改善深层神经网络

第一周 深度学习的实用层面

1.1 训练\开发\测试集

应用机器学习是一个高度迭代的过程，为了得到一个称心的神经网络，我们往往需要反复进行调参、测试，在测试过程中，一个好的样本就显得十分重要。

一般地，在训练神经网络过程中，我们将数据分为训练集、开发集、测试集。训练集主要是由于神经网络权重、阈值等参数的训练；开发集主要用于超参数的调试以及算法的比较；测试集用于最终评估算法的效果，不参与任何参数、超参数的调试。

在小规模数据时代，训练集、开发集、测试集的比例大概是 6：2：2（如果没有明确的开发集则是训练集、测试集的比例为 7：3）；在大数据时代，开发集、测试集的占比则被大大压缩，百万条数据规模的上述比例往往变为 98：1：1，超过百万数据规模上述比例变为 99.5：0.4：0.1

注意：①训练集和测试集的数据可能存在差别，但开发集和测试集应当遵循同一分布：以《神经网络和深度学习》一门课中的 logistic 回归编程作业为例，训练集往往是在网上下载的猫的图片，而测试集则可能是随手拍摄猫的照片，二者质量差异很大，在这种问题中，为了保证算法的有效性，开发集和测试集应当遵循同一分布。 ②可以没有测试集，因为开发集很大程度上可以代替测试集的一些功能

1.2 偏差\方差

偏差与方差是统计学中的两个概念，偏差可以理解为预测值相当于真实值的偏移程度，方差可以理解为预测值分布的集中情况，图 1 和图 2 可以便于理解方差与偏差的含义。

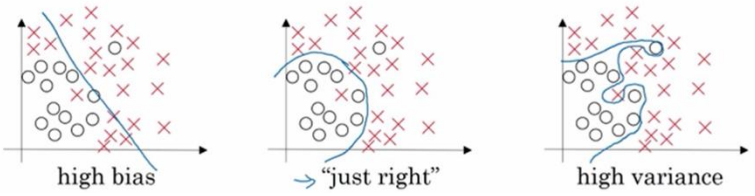


图 1

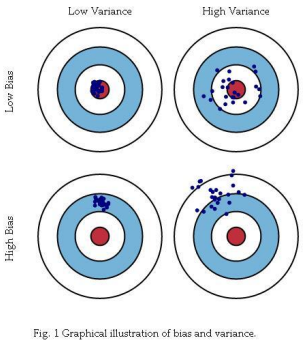
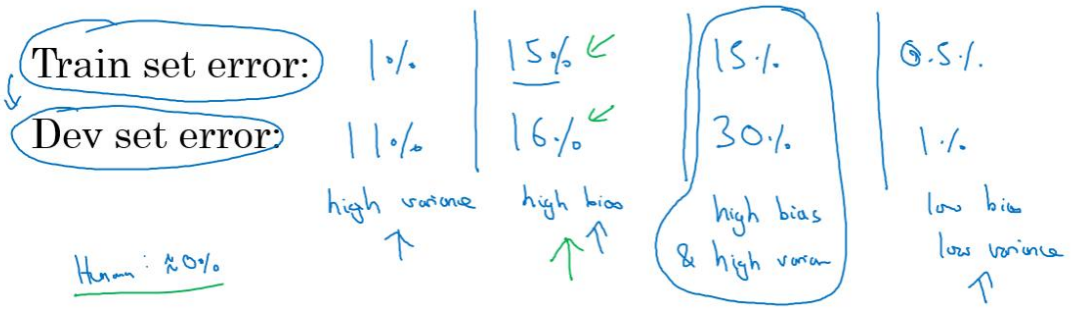


图 2

偏差较大时，预测数据严重偏离实际值，如图 1 最左侧曲线，表现为对训练集的欠拟合；方差较大时，预测数据较为分散，如图 1 最右侧曲线，表现为对训练集的过拟合。

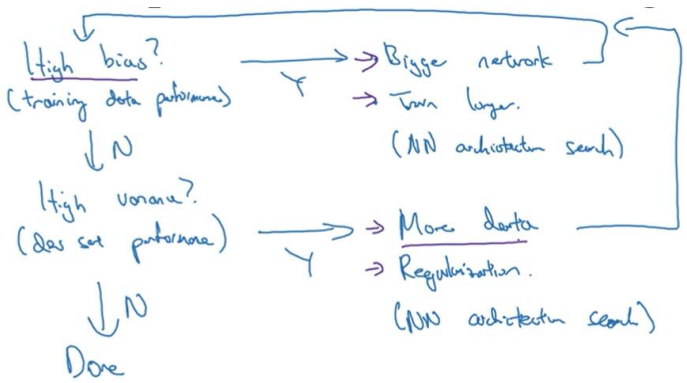
而在实践过程中，方差和偏差也可从训练集准确率和开发集错误率得知：



训练集错误率低而开发集错误率高，回归曲线则对训练集过拟合，该系统偏差低、方差高；训练集和开发集错误率均较高但相差不大，回归曲线对训练集欠拟合，该系统偏差高、方差低；训练集和开发集错误率均较高但开发集明显更高，该系统偏差高、方差高，造成原因是回归曲线对部分点过拟合，对整体欠拟合，这种情况容易出现在高维训练集中；训练集和开发集错误率均很低，回归曲线对训练集适度拟合，偏差、方差均低。

1.3 机器学习基础

如何调试神经网络？



如右图所示，在神经网络完成训练后，首先对其偏差进行评估，如果偏差过大存在欠拟合现象，则可以选择构建一个更大的神经网络（一般都有效）、训练更长时间（不一定有效）、新的神经网络架构（不一定有效）等方法对神经网络进行修改，训练，直到偏差下降到可接受范围内。然后就是要对方差进行评估，如果方差过大，存在过拟合现象，在条件允许的情况下可以投入更多的训练数据，如果没有更多的训练数据，则可以通过正则化减少过拟合，同时选择新的神经网络结构也是可以的，但不一定有效。

偏差和方差的权衡：在机器学习发展的初期，偏差和方差的权衡往往是比较重要的内容，因为当时并没有很好的方法来降低一个而不影响另一个，很多时候在降低偏差的时候方差会上升。但在大数据时代，我们可以通过构建一个更大的的神经网络、喂更多的数据并且合理地进行正则化使得两者均保持在较低水平。

1.4 正则化

1、logistic 回归中的 L2 正则化

在 logistic 回归中，L2 正则化就是在成本函数中加入w和b的第二范数，而在实际操作中b的第二范数往往对结果影响不大，故可以省略。

即 $J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L\left(\hat{y}^{(i)},y^{(i)}\right)+\frac{\lambda}{2m}\|w\|_2^2$，其中$\|w\|_2^2=\sum_{j=1}^{n_x}w_j^2=w^Tw$，$\lambda$为正则化参数

注意：①在正则化过程中，是否加入b的第二范数往往对结果影响不大，故可以

省略 ②在编程过程中lambda是Python中的保留字,故常用lambd来代替lambda作为λ的符号 ③上述的正则化叫做L2正则化，相应地也有L1正则化，即 $J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}\|w\|_1$，其中$\|w\|_1=\sum_{j=1}^{n_x}|w_j|$，但不常用。这种方法会使得$w$变为稀疏项，可以减少一定的存储空间，但效果不明显

2、神经网络中的L2正则化

神经网络的正则化与logistic回归大体类似，只是在正则化过程中需要加入所有w的第二范数，具体表达式如下：

$$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}\sum_{l=1}^{L}\left\|W^{[l]}\right\|_F^2$$

其中$\left\|w^{[l]}\right\|_F^2=\sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}\left(w_{ij}^{[l]}\right)^2$

这样，更新参数时，w的下降速率就会增大，具体如下

$$\mathrm{d}W^{[l]}=(from\ backprop)+\frac{\lambda}{m}W^{[l]}$$

$$W^{[l]}=W^{[l]}-\alpha\mathrm{d}W^{[l]}=\left(1-\frac{\alpha\lambda}{m}\right)W^{[l]}-\alpha(from\ backprop)$$

其中$(from\ backprop)$是正常反向传播得到的$\mathrm{d}w^{[l]}$

因为系数$\left(1-\frac{\alpha\lambda}{m}\right)$的存在，L2正则化也被称为“权重衰减”

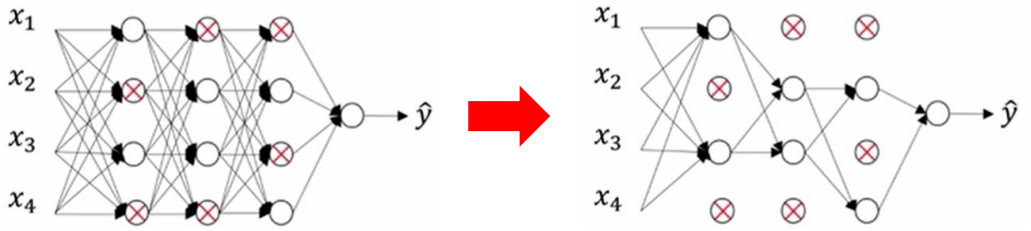
1.5 为什么正则化可以减少过拟合？

一方面，从直观上看，正则化参数λ越大，$W^{[l]}$将会越小，即神经网络中很多神经元的权重将会下降，这样神经网络就会慢慢趋于简单，简单的神经网络自然很难拟合出复杂的函数，这样就降低了拟合程度。

