Regularization: Add term to loss 正则化：增加损失项

Regularization: Add term to loss

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

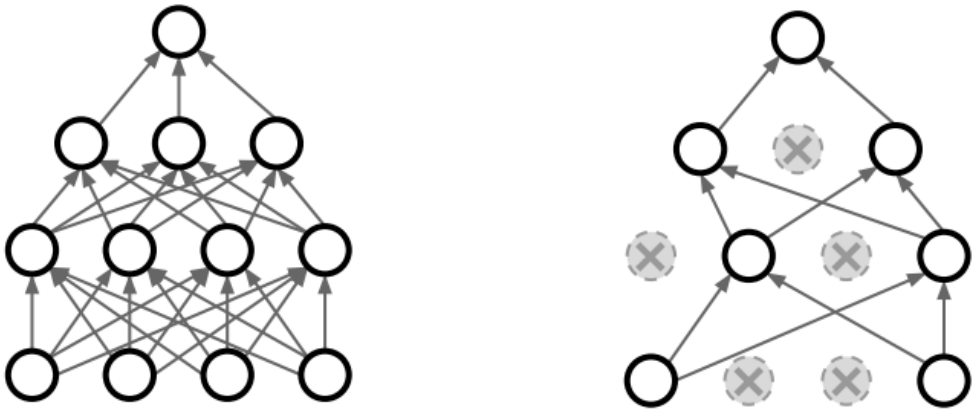
In common use:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$	(Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l}\rvert$	
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}\rvert$	

这里面我们在损失函数上加入额外的一项，我们有一项是让模型拟合数据，另一项则是正则项，L2正则化在神经网络中可能意义并不是很明确，有时候我们会在神经网络中选择其他方案。

Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



一个在神经网络中非常常用的方法就是dropout，Dropout非常简单，每次在网络中正向传递时，我们在每一层随机。将一部分神经元置零，每次正向传递时，随机被置零的神经元都不是完全相同的，。每次处理网络中的一层，我们经过一层网络，算出这一层的值，随机将其中一些置成零，然后继续在网络中前进，如果我们把左边这个全连接网络和右边经过dropout的版本进行对比，你会发现dropout后的网络，像是同样的网络变小了一号，我们只用到了其中一部分神经元，而且每次遍历，每次正向传递都是不同的部分。

Regularization: Dropout

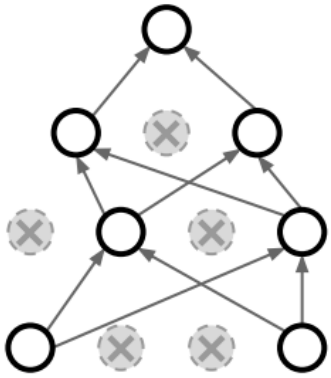
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

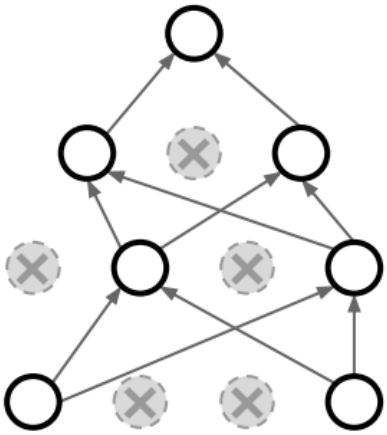
Example forward pass with a 3-layer network using dropout



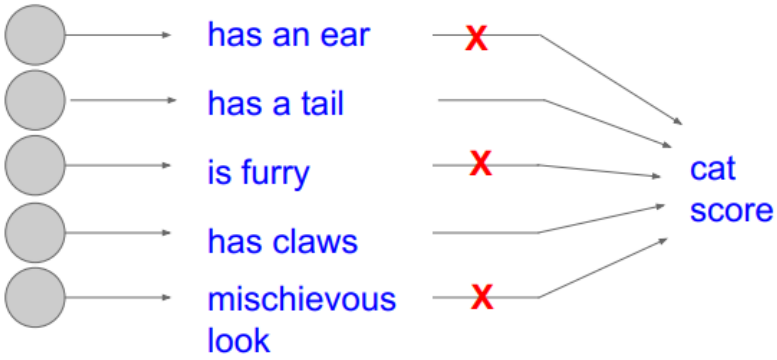
Dropout的实现非常简单，只需要两行代码，每行dropout一次。这是一个三层的神经网络，我们加上了dropout，唯一需要我们做的就是加上这行，随机将一部分神经元置零，这个实现起来非常简单。

Regularization: Dropout

How can this possibly be a good idea?



