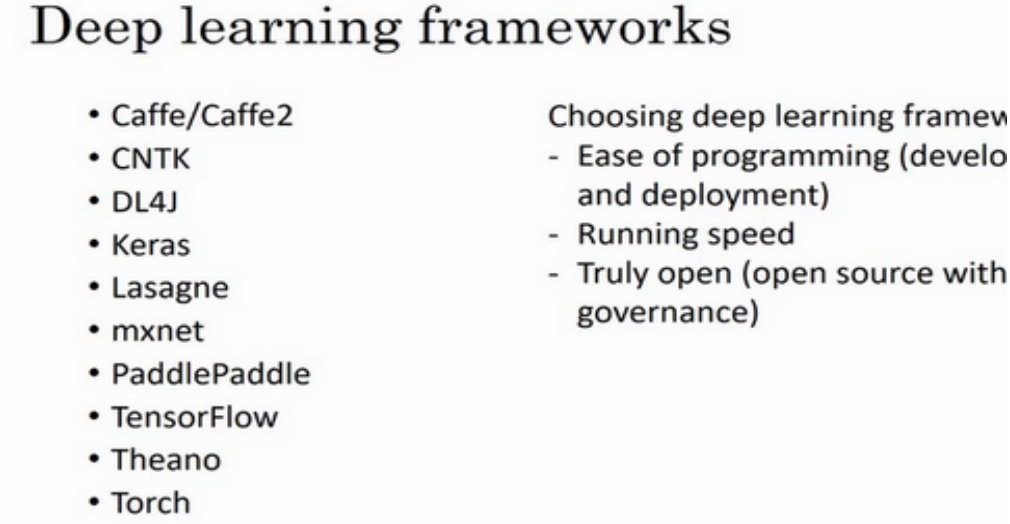
3.10 深度学习框架（Deep Learning frameworks）

你已经差不多从零开始学习了使用 **Python** 和 **NumPy** 实现深度学习算法，很高兴你这样做了，因为我希望你理解这些深度学习算法实际上在做什么。但你会发现，除非应用更复杂的模型，例如卷积神经网络，或者循环神经网络，或者当你开始应用很大的模型，否则它就越来越不实用了，至少对大多数人而言，从零开始全部靠自己实现并不现实。

幸运的是，现在有很多好的深度学习软件框架，可以帮助你实现这些模型。类比一下，我猜你知道如何做矩阵乘法，你还应该知道如何编程实现两个矩阵相乘，但是当你在建很大的应用时，你很可能不想用自己的矩阵乘法函数，而是想要访问一个数值线性代数库，它会更高效，但如果你明白两个矩阵相乘是怎么回事还是挺有用的。我认为现在深度学习已经很成熟了，利用一些深度学习框架会更加实用，会使你的工作更加有效，那就让我们来看下有哪些框架。



现在有许多深度学习框架，能让实现神经网络变得更简单，我们来讲主要的几个。每个框架都针对某一用户或开发群体的，我觉得这里的每一个框架都是某类应用的可靠选择，有很多人写文章比较这些深度学习框架，以及这些深度学习框架发展得有多好，而且因为这些框架往往不断进化，每个月都在进步，如果你想看看关于这些框架的优劣之处的讨论，我留给你自己去网上搜索，但我认为很多框架都在很快进步，越来越好，因此我就不做强烈推荐了，而是与你分享推荐一下选择框架的标准。

一个重要的标准就是便于编程，这既包括神经网络的开发和迭代，还包括为产品进行配置，为了成千上百万，甚至上亿用户的实际使用，取决于你想要做什么。

第二个重要的标准是运行速度，特别是训练大数据集时，一些框架能让你更高效地运行和训练神经网络。

还有一个标准人们不常提到，但我觉得很重要，那就是这个框架是否真的开放，要是一个框架真的开放，它不仅需要开源，而且需要良好的管理。不幸的是，在软件行业中，一些公司有开源软件的历史，但是公司保持着对软件的全权控制，当几年时间过去，人们开始使用他们的软件时，一些公司开始逐渐关闭曾经开放的资源，或将功能转移到他们专营的云服务中。因此我会注意的一件事就是你能否相信这个框架能长时间保持开源，而不是在一家公司的控制之下，它未来有可能出于某种原因选择停止开源，即便现在这个软件是以开源的形式发布的。但至少在短期内，取决于你对语言的偏好，看你更喜欢 **Python**，**Java** 还是 **C++**或者其它什么，也取决于你在开发的应用，是计算机视觉，还是自然语言处理或者线上广告，等等，我认为这里的多个框架都是很好的选择。

