

# 从零开始的深度学习课程

## I 基础原理篇

谢晋璟

Commerce Data Science Team

CDL数据科学社区（原BDA社区）

数据科学工作室

技术茶会

2016-07-15

# 引子



# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# 课程背景

## 背景

针对完全没有机器学习基础，或是使用过机器学习的library，但希望更进一步了解的工程师。

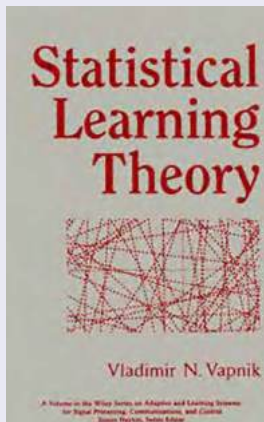
## 大纲

从零开始的《入门篇》包括：

- 基础原理篇
- 工具篇
- 应用篇

# 基础原理篇

Not This



But ...

# 原理篇的内容

原理篇里我们讲述的不是指类似上面的讲述为什么它能工作的原理。  
而是具体的**How to do it**.

## 我们会遇到下面的内容

- 常见的Machine Learning的普遍形式。
- Loss Function概念，各种loss function
- 求导，复合函数求导，向量函数求导（求梯度，Jacobian），张量的计算
- Neuron network作为复合的向量函数
- Backpropagation算法，一般形式和Neuron network下推导
- 具体例子：Affine层，ReLU层，卷积层，Pooling层

# 不解释了！上车！



# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度



# ML的最常见形式

市面上最常见最简单的ML系统的输出都可以看作是如下的东西:

$$f: \mathcal{X} \rightarrow \mathcal{S}$$

这里:

- $\mathcal{X}$  是**feature**空间, 代表可以观测到的一些信号。
- $\mathcal{S}$  是**score**空间, 代表你期望得到的预测值。这里的**score**不一定是最终的输出结果。
- $f$  就是**prediction function**, (ML里术语也叫**hypotheses**)

ML系统的中**training**的目的, 就是从一个很大的预测函数集合 $\mathcal{H}$  (一般这个 $\mathcal{H}$ 被称为**hypotheses set**) 中找到一个 $f$ , 使得它满足一定的要求。

# Lost Function

什么样的 $f$ 才是我们想要的？

我们需要一个评价标准，一般称为损失函数 $L$ ，损失越小的 $f$ 就越好。

常见的 $L$ 都是point level的

即对于每一条数据点 $(x_i, y_i)$ ，计算一个实数 $L(X, f(X), Y)$

$$L : \mathcal{X} \times \mathcal{S} \times \mathcal{Y} \rightarrow \mathbb{R}$$

这里 $L$ 依赖于输入数据里有一个“教导”信号 $Y$ （告诉你正确的应该怎么做—类似喋喋不休的人生导师:），这类 $L$ 诱导的方法被称为supervised Learning。

如果数据没有告诉你正确的如何做，只告诉你做对或做错，这种一般归类于**Reinforcement learning**。本课程里不涉及。

如果数据没有 $Y$

我们必须选择不依赖于 $Y$ 的 $L$ ,这类 $L$ 就诱导了所谓的**unsupervised learning**。

需要注意的是：我们这里假设 $L$ 都是**point level**的

而对 $f$ 好坏的整体评估，一般使用平均损失

$$E[L(X, f(X), Y)]$$

在实际应用中， $E$ 通常用经验平均代替：

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

这里 $N$ 是用于评估的数据集的个数。如果评估的数据集是用了训练数据，上面的估计就叫做 $E_{\text{in}}$ （**in sample error**）；评估数据集是测试数据，上面的估计就叫做 $E_{\text{out}}$ （**out of sample error**）。

# Lost Function例子

- 平方误差:

$$L(x, y_{\text{pred}}, y) = (y - y_{\text{pred}})^2$$

这里的score空间和最终的输出结果空间 $\mathcal{Y}$ 是一致的

- 0/1-Logistic的极大似然:

$$L(x, s, y) = -\log(\varphi(s)^y \cdot (1 - \varphi(s))^{1-y})$$

这里score空间是 $\mathbb{R}$ , output空间 $\mathcal{Y} = \{0, 1\}$ 的。

注:  $\varphi(s) = \frac{1}{1+e^{-s}}$

- $\pm 1$ -Logistic的极大似然:

$$L(x, s, y) = -\log(\varphi(y \cdot s))$$

这里score空间还是 $\mathbb{R}$ , 但 $\mathcal{Y} = \{-1, 1\}$ 的, 所以公式化简了。

# Lost Function例子

- Cross-Entropy loss(softmax的极大似然)

$$L(x, s, y) = -s_y + \log\left(\sum_{i=1}^m e^{s_i}\right)$$

这里score是个 $\mathbb{R}^m$ 向量，而 $y \in \mathcal{Y} = \{1, 2, \dots, m\}$ 。

- SVM loss

$$L(x, s, y) = \sum_{i \neq y} (s_i - s_y + 1)_+$$

当正确的类 $y$ 的score:  $s_y$ 比其他类score:  $s_i$ 都大上至少一个margin（这里是1）时才不产生loss。

# 如何求 $f$ ?

## 最优化的思想

为了能使用数学求解，我们需要hypotheses的集合 $\mathcal{H}$ 能够被参数化。这种参数化了的空间里的成员一般可以记作 $f(X; \theta)$ 形状，参数空间在没有其他约束的情况下就是向量空间 $\theta \in \mathbb{R}^n$ 。而我们的目标：寻找一个 $f$ ，就转换成下面的数学问题，

$$\min_{\theta} E[L(X, f(X; \theta), Y)]$$

如果把后面的一堆东西看做一个关于 $\theta$ 的函数：

$$g(\theta) = E[L(X, f(X; \theta), Y)]$$

这就是一个求函数最小值的问题。

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# 函数极值如何求？

答案

求导

复习

传说中挂了很多高数告诉我们，一元函数 $g(x)$ 如果可以求导，那么它的极值点的导数等于0。

所以求出导数等于0的点，就是可能的最小值点。

比如 $x^2 + ax + b$  在 $2x + a = 0$  即 $x = -a/2$  处取到极值。

对于多元函数，极值点的必要条件是：每个方向的偏导都等于0：

$$\frac{\partial g(x_1, x_2, \dots, x_i, \dots, x_n)}{\partial x_i} = 0, i = 1, \dots, n$$



## closed form

如果求解上面的方程组很容易，或有公式解，那么问题到此就可以解决了

## optimization algorithm

大部分情况没有公式解。就需要使用优化算法。但为什么要自己求导？很多优化算法不是只要一个目标函数么？

因为这样速度快啊！

# 复合函数求导

## 链式法则

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$$

## 多元函数情况

令  $f(z_1, z_2, \dots, z_k)$  为多元函数,  $g_i(x), i = 1, \dots, k$  为一组函数。

$$\frac{d}{dx} f(g_1(x), g_2(x), \dots, g_k(x)) = \frac{\partial f}{\partial z_1} g'_1(x) + \dots + \frac{\partial f}{\partial z_k} g'_k(x)$$

