

# 目录

1	Tensorflow 基础	1
1.1	Tensorflow 基础函数	1
1.1.1	Variable	1
1.1.2	placeholder	1
1.1.3	batch normalization	2
1.1.4	常见的的激活函数	2
1.2	relu 函数	3
1.2.1	relu	3
1.2.2	relu6	3
1.2.3	sigmoid	5
1.2.4	relu 和 softplus	6
1.2.5	dropout	8
1.3	卷积函数	8
1.4	池化	9
1.5	常见的分类函数	10
1.6	优化方法	10
1.6.1	BGD	11
1.6.2	SGD	11
1.6.3	momentum	11
1.6.4	Nesterov Momentum	11
1.6.5	Adagrad	12
1.6.6	RMSprop	12
1.6.7	Adam	12
1.6.8	构造简单的神经网络拟合数据	14
1.7	TensorBoard	15
1.7.1	TensorBoard Histogram Dashboard	17

1.7.2	一个简单的例子 . . . . .	17
1.7.3	Overlay Mode . . . . .	20
1.7.4	多个分布 . . . . .	21
1.7.5	更多分布 . . . . .	23
1.7.6	poisson 分布 . . . . .	26
1.7.7	结合所有的数据到一张图向上 . . . . .	27
1.8	CNN 手写体数据识别 . . . . .	28
1.8.1	mnist 数据集 . . . . .	28
1.9	RNN . . . . .	32
1.9.1	向量字表示 . . . . .	32
1.9.2	RNN . . . . .	40
2	Tensorflow 进阶	45
2.1	模型存储和加载 . . . . .	45
2.2	用 GPU . . . . .	46
2.2.1	手工配置设备 . . . . .	46
2.2.2	允许 GPU 的内存增长 . . . . .	47
3	Performance	49
3.1	最好的实践 . . . . .	49
3.2	从源代码创建安装 . . . . .	49
3.2.1	利用队列读取数据 . . . . .	50
3.2.2	在 CPU 上的预处理 . . . . .	50
3.2.3	用大文件 . . . . .	51
3.2.4	用 NCHW 图像数据格式 . . . . .	51
3.2.5	用融批规范 . . . . .	51
4	常用的 python 模块	53
4.1	Argparse . . . . .	53
4.1.1	ArgumentParser 对象 . . . . .	54
4.1.2	prog . . . . .	54
4.1.3	add_argument() 方法 . . . . .	59
4.2	path . . . . .	87
4.2.1	函数说明 . . . . .	87
4.2.2	例子 . . . . .	89
4.2.3	常见问题 . . . . .	90

目录	iii
5 re	101
5.1 正则表达式介绍 . . . . .	101
5.2 RE 库的主要功能函数 . . . . .	104
5.2.1 re 表达式中的 flags . . . . .	106
6 sys	111
7 url	119
8 requests	121
8.1 快速上手 . . . . .	121
8.1.1 发送请求 . . . . .	121
8.1.2 requests 库的 7 个主要方法 . . . . .	121
8.1.3 request 对象的属性 . . . . .	122
8.1.4 理解 encoding 和 apparent_encoding . . . . .	122
8.1.5 理解 Requests 库的异常 . . . . .	122
8.1.6 HTTP 协议 . . . . .	122
9 Tensorflow API	127
9.1 tf.squeeze . . . . .	127
9.2 tf.stack . . . . .	127
9.3 tf.metrics . . . . .	128
9.4 tf.reshape . . . . .	129
9.5 tf.image . . . . .	130
9.5.1 tf.image.decode_gif . . . . .	130
9.5.2 tf.image.decode_jpeg . . . . .	130
9.5.3 tf.image.encode_jpeg . . . . .	131
9.5.4 tf.image.decode_png . . . . .	131
9.5.5 tf.image.encode_png . . . . .	132
9.5.6 tf.image.decode_image . . . . .	132
9.5.7 tf.image.resize_images . . . . .	132



# Chapter 1

## Tensorflow 基础

### 1.1 Tensorflow 基础函数

#### 1.1.1 Variable

```
#tensorflow 1.2.1
import tensorflow as tf
var = tf.Variable(0)
add_operation = tf.add(var,1)
update_operation = tf.assign(var,add_operation)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for _ in range(3):
        sess.run(update_operation)
        print(sess.run(var))
```

#### 1.1.2 placeholder

```
#tensorflow 1.2
import tensorflow as tf
x1 = tf.placeholder(dtype=tf.float32,shape=None)
y1 = tf.placeholder(dtype=tf.float32,shape=None)
z1 = x1+y1
x2 = tf.placeholder(dtype=tf.float32,shape=[2,1])
y2 = tf.placeholder(dtype=tf.float32,shape=[1,2])
z2 = tf.matmul(x2,y2)
with tf.Session() as sess:
    z1_value = sess.run(z1,feed_dict={x1:1,y1:2})
```

```

z1_value,z2_value = sess.run([z1,z2],feed_dict={x1:1,y1:2,x2:[[2],[2]],y2: [[3,3]]})
print(z1_value)
print(z2_value)

```

### 1.1.3 batch normalization

§ 数据 x 为 Tensor。

- mean: 为 x 的均值，也是一个 Tensor。
- var: 为 x 的方差，也为一个 Tensor。
- offset: 一个偏移，也是一个 Tensor。
- scale: 缩放倍数，也是一个 Tensor。
- variable\_epsilon, 一个不为 0 的浮点数。
- name: 操作的名字，可选。

batch normalization 计算方式是:

$$x = (x - \bar{x}) / \sqrt{Var(x) + variable\_epsilon} \quad (1.1)$$

$$x = x \times scale + offset \quad (1.2)$$

(1.3)

$$\text{均值: } \bar{x} = \frac{1}{m} \sum_{i=1}^m x_i \quad (1.4)$$

$$\text{方差: } \sigma^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \bar{x})^2 \quad (1.5)$$

### 1.1.4 常见的激活函数

- relu
- sigmoid
- tanh
- elu
- bias\_add

- relu6
- softplus
- softsign

## 1.2 relu 函数

### 1.2.1 relu

relu 函数在自变量  $x$  小于 0 时值全为 0, 在  $x$  大于 0 时, 值和自变量相等。

```
import tensorflow as tf
import matplotlib.pyplot as plt
x = tf.linspace(-10., 10., 100)
y = tf.nn.relu(x)
with tf.Session() as sess:
    [x,y] = sess.run([x,y])
    plt.plot(x,y,'r',6,6,'bo')
    plt.title('relu')
    ax = plt.gca()
    ax.annotate("", xy=(6, 6), xycoords='data',
                xytext=(6, 4.5), textcoords='data',
                arrowprops=dict(arrowstyle="->",
                               connectionstyle="arc3"),
                )
    ax.annotate("", xy=(6, 6), xycoords='data',
                xytext=(10, 6), textcoords='data',
                arrowprops=dict(arrowstyle="->",
                               connectionstyle="arc3"),
                )
    ax.grid(True)
    plt.xlabel('x')
    plt.ylabel('relu(x)')
    plt.savefig('relu.png',dpi = 600)
```

### 1.2.2 relu6

relu6 函数和 relu 不同之处在于在  $x$  大于等于 6 的部分值保持为 6。

```
import tensorflow as tf
import matplotlib.pyplot as plt
```

```
x = tf.linspace(-10.,10.,100)
y = tf.nn.relu6(x)
with tf.Session() as sess:
    [x,y] = sess.run([x,y])
plt.plot(x,y,'r',6,6,'bo')
plt.title('relu6')
ax = plt.gca()
ax.annotate("",
            xy=(6, 6), xycoords='data',
            xytext=(6, 4.5), textcoords='data',
            arrowprops=dict(arrowstyle="->",
                           connectionstyle="arc3"),
            )
ax.grid(True)
plt.xlabel('x')
plt.ylabel('relu(x)')
plt.savefig('relu6.png',dpi = 600)
```

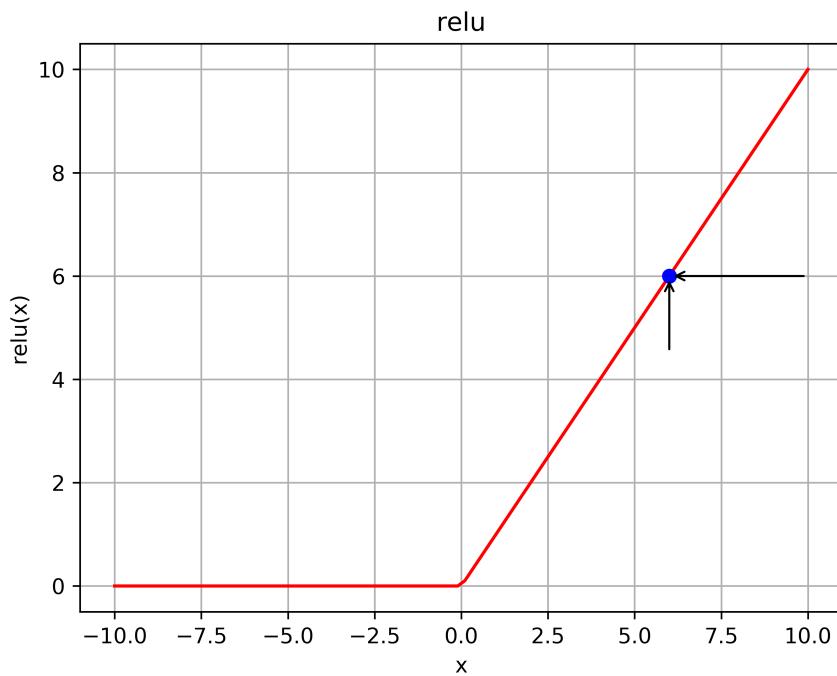


图 1.1: relu

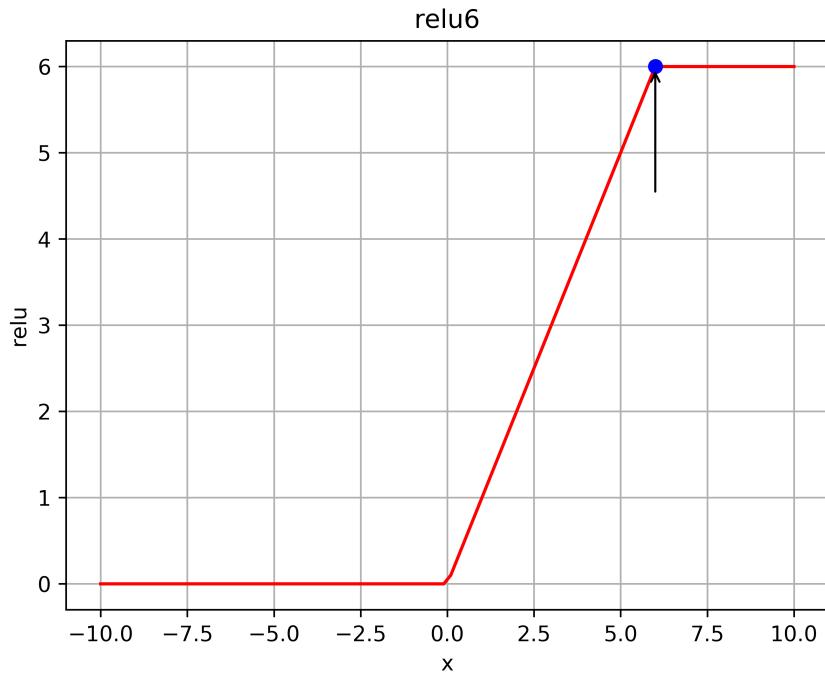


图 1.2: relu6

## 1.2.3 sigmoid

```

import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
x = tf.linspace(-10.,10.,100)
y1 = tf.nn.sigmoid(x)
y2 = tf.nn.tanh(x)
red_patch = mpatches.Patch(color = 'red',label = 'sigmoid')
blue_patch = mpatches.Patch(color = 'blue',label = 'tanh')
with tf.Session() as sess:
    [x,y1,y2] = sess.run([x,y1,y2])
    plt.plot(x,y1,'r',x,y2,'b')
    ax = plt.gca()
    ax.annotate(r"\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}",
                xy=(0,0),xycoords="data",
                xytext=(1,0),textcoords="data",
                arrowprops=dict(arrowstyle="->",
                               connectionstyle="arc3"),
                )
    ax.annotate(r"\$sigmoid(x) = \frac{1}{1+e^{-x}}\$",
                xy=(0,0),xycoords="data",
                xytext=(1,0),textcoords="data",
                arrowprops=dict(arrowstyle="->",
                               connectionstyle="arc3"),
                )

```

```

xy=(0,0.5), xycoords="data",
xytext=(1,0.5), textcoords="data",
arrowprops=dict (arrowstyle="->",
connectionstyle="arc3"),
)
plt.xlabel('x')
plt.grid(True)
plt.legend(handles = [red_patch,blue_patch])
plt.savefig('activate.png',dpi=600)

```

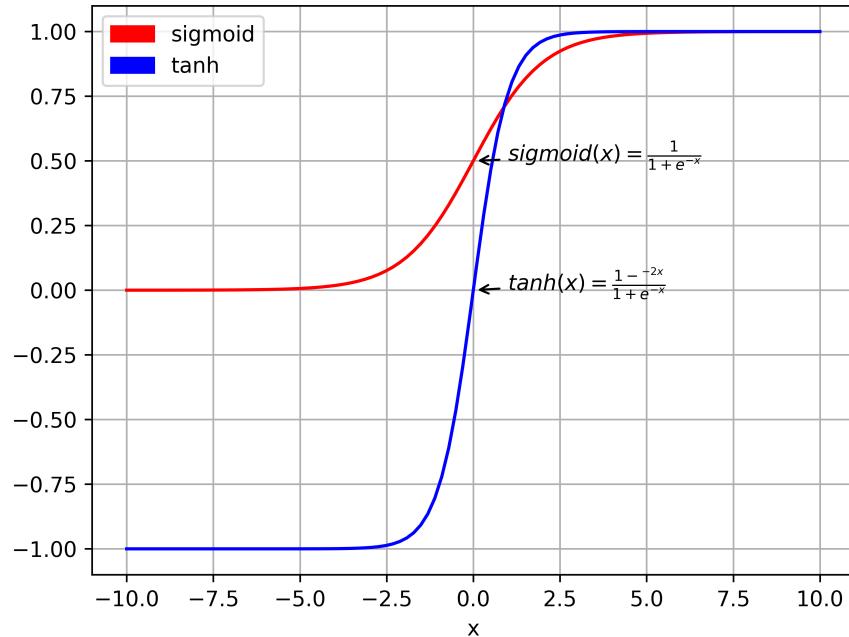


图 1.3: activate\_fun

#### 1.2.4 relu 和 softplus

```

import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
x = tf.linspace(-10.,10.,100)
y2 = tf.nn.softplus(x)
y3 = tf.nn.relu(x)
blue_patch = mpatches.Patch(color = 'blue',label = 'softplus')
yellow_patch = mpatches.Patch(color = 'yellow',label = 'relu')

```

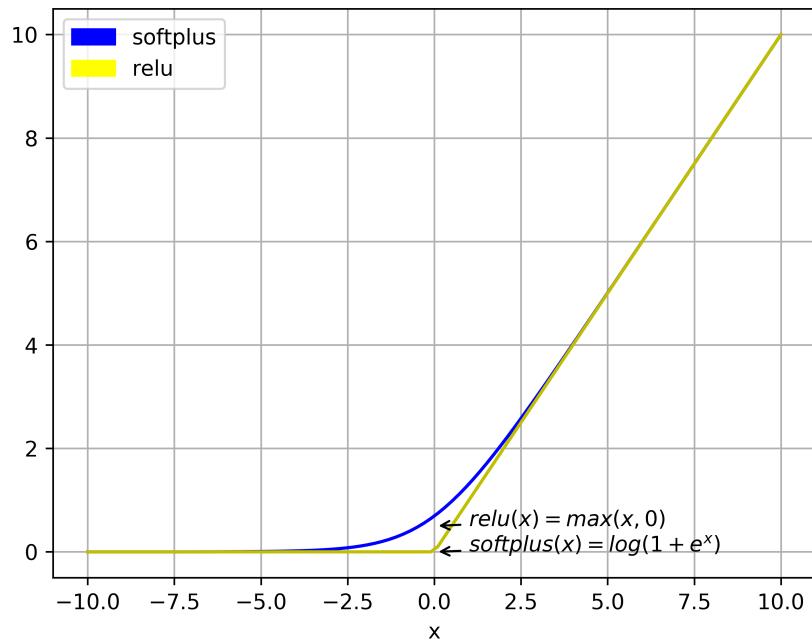
## 1.2. RELU 函数

7

```

with tf.Session() as sess:
    [x,y2,y3] = sess.run([x,y2,y3])
    plt.plot(x,y2,'b',x,y3,'y')
    ax = plt.gca()
    plt.xlabel('x')
    ax.annotate(r"$softplus(x)=\log(1+e^x)$",
                xy=(0,0),xycoords="data",
                xytext=(1,0),textcoords="data",
                arrowprops=dict(arrowsize=2,
                                connectionstyle="arc3"),
                )
    ax.annotate(r"$relu(x)=\max(x, 0)$",
                xy=(0,0.5),xycoords="data",
                xytext=(1,0.5),textcoords="data",
                arrowprops=dict(arrowsize=2,
                                connectionstyle="arc3"),
                )
    plt.grid(True)
    plt.legend(handles = [blue_patch,yellow_patch])
    plt.savefig('relu_softplus.png',dpi=600)

```



### 1.2.5 dropout

将神经元以概率 keep\_prob 绝对是否被抑制。如果被抑制该神经元的输出为 0 如果不被抑制,该神经元的输出将被放大到原来的  $1/\text{keep\_prop}$ 。默认情况下,每个神经元是否被抑制是相互独立的。但是是否被抑制也可以通过 noise\_shape 来调节。当 noise\_shape[i]=shape(x)[i] 时,x 中的元素相互独立。如果 shape(x)=[k,1,1,n], 那么每个批通道都是相互独立的, 但是每行每列的数据都是关联的, 也就是说要么都为 0, 要么还是原来的值。

```
import tensorflow as tf
a = tf.constant([[-1., 2., 3., 4.]])
with tf.Session() as sess:
    b = tf.nn.dropout(a, 0.5, noise_shape=[1, 4])
    print(sess.run(b))
    c = tf.nn.dropout(a, 0.5, noise_shape=[1, 1])
    print(sess.run(c))
```

`[[ -2. 0. 0. 8.]]`

`[[ -0. 0. 0. 0.]]`

当输入数据特征相差明显时, 用 tanh 效果会很好, 但在循环过程中会不断扩大特征效果并显示出来。当特征相差不明显时, sigmoid 效果比较好。同时, 用 sigmoid 和 tanh 作为激活函数时, 需要对输入进行规范化, 否则激活厚的值全部进入平坦区, 隐藏层的输出会趋同, 丧失原来的特征表达, 而 relu 会好很多, 优势可以不需要输入规范化来避免上述情况。因此, 现在大部分卷积神经网络都采用 relu 作为激活函数。

## 1.3 卷积函数

`tf.nn.conv2d(input,filter,padding,stride=None,dilation_rate=None,name=None,data_format=None)`

- input: 一个 tensor, 数据类型必须是 float32, 或者是 float64
- filter: 一个 tensor, 数据类型必须和 input 相同。
- strides: 一个长度为 4 的一组证书类型数组, 每一维对应 input 中每一维对应移动的步数, strides[1] 对应 input[1] 移动的步数。
- padding: 有两个可选参数'VALID' (输入数据维度和输出数据维度不同) 和'SAME' (输入数据维度和输出数据维度相同)
- use\_cudnn\_on\_gpu: 一个可选的布尔值, 默认情况下时 True。
- name: 可选, 操作的一个名字。

```

import tensorflow as tf
input_data = tf.Variable(tf.random_normal(shape = [10,9,9,3],mean=0,stddev=1),
                       dtype = tf.float32)
kernel = tf.Variable(tf.random_normal(shape = [2,2,3,2],mean = 0,stddev=1,dtype=
                                      tf.float32))

y = tf.nn.conv2d(input_data,kernel,strides=[1,1,1,1],padding='SAME')
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    print(sess.run(y).shape)

```

输出形状为 [10,9,9,2]。

## 1.4 池化

池化函数	功能
① tf.nn.avg_pool(value,ksize,strides,padding,data_format='NHWC',name =None)	平均池化
② tf.nn.max_pool(value,ksize,strides,padding,data_format='NHWC',name =None)	最大池化
③ tf.nn.max_pool_with_argmax(input,ksize,strides,padding,Targmax=None,name =None)	最大池化返回索引
④ tf.nn.avg_pool3d(input,ksize,strides,padding,name =None)	三维状态下平均池化
⑤ tf.nn.max_pool3d(input,ksize,strides,padding,name =None)	三维状态下最大池化
⑥ tf.nn.fractional_avg_pool(value,pooling_ratio,pseudo_random=None,overlapping=None,seed1=None,seed2=None,name = None)	三维状态下的平均池化
⑦ tf.nn.avg_max_pool(value,pooling_ratio,pseudo_random=None,overlapping=None,seed1=None,seed2=None,name = None)	三维状态下的最大池化
⑧ tf.nn.pool(input,window_shape,pool_type,padding,dilation_rate = None,strides=None,name=None,data_format=None)	执行一个 N

- value: 一个四维 Tensor, 维度时 [batch,height,width,channels]。
- ksize: 一个长度不小于 4 的整型数据, 每一位上的值对应于输入数据 Tensor 中每一维窗口对应值。
- stride: 一个长度不小于 4 的整型列表。该参数指定窗口在输入数据 Tensor 每一维上的步长。
- padding: 一个字符串, 取值为 SAME 或者 VALID。

- data\_format:NHWC。

## 1.5 常见的分类函数

`tf.nn.sigmoid_cross_entropy_with_logits(logits,targets,name=None)`

- logits:[batch\_size,num\_classes]
- targets:[batch\_size,size]
- 输出： loss[batch\_size,num\_classes]

最后已成不需要进行 sigmoid 操作。

`tf.nn.softmax(logits,dim=-1,name=None)`: 计算 Softmax

$$\text{softmax} = \frac{x^{\logits}}{\text{reduce\_sum}(e^{\logits}, dim)}$$

`tf.nn.log_softmax(logits,dim=-1,name = None)` 计算 log softmax

$$\text{logsoftmax} = \logits - \log(\text{reduce\_softmax}(\exp(\logits), dim))$$

`tf.nn.softmax_cross_entropy_with_logits(_setinel=None,labels=None,logits=None,dim=-1,name=None)` 输出 loss:[batch\_size] 保存的时 batch 中每个样本的交叉熵。`tf.nn.sparse_softmax_cross_entropy`

- logits: 神经网络最后一层的结果。
- 输入 logits:[batch\_size,num\_classes],labels:[batch\_size], 必须在 [0,num\_classes]
- loss[batch], 保存的是 batch 每个样本的交叉熵。

## 1.6 优化方法

- `tf.train.GradientDescentOptimizer`
- `tf.train.AdadeltaOptimizer`
- `tf.train.AdagradDAOptimizer`
- `tf.train.AdagradOptimizer`
- `tf.train.MomentumOptimizer`
- `tf.train.AdamOptimizer`
- `tf.train.FtrlOptimizer`
- `tf.train.RMSPropOptimizer`

### 1.6.1 BGD

BGD(batch gradient descent) 批量梯度下降。这种方法是利用现有的参数对训练集中的每一个输入生成一个估计输出  $y_i$ , 然后跟实际的输出  $y_i$  比较, 统计所有的误差, 求平均后的到平均误差作为更新参数的依据。啊他的迭代过程是:

1. 提取巡检集中所有内容  $\{x_1, \dots, x_n\}$ , 以及相关的输出  $y_i$ ;
2. 计算梯度和误差并更新参数。

这种方法的优点是: 使用所有数据计算, 都保证收敛, 并且并不需要减少学习率缺点是每一步需要使用所有的训练数据, 随着训练的进行, 速度会变慢。那么如果将训练数据拆分成一个个 batch, 每次抽取一个 batch 数据更新参数, 是不是能加速训练? 这就是 SGD。

### 1.6.2 SGD

SGD(stochastic gradient descent): 随机梯度下降。这种方法的主要思想是将数据集才分成一个个的 batch, 随机抽取一个 batch 计算并更新参数, 所以也称为 MBGD(minibatch gradient descent) SGD 在每次迭代计算 mini-batch 的梯度, 然后队参数进行更新。和 BGD 相比, SGD 在训练数据集很大时也能以较快的速度收敛, 但是它有两个缺点:

1. 需要手动调整学习率, 但是选择合适的学习率比较困难。尤其在训练时, 我们常常想队常出现的特征更新速度快点, 队不长出现的特征更新速度慢些, 而 SGD 对更新参数时对所有参数采用一样的学习率, 因此无法满足要求。
2. SGD: 容易收敛到局部最优。

### 1.6.3 momentum

Momentum 时模拟物理学中的动量概念, 更新时在一定程度上保留之前的更新方向, 利用当前批次再次微调本次更新参数, 因此引入了一个新的变量  $v$ , 作为前几次梯度的累加。因此, momentum 能够更新学习率, 在下降初期, 前后梯度方向一致时能加速学习: 在下降的中后期, 在局部最小值附近来回振荡, 能够抑制振荡加快收敛。

### 1.6.4 Nesterov Momentum

标准的 Monentum 法首先计算一个梯度, 然后子啊加速更新梯度的方向进行一个大的跳跃 Nesterov 首先在原来加速的梯度方向进行一个大的跳跃, 然后在改为值设置计算梯度值, 然后用这个梯度值修正最终的更新方向。

### 1.6.5 Adagrad

Adagrade 能够自适应的为各个参数分配不同的学习率，能够控制每个维度的梯度方向，这种方法的优点是能实现学习率的自动更改，如果本次更新时梯度大，学习率就衰减得快，如果这次更新时梯度小，学习率衰减得就慢些。

### 1.6.6 RMSprop

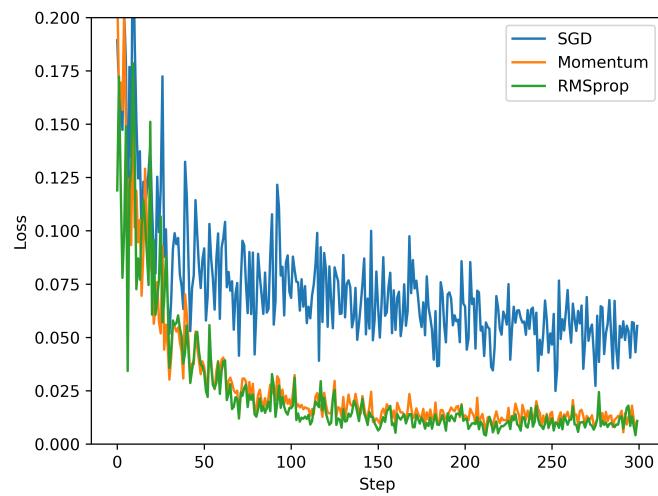
和 Momentum 类似，通过引入衰减系数使得每个回合都衰减一定比例。在实践中，对循环神经网络效果很好。

### 1.6.7 Adam

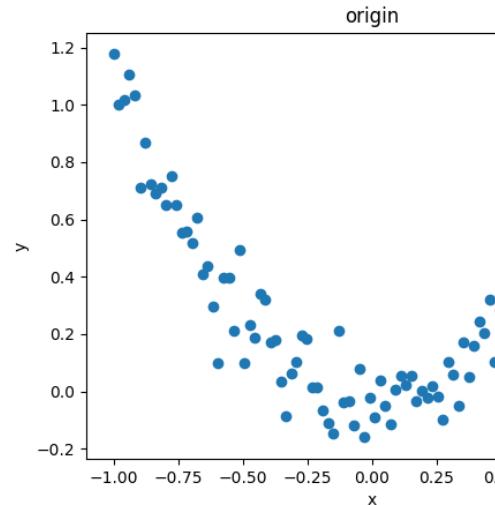
名称来自自适应矩阵 (adaptive moment estimation).Adam 更均损失函数针对每个参数的一阶矩，二阶矩估计动态调整每个参数的学习率。

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
tf.set_random_seed(0)
np.random.seed(0)
LR = 0.01
BATCH_SIZE = 32
x = np.linspace(-1, 1, 100).reshape(-1, 1)
noise = np.random.normal(0, 0.1, size=x.shape)
y = np.power(x, 2)+noise
class Net:
    def __init__(self, opt, **kwargs):
        self.x = tf.placeholder(tf.float32, [None, 1])
        self.y = tf.placeholder(tf.float32, [None, 1])
        l = tf.layers.dense(self.x, 20, tf.nn.relu)
        out = tf.layers.dense(l, 1)
        self.loss = tf.losses.mean_squared_error(self.y, out)
        self.train = opt(LR, **kwargs).minimize(self.loss)
net_SGD = Net(tf.train.GradientDescentOptimizer)
net_momentum = Net(tf.train.MomentumOptimizer, momentum=0.9)
net_RMSprop = Net(tf.train.RMSPropOptimizer)
net_Adam = Net(tf.train.AdamOptimizer)
nets = [net_SGD, net_momentum, net_RMSprop, net_Adam]
sess = tf.Session()
sess.run(tf.global_variables_initializer())
losses_his = [[], [], []]
for step in range(300):
```

