

3. Hello, Tensor World!

让我们来分析一下你刚才运行的 Hello World 的代码。代码如下：

```
import tensorflow as tf

# Create TensorFlow object called hello_constant
hello_constant = tf.constant('Hello World!')

with tf.Session() as sess:
    # Run the tf.constant operation in the session
    output = sess.run(hello_constant)
    print(output)
```

Tensor

在 TensorFlow 中，数据不是以整数、浮点数或者字符串形式存储的。这些值被封装在一个叫做 **tensor** 的对象中。在 `hello_constant = tf.constant('Hello World!')` 代码中，`hello_constant` 是一个 0 维度的字符串 **tensor**，**tensor** 还有很多不同大小：

```
# A is a 0-dimensional int32 tensor
A = tf.constant(1234)
# B is a 1-dimensional int32 tensor
B = tf.constant([123,456,789])
# C is a 2-dimensional int32 tensor
C = tf.constant([ [123,456,789], [222,333,444] ])
```

`tf.constant()` 是你在本课中即将使用的多个 TensorFlow 运算之一。`tf.constant()` 返回的 **tensor** 是一个常量 **tensor**，因为这个 **tensor** 的值不会变

Session

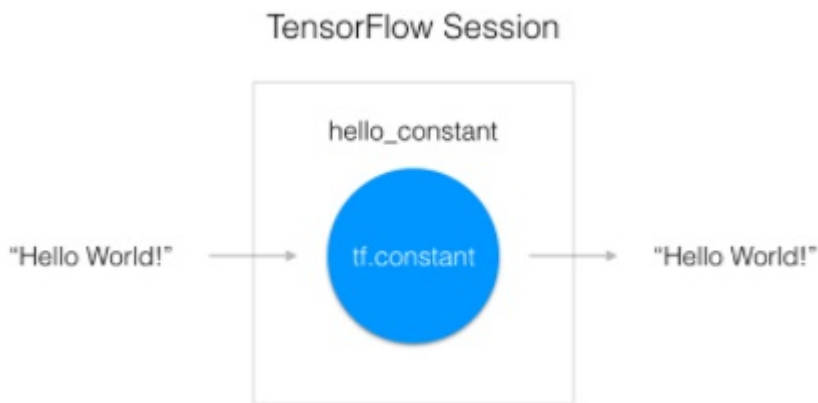
TensorFlow 的 **api** 构建在 **computational graph** 的概念上，它是一种对数学运算过程进行可视化的方法（在 MiniFlow 这节课中学过）。让我们把你刚才运行的 TensorFlow 代码变成一个图：

□

如上图所示，一个 "TensorFlow Session" 是用来运行图的环境。这个 **session** 负责分配 GPU(s) 和 / 或 CPU(s)，包括远程计算机的运算。让我们看看如何使用它：

```
with tf.Session() as sess:
    output = sess.run(hello_constant)
    代码已经从之前的一行中创建了 tensor hello_constant。接下来是在 session 里对 tensor 求值。

这段代码用 tf.Session 创建了一个 sess 的 session 实例。然后 sess.run() 函数对 tensor 求值，并返回结果。
```



如上图所示，一个“TensorFlow Session”是用来运行图的环境。

这个session负责分配GPU(s)和/或CPU(s)，包括远程计算机的运算。让我们看看如何使用它：

```
with tf.Session() as sess:
    output = sess.run(hello_constant)
```

代码已经从之前的一行中创建了tensor `hello_constant`。接下来是在session里对tensor求值。

这段代码用`tf.Session` 创建了一个`sess`的session实例。然后`sess.run()`函数对tensor求值，并返回结果。

4.TensorFlow 输入

在上一小节中，你向 session 传入一个 tensor 并返回结果。如果你想使用一个非常量（non-constant）该怎么办？这就是 `tf.placeholder()` 和 `feed_dict` 派上用场的时候了。这一节将向你讲解向 TensorFlow 传输数据的基础知识。

`tf.placeholder()`

很遗憾，你不能把数据集赋值给 `x` 再将它传给 TensorFlow。因为之后你会想要你的 TensorFlow 模型对不同的数据集采用不同的参数。你需要的是 `tf.placeholder()`！

数据经过 `tf.Session.run()` 函数得到的值，由 `tf.placeholder()` 返回成一个 tensor，这样你可以在 session 运行之前，设置输入。

Session 的 `feed_dict`

```
x = tf.placeholder(tf.string)

with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'Hello World'})
```

用 `tf.Session.run()` 里的 `feed_dict` 参数设置占位 tensor。上面的例子显示 tensor `x` 被设置成字符串 "Hello, world"。如下所示，也可以用 `feed_dict` 设置多个 tensor。

```
x = tf.placeholder(tf.string)
y = tf.placeholder(tf.int32)
z = tf.placeholder(tf.float32)

with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'Test String', y: 123, z: 45.67})
```

注意：

如果传入 `feed_dict` 的数据与 `tensor` 类型不符，就无法被正确处理，你会得到 `"ValueError: invalid literal for..."`。

5. TensorFlow 数学

获取输入很棒，但是现在你需要使用它。你将使用每个人都懂的基础数学运算，加、减、乘、除，来处理 `tensor`。（更多数学函数请查看文档）。

加法

```
x = tf.add(5, 2) # 7
```

从加法开始，`tf.add()` 函数如你所想，它传入两个数字、两个 `tensor`、或数字和 `tensor` 各一个，以 `tensor` 的形式返回它们的和。

减法和乘法

这是减法和乘法的例子：

```
x = tf.subtract(10, 4) # 6
y = tf.multiply(2, 5) # 10
```

`x tensor` 求值结果是 6，因为 $10 - 4 = 6$ 。`y tensor` 求值结果是 10，因为 $2 * 5 = 10$ 。是不是很简单！

类型转换

为了让特定运算能运行，有时会对类型进行转换。例如，你尝试下列代码，会报错：

```
tf.subtract(tf.constant(2.0),tf.constant(1)) # Fails with ValueError: Tensor conversion requested dtype float32 for Tensor with dtype int32:
```

