1. data set: training dat, dev, test:

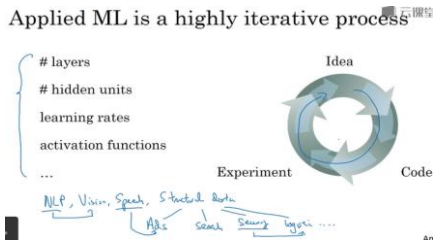
data set: helps for finding high performance neural network.
deep learning is a highly iterative process, try ideas-code-experiment to find better hyperamter values
-->good data seting will faster this process.

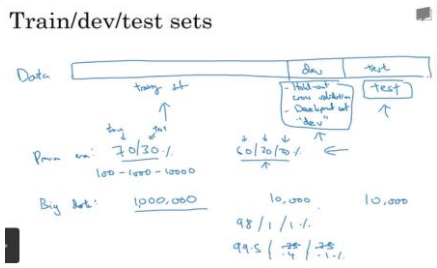
Note: one domain from one application area often do not transfer to other application areas, and hard to find good hyperparameters for one application in the start.**--->highly iterative process to find hyperparameters.**

1.1 data set splitted into: 3 part (data size m is small)
>1. Training set: 60% , train algorithm;
>2. dev set: 20% , evaluate algorithm performance with different models/hyperameters
>3. Test set: 20% ,run final algorithm, get final algorithm performance

data splitling fraction:
when have very large data set, dev, test set fraction could be smaller.
as dev set just need to be big engough to evaluate different algorithm performance choices and quickly decide which one is doing better.
(test set: just need to be big engough to have high confidence for final algorithm performance)

bigger data size, dev, test set fraction smaller. small data size more resonalbe to use 60%, 20%, 20% for splitting.

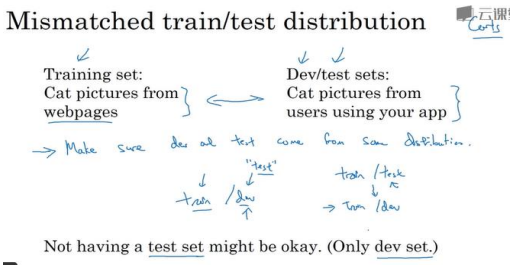




1.2: train , test set mismatch distribution

>1. Make sure dev and test set come from the same distribution.
Dev set used to evlaue different algoirthm choise and decide hyperparameter, so algorithm performance showed in dev, have to keep aligned in test set./real cared data

>2. Training set may mismatch dev/test set distribution:
to get more data for training set , may use all sorts of data. have devation from dev/test set.
but as long as dev and test set is same distrubiton, the progress in machine learning algorithm will be faster
(note: dev set: is the representative of real cared object, + singular number evaluation-->decide algorithm target)



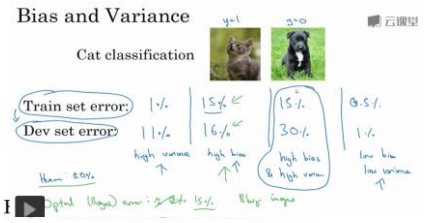
>3. Not haveing a test et might be ok (only dev set)
test set is to give unbias estimate peformance of selected final algorithm . if do not need unbaised algorithm performance, might be ok do not have test set.
(note: or dev set is large enough->algorithm will not overfit dev set: could cover all real cared data feature)

note: since algorithm have fit dev set, performance get in dev set is actually biased.

train , dev, test set also helps for judging bias, variance probelm, helping for improving algorithm performance.

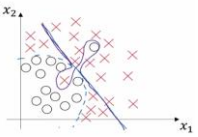
2. Bias and varaiance

Train set error;
Dev set error:
-->
Variance: algorithm's ability to generalize from training set to dev/test set. Reflected by gap between J_dev and J_train:
(note: algorithm generalizibility is decided by:
>>1. algorithm archtecture: esp. when overfit training set-->regularization (influence j_dev mainly)
>>2. training set: big engough to cover dev/test set feature: -->bigger training set (influence j_dev mainly)
)



Bias: algorithm's performance on training set. reflected by gap between J_train and **Bayes error/ J_human (human error).**

may have both bias and varaince error at same time: happens when overfit part of training set and undefit another part of training set. e.g: decision boundary most part is straight line -underfitting most training set, yet in the middle overfit some training examples.



judging bias or varaince based on training error and dev error, assumption/conditons is: training set and dev set are from same distribution.
(note: if training set come from different distribution from dev set : add another 'training-dev' set to judge variance problem)

Note: 贝叶斯误差
贝叶斯误差（bayes error rate)是指在现有特征集上，任意可以基于特征输入进行随机输出的分类器所能达到最小误差。也可以叫做最小误差。

1. 贝叶斯误差的定义有两个关键点：
1）给定特征集后的最小误差：即可以认为我们的训练集无限大且已经按真实分布穷举了所有可能的特征组合后，任何分类器所能达到的误差下限。产生贝叶斯误差的本质原因是特征集不足以推理出准确预测值，否则贝叶斯误差为0.

2）概率特性："最小误差"是限定在概率随机性条件下的（如果是开了实例级别的"天眼"的最小误差，显然无论任何情形下最小误差都是0，这个概念将没有任何意义。贝叶斯误差只能开概率级别的"天眼"）。即： 2.1）如果场景是给定特征输入，输出值是唯一的，则贝叶斯误差为0。例如特征是人的净高和鞋底厚度，回归目标是人的头顶距离地面高度。那贝叶斯误差为0. 在训练集准确的情况下，线性回归也可以达到相同误差。 2.2）反之，如果输出值是有一定随机性的，贝叶斯误差是此时选概率最大输出作为输出值所能达到的误差。比如，特征为头发长短，目标值为男、女分类label。假设在长发下，真实世界中女性概率90%，10%男性。反之女性10%，男性90%。那么贝叶斯概率误差即为认定长发为女，短发为男的误差。显然其贝叶斯误差为10%。

比较重要的是理解输入分布是真实的，但输出只能是一个值，所以会有误差。如果输出也可以是分布，那就没有误差了。

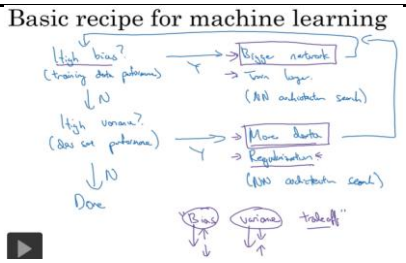
2. 贝叶斯误差的作用:
直观上可以这么理解，贝叶斯误差是在给定特征集的情况下，假设数据无限（且准确），依靠统计所能得到的最小误差。它是我们通过增加数据集/优化数据集分布/提升模型学习能力/防止过拟合等等措施后所能达到的误差下限。如果当前算法已经能达到接近贝叶斯误差的误差，则在不动特征的前提下（比如DNN输入特征为原始文本/像素时，输入特征就动不了）我们已经没有继续优化的意义了。-- 然而，事实上，贝叶斯误差无法求得（因为求得的前提是你知道真实分布，你知道真实分布还做什么机器学习呀）。目前就我知道的，业界并不会真正使用该指标，该指标更多的还是出现在学术探讨中。

3. Basic recipe for machine learing

>1. first , High bias: algorightm performance on training set--------->bigger network (more hidden units/ layer number) or training longer/ new NN architecture
Algorithm shoud at least fit or overfit training set.

>2. High variance: algorithm performance on dev set---------->more data, regulization, / new NN architecture

Above two steps' tool, do not hurting the other one when decreasing bias/variance, so no need to consider trade off bias and variance.
(note: does not mean do not influence. e.g: bigger network: will reduce both bias and variance.)



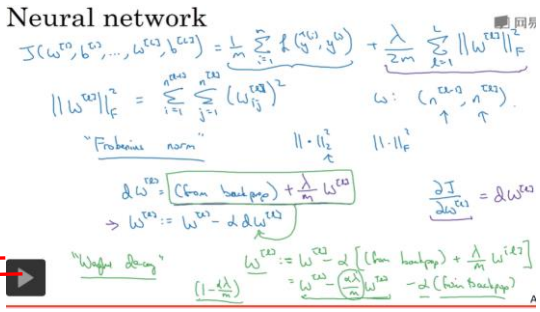
4. Regulization for overfitting

>add weight w in cost function: L2 regulization more popular (logistic regression) / Frobenius norm (sum of square of element in matrix)

>add weight w in cost function: L1 regulization--->
weight will be sparse, W vector will have a lot of 0 in it.?

>3. Could also add b in regulization, but normally do not do it, because:
weight W is high dimensional parameter vector (logistic regression) / matrix (NN), especially with a high variance problem, could not fit all the parameters well. whereas b is just a single value/ vector, so almost all the parameters are in w rather than b.
In practice, adding b does not make much difference, just one paramete over a very large number of parameters.

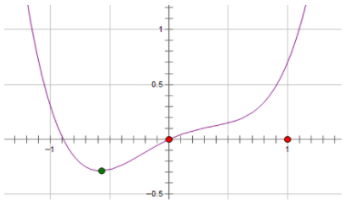
>4. Regulization parameter: lambda: in python will use 'lambd' as lambda is a reserved keyword in python.
>5. L2 regulization also called weight decay: as in gradient descent , L2 also involved--->
W:= (1-lambda/m) W- dW
origian W matrix is mulitiplying a fractor <1



l1 相比于 l2 为什么容易获得稀疏解？

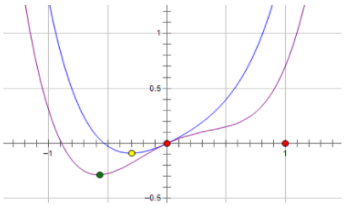
note:
l_0: = f(w): withoug regularization
l_1 = l_0 + L1: with L1 regularization
l_2= l_0 + L2: with L2 regularization
minimize loss: try to make dl_i/dw =0:
L2: dw_2 = dw_0 + lambd/m * w = dw_0 (when w ==0)
L1: dw_1 = dw_0 + C*-1/1 (when w ==0, could set C to make dw_1 ==0)
(只要系数C 大于|dw_0 (w=0)|: w=0就会变成一个极小值点。)

假设费用函数 L 与某个参数 x 的关系如图所示：



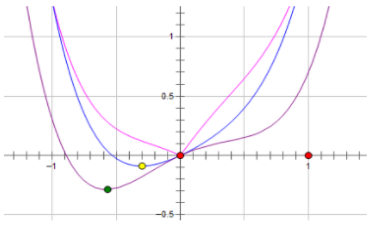
则最优的 x 在绿点处，x 非零。

现在施加 L2 regularization，新的费用函数（$L + Cx^2$）如图中蓝线所示：



最优的 x 在黄点处，x 的绝对值减小了，但依然非零。

而如果施加 L1 regularization，则新的费用函数（$L + C|x|$）如图中粉线所示：



最优的 x 就变成了 0。这里利用的就是绝对值函数的尖峰。

两种 regularization 能不能把最优的 x 变成 0，取决于原先的费用函数在 0 点处的导数。
如果本来导数不为 0，那么施加 L2 regularization 后导数依然不为 0，最优的 x 也不会变成 0。
而施加 L1 regularization 时，只要 regularization 项的系数 C 大于原先费用函数在 0 点处的导数的绝对值，x = 0 就会变成一个极小值点。
上面只分析了一个参数 x。事实上 L1 regularization 会使得许多参数的最优值变成 0，这样模型就稀疏了。

Yihua 2018-12-11
你拿不加正则化项但经过原点的loss函数来讲解会让很多人不明白。应该用：不加L1正则化项loss函数不过原点，加了L1通过调整C的值能使loss函数最小值在原点。而加了L2无论如何调整C也使函数的最小值到不了原点，这样的例子来讲才能让人明白。
4 回复 踩 举报

王婆 Maigo (作者) 回复 Yihua 2018-12-11
嗯……其实我把损失函数向上平移一下就好了……

https://zhuanlan.zhihu.com/p/35356992 Help understanding

L1和L2正则化：

我们所说的正则化，就是在原来的loss function的基础上，加上了一些正则化项或者称为模型复杂度惩罚项。现在我们还是以最熟悉的线性回归为例子。

优化目标：

$$\min\ 1/N * \sum_{i=1}^{N}(y_i - \omega^T x_i)^2 \quad \text{式子（1）}$$

加上L1正则项（lasso回归）：

$$\min\ 1/N * \sum_{i=1}^{N}(y_i - \omega^T x_i)^2 + C||\omega||_1 \quad \text{式子（2）}$$

加上L2正则项（岭回归）：

$$\min\ 1/N * \sum_{i=1}^{N}(y_i - \omega^T x_i)^2 + C||\omega||_2^2 \quad \text{式子（3）}$$

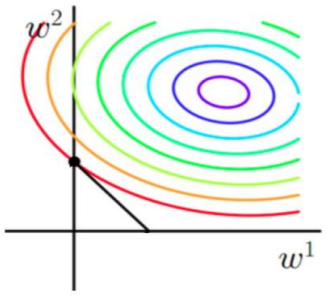
结构风险最小化角度：

结构风险最小化：在经验风险最小化的基础上（也就是训练误差最小化），尽可能采用简单的模型，以此提高泛化预测精度。

那现在我们就看看加了L1正则化和L2正则化之后，目标函数求解的时候，最终解有什么变化。

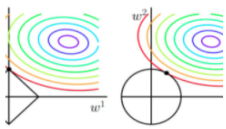
图像解释（假设X为一个二维样本，那么要求解参数 ω 也是二维）：

- 原函数曲线等高线(同颜色曲线上，每一组 ω_1 ，ω_2 带入值都相同)



目标函数等高线

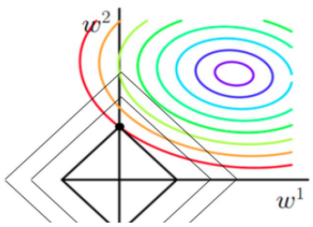
- L1和L2加入后的函数图像：



加入L1和L2正则的等高线

从上边两幅图中我们可以看出：

- 如果不加L1和L2正则化的时候，对于线性回归这种目标函数凸函数的话，我们最终的结果就是最里边的紫色的小圈圈等高线上的点。
- 当加入L1正则化的时候，我们先画出 $|\omega_1| + |\omega_2| = F$ 的图像，也就是一个菱形，代表这些曲线上的点算出来的 1范数 $|\omega_1| + |\omega_2|$ 都为F。那我们现在的目标是不仅是原曲线算得值要小（越来越接近中心的紫色圈圈），还要使得这个菱形越小越好（F越小越好）。那么还和原来一样的话，过中心紫色圈圈的那个菱形明显很大，因此我们要取到一个恰好的值。那么如何求值呢？



带L1正则化的目标函数求解

1. 以同一条原曲线目标等高线来说，现在以最外圈的红色等高线为例，我们看到，对于红色曲线上的每个点都可以做一个菱形，根据上图可知，当这个菱形与某条等高线相切（仅有一个交点）的时候，这个菱形最小，上图相割对比较大的两个菱形对应的1范数更大。

用公式说这个时候能使得在相同的 $1/N * \sum_{i=1}^{N}(y_i - \omega^T x_i)^2$ 下，由于相切的时候的 $C||\omega||_1$ 小，即 $|\omega_1| + |\omega_2|$ 小，所以：

能够使得 $1/N * \sum_{i=1}^{N}(y_i - \omega^T x_i)^2 + C||\omega||_1$ 更小。

2. 有了1.的说明，我们可以看出，最终加入L1范数得到的解，一定是某个菱形和某条原函数等高线的切点。现在有个比较重要的结论来了，**我们经过观察可以看到，几乎对于很多原函数等高曲线，和某个菱形相交的时候及其容易相交在坐标轴（比如上图），也就是说最终的结果，解的某些维度及其容易是0，比如上图最终解是 $\omega =$ （0，x），这也就是我们所说的L1更容易得到稀疏解（解向量中0比较多）的原因。**

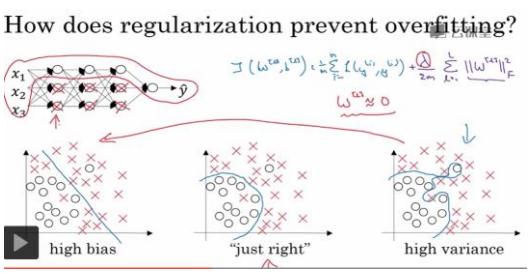
5.How regulization work:

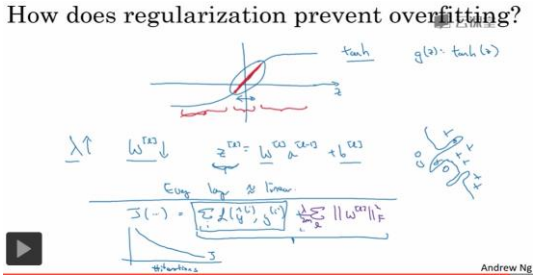
Cost function J= J_0 + regulization
>1. lambda is very big--->many hidden units weight is close 0, elimininting these hidden units influnce, close to logistic regressioin (for each layer)--->simpler NN (might do not change deepth)--->pron to underfitting
find a reasonable lambda--> to just fit

>Intuition 2: e.g activatio function is tanh: regulization make to use linear part of activation function-fast gradient descent.
if lambda is large-->w is smaller--->Z=w.x +b is smaller, reach the rigion of z, corresponds to activatioin linear part g(z) is almost linear,
--->every layer activation function will be like linear function, just like linear regression, -->whole NN is like linear network, could not fit complicated-non linear decsion boundary.-->prevent overfitting

Note:
>1. cost function changed when intro. regulization-->when debugging/plotting cost vs iteration, need to use new cost J, which should be signle decreasing. if use old J without regulizaiton, J curve may not signal decrease, not good for debugging;
>2. When check algorithm performance on dev/test, still use origianl cost without regulization;
regulization is only added in cost function to reduce hidden units influence and train algorithm in training set, while real cost error still be the old formula without regulization.

(note:
Variance: -->trained algorithm ability to generalize from training set to dev set.
High variance: due to : 1. training set feature could not be representative of dev /test set , and or algorithm learned fake features of training set; 2. algorithm learned fake features of training set due to: 'overfit' (may caused by too complicated algorithm or too small training set)
-->solution:
1. first Get more training data (more similar to dev/test set distribution-need do error analyais for mismatch ceiling performance): to cover all features of dev set. then if still have variance
2. intro. regularization: try to reduce weight of fake features algorithm has learned.-since no idea which feature is fake, reduce all features weight.
reducing weight:-->simpler architecture: due to :
1. simpler computation in each layer: less units weight-may less hidden units inovled-less fake features
2. activation function in each layer: pron to like linear function-simpler fitting, learn less complicated function.-less fake features
)

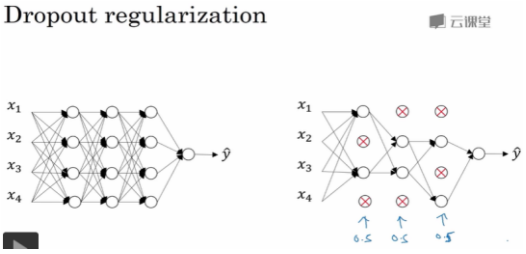




6. Drop out
6.1 Idea: go through each layer unit and set some probability of eliminating a node in neural network.-->get simpler NN
Remove nodes in layers in both forward and backward propagation(one iteration), and train NN on examples.

6.2 Function: have regulization effect as much smaller networks are being trained.

Note: nodes remvoed in layers is randomly,--->different iteration will elimate different nodes in layers. (even same iteration, for different batch samples, elimated nodes also could be different-acc. to possbility matrix: same dimension as a[l])
and for each example, would train it using one of these neural reduced networks.
(different iteration, use different architecture actually)



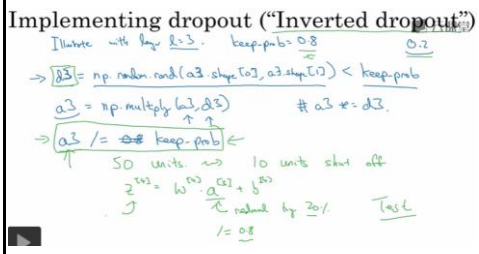
6.3 Implementing method:
>1. inverted dropout.
e.g: adding drop out on layer 3:

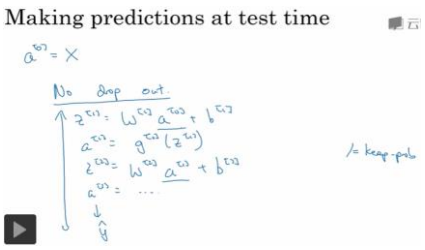
dropout vector: for each example and for each hidden unit, (value is true or false), have same dimension with layer output a[l] (n_l, m)
>>1. have same dimension as A[3]: d3 = np.random.rand (A[3].shape[0], A[3].shape[1]) < keep-prob
>>2. keep-prob is a real value, is the probability that a given hidden unit will be kept.
if keep-prob=0.8-->for each example and for each hidden unit, there's 0.8 chance that the corresponding d3 will be one.

>>3. Activation function/layer output: A[3] = np.multiply (A[3], d[3]) or A[3] *=d3 (element wise)
when use phython, dropout vector is a boolean array : value is true and false rather htan 1 and 0, but multiply operation works and will interpret the true and false values as one and zero.

>>4. A[3]/=keep-prob--called inverted dropout, as nomater what value keep-prob is , A[3] expected mean value remains same.
in order to not reduce the expected value of Z[4].??(drop out is meant to make algorithm simpler, should ouput less mean, why inverted to expected same mean??
If keep-prob=0.8-->A[3] only keep 80% units , others is 0
-->1. Z[4] = w[4]. A[3] + b[4], to keep E (Z[4]) do not change, A[3]/ 0.8, to correct / just a bumpthat back up by the roughly 20% that need, E [A[3]] also do not change much. activation function expected value do not change. (activation expectation and variance influence gradient exploring/vanishing)

--->: Inverted dropout A[3]/=keep-prob, makes test time easiser as have less of a scaling problem.
-->even do not implement dropout at test time to the scaling, the expected value of these activations A[l] do not change.
do not need to add extra scaling parameter at test time, which is different from that at training time. (note: at time test , do not implement dropout , just set keep-prob=1)





Note: drop out
1. for different example, zero out different hidden units; dropout vector dimension is: [n_l, m];
2. for each iteration, also randomly zero out different hidden units: randomly set droput vector in every iteration: for each hidden units in each layer and each example, same dimension as Z[l] /A[l]

>6.5 At test time, no drop out-set keep-prob=1:
as do not want output is random, otherwise just add noise to prediction.
Alternative: run prediction on test time many times with different hidden units randomly dropped out and average cross them.--->computationally inefficient and will give roughly same result of without dropout.

7 Why drop out work

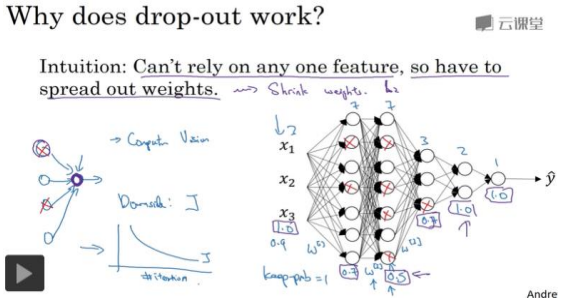
Intuition 1: on every iteration, randomly zeoro out hidden units, works on smaller NN;

regulizaiton effect:
Intuition 2: can not rely on any one feature, so have to spread out weights: (weights smaller??)
for one node: Z[l]_i: W[l]_i . A[l-1] +b[l]_i
Input of layer: A[l-1]: any one of input feature/ anyone of its own input, could go away at random, so reluctant to put all of its bets/too much weight on any one input (feature), as it might go away-->spread weight to all input feature./unite.

