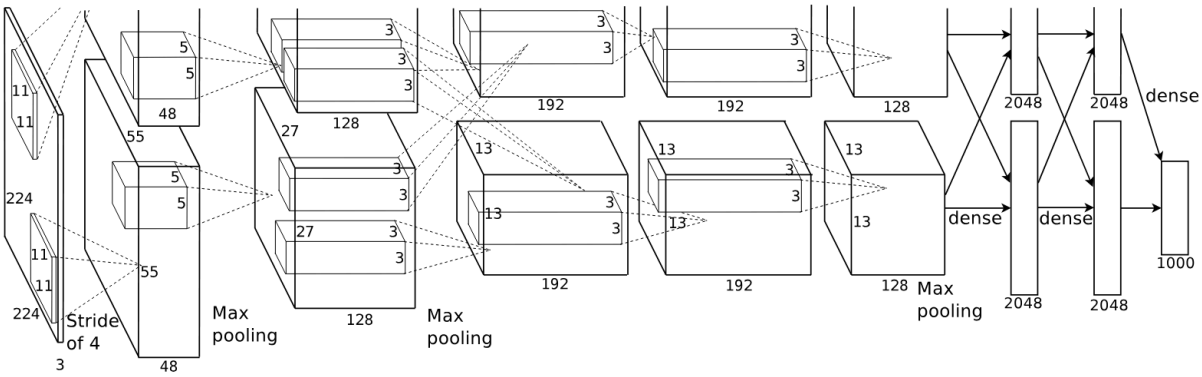
LECTURE NOTE

STANFORD CS231N · CONVOLUTIONAL NEURAL NETWORKS FOR VISUAL RECOGNITION



Yifeng Xu
Harbin Institute of Technology, Shenzhen

2021.02 – 2021.04

Course

Convolutional Neural Networks for Visual Recognition (Stanford CS231n, Spring 2017)

Lecturer

Prof. Fei-Fei Li

Dr. Justin Johnson

Dr. Serena Yeung

Resources

Website: `https://cs231n.stanford.edu/2017/`

Online Video

BiliBili: `https://www.bilibili.com/video/BV1nJ411z7fe`

目录

1 Image Classification

1.1 Data-Driven Approach

不同于传统的算法，统计机器学习的算法是**数据驱动**的。以图像分类任务为例，数据驱动方法包含以下 3 个步骤：

1. 收集数据并标注；
2. 使用机器学习方法训练一个分类器；
3. 使用该分类器对新的图片进行分类。

实现上，一个统计机器学习算法至少包含两个函数：`train` 和 `predict`，前者进行模型训练，后者使用训练好的模型进行预测。

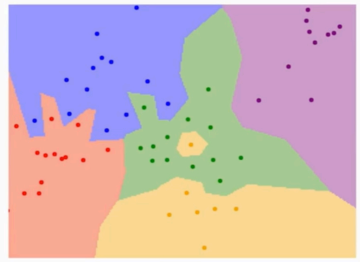
1.2 Nearest Neighbor

最近邻分类器是最简单的分类器。仍然以图像分类为例，最近邻的 `train` 过程仅仅是记录下所有数据及其标签，`predict` 过程是选取数据集中与当前图片最相近的图片，并以其标签作为结果输出。

尽管最近邻很简单，但是仍然有一个问题需要解决：怎么评判两张图片的相似程度。由于图片可以看作像素点组成的 $H \times W \times C$ 向量，所以问题转化为两个向量相似程度的评判。设 I_1, I_2 是要比较的两个图片向量，上标 p 表示向量的第 p 个元素，常用的距离有：

- L1 distance（曼哈顿距离）：$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$
- L2 distance（欧几里得距离）：$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$

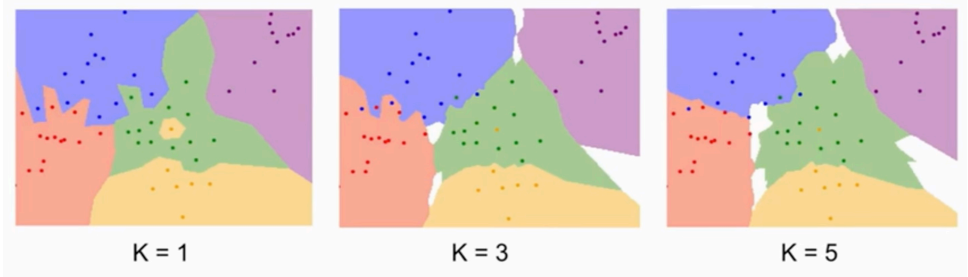
最近邻分类器准确率较差，这是因为如果出现一个噪声点，它将对周围一圈产生误导，如下图所示。

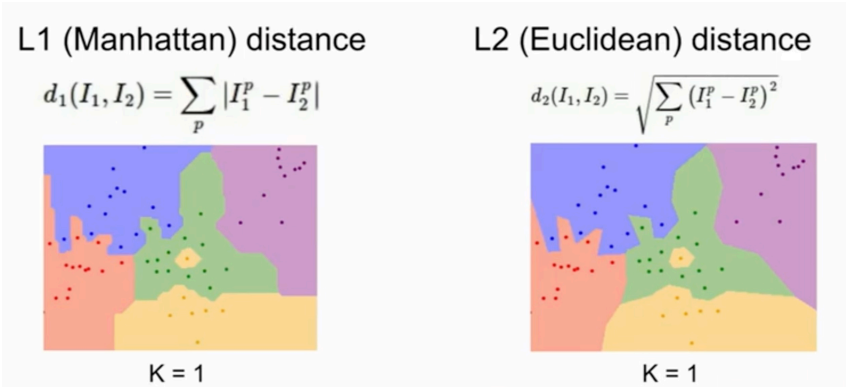


1.3 K-Nearest Neighbor

为了解决最近邻的问题，k 近邻查看 k 个最近的点，并以其中最多的标签作为结果。

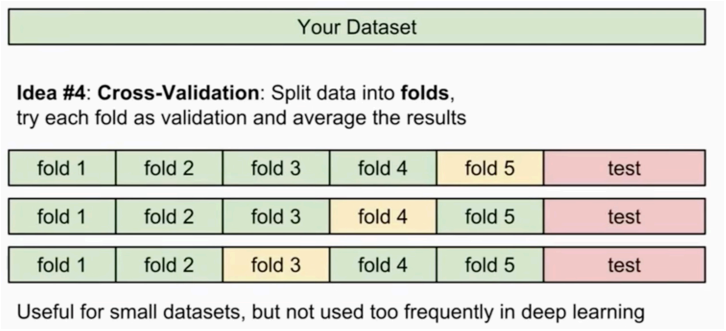
超参数 在 k 近邻算法中，k 如何取值是我们设定的，而非模型学习出的，这类参数被称作超参数。k 近邻算法的另一个超参数是距离函数的选取，选 L1 distance 和选 L2 distance 的结果会不同。





训练、验证与测试集 为了找寻最好的超参数，我们需要将数据集分为 3 类：训练集、验证集、测试集. 选取一组超参数后，我们在训练集上训练模型，在验证集上测试该模型的性能，调整超参数使得性能达到最佳，最后在测试集进行测试，并以测试集上的性能作为该模型的真正性能。

交叉验证 在数据集不是很多的时候，我们可以选择 k 折交叉验证的方式代替上述方式。我们分出一部分测试集后，把剩下的数据集分成 k 份（fold），循环地将每一份都视为验证集、其他份视为训练集进行 k 次训练，以 k 个训练性能的平均视为当前超参数下的训练性能，然后对超参数进行调整使得性能达到最佳，同样最后以在测试集上的性能作为真正性能。



KNN 不被使用 KNN 算法非常简单，但是问题也非常突出：

1. 它基本没有训练过程，仅仅是把数据集记录下来，而在预测时需要遍历整个数据集，非常耗时（如果使用 KD-Tree 实现，随机数据下复杂度是 log 的，要好一些，但是仍然不满意）；这与我们需要的正好相反——我们愿意花费较多的时间对模型进行训练，但一旦训练好了，在对新数据预测时要很快地出结果。
2. 不同的图像之间可以有相同的距离，这种度量方式对图像而言不太好。
3. 当向量维度很高时，我们的数据点在高维空间中的分布是稀疏的，所以所谓的“最近点”可能其实并没有多么相似。

2 Linear Classification

2.1 Linear Classifier

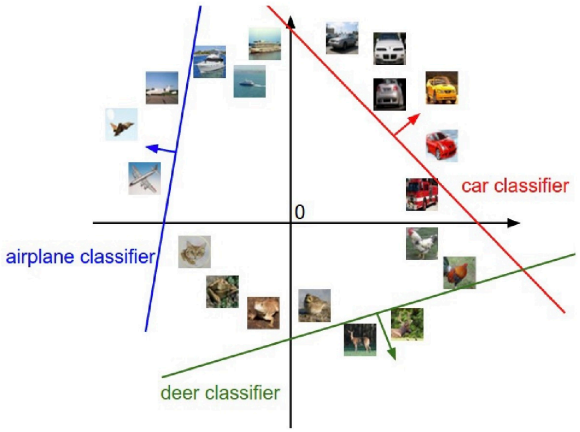
设有数据集 $\{(x_i, y_i) \mid i = 1, 2, \ldots, N\}$，其中 $x_i \in \mathbb{R}^D$, $y_i \in \{1, \ldots, K\}$. 定义 score function: $f : \mathbb{R}^D \mapsto \mathbb{R}^K$，即输入一个由图片像素构成的 D 维向量，输出这个图片属于各个类别的 score. 我们认为 score 越高，图片越可能属于这个类别。

对于 linear classifier 来说，它的 score function 为：

$$f(x; W, b) = Wx + b$$

其中，$W \in \mathbb{R}^{K \times D}$, $b \in \mathbb{R}^K$ 是参数，分别称作 weights 和 bias. 由于第 i 类的得分为 W 的第 i 行与 x 的内积加上 b 的第 i 个元素，因此类别之间的得分相互独立，写成矩阵只是为了方便而已。

为直观理解 linear classifier，注意到 $z = wx + b$ 是一个 $\mathbb{R}^{D+1}$ 中的超平面，法向量为 $(w, -1)$，截距为 b. 输入的 x 在超平面上对应位置的 “高度” 越高，就越可能属于这一超平面代表的那一类。下面是 $D = 2$ 的一个例子，线条表示超平面与 $z = 0$ 的交线，沿着箭头方向走，$wx + b$ 变大。



为了获得参数 W, b，我们需要定义损失函数 (loss function)。训练 W, b 就是最小化损失函数的过程，机器学习问题最终归结为一个优化问题。

2.2 Multiclass SVM

Multiclass SVM 是一种常用的损失函数. 对于数据 (x_i, y_i)，其损失定义为：

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

其中，s_j 表示 x_i 在第 j 类上的得分，即 $s_j = f(x_i; W, b)_j$.

对这个损失函数的理解是，如果正确分类的得分 s_{y_j} 比错误分类的得分 s_j 还高一个 Δ，那么损失就是 0；否则，损失是 $s_j + \Delta$ 比 s_{y_j} 多出的部分。由于这个函数的图像形状像铰链（合叶），所以这种损失函数也称作 **hinge loss**.

在机器学习课程中我们学过正则化。若使用 L2 regularization，则 Multiclass SVM 的完整形式是：

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W,b)_j - f(x_i;W,b)_{y_i} + \Delta) + \lambda\sum_k\sum_l W_{kl}^2$$

值得注意的是，上式中的 Δ 并不是需要调节的超参数，取 $\Delta = 1$ 即可。这是因为 W 的整体放缩可以在不改变 s_j 和 s_{y_i} 的大小关系的条件下改变它们的差值，所以 Δ 取值并不影响结果。

2.3 Softmax classifier

Softmax classifier 是二元逻辑回归分类器在多分类上的扩展，不同于 multiclass SVM，softmax classifier 的输出有一个概率的解释。具体而言，对于数据 (x_i, y_i)，设 f_j 表示它在第 j 类上的得分，softmax classifier 视之为尚未标准化的对数概率，并采取 **cross-entropy loss** 作为损失函数：

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

其中，函数 $f_j(z) = e^{z_j}/\sum_k e^{z_k}$ 称作 **softmax function**.

同样的，总的损失函数定义为各 L_i 的平均值加上正则化项：

$$L = -\frac{1}{N}\sum_{i=1}^{N}\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) + \lambda\sum_k\sum_l W_{kl}^2$$

从信息论角度理解，设真实概率分布为 p，估计概率分布为 q，则 cross-entropy 定义为：

$$H(p,q) = -\sum_x p(x)\log q(x)$$

在图像分类问题中，我们估计的各个分类上的概率分布为 $q = e^{f_{y_i}}/\sum_j e^{f_j}$，而真实分布为 $p = [0,\ldots,1,\ldots,0]$（即正确的分类为 1，其余为 0 ）。Softmax classifier 最小化的就是 $H(p,q)$.

从概率论角度理解，

$$\mathbb{P}(y_i|x_i;W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

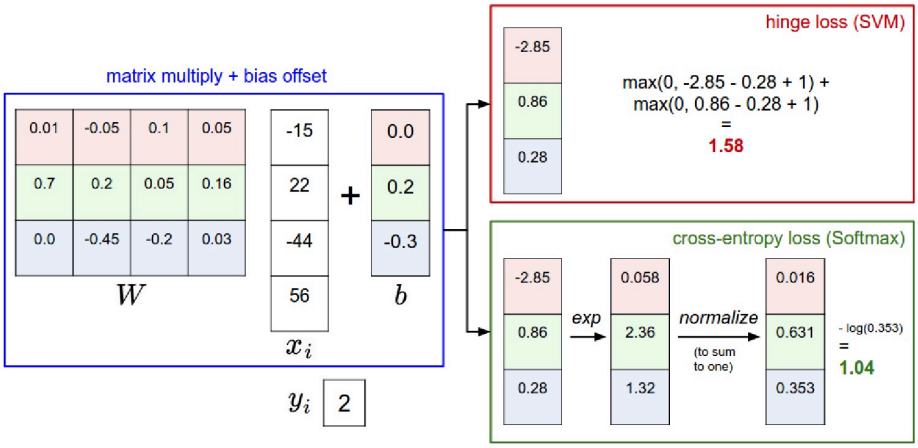
可以解释为当参数为 W, b 时，x_i 的类别为 y_i 的概率，因此最小化 L_i 等价于实施极大似然估计。

值得注意的是，在编写代码时，e^{f_j} 可能太大以至于计算精度较低，但是注意到：

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C\sum_j e^{f_j}} = \frac{e^{f_{y_i}+\ln C}}{\sum_j e^{f_j+\ln C}}$$

所以我们可以取 $\ln C = -\max_j f_j$ 来解决这个问题。

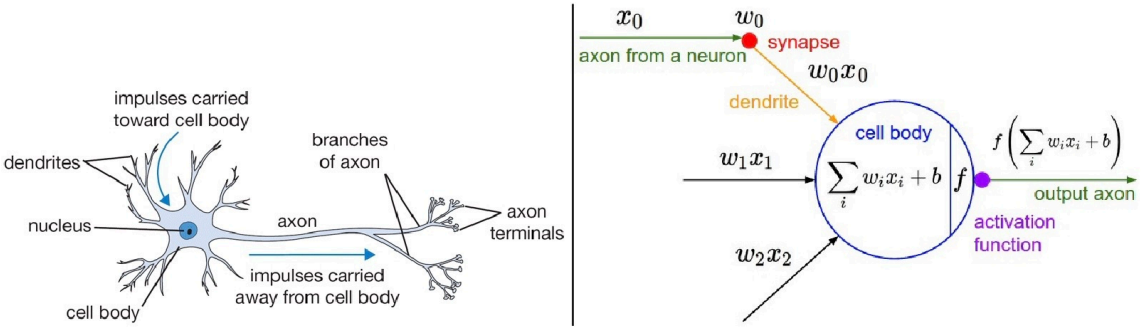
2.4 SVM vs. Softmax



3 Neural Networks

3.1 Modeling One Neuron

神经元 一个神经元是其输入的线性组合加上偏置项，再取激活函数 f 之后输出。



常用激活函数

名称	表达式
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$
Tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
ReLU	$f(x) = \max(0, x)$
Leaky ReLU / PReLU	$f(x) = [x < 0](\alpha x) + [x \geqslant 0]x$
Maxout	$f(x) = \max(w_1x + b_1, w_2x + b_2)$

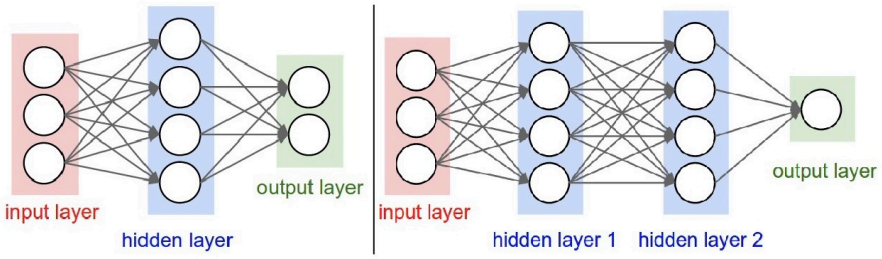
- Sigmoid：函数曾经很常用，但最近很少用了，这是因为它有两个大弊端：
  - Sigmoids 会 saturate，导致**梯度消失**。由于 $\sigma(x)$ 在 $|x|$ 较大时梯度非常小，而神经网络反向传播的本质是链式求导法则，要把梯度一级一级地乘起来，所以一旦遇到梯度很小的地方，梯度就无法继续反向传播了，这将导致前层的神经元的参数几乎不怎么更新。
  - Sigmoid 的输出不是以 0 为中心的，这意味着后一层神经元接受的输入都是大于 0 的数，从而导致参数 w 的梯度全为正或者全为负。
- Tanh: $\tanh(x) = 2\sigma(2x) - 1$，所以依然存在梯度消失的问题。
- ReLU：
  - (+) 与 sigmoid 或 tanh 相比，ReLU 大大加速了 SGD 的收敛速度；
  - (+) ReLU 的计算非常简单，而 sigmoid 和 tanh 涉及到指数等比较耗时的计算；
  - (-) 使用 ReLU 容易导致神经元“死亡”. 如果学习率设置的过大，容易使得参数变化过大，导致对数据集中的所有点 $wx + b$ 都小于 0，于是梯度不可逆地变成 0.
- Leaky ReLU：为了解决 ReLU 中神经元“死亡”的问题，引入一个小常数 α.
- PReLU：Leaky ReLU 中的 α 被视作神经元的一个参数参与优化。
- Maxout：ReLU / Leaky ReLU 是 Maxout 的特殊情况。

单神经元实现线性分类器 选取合适的损失函数后，一个神经元可以用来实现一个 linear classifier.