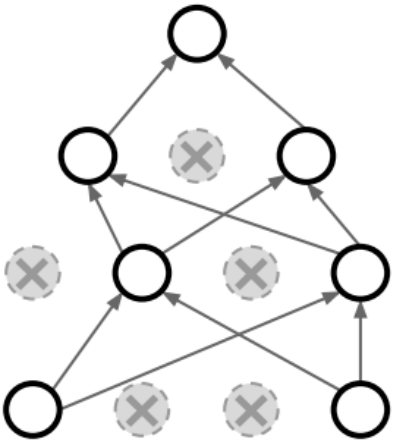
Forces the network to have a redundant representation;
Prevents co-adaptation of features



但问题是这个想法为什么可取？我们认真地在训练时将一部分神经元置成零，看看这样做是否有意义，一个比较勉强的解释是，人们觉得dropout避免了特征间的相互适应，假设我们要分类判断是不是猫，可能在网络里，一个神经元学到了有一只耳朵，一个学到了尾巴，一个学到了输入图像有毛，然后这些特征被组合到一起，来判断是否是猫，但是在加入dropout后，判断是不是猫时，网络就不能依赖这些特征组合在一起，给出的结果，而是要通过不同的零散的特征来判断，这也许某种程度上抑制了过拟合。

Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

另一种最近出现的关于Dropout的解释是，这是在单一模型中进行集成学习，如果你们观察左图，在dropout之后，我们是在一个子网络中，用所有神经元的子集进行运算，每一种可能的dropout方式都可以产生一个不同的子网络，所以dropout像是同时对一群共享参数的网络进行集成学习。顺便说一下，因为dropout的可能性随神经元个数呈指数倍增长，你不可能穷举每种情况，这可以看作是一个超级无比巨大的网络集合在同时被训练。

Dropout: Test time

Dropout makes our output random!

Output (label) Input (image) Random mask

$$y = f_W(x, z)$$

Want to “average out” the randomness at test-time

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

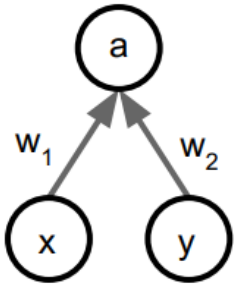
But this integral seems hard …

那么测试的时候又是什么样呢？当我们使用了dropout，我们把神经网络基本的运算都改变了，之前我们的神经网络里有权重w的函数f，输入x，并且得到输出y，但是现在我们额外有了一个输入z表示dropout中被置零的项，z的值是随机的，测试时引入一些随机性可能不是一个好主意，所以我们就想要平均这个随机性，如果把它写出来，你可以想象，通过一些积分来边缘化随机性，但是在实践中，这个积分是完全难以处理的，我们不知道怎样对这进行求解，一脸懵逼，你可能想做的一件事是，通过采样来逼近这个积分，在这儿，你可以对z的多次取样，然后在测试的时候把它们平均化，但是这仍然会引入一些随机性，这有点不好。

Dropout: Test time

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

辛运的是，在dropout的情况下，实际上我们可以用一种省事的方式局部逼近这个积分，如果我们考虑单个神经元输出是a，输入是x和y，以及两个权重，然后在测试时我们得到a的值是w1 * x + w2 * y 。现在想象一下，我们训练了这个网络，在训练期间，我们使用了dropout，丢弃神经网络单元的概率是0.5，现在，这个例子中，训练期间的期望值，可以算出解析解，有四个可能的dropout的掩码集合，我们将通过这四个掩码得到的值进行平均，我们可以看到a的期望，在训练期间为0.5 * (w1 * x + w2 * y)，这里有一点不统一，测试时平均值是w1 * x + w2 * y，训练时却只有一半。我们能做的一件省事的事情就是，在测试时我们没有任何随机性，而是用dropout的概率乘以这个输出，现在这些期望值是一样的，这有点像简易地局部逼近这个复杂的积分，这就是人们对于dropout，在实践中非常普遍的做法。

```
""" 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

在dropout时，对于预测函数，我们用dropout的概率乘以我们输出层的输出。总结起来，dropout在正向传播中非常简单，你只需要添加两行到你的实现中随机对一些节点置零，然后在测试时的预测函数内，你仅仅增加了一点点乘法，乘以你的概率，Dropout超级简单，它有时对于正则化神经网络有很大帮助。

More common: “Inverted dropout”

```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!

