Debug

1. Tuning Process

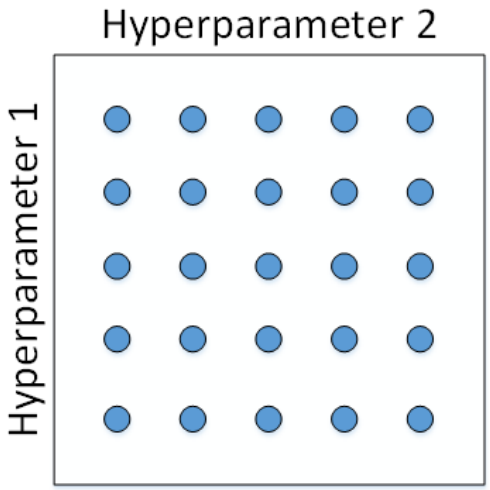
深度神经网络需要调试的超参数（Hyperparameters）较多，包括：

α：学习因子
β：动量梯度下降因子
$\beta_1, \beta_2, \varepsilon$：Adam算法参数
layers：神经网络层数
hidden units：各隐藏层神经元个数
learning rate decay：学习因子下降参数
mini-batch size：批量训练样本包含的样本个数

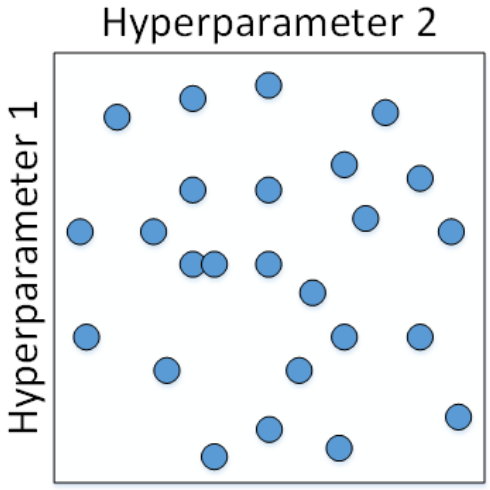
超参数之间也有重要性差异。通常来说，学习因子α是最重要的超参数，也是需要重点调试的超参数。动量梯度下降因子β、各隐藏层神经元个数hidden units和mini-batch size的重要性仅次于α。然后就是神经网络层数layers和学习因子下降参数learning rate decay。最后，Adam算法的三个参数$\beta_1, \beta_2, \varepsilon$,ε一般常设置为0.9，0.999和$10^{-8}$，不需要反复调试。当然，这里超参数重要性的排名并不是绝对的，具体情况，具体分析。

如何选择和调试超参数？传统的机器学习中，**对每个参数等距离选取任意个数的点**，然后，分别使用不同点对应的**参数组合进行训练**，最后根据验证集上的表现好坏，来选定最佳的参数。

例如有两个待调试的参数，分别在每个参数上选取5个点，这样构成了5×5=25中参数组合，如下图所示：



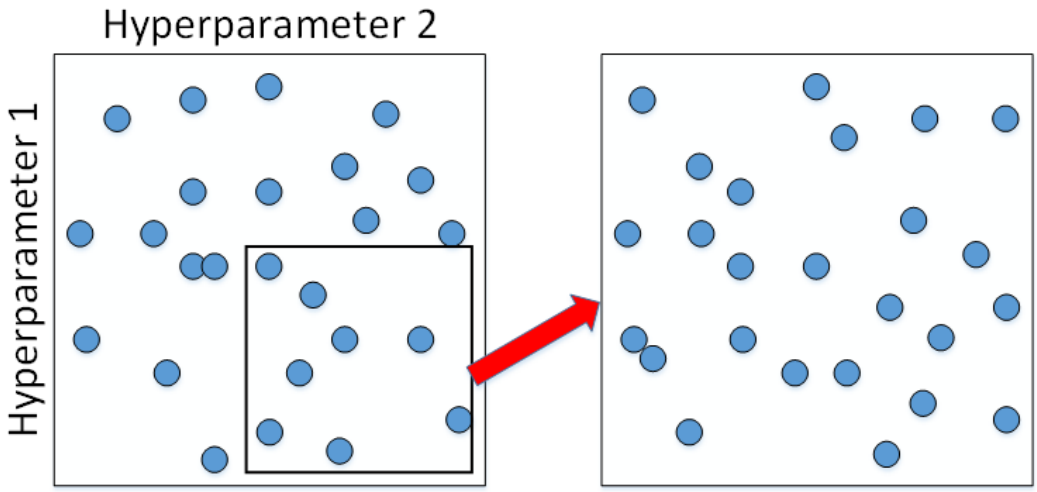
这种做法在参数比较少的时候效果较好。但是在深度神经网络模型中，我们不采用这种均匀间隔取点的方法，比较好的做法是使用**随机选择**。也就是说，对于上面这个例子，我们随机选择25个点，作为待调试的超参数，如下图所示：