这是因为常量 1 是整数，但是常量 2.0 是浮点数，`subtract` 需要它们的类型匹配。

在这种情况下，你可以确保数据都是同一类型，或者强制转换一个值为另一个类型。这里，我们可以把 2.0 转换成整数再相减，这样就能得出正确的结果：

```
tf.subtract(tf.cast(tf.constant(2.0), tf.int32), tf.constant(1)) # 1
```

6. TensorFlow 里的线性函数

神经网络中最常见的运算，就是计算输入、权重和偏差的线性组合。回忆一下，我们可以把线性运算的输出写成：

$$y = xW + b$$

这里 `W` 是连接两层的权重矩阵。输出 `y`，输入 `x`，偏差 `b` 全部都是向量。

TensorFlow 里的权重和偏差

训练神经网络的目的是更新权重和偏差来更好地预测目标。为了使用权重和偏差，你需要一个能修改的 `Tensor`。这就排除了 `tf.placeholder()` 和 `tf.constant()`，因为它们的 `Tensor` 不能改变。这里就需要 `tf.Variable` 了。

`tf.Variable()`

```
x = tf.Variable(5)
```

`tf.Variable` 类创建一个 `tensor`，其初始值可以被改变，就像普通的 Python 变量一样。该 `tensor` 把它的状态存在 `session` 里，所以你必须手动初始化它的状态。你

将使用 `tf.global_variables_initializer()` 函数来初始化所有可变 `tensor`。

初始化

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

`tf.global_variables_initializer()` 会返回一个操作，它会在 `graph` 中初始化所有的 `TensorFlow` 变量。你可以通过 `session` 来调用这个操作来初始化所有上面的变量。用 `tf.Variable` 类可以让我们改变权重和偏差，但还是要选择一个初始值。

从正态分布中取随机数来初始化权重是个好习惯。随机化权重可以避免模型每次训练时候卡在同一个地方。在下节学习梯度下降的时候，你将了解更多相关内容。

类似地，从正态分布中选择权重可以避免任意一个权重与其他权重相比有压倒性的特性。你可以用 `tf.truncated_normal()` 函数从一个正态分布中生成随机数。

tf.truncated_normal()

```
n_features = 120
n_labels = 5
weights = tf.Variable(tf.truncated_normal((n_features, n_labels)))
```

`tf.truncated_normal()` 返回一个 `tensor`，它的随机值取自一个正态分布，并且它们的取值会在这个正态分布平均值的两个标准差之内。

因为权重已经被随机化来帮助模型不被卡住，你不需要再把偏差随机化了。让我们简单地把偏差设为 0。

tf.zeros()

```
n_labels = 5
bias = tf.Variable(tf.zeros(n_labels))
```

`tf.zeros()` 函数返回一个都是 0 的 `tensor`。

7. TensorFlow Softmax

`Softmax` 函数可以把它的输入，通常被称为 `logits` 或者 `logit scores`，处理成 0 到 1 之间，并且能够把输出归一化到和为 1。这意味着 `softmax` 函数与分类的概率分布等价。它是一个网络预测多分类问题的最佳输出激活函数。



softmax 函数的实际应用示例

TensorFlow Softmax

当我们用 `TensorFlow` 来构建一个神经网络时，相应地，它有一个计算 `softmax` 的函数。

```
x = tf.nn.softmax([2.0, 1.0, 0.2])
```

就是这么简单，`tf.nn.softmax()` 直接为你实现了 `softmax` 函数，它输入 logits，返回 `softmax` 激活函数。

8.TensorFlow 中的交叉熵（Cross Entropy）

与 `softmax` 一样，TensorFlow 也有一个函数可以方便地帮我们实现交叉熵。

$$D(\hat{\mathbf{y}}, \mathbf{y}) = - \sum_j y_j \ln \hat{y}_j$$

Cross entropy loss function 交叉熵损失函数

让我们把你从视频当中学到的知识，在 TensorFlow 中来创建一个交叉熵函数。创建一个交叉熵函数，你需要用到这两个新的函数：

`tf.reduce_sum()`

`tf.log()`

Reduce Sum

```
x = tf.reduce_sum([1, 2, 3, 4, 5]) # 15
```

`tf.reduce_sum()` 函数输入一个序列，返回它们的和

Natural Log

```
x = tf.log(100) # 4.60517
```

`tf.log()` 所做跟你所想的一样，它返回所输入值的自然对数。

9.Mini-batching

在这一节，你将了解什么是 `mini-batching`，以及如何在 TensorFlow 里应用它。

`Mini-batching` 是一个一次训练数据集的一小部分，而不是整个训练集的技术。它可以使内存较小、不能同时训练整个数据集的电脑也可以训练模型。

`Mini-batching` 从运算角度来说是低效的，因为你不能在所有样本中计算 `loss`。但是这点小代价也比根本不能运行模型要划算。

它跟随机梯度下降（SGD）结合在一起用也很有帮助。方法是在每一代训练之前，对数据进行随机混洗，然后创建 `mini-batches`，对每一个 `mini-batch`，用梯度下降训练网络权重。因为这些 `batches` 是随机的，你其实是在对每个 `batch` 做随机梯度下降（SGD）。

让我们看看你的机器能否训练出 MNIST 数据集的权重和偏置项。

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
```

```

n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# Import MNIST data
mnist = input_data.read_data_sets('/datasets/ud730/mnist', one_hot=True)

# The features are already scaled and the data is shuffled
train_features = mnist.train.images
test_features = mnist.test.images

train_labels = mnist.train.labels.astype(np.float32)
test_labels = mnist.test.labels.astype(np.float32)

# Weights & bias
weights = tf.Variable(tf.random_normal([n_input, n_classes]))
bias = tf.Variable(tf.random_normal([n_classes]))

```

10. Epochs（代）

一个 epoch（代）是指整个数据集正向反向训练一次。它被用来提示模型的准确率并且不需要额外数据。本节我们将讲解 TensorFlow 里的 epochs，以及如何选择正确的 epochs。

下面是训练一个模型 10 代的 TensorFlow 代码