另一方面，假设激活函数为$g(z)=tanh(z)$，由于$W^{[l]}$变小，Z则会变小，在原点的一个较小邻域内，$g(z)=tanh(z)$会趋于线性，这样神经网络将会趋于线性输出，在《神经网络与深度学习》一课中已学到，如果激活函数都是线性函数，那么神经网络的输出也将是对X的线性组合，这样的输出也不可能拟合一个复杂的函数，这样也会降低拟合程度。

1.6 Dropout正则化

Dropout正则化，也叫随机失活，顾名思义，其核心思想也是缩小神经网络的规模，不同于L2正则化减少权重的缩减方式，Dropout正则化是通过逐层遍历神经元,并随机删除若干神经元的方法来实现缩小神经网络的目的。实现步骤：首先遍历神经网络并选择需要随机失活的神经元,然后删除这些选中的神经元以及其与其他神经元之间的连接。直观表示如下图所示。



反向随机失活：伪代码如下

```
import numpy as np

keepprob = 0.8

a[l] = np.random.rand(4,3)

d[l] = np.random.rand(al.shape[0],al.shape[1]) < keepprob

a[l] *= d[l]

a[l] /= keepprob
```

首先，我们要设定随机失活的概率 keepprob，上图中表示随机失活的概率为 20%。矩阵 d[1]的作用是产生一个随机布尔矩阵，表示随机失活神经元的位置，将其与 a[1]相乘，a[1]中剩余的元素就是经过一轮迭代所剩下的元素。最后，为了使输出值的期望保持不变，将 a[1]扩张。这样就完成了第 1 层的反省随机失活。

注意：在测试阶段不要使用反省随机失活。

1.7 理解 Dropout

为什么 Dropout 能够工作？

直观上理解，Dropout 会随机删除神经网络中的一些神经元，使得神经网络规模变小，这样可以有效地减少过拟合现象的发生。

对于单个神经元而言，由于每个输入特征都有可能会被随机删除，那么权重将会被分散，这样也会减少过拟合现象的发生。

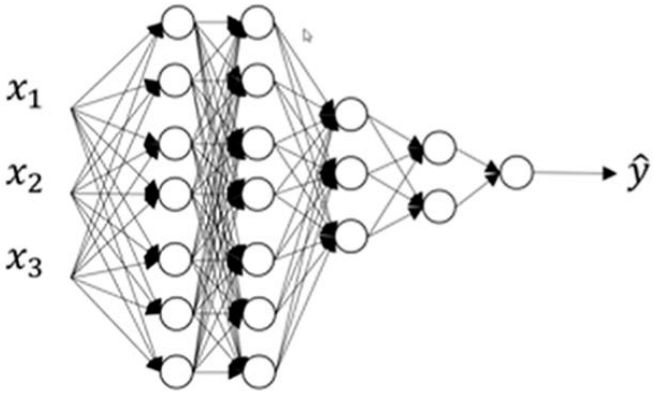
Dropout 的使用技巧

类似 L2 正则化，Dropout 的使用也存在着调整超参数的问题，下面将介绍 Dropout 中超参数 keepprob 的选择问题。

绝大多数情形中，对于神经网络的某一隐层而言，输入特征越多，神经元数目越多，$W^{[l]}$矩阵维度就会越大，这样就容易出现权重分配不均的情况，进而导致过拟合的发生；而对于输入特征较少的层，过拟合现象发生的机率不大，并且如果再对其进行 dropout，很可能会造成重要特征的丢失，进而导致输出结果存在较大偏差。针对这种情况，我们可以对不同的隐层设置不同的 keepprob。对于$W^{[l]}$较大的隐层，keepprob 可以设置为较小的数，例如 0.5；对于$W^{[l]}$较小的

隐层，keepprob 则可以设置为较大的数，例如 0.9，1。

以右图为例，各隐层的超参数 keepprob 可以分别设置为 1.0，0.7，0.5，0.9，1.0，1.0。



注意：一般情况下对于输入 X，通常不会进行 dropout。但在计算机视觉中，由于输入 X 过于庞大并且训练集较小，可以尝试对 X 进行 dropout。

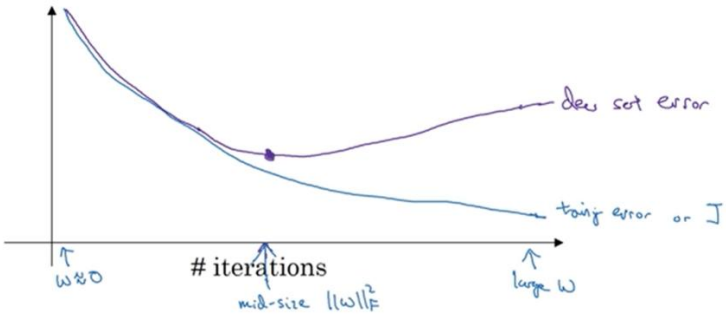
1.8 其他正则化方法

1、数据扩增

增大训练集可以解决过拟合现象的产生，但在很多场景下（如计算机视觉），增大训练集的成本高昂或者根本没有办法获取更多的数据。数据扩增可以根据已有数据额外生成假训练数据，进而部分地达到增大训练集的效果。

如上图所示，我们可以通过将原图剪辑、反向、放大、缩小、旋转等方法获得新的图片，进而完成数据扩增。

2、Early stopping

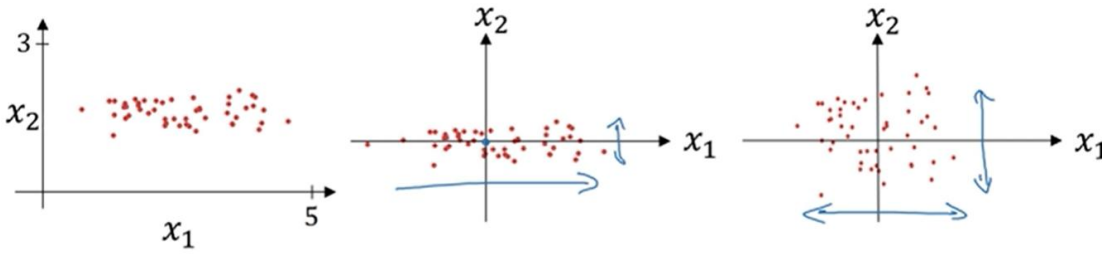


如上图所示，在迭代过程中，我们可以同时绘制训练集和开发集上的成本函数。一般地，随着迭代的进行，训练集上的成本函数会单调下降，而开发集的成本函数会先降后升。Early stopping 的思想就是在迭代过程中，选择一个合适的中间点停止迭代，这样可以有效地避免过拟合的发生。

但 Early stopping 也存在很大的问题，首先要从机器学习的步骤讲起。机器学习大致可以分为两个步骤：优化和正交化，优化过程就是对成本函数 J 进行凸优化，进而降低偏差；正交化则是为了降低方差。Early stopping 利用提前

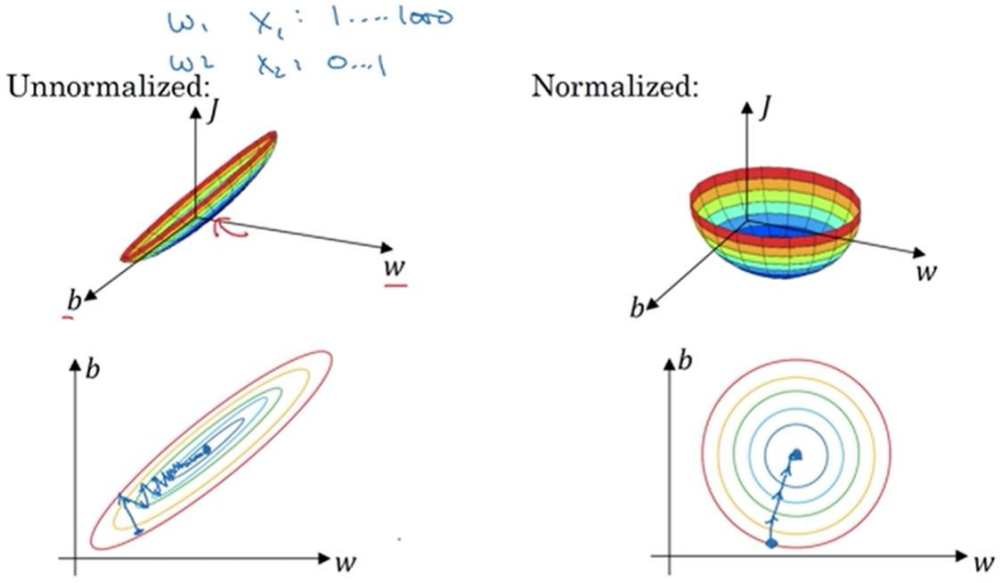
退出迭代的方式同时解决上述两个问题。但是可能存在着在开发集成本函数极值点出现时，训练集拟合度还不高，这样就无法达到预测的目的。