顺便说一下，有时你会看到一个常见的技巧，反转dropout，也许在测试的时候，你更关心效率，所以你想在测试的时候消除乘以概率p的这一额外的乘法，那么你可以做的是，在测试的时候，你使用整个权重矩阵，但是在训练的时候，除以p，因为训练可能发生在GPU上，你真的不在乎在训练的时候，做一个额外的乘法运算，然而在测试的时候，你想要这个过程尽可能高效。

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

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

Example: Batch Normalization

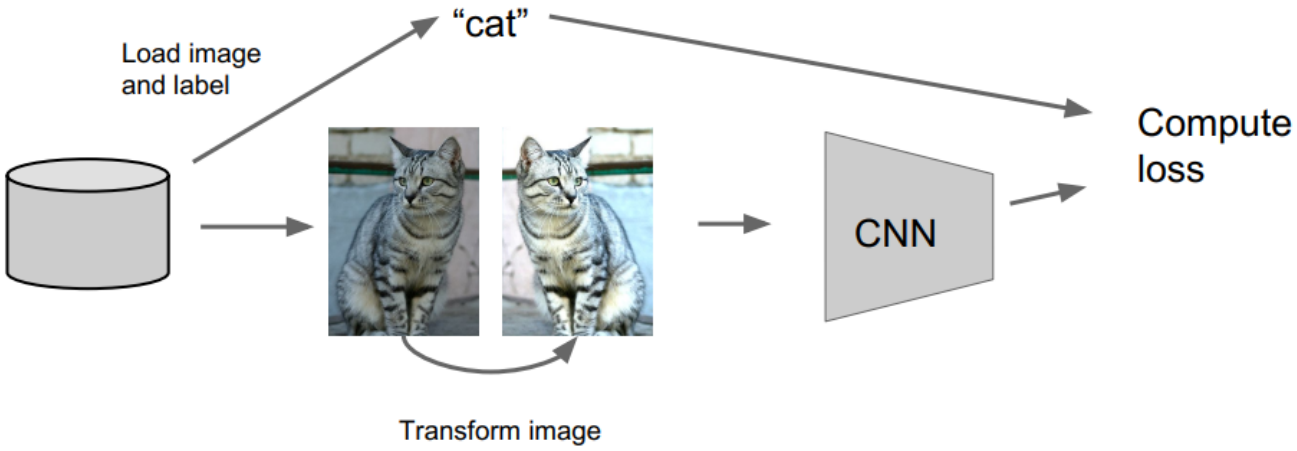
Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Dropout在我们看来是这样一个具体实例，这里有一个更通用的正则化策略，在训练期间，我们给网络添加一些随机性，以防止它过拟合训练数据，一定程度上扰乱它，防止它完美地拟合训练数据，现在在测试的时候，我们要抵消掉所有随机性，希望能够提高我们的泛化能力，dropout可能是最常见的使用这种策略的例子，但是实际上，Batch Normalization也符合这个想法，记得在Batch Normalization中，在训练的时候，一个数据点可能和其它不同的数据点出现在不同的小批量中，对于单个数据点来说，在训练过程中该点会如何被正则化，具有一定的随机性，但是在测试过程中，我们通过使用一些基于全局估计的正则化来抵消掉这个随机性，而不是采用每一小批量估算，实际上，Batch Normalization倾向于具有和dropout类似的正则化效果，因为它们在训练的时候，都随机引入某种随机性或者噪声，但是又在测试的时候抵消掉它们。实际上，当你使用Batch Normalization来训练神经网络的时候，有时你一点儿都不会使用dropout，仅仅是Batch Normalization给你的

网络，增加了足够的正则化效果，dropout某种程度上更好，因为你实际上可以通过改变参数p调整正则化的力度，但是在Batch Normalization中，并没有这种控制机制。

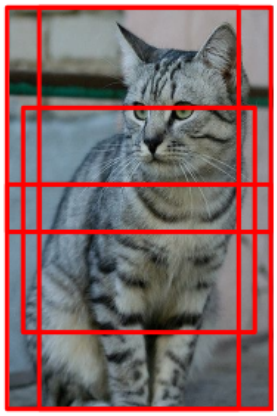
Regularization: Data Augmentation



另一种符合这种范式的策略就是这种数据增强的想法，在训练的时候，有一个最初的版本，我们有自己的数据，也有自己的标签，在每一次迭代中，我们使用它去更新我们的卷积神经网络，但是我们可以做的是在训练过程中，以某种方式随机地旋转图像，使得标签可以保留不变，现在我们用这些随机转换的图像进行训练，而不是原始的图像，有时你可能会看到随机的水平翻转。

Data Augmentation

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

假如，你采用了一张猫的图像并水平翻转了它，它依旧是一只猫，你可以从图像中随机抽取不同尺度大小的裁剪图像，因为猫随机地裁剪图像依然是一只猫，然后在测试过程中，通过评估一些固定的裁剪图像来抵消这种随机性，通常是四个角落和中间，以及它们的翻转。比较常见的就是，当你阅读ImageNet上论文的时候，他们会总结，他们模型的单个裁剪图像效果，这个就像整个图像一样，和它们模型的10种裁剪方式的效果，包括这5种标准裁剪，加上它们的翻转。

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)