```

from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
import numpy as np
from helper import batches # Helper function created in Mini-batching section

def print_epoch_stats(epoch_i, sess, last_features, last_labels):
    """
    Print cost and validation accuracy of an epoch
    """
    current_cost = sess.run(
        cost,
        feed_dict={features: last_features, labels: last_labels})
    valid_accuracy = sess.run(
        accuracy,
        feed_dict={features: valid_features, labels: valid_labels})
    print('Epoch: {:<4} - Cost: {:<8.3} Valid Accuracy: {:<5.3}'.format(
        epoch_i,
        current_cost,
        valid_accuracy))

n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# Import MNIST data
mnist = input_data.read_data_sets('/datasets/ud730/mnist', one_hot=True)

# The features are already scaled and the data is shuffled
train_features = mnist.train.images
valid_features = mnist.validation.images
test_features = mnist.test.images

train_labels = mnist.train.labels.astype(np.float32)
valid_labels = mnist.validation.labels.astype(np.float32)
test_labels = mnist.test.labels.astype(np.float32)

# Features and Labels
features = tf.placeholder(tf.float32, [None, n_input])
labels = tf.placeholder(tf.float32, [None, n_classes])

# Weights & bias
weights = tf.Variable(tf.random_normal([n_input, n_classes]))
bias = tf.Variable(tf.random_normal([n_classes]))

# Logits - xW + b
logits = tf.add(tf.matmul(features, weights), bias)

# Define loss and optimizer
learning_rate = tf.placeholder(tf.float32)

```

```

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(cost)

# Calculate accuracy
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

init = tf.global_variables_initializer()

batch_size = 128
epochs = 10
learn_rate = 0.001

train_batches = batches(batch_size, train_features, train_labels)

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

    # Training cycle
    for epoch_i in range(epochs):

        # Loop over all batches
        for batch_features, batch_labels in train_batches:
            train_feed_dict = {
                features: batch_features,
                labels: batch_labels,
                learning_rate: learn_rate}
            sess.run(optimizer, feed_dict=train_feed_dict)

        # Print cost and validation accuracy of an epoch
        print_epoch_stats(epoch_i, sess, batch_features, batch_labels)

    # Calculate accuracy for test dataset
    test_accuracy = sess.run(
        accuracy,
        feed_dict={features: test_features, labels: test_labels})

print('Test Accuracy: {}'.format(test_accuracy))

```

Running the code will output the following:

```

Epoch: 0   - Cost: 11.0   Valid Accuracy: 0.204
Epoch: 1   - Cost: 9.95  Valid Accuracy: 0.229
Epoch: 2   - Cost: 9.18  Valid Accuracy: 0.246
Epoch: 3   - Cost: 8.59  Valid Accuracy: 0.264
Epoch: 4   - Cost: 8.13  Valid Accuracy: 0.283
Epoch: 5   - Cost: 7.77  Valid Accuracy: 0.301
Epoch: 6   - Cost: 7.47  Valid Accuracy: 0.316
Epoch: 7   - Cost: 7.2   Valid Accuracy: 0.328
Epoch: 8   - Cost: 6.96  Valid Accuracy: 0.342
Epoch: 9   - Cost: 6.73  Valid Accuracy: 0.36
Test Accuracy: 0.3801000118255615

```

每个 epoch 都试图走向一个低 cost，得到一个更好的准确率。

模型直到 Epoch 9 准确率都一直有提升，让我们把 epochs 的数字提高到 100。

```

...
Epoch: 79 - Cost: 0.111 Valid Accuracy: 0.86
Epoch: 80 - Cost: 0.11  Valid Accuracy: 0.869
Epoch: 81 - Cost: 0.109 Valid Accuracy: 0.869
....
Epoch: 85 - Cost: 0.107 Valid Accuracy: 0.869
Epoch: 86 - Cost: 0.107 Valid Accuracy: 0.869
Epoch: 87 - Cost: 0.106 Valid Accuracy: 0.869
Epoch: 88 - Cost: 0.106 Valid Accuracy: 0.869
Epoch: 89 - Cost: 0.105 Valid Accuracy: 0.869
Epoch: 90 - Cost: 0.105 Valid Accuracy: 0.869
Epoch: 91 - Cost: 0.104 Valid Accuracy: 0.869
Epoch: 92 - Cost: 0.103 Valid Accuracy: 0.869
Epoch: 93 - Cost: 0.103 Valid Accuracy: 0.869
Epoch: 94 - Cost: 0.102 Valid Accuracy: 0.869
Epoch: 95 - Cost: 0.102 Valid Accuracy: 0.869

```

```
Epoch: 96 - Cost: 0.101 Valid Accuracy: 0.869
Epoch: 97 - Cost: 0.101 Valid Accuracy: 0.869
Epoch: 98 - Cost: 0.1 Valid Accuracy: 0.869
Epoch: 99 - Cost: 0.1 Valid Accuracy: 0.869
Test Accuracy: 0.869600006198883
```

从上述输出来看，在 epoch 80 的时候，模型的验证准确率就不提升了。让我们看看提升学习率会怎样。

learn_rate = 0.1

```
Epoch: 76 - Cost: 0.214 Valid Accuracy: 0.752
Epoch: 77 - Cost: 0.21 Valid Accuracy: 0.756
Epoch: 78 - Cost: 0.21 Valid Accuracy: 0.756
...
Epoch: 85 - Cost: 0.207 Valid Accuracy: 0.756
Epoch: 86 - Cost: 0.209 Valid Accuracy: 0.756
Epoch: 87 - Cost: 0.205 Valid Accuracy: 0.756
Epoch: 88 - Cost: 0.208 Valid Accuracy: 0.756
Epoch: 89 - Cost: 0.205 Valid Accuracy: 0.756
Epoch: 90 - Cost: 0.202 Valid Accuracy: 0.756
Epoch: 91 - Cost: 0.207 Valid Accuracy: 0.756
Epoch: 92 - Cost: 0.204 Valid Accuracy: 0.756
Epoch: 93 - Cost: 0.206 Valid Accuracy: 0.756
Epoch: 94 - Cost: 0.202 Valid Accuracy: 0.756
Epoch: 95 - Cost: 0.2974 Valid Accuracy: 0.756
Epoch: 96 - Cost: 0.202 Valid Accuracy: 0.756
Epoch: 97 - Cost: 0.2996 Valid Accuracy: 0.756
Epoch: 98 - Cost: 0.203 Valid Accuracy: 0.756
Epoch: 99 - Cost: 0.2987 Valid Accuracy: 0.756
Test Accuracy: 0.755600053882599
```

看来学习率提升的太多了，最终准确率更低了。准确率也更早的停止了改进。我们还是用之前的学习率，把 epochs 改成 80

