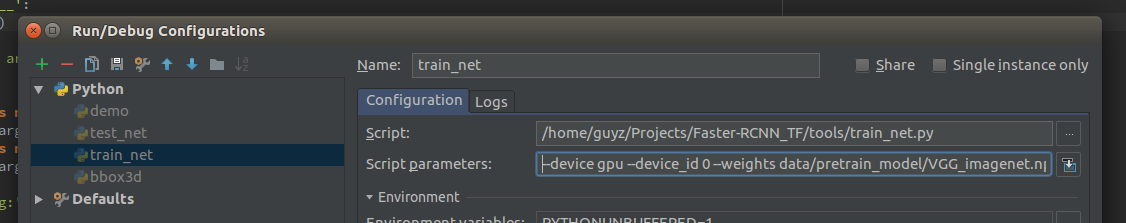
ipdb> pp locals()

Then after catching modify the code and add/modify to have fixed file opened :

filepath = 'experiment.yaml'

with open(filepath) as fp:

2nd way:



Drill

https://www.tensorflow.org/tutorials/mnist/tf/

Python slicing: *Use slicing to pull out the subarray consisting:*

[*http://cs231n.github.io/python-numpy-tutorial/*](http://cs231n.github.io/python-numpy-tutorial/)

Example

<https://www.tensorflow.org/tutorials/mnist/beginners/>

Install Tensor flow

<https://www.tensorflow.org/get_started/os_setup#optional-install-cuda-gpus-on-linux>

<https://www.tensorflow.org/get_started/os_setup#optional_install_cuda_gpus_on_linux>

Images normalization

Initialization of the coefficients

Numpy drill

<http://cs231n.github.io/python-numpy-tutorial/>

Train : CFAR 10 with NN

Analysing :

# **Class activation maps in Keras for visualizing where deep learning networks pay attention :**

<https://jacobgil.github.io/deeplearning/class-activation-maps>

And t-SNE metric :

<http://cs.stanford.edu/people/karpathy/cnnembed/>

Implementation : <http://lvdmaaten.github.io/tsne/>

* Images normalization
* Initialization of the coefficients in many ways
* Regularization
* Confusion matrix
* Test set performance

# [In TensorFlow, what is the difference between Session.run() and Tensor.eval()?](http://stackoverflow.com/questions/33610685/in-tensorflow-what-is-the-difference-between-session-run-and-tensor-eval)

If you have a Tensor t, calling [t.eval()](https://www.tensorflow.org/api_docs/python/framework/core_graph_data_structures#Tensor.eval) is equivalent to calling tf.get\_default\_session().run(t).

You can make a session the default as follows:

t = tf.constant(42.0) sess = tf.Session() with sess.as\_default(): # or `with sess:` to close on exit assert sess is tf.get\_default\_session() assert t.eval() == sess.run(t)

The most important difference is that you can use sess.run() to fetch the values of many tensors in the same step:

### 

### 

### How to set adaptive learning rate for GradientDescentOptimizer?

# <http://stackoverflow.com/questions/33919948/how-to-set-adaptive-learning-rate-for-gradientdescentoptimizer>

# TensorBoard

python "C:\Program Files\Anaconda3\Lib\site-packages\tensorflow\tensorboard\tensorboard.py" --logdir=mnist\_tf\_log

<http://stackoverflow.com/questions/38008512/how-can-i-get-the-value-of-the-error-during-training-in-tensorflow>

[http://localhost:6006](http://localhost:6006/)

|  |
| --- |
| As some person say, TensorBoard is the one for that purpose.  Here I can give you how to.  First, let's define a function for logging min, max, mean and std-dev for the tensor.  def variable\_summaries(var, name): with tf.name\_scope("summaries"): mean = tf.reduce\_mean(var) tf.scalar\_summary('mean/' + name, mean) with tf.name\_scope('stddev'): stddev = tf.sqrt(tf.reduce\_sum(tf.square(var - mean))) tf.scalar\_summary('stddev/' + name, stddev) tf.scalar\_summary('max/' + name, tf.reduce\_max(var)) tf.scalar\_summary('min/' + name, tf.reduce\_min(var)) tf.histogram\_summary(name, var)  Then, create a summarize operation after you build a graph like below. This code saves weight and bias of first layer with cross-entropy in "mnist\_tf\_log" directory.  variable\_summaries(W\_fc1, "W\_fc1") variable\_summaries(b\_fc1, "b\_fc1") tf.scalar\_summary("cross\_entropy:", cross\_entropy) summary\_op = tf.merge\_all\_summaries() summary\_writer = tf.train.SummaryWriter("mnist\_tf\_log", graph\_def=sess.graph)  Now you're all set. You can log those data by returning summary\_op and pass it to summary\_writer.  Here is an example for logging every 10 training steps.  for i in range(1000): batch\_xs, batch\_ys = mnist.train.next\_batch(100) if i % 10 == 0: \_, summary\_str = sess.run( [train\_step, summary\_op], feed\_dict={x: batch\_xs, y\_: batch\_ys}) summary\_writer.add\_summary(summary\_str, i) summary\_writer.flush() else: sess.run( train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys}) |

===============================

## **Variable Scope Example**

Variable Scope mechanism in TensorFlow consists of 2 main functions:

* tf.get\_variable(<name>, <shape>, <initializer>): Creates or returns a variable with a given name.
* tf.variable\_scope(<scope\_name>): Manages ***namespaces*** for names passed to tf.get\_variable().

tf.variable\_scope() comes into play: it pushes a namespace for variables.

