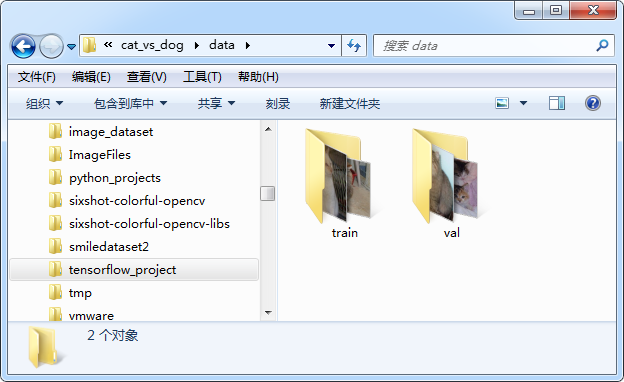
# 使用tensorflow实现猫狗分类

## 数据准备

训练集，使用10000张猫图像和10000张狗图像，测试集，使用2500张猫图像和2500张狗图像。



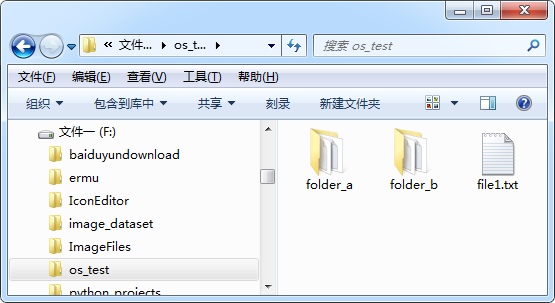
### 生成图像和标签列表

编写input\_data.py文件，代码如下：

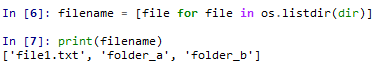
|  |
| --- |
| *# -\*- coding: utf-8 -\*-* **import** tensorflow **as** tf **import** numpy **as** np **import** os  **def** get\_files(file\_dir):  cats=[]  label\_cats=[]  dogs=[]  label\_dogs=[]  **for** file **in** os.listdir(file\_dir):  name=file.split(sep=**'.'**)  **if** name[0]==**'cat'**:  cats.append(file\_dir+file)  label\_cats.append(0)  **else**:  dogs.append(file\_dir+file)  label\_dogs.append(1)  print(**'cat number: %d, dog number: %d'**%(len(cats),len(dogs)))   image\_list=np.hstack((cats,dogs))  label\_list=np.hstack((label\_cats, label\_dogs))   temp=np.array([image\_list, label\_list])  temp=temp.transpose()  np.random.shuffle(temp)   image\_list=list(temp[:,0])  label\_list=list(temp[:,1])  **label\_list = [int(i) for i in label\_list]**  **return** image\_list, label\_list |

代码中，定义了一个get\_files函数用于返回图像的路径和标签列表。

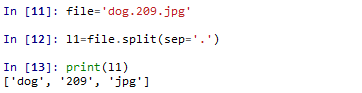
os.listdir可以返回路径下的文件和文件夹列表，例如，有以下文件夹：



测试以下代码：

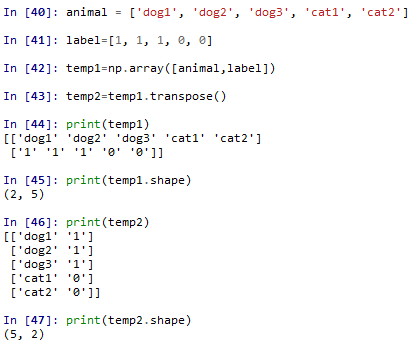


由于使用的数据都是以dog.1.jpg、cat.1.jpg这样的形式命名的，因此可以从文件名中判断该图像属于dog或cat，使用字符串的split方法可以将其拆分，测试代码：



Numpy的hstack函数可以将两个列表水平拼接起来。为了后面进行洗牌打乱图像顺序(使用numpy的np.random.shuffle函数)，需要把图像和其标签建立对应关系并转成numpy的array对象。

