Tensorflow

What?

深度学习框架，用于训练神经网络模型。

什么是深度学习？

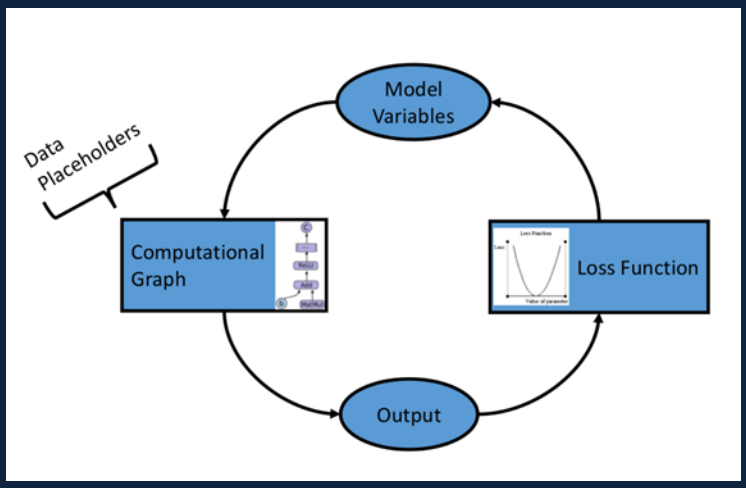
多层的神经网络。

Why?

深度学习需要大量的计算。它通常包含具有许多节点的神经网络，并且每个节点都有许多需要在学习过程中必须不断更新的连接。换句话说，神经网络的每一层都有成百上千个相同的人工神经元在执行相同的计算。因此，神经网络的结构适用于 GPU（图形处理单元）可以高效执行的计算类型（GPU 是专门为并行计算相同指令而设计的）。

How?

- 用 TensorFlow 之前你需要了解关于 TensorFlow 的以下基础知识:
- 使用图(graphs) 来表示计算.
- 在会话(Session) 中执行图.
- 使用张量(tensors) 来代表数据.
- 通过变量(Variables) 维护状态.
- 使用供给(feeds) 和取回(fetches) 将数据传入或传出任何操作.



1. 图

What?

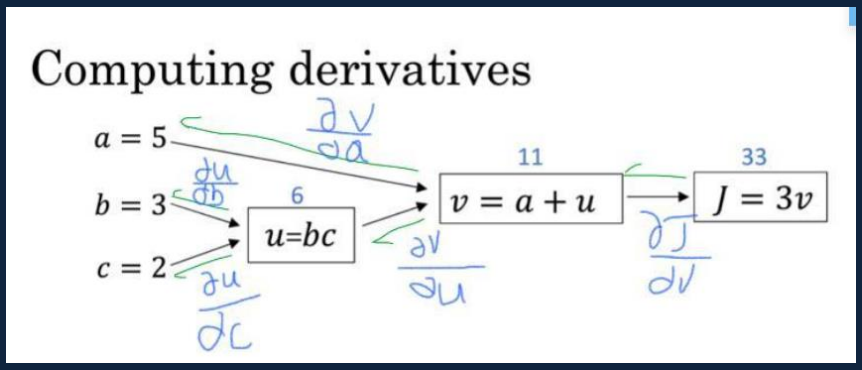
结点和有向边

比如说 J=a+bc 在图计算的表示中：

1.节点表示某种运算

2.有向边表示数据和数据的流向

比如 节点 u=a+u，有两条边流进来，就说明这个节点是需要这两条边的数据。这是“需”而 a=5 会自发的流向 v=a+u，这是“供”（5，3，2 也都是节点）



Why?

利用这个图来表示计算而不是一般的计算是因为，正向可以方便取中间值求结果，反向可以很方便的求偏导。

Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation
3. Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices
4. Many common machine learning models are taught and visualized as directed graphs

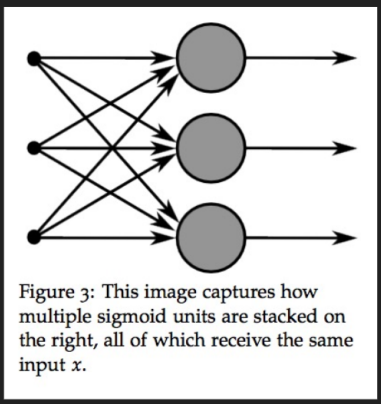


Figure 3: This image captures how multiple sigmoid units are stacked on the right, all of which receive the same input x.