```
Epoch: 65 - Cost: 0.122 Valid Accuracy: 0.868
Epoch: 66 - Cost: 0.121 Valid Accuracy: 0.868
Epoch: 67 - Cost: 0.12 Valid Accuracy: 0.868
Epoch: 68 - Cost: 0.119 Valid Accuracy: 0.868
Epoch: 69 - Cost: 0.118 Valid Accuracy: 0.868
Epoch: 70 - Cost: 0.118 Valid Accuracy: 0.868
Epoch: 71 - Cost: 0.117 Valid Accuracy: 0.868
Epoch: 72 - Cost: 0.116 Valid Accuracy: 0.868
Epoch: 73 - Cost: 0.115 Valid Accuracy: 0.868
Epoch: 74 - Cost: 0.115 Valid Accuracy: 0.868
Epoch: 75 - Cost: 0.114 Valid Accuracy: 0.868
Epoch: 76 - Cost: 0.113 Valid Accuracy: 0.868
Epoch: 77 - Cost: 0.113 Valid Accuracy: 0.868
Epoch: 78 - Cost: 0.112 Valid Accuracy: 0.868
Epoch: 79 - Cost: 0.111 Valid Accuracy: 0.868
Epoch: 80 - Cost: 0.111 Valid Accuracy: 0.869
Test Accuracy: 0.86909999418258667
```

准确率只到 0.86。这有可能是学习率太高造成的。降低学习率需要更多的 epoch，但是可以最终得到更好的准确率。

在接下来的 TensorFlow Lab 里，你有机会选择你自己的学习率，epoch 数，batch size 来提升模型的准确率。

11. TensorFlow 实验准备：NotMNIST

TensorFlow 神经网络实验室



TensorFlow 实验室

我们准备了一个 Jupyter Notebook，它将引导你完成在 TensorFlow 中创建单层神经网络。你将实现数据规范化，然后使用 TensorFlow 构建和训练网络。

Notebook

Notebook 和相关文件都可以从 我们的 Github 仓库 获得。你可以克隆这一 Github 仓库，或者直接下载为其 Zip 压缩文件。

使用 Git 克隆仓库。

```
git clone https://github.com/udacity/deep-learning.git
```

如果你对 Git 和 GitHub 不熟悉，我强烈建议你查看我们的课程。如果你不想使用 Git，则可以将该仓库以 Zip 文件形式下载下来。你可以在这里找到这个仓库。

查看 Notebook

在 notebook 文件目录中，启动你的 Jupyter notebook 服务器

```
jupyter notebook
```

你的页面应该会跳转到一个浏览器窗口。如果没有，请转到 <http://localhost:8888/tree>。如果你的其他 Notebook 服务器正在运行，端口号可能会有所不同。如果你找不到合适的服务器，请尝试 8889 而不是 8888。

你应该会看到 `intro_to_tensorflow.ipynb` notebook，这是你即将要工作的 notebook。notebook 有 3 个问题要解决：

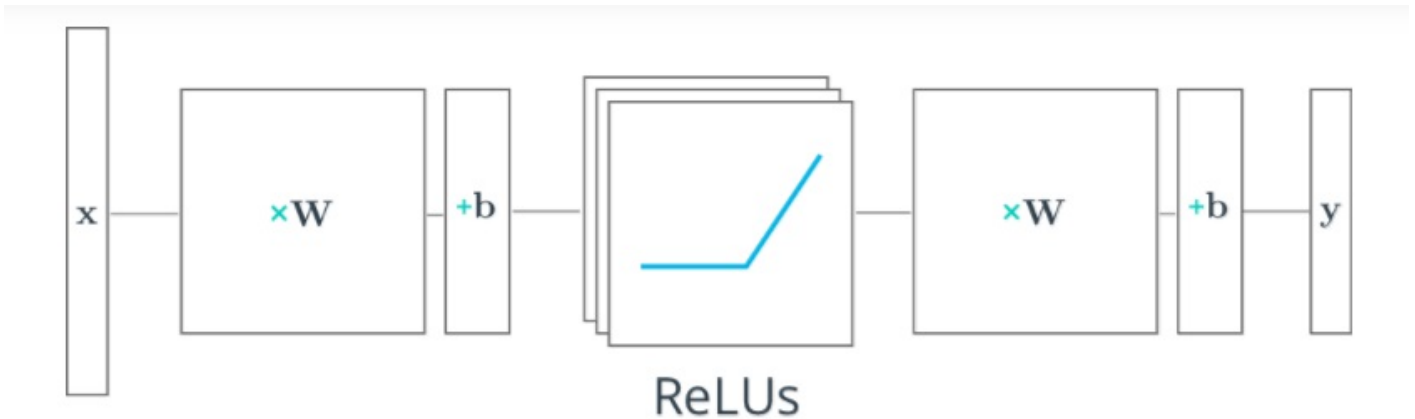
问题1: 标准化功能
问题2: 使用 TensorFlow 操作来创建特征，标签，重量和偏差张量
问题3: 调整学习速度，步数和批大小以获得最佳准确性

这是一个自我评估的实验室。你可以将你的答案和解决方案在这里进行比较。如果你在完成实验时遇到任何困难，优达学城 会提供一些服务来回答你可能遇到的任何问题。

帮助指南

请记住，你可以从导师，论坛（单击教室左侧的链接）或 Slack 频道获得学习帮助。你还可以查看以前课程中的页面来回顾学习到的知识。

13. 两层神经网络



在之前的课程和实验室中，你已经学会了如何建立一层神经网络。现在，你将学习如何使用TensorFlow构建多层神经网络。添加一个隐藏层到网络让它建模更复杂的功能。而且，在隐藏层上使用非线性激活函数可以模拟非线性函数。

我们将学习在TensorFlow中实现的第一件事是ReLU隐藏层。ReLU是一个非线性函数，又称修正线性单元。ReLU函数对于所有负的输入，返回为 0，ReLU 函数对所有正的输入，返回 x ；所有 $x > 0$ 的输入，返回 xx 。

和之前一样，下面的节点将建立在深度学习课程基础上。如果你需要重新理清思绪，你可以回顾一遍。

ReLU: 参考本章训练神经网络-其他激活函数 Feedforward: 参考本章神经网络简介-前向反馈 Dropout: 参考本章训练神经网络-Dropout

14. TensorFlow ReLUs

TensorFlow 提供了 ReLU 函数 `tf.nn.relu()`，如下所示：