同样，在数据增强中，有时会使用色彩抖动，你可能会在训练的时候随机改变图像的对比度和亮度，你也可以通过色彩抖动来得到一些更复杂的结果，当你试图在你的数据空间的主成分分析方向上产生色彩抖动，或者其它什么时候。你会以某种与数据相关的方式进行色彩抖动，但是这是不太常见的。

Data Augmentation
Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

一般来说，数据增强是非常普遍的事情，你可以将其应用于任何问题，不管你想要解决什么问题，可以考虑，在不更改标签的前提下，对数据进行转换，现在在训练的时候，你只需将这些随机转换应用于你的输入数据，这种方式对网络有正则化效果，因为你在训练的时候，你又增加了某种随机性，然后在测试的时候将它们淡化。

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

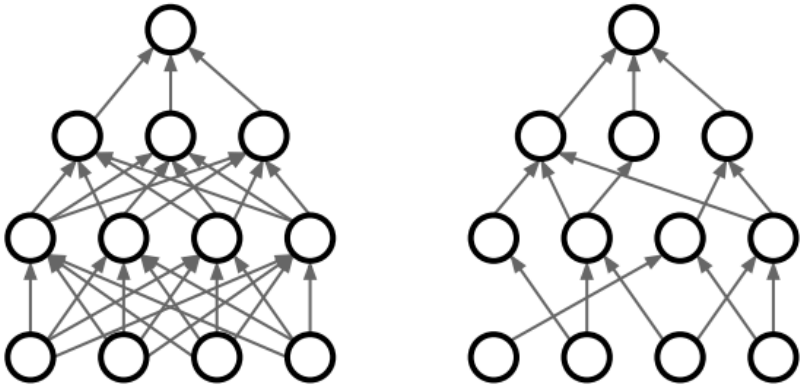
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



现在我们已经看到了这种模式的三个例子，Dropout、Batch Normalization和数据增强，但是还有很多其他的例子，一旦你学会这个模式，当你阅读其它论文的时候，你可能会认出它们。还有一种与dropout相关的算法，叫做dropconnect，DropConnect是同样的想法，但不是在每次正向传播中将激活函数置零，而是随机将权重矩阵的一些值置零，它们有一样的效果。

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

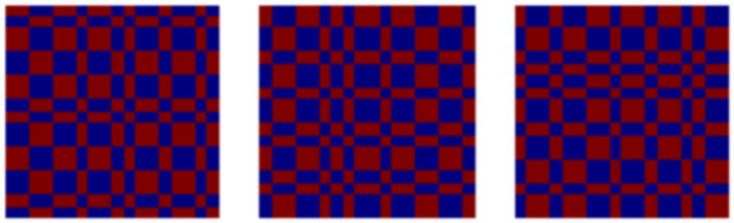
Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

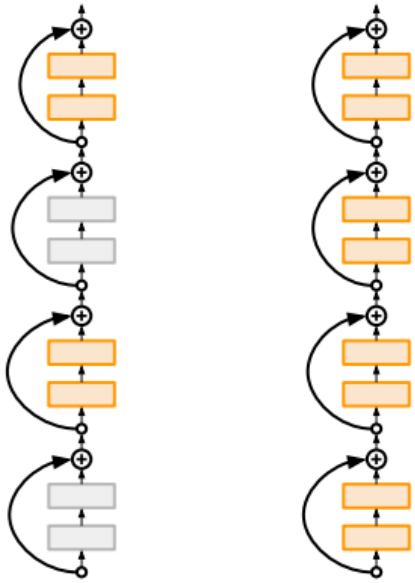


另一个我喜欢的想法，虽然不太常用，但是我认为是一个非常好的想法，就是部分最大化池化的想法，一般的，当你进行2 * 2的最大池化时，以前我们会把固定的2 * 2的区域，在前向传播的前面进行池化，但是现在通过部分最大化池化，每次我们在池化层操作时，我们将随机池化，我们正在池化的区域，如右图所示，展示了3个不同的，在训练时可能遇到的随机池化区域，在测试的时候，有很多方法可以得到抵消随机性，要么使用一些固定的池化区域，要么选择很多样本对它们取平均，这是一个很好的想法，虽然并不常用。

Regularization: A common pattern

Training: Add random noise
Testing: Marginalize over the noise

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth



Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

另一个令人眼前一亮的论文是2016年发表的，它就是随机深度，如左图所示，我们有一个很深的网络，在训练时，我们随机的从网络中丢弃部分层，在训练时，我们消除一些层，只用部分层，在测试的时候，我们用全部的网络，有点不可思议，这是一个神奇的研究，效果有点趋向Drouput的正则化效果和其他的类似策略，这是非常前沿的研究，在实际操作的时候并不常用，但他的想法不错！

大多数时候使用Batch Normalization就够了，可以加Dropout。