A neural net graph from Stanford's CS224N course

64

How?

创建节点（输出）和边（输入）

Implement?

```
import tensorflow as tf

 # 创建一个常量 op, 产生一个 1x2 矩阵. 这个 op 被作为一个节点

 # 加到默认图中.

 #

 # 构造器的返回值代表该常量 op 的返回值.

matrix1 = tf.constant([[3., 3.]])

 # 创建另外一个常量 op, 产生一个 2x1 矩阵.

matrix2 = tf.constant([[2.],[2.]])

 # 创建一个矩阵乘法 matmul op , 把 'matrix1' 和 'matrix2' 作为输入.

 # 返回值 'product' 代表矩阵乘法的结果.

product = tf.matmul(matrix1, matrix2)
```

这样图就设计好了，之后应该怎么获取图的值呢？-Create a session.

2. 在一个会话中启动图

   What?

   通过会话启动图

   Why?

   获取图的值

   How?

   第一步是创建一个 Session 对象, tf.Session(),

   第二步通过 run()方法 feed，然后 fetch 到该图的值

   第三步关闭会话使得本次运行的资源可以被释放。

   Implement:用 with as 语句简化代码,这样就可以不用调用 Sess.close()函数关闭对话。

```
with tf.Session() as sess:
  result = sess.run([product])
  print result
```

TensorFlow 还提供了填充机制，可以在构建图时使用 tf.placeholder()临时替代任意操作的张量，在调用 Session 对象的 run()方法去执行图时，使用填充数据作为调用的参数，调用结束后，填充数据就消失。代码示例如下：

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.multiply(input1, input2)
with tf.Session() as sess:
  print sess.run([output], feed_dict={input1:[7.], input2:[2.]})
# 输出 [array([ 14.], dtype=float32)]
```

3. Tensor

What?

An n-dimensional array

0-d tensor: scalar (number)

1-d tensor: vector

2-d tensor: matrix

and so on

How?

Here we will cover the main ways to create tensors in TensorFlow:

1. Fixed tensors:
   - Create a zero filled tensor. Use the following:
     `zero_tsr = tf.zeros([row_dim, col_dim])`
   - Create a one filled tensor. Use the following:
     `ones_tsr = tf.ones([row_dim, col_dim])`
   - Create a constant filled tensor. Use the following:
     `filled_tsr = tf.fill([row_dim, col_dim], 42)`
   - Create a tensor out of an existing constant. Use the following:
     `constant_tsr = tf.constant([1,2,3])`

4. 变量

What?

在图中有固定位置的变量（一般为需要更新的参数），不会像 tensor 那样流动

How?

1) 创建变量对象 Variable()
2) 需要初始化

```
my_var = tf.Variable(tf.zeros([2,3]))
sess = tf.Session()
initialize_op = tf.global_variables_initializer ()
sess.run(initialize_op)
```

5. 设备

如果机器上有超过一个可用的 GPU，除第一个外的其它 GPU 默认是不参与计算的. 为了让 TensorFlow 使用这些

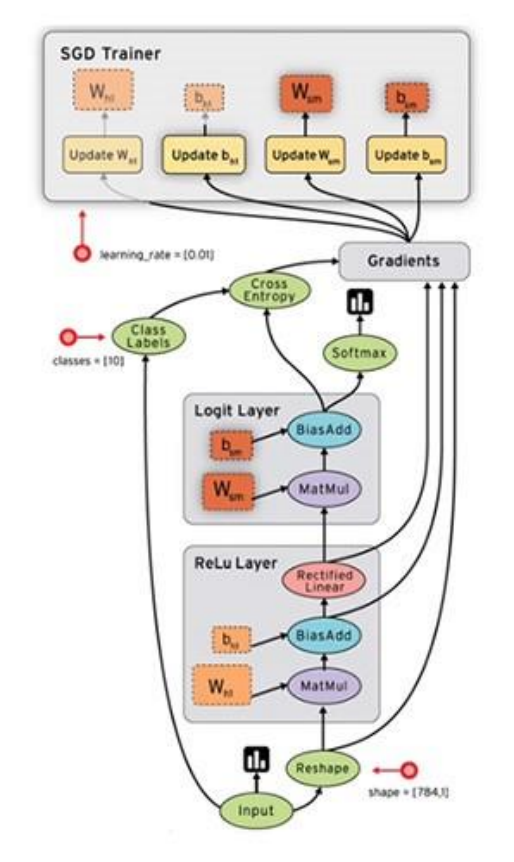
GPU，你必须将 op 明确指派给它们执行. with...Device 语句用来指派特定的 CPU 或 GPU 执行操

作:

```
with tf.Session() as sess:
  with tf.device("/gpu:1"):
    matrix1 = tf.constant([[3., 3.]])
    matrix2 = tf.constant([[2.],[2.]])
    product = tf.matmul(matrix1, matrix2)
    ...
```

设备用字符串进行标识. 目前支持的设备包括:

- `"/cpu:0"`: 机器的 CPU.
- `"/gpu:0"`: 机器的第一个 GPU, 如果有的话.
- `"/gpu:1"`: 机器的第二个 GPU, 以此类推.



了解了 TF 的基础知识之后，TF 是怎样实现神经网络呢？有如下几个步骤：

1. **提取特征向量**作为神经网络的输入。（洗数据）
2. 定义如何从神经网络的**输入得到输出**（前向传播算法）
3. **优化**神经网络的模型，通过训练数据来调整神经网络参数的取值。（反向传播算法）

- 神经网络

  What?

  由神经元和连接构成

  至少三个层（输入层，隐藏层，输出层）

  主要用于分类

  Why?

  对大规模训练样本效果好

  How?

  训练权值（连接）

  一个神经网络的训练算法就是让神经元之间的权重的值调整到最佳，以使得整个网络的预测效果最好。

- 神经元

  What?

  神经元是多个输入和一个输出的结构

  神经元的输入可以是整个神经网络的输入，也可以是其他神经元的输出

  神经元的输出是所有输入的加权和。

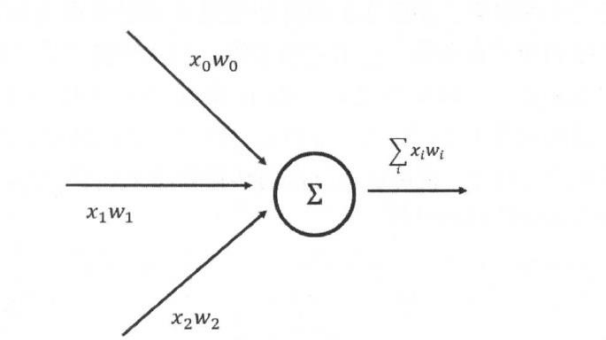


图 3-4　神经元结构示意图

- 前向传播算法：

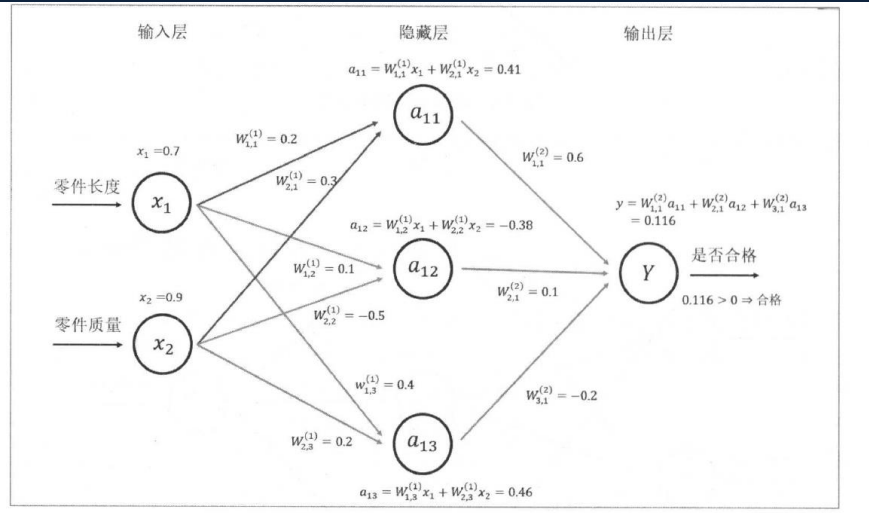


图 3-6　神经网络前向传播算法示意图

- 计算神经网络的前向传播结果需要三部分信息。

  第一个是神经网络的输入，就是从实体中提取的特征向量。

  第二个是神经元，不同神经元之间输入输出的连接关系。

  第三个是连接的权重

  给定神经网络的输入、神经网络的结构以及边上权重，就可以通过前向传播算法来计算出神经网络的输出。

- 实现：

