深度学习基础及数学原理

Hao Zhang

haomoodzhang@gmail.com

2016 年 9 月 26 日

目录

Chapter 1

引言

图像分类 *(image classification)* 问题是指, 假设给定一系列离散的类别 *(categories)*(如猫, 狗, 飞机, 货车, ...), 对于给定的图像, 从这些类别中赋予一个作为它的标记 *(label)*. 图像分类问题是计算机视觉领域的核心问题之一, 也与目标检测 *(object detection)*, 目标分割 *(object segmentation)* 等其他计算机视觉领域核心问题有密切的联系. [1] 当今, 我们正处在信息时代和数字时代, 充斥着大量的数字图像, 诸多的实际应用场景需要计算机能正确和高效地理解图像, 图像分类正是理解图像的基础.

人类从图像中进行目标识别非常容易, 但是计算机看到的图像是一组 0 至 255 之间的数字, 语义鸿沟 *(semantic gap)* 的存在使图像识别成为一项极具挑战性的任务. 除此之外, 拍摄视角, 光照, 背景, 遮挡等因素可能使具有相同类别的图像之间像素值会十分不相似而不同类别的图像之间像素值很相似, 这使得我们不能通过显式地指定若干规则来对图像中的目标进行识别. 因此, 我们借鉴人类的学习过程, 给定很多的训练数据, 让计算机从训练数据中学习如何做分类, 这叫做数据驱动过程 *(data-driven approach)*.

使用深度学习中卷积神经网络 *(convolutional neuralnetwork, CNN)* 是现在进行图像识别的主流方法, 目前效果最佳的卷积神经网络做图像分类的准确率已经超过人 [28]. 除图像分类外, 卷积神经网络还广泛应用于很多领域, 如目标识别 [8], 图像分割 [22], 视频分类 [15], 场景分类 [36], 人脸识别 [32], 深度估计 [5], 从图像中生成语言描述 [14] 等.

本文剩余部分将如下安排: 第 2 章将简述图像识别问题的挑战以及为

[1] 在下文将这些任务统称为图像识别 *(image recognition)*.

了应对挑战采用的数据驱动过程; 第 3 章将以 softmax 线性分类器为例介绍数据驱动过程的各个流程; 第 4 章使用神经网络扩展 softmax 线性分类器以解决非线性问题; 第 5 章讨论卷积神经网络的组成及学习方法; 第 6 章将讨论一些具体实现细节.

Chapter 2

图像识别问题的挑战及数据驱动过程

图像识别任务面临着诸多挑战, 这使得它自计算机视觉领域 1966 年诞生以来就成为一个十分活跃的子领域. 本章将简要讨论图像识别问题的挑战以及为了应对这些挑战而使用的数据驱动过程.

2.1 图像分类问题的挑战

虽然从图像中识别一个对象对人类来说非常的简单, 但图像识别对计算机来说是一项极具挑战性的工作. 在计算机内, 图像是由一个很大三维数组表示的. 比如一张 1024×768 的图像, 它拥有 R, G, B 三个分量, 因此, 这张图像有 $1024 \times 768 \times 3 = 2,359,296$ 个像素, 每个像素是一个 0(黑) 到 255(白) 之间的整数. 这种现象, 称为语义鸿沟. 图像分类的任务是将这两百万个数字映射到一个标记, 比如 “猫”.

除了语义鸿沟之外, 图像识别还有其他的一些挑战, 见图2.1.

- 视角变化. 一个相同的目标相对摄像机可以有不同的朝向.
- 尺度变化. 不仅是占据图像的相对大小, 目标在真实世界的大小也会发生变化.
- 形变. 许多目标并不是刚体, 有时会有很极端的形变.
- 遮挡. 目标可能被遮挡, 因此只有一小部分是可见的.
- 光照改变. 光照会对像素值的大小产生巨大的变化.

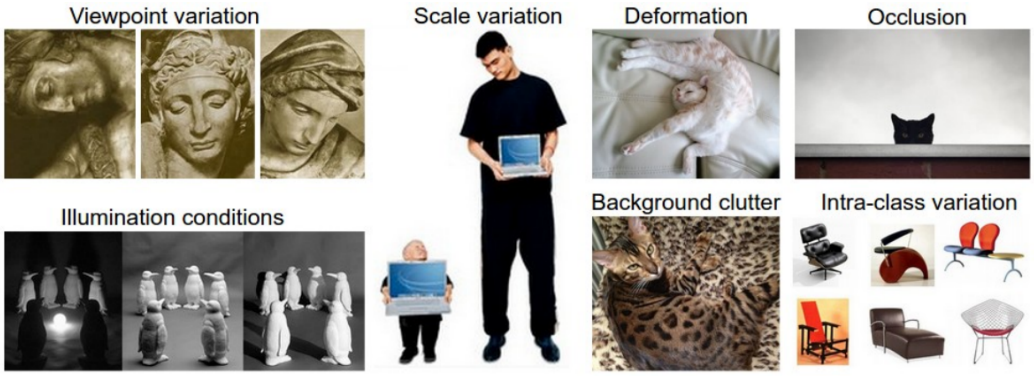


图 2.1: 图像识别问题的挑战. 图像来源于 [7].

- 背景融合. 目标可能会和背景混为一体, 使得它非常难以被认出.
- 类内变化. 相同类的不同个体之间可能会变的完全不同.

2.2 数据驱动过程

我们该怎样写出一个算法对图像进行分类呢? 和写出一个排序算法不同, 我们不知道如果通过指定一系列识别规则的方法来识别图像中的目标并且能应对上述的这些挑战. 回想我们人类能对图像内容进行有效的识别, 是因为我们之前已经积累了许多经验, 通过对经验的利用, 从而对新情况做出判断 [37].

在计算机系统中, 经验通常是以数据的形式存在. 我们将提供给计算机每个类别的许多*实例 (examples)*, 它们组成了*训练集 (training set)*, 利用*学习算法 (learning algorithms)* 从训练集中产生*分类器 (classifier)* 或模型 *(model)*. 在面对新情况时 (例如看到一张以前未出现的图像), 模型会提供相应的判断. 这个过程, 叫做数据驱动过程.

Chapter 3

线性分类器

本章将以 softmax 线性分类器为例, 讨论数据驱动过程的各组成部分. 同时本章是后文非线性分类器和深度学习的铺垫.

3.1 训练数据

给定由 m 张图像组成的训练集, 每个图像的标记是 K 个不同类中的一个,

$$D = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{m}\,, \tag{3.1}$$

$$\vec{x}^{(i)} \in \mathbb{R}^n, \forall i\,, \tag{3.2}$$

$$y^{(i)} \in \{0, 1, 2, 3, \ldots, K-1\}, \forall i\,. \tag{3.3}$$

其中, i 用于对训练实例进行索引. $\vec{x}^{(i)}$ 是第 i 张图像展成列向量之后的结果. n 是每个向量的维数, 若图像的大小是 $1024 \times 768 \times 3$, 则 $n = 1024 \times 768 \times 32 = 359296$. $y^{(i)}$ 是 $\vec{x}^{(i)}$ 对应的标记.

3.2 假设函数

在给定训练集后, 我们将从训练集中学到一个映射 h, 称为*假设 (hypothesis)*, 使得 $h(\vec{x})$ 给出了 $\vec{x}$ 属于某个类的信心或分数, 从而能进一步对 y 做预测, 见图3.1. 对于不同的分类模型, h 有不同的函数形式.

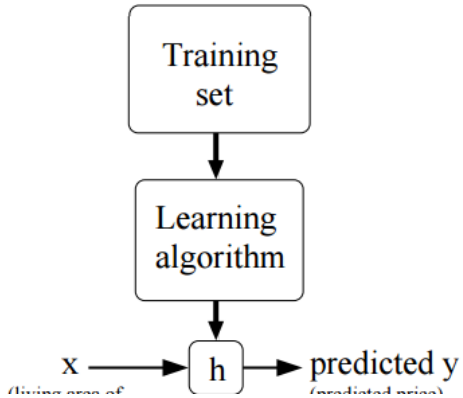


图 3.1: 假设函数. 图像来源于 [25].

3.2.1 线性分类模型的假设函数

在线性分类模型中, 假设函数 *(hypothesis)* h 采用的是一种最简单的线性映射

$$h(\vec{x}; W, \vec{b}) = W\vec{x} + \vec{b}, \tag{3.4}$$

$$W \in \mathbb{R}^{K\times n}, \vec{b} \in \mathbb{R}^{K}. \tag{3.5}$$

$W, \vec{b}$ 是函数的参数, W 通常叫做权值 *(weights)*, $\vec{b}$ 通常叫做偏置向量 *(bias vector)*.

我们的目标是设置 $W, \vec{b}$ 使得我们计算的分数匹配真实值 *(ground truth)*, 这个过程称作学习参数 $W, \vec{b}$. 一旦学习过程结束, 我们可以丢弃训练集, 通过参数 $W, \vec{b}$ 即可做预测.

3.2.2 对假设函数的理解

公式3.4可以看作是同时有 K 个分类器进行计算

$$h(\vec{x}; W, \vec{b}) = W\vec{x} + \vec{b} \tag{3.6}$$

$$= \begin{bmatrix} \vec{w}_0^T \\ \vec{w}_1^T \\ \vdots \\ \vec{w}_{K-1}^T \end{bmatrix} \vec{x} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{K-1} \end{bmatrix} \tag{3.7}$$

$$= \begin{bmatrix} \vec{w}_0^T \vec{x} + b_0 \\ \vec{w}_1^T \vec{x} + b_1 \\ \vdots \\ \vec{w}_{K-1}^T \vec{x} + b_{K-1} \end{bmatrix} . \tag{3.8}$$

其中 d 用于对 W 中各元素索引. $\vec{w}_d^T \vec{x} + b_d$ 计算的是 $\vec{x}$ 属于第 d 个类的信心或分数.

另外一种理解方式是将 $\vec{w}_d^T \vec{x}$ 看作是一种相似度的度量, 每个 $\vec{w}_d$ 代表了每个类的一个原型 *(prototype)*. 预测的过程是将 $\vec{x}$ 与各个类的原型相比较, 找到一个与 $\vec{x}$ 最相似的类.

3.2.3 Softmax 分类器的假设函数

Softmax 分类器是一种线性分类器, 它对于 $\vec{s} = h(\vec{x}) = W\vec{x} + \vec{b}$ 计算得到的各类的分数有概率上的理解

$$\Pr(y = k|\vec{x}) = \frac{\exp(s_k)}{\sum_{j=0}^{K-1} \exp(s_j)} \in [0, 1], \forall k \,. \tag{3.9}$$

其中 $\frac{\exp(s_k)}{\sum_{j=0}^{K-1} \exp(s_j)}$ 称为 softmax 函数.

3.3 损失函数

在给定假设函数 h 后, 我们可以计算对于每个数据 $\vec{x}^{(i)}$ 得到的对各个类的分数. 我们需要一个能通过比较假设函数得到的分数与数据真实值 $y^{(i)}$ 相符 (或不相符) 程度的度量, 通过这个度量来定量的表示当前参数的好坏. 损失函数 *(loss function)* 计算的是不相符的程度, 即当损失函数高的时候当前参数 $W, \vec{b}$ 表现很差, 当损失函数低的时候当前参数 $W, \vec{b}$ 表现很好.

