深度学习的实用层面

本文尝试对吴恩达深度学习课程，第二节《改善深层神经网络》，第一周课程《深度学习的实用层面》做一个简单总结，主要包括以下几个方面的内容：

1、训练集、开发集、测试集（Train,Dev,Test Set）

2、偏差、方差分析

3、正则化

L2正则化

L1正则化

作为约束的范数惩罚

dropout

early stopping

参数绑定与参数共享

4、神经网络的权重初始化

训练集、开发集、测试集（Train,Dev,Test Set）

机器学习设计很多超参数的选择，是一个高度迭代的过程，首先有一个初步的想法，编码实现，进行试验，观察结果，然后修改自己的想法，再次进行编码和试验，如此往复，直到达到期望的目标。循环该过程的效率是决定项目进展速度的一个关键因素，而创建高质量的训练数据集、 验证集和测试集也有助于提高循环的效率。

小数据时代七三分或者六二二分是个不错的选择。

在大数据时代各个数据集的比例可能需要变成 98%：1%：1%（1% 也足够完成验证和测试），甚至训练集的比例更大。

有些时候训练集和验证集、测试集的数据有所不同时，比如训练集的图片高像素、高质 量，而验证集和测试集则像素较低，那么有一条经验法则是 ：

确保验证集和测试集来自于同一分布

但由于深度学习需要大量的训练数据，为了获取更大规模的训练数据集，可能会进行 网页抓取，代价是训练集数据与验证集和测试集数据有可能不是来自于同一分布。 还有一种情况，就算没有测试集也是可以的，测试集的目的是对最终所选定的神经网络 系统做出无偏估计，如果不需要无偏估计也可以不设置测试集。 所以搭建训练验证集合测试集能够加速神经网络的集成，也可以更有效地衡量算法的偏 差和方差，从而帮助我们更高效地选择合适的方法来优化你的算法。

偏差、方差分析（Bias/Variance）

在《从VC维角度理解正则化与偏差方差权衡》一文中对偏差方差权衡的理论做了比较充分的说明，这里只列出公式，并给出简单说明：$E_{out} \leq E_{in} + \sqrt{\frac{8}{N} log \frac{4(2N)^{d_{vc}}}{\delta}}$

其中E_{out}为泛化误差，E_{in}为训练误差，N为样本数量，δ为置信度，d_{vc}为假设空间的VC维（大部分时候，d_{vc}与模型参数量成正比），d_{vc}越大，偏差E_{in}会越小，但方差项 $\sqrt{\frac{8}{N} log \frac{4(2N)^{d_{vc}}}{\delta}}$ 会比较大，如果d_{vc}太小，那么方差项 $\sqrt{\frac{8}{N} log \frac{4(2N)^{d_{vc}}}{\delta}}$ 会比较小，但是偏差E_{in}可能会比较大，这就是偏差方差权衡的数学基础。

高偏差

如果出现偏差项E_{in}比较大,那么则需要比较大d_{vc},有以下三种方式可以解决高偏差的问题：

1、可以训练一个更大的网络，

2、训练更长时间，选择更好的优化方法，选择更好的激活函数（训练更长时间，模型会与训练集更吻合，使E_{in}更小）

3、选择不同的网络架构（同样的参数量，更优秀的网络架构，可能有更丰富的表达能力）

优化方法、激活函数，网络架构将会在后续文章中专门论述，这里就不再说明。

高方差

如果出现偏差项E_{in}比较小,但E_{out}比较大情况，那么则需要比较小d_{vc}，或者更多的数据量,有以下三种方式可以解决高方差差的问题：

1、收集更多的数据

2、使用规则化（正则化是调节假设空间d_{vc}的有效手段，《从VC维角度理解正则化与偏差方差权衡》有说明）

3、dropout（dropout减弱overfitting的影响，其实与正则化不太一致，个人认为不应该理解为正则化，事实上这是一种通过集成来减弱overftting的手段，关于其理论基础会在下文对dropout的说明中详细解释）

4、early stopping

5、选择不同的网络架构（但优秀的网络架构可以有效减弱overfitting，例如CNN稀疏连接和参数共享机制可以在保持模型表达能力的基础上，有效减小参数量，减弱overfitting）

