[Stanford CS20SI]

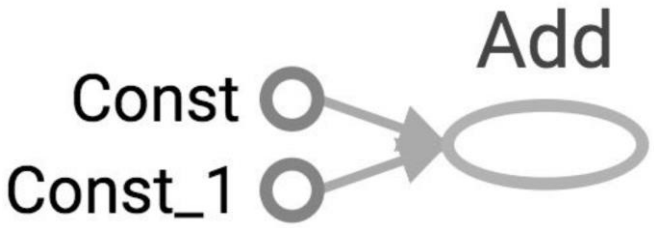
- Tensorflow 可视化——TensorBoard

```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)

with tf.Session() as sess:
        writer = tf.summary.FileWriter('./graphs', sess.graph)
        print sess.run(x)

# close the writer when you're done using it
writer.close()
```

```
$ python [yourprogram.py]
$ tensorboard --logdir="./graphs"
```



- 创建 Tensor 张量

  注意 tf 中张量是不可迭代的

  - tf.zeros()、tf.zeros_like()
  - tf.ones()、tf.ones_like()
  - tf.fill()
  - tf.linspace()、tf.range()
  - tf.random_normal() 、 tf.truncated_normal() 、 tf.random_uniform 、 tf.random_shuffle() 、 tf.random_crop()、tf.multinomial()、tf.random_gamma()

- 运算

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural network building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

- 数据类型（尽可能用 tf 类型）
  - Python 原生类型
    - 字符串、布尔、数值型都作为 0 维张量
    - 列表作为 1 维张量（向量）、列表的列表作为 2 维张量（矩阵）、依次类推
    - python 的数值长度不确定，tf 需要去推断具体的数据类型
  - Tf

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_UINT16	tf.uint16	16 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.
DT_QUINT8	tf.quint8	8 bits unsigned integer used in quantized Ops.

  - Numpy

    通常 tf 类型和 numpy 类型可交换使用
    - Numpy 类型没有 string
    - Numpy 不支持自动微分和 GPU 运算
- 变量
  - 创建：tf.Variable()
  - 使用前必须初始化
    - 初始化所有变量

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
        tf.run(init)
```

    - 初始化部分变量

```
init_ab = tf.variables_initializer([a, b], name="init_ab")
with tf.Session() as sess:
        tf.run(init_ab)
```

- 初始化单个变量

```
W = tf.Variable(tf.zeros([784,10]))
with tf.Session() as sess:
        tf.run(W.initializer)
```

- 获取变量的值：tf.Variable.eval()
- 为变量赋值：tf.Variable.assign()

  注意 tf.Variable.assign()只是描述了一个赋值 op，真正的执行需要 Session.run()
- 自增+=和自减-=

```
W = tf.Variable(10)

with tf.Session() as sess:
        sess.run(W.initializer)
        print sess.run(W.assign_add(10)) # >> 20
        print sess.run(W.assign_sub(2)) # >> 18
```

- 占位符、投喂字典

  声明占位符和 op：

```
# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # Short for tf.add(a, b)
```

如果直接 run 会出错，因为 a 还没有值：

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

>> NameError
```

可以在 run 的时候提供 feed_dict 对 a 指定值：

```
with tf.Session() as sess:
        # feed [1, 2, 3] to placeholder a via the dict {a: [1, 2, 3]}
        # fetch value of c
        print(sess.run(c, {a: [1, 2, 3]}))

>> [6. 7. 8.]
```

feed_dict 也可以用来覆盖替换图中的一个 op：

（如这里用值 15 替代了 add(2, 5)的 op）

```
# create Operations, Tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)

# start up a `Session` using the default graph
sess = tf.Session()

# define a dictionary that says to replace the value of `a` with 15
replace_dict = {a: 15}

# Run the session, passing in `replace_dict` as the value to `feed_dict`
sess.run(b, feed_dict=replace_dict) # returns 45
```

可用 tf.Graph.is_feedable(tensor)来判断一个 tensor 是否可被投喂

- Lazy loading

比较：

```
x = tf.Variable(10, name='x')
y = tf.Variable(20, name='y')
z = tf.add(x, y)

with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for _ in range(10):
                sess.run(z)
        writer.close()
```

和

```
x = tf.Variable(10, name='x')
y = tf.Variable(20, name='y')

with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for _ in range(10):
                sess.run(tf.add(x, y)) # create the op add only when you need to compute it
        writer.close()
```