```
# Hidden Layer with ReLU activation function
# 隐藏层用 ReLU 作为激活函数
hidden_layer = tf.add(tf.matmul(features, hidden_weights), hidden_biases)
hidden_layer = tf.nn.relu(hidden_layer)

output = tf.add(tf.matmul(hidden_layer, output_weights), output_biases)
```

上面的代码把`tf.nn.relu()` 放到隐藏层，就像开关一样把负权重关掉了。在激活函数之后，添加像输出层这样额外的层，就把模型变成了非线性函数。这个非线性的特征使得网络可以解决更复杂的问题。

15. TensorFlow 中的深度神经网络

你已经学过了如何用 TensorFlow 构建一个逻辑分类器。现在你会学到如何用逻辑分类器来构建一个深度神经网络。

详细指导

接下来我们看看如何用 TensorFlow 来构建一个分类器来对 MNIST 数字进行分类。如果你要在自己电脑上跑这个代码，文件在这儿。你可以在Aymeric Damien的 GitHub repository里找到更多的 TensorFlow 的例子。

代码

TensorFlow MNIST

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(".", one_hot=True, reshape=False)
```

你可以使用 TensorFlow 提供的 MNIST 数据集，他把分批和 one-hot 编码（独热码）都帮你处理好了。

学习参数 Learning Parameters

```
import tensorflow as tf
```

```
# 参数 Parameters
learning_rate = 0.001
training_epochs = 20
batch_size = 128 # 如果没有足够内存，可以降低 batch size
display_step = 1
```

```
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
```

这里的关注点是神经网络的架构，不是调参，所以这里直接给你了学习的参数。

隐藏层参数 Hidden Layer Parameters

```
n_hidden_layer = 256 # layer number of features 特征的层数
```

`n_hidden_layer` 决定了神经网络隐藏层的大小。也被称作层的宽度。

权重和偏置项 Weights and Biases

```
# Store layers weight & bias
# 层权重和偏置项的储存
weights = {
    'hidden_layer': tf.Variable(tf.random_normal([n_input, n_hidden_layer])),
    'out': tf.Variable(tf.random_normal([n_hidden_layer, n_classes]))
}
biases = {
    'hidden_layer': tf.Variable(tf.random_normal([n_hidden_layer])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

深度神经网络有多个层，每个层有自己的权重和偏置项。`'hidden_layer'` 的权重和偏置项只属于隐藏层（`hidden_layer`），`'out'` 的权重和偏置项只属于输出层（`output layer`）。如果神经网络比这更深，那每一层都有权重和偏置项。

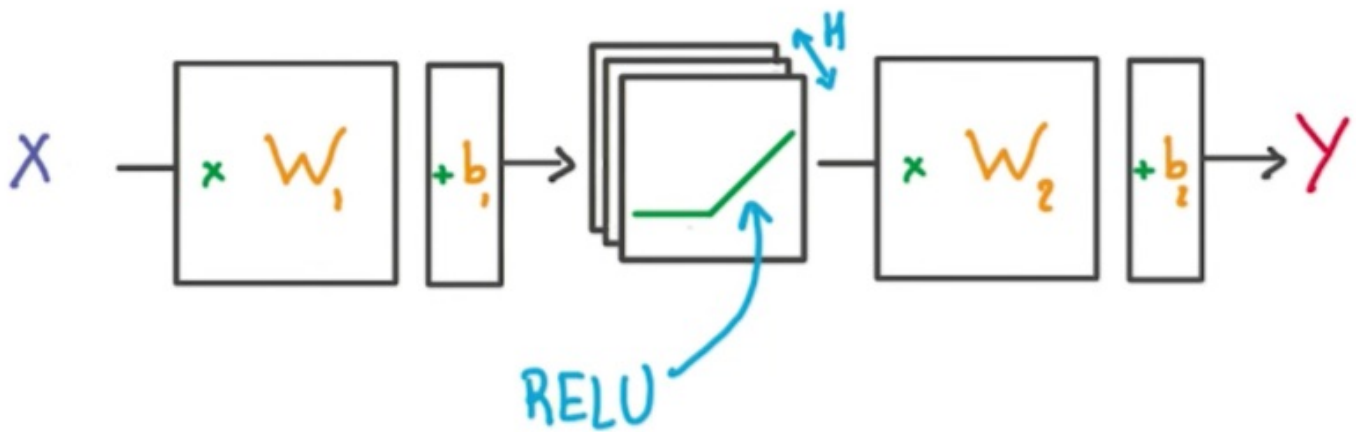
输入 Input

```
# tf Graph input
x = tf.placeholder("float", [None, 28, 28, 1])
y = tf.placeholder("float", [None, n_classes])

x_flat = tf.reshape(x, [-1, n_input])
```

MNIST 数据集是由 28px * 28px 单通道图片组成。`tf.reshape()`函数把 28px * 28px 的矩阵转换成了 784px * 1px 的单向向量 `x`。

多层感知器 Multilayer Perceptron



```
# Hidden layer with RELU activation
# ReLU作为隐藏层激活函数
layer_1 = tf.add(tf.matmul(x_flat, weights['hidden_layer']),\
    biases['hidden_layer'])
layer_1 = tf.nn.relu(layer_1)
# Output layer with linear activation
# 输出层的线性激活函数
logits = tf.add(tf.matmul(layer_1, weights['out']), biases['out'])
```

你之前已经见过