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# 梯度和Jacobian

现实情况中，参数空间基本上不可能是一维的，对目标函数 $g$ ，所有方向偏导形成一个向量而非一个数：

## Gradient

$$\nabla_{\theta} g = \left( \frac{\partial g}{\partial \theta_1} \quad \frac{\partial g}{\partial \theta_2} \quad \cdots \quad \frac{\partial g}{\partial \theta_n} \right)$$

所以一个标量（0-阶张量）通过求梯度变成了一个向量（1-阶张量）。如果原来的 $g$ 是向量映射 $g = (g_1, g_2, \dots, g_m)$ ，求导就得到Jacobian矩阵。

## Jacobian

$$\frac{\partial g}{\partial X} = \begin{pmatrix} \frac{\partial g_1}{\partial X_1} & \frac{\partial g_1}{\partial X_2} & \cdots & \frac{\partial g_1}{\partial X_n} \\ \frac{\partial g_2}{\partial X_1} & \frac{\partial g_2}{\partial X_2} & \cdots & \frac{\partial g_2}{\partial X_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial X_1} & \frac{\partial g_m}{\partial X_2} & \cdots & \frac{\partial g_m}{\partial X_n} \end{pmatrix}$$

所以一个向量（1-阶张量）通过求导变成了一个矩阵（2-阶张量）。从这两个例子知道，求导算子会提升张量的阶。

# Hessian-二阶张量

如果要计算的 $g$ 是某个函数的梯度:  $\mathbf{g} = \nabla g = (\frac{\partial g}{\partial X_1}, \dots, \frac{\partial g}{\partial X_n})$ , 它的Jacobian就是原来 $g$ 的Hessian (二阶导数)

## Hessian

$$\mathbf{H}(g) = \begin{pmatrix} \frac{\partial^2 g}{\partial X_1^2} & \frac{\partial^2 g}{\partial X_1 \partial X_2} & \cdots & \frac{\partial^2 g}{\partial X_1 \partial X_n} \\ \frac{\partial^2 g}{\partial X_2 \partial X_1} & \frac{\partial^2 g}{\partial X_2^2} & \cdots & \frac{\partial^2 g}{\partial X_2 \partial X_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 g}{\partial X_n \partial X_1} & \frac{\partial^2 g}{\partial X_n \partial X_2} & \cdots & \frac{\partial^2 g}{\partial X_n^2} \end{pmatrix}$$

所以多元标量函数求高阶导, 会产生高阶的张量。

# 题外话

曾经有大师说过

“这世界上没有什么不能通过求导解决的，如果有，那就再求一次。。。 ”

运筹学中的优化算法大部分也都用到了二阶导的信息，用到了Hessian。然而

Hessian存储和计算（包括产生和求逆）都耗费大量内存和计算资源，在大规模的神经网络计算中并没有使用，而是用一些拟二阶方法代替（用很容易得到的矩阵，比如对角阵去近似Hessian）。

高于二阶的导数在实际应用的优化算法中出现是非常少见的。

# 链式法则的向量（张量）情况

考虑这样的函数：

$$l(x) = f \circ \mathbf{g} \circ \mathbf{h}(x) = f(g_1(h_1(x), \dots, h_k(x)), \dots, g_n(h_1(x), \dots, h_k(x)))$$

运用链式法则

$$l'(x) = \sum_i \frac{\partial f}{\partial g_i} \sum_j \frac{\partial g_i}{\partial h_j} h'_j(x) = \sum_{i,j} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial h_j} h'_j(x)$$

可以看作是三个矩阵相乘

$$\nabla_{\mathbf{g}} f \cdot \left[ \frac{\partial \mathbf{g}}{\partial \mathbf{h}} \right]_{ij} \cdot \mathbf{h}'$$

这里  $\left[ \frac{\partial \mathbf{g}}{\partial \mathbf{h}} \right]_{ij}$  就是一个二阶张量（ $\mathbf{g}$ 的Jacobian矩阵）

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度



# 例子：两层神经网络

抛开那些神经网络的图示啊，动力学特性啊，它的结构异常简单，就是多个向量多元函数的复合。

## 两层网络

第一层输入是 $X$ ，输出是隐层 $Z$

$$Z = (f_1(X; W_1), \dots, f_h(X; W_1))$$

第二层输入是 $Z$ ，输出是score向量 $S$

$$S = (g_1(Z; W_2), \dots, g_t(Z; W_2))$$

这里 $f$ 和 $g$ 都是向量函数，他们的复合就是我们一开始说的prediction function。不同的参数取值 $W_1, W_2$ 给出了不同的hypotheses，训练的过程就是找到一个使得 $L(S, y)$  在全局上最优的过程。

注：这里 $L$ 不依赖于 $X$ ，所以我们省略 $X$ 。

# 神经网络就像乐高积木

每一层的向量映射 $f^i$ 就如同乐高里的小积木，函数复合就是“拼”积木，不同的拼装方式得到不同的hypotheses。



在文献和代码里，这些 $f$ 常常被组织成层（Layer）。最简单的情况，一个 $f$ 就是一个Layer。也可以两个 $f$ 复合起来当作一个Layer。这和写代码的喜好有关，不影响实质。

# 功能层面

任何神经网络结构都有至少需要两个功能：预测和更新权值。

## 预测

给定一个输入数据，计算一个**score**（某种预测结果）

而训练的目的在于通过计算**Lost Function**（例如，前面提到的**softmax**）得到的损失值 $l = L(S, y)$ 来不断修正我们的**hypotheses**。

## 更新权值

将损失信号 $l$ 正确地反映为自身参数的修正量

由于预测时，数据是一层一层从下到上流动，称为**向前**过程。而反馈信号是从上到下一层层返回来，称为**向后**过程。

## Forward/Backward

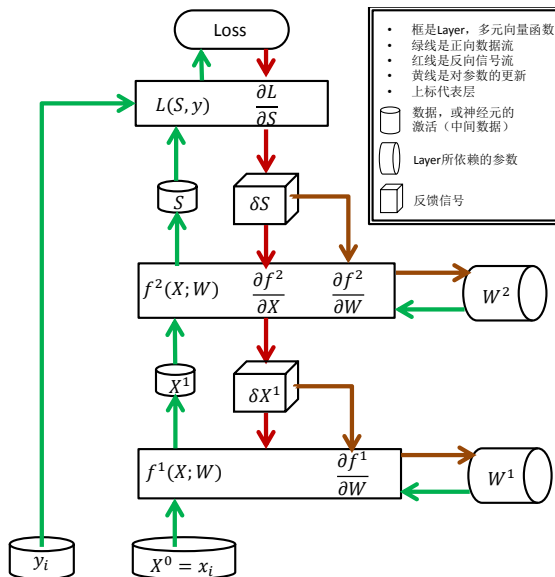
**向前** 数据一层层的被该层的 $f^i$ 作用，传到下一层

**向后** 从**Loss**算出的反馈信号一层层的通过 $f^i$ 的**Jacobian**传回上一层

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# Looking from 30000 inch height