高方差与高偏差

对模型训练时，可能会出现，对整体趋势没有很好的拟合，但对局部又出现过拟合的情况，此时，就会同时出现高偏差与高偏差的情况。

正则化(regularization)

《从VC维角度理解正则化与偏差方差权衡》一文中已经详细说明，正则化是调节假设空间d_{vc}的有效手段，接下来详细介绍一下L2正则化，L1正则化，并说明为什么L2正则化后的权值会比较小，而L1正则化，则会产生大量稀疏权值，最后使用python 实现神经网络中的L2正则化（使用最为广泛）。

L2 正则化

$L_{reg} = L_{org} + \frac{\lambda}{2N} \sum_l \sum_k \sum_j (W_{kj}^l)^2$

其中 L_{org}为不做规则化的cost函数，L_{reg} 为正则化后的cost函数，W_{kj}^{l}为连接第l层第k个神经元与第l-1层第j个神经元的权重，（需要说明的是，由于W一般是很高维的向量，偏置b相对W可忽略，因此，一般不加入到规则化的cost函数中）

经过推导，可得到更新过程如下：

$$W^{t+1} = (1-\eta\lambda)W^{t} - \eta\frac{\alpha L_{reg}}{\alpha W^{t}}$$

从上式可以看到：更新W时，除了梯度项，还会还会减去一个$\eta\lambda W^{t}$,如果W^{t}比较大，那么会减去一个比较大的值，如果W^{t}比较小，则会减去一个比较小的值，因此经过多轮更新后，W平均都比较小，但0值也很小。

L1 正则化

$$L_{reg} = L_{org} + \frac{\lambda}{2N}\sum_{l}\sum_{k}\sum_{j}|W_{kj}^{l}|$$

经过推导，可得到更新过程如下：

$$W^{t+1} = W^{t} - \eta\frac{\alpha L_{reg}}{\alpha W^{t}} - \eta\lambda sign(W^{t})$$

上式中sign表示取正负号，从上式可以看到，更新W时，除了梯度项，如果W为正，那么会固定减去$\eta\lambda$,如果W为负，则会加上一个$\eta\lambda$,因此，经过多轮更新后，会有大量W变为0，即产生稀疏解。

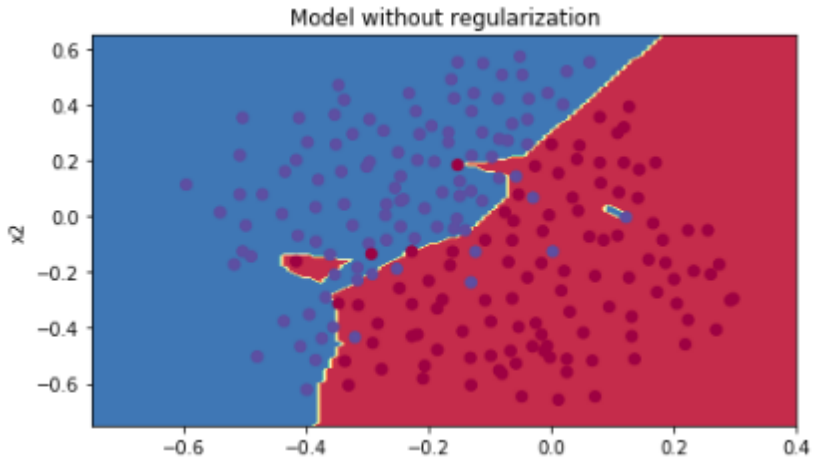
假设模型有三层，权值分别为W^{1}, W^{2}, W^{3},不做规则化的cost的函数为compute_cost(A3, Y)