3.3.1 交叉熵损失

通过最大似然估计 *(maximum likelihood estimate)* 可以得到 softmax 分类器的损失函数.

$$W^\star, \vec{b}^\star = \arg\max_{W,\vec{b}} L(W, \vec{b}) \tag{3.10}$$

$$= \arg\max_{W,\vec{b}} p(D; W, \vec{b}) \tag{3.11}$$

$$= \arg\max_{W,\vec{b}} \prod_{i=1}^{m} p(y^{(i)}|\vec{x}^{(i)}; W, \vec{b}) \qquad \text{(假设数据间独立同分布 (iid))} \tag{3.12}$$

$$= \arg\max_{W,\vec{b}} \log \prod_{i=1}^{m} p(y^{(i)}|\vec{x}^{(i)}; W, \vec{b}) \tag{3.13}$$

$$= \arg\max_{W,\vec{b}} \sum_{i=1}^{m} \log p(y^{(i)}|\vec{x}^{(i)}; W, \vec{b}) \tag{3.14}$$

$$= \arg\max_{W,\vec{b}} \sum_{i=1}^{m} \log \frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{j=0}^{K-1} \exp(s^{(i)}_j)} \qquad (\vec{s}^{(i)} = W\vec{x}^{(i)} + \vec{b}) \tag{3.15}$$

$$= \arg\min_{W,\vec{b}} \sum_{i=1}^{m} -\log \frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{j=0}^{K-1} \exp(s^{(i)}_j)} \tag{3.16}$$

$$= \arg\min_{W,\vec{b}} \frac{1}{m} \sum_{i=1}^{m} -\log \frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{j=0}^{K-1} \exp(s^{(i)}_j)} \tag{3.17}$$

$$= \arg\min_{W,\vec{b}} \frac{1}{m} \sum_{i=1}^{m} \left(-s^{(i)}_{y^{(i)}} + \log(\sum_{j=0}^{K-1} \exp(s^{(i)}_j)) \right) \tag{3.18}$$

$$\stackrel{\text{def}}{=} \arg\min_{W,\vec{b}} \frac{1}{m} \sum_{i=1}^{m} \text{err}(W, \vec{b}; \vec{x}^{(i)}, y^{(i)}) \tag{3.19}$$

$$\stackrel{\text{def}}{=} \arg\min_{W,\vec{b}} J(W, \vec{b})\,. \tag{3.20}$$

其中, $\text{err}(W, \vec{b}; \vec{x}^{(i)}, y^{(i)}) = -\log \frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{j=0}^{K-1} \exp(s^{(i)}_j)} = -s^{(i)}_{y^{(i)}} + \log(\sum_{j=0}^{K-1} \exp(s^{(i)}_j))$ 称为交叉熵损失 *(hinge loss)*. 在后文, 为了叙述方便, 将 $\text{err}(W, \vec{b}; \vec{x}^{(i)}, y^{(i)})$ 记做 $\text{err}^{(i)}$.

3.3.2 正则化

在上一子节我们知道了对 softmax 来说, 好的 $(W, \vec{b})$ 使得

$$J(W, \vec{b}) = \frac{1}{m} \sum_{i=1}^{m} -\log \frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{j=0}^{K-1} \exp(s^{(i)}_j)} \tag{3.21}$$

达到最小值. 但是根据式3.21计算得到的 $(W, \vec{b})$ 并不唯一. 假设 $(W^\star, \vec{b}^\star)$ 是最优解, 那么给 $W^\star$ 和 $\vec{b}^\star$ 中所有元素都加一个常数 $c, c > 0$ 得到的结果也是式3.21的最优值.

为了解决这个问题, 我们将对 $(W, \vec{b})$ 的取值设定一些偏好, 这个偏好的

设定通过在损失函数中添加正则项 *(regularization)*$\Omega(W)$ 来实现. 最常用的正则项利用 ℓ_2 范数 *(norm)*(对应于矩阵是 ℓ_F 范数)

$$\Omega(W) = \|W\|_F^2 = \sum_{d=0}^{K-1}\sum_{j=0}^{n-1} W_{dj}\,. \tag{3.22}$$

ℓ_2 正则化倾向于小而且分散的权值, 因此分类器倾向于考虑 $\vec{x}$ 的全部维度同时每个维度对输出的影响较小, 而不是只考虑 $\vec{x}$ 的几个维度同时这几个维度对输出有很大影响. 这样的正则化有助于提高泛化 *(generalization)* 能力.

这样, 完整的损失函数变成

$$J(W,\vec{b}) = \frac{1}{m}\sum_{i=1}^{m} -\log\frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{j=0}^{K-1}\exp(s^{(i)}_j)} + \frac{\lambda}{2}\|W\|_F^2 \tag{3.23}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\mathrm{err}^{(i)} + \frac{\lambda}{2}\|W\|_F^2\,. \tag{3.24}$$

其中, λ 是一个数字, 用于调节正则化对损失函数的影响, $\frac{1}{2}$ 是为了随后的数学计算方便一些.

3.4 优化

我们的目的是找到 $(W,\vec{b})$, 最小化损失函数

$$W^\star,\vec{b}^\star = \arg\min_{W,\vec{b}} J(W,\vec{b})\,. \tag{3.25}$$

这是个无约束的优化问题.

3.4.1 梯度下降

当训练集给定后, 损失函数 J 只是参数 $(W,\vec{b})$ 的函数. 通常 $(W,\vec{b})$ 维数很高, 我们很难可视化 $(W,\vec{b})$ 与 J 的关系. 但是, 通过选取参数空间的某两个方向 (θ_0,θ_1), 做出 J 的值沿着 (θ_0,θ_1) 方向变化的曲线, 即计算 $J(W+c_0\theta_0+c_1\theta_1,\vec{b})$ 随 (c_0,c_1) 的变化, 其中 $(W,\vec{b})$ 是随机给定的初始点, 对 $\vec{b}$ 的可视化方法也是一样的. 根据曲线, 我们仍然可以得到一些直观上的理解, 见图3.2.

直接找到最佳的参数 $(W,\vec{b})$ 通常是很难的 (尤其在神经网络中, $J(W,\vec{b})$ 是非凸函数). 因此, 我们的策略是开始时随机选定一组参数 $(W,\vec{b})$, 之后迭

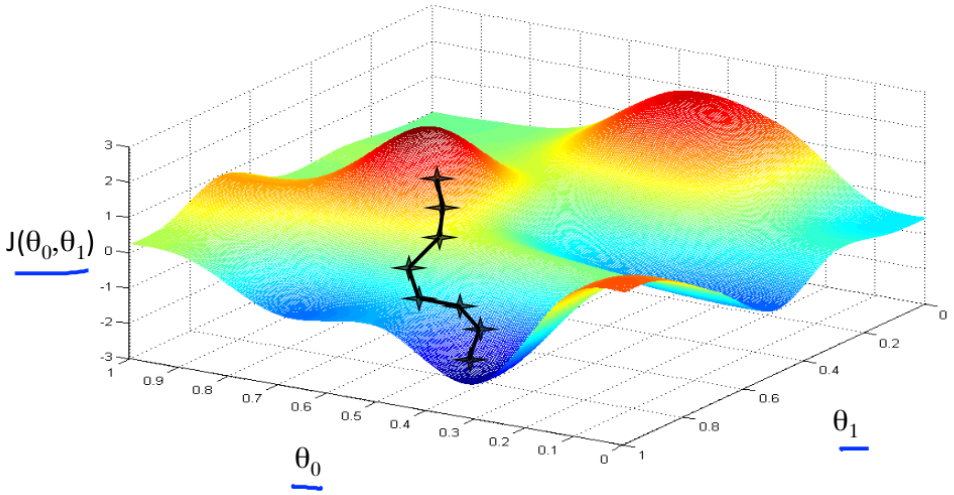


图 3.2: 梯度下降过程. 图像来源于 [26].

代地修正 $(W, \vec{b})$ 使得 $J(W, \vec{b})$ 越来越小. 对应于图3.2, 开始时随机位于 J 曲线上的某一点, 之后每次朝着到达谷底最快的方向前进一步.

将 W 展成列向量并与 $\vec{b}$ 拼接成一个大的列向量 $\vec{\theta}$, $\vec{\theta}$ 的维度是 $K \times n + K$. 因此, 优化问题可以写成

$$\min_{\vec{\theta}} J(\vec{\theta}) \,. \tag{3.26}$$

这仍然是个无约束的优化问题.

在每一步, 我们要找到一个方向 $\|\vec{v}\| = 1$, 使得 J 在这个方向下降最快, 然后朝着这个方向前进 $\eta > 0$ 大小. 优化变成带约束的优化问题

$$\min_{\|\vec{v}\|=1} J(\vec{\theta} + \eta\vec{v}) \,. \tag{3.27}$$

当 η 很小时, J 可以用泰勒展开近似:

$$\begin{aligned}
\vec{v}^{\star} &= \arg\min_{\|\vec{v}\|=1} J(\vec{\theta} + \eta\vec{v}) && (3.28)\\
&\approx \arg\min_{\|\vec{v}\|=1} J(\vec{\theta}) + \eta\vec{v}^{T}\nabla J(\vec{\theta}) && (3.29)\\
&= \arg\min_{\|\vec{v}\|=1} J(\vec{\theta}) + \eta\|\vec{v}\|\|\nabla J(\vec{\theta})\|\cos\langle\vec{v}, \nabla J(\vec{\theta})\rangle && (3.30)\\
&= \arg\min_{\|\vec{v}\|=1} J(\vec{\theta}) + \eta\|\nabla J(\vec{\theta})\|\cos\langle\vec{v}, \nabla J(\vec{\theta})\rangle && (3.31)
\end{aligned}$$

$$= \arg\min_{\|\vec{v}\|=1} \cos\langle \vec{v}, \nabla J(\vec{\theta})\rangle \tag{3.32}$$

$$= -\frac{\nabla J(\vec{\theta})}{\|\nabla J(\vec{\theta})\|}\,. \tag{3.33}$$

因此, 最佳的方向 $\vec{v}$ 是梯度 $\nabla J(\vec{\theta})$ 的反方向.

η 的选择对优化的收敛有决定性的影响. η 过小时收敛过慢, η 过大时收敛过程会很不稳定. 我们希望 η 在开始时大一些以更快收敛, 而在之后变小使得不会错过最优值. 可以取 $\eta = \alpha\|\nabla J(\vec{\theta})\|$ 达到. 开始时 $\|\nabla J(\vec{\theta})\|$ 较大, 在收敛时 $\|\nabla J(\vec{\theta})\| = 0$. 更新规则是

$$\vec{\theta} \leftarrow \vec{\theta} - \eta\frac{\nabla J(\vec{\theta})}{\|\nabla J(\vec{\theta})\|} \tag{3.34}$$

$$= \vec{\theta} - \alpha\|\nabla J(\vec{\theta})\|\frac{\nabla J(\vec{\theta})}{\|\nabla J(\vec{\theta})\|} \tag{3.35}$$

$$= \vec{\theta} - \alpha\nabla J(\vec{\theta})\,. \tag{3.36}$$

式3.36就是*梯度下降 (gradient descent, GD)*, α 是*学习速率 (learning rate)*.

用 $(W, \vec{b})$ 表示的梯度下降规则是

$$W \leftarrow W - \alpha\nabla_W J(W, \vec{b}) = W - \alpha\frac{1}{m}\sum_{i=1}^{m}\nabla_W \mathrm{err}^{(i)} - \alpha\lambda W\,, \tag{3.37}$$

$$\vec{b} \leftarrow \vec{b} - \alpha\nabla_{\vec{b}} J(W, \vec{b}) = \vec{b} - \alpha\frac{1}{m}\sum_{i=1}^{m}\nabla_{\vec{b}}\mathrm{err}^{(i)}\,. \tag{3.38}$$

注意式3.37, 3.38是同时对 $(W, \vec{b})$ 进行更新, 而不是先更新 W, 再更新 $\vec{b}$.

3.4.2 随机梯度下降

在更新规则式3.37, 3.38中, 每更新一次, 需要把所有的训练集看过一遍, 因此它也称为*批量梯度下降 (batch gradient descent)*. 当 m 很大时, 例如在 ImageNet[3] 中 m 是百万级别, 这样的更新就显得效率很低.

如果利用 ∇J 的无偏估计来替代 ∇J, 这样每更新一次只需要一个训练实例

$$W \leftarrow W - \alpha\nabla_W \mathrm{err}^{(i)} - \alpha\lambda W\,, \tag{3.39}$$

$$\vec{b} \leftarrow \vec{b} - \alpha\nabla_{\vec{b}}\mathrm{err}^{(i)}\,. \tag{3.40}$$

这种更新规则为随机梯度下降 *(stochastic gradient descent, SGD)*. SGD 通常比 GD 更快地收敛. 在神经网络这样的非凸优化中, 存在着多个局部最优点, GD 找到的局部最优点取决于参数初始化时所在的位置. 而在 SGD 中, 由于在更新过程中存在噪声, 噪声可能使参数跳到另一个比当前局部最优更好的局部最优. SGD 也可以跟踪数据分布的变化 [21].

在实际中, GD 和 SGD 都较少用到, GD 每更新一个需要看到 m 个训练数据, SGD 每更新一次需要看到 1 个数据. 小批量梯度下降 *(mini-batch gradient descent)* 介于两者之间, 每更新一次需要看到 m' 个数据, m' 被称为批量 *(batches)*. 小批量梯度下降继承了 SGD 的优点, 同时可以利用高效的向量化代码优化的技巧, 因此在涉及大数据优化时经常被用到. 由于批量大小是一个在 1 和 m 之间的人为指定超参数, 因此在下文的计算中, m 即可以代表训练数据大小, 也可以代表批量大小.

