第六章　神经网络与深度学习

周世斌

中国矿业大学　　计算机学院

May. 2022

参数范数惩罚

正则化在深度学习的出现前就已经应用了数十年。线性模型如线性回归和逻辑斯蒂回归以使用简单、直接有效的正则化策略。

许多正则化方法通过对目标函数 J 添加一个参数范数惩罚 $\Omega(\theta)$，限制模型（如神经网络、线性回归或逻辑斯蒂）的学习能力。我们将正则化后的目标函数记为 $\tilde{J}$：

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta),$$

其中 $\alpha \in [0, \infty)$ 是权衡范数惩罚项 Ω 和标准目标函数 $J(X; \theta)$ 相对贡献的超参数。将 α 设为 0 就是没有正则化。越大的 α，对应于越多的正则化惩罚。

当我们的训练算法最小化正则化后的目标函数 $\tilde{J}$ 时，它会降低原始目标 J 关于训练数据的误差并同时减小参数 θ 的规模（或在某些衡量下参数子集的规模）。参数范数 Ω 的不同选择会导致不同的优先解。

L^2 参数正则化

已经看到最简单和最常见的参数范数惩罚是，通常被称为权重衰减的 L^2 参数范数惩罚。这个正则化策略通过向目标函数添加一个正则项 $\Omega(\theta) = \frac{1}{2}\|w\|_2^2$，使权重更加接近原点[1]。

在其他学术圈，L^2 也被称为岭回归或 tikhonov 正则化。

[1]更一般地，我们可以将参数正则化为接近空间中的任意特定点，令人惊讶的是仍有正则化效果，并且更接近真实值将获得更好的结果。当我们不知道正确的值应该是正还是负，零是有意义的默认值。由于将模型参数正则化为零的情况更常见，我们将关注这种特殊情况。

我们可以研究正则化后目标函数的梯度，洞察一些权值衰减的正则化表现。为了简单起见，我们假定其中没有偏差参数，因此 θ 就是 w。这样一个模型具有以下总的目标函数：

$$\tilde{J}(w; X, y) = \frac{\alpha}{2} w^\top w + J(w; X, y), \tag{1}$$

与之对应的梯度为

$$\nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_w J(w; X, y). \tag{2}$$

使用单步梯度下降更新权重，即执行以下更新：

$$w \leftarrow w - \epsilon(\alpha w + \nabla_w J(w; X, y)). \tag{3}$$

换种写法就是：

$$w \leftarrow (1 - \epsilon\alpha)w - \epsilon\nabla_w J(w; X, y). \tag{4}$$

我们可以看到，加入权重衰减后会修改学习规则，即在每步执行通常的梯度更新之前先收缩权重向量（将权重向量乘以一个常数因子）。这是单个步骤发生的变化。但是，在训练的整个过程会发生什么？

L^1 参数正则化

L^2 权值衰减是权值衰减最常见的形式，我们还可以使用其他的方法惩罚模型参数的大小。另一种选择是使用 L^1 正则化。

对模型参数 w 的 L^1 正则化形式定义为：

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|, \tag{5}$$

与 L^2 权值衰减类似，L^1 权值衰减的强度也可以通过缩放惩罚项 Ω 的正超参数 α 控制。因此，正则化的目标函数 $\tilde{J}(w; X, y)$ 如下给出

$$\tilde{J}(w; X, y) = \alpha \|w\|_1 + J(w; X, y), \tag{6}$$

对应的梯度 (实际上是次梯度)：

$$\nabla_w \tilde{J}(w; X, y) = \alpha \mathsf{sign}(w) + \nabla_w J(w; X, y), \tag{7}$$

其中 $\mathsf{sign}(w)$ 只是简单地取 w 各个元素的符号。

相比 L^2 正则化，L^1 正则化会产生更稀疏的解。此处稀疏性指的是一些参数具有 0 的最优值。由 L^1 正则化导出的稀疏性质已经被广泛地用于特征选择机制。特征选择从可用的特征子集选择应该使用的子集，简化了机器学习问题。特别是著名的 LASSO（Tibshirani，95）（Least Absolute Shrinkage and Selection Operator）模型将 L^1 惩罚和线性模型结合，并使用最小二乘代价函数。L^1 惩罚使部分子集的权重为零，表明相应的特征可以被安全地忽略。

深度模型中常用正则化技巧

1. 数据集增强
2. 输入层注入噪声
3. 输出层注入噪声
4. 提前终止
5. Dropout

数据集增强

让机器学习模型泛化得更好的最好办法是使用更多的数据进行训练。当然，在实践中，我们拥有数据量是有限的。解决这个问题的一种方法是创建假数据并把它添加到训练集。对于一些机器学习任务，创建新的假数据相当简单。

对分类来说这种方法是最简单的。分类器需要一个复杂的高维输入 x，并用单个类别标识 y 概括 x。这意味着分类面临的一个主要任务是要对各种各样的变换保持不变。我们可以轻易通过转换训练集中的 x 来生成新的 (x, y) 对。

这种方法对于其他许多任务来说并不那么容易。例如，除非我们已经解决了密度估计问题，否则在密度估计任务中生成新的假数据是困难的。

数据集增强对一个具体的分类问题来说是特别有效的方法：对象识别。图像是高维的并包括各种巨大的变化因素，其中有许多可以轻易地模拟。即使模型已使用卷积和池化技术对部分平移保持不变，沿训练图像每个方向平移几个像素的操作通常可以大大改善泛化。许多其他操作如旋转图像或缩放图像也已被证明非常有效。

我们必须要小心，不能应用改变正确类别的转换。例如，光学字符识别任务需要认识到 “b” 和 “d” 以及 “6” 和 “9” 的区别，所以对这些任务来说，水平翻转和旋转 180° 并不是适当的数据集增强方式。

输入层注入噪声

在神经网络的输入层注入噪声也可以被看作是数据增强的一种形式。对于许多分类甚至一些回归任务，即使小的随机噪声被加到输入，任务仍应该是能解决的。然而，神经网络被证明对噪声不是非常健壮。

改善神经网络健壮性的方法之一是简单地将随机噪声施加到输入再进行训练。输入噪声注入是一些无监督学习算法的一部分，如去噪自编码器。向隐藏单元施加噪声也是可行的，这可以被看作在多个抽象层上进行的数据集增强。Poole 最近表明，噪声的幅度被细心调整后，该方法是非常高效的。dropout 方法是一个的强大正则化策略，可以被看作通过与噪声相乘构建新输入的过程。

输出层注入噪声

大多数数据集的 y 标签都有一定错误。当 y 是错误的，对最大化 $\log p(y \mid x)$ 会是有害的。为了防止这一点的一种方法是显式地对标签上的噪声进行建模。例如，我们可以假设，对于一些小常数 ϵ，训练集标记 y 是正确的概率是 $1-\epsilon$，任何其他可能的标签可能是正确的。

这个假设很容易就能解析地与代价函数结合，而不用显式地采噪声样本。例如，标签平滑 (label smoothing) 基于 k 个输出的 softmax 函数，把明确分类 0 和 1 替换成 $\frac{\epsilon}{k-1}$ 和 $1-\epsilon$，对模型进行正则化。标准交叉熵损失可以用在这些非确切目标的输出上。使用 softmax 函数和明确目标的最大似然学习可能永远不会收敛——softmax 函数永远无法真正预测 0 概率或 1 概率，因此它会继续学习越来越大的权重，使预测更极端。使用如权值衰减等其他正则化策略能够防止这种情况。标签平滑的优势是能防止模型追求明确概率而不妨碍正确分类。这种策略自 20 世纪 80 年代就已经被使用，并在现代神经网络继续保持显著特色。

提前终止

当训练有足够的表示能力甚至会过度拟合的大模型时，我们经常观察到，训练误差会随着时间的推移逐渐降低但验证集的误差会再次上升。

这意味着如果我们返回使验证集误差最低的参数设置，就可以获得更好的模型（因此，有希望获得更好的测试误差）。在每次验证集误差有所改善后，我们存储模型参数的副本。当训练算法终止时，我们返回这些参数而不是最新的参数。当验证集上的误差在事先指定的循环内没有进一步改善时，算法就会终止。

