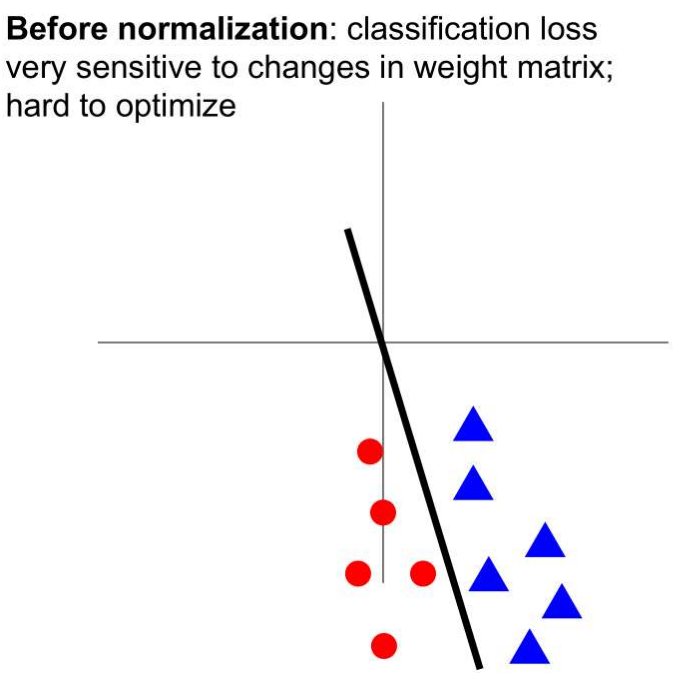
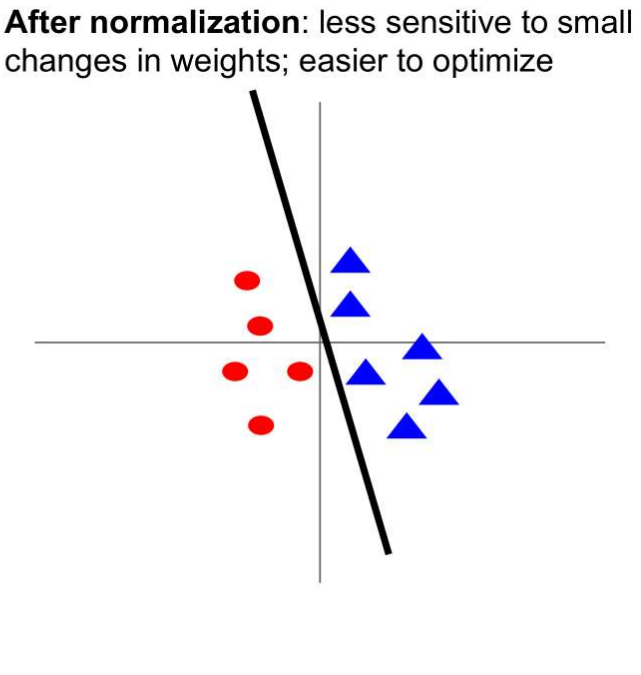
Lecture 7 Training Neutral Network, Part 2

一、关于Data processing的一些补充





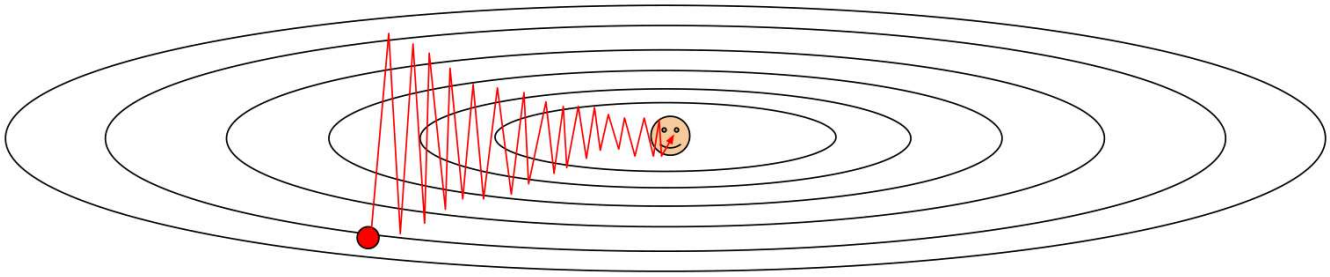
上面两张图分别是经过标准化和未经过标准化的数据的图，其实我这里讲的主要是为什么需要去做data processing。

可以很明显的认识到：对于第一个未经过标准化的数据来说，分割线的一点小小的改变都会影响分类的精度，这是因为数据的平均值原理原点的原因；对于经过标准化的数据来说，这个影响就会小很多，因为数据中心在零点，允许的空间就会稍微大一些。

二、Fancier Optimization (针对SGD的梯度选择方式的优化) —— 提升网络在训练中的表现

1. SGD的问题：

首先，SGD在二维的梯度下降中可能会出现这样的问题：



这个问题其实可以用玉米夹饼的比喻来解释。就是他可能不会沿着最近的方向到达loss最小的地方，而是会荡来荡去的减小，这个问题主要是来自于梯度下降过快导致的某一维变量轻微的减小就会导致loss下降方向的巨大偏差。

其次是关于局部最小值和鞍点的问题：

在局部最小值的点，会出现梯度为0的情况，这在**一维**的问题中非常危险，因为这种现象并不是很不常见，而是非常常见。这样的点带来的直接问题就是在该点的梯度为0，参数的优化也就随之停止，这样相当于是达到了一个局部最优值，但显然没有办法达到全局最优。

当然这个情况在**多维**的条件下就不是很常见的，就变成一个很少见的问题。原因是，很难达到一种情况就是超级多的参数在某一时刻全部为局部最优解，梯度为0。这种情况不真实。所以我们在多维的情况下只会去考虑**鞍点**。

鞍点是梯度相当小的情况，但不存在局部最优。这种情况在多维下还是比较常见的。当他出现鞍点的时候，当很多个参数同时出现梯度较小的时候，就会出现拟合相当慢的情况，这样就会大大降低我们的训练效率。

还有一个关于minibatch导致的噪声问题

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$

因为在SGD中，基本都是通过设置minibatch来计算梯度的，因为如果整体计算梯度的代价太过于巨大。但是这也导致了一个比较严重的问题，就是**噪声**问题。小样本的采样必然存在着坏点对于结果的影响，这个是毋庸置疑的。