1.9 归一化输入



为了提高算法效率，需要对输入进行归一化。以上图为例，原始输入 X 是一个二维列向量，如第一个图像所示。归一化分为两个步骤：第一步是零均值化，即令$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$，$X = X - \mu$，这样就使得 x_1、x_2 的均值都为 0，如第二个图像所示；第二步是归一化方差，即令$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$（由于已经完成零均值化，$\sigma^2$ 即为方差），$x = x / \sigma^2$，这样就令x_1、x_2 的方差均相等，如第三个图像所示。直观上理解，对于 m 维列向量 X，归一化输入就是将其图像变成一个中心在原点的“高维正方体”。

注意：若对训练集进行归一化，务必要用相同的 μ 和 σ^2 对测试集进行归一化。

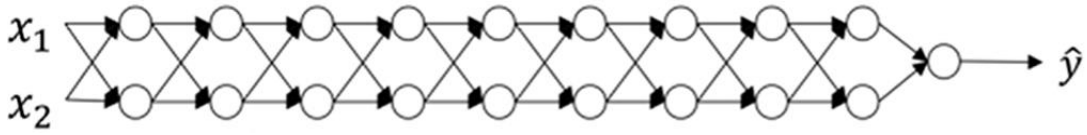


仍以原始输入 X 是一个二维列向量为例，如上图所示，假如不进行归一化输入，那么 x_1、x_2 的范围可能会差异很大，这样会导致成本函数变得“狭长”，这样的函数学习率不能过高，所以梯度下降可能会很慢；但归一化输入后，成本函数就较为“圆滑”，这样的函数有利于提高优化算法的效率。

1.10 梯度消失与梯度爆炸

梯度消失和梯度爆炸指的是在训练神经网络过程中，导数有时会变得很小，

有时会变得非常大的现象。本节将会介绍这种情况产生的原因。



不妨设上图的神经网络激活函数均为恒等函数 $g(z)=z$，则

$\hat{y}=w^{[l]}w^{[l-1]}\ldots w^{[2]}w^{[1]}X$（1），由（1）式可知，在正向传播过程中，输入 X 会乘上很多的系数，这种系数以指数形式增长，在神经网络很深的情况下，这种指数增长会使得 $\hat{y}$ 趋于 0 或者增长很大，这样就会极大地降低迭代的效率。

1.11 神经网络的权重初始化

上一节中所提到的梯度消失与梯度爆炸问题至今仍未有一个完全有效的解决方法，但是更好地、更谨慎地进行权重随机初始化可以有效缓解上述问题。

对于单个神经元 $z=w_1x_1+w_2x_2+\cdots+w_nx_n$，观察该式不难发现，z 会随着 n 的增大而增大。我们想要 z 的均值为 1，那么我们就只需要令 w_i 的方差为 $\frac{1}{n}$。

由上述推导可知，对于一层神经网络，我们可以进行如下的初始化方式

$$W^{[l]}=np.random.randn\left(n^{[l-1]},n^{[l]}\right)*\left(\frac{1}{n^{[l-1]}}\right)$$

对于不同的激活函数，可以有不同的初始化方式

ReLU 函数：$W^{[l]}=np.random.randn\left(n^{[l-1]},n^{[l]}\right)*np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$

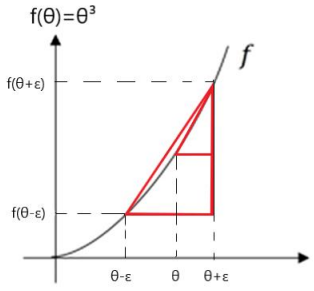
Tanh 函数：$W^{[l]}=np.random.randn\left(n^{[l-1]},n^{[l]}\right)*np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

或 $W^{[l]}=np.random.randn\left(n^{[l-1]},n^{[l]}\right)*np.sqrt\left(\frac{2}{n^{[l-1]}+n^{[l]}}\right)$

1.12 梯度的数值逼近

在实施反向传播过程中，有一种叫做梯度检验的测试，可以确保反向传播的正确进行。为了建立梯度检验，我们首先要了解如何对梯度进行数值逼近。

以 $f(\theta)=\theta^3$ 为例



由导数的定义 $f'(\theta)=\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}$ 可知，双边误差 $\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}$（即大三角形底角的正切）比单边误差 $\frac{f(\theta+\varepsilon)-f(\theta)}{\varepsilon}$（即小三角形底角的正切），更接近该点导数的真实值。实践证明，当 $\varepsilon=0.01$ 时，逼近的误差就已经接近 0.00001，故我们可以利用双边误差对函数的梯度进行数值逼近。

1.13 梯度检验

本节将介绍梯度检验的实现步骤

首先，将$W^{[1]}$、$b^{[1]}$、$W^{[2]}$、$b^{[2]}$、…、$W^{[L]}$、$b^{[L]}$转化为一个向量θ

相应地,将$dW^{[1]}$、$db^{[1]}$、$dW^{[2]}$、$db^{[2]}$、…、$dW^{[L]}$、$db^{[L]}$转化为一个向量$\mathrm{d}\theta$

根据隐层层数，将向量θ展开，即 $\theta=(\theta_1,\theta_2,\cdots,\theta_L)$,相应地，将向量$d\theta$展开，即 $d\theta=(d\theta_1,d\theta_2,\cdots,d\theta_L)$； $J(\theta)=J(\theta_1,\theta_2,\cdots,\theta_L)$

对于向量$\mathrm{d}\theta$的每个元素 $\mathrm{d}\theta_{approx}^{[i]}=\frac{J(\theta_1,\theta_2,\cdots,\theta_i+\varepsilon,\cdots)-J(\theta_1,\theta_2,\cdots,\theta_i-\varepsilon,\ldots)}{2\varepsilon}$

最终得到$\mathrm{d}\theta_{approx}=(\mathrm{d}\theta_{approx}^{[1]},\mathrm{d}\theta_{approx}^{[2]},\ldots,\mathrm{d}\theta_{approx}^{[L]})$

现在，我们得到各梯度的逼近值$\mathrm{d}\theta_{approx}$与精确值$d\theta$，下面将比较两者是否相近

计算：偏移率$\delta=\frac{\|\mathrm{d}\theta_{approx}-\mathrm{d}\theta\|_2}{\|\mathrm{d}\theta_{approx}\|_2+\|\mathrm{d}\theta\|_2}$

①若$\delta=a\times10^{-7}$,则梯度计算基本无误,基本不用检查 ②若$\delta=a\times10^{-5}$,则梯度计算可能出错，需要检查 ③若$\delta=a\times10^{-3}$，则梯度计算大概率出错，必须检查 ④若$\delta>a\times10^{1}$，再不去检查等着挨骂呢？

1.14 关于梯度检验实现的注记

本节将介绍一些在实现梯度检验时的注意事项

①不要在训练神经网络过程中使用梯度检验，只是在 debug 时候用：梯度检验过程很慢，训练过程中没必要开启，浪费资源

②如果梯度检验失败，先逐项检查：如果梯度检查失败，首先应该遍历向量$\mathrm{d}\theta_{approx}$的找出使得偏差出现的$\mathrm{d}\theta_{approx}^{[i]}$，确定 bug 出现的位置

③注意正则项：注意到正则化后的成本函数 $J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}\|w\|_2^2$ 中含有正则项，在梯度数值逼近时不要落下

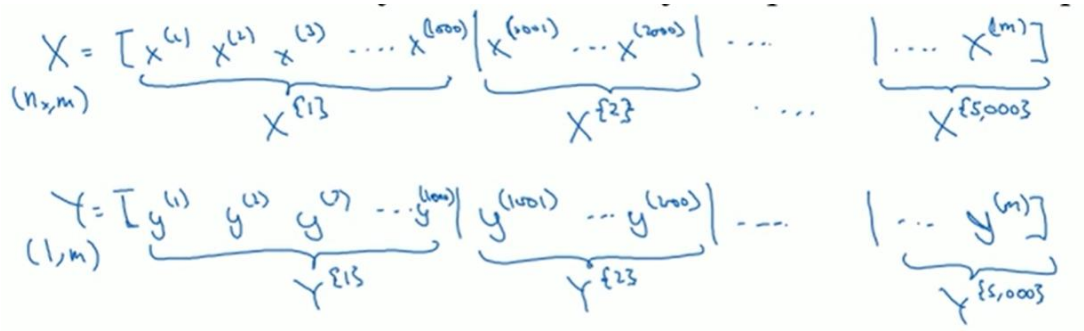
④梯度检验与 Dropout 不能同时使用：使用 Dropout 后，成本函数就没有一个明确的定义，并且会随着迭代而变化，使用梯度检验没有意义；可以先关闭 Dropout，进行梯度检验无误后再开启