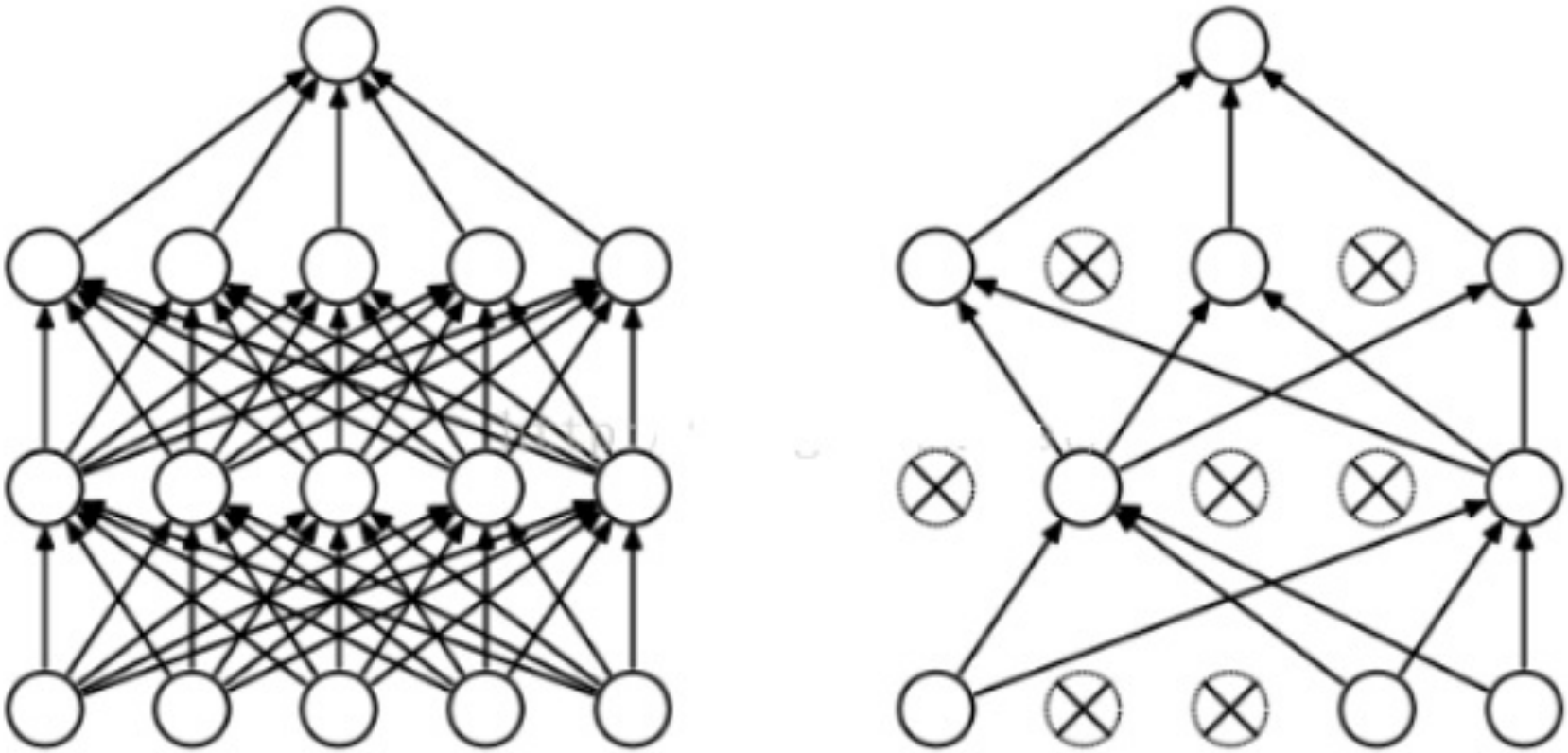
这种策略被称为提前终止。这可能是深度学习中最常用的正则化形式。它的流行主要是因为有效性和简单性。

我们可以认为提前终止是非常高效的超参数选择算法。按照这种观点，训练步数仅是另一个超参数。

在提前终止的情况下，我们通过拟合训练集的步数来控制模型的有效容量。大多数超参数的选择必须使用昂贵的猜测和检查过程，我们需要在训练开始时猜测一个超参数，然后运行几个步骤检查它的训练效果。“训练时间”是唯一只要跑一次训练就能尝试很多值的超参数。通过提前终止自动选择超参数的唯一显著的代价是训练期间要定期评估验证集。理想情况下，这可以并行在与主训练过程分离的机器上，或独立的 CPU，或独立的 GPU 上完成。如果没有这些额外的资源，可以使用比训练集小的验证集或较不频繁地评估验证集来减小评估代价，较粗略地估算取得最佳的训练时间。

Dropout

Dropout 提供了正则化一大类模型的方法，计算方便但功能强大。Dropout 说的简单一点就是我们让在前向传导的时候，让某个神经元的激活值以一定的概率 p，让其停止工作，示意图如下：



(a) Standard Neural Net

(b) After applying dropout.

以前我们网络的计算公式是：

$$
\begin{aligned}
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}),
\end{aligned}
$$

采用 dropout 后计算公式就变成了：

$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
\tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}).
\end{aligned}
$$

深度学习算法在许多情况下都涉及到优化。例如，模型中的推断（如PCA）涉及到求解优化问题。我们经常使用解析优化去证明或设计算法。在深度学习涉及到诸多优化问题中，最难的是神经网络训练。甚至是用几百台机器投入几天到几个月去解决一个神经网络训练问题，也是很常见的。因为这其中的优化问题很重要，代价也很高，因此开发了一组专门的优化技术。本章会介绍神经网络训练中的这些优化技术。

优化通常是一个极其困难的任务。通常，机器学习会小心设计目标函数和约束，以确保优化问题是凸的，从而避免一般优化问题的复杂度。在训练神经网络时，我们肯定会遇到一般的非凸情况。即使是凸优化，也并非没有任何问题。

神经网络优化中的挑战

1. 病态问题
2. 悬崖与梯度爆炸
3. 长期依赖
4. ...

病态

在优化凸函数时，会遇到一些挑战。这其中最突出的是 Hesse 矩阵 H 的病态（ill-conditioning）。这是数值优化，凸优化或其他形式的优化中普遍存在的问题，

病态问题一般被认为存在于神经网络训练过程中。病态体现在随机梯度下降会“卡”在某些情况，此时即使很小的更新步长也会增加代价函数。

悬崖与梯度爆炸

多层神经网络通常有像悬崖一样的斜率较大区域. 这是由于几个较大的权重相乘导致的。遇到斜率极大的悬崖结构时，梯度更新会很大程度地改变参数值，通常会跳过这类悬崖结构。

这些非线性在某些区域会产生非常大的导数。当参数接近这样的悬崖区域时，梯度下降更新可以使参数弹射得非常远，可能会无效化已经完成的大量优化工作。

长期依赖

当计算图变得非常之深时，神经网络优化算法会面临的另外一个难题就是长期依赖问题，由于变深的结构使模型丧失了学习到先前信息的能力，让优化变得极其困难。

例如，假设某个计算图中包含一条重复与矩阵 W 相乘的路径。那么 t 步后，相当于和 W^t 相乘。假设 W 有特征值分解 $W = V\mathrm{diag}(\lambda)\, V^{-1}$。在这种简单的情况下，很容易看到

$$W^t = (V\mathrm{diag}(\lambda)\,V^{-1})^t = V\mathrm{diag}(\lambda)^t V^{-1}.$$

当特征值 λ_i 不在 1 附近时，若在量级上大于 1 则会爆炸；若小于 1 时则会消失。梯度消失（或弥散）问题（**vanishing gradient problem**）和梯度爆炸问题（**exploding gradient problem**）是指该计算图上的梯度也会因为 $\mathrm{diag}(\lambda)^t$ 大幅度变化。消失的梯度使得难以知道参数朝哪个方向移动能够改进代价函数，而爆炸梯度会使得学习不稳定。之前描述的诱发梯度截断的悬崖结构便是爆炸梯度现象的一个例子。

此处描述的每个时间点重复与 W 相乘非常类似于寻求矩阵 W 的最大特征值及对应的特征向量的幂函数。从这个观点来看，$x^\top W^t$ 最终会丢弃 x 中所有与 W 的主特征向量垂直的成分。

1. 随机梯度下降
2. 动量
3. AdaGrad
4. RMSProp
5. Adam 算法
6. ...

随机梯度下降及其变种很可能是一般机器学习中用得最多的优化算法，特别是在深度学习中。通过计算独立同分布地从数据生成分布中抽取的 m 个小批量样本的梯度均值，我们可以得到梯度的无偏估计。

下面算法展示了如何沿着这个梯度估计下降。

Algorithm 4.1 随机梯度下降 (SGD) 在第 k 个训练迭代的更新

Require: 学习率 ϵ_k

Require: 初始参数 θ

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{x^{(1)}, \ldots, x^{(m)}\}$ 的小批量，对应目标为 $y^{(i)}$。

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

应用更新：$\theta \leftarrow \theta - \epsilon\hat{g}$

end while

随机梯度下降算法中的一个关键参数是学习速率。之前，我们介绍的随机梯度下降使用固定的学习速率。在实践中，有必要随着时间的推移逐渐降低学习速率，因此我们将第 k 步迭代的学习速率记作 ϵ_k。

这是因为随机梯度下降梯度估计引入的噪源（m 个训练样本的随机采样）并不会在极小值处消失。相比之下，当我们使用批量梯度下降到达极小值时，整个代价函数的真实梯度会变得很小，甚至为 $\mathbf{0}$，因此批量梯度下降可以使用固定的学习速率。保证随机梯度下降收敛的一个充分条件是

$$\sum_{k=1}^{\infty} \epsilon_k = \infty,$$

且

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$

实践中，一般会线性衰减学习速率到第 τ 次迭代：

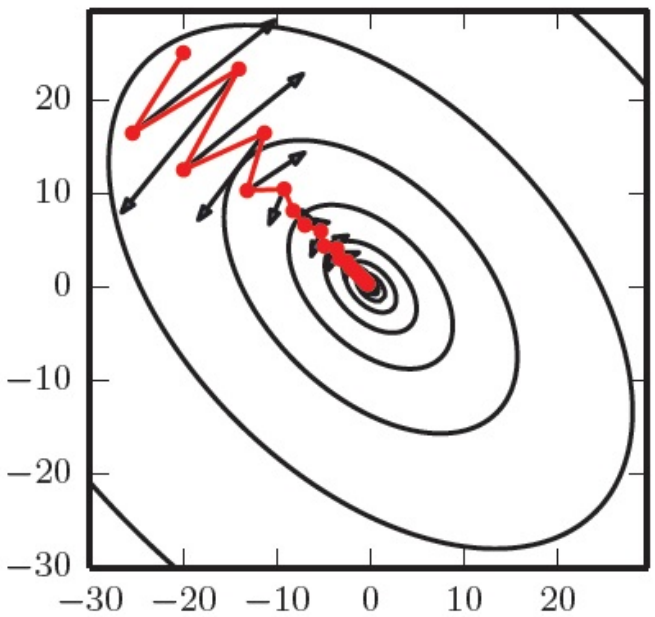
$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau$$