程序框架就讲到这里，通过提供比数值线性代数库更高程度的抽象化，这里的每一个程序框架都能让你在开发深度机器学习应用时更加高效。

3.11 TensorFlow

欢迎来到这周的最后一个视频，有很多很棒的深度学习编程框架，其中一个是**TensorFlow**，我很期待帮助你开始学习使用 **TensorFlow**，我想在这个视频中向你展示 **TensorFlow** 程序的基本结构，然后让你自己练习，学习更多细节，并运用到本周的编程练习中，这周的编程练习需要花些时间来做，所以请务必留出一些空余时间。

先提一个启发性的问题，假设你有一个损失函数J需要最小化，在本例中，我将使用这个高度简化的损失函数，$Jw = w^2 - 10w + 25$，这就是损失函数，也许你已经注意到该函数其实就是$(w-5)^2$，如果你把这个二次方式子展开就得到了上面的表达式，所以使它最小的w值是 5，但假设我们不知道这点，你只有这个函数，我们来看一下怎样用 **TensorFlow** 将其最小化，因为一个非常类似的程序结构可以用来训练神经网络。其中可以有一些复杂的损失函数$J(w,b)$取决于你的神经网络的所有参数，然后类似的，你就能用 **TensorFlow** 自动找到使损失函数最小的w和b的值。但让我们先从左边这个更简单的例子入手。

In [1]:
```
import numpy as np
import tensorflow as tf
```

In [2]:
```
w=tf.Variable(0,dtype=tf.float32)
cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init=tf.global_variables_initializer()
session=tf.Session()
session.run(init)
print(session.run(w))
```

```
0.0
```

我在我的 **Jupyter notebook** 中运行 **Python**，

```
import numpy as np
import tensorflow as tf
#导入 TensorFlow
w = tf.Variable(0,dtype = tf.float32)
#接下来，让我们定义参数 w，在 TensorFlow 中，你要用 tf.Variable()来定义参数
#然后我们定义损失函数：
cost = tf.add(tf.add(w**2,tf.multiply(- 10.,w)),25)
#然后我们定义损失函数 J
```

#然后我们再写：

```
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
```

#(让我们用 0.01 的学习率，目标是最小化损失)。

#最后下面的几行是惯用表达式：

```
init = tf.global_variables_initializer()
session = tf.Sessions()
```

#这样就开启了一个 TensorFlow session。

```
session.run(init)
```

#来初始化全局变量。

#然后让 TensorFlow 评估一个变量，我们要用到：

```
session.run(w)
```

#上面的这一行将 w 初始化为 0，并定义损失函数，我们定义 train 为学习算法，它用梯度下降法优化器使损失函数最小化，但实际上我们还没有运行学习算法，所以#上面的这一行将 w 初始化为 0，并定义损失函数，我们定义 train 为学习算法，它用梯度下降法优化器使损失函数最小化，但实际上我们还没有运行学习算法，所以session.run(w)评估了 w，让我::

```
print(session.run(w))
```

#所以如果我们运行这个，它评估w等于 0，因为我们什么都还没运行。

#现在让我们输入：

```
session.run(train)
```

，它所做的就是运行一步梯度下降法。

#接下来在运行了一步梯度下降法后，让我们评估一下 w 的值，再 print：

```
print(session.run(w))
```

#在一步梯度下降法之后，w 现在是 0.1。

```
In [3]: session.run(train)
        print(session.run(w))

        0.1
```

现在我们运行梯度下降 1000 次迭代：

```
In [4]: for i in range(1000):
            session.run(train)
        print(session.run(w))

        4.99999
```

这是运行了梯度下降的 1000 次迭代，最后w变成了 4.99999，记不记得我们说$(w-5)^2$

最小化，因此w的最优值是 5，这个结果已经很接近了。

希望这个让你对 **TensorFlow** 程序的大致结构有了了解，当你做编程练习，使用更多 **TensorFlow** 代码时，我这里用到的一些函数你会熟悉起来，这里有个地方要注意，w是我们想要优化的参数，因此将它称为变量，注意我们需要做的就是定义一个损失函数，使用这些 add 和 multiply 之类的函数。**TensorFlow** 知道如何对 add 和 mutiply，还有其它函数求导，这就是为什么你只需基本实现前向传播，它能弄明白如何做反向传播和梯度计算，因为它已经内置在 add，multiply 和平方函数中。