```
index = np.random.randint(0, x.shape[0], BATCH_SIZE)
b_x = x[index]
b_y = y[index]
for net, l_his in zip(nets, losses_his):
    _, l = sess.run([net.train, net.loss], {net.x:b_x, net.y:b_y})
    l_his.append(l)
labels = ['SGD', 'Momentum', 'RMSprop', 'Adam']
for i, l_his in enumerate(losses_his):
    plt.plot(l_his, label=labels[i])
plt.legend(loc='best')
plt.xlabel('Step')
plt.ylabel('Loss')
plt.ylim(0, 0.2)
plt.savefig('Opt.png', dpi=600)
```



### 1.6.8 构造简单的神经网络拟合数据



原始数据为  $y=x^2$  的基础上添加随机噪声。原始数据的散点图如下

```
#tensorflow 1.2.1
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
tf.set_random_seed(0)
np.random.seed(0)
#生成数据
step = 100
x = np.linspace(-1,1,step).reshape(-1,1)
noise = np.random.normal(0,0.1,size=x.shape)
y = np.power(x,2)+noise

tf_x = tf.placeholder(tf.float32,x.shape)
tf_y = tf.placeholder(tf.float32,x.shape)
l1 = tf.layers.dense(tf_x,10,tf.nn.relu)
output = tf.layers.dense(l1,1)

loss = tf.losses.mean_squared_error(tf_y,output)
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5)
train_op = optimizer.minimize(loss)

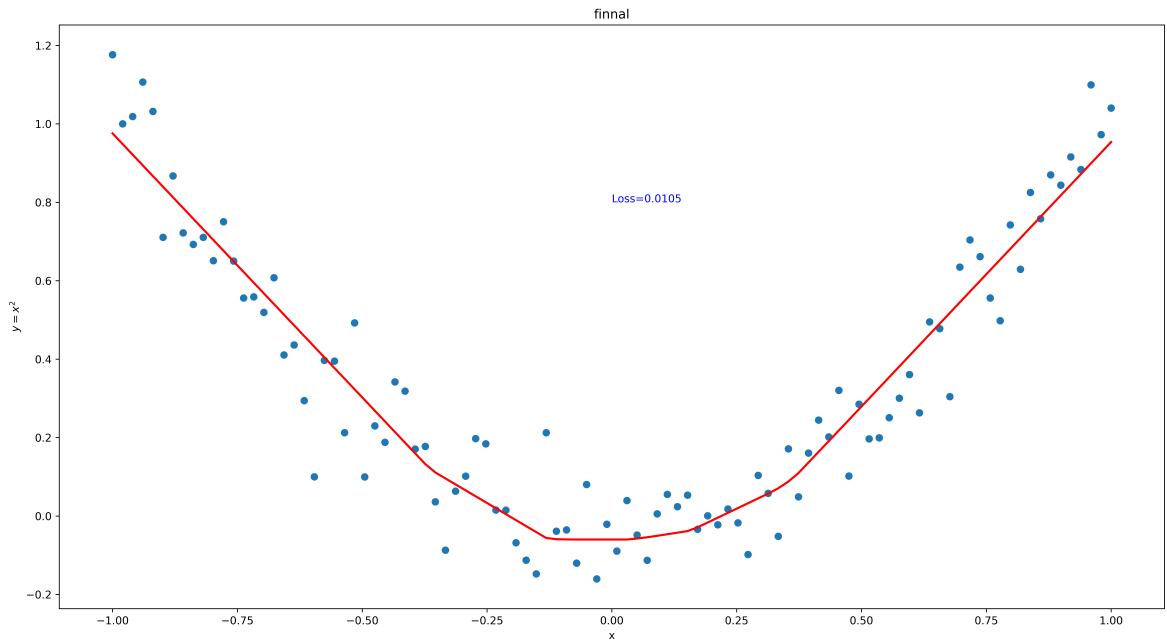
sess = tf.Session()
sess.run(tf.global_variables_initializer())
plt.ion()
for step in range(100):
    _,l,pred = sess.run([train_op,loss,output],{tf_x:x,tf_y:y})
```

```

if step%5==0:
    plt.cla()
    plt.scatter(x,y)
    plt.title(r'y=x^2+noise')
    plt.plot(x,pred,'r-',lw=2)
    plt.text(0,0.8,'Loss=%f'%l,fontdict={'size':10,'color':'blue'})
    plt.xlabel("x")
    plt.ylabel(r'y=x^2')
    plt.pause(0.1)
plt.ioff()
plt.show()

```

最终拟合数据:



## 1.7 TensorBoard

```

import tensorflow as tf
import matplotlib.pyplot as plt

tf.set_random_seed(1)
x0 = tf.random_normal((100,2),2,2,tf.float32,0)
y0 = tf.zeros(100)

```

```

x1 = tf.random_normal((100,2), -2, 2, tf.float32, 0)
y1 = tf.ones(100)
x = tf.reshape(tf.stack((x0,x1), axis=1), (200,2))
y = tf.reshape(tf.stack((y0,y1), axis=1), (200,1))
with tf.Session() as sess:
    x = sess.run(x)
    y = sess.run(y)

tf_x = tf.placeholder(tf.float32, x.shape)      # input x
tf_y = tf.placeholder(tf.int32, y.shape)      # input y

# neural network layers
l1 = tf.layers.dense(tf_x, 10, tf.nn.relu)          # hidden layer
output = tf.layers.dense(l1, 2)                      # output layer

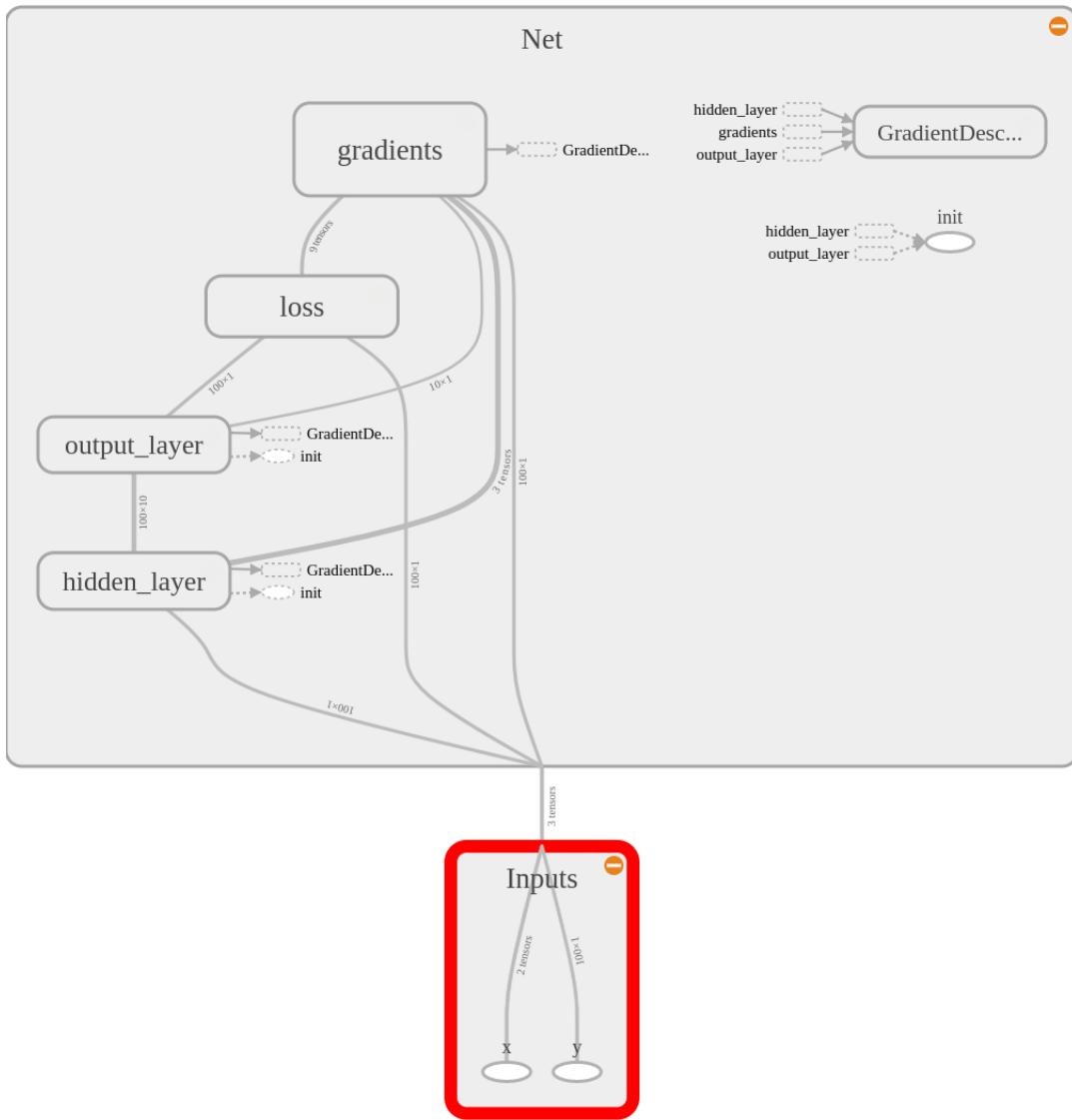
loss = tf.losses.sparse_softmax_cross_entropy(labels=tf_y, logits=output)
                                         # compute cost
accuracy = tf.metrics.accuracy(                # return (acc, update_op), and create 2
                                labels=tf.squeeze(tf_y), predictions=tf.argmax(output, axis=1),)[1]
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.05)
train_op = optimizer.minimize(loss)

sess = tf.Session()

                                         # control training and others
init_op = tf.group(tf.global_variables_initializer(), tf.
                   local_variables_initializer())
sess.run(init_op)      # initialize var in graph

plt.ion()    # something about plotting
for step in range(100):
    _, acc, pred = sess.run([train_op, accuracy, output], {tf_x: x, tf_y: y})
    if step % 2 == 0:
        plt.cla()
        plt.scatter(x[:, 0], x[:, 1], c=pred.argmax(1), s=100, lw=0, cmap='RdYlGn')
        plt.text(1.5, -4, 'Accuracy=% .2f' % acc, fontdict={'size': 20, 'color': 'red'})
        plt.pause(0.1)
plt.ioff()
plt.show()

```



### 1.7.1 TensorBoard Histogram Dashboard

TensorBoard Histogram Dashboard 显示 TensorFlow 图中的 Tensor 如何随着时间变化。

### 1.7.2 一个简单的例子

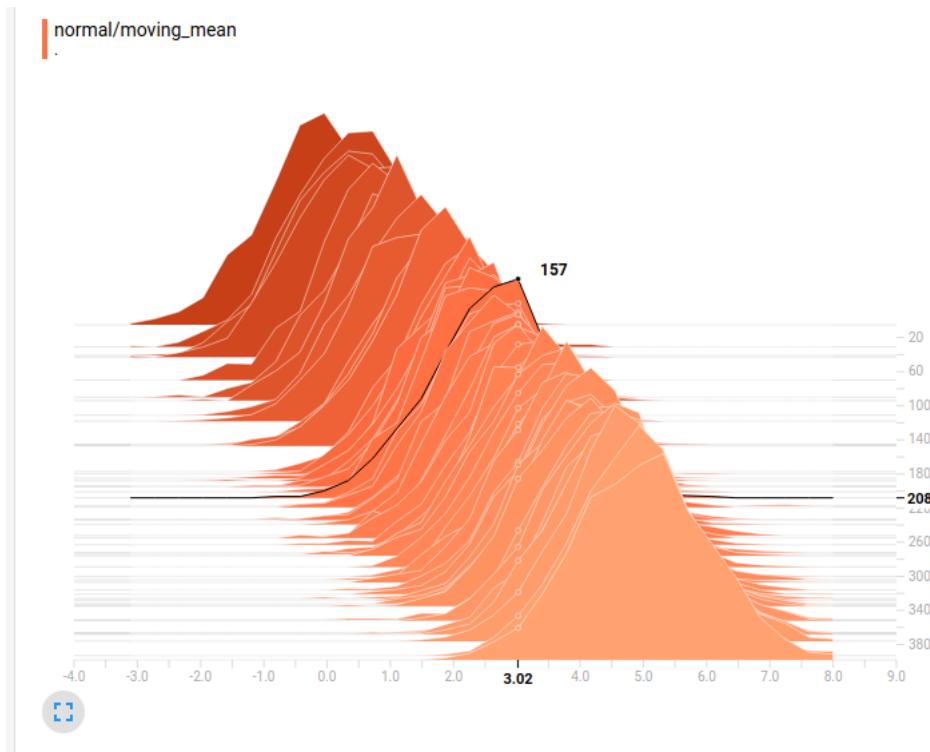
正态分布变量，均值随着和时间移动。TensorFlow 有一个操作 `tf.random_normal` 可以完美的达到这个目的。正如通常情况下 TensorBoard，我们将用 `summary op` 融合数据

据。在这种情况下'tf.summary.histogram'。这里有一个代码段将生成一些包含正态分布直方图数据的总结，这里均值随着时间增大。

```
import tensorflow as tf
k = tf.placeholder(tf.float32)
mean_moving_normal = tf.random_normal(shape=[1000], mean=(5*k), stddev=1)
summaries = tf.summary.histogram('normal/moving_mean', mean_moving_normal)
sess = tf.Session()
writer = tf.summary.FileWriter('./histogram_example')
N = 400
for step in range(N):
    k_val = step/float(N)
    summ = sess.run(summaries, feed_dict={k:k_val})
    writer.add_summary(summ, global_step=step)
```

在当前代码中运行下边的代码启动 TensorFlow 载入数据

```
tensorboard --logdir=./histogram_example
```



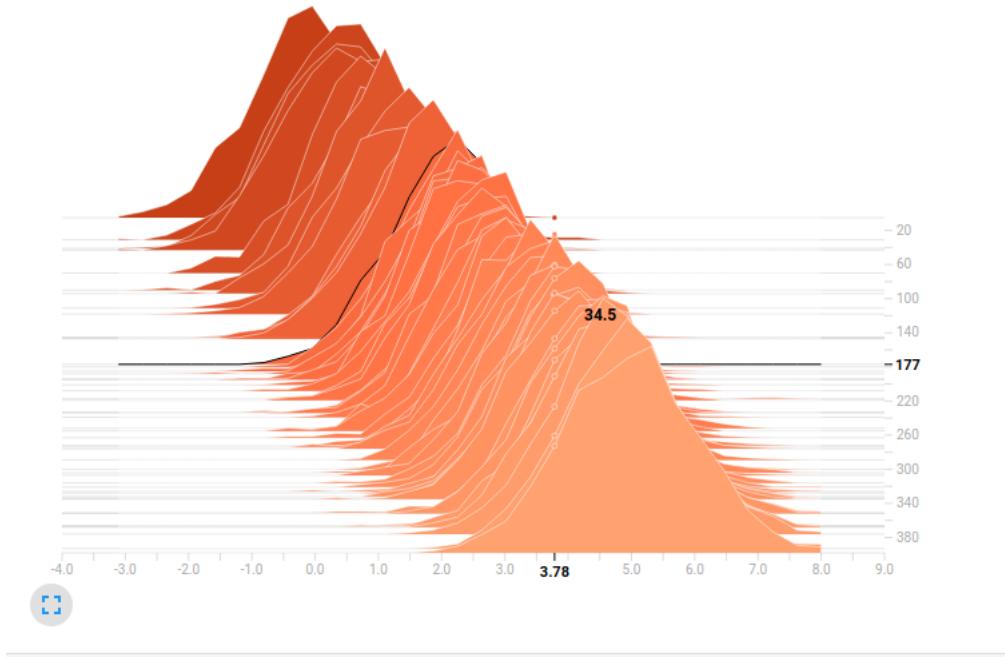
tf.summary.histogram 接受任一尺寸和大小的 Tensor，压缩他们进入直方数据结构组成一些小的数据宽度和数量组层的 bin 将该够，例如我们像组成数 [0.5,1.1,1.3,2.2,2.9,2.99] 成 3

个 bin，我们可以创建三个 bin：一个包含 0 到 1 之间的一切 (0.5)，一个包含 1-2(1.1,1.3) 之间，一个包含 2-3(2.2,2.9,2.99)

TensorFlow 用类是的方法创建 bins，但是不想我们上面的例子，它不创建整数读额 bins，瑞与大型数据，稀疏数据，这样的也许导致上千个 bin，bins 时指数分布时，一些 bins 相比于一些非常大数的 bin 接近于 0。然而，可视化指数分布 bin 时一个技巧，如果高被编码为数量，bin 宽度更大的空间，甚至他们有相同的元素，相比较之下统计数量使得豪赌比较变得可能，直方图采集数据仅均匀的 bins，这可能导致不幸的人工操作。

在直方图可视化器的每一个切片显示为一个单个的直方图。切片安装步数组组织。例老的切片 (e.g. step 0) 比较靠后变为更深，然而新的 slices 接近于前景色，颜色更轻，右边的 y 轴显示了步数。

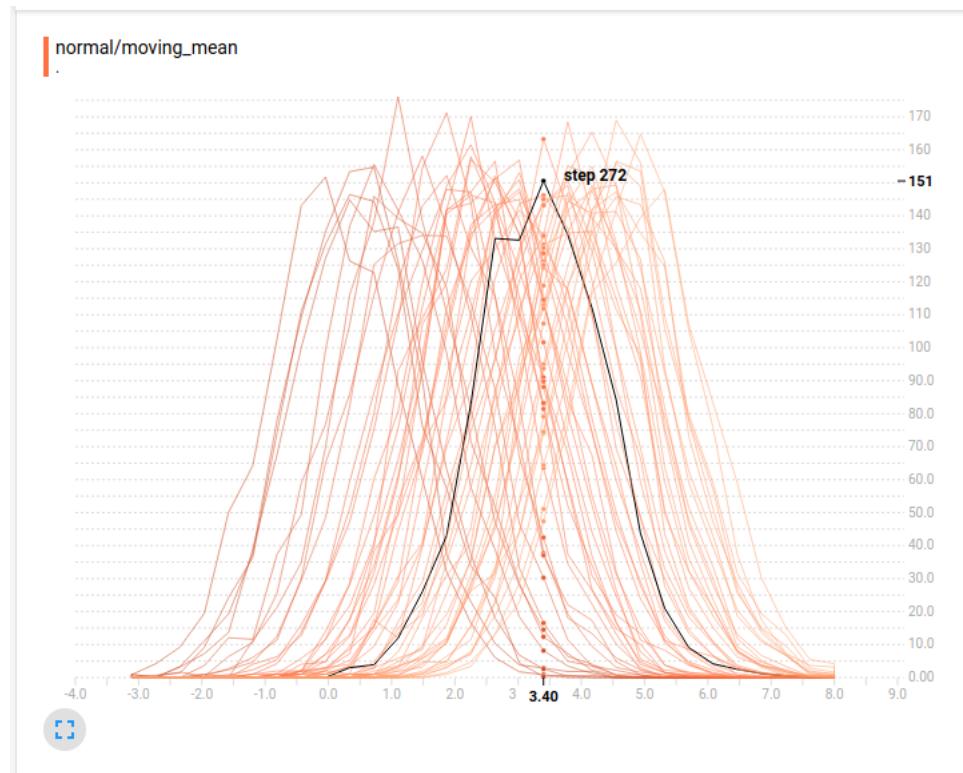
你可以在直方图上滑动鼠标看到更多的详细星系。你如下面的图你可以看到直方图的时间不为 177 有一个 bin 中心在 3.78 有 bin 中有 34.5 个元素。



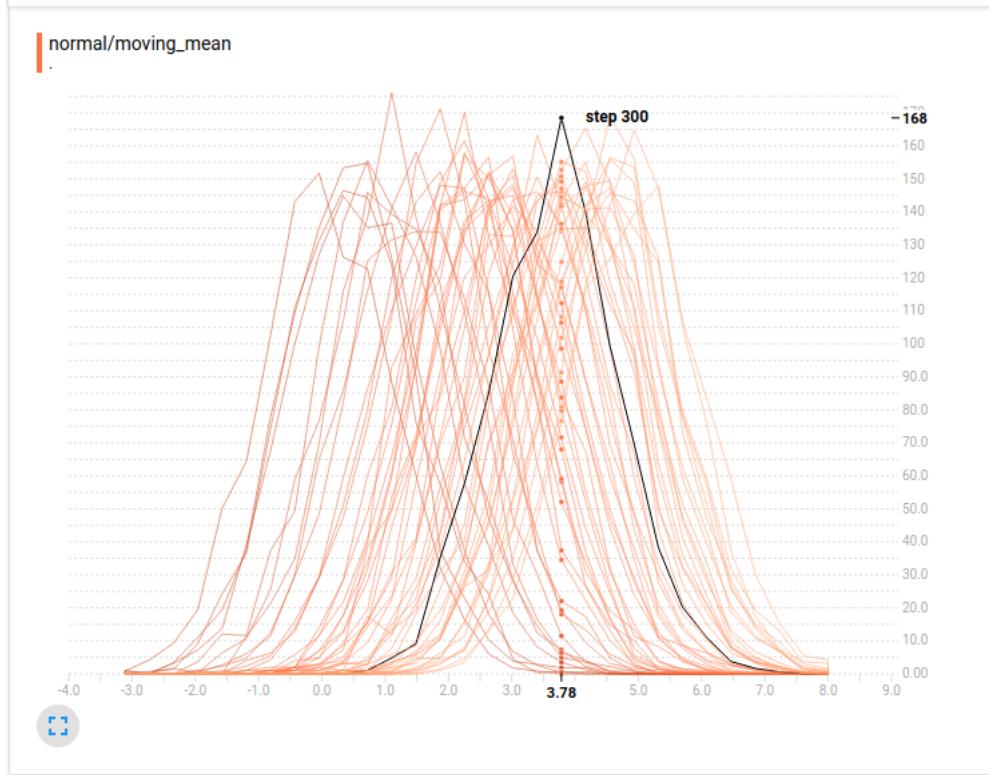
你也许注意到注意到直方切片在统计步数和时间上不总是偶数，这是因为 TensorBoard 用[reservoir sampling](#)保持直方图的子集，为了节约内存，Reservior sampling 保证每个采样有一个相等的可能性被包含进去，但是因为它时一个随机算法，采样并不在每个偶数步发生。

### 1.7.3 Overlay Mode

控制面板上允许你打开直方图模式为 offset 为 overlay。在 offset 模式下，可视化转动 45 度，因此单个的直方图切片不再展开，而是所有的图共享一个相同的 y 轴上。



现在表上的每个切片被线分开，y 轴显示每个 bucket 项目数量，深色线时老的，早期的时间不，浅色线时最近的新时间不，你可以用鼠标在表上查看更多的信息。



overlay 可视化在你想直接比较不同直方图的数量。

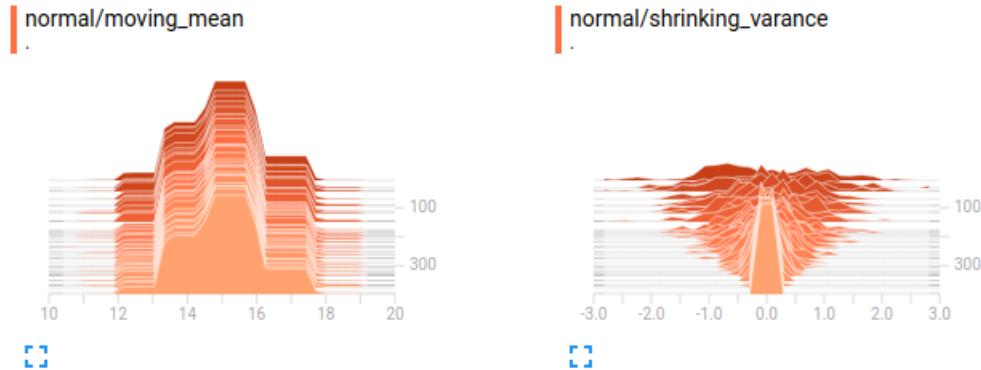
#### 1.7.4 多个分布

直方图控制面板对多分布下的可视化很有用，当我们通过链接两个不同的正态分布构造一个简单的二两分布，代码如下：

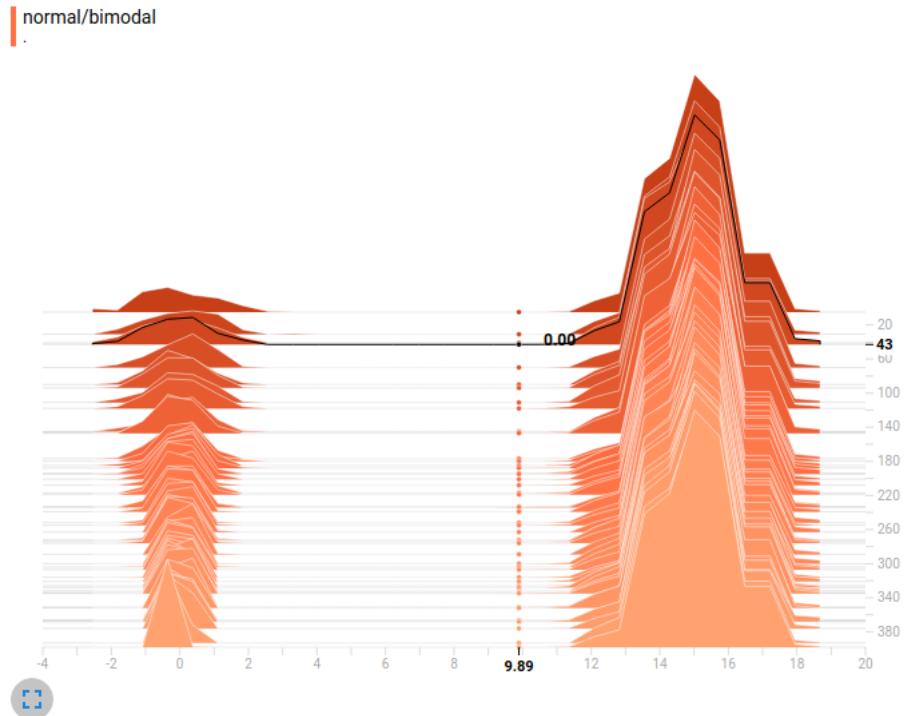
```
import tensorflow as tf
k = tf.placeholder(tf.float32)
mean_moving_normal = tf.random_normal(shape=[1000], mean=(3*5), stddev=1)
tf.summary.histogram('normal/moving_mean', mean_moving_normal)
variance_shrinking_normal = tf.random_normal(shape=[100], mean=0, stddev=1-(k))
tf.summary.histogram('normal/shrinking_varance', variance_shrinking_normal)
normal_combined = tf.concat([mean_moving_normal, variance_shrinking_normal], 0)
tf.summary.histogram('normal/bimodal', normal_combined)
summaris = tf.summary.merge_all()
sess = tf.Session()
writer = tf.summary.FileWriter('./histgram_example1')
N = 400
for step in range(N):
```