- **Binary Softmax classifier**：视神经元的输出 $\sigma\left(\sum_i w_ix_i + b\right)$ 为分类为正类的概率，采取 cross-entropy loss 训练，则得到 binary Softmax classifier，也即逻辑回归.
- **Binary SVM classifier**：使用 max-margin hinge loss 训练，则得到 Binary SVM.

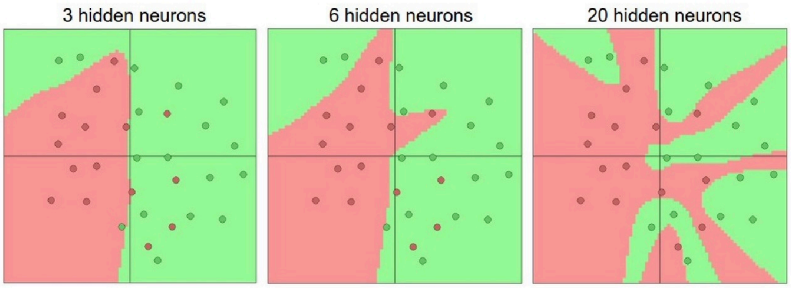
3.2 Neural Network Architectures

一个输入层，一个输出层，若干隐藏层。对于最普通的神经网络而言，层与层之间全连接。输出层的神经元经常没有激活函数，这样输出可以被视作分类的得分或者一些取实值的目标（比如回归值）。

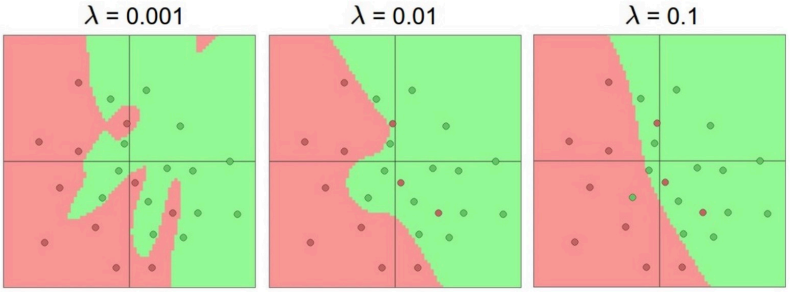


单层神经网络可以以任意精度近似任何函数，严格证明在 Approximation by superpositions of a sigmoidal function (1989) 这篇论文中，直观解释可以在博客 Neural Networks and Deep Learning 中找到。虽然如此，实践中多层的神经网络效果更好，也更容易学习。

神经网络中，隐藏层神经元数量越多，其表达能力越强大：



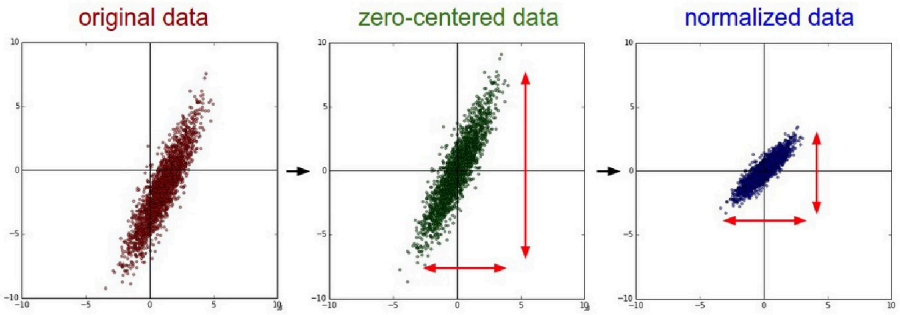
但这也意味着越容易过拟合，可以采取正则化的方式避免过拟合：



3.3 Data Preprocessing

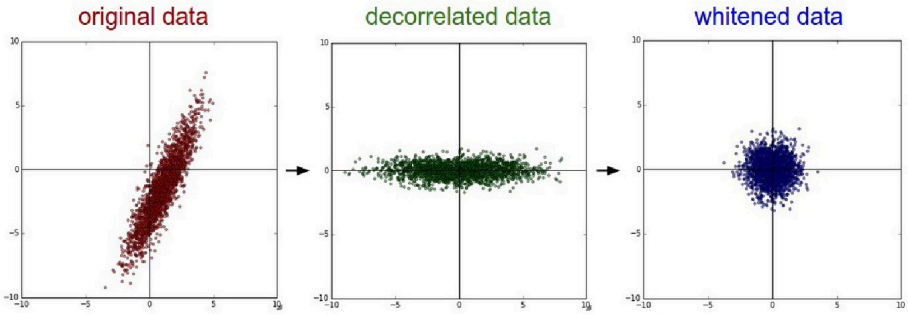
Mean subtraction 将所有数据减去平均值，即中心化。

Normalization 将数据的各维度缩放到大致相同的规模。一般有两种方式：一是对中心化后的数据除以各维度的标准差，使得各维度的数据均值为 0，方差为 1；二是将最大值和最小值缩放为 1 和 -1，这样所有数据都在 $[-1, 1]$ 之中。



PCA 主成分分析 (PCA) 是一种常用的数据降维方法，具体内容可见机器学习笔记。

Whitening 假设数据是一个多元高斯分布，则 whitening 将数据变为均值为 0，协方差矩阵为单位矩阵的多元高斯分布。就具体实现而言，在主成分分析中，如果没有使维度减小，则其效果为去相关性 (decorrelate)；对去相关性后的数据除以各维度的特征值就得到了 whitening 的数据。



注释. 卷积神经网络一般不采用 PCA 和 whitening，但是中心化以及 normalization 很重要。

注意. 在数据预处理时，预处理的统计量应该由训练集计算而来，然后应用到验证集或测试集上。**不能够**先在整个数据集上预处理，再划分训练/验证/测试集。

3.4 Weight Initialization

全零初始化（陷阱） 全部初始化为 0 是错误的做法，因为这样同一层的所有神经元都没有区别，它们将输出同样的值、具有同样的梯度、进行同样的更新（因为它们是对称的），这是没有意义的。

小随机数 初始化为接近 0 的随机数是经常使用的方法。一般而言，可以取 0.01 倍的标准正态分布或者均匀分布等。

方差修正 上一个方法存在一个问题，即神经网络输出的方差将随着输入量的增大而增大。在 Neural Networks and Deep Learning 中作者举了一个例子说明这一点：不妨假设 1000 个输入中有 500 个 0 和 500 个 1，考察第一个隐藏层中的一个神经元，$z = \sum_j w_j x_j + b$ 将是 501 个标准正态分布

之和，故服从 $N(0, 501)$，方差高达 501. 为了解决这个问题，我们计算：

$$\begin{aligned}
\mathrm{var}(z) &\approx \mathrm{var}\left(\sum_j w_j x_j\right) && \text{for simplicity, ignore bias} \\
&= \sum_j \mathrm{var}(w_j x_j) && \text{independence} \\
&= \sum_j (\mathbb{E}[w_j])^2 \mathrm{var}(x_j) + (\mathbb{E}[x_j])^2 \mathrm{var}(w_j) + \mathrm{var}(x_j)\mathrm{var}(w_j) \\
&= \sum_j \mathrm{var}(x_j)\mathrm{var}(w_j) && \text{assume } \mathbb{E}(x) = \mathbb{E}(w) = 0 \\
&= n\ \mathrm{var}(w)\ \mathrm{var}(x)
\end{aligned}$$

因此，若要 $\mathrm{var}(z) = \mathrm{var}(x)$，需要 $\mathrm{var}(w) = 1/n$，故我们只需要将初始化的值除以 $1/\sqrt{n}$ 即可。实践中这样做能加快收敛。

> 注释. 在 Understanding the difficulty of training deep feedforward neural networks by Glorot et al. 这篇论文中，作者最终建议用 $\mathrm{var}(w) = 2/(n_{\text{in}} + n_{\text{out}})$ 来初始化；在 Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification by He et al. 中，作者对 ReLU 神经网络进行分析，得到结论是方差应初始化为 $2/n$.
> **实践中，建议采用 ReLU 神经元并将权重方差设为 $2/n$.**

稀疏初始化 另一种初始化方式是将权重矩阵初始化为 0，但是为了打破对称性，每个神经元随机地连接下一层若干神经元（连接数目可由一个小的高斯分布生成）。

偏置初始化 偏置项常初始化为 0.

Batch Normalization 在全连接层/卷积层与激活函数之间插入一个 BatchNorm layer，因为 normalization 是可导的操作，这么做是可行的，具体查看论文：Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

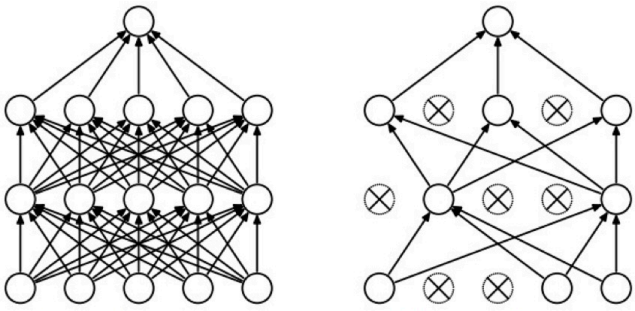
3.5 Regularization

L2 正则化 最常见的正则化方法，向损失函数加上 $\frac{1}{2}\lambda w^2$. 其特点是使得网络倾向于分散的权重。

L1 正则化 相对常见的正则化方法，向损失函数加上 $\lambda|w|$. 其特点是使得网络的权重倾向于稀疏（即许多非常接近 0 的数字，与 L2 正好相反）。结合 L1 和 L2 可得 Elastic net regularization，即向损失函数加上 $\lambda_1|w| + \lambda_2 w^2$. 一般而言，如果不是特别关注某些特征的选择，L2 比 L1 效果会更好。

最大范数约束 给神经元的权重向量 w 以约束：$||w||_2 < c$，c 一般取 3 或 4。

Dropout 及其有效和简单的正则化方法，在 Dropout: A Simple Way to Prevent Neural Networks from Overfitting by Srivastava et al. 中提出，能作为其他正则化方法的补充。实现方法是在训练中让神经元以概率 p（一个超参数）激活或设置为 0.



(a) Standard Neural Net (b) After applying dropout.

偏置项 偏置项没有与数据直接相乘，因此无需正则化，即便正则化对结果的影响也微乎其微。

3.6 Loss Functions

分类 常用 SVM loss:

$$L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1)$$

和 cross-entropy loss:

$$L_i = -\ln\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

属性分类 不一定被分为某一类，而可以分到若干类，设 y_{ij} 表示第 i 个图像是否属于第 j 类。我们可以对每一类训练一个 logistic regression classifier，然后用负的对数似然作为损失函数：

$$L_i = -\sum_j y_{ij} \ln \sigma(f_j) + (1 - y_{ij}) \ln(1 - \sigma(f_j))$$

回归 可采取 L2 或 L1 距离作为损失函数，即：

$$L_i = ||f - y_i||_2^2 = \sum_j (f_j - (y_i)_j)^2$$

或

$$L_i = ||f - y_i||_1 = \sum_j |f_j - (y_i)_j|$$

3.7 Gradient Checks

神经网络反向传播过程很容易写出 bug，为了检验我们计算的梯度是否正确，可以将解析梯度和数值梯度进行比较。这里有一些实现的 tips：

- 计算 numerical gradient 时，使用 centered formula，即：

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \approx \frac{f(x+h) - f(x-h)}{2h}$$

  而非 $\frac{\mathrm{d}f(x)}{\mathrm{d}x} \approx \frac{f(x+h)-f(x)}{h}$，这是因为前者误差为 $O(h^2)$，而后者有 $O(h)$. （泰勒展开即可证）
- 使用相对误差作为比较基准。如果使用绝对误差，比如 10^{-4}，那么在梯度本来就小于 10^{-4} 的

地方，这个阈值没有什么作用。因此，我们使用相对误差：

$$\frac{|f'_{\text{analytic}} - f'_{\text{numerical}}|}{\max\left(|f'_{\text{analytic}}|, |f'_{\text{numerical}}|\right)}$$

分母用 max 或者加和或者取任意一个都是可以的，但是要小心除零。

- 注意目标函数的不可导点。跨越了不可导点的 numerical gradient 可能和实际的 gradient 有较大差距，如果误差较大需要检查是不是这个原因。
- 只使用少量数据。解决上一条问题的一个方法是 gradient checks 时只用少量的数据，这样目标函数的不可导处将减少；同时只用检验 $2 \sim 3$ 处的梯度就够了。
- h 不宜设置得太小。理论上 h 越小越精确，但由于计算机浮点数的精度误差，太小了可能反而不精确。
- 小心正则化项占了梯度的主要部分。如果正则化项在梯度中占比很大，它可能掩盖住了 back propagation 的错误。一个方式是在梯度检验时去掉正则化。
- 记住去掉 dropout/augmentations. 梯度检验时需要去除带有**随机性质**的量，比如 dropout 或者数据的随机 augmentation. 提前设置一个随机数种子也是可以的。

3.8 Sanity Checks

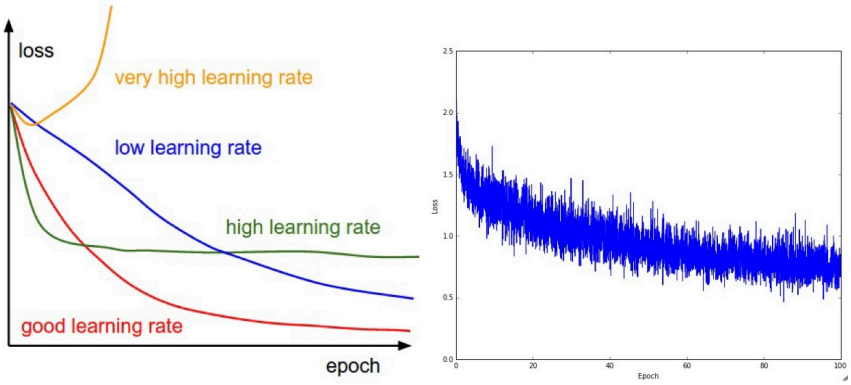
在开始训练之前，可以先进行 sanity checks，以保证我们的代码是正确的。