其中 $\alpha = \frac{k}{\tau}$。在 τ 步迭代之后，一般使 ϵ 保持常数。

动量

虽然随机梯度下降仍然是非常受欢迎的优化方法，但学习速率有时会很慢。动量方法 (Polyak,1964) 旨在加速学习，特别是处理高曲率，小但一致的梯度，或是带噪声的梯度。动量算法积累了之前梯度指数级衰减的移动平均，并且继续沿该方向移动。

动量的效果如图所示：



动量的主要目的是解决两个问题：Hessian 矩阵的不良条件数和随机梯度的方差。我们通过此图说明动量如何克服这两个问题的第一个。轮廓线描绘了一个二次损失函数（具有不良条件数的 Hessian 矩阵）。横跨轮廓的红色路径表示动量学习规则所遵循的路径，它使该函数最小化。我们在每个步骤画一个箭头，指示梯度下降将在该点采取的步骤。我们可以看到，一个条件数较差的二次目标函数看起来像一个长而窄的山谷或陡峭的峡谷。动量正确地纵向穿过峡谷，而梯度步骤则会浪费时间在峡谷的窄轴上来回移动。

动量

从形式上看，动量算法引入了变量 v 充当速度的角色——它代表参数在参数空间移动的方向和速度。速度被设为负梯度的指数衰减平均。名称动量来自物理类比，根据牛顿运动定律，负梯度是移动参数空间中粒子的力。动量在物理学上是质量乘以速度。在动量学习算法中，我们假设是单位质量，因此速度向量 v 也可以看作是粒子的动量。超参数 $\alpha \in [0, 1)$ 决定了之前梯度的贡献衰减得有多快。更新规则如下：

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta), y^{(i)}) \right),$$

$$\theta \leftarrow \theta + v.$$

速度 v 累积了梯度元素 $\nabla_\theta(\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)}; \theta), y^{(i)}))$。相对于 ϵ 的 α 越大，之前梯度对现在方向的影响也越大。

带动量的随机梯度下降 (SGD) 算法如下所示。

Algorithm 4.2 使用动量的梯度下降 (SGD)

Require: 学习率 ϵ，动量参数 α

Require: 初始参数 θ，初始速度 v

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{x^{(1)}, \ldots, x^{(m)}\}$ 的小批量，对应目标为 $y^{(i)}$。

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

计算速度更新：$v \leftarrow \alpha v - \epsilon g$

应用更新：$\theta \leftarrow \theta + v$

end while

之前，步长只是梯度范数乘以学习率。现在，步长取决于梯度序列的大小和排列。当许多连续的梯度指向相同的方向时，步长最大。如果动量算法总是观测到梯度 g，那么它会在方向 $-g$ 上不停加速，直到达到最后速度的步长为

$$\frac{\epsilon\|g\|}{1-\alpha}.$$

将动量的超参数视为 $\frac{1}{1-\alpha}$ 有助于理解。例如，$\alpha = 0.9$ 对应着最大速度 10 倍于梯度下降算法。

在实践中，α 的一般取值为 0.5，0.9 和 0.99。和学习速率一样，α 也会随着时间变化。一般初始值是一个较小的值，随后会慢慢变大。

受 Nesterov 加速梯度算法 (Nesterov,1983,2004) 启发，Sutskever(2013) 提出了动量算法的一个变种。这种情况的更新规则如下：

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left[\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta + \alpha v), y^{(i)}) \right],$$

$$\theta \leftarrow \theta + v,$$

其中参数 α 和 ϵ 发挥了和标准动量方法中类似的作用。Nesterov 动量和标准动量之间的区别体现在梯度计算上。Nesterov 动量中，梯度计算在施加当前速度之后。因此，Nesterov 动量可以解释为往标准动量方法中添加了一个校正因子。

完整的 Nesterov 动量算法如下所示。

Algorithm 4.3 使用 Nesterov 动量的随机梯度下降 (SGD)

Require: 学习率 ϵ，动量参数 α

Require: 初始参数 θ，初始速度 v

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{x^{(1)}, \ldots, x^{(m)}\}$ 的小批量，对应目标为 $y^{(i)}$。

应用临时更新：$\tilde{\theta} \leftarrow \theta + \alpha v$

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

计算速度更新：$v \leftarrow \alpha v - \epsilon g$

应用更新：$\theta \leftarrow \theta + v$

end while

在凸批量梯度的情况下，Nesterov 动量将额外误差收敛率从 $O(1/k)$（k 步后）改进到 $O(1/k^2)$，如 (Nesterov,83) 所示。不幸的是，在随机梯度的情况下，Nesterov 动量没有改进收敛率。

神经网络研究员早就意识到学习率肯定是难以设置的超参数之一，因为它对模型的性能有显著的影响。损失通常高度敏感于参数空间中的某些方向，而不敏感于其他。动量算法可以在一定程度缓解这些问题，但这样做的代价是引入了另一个超参数。在这种情况下，自然会问有没有其他方法。如果我们相信方向敏感度有些轴对齐，那么每个参数设置不同的学习率，在整个学习过程中自动适应这些学习率便是有道理的。

delta-bar-delta 算法 (jacobs,1988) 是一个早期的在训练时适应模型参数单独学习速率的启发式方法。该方法基于一个很简单的想法，如果损失对于某个给定模型参数的偏导保持相同的符号，那么学习速率应该增加。如果对于该参数的偏导变化了符号，那么学习率应减小。当然，这种方法只能应用于全批量优化中。

最近，一些增量（或者基于小批量）的算法被提出适应模型参数的学习率。这节将简要回顾一些这种算法。

AdaGrad

AdaGrad 算法，独立地适应所有模型参数的学习速率，放缩每个参数反比于其所有梯度历史平方值总和的平方根 (Duchi,2011)。具有损失最大偏导的参数相应地有一个快速下降的学习速率，而具有小偏导的参数在学习速率上有相对较小的下降。净效果是在参数空间中更为平缓的倾斜方向会取得更大的进步。
在凸优化背景中，AdaGrad 算法具有一些令人满意的理论性质。然而，经验上已经发现，对于训练深度神经网络模型而言，从训练开始时积累梯度平方会导致有效学习速率过早和过量的减小。AdaGrad 在某些深度学习模型上效果不错，但不是全部。

Algorithm 4.4 AdaGrad 算法

Require: 全局学习率 ϵ

Require: 初始参数 θ

Require: 小常数 δ，为了数值稳定大约设为 10^{-7}

初始化梯度累积变量 $r = 0$

while 没有达到停止规则 **do**

从训练集中采包含 m 个样本 $\{x^{(1)}, \ldots, x^{(m)}\}$ 的小批量，对应目标为 $y^{(i)}$。

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

累积平方梯度：$r \leftarrow r + g \odot g$

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

应用更新：$\theta \leftarrow \theta + \Delta\theta$

end while

RMSProp

RMSProp 算法 (Hinton,2012) 修改 AdaGrad 以在非凸设定下效果更好，改变梯度积累为指数加权的移动均值。AdaGrad 旨在应用于凸问题时快速收敛。当应用于非凸函数训练神经网络时，学习轨迹可能穿过了很多不同的结构，最终到达一个局部是凸碗的区域。AdaGrad 根据平方梯度的整个历史收缩学习率，可能使得学习率在达到这样的凸结构前就变得太小了。RMSProp 使用指数衰减平均以丢弃遥远过去的历史，使其能够在找到凸碗状结构后快速收敛，它就像一个初始化于该碗状结构的 AdaGrad 算法实例。

RMSProp 相比于 AdaGrad，使用移动均值引入了一个新的超参数 ρ，控制移动平均的长度范围。

经验上，RMSProp 已被证明是一种有效且实用的深度神经网络优化算法。目前它是深度学习从业者经常采用的优化方法之一。

Algorithm 4.5 RMSProp 算法

Require: 全局学习率 ϵ，衰减速率 ρ

Require: 初始参数 θ

Require: 小常数 δ，通常设为 10^{-6}（用于被小数除时的数值稳定）

初始化累积变量 $r = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{x^{(1)}, \ldots, x^{(m)}\}$ 的小批量，对应目标为 $y^{(i)}$。

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

累积平方梯度：$r \leftarrow \rho r + (1 - \rho) g \odot g$

计算参数更新：$\Delta\theta = -\frac{\epsilon}{\sqrt{\delta + r}} \odot g$ ($\frac{1}{\sqrt{\delta + r}}$ 逐元素应用)

应用更新：$\theta \leftarrow \theta + \Delta\theta$

end while

Algorithm 4.6 使用 Nesterov 动量的 RMSProp 算法

Require: 全局学习率 ϵ，衰减速率 ρ，动量系数 α

Require: 初始参数 θ，初始参数 v

初始化累积变量 $r = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{x^{(1)}, \ldots, x^{(m)}\}$ 的小批量，对应目标为 $y^{(i)}$。

计算临时更新：$\tilde{\theta} \leftarrow \theta + \alpha v$

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

累积梯度：$r \leftarrow \rho r + (1 - \rho) g \odot g$

计算速度更新：$v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot g$ ($\frac{1}{\sqrt{r}}$ 逐元素应用)