2. SGD + Momentum (随机梯度下降法的动量改进)

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

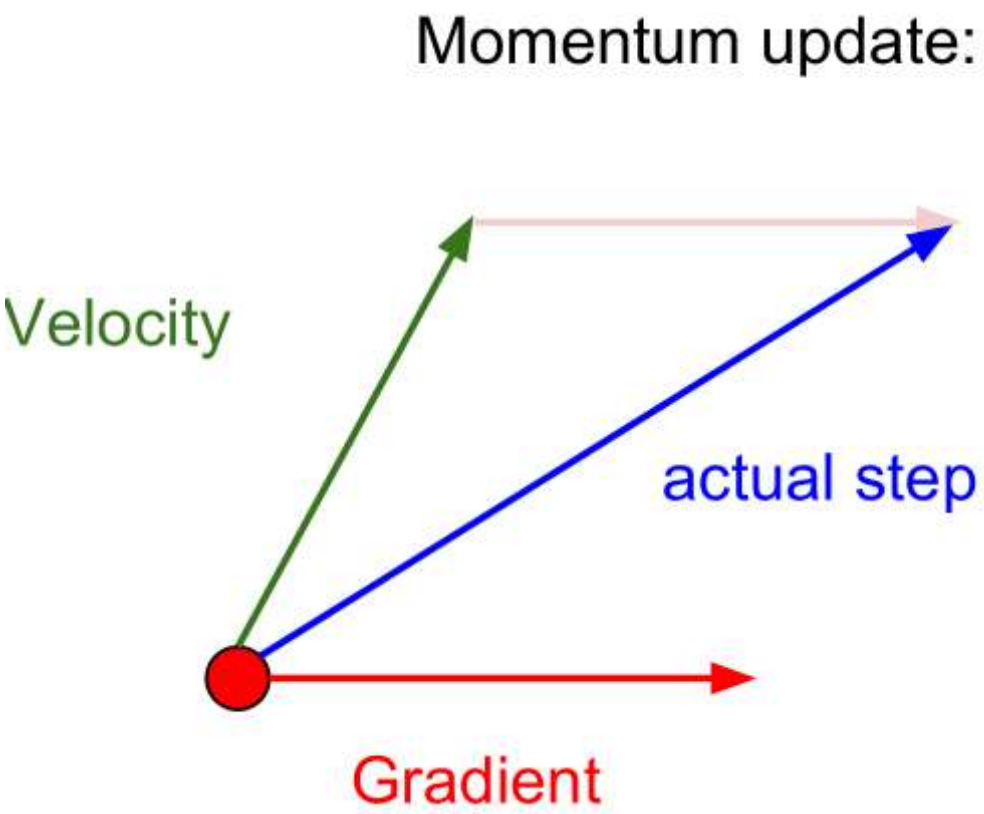
SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

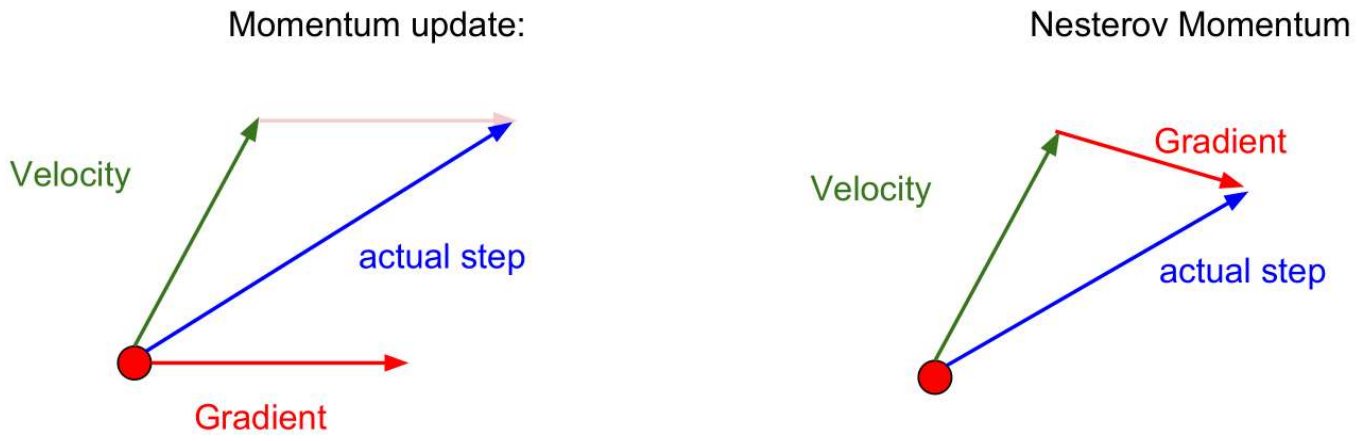
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

我们在SGD中加入偏移量**Momentum**，使得参数下降的方向与原随机梯度下降的方向有所偏差。这种偏差源于Momentum量对于之前动量的积累，目的就是为了不让梯度方向作为当前时刻参数改变的唯一标准。



3. Nesterov Momentum

这个方法是对前面方法的一种改进，改进的方式其实可以用物理学的常识来理解。两种方法的对比如下图：



可以很明显的看出两种方法的差距 —— 其实就是两种方法取梯度的点不同。在实际应用中，后一种的拟合效率要比前一种好很多，原因不知为何，但是效果摆在这里。

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

3. AdaGrad

这个算法其实是为了放缩梯度而设计的。他的主要思想可以看下面的公式解释，其实比较好理解，就是对于梯度的平方做一个累积，这样梯度越大的方向参数的更新也就相对变慢，梯度较小的方向参数的更新相对于前者就变快，最终达到一个稳定的值。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

累积平方梯度：$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

计算更新：$\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \boldsymbol{g}$ （逐元素地应用除和求平方根）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

https://blog.csdn.net/program_developer

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

4. RMSProp

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

与之前的AdaGrad唯一的不同就在于他并不是简单的将梯度做累加，而是做一个衰减的累加。但是这可能会出现训练的速度不停的减慢的情况，这并不是我们想要的。

5. Adam(almost)

有点像是带动量的AdaGrad，结合了动量和AdaGrad的优点。

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Problem：在一开始的时候，如果我们将second_momentum初始化成0，会出现什么样的结果?