随机化选择参数的目的是为了尽可能地得到更多种参数组合。还是上面的例子，如果使用均匀采样的话，每个参数只有5种情况；而使用随机采样的话，每个参数有25种可能的情况，因此更有可能得到最佳的参数组合。

这种做法带来的另外一个好处就是对重要性不同的参数之间的选择效果更好。假设hyperparameter1为α，hyperparameter2为ε，显然二者的重要性是不一样的。如果使用第一种均匀采样的方法，ε的影响很小，相当于只选择了5个α值。而如果使用第二种随机采样的方法，ε和α都有可能选择25种不同值。这大大增加了α调试的个数，更有可能选择到最优值。其实，在实际应用中完全不知道哪个参数更加重要的情况下，随机采样的方式能有效解决这一问题，但是均匀采样做不到这点。

在经过随机采样之后，可能得到某些区域模型的表现较好。然而，为了得到更精确的最佳参数，我们应该继续**对选定的区域进行由粗到细的采样**（coarse to fine sampling scheme）。也就是放大表现较好的区域，再对此区域做更密集的随机采样。例如，对下图中右下角的方形区域再做25点的随机采样，以获得最佳参数。



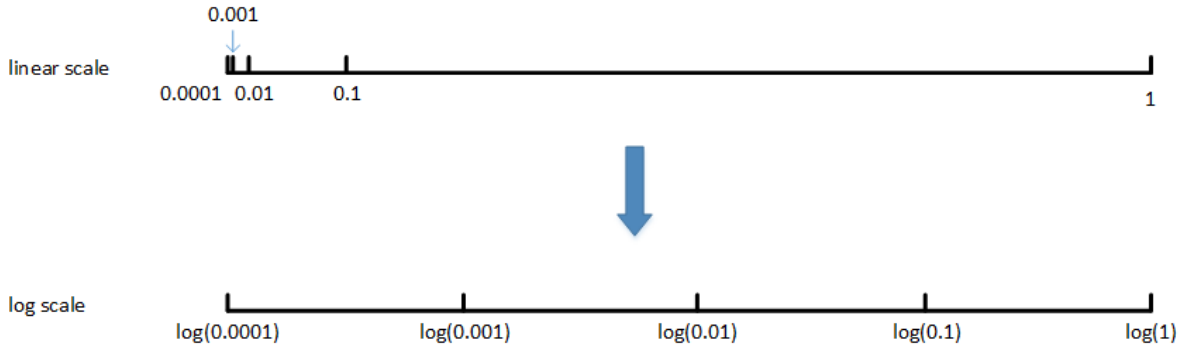
2. Using an appropriate scale to pick hyperparameters

上一部分讲的调试参数使用随机采样，对于某些超参数是可以进行尺度均匀采样的，但是某些超参数需要选择不同的合适尺度进行随机采样。

例如对于超参数layers和hidden units，都是正整数，是可以进行均匀随机采样的，即超参数每次变化的尺度都是一致的（如每次变化为1，犹如一个刻度尺一样，刻度是均匀的）。

但是，对于某些超参数，可能需要非均匀随机采样（即非均匀刻度尺）。例如超参数α，待调范围是[0.0001, 1]。如果使用均匀随机采样，那么有90%的采样点分布在[0.1, 1]之间，只有10%分布在[0.0001, 0.1]之间。这在实际应用中是不太好的，因为最佳的α值可能主要分布在[0.0001, 0.1]之间，而[0.1, 1]范围内$alpha$值效果并不好。因此更关注的是区间[0.0001, 0.1]，应该在这个区间内细分更多刻度。