```
a = tf.matmul(x, w1)
y = tf.matmul(a, w2)
```

- 反向传播算法：

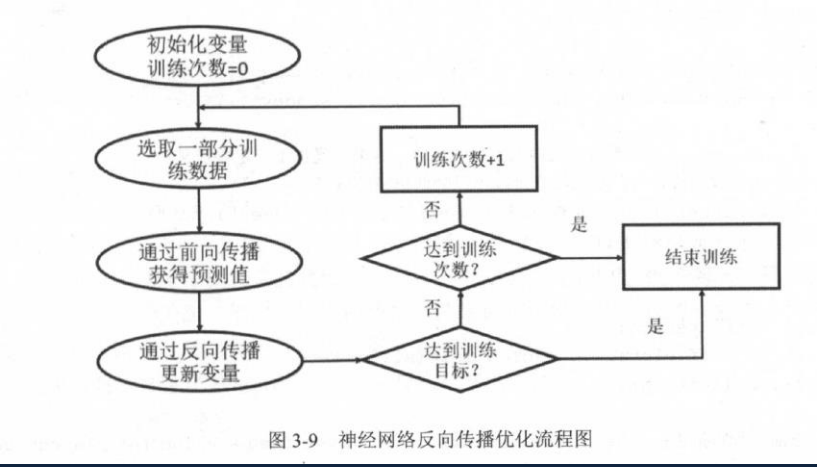


图 3-9　神经网络反向传播优化流程图

What?

在神经网络优化算法中，最常用的方法是反向传播算法（ backpropagation ）

Why?

需要调整神经网络中的参数对训练数据进行拟合。

How?

第一步是使用 TensorFlow 表达一个 batch 的数据

Why batch?

batch 的思想，至少有两个作用，一是更好的处理非凸的损失函数，非凸的情况下， 全样本就算工程上算的动， 也会卡在局部优上， 批表示了全样本的部分抽样实现， 相当于人为引入修正梯度上的采样噪声，使“一路不通找别路”更有可能搜索最优值；二是合理利用内存容量。如果数据集较小，可以采用全数据集（Full batch learning）的形式，这样有两个显然的好处：1.由全数据集计算的梯度能够更好的代表样本总体，从而更准确的朝向极值所在的方向；2.不同权重的梯度值差别很大，因此选取一个全局的学习率会比较困难。

第二步是用这个 batch 的样例，通过前向传播算法得到神经网络模型的预测结果。（前向传播算法）

第三步是以计算出当前神经网络模型的预测答案与正确答案之间的差距。(计算损失函数)

第四步是基于预测值和真实值之间的差距，反向传播算法会相应更新神经网络参数的取值，然后反复迭代（反向传播算法）

```
y=tf.sigmoid(y)
# 定义损失函数来刻画预测值与真实值得差距。
cross_entropy = -tf.reduce_mean(
    y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0))
    +(1-y)*tf.log(tf.clip_by_value(1-y, 1e-10, 1.0)))
# 定义学习率，在第 4 章中将更加具体的介绍学习率。
learning_rate = 0.001
# 定义反向传播算法来优化神经网络中的参数。
train_step =\
    tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