# 前向传播信号流

我们约定上标代表层而非指数，从一个输入数据计算loss过程如下：

**输入**  $X^0 = x$ ，这里  $(x, y) \in \mathcal{D}$  是一个数据点

**Forward**  $X^i = \mathbf{f}^i(X^{i-1}; W^i)$ ，每个  $X^i$  可以想象成  $i$  层神经元的激活，参数  $W^i$  决定的  $\mathbf{f}^i(\cdot, W^i)$  是第  $i-1$  层神经元的激活信号到第  $i$  层神经元的传播通路。

用  $h_i$  记该层的神经元个数，那么  $\mathbf{f}^i$  是一个多元向量映射

$$\mathbf{f}^i : \mathbb{R}^{h_{i-1}} \rightarrow \mathbb{R}^{h_i}$$

**Score** 用  $q$  记总的层数，最后一层输出就是score，记为

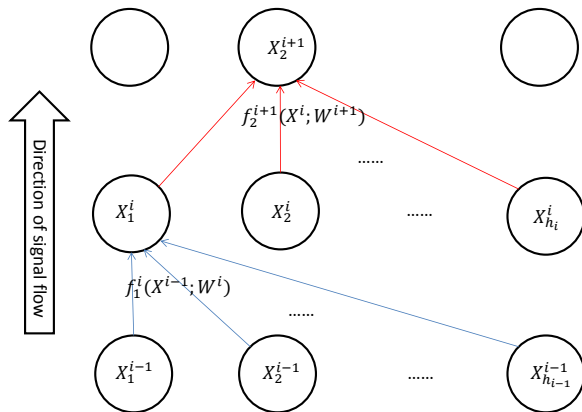
$$S = X^q = \text{Net}(x; W)$$

这里  $W = \{W^1, W^2, \dots, W^q\}$  是该网络Net的全体参数

**Loss** 这个数据点  $(x, y)$  的训练损失就是

$$l(\text{Net}(x), y) = L(S, y)$$

# feed forward



# 后(反)向传播信号流

定义第 $i$ 层上反馈信号

$$\delta X^i = \left( \frac{\partial l}{\partial X_1^i}, \frac{\partial l}{\partial X_2^i}, \dots, \frac{\partial l}{\partial X_{h_i}^i} \right)$$

那么反馈信号的流动是：

输入  $\delta l = \left( \frac{\partial l}{\partial s_1}, \frac{\partial l}{\partial s_2}, \dots, \frac{\partial l}{\partial s_{h_q}} \right) = \delta X^q$

后向传播  $\delta X^i = \delta X^{i+1} \cdot \frac{\partial X^{i+1}}{\partial X^i}$ , 这里

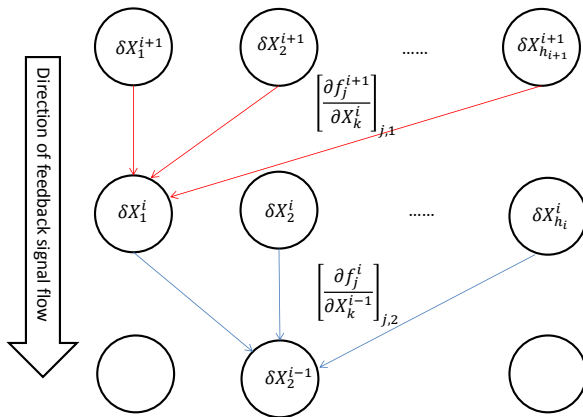
$$\frac{\partial X^{i+1}}{\partial X^i} = \left[ \frac{\partial f_j^{i+1}}{\partial X_k^i} \right]_{jk}$$

是 $f^{i+1}$ 对 $X^i$ 的Jacobian矩阵。

美妙的是，只要Jacobian算好，那么向后传播就是个矩阵乘法而已。



# backpropagation



# 后(反)向传播信号流

我们最终的目的是得到所有层的参数 $W^i$ 的修正量 $dW^i$   
(下面也叫做 $l$ 对 $W^i$ 的反馈)。

对 $\frac{\partial l}{\partial W^i}$ ，利用链式法则，我们有

$$\frac{\partial l}{\partial W^i} = \frac{\partial l}{\partial X^i} \cdot \frac{\partial X^i}{\partial W^i} = \delta X^i \cdot \left[ \frac{\partial f_j^i}{\partial W_k^i} \right]_{jk}$$

其中

$$\left[ \frac{\partial f_j^i}{\partial W_k^i} \right]_{jk}$$

是 $f^i$ 对 $W^i$ 的Jacobian矩阵。

# 后(反)向传播中对参数的update

## 最简单的参数更新

$$W^i \leftarrow W^i - \lambda \cdot \nabla_{W^i} l = W^i - \lambda \cdot \delta X^i \cdot \left[ \frac{\partial f_j^i}{\partial W_k^i} \right]_{jk}$$

$\lambda$ 是learning rate, 代表每次更新的步长。后面的 $\delta X^i \cdot \left[ \frac{\partial f_j^i}{\partial W_k^i} \right]_{jk}$ 就是这里的dW.

我们需要两部分

### local gradient部分

$$\left[ \frac{\partial f_j^i}{\partial W_k^i} \right]_{jk}$$

一般只需要前向时的 $X^{i-1}$ ,  $X^i$ 就可以算出的

### 反馈部分

$$\delta X^i$$

由更高的层后向传播而来

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - **Batched BP**
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# 然而

# 事情并没有那么简单

我们不是每次只考虑一个数据点的

# Batched Update的概念

回忆：经验平均损失

$$E_{\text{empirical}}[L(X, \mathbf{f}(X), Y)] = \frac{1}{N} \sum_{i=1}^N L(x_i, \mathbf{f}(x_i), y_i)$$

batched update

每做一次update，就要遍历一遍数据的方式称为batched update。

这种方法缺点是收敛很慢。

stochastic gradient descent

另一个极端是每看一条数据，就计算这个数据的loss，update相应的参数

它的缺点是数据的variation很大，训练过程中参数振荡地厉害。



# Mini-Batch

## Mini Batch

一个折衷的办法就是每次sample出 $B$ 条数据，在它上面计算经验损失；然后就Update。 $B$ 是一个事先选定的参数。

设当前Mini-Batch的数据是 $(x_i, y_i), i = 1, \dots, B$ ，那么这个Mini-Batch上目标函数变成

## Objective function

$$\text{Obj} = \frac{1}{B} \sum_{i=1}^B l(\text{Net}(x_i), y_i)$$

下面推导这种情况下反向传播的形式

# Mini-Batch Update Rule

## Mini Batch Update Rule

$$W^i \leftarrow W^i - \lambda \cdot \frac{\partial \text{Obj}}{\partial W^i}$$

where

$$\frac{\partial \text{Obj}}{\partial W^i} = \frac{1}{B} \sum_{j=1}^B \frac{\partial l^j}{\partial W^i}$$

where

$$l^j = l(\text{Net}(x_j; W), y_j)$$

# Tensor Version

## Batch version Tensor: $\delta X_k^{i,j}$

每层上的 $X^i$ 现在是个二阶的张量，上标分别代表层和第几个样本，下标是**feature**（神经元）编号

$$X_k^{i,j}, j = 1 \dots B; k = 1 \dots h_i$$

更高的层传至该层的反馈由定义可知：

$$\begin{aligned}\delta X_k^{i,j} &= \frac{\partial \text{Obj}}{\partial X_k^{i,j}} = \frac{\partial}{\partial X_k^{i,j}} \left[ \frac{1}{B} \sum_m l^m \right] \\ &= \frac{1}{B} \cdot \frac{\partial l^j}{\partial X_k^{i,j}}, i = 1 \dots q\end{aligned}$$