则：添加正则化后，总的cost为$cost_{reg}$,python实现如下

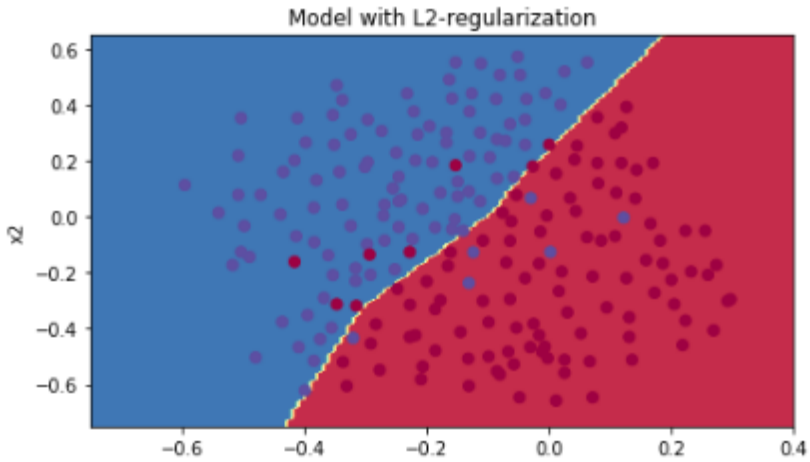
```
def compute_cost_with_regularization(A3, Y, parameters, lambd):
"""
Implement the cost function with L2 regularization.

Arguments:
A3 -- post-activation, output of forward propagation, of shape (output size, number of examples)
Y -- "true" labels vector, of shape (output size, number of examples)
parameters -- python dictionary containing parameters of the model

Returns:
cost - value of the regularized loss function (formula (2))
"""
m = Y.shape[1]
W1 = parameters["W1"]
W2 = parameters["W2"]
W3 = parameters["W3"]

cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost

### START CODE HERE ### (approx. 1 line)
L2_regularization_cost =
(np.sum(np.square(W1))+np.sum(np.square(W2))+np.sum(np.square(W3)))*lambd/m/2
### END CODER HERE ###

cost = cross_entropy_cost + L2_regularization_cost

return cost
```

更新过程为：

```
def backward_propagation_with_regularization(X, Y, cache, lambd):
"""
Implements the backward propagation of our baseline model to which we added an L2
regularization.

Arguments:
X -- input dataset, of shape (input size, number of examples)
Y -- "true" labels vector, of shape (output size, number of examples)
cache -- cache output from forward_propagation()
lambd -- regularization hyperparameter, scalar

Returns:
gradients -- A dictionary with the gradients with respect to each parameter, activation and pre-
activation variables
"""

m = X.shape[1]
(Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache

dZ3 = A3 - Y

### START CODE HERE ### (approx. 1 line)
dW3 = 1./m * np.dot(dZ3, A2.T) + lambd/m*W3
### END CODE HERE ###
db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

dA2 = np.dot(W3.T, dZ3)
dZ2 = np.multiply(dA2, np.int64(A2 > 0))   #relu 函数导数
### START CODE HERE ### (approx. 1 line)
dW2 = 1./m * np.dot(dZ2, A1.T) + lambd/m*W2
### END CODE HERE ###
db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
### START CODE HERE ### (approx. 1 line)
dW1 = 1./m * np.dot(dZ1, X.T) + lambd/m*W1
### END CODE HERE ###
db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)

gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,"dA2": dA2,
             "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
             "dZ1": dZ1, "dW1": dW1, "db1": db1}

return gradients
```



不做正则化分类边界



L2正则化后分类边界

上边两张图分别为不做正则化和L2正则化后的分类边界，可以明显看到，L2正则化后边界更加平滑，而原始分类边界则存在许多异常点，存在过拟合的风险。

不做正则化训练集准确率可以达到94.8 %，但测试集准确率只有91.5%，存在明显的过拟合问题。

L2正则化后训练集准确率为93.8%，测试集的准确率为93%，基本不存在过拟合，测试集准确率明显提高。

作为约束的范数惩罚

L2和L1正则化是一种广泛使用的正则化技术，事实上，我们可以在原始的目标函数上添加一系列惩罚项，并通过广义拉格朗日函数，将问题变为无约束的最优化问题，例如（SVM）

Dropout

L2,L1正则化通过降低假设空间d_{vc}，从而避免过拟合，Dropout能够有效减弱过拟合的风险是因为它相当于集成了很多个不同的神经网络，是一种集成(ensemble)的方法。