Spreading weight tend to have an effect of shrinking the L2 of weights--->sort of regulization effect.:
e.g: without dropt out W= [w1;w2], with drop out W'=1/2 [w1+w2, w1+w2]
-> ||W||^2 -||W'||^2 = 1/2 * (w1-w2) ^2 >=0
->Weight L2 with dropout, becomes smaller.

Note: L2 penalty on different weights/ are different, depending on the size of the activations being muliplied that way.
（for activation function tanh, sigmoid, smaller weight-->smaller Z---> pron to be the region where activation function is linear--->simplier whole neural network).

Summarize:
it's possible to show dropout has a similar effect to L2 regulization (penalizing weight), only the L2 regularization applied to different weights can be different and even more adaptive to the scale of diffrent inputs.
(note: drop out 'even' weight for activations/hidden units in each layer, while L2 regularization do not even weight-balance cost and weight norm)



note:
dropout for regularization: reducing fake feature influnce learned by algorithm from training set.

Reducing each feature weight of each units in each layer , by spreading weights to all units-could not rely on specify feature due to random drop out.-->
1. reducing weight norm ||W||2-regulization effect
2. more robust to hidden units
(note: in this way, seems to force each of the units in each layer learn similar feature-redundant??
over fitting due to fake feauture in hidden units-->drop out make each hidden units learn similar feature, reduce genrating fake features?)

Note:
1. Keep-prob: hyperparameter, can be vary by layer, set low value for layers with many parameter/units, and 1 for layer do not want dropout.
2. Could apply droput to input also: eliminate input feature. normaly set 1 for input layer.

3. Keep-prob: is hyperparameter, need to search using dev set. (how to carry out, seperately for each layer, or make a keep -prob matrix for all layers and try on dev?)

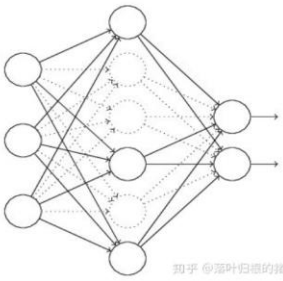
4. Drop out use more often in computer vision, where input image as too many axiles/features （applied also on input?).
and no engough data, so alwasy overfitting.

>5. Dropout have regulization effect, and used to prevent overfitting.

disadvange: debugging tool (plotting J vs iteration)
>1. Cost function is no longer well-defined: every iteration, randomly zero out hidden nodes. while well-defined cost function singal decrease every iteration---> loose debugging tool (plotting J vs iteration)
-->turn off dropout, run code make sure well-definied cost funtion is monotonicaly decreasing, -->turn on drop out-hope works well.

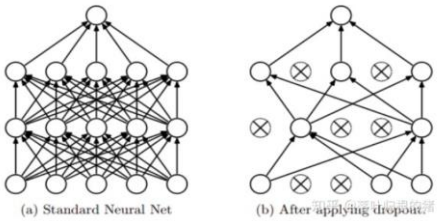
2. 什么是dropout? —— Dropout 会随机删除一些神经元，以在不同批量上训练不同的神经网络架构。

在训练时，每次随机（如50%概率）忽略隐层的某些节点；这样，我们相当于随机从2^H个模型中采样选择模型；同时，由于每个网络只见过一个训练数据（每次都是随机的新网络），所以类似 bagging 的做法，这就是我为什么将它分类到「结合多种模型」中；

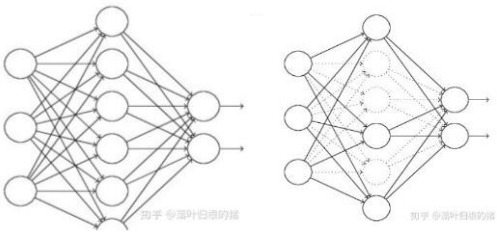


此外，而不同模型之间权值共享（共同使用这 H 个神经元的连接权值），相当于一种权值正则方法，实际效果比 L2 regularization 更好。

我们在前向传播的时候，让某个神经元的激活值以一定的概率p停止工作，这样可以使模型泛化性更强，因为它不会太依赖某些局部的特征。



3. dropout如何工作的?



输入是x输出是y，正常的流程是：我们首先把x通过网络前向传播，然后把误差反向传播以决定如何更新参数让网络进行学习。使用Dropout之后，过程变成如下：

（1）首先随机（临时）删掉网络中一半的隐藏神经元，输入输出神经元保持不变（图中虚线为部分临时被删除的神经元）

（2）然后把输入x通过修改后的网络前向传播，然后把得到的损失结果通过修改的网络反向传播。一小批训练样本执行完这个过程后，在没有被删除的神经元上按照随机梯度下降法更新对应的参数（w，b）。

（3）然后继续重复这一过程：

a. 恢复被删掉的神经元（此时被删除的神经元保持原样，而没有被删除的神经元已经有所更新）

b. 从隐藏层神经元中随机选择一个一半大小的子集临时删除掉（备份被删除神经元的参数）。

c. 对一小批训练样本，先前向传播然后反向传播损失并根据随机梯度下降法更新参数（w，b）（没有被删除的那一部分参数得到更新，删除的神经元参数保持被删除前的结果）。

不断重复这一过程。

4. 为什么dropout可以解决过拟合：

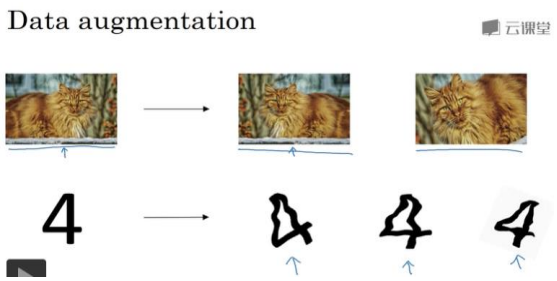
（1）取平均的作用： 先回到标准的模型即没有dropout，我们用相同的训练数据去训练5个不同的神经网络，一般会得到5个不同的结果，此时我们可以采用“5个结果取均值”或者“多数取胜的投票策略”去决定最终结果。例如3个网络判断结果为数字9,那么很有可能真正的结果就是数字9，其它两个网络给出了错误结果。这种“综合起来取平均”的策略通常可以有效防止过拟合问题。因为不同的网络可能产生不同的过拟合，取平均则有可能让一些“相反的”拟合互相抵消。dropout掉不同的隐藏神经元就类似在训练不同的网络，随机删掉一半隐藏神经元导致网络结构已经不同，整个dropout过程就相当于对很多个不同的神经网络取平均。而不同的网络产生不同的过拟合，一些互为“反向”的拟合相互抵消就可以达到整体上减少过拟合。

（2）减少神经元之间复杂的共适应关系： 因为dropout程序导致两个神经元不一定每次都在一个dropout网络中出现。这样权值的更新不再依赖于有固定关系的隐含节点的共同作用，阻止了某些特征仅仅在其它特定特征下才有效果的情况 。迫使网络去学习更加鲁棒的特征 ，这些特征在其它的神经元的随机子集中也存在。换句话说假如我们的神经网络是在做出某种预测，它不应该对一些特定的线索片段太过敏感，即使丢失特定的线索，它也应该可以从众多其它线索中学习一些共同的特征。从这个角度看dropout就有点像L1，L2正则，减少权重使得网络对丢失特定神经元连接的鲁棒性提高。

（3）**Dropout类似于性别在生物进化中的角色：** 物种为了生存往往会倾向于适应这种环境，环境突变则会导致物种难以做出及时反应，性别的出现可以繁衍出适应新环境的变种，有效的阻止过拟合，即避免环境改变时物种可能面临的灭绝。

8. Other regulizatioin methods

8.1. More data: by data augmentation (getting more data will be expensive) by: roating , cropping randomly, distortion
-->in this way, training set is a bit redundant, get less info. than brand new examples.



8.2. Early stopping
>1. defination: stopping the neural network training earlier.

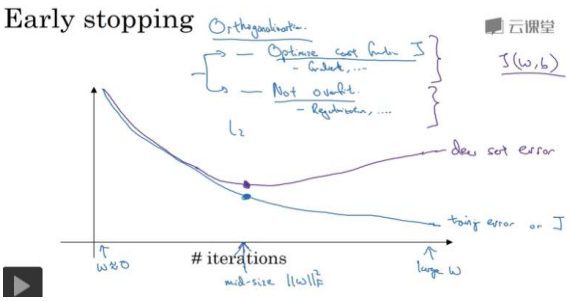
>2. Plotting:
J_train vs iteration: signal decrease;
J_dev vs iteration: decrease then increase (overfit).
early stopping: stop iteration when dev and train error seems ok.

>3.Idea:
Weight W start from very small close to 0; randomly initialize to small value, and increase to bigger value as iteration times increase.
early stopping iteration-->weight is not so big, mid-size W, similar to L2 regulization by picking a neural network with smaller norm for parameters 2, -->overfitting less.
optimzing J and prevent overfitiing is orthogonalized.

>4 disadvantage:

Machine learning optimizing Standard process: orthogonalizaton （正交化）
>>>1. Optimize cost function J: using gradient descent /momentum...: J_train go to flattern after enough iterations:
in this step , only need to focus on w and b, to make J small. no consider other things (hyperparameter, or architecture..).

>>>2. after finish J optimization, need to reduce overfitting: intro. regulization/more data, etc.
this is completely seperated task to reduce variance,
optimzing J and prevent overfitiing is orthogonalized: only work on one task at one time.



In early stopping, could not work on above two task independently: minimize J_train-bias, minimize J_dev-variance.
idea is using early stop fix two problem bias and variance at same time.--->make things to try more complicated to think about.
alternative is to use L2 , then could train neural network as long as possible. whiel disadvantage is have to try a lot of values of regularization parameter lambda, -make searching over many values of lambda more computationally expensive.

>5.Advantage: running the gradient descent process just once, get to try out values of small , mid-size and large W.
Without needing to try a lot of values of L2 regularization hyperparameter lambda.

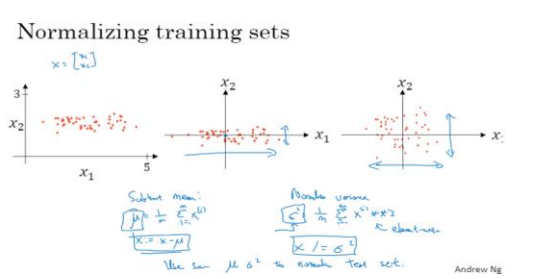
9. Normalizing training set-speed training

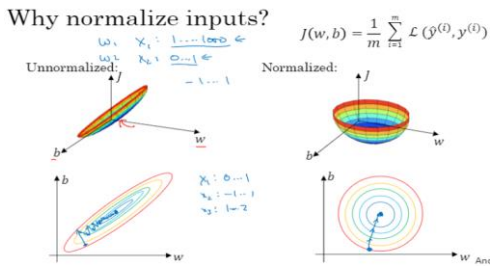
9.1 Normalize method:
normalizing training examples m: 输入的每个feature, 在所有样件上平均， 归一化，以使所有样件分布在（0,1）范围内。
1rst normalize mean and 2nd normalize variance for each feature
--> each feature have 0 mean, 1 variance over all examples:

Note: normalize dev/test by using same u and variance training set used for normalization.--->all data go through same transformation.

9.2: normalization training set -->motivation:
when feature are all on similar scales, cost function will be more round and easier to optimize.
make cost function more symmetric, --->less sensive about intialization point/start point for gradient descent and could use bigger learning rate ---> less step to find optima, algorithm run faster.: Derivative of J over each parameter w_i is similar-->gradient descent step for each parameter is similar-->less sensitive to start point and total cost move forward faster
if cost functio is not symmetric, while doing gradient descent, start point/intialization weight is important (derivative on different start point may differ much, and derivative direction not always point to optima, gradient descent step also differe much) and the tour to final optima is more tortuous.

-->when features scales from a large range, necessary to do normalization. or if features aleady in similar range (around 0-1), then do not need normalization -yet better normalize also.

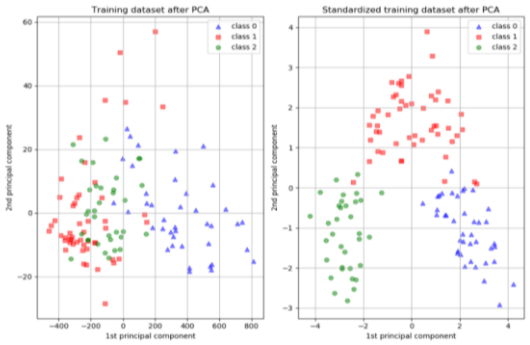




normalization

什么时候需要feature scaling?

- 涉及或隐含**距离计算**的算法，比如K-means、KNN、PCA、SVM等，一般需要feature scaling，因为
  - **zero-mean一般可以增加样本间余弦距离或者内积结果的差异**，区分力更强，假设数据集集中分布在第一象限远远的右上角，将其平移到原点处，可以想象样本间余弦距离的差异被放大了。在模版匹配中，zero-mean可以明显提高响应结果的区分度。
  - 就欧式距离而言，**增大某个特征的尺度，相当于增加了其在距离计算中的权重，如果有明确的先验知识表明某个特征很重要，那么适当增加其权重可能有正向效果，但如果没有这样的先验，或者目的就是想知道哪些特征更重要，那么就需要先feature scaling，对各维特征等而视之。**
  - 增大尺度的同时也增大了该特征维度上的方差，PCA算法倾向于关注方差较大的特征所在的坐标轴方向，其他特征可能会被忽视，因此，在PCA前做Standardization效果可能更好，如下图所示，图片来自scikit learn-Importance of Feature Scaling，



request%255Fid%2522%253A%252216287393881678027153790 6%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=162873938816780271537906&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-

- 损失函数中含有**正则项**时，一般需要feature scaling：对于线性模型$y = wx + b$而言，x的任何线性变换（平移、放缩），都可以被w和b吸收掉，理论上，不会影响模型的拟合能力。但是，如果损失函数中含有正则项，如入$||w||^2$，λ为超参数，其对w的每一个参数施加同样的惩罚，但对于某一维特征x_i而言，其scale越大，系数w_i越小，其在正则项中的比重就会变小，相当于对w_i惩罚变小，即损失函数会相对忽视那些scale增大的特征，这并不合理，所以需要feature scaling，使损失函数平等看待每一维特征。
- **梯度下降算法，需要feature scaling。**梯度下降的参数更新公式如下，

$$W(t+1) = W(t) - \eta \frac{dE(W)}{dW}$$

$E(W)$为损失函数，**收敛速度取决于：参数的初始位置到local minima的距离，以及学习率η的大小。**一维情况下，**在local minima附近**，不同学习率对梯度下降的影响如下图所示，

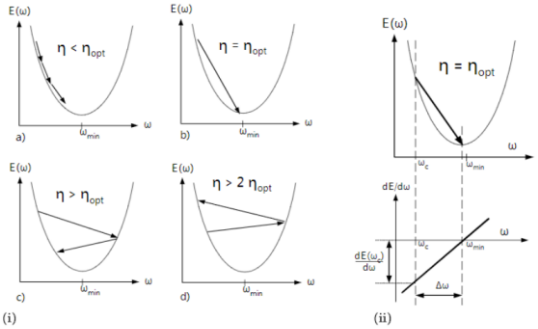


Fig. 6. Gradient descent for different learning rates.

多维情况下可以分解成多个上图，每个维度上分别下降，参数W为向量，但学习率只有1个，即所有参数维度共用同一个学习率（暂不考虑为每个维度都分配单独学习率的算法）。**收敛意味着在每个参数维度上都取得极小值，每个参数维度上的偏导数都为0，但是每个参数维度上的下降速度是不同的，为了每个维度上都能收敛，学习率应取所有维度在当前位置合适步长中最小的那个。**下面讨论feature scaling对gradient descent的作用，

多维情况下可以分解成多个上图，每个维度上分别下降，参数W为向量，但学习率只有1个，即所有参数维度共用同一个学习率（暂不考虑为每个维度都分配单独学习率的算法）。**收敛意味着在每个参数维度上都取得极小值，每个参数维度上的偏导数都为0，但是每个参数维度上的下降速度是不同的，为了每个维度上都能收敛，学习率应取所有维度在当前位置合适步长中最小的那个。**下面讨论feature scaling对gradient descent的作用，

- **zero center与参数初始化相配合，缩短初始参数位置与local minimum间的距离，加快收敛。**模型的最终参数是未知的，所以一般随机初始化，比如从0均值的均匀分布或高斯分布中采样得到，对线性模型而言，其分界面初始位置大致在原点附近，bias经常初始化为0，则分界面直接通过原点。同时，为了收敛，学习率不会很大。而每个数据集的特征分布是不一样的，如果其分布集中且距离原点较远，比如位于第一象限远远的右上角，分界面可能需要花费很多步骤才能"爬到"数据集所在的位置。所以，无论什么数据集，先平移到原点，再配合参数初始化，可以保证分界面一定会穿过数据集。此外，**outliers常分布在数据集的外围**，与分界面从外部向内挪动相比，从中心区域开始挪动可能受outliers的影响更小。
- 对于采用均方误差损失LMS的线性模型，损失函数恰为二阶，如下图所示

$$E(W) = \frac{1}{2P}\sum_{p=1}^{P}\left|d^p - \sum_i w_i x_i^p\right|^2$$

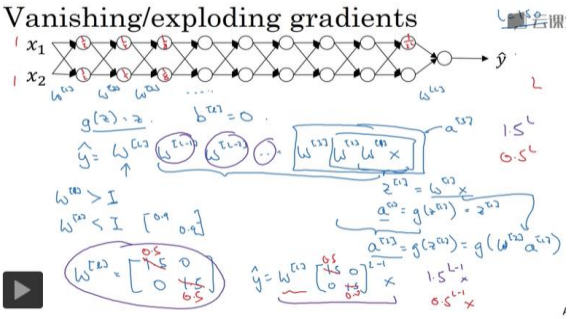
不同方向上的下降速度变化不同（二阶导不同，曲率不同），恰由输入的协方差矩阵决定，**通过scaling改变了损失函数的形状，减小不同方向上的曲率差异**，将每个维度上的下降分解来看，给定一个下降步长，如果不够小，有的维度下降的多，有的下降的少，有的还可能在上升，损失函数的整体表现可能是上升也可能是下降，就会不稳定。**scaling后不同方向上的曲率相对更接近，更容易选择到合适的学习率，使下降过程相对更稳定。**

- 另有从Hessian矩阵特征值以及condition number角度的理解，详见Lecun paper-Efficient BackProp中的Convergence of Gradient Descent一节，有清晰的数学描述，同时还介绍了白化的作用——解除特征间的线性相关性，使每个维度上的梯度下降可独立看待。
- 文章开篇的椭圆形和圆形等高线图，仅在采用均方误差的线性模型上适用，其他损失函数或更复杂的模型，如深度神经网络，**损失函数的error surface可能很复杂，并不能简单地用椭圆和圆来刻画**，所以用它来解释feature scaling对所有损失函数的梯度下降的作用，似乎过于简化，见Hinton vedio-3.2 The error surface for a linear neuron。
- 对于损失函数不是均方误差的情况，只要权重w与输入特征x间显相乘关系，损失函数对w的偏导必然含有因子x，w的梯度下降速度就会受到特征x尺度的影响。理论上为每个参数都设置上自适应的学习率，可以吸收掉x尺度的影响，但在实践中出于计算量的考虑，往往还是所有参数共用一个学习率，此时x尺度不同可能会导致不同方向上的下降速度悬殊较大，学习率不容易选择，下降过程也可能不稳定，通过scaling可对不同方向上的下降速度有所控制，使下降过程相对更稳定。
- 对于传统的神经网络，对输入做feature scaling也很重要，因为采用sigmoid等有饱和区的激活函数，如果输入分布范围很广，参数初始化时没有适配好，很容易直接陷入饱和区，导致**梯度消失**，所以，需要对输入做Standardization或映射到$[0, 1]$、$[-1, 1]$，配合精心设计的参数初始化方法，对值域进行控制。但自从有了Batch Normalization，每次线性变换改变特征分布后，都会重新进行Normalization，似乎可以不太需要对网络的输入进行feature scaling了？但习惯上还是会做feature scaling。

什么时候不需要Feature Scaling?

- 与距离计算无关的概率模型，不需要feature scaling，比如Naive Bayes；
- 与距离计算无关的基于树的模型，不需要feature scaling，比如决策树、随机森林等，树中节点的选择只关注当前特征在哪里切分对分类更好，即只在意特征内部的相对大小，而与特征间的相对大小无关。

10: Gradients vanishing /exploding



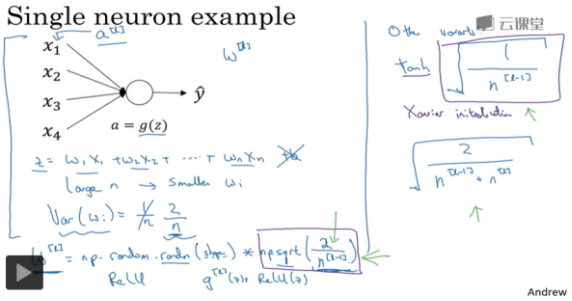
Definiation: in deep NN, while training, **derivative**/slop sometimes get either very big or very small.

E.g: activation function for all layers: g(z)=z linear --->
y= W[L]. W[L-1].....W[1].X
make W[l] all = [1.5, 0; 0, 1.5] ----->y =W[L]^L . X
in a very deep network, output y value will explode;
if w[l]= [0.5, 0; 0, 0.5] -->in a very deep network , output y will vanish

in a very deep network, if weight is little bigger than 1, output y value/activation value will explode; if weight value is a bit less than 1, output value will vanish.

same argument also applied to derivatives, exploding/vashining as function of the layer numbers.--->difficult for learning, as gradient descent step too big or too small.