前者只创建一个 add op，而后者会创建 10 个 add op

解决方法：

1. 尽可能拆分操作 op 及其执行
2. 用 property 修饰类方法保证它们只被加载一次

Structuring Your TensorFlow Models - Danijar Hafner

装饰器 Decorator – 廖雪峰

Python 进阶之“属性（property）”详解 – 伯乐在线

- 优化器会自动优化搜索各个变量 Variable 的最优解，如果有变量不需要优化，那么在声明时指定关键字参数 trainable=False 即可
- 把学习率作为优化的参数

```
global_step = tf.Variable(0, trainable=False, dtype=tf.int32)

learning_rate = 0.01 * 0.99 ** tf.cast(global_step, tf.float32)

increment_step = global_step.assign_add(1)

optimizer = tf.train.GradientDescentOptimizer(learning_rate) # learning rate can be a tensor
```

训练时可以直接最小化损失函数——

```
train_step = optimizer.minimize(loss)
```

也操作训练的过程——

```
# compute the gradients for a list of variables.

grads_and_vars = optimizer . compute_gradients ( loss ,  < list of variables >)

# grads_and_vars is a list of tuples (gradient, variable). Do whatever you

# need to the 'gradient' part, for example, subtract each of them by 1.

subtracted_grads_and_vars = [( gv [ 0 ]  -  1.0 ,  gv [ 1 ])  for  gv in  grads_and_vars]

# ask the optimizer to apply the subtracted gradients.

optimizer . apply_gradients ( subtracted_grads_and_vars)
```

TF 也提供了一个独立的梯度计算函数 tf.gradients()；

TF 预置了如下优化器：

tf.train.GradientDescentOptimizer
tf.train.AdadeltaOptimizer
tf.train.AdagradOptimizer
tf.train.AdagradDAOptimizer
tf.train.MomentumOptimizer
tf.train.AdamOptimizer
tf.train.FtrlOptimizer
tf.train.ProximalGradientDescentOptimizer
tf.train.ProximalAdagradOptimizer
tf.train.RMSPropOptimizer

不同优化器的比较：

[An overview of gradient descent optimization algorithms – Sebastian Uder]

[《An overview of gradient descent optimization algorithms》翻译]

- 三种梯度下降
  - 批量梯度下降，每一趟都用整个训练集来更新参数，每次更新都会朝正确的方向进行，最后能保证收敛于极值点，但每次学习时间过程，消耗大量内存，不能在线更新；
  - 随机梯度下降，每一趟随机选择一个样本来更新参数，学习非常快速，可以在线更新，但每次更新可能不会按正确方向进行，存在优化波动，波动也有可能使得参数跳出局部极小值而搜索到更好的局部极小值，迭代次数增多，收敛速度变慢，但最终与批量梯度下降具有相同的收敛性；
  - 小批量梯度下降，批量梯度下降和随机梯度下降的折中，每一趟随机选取若干样本来更新参数；
- 优化算法
  - 动量

    参数更新时加上上一次更新的动量，防止出现在峡谷地区（某些方向比其他方向陡峭得多）附近振荡而降低收敛速度的情况；

    $$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
    $$\theta = \theta - v_t$$

    加快与上一次梯度方向相同的参数的更新速度，降低与上一次梯度方向相反的参数的更新速度
  - NAG，涅斯捷罗夫梯度加速

    在损失函数中减去动量项

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

根据损失函数的斜率做到自适应更新，加速 SGD 的收敛，阻止过快更新来提高响应性（RNNs）

- Adagrad

  适合处理稀疏特征数据，降低非稀疏特征的学习率，提高稀疏特征的学习率，能够很好的提高 SGD 的鲁棒性。但需要更多的计算时间，同时学习率不断衰减至一个非常小的值；

  Adadelta 是 Adagrad 的扩展算法，缓解了学习速率衰减过快的问题；

  RMSprop 是 Adadelta 的中间形式，也是为了学习率衰减过快的问题

  Adam 在 RMSprop 的基础上使用动量与偏差修正

- 如何选择 SGD 优化器
  - 稀疏特征最好使用自适应学习速率的优化器
  - Adagrad、Adadelta、RSMprop、Adam 中，Adam 可能表现更优一些
  - 如果在意收敛速度或者训练一个复杂的网络最好使用自适应学习速率的优化器
- 其他优化策略
  - Shuffling and Curriculum Learning

    每次迭代中随机打乱训练集中的样本
  - Batch normalization

    在每次小批量梯度下降反向传播之后重新对参数进行 0 均值 1 方差标准化
  - Early stopping

    在验证集上如果连续的多次迭代过程中损失函数不再显著地降低，那么应该提前结束训练
  - Gradient noise

    在每次迭代计算梯度中加上一个高斯分布的随机误差