通常的做法是**将linear scale转换为log scale**，将均匀尺度转化为非均匀尺度，然后再在log scale下进行均匀采样。这样，[0.0001, 0.001]，[0.001, 0.01]，[0.01, 0.1]，[0.1, 1]各个区间内随机采样的超参数个数基本一致，也就扩大了之前[0.0001, 0.1]区间内采样值个数。



一般解法是，如果线性区间为[a, b]，令m=log(a)，n=log(b)，则对应的log区间为[m,n]。对log区间的[m,n]进行随机均匀采样，然后得到的采样值r，最后反推到线性区间，即10^r。10^r就是最终采样的超参数。相应的Python语句为：

```
m = np.log10(a)
n = np.log10(b)
```

```
r = np.random.rand()
r = m + (n-m)*r
r = np.power(10,r)
```

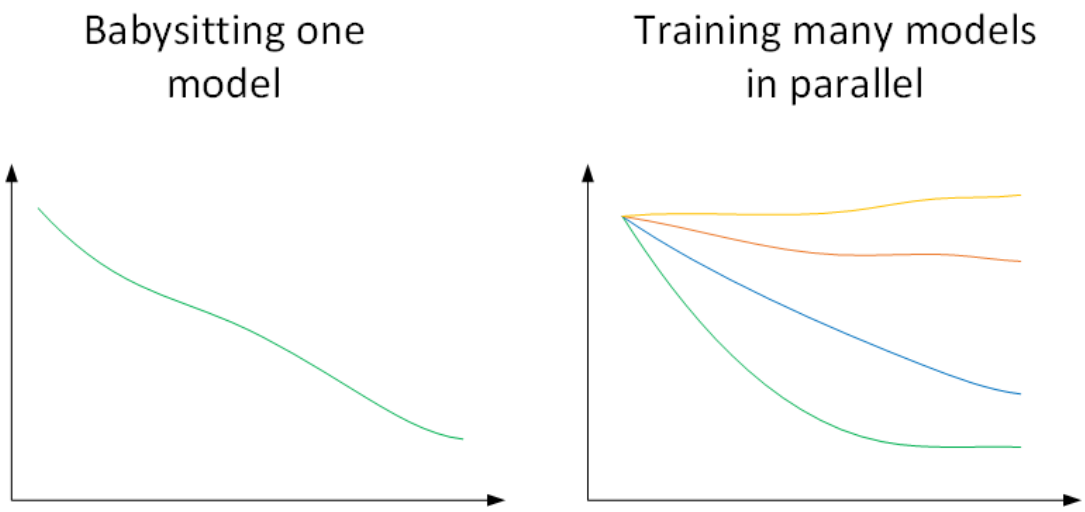
除了α之外，动量梯度因子β也是一样，在超参数调试的时候也需要进行非均匀采样。一般β的取值范围在[0.9, 0.999]之间，那么$1-\beta$的取值范围就在[0.001, 0.1]之间。那么直接对$1-\beta$在[0.001, 0.1]区间内进行log变换即可。

这里解释下为什么β也需要向α那样做非均匀采样。假设β从0.9000变化为0.9005，那么$\frac{1}{1-\beta}$基本没有变化。但假设β从0.9990变化为0.9995，那么$\frac{1}{1-\beta}$前后差别1000。β越接近1，指数加权平均的个数越多，变化越大。所以对β接近1的区间，应该采集得更密集一些。

3. Hyperparameters tuning in practice: Pandas vs. Caviar

经过调试选择完最佳的超参数并不是一成不变的，一段时间之后（例如一个月），需要根据新的数据和实际情况，再次调试超参数，以获得实时的最佳模型。

在训练深度神经网络时，一种情况是受计算能力所限，只能**对一个模型进行训练**，调试不同的超参数，使得这个模型有最佳的表现。称之为**Babysitting one model**。
另外一种情况是可以**对多个模型同时进行训练**，每个模型上调试不同的超参数，根据表现情况，选择最佳的模型。称之为**Training many models in parallel**。



因为第一种情况只使用一个模型，所以类比做Panda approach；第二种情况同时训练多个模型，类比做Caviar approach。使用哪种模型是由计算资源、计算能力所决定的。一般来说，对于非常复杂或者数据量很大的模型，使用Panda approach更多一些。

4. Normalizing activations in a network

Sergey Ioffe和Christian Szegedy两位学者提出了Batch Normalization方法。**Batch Normalization不仅可以让调试超参数更加简单，而且可以让神经网络模型更加“健壮”**。也就是说较好模型可接受的超参数范围更大一些，包容性更强，使得更容易去训练一个深度神经网络。

训练神经网络时，**标准化输入可以提高训练的速度**。方法是对训练数据集进行归一化的操作，即将原始数据减去其均值μ后，再除以其方差σ^2。但是标准化输入只是对输入进行了处理，那么对于神经网络，又该如何对各隐藏层的输入进行标准化处理呢?