11. Weight initilization for deep networks



Weight initilization: partial solution for gradient descent exploding/vanishing: does not solve it entirely, but helps a lot;
By better or more carfull choice of the random initialization for nerual network.
-->make derivative of activation function in each hidden layer, close to 1
e.g:
forward: A1=g1(W1*X), A2= g2(w2*A1),....A[L] = gL (wL* A[L-1])
backword: dA[L-1] = W[L]*dA[l]. *g'_L , **dA[L-2] = W[L-1]*dA[l-1]. *g'_(L -1)=W[L-1]*W[L]*dA[l]. *g'_L. *g'_(L -1)....**
1. output/activation of each layer will not explode, when g[l] not explode /vanish
(hidden layer activation funciton: sigmoid, tanh, -->-1<=g[l]<=1, ok for not exploding tanh not exploing only when z=wx+b: ~=1)

2. derivative will not explode or vanish only when derivative of activation function g'[l] close to 1.
(note: so more often choose hidden layer activation function Relu)

11.1 : Singal neuron example

z= w_1*x_1 +...w_m*x_n ,
in case Z is too large or too small-----> larger is n, smaller is w_i
could set weight variance:
V(w) = 1/sqrt (n) (n is the input feature of this neuron) for tanh activation functionn
V(w)= 2/sqrt (n) for Relu activation function

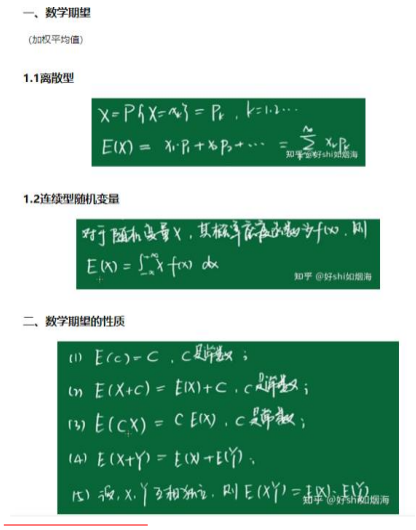
W[l] = np. random.randn(shape) * np.sqrt (1 or 2/n_l-1)

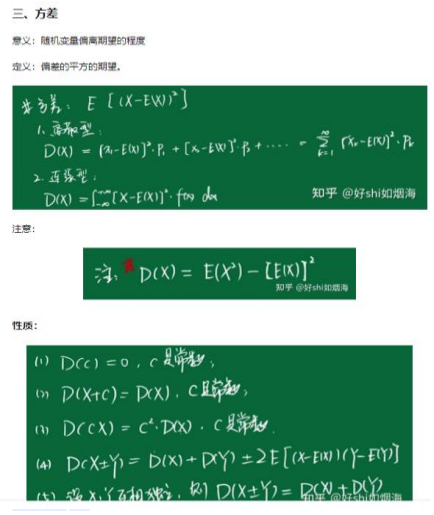
-->if input feature have 0 mean and 1 variance, then Z also take on a similar scale with above intialized w.
(note: D(z) = D(w1)*D(X_1) +.....+D(Wn)*D(X_n): in conditions Wi and X_i are independent)
does not solve but definitely helps reduce the vanishing , exploding gradients problem, as set each layer weight not too bigger than 1, and not too much less than 1.
Note: weight variance could be another hyperparameter to tune-but have low priority than other hyperameters.
W[l] = np. random.randn(shape) * np.sqrt (v(w))

E(X) =0, D(X)=1= E((X-E(X))^2) = E(X^2) - [E(X)]^2
E(Z)=E(w*x), if w, x unrelvent, -->E(Z) =E(W)*E(X)=0
D(Z)=D(W*X), ifw, x unrelevent-->D(Z) =E(W^2)
D(W) = E((W-E(W))^2)
D(X+Y) = D(X) +D(Y) if X, Y are independent.

E(X), D(X) defination:

https://www.zhihu.com/search?type=content&q=%E6%A6%82%E7%8E%87%E7%BB%9F%E8%AE%A1%20%E6%9C%9F%E6%9C%9B%20%E6%96%B9%E5%B7%AE





四、常见分布的期望与方差

分布	参数	分布律或概率密度	数学期望	方差
0-1 分布	p	$P\{x=k\}=p^k(1-p)^{1-k}$ $\{k=0,1\}$	p 知乎 @好shi如烟海	$p(1-p)$
二项分布 $B(n,p)$	n,p	$P\{x=k\}=C_n^k p^k(1-p)^{n-k}$	np 知乎 @好shi如烟海	$np(1-p)$
泊松分布 $P(\lambda)$	λ	$P\{x=k\}=\frac{\lambda^k e^{-\lambda}}{k!}$	λ 知乎 @好shi如烟海	λ
均匀分布 $U(a,b)$	a,b	$f(x)=\frac{1}{b-a},(a<x<b)$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
正态分布 $N(\mu,\sigma^2)$	μ,σ	$f(x)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	μ	σ^2
指数分布 $e(\lambda)$	λ	$f(x)=\begin{cases}\lambda e^{-\lambda x}, & x>0\\ 0, & \text{其他}\end{cases}$	$\frac{1}{\lambda}$ 知乎 @好shi如烟海	$\frac{1}{\lambda^2}$

深度学习中常见的权重初始化方法

梯度消失和梯度爆炸

反向传播的详细推导可以参见：

小糊糊：反向传播算法详解
zhuanlan.zhihu.com

从反向传播的推导公式我们知道

$$dW^{[l]}=\frac{1}{m}dZ^{[l]}\cdot A^{[l-1]^{\mathrm{T}}} \quad (1)$$

$$db^{[l]}=\frac{1}{m}np.sum(dZ^{[l]},axis=1,keepdim=True) \quad (2)$$

$$dZ^{[l]}=W^{[l+1]^{\mathrm{T}}}dZ^{[l+1]}*g^{[l]}(Z^{[l]}) \quad (3)$$

从上面式（2）我们得到了反向传播的核心递推公式。假设网络共 L 层，我们计算第 l 层权重的梯度 $dW^{[l]}$ 需要计算 $W^{[l+1]^{\mathrm{T}}}\cdot W^{[l+2]^{\mathrm{T}}}\cdot\ldots\cdot W^{[L]^{\mathrm{T}}}$ 和 $g^{[l]}(Z^{[l]})\cdot g^{[l+1]}(Z^{[l+1]})\cdot\ldots\cdot g^{[L-1]}(Z^{[L-1]})$。

可以看到梯度的大小与两个因素相关：权重矩阵和激活函数。

举个极端例子，假如权重矩阵都是一维的，且权重矩阵的值都是 t，那么：

$$W^{[l+1]^{\mathrm{T}}}\cdot W^{[l+2]^{\mathrm{T}}}\cdot\ldots\cdot W^{[L]^{\mathrm{T}}}=t^{L-l}$$

那么对于一些比较深的网络，对于一些靠近输入层的层，$L-l$ 较大。若 $|t|<1$，则 $|t^{L-l}|$ 的值接近于0，导致 $dW^{[l]}$ 和 $db^{[l]}$ 接近于0，发生梯度消失。若 $|t|>1$，则 $|t^{L-l}|$ 的值很大，导致 $dW^{[l]}$ 和 $db^{[l]}$ 很大，发生梯度爆炸。

另外从激活函数方面来看，若激活函数的导数绝对值都小于1，那么导数梯度消失；若激活函数的导数绝对值都大于1，那么导数梯度爆炸。这也是为啥Relu激活函数表现好的原因，人家梯度刚好是1。

下面我们来讨论一下深度学习中常用的参数初始化方法，主要参考Keras的官方文档。

首先我们讨论将权值矩阵初始化为常数可以吗？

Constant

```
keras.initializers.Constant(value=0)
```

将张量初始值设为一个常数的初始化器。

每层所有神经元相等，效果等价于一个神经元，这无疑极大限制了网络的能力。

特别地，全0初始化，根据式（3），反向传播时所有的梯度为0，权重不会更新，一直保持为0。

很明显，在深度学习中将权值矩阵初始化为常数是不合适的。

通过上面的讨论，全0、常数、过大、过小的权重初始化都是不好的，那什么样的初始化是好的呢？

什么样的初始化是好的？

- 因为对参数 w 的大小和正负缺乏先验知识，w 应为为随机数，且期望 $E(w)=0$；
- 在限定期望 $E(w)=0$ 后，为了防止梯度消失和梯度爆炸，参见公式（3），权重不易过大或过小，所以**要对权重的方差 $Var(w)$ 有所控制**；
- 参见公式（1），$dW^{[l]}$ 还与 $A^{l-1^{\mathrm{T}}}$ 相关，所以我们希望**不同激活层输出的方差相同**，即 $Var(a^{[l]})=Var(a^{[l-1]})$，这也就意味**不同激活层输入的方差相同**，即 $Var(z^{[l]})=Var(z^{[l-1]})$；
- 权重初始化时，权重的数值范围（方差）应考虑到前向和后向两个过程。数值太大，前向时可能陷入饱和区，反向时可能梯度爆炸；数值太小，反向时可能梯度消失。

glorot（Xavier）
论文地址：Understanding the difficulty of training deep feedforward neural networks

核心思想：正向传播时，激活值的方差保持不变；反向传播时，关于状态值的梯度的方差保持不变。

he初始化
论文地址：Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

核心思想：正向传播时，状态值的方差保持不变；反向传播时，关于激活值的梯度的方差保持不变。

glorot（Xavier）

论文地址：Understanding the difficulty of training deep feedforward neural networks

核心思想：正向传播时，激活值的方差保持不变；反向传播时，关于状态值的梯度的方差保持不变。

假设现在有一批输入数据 $X=\{x^{(1)},x^{(2)},\ldots\}$，$x^{(1)},x^{(2)},\ldots$ 是独立同分布的，且 $E(x)=0$。

假设同一层（例如第 l 层有 M 个输入节点，N 个输出节点）的权重 $w_{11}^{[l]},w_{12}^{[l]},\ldots,w_{ij}^{[l]},\ldots,w_{NM}^{[l]}$ 也是独立同分布。且期望 $E(w_{ij}^{[l]})=0$。

假设激活函数为类似于 $tanh$ 的函数，在0处的导数约等于1，例如 $g'(0)\approx 1$。

因为每层的权重 $w_{11}^{[l]},w_{12}^{[l]},\ldots,w_{ij}^{[l]},\ldots,w_{NM}^{[l]}$ 和输入 $a_1^{[l-1]},\ldots,a_m^{[l-1]},\ldots,a_M^{[l-1]}$ 相互独立，所以两者之积 $w_{ij}^{[l]}a_m^{[l-1]}$ 构成的随机变量 $z_1^{[l]},\ldots,z_n^{[l]},\ldots z_N^{[l]}$ 亦相互独立，且同分布。

因为 $E(x)=0$，$E(w_{ij}^{[l]})=0$，所以 $E(a_m^{[l]})=0$。

可以推得：$Var(w_{ij}^{[l]}a_m^{[l-1]})=Var(w_{ij}^{[l]})Var(a_m^{[l-1]})$

因为初始化时期望 $E(w)=0$，w 在0附近，而且 $E(x)=0$，所以 $g'(z_n^{[l]})\approx 1$，**$tanh$ 在初始化时期激活函数的活动范围在0附近类似于线性函数。**

$$Var(dW^{[l]})=Var(x)Var(dz^{[L]})\prod_{i=1}^{l-1}n^{[i-1]}Var(W^{[i]})\prod_{i=l+1}^{L}n^{[i]}Var(W^{[i]})$$

he初始化

论文地址：Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

核心思想：**正向传播时，状态值的方差保持不变；反向传播时，关于激活值的梯度的方差保持不变。**

Xavier假定激活函数在0附近为线性函数，这对于Relu是不成立的。**he初始化是针对Relu激活函数的初始化方法。**

因为 $W^{[l]}$ 是0均值，所以在**前向传播过程中**有：

$$Var(z^{[l]})=n^{[l-1]}Var(W^{[l]}a^{[l-1]})=n^{[l-1]}Var(W^{[l]})E((a^{[l-1]})^2)$$

其中 $(a^{[l-1]})^2$ 表示 $a^{[l-1]}$ 的平方。因为激活函数是Relu函数，所以 $a^{[l-1]}$ 不是0均值的。所以 $E((a^{[l-1]})^2)\neq Var(a^{[l-1]})$。

如果我们令 $W^{[l-1]}$ 为以0为中心的对称分布且 $b^{[l-1]}=0$，则 $z^{[l-1]}$ 也将是以0为中心的对称分布，且 $E(z^{[l-1]})=0$。

那么当激活函数为 $Relu$ 时有：$E((a^{[l-1]})^2)=\frac{1}{2}E((z^{[l-1]})^2)=\frac{1}{2}Var(z^{[l-1]})$

那么有 $Var(z^{[l]})=n^{[l-1]}Var(W^{[l]})E((a^{[l-1]})^2)=\frac{1}{2}n^{[l-1]}Var(W^{[l]})Var(z^{[l-1]})$

所以 $Var(z^{[l]})=Var(z^{[1]})\prod_{i}^{l}[\frac{1}{2}n^{[i-1]}Var(W^{[i]})]$

为了使前向传播不同激活层输出的方差相同，反向传播中不同层输入的梯度相同

即有对于任意的 i, j ，有：

$Var(a^{[i]}) = Var(a^{[j]})$

$Var(dz^{[i]}) = Var(dz^{[j]})$

解得对于同一个 $W^{[i]}$ ： $\forall i, n^{[i-1]}Var(W^{[i]}) = 1, n^{[i]}Var(W^{[i]}) = 1$

综合以上两式得 $\forall i, Var(w^{[i]}) = \frac{2}{n^{[i-1]} + n^{[i]}}$.

如果我们使用均匀分布来初始化，假设均匀分布的范围为 $[-a, a]$,则均匀分布的方差为：

$Var(uniform) = \frac{[a-(-a)]^2}{12} = \frac{a^2}{3}$.

令 $\frac{a^2}{3} = \frac{2}{n^{[i-1]} + n^{[i]}}$ 解得： $a = \sqrt{\frac{6}{n^{[i-1]} + n^{[i]}}}$

所以最终Xavier初始化均匀分布形式为 $(-\sqrt{\frac{6}{n^{[i-1]} + n^{[i]}}}, \sqrt{\frac{6}{n^{[i-1]} + n^{[i]}}})$.

若是使用正态分布，正态分布方差为：

$Var(normal) = \sigma^2$

令 $\sigma^2 = \frac{2}{n^{[i-1]} + n^{[i]}}$ 即可，

所以Xavier的正态分布形式为 $(u, \sigma^2) = (0, \frac{2}{n^{[i-1]} + n^{[i]}})$.

令前向传播过程中各层状态值的方差保持不变，有：

$\forall l, \frac{1}{2} n^{[l-1]} Var(W^{[l]}) = 1$

前向过程对应的高斯分布为 $N(0, \sqrt{\frac{2}{n^{[l-1]}}})$,均匀分布为 $U(-\sqrt{\frac{6}{n^{[l-1]}}}, \sqrt{\frac{6}{n^{[l-1]}}})$

在反向传播过程中，有：

$dz^{[l]} = f'(z^{[l]}) da^{[l]}$,因为 f' 是Relu激活函数，所以 $f'(z^{[l]})$ 为0或1的概率是相同的。

所以 $E(dz^{[l]}) = \frac{1}{2} E(da^{[l]}) = 0$. （由期望的定义可以推得）

所以 $E((dz^{[l]})^2) = Var(dz^{[l]}) = \frac{1}{2} Var(da^{[l]})$ （由方差的定义可以推得）

于是： $Var(da^{[l-1]}) = n^{[l]} Var(W^{[l]}) Var(dz^{[l]}) = \frac{1}{2} n^{[l]} Var(W^{[l]}) Var(da^{[l]})$

所以 $Var(da^{[l-1]}) = Var(da^{[l]}) \prod_{i=l}^{L} \frac{1}{2} [n^{[l]} Var(W^{[l]})]$

令前向传播过程中关于激活值的梯度的方差保持不变，有：

$\forall l, \frac{1}{2} n^{[l]} Var(W^{[l]}) = 1$

反向过程对应的高斯分布为 $N(0, \sqrt{\frac{2}{n^{[l]}}})$,均匀分布为 $U(-\sqrt{\frac{6}{n^{[l]}}}, \sqrt{\frac{6}{n^{[l]}}})$.

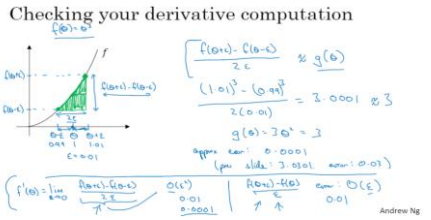
原论文中指出不管是按照前向过程还是反向传播过程得到的初始化方法初始化参数对结果影响不大。两个过程得到的结果只是一个是使用输入层单元个数，一个是使用输出层单元个数。

12. Numerical approximation of gradients_two side differance

for checking if gradient descent is working properly. Use two side differance.

Gradient descent estiamtion:

>1. Two side estimation: error ~ε^2 (while ε close to 0)
while 1 side estimation: error ~ε



13. Gradient check:
to debug if implementing gradient descent correctly.

Process:
>1. Take all parameters w[1], b[1], ...w[L], b[L] and reshape into a big vector θ.
--> J(w, b) = J (θ)= J(θ1,θ2, θ3 ...)

>2. Take d w[1], db[1], ...dw[L], db[L] and reshape into a big vector dθ.

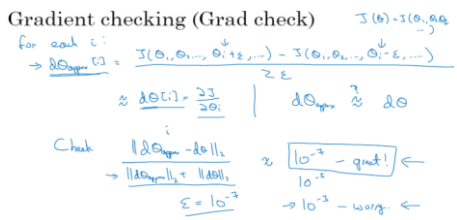
>3. for evey element in vector θ
>>>3.1 Compute estimated dθ_i by two side differance:
dθ_i _app= { J(θ1,θ2, θ_i+ε ...) - J(θ1,θ2, θ_i-ε ...) } /2/ε
>>>3.2 Compute derivrative dθ_i by formula:
dθ_i = dJ/dθ_i

-->after go thourgh all element in vector θ, get two vectors: dθ_app, dθ_i

>4. check similarity of two vectors: dθ_app, dθ_i by vector distance:
||dθ_app- dθ_i ||2 / (||dθ_app-||2 + || dθ_i ||2

Euclidean distance: ||dθ_app- dθ_i ||2
sum of element in distance vector and then take squart root.
Euclidean length: ||dθ||2. used here just incase any of these two vectors is too small or large, turn result into a ration
could set: ε=10^-7,
reslt : 10^-7 , may ok;
result :10^-5: concerned, double check components of vector, make sure none of the components are too large.
result 10^-3: worry there is a bug.--->
look at each component of thest two vectors, find the component of dθ_app that is very different from dθ_i , then to track down derivative computation might be incorrect.

Note:
Implement foreprop, backprop, and do grad check, and do debug if need.



14. Grad check tips:

>1. First, don't use grad check in training, only to debug.

computing dθ_i_approx , for all the values of i, this is a very slow computation.
So to implement gradient descent, you'd use backprop to compute dθ and just use backprop to compute the derivative. And it's only when you're debugging that you would compute this to make sure it's close to d θ. But once you've done that, then you would turn off the grad check, and don't run this during every iteration of gradient descent, because that's just much too slow.

>2. Second, if an algorithm fails grad check, look at the components, look at the individual components, and try to identify the bug.

Remember, different components of θ_i correspond to different components of b[l] and w[l].

if dθ_i_approx is very far from dθ, look at the different values of i to see which are the values of dθ_i_approx that are really very different than the values of dθ.
e.g: if find that the values of dθ_approx or dθ_, they're very far off, all correspond to db[l] for some layer , but the components for dw[l] are quite close, then maybe you find that the bug is in how you're computing db, the derivative with respect to parameters b.
This doesn't always let you identify the bug right away, but sometimes it helps you give you some guesses about where to track down the bug.

>3. Rember regularization:
when doing grad check, remember your regularization term if you're using regularization.
(note: debugging is used in training ; while regularization also only used in training to penalt weights.)

J(θ) = original cost +regularization (w)
dθ: gradient of J over θ-consider regularization term in derivative computation

grad check purpose is to check cost J derivative computation during gradient descent-minimizing cost and updating parameter.-->regularization need to involved in grad check if cost J have regularization, to check if optimzing parameter derivative computation ok or not.

>4. grad check doesn't work with dropout

as in every iteration, dropout is randomly eliminating different subsets of the hidden units. There isn't an easy to compute cost function J that dropout is doing gradient descent on.

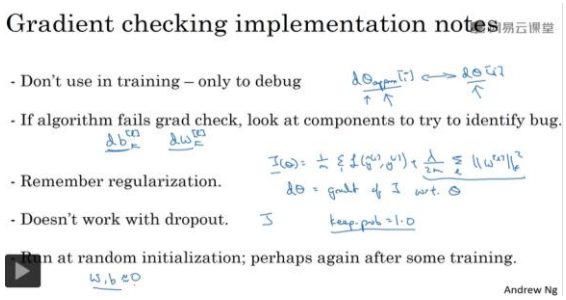
It turns out that dropout can be viewed as optimizing some cost function J, but it's cost function J defined by summing over all exponentially large subsets of nodes they could eliminate in any iteration. So the cost function J is very difficult to compute, and you're just sampling the cost function every time you eliminate different random subsets in those we use dropout. So it's difficult to use grad check to double check your computation with dropouts.

So implement grad check without dropout, can set keep-prob and dropout to be equal to 1.0. And then turn on dropout and hope that my implementation of dropout was correct.

There are some other things could do: like fix the pattern of nodes dropped and verify that grad check for that pattern of algorithm is correct.
but in practice do not usually do that. So my recommendation is turn off dropout, use grad check to double check that your algorithm is at least correct without dropout, and then turn on dropout.

>5. Run at beginning (at ranodm initialization); perhaps run grad check again after some training /iterations.
This is a subtlety. It is not impossible, rarely happens, but it's not impossible that your implementation of gradient descent is correct when w and b are close to 0, so at random initialization. But it gets more inaccurate when w and b become large.? ?

So one thing you could do, I don't do this very often, but one thing you could do is:
>>>1. run grad check at random initialization
>>>2. train the network for a while so that w and b have some time to wander away from 0, from your small random initial values.
>>>3. run grad check again after you've trained for some number of iterations.



This week summary:
Learned about how to set up your train, dev, and test sets, how to analyze bias and variance and what things to do if you have high bias versus high variance versus maybe high bias and high variance.
You also saw how to apply different forms of regularization, like L2 regularization and dropout on your neural network.
So some tricks for speeding up the training of your neural network. And then finally, gradient checking.

1. Mini-batch gradient desent

applying machine learning is a highly empirical process, is a highly iterative process. In which you just had to train a lot of models to find one that works really well.

So, it really helps to really train models quickly. One thing that makes it more difficult is that Deep Learning tends to work best in the regime of big data. We are able to train neural networks on a huge data set and training on a large data set is just slow. So, what you find is that having fast optimization algorithms, having good optimization algorithms can really speed up the efficiency of you and your team.

Vectorization allows to efficiently compute on m examples. But when m is too large, it's too slow.
Only do one gradient step after computing all examples .

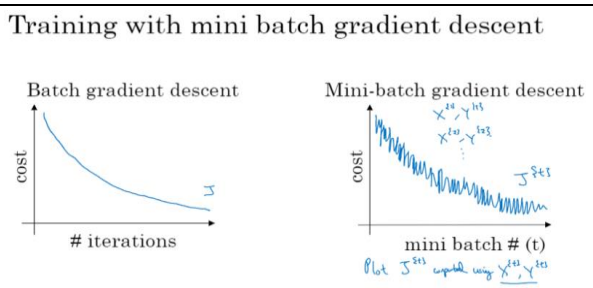
Mini-batch one epoch:
one epoch: a single pass through training set:

1.0: Minibach process:
Run one epoch:
>1. for each min-batch examples, run forward and backward, do one iteration, updating paramter;
>2. go though all mini-batches, -->algorithm do multiple times(= training data size m / mini-batch size) iterations, updated parameter multi times;
Run multi epochs:
>3. repeate a certain number epoch.

