

目录

1	Tensorflow 基础	3
1.1	Tensorflow 基础函数	3
1.1.1	Variable	3
1.1.2	placeholder	3
1.1.3	batch normalization	4
1.1.4	常见的的激活函数	4
1.2	relu 函数	5
1.2.1	relu	5
1.2.2	relu6	5
1.2.3	sigmoid	7
1.2.4	relu 和 softplus	8
1.2.5	dropout	10
1.3	卷积函数	10
1.4	池化	11
1.5	常见的分类函数	12
1.6	优化方法	12
1.6.1	BGD	13
1.6.2	SGD	13
1.6.3	momentum	13
1.6.4	Nesterov Momentum	13
1.6.5	Adagrad	14
1.6.6	RMSprop	14
1.6.7	Adam	14
1.6.8	构造简单的神经网络拟合数据	16
1.7	TensorBoard	17
1.8	CNN 手写体数据识别	21

2	目录
1.8.1	mnist 数据集 21
2	Tensorflow 进阶 25
2.1	模型存储和加载 25
2.2	用 GPU 26
2.2.1	手工配置设备 26
2.2.2	允许 GPU 的内存增长 27
3	常用的 python 模块 29
3.1	Argparse 29
3.1.1	ArgumentParser 对象 30
3.1.2	prog 30
3.1.3	add_argument() 方法 35
3.2	path 63
4	re 67
4.1	正则表达式介绍 67
4.2	RE 库的主要功能函数 69
5	sys 73
6	url 81
7	requests 83
7.1	快速上手 83
7.1.1	发送请求 83
8	Tensorflow API 85
8.1	tf.squeeze 85
8.2	tf.stack 85
8.3	tf.metrics 86
8.4	tf.reshape 87
8.5	tf.image 88
8.5.1	tf.image.decode_gif 88
8.5.2	tf.image.decode_jpeg 88
8.5.3	tf.image.encode_jpeg 89
8.5.4	tf.image.decode_png 89
8.5.5	tf.image.encode_png 90

目录	3
8.5.6 tf.image.decode_image	90
8.5.7 tf.image.resize_images	90

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:

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

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

$$(1.3)$$

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

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

1.1.4 常见的的激活函数

- 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('relu6(x)')
plt.savefig('relu6.png',dpi = 600)

```

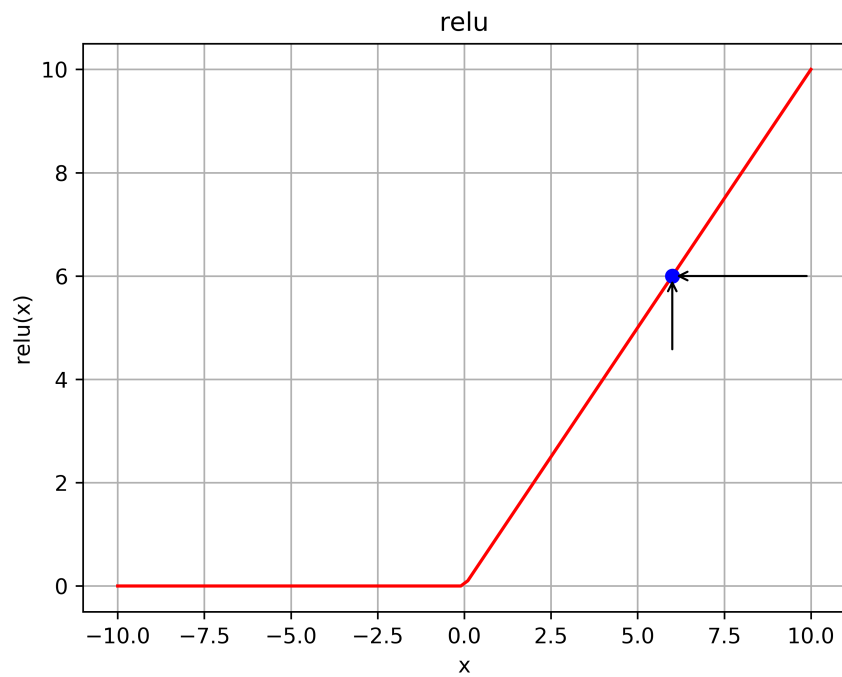


图 1.1: relu

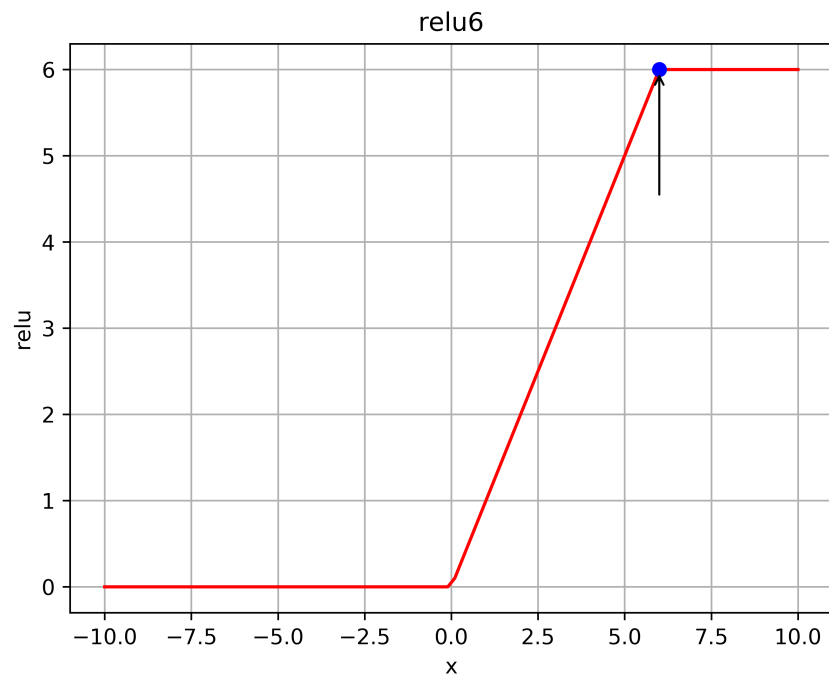


图 1.2: relu6

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"$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-x}}$",
            xy=(0,0), xycoords="data",
            xytext=(1,0), textcoords="data",
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3"),
            )
ax.annotate(r"$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$",
```

```

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)

```

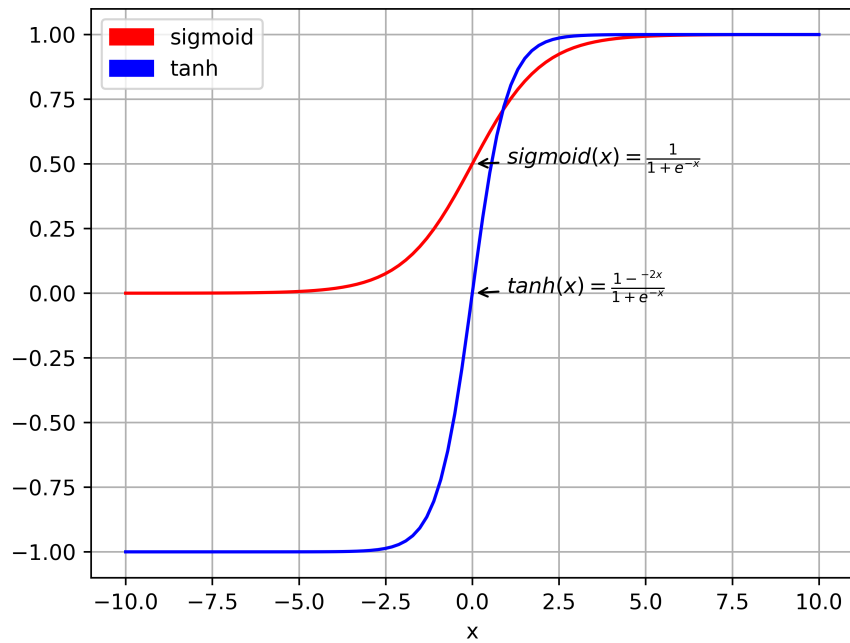


图 1.3: activate_fun

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')