```
k_val = step/float(N)
summ = sess.run(summaris, feed_dict={k:k_val})
writer.add_summary(summ, global_step=step)
```

上面的例子是滑动平均，现在我们已有一个收缩的变量分布。



当我们链接她们在一起，我们得到一个清晰解释分歧，二进制结构的表格：



### 1.7.5 更多分布

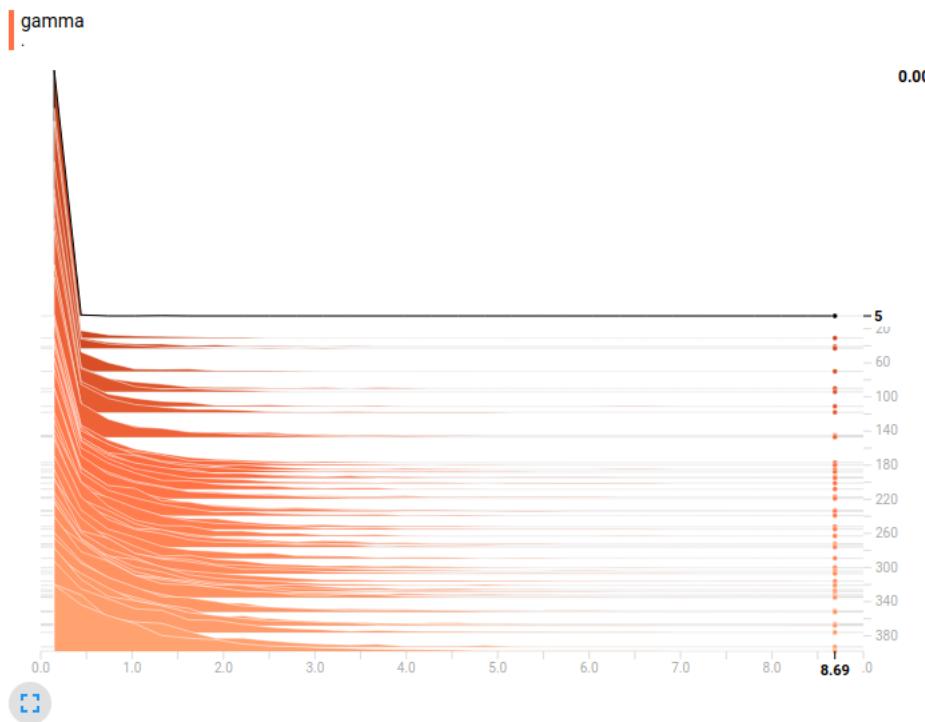
生成可视化更多分布，结合他们到表中：

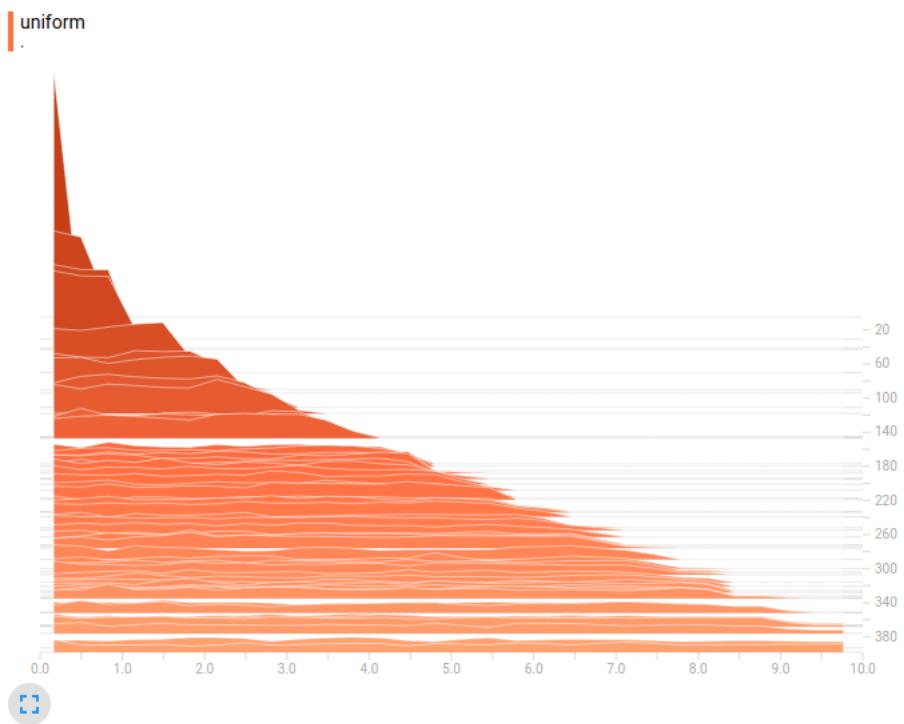
```

import tensorflow as tf
k = tf.placeholder(tf.float32)
# Make a normal distribution ,with a shift mean
mean_moving_normal = tf.random_normal(shape=[1000],mean=(5*k),stddev=1)
tf.summary.histogram('normal/moving_mean',mean_moving_normal)
variance_shrinking_normal = tf.random_normal(shape=[1000],mean=0,stddev=1-(k))
tf.summary.histogram('normal/shrinking_variance',variance_shrinking_normal)
normal_combined = tf.concat([mean_moving_normal,variance_shrinking_normal],0)
tf.summary.histogram("normal/bimodal",normal_combined)
#add gamma distribution
gamma = tf.random_gamma(shape=[1000],alpha=k)
tf.summary.histogram('gamma',gamma)
poisson = tf.random_poisson(shape=[1000],lam=k)
tf.summary.histogram('poisson',poisson)
#add a uniform distribution
uniform = tf.random_uniform(shape=[1000],maxval=k*10)
tf.summary.histogram('uniform',uniform)
#finnally combine everything together

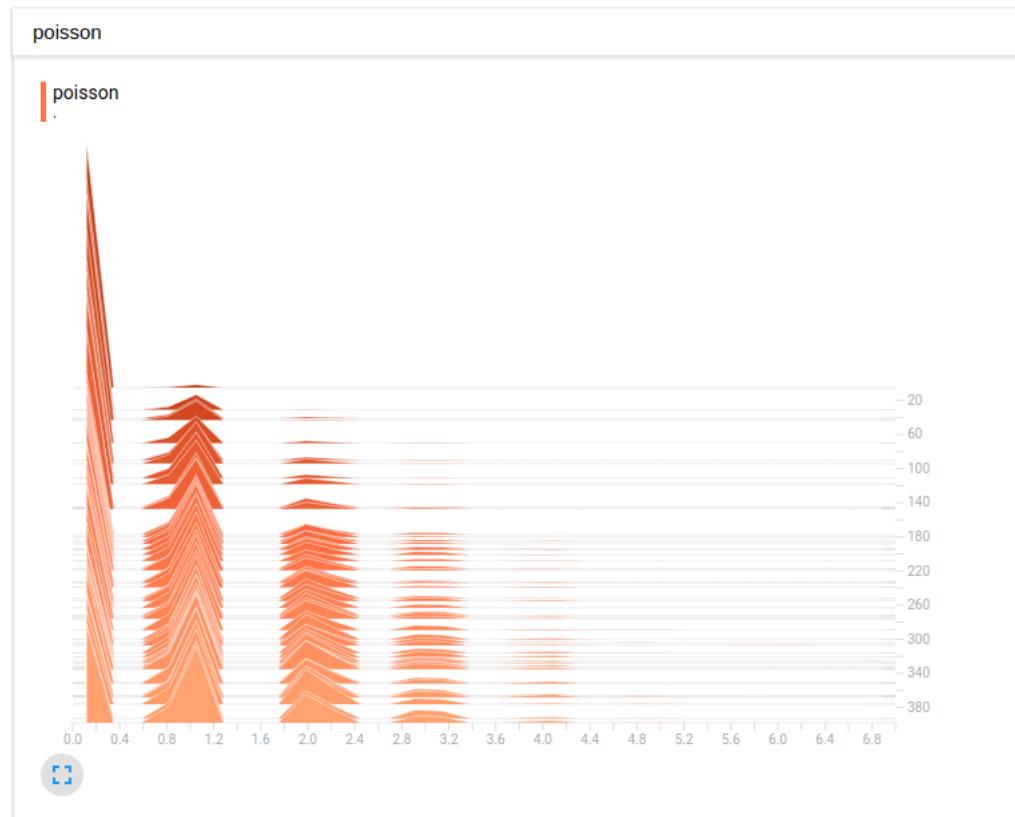
all_distributions = [mean_moving_normal,variance_shrinking_normal,gamma,poisson,
                     uniform]
all_combined = tf.concat(all_distributions,0)
tf.summary.histogram('all_combined',all_combined)
summaries = tf.summary.merge_all()
sess = tf.Session()
writer = tf.summary.FileWriter('./histogram_example2')
N = 400
for step in range(N):
    k_val = step/float(N)
    summ = sess.run(summaries,feed_dict={k:k_val})
    writer.add_summary(summ,global_step=step)

```



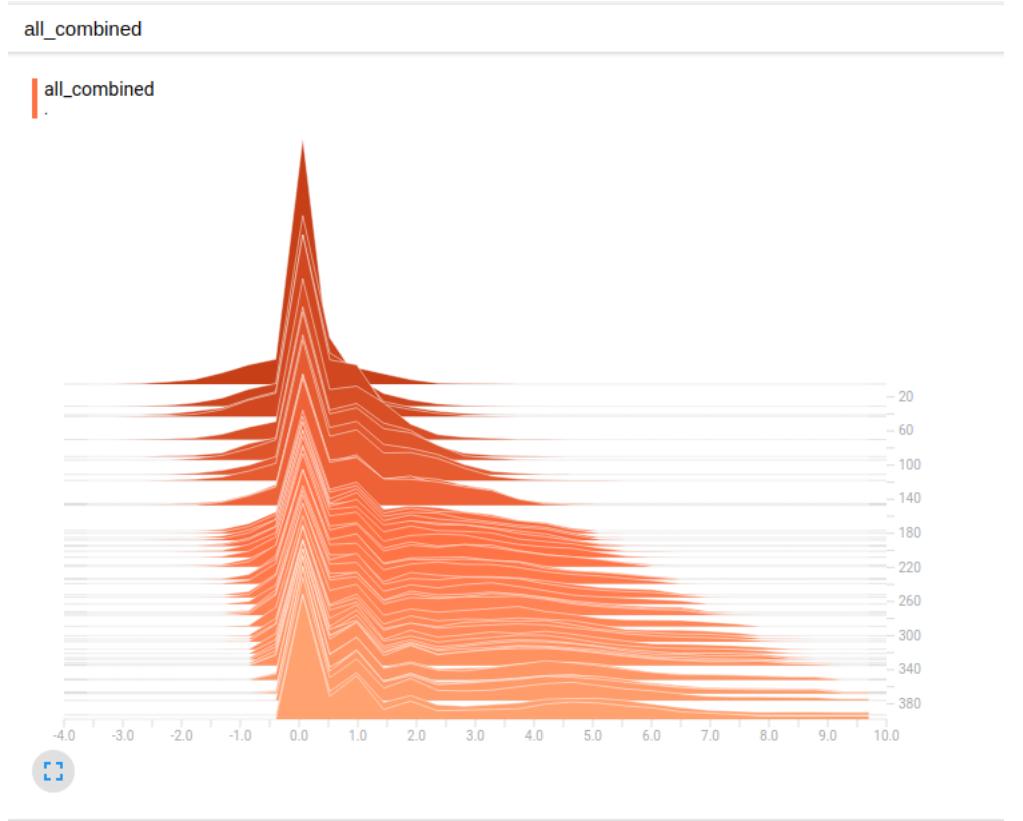


### 1.7.6 poisson 分布



poisson 分布定义在整数上，因此所有被生成的值都是整数，直方图压缩移动数据到浮点 bins，导致可视化在整数值上显示一点点突起。

### 1.7.7 结合所有的数据到一张图向上



```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
tf.set_random_seed(0)
np.random.seed(0)
x = np.linspace(-1,1,100).reshape(-1,1)
noise = np.random.normal(0,0.1,size=x.shape)
y = np.power(x,2)+noise
def gendata():
    t = np.linspace(-1,1,100).reshape(-1,1)
def save():
    print('This is save')
    tf_x = tf.placeholder(tf.float32,x.shape)
    tf_y = tf.placeholder(tf.float32,y.shape)
    l = tf.layers.dense(tf_x,10,tf.nn.relu)
    o = tf.layers.dense(l,1)
```

```

loss = tf.losses.mean_squared_error(tf_y,o)
train_op = tf.train.GradientDescentOptimizer(learning_rate=0.5).minimize(
    loss)
sess = tf.Session()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
for step in range(100):
    sess.run(train_op,{tf_x:x,tf_y:y})
saver.save(sess,'params',write_meta_graph=False)
pred,l = sess.run([o,loss],{tf_x:x,tf_y:y})
plt.figure(1,figsize=(10,5))
plt.subplot(121)
plt.scatter(x,y)
plt.plot(x,pred,'r-',lw=5)
plt.text(-1,1.2,'save loss=%f'%l,fontdict={'size':15,'color':'red'})
def reload():
    print('This is reload')
    tf_x = tf.placeholder(tf.float32,x.shape)
    tf_y = tf.placeholder(tf.float32,y.shape)
    l_ = tf.layers.dense(tf_x,10,tf.nn.relu)
    o_ = tf.layers.dense(l_,1)
    loss_ = tf.losses.mean_squared_error(tf_y,o_)
    sess = tf.Session()
    saver = tf.train.Saver()
    saver.restore(sess,'params')
    pred,l = sess.run([o_,loss_],{tf_x:x,tf_y:y})
    plt.subplot(122)
    plt.scatter(x,y)
    plt.plot(x,pred,'r-',lw=5)
    plt.text(-1,1.2,'Reload Loss=%f'%l,fontdict={'size':15,'color':'red'})
    plt.show()
save()
tf.reset_default_graph()
reload()

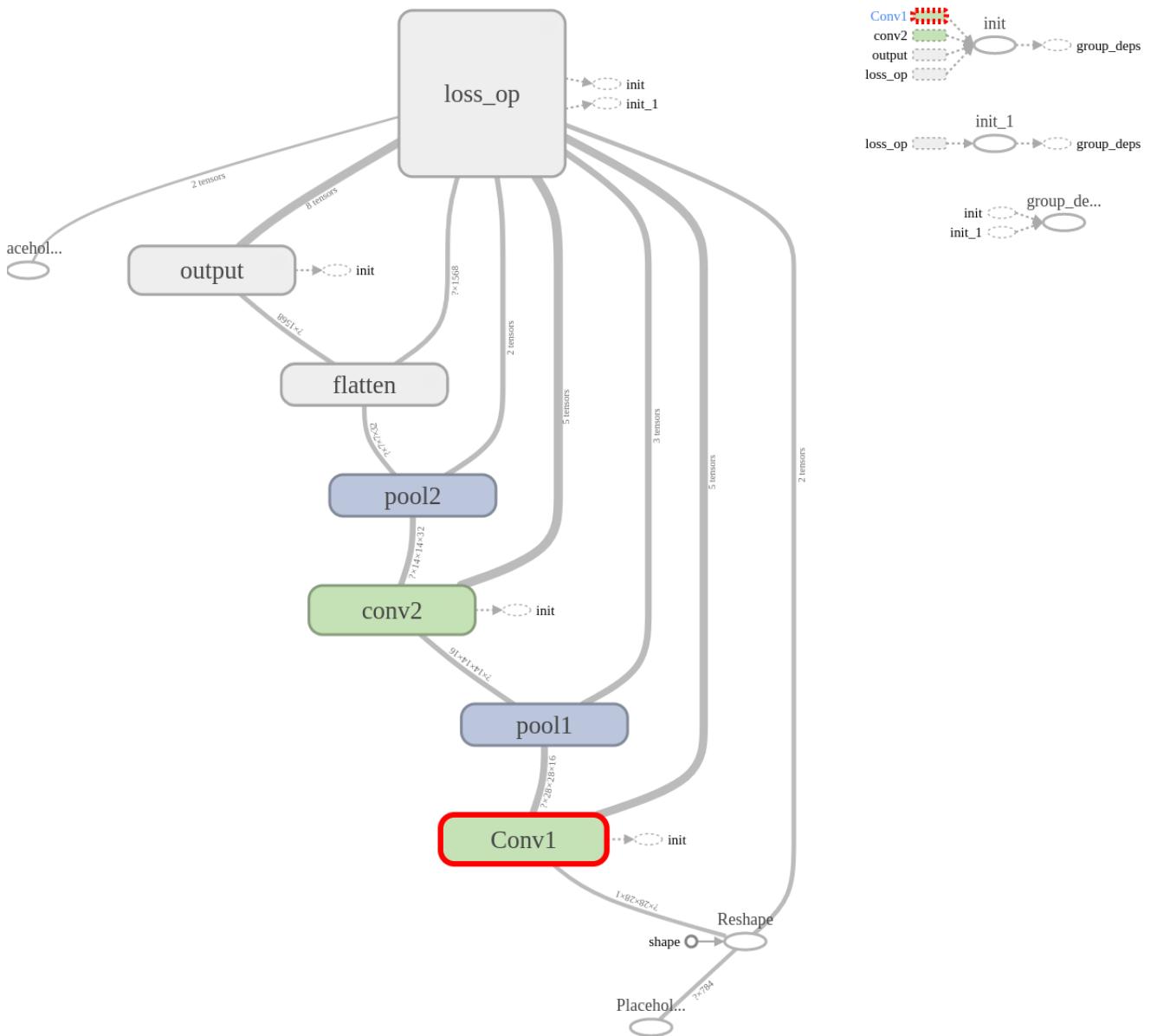
```

## 1.8 CNN 手写体数据识别

### 1.8.1 mnist 数据集

手写体数据训练集有 55000 张手写体数据图片。测试集有 10000 张图片。每张图片是大小为 32\*32 的灰度图片。卷积神经网络结构：

- 第一层卷积层：卷积核 16 个，卷积核大小为  $5 \times 5$ ,strides=1,padding 为 SAME，激活函数为 relu(输出大小为  $28 \times 28 \times 16$ )。
- 第一层池化层：池化层大小为 2,strides 为 2( $14 \times 14 \times 16$ )。第二层卷积层：卷积核 32，大小为  $5 \times 5$ ,strides=1,padding 为 SAME，激活函数为 relu。 $(14 \times 14 \times 32)$
- 第二层池化层：池化层大小为 2,strides 为 2( $7 \times 7 \times 32$ )。
- flatten:1568。



```

import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data

tf.set_random_seed(0)
np.random.seed(0)

```

```

BATCH_SIZE = 50
LR = 0.001
mnist = input_data.read_data_sets('/home/hpc/文档/mnist_tutorial/mnist', one_hot
                                  = True)
test_x = mnist.test.images[:2000]
test_y = mnist.test.labels[:2000]

tf_x = tf.placeholder(tf.float32, [None, 28*28])
images = tf.reshape(tf_x, [-1, 28, 28, 1])
tf_y = tf.placeholder(tf.int32, [None, 10])
with tf.variable_scope('Conv1'):
    conv1 = tf.layers.conv2d(
        inputs = images,
        filters = 16,
        kernel_size = 5,
        strides = 1,
        padding = 'same',
        activation = tf.nn.relu
    )
    tf.summary.histogram('conv1', conv1)
with tf.variable_scope('pool1'):
    pool1 = tf.layers.max_pooling2d(
        conv1,
        pool_size=2,
        strides =2
    )
    tf.summary.histogram('max_pool1', pool1)
with tf.variable_scope('conv2'):
    conv2 = tf.layers.conv2d(pool1, 32, 5, 1, 'SAME', activation=tf.nn.relu)
    tf.summary.histogram('conv2', conv2)
with tf.variable_scope('pool2'):
    pool2 = tf.layers.max_pooling2d(conv2, 2, 2)
    tf.summary.histogram('max_pool', pool2)
with tf.variable_scope('flatten'):
    flat = tf.reshape(pool2, [-1, 7*7*32])
with tf.variable_scope('output'):
    output = tf.layers.dense(flat, 10)
with tf.variable_scope('loss_op'):
    loss = tf.losses.softmax_cross_entropy(onehot_labels=tf_y, logits=output)
    train_op = tf.train.AdamOptimizer(LR).minimize(loss)
    accuracy = tf.metrics.accuracy(labels = tf.argmax(tf_y, axis=1), predictions =
                                    tf.argmax(output, axis=1),)[1]
    tf.summary.scalar('loss', loss)

```

```

tf.summary.scalar('accuracy', accuracy)
sess = tf.Session()
merge_op = tf.summary.merge_all()
init_op = tf.group(tf.global_variables_initializer(), tf.
                   local_variables_initializer())
sess.run(init_op)
writer = tf.summary.FileWriter('./log', sess.graph)
for step in range(600):
    b_x, b_y = mnist.train.next_batch(BATCH_SIZE)
    _, loss_, result = sess.run([train_op, loss, merge_op], {tf_x:b_x, tf_y:b_y})
    writer.add_summary(result, step)
    if step%50 == 0:
        accuracy_, flat_representation = sess.run([accuracy, flat], {tf_x:test_x,
                                                                     tf_y:test_y})
        print('Step:', step, '| train loss: %.4f | loss_ : %.2f | test accuracy: %.2f %'
              accuracy_)
test_output = sess.run(output, {tf_x:test_x[:10]})  

pred_y = np.argmax(test_output, 1)

```

## 1.9 RNN

### 1.9.1 向量字表示

#### Vector Representation of Words

通常图像或音频系统处理的是由图片中所有单个原始像素点强度值或者音频中功率谱密度的强度值，把它们编码成丰富、高维度的向量数据集。对于物体或语音识别这一类的任务，我们所需的全部信息已经都存储在原始数据中（显然人类本身就是依赖原始数据进行日常的物体或语音识别的）。然后，自然语言处理系统通常将词汇作为离散的单一符号，例如”cat”一词或可表示为 Id537，而”dog”一词或可表示为 Id143。这些符号编码毫无规律，无法提供不同词汇之间可能存在的关联信息。换句话说，在处理关于”dogs”一词的信息时，模型将无法利用已知的关于”cats”的信息（例如，它们都是动物，有四条腿，可作为宠物等等）。可见，将词汇表达为上述的独立离散符号将进一步导致数据稀疏，使我们在训练统计模型时不得不寻求更多的数据。而词汇的向量表示将克服上述的难题。向量空间模型(VSMs) 将词汇表达（嵌套）于一个连续的向量空间中，语义近似的词汇被映射为相邻的数据点。向量空间模型在自然语言处理领域中有着漫长且丰富的历史，不过几乎所有利用这一模型的方法都依赖于 分布式假设，其核心思想为出现于上下文情景中的词汇都有相类似的语义。采用这一假设的研究方法大致分为以下两类：基于计数的方法 (e.g. 潜在语义分析)，和 预测方法 (e.g. 神经概率化语言模型)。

其中它们的区别在如下论文中又详细阐述 Baroni :et al, 不过简而言之：基于计数的方法计算某词汇与其邻近词汇在一个大型语料库中共同出现的频率及其它统计量，然后将这些统计量映射到一个小型且稠密的向量中。预测方法则试图直接从某词汇的邻近词汇对其进行预测，在此过程中利用已经学习到的小型且稠密的嵌套向量。

Word2vec 是一种可以进行高效率词嵌套学习的预测模型。其两种变体分别为：连续词袋模型（CBOW）及 Skip-Gram 模型。从算法角度看，这两种方法非常相似，其区别为 CBOW 根据源词上下文词汇（'the cat sits on the'）来预测目标词汇（例如，'mat'），而 Skip-Gram 模型做法相反，它通过目标词汇来预测源词汇。Skip-Gram 模型采取 CBOW 的逆过程的动机在于：CBOW 算法对于很多分布式信息进行了平滑处理（例如将一整段上下文信息视为一个单一观察量）。很多情况下，对于小型的数据集，这一处理是有帮助的。相形之下，Skip-Gram 模型将每个“上下文-目标词汇”的组合视为一个新观察量，这种做法在大型数据集中会更为有效。本教程余下部分将着重讲解 Skip-Gram 模型。

## 处理噪声的对比训练

神经概率化语言模型通常使用极大似然法 (ML) 进行训练，其中通过 softmax function 来最大化当提供前一个单词 h (代表"history")，后一个单词的概率  $w_t$ (目标词概率)

$$P(w_t|h) = \text{softmax}(\text{score}(w_t, h)) = \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\}}$$

当  $\text{score}(w_t, h)$  计算了文字  $w_t$  和 上下文  $h$  的相容性（通常使用向量积）。我们使用对数似然函数来训练训练集的最大值，比如通过：

$$J_{ML} = \log P(w_t|h) = \text{score}(w_t, h) - \log(\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\})$$

这里提出了一个解决语言概率模型的合适的通用方法。然而这个方法实际执行起来开销非常大，因为我们需要去计算并正则化当前上下文环境  $h$  中所有其它 V 单词  $w'$  的概率得分，在每一步训练迭代中。

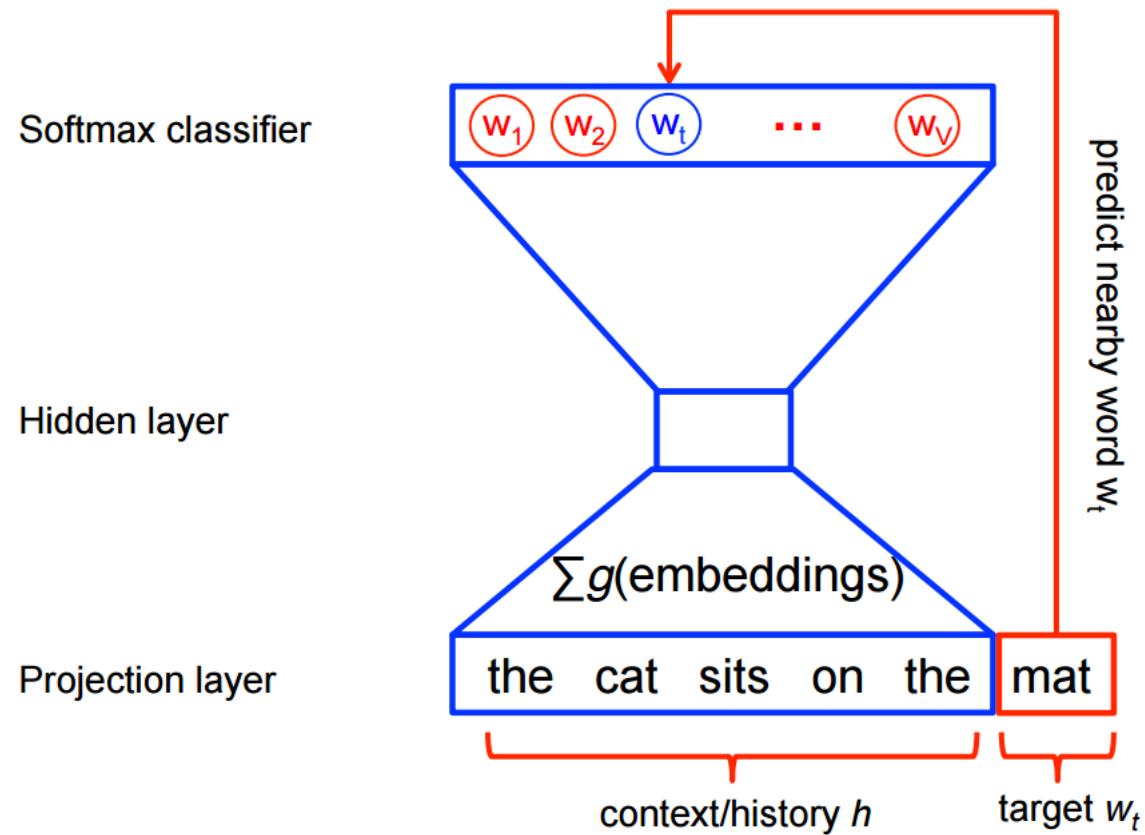


图 1.4: CBOW 方法

从另一个角度来说，当使用 word2vec 模型时，我们并不需要对概率模型中的所有特征进行学习。而 CBOW 模型和 Skip-Gram 模型为了避免这种情况发生，使用一个二分类器（逻辑回归）在同一个上下文环境里从  $k$  虚构的（噪声）单词  $\hat{w}$  区分真正的目标单词  $w_t$ ，下面详细参数 CBOW 模型，对于 Skip-Gram 模型只要简单的反向操作即可。

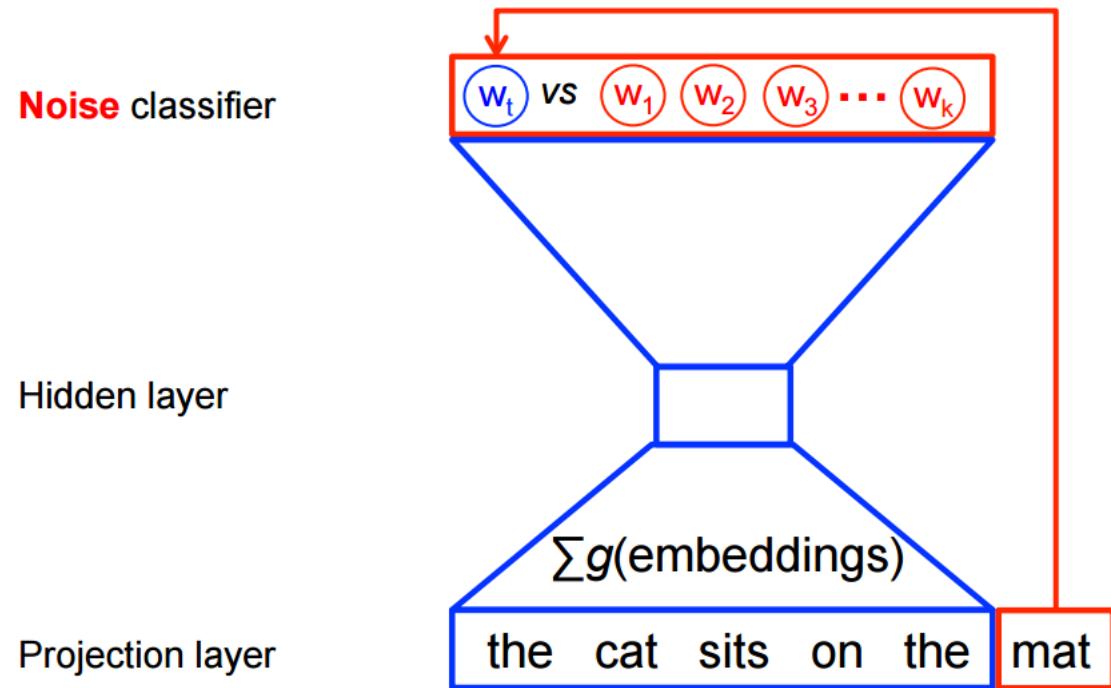


图 1.5: Skip-Gram

从数学的角度来说，我们的目标是对每个样本最大化:

$$J_{NEG} = \log Q_\theta(D = 1|w_t, h) + k \mathop{E}_{\hat{w} \sim P_{noise}} [\log Q_\theta(D = 0|\hat{w}, h)]$$

其中  $Q_\theta(D = 1|w, h)$  代表的是当前上下文  $h$ ，根据所学得嵌套向量  $\theta$  目标单词  $w$  使用二分类逻辑回归计算得出的概率。在实践中，我们通过在噪声分布中绘制比对文字来获得近似的期望值（通过计算蒙特卡洛平均值）。

当真实地目标单词被分配到较高的概率，同时噪声单词的概率很低时，目标函数也就达到最大值了。从技术层面来说，这种方法叫做**负抽样**，而且使用这个损失函数在数学层面上也有很好的解释：这个更新过程也近似于 softmax 函数的更新。这在计算上将会有很大的优势，因为当计算这个损失函数时，只是有我们挑选出来的  $k$  个 噪声单词，而没有使用整个语料库  $V$ 。这使得训练变得非常快。我们实际上使用了与**noise-contrastive estimation (NCE)**介绍的非常相似的方法，这在 TensorFlow 中已经封装了一个很便捷的函数 `tf.nn.nce_loss()`。

### Skip-gram 模型

下面来看一下这个数据集

the quick brown fox jumped over the lazy dog

我们首先对一些单词以及它们的上下文环境建立一个数据集。我们可以以任何合理的方式定义‘上下文’，而通常上这个方式是根据文字的句法语境的（使用语法原理的方式处理当前目标单词可以看一下这篇文献 [Levy et al.](#)，比如说把目标单词左边的内容当做一个‘上下文’，或者以目标单词右边的内容，等等。现在我们把目标单词的左右单词视作一个上下文，使用大小为 1 的窗口，这样就得到这样一个由(上下文, 目标单词)组成的数据集：

([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...

前文提到 Skip-Gram 模型是把目标单词和上下文颠倒过来，所以在这个问题中，举个例子，就是用'quick' 来预测'the' 和'brown'，用'brown' 预测'quick' 和'brown'。因此这个数据集就变成由(输入, 输出)组成的：

(quick, the), (quick, brown), (brown, quick), (brown, fox), ...

目标函数通常是对整个数据集建立的，但是本问题中要对每一个样本（或者是一个 batch\_size 很小的样本集，通常设置为  $16 \leq \text{batch\_size} \leq 512$ ）在同一时间执行特别的操作，称之为[随机梯度下降 \(SGD\)](#)。我们来看一下训练过程中每一步的执行。

假设用 t 表示上面这个例子中 quick 来预测 the 的训练的单个循环。用 num\_noise 定义从噪声分布中挑选出来的噪声（相反的）单词的个数，通常使用一元分布， $P(w)$ 。为了简单起见，我们就定 num\_noise=1，用 sheep 选作噪声词。接下来就可以计算每一对观察值和噪声值的损失函数了，每一个执行步骤就可表示为：

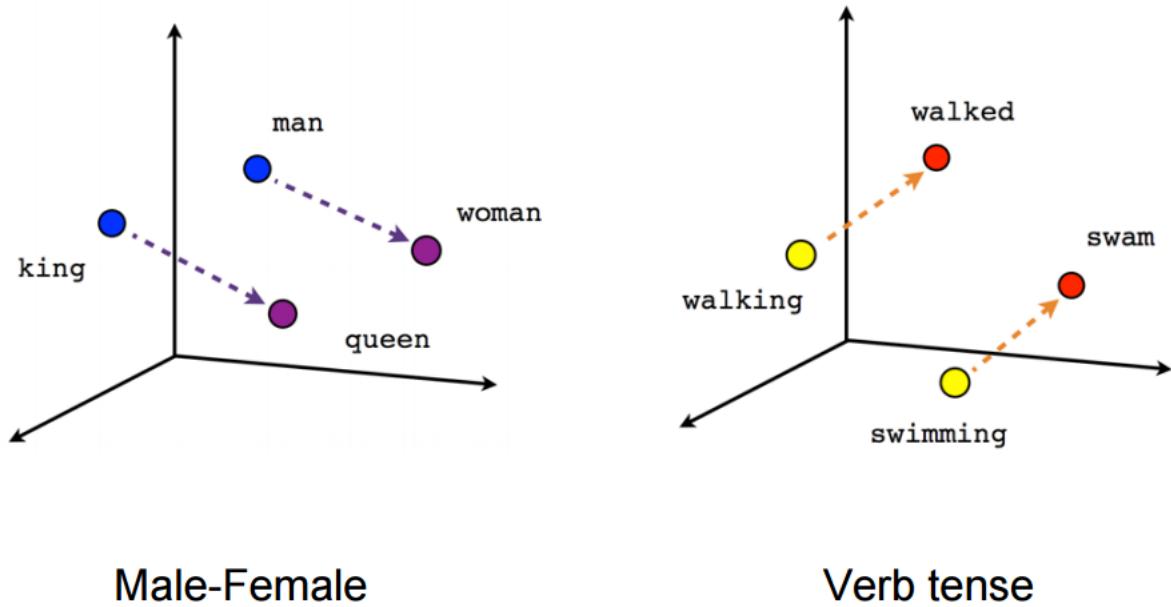
$$J_{NEG}^{(t)} = \log Q_\theta(D = 1 | the, quick) + \log(Q_\theta(D = 0 | sleep, quick))$$

整个计算过程的目标是通过更新嵌套参数  $\theta$  来逼近目标函数（这个例子中就是使目标函数最大化）。为此我们要计算损失函数中嵌套参数  $\theta$  的梯度，比如

$$\frac{\partial}{\partial} J_{NEG}$$

（幸好 TensorFlow 封装了工具函数可以简单调用！）。对于整个数据集，当梯度下降的过程中不断地更新参数，对应产生的效果就是不断地移动每个单词的嵌套向量，直到可以把真实单词和噪声单词很好得区分开。

我们可以把学习向量映射到 2 维中以便我们观察，其中用到的技术可以参考 [t-SNE 降维技术](#)。当我们用可视化的方式来观察这些向量，就可以很明显的获取单词之间语义信息的关系，这实际上是非常有用的。当我们第一次发现这样的诱导向量空间中，展示了一些特定的语义关系，这是非常有趣的，比如文字中 male-female, gender 甚至还有 country-capital 的关系，如下方的图所示（也可以参考 [Mikolov et al., 2013](#) 论文中的例子）。



这也解释了为什么这些向量在传统的 NLP 问题中可作为特性使用，比如用在对一个演讲章节打个标签，或者对一个专有名词的识别（看看如下这个例子 [Collobert et al.](#) 或者 [Turian et al.](#)）。

不过现在让我们用它们来画漂亮的图表吧！

这里谈得都是嵌套，那么先来定义一个嵌套参数矩阵。我们用唯一的随机值来初始化这个大矩阵。