- 按概率推测损失函数初始值。例如一个十分类问题，随机初始化后我们应期望得到 1/10 的正确率，也即负对数概率的损失函数大致为：$-\ln(0.1) = 2.302$.
- 过拟合一个极小的数据集。用一个极小的数据集去训练，在去除正则化的情况下，我们应该期望得到 100% 的正确率或损失函数为 0，即在这个极小的数据集上过拟合。

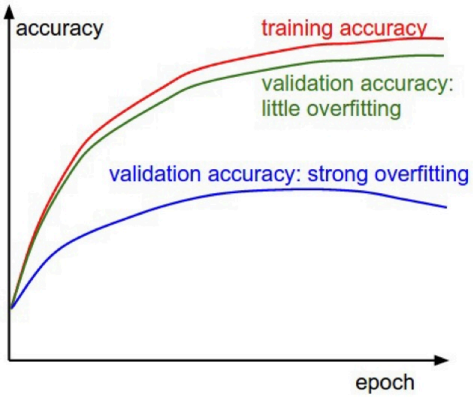
3.9 Babysitting the learning process

学习过程中我们可以监测一些值随着迭代次数的增加的变化。

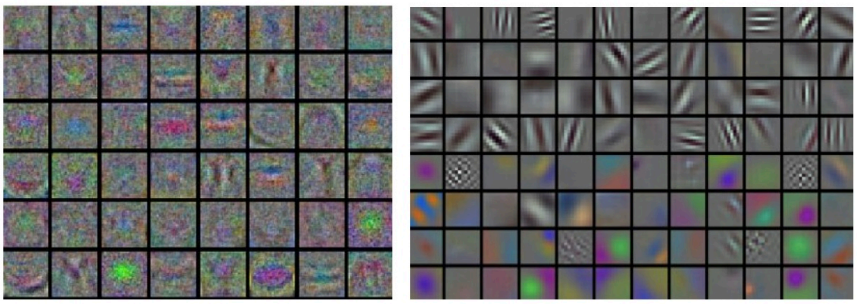
- **Loss function**. 作 loss - epoch 图能直观的告诉我们学习的状况，帮助我们调整学习率等参数。



- **Train/val accuracy**. 作训练集和验证集上的准确率的图能直观的告诉我们是否过拟合/欠拟合，帮助我们调整正则化项。



- **Ratio of weights updates**. 检测参数更新的幅度，帮助我们判断学习率是否合适，合适的幅度应该在 10^{-3} 左右。
- **First-layer Visualizations**. 在图像处理的网络中，我们可以把第一层的神经元的参数还原成图像可视化。



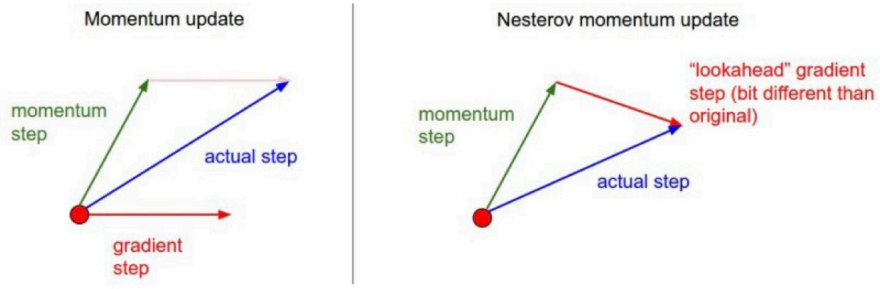
3.10 Parameter updates

SGD 及其延伸

- **Vanilla update**. 朴素的随机梯度下降
- **Momentum update**. 朴素的 SGD 在每一处都向梯度下降的方向更新，就像一个人下山一样。而如果我们做另一个比喻，想象一个小球从山坡滚下，它在每一处将具有保持原方向的惯性，这就是 momentum SGD 的灵感来源。具体地，momentum SGD 更新如下：

$$v := \mu v - \alpha \, \mathrm{d}x$$
$$x := x + v$$

  其中，α 为学习率，μ 可以看作是摩擦因子。
- **Nesterov Momentum**. 这是另一种 momentum update. 不同于上一条，Nesterov Momentum 计算随“惯性”移动一段距离之后的梯度再更新，见下图：



具体地，Nesterov Momentum 更新如下：

$$\begin{aligned} x_{\text{ahead}} &:= x + \mu v \\ v &:= \mu v - \alpha \, \mathrm{d}x_{\text{ahead}} \\ x &:= x + v \end{aligned}$$

学习率退火 在学习过程中不断减小学习率往往有帮助，一下三种方式是常用的 learning rate decay 实现方式：

- **Step decay**. 每迭代几次后将 learning rate 减小，例如每 20 次迭代后减小到原来的 0.1，或者每 5 次迭代减小一半等。这些参数依赖于具体问题和模型。一种启发式方法是当 validation error 停止减小时就减小 learning rate.
- **Exponential decay**. 依 $\alpha = \alpha_0 e^{-kt}$ 减小，其中 α_0, k 是超参数，t 是迭代次数。
- $1/t$ **decay**. 依 $\alpha = \alpha_0/(1+kt)$ 减小。

实践中，step decay 稍稍更受喜爱一些，因为其解释性比其他的强。

二阶方法 梯度下降及其延伸算法只需要用到一阶导数，即梯度。还有些最优化算法需要用到二阶导数。例如 Newton's method：

$$x := x - [Hf(x)]^{-1} \nabla f(x)$$

其中，$Hf(x)$ 表示 Hessian matrix，$\nabla f(x)$ 表示 $f(x)$ 的梯度向量。

注意到上述更新中不含有学习率这一超参数，这也是其优于一阶方法所在。但是，Newton's method 有一个及其显著的缺点，即 Hessian matrix 过于庞大，n 个参数的 Hessian matrix 是 $n \times n$ 的，计算耗时且难以存储。因此，出现了众多 quasi-Newton methods，例如 L-BFGS，它们不需要将 Hessian matrix 完整计算出来.

逐参数自适应学习率 在之前的算法中，学习率是全局的且对所有参数都一样。调整学习率是一个耗时耗力的过程，因此人们发明了许多自动调整学习率的方法，且这些学习率甚至能对每一个参数进行调整。虽然这些方法也有超参数，但它们与纯粹的学习率相比，能在更大的范围内表现较好，因而更好调参。

- **Adagrad**：

$$\begin{aligned} \text{cache} &:= \text{cache} + (\mathrm{d}x)^2 \\ x &:= x - \alpha \, \frac{\mathrm{d}x}{\sqrt{\text{cache}} + \text{eps}} \end{aligned}$$

  注意，上式中对向量的运算符定义为与 `numpy` 相同。对上式的理解是：cache 一路记录了每个参数的梯度平方和，如果某个参数有较大的梯度，那么它的学习率将减小；相反，梯度较小的参数将有较大的学习率。

- **RMSprop**：RMSprop 在 Adagrad 的基础上进行了简单的修改：

$$\text{cache} := \text{decay rate} \times \text{cache} + (1 - \text{decay rate}) \times (\mathrm{d}x)^2$$
$$x := x - \alpha \frac{\mathrm{d}x}{\sqrt{\text{cache}} + \text{eps}}$$

  也即是增加了 decay rate 这一超参数，一般取值为 $0.9, 0.99, 0.999$.
- **Adam**：可以看作是 RMSprop 带 momentum 的版本，其（简化的）更新如下：

$$m := \beta_1 m + (1 - \beta_1)\mathrm{d}x$$
$$v := \beta_2 v + (1 - \beta_2)(\mathrm{d}x)^2$$
$$x := x - \alpha \frac{m}{\sqrt{v} + \text{eps}}$$

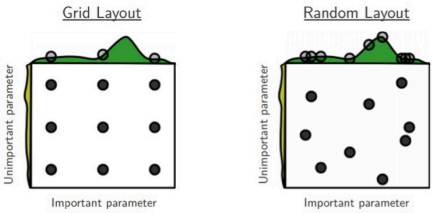
  典型的参数是：$\text{eps} = 10^{-8}, \beta_1 = 0.9, \beta_2 = 0.999$.
  完整的 Adam 算法存在一个 bias correction 机制：

$$m := \beta_1 m + (1 - \beta_1)\mathrm{d}x$$
$$mt := m/(1 - \beta_1^t)$$
$$v := \beta_2 v + (1 - \beta_2)(\mathrm{d}x)^2$$
$$vt := v/(1 - \beta_2^t)$$
$$x := x - \alpha \frac{mt}{\sqrt{vt} + \text{eps}}$$

3.11 Hyperparameter optimization

超参数的优化是一个黑盒优化问题，其优化的目标函数是神经网络的性能，因而有一个显著的特点：得到每一组超参数的结果会花费巨额的时间。因此我们并不能直接套用一般的优化方法对超参数进行优化。一般的，超参数优化方法有以下几种：

- **网格搜索**：即每个超参数设置几个值，暴力遍历所有的可能。
- **随机搜索**：在论文 Random Search for Hyper-Parameter Optimization by Bergstra and Bengio 中，作者论证了随机搜索比暴力遍历更为优秀，且更容易实现。



- **贝叶斯超参数优化**：利用贝叶斯方法进行超参数的寻找。

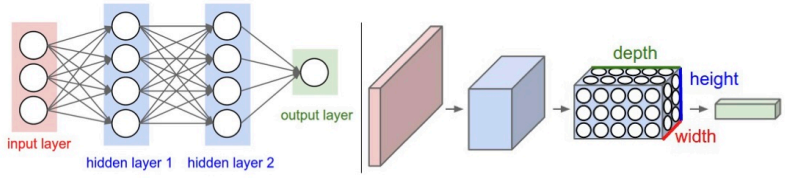
3.12 Model Ensembles

实践中，一个提升神经网络性能的方法是训练多个独立的模型，使用它们的平均预测结果。这些独立的模型可以是不同的模型，同一种模型、不同初始化条件，甚至是同一个模型在训练的不同时间点处等。除了对模型结果进行平均，还可以对各模型的参数进行平均，有时也能提高神经网络性能。

4 Convolutional Neural Networks

4.1 Architecture Overview

在图像处理的任务中，图片的特征具有局部性，而全连接的传统神经网络引入了太多冗余的参数，既浪费又难以训练。因此，CNN 被提出以解决这个问题。CNN 的架构和传统神经网络架构类似，都具有输入层、隐藏层和输出层。不同的是，CNN 中每一层都是 3 维的神经元，如下图所示：



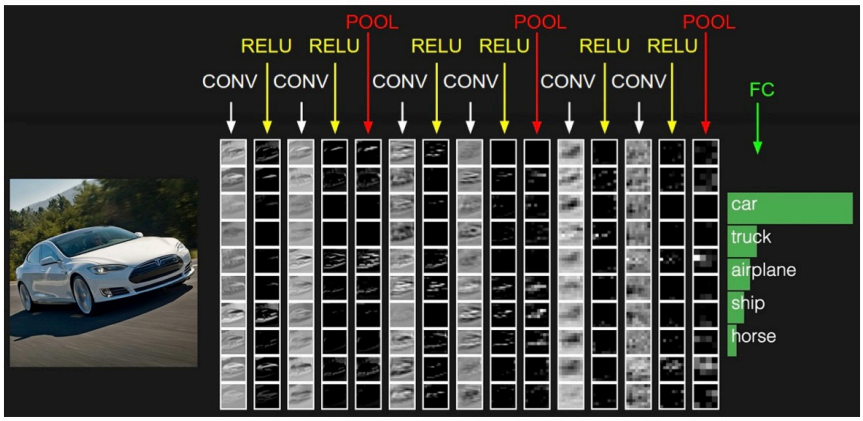
例如，CIFAR-10 数据集的输入可以是 $32 \times 32 \times 3$ 的，即 RGB 三个通道分别有 32×32 的像素；其输出可以是 $1 \times 1 \times 10$，分别表示 10 个分类上的得分。

下面来详细分析 CNN 中各种层的结构。

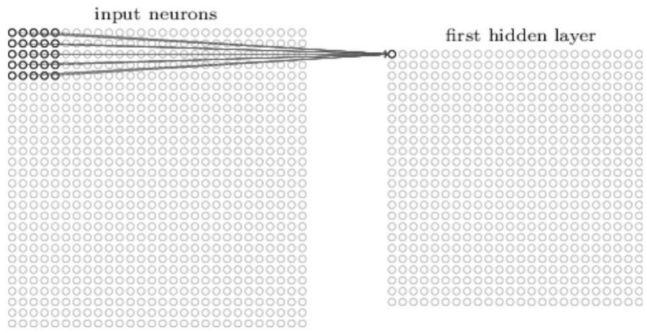
4.2 Layers used to build CNN

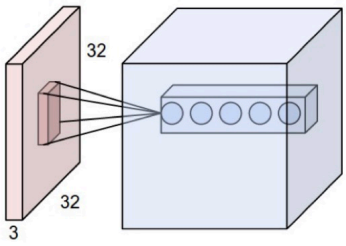
CNN 中主要有三种层：**Convolutinal Layer, Pooling Layer, Fully-Connected Layer**，将它们堆叠在一起就可以形成一个 CNN 的架构。

例如，[INPUT - CONV - RELU - POLL - FC] 就是一种简单的 CNN 架构。其中，CONV / FC 层具有需要学习的参数，而 RELU / POOL 层是固定的函数；CONV / FC / POOL 层都有超参数，而 RELU 没有。



Convolutional Layer Convolutional Layer 是 CNN 的核心。其基本思想很简单，就是用一个**卷积核**（**filter / kernel**）在输入图像上扫一遍，得到输出图像，其中卷积核的参数就是神经网络要学习的参数。设输入深度为 D_1，则这个卷积核的深度也是 D_1，即将输入图像某一区域**所有层**的数据一起做线性组合；如果输入的深度是 D_2，那么我们需要 D_2 个不同的卷积核，形成输出的不同层。





正式地说，设输入是一个 $W_1 \times H_1 \times D_1$ 的 3 维数组（即上一层），Convolutional Layer 有以下几个超参数：

- K：filters 的个数
- F：filters 的边长
- S：filters 进行扫描时的步长
- P：zero padding，在输入图像的边缘补充的长度

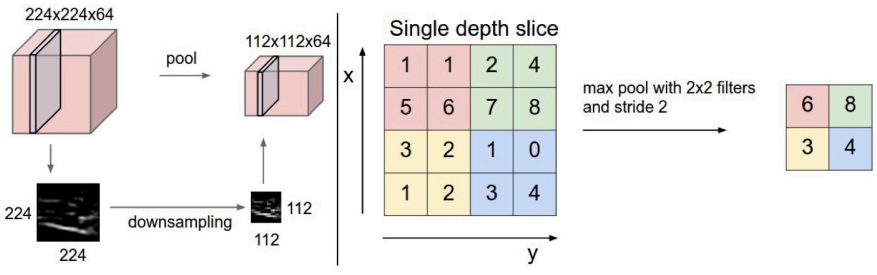
输出一个 $W_2 \times H_2 \times D_2$ 的 3 维数组，其中：

$$W_2 = (W_1 - F + 2P)/S + 1$$
$$H_2 = (H_1 - F + 2P)/S + 1$$
$$D_2 = K$$

在进行卷积操作时，我们需要注意计算得到的 W_2 和 H_2 是不是整数，如果不是，需要调整 P 或者 S。另外，注意对于输出的每一个 depth slice，其上的神经元是由同一个卷积核算出来的，分享了同样的参数，这有助于减少 CNN 的参数数量。

im2col 是实现卷积操作的一个方法，正如其名 image to column，它将二维图像变成一个列向量以方便地实施卷积操作。举例说明，假设输入图像是 $[227 \times 227 \times 3]$，使用 $[11 \times 11 \times 3]$ 的卷积核并取步长为 4，想要得到 $[55 \times 55 \times 96]$ 的输出。那么，我们将 $[11 \times 11 \times 3]$ 的卷积核拉伸成 $11 \times 11 \times 3 = 363$ 维的**行向量**，各卷积核拼接起来得到 $[96 \times 363]$ 的参数矩阵；然后把输入图像中对应 55×55 个位置的 $[11 \times 11 \times 3]$ 全拿出来拉伸成**列向量**并顺次拼接成 $[363 \times 3025]$ 的矩阵；二者相乘即可得到 $[96 \times 3025]$ 的矩阵，每一行的列向量还原成 $[55 \times 55]$ 即得到输出结果。

Pooling Layer　插入 Pooling Layer 是为了减小当前层的大小，与 Convolutional Layer 类似，它用一个 filter 扫描整个图像，然后做某种无参数的计算，如取 max (*max pooling*) 或者 L2 norm (*L2-norm pooling*) 或者平均值 (*average pooling*).