第二周 优化算法

2.1 Mini-batch 梯度下降法

深度学习在实际应用中，往往面临着训练集巨大的问题，这使得神经网络的运行时间特别长，传统的 Batch gradient descent（批梯度下降）要遍历整个训练集才能计算相应地成本函数，更新相应的参数，这种方式处理较大的训练集时计算速度慢，计算时间过长。Mini-batch gradient descent（小批梯度下降）将训练集分为若干个子集，每个子集分别进行正向传播与反向传播并计算成本函数，更新相应的参数，这种化整为零的思想可以提高计算的效率。下面将介绍 Mini-batch 梯度下降法的基本步骤。

假设训练集有 500 万个样本，Mini-batch 梯度下降法首先要做的是按照一定的步长对训练集进行划分，本节中以 1000 为步长进行划分



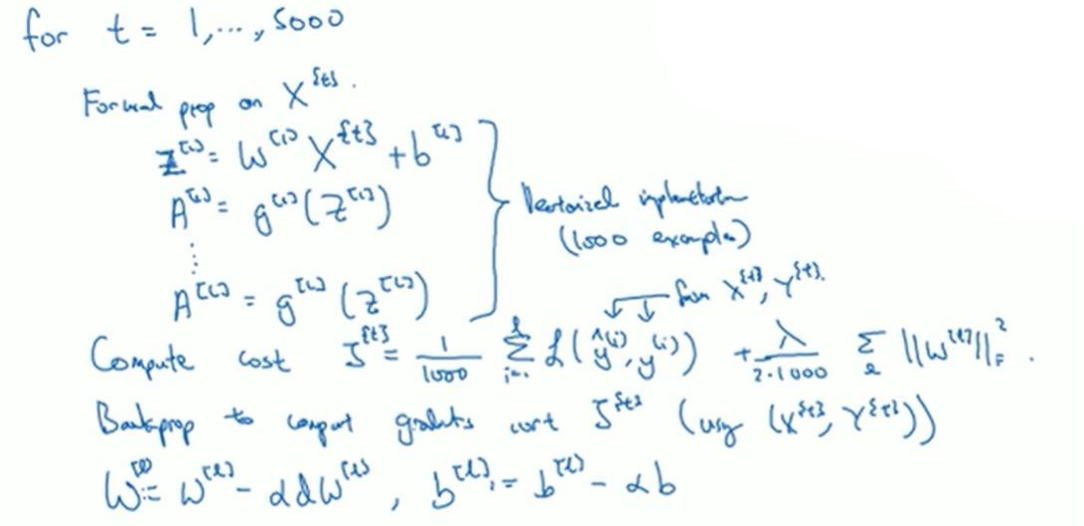
$$X = \left[x^{(1)}, \cdots x^{(1000)}, x^{(1001)}, \cdots, x^{(2000)}, \cdots, \cdots x^{(m)}\right] = \left[X^{\{1\}}, X^{\{2\}}, \cdots, X^{\{5000\}}\right]$$

$$Y = \left[y^{(1)}, \cdots y^{(1000)}, y^{(1001)}, \cdots, y^{(2000)}, \cdots, \cdots y^{(m)}\right] = \left[Y^{\{1\}}, Y^{\{2\}}, \cdots, Y^{\{5000\}}\right]$$

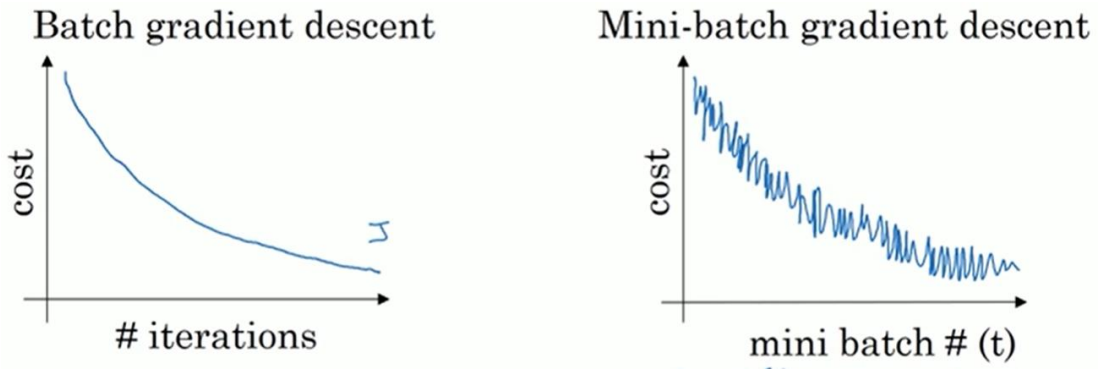
考察子集 $X^{\{t\}}$，$Y^{\{t\}}$ 是一个 n_x×1000 矩阵，$Y^{\{t\}}$ 是一个 1×1000 矩阵

复习：$X^{(i)}$：第 i 个样本、$A^{[l]}$：第 l 隐层输出、$X^{\{t\}}$：第 t 个 Mini-batch

在完成了对训练集的划分后，就可以将每个 Mini-batch 视为一个单独的训练集，应用梯度下降法得到最优解，并用一个 for 循环来遍历所有的 Mini-batch，伪代码如下图所示。

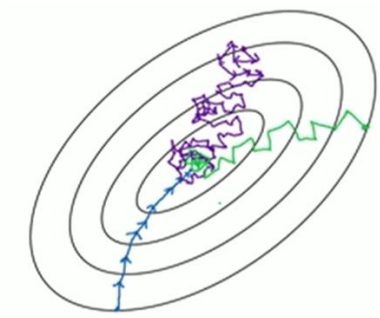


2.2 理解 Mini-batch 梯度下降法



由上图所示，在批梯度下降法中，成本函数 J 是迭代次数的严格递减函数，但在小批梯度下降法中，成本函数 J 总体上随着迭代次数增加而下降，但会有一定的摆动。

对于 Mini-batch 梯度下降法，Mini-batch 大小的选择会对整体效果造成较大的影响，具体如下表

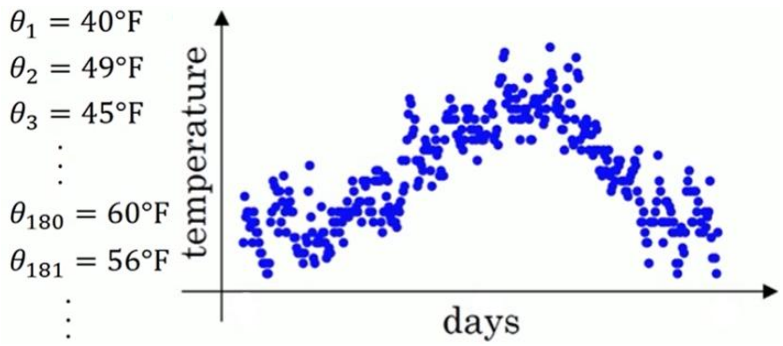
Mini-batch size	1	(1,m)	m
名称	随机梯度下降	小批梯度下降	批梯度下降
特点 	如紫线所示，随机梯度下降过程会有很大的噪音，并且不会收敛到最优解，只是在最优解附近震荡；单次迭代时间短，但失去了向量化所带来的效率	如绿线所示，小批梯度下降过程会有一定摆动，但大体上还是递减收敛到最优解；单次迭代时间短并且能够有效利用向量化	如蓝线所示，批梯度下降过程会严格递减收敛到最优解；但单次迭代时间长，计算量大

Mini-batch size 的选择方法：①训练集较小时（$m < 2000$），直接使用批梯度下降（$Mini-batch\ size = m$） ②一般情况下，$Mini-batch\ size = 64,128,256,512$，使用 2 的 n 次方会提高运算效率

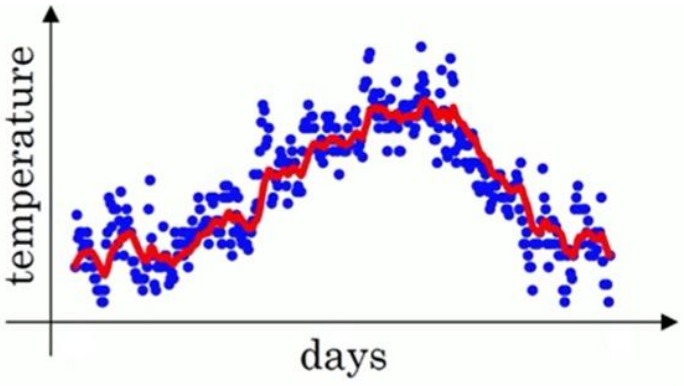
2.3 指数加权平均

本节将以预测伦敦一年的气温为例讲解指数加权平均

首先，我们得到气温数据，并将其绘制为散点图，如下图所示



为了估计气温的变化趋势，我们可以利用计算局部平均值的方式来对气温变化趋势进行估计，将下式中的v_i绘制如右图所示

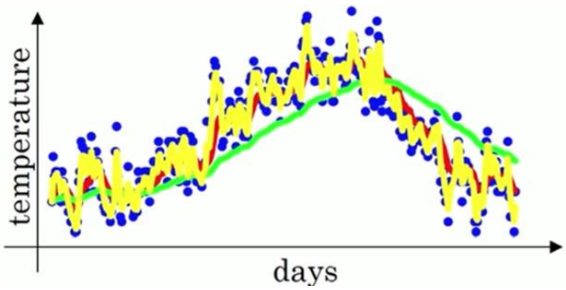


$$v_0 = 0$$
$$v_1 = 0.9v_0 + 0.1\theta_1$$
$$v_2 = 0.9v_1 + 0.1\theta_2$$
$$\cdots$$
$$v_m = 0.9v_{m-1} + 0.1\theta_m$$