## Chain Rule in Tensor Form

$$\begin{aligned}\delta X_k^{i,j} &= \frac{1}{B} \cdot \frac{\partial l^j}{\partial X_k^{i,j}} = \frac{1}{B} \sum_{u,v} \frac{\partial l^j}{\partial X_v^{i+1,u}} \cdot \frac{\partial X_v^{i+1,u}}{\partial X_k^{i,j}} \\&= \frac{1}{B} \sum_v \frac{\partial l^j}{\partial X_v^{i+1,j}} \cdot \frac{\partial X_v^{i+1,j}}{\partial X_k^{i,j}} = \frac{1}{B} \sum_v (B \cdot \delta X_v^{i+1,j}) \cdot \frac{\partial X_v^{i+1,j}}{\partial X_k^{i,j}} \\&= \sum_v \frac{\partial X_v^{i+1,j}}{\partial X_k^{i,j}} \cdot \delta X_v^{i+1,j}\end{aligned}$$

用 $x^{i,j}$ 代表数据 $x_j$ 在forward过程中传到第 $i$ 层时的值，它是个 $\mathbb{R}^{h_i}$ 中的向量。

## Shared Jacobian

By definition,

$$\frac{\partial X_v^{i+1,j}}{\partial X_k^{i,j}} = \frac{\partial f_v^{i+1}(x^{i,j})}{\partial X_k^{i,j}}$$

而记 $J_X(\mathbf{f}^{i+1})(x)$ 为 $\mathbf{f}^{i+1}$ 对 $X$ 的Jacobian在向量 $x$ 上的取值，那么上式用矩阵形式写出就是：

$$\left[ \frac{\partial X_v^{i+1,j}}{\partial X_k^{i,j}} \right]_{vk} = J_X(\mathbf{f}^{i+1})(x^{i,j})$$

所以Chain Rule化简为

$$\delta X^{i,j} = \delta X^{i+1,j} \cdot J_X(\mathbf{f}^{i+1})(x^{i,j}) \quad , \quad j = 1 \dots B$$

这里 $J_X(\mathbf{f}^{i+1})$ 如果有解析形式，那么只要对每个数据点 $x^{i,j}$ 代入计算即可。

# Batched Forward/Backward/Update

用  $J_{v,k}^{i,j}$  记  $J_X(\mathbf{f}^i)(x^{i-1,j})_{v,k}$ , 前向, 后向和update rule就是:

## Tensor Form Forward and Backward

前向:

$$X_k^{i,j} = f_k^i(X^{i-1,j})$$

后向:

$$\delta X_k^{i,j} = \sum_v \delta X_v^{i+1,j} J_{v,k}^{i+1,j}$$

## Tensor Form Update Rule

$$\begin{aligned} \frac{\partial \text{Obj}}{\partial W^i} &= \frac{1}{B} \sum_j \frac{\partial \ell^j}{\partial W^i} = \frac{1}{B} \sum_{j,k} \frac{\partial \ell^j}{\partial X_k^{i,j}} \frac{\partial X_k^{i,j}}{\partial W^i} \\ &= \sum_{j,k} \delta X_k^{i,j} \cdot \nabla_{W^i}(\mathbf{f}_k^i)(x^{i-1,j}) \end{aligned}$$

# 看到这里，大家有没有觉得



然而

革命尚未成功

同志仍需努力

# What's next?

## 从一般

我们无法进一步化简，因为这里假设的  $f^i$  和  $W^i$  都是一般的形状，没有特殊的形式。

然而！

## 到特殊

在神经网络，或是深度学习里， $f^i$  和  $W^i$  都有一定的形状和性质，无论是从编写代码上，还是实际的计算里，可以达到很高的效率。我们下面就开始讨论具体的  $f$ 。



# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - **Affine和ReLU层定义**
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# Affine Transform (Layer)

## Affine Transform

令 $W$ 为 $h_i \times h_{i-1}$ 矩阵,  $b$ 为 $\mathbb{R}^{h_i}$ 向量, 那么下式

$$Y = W \cdot X + b$$

就定义了 $\mathbb{R}^{h_{i-1}} \rightarrow \mathbb{R}^{h_i}$ 映射。(称为仿射变换) 对应于我们以前的定义

$$f_k = W(k, :) \cdot X + b_k$$

原来定义中的 $W^i$ 就是

$$[\text{vec}(W), b]$$

这里 $\text{vec}$ 是把一个矩阵, 或是张量平坦化成一个向量的算子。

**Affine Transform**由参数 $W, b$ 唯一决定。平坦化相当于忘记了它们的形状(上一节做法)。然而这里 $W$ 有矩阵形状, 我们要推导保持这个形状下的公式。用 $W_{kj}^i, b_k^i$ 张量来记这两组参数。

# Rectifier Linear Unit

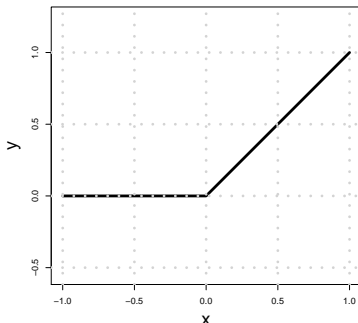
ReLU是一个不依赖于参数的映射：

ReLU

$$Y_k = (X_k)_+ = \max(0, X_k), \quad k = 1, \dots, h_i$$

就是每个分量做截断：小于零的令它为0，正部不变。

ReLU



# Affine ReLU Layer

一个ReLU复合上一个Affine就构成了neuron network中最常见，也是最简单的非线性，可训练的Layer。

前向

$$Y_k = (W(k, :) \cdot X^{i-1})_+, \quad k = 1, \dots, h_i$$

复合函数的Jacobian由链式法则推导，下面分别讨论之。

首先，为了推导方便起见，需要引入如下记号：

Kronecker delta记号

$$\delta_{mn} = \begin{cases} 1 & m = n \\ 0 & m \neq n \end{cases}$$

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - **Affine和ReLU层的Jacobian**
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# Jacobian of Affine mapping

回忆:  $Y = WX + b$

Jacobian w.r.t.  $X$

$$\frac{\partial Y_k}{\partial X_j} = W(k, j)$$

这里 $b$ 不出现在对 $X$ 的Jacobian的计算中

$$J_X(\text{Affine}, W) = W$$

然而 $W(u, v)$ 有了形状,  $Y$ 对 $W$ 的“导数”就写不成矩阵了, 它是一个3-阶张量。对 $b$ 仍然可以写成Jacobian矩阵。

Differential w.r.t.  $W, b$

$$\frac{\partial Y_k}{\partial W(u, v)} = \delta_{ku} \cdot X_v$$

$$\frac{\partial Y_k}{\partial b_u} = \delta_{ku}$$

# Jacobian of ReLU

回忆:  $Y_k = \max(0, X_k)$

相信聪明的你已经注意到了

**它在0点不可导!**

然而, 除去0点外, **ReLU**是光滑从而可以求任意阶导的。0点导数没有定义, 但对我们实际使用没有影响, **ReLU**只是在一个测度为0的集合上不可导而已。