Fully-connected Layer CNN 中的全连接层和传统神经网络的全连接层一模一样，一般最后连接输出层的时候用全连接层。

FC Layer 可以和 Conv Layer 相互转化：

- Conv 转 FC：只需要强行设定 FC 中某些权重为 0；
- FC 转 Conv：设定 Conv 卷积核大小与输入相同，那么一次卷积运算相当于 FC 前一层到后一层的某一个神经元的运算。

4.3 CNN Architectures

上一节讲了 CNN 中主要的三种层，这一节讲如何把它们的组合在一起形成 CNN.

Layer Patterns 最常见的各种层的组合是：

$$\texttt{INPUT} \to [[\texttt{CONV} \to \texttt{RELU}] \times N \to \texttt{POOL}] \times M \to [\texttt{FC} \to \texttt{RELU}] \times K \to \texttt{FC}$$

即若干 CONV-RELU layers，选择性地接 POOL layer，重复该结构直到最后接上传统的神经网络。

实践中注意一点：相比用一个具有较大卷积核的卷积层，更好的是使用多个具有较小卷积核的卷积层堆叠起来，因为一方面，堆叠起来的层引入更多非线性因素，使神经网络更强大；另一方面，后者具有更少的参数。

Layer Sizing Patterns

- **input layer**：最好能被 2 整除很多次；
- **conv layers**：应使用小的 filters（如 3×3 或最多 5×5），使用步长 $S = 1$，并且使用 zero padding 保证输出图像的大小和输入的大小相等（取 $P = (F - 1)/2$ 即可），这样能够防止略过图像边缘的信息，而把缩小 feature map 的任务交给 Pooling layer；
- **pool layers**：负责缩小输入的大小。最常用一个 2×2 的 filter 以步长为 2 扫描一遍，丢掉恰好 75% 的结果；也可以使用 3×3 的 filter，但是基本不用 > 3 的 filter，因为这样太过 aggresive 而使得结果变差。

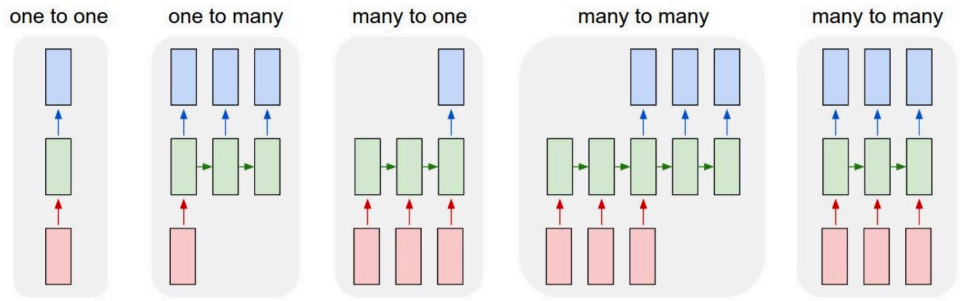
Case studies 有一些经典的 CNN 结构有一个名字，例如：

- **LeNet**：由 Yann LeCun 在 1990's 实现。
- **AlexNet**：由 Alex Krizhevsky, Ilya Sutskever 和 Geoff Hinton 实现，在 2012 年 ImageNet ILSVRC 比赛中以绝对优势夺冠，并使得 CNN 从此广受欢迎。
- **ZF Net**：由 Matthew Zeiler 和 Rob Fergus 实现，在 ILSVRC 2013 夺冠。
- **GoogLeNet**：由 Szeged 等人实现，在 ILSVRC 2014 夺冠。
- **VGGNet**：由 Karen Simonyan 和 Andrew Zisserman 实现，是 ILSVRC 2014 的第二名。
- **ResNet**：由何恺明等人实现，在 ILSVRC 2015 夺冠。

5 Recurrent Neural Networks

5.1 RNN: Process Sequences

Overview 所谓循环神经网络，可以看作是有时序性的神经网络，有时序电路的那种感觉。



上图中，横向从左到右可以看作为若干时刻。普通的神经网络是 one to one 的结构——一个输入层，经过一系列隐藏层，到达一个输出层，这些步骤都是在一个时刻完成的；而 RNN 可以处理序列型的数据，其可以是 one to many, many to one, many to many 等结构。

- one to many: 某一时刻给一个输入，在之后的若干时刻都有输出。例如 image captioning.
- many to one: 在连续的几个时刻给输入，直到最后一个时刻给出输出。例如 audio prediction.
- many to many: 在连续的几个时刻给输入，在之后的若干时刻都有输出。例如 video captioning.
- many to many: 在连续的几个时刻给输入，同时不断地输出。例如 frame level video classification.

Forward RNN 向前传播的 key idea 是：每一个神经元有一个“隐藏”的和时序相关的向量 h_t，它根据某**不随时序变化**的参数 W 和当前的输入 x_t 更新，即：

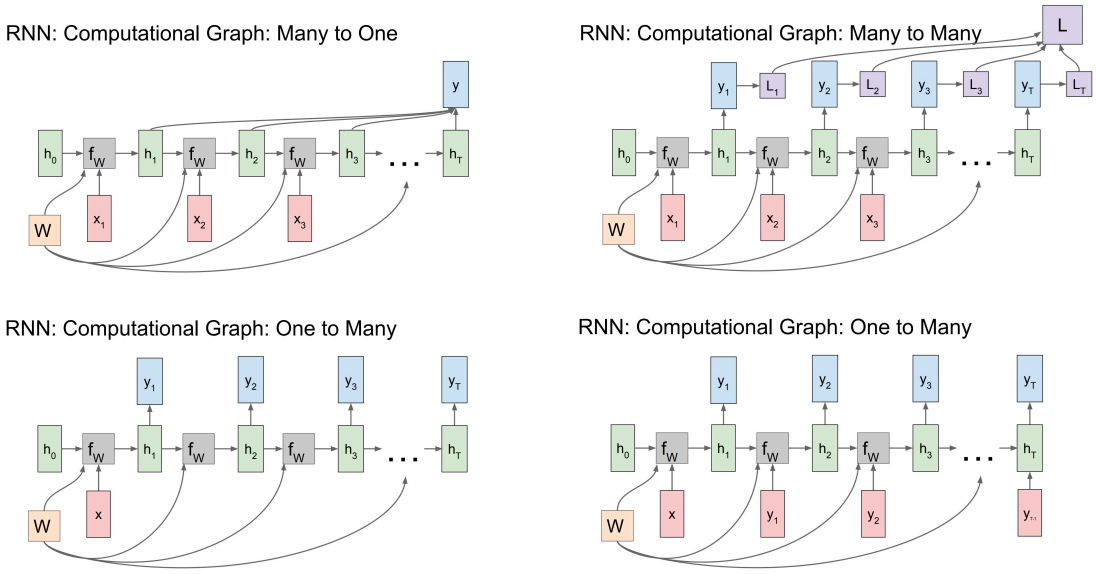
$$h_t = f_W(h_{t-1}, x_t)$$

随后可以根据情况（one to many / many to one / many to many）决定如何用 h_t 去更新输出 y_t.

例如，一个简单的情形可以是：

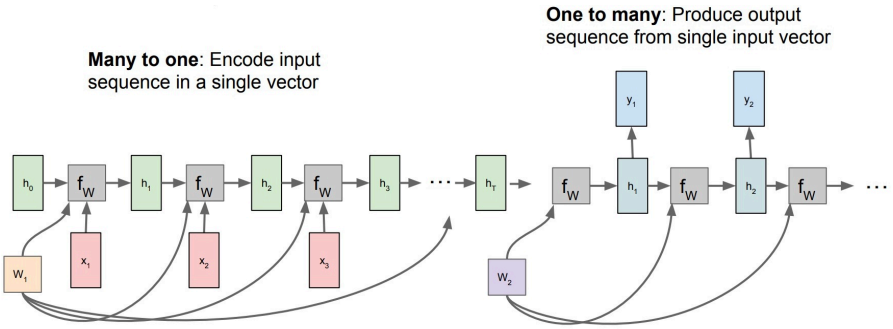
$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\ y_t &= W_{hy}h_t \end{aligned}$$

Computational Graph 为了方便 Backpropagation 的推导，computational graph 是非常重要的技巧。显然，对于不同结构（one to many / many to one / many to many, etc.），它们的 computational graph 会不同，但大同小异：

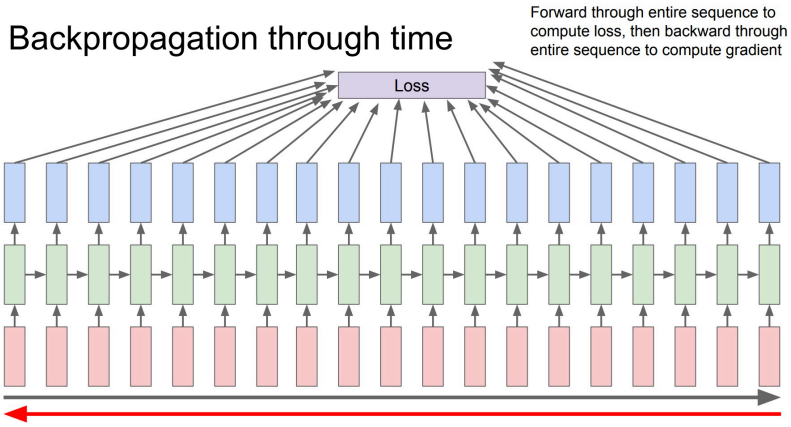


注意，在 one to many 结构中，我们可以用前一时刻的输出作为下一时刻的输入。

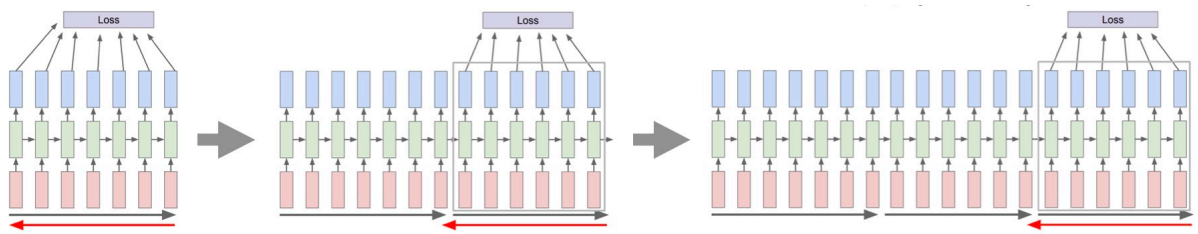
另外我们还可以把 many to one 和 one to many 连起来，形成 sequence to sequence 的效果。



Backpropagation 向前传播是按照时序计算的，于是反向传播就逆着时序传播。



但是这里有一个问题，如果时序序列很长，这个过程会占用很大的内存。解决方案是 **Truncated** Backpropagation，把整个时序分段，每次向前传播一段后就对这段反向传播。



RNN Tradeoffs RNN 的优点有：

- 可以处理任意长度输入。
- 时间步 t 理论上可以使用前序时间步的所有信息。
- 模型大小不随输入长度的增加而增加。
- 每个时间步用同一套模型权重，对输入的处理存在一种对称性。

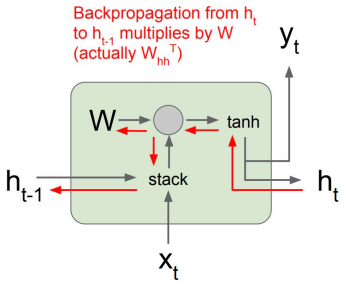
RNN 的缺点有：

- 循环计算较慢
- 实际使用中，难以获取到非常多步之前的信息。

5.2 LSTM (Long Short Term Memory)

RNN Gradient Flow RNN 的在一个时钟中的更新为：

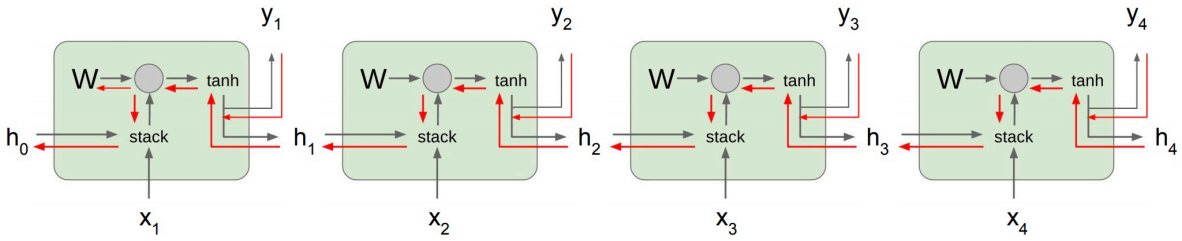
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) = \tanh\left(W\begin{pmatrix}h_{t-1}\\x_t\end{pmatrix}\right)$$



在这一个时钟中，我们有：

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'\left(W_{hh}h_{t-1} + W_{xh}x_t\right)W_{hh}$$

考虑整个时序：



我们 Backpropagation 的目的是找到 $\partial L/\partial W$：

$$\frac{\partial L}{\partial W}=\sum_{t=1}^{T}\frac{\partial L_t}{\partial W}$$

若仅考虑 $\partial L_T/\partial W$：

$$\begin{aligned}\frac{\partial L_T}{\partial W}&=\frac{\partial L_T}{\partial h_T}\frac{\partial h_T}{\partial h_{T-1}}\cdots\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W}\\&=\frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T}\frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial h_1}{\partial W}\\&=\frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T}\tanh'(W_{hh}h_{t-1}+W_{xh}x_t)\right)W_{hh}^{T-1}\frac{\partial h_1}{\partial W}\end{aligned}$$

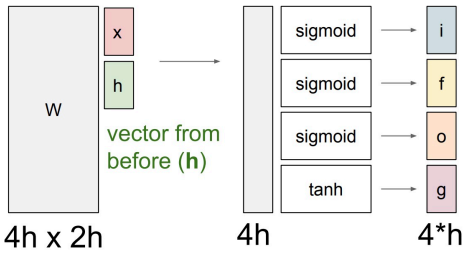
由于 $\tanh'(x)\leqslant 1$（当且仅当 $x=0$ 时取等），所以上式中括号内的乘积将非常小，这导致**梯度消失**。即便不考虑括号那一项，注意 W_{hh}^{T-1} 这一项，如果 W_{hh} 的最大奇异值 >1，该项将很大，导致**梯度爆炸**；而如果 W_{hh} 最大奇异值 <1，该项将很小，导致**梯度消失**。总而言之，梯度在 RNN 中的传播是困难的，于是我们思考改进 RNN 的结构来解决这个问题。

LSTM　LSTM 在普通 RNN 的基础上多加了四个中间变量，将一个时钟中的更新定义为：

$$\begin{cases}\begin{pmatrix}i\\f\\o\\g\end{pmatrix}=\begin{pmatrix}\sigma\\\sigma\\\sigma\\\tanh\end{pmatrix}W\begin{pmatrix}h_{t-1}\\x_t\end{pmatrix}\\c_t=f\odot c_{t-1}+i\odot g\\h_t=o\odot\tanh(c_t)\end{cases}$$

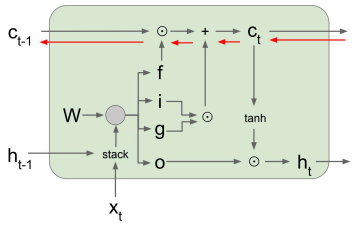
其中：

- i: Input gate, whether to write to cell
- f: Forget gate, whether to erase cell
- o: Output gate, how much to reveal cell
- g: how much to write to cell