显然就是在一开始的timestep中，second_momentum是0，那么除以一个很小的数步长就超级大了。

6. Adam(full form) —— 最常用

对上一个的初始化问题做了修改，这样就再也不怕啦

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

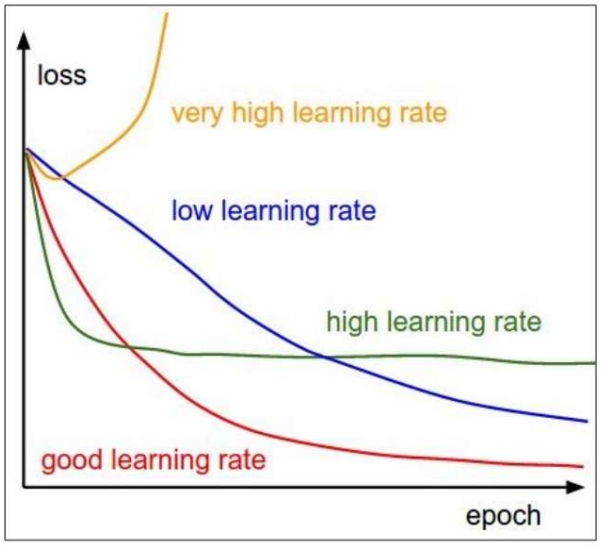
Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9,
beta2 = 0.999, and learning_rate = 1e-3 or 5e-4
is a great starting point for many models!

7. Learning Rate Decay (学习率的衰减方法)



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

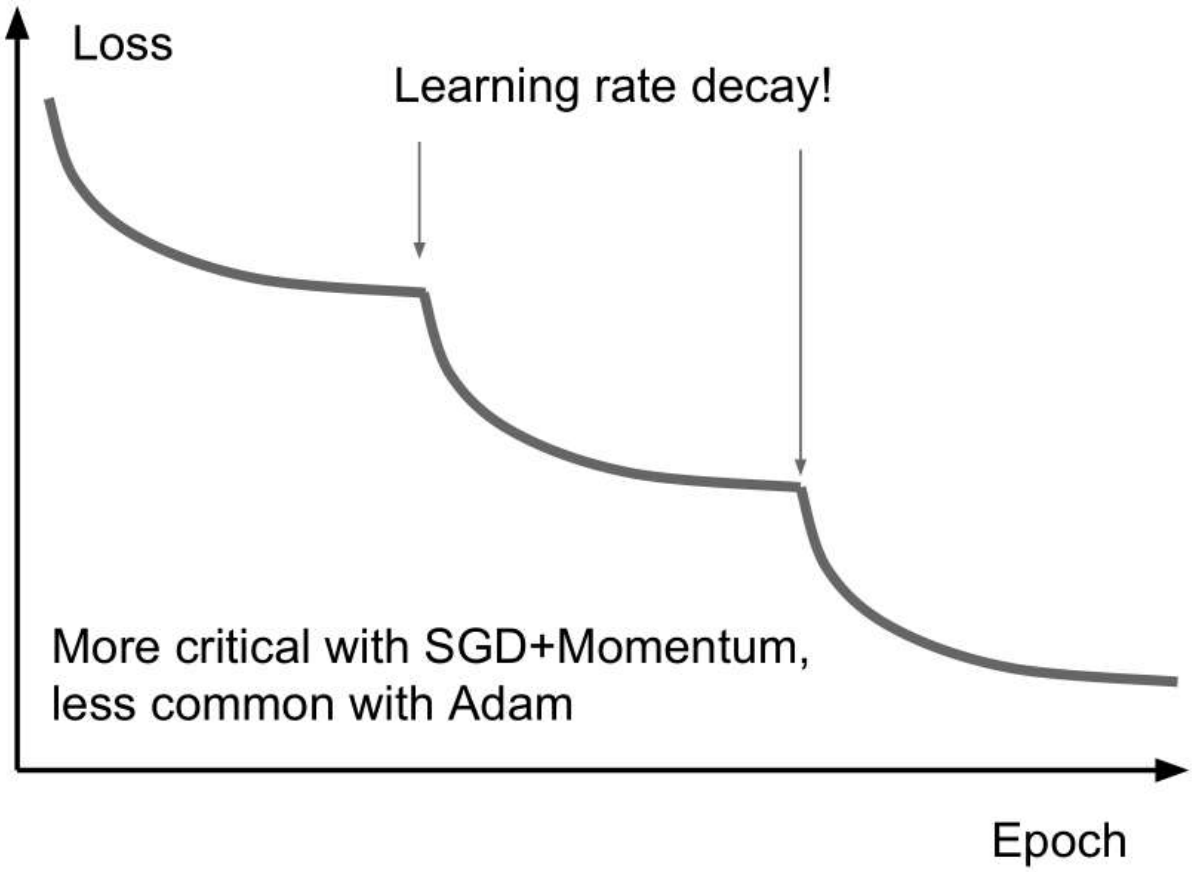
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