Jacobian w.r.t.  $X$

$$\frac{\partial Y_k}{\partial X_v} = \delta_{kv} \cdot 1_{X_v > 0}$$

这里 $1_{X_v > 0}$ 是示性函数, 它在条件为真时取1, 其他时候取0。

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量，Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - **Affine和ReLU层的BP**
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度



# Backpropagation on ReLU

ReLU比较简单，先看它。

ReLU不带参数，只负责将 $\delta X$ 向更低的层传递

## ReLU's backward

$$\delta X_k^{i,j} = \sum_v \delta_{k,v} \cdot 1_{X_v^{i,j} > 0} \cdot \delta X_v^{i+1,j} = 1_{X_k^{i,j} > 0} \cdot \delta X_k^{i+1,j}$$

## 物理意义解释

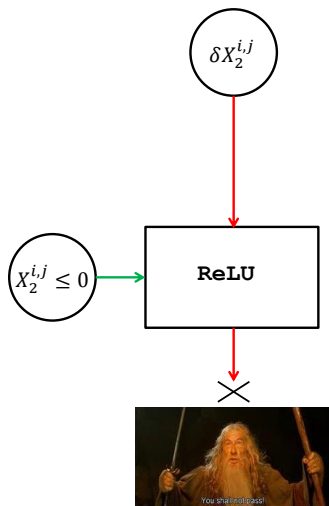
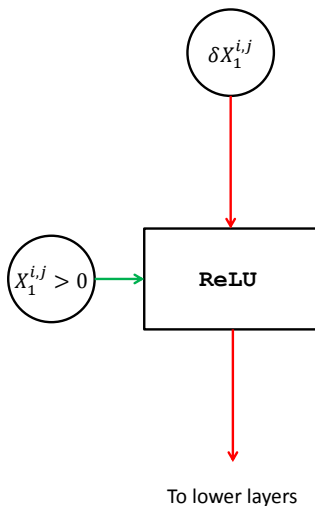
非常直观：

如果当前的 $i$ 层的第 $k$ 个神经元对第 $j$ 组数据是激活的 $X_k^{i,j} > 0$ ，那么就导通。让上层的反馈信号 $\delta X_k^{i+1,j}$ 通过。

不然，反馈信号就被截断于此。

# ReLU backward flow

注意反向不再是ReLU函数了，它是由 $X_k^{i,j}$ 控制的，输出是 $\delta X_k^{i+1,j}$ 或0.



# Backpropagation on Affine Layer

**Affine Layer**一方面负责将 $\delta X$ 向更低的层传递；一方面还要计算反馈给自己参数 $W, b$ 的梯度 $dW, db$ 。

## Affine Layer backward

$$\delta X_k^{i-1,j} = \sum_v W^i(v, k) \delta X_v^{i,j}$$

这是可以写成矩阵形式的：

$$\delta X^{i-1,j} = \delta X^{i,j} \cdot W^i$$

物理意义也非常直观：

正向是

$$X^{i,j} = W^i \cdot X^{i-1,j} + b^i$$

反向是转置（如果我们把 $\delta X^i$ 变成列向量）

$$(\delta X^{i-1,j})^T = (W^i)^T \cdot (\delta X^{i,j})^T$$

# Update rule on Affine Layer

## Affine Layer update rule on $W$

$$\begin{aligned}
 \frac{\partial \text{Obj}}{\partial W^i(u, v)} &= \sum_{j,k} \frac{\partial \text{Obj}}{\partial X_k^{i,j}} \frac{\partial X_k^{i,j}}{\partial W^i(u, v)} \\
 &= \sum_{j,k} \delta X_k^{i,j} \cdot (\delta_{k,u} \cdot X_v^{i-1,j}) \\
 &= \sum_j \delta X_u^{i,j} \cdot X_v^{i-1,j}
 \end{aligned}$$

## 物理意义

将反馈信号 $\delta X^i$ 按照当前层输入的激活度 $X^{i-1}$ 的比例传到该Layer的参数 $W^i$ 上。

我们下面试着把它写成矩阵形式得到更直观感受。

一般地，我们习惯上把数据写成矩阵时，每一行是一条记录。不同的列代表不同的**feature**，不同的行代表不同的数据样本。所以可以写成如下的形式。

# 数据矩阵排列约定和update rule矩阵形式

$\delta X^i$ 写成矩阵

$$\delta X^i = \begin{pmatrix} \delta X_1^{i,1} & \dots & \delta X_{h_i}^{i,1} \\ \delta X_1^{i,2} & \dots & \delta X_{h_i}^{i,2} \\ \vdots & & \vdots \\ \delta X_1^{i,B} & \dots & \delta X_{h_i}^{i,B} \end{pmatrix}$$

$X^{i-1}$ 写成矩阵

$$X^{i-1} = \begin{pmatrix} X_1^{i-1,1} & \dots & X_{h_{i-1}}^{i-1,1} \\ X_1^{i-1,2} & \dots & X_{h_{i-1}}^{i-1,2} \\ \vdots & & \vdots \\ X_1^{i-1,B} & \dots & X_{h_{i-1}}^{i-1,B} \end{pmatrix}$$

Affine Layer update rule: derivation

$$\begin{aligned} \left[ \frac{\partial \text{Obj}}{\partial W^i(u, v)} \right]_{u,v} &= \sum_j (\delta X^{i,j})^T \cdot X^{i-1,j} \\ &= (\delta X^i)^T \cdot X^{i-1} \end{aligned}$$

Matrix form

$$dW = (\delta X^i)^T \cdot X^{i-1}$$

# Update rule for $b$

不要忘了还有一部分参数在 $b$ 中

$$\frac{\partial \text{Obj}}{\partial b_k^i} = \sum_{j,u} \frac{\partial \text{Obj}}{\partial X_u^{i,j}} \frac{\partial X_u^{i,j}}{\partial b_k^i} = \sum_{j,u} \delta X_u^{i,j} \cdot \delta_{k,u} = \sum_j \delta X_k^{i,j}$$

矩阵形式

$$d b^i = (\delta X^i)^T \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

或写成

矩阵形式

$$d(b^i)^T = \mathbf{1}^T \cdot \delta X^i$$

# Affine Layer Summary

## 使用记号

$$\mathbf{1} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \quad X^{i-1} = \begin{pmatrix} X_1^{i-1,1} & \cdots & X_{h_{i-1}}^{i-1,1} \\ X_1^{i-1,2} & \cdots & X_{h_{i-1}}^{i-1,2} \\ \vdots & & \vdots \\ X_1^{i-1,B} & \cdots & X_{h_{i-1}}^{i-1,B} \end{pmatrix}$$