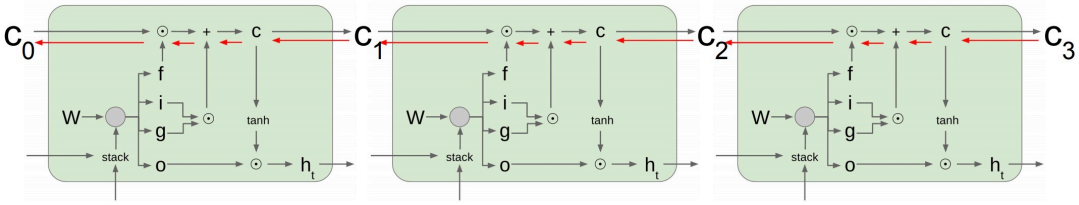


注意，计算上述四个 gate 各自的 W 是不同的，而上式中的 W 表示把它们写在一起的矩阵。

在一个时钟中，LSTM 的从 c_t 到 c_{t-1} 的梯度更新为：



整个时序上，gradient flow 显得很顺畅：

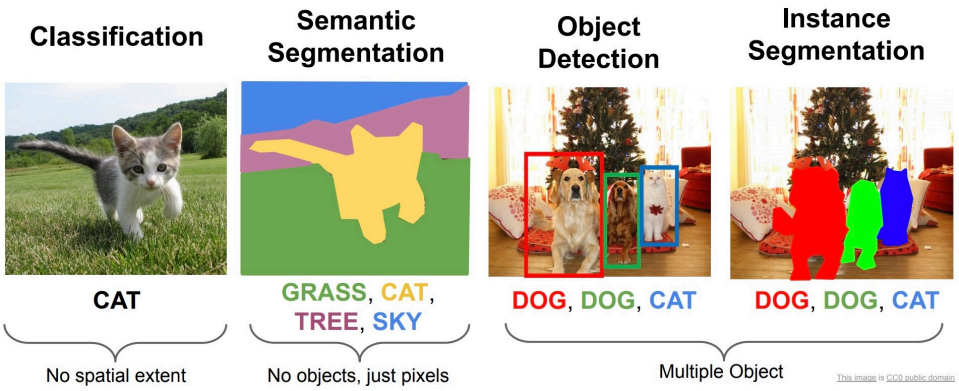


虽然 LSTM 不能保证不会发生梯度爆炸或梯度消失，但是它的 gradient flow 机制确实使得神经网络更容易训练。梯度在 LSTM 中的反向传播好似走了一条 high way，这一点上 LSTM 与 ResNet 有异曲同工之妙。

6 Detection and Segmentation

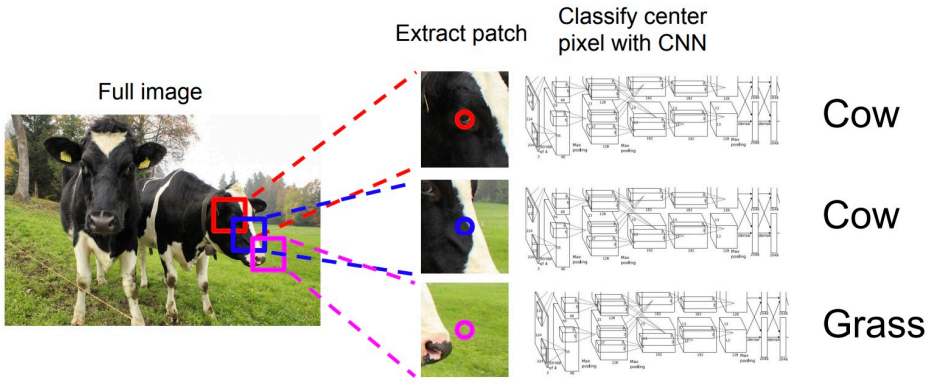
6.1 Overview

计算机视觉中的四个典型任务：分类、语义分割、目标检测、实例分割。

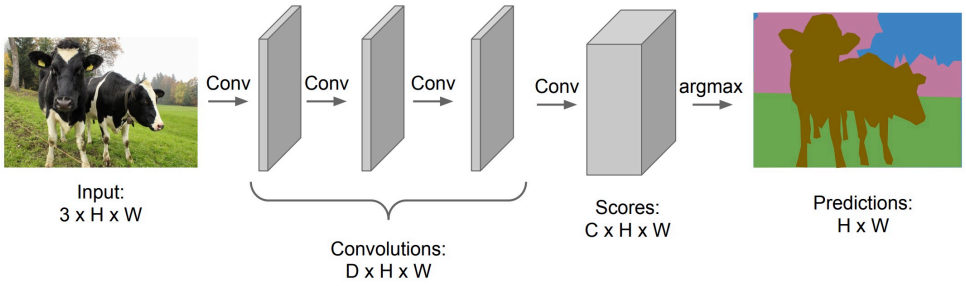


6.2 Semantic Segmentation

对图像的每一个像素进行分类。



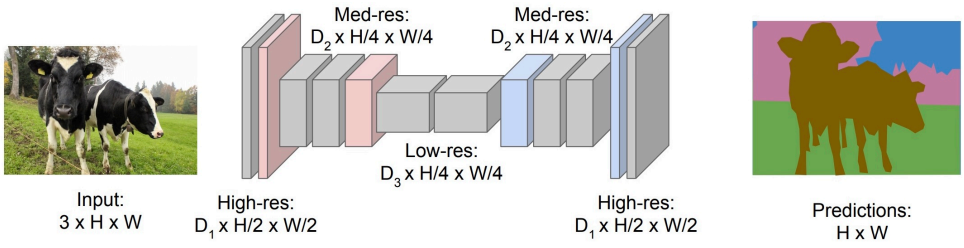
Idea 1: Sliding Window 对每一个像素点用一个神经网络做一次分类。效率很差，进行了许多重复计算，nobody use it.



Idea 2: Fully Convolutional 经过一个卷积神经网络，每一个卷积层都**保持原图像的大小**，最终对每个像素都得到其各分类的得分，从而得到每个像素的分类结果。

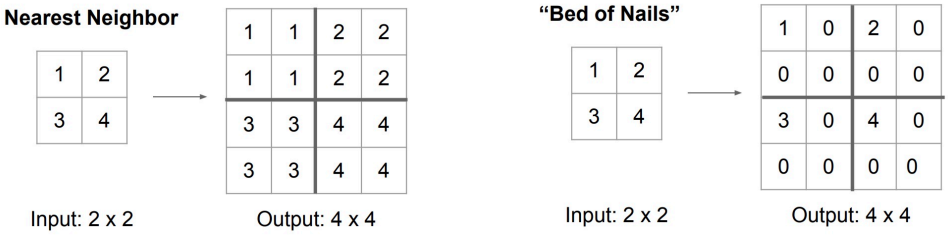
问题：卷积非常耗时，因为每一个卷积层都保持原图像大小。

为了解决这个问题，我们可以先通过 max-pool 或者 conv 层进行 **downsampling**，然后再 **upsampling** 回原大小：

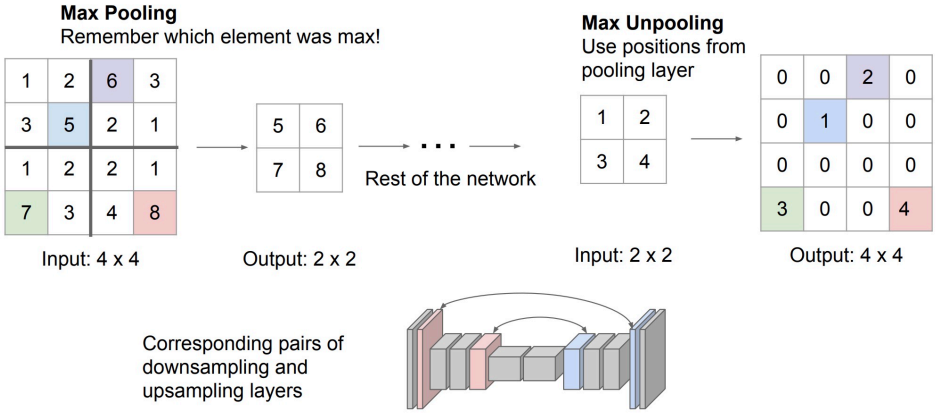


我们很清楚如何 downsampling，问题在于如何进行 **upsampling**：

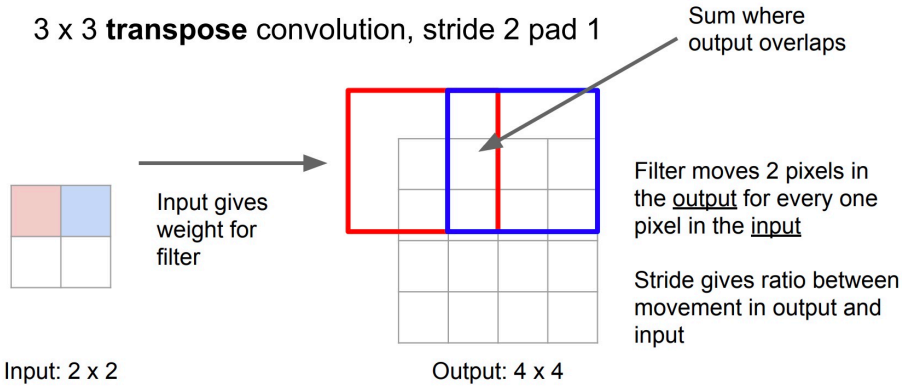
- **Unpooling**：



- **Max-unpooling**：



- **Transpose Convolution**：
  视输入矩阵为 filter 的权重，按权重将 filter 累加到输出矩阵中。



Transpose Convolution 只能恢复卷积前后的大小，并不能恢复值，其原理是：

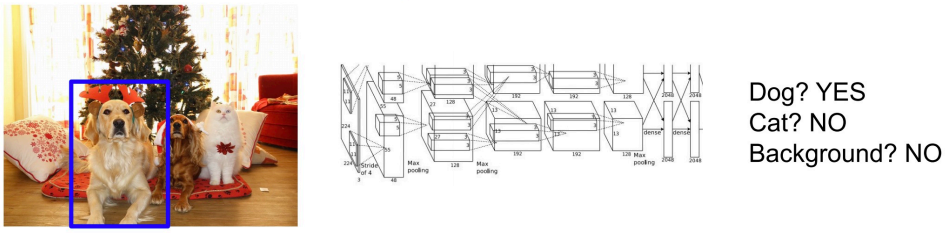
在 Convolution 操作时，我们可以将输入图像拉伸为 1 维向量 $X \in \mathbb{R}^D$，输出图像拉伸为 1 维向量 $Y \in \mathbb{R}^{D'}$，则卷积核可以写作一个稀疏矩阵 $C \in \mathbb{R}^{D' \times D}$，使得 $CX = Y$. 在 Upsampling 中，已知 C, Y，欲得到 X，Transpose Convolution 的操作是：$X = C^T Y$，这也是其名称的由来。显然这样做并没有恢复值，除非 $CC^T = I$.

6.3 Object Detection

Idea 1: Regression 构造神经网络输出检测物体的类别以及框住物体的矩形框的坐标 (x, y) 和大小 (w, h).

由于在 Object Detection 中，我们并不能实现知道输入图像中有多少个 object，所以其输出大小是不固定的，并且当 object 很多时输出将很大。因此这不是一个很好的方法。

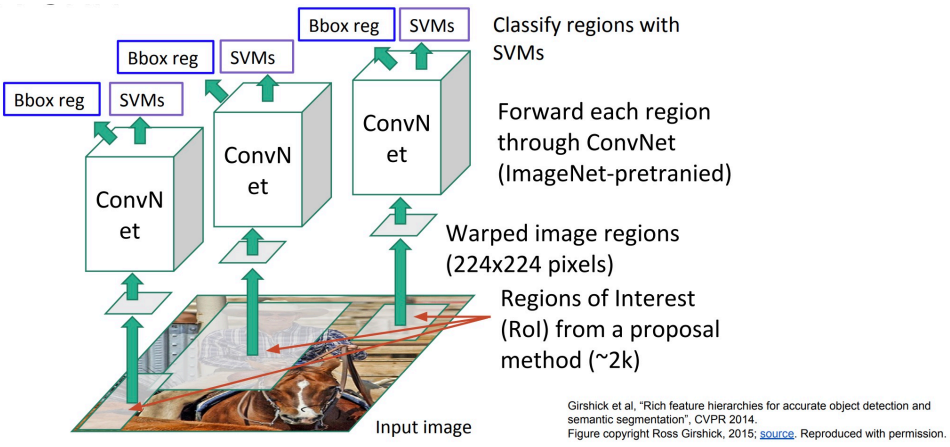
Idea2: Sliding Window 滑动一个矩形框，每次对矩形框内的图像进行分类。



该方法也有很明显的问题：矩形框的位置和大小都是未知的，我们需要暴力遍历各种矩形框，每次都通过一个巨大的 CNN 网络得到类别，这无疑十分耗时。

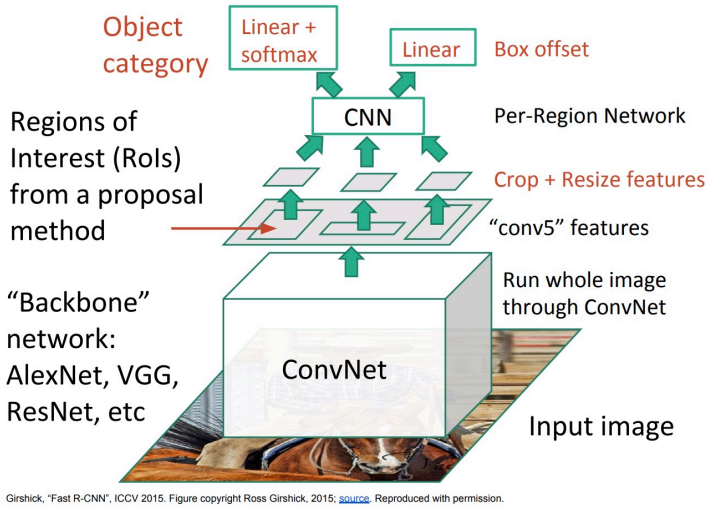
Idea 3: Region Proposals & R-CNN

- **Region Proposals**: 先通过一个网络给出物体可能存在的若干矩形范围，后续工作只需在这些范围内进行。**Selective Search** 是一种给出 Region Proposals 的方法，其运行速度非常快。
- **R-CNN**：在每一个给定的矩形范围内，我们可以构建 CNN 网络，输出该矩形内物体的类别以及这个矩形区域应该如何修正。注意由于每个矩形大小不同，而我们的网络要求特定大小的输入，所以我们会先对这些矩形进行变形使之符合网络的输入要求。具体见下图：



该方法的不足在于，对每个矩形区域进行计算依然十分耗时耗力。

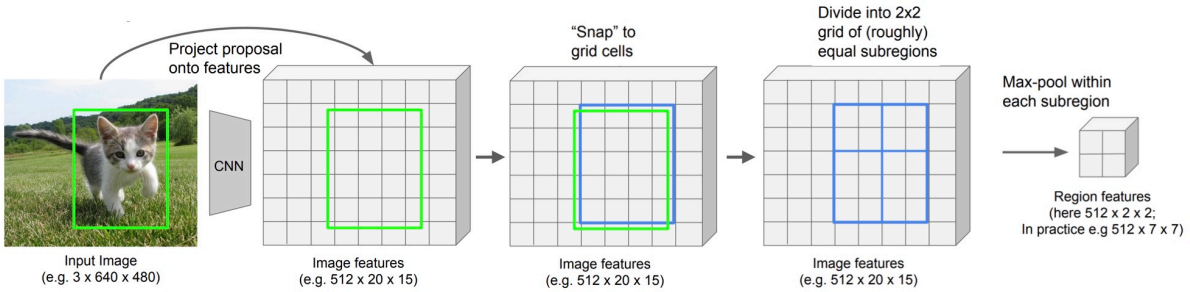
- **Fast R-CNN**：Idea 是我们先把图像输入进 CNN 网络，再对网络的输出结果进行矩形区域划分，这样图像只会经过一次 CNN 网络，从而提高运行速度。



Girshick, "Fast R-CNN", ICCV 2015. Figure copyright Ross Girshick, 2015; source. Reproduced with permission.

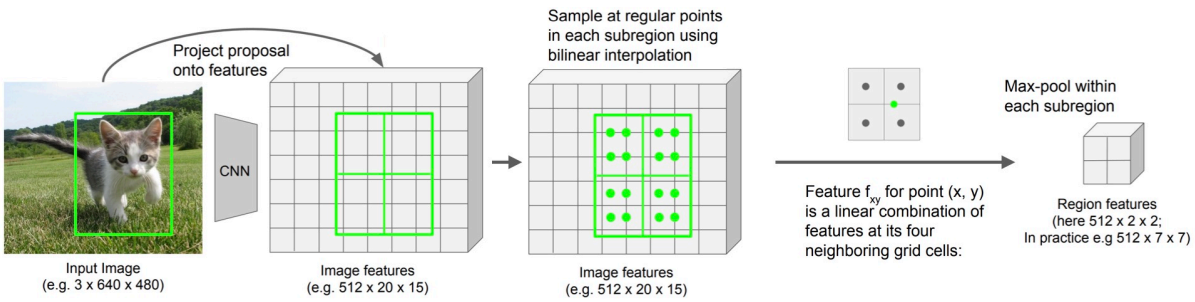
这里有个问题是如何在 CNN 网络的输出结果上进行 Region Proposals：

- **RoI (Region of Interest) Pool**：首先把原图像上的矩形区域按比例投影到输出矩阵上，然后取整使矩形端点落在整点上，对于这个矩形，我们将其尽可能地等分成需要的形状，并在每一块中作 max-pool，最终得到该区域的 feature.