注意红色代码部分特别重要，但又容易忽视，label\_list的元素为numpy.str\_数据类型，需要将其转成int类型。



### 1.2 生成batch

|  |
| --- |
| **def** get\_batch(image, label, image\_w, image\_h, batch\_size, capacity):   image=tf.cast(image,tf.string)  label=tf.cast(label,tf.int32)   input\_queue=tf.train.slice\_input\_producer([image,label])  label=input\_queue[1]  image\_content=tf.read\_file(input\_queue[0])  image=tf.image.decode\_jpeg(image\_content,channels=3)  image=tf.image.resize\_image\_with\_crop\_or\_pad (image,image\_w,image\_h)  image=tf.image.per\_image\_standardization(image)  image\_batch, label\_batch=tf.train.batch(  [image,label],  batch\_size=batch\_size,  num\_threads=64,  capacity=capacity  )  **return** image\_batch, label\_batch |

通过get\_files函数获得的图像路径和标签是python的类型，我们要将其装成tensorflow对应的类型，这可以使用tensorflow的cast函数完成。

接下来我们使用slice\_input\_producer来生成队列queue，输入参数是图像路径及标签列表，生成的queue第0个元素是image路径，第1个元素是标签。我们使用read\_file函数来读取图像，完成后需要使用tf.image.decode\_jpg函数来解码。

由于图像大小不一，我们为了把所有图像统一到一个大小，需要对大图像进行裁剪，对小图像进行扩充，这可以使用函数tf.image.resize\_image\_with\_crop\_or\_pad函数来实现。

接下来需要对图像进行标准化，减去均值，除以方差，使用函数tf.image.per\_image\_standardization函数。

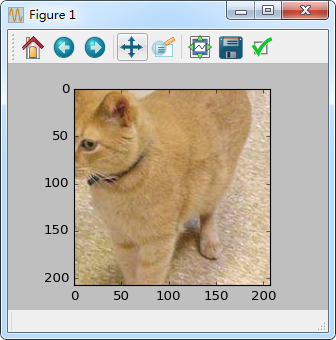
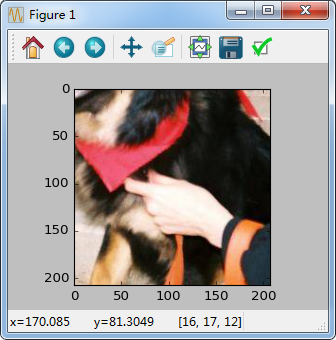
打包成batch，使用函数tf.train.batch函数。

以上用到的函数都可以到tensorflow官网上查找其用法。

### 测试batch

为测试以上对数据的读取有没有问题，查看原始的图像效果，我们把get\_batch中的代码image=tf.image.per\_image\_standardization(image)注释掉，编写以下测试代码：

|  |
| --- |
| **import** matplotlib.pyplot **as** plt **import** tensorflow **as** tf **import** input\_data **import** numpy **as** np   BATCH\_SIZE=2 CAPACITY=256 IMG\_W=208 IMG\_H=208 file\_dir=**'F:/tensorflow\_project/cat\_vs\_dog/data/train/'** image\_list, label\_list = input\_data.get\_files(file\_dir) image\_batch, label\_batch=input\_data.get\_batch(image\_list, label\_list, IMG\_W, IMG\_H, BATCH\_SIZE, CAPACITY)  **with** tf.Session() **as** sess:  i = 0  coord = tf.train.Coordinator()  threads = tf.train.start\_queue\_runners(coord=coord)  **try**:  **while not** coord.should\_stop() **and** i<1:  img,label=sess.run([image\_batch, label\_batch])   **for** j **in** np.arange(BATCH\_SIZE):  print(**'label: %d'**%label[j])  plt.imshow(img[j,:,:,:])  plt.show()  i+=1  **except** tf.errors.OutOfRangeError:  print(**'done!'**)  **finally**:  coord.request\_stop()  print(**'finish!'**)  coord.join(threads) |