## Forward/Backward Matrix form

$$X^i = X^{i-1} \cdot (W^i)^T + \mathbf{1} \cdot (b^i)^T$$

$$\delta X^{i-1} = \delta X^i \cdot W^i$$

## Update rule for $W^i, b^i$

$$d(W^i)^T = (X^{i-1})^T \cdot \delta X^i$$

$$d(b^i)^T = \mathbf{1}^T \cdot \delta X^i$$

# AffineReLU Layer Summary

## 使用记号

$(X > 0)$  = elementwise boolean matrix

$A \odot B$  = elementwise matrix multiplication

## AffineReLU Forward/Backward Matrix form

$$X^i = (X^{i-1} \cdot (W^i)^T + \mathbf{1} \cdot (b^i)^T)_+$$

$$\delta X^{i-1} = (\delta X^i \odot (X^i > 0)) \cdot W^i$$

## AffineReLU Update rule for $W^i, b^i$

$$d(W^i)^T = (X^{i-1})^T \cdot (\delta X^i \odot (X^i > 0))$$

$$d(b^i)^T = \mathbf{1}^T \cdot (\delta X^i \odot (X^i > 0))$$



## 注: ReLU的意义

这里, ReLU虽然非常简单,但是又不可或缺! 么有它,再多Affine的复合还是Affine。我们的 $\mathcal{H}$ 就是全体Affine变换而已。然而平面上的Affine就是条线,它甚至无法学习XOR这个函数。

ReLU就是星星之火点燃了无限的可能性,让 $\mathcal{H}$ 丰满起来。

除了ReLU,我们也可以使用其他的非线性变换和Affine结合。比如

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

或ELU (Exponential Linear Units)

$$f(x) = \begin{cases} x & x > 0 \\ \alpha \cdot (e^x - 1) & x \leq 0 \end{cases}$$

或带参数 $w_j^k, b_j^k$ 的maxout layer

$$f_k(X) = \max_{1 \dots t} ((w_1^k)^T X + b_1^k, \dots, (w_t^k)^T X + b_t^k)$$

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# 为什么要用卷积

## 例子

考虑一个 $640 \times 480$ 的图像，每个像素是个输入。如果输出也是同样维度的图像，那么Affine变换需要 $849347481600 \approx 85\text{billion}$ 个参数。

然而，有意义的部分都聚成小块，比如 $10 \times 10$ 的块，当然它可能在原来图像的某个位置上。

如果我们的映射只是对这每个每个的小块做，那么就无需上面那么庞大的参数集了。正是这种想法把卷积引入图像处理。

下面约定输入输出的记号：

## 一条记录上的数据 $x_i$

上两节的一个数据点上的输入 $x_i$ 都是向量。这里的 $x_i$ 本身就有形状，对于有色的图像，它是三阶张量。记为 $X_{u,v}^c$ 。而这一节中Layer的输出也是三阶张量，记为 $Y_{u',v'}^{c'}$ 。

# 什么是卷积？

## 卷积算子

数学上，卷积用于从两个函数得到一个新函数

$$(f * g)(\tau) = \int f(x)g(\tau - x) \mathrm{d} x$$

## 平面卷积算子

如果 $A_{x,y}, B_{x,y}$ 是两个二阶张量，那么它们卷积我们约定为：

$$(A * B)_{u,v} = \sum_{i,j} A_{i,j} \cdot B_{u+i,v+j}$$

## 三维卷积算子

如果 $A_{x,y}^c, B_{x,y}^c$ 是两个三阶张量，那么它们卷积我们约定为：

$$(A \star B)_{u,v} = \sum_{i,j,k} A_{i,j}^k \cdot B_{u+i,v+j}^k$$

# 图像处理用的卷积层

## 卷积层 $f$ 的基本形式

$$Y^{c'} = f^{c'}(X) = K^{c'} \star X$$

写成分量形式:

$$Y_{u',v'}^{c'} = f^{c'}(X)_{u',v'} = \sum_{i=-d}^d \sum_{j=-d}^d \sum_{k=1}^{n_c} K(i,j)_k^{c'} \cdot X_{u'+i,v'+j}^k$$

## 解释

- $f$ 由一系列的 $f^{c'}$ 组成
- 每个 $f^{c'}$ 由 $K^{c'}$ 定义
- $K(i,j)_k^{c'}$ 被称为**Kernel**。指标 $i,j$ 是曲面指标, 指标 $k$ 对应输入**Channel**,  $c'$ 对应输出的**Channel**。它是个四阶张量, 就是CNN要学习的参数
- 我们约定上式的加项里, 如果 $u'+i, v'+j$ 越界 (小于0或大于输入的行列数), 那么这项不含入。

## 卷积层 $f$ 的完整形式

完整形式只需加上**bias**项:

$$Y^{c'} = K^{c'} \star X + b^{c'}$$

相应的分量形式:

$$Y_{u',v'}^{c'} = \sum_{i=-d}^d \sum_{j=-d}^d \sum_{k=1}^{n_c} K(i,j)_k^{c'} \cdot X_{u'+i,v'+j}^k + b^{c'}$$

使用 $d = 2$ , 对于开始的例子**Kernel**大小是 $(2d + 1)^2 n_c n_{c'}$ ,  
即 $5 \times 5 \times 3 \times 3$ , 我们只需要 $25 \times 9 + 1 = 226$ 个参数。

## 和Affine的关系

它其实可以看作**Affine**变换, 只是很多位置**weight**是0, 而非零的地方的**weight**被共享。**Kernel**就是这些被不同位置的输出所共享的**weight**。

这是向前的形式, 我们下面推导向后的公式。

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# Local gradient分量形式

## 向下层传递

$$\begin{aligned}\frac{\partial Y_{u',v'}^{c'}}{\partial X_{u,v}^c} &= \sum_{i,j,k} K(i,j)_k^{c'} \cdot \delta_{u'+i,u} \cdot \delta_{v'+j,v} \cdot \delta_{k,c} \\ &= K(u-u', v-v')_c^{c'} \cdot 1_{|u-u'| \leq d, |v-v'| \leq d}\end{aligned}$$

## 对 $K$ 的更新

$$\begin{aligned}\frac{\partial Y_{u',v'}^{c'}}{\partial K(i',j')_{k'}^c} &= \delta_{c,c'} \cdot \sum_{i,j,k} X_{u'+i,v'+j}^k \cdot \delta_{i,i'} \cdot \delta_{j,j'} \cdot \delta_{k,k'} \\ &= X_{u'+i',v'+j'}^{k'} \cdot \delta_{c,c'}\end{aligned}$$

## 对 $b$ 的更新

$$\frac{\partial Y_{u',v'}^{c'}}{\partial b^c} = \delta_{c,c'}$$



# 卷积层BP

为了减少符号混乱，层的指标使用希腊字母 $\iota$ 写到括号里。

第 $\iota$ 层的参数 $K(\iota), b(\iota)$

$$X(\iota)_{u,v}^c = \sum_{i,j,k} K(\iota)(i,j)_k^c X(\iota-1)_{u+i,v+i}^k + b(\iota)^c$$