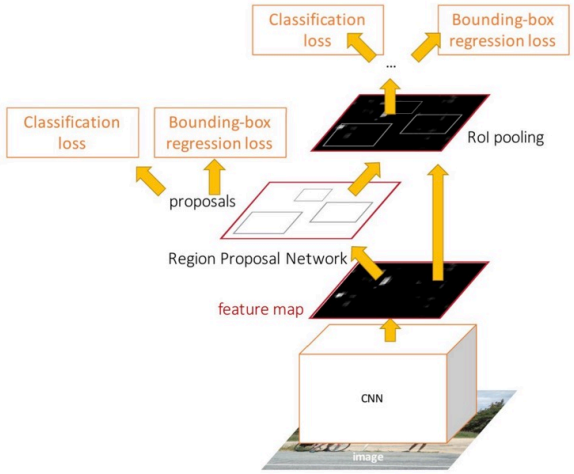


由于我们在这个过程中进行了两次近似——取整和划分，所以这种方法的问题在于我们得到的区域特征稍有偏离。

- **RoI Align**：仍然把原图像上的矩形区域投影到输出矩阵上，然后直接等距离划分成需要的形状；现在考察每一个小块，它们的顶点不一定在整点上，我们在其中取 4 个位置，对每个位置用它周围的 4 个整点的值作双线性插值，然后取 max-pool，最终得到该区域的 feature.

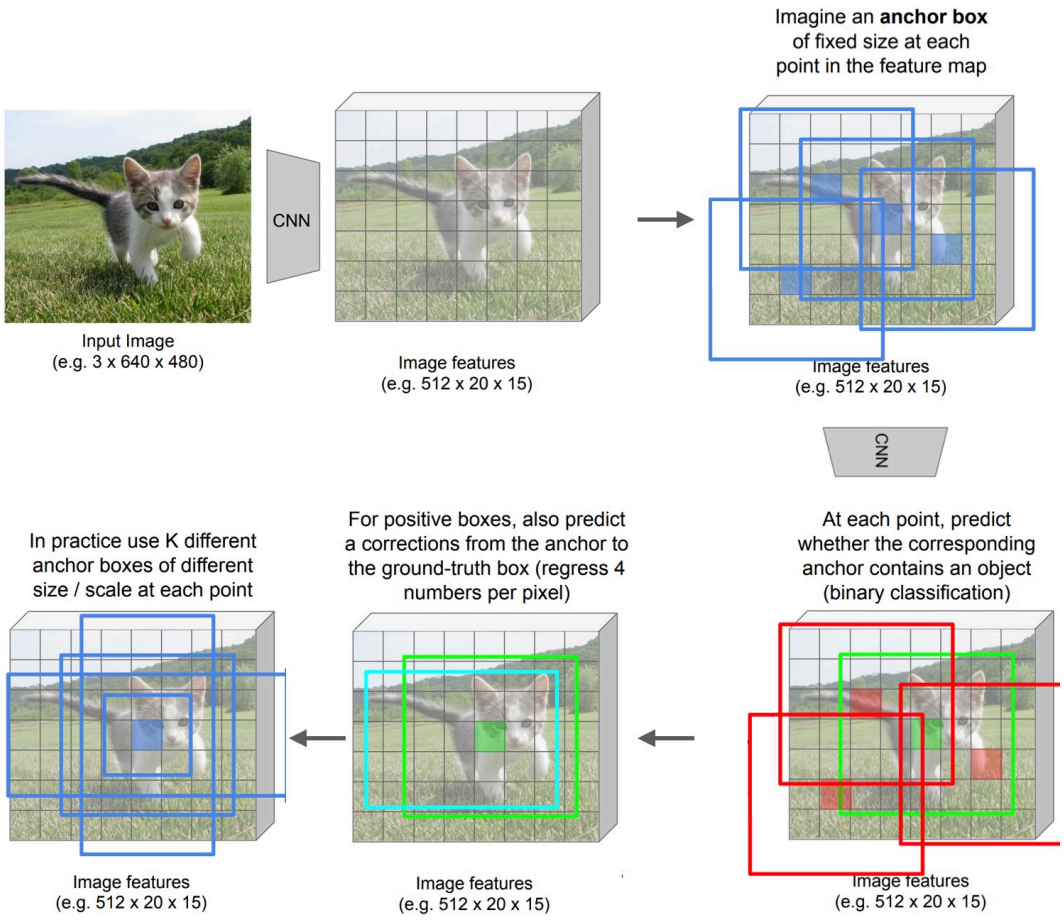


- **Faster R-CNN**：实测中发现，在 Fast R-CNN 中，Region Proposals 占了大部分时间。为了提高这部分的速度，Faster R-CNN 用一个 **Region Proposal Network** 在 CNN 的输出矩阵上作矩形区域划分，其余部分和 Fast R-CNN 相同。



Region Proposal Network (RPN) 的原理如下：

1. 想象以 feature map 的每一个位置为中心有固定大小的 anchor box，使用一个 CNN 预测这些 anchor box 中是否有物体，以及对 box 四边的修正值。
2. 对于每一个位置，计算 K 个不同的 anchor box，这样我们得到了 $K \times H \times W$ 个数值表示该 anchor 处是否有物体，以及 $4K \times H \times W$ 个数值表示 box 四边的修正值。
3. 取得分最高的约 300 个 anchor box 作为我们的 region proposals.



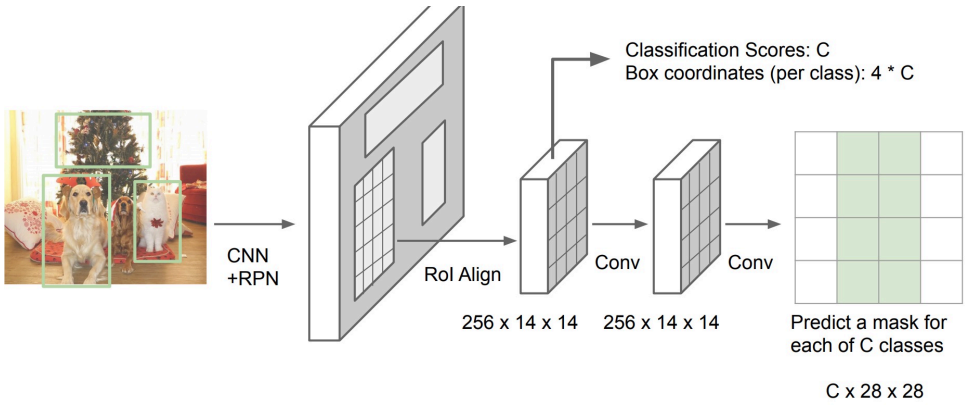
在训练时，我们会将四个 loss 一起训练——RPN 二分类（判断是否是物体）、RPN anchor box 位置的 loss、最终分类（判断是何物）的 loss、最终 box 的坐标的 loss.

- **YOLO / SSD / RetinaNet**：Faster R-CNN 其实有两个阶段：第一个阶段是得到 region proposals，通过一个基础的 CNN 和 RPN 完成，第二个阶段是对每一个 region 进行分类以及修正边界，通过 RoI pool / align 之后输入到 CNN 完成。而 YOLO / SSD / RetinaNet 等网络只用一个阶段，具体的内容不在此课程中讲授。

可以看出，Object Detection 有很多选择：首先，基础的网络有很多选择（VGG16 / ResNet-101 / Inception V2……）；接下来，我们可以选择二阶段的 Faster R-CNN 或者单阶段的 YOLO / SSD 或者混合的 R-FCN……这其中就有许多的 trade-offs. 例如，Faster R-CNN 比 SSD 更慢，但是准确率比后者更高；又如网络越大越深，效果越好，但训练也越难，耗时也越多……

这里有一篇综述论文：Zou et al, “Object Detection in 20 Years: A Survey” , arXiv 2019

6.4 Instance Segmentation



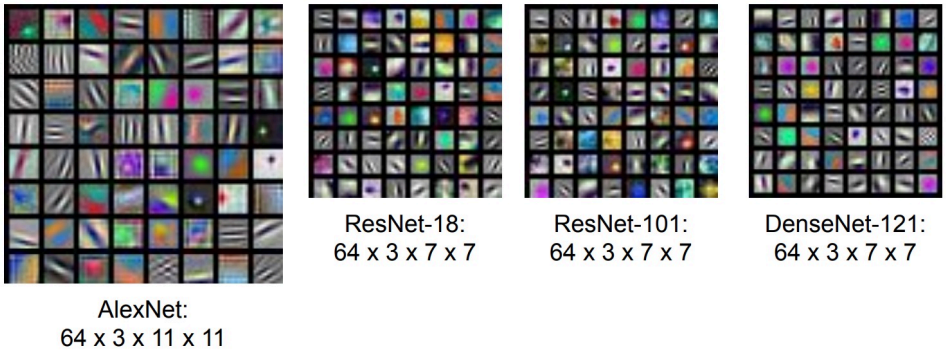
Mask R-CNN 类似于之前的 Faster R-CNN，不过最后一个卷积网络不是对区域内容进行分类，而是对每一类输出一个 mask。

Mask R-CNN 还可以做 pose estimation，即预测人体的姿势。

7 Visualizing and Understanding

7.1 First Layer: Visualize Filters

对于第一层卷积层，我们可以可视化每个卷积核的权重，并得到一系列可解释的图片：

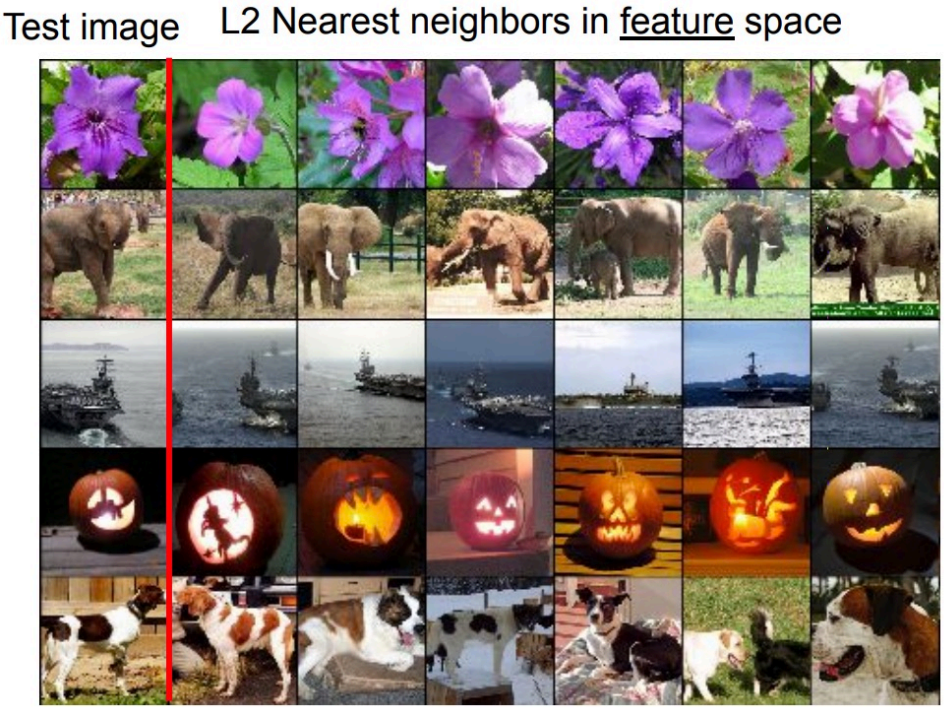


可以看出，第一层卷积层在寻找一些点与线，以及对立的颜色，可以认为是在寻找物体与物体之间的分割线。

为什么可视化卷积核的权重是有道理的呢？卷积核所做的就是与所有训练数据进行内积运算，而我们知道，两个向量越相近，其内积结果越大，因此，卷积核图片上的一条白线，可以认为是众多数据中存在这么一条线，使得卷积核学到了这一特征。

7.2 Last Layer

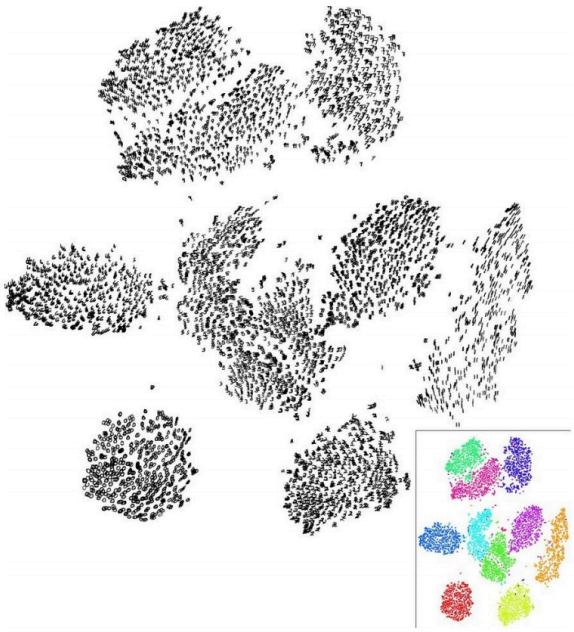
Nearst Neighbors 最后一层的输出其实可以看作是每个图像的特征向量，因此我们可以画出 nearst neighbors：



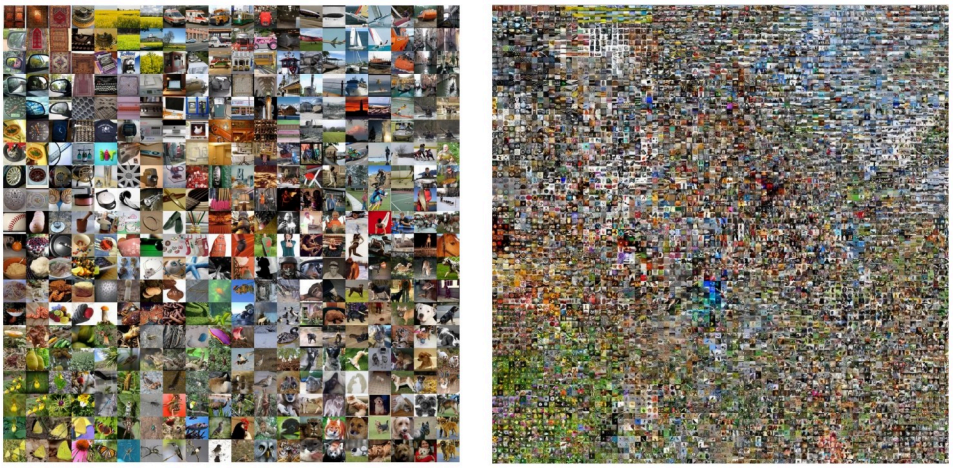
可以看见，这比我们在第一节课直接用像素空间进行距离的衡量要准确很多。

Dimensionality Reduction 将最后一层的向量使用降维方法，例如简单的 PCA 或更强大的 **t-SNE** 降维至 2 维并作图。

在 MNIST 上，我们可以得到：



在 CIFAR-10 上，我们把降维后的点放置在网格中（准确来讲，是找出与每个网格点中心最相似的图片），得到如下可视化结果（高清版本参见 `http://cs.stanford.edu/people/karpathy/cnnembed/`）：

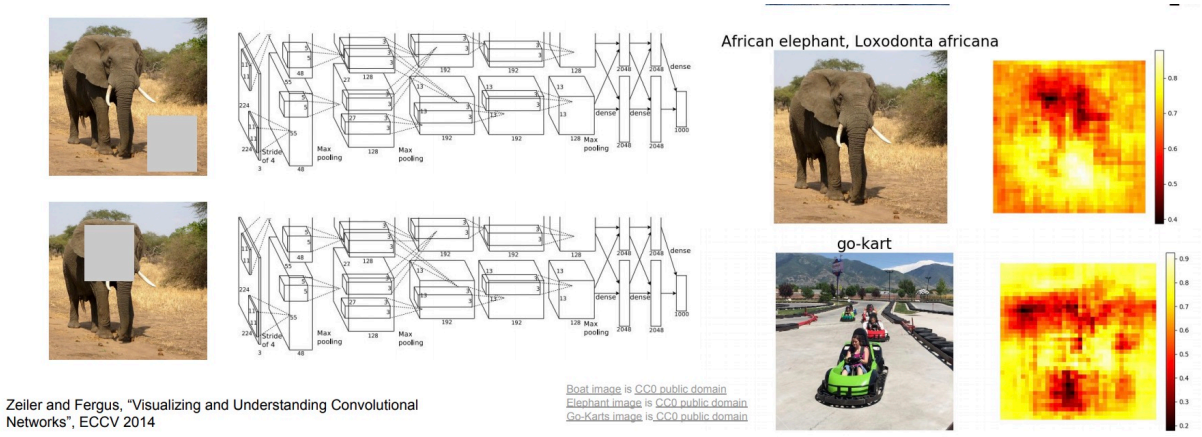


7.3 Maximally Activating Patches

从网络中选择某层的某个 channel，输入若干图片并记录这个 channel 上的数值。由于使用的是卷积神经网络，所以每一个神经元其实感受的是输入图像的某一些小片。将最大激活的图像小片可视化出来，可以得到：