```
tf.add(tf.matmul(x_flat, weights['hidden_layer']), biases['hidden_layer'])
```

，也就是 $xw + b$ 。把线性函数与 ReLU 组合在一起，形成一个2层网络。

优化器 Optimizer

```
# Define loss and optimizer
# 定义误差值和优化器
cost = tf.reduce_mean(\
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)
```

这跟 Intro to TensorFlow lab 里用到的优化技术一样。

Session

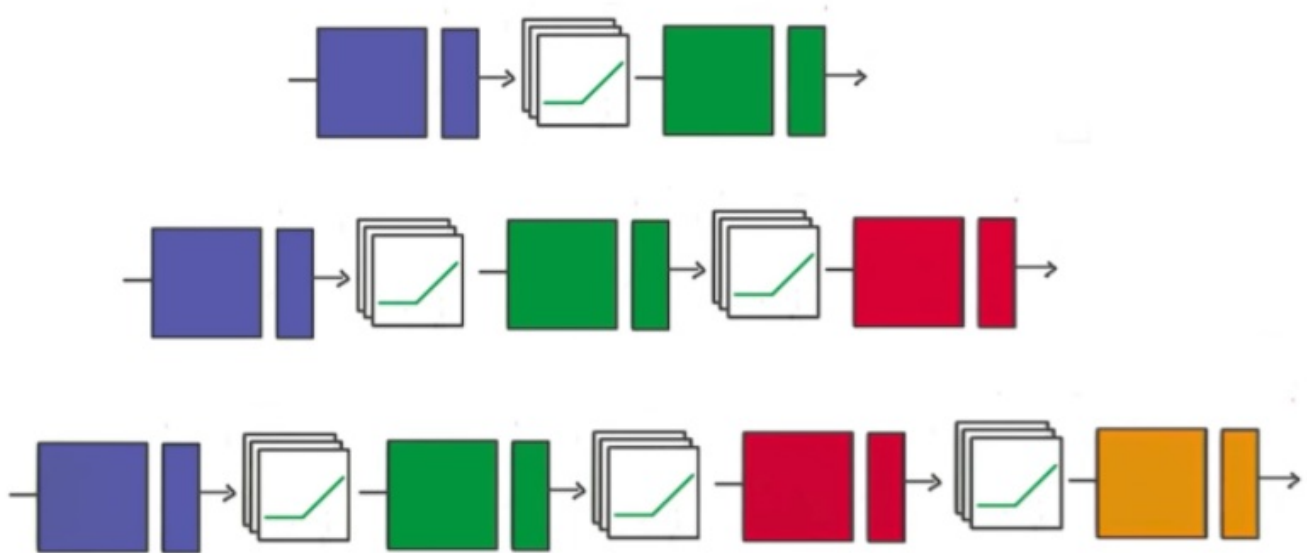
```
# Initializing the variables
# 初始化变量
init = tf.global_variables_initializer()

# Launch the graph
# 启动图
with tf.Session() as sess:
    sess.run(init)
    # Training cycle
    # 训练循环
    for epoch in range(training_epochs):
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        # 遍历所有 batch
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            # Run optimization op (backprop) and cost op (to get loss value)
            # 运行优化器进行反向传导、计算 cost (获取 loss 值)
```

```
sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
```

TensorFlow 中的 MNIST 库提供了分批接收数据的能力。调用`mnist.train.next_batch()`函数返回训练数据的一个子集。

深度神经网络



就是这样！从一层到两层很简单。向网络中添加更多层，可以让你解决更复杂的问题。

16. 保存和读取 TensorFlow 模型

训练一个模型的时间很长。但是如果你一旦关闭了 TensorFlow session，你所有训练的权重和偏置项都丢失了。如果你计划在之后重新使用这个模型，你需要重新训练！

幸运的是，TensorFlow 可以让你通过一个叫 `tf.train.Saver` 的类把你的进程保存下来。这个类可以把任何 `tf.Variable` 存到你的文件系统。

保存变量

让我们通过一个简单地例子来保存 weights 和 bias Tensors。第一个例子你只是存两个变量，后面会教你如何把一个实际模型的所有权重保存下来。

```
import tensorflow as tf

# The file path to save the data
# 文件保存路径
save_file = './model.ckpt'

# Two Tensor Variables: weights and bias
# 两个 Tensor 变量: 权重和偏置项
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))

# Class used to save and/or restore Tensor Variables
# 用来存取 Tensor 变量的类
saver = tf.train.Saver()

with tf.Session() as sess:
    # Initialize all the Variables
    # 初始化所有变量
    sess.run(tf.global_variables_initializer())

    # Show the values of weights and bias
    # 显示变量和权重
    print('Weights:')
    print(sess.run(weights))
    print('Bias:')
    print(sess.run(bias))
```

```
# Save the model
# 保存模型
saver.save(sess, save_file)
```

Weights:

```
[[-0.97990924 1.03016174 0.74119264]
```

```
[-0.82581609 -0.07361362 -0.86653847]]
```

Bias:

```
[ 1.62978125 -0.37812829 0.64723819]
```

weights 和 bias Tensors 用 `tf.truncated_normal()` 函数设定了随机值。用 `tf.train.Saver.save()` 函数把这些值被保存在 `save_file` 位置，命名为 "model.ckpt"，（".ckpt" 扩展名表示 "checkpoint"）。

如果你使用 TensorFlow 0.11.0RC1 或者更新的版本，还会生成一个包含了 TensorFlow graph 的文件 "model.ckpt.meta"。

加载变量

现在这些变量已经存好了，让我们把它们加载到新模型里。

```
# Remove the previous weights and bias
# 移除之前的权重和偏置项
tf.reset_default_graph()

# Two Variables: weights and bias
# 两个变量：权重和偏置项
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))

# Class used to save and/or restore Tensor Variables
# 用来存取 Tensor 变量的类
saver = tf.train.Saver()

with tf.Session() as sess:
    # Load the weights and bias
    # 加载权重和偏置项
    saver.restore(sess, save_file)