3.4.3 softmax 优化规则

利用微积分, 很容易计算对于交叉熵损失对参数的导数.

$$\vec{s}^{(i)} = W\vec{x}^{(i)} + \vec{b}, \forall i\,, \tag{3.41}$$

$$\begin{aligned} \mathrm{err}^{(i)} &= -\log \frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{j=0}^{K-1}\exp(s^{(i)}_j)} && (3.42)\\ &= -s^{(i)}_{y^{(i)}} + \log(\sum_{j=0}^{K-1}\exp(s^{(i)}_j))\,. && (3.43) \end{aligned}$$

因此

$$\begin{aligned} \frac{\partial \mathrm{err}^{(i)}}{\partial s^{(i)}_j} &= \frac{\partial}{\partial s^{(i)}_j}\log(\sum_{j=0}^{K-1}\exp(s^{(i)}_j)), j \neq y^{(i)} && (3.44)\\ &= \frac{\exp(s^{(i)}_j)}{\sum_{k=0}^{K-1}\exp(s^{(i)}_k)}\,, && (3.45)\\ \frac{\partial \mathrm{err}^{(i)}}{\partial s^{(i)}_{y^{(i)}}} &= \frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{k=0}^{K-1}\exp(s^{(i)}_k)} - 1\,. && (3.46) \end{aligned}$$

将上面两式合并写成向量形式

$$\nabla_{\vec{s}^{(i)}}\mathrm{err}^{(i)} = \frac{\exp(\vec{s}^{(i)})}{\sum_{k=0}^{K-1}\exp(s^{(i)}_k)} - \vec{e}_{y^{(i)}}\,. \tag{3.47}$$

其中 $\vec{e}_{y^{(i)}}$ 是个 K 维向量, 在 $y^{(i)}$ 的位置为 1, 其余位置为 0.

利用链式求导法则可得

$$\nabla_W \mathrm{err}^{(i)} = (\nabla_{\vec{s}^{(i)}} \mathrm{err}^{(i)}) \vec{x}^{(i)T}, \tag{3.48}$$

$$\nabla_{\vec{b}} \mathrm{err}^{(i)} = \nabla_{\vec{s}^{(i)}} \mathrm{err}^{(i)}. \tag{3.49}$$

之后利用式3.37, 3.38就可以对 $W, \vec{b}$ 进行更新.

总之, softmax 的完整训练过程可用算法1表示.

Algorithm 1 softmax 训练算法

init. $W \sim N(0, 0.01^2), \vec{b} = \vec{0}$ // W 初始值随机从高斯分布中采样, $\vec{b}$ 初始定为 0.

for $t = 1$ **to** T // 共更新参数 T 次.

sample a batch of data $\{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$

for $i = 1$ **to** m

$\vec{s}^{(i)} = h(\vec{x}^{(i)}) = W\vec{x}^{(i)} + \vec{b}$ // 计算假设函数.

$\mathrm{err}^{(i)} = -\log \frac{\exp(s_{y^{(i)}}^{(i)})}{\sum_{j=0}^{K-1} \exp(s_j^{(i)})}$ // 计算交叉熵损失

$\nabla_{\vec{s}^{(i)}} \mathrm{err}^{(i)} = \frac{\exp(\vec{s}^{(i)})}{\sum_{k=0}^{K-1} \exp(s_k^{(i)})} - \vec{e}_{y^{(i)}}$

$\nabla_W \mathrm{err}^{(i)} = (\nabla_{\vec{s}^{(i)}} \mathrm{err}^{(i)}) \vec{x}^{(i)T}$ // 计算 $\mathrm{err}^{(i)}$ 对参数的梯度.

$\nabla_{\vec{b}} \mathrm{err}^{(i)} = \nabla_{\vec{s}^{(i)}} \mathrm{err}^{(i)})$

$J = \frac{1}{m} \sum_{i=1}^{m} \mathrm{err}^{(i)} + \frac{\lambda}{2} \|W\|_F^2$ // 计算损失函数.

$\nabla_W J = \frac{1}{m} \sum_{i=1}^{m} \nabla_W \mathrm{err}^{(i)} + \lambda W$

$\nabla_{\vec{b}} J = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\vec{b}} \mathrm{err}^{(i)}$

$W \leftarrow W - \alpha \nabla_W J$ // 参数更新.

$\vec{b} \leftarrow \vec{b} - \alpha \nabla_{\vec{b}} J$

3.5 预测和评估

让分类模型在一组从未在训练集中出现过的图像组成的测试集上做预测, 通过比较预测的标记和测试集真实的标记来评价这个模型的好坏. 我们希望一个好的分类模型会有很多的预测标记和真实标记相一致.

3.5.1 预测

对于一个未知输入数据 $\vec{x}$, 分类模型的预测是

$$\hat{y} = \arg\max_k h(\vec{x})_k = \arg\max_k s_k \,. \tag{3.50}$$

即找到线性分类器输出分数最大的那项对应的类作为 $\vec{x}$ 的标记.

3.5.2 评估

给定测试集

$$D_{test} = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{m_{test}} \,, \tag{3.51}$$

对每个 $\vec{x}^{(i)}$, 根据式3.50计算模型的预测值 $\hat{y}^{(i)}$. 用准确率

$$Acc = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} 1\{\hat{y}^{(i)} = y^{(i)}\} \tag{3.52}$$

表示分类器的性能. 其中 $1\{\cdot\}$ 是指示函数,

$$1\{\texttt{true statement}\} = 1 \,, \tag{3.53}$$

$$1\{\texttt{false statement}\} = 0 \,. \tag{3.54}$$

若分类器将全部测试样本分对, 则 $Acc = 1$, 其余情况下 $Acc < 1$.

Chapter 4

前馈神经网络

多层前馈神经网络 (multilayer feedforward nueral network), 也称为*多层感知器 (multilayer perceptrons, MLP)*, 它利用上一章讨论的线性分类器处理非线性问题. 本章讨论的神经网络就是这种多层前馈神经网络. 在本章, 将讨论为什么要使用神经网络及深度学习, 神经网络的基本架构和学习方法. 本章也是下一章深度学习的铺垫.

4.1 特征/表示学习

4.1.1 线性模型

上一章讨论的 softmax 分类器代表了一系列的线性模型. 它们的训练是凸优化问题, 最终有理论保证能收敛到全局最优, 和参数的初始位置无关. 但是线性分类器只能用多个超平面把输入空间划分为几个简单的区域. 但是对很多问题, 直接把图像原始像素拿去做训练, 线性模型不能将它们分开, 即原始像素不是线性可分的.

4.1.2 特征工程

针对线性模型要求数据线性可分, *特征工程 (feature engineering)* 设计一种最适合当前任务的数据的*表示 (representation)*$\phi(\vec{x})$ 作为输入的特征而不是直接使用原始像素 $\vec{x}$, 使得 $\phi(\vec{x})$ 线性可分. 比如图4.3, 左图是原始的数据 $\vec{x}$, 包括两个类 (红色和蓝色), 每个类的数据各自分布在一个正弦曲线上.

这两类数据不是线性可分的, 线性模型无法将这两类分开. 如果设法设计一种表示 $\phi(\vec{x})$, 使得数据的分布变成右图那样, 数据将线性可分, 线性模型将能够把这两类分开. SIFT 和 HOG 特征都是特征工程的两个例子.

SIFT[23] 让图像和不同尺度下的高斯滤波器进行卷积, 从得到的图像的差异 (高斯差, *DoG*) 中找到兴趣点 (一般是 DoG 之后的极大, 极小值点), 以关键点相邻梯度方向分布作为指定方向参数, 使关键点描述拥有旋转不变性. SIFT 寻找可以用来识别目标的局部图像特征, 因此可以用来解决图像中目标有遮挡的情况, 见图4.1.

图 4.1: SIFT 可用于有遮挡情况下的目标识别. 图像来源于 [23].

HOG[2] 将图像分成小的连通区域, 计算各区域内各像素点的梯度或边缘的方向直方图, 之后把这些直方图组合起来构成图像特征, 见图4.2. 和 SIFT 相似, 两者都利用了图像的局部信息来做识别.

$\phi(\vec{x})$ 的设计需要领域的先验知识, 而且和需要解决的任务密切相关. 模型的的性能依赖于数据的表示, 很多情况下, 我们很难设计合适的能应对多种图像识别问题挑战的数据特征, 因此需要大量的工作不断设计更好的特征.

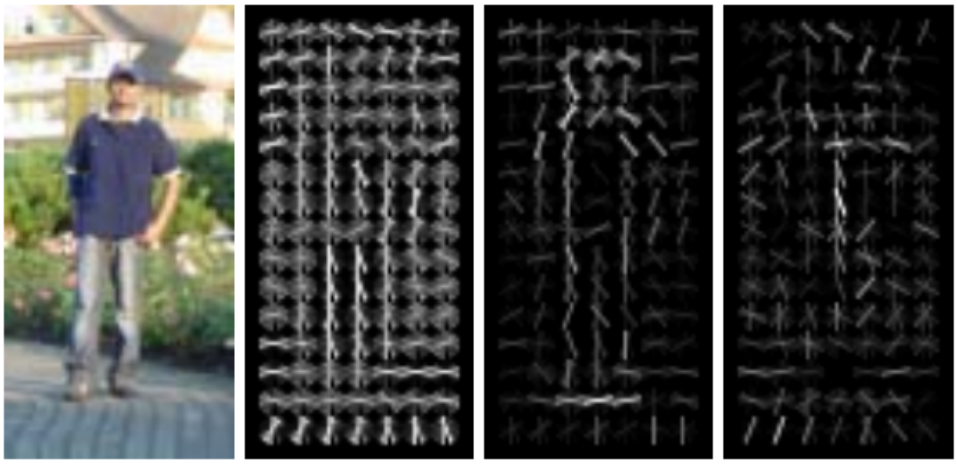


图 4.2: HOG 特征. 图像来源于 [2].

图 4.3: 通过某种数据的表示 ϕ 使得 $\phi(\vec{x})$ 之间线性可分. 图像来源于 [19].

4.1.3 核方法

在核方法中使用固定的 ϕ 将 $\vec{x}$ 映射到一个高维空间, 同时使优化问题仍然保持是凸优化问题. 映射 $\phi(\vec{x})$ 使模型的 *VC 维 (VC dimension)* 提高, 会有潜在的过拟合 *(overfitting)* 风险. 对 ϕ(核) 的选取是一种先验知识, 称为核工程 *(kernel engineering)*. 高斯核 *(Gaussian kernel)*$k(\vec{x}, \vec{x}') = \exp(-\gamma\|\vec{x} - \vec{x}'\|^2)$ 有十分宽的先验分布, 并且很平滑, 是一种广泛使用的核.

4.1.4 表示学习

机器学习算法是从数据中学习 *(learning from data)*. 我们能不能利用机器学习算法不仅学习从数据的表示到输出的映射, 也学习到合适的数据的表

示呢? 如果学到的表示可以不受光照, 视角等的影响, 反应了数据的本质特征, 这样学到的特征会比人为设计的特征性能更好, 而且这会使 AI 系统有更好的普适性. 从数据中学到合适表示的过程称为*表示学习 (representation learning)*. *自编码器 (autoencoder)*[1] 就是表示学习的一个例子, 它将输入转化为一个不同的表示.

4.1.5 深度学习

直接从输入学习到如何提取合适的, 能应对多种图像识别问题挑战的数据的表示 ϕ 仍然是很难的. *深度学习 (deep learning)* 将数据的表示分级, 高级的表示建立在低级的表示上, 机器将从数据简单的表示中学习复杂的表示.

比如在图4.4中, 直接从原始图像像素中学习到合适的表示是很难的. 深度学习把这样一个复杂的问题分成一系列嵌套的简单的表示学习问题

$$\phi(\vec{x}) = \phi_3(\phi_2(\phi_1(\vec{x}))) \tag{4.1}$$

每个小的表示映射可由神经网络中的一层建模:

- 第一层 ϕ_1: 从图像的像素和邻近像素的像素值中识别边缘.
- 第二层 ϕ_2: 将边缘整合起来识别轮廓和角点.
- 第三层 ϕ_3: 提取特定的轮廓和角点作为输入的特征.
- 最后通过一个线性分类器识别图像中的目标.