对了，要是觉得这种写法不好看的话，**TensorFlow** 其实还重载了一般的加减运算等等，因此你也可以把$cost$写成更好看的形式，把之前的 cost 标成注释，重新运行，得到了同样的结果。

```
In [5]: w=tf.Variable(0,dtype=tf.float32)
        #cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
        cost=w**2-10*w+25
        train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

        init=tf.global_variables_initializer()
        session=tf.Session()
        session.run(init)
        print(session.run(w))

        0.0

In [6]: session.run(train)
        print(session.run(w))

        0.1

In [7]: for i in range(1000):
            session.run(train)
        print(session.run(w))

        4.99999
```

一旦w被称为 TensorFlow 变量，平方，乘法和加减运算都重载了，因此你不必使用上面这种不好看的句法。

TensorFlow 还有一个特点，我想告诉你，那就是这个例子将w的一个固定函数最小化了。如果你想要最小化的函数是训练集函数又如何呢？不管你有什么训练数据x，当你训练神经网络时，训练数据x会改变，那么如何把训练数据加入 **TensorFlow** 程序呢？

我会定义x，把它想做扮演训练数据的角色，事实上训练数据有x和y，但这个例子中只有x，把x定义为：

`x = tf.placeholder(tf.float32,[3,1])`，让它成为[3,1]数组，我要做的就是，因为$cost$这个二次方程的三项前有固定的系数，它是$w^2 + 10w + 25$，我们可以把这些数字

1，-10 和 25 变成数据，我要做的就是把$cost$替换成：

`cost = x[0][0]*w**2 +x[1][0]*w + x[2][0]`，现在x变成了控制这个二次函数系数的数据，这个 **placeholder** 函数告诉 **TensorFlow**，你稍后会为x提供数值。

让我们再定义一个数组，`coefficient = np.array([[1.],[-10.],[25.]])`，这就是我们要接入x的数据。最后我们需要用某种方式把这个系数数组接入变量x，做到这一点的句法是，在训练这一步中，要提供给x的数值，我在这里设置：

`feed_dict = {x:coefficients}`

好了，希望没有语法错误，我们重新运行它，希望得到和之前一样的结果。

```
In [11]: coefficients=np.array([[1.],[-10.],[25.]])

         w=tf.Variable(0,dtype=tf.float32)
         x=tf.placeholder(tf.float32,[3,1])
         #cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
         #cost=w**2-10*w+25
         cost=x[0][0]*w**2+x[1][0]*w+x[2][0]
         train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

         init=tf.global_variables_initializer()
         session=tf.Session()
         session.run(init)
         print(session.run(w))

         0.0

In [12]: session.run(train,feed_dict={x:coefficients})
         print(session.run(w))

         0.1

In [13]: for i in range(1000):
             session.run(train,feed_dict={x:coefficients})
         print(session.run(w))

         4.99999
```

现在如果你想改变这个二次函数的系数，假设你把：

`coefficient = np.array([[1.],[-10.],[25.]])`

改为：

`coefficient = np.array([[1.],[-20.],[100.]])`

现在这个函数就变成了$(w-10)^2$，如果我重新运行，希望我得到的使$(w-10)^2$最小化的w值为 10，让我们看一下，很好，在梯度下降 1000 次迭代之后，我们得到接近 10 的w。

In [14]:
```
coefficients=np.array([[1.],[-20.],[100.]])

w=tf.Variable(0,dtype=tf.float32)
x=tf.placeholder(tf.float32,[3,1])
#cost=tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
#cost=w**2-10*w+25
cost=x[0][0]*w**2+x[1][0]*w+x[2][0]
train=tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init=tf.global_variables_initializer()
session=tf.Session()
session.run(init)
print(session.run(w))
```

```
0.0
```

In [15]:
```
session.run(train,feed_dict={x:coefficients})
print(session.run(w))
```

```
0.2
```

In [16]:
```
for i in range(1000):
    session.run(train,feed_dict={x:coefficients})
print(session.run(w))
```

```
9.99998
```

在你做编程练习时，见到更多的是，**TensorFlow** 中的 **placeholder** 是一个你之后会赋值的变量，这种方式便于把训练数据加入损失方程，把数据加入损失方程用的是这个句法，当你运行训练迭代，用 `feed_dict` 来让 `x=coefficients`。如果你在做 **mini-batch** 梯度下降，在每次迭代时，你需要插入不同的 **mini-batch**，那么每次迭代，你就用 `feed_dict` 来喂入训练集的不同子集，把不同的 **mini-batch** 喂入损失函数需要数据的地方。