```
embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

对噪声-比对的损失计算就使用一个逻辑回归模型。对此，我们需要对语料库中的每个单词定义一个权重值和偏差值。（也可称之为输出权重与之对应的输入嵌套值）。定义如下：

```
nce_weights = tf.Variable(
    tf.truncated_normal([vocabulary_size, embedding_size],
                        stddev=1.0 / math.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
```

我们有了这些参数之后，就可以定义 Skip-Gram 模型了。简单起见，假设我们已经把语料库中的文字整型化了，这样每个整型代表一个单词（细节请查看 `_basic.py`）。Skip-Gram 模型有两个输入。一个是一组用整型表示的上下文单词，另一个是目标单词。给这些输入建立占位符节点，之后就可以填入数据了。

```
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
```

然后我们需要对批数据中的单词建立嵌套向量，TensorFlow 提供了方便的工具函数。

```
embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

好了，现在我们有了每个单词的嵌套向量，接下来就是使用噪声-比对的训练方式来预测目标单词。

```
loss = tf.reduce_mean(
    tf.nn.nce_loss(nce_weights, nce_biases, embed, train_labels,
                   num_sampled, vocabulary_size))
```

我们对损失函数建立了图形节点，然后我们需要计算相应梯度和更新参数的节点，比如说在这里我们会使用随机梯度下降法，TensorFlow 也已经封装好了该过程。

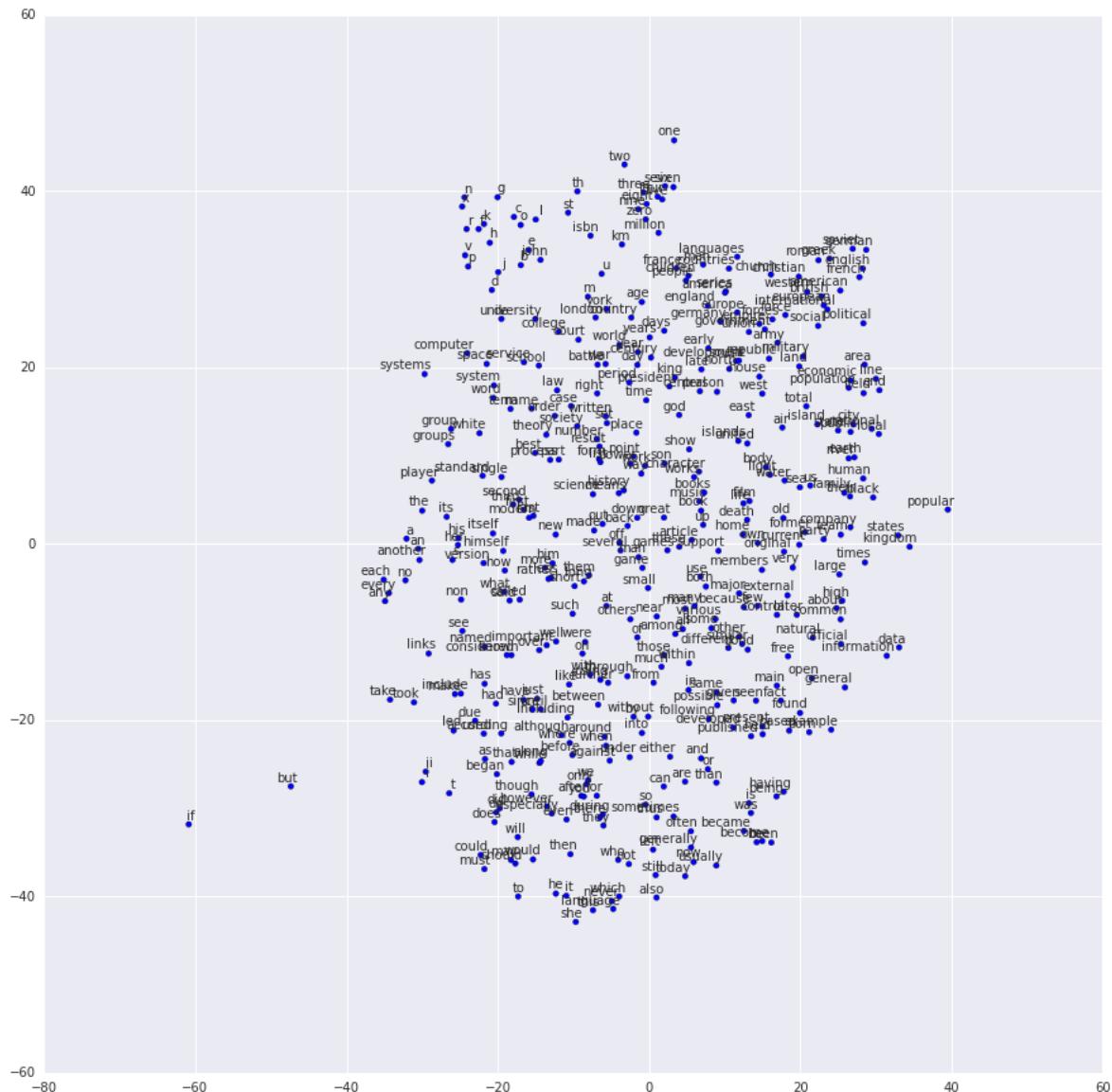
```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0).minimize(loss)
```

## 训练过程

训练的过程很简单，只要在循环中使用 `feed_dict` 不断给占位符填充数据，同时调用 `session.run` 即可。

```
for inputs, labels in generate_batch(...):
    feed_dict = {training_inputs: inputs, training_labels: labels}
    _, cur_loss = session.run([optimizer, loss], feed_dict=feed_dict)
```

嵌套学习结果可视化



Et voilà! 与预期的一样，相似的单词被聚类在一起。对 word2vec 模型更复杂的实现需要用到 TensorFlow 一些更高级的特性，具体是实现可以参考[word2vec.py](#)

嵌套学习的评估：类比推理

词嵌套在 NLP 的预测问题中是非常有用且使用广泛地。如果要检测一个模型是否是可以成熟地区分词性或者区分专有名词的模型，最简单的办法就是直接检验它的预测词性、

语义关系的能力，比如让它解决形如 king is to queen as father is to ? 这样的问题。这种方法叫做类比推理，可参考 Mikolov and colleagues，数据集下载地址为:[questions-words.txt](#)。To see how we do this evaluation 如何执行这样的评估，可以看 build\_eval\_graph() 和 eval() 这两个函数在下面源码中的使用 [word2vec.py](#)

超参数的选择对该问题解决的准确性有巨大的影响。想要模型具有很好的表现，需要有一个巨大的训练数据集，同时仔细调整参数的选择并且使用例如二次抽样的一些技巧。不过这些问题已经超出了本教程的范围。

## 优化实现

以上简单的例子展示了 TensorFlow 的灵活性。比如说，我们可以很轻松得用现成的 tf.nn.sampled\_softmax\_loss() 来代替 tf.nn.nce\_loss() 构成目标函数。如果你对损失函数想做新的尝试，你可以用 TensorFlow 手动编写新的目标函数的表达式，然后用控制器执行计算。这种灵活性的价值体现在，当我们探索一个机器学习模型时，我们可以很快地遍历这些尝试，从中选出最优。

一旦你有了一个满意的模型结构，或许它就可以使实现运行地更高效（在短时间内覆盖更多的数据）。比如说，在本教程中使用的简单代码，实际运行速度都不错，因为我们使用 Python 来读取和填装数据，而这些在 TensorFlow 后台只需执行非常少的工作。如果你发现你的模型在输入数据时存在严重的瓶颈，你可以根据自己的实际问题自行实现一个数据阅读器，参考 新的数据格式。对于 Skip-Gram 模型，我们已经完成了如下这个例子 [word2vec.py](#)。

如果 I/O 问题对你的模型已经不再是个问题，并且想进一步地优化性能，或许你可以自行编写 TensorFlow 操作单元，详见 添加一个新的操作。相应的，我们也提供了 Skip-Gram 模型的例子 [optimized.py](#)。请自行调节以上几个过程的标准，使模型在每个运行阶段有更好的性能。

### 1.9.2 RNN

此教程将展示如何在高难度的语言模型中训练循环神经网络。该问题的目标是获得一个能确定语句概率的概率模型。为了做到这一点，通过之前已经给出的词语来预测后面的词语。我们将使用 PTB(Penn Tree Bank) 数据集，这是一种常用来衡量模型的基准，同时它比较小而且训练起来相对快速。

语言模型是很多有趣难题的关键所在，比如语音识别，机器翻译，图像字幕等。它很有意思—可以参看 [here](#)。

本教程的目的是重现 [Zaremba et al., 2014](#) 的成果，他们在 PTB 数据集上得到了很棒的结果。

## 下载及准备数据

本教程需要的数据在 data/ 路径下，来源于 Tomas Mikolov 网站上的[PTB 数据集](#)

该数据集已经预先处理过并且包含了全部的 10000 个不同的词语，其中包括语句结束标记符，以及标记稀有词语的特殊符号 (<unk>)。我们在 reader.py 中转换所有的词语，让他们各自有唯一的整型标识符，便于神经网络处理。

## LSTM

模型的核心由一个 LSTM 单元组成，其可以在某时刻处理一个词语，以及计算语句可能的延续性的概率。网络的存储状态由一个零矢量初始化并在读取每一个词语后更新。而且，由于计算上的原因，我们将以 batch\_size 为最小批量来处理数据。

基础的伪代码就像下面这样：

```
lstm = rnn_cell.BasicLSTMCell(lstm_size)
state = tf.zeros([batch_size, lstm.state_size])

loss = 0.0
for current_batch_of_words in words_in_dataset:
    output, state = lstm(current_batch_of_words, state)

    logits = tf.matmul(output, softmax_w) + softmax_b
    probabilities = tf.nn.softmax(logits)
    loss += loss_function(probabilities, target_words)
```

## 截断反向传播

为使学习过程易于处理，通常的做法是将反向传播的梯度在（按时间）展开的步骤上照一个固定长度 (num\_steps) 截断。通过在一次迭代中的每个时刻上提供长度为 num\_steps 的输入和每次迭代完成之后反向传导，这会很容易实现。

一个简化版的用于计算图创建的截断反向传播代码：

```
words = tf.placeholder(tf.int32, [batch_size, num_steps])

lstm = rnn_cell.BasicLSTMCell(lstm_size)
initial_state = state = tf.zeros([batch_size, lstm.state_size])

for i in range(len(num_steps)):
    output, state = lstm(words[:, i], state)

    # ...
```

```
final_state = state
```

下面展现如何实现迭代整个数据集：

```
numpy_state = initial_state.eval()
total_loss = 0.0
for current_batch_of_words in words_in_dataset:
    numpy_state, current_loss = session.run([final_state, loss],
        feed_dict={initial_state: numpy_state, words: current_batch_of_words})
    total_loss += current_loss
```

## 输入

在输入 LSTM 前，词语 ID 被嵌入到了一个密集的表示中（查看 矢量表示教程）。这种方式允许模型高效地表示词语，也便于写代码：

```
# embedding_matrix 张量的形状是: [vocabulary_size, embedding_size]
word_embeddings = tf.nn.embedding_lookup(embedding_matrix, word_ids)
```

嵌入的矩阵会被随机地初始化，模型会学会通过数据分辨不同词语的意思。

## 损失函数

我们想使目标词语的平均负对数概率最小  $loss = -\frac{1}{N} \sum_{i=1}^N \ln p_{target_i}$  实现起来并非很难，而且函数 `sequence_loss_by_example` 已经有了，可以直接使用。

论文中的典型衡量标准是每个词语的平均困惑度（perplexity），计算式为

$$e^{-\frac{1}{N} \sum_{i=1}^N \ln p_{target_i}} = e^{loss}$$

同时我们会观察训练过程中的困惑度值（perplexity）

## 多个 LSTM 层堆叠

要想给模型更强的表达能力，可以添加多层 LSTM 来处理数据。第一层的输出作为第二层的输入，以此类推。

类 `MultiRNNCell` 可以无缝的将其实现：

```
lstm = rnn_cell.BasicLSTMCell(lstm_size)
stacked_lstm = rnn_cell.MultiRNNCell([lstm] * number_of_layers)

initial_state = state = stacked_lstm.zero_state(batch_size, tf.float32)
for i in range(len(num_steps)):
    # 每次处理一批词语后更新状态值。
    output, state = stacked_lstm(words[:, i], state)
```

```
# 其余的代码.  
# ...  
  
final_state = state
```

## 编译并运行代码

首先需要构建库，在 CPU 上编译：

```
bazel build -c opt tensorflow/models/rnn/ptb:ptb_word_lm
```

如果你有一个强大的 GPU，可以运行

```
bazel build -c opt --config=cuda tensorflow/models/rnn/ptb:ptb_word_lm
```

运行模型：

```
bazel-bin/tensorflow/models/rnn/ptb/ptb_word_lm \  
--data_path=/tmp/simple-examples/data/ --alsologtosterr --model small
```

教程代码中有 3 个支持的模型配置参数：“small”，“medium” 和“large”。它们指的是 LSTM 的大小，以及用于训练的超参数集。

模型越大，得到的结果应该更好。在测试集中 small 模型应该可以达到低于 120 的困惑度 (perplexity)，large 模型则是低于 80，但它可能花费数小时来训练。



## Chapter 2

# Tensorflow 进阶

## 2.1 模型存储和加载

- 生成 checkpoint 文件, 扩展名一般为.ckpt, 通过在 tf.train.Saver 对象上调用 Saver.saver() 生成。它包含权重和其他程序中定义的变量, 不包含 图的结构。如果需要在另一个程序中使用, 需要重建图形结构, 并告诉 Tensorflow 如何处理这些权重。
- 生成 (graph proto file), 这是一个二进制文件, 扩展名一般是.pb, 用 tf.train.write\_graph() 保存每, 只包含图形结构, 不包含权重, 然后使用 tf.import\_graph\_def() 加载 图形。

## 2.2 用 GPU

在 Tensorflow 中 CPU,GPU 用字符串表示

- "cpu:0": 机器上的 CPU
- "gpu:0": 机器上的 GPU
- "gpu:1": 机器上的第二块 GPU

如果 TensorFLow 操作有 GPU 和 CPU 实现，GPU 将被优先指定，例如 matmul 有 CPU 和 GPU 内核，在系统上有 cpu:0 和 gpu:0,gpu:0 将优先运行 matmul。布置采集设备

找到你的操作和 tensor 上的设备，创建一个会话 log\_device\_placement 配置设置为 True

```
import tensorflow as tf
a = tf.reshape(tf.linspace(-1., 1., 12), (3, 4))
b = tf.reshape(tf.sin(a), (4, 3))
c = tf.matmul(a, b)
with tf.Session() as sess:
    print(sess.run(c))
```

输出参数:

```
[[ 0.87280041  0.44710392  0.00666773]
 [ 0.43973413  0.44710392  0.4397341 ]
 [ 0.00666779  0.44710392  0.87280059]]
```

### 2.2.1 手工配置设备

如果你想将你的操作运行在指定的设备中而不由 tensorflow 是自动为你选择，你可以用 tf.device 创建一个设备，左右的操作将在同一个设备上指定。

```
import tensorflow as tf
with tf.device('/cpu:0'):
    a = tf.constant([1., 2., 3., 4., 5., 6.], shape=(2, 3), name='a')
    b = tf.reshape(a, shape=(3, 2))
    c = tf.matmul(a, b)
    with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
        print(sess.run(c))
```

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: TITAN Xp, pci bus id: 0000:06:00.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: TITAN Xp, pci bus id: 0000:05:00.0
Reshape: (Reshape): /job:localhost/replica:0/task:0/cpu:0
MatMul: (MatMul): /job:localhost/replica:0/task:0/cpu:0
Reshape/shape: (Const): /job:localhost/replica:0/task:0/cpu:0
a: (Const): /job:localhost/replica:0/task:0/cpu:0
[[ 22.  28.]
 [ 49.  64.]]
```

正如你看到的 a,b 被复制到 cpu:0, 因为设备没有明确指定, Tensorflow 将选择操作和可用的设备 (gpu:0)

### 2.2.2 允许 GPU 的内存增长

默认情况下 Tensorflow 将映射所有的 CPUs 的显存到进程上, 用相对精确的 GPU 内存资源减少内存的碎片化会更高效。通常有些程序希望分贝可用内存的一部分, 或者增加内存的需要两。在会话中 tensorflow 提供了两个参数 控制它。第一个参数是 allow\_growth 选项, 根据运行情况分配 GPU 内存: 它开始分配很少的内存, 当 Session 开始运行 需要更多 GPU 内存是, 我们同感 Tensorflow 程序扩展 GPU 的内存区域。注意我们不释放内存, 因此这可能导致更多的内存碎片。为了开启这个选项, 可以通过下面的设置

```
config = tf.ConfigProto()
config.gpu_option.allow_growth = True
sess = tf.Session(config=config, ...)
```

第二种方法是 per\_process\_gpu\_memory\_fraction 选项, 决定 GPU 总体内存中多少应给被分配, 例如你可以告诉 Tensorflow 分配 40% 的 GPU 总体内存。

```
config = tf.ConfigProto()
config.gpu_option.per_process_gpu_memory_fraction = 0.4
sess = tf.Session(config = config)
```

如果你想限制 Tensorflow 程序的 GPU 使用量, 这个参数是很有用的。

在多 GPU 系统是使用 GPU

如果你的系统上有超过一个 GPU, 你的 GPU 的抵消的 ID 将被默认选中, 如果你想运行在不同的 GPU 上, 你需要指定 你想要执行运算的 GPU

```
import tensorflow as tf
with tf.device('/gpu2:0'):
    a = tf.constant([1., 2., 3., 4., 5., 6.], shape=(2, 3), name='a')
    b = tf.reshape(a, shape=(3, 2))
    c = tf.matmul(a, b)
    with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
        print(sess.run(c))
```

如果你指定的设备不存在, 你将个到一个 InvalidArgumentError:

```
InvalidArgumentError (see above for traceback): Cannot assign a device for operation 'Reshape': Operation was explicitly assigned to /device:GPU:2 but available devices are [ /job:localhost/replica:0/task:0/cpu:0, /job:localhost/replica:0/task:0/gpu:0, /job:localhost/replica:0/task:0/gpu:1 ]. Make sure the device specification refers to a valid device.
[[Node: Reshape = Reshape[T=DT_FLOAT, Tshape=DT_INT32, _device="/device:GPU:2"](a, Reshape/shape)]]
```

如果你想 Tensorflow 在万一指定的设备不存在时自动选择一个存在的设备，你可以在创建会话时配置中设置 allow\_soft\_placement 为 True

```
with tf.device('/gpu:2'):
    a = tf.constant([1., 2., 3., 4., 5., 6.], shape=[3, 2], name='a')
    b = tf.constant([1., 2., 3., 4., 5., 6.], shape=[2, 3], name='b')
    c = tf.matmul(a, b)
with tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                      log_device_placement=True)) as sess:
    print(sess.run(c))
```

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: TITAN Xp, pci bus id: 0000:06:00.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: TITAN Xp, pci bus id: 0000:05:00.0
Reshape: (Reshape): /job:localhost/replica:0/task:0/gpu:0
MatMul: (MatMul): /job:localhost/replica:0/task:0/gpu:0
Reshape/shape: (Const): /job:localhost/replica:0/task:0/gpu:0
a: (Const): /job:localhost/replica:0/task:0/gpu:0
[[ 22.  28.]
 [ 49.  64.]]
```

## 用多 GPU

如果你想在多张 GPU 上运行 Tensorflow，你可以在 multi-tower fashion 上构造你的模型，每个 tower 被指定到不同的 GPU 上。例如：

```
c = []
for d in ['/gpu:0', '/gpu:1']:
    with tf.device(d):
        a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
        b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
        c.append(tf.matmul(a, b))
    with tf.device('/cpu:0'):
        sum = tf.add_n(c)
    # Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                         log_device_placement=True))
    # Runs the op.
print(sess.run(sum))
sess.close()
```

# Chapter 3

## Performance

这个导航包含一个优化你的 TensorFlow 代码的集合。对于 Tensorflow 用户来说这是最好的应用，正如在这个文档中最好的时间，高性能模式为在不同的硬件上创建模型文档链接到示例代码。

### 3.1 最好的实践

尽管优化实现不同类型的模型可能不同，下面是通过 tensorflow 实现性能的几个最好的方式，尽管这些暗示在基于图像的模型，我们将增加一些技巧到所有类型的模型。下面列出了最好实践的关键：

- 从原来码编译安装
- 利用队列读取数据
- 在 CPU 上预处理
- 用 NCHW 图像格式
- 在 GPU 上放共享参数
- 用融合的批处理规范

下面章节时处理的详细信息。

### 3.2 从源代码创建安装

为了安装最优化的 TensorFlow 版本，通过源代码编译安装 Tensorflow。从原来码编译优化目标硬件确保最新的 CUDA 平台和 CuDNN 库被用高性能安装。

对于多数稳定的实验，从最新版的[latest release](#)分支编译。为了得到最新性能改变接受一些稳定性风险，从[master](#)编译。

如果你需要在不同的目标硬件平台上编译 TensorFlow，交叉编译最优化目标平台。下面的目录是一个例子高数 bazel 为指定平台编译

```
# This command optimizes for Intel's Broadwell processor
bazel build -c opt --copt=-march="broadwell" --config=cuda //tensorflow/tools/
                                         pip_package:build_pip_package
```

### 环境，构建，安装技巧

- 编译最高级别的[GPU 支持](#)，e.g. P100: 6.0, Titan X (pascal): 6.2, Titan X (maxwell): 5.2, and K80: 3.7.
- 安装最新版的 CUDA 平台和 cuDNN 库
- 确保你的 gcc 版本支持对目标 cpu 所有的优化，推荐最小的 gcc 版本为 4.8.3
- TensorFlow 在启动时检查是否已经在 cpu 上编译优化过，如果优化不被包含，TensorFlow 将 chxuan 警告，e.g. AVX, AVX2 和 FMA 设备不被包含。

#### 3.2.1 利用队列读取数据

在利用 GPUs 时性能很差或者没有设置高效的 pipeline 导致缺乏数据，确保设置输入 pipeline 高效利用队列和流数据，一种识别 GPU 处于饥饿状态的方法时生成和查询时间线。一个相信的时间线指南不存在，但是一个快速生成时间线的例子在[XLA JIT](#)部分存在，另一个检查是否 GPU 被充分使用时运行 nvidia-smi 查看，如果 GPU 利用没有达到 100% 这样 GPU 没有足够的快的得到数据。

除非指定一个特殊的情形或者示例代码，没有从 Python 变量给予数据到会话，e.g.