7.4 Occlusion Experiments

遮盖住输入图像的一部分之后再输入进 CNN，记录分类结果的概率。可以据此做出遮盖不同位置导致不同概率的热点图，如下：

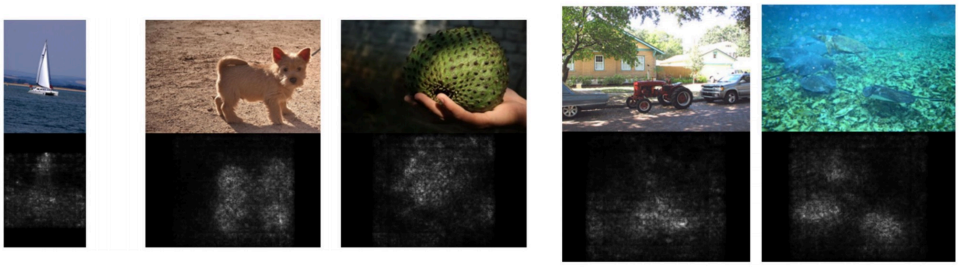


显然，那些导致分类概率急剧变小的像素就是图像里关键的地方。

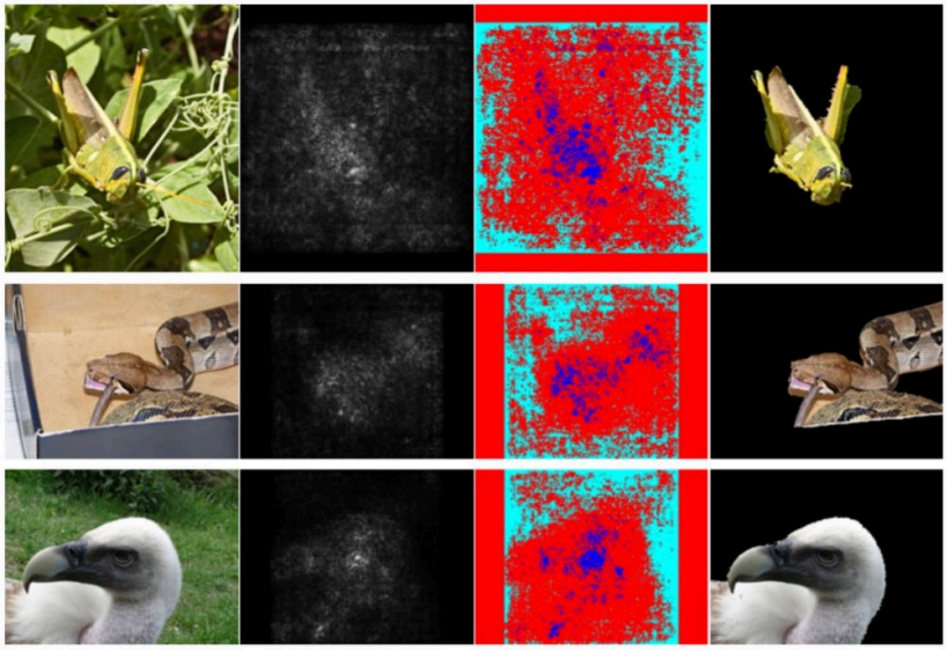
7.5 Saliency Maps

Saliency Maps 和 Occlusion 的目的一样，都是想知道哪些像素对分类结果很重要。

计算每一类的得分对图像像素的梯度，取绝对值并取 RGB 三个 channel 中的最大值。这样做让我们知道如果一个像素发生了微小扰动，相应类的分类得分将发生多大的变化。如果变化很大，自然这个像素对该分类非常重要。



由此，我们可以想到把 saliency maps 用在 semantic segmentation 中，并且这样做不需要 semantic segmentation 的训练数据。使用 **GrabCut** 算法可以做到这一点：

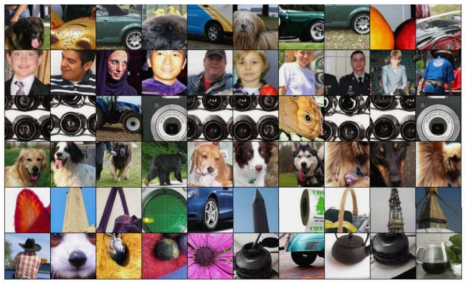


7.6 Intermediate features via (guided) backprop

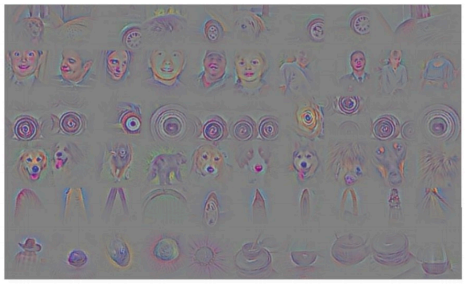
借助和 Saliency Maps 类似的想法，如果我们取出神经网络中的某一个神经元并计算其对输入图像各个像素的梯度，那我们也能知道哪些像素对这个神经元非常重要。然而，与通常的 back propagation 不同，我们这时使用 guided backprop，只对通过 ReLU 的正的梯度进行反向传播，这样得到的图像更加清晰。

Maximally activating patches
(Each row is a different neuron)

Guided Backprop



Maximally activating patches
(Each row is a different neuron)



Guided Backprop

7.7 Gradient Ascent

对于神经网络中的一个神经元，我们想知道什么类型的输入能够激活这个神经元。注意现在和之前的区别在于，我们现在不依赖于输入的数据，而是生成一个使得神经元激活的输入类型。为解决这个问题，我们可以采取 Gradient Ascent.

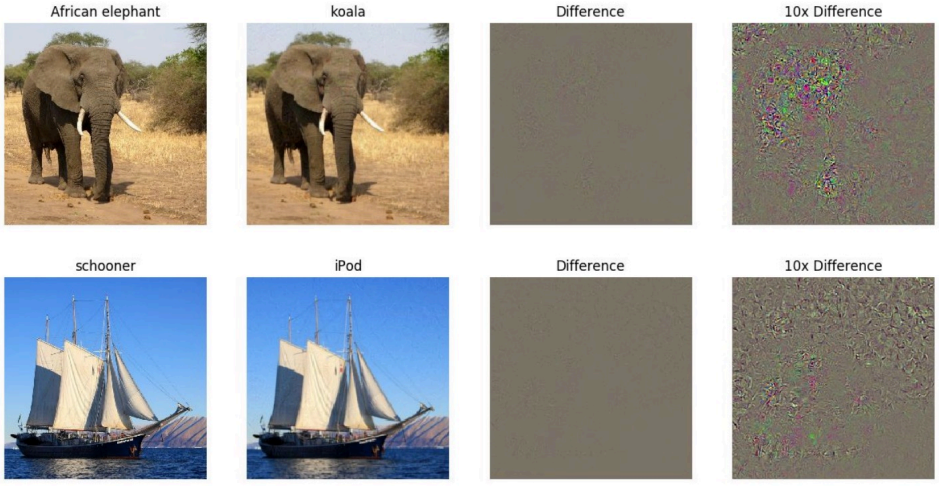
在 Gradient Descent 中，我们改变参数的值来最小化结果；而 Gradient Ascent 是一个相反的过程，我们固定参数而去改变输入图像的像素，以使得神经元被最大激活或者某一类的得分最大。同时我们需要正则化项，以避免我们生成的图像对我们的网络结构过拟合：

$$I^* = \arg\max_I f(I) + R(I)$$

其中 $f(I)$ 表示神经元的值或者某一类的得分，而 $R(I)$ 是正则化项。

7.8 Fooling Images / Adversarial Examples

首先任选一个输入图像、任选一类，然后更改图像来使得选定类别的分值最大，反复此操作直到网络被 fool. 结果我们发现，一个被更改为错误分类的图像其实并没有肉眼可见的变化。



7.9 DeepDream: Amplify existing features

对于给定一张图像和选定 CNN 中的某层，反复执行：

1. 前向传播到选定的层；
2. 设置该层的梯度等于其激活值；

3. 反向传播，计算对输入图像的梯度；
4. 更新图像

这样做以使得图像的特征被增强。

8 Generative Models

8.1 Generative Models

Generative models 隶属于 unsupervised learning 的范畴，其数据集没有标签。其目的是根据输入学会一种数据集的分布，并生成具有这种分布的新的图像。

8.2 Pixel RNN & Pixel CNN

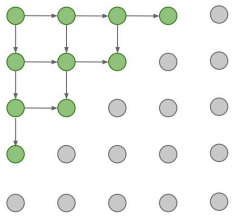
Pixel RNN 和 Pixel CNN 都属于 **Fully visible belief network**，其思想是对于图像 x，计算其似然 $p_\theta(x)$：

$$p_\theta(x) = \prod_{i=1}^{n} p_\theta(x_i \mid x_1, \ldots, x_{i-1})$$

其中 x_i 是 x 的一个像素。$p_\theta(x_i \mid x_1, \ldots, x_{i-1})$ 含义是，在已经生成像素点 $x_1 \ldots x_{i-1}$ 的条件下，下一个像素点为 x_i 的概率。那么根据极大似然法的思想，我们想生成一个好的图像 x，目标就是最大化似然函数 $p_\theta(x)$.

这跟网络有什么关系呢？上式的计算未免过于复杂，而我们知道神经网络善于对一个复杂的计算过程建模，因此我们可以设计一些网络达到目的。

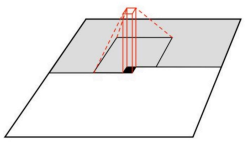
Pixel RNN 我们从左上角开始，按照下图所示顺序生成新的像素：



自然而然地，这个时序过程可以用 RNN/LSTM 来完成。

其缺点是每个像素是顺次生成的，这导致网络的生成速度较慢，其训练速度也慢。

Pixel CNN 把 RNN 换成 CNN，根据周围像素生成新的像素。



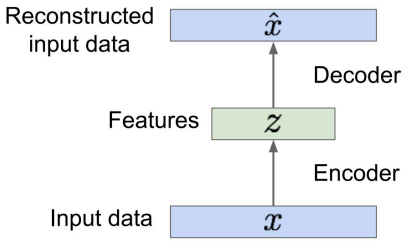
训练比 Pixel RNN 快，但生成依旧是顺次生成，速度依旧较慢。

8.3 Variational Autoencoders (VAE)

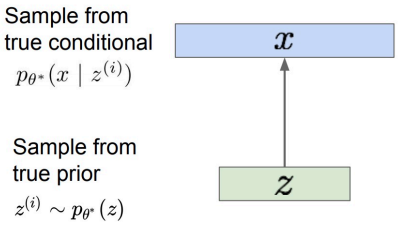
Autoencoders 在学习 VAE 之前，我们首先需要了解 Autoencoders.

Autoencoders 是一种 unsupervised 的 dimensionality reduction 的方法。其思想是，用一个 CNN 将输入数据 x 降维为 z，然后再用一个 CNN 将降维后的数据 z 恢复为原大小 $\hat{x}$，并定义 loss function 为：$||x - \hat{x}||^2$，这样在训练后，前一个 CNN 就可以作为数据降维的 encoder 了。注意这个过程并没有使用标签，所以这是 unsupervised 的。

Autoencoders 可以用于 supervised model 的初始化，帮助模型的学习。



VAE 我们假设真实数据 $\{x^{(i)}\}_{i=1}^N$ 是由一个未知的、隐藏的 z 采样得到的，也就是说，给定 $z^{(i)}$，$x^{(i)}$ 采样自概率分布 $p_\theta(x \mid z^{(i)})$，而这里的 $z^{(i)}$ 又采样自一个先验概率分布：$p_\theta(z)$.



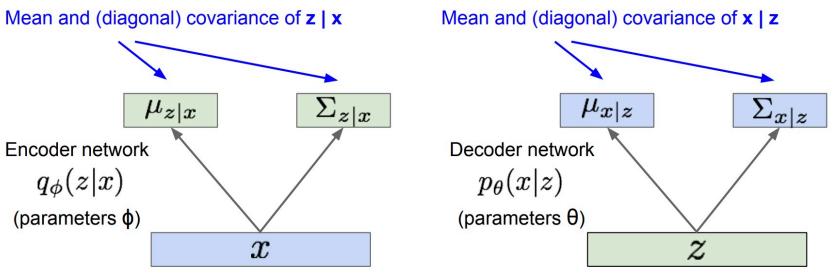
我们可以合理地选取高斯分布为 z 的先验分布；又由于 $p_\theta(x \mid z^{(i)})$ 是一个复杂的东西，所以我们可以用一个 decoder network 对它进行建模。那如何训练这个神经网络呢？不同于 Fully visible belief network，VAE 将似然写作：

$$p_\theta(x) = \int p_\theta(z) p_\theta(x \mid z) \mathrm{d}z$$

根据极大似然法的思想，最大化这个似然函数就是训练神经网络的过程了。

这时问题出现了，因为积分的存在，我们无法处理 $p_\theta(x)$ 这个函数，也就无法训练神经网络；我们也没法处理后验分布：$p_\theta(z \mid x) = p_\theta(x \mid z) p_\theta(z) / p_\theta(x)$。

对于第二个问题，我们再定义一个神经网络 $q_\phi(z \mid x)$ 去近似 $p_\theta(z \mid x)$：

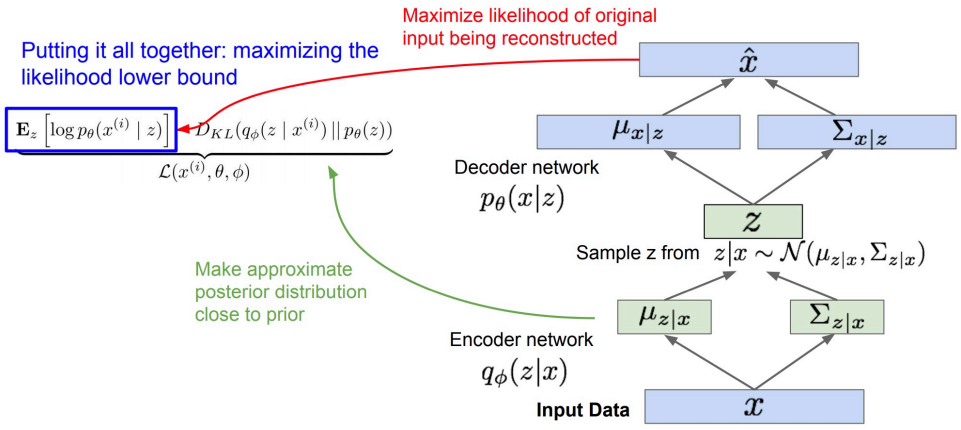


两个神经网络的输出都是均值和方差，如此，在 inference 阶段，我们可以选取以该均值和方差为统计量的正态分布作为采样的概率分布。

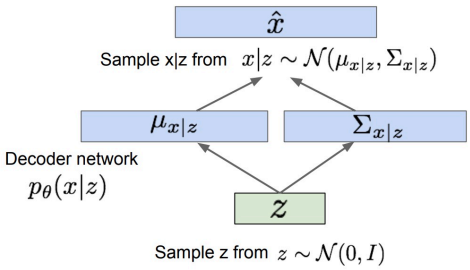
对第一个问题的解决方法是，我们训练一个 $p_\theta(x)$ 的下界：

$$
\begin{aligned}
\log p_\theta(x^{(i)}) &= \mathbb{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta(x^{(i)})\right] \\
&= \mathbb{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\right] \\
&= \mathbb{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\frac{q_\phi(z \mid x^{(i)})}{q_\phi(z \mid x^{(i)})}\right] \\
&= \mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - \mathbb{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z)}\right] + \mathbb{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z \mid x^{(i)})}\right] \\
&= \mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)})||p_\theta(z)) + D_{KL}(q_\phi(z \mid x^{(i)})||p_\theta(z \mid x^{(i)})) \\
&\geqslant \mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)})||p_\theta(z))
\end{aligned}
$$