程序中使用Coordinator()及start\_queue\_runners来监控queue的状态，再使用try…exctpt…finally函数时刻对queue状态进行检查，可参考tensorflow官网：<https://www.tensorflow.org/versions/master/programmers_guide/reading_data>

## 模型定义

### 2.1 网络结构

这里的网络使用两个卷积层后接3个全连接层，代码如下：

|  |
| --- |
| *# -\*- coding: utf-8 -\*-* **import** tensorflow **as** tf  **def** inference(images, batch\_size, n\_classes):  *#第1层卷积层,包括conv、pool和norml层* **with** tf.variable\_scope(**'conv1'**) **as** scope:  weights=tf.get\_variable(**'weights'**,  shape=[3,3,3,16],*#卷积核尺寸3，3，输入通道3，输出通道16* dtype=tf.float32,  initializer=tf.truncated\_normal\_initializer(stddev=0.1,dtype=tf.float32)  )  biases=tf.get\_variable(**'biases'**,  shape=[16],  dtype=tf.float32,  initializer=tf.constant\_initializer(0.1)  )  conv=tf.nn.conv2d(images,weights,strides=[1,1,1,1],padding=**'SAME'**)  pre\_activation=tf.nn.bias\_add(conv,biases)  conv1=tf.nn.relu(pre\_activation,name=**'conv1'**)   **with** tf.variable\_scope(**'pooling\_lrn'**) **as** scope:  pool1=tf.nn.max\_pool(conv1,  ksize=[1,3,3,1],  strides=[1,2,2,1],  padding=**'SAME'**,  name=**'pooling1'** )  norm1=tf.nn.lrn(pool1,depth\_radius=4,bias=1.0,alpha=0.001/9.0,  beta=0.75,name=**'norm1'**)  *# 第2层卷积层,包括conv、pool和norml层* **with** tf.variable\_scope(**'conv2'**) **as** scope:  weights = tf.get\_variable(**'weights'**,  shape=[3, 3, 16, 16], *# 卷积核尺寸3，3，输入通道16，输出通道16* dtype=tf.float32,  initializer=tf.truncated\_normal\_initializer(stddev=0.1, dtype=tf.float32)  )  biases = tf.get\_variable(**'biases'**,  shape=[16],  dtype=tf.float32,  initializer=tf.constant\_initializer(0.1)  )  conv = tf.nn.conv2d(norm1, weights, strides=[1, 1, 1, 1], padding=**'SAME'**)  pre\_activation = tf.nn.bias\_add(conv, biases)  conv2 = tf.nn.relu(pre\_activation, name=**'conv2'**)   **with** tf.variable\_scope(**'pooling\_lrn'**) **as** scope:  pool2 = tf.nn.max\_pool(conv2,  ksize=[1, 3, 3, 1],  strides=[1, 2, 2, 1],  padding=**'SAME'**,  name=**'pooling2'** )  *#第一层全连接层* **with** tf.variable\_scope(**'local3'**) **as** scope:  reshape=tf.reshape(pool2, shape=[batch\_size,-1])  dim=reshape.get\_shape()[1].value  weights=tf.get\_variable(**'weights'**,  shape=[dim,128],  dtype=tf.float32,  initializer=tf.truncated\_normal\_initializer(stddev=0.005,dtype=tf.float32)  )  biases=tf.get\_variable(**'biases'**,  shape=[128],  dtype=tf.float32,  initializer=tf.constant\_initializer(0.1)  )  local3=tf.nn.relu(tf.matmul(reshape,weights)+biases,name=**'local3'**)   *# 第二层全连接层* **with** tf.variable\_scope(**'local4'**) **as** scope:  weights = tf.get\_variable(**'weights'**,  shape=[128, 128],  dtype=tf.float32,  initializer=tf.truncated\_normal\_initializer(stddev=0.005, dtype=tf.float32)  )  biases = tf.get\_variable(**'biases'**,  shape=[128],  dtype=tf.float32,  initializer=tf.constant\_initializer(0.1)  )  local4 = tf.nn.relu(tf.matmul(local3, weights) + biases, name=**'local4'**)   *# 第二层全连接层* **with** tf.variable\_scope(**'local5'**) **as** scope:  weights = tf.get\_variable(**'weights'**,  shape=[128, n\_classes],  dtype=tf.float32,  initializer=tf.truncated\_normal\_initializer(stddev=0.005, dtype=tf.float32)  )  biases = tf.get\_variable(**'biases'**,  shape=[n\_classes],  dtype=tf.float32,  initializer=tf.constant\_initializer(0.1)  )  local5 = tf.matmul(local4, weights) + biases  **return** local5 |