这样的一种解决问题的策略称为*连接主义 (connectionism)*. 虽然每一层都是相对简单的运算, 但是多层结合起来会展示出模型强大的力量.

有关经典机器学习, 表示学习, 和深度学习的对比, 见图4.5, 图中深色的部分代表模型能从数据中学习的部分. 深度学习的具体内容见下一章, 本章剩余部分将介绍深度学习的基础—前馈神经网络.

4.2 假设函数

4.2.1 数据表示

神经网络中, 上一章讨论的数据驱动过程的各个流程没有改变, 只是在线性分类器中使用的假设函数

$$h(\vec{x}) = W\vec{x} + \vec{b} \tag{4.2}$$

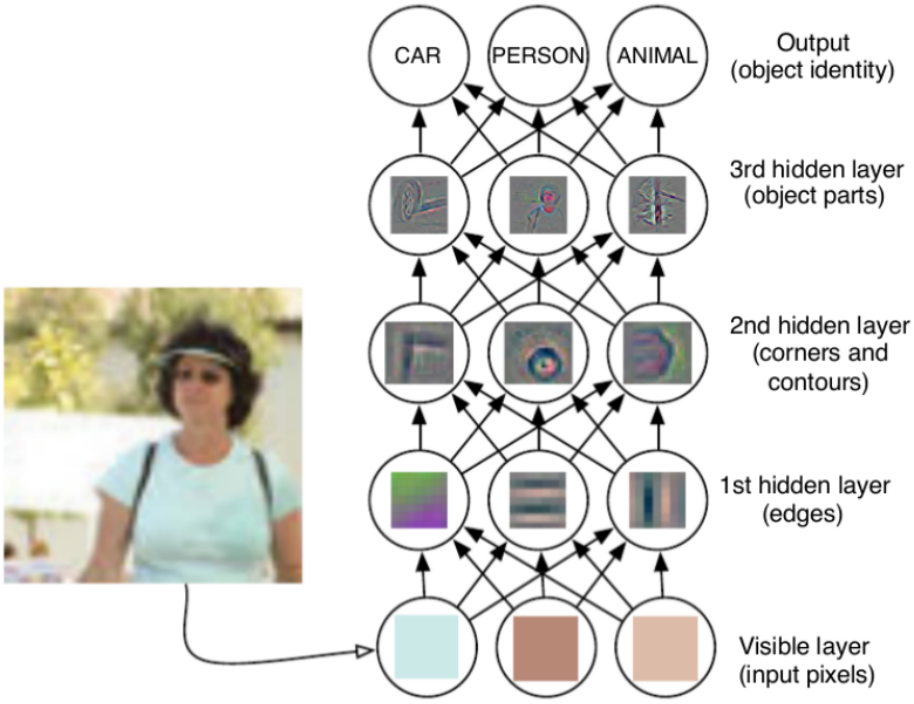


图 4.4: 一个深度学习模型. 图像来源于 [9].

里原始图像像素输入 $\vec{x}$ 变成了可学习的数据的表示 $\phi(\vec{x})$.

$$h(\vec{x}) = W\phi(\vec{x}) + \vec{b}. \tag{4.3}$$

表示 $\phi(\vec{x})$ 又由 $L-1$ 个嵌套的相对简单的表示组成

$$\phi(\vec{x}) = \phi_{L-1}(\phi_{L-2}(\dots\phi_2(\phi_1(\vec{x})))) . \tag{4.4}$$

每个表示 $\phi_l(\vec{a})$ 是由线性运算和非线性运算组成. 最简单的线性运算是*仿射 (affine)* 运算

$$\vec{z} = W^{(l)}\vec{a} + \vec{b}^{(l)}, \forall l = 1, 2, \dots, L-1 . \tag{4.5}$$

其中 $W^{(l)}, \vec{b}^{(l)}$ 是可以从数据中学习到的参数. 非线性运算又称为神经网络中的*激活函数 (activation function)*

$$\phi_l(\vec{a}) = \max(0, \vec{z}), \forall l = 1, 2, \dots, L-1 . \tag{4.6}$$

其中

$$\max(0, s) = 1\{s > 0\}s \tag{4.7}$$

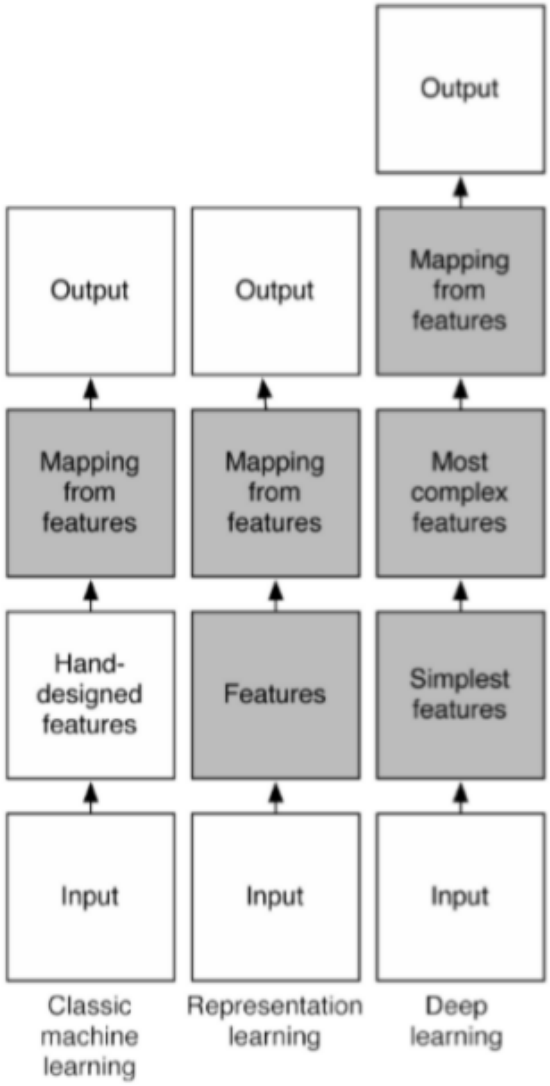


图 4.5: 经典机器学习, 表示学习, 和深度学习的对比. 图像来源于 [9].

称为线性整流层单元 *(rectified linear units, ReLU, relu)*. 它将 s 中小于 0 的部分置为 0, 见图4.6. 它将逐元素的应用于 $\vec{z}$. 激活函数的作用至关重要. 如果没有激活函数, 多个线性运算的嵌套还是一个线性运算, 因此 $h(\vec{x})$ 仍然是 $\vec{x}$ 的一个线性运算. 除了 relu 之外, 还有其他的激活函数, 比如 sigmoid

函数,

$$\sigma(s) = \frac{1}{1+\exp(-s)}\,, \tag{4.8}$$

见图4.7. 但是 sigmoid 函数计算比 relu 复杂, 性能也不如 relu[18], 所以现在除了在 LSTM[12] 中, 已经较少为人使用.

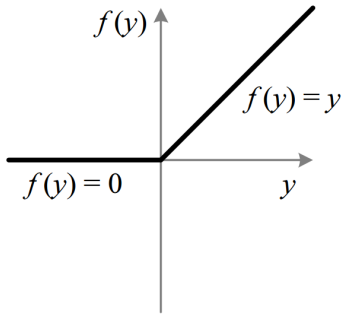


图 4.6: ReLU. 图像来源于 [11].

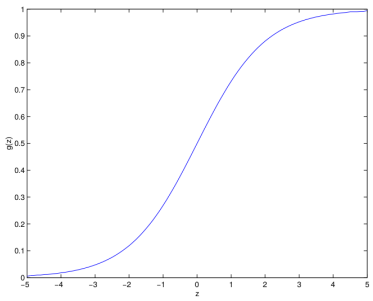


图 4.7: Sigmoid 函数. 图像来源于 [25].

如果令

$$\vec{a}^{(0)} = \vec{x}\,. \tag{4.9}$$

可将上面的式子用递归的形式简化表示为

$$\vec{a}^{(l)} = \max(0, W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)}), \forall l = 1, 2, \ldots, L-1\,, \tag{4.10}$$

$$h(\vec{x}) = W^{(L)}\vec{a}^{(L-1)} + \vec{b}^{(L)}\,. \tag{4.11}$$

其中 $W^{(L)}, \vec{b}^{(L)}$ 是式4.3中的 $W, \vec{b}$. $\vec{a}^{(L-1)}$ 即是数据的表示 $\phi(\vec{x})$. 通过这两式, 即可计算神经网络中的假设函数.

4.2.2 人工神经元模型

神经网络模型的设计最初是受到生物体内神经元和神经元之间突触连接形式的启发, 见图4.8的左图. 每个神经元 *(neuron)* 通过树突 *(dendrites)* 从别的神经元接受信号, 在细胞核 *(nucleus)* 内处理, 再由轴突 *(axon)* 将信号传输出去. 轴突在伸出后分叉, 每个又与其他的神经元的树突通过突触 *(synapses)* 相连接.

图4.8的右图是对生物神经元的一种数学上抽象出来的模型. 神经元通过树突接收到来自其他神经元传来的信号, 这些输入信号通过带权重的连接进行传递, 在细胞核内进行汇总, 然后通过激活函数处理以产生神经元的输出. 如果输入是 $\vec{x}$, 权重是 $\vec{w}$, 则神经元完成的计算是

$$z = \vec{w}^T\vec{x} + b = \sum_{j=0}^{n-1} w_j x_j + b\,, \tag{4.12}$$

$$a = \max(0, z) = \max(0, \vec{w}^T\vec{x} + b)\,. \tag{4.13}$$

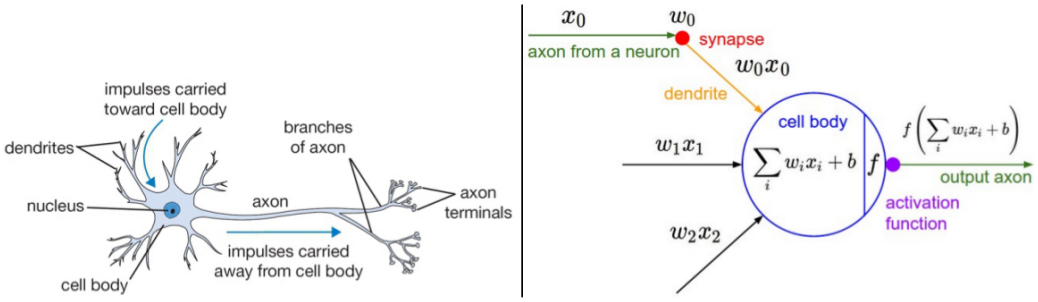


图 4.8: 生物神经元及相应的数学抽象模型. 图像来源于 [7].

如果将 n' 个神经元堆叠起来, 每个神经元接收同样的输入, 但是每个神经元有各自权重, 神经元之间没有连接, 那么每个神经元的输出

$$a_k = \max(0, z_k) = \max(0, \vec{w}_d^T\vec{x}) + b_d, \forall d = 0, 1, 2, \ldots, n' - 1\,. \tag{4.14}$$

用向量形式写出来是

$$\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n'-1} \end{bmatrix} \tag{4.15}$$

$$= \begin{bmatrix} \max(0, z_0) \\ \max(0, z_1) \\ \vdots \\ \max(0, z_{n'-1}) \end{bmatrix} \tag{4.16}$$

$$= \begin{bmatrix} \max(0, \vec{w}_0^T \vec{x} + b_0) \\ \max(0, \vec{w}_1^T \vec{x} + b_1) \\ \vdots \\ \max(0, \vec{w}_{n'-1}^T \vec{x} + b_{n'-1}) \end{bmatrix} \tag{4.17}$$

$$= \max(0, \begin{bmatrix} \vec{w}_0^T \\ \vec{w}_1^T \\ \vdots \\ \vec{w}_{n'-1}^T \end{bmatrix} \vec{x} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n'-1} \end{bmatrix}) \tag{4.18}$$

$$= \max(0, W\vec{x} + \vec{b}) . \tag{4.19}$$

4.2.3 神经网络架构

神经网络的假设函数是

$$\vec{a}^{(l)} = \max(0, W^{(l)} \vec{a}^{(l-1)} + \vec{b}^{(l)}), \forall l = 1, 2, \ldots, L-1 , \tag{4.20}$$

$$h(\vec{x}) = W^{(L)} \vec{a}^{(L-1)} + \vec{b}^{(L)} . \tag{4.21}$$

这是一种 L 层的递归形式. 如果每一层都用多个神经元组成, 第 l 层神经元的输出 $(\vec{a}^{(l)})$ 作为第 $l+1$ 层神经元的输入, 神经元之间没有同层连接, 相邻层的神经元之间采用全连接, 不存在跨层连接, 这叫做一层全连接层 *(fully-connected layer, fc)*. 这样, 神经网络的假设函数可以用 L 层的全连接层建模, 见图4.9.