```
# Using feed_dict often results in suboptimal performance when using large
                           inputs.
sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

#### 3.2.2 在 CPU 上的预处理

将预处理操作放在 CPU 上可能对性能提升很重要，当预处理发生在 GPU，数据流使从 CPU->GPU(预处理)->CPU->GPU(训练)。这数据被限制在 CPU 和 GPU 之间，当预处理被放在 CPU 上，数据流是 CPU(预处理)->GPU(训练)。另一个好处是在 CPU 上预处理释放 GPU 时间让其集中训练。

将预处理放在 CPU 上可能导致对 sample/sec 处理速度 6 倍以上的处理性能增加，将导致训练时间缩短为原来的  $\frac{1}{6}$ ，确保预处理在 CPU 上，按照如下操作：

```
with tf.device('/cpu:0'):
    # function to get and process images or data.
    distorted_inputs = load_and_distort_images()
```

### 3.2.3 用大文件

在一些情形下，CPU 和 GPU 可能同感 I/O 操作获取数据时对数据处于饥饿状态。如果你正用一些小文件形成输入数据集，你也许被你的文件系统限制了速度。如果你在 SSD 上而不是 HDD 上存储你的输入数据你的训练循环运行更快。如果是这样你应该通过创建一些大的 TFRecord 文件预处理你的输入数据。

### 3.2.4 用 NCHW 图像数据格式

图像数据格式涉及到图像的批量表示。TensorFlow 支持 NHWC(TensorFlow 默认) 和 NCHW(cuDNN 默认)，N 时图像的批数，H 时图像垂直方向的像素数量，W 是水平方向的像素，C 时图像的通道数，尽管 cuDNN 能处理上面两种格式，但是它处理默认格式更快。最好的实现是用 NCHW 和 NHWC 构建模型正如通常在 GPU 上用 NCHW 训练然后在 CPU 上用 NHWC 推断。

TensorFlow 用这两个格式是的一个简单的历史因为它在 CPUs 上运行快点，然后 TensorFlow 团队发现当 NVIDIA cuDNN 库时 NCHW 运行更好。当即用户推荐在他们的模型中支持两种格式，在很长一段时期，我们计划重写图转化两种格式。

### 3.2.5 用融批规范

当用批规范 `tf.contrib.layers.batch_norm` 设置属性 `fused=True`:

```
bn = tf.contrib.layers.batch_norm(
    input_layer, fused=True, data_format='NCHW',
    scope=scope, **kwargs)
```

在没有融合批规范计算几个单独的操作。融合批规范结合单个操作进入内核，运行更快。



## Chapter 4

# 常用的 python 模块

### 4.1 Argparse

argparse 模块是一个用户友好的命令行接口，当用户每有给定可用的参数时，argparser 能自动生成帮助和使用信息。

```
import argparse
parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+', help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const', const=sum,
                    default=max, help='sum the integers( default : find the max )')
args = parser.parse_args()
print(args.accumulate(args.integers))
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ vim code/demo1.py
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo1.py
usage: demo1.py [-h] [--sum] N [N ...]
demo1.py: error: the following arguments are required: N
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo1.py 1 2 3 4
4
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo1.py --sum 1 2 3 4
10
```

代码能根据传入的参数选择相应的函数计算。

- 创建一个 parser
- 增加 arguments
- 解析参数

### 4.1.1 ArgumentParser 对象

```
class argparse.ArgumentParser(prog=None, usage=None, description=None, epilog=None,
parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None,
argument_default=None, conflict_handler='error', add_help=True, allow_abbrev=True)
```

- prog: 程序的名字 (默认为 sys.argv[0])
- usage: 描述程序用法的字符串。 (默认通过 arguments 增加到 parser)
- description: argument 帮助前的文本展示。 (默认为:None)
- epilog: argument 帮助之后的文本展示。 (默认为:None)
- parents: 应该被包含的列表对象。
- formatter\_class: 自定义输出帮助的类。
- prefix\_chars: 参数前面的字符。 (默认为'-')
- fromfile\_prefix\_chars: 应该被读的文件的字符串。
- argument\_default: 参数的全局值。 (default:None)
- conflict\_handler: 解决冲突选项的策略。 (通常不是必需的)
- add\_help: 增加-h/-help 选项到 parser。 (默认为 True)
- allow\_abbrev: 如果缩略不冲突, 可以允许长的选项被缩略。 (默认为 True)

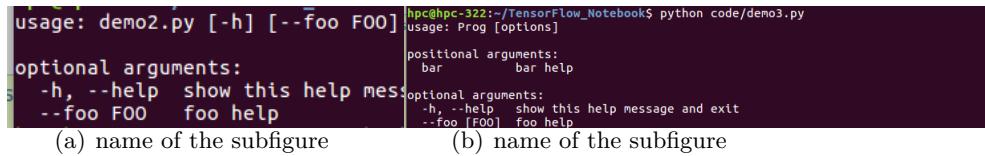
### 4.1.2 prog

默认情况下 ArgumentParser 对象用 sys.argv[0] 决定如何显示程序的名字。

```
#filename : arg1.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

默认情况下 ArgumentParser 从包含用法信息的参数计算 usage message。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```



大多数的 ArgumentParser 构造体用 `description=` 关键字，这个参数给出一个简单的程序说明其如何工作的。在帮助信息中表述在命令行和帮助信息之间。

```
import argparse
parser = argparse.ArgumentParser(description='A foo that bars')
parser.print_help()
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo4.py
usage: demo4.py [-h]

A foo that bars

optional arguments:
  -h, --help  show this help message and exit
```

一些程序喜欢在参数表述后添加一些额外的信息说明，这些说明可以通过 ArgumentParser 中的 `epilog=` 参数指定。

```
import argparse
parser = argparse.ArgumentParser(description='A foo that bars',
epilog="And that's how you'd foo a bar")
parser.print_help()
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo5.py
usage: demo5.py [-h]

A foo that bars

optional arguments:
  -h, --help  show this help message and exit

And that's how you'd foo a bar
```

有时候一些 parser 共享一些参数，相比于重复定义这些参数，一个单个的 parser 通过传递 `parents` 给 ArgumentParser。`parents=` 参数得到一个 ArgumentParser 对象的列表对象，从中收集所有的位置和选项行为

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

大多数的 parent parser 指定 `add_help=False`，因此 ArgumentParser 将看到两个帮助选项（一个在 parent 一个在 child）同时报错。你必须在通过 `parsers=` 传递前必须完全初始化

parser, 如果你在 child parser 改变 parent parsers, 改变将不被反映到 child.formatter\_class ArgumentParser.durian 允许指定可用的格式化类自定义格式, 当前有 4 个类:

- argparse.RawDescriptionHelpFormatter
- argparse.RawTextHelpFormatter
- argparse.ArgumentDefaultHelpFormatter
- argparse.MetavarTypeHelpFormatter

RawDescriptionHelpFormatter 和 RawTextHelpFormatter 在如何显示说明上给与更多控制, 默认 ArgumentParser 对 description 和 epilog 在命令终端一行显示。

```
import argparse
parser = argparse.ArgumentParser(prog='PROG', description='''
    this
    description was indented weird
but that is okey ''',
epilog=''''
likewise for this epilog whose whitespace will be
cleaned up and whose words will be wrapped
across a couple lines ''')
parser.print_help()
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo/.py
usage: PROG [-h]

this description was indented weird but that is okey
optional arguments:
  -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

传递 RawDescriptionHelpFormatter 作为 formatter\_class= 让 description 和 epilog 正确显示。RawTextHelpFormatter 主要维持素有的帮助文本, 值描述的信息。

ArgumentDefaultHelpFormatter: 自动增加关于值的默认信息。

```
import argparse
parser = argparse.ArgumentParser(prog='Prog',
formatter_class = argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--foo', type=int, default=42, help='FOO')
parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR! ')
parser.print_help()
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo9.py
usage: Prog [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
  bar          BAR! (default: [1, 2, 3])

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO    FOO (default: 42)
```

MatavarTypeHelpFormatter 用 type 显示参数显示值的名字。

```
import argparse
parser = argparse.ArgumentParser(prog='Prog',
                                formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--foo', type=int, default=42, help='FOO')
parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
parser.print_help()
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo10.py
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help    show this help message and exit
  --foo int
```

prefix\_chars, 大多数命令行参数选项用-, 比如-f/-foo。parsers 需要支持不同的或者说另外的前缀, 像 +f 或者/fo 可以设置 prefix\_chars= 参数指定。prefix\_chars 默认默认为-, 用非-字符能禁用-f/-foo 这种类型的选项。

```
import argparse
parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+-')
parser.add_argument('+f')
parser.add_argument('++bar')
parser.parse_args('+f X ++bar Y'.split())
```

fromfile\_prefix\_chars, 有时我们处理一个长的参数列表, 将参数保存在文件中比直接在命令行中更容易理解, 如果 fromfile\_prefix\_chars= 参数给 ArgumentParse 结构体, 指定的参数将被作为文件, 被下面的参数取代。例如

```
import argparse
with open('args.txt', 'w') as fp:
    fp.write('-f\nbar')
parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
parser.add_argument('-f')
parser.parse_args(['-f', 'foo', '@args.txt'])
```

默认从一个文件读取参数, 上面的表达式 ['-f', 'foo', '@args.txt'] 等于表达式 ['-f', 'foo', '-f', 'bar'], fromfile\_prefix\_chars 参数默认为 None, 意味着参数不被当作文件。argument\_default

通常通过传递 add\_argument 或者通过调用 set\_defaults() 方法指定名字和值对，然而有时候通过给参数指定一个简单的 parser-wide 是有用的，这可以通过传递 argument\_default=关键字到 ArgumentParser，例如调用其全局抑制属性在 parse\_args() 调用，我们用 argument\_default=SUPPRESS：

```
import argparse
parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
parser.add_argument('--foo')
parser.add_argument('bar', nargs='?')
parser.parse_args(['--foo', '1', 'BAR'])
print(parser.parse_args([]))
```

#### allow\_abbrev

通常我们传递一个参数 liebhiao 给 ArgumentParser 的方法 parse\_args()，如果选项参数太长的话。特征展示可能通过设置 allow\_abbrev 设置为 False 被禁用。

```
import argparse
parser = argparse.ArgumentParser(prog='Prog', allow_abbrev=False)
parser.add_argument('--foobar', action='store_true')
parser.add_argument('--fooley', action='store_true')
parser.parse_args(['--foon'])
```

```
hpc@hpc-322:~/TensorFlow_Notebook/code$ python demo14.py
usage: Prog [-h] [--foobar] [--fooley]
Prog: error: unrecognized arguments: --foon
```

#### conflict\_handler

ArgumentParser 对象不允许相同的选项字符串有两个行为，默认情况下当已经一偶选项字符串使用时尝试穿件一个新的参数 ArgumentParser 对象将报出异常。

```
In [1]: import argparse
In [2]: parser = argparse.ArgumentParser(prog='PROG')
In [3]: parser.add_argument('-f', '--foo', help='old foo help')
Out[3]: _StoreAction(option_strings=['-f', '--foo'], dest='foo', nargs=None, const=None, default=None, type=None, choices=None, help='old foo help', metavar=None)
In [4]: parser.add_argument('--foo', help='new foo help')
      File "<ipython-input-4-b0dbd0131b6e>", line 1
          parser.add_argument('--foo', help='new foo help')
                                         ^
SyntaxError: invalid syntax
```

有时候覆盖掉就得参数时有用的，为了得到参数的行为值'resolvce' 可能被应用在 conflict\_handler=参数。

```
import argparse
parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
parser.add_argument('-f', '--foo', help='old foo help')
parser.add_argument('--foo', help='new foo help')
```

```
parser.print_help()
```

```
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  -f FOO      old foo help
  --foo FOO   new foo help
```

如果所有的选项字符串被覆盖，ArgumentParser 对象仅仅移除一个行为，因此上面的例子中，就得行为-f/-foo 行为保留-f 行为，因为仅仅-foo 选项字符串被覆盖。add\_help 默认情况下 ArgumentParser durian 增加帮助信息到显示的消息中，例如：

```
import argparse
parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+', help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const', const=sum,
                    default=max, help='sum the integers (default: find the max)')
args = parser.parse_args()
print(args.accumulate(args.integers))
```

```
usage: demo1.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N           an integer for the accumulator

optional arguments:
  -h, --help  show this help message and exit
  --sum       sum the integers(default:find the max)
```

```
import argparse
parser = argparse.ArgumentParser(description='Process some integers.', add_help=False)
parser.add_argument('integers', metavar='N', type=int, nargs='+', help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const', const=sum,
                    default=max, help='sum the integers (default: find the max)')
args = parser.parse_args()
print(args.accumulate(args.integers))
```

```
usage: demo1.py [--sum] N [N ...]
demo1.py: error: the following arguments are required: N
```

### 4.1.3 add\_argument() 方法

ArgumentParser.add\_argument(name or flags..., action][, nargs][, const][, default][, type][, choices][, required][, help][, metavar][, dest]) 定一个一个命令行参数应该被如何解

析，每一个参数自己有自己的详细描述，如下：

- name or flags: 名字或者选项字符串， foo 或者 (-f,--foo)。
- action: 参数出现在命令行后采取的基本的行为。
- nargs: 命令行参数应该被使用的参数的数量。
- const:action 和 nargs 选项要求的常数值。
- default: 缺乏参数的默认值。
- type: 传递参数读取的数据类型。
- choices: 参数的允许值的容器。
- required: 是否命令行选项被忽略。
- help: 简易的参数说明。
- metavar: 在 usage 消息的名字。
- dest: 增加到 parse\_args() 返回对象的属性的名字。

name 或者 flags

当 parse\_args() 被调用的时候。选项参数通过-前缀识别。

```
import argparse
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-f', '--foo')
parser.add_argument('bar')
print(parser.parse_args(['BAR']))
print(parser.parse_args(['BAR', '--foo', 'FOO']))
```

```
hpc@hpc-322:~/TensorFlow_Notebook/code$ python demo16.py
Namespace(bar='BAR', foo=None)
Namespace(bar='BAR', foo='FOO')
```

action

- 'store': 仅仅保存参数的值，例如

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo')
parser.parse_args('--foo 1'.split())
```

输出 Namespace(foo='1')

- 'store\_true': 存储 const 参数指定的值，'store\_const' 行为通常用于指定一些 flag。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', action='store_const', const=42)
parser.add_argument('--foo')
```

输出:Namespace(foo=42)

- 'store\_true' 和 'store\_false' 指定 'store\_const'。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', action='store_true')
parser.add_argument('--bar', action='store_false')
parser.add_argument('--baz', action='store_false')
parser.parse_args('--foo --bar'.split())
```

输出:Namespace(foo=True, bar=False, baz=True)

- 'append': 一个存储列表，添加每个参数值到列表中，允许选项被多次指定时很有用。

```
parser = argparse.ArgumentParser()
parser.add_argument('--str', dest='types', action='append_const', const=str)
parser.add_argument('--int', dest='types', action='append_const', const=int)
parser.parse_args('--str --int'.split())
```

输出:Namespace(type=[<class 'str'>, <class 'int'>])

- 'count': 关键参数出现的次数。

```
parser = argparse.ArgumentParser()
parser.add_argument('--verbose', '-v', action='count')
parser.parse_args(['-vvv'])
```

输出:Namespace(verbose=3)

- help: 打印当前 parser 所有选项的帮助信息， 默认帮助行为被添加到 parser。
- version: add\_argument 调用指定 version= 关键字

```
import argparse
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('--version', action='version', version='(%prog) 2.0')
parser.parse_args(['--version'])
```

输出 PROG 2.0。

- 你可以通过传递行为子类或者其它对象的接口传递给 action，推荐的方法是扩展 Action，覆盖掉 \_\_call\_\_ 方法和 \_\_init\_\_。

```
class FooAction(argparse.Action):
    def __init__(self, option_strings, dest, nargs=None, **kwargs):
        if nargs is not None:
            raise ValueError("nargs not allowed")
    def __call__(self, parser, namespace, values, option_string=None):
        print('%r%r%r' % (namespace, values, option_string))
        setattr(namespace, self.dest, values)
parser = argparse.ArgumentParser()
parser.add_argument('--foo', action=FooAction)
parser.add_argumentParser('bar', action=FooAction)
args = parser.parse_args('1 -- foo 2'.split)
```

输出：

Namespace(bar=None,foo=None) '1' None

Namespace(bar=1,foo=None) '2' '--foo'

nargs

- N: 一个整数，命令行下的参数被放到一起成为一个列表：

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', nargs=2)
parser.add_argument('bar', nargs=1)
parser.parse_args('c --foo a b'.split())
```

输出:Namespace(bar=['c'],foo=['a','b'])

- ?：根据不同情况生成不同的值，如果没有参数指定它的值来自默认生成如果有一个带有-前缀的参数值将被 const 参数生成，如果指定了值将生成指定值。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', nargs='?', const='c', default='d')
parser.add_argument('bar', nargs='?', default='d')
parser.parse_args(['XX', '--foo', 'YY'])
parser.parse_args(['XX', '--foo'])
parser.parse_args([])
```

分别输出：

Namespace(bar='XX',foo='YY')

Namespace(bar='XX',foo='x')

```
Namespace(bar='d',foo='d')
```

用 nargs='?' 更常用的用法时允许选项输入输出文件:

```
parser = argparse.ArgumentParser()
parser.add_argument('infile',nargs='?',type=argparse.FileType('r'),default=sys.stdin)
parser.add_argument('outfile',nargs='?',type=argparse.FileType('w'),default=sys.stdout)
parser.parse_args(['input.txt','output.txt'])
```

输出:Namespace(infile=<\_io.TextIOWrapper name='input.txt',encoding='UTF-8'>, outfile=<\_io.TextIOWrapper name='output.txt' encoding='UTF-8'>) parser.parse\_args([])

输出: Namespace(infile=<io.TextIOWrapper name='<stdin>' encoding='UTF-8'>, outfile=<\_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)

- \*: 所有的命令行参数将被放到一个列表中。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo',nargs='*')
parser.add_argument('--bar',nargs='*')
parser.add_argument('--barz',nargs='*')
parser.parse_args('a b --foo x y --bar 1 2'.split())
```

输出:Namespace(bar=['1','2'],baz=['a','b'],foo=['x','y'])

- +: 所有的命令行参数将被添加到一个列表中, 至少需要一个参数否则将报错。

```
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('foo',nargs='+')
parser.parse_args(['a','b'])
parser.parse_args([])
```

输出:Namespace(foo=['a',nargs='+'])

usage: PROG [-h] foo [foo ...]

PROG: error: too few arguments

- argparse.REMAINDER: 所有已经存在的参数被添加到一个列表。

```
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('--foo')
parser.add_argument('command')
parser.add_argument('args',nargs=argparse.REMAINDER)
print(parser.parse_args('--foo B cmd --arg1 xx zz'.split()))
```

输出:Namespace(args=['-arg1','XX','ZZ'],command='cmd',foo='B') 如果 nargs 参数没有提供, argument 由 action 决定, 通常这意味着一个的命令行参数被使用一个项目被产生。

#### const

const 参数被用在保存没有被命令行读入的常数来常数值, 两个常见的用法如下:

- 当 add\_argument() 调用的时候设置了 action='store\_const' 或者是 action='append\_const' 通过增加 const 值到一个 parse\_args() 返回的对象的属性。
- 当 add\_argument() 通过选项字符串 (像-f 或者-foo) 和 nargs='?' , 这将穿件一个由 0 行或者一行参数跟着的选项, 当解析命令行时, 如果选项字符串遇到没有命令行参数的时候, 值 const 将被用来替代。'store\_const' 和'append\_const' 行为, const 关键字参数必须给定, 对于其它行为, 默认为 None。

#### default

所有的参数和一些位置的参数在命令行下可能被忽略, add\_argument() 参数 default 的值默认为 None, 指定当没有参数时什么值被使用。没有指定选项字符串, default 的值将取代参数。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', default=42)
parser.parse_args(['--foo', '2'])
parser.parse_args([])
```

输出: Namespace(foo='2')

Namespace(foo=42)

如果默认值是一个字符串, parser 解析值就好象命令行参数一样, 类似的, parser 应用任何 type 转换参数, 如果在设置属性值前 Namespace 返回值, 否则 parser 用下面的值。

```
parser = argparse.ArgumentParser()
parser.add_argument('--length', default=42, type=int)
parser.add_argument('--width', default=10.5, type=float)
parser.parse_args()
```

输出: Namespace(length=10, width=10.5)

对于参数为'?' 或者'\*', 命令行没有值的时候 default 值将被使用

```
parser = argparse.ArgumentParser()
parser.add_argument('foo', nargs='?', default=42)
parser.parse_args(['a'])
parser.parse_args([])
```

分别输出:

```
Namespace(foo='a')
```

```
Namespace(foo=42)
```

如果 default=argparse.SUPPRESS 如果没有命令行参数将导致没有属性被添加。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', default=argparse.SUPPRESS)
parser.parse_args([''])
parser.parse_args(['--foo', '1'])
```

分别输出:

```
Namespace()
```

```
Namespace(foo='1')
```

type

默认 ArgumentParser 对象读命令行参数为字符串，然而，经常命令行应该以另一种数据类型解析，像 float, int, add\_argument() 的 type 关键字允许需要的类型检查和转换被执行，常用的内部数据类型和参数可以被作为 type 的值直接使用。

```
parser = argparse.ArgumentParser()
parser.add_argument('foo', type=int)
parser.add_argument('bar', type=open)
parser.parse_args('2 temp.txt'.split())
```

输出:Namespace(bar=<\_io.TextIOWrapper name='temp.txt' encoding='UTF-8',foo=2) 为了能轻松的使用多种文件类型,argparse 模块提供了工厂 FileType,利用 mode=,bufsize=,encoding= 和 error= 参数，例如 FileType('w') 可以被用来创建一个可写的文件。

```
parser = argparse.ArgumentParser()
parser.add_argument('bar', type=argparse.FileType('w'))
parser.parse_args(['output'])
```

输出:Namespace(bar=<\_io.TextIOWrapper name='out.txt' encoding='UTF-8';>) type 能够调用一个字符串参数返回转换过值的参数

```
import math
import argparse
def perfect_square(string):
    value = int(string)
    sqrt = math.sqrt(value)
    if sqrt != int(sqrt):
        msg = '%r is not a perfect square' % string
        raise argparse.ArgumentTypeError(msg)
    return value
```

```
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('foo', type=perfect_square)
print(parser.parse_args(['9']))
print(parser.parse_args(['7']))
```

输出: Namespace(foo=9)

usage: PROG [-h] foo

PROG: error: argument foo: '7' is not a perfect square

choise

choise 参数在检查值的范围时很方便。

```
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('foo', type=int, choices=range(5, 10))
parser.parse_args(['7'])
parser.parse_args(['11'])
```

分别输出:Namespace(foo=7)

usage: PROG [-h] 5,6,7,8,9

PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)

choise

一些命令行参数从一些限定值的中选定,可以通过传递 choice 关键字参数给 add\_argument(),当命令行解析的时候,值将被检查如果不在可接受值范围内将显示错误消息。

```
parser = argparse.ArgumentParser(prog='game.py')
parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
parser.parse_args(['rock'])
parser.parse_args(['file'])
```

分别输出:

Namespace(move='rock')

usage: game.py [-h] rock,paper,scissors

game.py: error: argument move: invalid choice: 'fire' (choose from 'rock', 'paper', 'scissors')  
choice 选项检查在转化数据类型后进行。

```
parser = argparse.ArgumentParser(prog='doors.py')
parser.add_argument('door', type=int, choices=range(1, 4))
print(parser.parse_args(['3']))
print(parser.parse_args(['4']))
```

分别输出:

Namespace(door=3)

usage: doors.py [-h] 1,2,3

doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)

任何支持 in 操作的对象都能被传递给 choise 作为值，因此 dict, set 对象都是常用的支待的对象。required

通常 argparse 模块假设 flag 像可以被省略的-f 和-bar,, 为了一个选项必需要设置 required=True。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', required=True)
parser.parse_args(['--foo', 'BAR'])
parser.parse_args([])
```

分别输出：

Namespace(foo='BAR')

usage: argparse.py [-h] [-foo FOO]

argparse.py: error: option -foo is required

正如上例，如果 parse\_args() 的 required 被标记，如果不给值将报错。help

help 值包含一些简单的参数说明，当用户要求帮助的时候（通常用-h 或者--help）, help 描述信息将被展示

```
parser = argparse.ArgumentParser(prog='frobble')
parser.add_argument('--foo', action='store_true', help='foo the bars before
                           frobbing')
parser.add_argument('bar', nargs='+', help='foo the bars before frobbed')
parser.parse_args(['-h'])
```

输出：

usage: frobble [-h] [-foo] bar [bar ...]

positional arguments:

bar one of the bars to be frobbled

optional arguments:

-h, --help show this help message and exit

--foo foo the bars before frobbing

help 字符串能包含多种格式像程序名字或者默认参数，可用的指定包含程序的名字,%(prog)s 和多数 add\_argument() 关键字，像%(default)s,%(type)s 等等。

```
parser = argparse.ArgumentParser(prog='frobble')
parser.add_argument('bar', nargs='?', type=int, default=42, help='the bar to %(prog)
                           s (default :%(default)s)')
parser.print_help()
```

输出:

```
usage: frobble [-h] [bar]
```

optional arguments:

bar the bar to frobble (default: 42)

optional arguments:

-h, --help show this help message and exit

帮助字符串支持% 格式, 如果你想一个% 出现在帮助字符串中, 你需要使用%% argparse 对于指定的选项通过设置 argparse.SUPPRESS 设置支持静默帮助。

```
parser = argparse.ArgumentParser(prog='frobble')
parser.add_argument('--foo', help=argparse.SUPPRESS)
parser.print_help()
```

输出:

```
usage: frobble [-h]
```

optional arguments:

-h, --help show this help message and exit

metavar

当 ArgumentParser 生成帮助消息的时候需要一些方法设计查询每个参数, 默认, ArgumentParser 对象用 dest 值作为每个对象的名字, 默认对于 action 位置的参数, dest 值被直接使用, 对于一些选项行为, dest 值时大写的。因此单个位置参数 dest='bar' 将被认做 bar, --foo 应该被跟着一个命令作为 FOO

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo')
parser.add_argument('bar')
parser.parse_args('X --foo Y'.split())
print.print_help()
```

分别输出:

```
Namespace(bar='X', foo='Y')
```

```
usage: [-h] [-foo FOO] bar
```

optional arguments:

bar

optional arguments:

-h, --help show this help message and exit  
 -foo FOO

一个可用的名字被 metavar 指定:

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', metavar='YYY')
parser.add_argument('bar', metavar='XXX')
parser.parse_args('X -- foo Y'.split())
parser.print_help()
```

Namespace(abr='X',foo='Y')  
 usage: [-h] [-foo YYY] XXX

positional arguments:

XXX

optional arguments:

-h, --help show this help message and exit  
 -foo YYY

注意 metavar 仅仅改变显示的名字, parse\_args() 属性的名字仍然由 dest 值决定。不同的 nargs 也许导致 metavar 被多次使用, 提供一个元组给 metavar 指定一个不同的显示。

```
parser = argparse.ArgumentParser(prog='prog')
parser.add_argument('-x', nargs=2)
parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
parser.print_help()
```

输出:

usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:

-h, --help show this help message and exit  
 -x X X  
 --foo bar baz  
 dest

大多数 ArgumentParser 行为增加一些值作为 parser\_args() 返回值的属性。属性的名字由 dest 决定