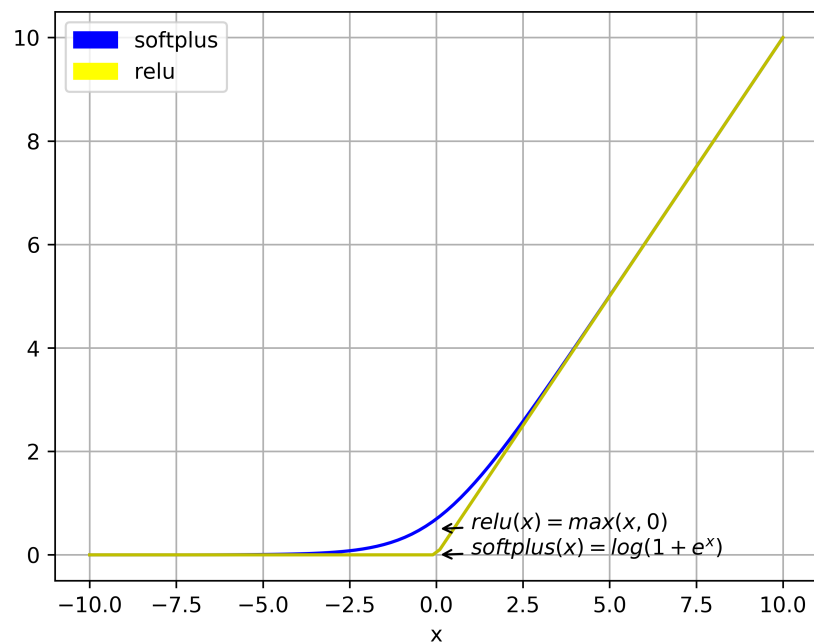
```

```

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"$\text{softplus}(x)=\log(1+e^x)$",
            xy=(0,0),xycoords="data",
            xytext=(1,0),textcoords="data",
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3"),
            )
ax.annotate(r"$\text{relu}(x)=\max(x,0)$",
            xy=(0,0.5),xycoords="data",
            xytext=(1,0.5),textcoords="data",
            arrowprops=dict(arrowstyle="->",
                            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
④ tf.nn.avg_pool3d(input,ksize,strides,padding,name =None)	三维状态下
⑤ tf.nn.max_pool3d(input,ksize,strides,padding,name =None)	三维状态下
⑥ tf.nn.fractional_avg_pool(value,pooling_ratio,pseudo_random=None,overlapping=None,seed1=None,seed2=None,name = None)	三维下的平均池化
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
⑧ tf.nn.pool(input>window_shape,pool_typing,padding,dilation_rate = None,strides=None,name=None,data_format=None)	执行一个 N