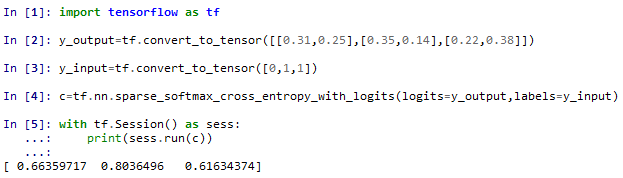
注意在local5层，没有使用激活函数，而是直接返回，因为在下面使用sparse\_softmax\_cross\_entropy\_with\_logits函数计算交叉熵时会在内部做这些操作。

### 2.2 损失函数、训练函数及评价函数

损失函数、训练及评价函数如下：

|  |
| --- |
| **def** losses(logist, labels):  **with** tf.variable\_scope(**'loss'**) **as** scope:  cross\_entropy=tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(  logits=logist,labels=labels,name=**'cross\_entropy'** )  loss=tf.reduce\_mean(cross\_entropy,name=**'loss'**)  tf.summary.scalar(**'cross\_entropy\_loss'**,loss)  **return** loss  **def** trainning(loss, learning\_rate):  **with** tf.name\_scope(**'optimizer'**):  optimizer=tf.train.AdamOptimizer(learning\_rate=learning\_rate)  global\_step=tf.Variable(0,name=**'global\_step'**,trainable=**False**)  train\_op=optimizer.minimize(loss,global\_step=global\_step)  **return** train\_op  **def** evaluation(logists, labels):  **with** tf.variable\_scope(**'accuracy'**) **as** scope:  correct=tf.nn.in\_top\_k(logists,labels,1)  correct=tf.cast(correct,tf.float32)  accuracy=tf.reduce\_mean(correct)  tf.summary.scalar(**'accuarcy'**, accuracy)  **return** accuracy |

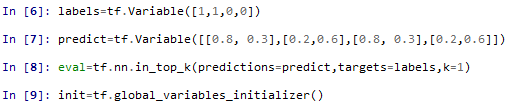
在损失函数中，使用函数sparse\_softmax\_cross\_entropy\_with\_logits来计算交叉熵损失，这个函数的参数包括预测结果，本例中，每一图像的预测结果是一个包括两个元素的向量，如[0.08993886 0.11971622]，另一个重要参数是标签，它表示正确分类结果的类别序号，如[0]或[1]，该函数的用法演示如下：

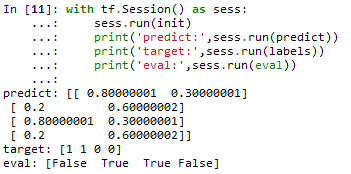


另外在evaluation函数中，使用了in\_to\_k函数，该函数声明如下：

in\_top\_k(predictions,targets,k,name=None)