其实在神经网络中，第l层隐藏层的输入就是第$l-1$层隐藏层的输出$A^{[l-1]}$。对$A^{[l-1]}$进行标准化处理，从原理上来说可以提高$W^{[l]}$和$b^{[l]}$的训练速度和准确度。这种对各隐藏层的标准化处理就是Batch Normalization。值得注意的是，实际应用中，一般是对$Z^{[l-1]}$进行标准化处理而不是$A^{[l-1]}$，其实差别不是很大。

Batch Normalization对第l层隐藏层的输入$Z^{[l-1]}$做如下标准化处理，忽略上标$[l-1]$:

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$$

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

其中，m是单个mini-batch包含样本个数，ε是为了防止分母为零，可取值10^{-8}。这样，使得该隐藏层的所有输入$z^{(i)}$**均值为0，方差为1**。

但是，大部分情况下并不希望所有的$z^{(i)}$均值都为0，方差都为1，也不太合理。通常需要对$z^{(i)}$进行进一步处理：

$$\tilde{z}^{(i)} = \gamma \cdot z_{norm}^{(i)} + \beta$$

上式中，γ和β是learnable parameters，类似于W和b一样，可以通过梯度下降等算法求得。这里，$gamma$和β的作用是**让$\tilde{z}^{(i)}$的均值和方差为任意值**，只需调整其值就可以了。例如，令：

$$\gamma = \sqrt{\sigma^2 + \varepsilon},\ \ \beta = u$$

则$\tilde{z}^{(i)} = z^{(i)}$，即identity function。可见，设置γ和β为不同的值，可以得到任意的均值和方差。

这样，通过Batch Normalization，对隐藏层的各个$z^{[l](i)}$进行标准化处理，得到$\tilde{z}^{[l](i)}$，替代$z^{[l](i)}$。

值得注意的是，输入的标准化处理Normalizing inputs和隐藏层的标准化处理Batch Normalization是有区别的。Normalizing inputs使所有输入的均值为0，方差为1。而Batch Normalization可使各隐藏层输入的均值和方差为任意值。实际上，从激活函数的角度来说，如果各隐藏层的输入均值在靠近0的区域即处于激活函数的线性区域，这样不利于训练好的非线性神经网络，得到的模型效果也不会太好。这也解释了为什么需要用γ和β来对$z^{[l](i)}$作进一步处理。

5. Fitting Batch Norm into a neural network

已经知道了如何对某单一隐藏层的所有神经元进行Batch Norm，接下来将研究如何把Bath Norm应用到整个神经网络中。

对于L层神经网络，经过Batch Norm的作用，整体流程如下：

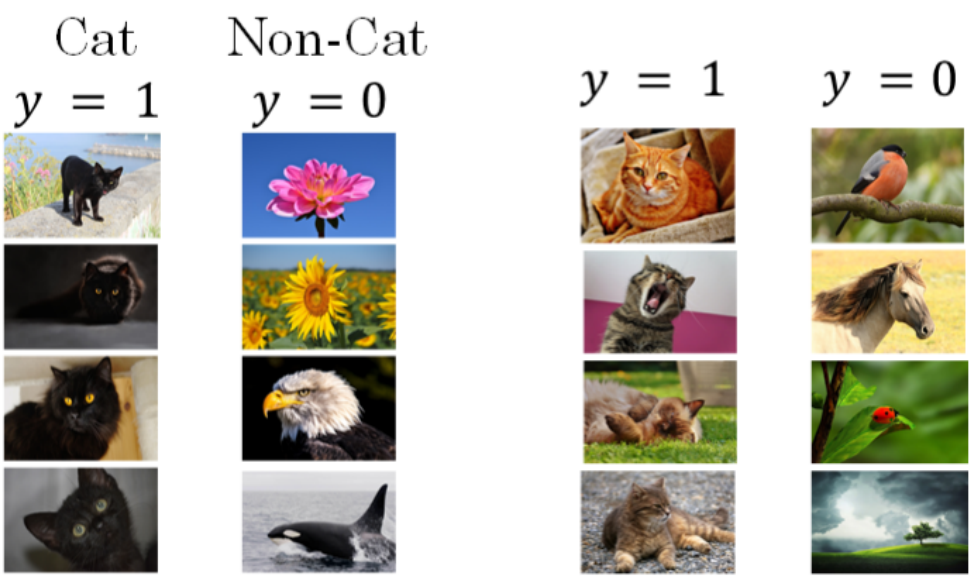
实际上，Batch Norm经常使用在mini-batch上，这也是其名称的由来。