```
parser = argparse.ArgumentParser()
parser.add_argument('bar')
parser.parse_args(['xxx'])
```

输出:Namespace(bar='xxx')

对于选项参数, dest 的值从选项字符串推断出, ArgumentParser 通过得到长的选项字符串删除初始化-字符串生成 dest 的值, 如果 meiyou 长的选项字符串提供, dest 将通过初始化字符-从第一个短的字符串选项得到。任何内部-字符将被转换为 \_ 字符确保字符串是一个可用的属性名字。

```
parser = argparse.ArgumentParser()
parser.add_argument('-f', '--foo-bar', '--foo')
parser.add_argument('-x', '-y')
parser.parse_args ['-f 1 -x 2'.split()]
parser.parse_args('--foo 1 -y 2'.split())
```

分别输出:

Namespace(foo\_bar=1,x='2')

Namespace(foo\_bar='1',x='2')

dest 允许自定义属性的名字:

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', dest='bar')
parser.parse_args('--foo XXX'.split())
```

输出:Namespace(bar='XXX')

Action class

Action classes 实现的 Action API, 一个命令行返回的可调的 API。任何这个 API 对象都可以被 zuoweiaction 参数传递给 add\_argument().class argparse.Action(option\_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None) Action 实力应该是可调用的, 因此子类必须被 \_\_call\_\_ 方法覆盖, 应该接受四个参数:

- parser: 包含这个 action 的 ArgumentParser。
- namespace:parser\_args() 返回的 Namespace 对象, 大多数行为通过 setattr() 增加一个属性到对象。
- varlue: 结合命令行参数和任何转化应用, 类型转换被 type 关键字指定。
- option\_string: 宣告像字符串被用于激活这个 action, option\_string 时一个选项, 将

缺席如果这个 action 和 positional 参数结合。`__call__` 方法也许执行任意行为，但是典型的设置基于 dest 和 value 的 namespace 属性。

`parse_args()` 方法:

`ArgumentParser.parse_args(args=None, namespace=None)` 转换参数字符串为对象指定他们作为 namespace 的属性。之前调用 `add_argument()` 决定 决定创建什么对象如何复制， 默认 argument 字符串来自 `sys.argv`, 一个新的空的 Namespace 对象被创建。Option value syntax

`parse_args` 方法支持多种方法指定选项的值，在最简单的情况下，这个选项和它的值被传递作为两个分开的参数:

```
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-x')
parser.add_argument('--foo')
parser.parse_argument('-x', 'X')
parser.parse_args(['--foo', 'FOO'])
```

分别输出:

`Namespace(foo=None,x='X')`

`Namespace(foo='FOO',x=None)`

对于短的选项，这个选项值可以被链接，多个短选项可以被-前缀连接在一起，只要最新的选项（非空）要求值:

```
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-x', action='store_true')
parser.add_argument('-y', action='store_true')
parser.add_argument('-z')
parser.parse_args(['-xyzZ'])
```

输出:`Namespace(x=True,y=True,z='Z')` 不可用的参数

当解析命令行时 `parse_args()` 检查多种错误，包括不明确的选项，不可用的类型， 错误的参数为值等等，当出现一个错误，它推出同时打印错误和用法信息。

```
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('--foo', type=int)
parser.add_argument('bar', nargs='?')

# invalid type
parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

# invalid option
```

```

parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

# wrong number of arguments
parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger

```

## 参数包含

当用户犯错时 `parse_args()` 方法尝试给出错误, 但是一些情况下固有的二义, 例如, 命令行参数-1 可能同时指定一个选项或者尝试提供一个指定位置参数, `parse_args()` 方法导致, 指定位置的参数仅仅用-开始如果他们看起来像负数在 `parser` 没有选像解析看起来像负数:

```

parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-x')
parser.add_argument('foo', nargs='?')

# no negative number options, so -1 is a positional argument
parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

# no negative number options, so -1 and -5 are positional arguments
parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-1', dest='one')
parser.add_argument('foo', nargs='?')

# negative number options present, so -1 is an option
parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

# negative number options present, so -2 is an option
parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

# negative number options present, so both -1s are options
parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]

```

```
PROG: error: argument -1: expected one argument
```

如果你有一个必须以-开始的参数而且不是负数，你可以插入'-'告诉 parse\_args() 之后的一切：

```
parser.parse_args(['--', '-f'])
```

输出:Namespace(foo='f',one=None) 参数缩略 如果缩略没有歧义 parser\_args() 方法默认允许长选项被简写为前缀。

```
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('-bacon')
parser.add_argument('-badger')
parser.parse_args ['-bac MMM'.split()]
Namespace(bacon='MMM', badger=None)
parser.parse_args ['-bad WOOD'.split()]
Namespace(bacon=None, badger='WOOD')
parser.parse_args ['-ba BA'.split()]
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

可能产生多个选项时错误产生，可以通过设置 allow\_abbrev 设置为 False 禁用。Beyond sys.argv

ArgumentParser 通常比 sys.argv 有用，可以穿地一个字符串列表到 parser\_args() 完成，这在测试交互式提示符很有用。

```
parser = argparse.ArgumentParser()
parser.add_argument('integers', metavar='int', type=int, choices=range(100),
nargs='+', help='an integer in range 0..9')
parser.add_argument('--sum', dest='accumulate', action='store_const',
const=sum, default=max, help='sum the integers (default: find the max)')
parser.parse_args(['1','2','3','4'])
parser.parse_args(['1','2','3','4'], '--sum')
```

输出结果分别为：

```
Namespace(accumulate=<built-in function max>, integers=[1,2,3,4])
```

```
Namespace(accumulate=<built-in function sum>, integers=[1,2,3,4])
```

Namespace 对象

class argparse.Namespace，简单的 parse\_args() 创建一个对象，保存属性返回它。这个类很简单，仅仅是一个可读表达的对象子类，如果你希望有字典类似的属性，你可以用标准的 python idiom()：

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo')
```

```
args = parser.parse_args(['--foo', 'BAR'])
var(args)
```

输出:'foo':'BAR'

当 ArgumentParser 指定属性到已经存在的对象时它是很有用的，相比于新的 Namespaced 对象，它可以指定 namespace= 关键参数获得。

```
class C:
    pass
C = C()
parser = argparse.ArgumentParser()
parser.add_argument('--foo')
parser.parse_args(args=['--foo', 'BAR'], namespace=C)
c.foo
```

输出:'BAR'

子命令: ArgumentParser.add\_subparsers([title][, description][, prog][, parser\_class][, action][, option\_string][, dest][, help][, metavar]) 一些程序分割他们的功能为一个子命令，例如， svn 程序可以有子命令 svn checkout,svn commit,svn update。当程序有一些要求不同类型命令行参数的不同的功能的时候分割功能的方法是一个好的想法，ArgumentParser 支持支持 add\_subparsers 一个子命令，add\_subparsers() 方法调用通常没有参数返回一个特殊的行为对象，这个对象是一个方法，add\_parser() 得到一个命令名字和任何 ArgumentParser 够草体参数，返回一个可以被修改的 ArgumentParser 对象。

- title: 帮助输出 sub-parser 组的标题，如果说明提供了的话默认”subcommands”，否则用参数作为标题。
- description: 在输出帮助中描述 sub-parser 组，默认是 None。
- prog:sub-command 的帮助信息，默认程序的名字和位置上的参数在 subparser 参数前。
- parser\_class: 用于创建一个 sub-parser 实例的类，默认时当 parser。
- action: 在命令行中参数出现厚的基础类型的行为。
- dest:sub-command 下属性的名字将被存储，默认没有值被存储。
- help: 在帮助输出的 sub-parser, 默认为 None。
- metavar: 在 help 中可用的子命令默认是 None 代表子命令 cmd1,cmd2,...

用法:

```
# create the top-level parser
parser = argparse.ArgumentParser(prog='PROG')
parser.add_argument('--foo', action='store_true', help='foo help')
subparsers = parser.add_subparsers(help='sub-command help')

# create the parser for the "a" command
parser_a = subparsers.add_parser('a', help='a help')
parser_a.add_argument('bar', type=int, help='bar help')

# create the parser for the "b" command
parser_b = subparsers.add_parser('b', help='b help')
parser_b.add_argument('--baz', choices='XYZ', help='baz help')

# parse some argument lists
parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

注意 parser\_args() 返回的 durian 将包含住 parser 和 subparser 命令行选中的参数，因此在上面的例子中，当一个命令被指定，仅仅 foo 和 bar 被呈现，当 b 被指定，仅仅 foo 和 baz 属性被呈现，类似的，subparser 要求帮助信息，仅仅这个 parser 的帮助信息被打印，帮助信息不包含父或者兄弟 parser 信息。(一个 subparser 命令的帮助消息，然而，可以被 help= 参数增加到上面)

```
parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
{a,b}    sub-command help
      a    a help
      b    b help

optional arguments:
-h, --help show this help message and exit
--foo    foo help

parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
bar    bar help
```

```

optional arguments:
  -h, --help    show this help message and exit

parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help    show this help message and exit
  --baz {X,Y,Z}  baz help

```

add\_subparsers() 方法也支持 title 和 description 关键参数, 当两者都呈现的时候在帮助输出 subparser 的命令将出现在自己的组。

```

>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                         description='valid subcommands',
...                                         help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage: [-h] {foo,bar} ...

optional arguments:
  -h, --help    show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}    additional help

```

更进一步, add\_parser 支持一个 aliases 参数, 允许多字符串访问同一个 subparser, 像 svm, 别名 co 作为 checkout 的简写。

```

>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')

```

一个类似的高效处理 sub-commands 结合 add\_subparsers() 方法调用 set\_default() 以至于每个 subparser 知道那个 python 函数应该被执行。

```

>>> # sub-command functions
>>> def foo(args):

```

```

...     print(args.x * args.y)

...
>>> def bar(args):
...     print('((%s))' % args.z)
...

>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))

```

你可以用 `parse_args()` 在参数解析完成后通过调用合适的函数做这个工作，结合函数和 `action` 像这个像这样典型的轻松的方法处理不同的行为，然而，如果它需要检查 `subparser` 的名字，`dest` 关键值通过 `add_subparsers()` 调用将发挥作用。

```

>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')

```

FileType 对象：

class argparse.FileType(mode='r', bufsize=-1, encoding=None, errors=None) FileType 工厂创建一个能被传递给 ArgumentParser.add\_argument() 的对象。参数有 FileType 对象将用要求的模式打开命令行参数作为文件，换从大小，编码，错误处理。

```
parser = argparse.ArgumentParser()
parser.add_argument('--raw', type=argparse.FileType('wb', 0))
parser.add_argument('out', mtype=argparse.FileType('w', encoding='UTF-8'))
parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
```

输出: Namespace(out=<\_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8', raw=<\_ioFileIO name='raw.dat' mode='wb'>)

FileType 对象明白伪参数同时自动转换 sys.stdin 为可读的 FileType 对象， sys.stdout 可写的 FileType 对象。

```
parser = argparse.ArgumentParser()
parser.add_argument('infile', type=argparse.FileType('r'))
parser.parse_args(['-'])
```

输出 Namespace(infile=<\_io.TextIOWrapper name='<stdin>' encoding='UTF-8') Argument group

ArgumentParser.add\_argument\_group(title=None, description=None)

默认情况下, ArgumentParser groups, 当显示帮助信息的时候命令行参数进入对应位置的参数和选项参数。当有一个比默认更好的概念上的参数组, 合适的组能被 add\_argument\_group() 创建:

```
parser = argparse.ArgumentParser(prog='PROG', add_help=False)
group = parser.add_argument_group('group')
group.add_argument('--foo', help='foo help')
group.add_argument('bar', help='bar help')
parser.print_help()
```

输出: usage: PROG [-foo FOO] bar

group:  
bar bar help  
-foo FOO foo help  
add\_argument\_group() 方法返回一个有 add\_argument() 方法的参数组对象。当一个参数增加到组中, parser 就当它为正常参数,但是在帮助信息中分组显示。add\_argument\_group() 方法接受 title 和 description 参数自定义显示:

```
parser = argparse.ArgumentParser(prog='PROG', add_help=False)
group1 = parser.add_argument_group('group1', 'group1 description')
```

```
group1.add_argument('foo', help='foo help')
group2 = parser.add_argument_group('group2', 'group2 description')
group2.add_argument('--bar', help='bar help')
parser.print_help()
```

usage: PROG [-bar BAR] foo

```
group1:
group1 description

foo foo help
```

```
group2:
group2 description

--bar BAR bar help
```

注意任何不再你的用户定义组中的参数将以对应位置参数和选项参数结束。Mutual exclusion

`ArgumentParser.add_mutually_exclusive_group(required=False)`

创建一个转悠的组，`argparse` 将确保唯一的参数在彼此的组被呈现在命令行。

```
parser = argparse.ArgumentParser(prog='PROG')
group = parser.add_mutually_exclusive_group()
group.add_argument('--foo', action='store_true')
group.add_argument('--bar', action='store_false')
parser.parse_args(['--foo'])
parser.parse_args(['--bar'])
parser.parse_args(['--foo', '--bar'])
```

分别输出：

`Namespace(bar=True,foo=True)`

`Namespace(bar=False,foo=False)`

`sage: PROG [-h] [-foo | --bar]`

`PROG: error: argument --bar: not allowed with argument -foo`

`add_mutually_exclusive_group()` 方法接受一个 `required` 参数，预示着最新的参数被要求。

```
parser = argparse.ArgumentParser(prog='PROG')
group = parser.add_mutually_exclusive_group(required=True)
group.add_argument('--foo', action='store_true')
group.add_argument('--bar', action='store_true')
```

```
parser.parse_args([])
```

输出:

```
usage: PROG [-h] (-foo | -bar)
```

```
PROG: error: one of the arguments -foo -bar is required
```

注意当前的 mutually exclusive 参数组不支持 title 和 description 参数。Parser defaults

```
ArgumentParser.set_defaults(**kwargs)
```

大多数时候,parse\_args() 返回的属性对象将被命令行参数和参数行为完全决定。set\_default() 允许一些额外的属性决定没有命令行增加时的行为:

```
parser = argparse.ArgumentParser()
parser.add_argument('foo', type=int)
parser.set_default(bar=42, baz='badger')
parser.parse_args(['736'])
```

输出:Namespace(bar=42,baz='badger',fpp=736) 注意 parser 级默认覆盖参数级。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', default='bar')
parser.set_defaults(foo='spam')
parser.parse_args([])
```

输出: Namespace(foo='spam') Parser 级别在多个 parser 时特别有用。ArgumentParser.get\_default(dest) 得到 namespace 属性的默认值, 正如设置 add\_argument() 或者 set\_defaults()

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', default='badger')
parser.get_default('foo')
```

输出:'baadger' Printing help

在一些典型的应用中 parse\_args() 将考虑打印用法和错误信息的格式, 然而一些格式方法是可用的: ArgumentParser.print\_usage(file=None): 打印 ArgumentParser 应该在命令行调用的简单描述, 如果 file 是 None, sys.stdout 被假定。ArgumentParser.print\_help(file=None): 打印程序的用法信息和 ArgumentParser 参数注册信息, 如果 file 为 None, sys.stdout 被假定。ArgumentParser.format\_usage(): 返回在命令行中 ArgumentParser 参数应该被如何调用的简要说明字符串。ArgumentParser.format\_help(): 返回一个包含程序用法和 ArgumentParser 参数注册信息的帮助字符串。Partial parsing

```
ArgumentParser.parse_known_args(args=None, namespace=None)
```

有时候一些脚本也许仅仅解析一些命令行参数, 传递参数到另一个脚本或者程序, 在这种情形下, parser.\_known\_args() 方法很有用, 它像 parser\_args() 除了当有额外的参数呈现的时候不生成错误, 相反, 它返回一个包含 populated namespace 和保留参数字符串的列表的两个元素的元组。

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', action='store_true')
parser.add_argument('bar')
parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
```

输出:(Namespace(bar='BAR', foo=True), ['--badger', 'spam']) Customizing file parsing

ArgumentParser.convert\_arg\_line\_to\_args(arg\_line)

从文件中读入的参数一行读一个，convert\_arg\_line\_to\_args() 能被覆盖。这个方法从参数文件得到一个简单的 arg\_line 字符串，返回一个参数列表，每读取一行方法被调用一次。一个有用的覆盖每这个方法是当空格分开的 word 为参数，下面的例子展示：

```
class NyArgumentParser(argparse.ArgumentParser):
    def convert_tag_line_to_args(self, arg_line):
        return arg_line.split()
```

Exiting method

ArgumentParser.exit(status=0, message=None): 这个方法终止程序，以指定的状态推出，如果参数被给，打印消息。ArgumentParser.error(message): 这个方法打印包含消息用法信息到标准错误终止程序以状态代码 2。Upgrading optparse code

最初 argparse 模块尝试用 optparse 维持兼容性，然而 optparse 很难扩展，特别是改变要求支持新的 nargs= 指定更好的用法消息。当大多数 optparse 已经被复制粘贴过或者 monkey-patched，它不再尝试维持向后兼容。, argparse 模块在一些方法改进了标准库 optparse:

- 处理位置参数。
- 支持子命令。
- 允许 + 和/前缀。
- 处理 0 或者更多 1 或者更多风格的参数。
- 处理更多的用法消息。
- 提供简单的接口自定义 type 和 action。

optparse 到 argparse 的并行升级

\item 用 ArgumentParser.add_argument() 调用取代 optparse.OptionParser.add_option()	调用。
\item 用 args=parser.parser_args() 取代 (options, args)=parser.parse_args() 增加	ArgumentParser.add_argument() 调用给指定位置的参数，记住显现的前向，现在在 argparse 上下文称为 args。

```
\item 用 type 和 action 取代 callback 行为和 callback\_* 关键参数。
\item 取代 type 关键字的字符串名字和相关的对象类型（如 int, float, complex 等等）
\item 用 Namespace 和 optparse.OptionError, optparse.OptionValueError 取代 optparse.
Value。
\item 用标准那得 Python 语法取代 \%default 或者 \%prog, \%( default)s 和 \%( prog)s。
\item 通过调用 parser.add\_argument(' --version ', action=' version ', version=' <the
version > ') 取代 OptionParser 结构体 version
。
```

setting 输入:

```
hpc@hpc322:~/文档/Tensorflow$ python code/arg1.py
usage: arg1.py [-h] echo
arg1.py: error: the following arguments are required: echo
hpc@hpc322:~/文档/Tensorflow$ python code/arg1.py -h
usage: arg1.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help  show this help message and exit
hpc@hpc322:~/文档/Tensorflow$
```

add\_argument 方法指定程序需要接受的命令参数，本例中为 echo，此程序运行必须指定一个参数，方法 parse\_args() 通过分析指定的参数返回数据 echo。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="show the help information", type=int)
args = parser.parse_args()
print(args.echo**2)
```

指定参数类型为 int， 默认为 string。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("Verbosity turned on")
```

```
Verbosity turned on
hpc@hpc322:~/文档/Tensorflow$ python code/arg3.py --verbosity a
Verbosity turned on
```

这里指定了--verbosity 程序就显示一些信息，如果不指定程序也不会出错，对应的变量就被设置为 None。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity", action="store_true")
```

```
args = parser.parse_args()
if args.verbosity:
    print("Verbosity turned on")
```

指定一个新的关键词 action, 赋值为 store\_true。如果指定了可选参数, args.verbosie 就赋值为 True, 否则就为 False。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="Increase output verbosity", action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

```
hpc@hpc322:~/文档/Tensorflow$ python code/arg4.py --help
usage: arg4.py [-h] [-v]

optional arguments:
  -h, --help      show this help message and exit
  -v, --verbose  Increase output verbosity
```

```
#args5.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int, help="display help information")
parser.add_argument("-v", "--verbose", action="store_true", help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print("The square of {} equals {}".format(args.square, answer))
else:
    print(answer)
```

输入参数 verbose 和整数 (4) 顺序不影响结果。python args5.py -verbose 4 和 python args5.py 4 -verbose

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int, help="display a square of a given number")
parser.add_argument("-v", "--verbose", type=int, help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose == 2:
```

```

    print("The square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2=={}".format(args.square, answer))
else:
    print(answer)

```

python args6.py 4 -v 0,1,2 通过指定不同的参数 v 为 0,1,2 得到不同的结果。

```

#arg7.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int, help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count", help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print("The square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)

```

这里添加参数 action="count", 统计可选参数出现的次数。python arg7.py 4 -v(出现一次), 对应结果为  $x^2 == 16$

python arg7.py 4 -vv(出现两次), 对应出现 The square of 4 equals 16

```

import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int, help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0, help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity>=2:
    print("The square of {} equals {}".format(args.square, answer))
elif args.verbosity>=1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)

```

加速让 default 参数。这默认为值 0, 当参数 v 不指定时参数就被置为 None, None 不能

和整型比较。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="The base")
parser.add_argument("y", type=int, help="The exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
elif args.verbosity >= 1:
    print("{}^{} == {}".format(args.x, args.y, answer))
else:
    print(answer)
```

为了让后面的参数不冲突，我们需要使用另一个方法：

```
#args10.py
import argparse
parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
parser.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quit", action="store_true")
parser.add_argument("x", type=int, help="The base")
parser.add_argument("y", type=int, help="The exponent")
args = parser.parse_args()
answer = args.x**args.y
if args.quit:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
else:
    print("{}^{} == {}".format(args.x, args.y, answer))
```

可以输入 python arg10.py 3 4 -vq 得到计算结果。

```
import argparse
parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quit", action="store_true")
parser.add_argument("x", type=int, help="The base")
parser.add_argument("y", type=int, help="The exponent")
args = parser.parse_args()
answer = args.x**args.y
```

```
if args.quit:  
    print(answer)  
elif args.verbose:  
    print("{} to the power {} equals {}".format(args.x, args.y, answer))  
else:  
    print("{}^{} == {}".format(args.x, args.y, answer))
```

这里参数 v 和 q 不能同时使用。

## 4.2 path

### 4.2.1 函数说明

- `os.path.abspath(path)`: 返回 path 的绝对路径, 在多数平台下, 相当于调用函数 `normpath(join(os.getcwd(),path))`
- `os.path.basename(path)`: 返回 path 的路径 base name, 第二个元素通过传递 path 给 `split()`, 注意这个结果不同于 unix 的 basename 程序, 这里 basename,'foo/bar' 然会 bar, 而 basename() 函数返回空字符串 ("")。
- `os.path.commonpath(paths)`: 返回 paths 队列中最长的 sub-path, 日国路径中包含绝对路径和相对路径的话将报 ValueError 或者如果 paths 是空, 不想 commonprefix(), 这个函数返回一个错的路径。
- `os.path.dirname(path)`: 返回目录的名字, 就是 path 用 split 分割厚的第一个元素。
- `os.path.exists(path)`: 如果春在路径 path 或者一个打开的文件描述返回 True。对于破掉的符号链接返回 False, 在一些平台, 如果权限不允许执行 `os.stat()` 即使存在物理路径这个函数也返回 False。
- `os.path.lexists(path)`: 如果路径存在返回 True, 对 broken 符号链接返回 True, 等效与 exists()。
- `os.path.expanduser(path)`: 在 Unix 和 Windows 上用 ~ 或者 user 取代用户路径的值。在 unix 上一个 被环境变量 HOME 替代 (如果设置了 HOME 环境变量的话), 否则当前用户的 home 目录通过内建模块 pwd 查找, 一个初始化 user 是寻找在 password 目录里面的目录。
- `os.path.expandvars(path)`: 返回环境变量的值, 子字符串形式时 `namename` 被环境变量名取代, 变形的变量名字和参考不存在的变量将不改变。
- `os.path.getatime(path)`: 返回上次访问路径的时间, 返回一个从 epoch 起经历的秒数, 如果文件不存在或不可访问则报 OSError。
- `os.path.getmtime(path)`: 返回最新修改路径的时间, 返回值时一个 epoch 其开始的秒数, 文件不存在或者不可范围跟时报 OSError。
- `os.path.getctime`: 返回系统的 ctime, 在 Unix 上时最新的 metadata 改变的时间, 在 windows 上时 path 创建的时间, 返回一个从 epoch 起经历的秒数, 如果文件不存在或不可访问则报 OSError。

- os.path.getsize(path): 返回字节表示的路径的大小，如果不存在文件或者文件不可范围跟将报出 OSError。
- os.path.isabs(path): 如果路径是绝对路径返回 True。
- os.path.isfile(path): 如果路径是文件将返回 True。
- os.path.isdir(path): 如果存在路径返回 True。
- os.path.islink(path): 如果路径查询一个目录入口时符号链接返回 True，如果 Python 运行时符号链接不支持将返回 False。
- os.path.ismount(path): 如果 path 是一个挂载点，返回 True。
- os.path.join(path,\*paths): 加入一个或者更多的组建，返回值是连接路径和任何成员的路径。
- os.path.normcase(path): 在 Unix, MAX OS 上返回路径不变，在一些敏感的文件系统上将转换路径为小写，在 windows 上将转化斜线为反斜线，如果 path 不是 str 或者 bytes 将报 TypeError。
- os.path.normpath(path): 删去冗余得分和服，因此 A//B,A/B,A./B,A/foo../B 将变为 A/B. 字符串操作也许改变包含符号链接的意义，在 windows 上它转化斜线为反斜线。
- os.path.realpath(path): 返回指定文件名的确定路径，消除路径中出现的任何符号链接。
- os.realpath(path,start=os.curdir): 从当前路径或者 start 路径返回相对的文件路径，这是一个路径计算：文件系统不妨问确定的存在的或者自然的路径或者 start。
- os.path.samefile(path1,path2): 如果 pathname 值访问相同的文件或者目录则返回 True，这有 device 名字和 i-node 数量决定，如果 os.stat() 调用 pathname 失败将报出异常。
- os.path.sameopenfile(fp1,fp2): 如果 fp1 和 fp2 指定的时相同的文件将返回 True。
- os.path.samestat(stat1,stat2): 如果元组 stat1 和 state2 查询的时相同的文件，返回 True，这个结构可需已经被 os.fstate(),os.lstat() 或者 os.stat() 返回，番薯通过 samefile() 和 sameopenfile() 实现基本的比较。
- os.path.split(path): 分割路径为 (head,tail)。tail 不包含斜线，如果以斜线将诶为，tail 将为空，如果没有斜线，头将为空，如果 path 时空，头尾都为空。后面的斜线从 head

删除出位它是 root(一个或者更多的斜线), 在所有的情况下 join(head,tail) 返回一个路径到相同位置作为路径。

- os.path.splitdrive(path): 返回 pathname 到 (drive,tail), 这里 drive 可以使挂载点或者空字符串。在系统上没有用驱动器指定, 驱动器将为空字符串, 在所有的倾向下, drive+tail 将时相同的路径。在 Windows 上, 分割 pathname 成 drive/UNC 共享点和相对路径, 如果路径包含驱动器驱动器将包含冒号 (splitdrive("c:/dir")) 返回 ("c:", "/dir"), 如果路径包含驱动 UNC 路径, 驱动器将包含主机名和 share, 但是不包含四个分隔符 splitdrive("//host/computer/dir")return("//host/computer","/dir")
- os.path.split(path): 分割路径名为 (root,ext) 像 root+ext == path, ext 时空或者以一个周期开头, 导致 basename 被忽略, splitext('.cshrc') 返回 ('.cshrc', '')
- os.path.supports\_unicode\_filenames(): 如果文件名时 unicode 编码的则为 True。

### 4.2.2 例子

#### 1. 获取文件名, 目录, 扩展, 新文路径。

```
import os
file_path = '~/iris_test.csv'
filename = os.path.basename(file_path)
new_dir = os.path.join('home', 'hpc', filename)
file_dir = os.path.dirname(file_path)
dir1 = '~/'
fulldir = os.path.expanduser(dir1)
sp = os.path.split(new_dir)
print('new_dir:', sp[0], 'ext:', sp[1])
```

#### 2. 查看文件同时打开文件

```
import os
path = '/etc'
filename = 'passwd'
if os.path.isdir(path):
    full_path = os.path.join(path, filename)
    if os.path.isfile(full_path):
        with open(full_path, 'r') as f:
            line = f.readlines()
            for _ in range(len(line)):
                print(line)
```

#### 3. 获取文件大小和修改时间

```
os.path.getsize('/etc/passwd')
os.path.gettime('/etc/passwd')
import time
time.ctime(os.path.gettime('/etc/passwd'))
```

#### 4. 获取当前目录里面的指定文件的文件名

```
dir_name = '/home/hpc/TensorFlow_Notebook/code'
pyfile = [name for name in os.listdir(dir_name) if name.endswith('.py')]
#or use glob and fnmatch
import glob
pyfiles = glob.glob(dir_name+'/*.py')
from fnmatch import fnmatch
pyfiles = [name for name in os.listdir(dir_name) if fnmatch(name, '*.py')]
```

#### 4. 获取指定目录的文件的相关信息