1.1: Cost fucntion:

J_train:
for batch gradient descent, J_train decrease signally;
While for mini-batch, J_train decrease with noise:
Noice due to : J_train = cost funstion with parameters updated in last iteration using last batch examples, computed loss on new batch examples- due to different batch examples used , J_train do not gurantee decrease very iteration.
there are harder mini-batch (J_train big), and easy mini-batch (J_train small)

(note: J_train decrease signally on conditions: learning rate is small engough.
input featurea are normalized-->algorithm/cost function treat each input feature equally-->w similar, dw_i similar also
-->gradient step : learning rate* dw_i (same learning rate for all feature weight derivatives) also similar-and easy find one learin rate fit all all feature weight derivatives/step-->cost function decrease.
in theory, if have sepearte, reasonable learnin rate for each (input) feature weight, could make sure cost fucntion decrease. -->using one learnig rate for all features, in case overshooting for large feature derivatives-normalize all features to make their weight derivatives similar.
mini-batch: cost funciton noise:
all training set feature normalize first-->feature weight derivative: average on mini-batch-->gradient step, update feature weight, update cost: cost function on this mini-batch examples decrease.
yet with these weight, cost on another unfitttted mini-batch: cost may bigger than befor updating on previous mini-batch: weight derivative, average on prevous mini-batch example, may overshoting on cost of next mini-batch.)

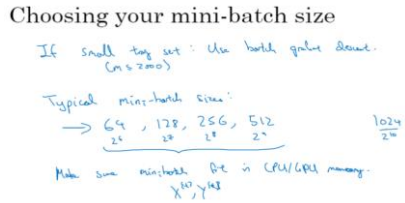


1.2 Mini-batch size:

>1. size = m, batch gradient descent:
gradient descent: take relatively low noise, large steps, keep marching to global optima-blue curve.

>2. size =1, stochastic gradient decsent: every example is its own mini-batch.
Take gradient descent on each one exmaple.
gradient descent: every iteration, take gradient descent on one signle example, so most of the time hit towards the global minimum, , but sometims hit in the wrong direction
(note: gradient decent direction based on one example, could not fit all the rest examples)

-->stochastic gradient descent can be extremely noisy, on average take good direction, but sometimes head in the wrong direction as well.
-->stochastic gradient descent won't ever converge, always just kind of oscillate and wander around the region of the minimum, but won't ever head to the minimum and stay there.



>3. size = 1-m: mini-batch:
if batch: size =m, processionog a huge training set on every iteration, takes too long per itration.
if m is small, batch gradient descent works fine.
if size=1, stochastic gradient descent, make process just after processing 1 example, and noisiness can be reduced by just using a smaller learning rate (but noise will not be eliminated as long as batch size < m?) .
but huge disadvante of stochastic gradient descent: loose almost all your speed up from vectorization. processing a single training example at a time, is very inefficient.

mini-batch is betwtten, size not too big/small --->gives the fastest learning:
do get a lot of vectorization, faster than processing the examples one at a time;
could also make process without needing to wait till process eniter training set.

mini-batch gradient descent:
>>>1. not guaranteed to always head toward the minimum, but tends to head more consistently in direction of the minimum than stochastic gradient descent.
>>>2. Does not always exactly converge or oscillate in a very small region. -can always reduce the learning rate slowly.

Summarize:

>1. if small training set, just use batch gradient descent; m<2000
>2. big training set, take mini-batch size: 64, 128, ...512 (2^9)
consider the way computer memory is laid out and accessed, sometimes could run faster if mini-batch size is a power of 2
>3, make sure all mini-batch fit in CPU/GPU memory (??). otherwise algorithm performace suddenly falls of a cliff, much worse.

>4. mini-batch size is actually hyperparameter, need to figure out which one is most sufficient of reducing the cost funtion J: try a different values and then pick one makes gradient descent optimization algorithm as efficient as possible.

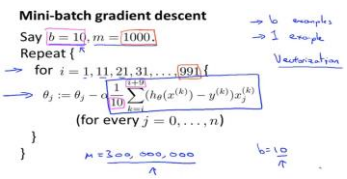
(note: mini-batch only better than stochastic when well vectorized, how to choose min-batch size in coding:
try different size and see runing time, perforamance vs iteration on dev set?)

1. Mini-batch gradient descent

>1. Batch gradient descen: Use all m examples in each iteration;
>2. Stochastic gradient descent: Use a single example in each iteration;
>3. Mini-batch gradient descent: Use b examples in each iteration

Parameter: b- "mini batch size" , type choice: 2-100 (common 10)

1.1 Mini-batch idea: Somewhat in-between Batch gradient descent and Stochastic gradient descent.
Rather than using one example at a time or m examples at a time we will use b examples at a time.



1.2 Mini-batch algorithm

e.g: b=10, m=1000.

>1. Inner loop: -one epoch (go through all training example one time)
>>> 1. one iteration: graddient descent on b-10 examples
>>> 2 go continualy 100 (m/b) itertions, each iterations with next, different 10 examples
-->go thourh all examples 1 time, 100 (m/b) itertions with b example for each iteration.
each iteration use different b examples.

>2. outer loop: repeate inner loop: -->go through again all the examples.

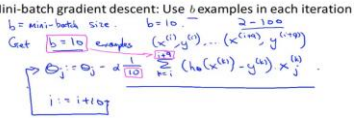
Mini-batch faster than batch gradient descent:

Mini-batch gradient descent

→ Batch gradient descent: Use all m examples in each iteration

→ Stochastic gradient descent: Use 1 example in each iteration

Mini-batch gradient descent: Use b examples in each iteration



1.3: Comparision: Mini-bacth vs Batch, stochasic,

>1. Mini-bacth vs Batch
Mini-batch faster :
(Each iteration) start making progress in modifying the parameters after looking at just b examples rather than needing to wait till have scan through every single training example.

>2. **Mini-bacth vs Stochastic gradient descent**
Mini-batch outperform stochastic : only if have a good vectorized implementation

In each iteration, sum over b examples (for the derivation of J over paramter θ) in a more vectorized way will allow to partially parallelize computation over the b examples.
using appropriate vectorization to compute the derivatives, can sometimes partially use the good numerical algebra libraries and parallelize gradient computations over the b examples.
Whereas with Stochastic gradient descent , just looking at one example at a time is less efficient to parallelize over.

>3, Disadvantage of mini-batch:
Have one more parameter to faddle with: mini-batch size b. which may therefore take time.
But if have a good vectorized implementation this can sometimes run even faster that Stochastic gradient descent.

Summary:
Mini-batch gradient descent algorithm in some sense does something that's somewhat in between what Stochastic gradient descent and batch gradient descent does.
And if choose a reasonable value of b (usually use 10, , anywhere from say 2 to 100 would be reasonably common), and use a good vectorized implementation, sometimes it can be faster than both Stochastic gradient descent and faster than Batch gradient descent.

2. Exponentially weighted average:

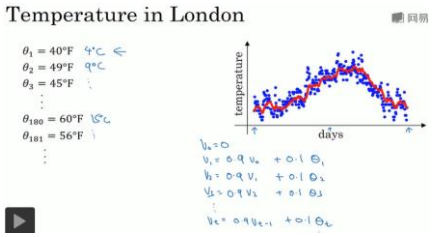
Algorithm faster than mini-batch:
2.1 e.g: temp. trend:

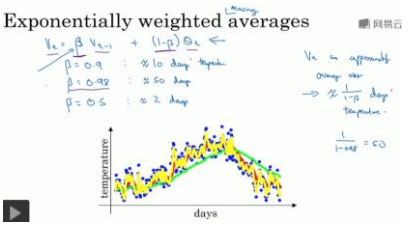
>1. ploting temp of eveyday. vs days ,--> a littlbe big noisy
>2. if want to compute the trends the local average / moving average / exponentailly weighted average of temp,
v0=0;
v1=0.9* v0 + 0.1 * t1;
v2=0.9* v1 + 0.1*t2
.....
-->Vt =β*V_t-1 + (1-β)* T_t: an approximately average over 1/ (1-β) days' temp.
ploting is more smoother as now is averaging over more days of temp.

β=0.5, about 2days average temp.- yellow line
β=0.9, about 10 days average temp.- red line
β=0.98, about 50 days average temp.- green line

---->β is bigger (weighting for previous data)
>>>1. ploting cure ship to further right, as now averaging over a much larger window of temp.
>>>2. by average over large window, this exponentially weighted average formula, adapts more slowly when the temp. changes.
giving lot of weight to the previous days' temp, much smaller weight to the value of right now.

find β works best: balancing sensitive to right now value and cover previous data.





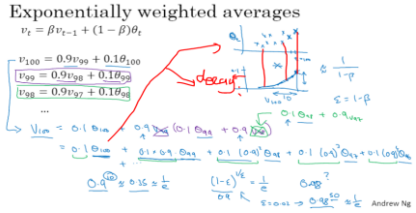
4. Understanding exponential weighted average

V_t = β * V_t -1 + (1-β) * T_t
= (1- β) *T_t + (1- β) * **β^(t-1)** *T_1 + ... (1- β) *β^(t-I) *T_i + ...+(1- β)* **β^t** * T_0

4.1: Plotting:
>>1. T_t vs time/day
vertical axis:temp of each day: T_0, ...T_t,
horizental axis: days: 1, , ...t, ...

>>2. Exponentially decaying function: [(1- β)* **β^t**, (1- β)* β^(t-1)....., (1- β)]
vertical axis: range: [(1- β)* β^t, (1- β)],
-weight for each day data, sum together to get current day t average data.
Note: decay fractor of previous i day: β^(i)
horizental axis: days: 1, , ...t, ...

>>3. take the element wise product between above two functions and sum it up:
take daily temp and mulitpy with corresponding exponentailly decaying function/weight, and then summing it up.



4.2: how many previous data to average: 1/ (1-β)

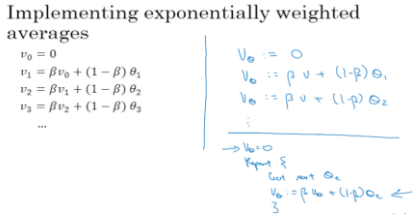
1/e = 0.35 = β^ (1/(1-β) = (1-β)^ (1/(β)

β=0.9, -->1/ (1-β)=0.9^10
-->take about 10 days, to decay from now day value weight (1- β) to around 1/3 .
Note: decay fractor of previous i day: β^(i)
--> only take last i days , whose decay fractor is β^i, take i make β^i = 1/e~=1/3, -->get i ~= 1/(1-β)
when i > 1/(1-β), decay <1/e, 30%, then do not consider that days data contribute to sum.
as if is computing an exponentially weighted average that focus on just the last 10 (1/1-β)days. as for more previous days, their weight decasy to more than 1/3 of the weight of the current day.

How to set β, balancing weight put on current day data.

Note: sum of each previous i-th day's weight:(1- β)* β ^i
sum = (1- β) * (1+β+β^2+...β^i+...+β^(t-1)) = (1- β) * (1-β^t)/(1-β) = 1-β^t-->~=1, when t is large enough.

for t days, V(t) = = (1- β) *T_t + (1- β) * β^1) *T_t-1 + ... (1- β) *β^(I) *T_(t-I) + ...+(1- β)* β^(t-1) * T_1 + β^t * V(0)
all days T_I weight sum= (1- β) (1+β+β^2+...+β^(t-1))=1-β^t
V_0 weight= β^t
total weight sum = 1



4.3: Implementing exponentially weighted averages:

advantage: takes very little memory, only one row number in compute memory:
V_θ =0
repeat:
get next θ_t
V_θ := β*V_θ + (1-β)*θ_t

Summarize:
Exponentially weighted average is not the most accurate way to compute an average, while most accurate way is to store all past 10/50 days data and average them-huge storage and computation cost.
while using exmponentially weighted average, only need to store one value V_θ , storage and computation effciently, and only need one row coding.

5. Bias correction in exponentially weighted average:

V_t = β * V_t -1 + (1-β) * T_t
= (1- β) *T_t + (1- β) * β^1 *T_(t-1) + ... (1- β) *β^(t-i)*T_i + ...+(1- β)* β^(t-1) * T_1 + β^t * V_0

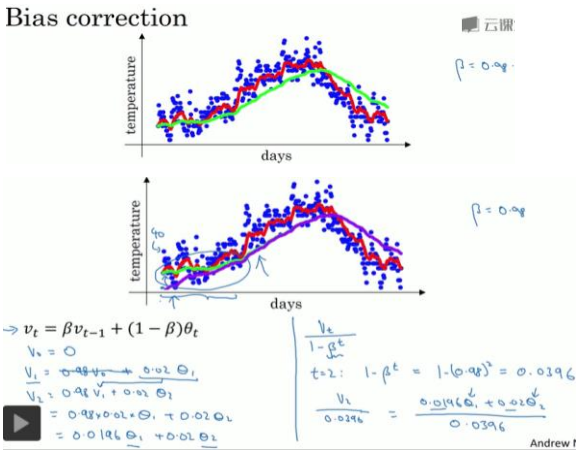
Bias in the inital phase : caused by v_0=0---> (should be green line, while is purple line with bias in initial phase)
v_1 = (1-β) * T_1
v_2 = β* v_1 + (1-β)*T_2=β(1-β) * T_1 + (1-β) * T_2..
no term of ' β^t * V_0', which influence V_i when i is small (β^t is big).

Bias correction:

V_t := 1/ (1-β^t)
>1. adding bias correction: intial phase change from purple line to green line.
>2. as t becomes bigger, β^t * V_0 nearly zeor, no bias. -purple line no bias in late phase
>3. as t becomes bigger, bias correction factor also close to 1, no bias correction function-green line aligned with purple line.

Note: bias correction use 1/ (1-β^t)--->sum of each day's weight =1
all days T_I weight sum= (1- β) (1+β+β^2+...+β^(t-1))=1-β^t, using correction -->sum =1

for t days, V(t) = = (1- β) *T_t + (1- β) * β^(1) *T_(t-1) + ... (1- β) *β^(l) *T_(t-l) + ...+(1- β)* β^(t-1) * T_1+ β^t*v(0)
all days T_I weight sum= (1- β) (1+β+β^2+...+β^(t-1))=1-β^t
V_0 weight= β^t
total weight sum = 1



Summarize:

In machine learning, for most implementations of the exponentially weighted average, people don't often bother to implement bias corrections , because most people would rather just wait that initial period and have a slightly more biased assessment , and then go from there.
But if concerned about the bias during this initial phase, while your exponentially weighted moving average is still warming up, then bias correction can help you get a better estimate early on.

6. Gradient descent with momentum
compute an exponentially weighted average of gradients.
6.1 Gradient descent problem:

During gradient descent, may have up and dow oscillation,---->
>1. slows down gradient descent, and prevents from using a much larger learning rate-which may end up overshooting and liverging;
>2. Oscillation could reduce by small learning rate

>3. another view: want learning a bit slower in oscillation -e.g vertial direction (less oscillation), but faster learning in no oscillation -e.g horizental direction: move fast from left to minimum.-right ex.g

6.2: Momentum in gradient descent:

>1. Principle: using a moving average of dw, db
>2. Function-Intuition: smooth out the step of gradient descent : oscillation reduced almost to 0 as averaged over positive and negatives (in the vertical direction)-average will close to 0;
while in the horizental diection, all the derivatives are pointing to the right direction, so averge in this direction will still be big.
(note: oscillation: due to weight derivative change direction <--learning rate/gradient step too big for this weight/direction;
-->by using moving average of weight derivative: will even out the direction of weight derivative: the derivative value pe se may no much change, but direction: +/- more stable: do not change so frequently-->show as oscillation)

-->gradient oscillate much less and moving quickly in horizental direction.
(optimizing in horizental direction-this parameter updating no change, optimizing in oscillaito directioin-this oscillating parameter derivative-faster)

>3. Formula:

V_dw = β*V_dw + (1-β)*dw
V_db = β*V_db + (1-β)*db

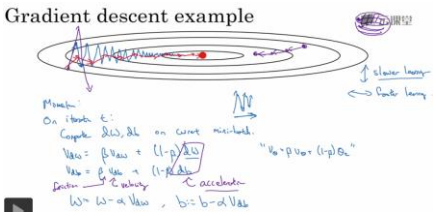
W:= - a * V_dw
b:= - a * V_db

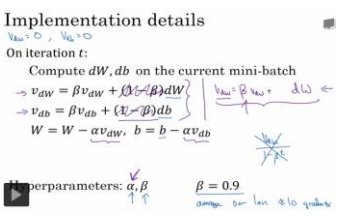
oscillation root cause: gradient step 'a*dw' too big
1. value |dw*a| is big-->updated paramter w' and previous parameter w will be on two sides of optima parameter w0-->dw' * dw <0: negative and positive direction; -->derivative direction changed
momentum: even out derivative dirction dw /db (no change on learning rate, which used for all parameters);
RTMS: reduce dw value :

(note 1: momentum: using expoentially weighted average of gradients over iterations.-parameter derivative direction even out: positive and negative (-dw:是向量， 有方向,momentum 主要改变方向，顺带会改变其大小)
for the osciallation direction-specific paramter derivation: gradient step |a*dw|too big, update parameter's derivative direction positive and negative changed from -dw to -dw':
Exponential weighted derivatives even direction-positive and negative close to 0):: positive and negative direction in past and current iteration will sum together, even out direction, very close to 0-also make |dw| close to 0: to keep dw always one direction postive or negative while |dw| value is small. (now learning on this derivative-parameter will be quite slow, how to deal with this??-->not slow,even out derivative direction and value, -->with step dw*a , march torwards to optima; now the gradient step dw*a: is a better step, previous too big to cause overshooting, even dw while no change learing rate, to actually make a better'gradient step' now for this osciallation weight/direction)
for no oscillaiton direction: parameter derivation average nearly no change.)

(note2: exponentially weighted average of gradients: not use on mini-batch examples/batch examples within one iteration: as need to consider all examples influence on parameter dervatives of one step, not few-1/β batch /iterations examples.) -->could use on mini-batch
(J_train decrease with noise due to different mini batch examples.--different big engough not able to use expoentailly weighted average of gradient in different batch??)

note 3: oscillation in some direction-specfic parameter derivative: during iterations:
all oscillations actually could be soved by reducing learning rate to small enough-**perfect solution is seperate, unique learning rate for each parameter, in every iteration.**
soluton: seperate learning rate for each parmaeter-->could be replaced by using same learning rate, with normalized input feature.
solution: unique learning rate for each iteration-->could be replaced by learning rate decay during iterations.
-->oscillation should be improved by all layer input featuring scalling & learning rate decay during iterations ?-->then no need of momentum)





6.4: Implementation details:
Initialize
V_dw=0; V_db=0

repeate iterations: for one iteration:
>1. on iteration t, compute parameter derivatives dw, db on mini-batch examples;

>2. update parameter derivatives:
V_dw = β*V_dw + (1-β)*dw
V_db = β*V_db + (1-β)*db

>3. Update parameter
W:= - a * V_dw
b:= - a * V_db

Hyperparameter: a, β
β: common take 0.9 -pretty robust value, could try different values.

Many updated formula in paper:
V_dw = β*V_dw + *dw
-->V_dw scaled by 1/(1- β),
while updating parameter, learning rate needs to change by a corresponding value 1/(1-β)

Note: recommend offical formula V_dw = β*V_dw + (1-β)*dw.
otherwise need to tune not only β, learning rate have to tune correspondingly.

6.3 Momentus intuition:

could take gradient descent as: ball rolling down from bowl-cost function.
dw, db : ball acceleration
V_dw, V_db: velocity-plays a role of fraction, prevens ball from speeding up without limit.

So rather than previous steps taking every single step independently, now the ball can roll downhill and gian momentum by accelerating down.

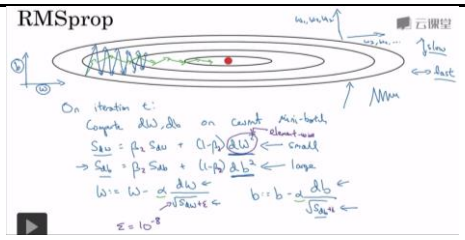
7. RMSProp: root mean square prop

>1. Idea: exponentially weighted average of derivative squares.- damping out ocillation by slowing down paramter update, parameters that caused these ocillation

>2. e.g: right picl: gradient ocilliate in vertial direction and moving in horizontal direction.
Take parameter b as vertical direction, and w as horizental ;
on one iteration t: compute dw, db on current mini-batch.

S_dw = β * S_dw + (1-β) * dw ^2 (element-wise squaring operation)
S_db = β * S_db + (1-β) * db ^2

W:= - a * dw/ sqrt (s_dw)
b:= - a * db/ sqrt (s_db)



-->if ocillate in paramter w/b direction is big, -->S_dw/ s_db is large, slow down paramter w/b update; reduce ocillation.
(note: ocillatioin: due to gradient step a* dw too big, -->
>>1. ocillate: derivatve direction change
>>2. ocillation magnitude: big if value|a* dw| big, otherwise small ocillation
RMSP: reduce big ocillation only: if was small ocillaiton, may make it a little bigger...but no care small ocillation)
)

-->if origianl gradient descent has no ocillate in parameter w/b, dw/db is small--->S_dw/S_db is smaller, faster move in this direction than direction have ocillation.
(faster than direction have ocillation: with RMSP, gradient step a* dw/db is bigger in direciton |dw/db| small-->faster move in non-ocillation direction than ocillation direction
possible be slower than without RMSP in non-ocillation direction-if |dw|>1)

Note:
>1. here take b, w as vertical and horizontal direction, in practice w, b is high dimenional, so the ocillation may happen on parameter w_i, b_i.
but in dimension getting these oscillations, will end up computing a larger sum: S_dw_i/ S_db_i, so end up damping out in which there are these oscillations.

>2. for ensuring numerical stability, in case s_dw/s_db too small, -->divide sqrt (s_dw + ε)
could set ε = 10^-8

(note: root cause for oscillation: parameter updating with gradient decsent : w= w-a * dw
gradient step : a*dw
1. oscillation happens when: w' = w-a*dw and w are on the two side of optima parameter w0-over shooting
<----gradient step a*dw is too big.
2. there are parameter gradient descent step ok: a*dw_i: ok, no overshooting for parameter w_i
while learnign rate a is same for all parameters-->
for oscillation direction: paramter derivative dw_i is big;
for no oscillation direction: parameter derivative dw_t is small;

for solving oscillation, reduce value a*dw while keep a no change.
-->1. oscillation direction-parameter: fractor 1/|dw_i|: <1-->gradient step reduce; help improve oscillation
-->2. no oscillationd irection-parameter: fractor 1/|dw_i|: may >1-->gradient step increase;- ->while turn to be oscillation?? this will be even out in next iteration.

with RMSProp, -->
>1.gradient descent end up ocillate less in parameter that cause ocillation (e.g vertical) and keep going in no occilation (e.g horizontal) direction.
>2. could use larger learning rate, fast learning.
(?faster learning on no-oscillation direction?
as increase leranign rate will make original oscillation direction more worse
-->dw kind of 'normalized' |dw| ~=1
need to find learning rate a, make gradient decent step a* dw: fit well for loss function with all parameters direction
)

8. Adam optimization algorithm

Adam: Adaptive Moment Estimation

RMSprop and Momentum, work well across a wide range of deep learning architectures.