希望这让你了解了 **TensorFlow** 能做什么，让它如此强大的是，你只需说明如何计算损失函数，它就能求导，而且用一两行代码就能运用梯度优化器，**Adam** 优化器或者其他优化器。

Code example

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]     # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()                        with tf.Session() as session:
session.run(init)                                 session.run(init)
print(session.run(w))                             print(session.run(w))

for i in range(1000):
     session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

这还是刚才的代码，我稍微整理了一下，尽管这些函数或变量看上去有点神秘，但你在做编程练习时多练习几次就会熟悉起来了。

```
session = tf.Session()                with tf.Session() as session:
session.run(init)                         session.run(init)
print(session.run(w))                     print(session.run(w))
```

还有最后一点我想提一下，这三行（蓝色大括号部分）在 **TensorFlow** 里是符合表达习惯的，有些程序员会用这种形式来替代，作用基本上是一样的。

但这个 **with** 结构也会在很多 **TensorFlow** 程序中用到，它的意思基本上和左边的相同，但是 **Python** 中的 **with** 命令更方便清理，以防在执行这个内循环时出现错误或例外。所以你也会在编程练习中看到这种写法。那么这个代码到底做了什么呢？让我们看这个等式：

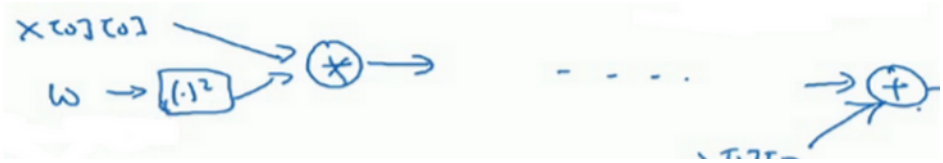
```
cost = x[0][0]*w**2 +x[1][0]*w + x[2][0]#(w-5)**2
```

TensorFlow 程序的核心是计算损失函数，然后 **TensorFlow** 自动计算出导数，以及如何最小化损失，因此这个等式或者这行代码所做的就是让 **TensorFlow** 建立计算图，计算图所做的就是取$x[0][0]$，取w，然后将它平方，然后$x[0][0]$和w^2相乘，你就得到了$x[0][0]*w^2$，以此类推，最终整个建立起来计算$cost = [0][0]*w**2 + x[1][0]*w + x[2][0]$，最后你得到了损失函数。

TensorFlow 的优点在于，通过用这个计算损失，计算图基本实现前向传播，**TensorFlow**

已经内置了所有必要的反向函数，回忆一下训练深度神经网络时的一组前向函数和一组反向函数，而像 **TensorFlow** 之类的编程框架已经内置了必要的反向函数，这也是为什么通过内置函数来计算前向函数，它也能自动用反向函数来实现反向传播，即便函数非常复杂，再帮你计算导数，这就是为什么你不需要明确实现反向传播，这是编程框架能帮你变得高效的原因之一。

如果你看 **TensorFlow** 的使用说明，我只是指出 **TensorFlow** 的说明用了一套和我不太一样的符号来画计算图，它用了$x[0][0]$，w，然后它不是写出值，想这里的w^2，**TensorFlow** 使用说明倾向于只写运算符，所以这里就是平方运算，而这两者一起指向乘法运算，以此类推，然后在最后的节点，我猜应该是一个将$x[2][0]$加上去得到最终值的加法运算。



为本课程起见，我认为计算图用第一种方式会更容易理解，但是如果你去看 **TensorFlow** 的使用说明，如果你看到说明里的计算图，你会看到另一种表示方式，节点都用运算来标记而不是值，但这两种呈现方式表达的是同样的计算图。

在编程框架中你可以用一行代码做很多事情，例如，你不想用梯度下降法，而是想用 **Adam** 优化器，你只要改变这行代码，就能很快换掉它，换成更好的优化算法。所有现代深度学习编程框架都支持这样的功能，让你很容易就能编写复杂的神经网络。

我希望我帮助你了解了 **TensorFlow** 程序典型的结构，概括一下这周的内容，你学习了如何系统化地组织超参数搜索过程，我们还讲了 **Batch** 归一化，以及如何用它来加速神经网络的训练，最后我们讲了深度学习的编程框架，有很多很棒的编程框架，这最后一个视频我们重点讲了 **TensorFlow**。有了它，我希望你享受这周的编程练习，帮助你更熟悉这些概念。