应用更新：$\theta \leftarrow \theta + v$

end while

Adam

Adam(kingma,2014) 是另一种学习率自适应的优化算法。“Adam” 这个名字派生自短语 “Adaptive moments”。早期算法背景下，它也许最好被看作结合 RMSProp 和具有一些重要区别的动量的变种。首先，在 Adam 中，动量直接并入了梯度一阶矩（带指数加权）的估计。将动量加入 RMSProp 最直观的方法是应用动量于缩放后的梯度。结合重放缩的动量使用没有明确的理论动机。其次，Adam 包括负责原点初始化的一阶矩（动量项）和（非中心的）二阶矩的估计修正偏置。RMSProp 也采用了（非中心的）二阶矩估计，然而缺失了修正因子。因此，不像 Adam，RMSProp 二阶矩估计可能在训练初期有很高的偏置。Adam 通常被认为对超参数的选择相当鲁棒，尽管学习速率有时需要从建议的默认修改。

Algorithm 4.7 Adam 算法

Require: 步长 ϵ（建议默认为：0.001）
Require: 矩估计的指数衰减速率，$\rho_1(0.9)$ 和 $\rho_2(0.99)$ 在区间 $[0,1)$ 内。
Require: 用于数值稳定的小常数 δ（建议默认为：10^{-8}）
Require: 初始参数 θ
初始化一阶和二阶矩变量 $s=0$, $r=0$，初始化时间步 $t=0$
while 没有达到停止准则 **do**
从训练集中 m 个样本 $\{x^{(1)},\ldots,x^{(m)}\}$ 的小批量，对应目标为 $y^{(i)}$。
计算梯度：$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta),y^{(i)})$
$t \leftarrow t+1$
更新有偏一阶矩估计：$s \leftarrow \rho_1 s+(1-\rho_1)g$
更新有偏二阶矩估计：$r \leftarrow \rho_2 r+(1-\rho_2)g\odot g$
修正一阶矩的偏差：$\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
修正二阶矩的偏差：$\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
计算更新：$\Delta\theta = -\epsilon\frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$（逐元素应用操作）
应用更新：$\theta \leftarrow \theta+\Delta\theta$
end while

在本节中，我们讨论了一系列算法，通过自适应每个模型参数的学习速率以解决优化深度模型中的难题。此时，一个自然的问题是：该选择哪种算法呢？

遗憾的是，目前在这一点上没有达成共识。(Schaul,2014) 展示了许多优化算法在大量学习任务上的价值比较。虽然结果表明，具有自适应学习率（由 RMSProp 和 AdaDelta 代表）的算法族表现得相当鲁棒，但没有单一的算法表现为最好的。

目前，最流行的活跃使用的优化算法包括 SGD，具动量的 SGD，RMSProp，具动量的 RMSProp，AdaDelta 和 Adam。此时，选择哪一个算法似乎主要取决于使用者对算法的熟悉程度（以便调节超参数）。

卷积网络

卷积网络是一种专门用来处理具有类似网格结构的数据的神经网络。是指那些至少在网络的一层中使用卷积运算来替代一般的矩阵乘法运算的神经网络。例如时间序列数据（可以认为是在时间轴上有规律地采样形成的一维网格）和图像数据（可以看作是二维的像素网格）。卷积网络在诸多应用领域都表现优异。卷积网络一词表明该网络使用了卷积这种数学运算。卷积是一种特殊的线性运算。

卷积运算

在通常形式中，卷积是对两个实值函数的一种数学运算。为了给出卷积的定义，我们从两个可能会用到的函数的例子出发。

假设我们正在用激光传感器追踪一艘宇宙飞船的位置。我们的激光传感器给出一个单独的输出 $x(t)$，表示宇宙飞船在时刻 t 的位置。x 和 t 都是实值的，这意味着我们可以在任意时刻从传感器中读出飞船的位置。

现在假设我们的传感器含有噪声。为了得到飞船位置的低噪声估计，我们对得到的测量结果进行平均。显然，时间上越近的测量结果越相关，所以我们采用一种加权平均的方法，对于最近的测量结果赋予更高的权值。我们可以采用一个加权函数 $w(a)$ 来实现，其中 a 表示测量结果据当前时刻的时间间隔。如果我们对任意时刻都采用这种加权平均的操作，就得到了对于飞船位置的连续估计函数 s：

$$s(t) = \int x(a)w(t-a)da. \tag{8}$$

这种运算就叫做卷积。

卷积运算

卷积运算通常用星号表示：

$$s(t) = (x * w)(t). \tag{9}$$

在我们的例子中，w 必须是一个有效的概率密度函数，否则输出就不再是一个加权平均。另外，w 在参数为负值时必须为 0，否则它会涉及到未来，这不是我们能够做到的。但这些限制仅仅是对我们这个例子来说。通常，卷积被定义在满足上述积分式的任意函数上，并且也可能被用于加权平均以外的目的。

在卷积神经网络的术语中，第一个参数（在这个例子中，函数 x）叫做输入，第二个参数（函数 w）叫做核。

在我们的例子中，激光传感器能够在任意时刻给出测量结果的想法是不现实的。一般地，当我们用计算机处理数据时，时间会被离散化，传感器会给出特定时间间隔的数据。所以比较现实的的假设是传感器每秒给出一次测量结果，这样，时间 t 只能取整数值。如果我们假设 x 和 w 都定义在整数时刻 t 上，就得到了离散形式的卷积：

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{10}$$

在机器学习的应用中，输入通常是高维数据数组，而核也是由算法产生的高维参数数组。我们把这种高维数组叫做张量。因为输入与核的每一个元素都分开存储，我们经常假设在存储了数据的有限点集以外，这些函数的值都为零。这意味着在实际操作中，我们可以统一地把无限的求和当作对有限个数组元素的求和来用。

最后，我们有时对多个维度进行卷积运算。例如，如果把二维的图像 I 作为输入，我们也相应的需要使用二维的核 K：

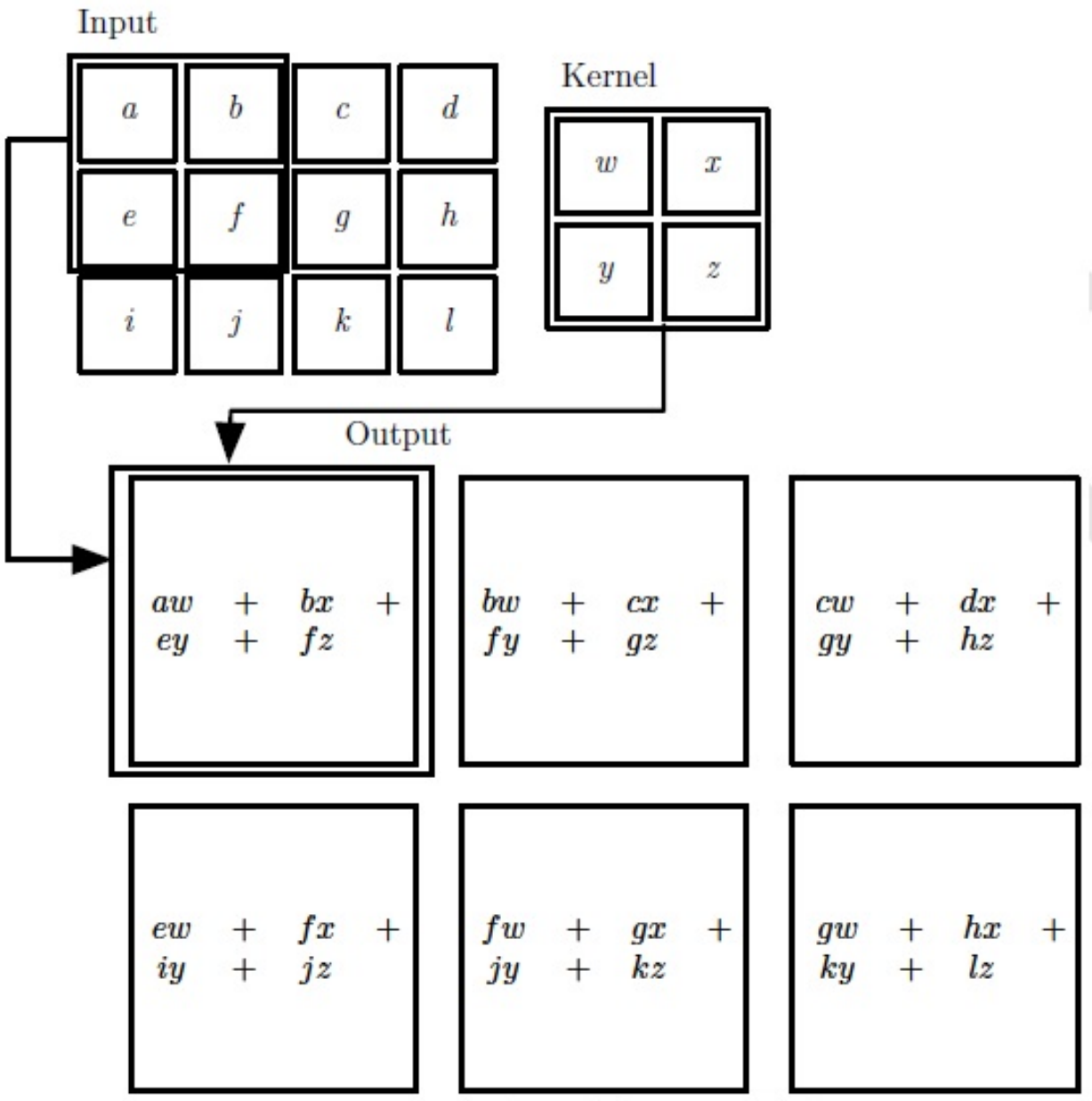
$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{11}$$