由上述推导，我们得到指数加权平均的一般形式：$v_t = \beta v_{t-1} + (1-\beta)\theta_t$

直观上，对于权重β，可以有这样的理解方式：每个值是$\frac{1}{1-\beta}$天的加权平均，权重逐天递增。

$\beta = 0.5$，可以视为两天的加权平均，这样拟合曲线有较大的噪声，但曲线能较快反应气温变化。



$\beta = 0.98$，可以视为五十天的加权平均，这样曲线明显右偏，对于气温变化存在着明显的滞后现象，原因是前 49 天的气温占了较大比重，第 50 天的气温对整体影响被削弱。

2.4 理解指数加权平均

对于式子$v_t = \beta v_{t-1} + (1-\beta)\theta_t$，以$T = 100$，$\beta = 0.9$ 为例，考察v_{100}的结构

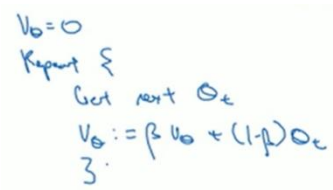
$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$
$$v_{98} = 0.9v_{97} + 0.1\theta_{98}$$
$$\cdots$$
$$v_1 = 0.9v_0 + 0.1\theta_0$$
$$v_0 = 0$$

$$v_{100} = 0.1\theta_{100} + 0.9v_{99} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9v_{48}) = \ldots$$
$$= 0.1\theta_{100} + 0.1 \times 0.9\theta_{99} + 0.1 \times (0.9)^2\theta_{98} + 0.1 \times (0.9)^3\theta_{97} + \ldots$$

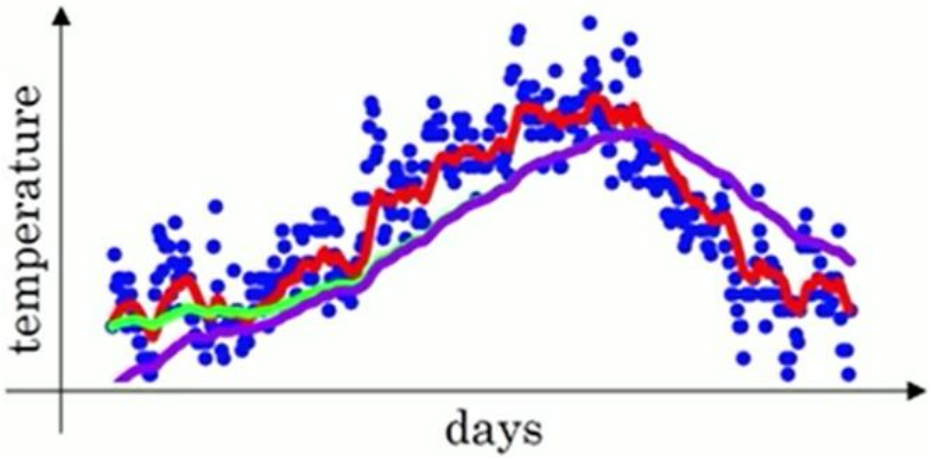
由v_{100}的结构可知，对于v_{100}前的项，与v_{100}距离越大，其权重越低

考察式子$[1-(1-\beta)]^{\frac{1}{1-\beta}}$，该式子的极限为$\frac{1}{e}$，所以说当$t < \frac{1}{1-\beta}$时，计算指数加权平均时，其权重可以忽略，所以说，粗略地看，指数加权平均可以视为v_t前$\frac{1}{1-\beta}$项权重逐次递增的加权平均。这样就解释了 2.3 节中$\beta = 0.9$时，我们可以视为 10 天的加权平均。

指数加权平均的实现也很简单，如右图所示，只需要使用 for 循环对v_t进行更新即可。



2.5 指数加权平均的偏差修正



对于 2.3 节中 $\beta = 0.98$，实际上，当按照 2.3 节中的公式执行运算时，会得到上图的紫色曲线。得出这个结果的原因并不难理解，对于 $v_1 = 0.98v_0 + 0.02\theta_0$，$v_0 = 0$，可以得出 $v_1 = 0.02\theta_0$，这样 v_1与θ_0就存在着较大的偏差，即紫色曲线所示的情况。

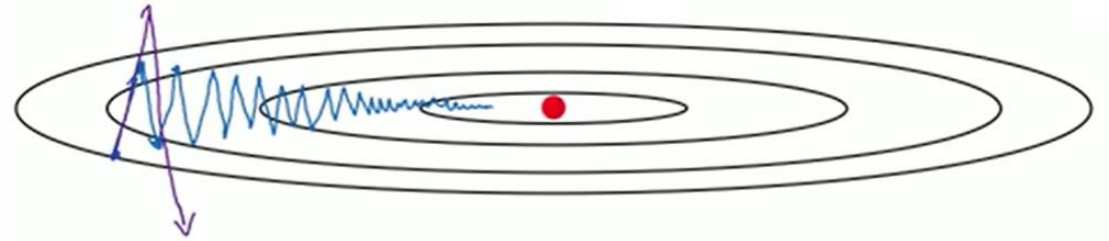
由于初始化 $v_0 = 0$，在指数加权平均初始时刻加权平均得到的预测值与真实值偏差较大，为了校对这种偏差，我们引入如下公式：

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t,\ \ v_t = \frac{v_t}{1-\beta^t}$$

在完成指数加权平均后，我们对结果进行放缩，这样可以有效避免前期偏差过大的现象出现，而随着 t 的不断增大，β^t 会很快衰减至 0，这样就得到了上图的绿色曲线。

2.6 动量梯度下降法

梯度下降法的缺点



在利用梯度下降法求最优解的过程中，我们很有可能遇到如上图所示的成本函数，这样我们就必须把学习率降低，否则在纵向上很可能超出成本函数的定义域；而图中成本函数横向上却需要一个较高的学习率，调低学习率会增加迭代次数，使收敛速度变慢。

动量梯度下降法在原有正向传播、反向传播等计算的基础上，增加求梯度指数加权平均值的计算，并用梯度的指数加权平均代替梯度进行迭代，具体方法如下所示：

$V_{\mathrm{d}w} = \beta V_{\mathrm{d}w} + (1-\beta)\mathrm{d}W$ ，$V_{\mathrm{d}b} = \beta V_{\mathrm{d}b} + (1-\beta)\mathrm{d}b$

$W = W - \alpha V_{\mathrm{d}W}$ ，$b = b - \alpha V_{\mathrm{d}b}$

解释：

如上图所示，纵向梯度一直处于摆动状态，其均值趋于 0，而横向梯度则基本为常数，这样就可以在纵向保持一个较低的步长而横向保持较大的步长，解决

了梯度下降法部分情况下学习率低的问题。

从物理学的角度类比，动量梯度下降法使得每次迭代均与前几次迭代相关联，同时，本次迭代也会影响后几次迭代，这样的做法可以类比成物理学中的动量（惯性）。（我觉得关系不大）

注意：①一般情况下，在动量梯度下降法中几乎不会使用偏差修正的方法，因为在迭代 20 次左右，初始条件偏差的影响就会变得很小 ②β 的最优参考值大概为 0.9 ③指数加权平均写成如下形式也是可以的

$V_{\mathrm{d}w}=\beta V_{\mathrm{d}w}+\mathrm{d}W$ ，$V_{db}=\beta V_{\mathrm{d}b}+\mathrm{d}b$

2.7 RMSprop

同动量梯度下降法功能类似，RMSprop 算法也是为了解决梯度下降法中横轴和纵轴（实际上大多数时候是高维空间，为了方便，在此以二维平面为例）所需的学习率不一致导致算法效率低下的问题。

RMSprop 算法是在原有正向传播、反向传播的基础上，加入了求加权方均根的步骤，具体方法如下：

$S_{dW}=\beta S_{dW}+(1-\beta)(dW)^2$ ，$S_{db}=\beta S_{db}+(1-\beta)(db)^2$

$W=W-\alpha\frac{\mathrm{d}W}{\sqrt{S_{dW}}}$ ，$b=b-\alpha\frac{\mathrm{d}b}{\sqrt{S_{db}}}$

注：在神经网络中 dW 是一个矩阵，$(dW)^2$是矩阵所有元素的平方

解释：以 W 为例，当 W 的梯度波动很大时，dW 相应地会很大，所以S_{dW} 也会很大。用 dW 除以一个较大的数，可以在不减小学习率α的条件下，减小 W 的波动程度。