The method [tf.get\_variable](https://www.tensorflow.org/versions/r0.8/api_docs/python/state_ops.html#get_variable) can be used with the name of the variable as argument to either create a new variable with such name or retrieve the one that was created before. This is different from using the [tf.Variable](https://www.tensorflow.org/versions/r0.8/api_docs/python/state_ops.html#Variable) constructor which will create a new variable

every time it is called (and potentially add a suffix to the variable name if a variable with such name already exists). It is for the purpose of the **variable sharing mechanism** that a separate type of scope (variable scope) was introduced.

As a result, we end up having two different types of scopes:

* *name scope*, created using [tf.name\_scope](https://www.tensorflow.org/versions/r0.8/api_docs/python/framework.html#name_scope) or [tf.op\_scope](https://www.tensorflow.org/versions/r0.8/api_docs/python/framework.html#op_scope)
* *variable scope*, created using [tf.variable\_scope](https://www.tensorflow.org/versions/r0.8/api_docs/python/state_ops.html#variable_scope) or [tf.variable\_op\_scope](https://www.tensorflow.org/versions/r0.8/api_docs/python/state_ops.html#variable_op_scope)

Both scopes have the same effect on all operations as well as variables created using tf.Variable, i.e. the scope will be added as a prefix to the operation or variable name.

======================

**self** : like this for defining method

**Logits simply means that the function operates on the unscaled output** of earlier layers and that the relative scale to understand the units is linear. It means, in particular, the sum of the inputs may not equal 1, that the values are *not* probabilities (you might have an input of 5).

tf.nn.softmax produces just the result of applying the [softmax function](https://en.wikipedia.org/wiki/Softmax_function) to an input tensor. The softmax "squishes" the inputs so that sum(input) = 1; it's a way of normalizing. The shape of output of a softmax is the same as the input - it just normalizes the values. The outputs of softmax *can* be interpreted as probabilities.

a = tf.constant(np.array([[.1, .3, .5, .9]])) print s.run(tf.nn.softmax(a)) [[ 0.16838508 0.205666 0.25120102 0.37474789]]

In contrast, tf.nn.softmax\_cross\_entropy\_with\_logits computes the cross entropy of the result after applying the softmax function (but it does it all together in a more mathematically careful way). It's similar to the result of:

sm = tf.nn.softmax(x) ce = cross\_entropy(sm)

The cross entropy is a summary metric - it sums across the elements. **The output of tf.nn.softmax\_cross\_entropy\_with\_logits on a shape [2,5] tensor is of shape [2,1] (the first dimension is treated as the batch).**

# **Creating Estimators in tf.contrib.learn**

tf.contrib.learn : tentative framework by the release

tf.get\_variable(): As you can see, tf.get\_variable() checks that already existing variables are not shared by accident. If you want to share them, you need to specify it by setting reuse\_variables() as follows.

Read output

for tensor:

bbox\_pred\_mean\_t = sess.run(net.bbox\_pred\_means,feed\_dict=feed\_dict)

For layer

bbox\_pred\_denorm\_t = sess.run(net.\_predictions['bbox\_pred\_denorm'],feed\_dict=feed\_dict)

**Install Tensorflow**

sudo apt-get install tortoisehg

sudo apt-get install python-pip python-dev

sudo apt-get install virtualenv # install virtualenv

virtualenv --system-site-packages ~/tensorflow # create virtualenv

source ~/tensorflow/bin/activate

pip install --upgrade tensorflow-gpu # install tensorflow

sudo pip install --upgrade pip

sudo pip install cython easydict

sudo pip install opencv-python

sudo apt-get install git

sudo pip install --upgrade cython

sudo pip install scipy

sudo pip install matplotlib

sudo apt-get install python-tk

pip install pyyaml

sudo pip install image

mkdir ~/Projects

cd Projects

git clone --recursive https://github.com/smallcorgi/Faster-RCNN\_TF.git

- add -D\_GLIBCXX\_USE\_CXX11\_ABI=0 to g++ cmd in make.sh

Build the cyto modules

cd $FCRN\_ROOT/lib

make

source ~/tensorflow27/bin/activate

Running tensorboard :

source ~/tensorflow/bin/activate

(tensorflow) hanochk@inv-lgc02:/usr/local/cuda/lib64$ tensorboard --logdir='/home/hanochk/Projects/fasterrcnn/log/'

Read output

for tensor:

bbox\_pred\_mean\_t = sess.run(net.bbox\_pred\_means,feed\_dict=feed\_dict)

For layer

bbox\_pred\_denorm\_t = sess.run(net.\_predictions['bbox\_pred\_denorm'],feed\_dict=feed\_dict)