    # Show the values of weights and bias
    # 显示权重和偏置项
    print('Weight:')
    print(sess.run(weights))
    print('Bias:')
    print(sess.run(bias))
```

Weights:

```
[[-0.97990924 1.03016174 0.74119264]
```

```
[-0.82581609 -0.07361362 -0.86653847]]
```

Bias:

```
[ 1.62978125 -0.37812829 0.64723819]
```

注意，你依然需要在 Python 中创建 weights 和 bias Tensors。`tf.train.Saver.restore()` 函数把之前保存的数据加载到 weights 和 bias 当中。

因为 `tf.train.Saver.restore()` 设定了 TensorFlow 变量，这里你不需要调用 `tf.global_variables_initializer()`了。

保存一个训练好的模型

让我们看看如何训练一个模型并保存它的权重。

从一个模型开始：

```
# Remove previous Tensors and Operations
# 移除之前的 Tensors 和运算
```

```

tf.reset_default_graph()

from tensorflow.examples.tutorials.mnist import input_data
import numpy as np

learning_rate = 0.001
n_input = 784 # MNIST 数据输入 (图片尺寸: 28*28)
n_classes = 10 # MNIST 总计类别 (数字 0-9)

# Import MNIST data
# 加载 MNIST 数据
mnist = input_data.read_data_sets('.', one_hot=True)

# Features and Labels
# 特征和标签
features = tf.placeholder(tf.float32, [None, n_input])
labels = tf.placeholder(tf.float32, [None, n_classes])

# Weights & bias
# 权重和偏置项
weights = tf.Variable(tf.random_normal([n_input, n_classes]))
bias = tf.Variable(tf.random_normal([n_classes]))

# Logits - xW + b
logits = tf.add(tf.matmul(features, weights), bias)

# Define loss and optimizer
# 定义损失函数和优化器
cost = tf.reduce_mean(\
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)

# Calculate accuracy
# 计算准确率
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

```

让我们训练模型并保存权重:

```

import math

save_file = './train_model.ckpt'
batch_size = 128
n_epochs = 100

saver = tf.train.Saver()

# Launch the graph
# 启动图
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # Training cycle
    # 训练循环
    for epoch in range(n_epochs):
        total_batch = math.ceil(mnist.train.num_examples / batch_size)

        # Loop over all batches
        # 遍历所有 batch
        for i in range(total_batch):
            batch_features, batch_labels = mnist.train.next_batch(batch_size)
            sess.run(
                optimizer,
                feed_dict={features: batch_features, labels: batch_labels})

        # Print status for every 10 epochs
        # 每运行10个 epoch 打印一次状态
        if epoch % 10 == 0:
            valid_accuracy = sess.run(
                accuracy,
                feed_dict={
                    features: mnist.validation.images,
                    labels: mnist.validation.labels})
            print('Epoch {:<3} - Validation Accuracy: {}'.format(

```

```
        epoch,
        valid_accuracy))

# Save the model
# 保存模型
saver.save(sess, save_file)
print('Trained Model Saved.')
```

Epoch 0 - Validation Accuracy: 0.06859999895095825

Epoch 10 - Validation Accuracy: 0.20239999890327454

Epoch 20 - Validation Accuracy: 0.36980000138282776

Epoch 30 - Validation Accuracy: 0.48820000886917114

Epoch 40 - Validation Accuracy: 0.5601999759674072

Epoch 50 - Validation Accuracy: 0.6097999811172485

Epoch 60 - Validation Accuracy: 0.642599999046326

Epoch 70 - Validation Accuracy: 0.6733999848365784

Epoch 80 - Validation Accuracy: 0.6916000247001648

Epoch 90 - Validation Accuracy: 0.7113999724388123

Trained Model Saved.

加载训练好的模型

让我们从磁盘中加载权重和偏置项，验证测试集准确率。

```
saver = tf.train.Saver()

# Launch the graph
# 加载图
with tf.Session() as sess:
    saver.restore(sess, save_file)

    test_accuracy = sess.run(
        accuracy,
        feed_dict={features: mnist.test.images, labels: mnist.test.labels})

    print('Test Accuracy: {}'.format(test_accuracy))
```

Test Accuracy: 0.7229999899864197

就是这样！你现在知道如何保存再加载一个 TensorFlow 的训练模型了。下一章节让我们看看如何把权重和偏置项加载到修改过的模型中。

17. 参数微调 把权重和偏置项加载到新模型中

很多时候你想调整，或者说“微调”一个你已经训练并保存了的模型。但是，把保存的变量直接加载到已经修改过的模型会产生错误。让我们看看如何解决这个问题。

命名报错

TensorFlow 对 Tensor 和计算使用一个叫 name 的字符串标识器，如果没有定义 name，TensorFlow 会自动创建一个。TensorFlow 会把第一个节点命名为_，把后续的命名为_。让我们看看这对加载一个有不同顺序权重和偏置项的模型有哪些影响：

```
import tensorflow as tf

# Remove the previous weights and bias
# 移除先前的权重和偏置项
tf.reset_default_graph()

save_file = 'model.ckpt'
```



```

# Two Tensor Variables: weights and bias
# 两个 Tensor 变量: 权重和偏置项
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))

saver = tf.train.Saver()

# Print the name of Weights and Bias
# 打印权重和偏置项的名字
print('Save Weights: {}'.format(weights.name))
print('Save Bias: {}'.format(bias.name))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.save(sess, save_file)

# Remove the previous weights and bias
# 移除之前的权重和偏置项
tf.reset_default_graph()

# Two Variables: weights and bias
# 两个变量: 权重和偏置项
bias = tf.Variable(tf.truncated_normal([3]))
weights = tf.Variable(tf.truncated_normal([2, 3]))

saver = tf.train.Saver()