- 在以上代码中， cross_entropy 定义了损失函数。第二行 train_ step 定义了反向传播的优化方法。在定义了反向传播算法之后，通过运行 sess.run(train_ step)就可以对所有在集合中的变量进行优化。
- 完整神经网络样例程序

l. 定义神经网络的结构和前向传播的输出结果。

```
import tensorflow as tf

# NumPy 是一个科学计算的工具包，这里通过 NumPy 工具包生成模拟数据集。
from numpy.random import RandomState

# 定义训练数据 batch 的大小。
batch_size = 8

# 定义神经网络的参数，这里还是沿用 3.4.2 小节中给出的神经网络结构。
w1 = tf.Variable(tf.random_normal([2, 3], stddev=1, seed=1))
w2 = tf.Variable(tf.random_normal([3, 1], stddev=1, seed=1))

# 在 shape 的一个维度上使用 None 可以方便使用不同的 batch 大小。在训练时需要把数据分
# 成比较小的 batch，但是在测试时，可以一次性使用全部的数据。当数据集比较小时这样比较
# 方便测试，但数据集比较大时，将大量数据放入一个 batch 可能会导致内存溢出。
x = tf.placeholder(tf.float32, shape=(None, 2), name='x-input')
y_ = tf.placeholder(tf.float32, shape=(None, 1), name='y-input')

# 定义神经网络前向传播的过程。
a = tf.matmul(x, w1)
y = tf.matmul(a, w2)

# 定义损失函数和反向传播的算法。
y=tf.sigmoid(y)
```

2. 定义损失函数以及选择反向传播优化的算法。

```
cross_entropy = -tf.reduce_mean(
    y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0))
    +(1-y)*tf.log(tf.clip_by_value(1-y, 1e-10, 1.0)))
train_step = tf.train.AdamOptimizer(0.001).minimize(cross_entropy)
```

4. 生成会话并且在训练、数据上反复运行反向传播优化算法。无论神经网络的结构如何变化，这三个步骤是不变的。

```
# 创建一个会话来运行 TensorFlow 程序。
with tf.Session() as sess:
    init_op = tf.global_variables_initializer()
    # 初始化变量。
    sess.run(init_op)

    print sess.run(w1)
    print sess.run(w2)
    '''
    在训练之前神经网络参数的值：
    w1 = [[-0.81131822, 1.48459876, 0.06532937]
          [-2.44270396, 0.0992484, 0.59122431]]
    w2 = [[-0.81131822], [1.48459876], [0.06532937]]
    '''

    # 设定训练的轮数。
    STEPS = 5000
    for i in range(STEPS):
        # 每次选取 batch_size 个样本进行训练。
        start = (i * batch_size) % dataset_size
        end =  min(start+batch_size, dataset_size)

        # 通过选取的样本训练神经网络并更新参数。
        sess.run(train_step,
                 feed_dict={x: X[start:end], y_: Y[start:end]})
        if i % 1000 == 0:
            # 每隔一段时间计算在所有数据上的交叉熵并输出。
            total_cross_entropy = sess.run(
                cross_entropy, feed_dict={x: X, y_: Y})
```

```
            print("After %d training step(s), cross entropy on all data is %g" %
                  (i, total_cross_entropy))
        '''
        输出结果:
        After 0 training step(s), cross entropy on all data is 1.89805
        After 1000 training step(s), cross entropy on all data is 0.655075
        After 2000 training step(s), cross entropy on all data is 0.626172
        After 3000 training step(s), cross entropy on all data is 0.615096
        After 4000 training step(s), cross entropy on all data is 0.610309

        通过这个结果可以发现随着训练的进行，交叉熵是逐渐变小的。交叉熵越小说明
        预测的结果和真实的结果差距越小。
        '''

print sess.run(w1)
print sess.run(w2)
'''
在训练之后神经网络参数的值:
w1 = [[0.02476984, 0.5694868, 1.69219422]
      [-2.19773483, -0.23668921, 1.11438966]]
w2 = [[-0.45544702], [0.49110931], [-0.9811033]]

可以发现这两个参数的取值已经发生了变化，这个变化就是训练的结果。
它使得这个神经网络能更好地拟合提供的训练数据。
'''
```