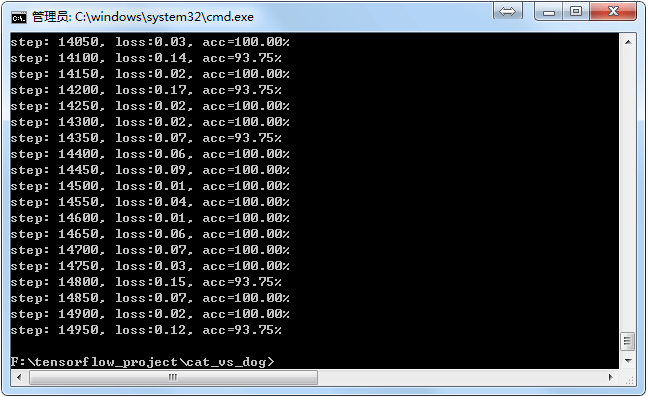
该函数判断target是否在top k预测之中，该函数演示如下：

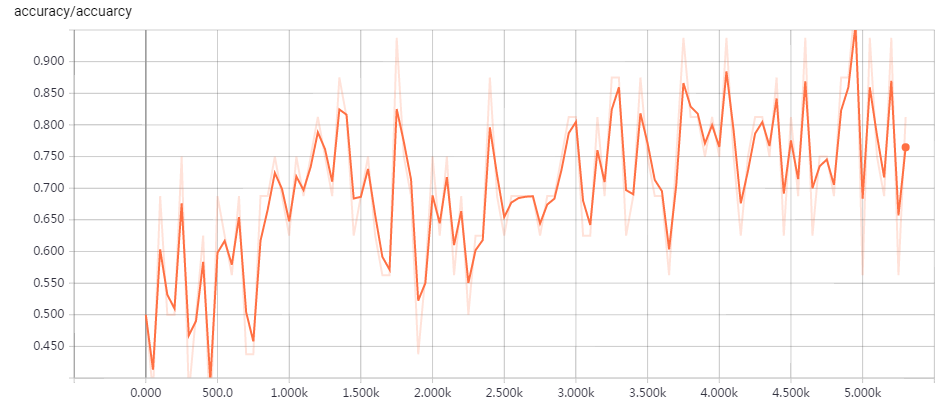


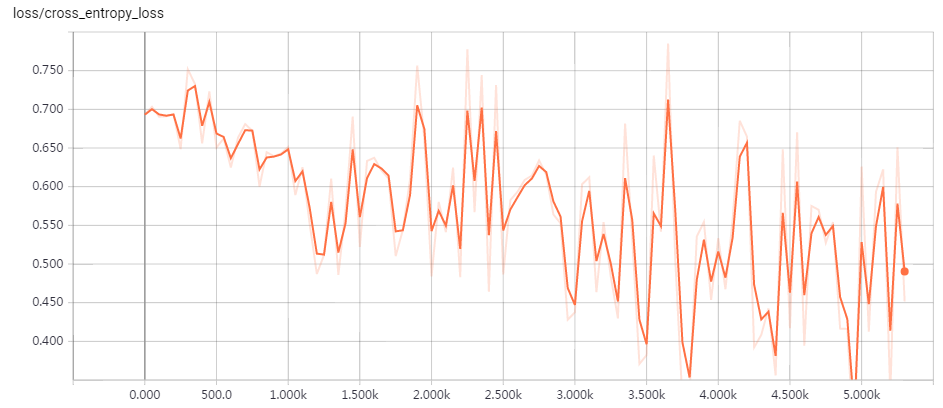


## 训练

|  |
| --- |
| *# -\*- coding: utf-8 -\*-* **import** os **import** tensorflow **as** tf **import** numpy **as** np **import** input\_data **import** model  N\_CLASS=2 IMG\_W=208 IMG\_H=208 BATCH\_SIZE=16 CAPACITY=2000 MAX\_STEP=15000 LEARNING\_RATE=0.0001  train\_dir=**'F:/tensorflow\_project/cat\_vs\_dog/data/train/'** log\_dir=**'F:/tensorflow\_project/cat\_vs\_dog/log/'   def** run\_training():  train,train\_label=input\_data.get\_files(train\_dir)  train\_batch, train\_label\_batch=input\_data.get\_batch(train,  train\_label,  IMG\_H,  IMG\_W,  BATCH\_SIZE,  CAPACITY  )  train\_logist=model.inference(images=train\_batch,batch\_size=BATCH\_SIZE,n\_classes=N\_CLASS)  train\_loss=model.losses(train\_logist,train\_label\_batch)  train\_op=model.trainning(train\_loss,learning\_rate=LEARNING\_RATE)  train\_acc=model.evaluation(train\_logist,train\_label\_batch)   summary\_op=tf.summary.merge\_all()  sess = tf.Session()  train\_writer=tf.summary.FileWriter(log\_dir, sess.graph)  saver=tf.train.Saver()   sess.run(tf.global\_variables\_initializer())  coord=tf.train.Coordinator()  threads=tf.train.start\_queue\_runners(sess,coord)   **try**:  **for** step **in** np.arange(2):  **if** coord.should\_stop():  **break** \_,tra\_loss,tra\_acc=sess.run([train\_op,train\_loss,train\_acc])  **if** step == 1:  print(**'logist: '**,sess.run(train\_logist))  print(**'label: '**, sess.run(train\_label\_batch))  **if** step%50==0:  print(**'step: %d, loss:%.2f, acc=%.2f%%'**%(step, tra\_loss,tra\_acc\*100.0))  summary\_str=sess.run(summary\_op)  train\_writer.add\_summary(summary\_str,step)   **if** step % 2000 == 0 **or** (step + 1) == MAX\_STEP:  checkpoint\_path = os.path.join(log\_dir,**'model.ckpt'**)  saver.save(sess, checkpoint\_path, global\_step=step)  **except** tf.errors.OutOfRangeError:  print(**"Done trainning..."**)  **finally**:  coord.request\_stop()  coord.join(threads)  run\_training() |