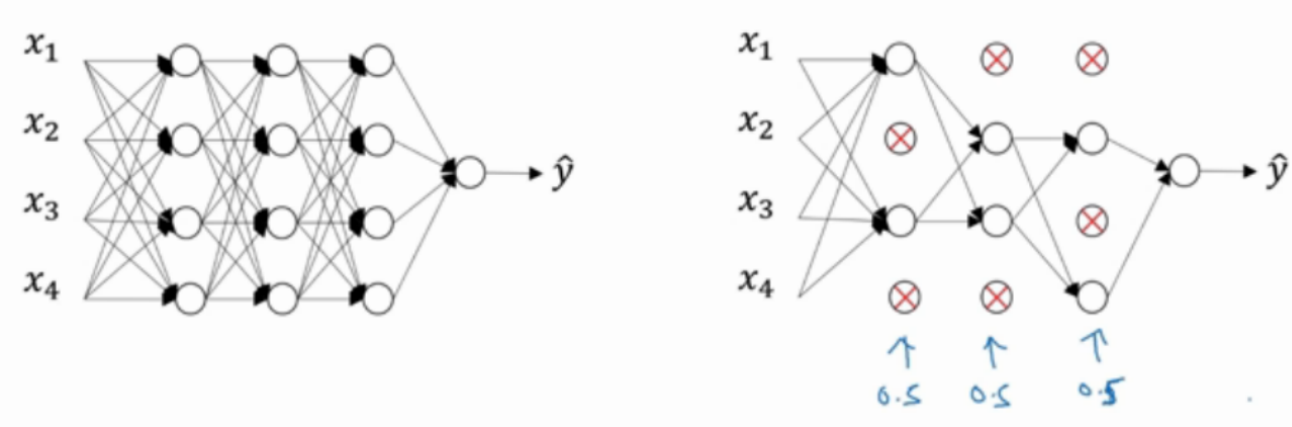
本节的内容主要包括以下几个方面

1、Dropout原理

2、使用python实现Dropout

3、说明为什么Dropout可以有效减弱overfitting

Dropout原理



如上图所示，Dropout 可以施加在任何隐藏层，假设对某一隐藏层施加keep_prob的dropout，则每一次迭代过程中，使该层中的每个神经元以(1 - keep_prob)的比例失活（shut down）（失活的神经元不参与本次参数迭代的forward和backward）。

python 实现Dropout

本文中Dropout实现采用invert dropout

带dropout的 forward

```
D1 = np.random.rand(A1.shape[0],A1.shape[1])
# Step 1: initialize matrix D1 = np.random.rand(..., ...)
D1 = (D1<keep_prob)
# Step 2: convert entries of D1 to 0 or 1 (using keep_prob as the threshold)
A1 = np.multiply(A1,D1)
# Step 3: shut down some neurons of A1
A1 = np.divide(A1,keep_prob)
# Step 4: scale the value of neurons that haven't been shut down
```

前两步生成用于控制失活的mask：D1

第三步 执行A1与D1的元素相乘，使A1以一定比例失活

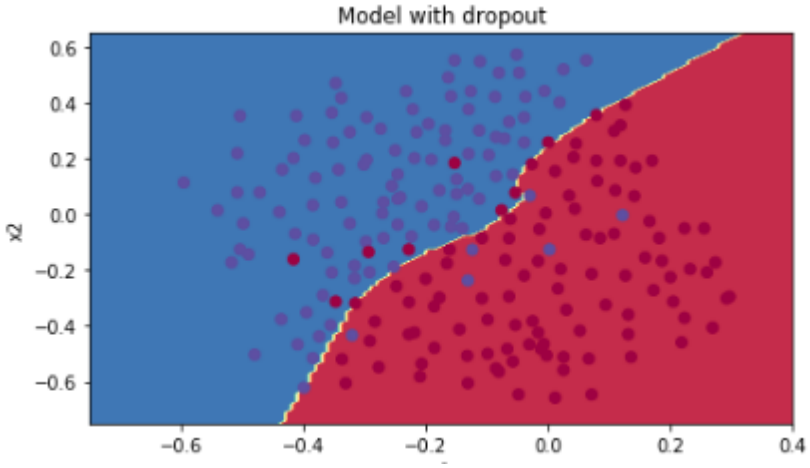
第四步是为了保证该层输出到下一层中的总期望值不发生改变，从而使得cost的期望值不发生改变。