```
import os
import os.path
import glob
import time
path_name = '/home/hpc/TensorFlow_Notebook/code'
pyfiles = glob.glob(path_name+'/*.py')
name_sz_data = [(name, os.path.getsize(name), os.path.getmtime(name)) for name in pyfiles]
file_metadata = [(name, os.stat(name)) for name in pyfiles]
for name, meta in file_metadata:
    print(name, '\t|', meta.st_size, '\t|', time.ctime(meta.st_mtime))
```

### 4.2.3 常见问题

1. 当你的程序获得目录中的一个文件列表，但是当试着打印文件名的时候文件崩溃，出现 UnicodeEncodeError 异常和一条奇怪的消息—surrogates not allow。

打印位置文件名时使用下面的方法可以避免下面的错误：

```
def bad_filename(filename):
    return repr(filename)[1:-1]
try:
    print(filename)
except UnicodeEncodeError:
    print(bad_filename(filename))
```

默认情况下，Python 假定所有文件名都已经根据 sys.getfilesystemencoding() 的值编码过了。但是，有一些文件系统并没有强制要求这样做，因此允许创建文件名没有正确编码的文

件。这种情况不太常见，但是总会有些用户冒险这样做或者是无意之中这样做了（可能是在一个有缺陷的代码中给 open() 函数传递了一个不合规范的文件名）。当执行类似 os.listdir() 这样的函数时，这些不合规范的文件名就会让 Python 陷入困境。一方面，它不能仅仅只是丢弃这些不合格的名字。而另一方面，它又不能将这些文件名转换为正确的文本字符串。Python 对这个问题的解决方案是从文件名中获取未解码的字节值比如 \xhh 并将它映射成 Unicode 字符 \udchh 表示的所谓的“代理编码”。当你有一个不合格的文件名在目录列表中的是后，python 会将其转化为 unicode 如果你有代码需要操作文件名或者将文件名传递给 open() 这样的函数，一切都能正常工作。只有当你想要输出文件名时才会碰到些麻烦（比如打印输出到屏幕或日志文件等）。特别的，当你想打印上面的文件名列表时，你的程序就会崩溃，崩溃的原因就是字符\udce4 是一个非法的 Unicode 字符。它其实是一个被称为代理字符对的双字符组合的后半部分，因此他是一个非法的 Unicode，所以唯一能称该输出的方法就是遇到不合法文件名时采取相应的补救措施。可以将上述代码修改为：

```
for name in files:
    try:
        print(name)
    except UnicodeEncodeError:
        print(bad_filename(name))
```

或者：

```
def bad_filename(filename):
    temp = filename.encode(sys.getfilesystemencoding(), errors='surrogateescape')
    return temp.decode('latin-1')
```

## 2. 不关闭一个以打开的文件前提下增加或改变它的 Unicode 编码。

```
>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f.buffer
<_io.BufferedReader name='sample.txt'>
>>> f.buffer.raw
<_io.FileIO name='sample.txt' mode='wb'>
>>>
```

在这个例子中，io.TextIOWrapper 是一个编码和解码 Unicode 的文本处理层，io.BufferedReader 是一个处理二进制数据的带缓冲的 I/O 层，io.FileIO 是一个表示操作系统底层文件描述符的原始文件。增加或改变文本编码会涉及增加或改变最上面的 io.TextIOWrapper 层。

detach() 会断开文件最顶层并返回第二层，之后顶层就没什么用了，例如

```
>>> f = open('text.txt', 'w')
>>> f = io.TextIOWrapper(f.buffer, encoding='latin-1')
```

```
>>> b = f.detach()
>>> f.write('hello')
ValueError
<ipython-input-21-0ec9cf64e174> in <module>()
----> 1 f.write('hello')

ValueError: underlying buffer has been detached
```

一旦断开最顶层后，你就可以给返回结果添加一个新的最顶层，比如：

```
>>> f = io.TextIOWrapper(b, encoding='latin-1')
<_io.TextIOWrapper name='text.txt' encoding='latin-1'>
```

在文本模式打开的文件中写入原始的字节数据（将数据直接写入缓冲区）

```
In [1]: import sys

In [2]: sys.stdout.write(b'Hello\n')
-----
TypeError
<ipython-input-2-51d3384e9645> in <module>()
----> 1 sys.stdout.write(b'Hello\n')

TypeError: write() argument must be str, not bytes

In [3]: sys.stdout.buffer.write(b'Hello\n')
Hello
```

类似的，能够读取文本的 buffer 属性来读取二进制数据。I/O 系统以层级结构的形式构建而成。文本文件是通过在一个拥有缓冲的二进制模式文件上增加一个 Unicode 编码/解码层来创建。buffer 属性指向对应的底层文件。如果你直接访问它的话就会绕过文本编码/解码层。

本小节例子展示的 sys.stdout 可能看起来有点特殊。默认情况下，sys.stdout 总是以文本模式打开的。但是如果你在写一个需要打印二进制数据到标准输出的脚本的话，你可以使用上面演示的技术来绕过文本编码层。3. 你有一个对应于操作系统上一个已经打开的 I/O 通道（比如文件，管道，套芥子等）的整形文件描述符，你想将它包装成一个更高层的 Python 文件对象。

一个文件描述符和一个打开的普通文件不一样。文件描述符仅仅是一个操作系统指定的整数，用来指代某系统的 I/O 通道。如果你碰巧有这么一个文件描述符你可以通过 shiyingopen() 函数来将其包装为一个 Python 的文件对象。你仅仅需要使用这个整数值的文件描述符作为第一个参数来替代文件名即可：

```
In [4]: import os
```

```
In [5]: fd = os.open('text.txt', os.O_WRONLY|os.O_CREAT)
In [6]: f = open(fd, 'wt')
In [7]: f.write('hello world\n')
In [8]: f.close()
```

当高层文件对象被关闭或者破坏的时候，底层文件描述符也会被关闭。如果这个并不是你想要的结果，你可以给 `open()` 函数传递一个可选的 `closefd=False`. 比如:

```
f = open(fd, 'wt', closefd=False)
```

在 Unix 系统中，这种包装文件描述符的技术可以很方便的将一个类文件接口作用于一个以不同方式打开的 I/O 通道上，如管道、套接字等。举例来讲，下面是一个操作管道的例子：

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_client(client_sock, addr):
    print('Got connection from', addr)

    # Make text-mode file wrappers for socket reading/writing
    client_in = open(client_sock.fileno(), 'rt', encoding='latin-1',
                      closefd=False)

    client_out = open(client_sock.fileno(), 'wt', encoding='latin-1',
                      closefd=False)

    # Echo lines back to the client using file I/O
    for line in client_in:
        client_out.write(line)
        client_out.flush()

    client_sock.close()

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        echo_client(client, addr)
```

需要重点强调的一点是，上面的例子仅仅是为了演示内置的 `open()` 函数的一个特性，并且也只适用于基于 Unix 的系统。如果你想将一个类文件接口作用在一个套接字并希望你的代码可以跨平台，请使用套接字对象的 `makefile()` 方法。但是如果考虑可移植性的话，那上面的解决方案会比使用 `makefile()` 性能更好一点。

你也可以使用这种技术来构造一个别名，允许以不同于第一次打开文件的方式使用它。例如，下面演示如何创建一个文件对象，它允许你输出二进制数据到标准输出（通常以文本模式打开）：

```
import sys
# Create a binary-mode file for stdout
bstdout = open(sys.stdout.fileno(), 'wb', closefd=False)
bstdout.write(b'Hello World\n')
bstdout.flush()
```

尽管可以将一个已存在的文件描述符包装成一个正常的文件对象，但是要注意的是并不是所有的文件模式都被支持，并且某些类型的文件描述符可能会有副作用（特别是涉及到错误处理、文件结尾条件等等的时候）。在不同的操作系统上这种行为也是不一样，特别的，上面的例子都不能在非 Unix 系统上运行。

### 5. 创建临时文件和文件夹，在程序执行完后自动销毁。

```
from tempfile import TemporaryFile
with TemporaryFile('w+t') as f:
    # Read/write to the file
    f.write('Hello world \n')
    f.write('testing\n')
    # Seek back to beginning and read the data
    f.seek(0)
    data = f.read()
# Temporary file is destroyed
```

或者，如果你喜欢，你还可以像这样使用临时文件：

```
f = TemporaryFile('w+t')
# Use the temporary file
...
f.close()
# File is destroyed
```

`TemporaryFile()` 的第一个参数时文件模式，通常来将文本模式使用 `w+t`, 二进制模式使用 `w+b`。这个模式同时支持读和写操作，在这里很有用，因为当你关闭文件去修改模式的时候，文件实际上已经不存在了。`TemporaryFile()` 另外还支持内置的 `open()` 函数一样的参数。比如：

```
with TemporaryFile('w+t', encoding='utf-8', errors='ignore') as f:
    ...
```

在大多数系统上，同感 `TemporaryFile()` 创建的文件都是匿名的，甚至连目录都没有。如果你想打破这个限制，可以使用 `NamedTemporaryFile()` 来代替。比如：

```
In [16]: with NamedTemporaryFile('w+t') as f:
    ...:     print('filename is:', f.name)
from tempfile import NamedTemporaryFile
with NamedTemporaryFile('w+t') as f:
    print('filename is:', f.name)
filename is: /tmp/tmp4dwoxytf
```

这里被打开的文件的 `f.name` 属性包含了临时文件的文件名。当你需要将文件传递给其它代码来打开这个文件的 `scipio`, 这个就很有用了, 和 `TemporaryFile()` 一样, 结果文件关闭时会被自动删除调。如果你不想这么做呢, 可以传递一个关键字参数 `delte=False` 即可。比如:

```
with NamedTemporaryFile('w+t', delete=False) as f:
    print('filename is:', f.name)
    ...
```

为了创建一个临时目录, 可以使以哦嗯 `tempfile.TemporaryDirectory()`。比如:

```
from tempfile import TemporaryDirectory

with TemporaryDirectory() as dirname:
    print('dirname is:', dirname)
    # Use the directory
    ...
# Directory and all contents destroyed
```

`TemporaryFile()`、`NamedTemporaryFile()` 和 `TemporaryDirectory()` 函数应该是处理临时文件目录的最简单的方式了, 因为它们会自动处理所有的创建和清理步骤。在一个更低的级别, 你可以使用 `mkstemp()` 和 `mkdtemp()` 来创建临时文件和目录。比如:

```
In [19]: tempfile.mkstemp()
Out[19]: (13, '/tmp/tmp6heplg63')

In [20]: tempfile.mkdtemp()
Out[20]: '/tmp/tmpcd70_9po'
```

但是, 这些函数并不会做进一步的管理了。例如, 函数 `mkstemp()` 仅仅就返回一个原始的 OS 文件描述符, 你需要自己将它转换为一个真正的文件对象。同样你还需要自己清理这些文件。

通常来讲, 临时文件在系统默认的位置被创建, 比如 `/var/tmp` 或类似的地方。为了获取真实的位置, 可以使用 `tempfile.gettempdir()` 函数。比如:

```
In [20]: tempfile.mkdtemp()
Out[20]: '/tmp/tmpcd70_9po'
```

```
In [21]: tempfile.gettempdir()
Out[21]: '/tmp'
```

所有和临时文件相关的函数都允许你通过使用关键值参数 prefix,suffix 和 dir 来自定义目录以及命名规则，比如：

```
In [24]: from tempfile import NamedTemporaryFile
In [25]: f = NamedTemporaryFile(prefix='mytemp', suffix='.txt', dir='/tmp')
In [26]: f.name
'/tmp/mytempw2pxl2v5.txt'
```

最后还有一点，尽可能以最安全的方式使用 tempfile 模块来创建临时文件。包括仅给当前用户授权访问以及在文件创建过程中采取措施避免竞态条件。

你需要将一个 Python 对象序列化为一个字节流，一边将它保存到一个文件，存储到数据库或者通过网络传输它。

用 pickle 模块将一个对象保存在一个文件中

```
import pickle
data = [1, 2, 3, 4]
f = open('sample', 'wb')
pickle.dump(data, f)
```

将一个对象保存在一个文件中

```
import pickle
data = range(10)
f = open('temp', 'wb')
pickle.dump(data)
```

如果想将一个对象转化为字符串，可以使用 pickle.dumps()：

```
s = pickle.dumps(data)
```

为了从字节流中恢复一个对象，使用 pickle.load() 或者 pickle.loads() 函数。比如：

```
# Restore from a file
f = open('somefile', 'rb')
data = pickle.load(f)

# Restore from a string
data = pickle.loads(s)
```

对于大多数应用程序来讲，dump() 和 load() 函数的使用就是你有效使用 pickle 模块所需的全部了。它可适用于绝大部分 Python 数据类型和用户自定义类的对象实例。如果你碰到某个库可以让你在数据库中保存/恢复 Python 对象或者是通过网络传输对象的话，那么很有可能这个库的底层就使用了 pickle 模块。

pickle 是一种 Python 特有的自描述的数据编码。通过自描述，被序列化后的数据包含每个对象开始和结束以及它的类型信息。因此，你无需担心对象记录的定义，它总是能工作。举个例子，如果要处理多个对象，你可以这样做：

```
>>> import pickle
>>> f = open('somedata', 'wb')
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump('hello', f)
>>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
>>> f.close()
>>> f = open('somedata', 'rb')
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'hello'
>>> pickle.load(f)
{'Apple', 'Pear', 'Banana'}
>>>
```

还能序列化成函数，类，接口，但是结果数据仅仅将他们的名称编码成对应的代码对象。例如

```
>>> import math
>>> import pickle.
>>> pickle.dumps(math.cos)
b'\x80\x03cmath\ncos\nq\x00.'
```

当数据反序列化回来的时候，会先假定所有的源数据时可用的。模块、类和函数会自动按需导入进来。对于 Python 数据被不同机器上的解析器所共享的应用程序而言，数据的保存可能会有问题，因为所有的机器都必须访问同一个源代码。有些类型的对象是不能被序列化的。这些通常是那些依赖外部系统状态的对象，比如打开的文件，网络连接，线程，进程，栈帧等等。用户自定义类可以通过提供 `__getstate__()` 和 `__setstate__()` 方法来绕过这些限制。如果定义了这两个方法，`pickle.dump()` 就会调用 `__getstate__()` 获取序列化的对象。类似的，`__setstate__()` 在反序列化时被调用。为了演示这个工作原理，下面是一个在内部定义了一个线程但仍然可以序列化和反序列化的类：

```
# countdown.py
import time
import threading

class Countdown:
    def __init__(self, n):
```

```

self.n = n
self.thr = threading.Thread(target=self.run)
self.thr.daemon = True
self.thr.start()

def run(self):
    while self.n > 0:
        print('T-minus', self.n)
        self.n -= 1
        time.sleep(5)

def __getstate__(self):
    return self.n

def __setstate__(self, n):
    self.__init__(n)

```

运行下面结构化代码

```

>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...
>>> # After a few moments
>>> f = open('cstate.p', 'wb')
>>> import pickle
>>> pickle.dump(c, f)
>>> f.close()

```

然后退出 Python 解析器并重启后再试验下:

```

>>> f = open('cstate.p', 'rb')
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...

```

你可以看到线程又奇迹般的重生了，从你第一次序列化它的地方又恢复过来。

pickle 对于大型的数据结构比如使用 array 或 numpy 模块创建的二进制数组效率并不是一个高效的编码方式。如果你需要移动大量的数组数据，你最好是先在一个文件中将其保存为数组数据块或使用更高级的标准编码方式如 HDF5 (需要第三方库的支持)。

由于 pickle 是 Python 特有的并且附着在源码上，所有如果需要长期存储数据的时候不应该选用它。例如，如果源码变动了，你所有的存储数据可能会被破坏并且变得不可读取。坦白来讲，对于在数据库和存档文件中存储数据时，你最好使用更加标准的数据编码格式如 XML, CSV 或 JSON。这些编码格式更标准，可以被不同的语言支持，并且也能很好的适应源码变更。



# Chapter 5

re

## 5.1 正则表达式介绍

操作符	说明	实例
[]	字符集合, 对单个字符给出取值范围	[abc]表示 a,b,c,[a-z]表示 a 到 z 的单个字符
.	任何单个字符	
[^]	非字符集, 对单个字符给出排除范围	[^ abc]表示非 a 或者 b 或者 c 的单个字符
*	前一个字符 0 次或者无限次扩展	abc* 表示 ab,abc,abcc 等
+	前一个字符 1 次或无限次扩展	abc+ 表示 abc,abcc,abccc 等
?	前一个字符 0 次或者一次扩展	abc? 表示 ac,abc
	左右表达式任一个	abc def 表示 abc 或者 def
{m}	扩展前一个字符 m 次	ab{2}c 表示 abc,abbc
{m,n}	扩展前一个字符 m 到 n 次, 包含 n	ab{1,2}c 表示 abc,abbc
^	匹配字符串开头	^abc 表示 abc 且在一个字符串开头
\$	匹配字符串结尾	abc\$ 表示 abc 且在一个字符串的结尾
()	分组标记, 内部只能使用   操作符	(abc) 表示 abc, (abc def) 表示 abc 或者 def
(?...)	这是一个扩展的符号, 第一个字符在'?'后面决定了深层的语法。扩展通常没有创建一个新的 group,(?P<name>... 时该规则惟一的特例)	
(?aiLmsux)	来自集合'a','i','L','m','s','u','x' 的一个或者多个字母, group 匹配空字符串字符给整个正则表达式设置相关的 flags: re.A,re.I,re.L,re.M,re.S,re.X。如果你洗完桑包含 flags 作为正则表达式的一部分而不是传递一个 flag 参数到 re.compile() 函数这就是很有用的, Flasg 应该首先用在表达式字符串。	

操作符	说明	实例
(?:...)	非捕获版本的正则括号，匹配括号中无论什么正则表达式，但是在执行一个匹配或者查询之后 group 中子字符串匹配不能被获得	
\d	数字等价与 [0-9]	
\D	非数字等价与 [0-9]	
\number	匹配相同 number 的组。组以 1 开始，例如 (.+) \1 匹配'the the'or'55 55'，但是'the the'(中间需要有空格)，这种特殊的序列仅仅被用来匹配 1 到 99 组。如果第一个数字为 0 或者是 3 为八进制的，他将被解释为一个 group match，在字符类 '[' and ']' 中，所有的数被当作字符。	
A 匹配	匹配字符串的开始	
\b	匹配空字符串，但是仅仅是单词前面或者后面的空字符串，单词被定义为一个 unicode 字母数字序列或下划线特征，因此单词为被空格或者为字母数字预示，非强跳得字符串，注意，\b 被定义为 a\w 和 a\W 之间，或者在\w 和单词开始之间，这意味着 r'\bfoo\b' 匹配'foo','foo.),(foo)',bar foo baz' 而不是'foobar' 或者'foo3'	
\B	匹配空字符串，但是仅仅当它不在单词的开头或者结尾时，这意味着 r'py\B' 匹配'python','py3','py2'，而不是'py','py.' 或者是'py!'\B 和\b 相反，因此单词时 unicode 字母数字或者下划线，尽管这能被 ASCII flag 改变	
\S	匹配不是任何不是空格的 unicode 字符，和\s 相反，如果 ASCII flag 被用这因为等于 [\t\n\r\f\v](但是 flag 影响整个正则表达式，因此在这种情况下 [\t\n\r\f\v])	
z	匹配字符串的尾部	
(?imsx-imsx:...)	在字符字母集合'i','m','s','x' 中，'-' 跟着的来自同样字母集合的一个或者更多字母)，对于部分表达式字母集合或者移去相关的 flags:re.I,re.M,re.S,re.X。	
<?P=name>	: 对于 group 的一个反向引用，它匹配之前 name 命名的 group 无论什么文本。	

(?+...)	一个注释，括号里面的内容被简单的忽视	
(?=...)	如果... 匹配下一步，不小于任何字符串。例如 Isaac (?=Asimov) 将匹配'Isacc' 如果它被'Asimov' 跟着的话。	
(?!....)	如果... 不匹配下一个，例如 Isaac (?!Asimov) 将匹配'Isaac'，仅仅是它没有'Asimov' 跟着。	
\w	单词字符，等价与 [A-Za-z0-9_]	

正则表达式的语法实例

P(Y YT YTH YTHO)?N	'PN', 'PYN', 'PYTN', 'PYTHN', 'PYTHON'
PYTHON+	'PYTHON', 'PYTHONN', 'PYTHONNN', ...
PY[TH]ON	'PYTON', 'PYHON'
PY[TH]?ON	'PYON', 'PYaON', 'PYbON', 'PYcON', ...
PY{:3}N	'PN', 'PYN', 'PYYN', 'PYYYN', ...

常用的正则表达式:

^[A-Za-Z]+\$	26 个字母组成的字符串
^[A-Za-z0-9]+\$	由 26 个字母和数字组成的字符串
^-?\d+\$	整数形式的字符串
^[0-9]*[1-9][0-9]* \$	正整数形式的字符串
[1-9]\d{5}	中国境内邮政编码，6 位
[\u4e00-\u9fa5]	匹配中文字符
\d{3}-\d{8} \d{4}-\d{7}	国内电话号码，010-68913536

匹配 IP 地址的正则表达式: \d+\.\d+\.\d+ 或者 \{1,3\}. 精确写法:

0-99:[1-9]?\\d

100-199:1\\d{2}

200-249:2[0-4]?\\d

250-255:25[0-5]

IP 地址的正则表达式:((([1-9]?\\d|1\\d{2}|2[0-4]\\d|25[0-5]).){3}(([1-9]?\\d|1\\d{2}|2[0-4]\\d|25[0-5]))

## 5.2 RE 库的主要功能函数

re.search()	在一个字符串搜索匹配正则表达式的第一位置。
re.match()	从一个字符的开始为值起匹配正则表达式，返回 match 对象。
re.fullmatch()	如果整个字符串匹配正则表达式然会相应的 match 对象，不匹配返回 None，注意这不同于 0 长度匹配
re.findall()	搜索字符串，以列表类型返回全部匹配的字串
re.split()	将一个字符串按照正则表达式匹配结果进行分割，返回列表类型
re.finditer()	搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素时 match 对象
re.sub()	在字符串中替换所有匹配正则表达式的子串，返回替换后的字符串。
re.subn()	执行替换操作凡是返回一个 (new_string,number_of_subs_made) 元组
re.escape(pattern)	转义素有的字符除了 ASCII 字母，数字和下划线，如果你想匹配一个也许有正则表达式在里面的任一字符串这是很有用的。
re.purge()	清除正则表达式缓存

re.search(pattern,string,flags=0): 在一个字符串中搜索匹配正则表达式的第一位置返回 match 对象。

- pattern: 正则表达式的字符串或原声字符串表示。
- string: 待匹配字符串。
- flags: 正则表达式使用时的控制标记。

5.2.1 re 表达式中的 flags	使\w \W\b\B\d\D \s\S 值执行 ASCII 匹配而不是 Unicode 匹配，仅仅对于 Unicode 样式有意义对 Byte 样式忽略。
re.A	
re.DEBUG	显示编译表达式的调试信息
re.I	忽略正则表达式的大小写，[A-Z] 能够匹配小写。
re.L	使得\w \W\b\B\d\D \s\S 依赖于当前现场，当现场机制不可信时不鼓励使用，在不管什么时候它处理一个 cultrue，你应该用 Unicode 匹配，这个 flag 仅仅可以被用在 bytes 样式中。
re.M	正则表达式中的操^ 作能够将给定字符串的每一行当作匹配开始
re.S	正则表达式中的. 操作能够匹配所有的字符，默认匹配除换行外的所有字符
re.VERBOSE(re.X)	这个 flag 通过允许你分割逻辑部分和增加注释允许你写的正则表达式更好，空 pattern 中的空格被忽略特别是当一个字符类或者当有为转义的反斜线时，当一行包含不饿时字符类得 # 和非转义斜线时，所有的左边以 # 开头的字符将被忽略
re,error(msg,pattern=None, pos=None)	<ul style="list-style-type: none"> <li>- msg: 非正式格式的错误消息</li> <li>- pattern: 正则表达式</li> <li>- pos: 在 pattern 编译失败的索引（也许是 None）</li> <li>- lineno: 对应位置的行（也许是 None）</li> <li>- colno: 对应位置的列（也许是 None）</li> </ul>

```
import re
match = re.match(r'1\d{5}', 'BIT 100081')
if match:
    match.group(0)
```

re.match(pattern,string,flags=0): 从一个字符串的开始位置起匹配正则表达式，返回 match 对象。

```
import re
match = re.match(r'1\d{5}', '100081 BIT')
if match:
    print(match.group(0))
```

re.findall(pattern,string,flags=0): 搜索字符串，以列表类型返回能匹配的子串。

```
import re
ls = re.findall(r'1\d{5}', 'BIT 100081 TSU100084')
```

re.split(pattern,string,maxsplit = 0,flags=0): 将字符串按照正则表达式匹配结果进行分割，返回列表类型。

maxsplit: 最大分割数，剩余部分作为最后一个元素输出。

```
import re
re.split(r'1\d{5}', 'BIT100081 TSU100084')
re.split(r'1\d{5}', 'BIT100081 TSU100084', maxsplit=1)
```

re.finditer(pattern,string,flags=0): 搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素时 matchdurian。

```
import re
for m in re.finditer(r'1\d{5}', 'BIT100081 TSU100084'):
    if m:
        print(m.group(0))
```

re.sub(pattern,repl,string,count=0,flags=0) 在一个字符串中替换所有匹配正则表达式的子串返回替代后的字符串。

- repl: 替换匹配字符串的字符串
- string: 待匹配字符串
- count: 匹配的最大替换次数

```
import re
re.sub(r'1\d{5}', '110', 'BIT100081 TSU100084')
```

Re 库的另一种等价用法:

```
rst = re.search(r'1\d{5}', 'BIT 100081')
```

等价于

```
pat = re.compile(r'1\d{5}')
pat.search('BIT 100081')
```

regex.search	在字符串中搜索匹配正则表达式的第一位置，返回 match 对象
regex.match()	在字符串的开始为值起配置正则表达式，返回 match 对象
regex.findall()	所有字符串，以列表类型返回全部能匹配的子串
regex.split()	将字符串按照正则表达式匹配结果进行分割，返回列表类型。
regex.finditer()	搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素是 match 对象
reg.sub()	在一个字符串中替换所有匹配正则表达式的子串，返回替换后的字符串

Match 对象：一次匹配的结果，包含匹配的很多信息。

```
match = re.search(r'1\d{5}', 'BIT 100081')
if match:
    print(match.group(0))
type(match)
```

match 对象的属性和方法

.string	待匹配的文本
.re	匹配时使用的 patter 对象 (正则表达式)
.pos	正则表达式搜索文本的开始位置
.endpos	正则表达式搜索文本的结束位置
.group(0)	获得匹配后的字符串
.start()	匹配字符串在原始字符串的开始位置
.end()	匹配字符串的结尾位置
.span()	返回 (.start(),.end())
.expand()	用 sub() 方法返回一个通过在 temple 字符串替代\的像\n 被转换成合适的字符串，数值反向索引 (\1,\2) 和 (\g<1>,\g<name>) 被相应组里面的内容取代\字符串
.__getitem__(g)	允许你轻松的访问一个 match 组
.groupdict(default=None)	返回一个包含所有子组的匹配对象，key 是子组的名字，被用在 groups 的默认参数 默认参数不参加匹配，默认值时 None。
.lastindex	最新匹配的组的整数索引，或者如果没有组被匹配就为 None。例如表达式 (a)b,((a)(b)) 和 ((ab)) 将有 lastindex == 1 如果应用的字符串'ab'，然而表达式 (a)(b) 将有 lastindex == 2，如果与应用在同一个字符串。
.lastgroup	最新匹配名字，如果 group 没有一个名字或者没有 group 就匹配为 None。
.re	正则表达式的 match() 或者 search() 方法生成的 match 实例

Re 库默认采用贪婪匹配，即输出匹配最长的字子串

```
match = re.search(r 'PY.*N', 'PYANBNCNDN')
match.group(0)
```

通常搜索的时候 PYAN 就能匹配出结果但是根据贪婪匹配，匹配待匹配字符串中最长的字符串。输出最短子串 PYAN。

```
match = re.search(r 'PY.*?N', 'PYANBNCNDN')
```

最小匹配操作符

操作符	说明
*?	前一个字符 0 次或者无限次扩展，最小匹配
+?	前一个字符 1 次或者浮现次扩展，最小匹配
??	前一个字符 0 次或者 1 次扩展，最小匹配
{m,n}?	扩展前一个字符串 m 到 n 次 (含 n)，最小匹配

```
import re
m = re.match(r'(\w+ \w+)', 'Isaac Newton, physicist')
m.group(0)
m.group(1)
m.group(2)
m.group(1, 2)
```