注意我们讲一个 L 层神经网络的时候, 是不算输入层的, 因此图4.9是 3 层神经网络. 根据式4.21, 输出层 (第 L 层) 没有激活函数.

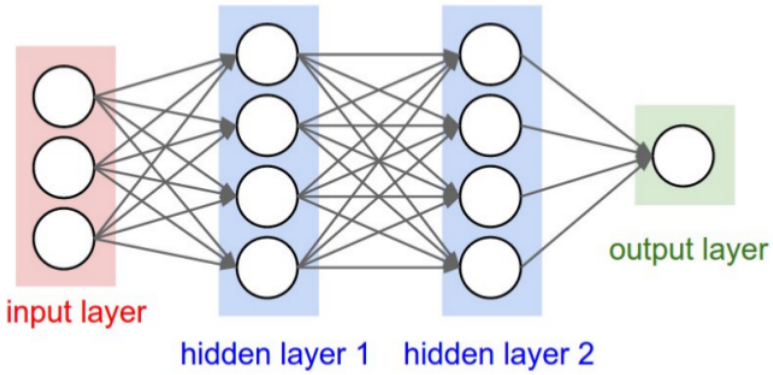


图 4.9: 多层前馈神经网络. 图像来源于 [7].

4.3 损失函数

神经网络对应的交叉熵损失函数为

$$J(W,\vec{b}) = \frac{1}{m}\sum_{i=1}^{m} -\log\frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{j=0}^{K-1}\exp(s^{(i)}_j)} + \sum_{l=1}^{L}\frac{\lambda}{2}\|W^{(l)}\|_F^2 \tag{4.22}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\mathrm{err}^{(i)} + \sum_{l=1}^{L}\frac{\lambda}{2}\|W^{(l)}\|_F^2\,. \tag{4.23}$$

其中

$$\vec{s}^{(i)} = h(\vec{x}^{(i)})\,. \tag{4.24}$$

$h(\vec{x}^{(i)})$ 是利用式4.21计算得到的.

4.4 优化

4.4.1 梯度下降

和在线性分类器中一样, 优化采用的是梯度下降, 参数更新规则是

$$W^{(l)} \leftarrow W^{(l)} - \alpha\nabla_{W^{(l)}}J = W^{(l)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\nabla_{W^{(l)}}\mathrm{err}^{(i)} - \alpha\lambda W^{(l)}, \forall l\,, \tag{4.25}$$

$$\vec{b}^{(l)} \leftarrow \vec{b}^{(l)} - \alpha\nabla_{\vec{b}^{(l)}}J = \vec{b}^{(l)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)}, \forall l\,. \tag{4.26}$$

因此, 关键是计算 $\nabla_{W^{(l)}}\mathrm{err}^{(i)}$ 和 $\nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)}$.

注意到

$$\nabla_{\vec{s}^{(i)}}\mathrm{err}^{(i)} = \frac{\exp(\vec{s}^{(i)})}{\sum_{k=0}^{K-1}\exp(s_k^{(i)})} - \vec{e}_{y^{(i)}}\,, \tag{4.27}$$

$$\vec{s}^{(i)} = h(\vec{x}^{(i)}) = W^{(L)}\vec{a}^{(L-1)} + \vec{b}^{(L)}\,. \tag{4.28}$$

因此 $\nabla_{W^{(L)}}\mathrm{err}^{(i)}$ 和 $\nabla_{\vec{b}^{(L)}}\mathrm{err}^{(i)}$ 很容易计算.

$$\nabla_{W^{(L)}}\mathrm{err}^{(i)} = (\nabla_{\vec{s}^{(i)}}\mathrm{err}^{(i)})\vec{x}^{(i)T}\,, \tag{4.29}$$

$$\nabla_{\vec{b}^{(L)}}\mathrm{err}^{(i)} = \nabla_{\vec{s}^{(i)}}\mathrm{err}^{(i)}\,. \tag{4.30}$$

但是 $\nabla_{W^{(l)}}\mathrm{err}^{(i)}$ 和 $\nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)}$, $l < L$ 呢? 想要显式的写出是 $\nabla_{W^{(l)}}\mathrm{err}^{(i)}$ 和 $\nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)}$ 的表达式是非常复杂和困难的.

4.4.2 误差反向传播

在第 l 层, 完成的运算是

$$\vec{z}^{(l)} = W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)}\,, \tag{4.31}$$

$$\vec{a}^{(l)} = \max(0, \vec{z}^{(l)})\,. \tag{4.32}$$

假设 $\nabla_{\vec{a}^{(l)}}\mathrm{err}^{(i)}$ 已知, 通过链式求导法则可以计算

$$\nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)} = (\frac{\partial\vec{a}^{(l)}}{\partial\vec{z}^{(l)}})^T\nabla_{\vec{a}^{(l)}}\mathrm{err}^{(i)} \tag{4.33}$$

$$= 1\{\vec{z}^{(l)} > 0\} \odot \nabla_{\vec{a}^{(l)}}\mathrm{err}^{(i)}\,. \tag{4.34}$$

其中 $\odot$ 代表的是逐元素相乘, 即向量的 *Hadamard* 积 *(Hadamard product)*. 同样的,

$$\nabla_{\vec{a}^{(l-1)}}\mathrm{err}^{(i)} = (\frac{\partial\vec{z}^{(l)}}{\partial\vec{a}^{(l-1)}})^T\nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)} \tag{4.35}$$

$$= W^T\nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)}\,, \tag{4.36}$$

$$\nabla_{W^{(l)}}\mathrm{err}^{(i)} = (\nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)})\vec{a}^{(l-1)T}\,, \tag{4.37}$$

$$\nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)} = \nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)}\,. \tag{4.38}$$

只要知道了 $\nabla_{\vec{a}^{(l)}}\mathrm{err}^{(i)}$, 就可以计算 $\nabla_{W^{(l)}}\mathrm{err}^{(i)}, \nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)}, \nabla_{\vec{a}^{(l-1)}}\mathrm{err}^{(i)}$. $\nabla_{W^{(l)}}\mathrm{err}^{(i)}$ 和 $\nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)}$ 用于更新第 l 层的参数. $\nabla_{\vec{a}^{(l-1)}}\mathrm{err}^{(i)}$ 传向第 $l-1$ 层, 用于进

一步计算第 $l-1$ 层的参数的导数. 因此, 从第 $l=L$ 层开始, 应用上面的公式, 直到第 1 层, 就可以计算出所有参数的导数. 这样的算法称为*误差反向传播算法 (error back-propagation, BP)*, 这本质上是微积分中链式求导法则的递归形式的应用.

总之, 神经网络的完整训练过程可用算法2表示.

4.5 预测和评估

神经网络中的预测和评估方法和线性分类器中的相同, 只不过 $\vec{s}^{(i)} = h(\vec{x}^{(i)})$ 是用式4.21计算得到的.

Algorithm 2 神经网络训练算法

for $l = 1$ **to** L // W 初始值随机从高斯分布中采样, $\vec{b}$ 初始定为 0.
  init. $W^{(l)} \sim N(0, 0.01^2), \vec{b}^{(l)} = \vec{0}$
for $t = 1$ **to** T // 共更新参数 T 次.
  sample a batch of data $\{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$
  for $i = 1$ **to** m
    $\vec{a}^{(0)} = \vec{x}^{(i)}$
    for $l = 1$ **to** $L-1$ // 前向传播计算假设函数.
      $\vec{z}^{(l)} = W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)}$
      $\vec{a}^{(l)} = \max(0, \vec{z}^{(l)})$
    $\vec{s}^{(i)} = h(\vec{x}^{(i)}) = W^{(L)}\vec{a}^{(L-1)} + \vec{b}^{(L)}$ // 计算假设函数.
    $\mathrm{err}^{(i)} = -\log \frac{\exp(s^{(i)}_{y^{(i)}})}{\sum_{k=0}^{K-1}\exp(s^{(i)}_k)}$ // 计算交叉熵损失
    $\nabla_{\vec{s}^{(i)}}\mathrm{err}^{(i)} = \frac{\exp(\vec{s}^{(i)})}{\sum_{k=0}^{K-1}\exp(s^{(i)}_k)} - \vec{e}_{y^{(i)}}$
    $\nabla_{W^{(L)}}\mathrm{err}^{(i)} = (\nabla_{\vec{s}^{(i)}}\mathrm{err}^{(i)})\vec{x}^{(i)T}$
    $\nabla_{\vec{b}^{(L)}}\mathrm{err}^{(i)} = \nabla_{\vec{s}^{(i)}}\mathrm{err}^{(i)}$
    $\nabla_{\vec{a}^{(L-1)}} = W^{(L)T}\nabla_{\vec{s}^{(i)}}\mathrm{err}^{(i)}$
    for $l = L-1$ **to** 1 // 反向传播计算 $\mathrm{err}^{(i)}$ 对参数的导数.
      $\nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)} = 1\{\vec{z}^{(l)} > 0\} \odot \nabla_{\vec{a}^{(l)}}\mathrm{err}^{(i)}$
      $\nabla_{W^{(l)}}\mathrm{err}^{(i)} = (\nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)})\vec{a}^{(l-1)T}$
      $\nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)} = \nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)}$
      $\nabla_{\vec{a}^{(l-1)}}\mathrm{err}^{(i)} = W^T\nabla_{\vec{z}^{(l)}}\mathrm{err}^{(i)}$
  $J = \frac{1}{m}\sum_{i=1}^{m}\mathrm{err}^{(i)} + \sum_{l=1}^{L}\frac{\lambda}{2}\|W^{(l)}\|_F^2$ // 计算损失函数.
  for $l = 1$ **to** L
    $\nabla_{W^{(l)}}J = \frac{1}{m}\sum_{i=1}^{m}\nabla_{W^{(l)}}\mathrm{err}^{(i)} + \lambda W^{(l)}$
    $\nabla_{\vec{b}^{(l)}}J = \frac{1}{m}\sum_{i=1}^{m}\nabla_{\vec{b}^{(l)}}\mathrm{err}^{(i)}$
    $W^{(l)} \leftarrow W^{(l)} - \alpha\nabla_{W^{(l)}}J$ // 参数更新.
    $\vec{b}^{(l)} \leftarrow \vec{b}^{(l)} - \alpha\nabla_{\vec{b}^{(l)}}J$

Chapter 5

卷积神经网络

卷积神经网络 (convolutional neural networks, CNN, ConvNets) 是流行的深度学习技术中的一种. 和上一章讨论的神经网络一样, CNN 也是由可学习的参数组成, 每一层也是进行一个线性运算和经过一个激活函数, 参数的学习也是根据 BP 算法. CNN 和神经网络的区别, CNN 的优点, 和 CNN 参数的具体学习过程将在本章进行讨论.

5.1 训练数据

CNN 要求输入是图像, 这样可以让我们依据图像的性质对 CNN 的结构进行设计, 使得 CNN 相比一般的神经网络结构更加高效性能更好. [1]

因此, 和线性分类器和神经网络的训练数据使用特征向量不同, CNN 的训练数据将由代表图像的张量 *(tensor)* 组成.

$$D = \{(\mathbf{X}^{(i)}, y^{(i)})\}_{i=1}^{m}, \tag{5.1}$$

$$\mathbf{X}^{(i)} \in \mathbb{R}^{H \times W \times D}, \forall i, \tag{5.2}$$

$$y^{(i)} \in \{1, 2, 3, \ldots, K\}, \forall i. \tag{5.3}$$

其中, $\mathbf{X}^{(i)}$ 是第 i 张图像, $H \times W \times D$ 是每个图像的尺寸. 注意这里有些滥用记号: 用 W 即表示可学习的参数, 也表示图像的宽; D 即表示整个训练集, 也表示图像的深度.

[1] 严格的说, CNN 也适用于输入不是图像的情况. 比如在自然语言处理 *(natural language processing, nlp)* 中, CNN 也可以用于对句子的建模 [13].