Loss到 $\iota$ 反馈

$$\delta X(\iota)_{u,v}^c = \frac{\partial l}{\partial X(\iota)_{u,v}^c}$$

$\delta X(\iota)$ 到 $\delta X(\iota-1)$ 的反向公式

$$\begin{aligned} \delta X(\iota-1)_{u,v}^c &= \frac{\partial l}{\partial X(\iota-1)_{u,v}^c} = \sum_{u',v',c'} \frac{\partial l}{\partial X(\iota)_{u',v'}^{c'}} \cdot \frac{\partial X(\iota)_{u',v'}^{c'}}{\partial X(\iota-1)_{u,v}^c} \\ &= \sum_{u',v',c'} \delta X(\iota)_{u',v'}^{c'} \cdot K(\iota)(u-u', v-v')_c^{c'} \\ &= \sum_{i=-d}^d \sum_{j=-d}^d \sum_{k=1}^{n_{c'}(\iota)} K(\iota)(i,j)_c^k \cdot \delta X(\iota)_{u-i,v-j}^k \end{aligned}$$

引入 $\tilde{K}(\iota)(i, j)_k^c = K(\iota)(-i, -j)_c^k$ , 即新张量 $\tilde{K}$ 是原来 $K$ 张量上两个指标的镜像, 另两个指标的转置, 那么

## 卷积层的BP

$$\delta X(\iota - 1)_{u,v}^c = \sum_{i=-d}^d \sum_{j=-d}^d \sum_{k=1}^{n_c(\iota)} \tilde{K}(\iota)(i, j)_k^c \cdot \delta X(\iota)_{u+i, v+j}^k$$

即

$$\delta X(\iota - 1)^c = \tilde{K}(\iota)^c \star \delta(\iota)$$

反向仍然是卷积变换, 只是Kernel变成了 $\tilde{K}(i, j)_c^c$ 。可以对比正向卷积:

## 卷积层的正向传播

$$X(\iota)^{c'} = K(\iota)^{c'} \star X(\iota - 1) + b(\iota)^{c'}$$

分量形式:

$$X(\iota)_{u',v'}^{c'} = \sum_{i=-d}^d \sum_{j=-d}^d \sum_{k=1}^{n_c(\iota)} K(\iota)(i, j)_k^{c'} \cdot X(\iota - 1)_{u'+i, v'+j}^k + b(\iota)^{c'}$$

# Weigth Update

卷积层的Weigth Update: d K张量

$$\begin{aligned}
 dK(\iota)(i', j')_{k'}^k &= \frac{\partial l}{\partial K(\iota)(i', j')_{k'}^k} = \sum_{u', v', c'} \frac{\partial l}{\partial X(\iota)_{u', v'}^{c'}} \cdot \frac{\partial X(\iota)_{u', v'}^{c'}}{\partial K(\iota)(i', j')_{k'}^k} \\
 &= \sum_{u', v', c'} \delta X(\iota)_{u', v'}^{c'} \cdot X(\iota - 1)_{u' + i', v' + j'}^{k'} \cdot \delta_{k, c'} \\
 &= \sum_{u', v'} \delta X(\iota)_{u', v'}^k \cdot X(\iota - 1)_{i' + u', j' + v'}^{k'}
 \end{aligned}$$

即 $\delta X(\iota)_{u', v'}^k$ 为Kernel的二维卷积 $dK(\iota)_{k'}^k = \delta X(\iota)^k * X(\iota - 1)^{k'}$

卷积层的Weigth Update: d b张量

$$db(\iota)^{c'} = \frac{\partial l}{\partial b(\iota)^{c'}} = \sum_{u'', v'', c''} \frac{\partial l}{\partial X(\iota)_{u'', v''}^{c''}} \cdot \frac{\partial X(\iota)_{u'', v''}^{c''}}{\partial b(\iota)^{c'}} = \sum_{u', v'} \delta X(\iota)_{u', v'}^{c'}$$

# Summary

回忆我们用 $\star$ 记录三维卷积，用 $*$ 记录二维卷积；前面公式写作：

正向

$$X(\iota)^{c'} = K(\iota)^{c'} \star X(\iota - 1) + b(\iota)^{c'}$$

反向

$$\delta X(\iota - 1)^c = \tilde{K}(\iota)^c \star \delta X(\iota)$$

update for  $K$

$$d K(\iota)_{k'}^k = \delta X(\iota)^k * X(\iota - 1)^{k'}$$

update for  $b$

$$d b(\iota)^c = \mathbf{1}^T \text{vec}(\delta X(\iota)^c)$$

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# Feed forward with stride

回到最简单的卷积算子

$$Y_{u',v'}^{c'} = \sum_{i,j,k} K(i,j)_k^{c'} \cdot X_{u'+i,v'+j}^k$$

这里输入尺寸和输出尺寸是一样的。

如果输出的尺寸 $u', v'$ 和输入尺寸不同，那么就要引入一个不同的**stride**（相当于对图像缩放） $s$ ：

带有**stride**的卷积层

$$Y_{u',v'}^{c'} = \sum_{i,j,k} K(i,j)_k^{c'} \cdot X_{u' \cdot s + i, v' \cdot s + j}^k$$

所以 $s$ 物理意义是，输出上每移动一个单位，输入的采样中心需要移动 $s$ 个单位。如果原图是 $n_x \times n_y$ ，那么输出是 $\frac{n_x}{s} \times \frac{n_y}{s}$ 尺寸的。

# stride local gradient and BP

## local gradient

$$\begin{aligned}\frac{\partial Y_{u',v'}^{c'}}{\partial X_{u,v}^c} &= \sum_{i,j,k} K(i,j)_k^{c'} \cdot \delta_{u's+i,u} \cdot \delta_{v's+j,v} \cdot \delta_{k,c} \\ &= K(u-u's, v-v's)_c^{c'}\end{aligned}$$

## BP

$$\begin{aligned}\delta X(\ell-1)_{u,v}^c &= \sum_{u',v',c'} \delta X(\ell)_{u',v'}^{c'} \cdot K(\ell)(u-u's, v-v's)_c^{c'} \\ &= \sum_{i=-d}^d \sum_{j=-d}^d \sum_{k=1}^{n_c(\ell)} K(\ell)(i,j)_c^k \cdot \delta X(\ell)_{\lfloor \frac{u-i}{s} \rfloor, \lfloor \frac{v-j}{s} \rfloor}^k\end{aligned}$$

# stride下和最简形式的联系

如果引入记号

 $\tilde{K}$ 

$$\tilde{K}(\iota)(i, j)_k^c = K(\iota)(-i, -j)_c^k$$

 $\widetilde{\delta X}$ 

$$\widetilde{\delta X}(\iota)^k = \delta X(\iota)^k \otimes (\mathbf{1} \cdot \mathbf{1}^T)$$

这里 $\mathbf{1} \cdot \mathbf{1}^T$ 是 $s \times s$ 阶全1的常数矩阵。右边Kronecker乘积操作相当于把反馈信号尺寸放大 $s$ 倍—原来每个位置用 $s \times s$ 的常数块代替。那么反向还是可以写成普通（3维）的卷积