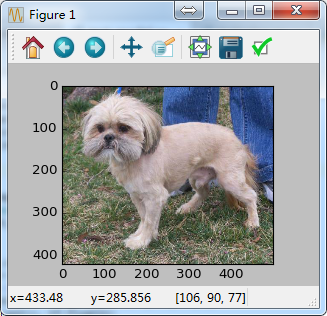


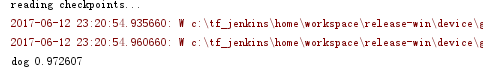


## 测试

载入训练好的模型进行测试，代码如下：

|  |
| --- |
| **from** PIL **import** Image **import** matplotlib.pyplot **as** plt **import** tensorflow **as** tf **import** input\_data **import** numpy **as** np **import** model  **def** get\_one\_image(train, ind):  img\_dir=train[ind]  image=Image.open(img\_dir)  plt.imshow(image)  image=image.resize([208,208])  image=np.array(image)  **return** image  **def** evaluate\_one\_image():  test\_dir=**'F:/tensorflow\_project/cat\_vs\_dog/data/val/'** test\_image, test\_label = input\_data.get\_files(test\_dir)   **with** tf.Graph().as\_default():  BATCH\_SIZE = 1  N\_CLASSES = 2  x=tf.placeholder(tf.float32, shape=[208,208,3])  image = tf.cast(x, tf.float32)  image = tf.image.per\_image\_standardization(image)  image = tf.reshape(image, [1, 208, 208, 3])   prediction = model.inference(image, BATCH\_SIZE, N\_CLASSES)  prediction = tf.nn.softmax(prediction)  log\_dir = **'F:/tensorflow\_project/cat\_vs\_dog/log/'** saver = tf.train.Saver()   **with** tf.Session() **as** sess:  print(**'reading checkpoints...'**)  ckpt = tf.train.get\_checkpoint\_state(log\_dir)  **if** ckpt **and** ckpt.model\_checkpoint\_path:  saver.restore(sess, ckpt.model\_checkpoint\_path)  **else**:  print(**'no checkpoint file found!'**)  **for** i **in** range(10):  image\_array = get\_one\_image(test\_image, i)  result = sess.run(prediction, feed\_dict={x: image\_array})  max\_idx = np.argmax(result)  **if** max\_idx == 0:  print(**'cat %.6f'** % result[:, 0])  **else**:  print(**'dog %.6f'** % result[:, 1])  plt.show()  evaluate\_one\_image() |





载入已训练模型的方法：

1. 定义一个Saver: saver = tf.train.Saver()
2. 从目录中载入模型：ckpt = tf.train.get\_checkpoint\_state(log\_dir)
3. 使用Saver的restore方法将模型载入会话：saver.restore(sess, ckpt.model\_checkpoint\_path)
4. 使用会话