5.2 假设函数

由于 CNN 的输入变成了张量, 因此神经网络中每一层的神经元将不再像神经网络中按一个维度排列成向量, 而是沿着三个维度 (高度, 宽度, 深度) 排列成张量. 最后一层 (第 L 层) 的神经元输出维数是 $1 \times 1 \times K$, 这将和普通神经网络一样, 是一个表示每个类分数的向量, 见图5.1.

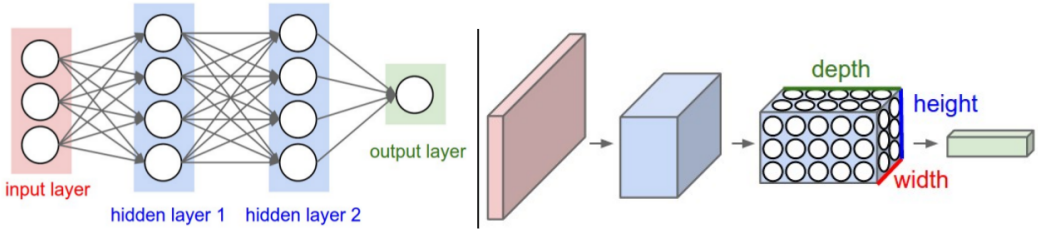


图 5.1: 普通神经网络 (左) 和 CNN(右) 的区别. 图像来源于 [7].

和普通神经网络每层都是一个全连接层 (仿射运算 +relu 非线性激活函数) 不同, CNN 在每层有 4 种选择: *卷积层 (convolutional layer, conv)*, *汇合层*[2]*(pooling layer, pool)*, 线性整流层 (relu), 和全连接层 (fc). 下面将分别分析每层的结构.

5.2.1 卷积层 (conv)

全连接仿射的张量扩展

上一章讨论的神经网络中, 每个神经元和上一层的所有神经元保持全连接. 假设在第 l 层, 输入是 $\vec{x} \in \mathbb{R}^{n_{l-1}}$. [3] $\vec{x}$ 经过一个仿射运算:

$$\vec{a} = W\vec{x} + \vec{b} \tag{5.4}$$

$$= \begin{bmatrix} \vec{w}_0^T \\ \vec{w}_1^T \\ \vdots \\ \vec{w}_{n_l-1}^T \end{bmatrix} \vec{x} + \vec{b} \tag{5.5}$$

[2] 也有其他文献将其翻译为池化层.

[3] 这里为了简化记号, 用 $\vec{x}$ 代表上一章使用的符号 $\vec{a}^{(l-1)}$.

$$= \begin{bmatrix} \vec{w}_0^T \vec{x} + b_0 \\ \vec{w}_1^T \vec{x} + b_1 \\ \vdots \\ \vec{w}_{n_l-1}^T \vec{x} + b_{n_l-1} \end{bmatrix} . \tag{5.6}$$

这可以认为是同时有 n_l 个 $\vec{w}_{d_l}, b_{d_l}$ 作用于 $\vec{x}$, 每个完成计算

$$a_{d_l} = \vec{w}_{d_l}^T \vec{x} + b_{d_l} = \vec{w}_{d_l} \odot \vec{x} + b_{d_l}, \forall d_l \, . \tag{5.7}$$

$\vec{w}_d$ 和 $\vec{x}$ 的维度保持一致.

当输入由一维向量 $\vec{x} \in \mathbb{R}^{n_{l-1}}$ 变成三维张量 $\mathbf{X} \in \mathbb{R}^{H_{l-1} \times W_{l-1} \times D_{l-1}}$ 时, 若保持全连接性质, 则权值也要变为三维张量 $\mathbf{W}_d \in \mathbb{R}^{H_{l-1} \times W_{l-1} \times D_{l-1}}$.

$$a_{d_l} = \mathbf{W}_{d_l} \odot \mathbf{X} + b_{d_l} = \sum_{i=0}^{H_{l-1}-1} \sum_{j=0}^{W_{l-1}-1} \sum_{d=0}^{D_{l-1}-1} \mathbf{X}(i,j,d) \mathbf{W}_{d_l}(i,j,d) + b_{d_l}, \forall d_l \, . \tag{5.8}$$

在这里及后文, 为了写法上简单一些, 使用 $\mathbf{X}(i,j,d)$ 表示 $\mathbf{X}$ 中位于位置 (i,j,d) 的元素.

全连接结构对图像的可扩展性并不好. 在 CIFAR-10[17] 数据集中, 图像的大小是 $32 \times 32 \times 3$. 因此在第一层中, 每个神经元有 $32 \times 32 \times 3 = 3072$ 个权值, 这个参数量还可以接受. 但是对于一个相对合理的图片大小, 比如 $200 \times 200 \times 3$, 每个神经元将有 $200 \times 200 \times 3 = 120,000$ 个权值. 大量的参数需要繁重的计算, 但更重要的是, 容易导致过拟合. 为了解决这个问题, 卷积层做了两个简化.

卷积层的两个简化

(1). 稀疏连接. 在全连接中, 每个输出 a_{d_l} 通过 $\mathbf{W}_{d_l}(i,j,d)$ 和每个输入神经元 $\mathbf{X}(i,j,d)$ 相连. 而在图像识别中, 关键性的图像特征, 边缘, 角点等只占据了整个图像的一小部分, 图像中相距很远的两个像素之间有相互影响的可能性很小. 因此, 每个神经元 a_{d_l} 只需要和一小部分输入神经元相连, 图5.2是一个一维的例子.

局部连接的空间范围 $F_1 \times F_2$ 称为*感受野 (receptive field)*, 而沿深度轴的连接数总是等于输入的深度. 也就是说, 沿高和宽轴是局部连接的, 沿深度轴是全连接的, 图5.3是一个三维的例子.

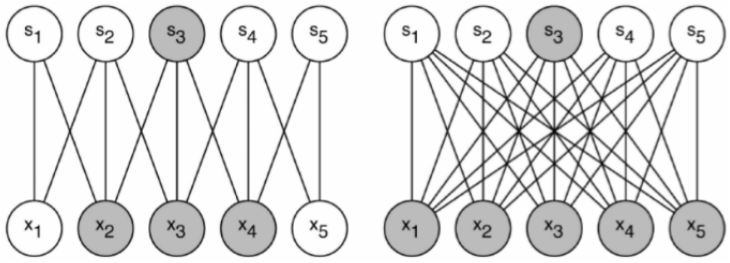


图 5.2: 一维的稀疏连接. 右图中每个输出神经元和所有的输入神经元相连接, 左图中限制了连接数目. 图像来源于 [9].

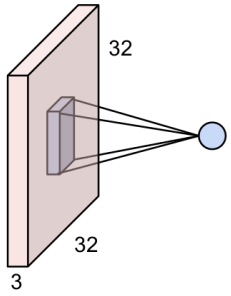


图 5.3: 三维的稀疏连接. 输入高和宽是 32, 深度是 3. 图像来源于 [7].

随着这个局部连接区域在输入张量上空间位置的变化 (沿着高轴和宽轴), 将得到一个二维的输出神经元排列, 称为*激活映射 (activation map)*, 见图5.4.

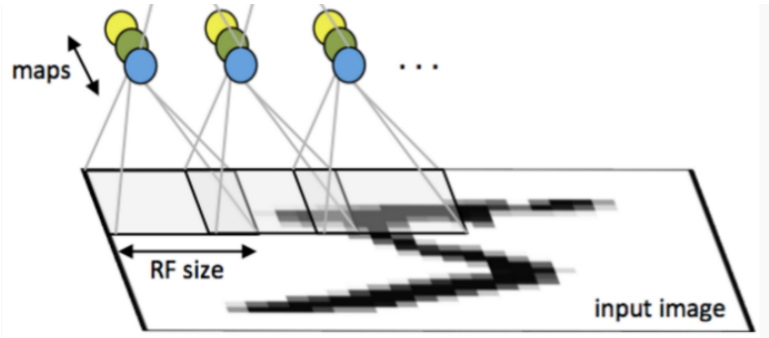


图 5.4: 激活映射. 图像来源于 [6].

每个 $\mathbf{W}_{d_l}$ 维度将变成 $F_1 \times F_2 \times D_{l-1}$, $\mathbf{W}_{d_l}$ 作用于输入神经元, 得到一个激活映射, D_l 个滤波器将得到 D_l 个激活映射. 将这 D_l 个激活映射沿着深度方向排列起来, 将得到一个输出神经元张量 $\mathbf{A} \in \mathbb{R}^{H_l \times W_l \times D_l}$, 见图5.5. 将这 D_l 个滤波器 $\mathbf{W}_{d_l}$ 沿着第四维排列起来, 将得到一个四维的参数矩阵 $\mathbf{W} \in \mathbb{R}^{H_{F1} \times W_{F2} \times D_{l-1} \times D_l}$.

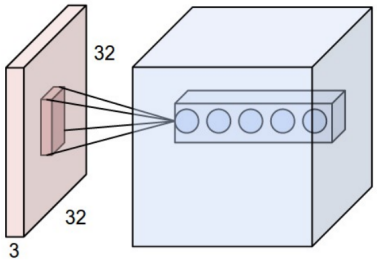


图 5.5: 输入神经元张量和输出神经元张量. 图像来源于 [7].

(2). 共享参数. 如果一组权重可以在图像中某个区域提取出有效的表示, 那么它也能在图像的另外的区域中提取出有效的表示. 也就是说, 如果一个图案出现在图像中的某个区域, 那么它也可以出现在图像中的其他任何区域. 因此在相同激活映射上的不同位置的神经元共享相同的权重, 用于发现图像中不同位置的相同模式.

通过这两个简化, 可以大幅减少参数的数量.

卷积

根据上面两个假设, 第 d_l 个输出映射的 i_l, j_l 位置的神经元将是输入神经元张量和第 d_l 个滤波器的卷积 *(convolution)* 卷积层的名字也因此得来,[4]每个 $\mathbf{W}_{d_l}$ 称为滤波器 *(filter)* 或核 *(kernel)*.

$$\mathbf{A}_{d_l} = \mathbf{X} * \mathbf{W}_{d_l} + b_{d_l} . \tag{5.9}$$

写成求和的形式是

$$\mathbf{A}(i_l, j_l, d_l) = \sum_{i=0}^{F_1-1} \sum_{j=0}^{F_2-1} \sum_{d=0}^{D_{l-1}-1} \mathbf{X}(i_l + i, j_l + j, d)\mathbf{W}(i, j, d, d_l) + b_{d_l}, \forall i_l, j_l, d_l . \tag{5.10}$$

一个二维卷积的例子见图5.6.

[4]严格的说, 应该是互相关函数 *(cross-correlation)*.

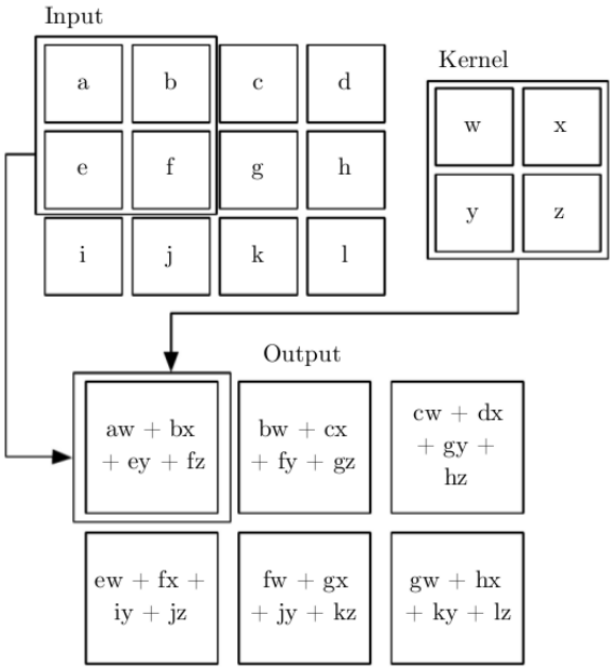


图 5.6: 二维卷积示例. 图像来源于 [9].

A 的维数 $H_l \times W_l \times D_l$ 的计算方法是: D_l 是第 l 使用的滤波器个数; H_l 和 W_l 由下式计算:

$$H_l = H_{l-1} - F_1 + 1\,, \tag{5.11}$$

$$W_l = W_{l-1} - F_2 + 1\,. \tag{5.12}$$

描述卷积层的四个量

之前已经讨论过其中的两个量: 滤波器数目 D_l, 和滤波器的感受野 $F_1 \times F_2$. 还有两个量将在这里讨论.