卷积运算可交换性的出现是因为我们相对输入了翻转，这意味着当 m 增大时，输入的索引增大，但核的索引相应的减小。翻转核的唯一目的就是为了得到可交换性。尽管可交换性在证明时很有用，但在神经网络的应用中却不是一个重要的性质。与之不同的是，许多神经网络库会实现一个相关的函数，称为互相关函数，和卷积运算几乎一样但是并不翻转核：

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{12}$$

许多机器学习的库使用互相关函数但是叫它卷积。这是因为核本身是对称的。这里我们遵循把两种运算都叫做卷积的这个传统。

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{13}$$



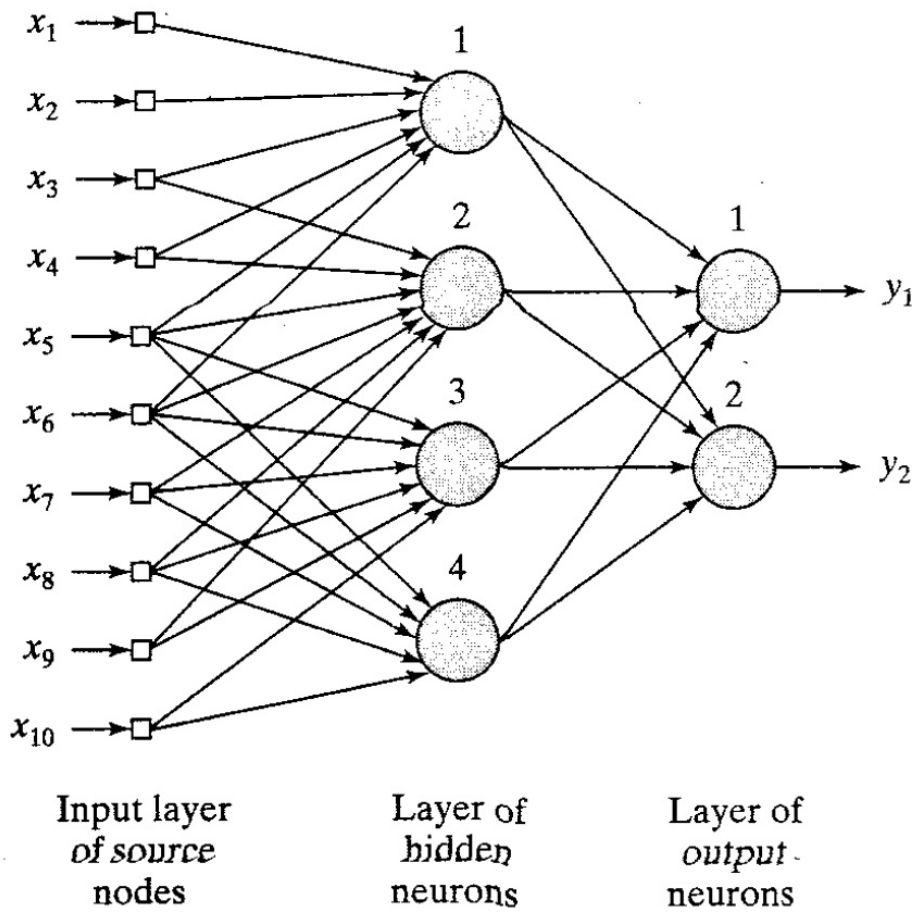
一个 2 维卷积的例子。我们限制只对核完全处在图像中的位置输出。我们用画有箭头的盒子来说明输出张量的左上角元素是如何通过对输入张量相应的左上角区域使用核进行卷积得到的。

卷积运算通过三个重要的思想来帮助改进机器学习系统：稀疏交互 sparse interactions、参数共享 parameter sharing、等变 equivariant representations。另外，卷积提供了一种处理大小可变的输入的方法。

传统的神经网络使用矩阵乘法来建立输入与输出的连接关系。其中，参数矩阵的每一个独立的参数都描述了每一个输入单元与每一个输出单元间的交互。这意味着每一个输出单元与每一个输入单元都产生交互。然而，卷积神经网络具有稀疏交互 sparse interactions，也叫做稀疏连接 sparse connectivity 或者稀疏权重 sparse weights 的特征。这通过使得核的规模远小于输入的规模来实现。举个例子，当进行图像处理时，输入的图像可能包含百万个像素点，但是我们可以通过只占用几十到上百个像素点的核来探测一些小的有意义的特征，例如图像的边缘。这意味着我们需要存储的参数更少，不仅减少了模型的存储需求，而且提高了它的统计效率。这也意味着为了得到输出我们只需要更少的计算量。这些效率上的提高往往是很显著的。

怎样在神经网络设计中加入先验信息

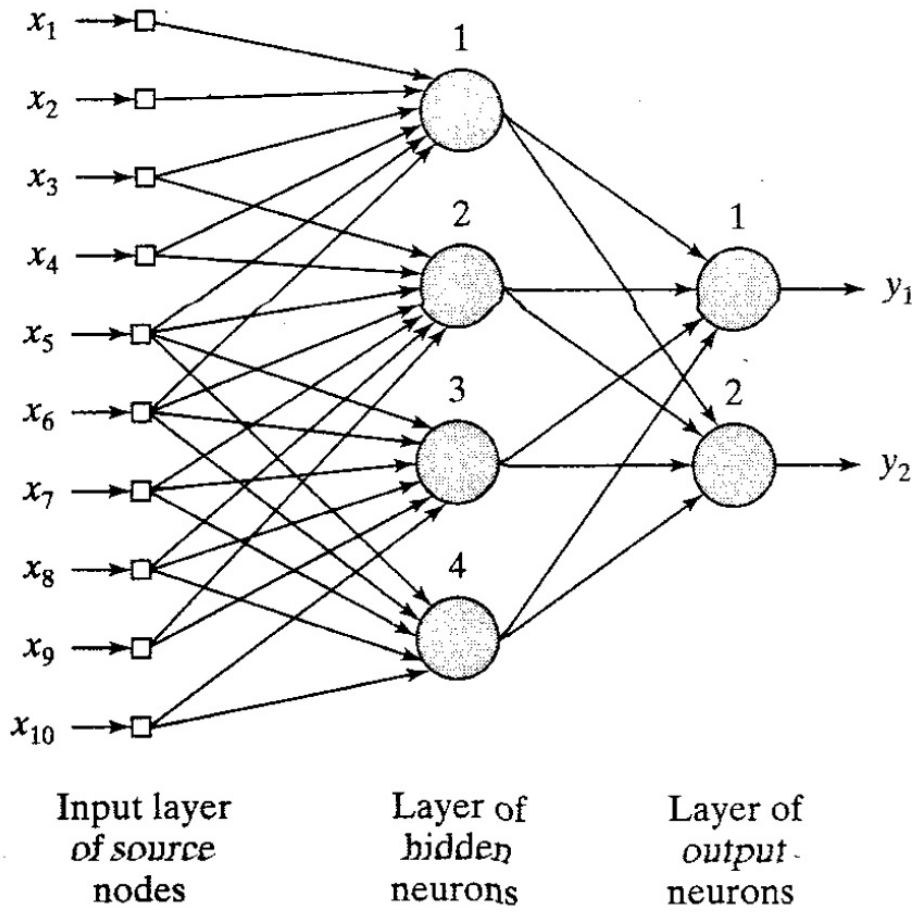
以此建立一种特定的网络结构，是必须考虑的重要问题。目前主要使用下面两种技术的结合