BP with stride

$$\delta X(\iota - 1)^c = \tilde{K}(\iota)^c \star \widetilde{\delta X}(\iota)$$



# stride下对 $K$ 的更新

## Local gradient, no stride

$$\frac{\partial Y_{u',v'}^{c'}}{\partial K(i',j')_{k'}^c} = X_{u'+i',v'+j'}^{k'} \cdot \delta_{c,c'}$$

只要在 $X$ 相关的指标中用 $u's$ 代替 $u'$ ，用 $v's$ 代替 $v'$

## Local gradient, with stride $s$

$$\frac{\partial Y_{u',v'}^{c'}}{\partial K(i',j')_{k'}^c} = X_{u's+i',v's+j'}^{k'} \cdot \delta_{c,c'}$$

## 卷积层的Weight Update, $dK$ 张量, with stride $s$

$$\begin{aligned} dK(\iota)(i',j')_{k'}^c &= \sum_{u',v'} \delta X(\iota)_{u',v'}^{k'} \cdot X(\iota-1)_{i'+u's,j'+v's}^{k'} \\ &= \sum_{i,j} X(\iota-1)_{i,j}^{k'} \cdot \delta X(\iota)_{\lfloor \frac{i-i'}{s} \rfloor, \lfloor \frac{j-j'}{s} \rfloor}^c \end{aligned}$$

# padding

而padding是为了处理边界问题。例如：当 $(u', v') = (0, 0)$ 时，对 $i < 0, j < 0$ 的指标，相当于pad了 $d$ 层0。

最简形式已经pad了 $d$ 层

$$X_{(-d:-1),:} = X_{:,-d:-1) = X_{(n_x:n_x+d),:} = X_{:,(n_y:n_y+d)} = 0$$

如果输出中不希望有padding成分，可以slice一个子块

$$Y_{d:(n_x-d-1),d:(n_y-d-1)}^{c'}$$

## 一般padding

如果我们希望pad其他尺寸 $p$ ，那么可以用

$$Y_{u',v'}^{c'} = \sum_{i,j,k} K(i,j)_k^{c'} \cdot X_{u'+i+(p-d),v'+j+(p-d)}^k$$

的公式。

它的相应公式可以类似推导。

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - **Pooling层**
- 6 Loss层的梯度

# Pooling Layer

上面我们看到使用不同**stride**可以改变图像大小。有时候需要一个强制的缩放，而不想要一个卷积层。这就是所谓的**Pooling**。假设我们把原来的 $s \times s$ 的小块缩成一个像素点，新像素点的强度取原来小块的最大值，就得到**max pooling**。

## max pooling

$$X(\iota)_{u',v'}^{c'} = \max_{i,j \in \{0, \dots, s-1\}} X(\iota-1)_{u's+i, v's+j}^{c'}$$

这是个**non-linear**映射

当然，另一个自然的想法是用小块的平均值作为缩小后的图的像素点的值。

## mean pooling

$$\begin{aligned} X(\iota)_{u',v'}^{c'} &= \text{Average}_{i,j \in \{0, \dots, s-1\}} (X(\iota-1)_{u's+i, v's+j}^{c'}) \\ &= \frac{1}{s^2} \mathbf{1}^T X(\iota-1)_{u's:(u's+s-1), v's:(v's+s-1)}^{c'} \mathbf{1} \end{aligned}$$

是**linear**的

# Local gradient for Pooling Layer

## gradient of max

在可以求导的点上偏导是  $\frac{\partial \max(X)}{\partial X_i} = \delta_{i, \arg\max(X)}$ , 所以

$$\nabla_X \max(X) = e_{\arg\max(X)}^T$$

## BP of max pooling

$$\delta X(\ell-1)_{u,v}^c = \begin{cases} \delta X(\ell)_{\lfloor \frac{u}{s} \rfloor, \lfloor \frac{v}{s} \rfloor}^c & X(\ell-1)_{u,v}^c = X(\ell-1)_{\lfloor \frac{u}{s} \rfloor, \lfloor \frac{v}{s} \rfloor}^c \\ 0 & \text{otherwise} \end{cases}$$

也就是取最大值处“导通”传回，其他地方阻断。

## mean pooling

$$\delta X(\ell-1)_{u,v}^c = \frac{1}{s^2} \delta X(\ell)_{\lfloor \frac{u}{s} \rfloor, \lfloor \frac{v}{s} \rfloor}^c$$

信号强度会变小  $s^2$  倍，所以实际中这种 pooling 不常用

# 目录

- 1 背景和大纲
- 2 ML和数学的复习
  - Machine Learning超简单的简介
  - 求导的复习
  - 梯度和张量, Jacobian和Hessian
- 3 反向传播算法
  - 神经网络概念
  - 一个数据点上的BP
  - Batched BP
- 4 具体的Layer: Affine和ReLU层
  - Affine和ReLU层定义
  - Affine和ReLU层的Jacobian
  - Affine和ReLU层的BP
- 5 卷积神经网络
  - 卷积运算
  - 卷积层BP
  - padding和stride
  - Pooling层
- 6 Loss层的梯度

# 最后一块拼图

## 平方误差

$$\delta S = \frac{d}{ds}(y - s)^2 = 2(s - y)$$

## Logistic 0/1

$$\delta S = \frac{d}{ds}L = \varphi(s) - y$$

## Logistic $\pm 1$

$$\delta S = \frac{d}{ds}L = -y \cdot \varphi(-y \cdot s)$$

# 最后一块拼图

用 $e_i$ 表示 $\mathbb{R}^m$ 的标准正交基（也称为one hot向量）

$$e_i = \underbrace{(0, \dots, 1, \dots, 0)^T}_{\text{i-th position is 1, other are zero}} \in \mathbb{R}^m$$

## Cross Entropy

引入记号 $\mathbf{p}$ 代表softmax给出的概率分布：

$$\mathbf{p} = \left( \frac{e^{s_1}}{\sum_j e^{s_j}}, \dots, \frac{e^{s_m}}{\sum_j e^{s_j}} \right)$$

用 $\mathbf{y}$ 代表 $y$ 给出的概率分布 $\mathbf{y} = e_y^T$ ，那么

$$\delta S = \nabla_S L = \mathbf{p} - \mathbf{y}$$



# 最后一块拼图

## SVM loss

用 $\Delta$ 记那些得分加上margin后比正确的类 $y$ 的得分还要高的类，那么

$$\delta S = \nabla_S L = \sum_{j \in \Delta} e_j^T - |\Delta| \cdot e_y^T$$

## Cross Entropy和SVM loss的物理意义

$$\delta S_{\text{Cross Entropy}} = (p_1, p_2, \dots, p_y - 1, \dots, p_m)$$

$$\delta S_{\text{SVM loss}} = (0, 1, \dots, 1_{\text{position} \in \Delta}, \dots, -|\Delta|, \dots, 0)$$

都是错误的方向上要减一个大的正数，正确方向上减一个大的负数。

# 终于告一段落啦



“雷姆雷姆，  
好像真的不难  
耶。。。”



“姐姐姐姐，  
数学-好有趣  
呢！”

# 下周再见



“今天讲了

- BP算法
- 具体公式的推导
- 卷积层

有点抽象哦”

“下次我们是要  
看看它们如何代  
码化的，对吗？  
真期待。。。 ”