- value: 一个四维 Tensor，维度为 [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

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

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

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

`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

Adagrad 能够自适应的为每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。

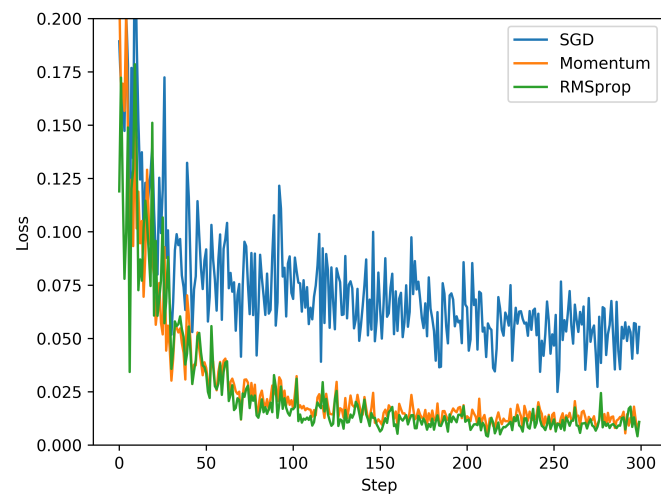
1.6.7 Adam

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。

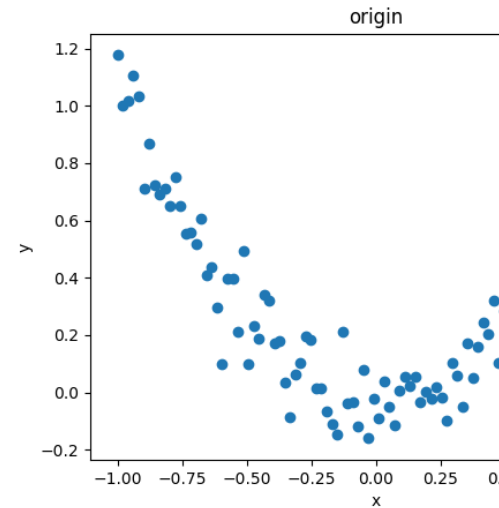
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
tf.set_random_seed(0)
np.random.seed(0)
LR = 0.01
BATCH_SIZE = 32
x = np.linspace(-1,1,100).reshape(-1,1)
noise = np.random.normal(0,0.1,size=x.shape)
y = np.power(x,2)+noise
class Net:
    def __init__(self,opt,**kwargs):
        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_hist in zip(nets, losses_hist):
    _, l = sess.run([net.train, net.loss], {net.x: b_x, net.y: b_y})
    l_hist.append(l)
labels = ['SGD', 'Momentum', 'RMSprop', 'Adam']
for i, l_hist in enumerate(losses_hist):
    plt.plot(l_hist, 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)

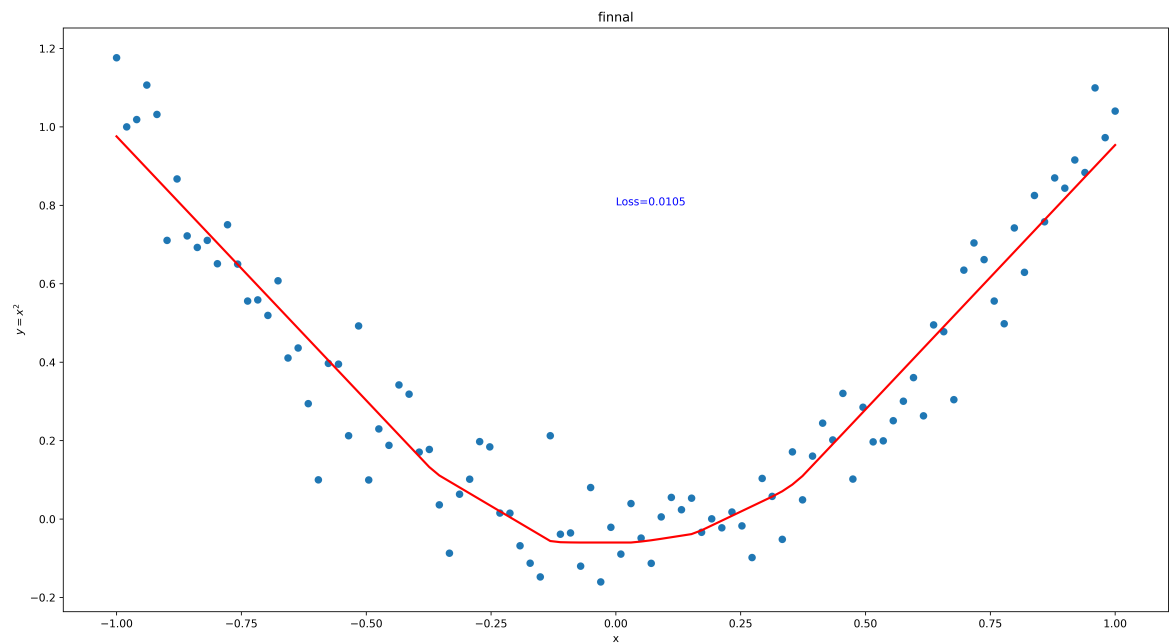
sess = tf.Session()
sess.run(tf.global_variables_initializer())
plt.ion()
for step in range(100):
    _,l,pred = sess.run([train_op,loss,output],[tf_x:x,tf_y:y])
```

```

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=%.4f'%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
                                           local variables
        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)