1. 通过使用称为接收域的局部连接，限制网络结构
2. 通过使用权值共享，限制突触权值的选择

联合利用接收域和权值共享的图例。所有四个隐神经元共享他们突触连接的相同权值集

怎样在神经网络设计中加入先验信息



每个隐藏神经元有 6 个局部连接，共有 4 个隐藏神经元，我们可以表示每个隐藏神经元的诱导局部域如下：

$$v_j = \sum_{i=1}^{6} w_i x_{i+j-1}, j = 1, 2, 3, 4$$

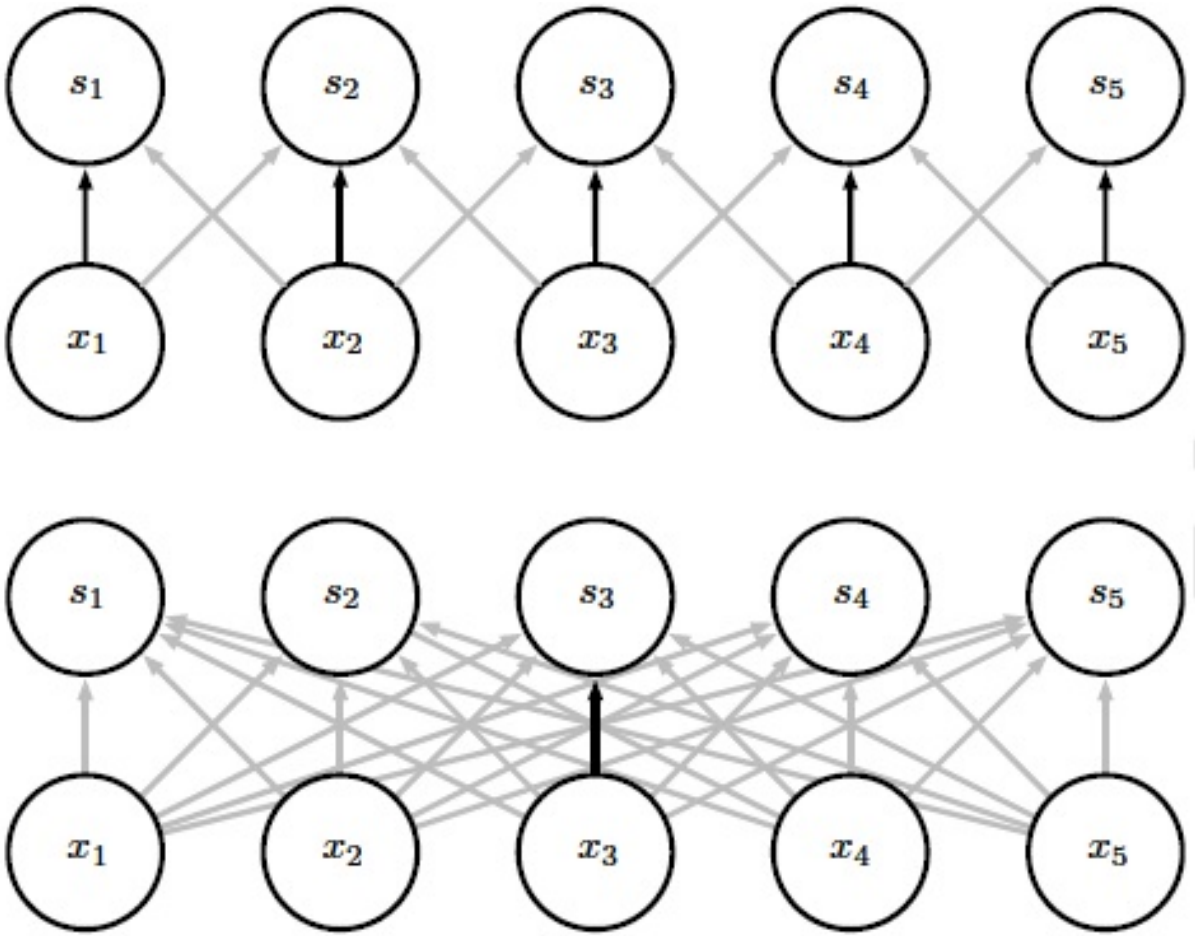
其中，$\{w_i\}_{i=1}^{6}$ 构成所有四个隐藏神经元共享的同一权值集。

这里描述的前馈网络使用局部连接和权值共享的方式，这样的前馈网络称为卷积网络。

参数共享

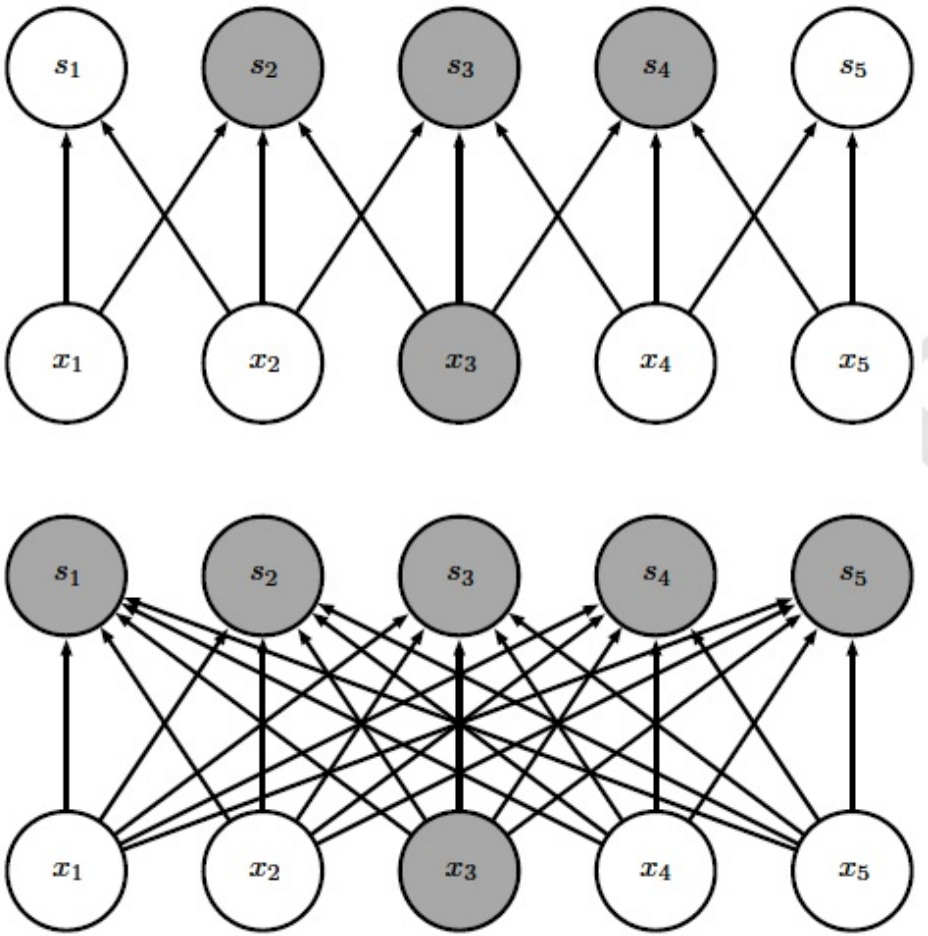
参数共享 parameter sharing 是指在一个模型的多个函数中使用相同的参数。在传统的神经网络中，当计算一层的输出时，权值矩阵的每一个元素只使用一次，当它乘以输入的一个元素后就再也不会用到了。作为参数共享的同义词，我们可以说一个网络含有绑定的权重，因为用于一个输入的权值也会被绑定在其他的权值上。在卷积神经网络中，核的每一个元素都作用在输入的每一位置上（除了一些可能的边界像素，取决于对于边界的决策设计）。卷积运算中的参数共享保证了我们只需要学习一个参数集合，而不是对于每一位置都需要学习一个单独的参数集合。因此，卷积在存储需求和统计效率方面极大地优于稠密矩阵的乘法运算。

参数共享



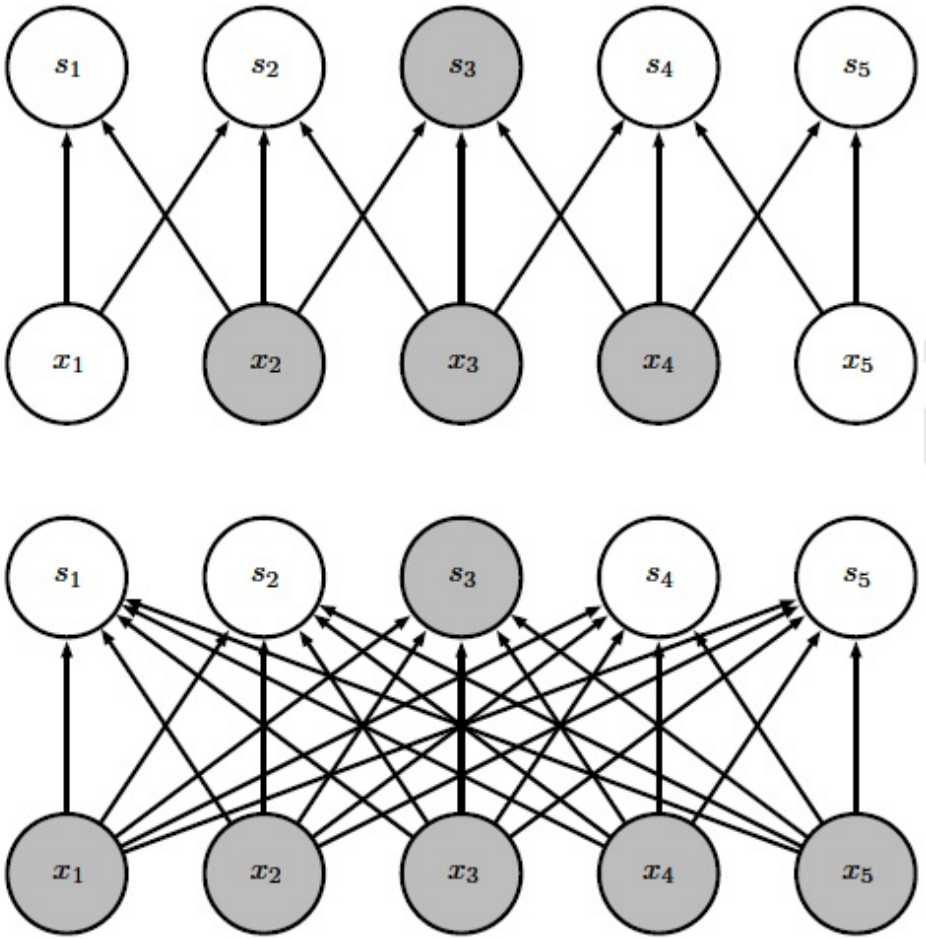
黑色箭头表示在两个不同的模型中使用了特殊参数的连接。(上图) 黑色箭头表示在卷积模型中 3 元素核的中间元素的使用。因为参数共享，这单个参数被用于所有的输入位置。(下图) 这单个黑色箭头表示在全连接模型中权重矩阵的中间元素的使用。这个模型没有使用参数共享，所以参数只使用了一次。

稀疏连接



对每幅图从下往上看。我们强调了一个输入单元 x_3 以及在 s 中受该单元影响的输出单元。(上图) 当 s 是由核宽度为 3 的卷积产生时，只有三个输出受到 x 的影响[2]。(下图) 当 s 是由矩阵乘法产生时，连接不再是稀疏的，所以所有的输出都会受到 x_3 的影响。