带dropout的 backward

```
dA1 = dA1*D1
# Step 1: Apply mask D1 to shut down the same neurons as during the forward propagation
dA1 = np.divide(dA1,keep_prob)
# Step 2: Scale the value of neurons that haven't been shut down
dZ1 = np.multiply(dA1, np.int64(A1 > 0))
dW1 = 1./m * np.dot(dZ1, X.T)
db1 = 1./m * np.sum(dZ1, axis=1, keepdims = True)
```

step1 ：反向传播时使用将dA1与mask D1进行元素相乘，使反向传播过程中也失活节点也不参与传播

step2：正向传播时执行了A1 = np.divide(A1,keep_prob) ，A1被scale，dA1也被scale，因此，反向传播时，应该恢复到原来的水平。



相比于不做dropout的分类边界，相对比较平滑，测试集上的准确率达到了95%，相比91%有明显的提高。

为什么Dropot可以减弱overfitting

为了说明Dropout为什么可以减弱overfitting，首先说明**模型集成为什么可以提升模型的泛化能力** 。

假定我们有k个学习器：{$h_1, h_2, ... h_k$)},通过加权平均结合缠身的集成来完成回归任务f(其他任务类似，这里重点说明集成的精神)。集成结构为H,对事例x，

定义学习器h_i的分歧(ambiguity)为：$A(h_i|x) = (h_i(x) - H(x))^2$

则集成的分歧为$\tilde{A}(h|x) = \sum_{i=1}^{k} w_i A(h_i|x) = \sum_{i=1}^{k} w_i((h_i|x) - H(x))^2$

这里的分歧表示个体学习器在样本x上的不一致性，反映了个体学习器的多样性。

个体学习器h_i的平方误差为：$E(h_i|x) = (f(x) - h_i(x))^2$

集成H的平方误差为：$E(H|x) = (f(x) - H(x))^2$

个体学习误差的加权平均值：$\tilde{E}(h|x) = \sum_{i=1}^{k} w_i . E(h_i, x)$

将$\tilde{A}(h|x)$配方,则有$\tilde{A}(h|x) = \sum_{i=1}^{k} w_i(h_x(x) - H(x))^2 = \sum_{i=1}^{k} w_i(h_i(x) - f(x) + f(x) - H(x))^2$

$\tilde{A}(h|x) = \sum_{i=1}^{k} w_i(h_i(x) - f(x))^2 - 2 * w_i * (f(x) - h_i(x)) * (f(x) - H(x)) + (f(x) - H(x))^2$

$H(x) = \sum_{i=1}^{k} w_i h_i(x)$

从而有

$\tilde{A}(h|x) = \sum_{i=1}^{k} w_i(h_i(x) - f(x))^2 - 2(f(x) - H(x))^2 + (f(x) - H(x))^2 = \sum_{i=1}^{k} w_i(h_i(x) - f(x))^2 - (f(x) - H(x))^2$

$\tilde{A}(h|x) = \sum_{i=1}^{k} w_i(E(h_i|x) - E(H, x)) = \tilde{E}(h|x) - E(H|x)$

对全体样本则有(根据均值定义)：

$\sum_{i=1}^{k} w_i \int A(h_i|x)p(x)dx = \sum_{i=1}^{k} w_i \int E(h_i|x)p(x)dx - \int E(H|x)p(x)dx$

令$\tilde{E} = \sum_{i=1}^{k} w_i \int E(h_i|x)p(x)dx$,个体学习器的泛化误差加权均值

令$\tilde{A} = \sum_{i=1}^{k} w_i \int A(h_i|x)p(x)dx$个体学习器的加权分歧值(所有数据上)