输出：

```
'Isaac Newton'
'Isaac'
'Newton'
('Isaac','Newton')
```

```
m = re.match(r'(\d+).(\d+)', '3.1415')
m.groups()
```

输出：

```
('3','1415')
```

```
m = re.match(r'(?P<first_name>\w+) (?P<last_name>\w+)', 'Malcolm Reynolds')
m.groupdict()
```

输出：'first\_name': 'Malcolm', 'last\_name': 'Reynolds'

# Chapter 6

## Sys

- sys.abiflags: 在 POSIX 体同上 Python 用标准的 configure 脚本编译, 包含 PEP3149 指定的 ABI flags。
- sys.argv: 传递给 Python 的命令行参数, argv[0] 是脚本的名字, 在解释器中如果命令行用-c 选项, argv[0] 被设置为'-c'。如果没有脚本名字被传递给 python 解释器, argv[0] 是空字符串。
- sys.base\_exec\_prefix: Python 启动时设置, 在 site.py 之前运行前设置为 exec\_prefix。如果不运行一个虚拟环境, 值保持不变, 如果 site.py 找到的虚拟环境被用了, prefix 和 exec\_prefix 的值将被改变到指向虚拟环境, 由于 base\_prefix 和 base\_exec\_prefix 将任何指向 python 安装的 base 环境 (虚拟换将被创建)。
- sys.base\_prefix: 在 site.py 运行前 python 启动中值和 prefix 相同。如果不运行在虚拟环境中, 值将保持不变 rugosasite.py 找到一个虚拟环境被用, prefix 和 exec\_prefix() 值将被改变到指向虚拟环境, 由于 base\_prefix 和 base\_exec\_prefix 将保留指向 python 安装的 base 环境 (虚拟换将被创建)。
- byteorder: 本地变量的指示器, 这将在 big-endian 平台有一个值'big','title' 在 little-endian 平台。
- sys.vuiltin\_module\_name: 被编译进 Python 解释器的模块的字符串元组。(信息在其他方法下不可用-modules.keys() 仅仅显示导入的模块)。
- sys.call\_tracing(func,args): 调用 func(\*args), 当 trace 使能时。trace 状态被后来保存和恢复。从 checkpoint 文件 debug 去玄幻调试其它代码。
- sys.copyright: 包含 python 解释器版权信息的字符串。

- sys.\_clear\_type\_cache(): 清除内部变量的缓存，类型缓存用来加速属性和方法的查找这个函数用来降低泄漏 debug 的非比要得查找。
- sys.\_current\_fnames(): 返回映射每个线程的标识符到函数调用时的线程栈的栈顶。注意 traceback 模块中的函数能编译调用被给定一个帧的栈。在调试线程锁时很有用：这个函数线程锁死操作，这样线程的调用被冻结和它们的死锁一样长。帧返回一个非死锁的线程也许忍受没有关系到当前这次调用的代码激活的线程检查帧。, 这个函数仅仅被用在内部或者特殊的目的。
- sys.\_debugmallocstats(): 打印 cpython 内存分贝其的低级的信息到标准的错误输出。如果 python 配置了-with-pydebug, 它也只行一些开销巨大的内部组成检查。
- sys.dllhandle: 指定处理 python dll 的整数，在 Windows 上可用。
- sys.displayhook(value): 如果值为 None, 函数打印 rep(value) 到 sys.stdout, 报春之在 builtins.\_。如果 repr(value) 时不可编码的 sys.stdout.encoding 和 sys.stdout.error 句柄，解码 sys.stdout.encoding 和 backslashplace 错误句柄。sys.displayhook 被调用在计算输入交互式 python 会话表达式的结果，显示值能通过指定参数被自定义。

```
def displayhook(value):
    if value is None:
        return
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.buffer.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

- sys.dont\_write\_bytecode: 如果为真，python 不尝试写.pyc 文件到源模块，值依赖-B 命令行选项和 PYTHONDONTWRITEBYTECODE 环境变量通过设置 True 或者 False 确定，但是你可以在你自己控制二进制文件生成。
- sys.exc\_info(type,value,traceback): 这个函数打印出一个给定的 traceback 和 sys.stderr 异常。当出现异常时，解释器设置三个参数异常类，异常实例和 traceback 对象调用

`sys.execpthook`。在交互式会话中这发生在控制被返回到终端前，在 Python 程序中仅当程序退出时被调用，在处理类似顶级异常可以通过指定另一个三个参数函数它哦`sys.exeothook`。

- `sys.__displayhook__`
- `sys.__excepthook__`: 这个对象在程序的开头包含 `displayhook` 和 `excepthook` 的初始值，他们被保存以至于他们被异常取代时 `displayhook` 和 `excepthook` 可以被恢复。
- `sys.exc_info`: 这个函数给出关于当前被处理的异常的信息的元组。信息返回被指定到当前线程和当前栈帧，如果当前栈帧没有处理异常，信息被调用的栈帧得到，或者它的调用器得到，因此直到在处理异常时栈帧被发现，这里处理一个异常被定义为处理一个异常发生。对于任何栈帧，仅仅当前异常信息被处理。如果在栈帧中没有异常被处理，返回包含三个 `None` 的元组。否则返回值为 (`type,value,traceback`)，他们分别为 `d` 得到的被处理异常的类型，异常实例，和 `traceback` 对象（压缩调用栈）。
- `sys.exec_prefix`: 一个字符串给 `site-specofic` 目录前缀到 `python` 文件安装平台之前，默认是’/usr/local’，这可以通过设置 `configure` 脚本`--exec-prefix` 参数被设置编译时间，特别是所有的配置文件(像 `pyconfig.h` 头文件)被安装子啊 `exec_prefix/lib/pythonX.Y/config` 和共享库模块被按转子啊 `exec_prefix/lib/pythonX.Y/lib-dynload`，这里 `X,Y` 代表跑一趟好哦那得版本。
- `sys.executable`: 给 `python` 解释器一个绝对路径字符串，如果 `python` 不能获得真是的执行路径，`sys.executable` 将为 `None`。
- `sys.exit([arg])`: 从 `python` 推出，`SystemExit` 异常时生成，选项参数可以被给定为整数(默认为 0)或者其它对象类型。如果时一个整数，0 被认为成功终止，任何非零数值被认为异常终止。多数系统要求值在 0-127 之间，否则将产生不确定结果，一些系统约定指定推出代码，但是通常不完善，Unix 程序生成用 2 代表命令行语法错误 1 代表其它错误，如果另一类型的对象被传递，`None` 相当于传递 0，其他队像被打印到 `stderr` 和推出代码为 1，类似的 `sys.exit("some error message")` 是当程序出错一个快速退出程序的方法。因此 `exit()` 当从主进程退出进程时产生异常。异常不被拦截。
- `sys.flags` 结构序列 `flags` 暴露命令行状态

attribute	flag
debug	-d
inspect	-i
interactive	-i
optimize	-O or -OO
dont_write_bytecode	-B
no_user_site	-s
no_site	-S
ignore_environment	-E
verbose	-v
bytes_warning	-b
quit	-q
hash_randomization	-R

- sys.float\_info: 一个结构序列保持 float 类型的信息，它包含精确度和内部表达式低级信息，值符合在头文件 float.h 中定义的浮点常数。

attribute	float.h macro	explanation
epsilon	DBL_EPSILON	1 和大于 1 的最新值之间的差作为浮点数
dig	DBL_DIG	浮点数能带秒的最大精度
mant_dig	DBL_MANT_DIG	浮点精度。base-radix 浮点数的精度
max	DBL_MAX	有限浮点数的最大值
max_exp	DBL_MAX_EXP	radix**e-1 代表的最大整数 e 代表无穷浮点数
max_10_exp	DBL_MAX_10_EXP	最大 e10**2 代表的最大浮点
radix	FLT_RADIX	指数表达式的基数
rounds	FLT_ROUNDS	整数常数代表 round 模式，这反映了系统在解释器启动时 FL

属性 sys.float\_info.dig 需要更进一步扩展，如果 s 时任何字符串表达一个十进制数，然后转换 s 为浮点数将恢复一个字符串表达式。

```
import sys
sys.float_info.dig
15
s = '3.14159265358979'      # decimal string with 15 significant digits
format(float(s), '.15g') # convert to float and back -> same value
'3.14159265358979'
```

但是对于字符串 sys/float\_info.dig 指定精度，这不总是 true。

```
s = '9876543211234567'      # 16 significant digits is too many!
format(float(s), '.16g') # conversion changes value
```

'9876543211234568'

- sys.float\_repr\_style: 指示 repr() 函数如何处理入店时的字符串。日国字符串有一个值'short' 然后对于一个有限的浮点数 x, repr(x) 产生一个短字符串 float(repr(x)) == x。否则 float\_repr\_style 有值'legacy' 和 repr(x) 行为正如子啊 python3.1 中的一样。
- sys.getalloctedblock(): 返回解释器当前分配的内存块数量, 这个函数在更重和调式内存泄漏时很有用, 因为解释器内部换传, 结果可能因为调用而不同, 你也许可以调用 \_clear\_type\_cache() 和 gc.collect() 得到雨鞋结果。如果 python 编译实现不能合理的计算这些信息, getalloctedbloacks() 允许返回 0。
- sys.getdefaultencoding(): 返回 Unicode 实现的字符串的默认编码的名字。
- sys.getdlopenflags(): 同 dlopen() 返回当前 flag 的值。flag 值的符号名字能被在 os 模块中找到
- sys.getfilesystemencoding(): 返回用于转换 Unicode 文件名和 bytes 文件名的编码名字, 为了最好的兼容性, str 应该在所有情况下被用在 filename, 尽管文件名作为 bytes 被支持, 函数接受 fanti 文件名应该支持 str 或者 butes 内部转换系统偏好的表达。编码总是兼容 ASCII os.fsencode() 和 os.fsdecode() 应该被用于保证正确的编码和错误的模型使用。
  - 在 MAC OS 上编码为 utf-8
  - Unix 编码时 locale 编码
  - 在 windows 上编码也许是'utf-8' 或者是'mbcs', 依赖于用户配置。
- sys.getfilesystemencodererrors(): 返回转换 unicode 文件名和 bytes 文件名错误模式的名字, 编码名字有 getfilesystemencoding() 指定的便阿妈名字。os.fsencode() 和 os.fsdecode() 用来确保争取的编码和错误模式使用。
- sys.getrecount(object): 返回 object 对象的引用返回的储量通常高于你认为的呀, 因为它包含临时引用作为 getrefcount() 的参数。
- sys.getsizeof(object,[,default]): 返回对象的比特大小, 对象可以使任何类型的对象, 所有内建的多项将返回争取的结果, 但是这没有保持新的第三方扩展作为实现。仅仅内存消耗直接属性到对象, 对象访问时没有内存消耗。如果内定默认将返回不提供均值到这个值, 否则, TypeError 将产生。getsizeof() 调用对象的 \_\_sizeof\_\_ 方法, 如果对象通过垃圾回收器管理增加一个额外的垃圾回收器。

- `sys.getrecursionlimit()`: 返回循环限制的当前值，最大的 python 解释器栈深度。这限制阻止由无限循环从 c 栈移除和 python 崩溃，它可以被 `setrecursionlimit()`。
- `sys.getsizeof(object,[,default])`: 返回对象的比特大小，对象可以使任何类型，所有内建的兑奖将被正确返回，但是不是必须保持 true 给第三方扩展当它的实现被指定，仅仅仅对象的直接内存消耗属性，不是独享引用的内存消耗。如果对象没有给定获取大小，默认将被返回，否则 `TypeError` 将被报出。
- `sys.getwitchinterval()` 返回解释器的线程交换区间。
- `sys.sys._getframe([depth])`: 从调用的栈返回一个帧对象，如果宣讲整数 `depth` 被给定，返回栈顶下的帧对象调用。如果 `depper` 比调用的栈深，`ValueError` 被报出。默认深度为 0，返回调用栈顶的帧。
- `sys.getprofile()`: 获取 `setprofile()` 设置的 profile 函数。
- `sys.gettrace()`: 得到 `settrace()` 的 trace 函数。
- `sys.getwindowsversion()`: 返回一个描述当前 windows 版本的描述的名字元组。命名元素时 `major,minor,build,platform,service_pack_minor,service_pack_major,suit_mask,product_type` 和 `platform_version.service_pack` 包含一个字符串，`platform_version` 一个三元组和所有其它值。这个组建可以同感 `name` 访问，因此 `sys.getwindowsversion()[0]` 被等同
 

(VER_NT_WORKSTATION)	系统是工	
platform 将被 2(VER_PLATFORM_WIN32_NI)	(VER_NT_DOMAIN_CONTROLLER)	是系统是
	(VER_NT_SERVER)	服

 函数包装 `WIN32 GetVersionEx()` 函数，windows 可用
- `sys.get_asyncgen_hooks()`: 返回一个类似名称元组的 `asyncgen_hooks` 对象，这里 `firstier` 和 `expected` 均可设为 `None` 或者获取异步生成器作为参数的函数，通过时间循环调度异步生成器终止。
- `sys.get_coroutine_wrapper()`: 返回 `None` 或者一个由 `set_coroutine_wrapper` 包装器。

- sys.has\_info: 数值 hash 实现参数给一个结构序列。
- |           |                                     |
|-----------|-------------------------------------|
| width     | hash 值的位宽                           |
| modulus   | 用于数值 hash 方案的主要模块 P                 |
| inf       | 返回正无穷大的 hash 值                      |
| nan       | 返回非数的 hash 值                        |
| imag      | 返回复数的虚部                             |
| algorithm | str,bytes 和 memoryview 的 hash 算法的实现 |
| hash_bits | hash 算法的内部输出大小                      |
| seed_bits | hash 算法的种子值                         |
- sys.hexversion: 单个证书的编码版本。这被保证增加，包括合适的 support non-product release 版本，例如。测试 Python 解释器时最新的版本 1.5.2 用

```

if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...

```

- sys.
- sys.
- sys.
- sys.
- sys.



## Chapter 7

url



# Chapter 8

## requests

### 8.1 快速上手

#### 8.1.1 发送请求

```
import requests
r = requests.get('https://github.com/timeline.json')
r = requests
```

#### 8.1.2 requests 库的 7 个主要方法

requests.request()	够找一个请求，支持一下各方法的基础方法
requests.get()	获取 HTML 网页的主要方法，对应于 HTTP 的 GET
requests.head()	获取 HTML 网页头信息的方法，对应于 HTTP 的 HEAD
requests.post()	向 HTML 网页提交 POST 请求的方法，对应于 HTTP 的 POST
requests.put()	像 HTML 网页提交 PUT 请求的方法，对应于 HTTP 的 PUT
requests.patch()	像 HTML 网页提交局部修改请求，对应于 HTMP 的 PATCH
requests.delete()	像 HTML 网页提交删除请求，对应于 HTTP 的 DELETE

requests.get(url,params=None,\*\*kwargs)

- url: 想要获取的网页的 url 链接。
- params:url 中额外的参数，字典或字节流格式，可选
- \*\*kwargs:12 个控制访问的参数。

### 8.1.3 request 对象的属性

属性	说明
r.status_code	HTTP 请求的返回状态, 200 表示连接成功, 404 表示失败
r.text	HTTP 响应内容的字符串形式, 即, url 对应的页面内容
r.encoding	从 HTTP header 中猜测响应的内容编码方式
r.apparent_encoding	从内容分析出的响应内容编码方式 (备选编码方式)
r.content	HTTP 响应内容的二进制形式

### 8.1.4 理解 encoding 和 apparent\_encoding

r.encoding: 从 HTTp header 中猜测的响应内容编码方式, 如果 header 中不存在 charset, 则认为编码为 ISO-8859-1 r.text 根据 r.encoding 显示网页内容

r.apparent\_encoding: 根据网页内容分析出的编码方式, 可以看做是 r.encoding 的备选。

### 8.1.5 理解 Requests 库的异常

异常	说明
requests.ConnectionError	网络链接错误异常, 如 DNS 查询时白, 拒绝链接
requests.HEEPError	HTTP 错误异常
requests.URLRequired	URL 缺失异常
requests.TooManyRedirects	超过最大重定向次数, 产生重定向异常
requests.ConnectTimeout	连接远程服务器超时异常
requests.Timeout	请求 URL 超时, 产生超时异常
requests.raise_for_status()	如果不是 200, 产生异常, requests.HTTPError

r.raise\_for\_status() 方法在内部判断 r.status\_code 是否等于 200, 不需要额外加 if 语句, 该语句便于利用 try-except 进行异常处理。

### 8.1.6 HTTP 协议

HTTP:Hypertext Transfer Protocol 超文本传输协议。  
HTTP 是一句“请求与响应”模式的, 武装到的应用层协议, HTTP 协议采用 URL 定位网

络资源的标志， URL 格式如下：

http://host[:port][path]  
 host: 合法的 Internet 主机域名和 IP 地址。  
 port: 端口号, 缺省端口为 80  
 path: 请求资源的路径

### HTTP 协议对资源的操作

方法	说明
GET	请求获取 url 位置的资源
HEAD	请求获取 URL 位置资源的响应消息报告, 即获得该资源的头部信息
POST	请求像 URL 位置的资源后附加新的数据
PUT	请求 URL 位置存储一个资源, 覆盖原 URL 位置的资源
PATCH	请求局部更新 URL 位置的资源, 即改变该处资源的部分内容
DELETE	请求删除 URL 位置存储的资源

#### head 方法的使用

```
In [3]: r = requests.head('http://www.baidu.com')
In [4]: r
Out[4]: <Response [200]>
In [5]: r.headers
Out[5]: {'Server': 'bfe/1.0.8.18', 'Date': 'Tue, 08 Aug 2017 11:46:32 GMT', 'Content-Type': 'text/html', 'Last-Modified': 'Mon, 13 Jun 2016 02:50:04 GMT', 'Connection': 'Keep-Alive', 'Cache-Control': 'private, no-cache, no-store, proxy-revalidate, no-transform', 'Pragma': 'no-cache', 'Content-Encoding': 'gzip'}
```

#### post 方法的使用 (像 URLPOST 一个表单, 自动编码为 form)

```
In [10]: payload = {'key1': 'value1', 'key2': 'value2'}
In [11]: r = requests.post('http://httpbin.org/post', data=payload)
In [12]: print(r.text)
{...
 "form": {"key2": "value2",
          "key1": "value1"}
```

```
    } .  
}
```

### put 方法

```
In [18]: r = requests.put('http://httpbin.org/put', data = payload)
```

```
In [19]: print(r.text)  
{  
    "args": {},  
    "data": "",  
    "files": {},  
    "form": {  
        "key1": "value1",  
        "key2": "value2"  
    },  
    "headers": {  
        "Accept": "*/*",  
        "Accept-Encoding": "gzip, deflate",  
        "Connection": "close",  
        "Content-Length": "23",  
        "Content-Type": "application/x-www-form-urlencoded",  
        "Host": "httpbin.org",  
        "User-Agent": "python-requests/2.18.3"  
    },  
    "json": null,  
    "origin": "210.47.0.232",  
    "url": "http://httpbin.org/put"  
}
```

```
requests.request(method,url,**kwargs)  
**kwargs 控制访问参数
```

params	字典或字节序列，作为参数增加到 url 中
data	字典，字节序列或文件对象，作为 Request 的内容
json	JSON 格式的数据，作为 Request 的内容
header	字典,HTTP 定制头
cookies	字典或 CookieJar，Request 中的 cokkie
auth	元组，支持 HTTP 认证功能
file	字典类型，传输文件
timeout	设定草食时间，s 为单位。
proxies	字典类型，设定访问代理服务器，可以增加登录认证
allow_redirects	:True/False, 默认为 True，重定向开关
stream	True/False，默认为 True，获取内容立即下载开关
verify	True/False, 默认为 True，认证 SSL 整数开关
cert	本地 SSL 整数路径



# Chapter 9

## Tensorflow API

### 9.1 tf.squeeze

tf.squeeze(input, axis=None, name=None, squeeze\_dims=None) 说明: 从指定的 Tensor 中移除 1 维度。

- input: tensor, 输入 Tensor。
- axis: 列表, 指定需要移除的位置的列表, 默认为空列表 [], 索引从 0 开始 squeeze 不为 1 的索引会报错。
- name: caozuoide 名字
- squeeze\_dims: 否决当前轴的参数。
- 返回一个 Tensor, 形状和 input 相同, 包含和 input 相同的数据, 但是不包含有 1 的元素。
- 异常: squeeze\_dims 和 axis 同时指定时会有 ValueError。

```
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
shape(squeeze(t)) ==> [2, 3]
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
shape(squeeze(t, [2, 4])) ==> [1, 2, 3, 1]
```

### 9.2 tf.stack

stack(values, axis=0, name='stack'): stack 一个 n 维 tensor 为 n+1 维 tensor。给定一个长度为 N 的形状为 (A,B,C) 的 tensor, 如果 axis==0 输出 tensor 的形状为 (N,A,B,C), 如

果 axis==1, 输出 tensor 的形状为 (A,N,B,C) # 'x' is [1,4]

```
# 'y' is [3,6]
# 'z' is [3,6]
stack([x,y,z])=>[[1,4],[2,5],[3,6]]
stack(x,y,z,axis=1)=>[[1,2,3],[4,5,6]]
tf.stack([x,y,z]) = np.asarray([x,y,z])
```

参数:

- 一个 Tensor 列表。
- 整数， 默认为 0, 支持负坐标。
- 操作的名字。

S 一个 stack 的 Tensor。

S ValueError: 如果 axis 超过 [-R+1,R+1)

#### Example

```
import tensorflow as tf
x = tf.constant([1,4])
y = tf.constant([2,5])
z = tf.constant([3,6])
r1 = tf.stack([x,y,z])
r2 = tf.stack([x,y,z],axis=1)
with tf.Session() as sess:
    print(sess.run(r1).shape)
    print(sess.run(r2).shape)
```

### 9.3 tf.metrics

accuracy(labels,predictions,weights=None,metrics\_collections,updates\_collections=None,name=None)

- labels:tensor, 和 predictions 的形状相同, 代表真实值。
- predictions:tensor, 代表预测值。
- weights:tensor, rank 可以为 0 或者 labels 的 rank, 必须能和 label 广播 (所有的维度必须是 1, 或者和 labels 维度相同)
- metrics\_collection:accuracy 应该被增加的一个 collection 列表选项。

- update\_collections:update\_op 应该添加的选项列表。
- name:variable\_scope 名字选项。
- accuracy: 返回值 tensor, 代表精度, 总共预测对的和总数的商。
- update\_op: 返回值适当增加 total 和 count 变量和 accuracy 匹配。
- valueerror: 异常如果 predictions 和 labels 有不同的形状, 或者 weight 不是 none 它的形状不合 prediction 匹配, 或者 metrics\_collections 会哦这 updates\_collections 不是一个 list 或者 tuple。

## 9.4 tf.reshape

```
tf.reshape(tensor,shape,name=None)
```

- Tensor: 一个 Tensor。
- shape: 一个列表, 数值类型为 int32 或者时 int64
- name: 操作的名字。

S 指定形状的 Tensor。

```
import tensorflow as tf
a = tf.linspace(0.,9.,10)
b = tf.reshape(a,[2,5])
with tf.Session() as sess:
    a = sess.run(a)
    b = sess.run(b)
print(a.shape)
print(b.shape)
```

## 9.5 tf.image

### 9.5.1 tf.image.decode\_gif

```
tf.image.decode_gif(contents,name=None)
```

- contents: 一个字符串 Tensor, GIF 编码的图像。
- name: 操作的名字。
- 返回一个 8 位无符号的 Tensor, 四维形状为 [num\_frames,height,width,3], 通道顺序是 RGB。

### 9.5.2 tf.image.decode\_jpeg

```
tf.image.decode_jpeg(contents,channels=None,ratio=None,fancy_upscaling=None,  
try_recover_truncated=None,acceptable_fraction=None,dct_methed=None,name=None)  
解码 JPEG 编码的图像为无符号的 8 位整型 tensor。
```

- contents: 一个字符串 tensor, JPEG 编码的图像。
- channels: 一个整数默认为, 0 代表编码图像的通道数 (JPEG 编码的图像), 1 代表灰度图, 3 带秒 RGB 图。
- ratio: 一个整数, 默认为 1, 取值可以是 1,2,4,8, 表示缩减图像的比例。
- fancy\_upscaling:bool 型, 默认为 True, 表示用慢但是更好的提高色彩浓度。
- try\_recover\_truncated:bool 型, 默认是 False, 如果时 True 尝试从截断的输入恢复图像。
- acceptable\_fraction:float 型, 默认是 1, 可接受的最小的截断输入的因子。
- dct\_methed:string 类型, 默认为 “”. 指定一个解压算法, 默认是 “” 由系统自行指定。可用的值有 [“INTEGER\_FAST”, “INTEGER\_ACCURATE”]
- name: 操作的名字。
- 返回值为一个 8 位无符号整型 Tensor, 3 维形状 [height,width,channels]

## 9.5.3 tf.image.encode\_jpeg

`tf.image.encode_jpeg(image,format=None,quality=None,progressive=None,optimize_size=None,chroma_downsampling=None,density_uint=None,x_density=None,y_density=None,xmp_metadata=None,`

- `image`: 一个 3 维 [height,width,channels]，8 位无符号整型 Tensor。
- `format:string` 类型，可以为 "", "grayscale", "rgb"，默认为 ""。如果 `format` 没有指定或者不为空字符串，默认格式从 `image` 的通道中选，1: 输出灰度图，3: 输出 RGB 图。
- `quality`: 整型，默认值为 95，代表压缩质量值 [0,100]，值越大越好，单速度越慢。
- `optimize_size:bool` 型，默认为 False，如果为 True 用 CPU/RAM 减少尺寸同时保证质量。
- `chroma_downsampling:bool` 型，默认为 True。
- `density_unit`: 一个字符串，可以为 "in", "cm"，指定 `x_density` 和 `y_density.in` 每 inch 的像素，cm 表示每厘米的像素。
- `x_density`: 一个整数，默认为 300，每个 density 单位的水平像素。
- `y_density`: 一个整数，默认为 6300，数值方向上每 density 单位的像素。
- `xmp_metadata:string` 类型，默认为 ""，如果为空，嵌入 XMP metadata 到图像头部。
- `name`: 操作的名字。
- `name`: 操作的名字。
- 返回 0 维字符串型 JPEG 编码的 Tensor。

## 9.5.4 tf.image.decode\_png

`tf.image.decode_png(contents,channels=None,dtype=None,name=None)` 解码 PNG 编码的图像为 8 位或者 16 位无符号整型 Tensor。

- `contents`: 一个 0 维 PNG 编码的图像的字符串的 Tensor。
- `channels`: 整型默认为 0，代表解码图像的通道，0 用 PNG 编码图像数，1: 代表输出灰度图像。3: 代表输出 RGB 图像。4: 代表输出 RGBA 图像。
- `dtype:tf.DType`, 值可以为 `tf.uint8,tf.uint16`, 默认为 `tf.uint8`。
- `name`: 操作的名字。
- 返回 3 维 [height,width,channels] 的 Tensor。

### 9.5.5 tf.image.encode\_png

```
tf.image.encode_png(image,compression=None,name=None)
```

- 一个 8 位或者 16 位的 3 维 Tensor, 形状为 [height,width,channels]
- compression: 一个整数, 默认为-1, 表示压缩等级。
- name: 操作的名字。
- 返回一个 0 维 string 型的 PNG-encoded 的 Tensor。

### 9.5.6 tf.image.decode\_image

```
tf.image.decode_image(contents,channels=None,name=None)
```

- contents: 0 维编码图像的字符串。
- channels: 整数, 默认为 0, 解码图像的通道数。
- name: 操作的名字。
- 返回 JPEG,PNG 的 8 位无符号的形状为 [height,width,num\_channels], GIF 文件的形状为 [num\_frames,height,width,3]
- ValueError: 通道数不正确。

### 9.5.7 tf.image.resize\_images

```
tf.image.resize_images(images,size,method=ResizeMethod.BILINEAR,align_corners=False)
```

- images: 形状为 [batch,height,width,channels]4 维 Tensor,3 为 Tensor, 形状为 [height,width,channels]
- size: 一位 32 整型 Tensor 元素为 new\_height,new\_width, 新的图像尺寸。
- method: ResizeMethod, 默认为 ResizeMethod.BILINEAR
  - ResizeMethod.BILINEAR: 二进制插值。
  - ResizeMethod.NEAREST\_NEIGHBOR:
  - ResizeMethod.BICUBIC:
  - ResizeMethod.AREA:
- align\_corners:bool 型, 如果为真提取对齐四个角, 默认为 False。

- 异常
  - ValueError: 图像形状和函数要求的不一样。
  - ValueError:size 是不可用的形状或者类型。
  - ValueError: 指定的方法不支持。
- 如果图像时 4 维 [batch,new\_height,new\_height,channels], 如果图像是 3 维, 形状为 [new\_height,new\_width,channels]