步幅 *(stride)*S 表示在一个激活映射中, 在空间上, 每跳过 S 个位置计算一个输出神经元, 见图5.7. 因此, 大的 S 可以使输出映射的尺寸变小.

$$\mathbf{A}(i_l, j_l, d_l) = \sum_{i=0}^{F_1-1} \sum_{j=0}^{F_2-1} \sum_{d=0}^{D_{l-1}-1} \mathbf{X}(i_l S+i, j_l S+j, d)\mathbf{W}(i, j, d, d_l)+b_{d_l}, \forall i_l, j_l, d_l\,. \tag{5.13}$$

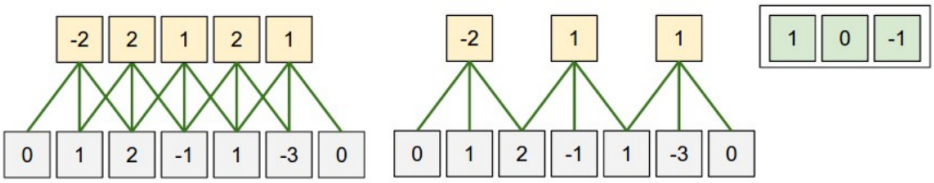


图 5.7: 步幅. 左图步幅为 1, 右图步幅为 2. 图像来源于 [7].

可以看出, 式5.10是 $S = 1$ 的特殊情况.

0-填充 (zero-padding)P. 有时在输入进行卷积之前, 我们会在输入的四周填充一些 0, 高和宽分别填充的大小是 P_1, P_2. 通过 0-填充, 可以使我们控制输出特征映射的大小.

当考虑到步幅和 0-填充后, 输出 $\mathbf{A}$ 宽和高的计算方法有所变化.

$$H_l = \frac{H_{l-1} - F_1 + 2P_1}{S} + 1\,, \tag{5.14}$$

$$W_l = \frac{W_{l-1} - F_2 + 2P_2}{S} + 1\,. \tag{5.15}$$

当 $S = 1$ 时, 通过设定

$$P_1 = \frac{F_1 - 1}{2}\,, \tag{5.16}$$

$$P_2 = \frac{F_2 - 1}{2}\,. \tag{5.17}$$

将保证

$$H_l = H_{l-1}\,, \tag{5.18}$$

$$W_l = W_{l-1}\,. \tag{5.19}$$

5.2.2 汇合层 (pool)

汇合层根据神经元空间上的局部统计信息进行采样, 在保留有用信息的同时减少神经元的空间大小, 进一步使参数量减少并降低过拟合的可能.

pool 操作在各深度分量上独立进行, 常用的是*最大汇合 (max-pooling)*.

$$\mathbf{A}(i_l, j_l, d_l) = \max_{0 \le i < F_1, 0 \le j < F_2} \mathbf{X}(i_l S + i, j_l S + j, d_l), \forall i_l, j_l, d_l\,. \tag{5.20}$$

常用的是每个滤波器的大小是 2×2, 步幅 $S = 2$, 在每个局部空间采用 max 操作, 舍弃 75% 的信息, 见图5.8.

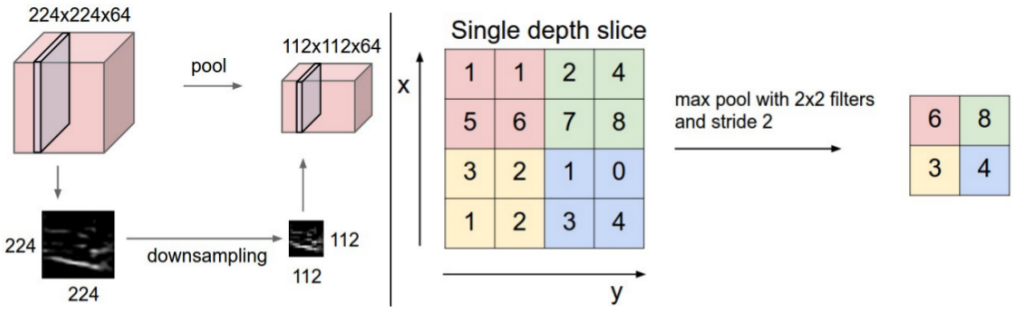


图 5.8: 最大化汇合. 图像来源于 [7].

5.2.3 线性整流层 (relu)

和在普通神经网络中 relu 操作相同, 线性整流层是逐元素的进行 relu 操作, 不会改变维度.

$$\mathbf{A}(i_l, j_l, d_l) = \max(0, \mathbf{X}(i_l, j_l, d_l)), \forall i_l, j_l, d_l \,. \tag{5.21}$$

5.2.4 全连接层 (fc)

和在普通神经网络中全连接操作相同, 每个输出神经元和所有的输入神经元保持连接, 在此不再赘述.

5.3 损失函数

损失函数和普通神经网络中损失函数相同, 只不过

$$\vec{s}^{(i)} = h(\mathbf{X}^{(i)}) \tag{5.22}$$

中 h 是由多个卷积层, 汇合层, relu 层, 和全连接层组合而成的.

5.4 优化

和普通神经网络中优化过程相同, 都是基于 BP 算法的梯度下降规则. 因此问题的关键是对四种不同结构如何计算出对参数和对上一层神经元的导数. 由于 relu 层和全连接层的计算方法和上一章中相同, 因此下面将介绍卷积层和汇合层的反向传播操作.

5.4.1 卷积层的反向传播

卷积层完成的操作是

$$\mathbf{A}(i_l, j_l, d_l) = \sum_{i=0}^{F_1-1} \sum_{j=0}^{F_2-1} \sum_{d=0}^{D_{l-1}-1} \mathbf{X}(i_l + i, j_l + j, d)\mathbf{W}(i, j, d, d_l) + b_{d_l}, \forall i_l, j_l, d_l \,. \tag{5.23}$$

假设 $\nabla_{\mathbf{A}} \mathrm{err}^{(i)}$ 已知, 需要计算 $\nabla_{\mathbf{W}} \mathrm{err}^{(i)}$ 和 $\nabla_{\tilde{b}} \mathrm{err}^{(i)}$ 用于更新参数, $\nabla_{\mathbf{X}} \mathrm{err}^{(i)}$ 用于将误差向上一层传播.

$$\frac{\partial \mathrm{err}^{(i)}}{\partial \mathbf{W}(i, j, d, d_l)} = \sum_{i_l=0}^{H_l-1} \sum_{j_l=0}^{W_l-1} \frac{\partial \mathrm{err}^{(i)}}{\partial \mathbf{A}(i_l, j_l, d_l)} \frac{\partial \mathbf{A}(i_l, j_l, d_l)}{\partial \mathbf{W}(i, j, d, d_l)} \tag{5.24}$$

$$= \sum_{i_l=0}^{H_l-1} \sum_{j_l=0}^{W_l-1} \frac{\partial \mathrm{err}^{(i)}}{\partial \mathbf{A}(i_l, j_l, d_l)} \mathbf{X}(i_l + i, j_l + j, d), \forall i, j, d \,, \tag{5.25}$$

$$\frac{\partial \mathrm{err}^{(i)}}{\partial b_{d_l}} = \sum_{i_l=0}^{H_l-1} \sum_{j_l=0}^{W_l-1} \frac{\partial \mathrm{err}^{(i)}}{\partial \mathbf{A}(i_l, j_l, d_l)} \frac{\partial \mathbf{A}(i_l, j_l, d_l)}{\partial b_{d_l}} \tag{5.26}$$

$$= \sum_{i_l=0}^{H_l-1} \sum_{j_l=0}^{W_l-1} \frac{\partial \mathrm{err}^{(i)}}{\partial \mathbf{A}(i_l, j_l, d_l)}, \forall d_l \,, \tag{5.27}$$

$$\frac{\partial \mathrm{err}^{(i)}}{\partial \mathbf{X}(i_l + i, j_l + j, d)} = \sum_{i_l=0}^{H_l-1} \sum_{j_l=0}^{W_l-1} \frac{\partial \mathrm{err}^{(i)}}{\partial \mathbf{A}(i_l, j_l, d_l)} \frac{\partial \mathbf{A}(i_l, j_l, d_l)}{\partial \mathbf{X}(i, j, d)} \tag{5.28}$$

$$= \sum_{i_l=0}^{H_l-1} \sum_{j_l=0}^{W_l-1} \frac{\partial \mathrm{err}^{(i)}}{\partial \mathbf{A}(i_l, j_l, d_l)} \mathbf{W}(i, j, d, d_l), \forall i, j, d \,. \tag{5.29}$$

注意到, 卷积操作的反向传播还是卷积操作, 只不过需要将滤波器空间翻转.

5.4.2 汇合层的反向传播

汇合层没有参数, 不需要进行参数更新, 因此汇合层在反向传播时需要完成的工作是将第 l 层的导数传播到第 $l-1$ 层.

首先讨论 max 操作在反向传播中的作用. 考虑一个简单的运算

$$z = \max(x, y) = 1\{x \geq y\}x + 1\{y > x\}y \,. \tag{5.30}$$

假设我们知道了 $\frac{\partial \mathrm{err}^{(i)}}{\partial z}$, 要计算 $\frac{\partial \mathrm{err}^{(i)}}{\partial x}$ 和 $\frac{\partial \mathrm{err}^{(i)}}{\partial y}$,

$$\frac{\partial \mathrm{err}^{(i)}}{\partial x} = \frac{\partial \mathrm{err}^{(i)}}{\partial z} \frac{\partial z}{\partial x} \tag{5.31}$$

$$= \frac{\partial \mathrm{err}^{(i)}}{\partial z} 1\{x \geq y\}\,, \tag{5.32}$$

$$\frac{\partial \mathrm{err}^{(i)}}{\partial y} = \frac{\partial \mathrm{err}^{(i)}}{\partial z}\frac{\partial z}{\partial y} \tag{5.33}$$

$$= \frac{\partial \mathrm{err}^{(i)}}{\partial z} 1\{y > x\}\,. \tag{5.34}$$

直观上讲, max 操作就像是导数的路由. 它会把导数 $\frac{\partial \mathrm{err}^{(i)}}{\partial z}$ 保持不变地只传给一个输入, 这个输入是所有输入变量中最大的一个, 其他输入变量得到的导数为 0.

汇合层前向传播是

$$\mathbf{A}(i_l, j_l, d_l) = \max_{0 \leq i < F_1, 0 \leq j < F_2} \mathbf{X}(i_l S + i, j_l S + j, d_l), \forall i_l, j_l, d_l\,. \tag{5.35}$$

在前向传播时, 需要记录下每个 $F_1 \times F_2$ 局部区域最大值所对应的索引. 反向传播时, 直接将第 l 层导数传播到这些索引位置, 第 $l-1$ 层其他位置导数置 0 即可. 因此, 汇合层的反向传播操作会十分高效.

5.5 预测和评估

CNN 的预测和评估方法和上一章讨论的神经网络的预测和评估方法相同, 在此不再赘述.

Chapter 6

实现细节

在做工程实现时, 除了理论推导要正确外, 还要考虑诸如计算机的数值稳定性, 计算速度, 收敛快慢等问题. 本章将就一些实现细节问题加以讨论.

6.1 Softmax 的数值稳定性问题

在计算 softmax 函数

$$f(\vec{s})_k = \frac{\exp(s_k)}{\sum_{j=0}^{K-1}\exp(s_j)}, \forall k = 0, 1, 2, \ldots, K-1 \tag{6.1}$$

时, 如果不注意细节, 很容易造成数值稳定性问题 [9].