注意：①在实际操作中，为了避免出现除 0 错误，W、b的更新公式更改如下：$W=W-\alpha\frac{\mathrm{d}W}{\sqrt{S_{dW}}+\varepsilon}$ ，$b=b-\alpha\frac{\mathrm{d}b}{\sqrt{S_{db}}+\varepsilon}$，在分母加上一个 ε，防止出现除 0 错误，一般地 $\varepsilon=10^{-8}$ ②实际操作中，往往 RMSprop 会和动量梯度下降法一同使用，现做出规定动量梯度下降法中的 β 为 β_1，RMSprop 中的 β 为 β_2

2.8 Adam 优化算法

Adam 优化算法实际上是将动量梯度下降法和 RMSprop 算法同时使用，实践证明，Adam 优化算法是为数不多广泛适用于各种结构的神经网络。其具体方法如下：（假设已经完成前向传播和反向传播）

$V_{\mathrm{d}w}=\beta_1 V_{\mathrm{d}w}+(1-\beta_1)\mathrm{d}W$ ，$V_{\mathrm{d}b}=\beta_1 V_{\mathrm{d}b}+(1-\beta_1)\mathrm{d}b$

$S_{dW}=\beta_2 S_{dW}+(1-\beta_2)(dW)^2$ ，$S_{db}=\beta_2 S_{db}+(1-\beta_2)(db)^2$

$V_{\mathrm{d}W}^{correct}=V_{\mathrm{d}W}/(1-\beta_1^t)$ ，$V_{\mathrm{d}b}^{correct}=V_{\mathrm{d}b}/(1-\beta_1^t)$ 注意：Adam 算法需要偏差修正

$S_{\mathrm{d}W}^{correct}=S_{\mathrm{d}W}/(1-\beta_2^t)$ ，$S_{\mathrm{d}b}^{correct}=S_{\mathrm{d}b}/(1-\beta_2^t)$

$W=W-\alpha\frac{V_{\mathrm{d}W}^{correct}}{\sqrt{S_{\mathrm{d}W}^{correct}}+\varepsilon}$ ，$b=b-\alpha\frac{V_{\mathrm{d}b}^{correct}}{\sqrt{S_{\mathrm{d}b}^{correct}}+\varepsilon}$

超参数设置：

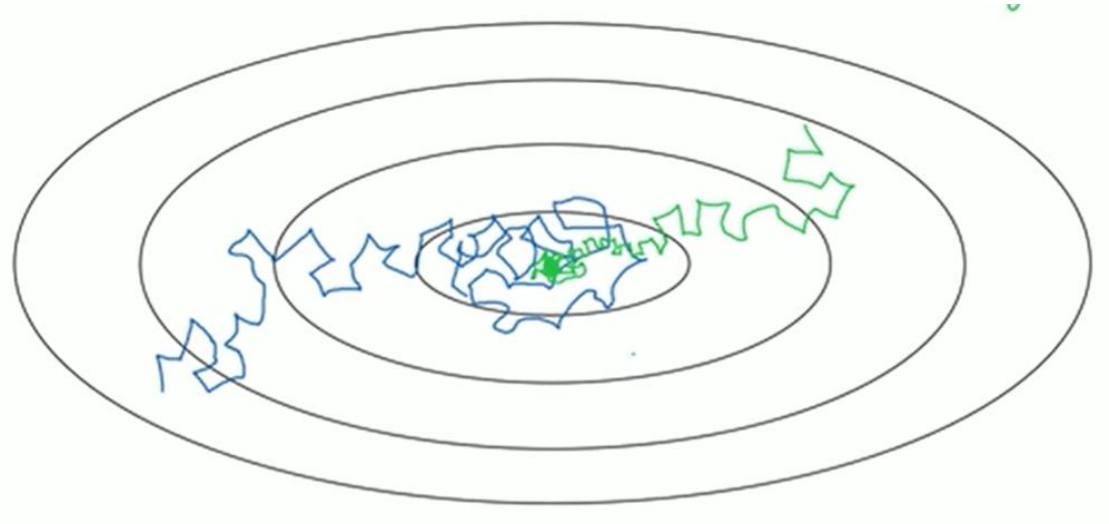
学习率α：没有固定的范围，需要在调试中确定

β_1：一般取$\beta_1=0.9$，也可以在调试中确定

β_2：一般取$\beta_2=0.999$，也可以在调试中确定

ε：一般取$\varepsilon=10^{-8}$，没人会去调试这个

2.9 学习率衰减



在 Mini-batch 梯度下降法迭代的过程中，由于 Mini-batch size 的选取可能比较小，这样会造成迭代不收敛至最优解，而是在最优解一个较大的邻域内震荡，如蓝色曲线所示，这个时候就需要适当缩减学习率，使其收敛至最优解一个较小的邻域内，如绿色曲线所示。

学习率衰减顾名思义，就是将学习率设置为一个迭代次数的递减函数，使得学习率随着迭代次数的增加而减少，进而达到上述目的。实现学习率衰减的函数比较多样，下面介绍几种常用的函数。

① $\alpha = \frac{1}{1+DecayRate\times EpochNum}\alpha_0$

② $\alpha = 0.95^{EpochNum}\cdot\alpha_0$

③ $\alpha = \frac{k}{\sqrt{EpochNum}}\cdot\alpha_0$

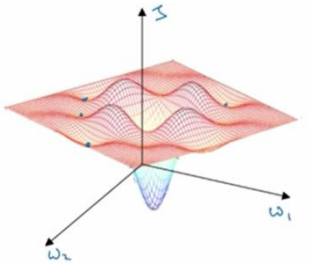
③离散衰减：

④ 手动调参：有多少人工，就有多少智能

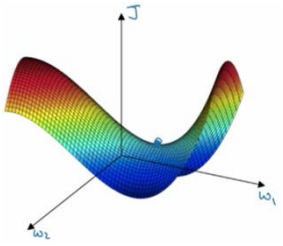
注意：$EpochNum$ 指的是遍历整个训练集的次数（不是 Mini-batch）

2.10 局部最优的问题

在深度学习的早期，人们往往会担心算法陷入一个很差的局部最优解，例如右图上。但事实上，成本函数往往被定义在一个很高维的空间（例如计算机视觉），在高维空间中，局部最优解并不是一个很大的问题，因为想要令所有的方向均为极大值或均为极小值，这样的概率特别小，基本可以忽略。



但深度学习中存在着遇到鞍点的问题，如右图下所示。不同于极值点需要所有的方向均为极大值或均为极小值，鞍点只需要所有方向导数均为 0，这样就大大增加了遭遇鞍点的概率。鞍点附近成本函数梯度趋于 0，会极大减慢算法迭代的速率。但鞍点问题也正是动量梯度下降法、RMSprop、Adam 算法等优化算法发挥作用的地方。



第三周 超参数调试、Batch 正则化和程序框架

3.1 调试处理

超参数的重要程度：

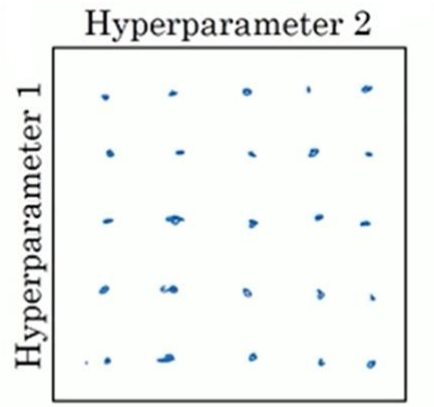
第一重要：学习率 α

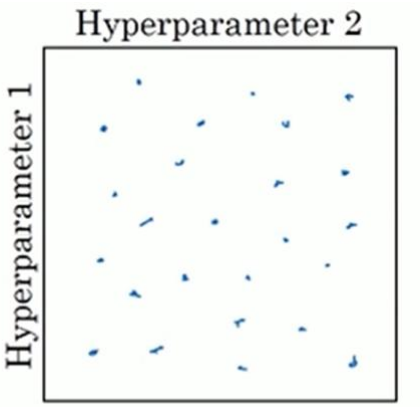
第二重要：动量梯度下降的 β 、隐层神经元数、Mini-batch size

第三重要：Adam 算法的 β_1、β_2、ε（几乎不调试）

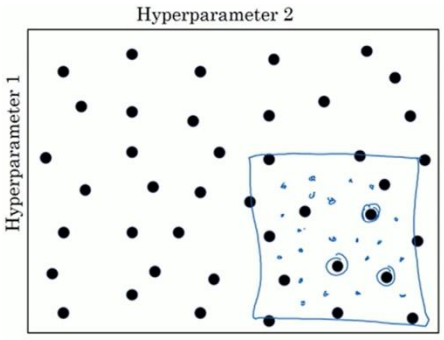
超参数取值的选择：本节以两个超参数为例，多个超参数的调试过程类似

随机取点：对于多个超参数，要进行如右图所示的随机取点而不是如左图所示的均匀取点。





对于两个重要程度差别较大的超参数，例如 α 和 ε，如左图所示的均匀取点最终的实际结果相当于是取了 5 个α 和 1 个 ε，造成了资源的浪费，所以说在多个超参数进行调试时，要随机取点，然后再进行分析。