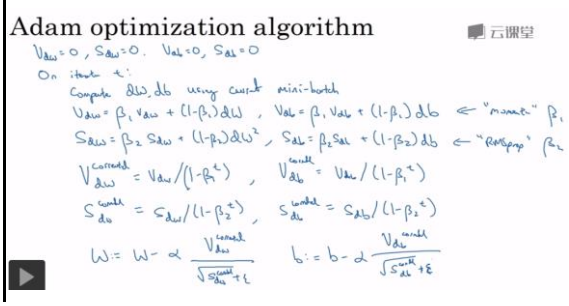
8.1 Principle: take Momentum and RMSprop together.

8.2 Formula:

>1. initialization: V_dw=0, V_db=0, S_dw=0, S_db=0
>2. for each iteration on mini-batch

V_dw = β1*V_dw + (1-β1)*dw
V_db = β1*V_db + (1-β1)*db

S_dw = β2* S_dw + (1-β2) * dw ^2 (element-wise squaring operation)
S_db = β2 * S_db + (1-β2) * db ^2



>3. Bias correction: on both momentum and RMSprop

V_dw_ correct = V_dw / (1-β1^t) ; V_db_ correct = V_db / (1-β1^t)

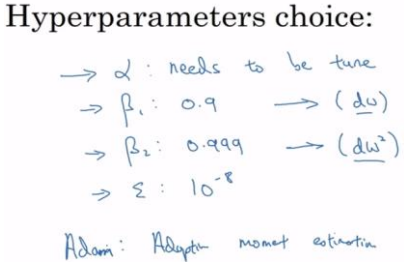
S_dw_ correct = S_dW / (1-β2^t) ; S_db_ correct = V_db/ (1-β2^t)

> 4. Parameter update
W:= - a * V_dw/ sqrt (S_dw_correct +ε)
b:= - a * V_db / sqrt (S_db_correct +ε)

8.3. Hyperparameter in Adam:

>1. a: learning rate , need to be tune.

>2. β1 , 0.9 : moving averaging; default
(note: consider previous 10 iterations dw' direction)
>2. β1 , 0.999 : RMSprop; default
(note: consider previous 1000 iterations |dw|^2-->gradient step|a*dw| much more stable-cons value, less sensitve to current dw status, why?? graident step should be smaller when reaching optima)
>4. ε: 10^-8; default. Does not matter much.



Adam: commonly used learning algorithm, proven to be very effective for many different neural networks.

>2. for each iteration on mini-batch

V_dw = β1*V_dw + (1-β1)*dw
V_db = β1*V_db + (1-β1)*db

S_dw = β2* S_dw + (1-β2) * dw ^2 (element-wise squaring operation)
S_db = β2 * S_db + (1-β2) * db ^2

Note:
momentum: reduce gradient descent ocillation by exponentially weighted average over derivative, of the past 1/ (1-β1) iterations.--->reduce dw/db oscillation , by averaging the posive and negative of dw_i/db_i that caused these ocillations (may change derivative direction, and slight 'scalling' effect by multiplying weight) .
RMSprop: reduce gradient descent ocillation by exponentiall weighted average over square derivative, of the past 1/ (1-β2) iterations.
--->reduce dw/db oscillation , by scalling the posive and negative of dw_i/db_i that caused these ocillations-(but do not change derivative direction-positive or negative dirction), using factor 1/ sqrt (s_dw_i/ db_i): make derivation caused oscillation smaller by scalling based on averaged past iteration derivation length.

Adam: for ocillations caused by certain paramters: average these positive and negative parameter derivative and then scalling by factor 1/ sqrt (s_dw_i/ db_i)

(note: for big ocillations:
>> RMSP: -->by scalling make it ocillate less
>> Momentum: -->by averaging derivative direction, make this small ocillations to nearly no occilation
>>RMSP: by scalling: make no-occilation direction, by small derivative bigger-->graident step bigger in non-occillatio direction
-->finnaly make all weights : gradient decent march torward non-occliate and step bigger.)

---
e.g: β1=β2=0.5, dw has ocsillation. weight for i1, i2, i3 value: (1-β1)*β1^2, (1-β1)*β1, (1-β1)
dw (iterative 1)=-5, dw_i2=7, dw_i3=- 4
-->V_dw_i3 = 7/8: derivation change from negative to positve, direction change
-->S_dw_i3 ~=25, RMSprop only-->dw_i3:=dw_13/5= -0.8: derivation direction no change, yet size becomes much less
-->V_dw +s_dw together -->v_dw_i3 /sqrt (s_dw_i3)= 7/8 /5 ~=0.2: derivative direction change, and size scalling to less.

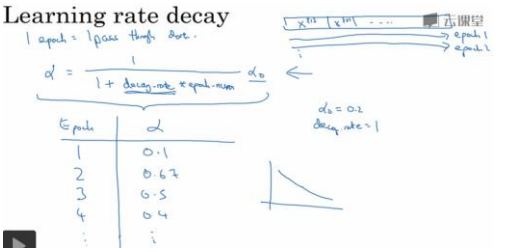
db (iterative 1)= 3, db_i2=5, db_i3=4; db no oscillation
-->V_db_i3 = 19/8: direction no change
-->S_db_i3 =67/8, RMSprop only-->dw_i3:=dw_13/sqrt (s_db_i3) ~= 4/3: derivation direction no change, yet size becomes much less
-->V_dw +s_dw together -->v_dw_i3 /sqrt (s_dw_i3) ~=19/24: derivative direction change, and size scalling to less.

9. Learning rate decay

9.1 Gradient descent on mini-batch in each iteration:
gradient descrease with noise in each mini-batch, while learning rate is fixed value, algorithm end up wandering around and never converge.
(note: this noise will help algorithm go out of local optima/-saddle point. ?)

Reduce learning rate in the late phase, will make algorithm end up in a tigher region around this minimum.

Intuition: in the initial phase, could afford to take much bigger steps-big learning rate, but as learning approaches converges, then having a slower learning rate allows to take smaller steps.



9.2 Learning rate decap implementation:

1 epoch: 1 pass through training set.
Leaning rate: a = 1/ (1+ decay rate * epoch number) * a_0

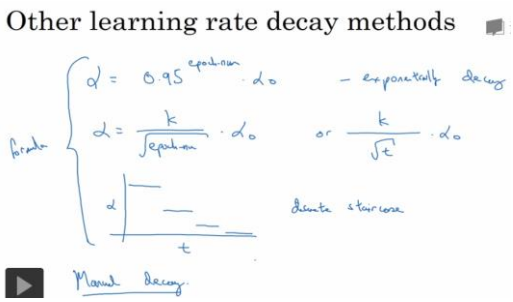
hyperameter:
>1. decay rate, need to tune
>2, a_0: need to tune

9.3 other decay method;

exponentially decay:
a = 0.95^epoch * a_0;

a= K(constant) / sqrt (epoch-number) * a_0; or a = k (constant) / sqrt (mini-batch number t) * a_0

or discrete learning rate: some steps have some learning rate.



Manully decay:
change learning rate manully after algorithm run some time: tunning learning rate by hand, hour by hour, or day by day.
woks only if training a small number of models.

Recommentd:
a_0 have high priority to tune, have huge impact. Learning rate decay does help , but a little bit lower down in tunning list

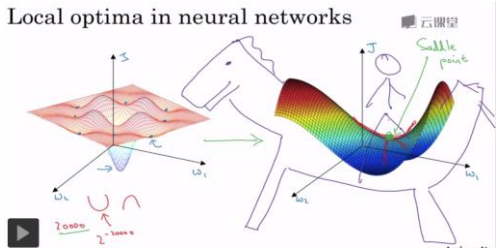
10: Local optima in deep learning

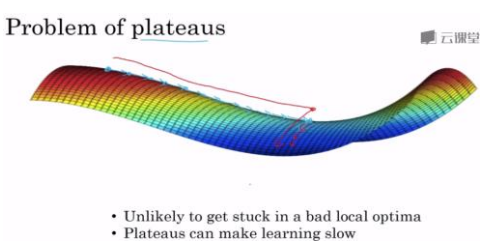
10.1: Intuition : local optima

>1. Intuition: for low dimension parameter
Previous thinking for local optima:for low dimension parameter:
cost function: have many local optima, where gradient easy to get stuck to find global optima.

>2. This intuition incorrect for deep learning:
Most points of 0 gradients, are not local optima, but saddle points instead.
for a function of high dimension, when gradient =0, then in this parameter direction it can either be convex light funciton, or concave light function.
so in this high dimension space, for 0 gradient point to be a local optima, all parameter directions need to be convex. the chance of this happening is very small. more likely some curve bend up in one direction , and bend down in other direction, rather than all bend upwards.

-->in very high dimensional space, actually more likely to run into a saddle point.
A lot of intuitions about low-dimensional spaces- like cost function local optima, really do not transfer to the very high-dimensional spaces





10.2 Plateaus problem for deep learning

Local optimum is not a problem for deep learning,, instead will meet plateaus problem.

Plateaus: the region where the derivative is close to zero for a long time.
E.g:
in the plateau region , surface is platten, gradient descent in all direciton close to 0,
take a very long time to slowly find way to maybe the saddle point on the plateau.-blue curve.
and then because of a random perturbation of left/right, then finally find way off the plateau- red line.
(mini-batch noise on cost function helps for going out of plateau.)

Plateau problem for deep learning: take a very long slop/step to the saddel point and then get off this plateau.

Summary:
>1. Unlikely to get stuck in a bad local optima;
on conditons: as long as training a reasonably large neural network.-->(high dimension space)
A lot of parameters, and cost function is defined over a relatively high dimensional space.

>2. Plateaus can make learning slow.
this is where Momentum / RMSprop/ Adam can really help learning algorithm as well.!!!
and this is cases where more sophisticated observation algorithms like Adam can actually speed up the rate at which you could move down the plateau and then get off the plateau.

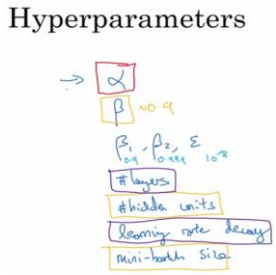
2-3_Hyperamter tuning

1. Hyperparameters

1.1. Hyperparameters

- α: learning rater: morst import hyperparameter to tune
- β, =0.9 good default, momentum
- mini-batch size: tune to make sure optimzation algorithm runing efficiently
- hidden units: w, b
- layers
- learnig rate decay,
- β1=0.9, β2=0.999,ε=10^-8:RTM, Adam: never tune in Adm

red: most important to tune
orange: second import to tune
Purple: thirsd important



1.2 Try random values:

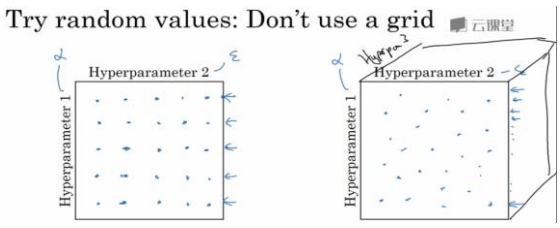
>.1: Grid:
sample points in grid (fix parameter 1, sampling parameer 2)-sample a row number, and then systematicall explore these values (fix sampled paramter 2, explore paramter 1)-exlore vertically.
then try all these points see which one works betst.

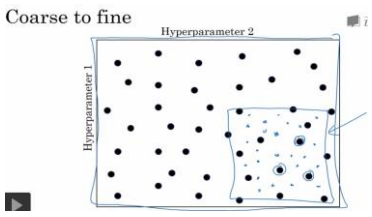
works ok when hyperparameter number was relatively small.

>2: Random choose for deep learning
in deep learning, use random choosing instead. and try these random points on algorithm.

>3 Reason for random choosing for deep learning
>>1. as could not know in advance which hyperparameter is the most important.
E.g, right pic.: hp1 is learning rate, hp2 is ε in RTM. E is not so important, 5 points: one hp1 with 5 different hp2, result no big different as rely more on hp1, same in 5 points-->25 points in grid actually could only test 5 hp1 values.
by using random points: righ pic.: all points have unique hp1, hp2.-->none of two points forced to have same hp2/hp1, both hp1, hp2 tested 25 values instead of 5. have more chance to find a value that works really well.

when have more paramters, sample paramter in higher dimension space-->
random choosing, ensure that you are more richly exploring set of possible values for the most important hyperparameters





>4: Coarse to fine
except above random choosing, second common practice in Sampling hyperametes is to use a coarse to fine sampling scheme.

>>>1. In above randome choosing sampling, run algorithm and find a points/ a few other points aournd it tended to work really well.

>>>2. Zoom in to a smaller region of the these hypperparameters points, and then sample more density within this space.
maybe again randomly, but to then focus more resources on searching within this parameter region-blue squre, if suspecting best setting /hyperparameters maybe in this region.

after doing a coarse sample of this entire square, then result tells to focus on a smaller square, then sample more densely into smaller square.

Coarse to fine search is frequently used.

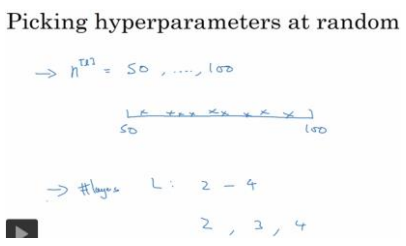
2. Using an appropriate scale to pick hypeparameters

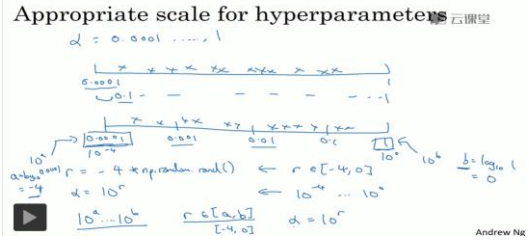
While searching hyperparamters, sampling at random over the range of hyperparamters, could allow to search over the hyperparameter space more efficiently.

But sampling random does not mean sampling 'uniformly' at random over the range of valid values. instead it's important to pick the appropriate scale. on which to explore the hyperparameters.

2.1 : Picking hyperparameters at random

>1. : Sampling uniformly at random over range considered
e.g:
hiddien units in layers: n_l: could picking some number vaues at random within number line: 50-100
hidden layer number L: sampling uniformy at random along 2-4 , might be reasonable/ or could use grid search for value 2, 3,4.





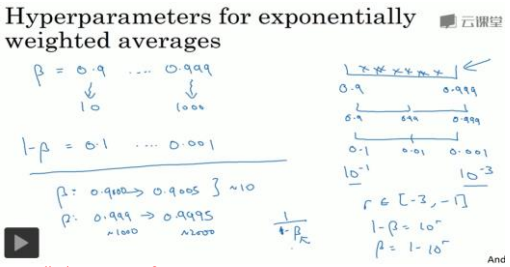
2.2: Appropriate scale for hyperparameters

>Log scale:
e.g learning rate: considering range: 0.0001 -1
if sampling uniformly at random, 90% percent sample would be between 0.1-1, -->90% of the resources to search between 0.1-1, only 10% to search 0.0001-0.1 -->more reasonable to search on a log scale, and sampling uniformly at random on this new scale. now have more resource to deviated to 0.0001 - 0.1.

code:
r=-4* np.random.rand() --->[-4, 0];
a=10^r; -->[10^-4, 10^0]

summarize:
if try to sample on range 10^a-10^b, on log scale, then sample r uniformly at random, between a and b.
if sample on log scale, take the low value , take logs to figure out what is a, and take log of high value to find b, then sampling r uniformly between a and b.

2.3 Hyperparameters for exponentailly weighted averages



β : considered range: 0.9 -0.999

Remember β means: -->using 1/ (1-β) previous data:
β : considered average days range is : 10 - 1000
Using linear scale on range 0.9-0.999, does not make sense;

better explore: 1-β : 0.1 - 0.0001. use log scale: --> [-3, -1]

Reason do not use linear scale directly is:
when sampling close to 1, the sensitivity of the result changes, even with very small changes to β.
e.g when β change from 0.9 (10 days)to 0.9005 (10.5 days), there is hardly any change in result, but when β change from 0.999 (1000days)to 0.9995 (2000 days average), this will have a huge impact on what algorithm is doing. (1/ (1-β) is very sensitive to β when it is close to 0)

need to sample more densely in the region of when β close to 1, so can **more efficient in terms of how distribute the samples to explore the space of possible outcomes more efficiently.**

Summary:
In case you don't end up making the right scaling decision on some hyperparameter choice, don't worry to much about it.
Even if you sample on the uniform scale, where some of the scale would have been superior, you might still get okay results, especially if you use a coarse to fine search: in later iterations, will focus more on the most useful range of hyperparameter values to sample.

3: Hyperparameters tuning in practice

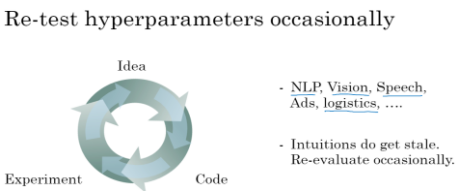
Item purpose: Final tips and tricks for how to organize hyperparameter search process.

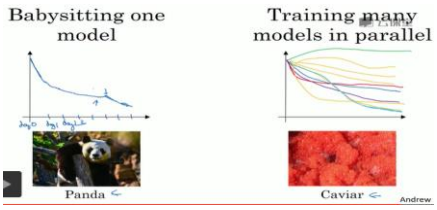
3.1 Re-test hyperparameters occasionally;

>1. cross-fertilization: NLP, Vision, Speech, Ads, logistics.....
Deep learning today is applied to many different application areas and that intuitions about hyperparameter settings from one application area may or may not transfer to a different one. There is a lot of cross-fertilization among different applications' domains, so for example, I've seen ideas developed in the computer vision community, such as Convnets or ResNets, which we'll talk about in a later course, successfully applied to speech. I've seen ideas that were first developed in speech successfully applied in NLP, and so on.

So one nice development in deep learning is that people from different application domains do read increasingly research papers from other application domains to look for inspiration for cross-fertilization.

>Intuitions do get stale. Re-evaluate occaionally.
In terms of your settings for the hyperparameters, though, I've seen that intuitions do get stale. So even if you work on just one problem, say logistics, you might have found a good setting for the hyperparameters and kept on developing your algorithm, or maybe seen your data gradually change over the course of several months, or maybe just upgraded servers in your data center. And because of those changes, the best setting of your hyperparameters can get stale. So I recommend maybe just retesting or reevaluating your hyperparameters at least once every several months to make sure that you're still happy with the values you have.





3.2 Two method in searching for hyperparameters,
>1. babysit one model.
>>Conditions: have maybe a huge data set but not a lot of computational resources- CPUs or GPU, train only one model or a very small number of models at a time.
>>Idea: gradually babysit that model even as it's training : watching performance and patiently nudging parameter up or down
Every day you kind of look at algorithm performance and try nudging up and down your parameters: kind of babysitting the model one day at a time even as it's training over a course of many days or over the course of several different weeks..

E.G: on first initialize your parameter as random and then start training. And gradually watch learning curve, maybe the cost function J or your dataset error or something else, gradually decrease over the first day. Then at the end of day one, you might say, gee, looks it's learning quite well, I'm going to try increasing the learning rate a little bit and see how it does. And then maybe it does better. And then that's your Day 2 performance. And after two days you say, okay, it's still doing quite well. Maybe I'll fill the momentum term a bit or decrease the learning variable a bit now, and then you're now into Day 3. And every day you kind of look at it and try nudging up and down your parameters. And maybe on one day you found your learning rate was too big. So you might go back to the previous day's model, and so on. But you're kind of babysitting the model one day at a time even as it's training over a course of many days or over the course of several different weeks.

also called panda approach:
When pandas have children, they have very few children, usually one child at a time, and then they really put a lot of effort into making sure that the baby panda survives. So that's really babysitting. One model or one baby panda.

>2 Train many models in parallel.
train different model with different hyperparameters setting at the same time, and generate different learning curve.
so this way you can try a lot of different hyperparameter settings and then just maybe quickly at the end pick the one that works best.

alsoc called caviar （鱼子酱）strategy:
like fish that lay over 100 million eggs in one mating season. But the way fish reproduce is they lay a lot of eggs and don't pay too much attention to any one of them, just see that hopefully one of them, or maybe a bunch of them, will do well.

Summary:
>1. So the way to choose between these two approaches is really a function of how much computational resources you have.
If have enough computers to train a lot of models in parallel, then by all means take the caviar approach and try a lot of different hyperparameters and see what works.

Otherwise, use babysitting , but could baby sitting on more than one models:
In some application domains, e.g online advertising settings and computer vision applications, there's just so much data and the models you want to train are so big that it's difficult to train a lot of models at the same time. It's really application dependent of course, but those communities use the panda approach a little bit more: kind of baby sitting on a single model along and nudging the parameters up and down and trying to make this one model work.
Although, even the panda approach, having trained one model and then seen it work or not work, maybe in the second week or the third week, maybe I should initialize a different model and then baby that one along just like even pandas, I guess, can have multiple children in their lifetime, even if they have only one, or a very small number of children, at any one time.

2.3-4 Normalizing activaitons in a network_Batch Normalization

Item purpose:
Make neural network much more robust to the choice of hyperparameters. It doesn't work for all neural networks, but when it does, it can make the hyperparameter search much easier and also make training go much faster.

In the rise of deep learning, one of the most important ideas has been an algorithm called batch normalization. Batch normalization makes hyperparameter search problem much easier, makes neural network much more robust. The choice of hyperparameters is a much bigger range of hyperparameters that work well, and will also enable you to much more easily train even very deep networks.

1. Normalizing: logistic and deep model

In logistic regression, normizing input feature could make cost function from elongated to more round over perameter w, b, easier for algorithm like gradient descent to optimize.
But in deep model, not only have input featurex x, but have actitavtion input a[l].
-->for any hidden layer, normlize a[l-1]/Z[l-1] to make training of w[l], b[l] more efficient?

Note: have choices to normalize a[l] or Z[l], better take z[l] as default. (why?)
对每层进行Batch Normalization的时候一般都是先进行normalization再传进active function的， 好处就是可以让输入更多地落在active function (比如sigmoid和tanh)的微分比较大的地方也就是零附近。

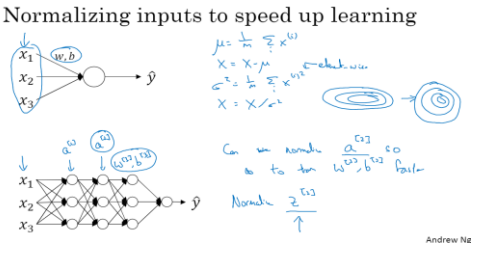
2. Implementing Batch Norm:

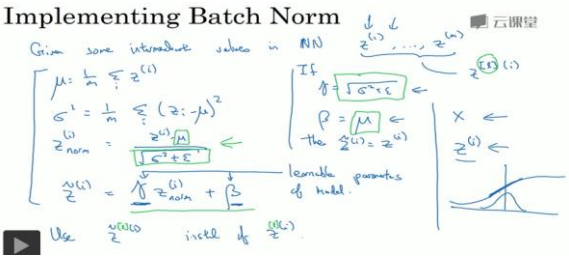
for layer l: Z[l] = [z[l]_1, ...z[l]_m] : m example. dimension: (n_l x m)
u= z[l] average over m example: dimension: (n_l x 1)
б^2= average variance of z[l] over all m examples: dimension: (n_l x 1)

z[l]_i_norm = z[l]_i -u)/ sqrt(б^2+ε) : mean 0 and variance 1; dimension: (n_l x 1)

But do not want hidden units always have 0 mean and 1 variance-->
~z[l]_i = γ (z[l]_i -u)/ sqrt(б^2+ε) + β

γ, β: learnable parameter using gradient descent (momentum, Adam, etc.)