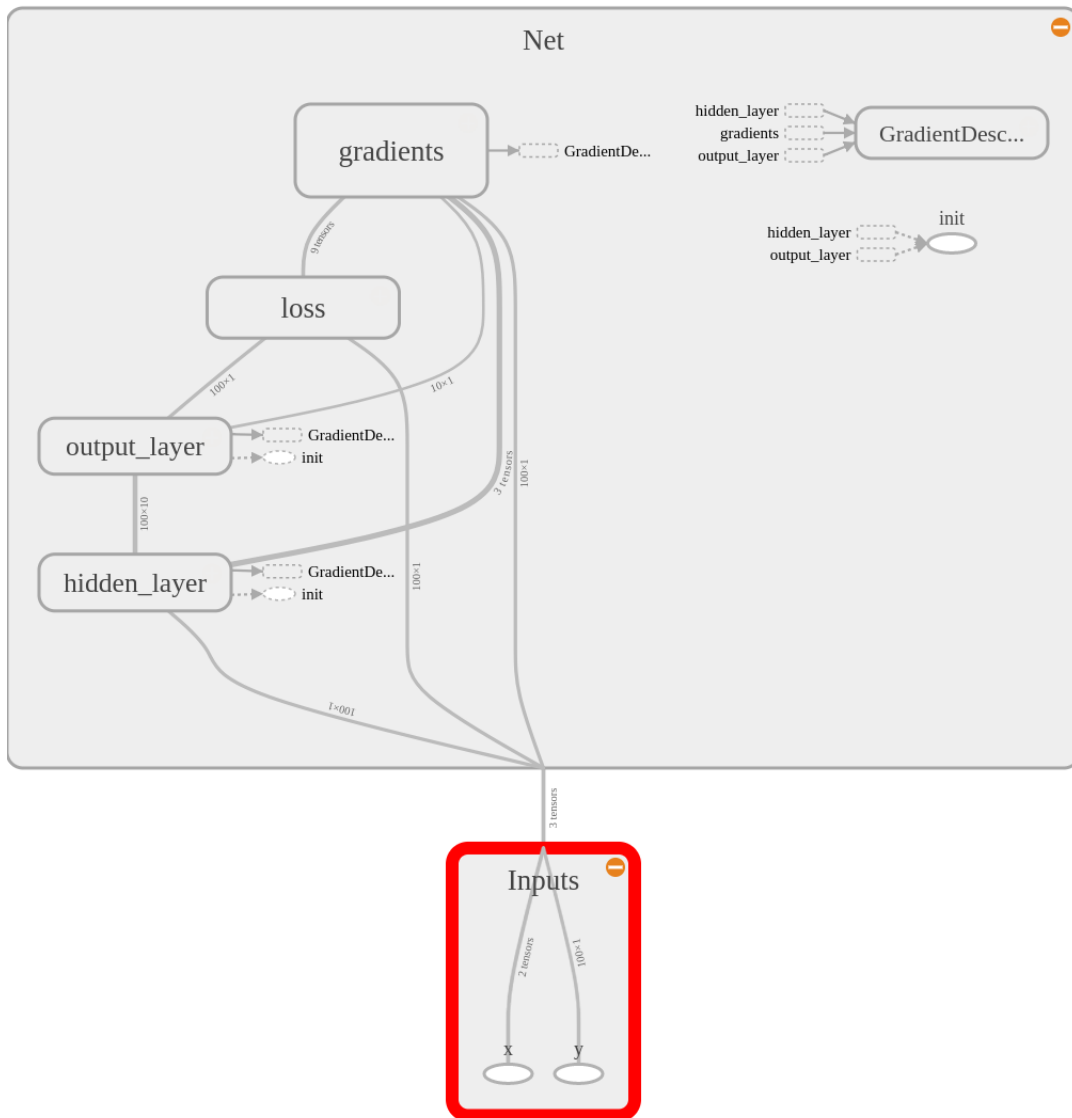
sess = tf.Session()

                                           # 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()

```



```
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=%0.4f'%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=%0.4f'%l, fontdict={'size': 15, 'color': 'red'})
    plt.show()
save()
tf.reset_default_graph()
reload()