精确搜索：在进行随机取点后，如果发现某一区域内出现比较好的结果，那么就在这个区域内进一步进行随机取点，找到更好的结果重复数次，直到找到最优结果。

3.2 为超参数选择合适的范围

不同的超参数有不同的最适范围，在最适范围内寻找最优解，需要在该范围内取样，很多时候均匀分布的取样并不能十分正确地找出最优解，这样就需要一些其他的取样方式。

以学习率 α 为例，假设 α 的最优范围为$(0.9, 0.999)$，如果使用均匀分布的抽样方式，90%的资源将会用于区间$(0.9, 0.9891)$的搜索，而 α 的比较敏感的区域是接近 1 的邻域，这样的搜索方式显然存在问题。

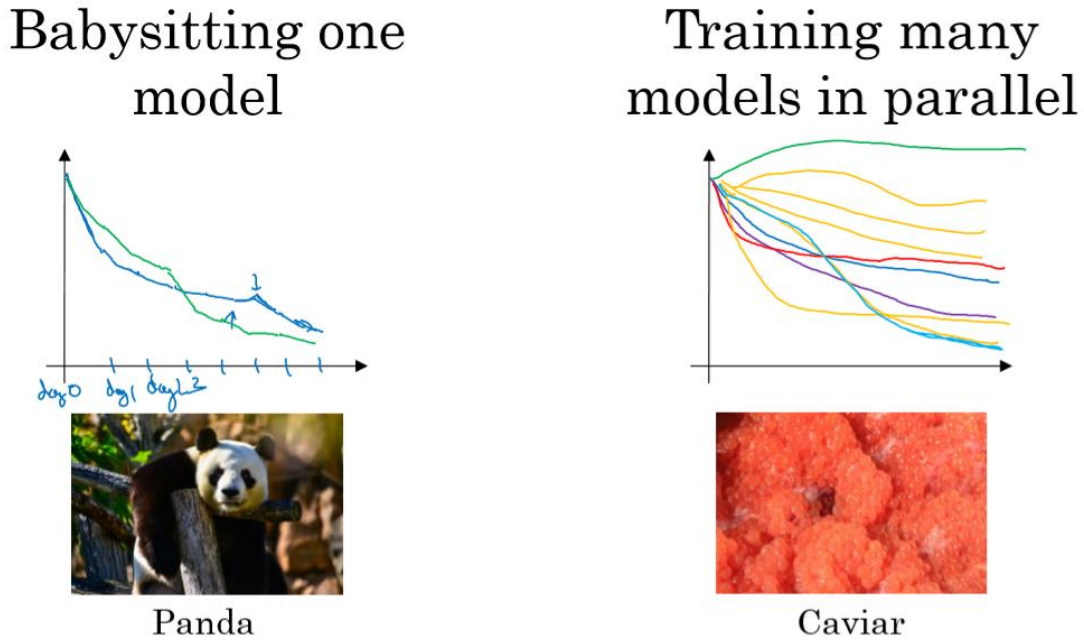
为了方便讨论，我们令 $\alpha' = 1 - \alpha$，原问题转化为 α' 在区间$(0.001, 0.1)$上的最优解问题。我们不妨将区间$(0.001, 0.1)$划分为$(0.001, 0.01)$和$(0.01, 0.1)$，即$(10^{-3}, 10^{-2})$和$(10^{-2}, 10^{-1})$。这样，我们可以在区间$(-3, -1)$上均匀取值，然后再进行乘方运算，具体步骤如下。

$r = -4 * \text{np.random.rand}()$

$\alpha' = 10^r$

3.3 超参数训练的实践：Pandas VS Caviar

调试超参数有两种可供选择的模式：Pandas model（熊猫模式）和 Caviar model（鱼子酱模式），两种模式的选择取决于你所拥有的计算资源。



如上图所示，熊猫模式是指：在模型训练过程中，根据成本函数的大小以及变化趋势逐渐调整超参数，进而得到最优解。这种模式一般在计算资源有限的情况下使用；鱼子酱模式是指：在模型训练之初就编写多个不同超参数的模型，各个模型同时进行训练，最终取结果最优的模型。

3.4 正则化的网络激活函数

在 1.9 节中，我们介绍了 logistic 回归中归一化输入的问题，先计算样本的方差和均值，再减去均值，除以方差得到归一化的输入，

即 $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ ，$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$，令 $X = X - \mu$， $x = x / \sigma^2$

本节将介绍在神经网络中的任一隐层进行归一化输入的方法，具体步骤如下：

现已知某一隐层的输出$z^{[l](1)}$、$z^{[l](2)}$、…、$z^{[l](m)}$,为了方便，简记为$z^{(i)}$

首先计算方差和均值 $\mu = \frac{1}{m}\sum_{i=1}^{m} z^{(i)}$ ，$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (z_i - \mu)^2$

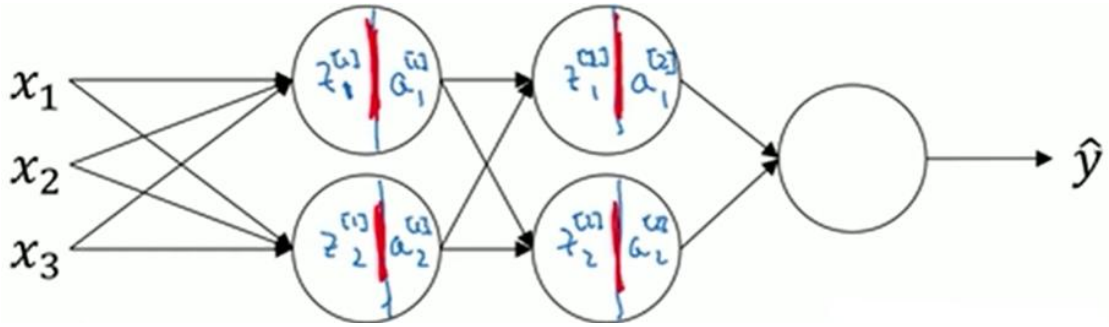
归一化：$Z_{norm}^{[l]} = \frac{z^{[l]} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ ……此处加入 ε 防止因 $\sigma^2 = 0$ 出现除零错误

此处已经将输出变为均值为 0，方差为 1 的归一化输出，但有时不同的分布对一些激活函数有更好的效果，故我们还会做进一步计算。

$\tilde{Z}^{[l]} = \gamma Z_{norm}^{[l]} + \beta$，其中$\gamma$ 、β为可学习的参数，随着迭代而更新

这样，我们用 $\tilde{Z}^{[l]}$ 代替$Z^{[l]}$作为第 l 层的输出，完成了归一化

3.5 将 Batch Norm 拟合进神经网络



以图示的神经网络为例，Batch Norm 流程图如下

$$X \xrightarrow{W^{[1]},b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]},\gamma^{[1]}} \tilde{Z}^{[1]} \rightarrow A^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{W^{[2]},b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]},\gamma^{[2]}} \tilde{Z}^{[2]} \rightarrow A^{[2]}$$

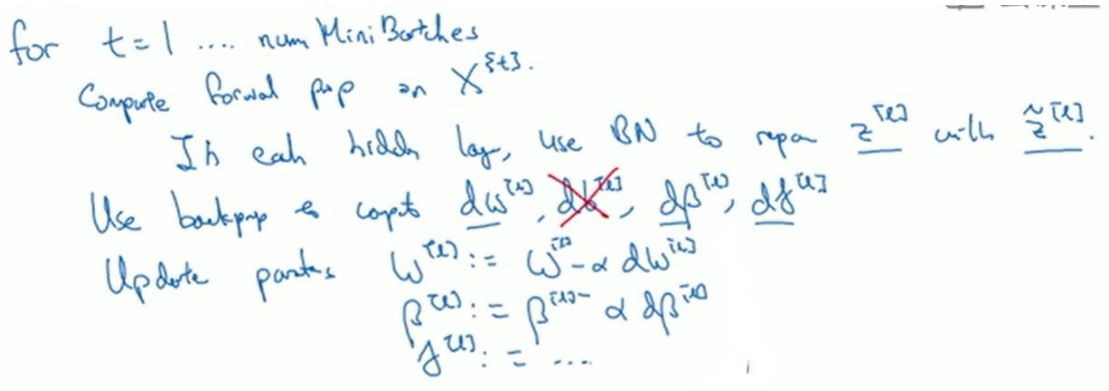
上述在具体实现过程中，往往只需要一行代码 tf.nn.batch.normalization。

注意到：对于进行 Batch Norm 的 $Z^{[l]}$ 而言，$b^{[l]}$ 是否存在对结果没有影响，因为在求平均的过程中，$b^{[l]}$ 的影响会被相应地在平均值中被中和。