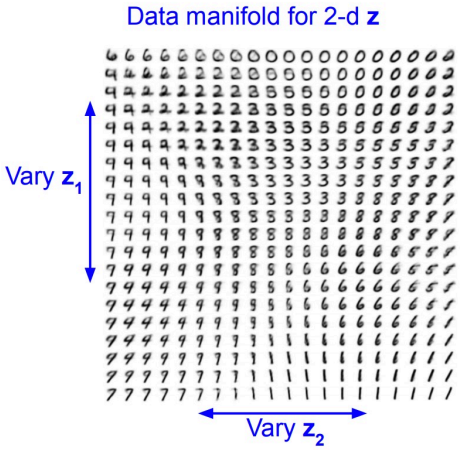
于是乎，我们训练神经网络的过程，就是最大化这个下界 $\mathcal{L}(x^{(i)},\theta,\phi) = \mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)})||p_\theta(z))$ 的过程。如何理解这个过程呢？最大化 $\mathcal{L}(x^{(i)},\theta,\phi)$，就要最大化第一项——即努力重新构造出输入数据，以及最小化第二项——即努力使得近似后验分布接近于我们预定的先验分布。综上，我们的训练过程如下：



现在我们训练好了一个 VAE 网络，就可以用它来生成数据了。从 $z \sim N(0, I)$ 对 z 进行采样，然后用训练的 decoder network 得到 $\mu_{x|z}$ 和 $\Sigma_{x|z}$，随后从 $x \mid z \sim N(\mu_{x|z}, \Sigma_{x|z})$ 得到生成的数据 $\hat{x}$：



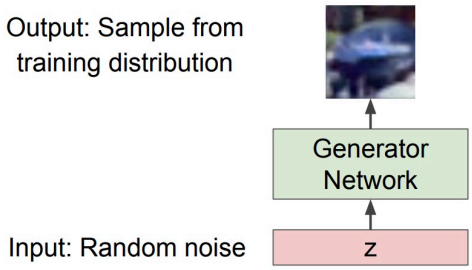
以下是生成 MNIST 数字的 VAE，选取 z 为 2 维时可以得到：



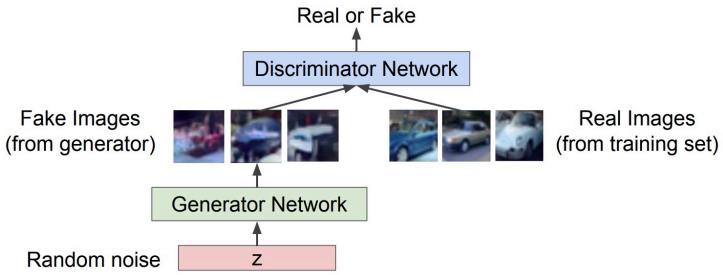
8.4 GANs

PixelCNNs 和 VAEs 都显式地对概率密度函数 $p_\theta(x)$ 进行了定义，我们是否可以不给出一个显式的概率密度函数呢？GANs 网络就是这样的。

我们没有直接的方法从训练集里找出一个概率分布并据此采样以生成新的图像，但我们能从一个简单分布采样，例如随机噪声；随后我们不断改变这个简单的分布，以最终逼近真正的分布。这个复杂的过程显然用神经网络建模是最好不过的了：



那么我们如何训练这个神经网络呢？方法是用两个神经网络进行博弈——Generator 负责生成新的图像，Discriminator 负责辨别输入图像是真的还是假的（输出真的概率）。训练时 Generator 的目标是尽可能地骗过 Discriminator，而 Discriminator 的目标就是不被 Generator 骗到。



我们的目标函数定义为 Minimax objective function：

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x\sim p_{data}} \ln D_{\theta_d}(x) + \mathbb{E}_{z\sim p(z)} \ln(1 - D_{\theta_d}(G_{\theta_g}(z)))\right]$$

先看内层 max 的部分，第一项中，x 取自真实分布，$D_{\theta_d}(x)$ 是 Discriminator 认为真的概率，所以这一项是要最大化真实图像是真的的概率；第二项中，z 取自生成网络，$1 - D_{\theta_d}(G_{\theta_g}(z))$ 是

Discriminator 认为假的概率，所以这一项是要最大化假图像是假的的概率。因此，内层值越高，代表 Discriminator 越准确，这正好不是 Generator 希望看到的，所以外层套一个 min，表示 Generator 希望 Discriminator 的最大得分尽可能低。这正是所谓的 minimax 算法。

训练时这个目标函数可以拆成两部分，对 Discirminator，用梯度上升使得：

$$\max_{\theta_d} \left[\mathbb{E}_{x\sim p_{data}} \ln D_{\theta_d}(x) + \mathbb{E}_{z\sim p(z)} \ln(1 - D_{\theta_d}(G_{\theta_g}(z)))\right]$$

对 Generator，用梯度下降使得：

$$\min_{\theta_g} \mathbb{E}_{z\sim p(z)} \ln(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

因为前一项与 θ_g 无关，所以只有这后一项。

然而在实践中，优化这个目标函数并不能工作得很好。这是因为 $y = \ln(1 - x)$ 的特性是在 x 接近 0 时梯度较小，x 接近 1 时梯度很大，于是在 Generator 这里，生成的图像很假的时候学习较慢，生成的图像已经很逼真的时候学习反而很快，不符合我们的预期。因此，我们改用梯度上升训练 Generator，使得：

$$\max_{\theta_g} \mathbb{E}_{z\sim p(z)} \ln(D_{\theta_d}(G_{\theta_g}(z)))$$

这样梯度就符合我们的预期了。

总结一下，训练 GANs 的流程为：

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)})))\right]$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by ascending its stochastic gradient (improved objective):

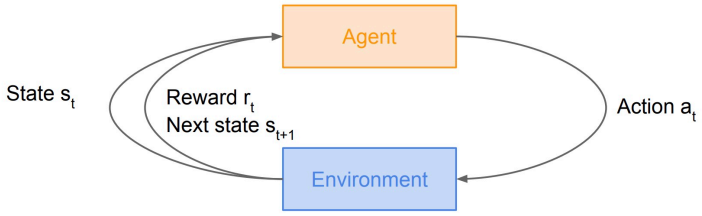
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

9 Deep Reinforcement Learning

9.1 Reinforcement Learning

强化学习要解决的问题类似于玩一个游戏：Environment 告诉 Agent 当前状态 s_t，Agent 据此做出一个动作 a_t，Environment 为该动作给出 r_t 的奖励，并更新状态为 s_{t+1}，然后反复上述过程。



9.2 Markov Decision Process

强化学习可以用马尔可夫过程进行数学形式化的表述，因为它满足 **Markov property**，即当前状态完全定义了未来的状态，或者说未来的状态仅受当前状态、而不受过去状态的影响。

马尔可夫过程定义为：$(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$，其中 $\mathcal{S}$ 表示所有可能的状态集合，$\mathcal{A}$ 表示所有可能的动作集合，$\mathcal{R}$ 表示奖励在给定 $\mathcal{S}$ 和 $\mathcal{A}$ 下的分布，$\mathbb{P}$ 表示状态的转移概率，γ 是 discount factor，用于对近期奖励和远期奖励进行加权。

马尔可夫过程的流程是：

- 在 $t = 0$ 时刻，environment 对初始状态进行采样 $s_0 \sim p(s_0)$；
- 接下来，反复执行以下步骤直到结束：
  - Agent 选择动作 a_t；
  - Environment 对奖励进行采样 $r_t \sim \mathcal{R}(\bullet \mid s_t, a_t)$；
  - Environment 对下一个状态进行采样 $s_{t+1} \sim \mathbb{P}(\bullet \mid s_t, a_t)$；
  - Agent 收到奖励 r_t 和下一个状态 s_{t+1}。

一个决策 π 就是一个从 $\mathcal{S}$ 到 $\mathcal{A}$ 的函数，决定在每一个状态下执行什么动作，而我们的目标就是找到最佳的决策 π^* 以使得累计奖励 $\sum_{t \geqslant 0} \gamma^t r_t$ 最大。但是，由于我们的众多变量都是随机变量，所以我们实际上要最大化的是累计奖励的期望，即：

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t \geqslant 0} \gamma^t r_t \mid \pi\right]$$

其中 $s_0 \sim p(s_0)$, $a_t \sim \pi(\bullet \mid s_t)$, $s_{t+1} \sim \mathbb{P}(\bullet \mid s_t, a_t)$.

服从一个决策将会产生一个样本路径（轨迹）：$s_0, a_0, r_0, s_1, a_1, r_1, \ldots$，我们需要一种方式量化从这个初始状态 s_0 开始的好坏程度，因此定义 **value function**：

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t \geqslant 0} \gamma^t r_t \mid s = s_0, \pi\right]$$

进一步的，我们还可以定义从初始状态 s_0 和初始动作 a_0 开始服从某一个决策 π 的好坏程度，称

为 **Q-value function**：

$$Q^{\pi}(s,a)=\mathbb{E}\left[\sum_{t\geqslant 0}\gamma^{t}r_{t}\mid s_{0}=s,a_{0}=a,\pi\right]$$

那么，最佳的 Q-value function Q^{*} 就是在给定 (s,a) 下找到使得 $Q^{\pi}(s,a)$ 最大的决策方式 π：

$$Q^{*}(s,a)=\max_{\pi}\mathbb{E}\left[\sum_{t\geqslant 0}\gamma^{t}r_{t}\mid s_{0}=s,a_{0}=a,\pi\right]$$

而一个核心的等式——**Bellman Equation** 告诉我们 Q^{*} 满足：

$$Q^{*}(s,a)=\mathbb{E}_{s'\sim\mathcal{E}}\left[r+\gamma\max_{a'}Q^{*}(s',a')\mid s,a\right]$$

注释. 推导：

$$\begin{aligned}Q^{\pi}(s,a)&=\mathbb{E}\left[r_{t}+\gamma r_{t+1}+\gamma^{2}r_{t+2}+\cdots\mid s_{0}=s,a_{0}=a,\pi\right]\\&=\mathbb{E}[r_{t}+\gamma(r_{t+1}+\gamma r_{t+2}+\cdots)\mid s_{0}=s,a_{0}=a,\pi]\\&=\mathbb{E}[r_{t}+\gamma Q^{\pi}(s',a')\mid s_{0}=s,a_{0}=a,\pi]\end{aligned}$$

其中，s',a' 是下一时刻的状态和动作。因此，$Q^{\pi}(s,a)$ 最大值就是当前奖励 r_t 加上在接下来的状态中用最佳的动作得到的最大收益。

为了求解上述最佳 Q-value functino Q^{*}，根据 Bellman Equation，我们可以迭代求解，迭代格式为：

$$Q_{i+1}(s,a)=\mathbb{E}\left[r+\gamma\max_{a'}Q_{i}(s',a')\mid s,a\right]$$

可以证明，该迭代格式收敛到 Q^{*}.

这么做理论上可行，但是实际中我们需要遍历所有可能的 (s,a) 对，这是不可行的。为了解决这个问题，我们可以找一个函数去近似 $Q(s,a)$，比如用一个神经网络——这就引出了 Q-Learning。

9.3 Q-Learning

正如前文所述，Q-Learning 的目的是用一个神经网络近似 $Q^{*}(s,a)$，即：

$$Q(s,a;\theta)\approx Q^{*}(s,a)$$

其中 θ 是我们的网络的参数。

要训练这个网络，关键是定义损失函数。由于 Q^{*} 需要满足 Bellman Equation，我们可以用当前网络到 Bellman Equation 的 MSE 误差作为损失函数：

$$\begin{aligned}L_{i}(\theta_{i})&=\mathbb{E}_{s,a\sim\rho(\bullet)}\left[(y_{i}-Q(s,a;\theta_{i}))^{2}\right]\\\text{where }\ y_{i}&=\mathbb{E}_{s\sim\mathcal{E}}\left[r+\gamma\max_{a'}Q(s',a';\theta_{i-1})\mid s,a\right]\end{aligned}$$

在训练玩游戏时，一个自然的想法是用最近的若干连续的画面帧作为神经网络的输入，但这么做将

会导致一些问题：首先这些样本将高度相关，以至于学习的效率低；其次，当前网络的参数将决定接下来输入样本的分布，例如当前的动作是向左走，那么接下来的训练样本将大量采样自画面左侧。为了解决这些问题，我们使用 **experience replay**：维护一个 replay memory table，其存储的是 (s_t, a_t, r_t, s_{t+1}) 组，在训练过程中不断更新其内容；在训练时，一个 minibatch 是从这个 table 中随机选取的，而非采取若干连续的状态，这样就解决了上述问题。

9.4 Policy Gradients

Q-Learning 的问题在于 Q-function 可以非常复杂，而我们需要的策略可能仅仅是一个非常简单的动作而已。这就引出了 Policy Gradients.

首先，我们定义决策空间为：$\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$. 对每个决策，我们定义其值为：

$$J(\theta) = \mathbb{E}\left[\sum_{t\geqslant 0} \gamma^t r_t \mid \pi_\theta\right]$$

我们的目标是最大化 $J(\theta)$，即找到 $\theta^* = \arg\max_\theta J(\theta)$. 容易想到我们对 θ 做梯度上升即可。

$J(\theta)$ 可写作：

$$J(\theta) = \mathbb{E}_{\tau\sim p(\tau;\theta)}[r(\tau)] = \int_\tau r(\tau)p(\tau;\theta)\mathrm{d}\tau$$

其中 $\tau = (s_0, a_0, r_0, s_1, \ldots)$ 是一个决策轨迹，$r(\tau)$ 为其奖励。现在我们求解其梯度：

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_\tau r(\tau)\nabla_\theta p(\tau;\theta)\mathrm{d}\tau \\ &= \int_\tau r(\tau)p(\tau;\theta)\nabla_\theta \ln p(\tau;\theta)\mathrm{d}\tau \\ &= \mathbb{E}_{\tau\sim p(\tau;\theta)}[r(\tau)\nabla_\theta \ln p(\tau;\theta)]\end{aligned}$$

由于 $p(\tau;\theta) = \prod_{t\geqslant 0} p(s_{t+1} \mid s_t, a_t)\pi_\theta(a_t \mid s_t)$，故：

$$\nabla_\theta \ln p(\tau;\theta) = \nabla_\theta \sum_{t\geqslant 0} \ln p(s_{t+1} \mid s_t, a_t) + \ln \pi_\theta(a_t \mid s_t) = \nabla_\theta \sum_{t\geqslant 0} \ln \pi_\theta(a_t \mid s_t)$$

于是乎，我们可以对轨迹 τ 进行采样，然后计算：

$$\nabla_\theta J(\theta) \approx \sum_{t\geqslant 0} r(\tau)\nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

但是这个 gradient estimator 很难达到我们想要的效果，因为其方差很大。为此，人们提出了若干减小方差的方法：

- Idea 1：

  $$\nabla_\theta J(\theta) \approx \sum_{t\geqslant 0}\left(\sum_{t'\geqslant t} r_{t'}\right)\nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

  用从当前决策开始直到结束的过程中得到的奖励代替 $r(\tau)$。

- Idea 2：

$$\nabla_\theta J(\theta) \approx \sum_{t\geqslant 0}\left(\sum_{t'\geqslant t}\gamma^{t'-t}r_{t'}\right)\nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

  加入一个 discount factor γ.
- Idea：引入 baseline function：

$$\nabla_\theta J(\theta) \approx \sum_{t\geqslant 0}\left(\sum_{t'\geqslant t}\gamma^{t'-t}r_{t'} - b(s_t)\right)\nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

如何选取 baseline 呢？一个最简单的方法是取已经遍历过的决策轨迹的奖励的移动平均。不过利用 Q-value function，我们能得到更好的 baseline：

$$\nabla_\theta J(\theta) \approx \sum_{t\geqslant 0}\left(Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t)\right)\nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

直观上讲，我们希望 $A^{\pi_\theta}(s_t,a_t) = Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t)$ 越大越好，因为这表示我们选取的动作 a_t 使得这之后我们得到的奖励期望值大于随便取一个动作得到的奖励期望值，我们称 $A^\pi(s,a)$ 为 **advantage function.**

现在我们得到了 Actor-Critic Algorithm，即结合 Policy Gradients 和 Q-learning 的算法：同时训练一个 actor，即决策，和 critic，即 Q-function——Actor 决定做一个动作，critic 告诉它这个动作的优劣以及应该如何调整；同时 critic 无需对所有 (s,a) 对进行学习，而只需要学习 actor 做出的那些决策。