从而有$E(H|x) = \tilde{E} - \tilde{A}$

上式清楚的说明了，只要个体学习器泛化误差越小，个体学习器之间的分歧越大，则集成模型的泛化误差就越小。

上式清晰的表明，增加模型的多样性，可以减小集成模型的泛化误差。而Dropout就是一种集成方法。

事实上不仅增加模型多样性，增加数据的多样性也可以有效提高模型泛化误差（比如batch Norm等）。

（上述证明来自周志华<机器学习>第8章集成学习，对其中的证明添加了中间几步，使证明更加清晰）

对某隐藏层施加dropout后，那么被施加dropout的每一层的每个节点被有可能失活，因此相当于训练了多个网络（假设神经元总数为N，则会有2^N种可能网络（输出不会过分依赖任何一个神经元的输出，减少了数据扰动产生的影响））。因此，dropout可以被认为是集成大量深层神经网络的bagging方法（当然，与完全训练多个模型进行bagging不同（普通的bagging方法不同的神经网络时独立的），dropout 共享参数，每个子模型继承父神经网络参数的不同子集，参数共享使得在有限可用的内存下表示指数级数量的模型成为可能， 因此dropout是一种廉价的bagging近似，事实上，Dropout不仅仅是训练一个Bagging的集成模型，而是一种特殊的共享隐藏单元的集成模型，意味着无论其他隐藏单元是否在模型中，每个隐藏单元都必须能够表现练好，相比独立模型集成，可以获得更好的泛化误差的提升。

Dropout强大的另一个原因来自施加到隐藏单元的掩码噪声，破坏提取的特征而不是原始值，让破坏的过程充分利用该模型迄今获得的关于输入分布的所有知识。

集成预测的是由不同分别的算术平均值给出的：$\frac{1}{k}\sum_{i=1}^{k} p^i(y,x)$

dropout 情况下。所有子网络算术平均值给出：$\sum_{\mu} p(\mu)p(y|x,\mu)$,其中$p(\mu)$是训练时采用$\mu$的概率分布，包含指数级的项，无法精确计算。

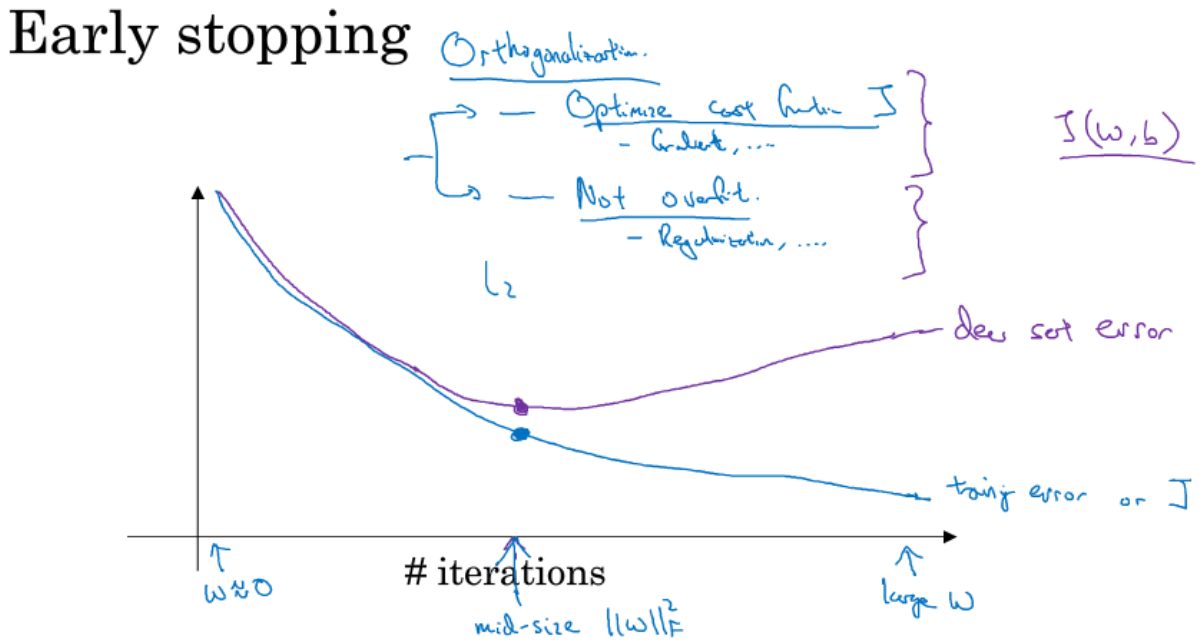
通常计算采用 **权重比例推断规则** ，有两种做法：1、将某一单元i输出的权重乘以该单元可能出现的概率，保证得到从该单元输出正确的期望值。2、在训练期间将单元的状态除以该单元出现的概率（上面的实现中，采用方法2），目标都是确保在测试时一个单元的期望总输入与在训练时该单元的期望总输入大致是相同的。权重比例推断规则对具有非线性的深度模型仅仅是一个近似，但在实践中效果很好。