衰减主要有三种方式：

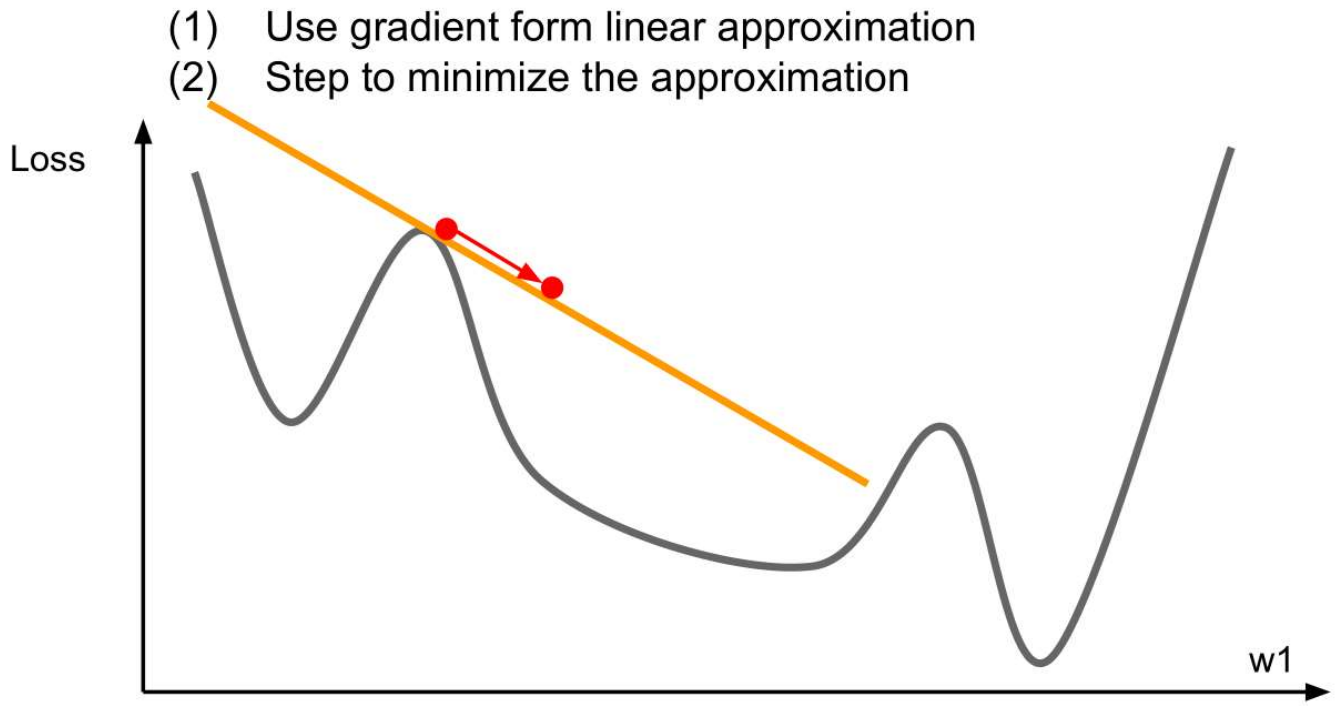
- 到了一定的迭代次数之后衰减



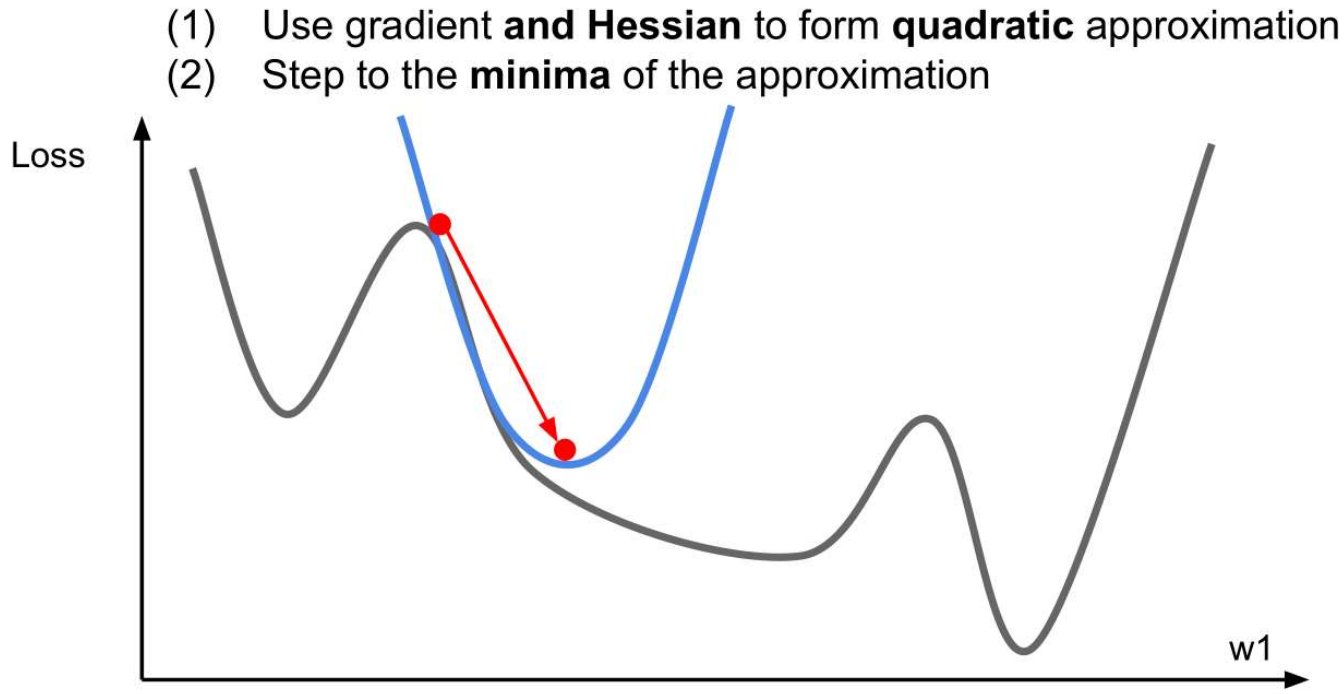
- 指数型持续衰减
- 根据时间的衰减

8. First-order and Second-order Optimization (一阶和二阶逼近优化)

对于一阶逼近优化，其实还是比较好理解的，就是对实际loss函数不断做一阶的逼近，然后按照逼近的结果进行参数的调整就好了，如下图，直接按照直线的趋势走就可以。



对于二阶逼近优化，情况其实差不多，他的做法就是用一阶的梯度和二阶的Hessian矩阵来获得二阶泰勒loss的逼近，这样就可以直接奔向二次函数的最低点以达到取最小的目的。



second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

上面是二阶泰勒表达式，可以看出，在表达式中没有超参，更没有学习率，并且用Hessian矩阵的话他的数据量是参数量的平方，也就意味着，如果有一亿个参数，就需要进行一亿平方次计算，这样的计算代价太过于巨大，所以就选择用下面的优化方法。

Second-Order Optimization

$$\theta^* = \theta_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

L-BFGS

- **Usually works very well in full batch, deterministic mode**
  i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

9. In Practice

- **Adam** is a good default choice in most cases
- If you can afford to do full batch updates then try out **L-BFGS** (and don’ t forget to disable all sources of noise)