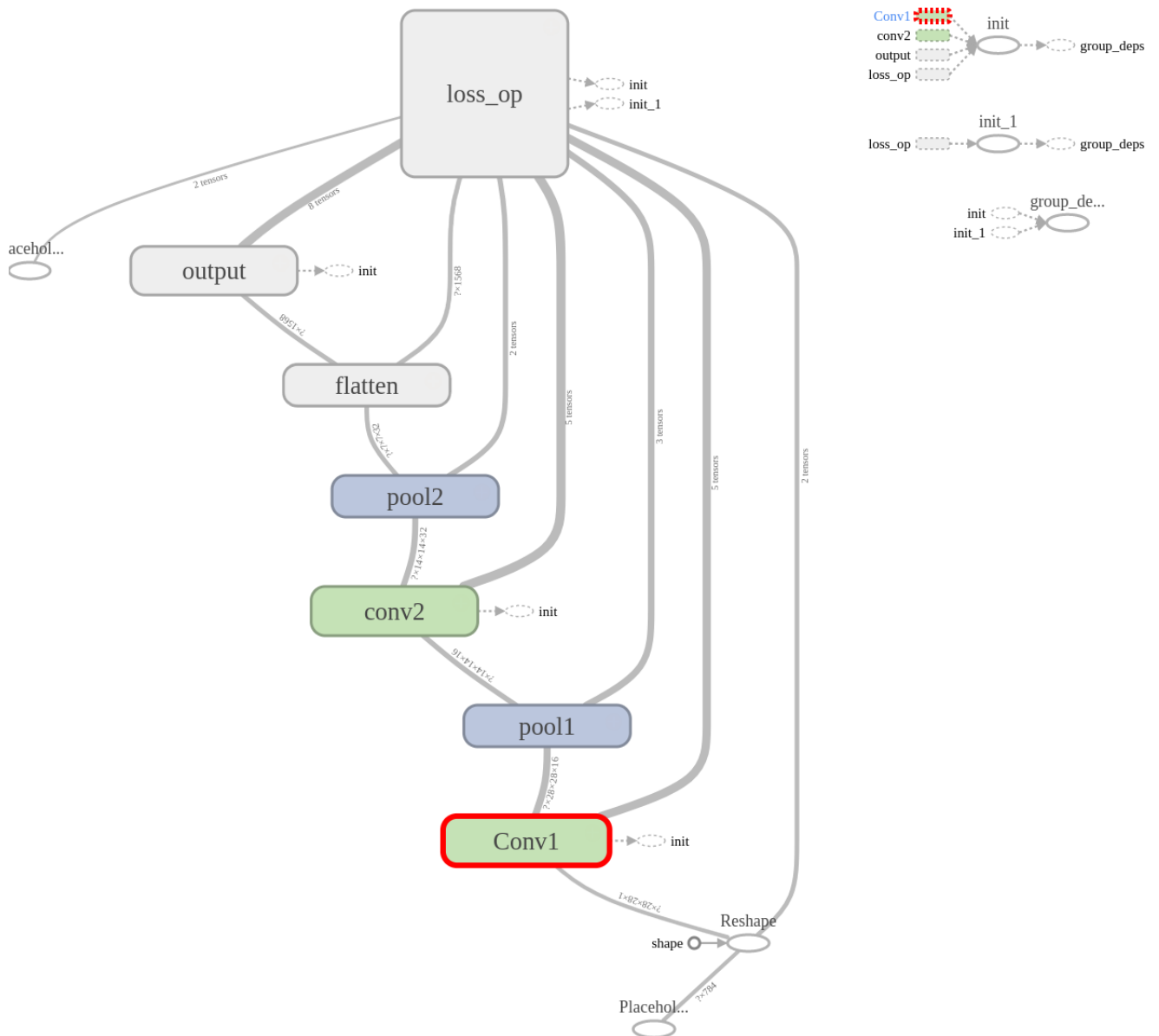
```

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 28×28 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 5×5 , 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 5×5 , 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_, '| test accuracy: %.2f' %
              accuracy_)
test_output = sess.run(output, {tf_x: test_x[:10]})
pred_y = np.argmax(test_output, 1)
```

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

常用的 python 模块

3.1 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，argprser 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

3.1.1 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: 自定义输出帮助类。
- prefix_chars: 参数前面的字符。(默认为 '-')
- fromfile_prefix_chars: 应该被读的文件字符串。
- 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)