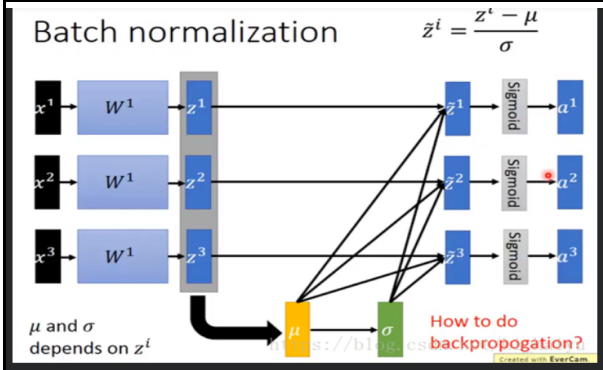


Intuition:
>1. Normalizing the input features x can help learning in a neural network, and what batch norm does is it applies that normalization process not just to the input layer, but to the values even deep in some hidden layer in the neural network. So it will apply this type of normalization to normalize the mean and variance of some of your hidden units' values, z.

>2. But one difference between the training input and these hidden unit values is you might not want your hidden unit values be forced to have mean 0 and variance 1.

e.g: while using sigmoid activation function, might want z to have a larger variance or have a mean that's different than 0, so have a wider range not only under sigmoid function linear area, in order to better take advantage of the nonlinearity of the sigmoid function rather than have all your values be in just this linear regime.

So parameters γ and β, can now make sure that z[l]_i values have the range of values that you want, normalizes mean and variance of these hidden unit values, z[l]_i to have some fixed mean and variance. (seperate γ, β for each units in layer l?-vector of output z size?)



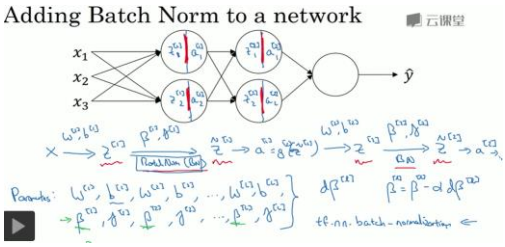
在每层的active function之前都进行一次batch normalization那么做了batch normalization之后的方向传播又是怎么进行的呢?

在方向后传播的时候，一路反向传播回来的时候是会通过σ然后通过μ然后去update Z的，如果在BP过程中没有考虑μ和σ的话是不对的，因为μ和σ是依赖于Z的，改动了Z就相当于改动了μ和σ，所以在training的时候μ和σ是会被考虑进去的。

现在我们把Z normalize成Z̃ ，然后在active function输入之前还可以增加β和γ参数，这两个参数在training的时候也会被学习到。

2.3-5 Batch norm into neuroal work

1. Add batch norm to neuroal work:



>1. Norm add between Z & a(Z).
Each hidden units can take it as computing two things: z and a.

In batch norm:
X---w[1], b[1]----->Z[1]------y[1], β[1] ----->~Z[1]-------->A[1] , ------w[2],b[2]----->Z[2]------y[2], β[2] ----->~Z[2]....

normalization: between Z[l] and A[l], and use ~z not the z_norm

Parameters: w[l], b[l], y[l], β[l].
b[l] could be removed now, and ~z mean now is controlled by β[l]

>2. gradient descent with batch norm: could still use momentum, RMSP, Adam;
>3. Batch norm appliciton in coding: could use one code in phython: tf.nn. Batch_normalization

2. Working with mini batch

In practice, Batch Norm is usually applied with mini-batches of your training set.

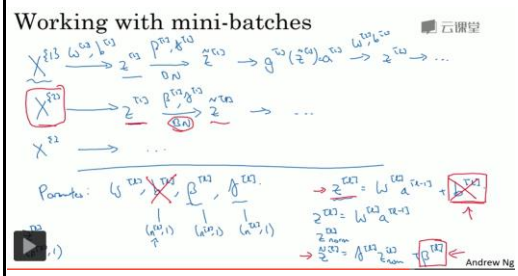
>1. Batch Norm in mini batch:
>>1. first mini-batch:
X{1}---w[1], b[1]----->Z[1]------y[1], β[1] ----->~Z[1]-------->A[1] , ------w[2],b[2]----->Z[2]------y[2], β[2] ----->~Z[2]....

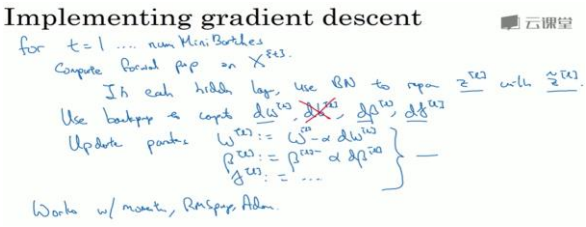
Note: computer mean and variance of the Z[l] on just this mini batch and then re-scale Z[l] by y, β to get ~ Z[L], all this is done on the first mini-batch.

finish one step gradient descent on the mini-batch. then go to second mini-batch, do something similar.
X{2}---w[1], b[1]----->Z[1]------y[1], β[1] ----->~Z[1]-------->A[1] , ------w[2],b[2]----->Z[2]------y[2], β[2] ----->~Z[2]....
Note: normalizing z[l] using just the data in second mini-batch: computing mean u and variance σ^2 on second mini-batch data, re-scale Z[l] by y, β to get ~ Z[l].

>2. b[l] removed as parameter or set to 0: as any constant adde will get cancelled out by the mean normalization step.
β[l] instead used to decide mean of ~z[l]

>3. Y[l] & β[l]: dimension: ((n[l],1)/ (n[l],m). Each hidden units in layer l(in all dimension) have 1 Y value for setting virance: scale the mean and variance of each of the hidden values (totall n[l] values)





3.Implementing batch norm in gradient descent with mini-batch

>1. one epoch: go trough all iterations in one epoch (iteration number = batch size /mini-size)

>2 Inner loop: for each iteration/mini-batch
>>1. foward pro:
in each hidden layer, use BN to replace Z[l] with Z~[[];
Make sure in that mini-batch, the values and version of the normalized mean and variance is ~Z[l];

>>2. back pro: compute dw[l], dY[l], dβ[l]
>>3. update parmaters: w[l]-=a*dw[l[; Y[l]-=a*dy, [l], β[l]-=a* dβ[l]
in this step could use momentum, RTM, Adam for fasting move

Note: Instead of taking this gradient descent update mini-batch, could use the updates given by these other algorithms as we discussed in the previous week's videos (Momentum, RTM, ADAM). Some of these other optimization algorithms as well can be used to update the parameters β and y that Batch Norm added to algorithm.

2.3-6 Why does Batch Norm work

Normalizing all input features X, to take on a similar range of values can speed up learning. So, one intuition behind why batch norm works is, this is doing a similar thing, but further values in your hidden units and not just for your input there.

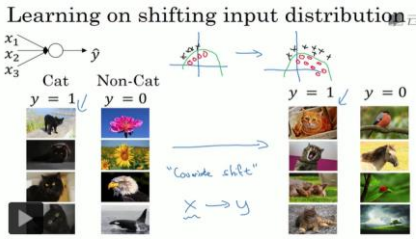
1. Learning on shifting input distribution

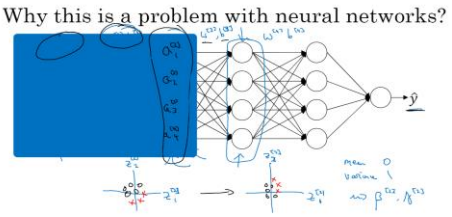
Weight, later or deeper that network. more robust to changes of weight in ealier layers

>1. covariate shif: Data distribution changing influence
e.g: shallow NN like logistic regression, for cat classification;
Have trained algorithm with training set of black cats, -->now apply classfier to colored cat,classfier may not doing well.

algorithm trained on one data set, may does not do well on data set generalized from origianl data set- decison bounday may could not apply well to the new generalized data set.

Covariate shift: data distribution changing .
for the learned X to Y mapping, if the distribution of X changes, then might need to retrain learning algorithm.
(note: normally require training set, dev set and test set same distribution, training set cover all the dev/test set features-->algorithm generalize better on dev/test set.
dev set and test set have to be same distribution; training set may come from different distribution -as need large data to train, data may from other distribution.)
And this is true even if the ground true function, mapping from X to Y, remains unchanged, which it is in this example, because the ground true function is: is this picture a cat or not. And the need to retrain your function becomes even more acute or it becomes even worse if the ground true function shifts as well.





2. covariate shif in neural network

2.1 e.g:
layer 3: w[3], b[3] , input a[2]
>1. Layer 3 purpose: take input a[2] and find a way to map a[2] to output a[3]= y^.
-->w[3], b[3], w[4], b[4] learned and updated to do this mapping well

>2. while whole network also adapping paramer in ealier layers w[1], ..b[2].
when these parameters change, a[2] as input to layer 3 also change--->

lalyer 3 have covariate shift problem. (input distribution changed)

-covariate shif: changes in previous layer (hyperpramater, traning data) will influence current layer hyperpramater : eg. Layer L , W[l], b[l] have trained based on input a[l-1], (who trained on batch X[t] training data), to estimate output y. if a[l-1] changed, W[l], b[l] normally have to change to estimate same y (if output do not change).

cost = sum cost in each layer c_l over all layers
-->cost error in hidden layers could not pass to next/cost function?-backprop;
covariate shift is forward prop, how to influence on backprop: ?-->
parameter update-->output of each layer change-->cost error_l shifting,

2.2 Batch norm helps covariate shifting

Batch norm: Make parameter e.g weight in the later or deeper network layer, more robust to changes of weight in ealier layers.
>1. batch norm: reduce the amount that the distribution of the hidden unite values shifting around.

when parameters in the neural nework ealier layer changes ---> the Z[l] will change.
Batch norm will make sure hidden unite values ((Z[l]_i)'s mean and virance remain same, -->holding hidden unite values in certain region under all changes; limit the amount to which updating the paramters in ealier layer can affect the distribution of hidden unit values. which is the next layer sees and has to learn on.

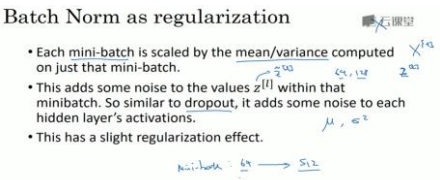
So batch norm reduces the problem of the input values changing, it really causes these values to become more stable, so that the later layers of the neural network has more firm ground to stand on.

>2. And even as the earlier layers keep learning, the amounts that this forces the later layers to adapt to as early as layer changes is reduced or, it weakens the coupling between what the early layers parameters has to do and what the later layers parameters have to do. --->allows each layer of the network to learn by itself, a little bit more independently of other layers, and this has the effect of speeding up of learning in the whole network.

>3. Eespecially from the perspective of one of the later layers of the neural network, the earlier layers don't get to shift around as much, because they're constrained to have the same mean and variance. And so this makes the job of learning on the later layers easier.

2.3 Batch norm as regulization

>1. Each mini-batch X_t, has the values Z_t, has the values Z_l, scaled by the mean and variance computed on just that one mini-batch.
--> as the mean and variance computed on just that mini-batch as opposed to computed on the entire data set, that mean and variance has a little bit of noise in it, the scaling process of Z [l] is a little bit noisy as well.



>2. This adds some noise to the values Z[l] within that minibatch.
So similar to dropout, it adds some noise to each hidden layer's activations.
dropout noise: takes a hidden unit (activation output) and multiplies it by zero with some probability.

So your dropout has multiple of noise because it's multiplied by zero or one, whereas batch norm has multiples of noise because of scaling by the standard deviation, as well as additive noise because it's subtracting the mean. while the estimates of the mean and the standard deviation are noisy.

So similar to dropout, batch norm therefore has a slight regularization effect : **by adding noise to the hidden units, it's forcing the downstream hidden units not to rely too much on any one hidden unit.**
note: regularization effect: reduce weight

>3. This has a slight regularization effect:
Because the noise added is quite small, and therefore it is a very slight regularization effect.

Could choose to use batch norm together with dropout if want the more powerful regularization effect of dropout.

Note:
>1. using a bigger mini-batch size, will reducing this noise and therefore also reducing this regularization effect.
So that's one strange property of dropout which is that by using a bigger mini-batch size, you reduce the regularization effect.
>2. Do not use batch norm as a regularizer.
That's really not the intent of batch norm, but sometimes it has this extra intended or unintended effect on your learning algorithm. But don't turn to batch norm as a regularization. Use it as a way to normalize your hidden units activations and therefore speed up learning.

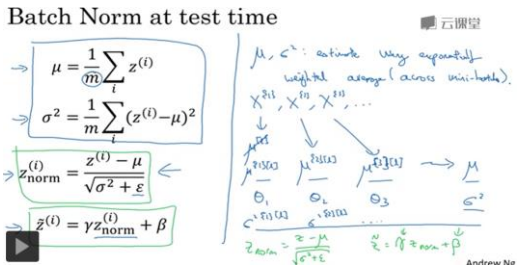
>3. Batch norm handles data one mini-batch at a time. It computes mean and variances on mini-batches.
So at test time, you try and make predictors, try and evaluate the neural network, you might not have a mini-batch of examples, you might be processing one single example at the time. So, at test time you need to do something slightly differently to make sure your predictions make sense.

2.3-7 Batch norm at test

1. mean& virance in minibatch-one layer: u=1/m sum z[i]: m mini batch size;
u and 6^2 in layer l, for scaling Z[l], are computed on the entire mini batch examples.

2. mean & virance in test-each layer:
when have only one example in test dev, normlization by using it's own u and 6^2 does not make sense(u=z, D(z)=0, -->~z no value).-->sepeate estimate of u and 6^2 :
Using expension weight average-across mini batch (save storage): averager the values in layer l over previous values used /computed (in different iterations/mini-batch):
e.g:
X{1}---->6^2 /u[l]_{1}: mean computed and used in scaling in layer l, with first mini-batch example.
X{2}---->6^2 /u[l]_{1}: mean computed and used in scaling in layer l, with second mini-batch example.
....
X{t}---->6^2 /u[l]_{t}: mean computed and used in scaling in layer l, with t-th mini-batch example.

keep running average of u and 6^2 that computed and used in each layer as train the neural network across different mini-batches.
and use in test example: Z_norm = z-u / sqrt (6^2 +ε)



2.3-8 Softmax layer_introduciton

1. Softmax:

Softmax: generalization of logistic regression, used when have mulitiple classification, :make predictions where you're trying to recognize one of multiple classes, rather than just recognize two classes.

Output: probability of each of classes: p(c_i |x).
C: classes number;
Output vector dimension: (C,1)
Output sum of all ouput is =1.

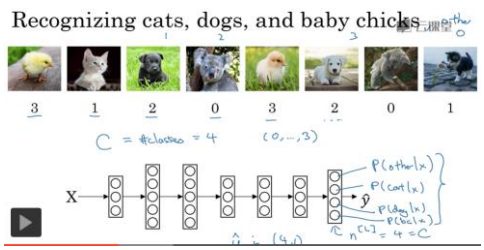
softmax layer used in neural network as output layer, in order to generate these ouputs.

2. Softmax layer

outlayer: Z[L] = W[L] . A[L-1] +b[L]

activation function:
t_i= e^Z[L]_i, same dimension as Z[l]
A[L]_i: =t)i/sum t_j over all j classes/units in output layer
= e^Z[L]_i / sum e^Z[L]_j over all elements.
A[[L]: Same dimension as Z[L]; normalize t.

activation function: A[L] = g(Z[L])
take a vector z[L] and output a vector A[L].
(note: for m examples: Z[L]: C X m matrix:
-->A[L] = e^Z[L] / (np. sum (e^Z[L]. dim=1)
)



3. softmax example: no hidden layer

Z[1] = W[1].X+b
A[1]= g)W[1])

softmax layer with C output classes can represent this type of decision boundaries-several linear decision boundaries, allows it to separate out the data into C classes.

Coloring in the right pic./input based on which one of the c outputs have the highest probability.
can kind of see it like a genralization of logistic regression, with sort of linear decision boundaries

Note:
>1. without any hidden layer, decision boundary between any two classes will be linear (e.g in right)
>2 with hidden layers, can learn more complex non-linear decision boundaries to separate out mulitple diffrent classes.

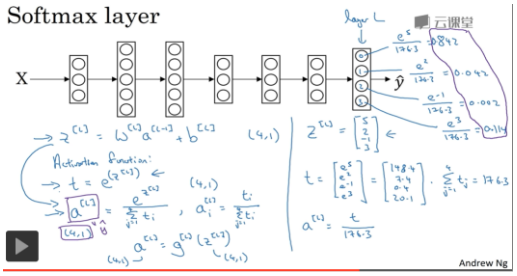
3. softmax example: no hidden layer
[x1, x2]T-->w: [neuro1, neuro2, neuro3]-->y [y1,y2,y3].
Z=W[w11, w12; w21, w22]* [x1, x2]T +b (default 0)=[y1, y2,y3]
y1=exp (w11*x1+w12*x2)/(y1+y2+y3)
y2=exp (w21*x1+w22*x2)/(y1+y2+y3)
y3=exp (w31*x1+w32*x2)/(y1+y2+y3)
y1+y2+y3=1

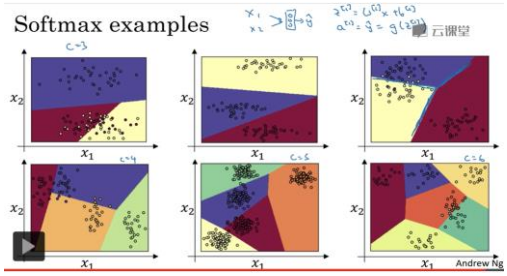
right pic: train softmax with three output labes on the data, colors in plot show output threshold in softmax classifier. and coloring in the input based on which one of the three output units have highest probability -->like generalization of logistic regression, with sort of linear boundaries (more complicated non-linear boundaries could be learned with hidden layers-activation function-->Y=g(Z))=g(w11*x1+w12*x2), but with more than two classes.

-Purpose: last layer activation fucntion is a=e^z[l]/sum e^z[l, i]-sum of output in all dimension (all classes)
output is probabilities for more than 2 classification: use e^z[i] function , make sure each unite value e^[l-i feature/dimension] probabity >0, divide sum of all out put probility to make sure: all sum output probality <1.

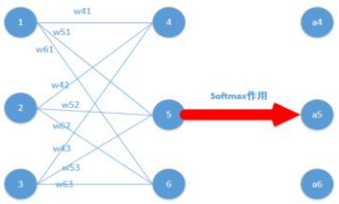
note: two classfication previous talked , activation fucntion is Relu/sigmoid, last layer z[l] is a number (dimension 1), -->out put is also a number -one dimension.

multi classfication last layer Z[l] dimension is classifcation number,-->out put a[l]=g[Z[l])=e^z[l] is also multi dimension.
- output: probability for each output/class unit with input x(i) : p(y1 | X [i]). output dimension number=classification number.
note: when classificaiton number is two, then equal to two classification: only one out put (as the other one could calculated by 1-y1 (sum output is 1)).
-softmax generalize logistic activation function to class C rather than just two classes.





softmax derivative



交叉熵函数形式如下：

$$Loss = -\sum_i y_i ln a_i$$

其中y代表我们的真实值，a代表我们softmax求出的值。i代表的是输出结点的标号！在上面例子，i就可以取值为4,5,6三个结点（**当然我这里只是为了简单，真实应用中可能有很多结点**）

现在看起来是不是感觉复杂了，居然还有累和，然后还要求导，每一个a都是softmax之后的形式！

但是实际上不是这样的，我们往往在真实中，如果只预测一个结果，那么在目标中只有一个结点的值为1，比如我认为在该状态下，我想要输出的是第四个动作（第四个结点），那么训练数据的输出就是a4 = 1,a5=0,a6=0，哎呀，这太好了，除了一个为1，其它都是0，那么所谓的求和符合，就是一个幌子，我可以去掉啦！

为了形式化说明，我这里认为训练数据的真实输出为第j个为1，其它均为0！

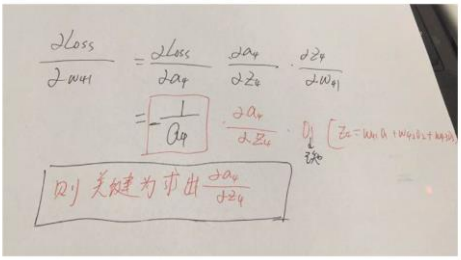
那么Loss就变成了 $Loss = -y_j ln a_j$,累和已经去掉了，太好了。现在我们要开始求导数了！

我们在整理一下上面公式，为了更加明白的看出相关变量的关系：

其中 $y_j = 1$,那么形式变为 $Loss = -ln a_j$

那么形式越来越简单了，求导分析如下：

参数的形式在该例子中，总共分为w41,w42,w43,w51,w52,w53,w61,w62,w63.这些，那么比如我要求出w41,w42,w43的偏导，就需要将Loss函数求偏导传到结点4，然后再利用链式法则继续求导即可，举个例子此时求w41的偏导为：



w51.....w63等参数的偏导同理可以求出，那么我们的关键就在于Loss函数对于结点4,5,6的偏导怎么求，如下：

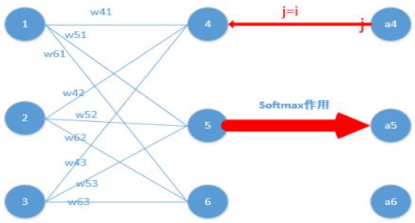
这里分为俩种情况：

if $j = i$:

$$\frac{\partial a_j}{\partial z_i} = \frac{\partial}{\partial z_i}\left(\frac{e^{z_j}}{\sum_k e^{z_k}}\right) = \frac{(e^{z_j})' \cdot \sum_k e^{z_k} - e^{z_j} \cdot e^{z_j}}{\left(\sum_k e^{z_k}\right)^2} = \frac{e^{z_j}}{\sum_k e^{z_k}} - \frac{e^{z_j}}{\sum_k e^{z_k}} \cdot \frac{e^{z_j}}{\sum_k e^{z_k}} = a_j(1 - a_j)$$

j=i对应例子里就是如下图所示：

比如我选定了j为4，那么就是说我现在求导传到4结点这！



那么由上面求导结果再乘以交叉熵损失函数求导

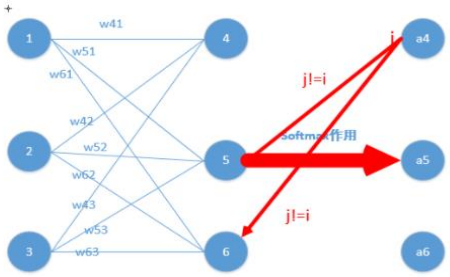
$Loss = -ln a_j$ ，它的导数为 $-\frac{1}{a_j}$,与上面 $a_j(1 - a_j)$ 相乘为 $a_j - 1$ （**形式非常简单，这说明我只要正向求一次得出结果，然后反向传梯度的时候，只需要将它结果减1即可，后面还会举例子！**）那么我们可以得到Loss对于4结点的偏导就求出了了（**这里假定4是我们的预计输出**）