三、Regularization (正则化) —— Dropout —— 提升模型在测试集上的表现

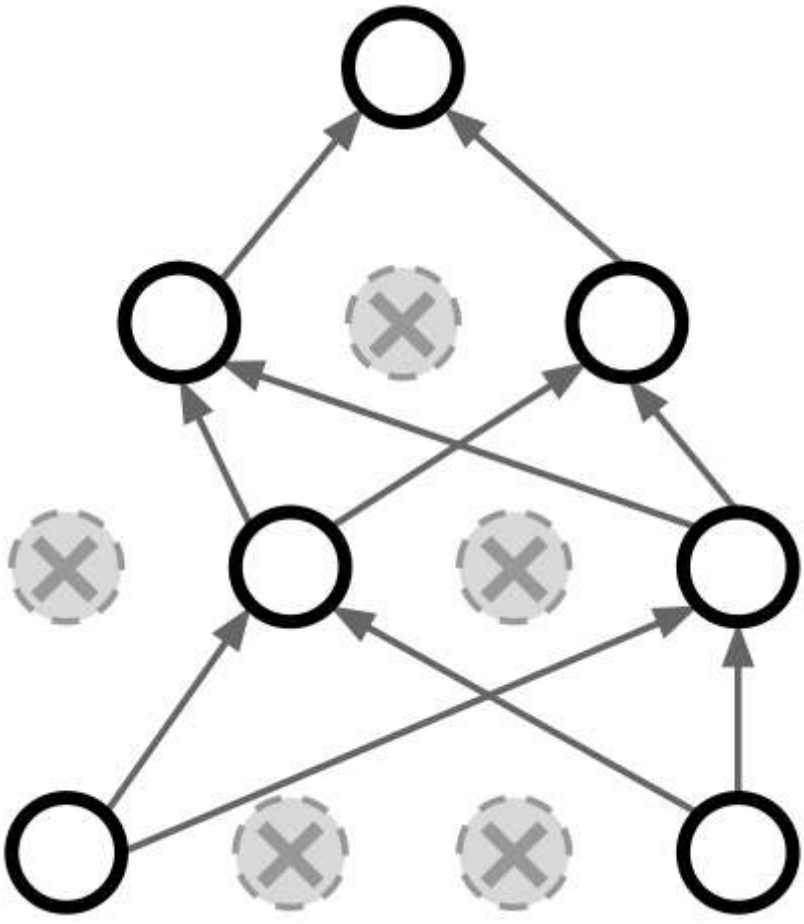
正则化其实我们在之前有说过，也做过，之前做的基本是L2的正则化，不知所云的正则，不知道为啥这样就可以正则。但是，接下来的dropout就是真的正则了。

1. 做法

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

上面这段代码是真的无敌，就直接实现了两层神经网络的dropout，悄无声息的。

其实他的做法也很好理解，对于U1和U2两行代码，其实就是产生一个同样大小的矩阵，矩阵的元素是0-1的值，因为我们要选择一半的神经元死掉，那么我们就将这些元素与0.5比，大的就赋值1，小的就赋值0，这样再与原矩阵相乘，就可以得到最终的一半神经元死掉的神经网络，也就是下图：

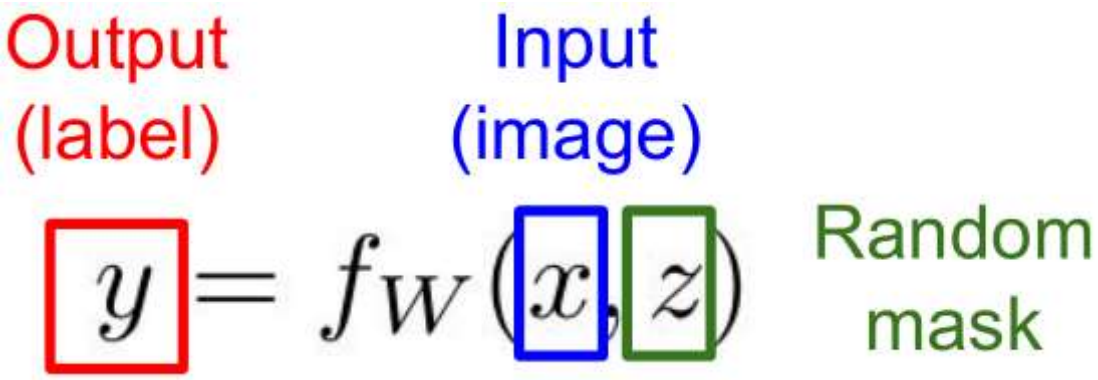


2. 可正则化原因 —— 主要是为了解决过拟合

- **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。
- **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。
- **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。

3. Test Time (predict阶段的问题)

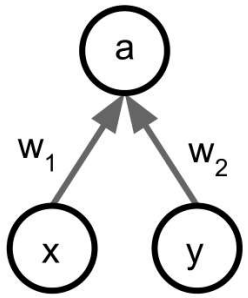
在使用了Dropout之后，我们会发现，结果的表达式发生了变化：



其中包含了一个random量，也就是z。这就使得predict的结果存在明显的随意性，不能保证其结果与训练时的参数有很好的统一性，这在数学期望上可以看出来：

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E[a] = w_1 x + w_2 y$

During training we have:

$$\begin{aligned} E[a] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

At test time, **multiply** by dropout probability

At test time all neurons are active always => We must scale the activations so that for each neuron: output at test time = expected output at training time