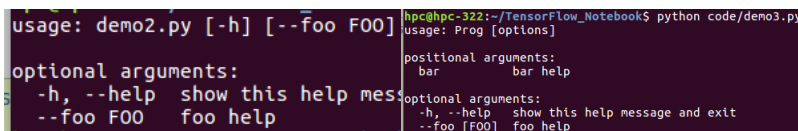
3.1.2 prog

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。

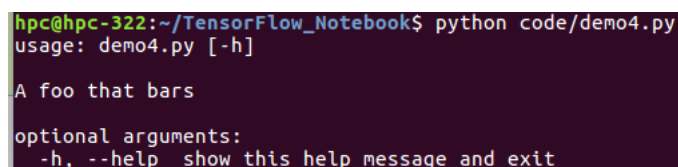
```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

(a) name of the subfigure
(b) name of the subfigure

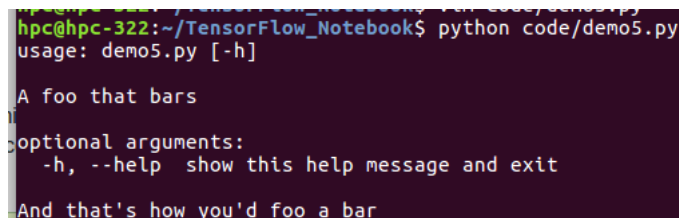
大多数的 `ArgumentParser` 构造体用 `description=` 关键字，这个参数给出一个简单的程序说明其如何工作的。在帮助信息中表述在命令行和帮助信息之间。

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



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()
```



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 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 wierd
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_Notebooks$ python code/demo/.py
usage: PROG [-h]

this description was indented wierd 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 F00] [bar [bar ...]]

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

optional arguments:
  -h, --help  show this help message and exit
  --foo F00   F00 (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/foo 就可以设置 prefix_chars= 参数指定。prefix_chars 默认默认为-, 用非-字符能禁用-f/-foo 这种类型性的选项。

```
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_c 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 `'resolve'` 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 F00] [--foo F00]

optional arguments:
  -h, --help  show this help message and exit
  -f F00      old foo help
  --foo F00   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 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
```

3.1.3 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: 传递参数读额数据类型。
- choices: 参数的允许值的容器。
- required: 是否命令行选项被乎略。
- 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,abr=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)s 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', 'b'], 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=int)
parser.parse_args()
```

输出: Namesapce(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, choice=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')

choise 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 choice 作为值, 因此 dict,set 对象都是常用的支持的对象。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 frobbled')
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)s)')
parser.print_help()
```

输出:

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

positional 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 值是大写的。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(parser.print_help())
```

分别输出:

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

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

positional 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 parse_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 meiyoiu 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 zuoiveiaction 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.parse_args() 返回的 Namespace 对象, 大多数行为通过 setattr() 增加一个属性到对象。
- value: 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` 可能有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` 没有选项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 -l: 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 `parse_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.args`

`ArgumentParser` 通常比 `sys.argv` 有用，可以穿地一个字符串列表到 `parse_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, choice=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.parse_args()` 返回的 `durian` 将包含住 `parser` 和 `subparser` 命令行选中的参数，因此上面的例子中，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 `discription` 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 subparse 的名字，`dest` 关键值通过 `add_argparsers()` 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', type=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 将确保惟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` 是 `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, `parse_known_args()` 方法很有用, 它像 `parse_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当空格分开的 word 为参数，下面的例子展示：

```

class NyArgumentParser(argparse.ArgumentParser):
    def convert_arg_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.parse_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
```

`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`。

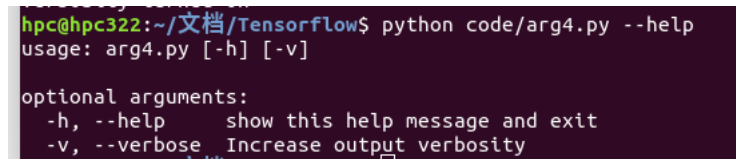
```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity", action="store_true")
```

```
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.verbose` 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", "--verbosity", type=int, 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)

```

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。这只默认为值 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：

```
#argsl0.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 argl0.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。

3.2 path

- `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, 第二个元素同感传递 `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, 且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` 分割后的第一个元素。
- `os.path.exists(path)`: 如果存在路径 `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 `$name` 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访问则报 `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访问将报出 `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)`: 删除冗余得分和服, 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, 如果以斜线将诒为, `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")` 返回 `("//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`。

Chapter 4

re

4.1 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	
[]	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
[^]	非字符集，对单个字符给出排除范围	[âbc]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
\d	数字等价与 [0-9]	
\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:

<code>^[A-Za-Z]+\$</code>	26 个字母组成的字符串
<code>^[A-Za-z0-9]+\$</code>	由 26 个字母和数字组成的字符串
<code>^-?\d+\$</code>	整数形式的字符串
<code>^[0-9]*[1-9][0-9]* \$</code>	正整数形式的字符串
<code>[1-9]\d5</code>	中国境内邮政编码，6 位
<code>[\u4e00-\u9fa5]</code>	匹配中文字符
<code>\d{3}-\d{8} \d{4}-\d{7}</code>	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])`

4.2 RE 库的主要功能函数

<code>re.search()</code>	在一个字符串搜索匹配正则表达式的第一个位置。
<code>re.match()</code>	从一个字符的开始为值起匹配正则表达式，返回 <code>match</code> 对象。
<code>re.findall()</code>	搜索字符串，以列表类型返回全部匹配的字符串
<code>re.split()</code>	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
<code>re.finditer()</code>	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 <code>match</code> 对象
<code>re.sub()</code>	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.search(pattern,string,flags=0)`: 在一个字符串中搜索匹配正则表达式的第一个位置返回 `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。

re.I	忽略正则表达式的大小写，[A-Z] 能够匹配小写。
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

```
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厚的字符串。

- 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个位置，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()	匹配字符串的将谗数位置
.span()	返回 (.start(),.end())

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

本章内容借鉴与北京理工大学的嵩天老师的 python 正则表达式相关内容。

Chapter 5

sys

- `sys.abiflags`: 在 POSIX 体系上 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_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 (虚拟环境将被创建)。
- `sys.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, 如果 `site.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 (虚拟环境将被创建)。
- `sys.byteorder`: 本地变量的指示器, 这将在 big-endian 平台有一个值 `'big'`, 在 little-endian 平台。
- `sys.modules`: 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.settrace(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特们的死锁一样长。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.excepthook(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.excepthook`。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.exeoohook`。

- `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-specific 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）被安装于啊 `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。如果是一个整数，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
<code>epsilon</code>	<code>DBL_EPSILON</code>	1 和大于 1 的最新值之间的差作为浮点数
<code>dig</code>	<code>DBL_DIG</code>	浮点数能带秒的最大精度
<code>mant_dig</code>	<code>DBL_MANT_DIG</code>	浮点精度。base-radix 浮点数的精度
<code>max</code>	<code>DBL_MAX</code>	有限浮点数的最大值
<code>max_exp</code>	<code>DBL_MAX_EXP</code>	$\text{radix}^{**}(\text{e}-1)$ 代表的最大整数 e 代表无穷浮点数
<code>max_10_exp</code>	<code>DBL_MAX_10_EXP</code>	最大 $\text{e}10^{**2}$ 代表的最大浮点
<code>radix</code>	<code>FLT_RADIX</code>	指数表达式的基数
<code>rounds</code>	<code>FLT_ROUNDS</code>	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 <code>sys.float_info.dig</code> 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.getallocatedblock()`: 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, `getallocatedbloacks()` 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既, 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.getwchinterval()` 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)
系统是服

 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。

- 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
 - hash_bits hash 算法的内部输出大小
 - seed_bits hash 算法的种子值
- sys.has_info: 数值 hash 实现参数给一个结构序列。

- sys.hexversion: 单个证书的编码版本。这被保证增加，包括合适的支持 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 6

url

Chapter 7

requests

7.1 快速上手

7.1.1 发送请求

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


Chapter 8

Tensorflow API

8.1 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]
```

8.2 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)
```

8.3 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 collectionn 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`。

8.4 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 `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)
```

8.5 tf.image

8.5.1 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。

8.5.2 tf.image.decode_jpeg

```
tf.image.decode_jpeg(contents,channels=None,ratio=None,fancy_upscaling=None,  
try_recover_truncated=None,acceptable_fraction=None,dct_method=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_method: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]

8.5.3 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 JPED 编码的 Tensor。

8.5.4 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: 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。

8.5.5 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。

8.5.6 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。

8.5.7 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樣。
 - ValueError:size 是不可以用的形状或者类型。
 - ValueError: 指定的方法不支持。
- 如果图像是 4 维 [batch,new_height,new_width,channels], 如果图像是 3 维, 形状为 [new_height,new_width,channels]