实践上dropout相比于其他正则化方法更有效，也可以与其他形式的正则化合并，得到进一步提升。

dropout训练非常快，只需要添加一些节点输出与mask的乘法。

Dropout的另一个优点是不怎么限制适用的模型和训练过程，可以在任何使用分布式表示且可以用随机梯度下降训练的模型上都表现良好。包括FC,CNN,RNN等。

early stopping



Andrew Ng

如上图所示，随着训练程度的增加，训练集上的误差不断减小，但验证集上的误差确呈现先下降后上升的缺失，因此，为了获得最好的泛化误差，可以进行early stopping。

接下来尝试说明，为什么early stopping可以有效减弱overfitting：

early stopping 是通过控制训练步数来控制模型的有效容量，可以理解为将优化的参数空间的初始值限定在一个小的领域内，因此可以理解为一种特殊的L2正则化，但early stopping 能检查训练集合验证集的测试误差，自动确定正则化的大小，而L2正则化则需要进行多组超参数测试，因此更具优势。

参数绑定与参数共享

L2正则化表达了，我们希望权重尽可能小的先验知识，L1正则化表达了我们希望连接尽可能稀疏的先验知识，有时我们可能无法精确地知道应该使用什么样的参数，但根据相关领域和模型结构方面的知识得知模型之间应该存在一些相关性，比如迫使某些参数相等，这种方式称为参数共享，这种正则化的优势是可以显著减小模型所占用的内存。例如卷积神经网络

神经网络中的权重初始化

深度神经网络模型通常是迭代的，要求使用者指定一个初始点，训练深度模型是一个非常困难的问题，大多数算法都很大程度上受到初始选择的影响，好的初始化方式，可以加速梯度下降的收敛速度，可以获得更小的训练误差。

现代初始化的策略是简单的，启发式的，事实上我们对于初始点如何影响泛化的理解还是非常原始的，几乎没有提供任何选择初始点的指导。

唯一可以确定的是，初始化参数需要在不同单元之间“破坏对称性”，如果具有相同激活函数的两个隐藏单元连接到相同的输入，那么这些单元必须具有不同的初始参数，如果初始相同，那么神经网络将以相同的方式更新这两个神经元。

偏置项一般挑选常数

通常我们将权重初始化为高斯分布和均匀分布，但这两者之间的差别也不是很大，但是初始分布的大小缺失对优化过程的结果和网络泛化性能有很大的影响。

如果模型初始化的太小，那么信号将在每层间传递时，逐渐缩小而难以产生作用，但如果初始化的太大，那么信号将在每层间传递时逐渐放大并导致发散和失效

m个输入，n个输出 的全连接层

1、$W_{i,j}$ 从 $U(-\frac{1}{\sqrt(m)}, \frac{1}{\sqrt(m)})$中采用

```
def initialize_parameters_he(layers_dims):
"""
Arguments:
layer_dims -- python array (list) containing the size of each layer.

Returns:
parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                W1 -- weight matrix of shape (layers_dims[1], layers_dims[0])
                b1 -- bias vector of shape (layers_dims[1], 1)
                ...
                WL -- weight matrix of shape (layers_dims[L], layers_dims[L-1])
                bL -- bias vector of shape (layers_dims[L], 1)
"""

np.random.seed(3)
parameters = {}
L = len(layers_dims) - 1 # integer representing the number of layers

for l in range(1, L + 1):
    ### START CODE HERE ### (≈ 2 lines of code)
    parameters['W' + str(l)] =
    np.random.randn(layers_dims[l],layers_dims[l-1])*np.sqrt(2/(layers_dims[l-1]))
    parameters['b' + str(l)] = np.zeros([layers_dims[l],1])
    ### END CODE HERE ###

return parameters
```

2、Xaiver 初始化 ：$W_{i,j}$ 从 $U(-\frac{6}{\sqrt(m+n)}, \frac{6}{\sqrt(m+n)})$中采用，Xaiver的目的是使权重初始化的正好合适，Xaiver就是让权重满足均值为0，方差为$\frac{2}{m+n}$

3、使用autoencoder 等无监督训练方法，使得每层变换时，尽可能的完整保留原始输入的信息，作为初始化的方式。

4、使用相关问题上训练得到的权重作为初始权重（迁移学习）