用代码来实现整个流程就如下图：

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Dropout Summary

drop in forward pass

scale at test time

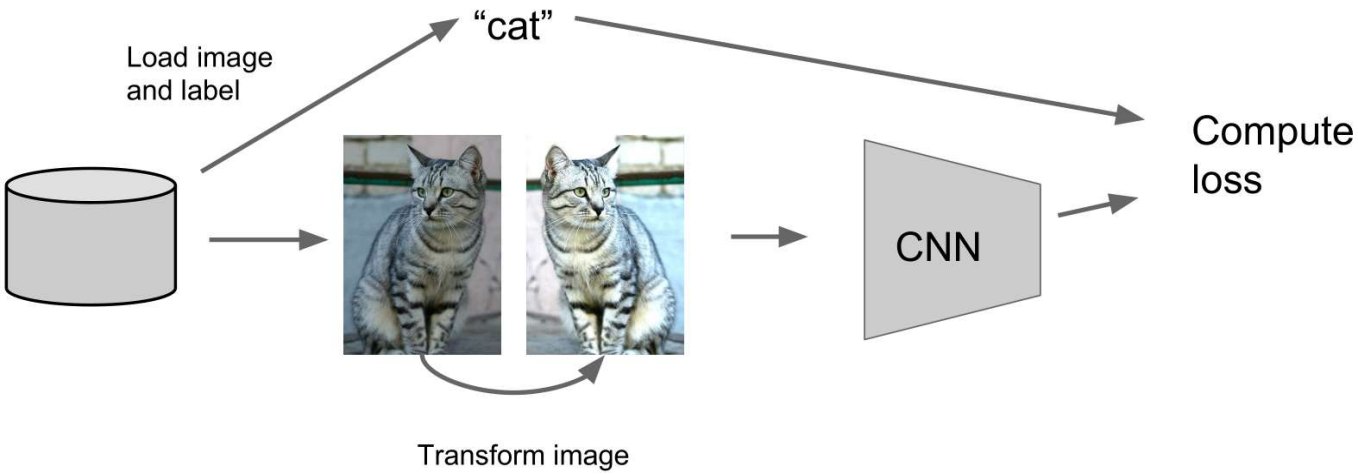
其实改一下代码也可以这样做：

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

四、Regularization (正则化) —— Data Augumentation (数据增强)

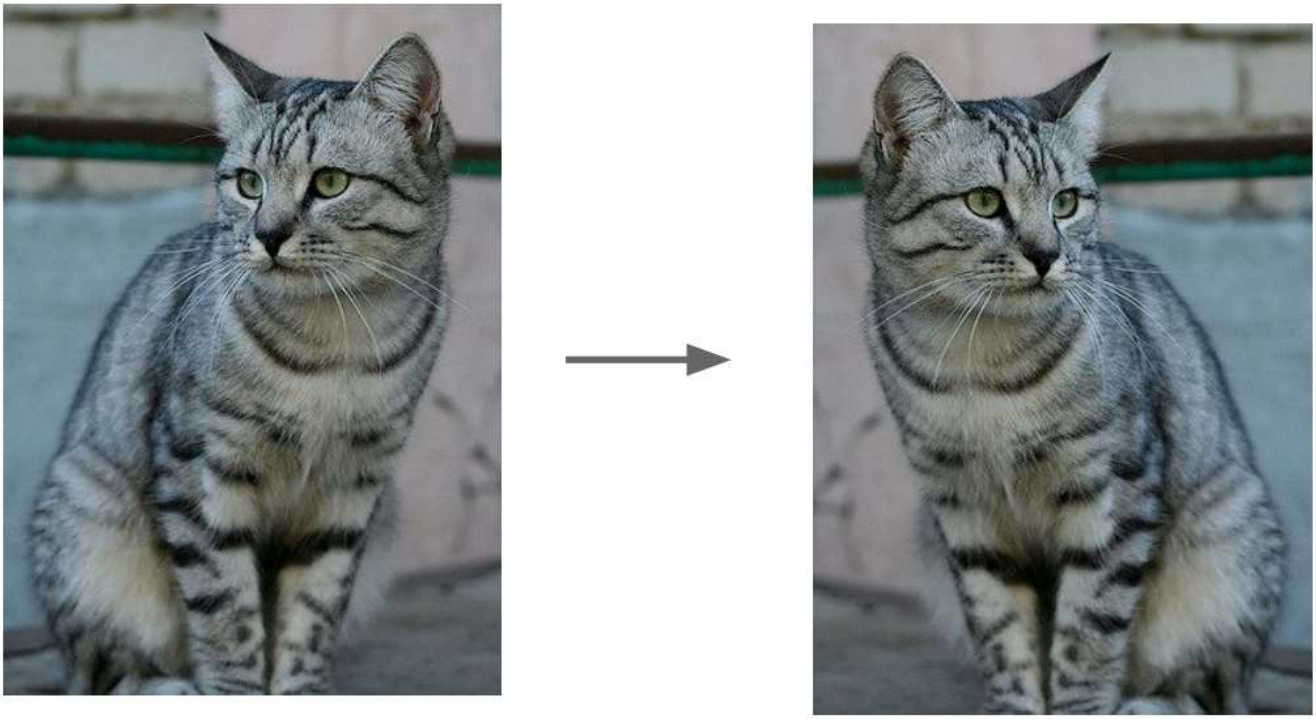
数据增强的基本思想就是，当我们手里的数据集不是很大的时候，我们可以通过一些措施对原有数据进行修改重新加入到数据集中，以达到数据集扩展的目的。



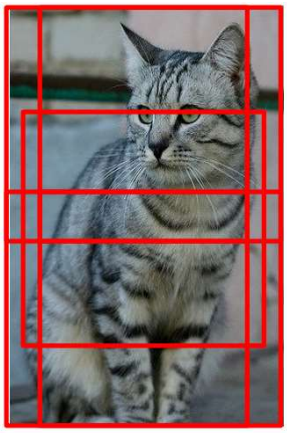
1. 基本方法

基本方法有以下几种：

(1) Horizontal Flips (水平翻转)



(2) Random crops and scales (对图片做随机裁剪和尺度改变)



Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

Testing: average a fixed set of crops

ResNet:

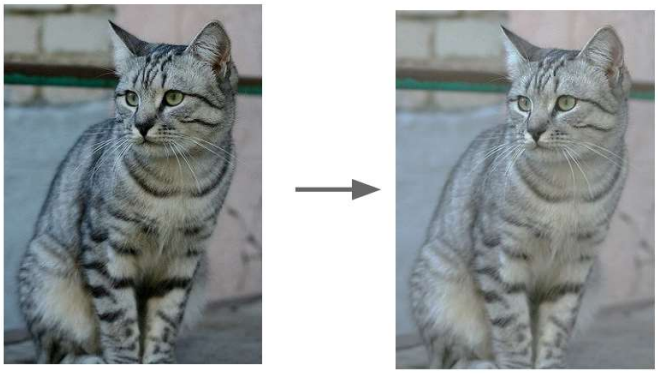
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

(3) Color Jitter (色彩变换)

Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

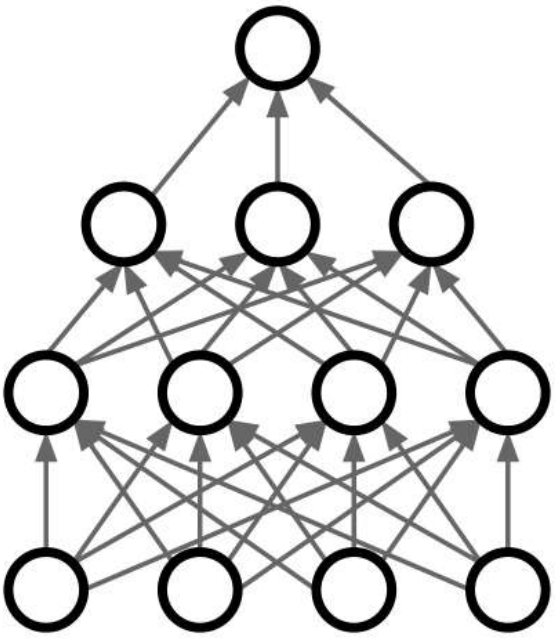
五、Regularization: A common pattern

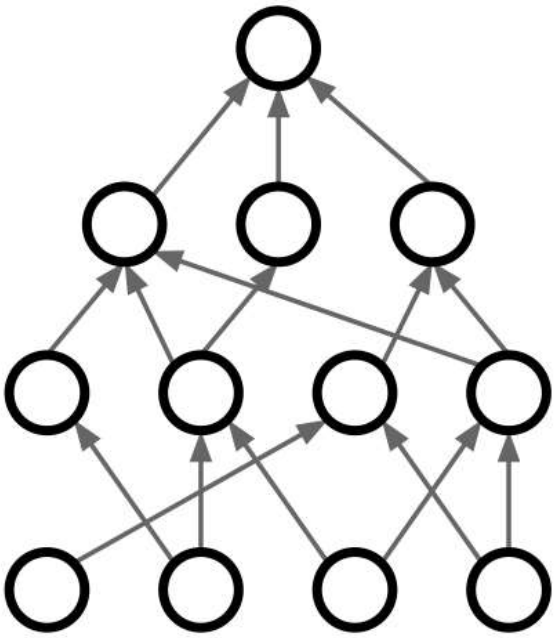
Training: Add random noise

Testing: Marginalize over the noise

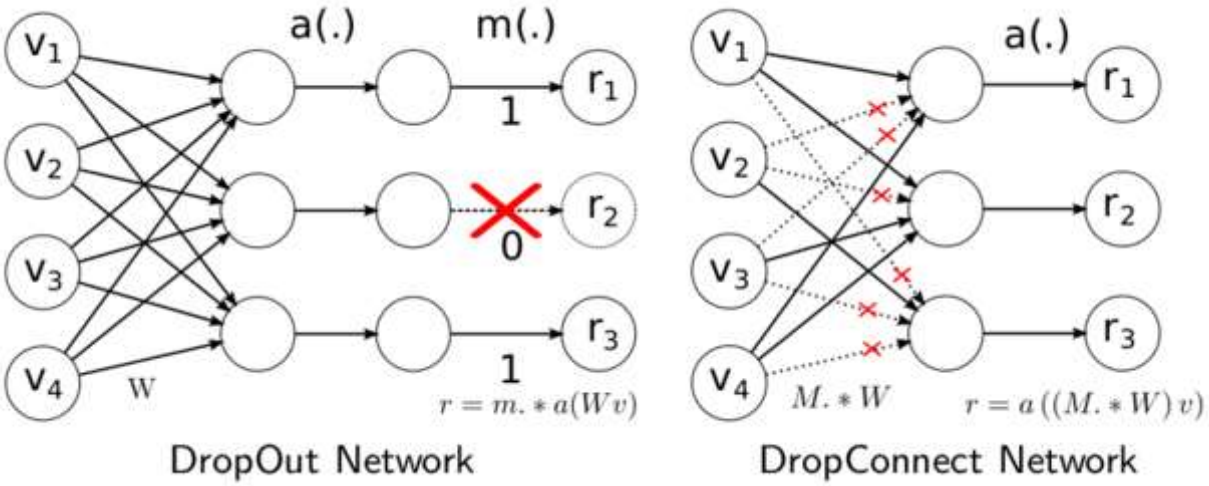
Examples:

- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect

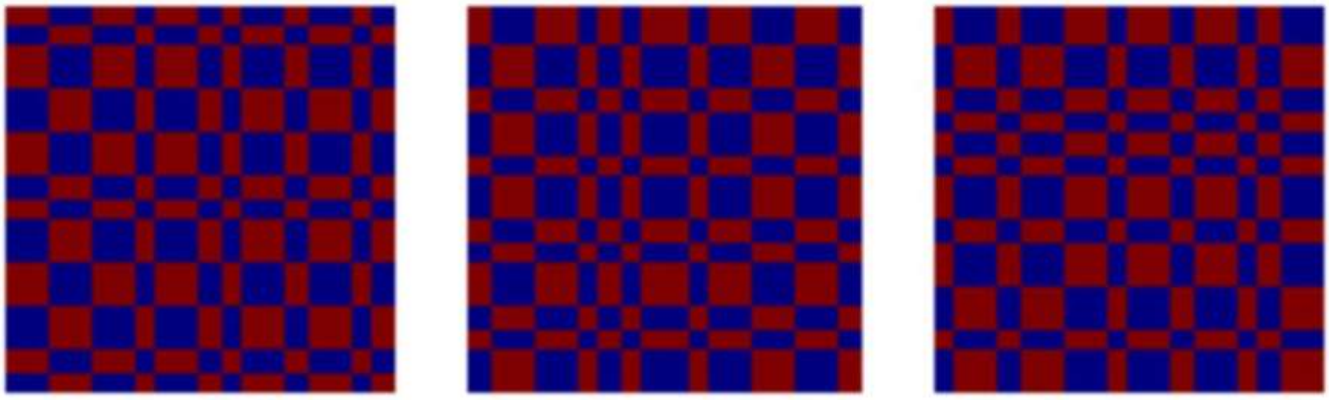




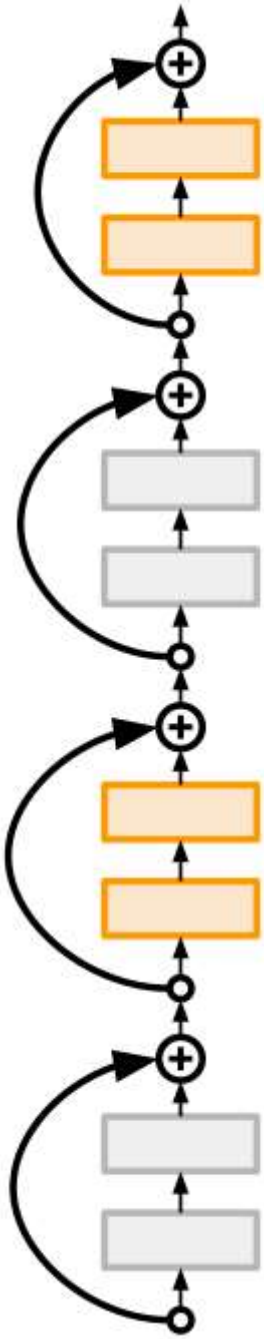
他与Dropout的区别其实就在于下面这张图。从图中其实可以看出来，Dropout是在激活函数计算完之后加了概率值，选择是否为0。而DropConnect则是先一步，在输入值乘以权重值的全连接阶段就去选择连接的活性，也就是选择一些神经元之间的连接被去掉。这样其实和后面的选择其实没啥大的区别，正则化的结果也是类似的，但是好处就在于：**他极大的减少了计算量。**

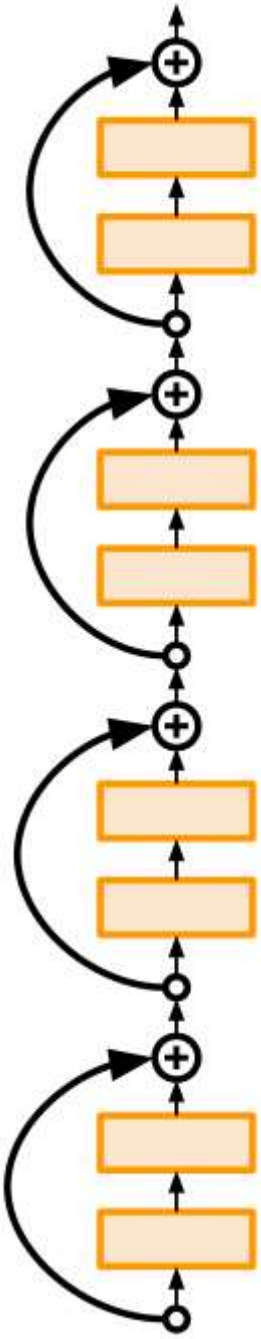


- Fractional Max Pooling (部分最大池化操作) 其实也差不多，就是随机选择一些区域池化，另一些区域不池化，结果其实和Dropout没啥大的区别。



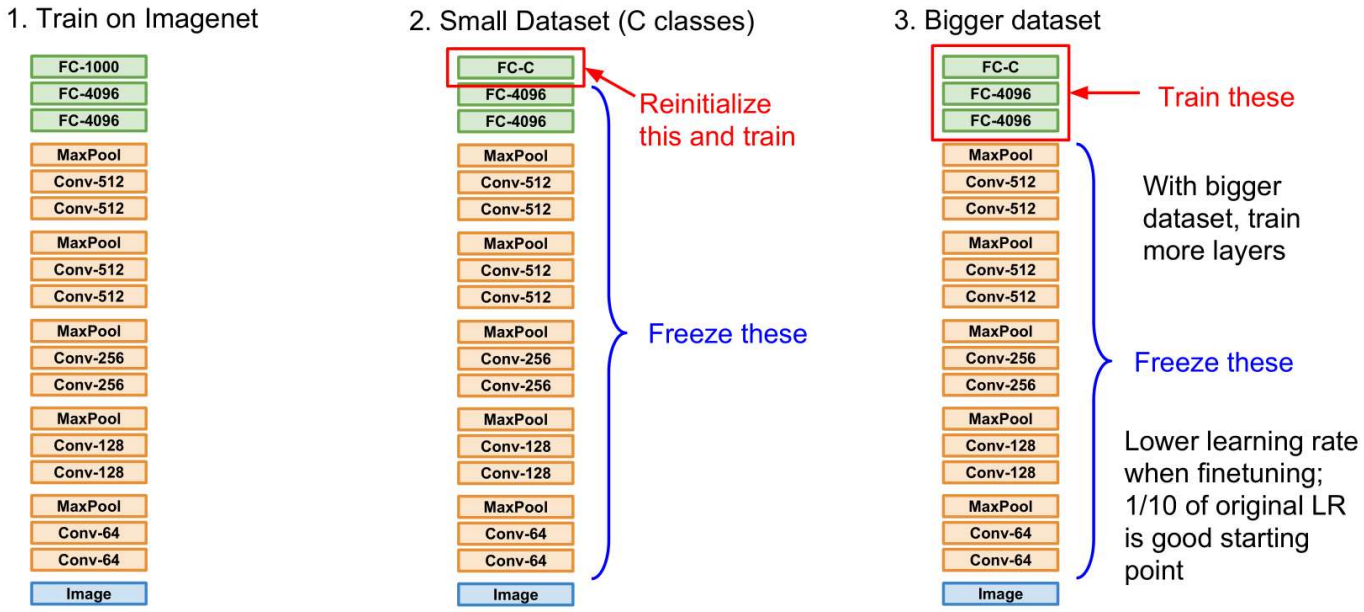
- Stochastic Depth (随机深度网络) 这个想法其实比较夸张，还处于研究阶段，并没有实际应用。他的主要想法是，因为网络深度超级大，所以我在训练过程中选择跳着训练，然后在test的时候用整体网络。这样做其实也是为了减少训练的计算量，与Dropout的结果是相似的。





六、Transfer Learning (迁移学习) —— 小样本训练方法

迁移学习的想法就是，我不需要去重新训练这么多的权重，不需要每一层都要训练，而是根据自己所掌握的数据大小去确定自己在已有的模型上对最后的几层做什么样的修改。因为选择的都是在ImageNet上训练好的模型，所以我们大多数只需要对最后的一层，最多三层做修改就可以。


1. Train on Imagenet
FC-1000
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image
2. Small Dataset (C classes)
FC-C
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image
Reinitialize this and train
Freeze these
3. Bigger dataset
FC-C
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image
Train these
With bigger dataset, train more layers
Freeze these
Lower learning rate when finetuning; 1/10 of original LR is good starting point