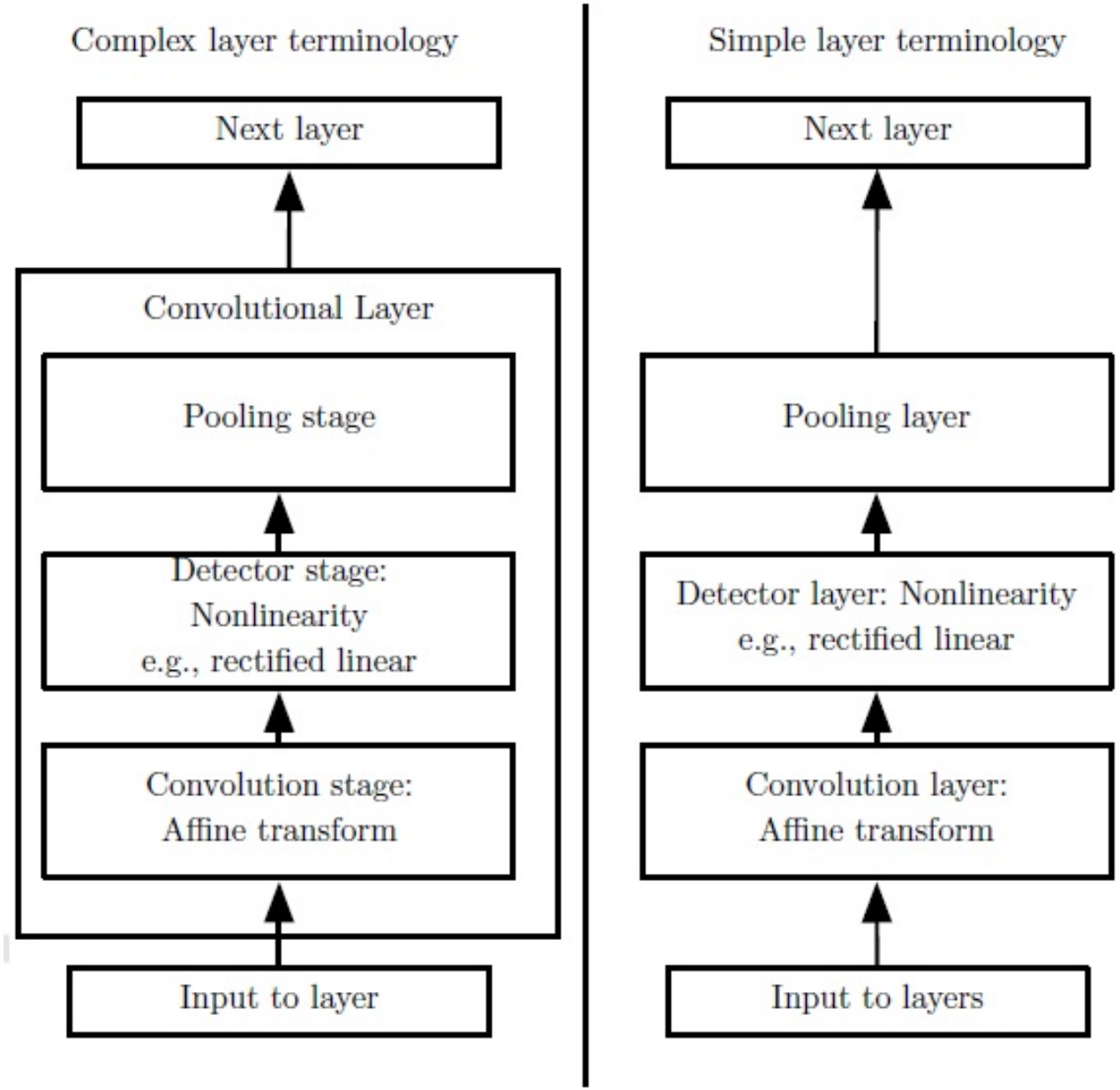
稀疏连接



对每幅图从上往下看。我们强调了一个输出单元 s_3 以及 x 中影响该单元的输入单元。这些单元被称为 s_3 的接收域。(上图) 当 s 是由核宽度为 3 的卷积产生时，只有三个输入影响 s_3。(下图) 当 s 是由矩阵乘法产生时，连接不再是稀疏的，所以所有的输入都会影响 s_3

Pooling

卷积神经网络的卷积层通常包含三级。在第一级中，卷积层并行地进行多个卷积运算来产生一组线性激活函数。在第二级中，非线性的激活函数如修正线性函数 ReLU 函数等作用在第一级中的每一个线性输出上。这一级有时也被称为探测级 detector stage。在第三级中，我们使用池化 pooling funciton 来更进一步地调整卷积层的输出。


Complex layer terminology
Next layer
Convolutional Layer
Pooling stage
Detector stage:
Nonlinearity
e.g., rectified linear
Convolution stage:
Affine transform
Input to layer
Simple layer terminology
Next layer
Pooling layer
Detector layer: Nonlinearity
e.g., rectified linear
Convolution layer:
Affine transform
Input to layers

TensorFlow 提供了池化操作的函数，最大池化和平均池化。

使用池化可以看作是增加了一个无限强的先验：卷积层学得的函数必须具有对少量平移的不变性。当这个假设成立时，池化可以极大地提高网络的统计效率。

局部平移不变性是一个很重要的性质，尤其是当我们关心某个特征是否出现而不关心它出现的具体位置时

卷积神经网络也许是生物学启发人工智能的最为成功的故事。虽然卷积神经网络已经被许多其他领域指导，但是神经网络的一些关键设计原则来自神经科学。

卷积神经网络的历史始于神经科学实验，远早于相关计算模型的发展。神经生理学家 David Hubel 和 Torsten Wiesel 合作多年，为了确定关于哺乳动物视觉系统如何工作的许多最基本的事实。他们的成就最终获得了诺贝尔奖。他们的发现对当代深度学习模型有最大影响的是基于记录猫的单个神经元的活动。他们观察了猫的脑内神经元如何响应投影在猫前面屏幕上精确位置的图像。他们的伟大发现是，处于视觉系统较为前面的神经元对非常特定的光模式（例如精确定向的条纹）反应最强烈，但对其他模式几乎完全没有反应。

他们的工作有助于表征大脑功能的许多方面。从深度学习的角度来看，我们可以专注于简化的，卡通形式的大脑功能视图。

在这个简化的视图中，我们关注被称为 V1 的大脑的一部分，也称为初级视觉皮层。V1 是大脑对视觉输入开始执行显著高级处理的第一个区域。在该卡通视图中，图像是由光到达眼睛并刺激视网膜（眼睛后部的光敏组织）形成的。视网膜中的神经元对图像执行一些简单的预处理，但是基本不改变它被表示的方式。然后图像通过视神经和称为外侧膝状核的脑部区域。这些解剖区域的主要作用是仅仅将信号从眼睛传递到位于头后部的 V1。

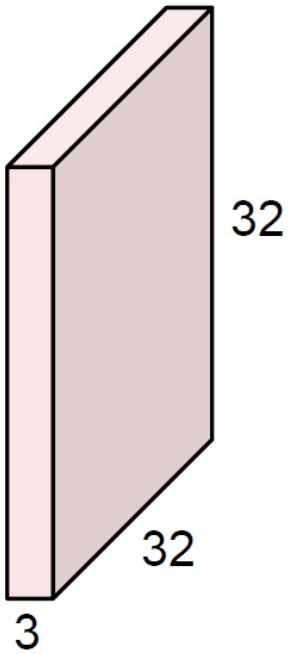
卷积层被设计为描述 V1 的三个性质：

1. V1 布置在空间图中。它实际上具有二维结构来反映视网膜中的图像结构。例如，到达视网膜下半部的光仅影响 V1 相应的一半。卷积网络通过用二维映射定义特征的方式来描述该特性。
2. V1 包含许多简单细胞。简单细胞的活动在某种程度上可以概括为在一个小的空间位置接受域内的图像的线性函数。卷积网络的检测器单元被设计为模拟简单细胞的这些性质。
3. V1 还包括许多复杂细胞。这些细胞响应类似于由简单细胞检测的那些特征，但是复杂细胞对于特征的位置微小偏移具有不变性。这启发了卷积网络的池化单元。复杂细胞对于照明中的一些变化也是不变的，不能简单地通过在空间位置上池化来刻画。这些不变性激发了卷积网络中的一些跨通道池化策略。