第二种情况为：

if $j \neq i$:

$$\frac{\partial a_j}{\partial z_i} = \frac{\partial}{\partial z_i}\left(\frac{e^{z_j}}{\sum_k e^{z_k}}\right) = \frac{0 \cdot \sum_k e^{z_k} - e^{z_j} \cdot e^{z_i}}{\left(\sum_k e^{z_k}\right)^2} = -\frac{e^{z_j}}{\sum_k e^{z_k}} \cdot \frac{e^{z_i}}{\sum_k e^{z_k}} = -a_j a_i$$

这里对应我的例子图如下，我这时对的是j不等于i，往前传：



那么由上面求导结果再乘以交叉熵损失函数求导

$Loss = -ln a_j$ ，它的导数为 $-\frac{1}{a_j}$,与上面 $-a_j a_i$ 相乘为 a_i （**形式非常简单，这说明我只要正向求一次得出结果，然后反向传梯度的时候，只需要将它结果保存即可，后续例子会讲到**）这里就求出了除4之外的其它所有结点的偏导，然后利用链式法则继续传递过去即可！我们的问题也就解决了！

loss function:

交叉熵代价函数（Cross-entropy cost function）是用来衡量人工神经网络（ANN）的预测值与实际值的一种方式。与二次代价函数相比，它能更有效地促进ANN的训练。在介绍交叉熵代价函数之前，本文先简要介绍二次代价函数，以及其存在的不足。

1. 二次代价函数的不足

ANN的设计目的之一是为了使机器可以像人一样学习知识。人在学习分析新事物时，当发现自己犯的错误越大时，改正的力度就越大。比如投篮：当运动员发现自己的投篮方向离正确方向越远，那么他调整的投篮角度就应该越大，篮球就更容易投进篮筐。同理，**我们希望：ANN在训练时，如果预测值与实际值的误差越大，那么在反向传播训练的过程中，各种参数调整的幅度就要更大，从而使训练更快收敛。**然而，如果使用二次代价函数训练ANN，看到的实际效果是，如果误差越大，参数调整的幅度可能更小，训练更缓慢。

以一个神经元的二类分类训练为例，进行两次实验（ANN常用的激活函数为sigmoid函数，该实验也采用该函数）：输入一个相同的样本数据x=1.0（该样本对应的实际分类y=0）；两次实验各自随机初始化参数，从而在各自的第一次前向传播后得到不同的输出值，形成不同的代价（误差）：

实验1：第一次输出值为0.82

实验2：第一次输出值为0.98

在实验1中，随机初始化参数，使得第一次输出值为0.82（该样本对应的实际值为0）；经过300次迭代训练后，输出值由0.82降到0.09，逼近实际值。而在实验2中，第一次输出值为0.98，同样经过300迭代训练，输出值只降到了0.20。

从两次实验的代价曲线中可以看出：**实验1的代价随着训练次数增加而快速降低，但实验2的代价在一开始下降得非常缓慢；直观上看，初始的误差越大，收敛得越缓慢。**

其实，误差大导致训练缓慢的原因在于使用了二次代价函数。二次代价函数的公式如下：

$$C=\frac{1}{2n}\sum_x\|y(x)-a^L(x)\|^2$$

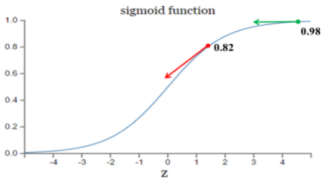
其中，C表示代价，x表示样本，y表示实际值，a表示输出值，n表示样本的总数。为简单起见，同样一个样本为例进行说明，此时二次代价函数为：

$$C=\frac{(y-a)^2}{2}$$

目前训练ANN最有效的算法是反向传播算法。简而言之，训练ANN就是通过反向传播代价，以减少代价为导向，调整参数。参数主要有：神经元之间的连接权重w，以及每个神经元本身的偏置b。调参的方式是采用梯度下降算法（Gradient descent），沿着梯度方向调整参数大小。w和b的梯度推导如下：

$$\frac{\partial C}{\partial w}=(a-y)\sigma'(z)x$$
$$\frac{\partial C}{\partial b}=(a-y)\sigma'(z)$$

其中，z表示神经元的输入，σ表示激活函数。从以上公式可以看出，w和b的梯度跟激活函数的梯度成正比，激活函数的梯度越大，w和b的大小调整得越快，训练收敛得就越快。而神经网络常用的激活函数为sigmoid函数，该函数的曲线如下所示：



如图所示，**实验2的初始输出值（0.98）对应的梯度明显小于实验1的输出值（0.82），因此实验2的参数梯度下降得比实验1慢。这就是初始的代价（误差）越大，导致训练越慢的原因。**与我们的期望不符，即：不能像人一样，错误越大，改正的幅度越大，从而学习得越快。

可能有人会说，那就选择一个梯度不变化或变化不明显的激活函数不就解决问题了吗？图样图森破，那样虽然简单粗暴地解决了这个问题，但可能会引起其他更多更麻烦的问题。而且，类似sigmoid这样的函数（比如tanh函数）有很多优点，非常适合用来做激活函数，具体请自行google之。

2. 交叉熵代价函数

换个思路，我们下换激活函数，而是换掉二次代价函数，改用交叉熵代价函数：

$$C=-\frac{1}{n}\sum_x[y\ln a+(1-y)\ln(1-a)]$$

其中，x表示样本，n表示样本的总数。那么，重新计算参数w的梯度：

$$\begin{aligned}\frac{\partial C}{\partial w_j}&=-\frac{1}{n}\sum_x\left(\frac{y}{\sigma(z)}-\frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial\sigma}{\partial w_j}\\&=-\frac{1}{n}\sum_x\left(\frac{y}{\sigma(z)}-\frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j\\&=\frac{1}{n}\sum_x\frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y)\\&=\frac{1}{n}\sum_x x_j(\sigma(z)-y)\end{aligned}$$

其中（具体证明见附录）：

$$\sigma'(z)=\sigma(z)(1-\sigma(z))$$

因此，w的梯度公式中原来的$\sigma'(z)$被消掉了；另外，该梯度公式中的$\sigma(z)-y$表示输出值与实际值之间的误差。所以，当误差越大，梯度就越大，参数w调整得越快，训练速度也就越快。同理可得，b的梯度为：

$$\frac{\partial C}{\partial b}=\frac{1}{n}\sum_x(\sigma(z)-y)$$

实际情况证明，交叉熵代价函数带来的训练效果往往比二次代价函数要好。

3. 交叉熵代价函数是如何产生的？

以偏置b的梯度计算为例，推导出交叉熵代价函数：

$$\begin{aligned}\frac{\partial C}{\partial b}&=\frac{\partial C}{\partial a}\cdot\frac{\partial a}{\partial z}\cdot\frac{\partial z}{\partial b}\\&=\frac{\partial C}{\partial a}\cdot\sigma'(z)\cdot\frac{\partial(wx+b)}{\partial b}\\&=\frac{\partial C}{\partial a}\cdot\sigma'(z)\\&=\frac{\partial C}{\partial a}\cdot a(1-a)\end{aligned}$$

在第1小节中，由二次代价函数推导出来的b的梯度公式为：

$$\frac{\partial C}{\partial b}=(a-y)\sigma'(z)$$

为了消掉该公式中的$\sigma'(z)$，我们想找到一个代价函数使得：

$$\frac{\partial C}{\partial b}=(a-y)$$

即：

$$\frac{\partial C}{\partial a}\cdot a(1-a)=(a-y)$$

对两侧求积分，可得：

$$C=-[y\ln a+(1-y)\ln(1-a)]+\text{constant}$$

而这就是前面介绍的交叉熵代价函数。

softmax的log似然代价函数（公式求导）

2. softmax配合log似然代价函数训练ANN

在上一篇博文 交叉熵代价函数 中讲到，二次代价函数在训练ANN时可能会导致训练速度变慢的问题。那就是，初始的输出值离真实值越远，训练速度就越慢。这个问题可以通过采用交叉熵代价函数来解决。其实，这个问题也可以采用另外一种方法解决。那就是采用softmax激活函数，并采用log似然代价函数（log-likelihood cost function）来解决。

log似然代价函数的公式为：

$$C=-\sum_k y_k\log a_k$$

其中，a_k表示第k个神经元的输出值，y_k表示第k个神经元对应的真实值，取值为0或1。

我们来简单理解一下这个代价函数的含义。在ANN中输入一个样本，那么只有一个神经元对应了该样本的正确类别；若这个神经元输出的概率值越高，则按照以上的代价函数公式，其产生的代价就越小；反之，则产生的代价就越高。

为了检验softmax和这个代价函数也可以解决上述所说的训练速度变慢问题，接下来的重点就是推导ANN的权重w和偏置b的梯度公式。以偏置b为例：

$$\begin{aligned}\frac{\partial C}{\partial b_j}&=\frac{\partial C}{\partial z_j}\frac{\partial z_j}{\partial b_j}\\&=\frac{\partial C}{\partial z_j}\cdot\frac{\partial(w_{ji}a_k+b_j)}{\partial b_j}\\&=\frac{\partial}{\partial z_j}\left(-\sum_k y_k\log a_k\right)\\&=-\sum_k y_k\frac{1}{a_k}\cdot\frac{\partial a_k}{\partial z_j}\\&\text{(use softmax derivative)}\end{aligned}$$

$$\begin{aligned}&=-y_j\frac{1}{a_j}\cdot\frac{\partial a_j}{\partial z_j}-\sum_{k\neq j}y_k\frac{1}{a_k}\cdot\frac{\partial a_k}{\partial z_j}\\&=-y_j\frac{1}{a_j}\cdot a_j(1-a_j)-\sum_{k\neq j}y_k\frac{1}{a_k}\cdot(-a_j a_k)\\&=-y_j+y_j a_j+\sum_{k\neq j}y_k a_j\\&=-y_j+a_j\sum_k y_k\\&=a_j-y_j\end{aligned}$$

同理可得：

$$\frac{\partial C}{\partial w_{jk}}=a_k^{L-1}(a_j^L-y_j)$$

从上述梯度公式可知，softmax函数配合log似然代价函数可以很好地训练ANN，不存在学习速度变慢的问题。

2.3-9 softmax understanding

1. hardmax: set max. probability =1, others 0.
while softmax is contrast to hardmax, more gentle mapping from z to these probabilities.

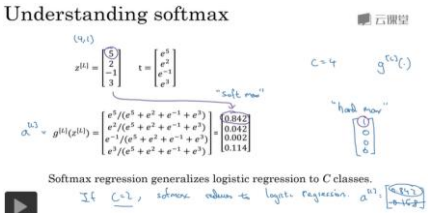
note: two classfication previous talked , activation fucntion is Relu/sigmoid, last layer z[l] is a number (dimension 1), -->out put is also a number -one dimension.

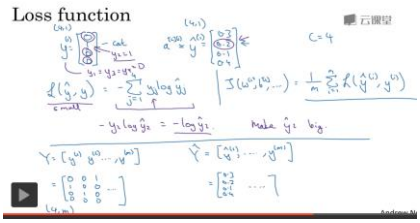
2. Train softmax regression

>1. Loss function singal input:J=sum y[i] * log (y^-[i]) over all classes,
singal example loss: sum of each class loss= label class * algorithm predicted probility for that class.
note: labeld y[i]=1, or 0 (1 means labeld y [i], corresponding to x[i]),-->loss function J=-y[i] * log(y^-[i])
min J-->max log (y^-[i])-->max y^[i]-corresponding
>2. minimize J, means looks at whatever the ground truth calss in training data, and try to make the corresponding probability of that class as high as possible.--a form of maximum likelyhood estimation.

>3. Batch cost function: batch input: J=1/m * sum (J[i])over all m training examples)
minimize batch cost function with gradient descent.

Note: labeled y and prediction matrix dimension is (C, m)

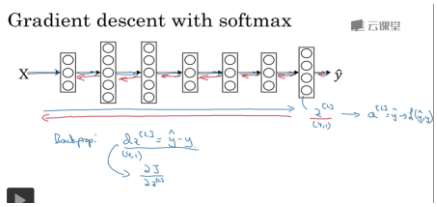




3. Gradient descent with softmax

forward:
Z[L] --->A[L]=y^ --->L(y^, y)

Backward:
see above



Note: when use framework, usually just need to focus on getting the forward prop right, and so long as specify a program framework, the forward prop pass, the program framework will figure out how to do back prop.
(note: when C=2, softmax reduce to sigmoid :
softmax: a_1 = e^z1/ (e^z1+e^z2) = 1/ (1+ e^(z2-z1);
sigmoid: a= 1/ (1+e^-z)
)

2.3_10 Framework_deep learning

Have learned to implement deep learning algorithms more or less from scratch using Python and NumPY, to understand what these deep learning algorithms are really doing.

But when implement more complex models, such as convolutional neural networks or recurring neural networks, or as you start to implement very large models that is increasingly not practical, at least for most people, is not practical to implement everything yourself from scratch. Fortunately, there are now many good deep learning software frameworks that can help you implement these models.

1. Deep learning frameworks
Below are some of the leading deep learning software frameworks. Each of these frameworks has a dedicated user and developer community and I think each of these frameworks is a credible choice for some subset of applications.

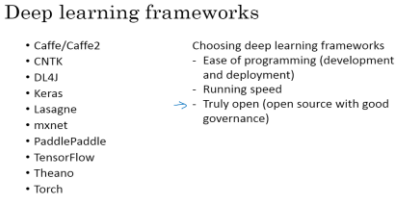
Note: these frameworks are often evolving and getting better month to month,

Criteria choosing frame work
>1. easy of programming: developing neural network and itrating on it , and deploying it for production
>2. running speed:
especially training on large data sets, some frameworks will let you run and train your neural network more efficiently than others.
>3. truly open (open source with good governance): how much you trust a framework will remain open source for a long time, rather than just being under the control of a single company.

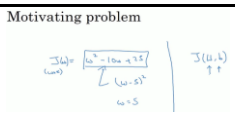
for a framework to be truly open, it needs not only to be open source but it needs good governance as well. Unfortunately, in the software industry some companies have a history of open sourcing software but maintaining single corporation control of the software. And then over some number of years, as people start to use the software, some companies have a history of gradually closing off what was open source, or perhaps moving functionality into their own proprietary cloud services. So one thing I pay a bit of attention to is how much you trust that the framework will remain open source for a long time rather than just being under the control of a single company, which for whatever reason may choose to close it off in the future even if the software is currently released under open source.

>4. But at least in the short term depending on yoru preferences of language, whethe prefer Python or Java or C++ or something else, and depending on what application you're working on, whether this can be division or natural language processing or online advertising or something else, Multiple of these frameworks could be a good choice.

Summarize:
on programming frameworks by providing a higher level of abstraction than just a numerical linear algebra library, any of these program frameworks can make it more efficient as develop machine learning applications.



2.3_11 TensorFlow Practice

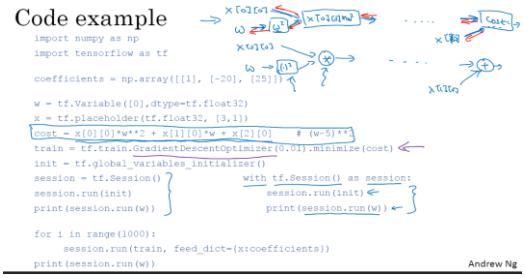


Motivation problem:
e.g: J(w) = w^2 -10w+25

a very similar structure of program can be used to train neural network where can have some complicated cost function J(w, b)--->then could use tensorflow automatically to find parameters that minimize this cost function.

1. Tensorflow coding:
>Define:
>>>1. variable (w),
>>>2. cost function: could use tensorflow add and multiply as well as other functions , tensoflow knows how to compute derivatives of these functions, which already built in to these functions.
-->this is why can only have to implement basically forward prop.

Note: could also use simple way of these functions in cost fuction formula, which are overloaded already in tensoflow.



2. Get training data into Tensorflow program

e.g:
define x (no y involved here): name placeholder, float 32, (3,1) vector;

note: placehoder in TensorFlow is a variable whose value assign later.
convenient way to get traiing data into the cost function (using feed_dict).
when use mini-batch, then on different iterations feed in different subsets of training sets.

-->x = tf.placeholder (tf.float32, [3,1])
cost = x[0][0] *w**2 + x[1][0]*w + x[2][0]

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

train = tf.train. Gradient Descent Optimizer(0.01).minimize (cost)
# could change'GradientDescentOpimizer' to use other method like Adma;
# 0.01 here is learning rate;
#purpose is minimize (cost)

for i in range (1000) # 1000 iterations.
session.run (train, feed_dict=(x: coefficients)) # coefficients value feed to x, which control cost function.

3. Below three lines of code are quite idiomatic in TensorFlow,
>1 .session = tf. Session ---> alternative : with tf.Session () as session: # with command in python is a little bit better at cleaning up in case there's error or exception while executing this inner loop.
>2. seesion. Run (int)
>3. print(session.run(w))

Summarize:
Tensorflow is powerful because all need to do is specify how to compute the cost fucntion, then it can take derivatives, apply a grdient optimizer or an Adam optimizer or some other optimize rwith just pretty much one or two lines of codes.

Cost function computation in Tensorflow:
compute forward acc. to computing graph,
and all necessary backward functions have been built-in.

Thus using the built-in functions to compute the forward function, it can automatically do the backward functions as well to implement back prop through even very complicated functions and compute derivatives.--->make computation more efficient.

Summary

1. Inpute normalizaton-->cost function is more like round bowl -in all paramter direction, fast gradient descent, faster iteration .

2. Batch norm: make each layer output shifting in limitation, each layer more indepenendt to train-->faster algorithm parameter traning.

3. Weight initilization: variance 1/n, or 2/n---->reduce gradient descent vanishing/exploring: by making variance of activation in forwad prop and derivative of state in backprop same (Tanh/sigmoid) or variance of state z in forward prop and derivative of activation in backprop same (Relu activatioin)

Loss function:

Principle: update parameter in big step when error /cost function is big.
Better: dw = (y^-y)*c (c: nothing to do with y^,y)
-->
1. choose cost functioin:
>>1. square error:
>>> ok for linear function: dw = (y^-y)*X.T
>>> non-ok for logistic function: dw = (y^-y) * [y^(1-y^)] * X.T-->Can not gurantte dw bigger when has worse prediction (y^-y) bigger

>>2. cross-error:
>>> for logistic function: dw = (y^-y) . X.T

>>3. softmax error:
>>> loss = log y^_i: corresponding to max probabilty: maximum likelyhood estimation

2. Choose activation function for output: linear , sigmoid, softmax

Normalizati

BatchNormalization、LayerNormalization、InstanceNorm、(

1、综述

1.1 论文链接

1、Batch Normalization
https://arxiv.org/pdf/1502.03167.pdf
2、Layer Normalizaiton
https://arxiv.org/pdf/1607.06450v1.pdf
3、Instance Normalization
https://arxiv.org/pdf/1607.08022.pdf
https://github.com/DmitryUlyanov/texture_nets

1.2 介绍

归一化层，目前主要有这几个方法，Batch Normalization（2015年）、Layer Normalization（2016年）、Instance Normalization（2017年）、Group Normalization（2018年）、Switchable Normalization（2018年）；

将输入的图像shape记为[N, C, H, W]，这几个方法主要的区别就是在，

- batchNorm是在batch上，对NHW做归一化，对小batchsize效果不好；
- layerNorm在通道方向上，对CHW归一化，主要对RNN作用明显；
- instanceNorm在图像像素上，对HW做归一化，用在风格化迁移；
- GroupNorm将channel分组，然后再做归一化；
- SwitchableNorm是将BN、LN、IN结合，赋予权重，让网络自己去学习归一化层应该使用什么方法。

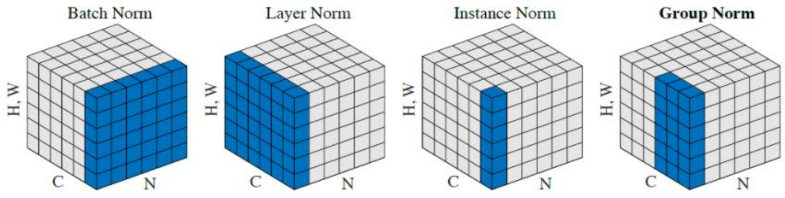


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

3、Layer Normalizaiton

batch normalization存在以下缺点：

- 对batchsize的大小比较敏感，由于每次计算均值和方差是在一个batch上，所以如果batchsize太小，则计算的均值、方差不足以代表整个数据分布；
- BN实际使用时需要计算并且保存某一层神经网络batch的均值和方差等统计信息，对于对一个固定深度的前向神经网络（DNN，CNN）使用BN，很方便；但对于RNN来说，sequence的长度是不一致的，换句话说RNN的深度不是固定的，不同的time-step需要保存不同的statics特征，可能存在一个特殊sequence比其他sequence长很多，这样training时，计算很麻烦。（参考于 https://blog.csdn.net/lqfarmer/article/details/71439314）

与BN不同，LN是针对深度网络的某一层的所有神经元的输入按以下公式进行normalize操作。

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

LN用于RNN效果比较明显，但是在CNN上，不如BN。

```
def ln(x, b, s):
    _eps = 1e-5
    output = (x - x.mean(1)[:,None]) / tensor.sqrt((x.var(1)[:,None] + _eps))
    output = s[None, :] * output + b[None,:]
    return output
```

用在四维图像上，

```
def Layernorm(x, gamma, beta):

    # x_shape:[B, C, H, W]
    results = 0.
    eps = 1e-5

    x_mean = np.mean(x, axis=(1, 2, 3), keepdims=True)
    x_var = np.var(x, axis=(1, 2, 3), keepdims=True0)
    x_normalized = (x - x_mean) / np.sqrt(x_var + eps)
    results = gamma * x_normalized + beta
    return results
```

4、Instance Normalization

BN注重对每个batch进行归一化，保证数据分布一致，因为判别模型中结果取决于数据整体分布。

但是图像风格化中，生成结果主要依赖于某个图像实例，所以对整个batch归一化不适合图像风格化中，因而对HW做归一化。可以加速模型收敛，并且保持每个图像实例之间的独立。

公式：

-

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}, \quad \mu_{ti} = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H} x_{tilm}, \quad \sigma_{ti}^2 = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H}(x_{tilm} - mu_{ti})^2.$$

代码：

```
def Instancenorm(x, gamma, beta):

    # x_shape:[B, C, H, W]
    results = 0.
    eps = 1e-5

    x_mean = np.mean(x, axis=(2, 3), keepdims=True)
    x_var = np.var(x, axis=(2, 3), keepdims=True0)
    x_normalized = (x - x_mean) / np.sqrt(x_var + eps)
    results = gamma * x_normalized + beta
    return results
```