当 $s_j = c, \forall j$ 时, 理论上

$$f(\vec{s})_k = \frac{\exp(s_k)}{\sum_{j=0}^{K-1}\exp(s_j)} = \frac{1}{K}, \forall k\,. \tag{6.2}$$

但在实际中考虑两种情况:

(1). c 是一个非常大的负数. $\exp(c)$ 会是一个非常接近 0 的数. 由于计算机能表示的浮点数精度是有限的, 非常接近 0 的数会近似为 0, 这是数值下溢 *(underflow)*. 因此

$$\exp(c) = 0\,, \tag{6.3}$$

$$f(\vec{s})_k = \frac{\exp(c)}{\sum_{j=0}^{K-1}\exp(c)} = \frac{0}{0}, \forall k\,. \tag{6.4}$$

(2). c 是一个非常大的正数. $\exp(c)$ 会是一个非常大的数. 由于计算机的有限的计算精度, 非常大的数字会被近似为 ∞, 这是数值上溢 *(overflow)*. 因此

$$\exp(c) = \infty\,, \tag{6.5}$$

$$f(\vec{s})_k = \frac{\exp(c)}{\sum_{j=0}^{K-1}\exp(c)} = \frac{\infty}{\infty}, \forall k\,. \tag{6.6}$$

注意到 softmax 函数满足性质

$$f(\vec{s})_k = \frac{\exp(s_k)}{\sum_{j=0}^{K-1}\exp(s_j)} \tag{6.7}$$

$$= \frac{\exp(s_k)c}{\sum_{j=0}^{K-1}\exp(s_j)c}, \forall c > 0 \tag{6.8}$$

$$= \frac{\exp(s_k + \log c)}{\sum_{j=0}^{K-1}\exp(s_j + \log c)} \tag{6.9}$$

$$= \frac{\exp(s_k + c')}{\sum_{j=0}^{K-1}\exp(s_j + c')}, \forall c'\,. \tag{6.10}$$

即在分子分母同乘以一个常数不会影响 softmax 的结果, 但是可以利用这个性质来解决数值稳定性问题.

取

$$c' = -\max_l s_l\,. \tag{6.11}$$

即在分子分母的指数上同时减去 $\vec{s}$ 最大分量, 使得在指数上最大的一项为 0, 这去掉了发生上溢的可能性. 同样的, 分母中至少有一项为 1, 这去掉了因为下溢而导致分子分母同时为 0 的可能性.

6.2 卷积操作的实现

在卷积层中需要完成操作

$$\mathbf{A}(i_l, j_l, d_l) = \sum_{i=0}^{F_1-1}\sum_{j=0}^{F_2-1}\sum_{d=0}^{D_{l-1}-1}\mathbf{X}(i_l+i, j_l+j, d)\mathbf{W}(i, j, d, d_l) + b_{d_l}, \forall i_l, j_l, d_l\,. \tag{6.12}$$

如果直接按上式在计算机中实现, 上式 3 重求和对应 3 重 **for** 循环, 对 i_l, j_l, d_l 遍历又需要 3 重 **for** 循环, 而且通常我们进行批梯度下降, 对批数据中每个训练示例遍历也需要 1 重 **for** 循环, 共计需要 7 重 **for** 循环. 7 重

for 循环的计算效率很低, 即使使用向量化技巧去掉 3 重求和对应的 3 重 **for** 循环, 还需要 4 重 **for** 循环, 这个计算代价仍然很高. 因此, 需要更加高效的方法计算卷积. 大致有以下两种方法.

6.2.1 傅里叶变换

根据傅里叶变换 *(Fourier transform)* 的性质, $f * g$ 的傅里叶变换等于 f 的傅里叶变换乘 g 的傅里叶变换

$$\mathfrak{F}(f * g) = \mathfrak{F}(f) \cdot \mathfrak{F}(g) . \tag{6.13}$$

这样可以将卷积运算转化为普通的乘法运算.

在卷积层内实现的卷积操作是

$$\mathbf{A}_{d_l} = \mathbf{X} * \mathbf{W}_{d_l} + b_{d_l}, \forall d_l . \tag{6.14}$$

利用傅里叶变换, 可以写成

$$\mathbf{A}_{d_l} = \mathfrak{F}^{-1}(\mathfrak{F}(\mathbf{X}) \odot \mathfrak{F}(\mathbf{W}_{d_l})) + b_{d_l}, \forall d_l . \tag{6.15}$$

通过快速傅里叶变换 *(fast Fourier transform, FFT)*, 可以在 $O(N \lg N)$ 的时间内计算完长 N 向量的傅里叶变换, FFT 也可以扩展到 2 维矩阵中. 但是利用 FFT 计算卷积在滤波器尺寸大的时候有很大加速, 对于通常使用的 3×3 滤波器加速不明显 [34]. 而且 FFT 无法很高效的处理步幅 $S > 1$ 的情况, 因此在实际上使用的并不多.

6.2.2 im2col

考虑到矩阵的乘法运算在计算机中已经有非常高效的实现方法, im2col 的思路是将卷积运算转化为普通的矩阵乘法运算. 方法如下:

第一步将和每个输出神经元相连的输入张量中的局部区域展成一个行向量, 将所有的向量拼接成一个矩阵 $\tilde{X}$. 每个局部区域的维度是 $F_1 \times F_2 \times D_{l-1}$, 共有 $H_l \times W_l$ 个这样的区域, 因此 $\tilde{X}$ 的维度是 $(H_l W_l) \times (F_1 F_2 D_{l-1})$.

$\mathbf{X}$ 和 $\tilde{X}$ 之间的对应关系可由如下方法算出 [35]: 用 (p, q) 索引 $\tilde{X}$ 中的元素, 用 (i, j, d, d_l) 索引 $\mathbf{W}$ 中的元素, 用 (i_l, j_l, d_l) 索引 $\mathbf{A}$ 中的元素, 用 $(i_{l-1}, j_{l-1}, d_{l-1})$ 索引 $\mathbf{X}$ 中的元素, 则有关系

$$i_{l-1} = i_l + i , \tag{6.16}$$

$$j_{l-1} = j_l + j\,, \tag{6.17}$$

$$p = i_l + H_l \times j_l\,, \tag{6.18}$$

$$q = i + F_1 \times j + F_1 \times F_2 \times d_{l-1}\,. \tag{6.19}$$

p 决定了对应 $\mathbf{A}$ 中的空间位置 (i_l, j_l), q 决定了对应 $\mathbf{X}$ 中的深度位置 d_{l-1} 和空间位移 (i, j), . 将 (i, j) 和 (i_l, j_l) 结合起来, 就可以得到对应 $\mathbf{X}$ 中的空间位置 (i_{l-1}, j_{l-1}).

第二步将每个滤波器 $\mathbf{W}_{d_l}$ 展成列向量, 将所有的列向量拼接成一个矩阵 $\tilde{W}$. 每个滤波器的维度是 $F_1 \times F_2 \times D_{l-1}$, 共有 D_l 个滤波器, 因此 $\tilde{W}$ 的维度是 $(F_1 F_2 D_{l-1}) \times D_l$.

第三步将 $\tilde{X}$ 和 $\tilde{W}$ 相乘, 再加上 $\vec{b}$.

$$\tilde{A} = \tilde{X}\tilde{W} + \vec{1}\vec{b}^T\,. \tag{6.20}$$

$\tilde{A}$ 的维度将是 $(H_l W_l) \times D_l$.

第四步将 $\tilde{A}$ 转为张量, 即可得到 $\mathbf{A} \in \mathbb{R}^{H_l \times W_l \times D_l}$.

这种方法的弊端是 $\tilde{X}$ 会占用很大的内存, 因为 $\mathbf{X}$ 中的每个元素会在 $\tilde{X}$ 中多次出现. 但是 im2col 利用了矩阵运算的高效实现, 因此实际中使用的比较多. 卷积层的前向, 反向, 汇合操作也可以用 im2col 实现.

6.3 参数更新

前面讨论的参数更新方法都是梯度下降

$$\vec{\theta} \leftarrow \vec{\theta} - \alpha \nabla J(\vec{\theta})\,. \tag{6.21}$$

事实上, 还有很多基于梯度下降的方法, 但是在深度网络中有着更好的收敛速率.

在梯度下降方法中, 确定合适的*超参数 (hyperparameter)*α 非常困难, 学习速率 α 选择的好坏直接影响到网络收敛情况及网络的性能. 我们希望有一种方法可以自适应的调整学习速率, 甚至对 $\vec{\theta}$ 中的每个参数 θ_j 有各自的学习速率. RMSprop[33] 就是这样一种方法, 它的参数更新方法如下

$$v_j \leftarrow \beta v_j + (1-\beta) \cdot (\frac{\partial J(\vec{\theta})}{\partial \theta_j})^2, \forall j\,, \tag{6.22}$$

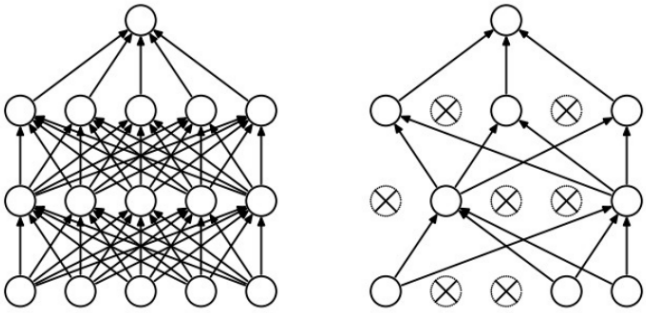
$$\theta_j \leftarrow \theta_j - \frac{\alpha}{\sqrt{v_j + \varepsilon}} \cdot \frac{\partial J(\vec{\theta})}{\partial \theta_j}, \forall j\,. \tag{6.23}$$

$\vec{v}$ 是缓存变量, 和 $\vec{\theta}$ 的大小相同, 用来记录导数的平方. 通过 $\vec{v}$ 调整各参数的学习速率. 如果某个参数持续的得到高的梯度, 它的学习速率将降低; 如果某个参数持续得到低的梯度, 它的学习速率将增大. β 通常被设为 $\beta \in \{0.9, 0.99, 0.999\}$. ε 通常设定为 $\varepsilon \in [10^{-8}, 10^{-4}]$, 用于避免除 0 错误.

除 RMSprop 外, 还有很多常用基于梯度下降的参数更新规则, 比如 Momentum[27], Nesterov Momentum[24], Adagrad[4], Adam[16].

6.4 退出

退出 *(dropout)*[30] 是一种非常简单而高效的正则化方法. 它以 p 的概率随机保留一些神经元, 其余的神经元输出将被设为 0, 见图6.1. 这可以认为是在训练过程中从原始网络中提取一个小的网络, 参数更新只在这个小的网络中进行. 在测试过程中不适用退出, 这可以认为是无数小的网络所做的预测的集成 *(ensemble)*.



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

图 6.1: 退出. 右图是左图神经网络丢弃之后的结果. 图像来源于 [30].

6.5 数据初始化

一个常用的数据初始化策略是, 对训练集中的每个特征, 减去它在训练集中的均值.

$$\hat{\mathbf{X}} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{X}^{(i)}, \tag{6.24}$$

$$\mathbf{X}^{(i)} \leftarrow \mathbf{X}^{(i)} - \hat{\mathbf{X}}, \forall i. \tag{6.25}$$

6.6 参数初始化

输入数据在初始化后是以 0 为中心的, 因此我们希望参数一部分大于 0, 一部分小于 0.

但是如果我们将全部的参数初始化为 0, 那么一层中所有的神经元将计算得到相同的输出, 在反向传播时将得到相同的导数, 参数将进行相同的更新, 也就是说, 这会导致一层中所有的神经元是冗余的.

因此, 我们将参数初始为很接近 0 但不全是 0 的数, 比如从高斯分布 $N(0, 0.01^2)$ 中采样作为初始值.

6.7 随机裁剪

很多情况下, 训练数据越多, 模型的性能越好. 随机裁剪 [18] 就是一种*数据扩充 (data augmentation)* 的方法. 在训练时, 随机在原图中裁剪一些区域, 将这些区域送给神经网络去训练. 在测试时, 取图像中的 5 个固定位置 (4 个角落 + 中央) 让神经网络预测, 将预测的结果平均之后作为最终的输出.

参考文献

[1] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[2] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[4] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[5] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. In *Advances in neural information processing systems*, pages 2366–2374, 2014.

[6] Andrew Ng et al. UFLDL tutorial. Stanford.

[7] Li Fei-Fei, Andrej Karpathy, and Justin Johnson. CS231n: Convolutional neural networks for visual recognition. Stanford, 2016.

[8] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmen-

tation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[14] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[15] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[16] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[19] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[20] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[21] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[22] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[23] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[24] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[25] Andrew Ng. CS229: Machine learning. Stanford.

[26] Andrew Ng. Machine learning. Coursera.

[27] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[28] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[31] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[32] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[33] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4:2, 2012.

[34] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

[35] Jianxin Wu. CNN for dummies. 2016.

[36] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Advances in neural information processing systems*, pages 487–495, 2014.

[37] 周志华. 机器学习. 清华大学出版社, 2016.