因为Batch Norm对各隐藏层$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$有去均值的操作，所以这里的常数项$b^{[l]}$可以消去，其数值效果完全可以由$\tilde{Z}^{[l]}$中的$\beta$来实现。因此，在使用Batch Norm的时候，可以忽略各隐藏层的常数项$b^{[l]}$。在使用梯度下降算法时，分别对$W^{[l]}$，$\beta^{[l]}$和$\gamma^{[l]}$进行迭代更新。除了传统的梯度下降算法之外，还可以使用动量梯度下降、RMSprop或者Adam等优化算法。

6. Why does Batch Norm work?

可以把输入特征做均值为0，方差为1的规范化处理，来加快学习速度。而Batch Norm也是对隐藏层各神经元的输入做类似的规范化处理。总的来说，**Batch Norm不仅能够提高神经网络训练速度，而且能让神经网络的权重W的更新更加“稳健”**，尤其在深层神经网络中更加明显。比如神经网络很后面的W对前面的W包容性更强，即前面的W的变化对后面W造成的影响很小，整体网络更加健壮。

举个例子来说明，假如用一个浅层神经网络（类似逻辑回归）来训练识别猫的模型。如下图所示，提供的所有猫的训练样本都是黑猫。然后，用这个训练得到的模型来对各种颜色的猫样本进行测试，测试的结果可能并不好。其原因是训练样本不具有一般性（即不是所有的猫都是黑猫），这种训练样本（黑猫）和测试样本（猫）分布的变化称之为covariate shift。



对于这种情况，如果**实际应用的样本与训练样本分布不同，即发生了covariate shift，则一般是要对模型重新进行训练的。**在神经网络，尤其是深度神经网络中，covariate shift会导致模型预测效果变差，重新训练的模型各隐藏层的$W^{[l]}$和$B^{[l]}$均产生偏移、变化。而**Batch Norm的作用是减小covariate shift的影响**，让模型变得更加健壮。Batch Norm减少了各层$W^{[l]}$、$B^{[l]}$之间的耦合性，让各层更加独立，实现自我训练学习的效果。也就是说，如果输入发生covariate shift，那么因为Batch Norm的作用，对个隐藏层输出$Z^{[l]}$进行均值和方差的归一化处理，$W^{[l]}$和$B^{[l]}$更加稳定，使得原来的模型也有不错的表现。针对上面这个黑猫的例子，如果使用深层神经网络，使用Batch Norm，那么该模型对花猫的识别能力应该也是不错的。

从另一个方面来说，**Batch Norm也起到轻微的正则化（regularization）效果。**具体表现在：
1.每个mini-batch都进行均值为0，方差为1的归一化操作
2.每个mini-batch中，对各个隐藏层的$Z^{[l]}$
添加了随机噪声，效果类似于Dropout
3.mini-batch越小，正则化效果越明显

但是，Batch Norm的正则化效果比较微弱，正则化也不是Batch Norm的主要功能。

7. Batch Norm at test time

训练过程中，Batch Norm是对单个mini-batch进行操作的，但在测试过程中，如果是单个样本，该如何使用Batch Norm进行处理呢?