6、Switchable Normalization

本篇论文作者认为，

- 第一，归一化虽然提高模型泛化能力，然而归一化层的操作是人工设计的。在实际应用中，解决不同的问题原则上需要设计不同的归一化操作，并没有一个通用的归一化方法能够解决所有应用问题；
- 第二，一个深度神经网络往往包含几十个归一化层，通常这些归一化层都使用同样的归一化操作，因为手工为每一个归一化层设计操作需要进行大量的实验。

因此作者提出自适配归一化方法——Switchable Normalization（SN）来解决上述问题。与强化学习不同，SN使用可微分学习，为一个深度网络中的每一个归一化层确定合适的归一化操作。

公式:

SN has an intuitive expression

$$\hat{h}_{ncij} = \gamma \frac{h_{ncij} - \Sigma_{k\in\Omega} w_k \mu_k}{\sqrt{\Sigma_{k\in\Omega} w'_k \sigma_k^2 + \epsilon}} + \beta, \qquad (3)$$

$$w_k = \frac{e^{\lambda_k}}{\Sigma_z e^{\lambda_z}}, \quad k \in \{\text{in}, \text{ln}, \text{bn}\},$$

$\omega_z \in \{\text{in}, \text{ln}, \text{bn}\}$

$$\mu_{\text{in}} = \frac{1}{HW}\sum_{h,w}^{H,W} h_{ncij},\ \sigma^2_{\text{in}} = \frac{1}{HW}\sum_{h,w}^{H,W}(h_{ncij} - \mu_{\text{in}})^2,$$

$$\mu_{\text{ln}} = \frac{1}{C}\sum_{c=1}^{C}\mu_{\text{in}},\ \sigma^2_{\text{ln}} = \frac{1}{C}\sum_{c=1}^{C}(\sigma^2_{\text{in}} + \mu^2_{\text{in}}) - \mu^2_{\text{ln}},$$

$$\mu_{\text{bn}} = \frac{1}{N}\sum_{n=1}^{N}\mu_{\text{in}},\ \sigma^2_{\text{bn}} = \frac{1}{N}\sum_{n=1}^{N}(\sigma^2_{\text{in}} + \mu^2_{\text{in}}) - \mu^2_{\text{bn}}, \quad (5)$$

代码：

```
def SwitchableNorm(x, gamma, beta, w_mean, w_var):
    # x_shape:[B, C, H, W]
    results = 0.
    eps = 1e-5

```

```
    mean_in = np.mean(x, axis=(2, 3), keepdims=True)
    var_in = np.var(x, axis=(2, 3), keepdims=True)

    mean_ln = np.mean(x, axis=(1, 2, 3), keepdims=True)
    var_ln = np.var(x, axis=(1, 2, 3), keepdims=True)

    mean_bn = np.mean(x, axis=(0, 2, 3), keepdims=True)
    var_bn = np.var(x, axis=(0, 2, 3), keepdims=True)

    mean = w_mean[0] * mean_in + w_mean[1] * mean_ln + w_mean[2] * mean_bn
    var = w_var[0] * var_in + w_var[1] * var_ln + w_var[2] * var_bn

    x_normalized = (x - mean) / np.sqrt(var + eps)
    results = gamma * x_normalized + beta
    return results
```

ion method

GroupNorm、SwitchableNorm总结

4、Group Normalization

https://arxiv.org/pdf/1803.08494.pdf

5、Switchable Normalization

https://arxiv.org/pdf/1806.10779.pdf

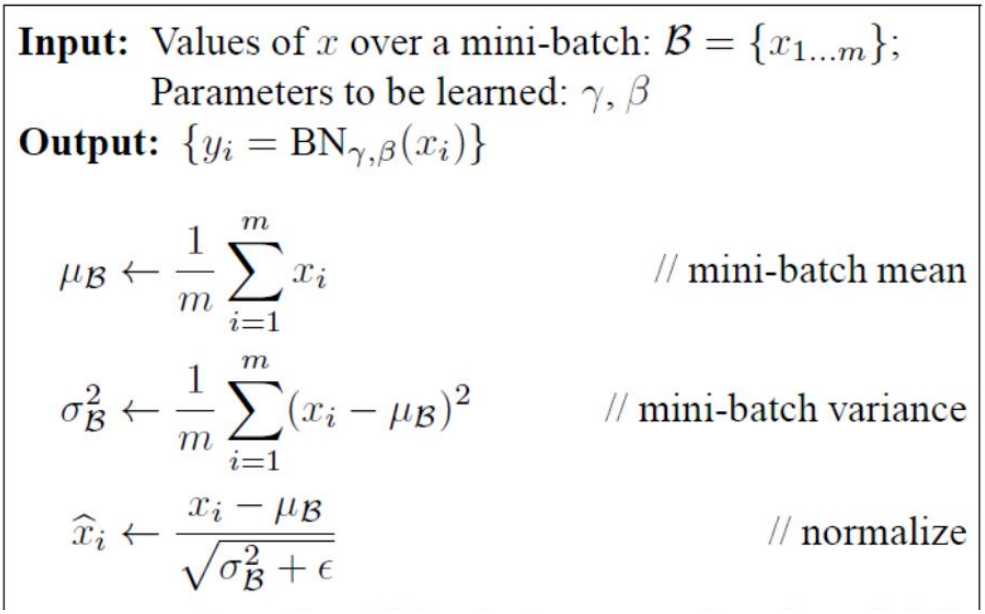
https://github.com/switchablenorms/Switchable-Normalization

2、Batch Normalization

首先，在进行训练之前，一般要对数据做归一化，使其分布一致，但是在深度神经网络训练过程中，通常以送入网络的每一个batch训练，这样每个batch具有不同的分布；此外，为了解决internal covarivate shift问题，这个问题定义是随着batch normalizaiton这篇论文提出的，在训练过程中，数据分布会发生变化，对下一层网络的学习带来困难。

所以batch normalization就是强行将数据拉回到均值为0，方差为1的正太分布上，这样不仅数据分布一致，而且避免发生梯度消失。

此外，internal corvariate shift和covariate shift是两回事，前者是网络内部，后者是针对输入数据，比如我们在训练数据前做归一化等预处理操作。



$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

算法过程：

- 沿着通道计算每个batch的均值u
- 沿着通道计算每个batch的方差σ^2
- 对x做归一化，x'=(x-u)/开根号(σ^2+ε)
- 加入缩放和平移变量γ和β ,归一化后的值，y=γx'+β

加入缩放平移变量的原因是：保证每一次数据经过归一化后还保留原有学习来的特征，同时又能完成归一化操作，加速训练。 这两个参数是用来学习的参数。

```python
import numpy as np

def Batchnorm(x, gamma, beta, bn_param):

    # x_shape:[B, C, H, W]
    running_mean = bn_param['running_mean']
    running_var = bn_param['running_var']
    results = 0.
    eps = 1e-5

    x_mean = np.mean(x, axis=(0, 2, 3), keepdims=True)
    x_var = np.var(x, axis=(0, 2, 3), keepdims=True0)
    x_normalized = (x - x_mean) / np.sqrt(x_var + eps)
    results = gamma * x_normalized + beta

    # 因为在测试时是单个图片测试，这里保留训练时的均值和方差，用在后面测试时用
    running_mean = momentum * running_mean + (1 - momentum) * x_mean
    running_var = momentum * running_var + (1 - momentum) * x_var

    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var

    return results, bn_param
```

5、Group Normalization

主要是针对Batch Normalization对小batchsize效果差，GN将channel方向分group，然后每个group内做归一化，算(C//G)*H*W的均值，这样与batchsize无关，不受其约束。

公式：

Group Norm. Formally, a Group Norm layer computes μ and σ in a set $\mathcal{S}_i$ defined as:

$$\mathcal{S}_i = \{k \mid k_N = i_N, \lfloor \frac{k_C}{C/G} \rfloor = \lfloor \frac{i_C}{C/G} \rfloor\}. \quad (7)$$

Here G is the number of groups, which is a pre-defined hyper-parameter ($G = 32$ by default). C/G is the number of channels per group. $\lfloor \cdot \rfloor$ is the floor operation, and "$\lfloor \frac{k_C}{C/G} \rfloor = \lfloor \frac{i_C}{C/G} \rfloor$" means that the indexes i and k are in the same group of channels, assuming each group of channels are stored in a sequential order along the C axis. GN computes μ and σ along the (H, W) axes and along a group of $\frac{C}{G}$ channels. The computation of GN is illustrated in

Figure 2 (rightmost), which is a simple case of 2 groups ($G = 2$) each having 3 channels.

代码：

```
def GroupNorm(x, gamma, beta, G=16):

    # x_shape:[B, C, H, W]
    results = 0.
    eps = 1e-5
    x = np.reshape(x, (x.shape[0], G, x.shape[1]/16, x.shape[2], x.shape[3]))

    x_mean = np.mean(x, axis=(2, 3, 4), keepdims=True)
    x_var = np.var(x, axis=(2, 3, 4), keepdims=True0)
    x_normalized = (x - x_mean) / np.sqrt(x_var + eps)
    results = gamma * x_normalized + beta
    return results
```

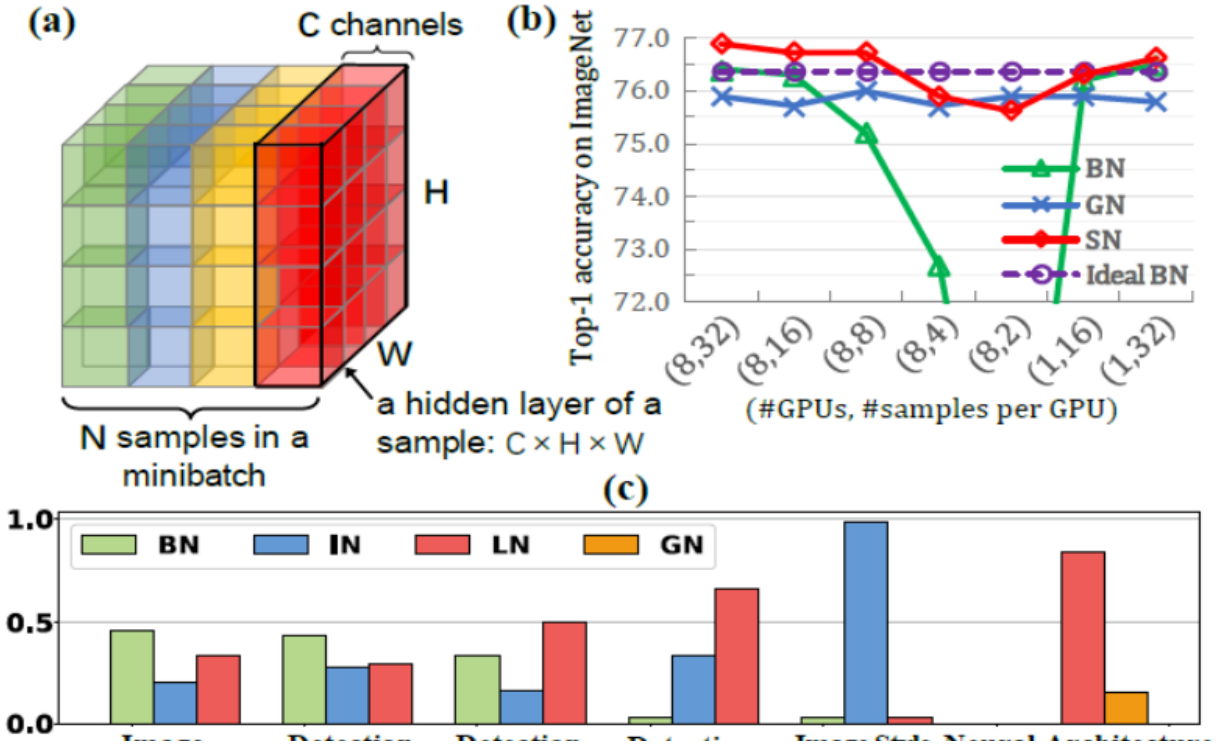
提出的原因：在小 batch 的样本中，Batch Normalization 估计的值不准。一般用在很大的模型中，这时 batch size 就很小。

思路：数据不够，通道来凑。 每个样本的特征分为几组，每组特征分别计算均值和方差。可以看作是 Layer Normalization 的基础上添加了特征分组。

注意事项：

- 不再有 running_mean 和 running_var
- γ 和 β 为逐通道的

结果比较：


(a)
C channels
H
W
N samples in a minibatch
a hidden layer of a sample: C × H × W
(b)
Top-1 accuracy on ImageNet
77.0
76.0
75.0
74.0
73.0
72.0
BN
GN
SN
Ideal BN
(8,32)
(8,16)
(8,8)
(8,4)
(8,2)
(1,16)
(1,32)
(#GPUs, #samples per GPU)
(c)
1.0
0.5
0.0
BN
IN
LN
GN

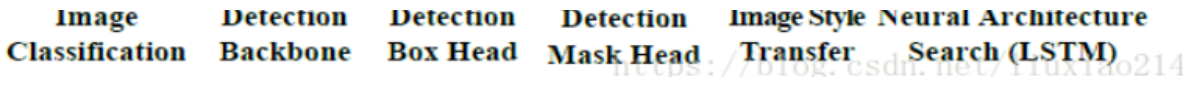


Figure 1: **(a)** In a CNN, the feature maps of a hidden convolutional layer is of size $N \times C \times H \times W$, which represents N samples in a minibatch, C channels of a layer, height and width of a channel respectively. We have $N = 4$ in (a) as an illustrative example. Different normalization methods compute statistics (means and variances) along different axes of N, C, H, W. **(b)** The top-1 accuracies of ResNet50 trained with SN on ImageNet are compared with BN and GN in different batch sizes. For instance, all methods are compared to an ideal case, 'ideal BN', whose accuracies are 76.4%, which is the result of (8,32) of BN. This ideal case isn't obtainable in practice. In fact, when the minibatch size decreases, BN's accuracies drop significantly, while SN and GN both maintain reasonably good performance. SN surpasses or is comparable to both BN and GN in all settings. **(c)** shows that SN adapts to various network architectures and tasks, by learning an importance weight to select each normalization method (*i.e.* a

ratio between 0 and 1; all the weights of each task sum to 1).

【深度学习】参数初始化

一、参数初始化原则

参数初始化作为模型训练的起点，决定了模型训练的初始位置。选择的好坏很大程度影响收敛的速度与最终的训练结果。一般来说，参数初始化遵循以下几点要求：

- 不建议同时初始化为一个值，容易出现“对称失效”
- 最好保证均值为0，正负交错，参数大致数量相等
- 初始化参数不要太大或太小，前者会导致梯度发散难以训练，后者会导致特征对后面的影响逐渐变小

二、常见的初始化方法

1、全零初始化与随机初始化

如果神经元的权重被初始化为0， 在第一次更新的时候，除了输出之外，所有的中间层的节点的值都为零。一般神经网络拥有对称的结构，那么在进行第一次误差反向传播时，更新后的网络参数将会相同，在下一次更新时，相同的网络参数学习提取不到有用的特征，因此深度学习模型都不会使用0初始化所有参数。
随机初始化顾名思义，一般值大了容易饱和，小了对后面的影响太小，激活函数的效果也不好。

3、Kaiming初始化

Kaiming初始化，也称之为he初始化。是残差网络的作者何凯明在这篇论文中提出了ReLU网络的初始化方法。

（1）正态化的kaiming初始化——he_normal

它从以 0 为中心，标准差为 stddev = sqrt(2 / fan_in) 的截断正态分布中抽取样本， 其中 fan_in是权值张量中的输入单位的数量

（2）标准化化的kaiming初始化——he_uniform

它从 [-limit，limit] 中的均匀分布中抽取样本， 其中 limit 是 sqrt(6 / fan_in)， 其中 fan_in 是权值张量中的输入单位的数量

4、lecun初始化

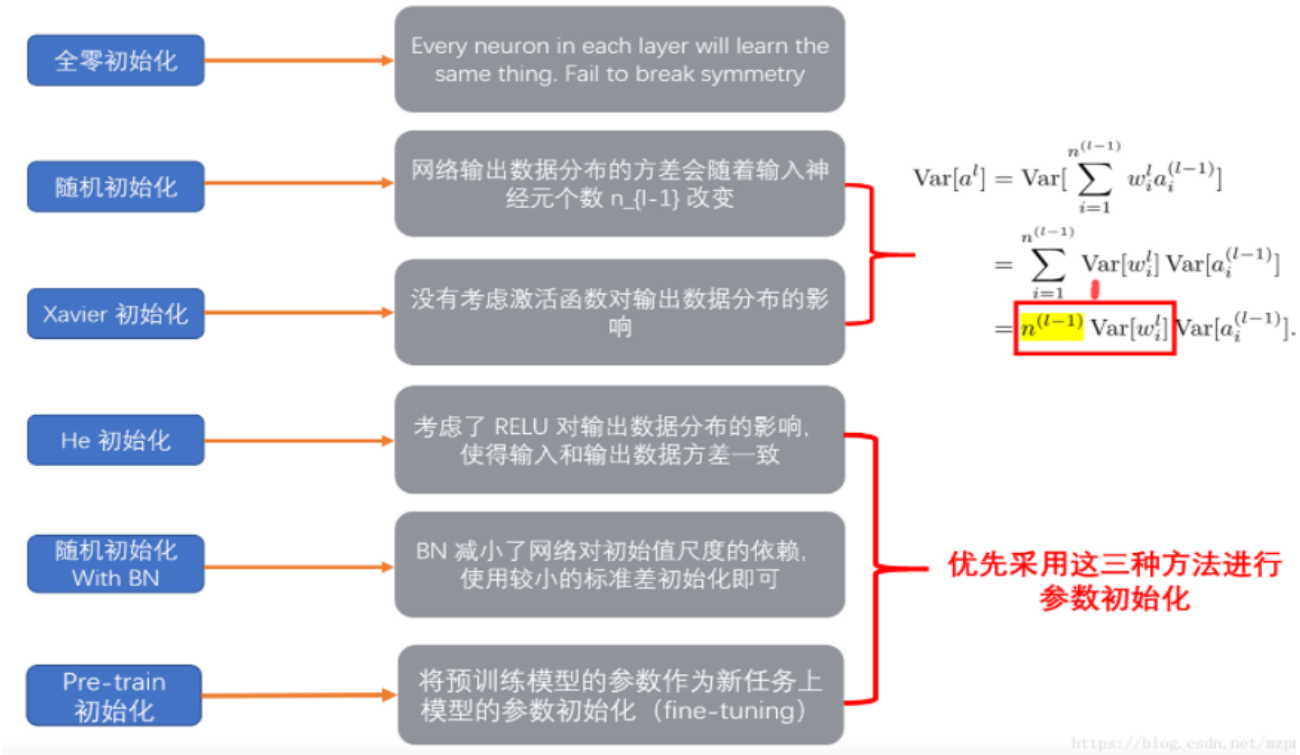
（1）正态化的lecun初始化——lecun_normal

它从以 0 为中心，标准差为 stddev = sqrt(1 / fan_in) 的截断正态分布中抽取样本， 其中 fan_in是权值张量中的输入单位的数量。

（2）标准化的lecun初始化——lecun_uniform

它从 [-limit，limit] 中的均匀分布中抽取样本， 其中 limit 是 sqrt(3 / fan_in)， fan_in 是权值张量中的输入单位的数量。

四、初始化选择建议


全零初始化
Every neuron in each layer will learn the same thing. Fail to break symmetry
随机初始化
网络输出数据分布的方差会随着输入神经元个数 n_{l-1} 改变
Xavier 初始化
没有考虑激活函数对输出数据分布的影响
$\mathrm{Var}[a^l] = \mathrm{Var}[\sum_{i=1}^{n^{(l-1)}} w_i^l a_i^{(l-1)}]$
$= \sum_{i=1}^{n^{(l-1)}} \mathrm{Var}[w_i^l]\,\mathrm{Var}[a_i^{(l-1)}]$
$= n^{(l-1)}\,\mathrm{Var}[w_i^l]\,\mathrm{Var}[a_i^{(l-1)}].$
He 初始化
考虑了 RELU 对输出数据分布的影响，使得输入和输出数据方差一致
随机初始化 With BN
BN 减小了网络对初始值尺度的依赖，使用较小的标准差初始化即可
Pre-train 初始化
将预训练模型的参数作为新任务上模型的参数初始化（fine-tuning）
优先采用这三种方法进行参数初始化

化与Batch Normalization

2、Glorot初始化方法

（1）正态化的Glorot初始化——glorot_normal

Glorot 正态分布初始化器，也称为 Xavier 正态分布初始化器。它从以 0 为中心，标准差为 stddev = sqrt(2 / (fan_in + fan_out)) 的截断正态分布中抽取样本。其中 fan_in 是权值张量中的输入单位的数量， fan_out 是权值张量中的输出单位的数量。

（2）标准化的Glorot初始化——glorot_uniform

Glorot 均匀分布初始化器，也称为 Xavier 均匀分布初始化器。

它从 [-limit，limit] 中的均匀分布中抽取样本， 其中 limit 是 sqrt(6 / (fan_in + fan_out))， fan_in 是权值张量中的输入单位的数量， fan_out 是权值张量中的输出单位的数量。

（3）Glorot初始化器的缺点

首先有一个共识必须先提出：神经网络如果保持每层的信息流动是同一方差，那么会更加有利于优化。不然大家也不会去争先恐后地研究各种normalization方法。不过，Xavier Glorot认为还不够，应该增强这个条件，好的初始化应该使得各层的激活值和梯度的方差在传播过程中保持一致，这个被称为Glorot条件。如果反向传播每层梯度保持近似的方差，则信息能反馈到各层。而前向传播激活值方差近似相等，有利于平稳地学习。当然为了做到这一点，对激活函数也必须作出一些约定。

(1) 激活函数是线性的，至少在0点附近，而且导数为1。

(2) 激活值关于0对称。

这两个都不适用于sigmoid函数和ReLU函数，而适合tanh函数。

三、Batch Normalization

随着网络层数的增加，分布逐渐发生偏移，之所以收敛慢，是因为整体分布往非线性函数取值区间的上下限靠近。这会导致反向传播时梯度消失。BN就是通过规范化的手段，把每层神经网络任意神经元这个输入值的分布强行拉回到均值0方差1的标准正态分布，使得激活输入值落入非线性函数中比较敏感的区域。可以让梯度变大，学习收敛速度快，能大大加快收敛速度。

这张图已经能很好地反映初始化的一些思路了，由于现在batch norm比较流行所以可能对初始化要求不是那么高，一般用relu的话使用he初始化会好一些，或者直接用norm batch。
使用 RELU（without BN） 激活函数时，最好选用 He 初始化方法，将参数初始化为服从高斯分布或者均匀分布的较小随机数。
使用 BN 时，减少了网络对参数初始值尺度的依赖，此时使用较小的标准差(eg：0.01)进行初始化即可，初始化方法没有那么重要了。

另外关于ReLU和BN的顺序，原paper建议将BN层放置在ReLU前，因为ReLU激活函数的输出非负，不能近似为高斯分布。实验表明，放在前后的差异似乎不大，甚至放在ReLU后还好一些。使用梯度下降法对参数进行优化学习时，非常关键的一个问题是如何合理地初始化参数值，为此提出了Xavier初始化和He初始化等方法，而使用BN操作后，对于参数初始化的要求就没那么高了。而且也可以使用大的学习率，另外BN也能够有效地缓解梯度消失的问题。