在实际操作中，往往 Batch Norm 会和 Mini-batch 相结合使用。具体过程与上述流程类似，只需将 X 替换为 $X^{\{1\}}$ 即可。为了书写方便，在如下表达式中，只有 $X^{\{1\}}$ 显式地表达 Mini-batch 的层数，其余略去。

$$X^{\{1\}} \xrightarrow{W^{[1]},b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]},\gamma^{[1]}} \tilde{Z}^{[1]} \rightarrow A^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{W^{[2]},b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]},\gamma^{[2]}} \tilde{Z}^{[2]} \rightarrow A^{[2]}$$

实现 Batch Norm 的伪代码如下：

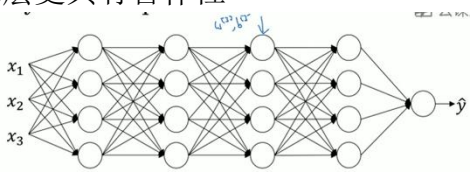


3.6 Batch Norm 为什么奏效

第一，同 1.9 节内容所述，Batch Norm 后的输入使得成本函数的定义域变得“圆滑”、“对称”，这样有利于梯度下降的迭代。

第二，Batch Norm 后的输入使得后面的隐层更具有鲁棒性

以右图所示的神经网络为例，考察其第三层，第三层的输入为 $A^{[2]}$ 。如果不进行 Batch Norm，对于不同的输入X，$A^{[2]}$的取值可能十



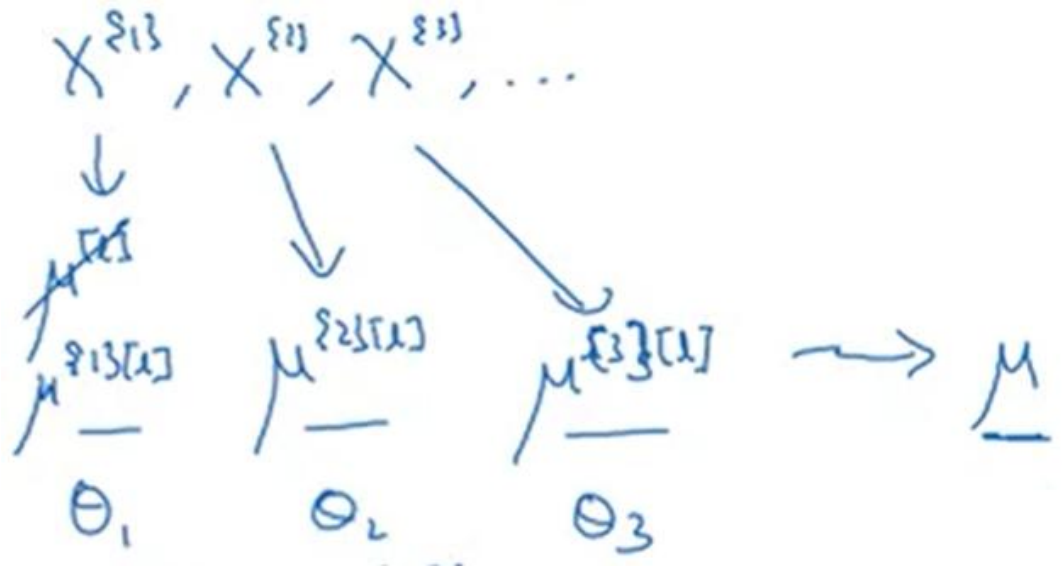
分分散并且规律性不大，这样就给第三层的学习造成了很大的障碍。使用了 Batch Norm 后，$A^{[2]}$的均值和方差均被限定在一个较小的范围内，$A^{[2]}$的分布较为集中，使得第三层可以在一个较为稳定的输入分布进行学习。

第三，Batch Norm 也具有一定的正则化效果。

3.7 测试时的 Batch Norm

在训练神经网络时，Batch Norm 将会对整个 Mini-batch 进行处理，进而得到相应地均值和方差。在测试神经网络时，对于单个输入 X 计算均值和方差是没有意义的，但是运行神经网络仍然需要一个均值和方差，本节将介绍这种情况的处理方法。

我们可以采用如下所示指数加权平均的方式来计算一个全局的 μ 和 σ^2

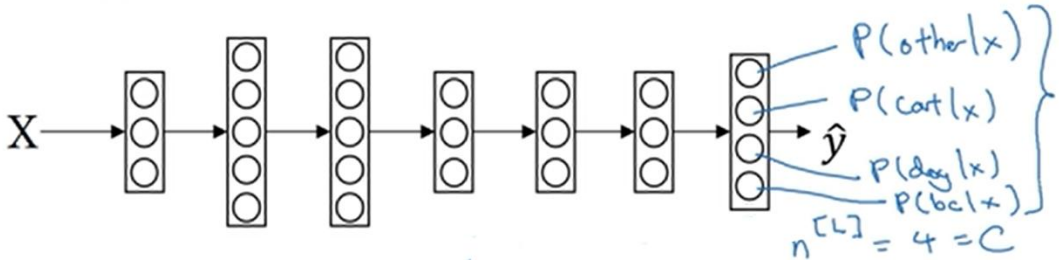


同时，在深度学习框架中也有相应默认的方式来估算全局的均值和方差。

3.8 Softmax 回归

在前方的介绍中，我们介绍都是二分分类，即输出结果只有 0，1 两种情况（不是猫，是猫），在本节中我们将介绍输出结果有多种情况的分类。

假设我们需要进行如下分类：是猫，是狗，是鸟，其他，我们可以构建如下的神经网络



输出层有四个神经元，输出结果是一个四维列向量，从上到下依次对应着其他、猫、狗、鸟的概率，并且纵向求和结果为 1。

分析输出层的输入与输出，输出层的输入为 $A^{[L-1]}$，在上图中是一个四维列

向量；输出层的输出也是一个四维列向量。这样，在输出层我们就需要一个不同于 logistic 回归的激活函数，因为无论是 sigmoid 函数还是 ReLU 函数，都是单值函数，无法实现上述功能。

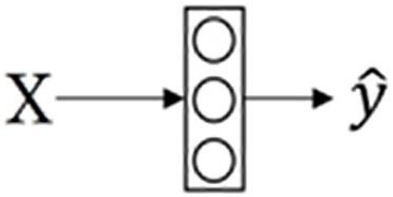
Softmax 激活函数

$$t = \mathrm{e}^{(z^{[L]})},\ a^{[L]} = \frac{\mathrm{e}^{z^{[L]}}}{\sum_{j=1}^{n^{[L]}} t_i}$$

其中$a^{[L]}$是一个$n^{[L]}$维列向量，其各行元素$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{n^{[L]}} t_i}$

直观理解 Softmax 回归

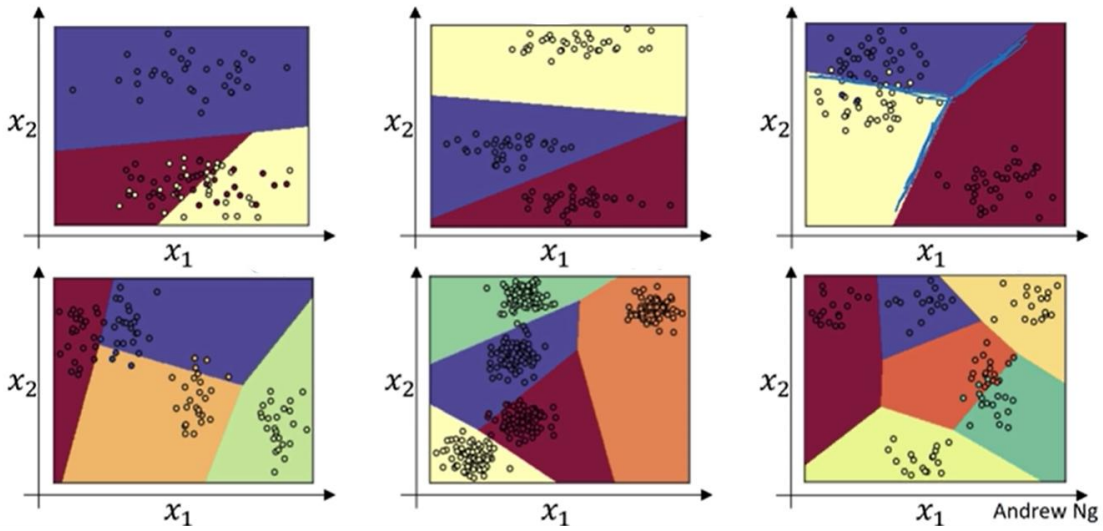
对于如右图所示没有隐层的神经网络，其运行步骤如下式：（不妨设 X 维二维列向量）



$$z^{[1]} = w^{[1]}x + b^{[1]}$$

$$a^{[1]} = \hat{y} = g\left(z^{[1]}\right)$$

其结果直观理解是对向量空间进行了划分，具体如下图所示



对于没有隐层的 Softmax 回归，其仅对空间进行了线性划分，而对于有多个隐层的 Softmax 回归，其可以对空间进行较为复杂的划分，进而实现本节初的功能。