首先，回顾一下训练过程中Batch Norm的主要过程：

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma \cdot z_{norm}^{(i)} + \beta$$

其中，μ和σ^2是对单个mini-batch中所有m个样本求得的。在测试过程中，如果只有一个样本，求其均值和方差是没有意义的，就需要对μ和σ^2进行估计。估计的方法有很多，理论上可以将所有训练集放入最终的神经网络模型中，然后将每个隐藏层计算得到的$\mu^{[l]}$和$\sigma^{2[l]}$直接作为测试过程的μ和σ^2来使用。但是，实际应用中一般不使用这种方法，而是使用指数加权平均（exponentially weighted average）的方法来预测测试过程单个样本的μ和σ^2。

指数加权平均的做法很简单，对于第l层隐藏层，考虑所有mini-batch在该隐藏层下的$\mu^{[l]}$和$\sigma^{2[l]}$，然后用指数加权平均的方式来预测得到当前单个样本的$\mu^{[l]}$和$\sigma^{2[l]}$。这样就实现了对测试过程单个样本的均值和方差估计。最后，再利用训练过程得到的γ和β值计算出各层的$\tilde{z}^{(i)}$值。

8. Softmax Regression

目前介绍的都是二分类问题，神经网络输出层只有一个神经元，表示预测输出$\hat{y}$是正类的概率$P(y=1|x)$，$\hat{y}>0.5$则判断为正类，$\hat{y}<0.5$则判断为负类。

对于多分类问题，用C表示种类个数，神经网络中输出层就有C个神经元，即$n^{[L]}=C$。其中，**每个神经元的输出依次对应属于该类的概率**，即$P(y=c|x)$。为了处理多分类问题，一般使用Softmax回归模型。Softmax回归模型输出层的激活函数如下所示：

$$z^{[L]}=W^{[L]}a^{[L-1]}+b^{[L]}$$

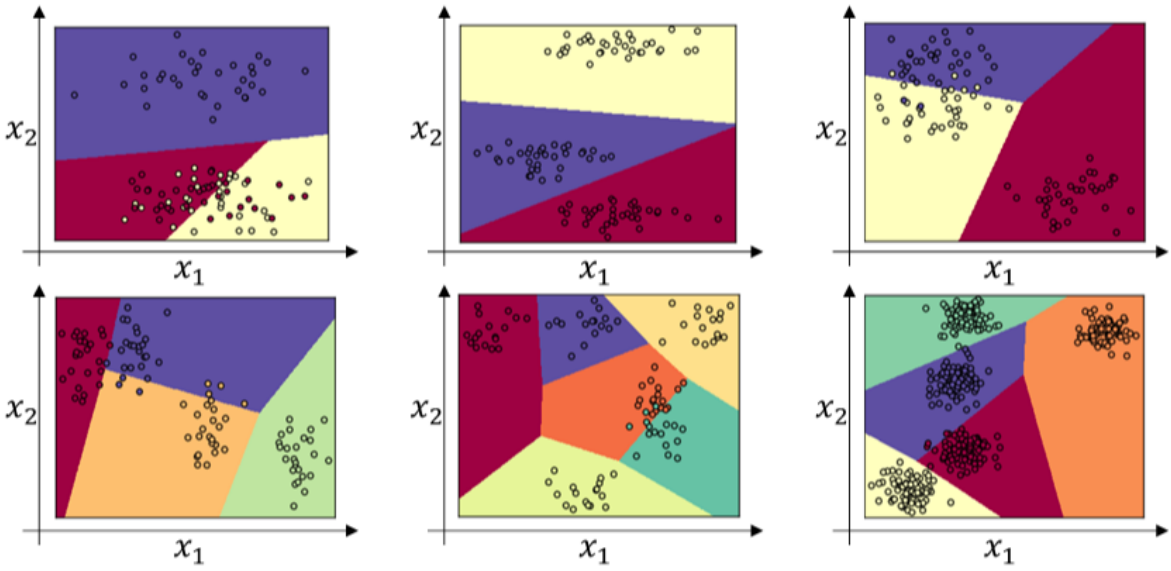
$$a_i^{[L]}=\frac{e^{z_i^{[L]}}}{\sum_{i=1}^{C}e^{z_i^{[L]}}}$$