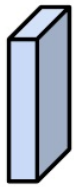
Convolution Layer

Filters always extend the full depth of the input volume

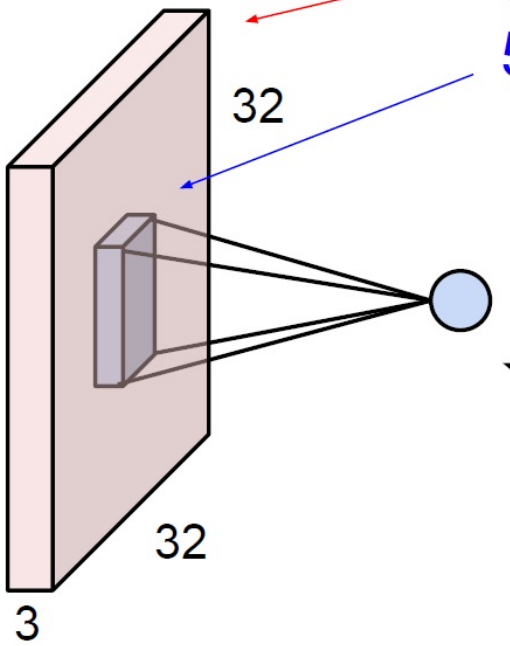
32x32x3 image



5x5x3 filter



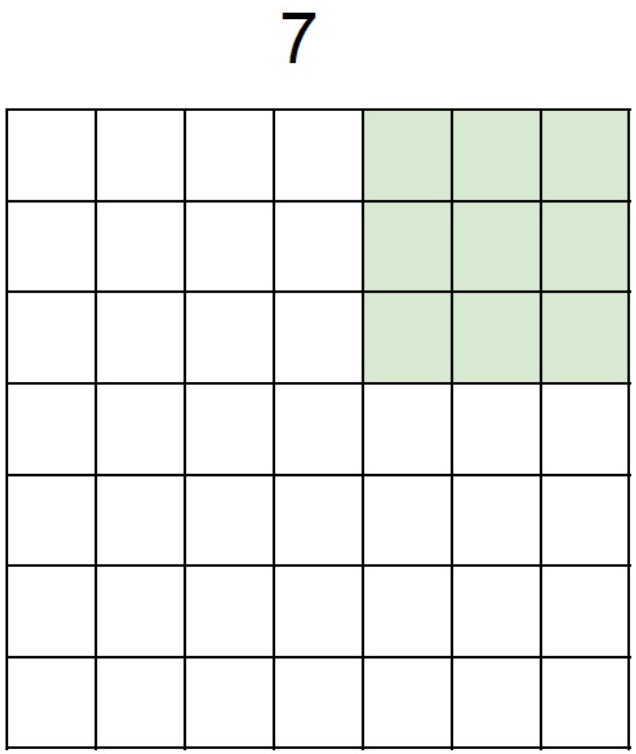
Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

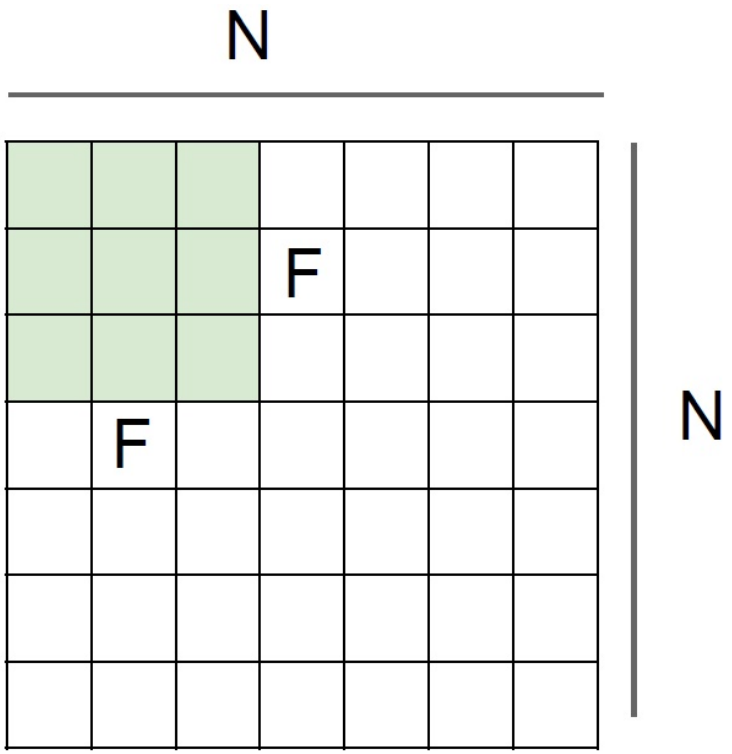
$$w^T x + b$$

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

=> 5x5 output



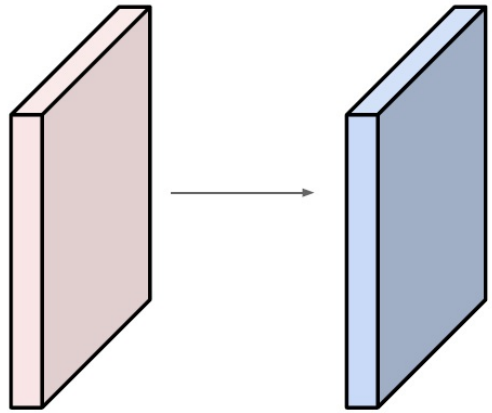
Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

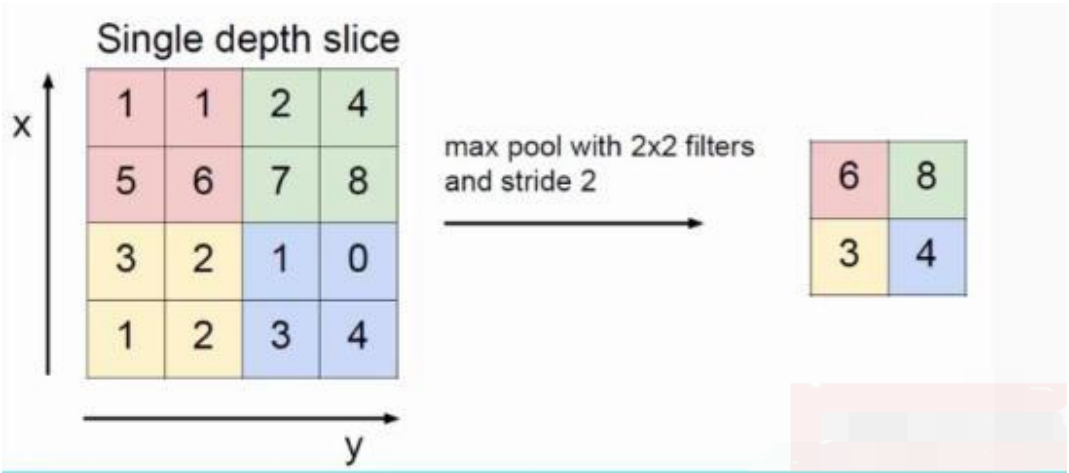
Examples time:



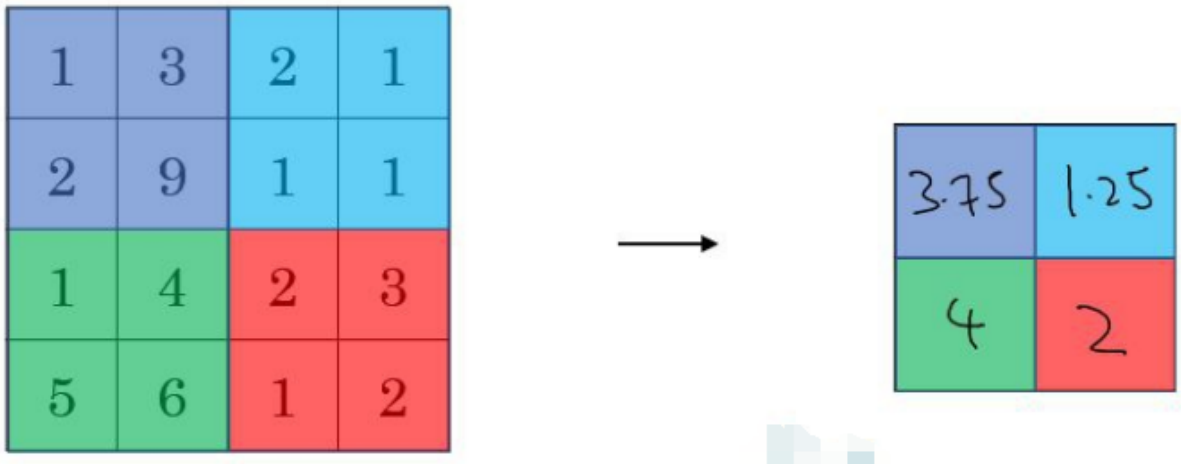
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

☞Max Pooling



☞Average Pooling



池化层没有参数
池化的作用：

- 保留主要特征的同时减少参数和计算量，防止过拟合。
- invariance(不变性)，这种不变性包括 translation(平移)，rotation(旋转)，scale(尺度)。

卷积神经网络实例

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
trainset = torchvision.datasets.MNIST(
    root='./mnist/',
    train=True,
    download=True,
    transform=transforms.ToTensor())

trainloader = torch.utils.data.DataLoader(
    trainset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    )
```

卷积神经网络实例

```
testset = torchvision.datasets.MNIST(
    root='./mnist/',
    train=False,
    download=True,
    transform=transforms.ToTensor())

testloader = torch.utils.data.DataLoader(
    testset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    )
```

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=1,
                      out_channels=6,
                      kernel_size=5,
                      stride=1,
                      padding=2) ,
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(6, 16, 5),
            nn.ReLU(),          #input_size=(16*10*10)
            nn.MaxPool2d(kernel_size=2, stride=2)  #output_size=(16*5*5)
        )
```

```
        self.fc1 = nn.Sequential(
            nn.Linear(in_features=16 * 5 * 5,
                      out_features = 120,  bias=True),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(120, 84),
            nn.ReLU()
        )
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size()[0], -1)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

```
EPOCH = 8
BATCH_SIZE = 64
LR = 0.001
net = LeNet().cuda()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9)
```

```
if __name__ == "__main__":
    net.train()
    for epoch in range(EPOCH):
        sum_loss = 0.0

        for i, data in enumerate(trainloader):
            inputs, labels = data
            inputs, labels = inputs.cuda(), labels.cuda()

            optimizer.zero_grad()

            # forward + backward
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
```

```
sum_loss += loss.item()
if i % 100 == 99:
    print('[%d, %d] loss: %.03f'
          % (epoch + 1, i + 1, sum_loss / 100))
    sum_loss = 0.0
```

```
net.eval()
correct = 0
total = 0
for data in testloader:
    images, labels = data
    images, labels = images.cuda(), labels.cuda()
    outputs = net(images)

    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum()
print('Accuracy: %d' % (100 * correct / total))
```