# Print the name of Weights and Bias
# 打印权重和偏置项的名字
print('Load Weights: {}'.format(weights.name))
print('Load Bias: {}'.format(bias.name))

with tf.Session() as sess:
    # Load the weights and bias - ERROR
    # 加载权重和偏置项 - 报错
    saver.restore(sess, save_file)

```

上述代码会有下列输出: Save Weights: Variable:0

Save Bias: Variable_1:0

Load Weights: Variable_1:0

Load Bias: Variable:0

...

InvalidArgumentError (see above for traceback): Assign requires shapes of both tensors to match.

...

你注意到, `weights` 和 `bias` 的 `name` 属性与你保存的模型不同。这是为什么代码报“Assign requires shapes of both tensors to match”这个错误。`saver.restore(sess, save_file)` 代码试图把权重数据加载到 `bias` 里, 把偏置项数据加载到 `weights` 里。

与其让 TensorFlow 来设定 `name` 属性, 不如让我们来手动设定:

```

import tensorflow as tf

tf.reset_default_graph()

save_file = 'model.ckpt'

# Two Tensor Variables: weights and bias
# 两个 Tensor 变量: 权重和偏置项
weights = tf.Variable(tf.truncated_normal([2, 3]), name='weights_0')
bias = tf.Variable(tf.truncated_normal([3]), name='bias_0')

saver = tf.train.Saver()

# Print the name of Weights and Bias
# 打印权重和偏置项的名称
print('Save Weights: {}'.format(weights.name))
print('Save Bias: {}'.format(bias.name))

```

```

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.save(sess, save_file)

# Remove the previous weights and bias
# 移除之前的权重和偏置项
tf.reset_default_graph()

# Two Variables: weights and bias
# 两个变量: 权重和偏置项
bias = tf.Variable(tf.truncated_normal([3]), name='bias_0')
weights = tf.Variable(tf.truncated_normal([2, 3]), name='weights_0')

saver = tf.train.Saver()

# Print the name of Weights and Bias
# 打印权重和偏置项的名称
print('Load Weights: {}'.format(weights.name))
print('Load Bias: {}'.format(bias.name))

with tf.Session() as sess:
    # Load the weights and bias - No Error
    # 加载权重和偏置项 - 没有报错
    saver.restore(sess, save_file)

print('Loaded Weights and Bias successfully.')

```

Save Weights: weights_0:0

Save Bias: bias_0:0

Load Weights: weights_0:0

Load Bias: bias_0:0

Loaded Weights and Bias successfully.

这次没问题! Tensor 名称匹配正确, 数据被正确加载。

18. TensorFlow Dropout

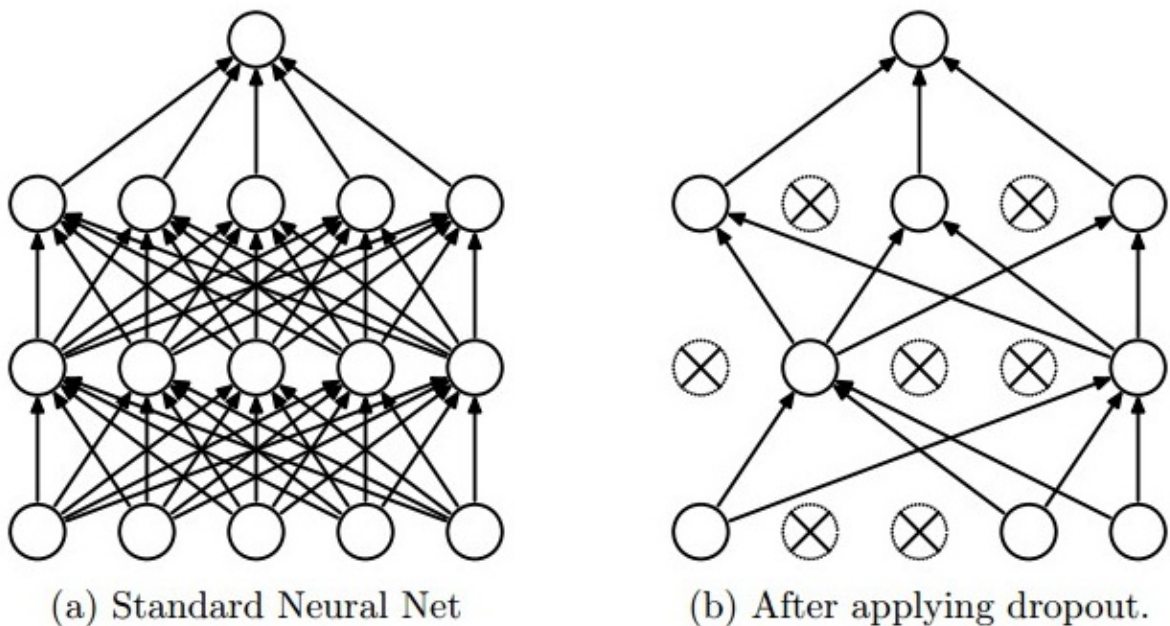


图 1 : 来自论文 "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" (<https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>)

Dropout 是一个降低过拟合的正则化技术。它在网络中暂时的丢弃一些单元（神经元），以及与它们的前后相连的所有节点。图 1 是 dropout 的工作示意图。

TensorFlow 提供了一个 `tf.nn.dropout()` 函数，你可以用来实现 dropout。

让我们来看一个 `tf.nn.dropout()` 的使用例子。

```
keep_prob = tf.placeholder(tf.float32) # probability to keep units

hidden_layer = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer = tf.nn.relu(hidden_layer)
hidden_layer = tf.nn.dropout(hidden_layer, keep_prob)

logits = tf.add(tf.matmul(hidden_layer, weights[1]), biases[1])
```

上面的代码展示了如何在神经网络中应用 dropout。 `tf.nn.dropout()` 函数有两个参数：

hidden_layer: 你要应用 dropout 的 tensor **keep_prob:** 任何一个给定单元的留存率（没有被丢弃的单元） `keep_prob` 可以让你调整丢弃单元的数量。为了补偿被丢弃的单元，`tf.nn.dropout()` 把所有保留下来的单元（没有被丢弃的单元）* $1/\text{keep_prob}$

在训练时，一个好的 `keep_prob` 初始值是 0.5。

在测试时，把 `keep_prob` 值设为 1.0，这样保留所有的单元，最大化模型的能力。