输出层每个神经元的输出$a_i^{[L]}$对应属于该类的概率，满足：

$$\sum_{i=1}^{C}a_i^{[L]}=1$$

所有的$a_i^{[L]}$，即$\hat{y}$，维度为(C, 1)。

下面给出几个简单的线性多分类的例子：



如果使用神经网络，特别是深层神经网络，可以得到更复杂、更精确的非线性模型。

9. Training a softmax classifier

Softmax classifier的训练过程与二元分类问题有所不同。先来看一下softmax classifier的loss function。举例来说，假如C=4，某个样本的预测输出$\hat{y}$和真实输出y为：

$$\hat{y}=\begin{bmatrix}0.3\\0.2\\0.1\\0.4\end{bmatrix}$$

$$y=\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

从$\hat{y}$值来看，$P(y=4|x)=0.4$，概率最大，而真实样本属于第2类，因此该预测效果不佳。定义softmax classifier的loss function为：

$$L(\hat{y},y)=-\sum_{j=1}^{4}y_j\cdot log\ \hat{y}_j$$

然而，由于只有当$j=2$时，$y_2=1$，其它情况下，$y_j=0$。所以，上式中的$L(\hat{y},y)$可以简化为：

$$L(\hat{y},y)=-y_2\cdot log\ \hat{y}_2=-log\ \hat{y}_2$$

要让$L(\hat{y},y)$更小，就应该让$\hat{y}_2$越大越好。$\hat{y}_2$反映的是概率，完全符合之前的定义。

所有m个样本的cost function为：

$$J=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y},y)$$

其预测输出向量$A^{[L]}$即$\hat{Y}$的维度为(4, m)。

softmax classifier的反向传播过程仍然使用梯度下降算法，其推导过程与二元分类有一点点不一样。因为只有输出层的激活函数不一样，先推导$dZ^{[L]}$：

$$da^{[L]}=-\frac{1}{a^{[L]}}$$

$$\frac{\partial a^{[L]}}{\partial z^{[L]}}=\frac{\partial}{\partial z^{[L]}}\cdot(\frac{e^{z_i^{[L]}}}{\sum_{i=1}^{C}e^{z_i^{[L]}}})=a^{[L]}\cdot(1-a^{[L]})$$

$$dz^{[L]}=da^{[L]}\cdot\frac{\partial a^{[L]}}{\partial z^{[L]}}=a^{[L]}-1=a^{[L]}-y$$

对于所有m个训练样本：

$$dZ^{[L]}=A^{[L]}-Y$$

可见$dZ^{[L]}$的表达式与二元分类结果是一致的，虽然推导过程不太一样。然后就可以继续进行反向传播过程的梯度下降算法了，推导过程与二元分类神经网络完全一致。

10. Deep learning frameworks

深度学习框架有很多，例如：
Caffe/Caffe2
CNTK
DL4J
Keras
Lasagne
mxnet
PaddlePaddle
TensorFlow
Theano
Torch

一般选择深度学习框架的基本准则是：
Ease of programming(development and deployment)
Running speed
Truly open(open source with good governance)

实际应用中，应该根据自己的需求选择最合适的深度学习框架。

11. TensorFlow

这里简单介绍一下最近几年比较火的一个深度学习框架：TensorFlow。

举个例子来说明，例如cost function是参数w的函数：

$$J=w^2-10w+25$$

如果使用TensorFlow对cost function进行优化，求出最小值对应的w，程序如下：

```
import numpy as np
import tensorflow as tf

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

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

> 0.1

```
for i in range(1000):
    session.run(train)
print(session.run(w))
```

> 4.99999

TensorFlow框架内可以直接调用梯度下降优化算法，不需要我们自己再写程序了，大大提高了效率。在运行1000次梯度下降算法后，w的解为4.99999，已经非常接近w的最优值5了。

针对上面这个例子，如果对w前的系数用变量x来代替，程序如下：

```
import numpy as np
import tensorflow as tf

cofficients = 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

```
session.run(train, feed_dict=(x:coefficients))
print(session.run(w))
```

> 0.1

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

> 4.99999

结果跟之前是一样的。除此之外，还可以更改x即cofficients的值，而得到不同的优化结果w。

另外，上段程序中的：

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

有另外一种写法：

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

TensorFlow的最大优点就是采用数据流图（data flow graphs）来进行数值运算。图中的节点（Nodes）表示数学操作，图中的线（edges）则表示在节点间相互联系的多维数据数组，即张量（tensor）。而且它灵活的架构让你可以在多种平台上展开计算，例如台式计算机中的一个或多个CPU（或GPU），服务器，移动设备等等。