第二部分 现代实践

- 分总结现代深度学习用于解决实际应用的现状
- 仅关注那些基本上已在工业中大量使用的技术方法

现代深度学习为监督学习提供了一个强大的框架：

- 添加更多层以及向层内添加更多单元，深度网络可以表示复杂性不断增加的函数
- 给定足够大的模型和足够大的标注训练数据集，深度学习可以通过深度学习将输入向量映射到输出向量，完成大多数对人来说能迅速处理的任务

这一部分描述参数化函数近似技术的核心：

- 前馈深度网络
- 正则化和优化
- 卷积网络
- 循环和递归神经网络
- 实践方法论和应用

第六章 深度前馈网络

前馈神经网络

- 多层感知机（MLP）
- 目标是寻找最佳函数近似$\mathrm{y} = f^*(\boldsymbol{x}; \boldsymbol{\theta})$
- 学习参数θ的值
- 模型的输出和模型本身之间没有反馈连接
- 链式结构：$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right))$
- 标签：$y \approx f^*(\boldsymbol{x})$

训练数据

- 直接指明了输出层必须产生一个接近$\boldsymbol{y}$的值
- 没有直接指明其他层应该怎么做：由学习算法决定

设计决策

- 选择一个优化模型、代价函数以及输出单元的形式
- 选择用于计算隐藏层值的激活函数
- 设计网络的结构

第六章 深度前馈网络

前馈神经网络

- 为了实现统计泛化
- 为了扩展线性模型来表示$\boldsymbol{x}$的非线性函数
- 用在一个变换后的输入$\boldsymbol{\phi}(\boldsymbol{x})$上

如何选择$\boldsymbol{\phi}(\boldsymbol{x})$

- 选择一个通用的$\boldsymbol{\phi}$
  - 拟合训练数据能力强
  - 泛化能力差、没有将足够的先验信息进行编码来解决高级问题
- 手动设计一个$\boldsymbol{\phi}$
  - 工作量大、不同领域之间很难迁
- 学习$\boldsymbol{\phi}$
  - 模型：$y = f(\boldsymbol{x};\theta,\boldsymbol{w}) = \boldsymbol{\phi}(\boldsymbol{x};\theta)^T\boldsymbol{w}$，学习得到$\boldsymbol{\theta},\boldsymbol{w}$
  - 放弃了训练问题的凸性
  - 但是兼顾1,2的优点

6.1 实例：学习XOR

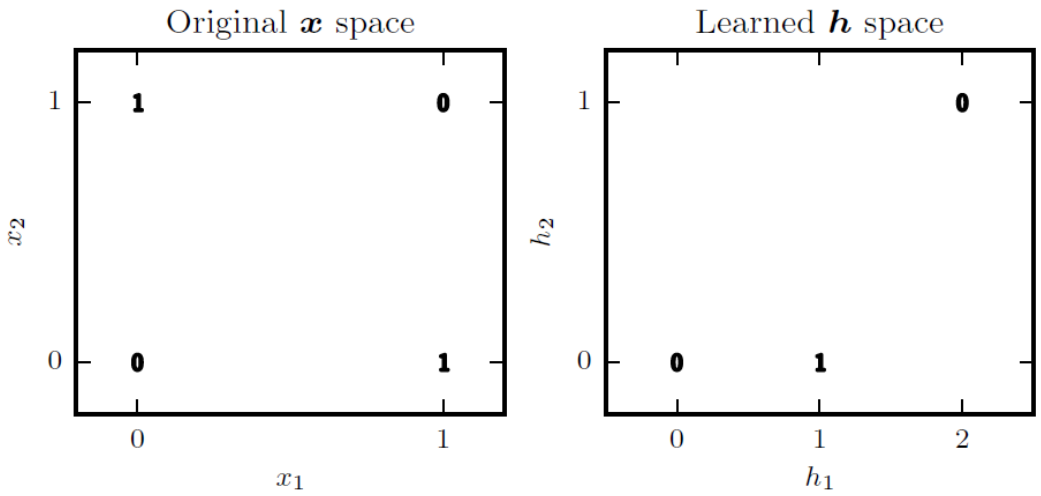
看作回归问题

- 损失函数：均方误差 $J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{x\in\mathbb{X}}(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta}))^2$
- 模型：$f(\boldsymbol{x};\boldsymbol{w},b) = \boldsymbol{x}^\top\boldsymbol{w} + b$
- 正规方程求解：$w = 0$，$\mathrm{b} = \frac{1}{2}$
- 显然，不正确。线性模型仅仅是在任意一点都输出0.5

解决方案

- 必须用非线性函数来描述这些特征，使用一个模型来学习一个不同的特征空间，引入一个非常简单的前馈神经网络
- 这个前馈网络有一个通过函数$f^{(1)}(\boldsymbol{x};\boldsymbol{W},c)$计算得到的隐藏单元的向量$\boldsymbol{h}$

6.1 实例：学习XOR



网络包含链接在一起的两个函数：

- $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, c)$
- $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$
- 完整模型：$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$
- 定义非线性函数：$g(z) = \max(0, z)$, $\boldsymbol{h} = g(\boldsymbol{W}^T \boldsymbol{x} + c)$
- 于是：$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b$

6.2 基于梯度的学习

代价函数的特点

- 神经网络的非线性导致大多数代价函数都变得非凸
- 使用迭代的、基于梯度的优化
- 随机梯度下降，没有收敛性保证，并且对参数的初始值很敏感
- 初始化为小随机数是十分重要的

代价函数的选择

- 使用最大似然原理：
  - 训练数据和模型预测间的交叉熵
- 更简单的方法
  - 仅仅预测在给定$\boldsymbol{x}$的条件下$\boldsymbol{y}$的某种统计量

完整代价函数：

- 完整代价函数 = 基本代价函数 + 正则项

输出单元：

- 任何可用作输出的神经网络单元，也可以被用作隐藏单元

6.2 基于梯度的学习

最大似然原理：

- 代价函数就是负的对数似然：$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$
- 代价函数的具体形式取决于$logp_{model}$的具体形式
- 优势：
  - 减轻了为每个模型设计代价函数的负担
  - 避免隐藏层单元或者输出单元输出的激活函数饱和
- 劣势：
  - 交叉熵代价函数时通常没有最小值

6.2 基于梯度的学习

学习条件统计量：

- 可以把代价函数看作是一个泛函
- 学习过程就被看作选择一个函数而不仅仅是选择一组参数
- 使用变分法导出两个结果：
  - 均方误差：$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2$
  - 平均绝对误差：$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||_1$
- 优势：
  - 仅仅预测在给定$\boldsymbol{x}$的条件下$\boldsymbol{y}$的某种统计量
- 劣势：
  - 在使用基于梯度的优化方法时往往成效不佳，会饱和

6.2 基于梯度的学习

输出单元：

- 用于高斯输出分布的线性单元
- 用于Bernoulli 输出分布的sigmoid 单元
- 用于Multinoulli输出分布的softmax单元
- 其他的输出类型

6.2 基于梯度的学习

用于高斯输出分布的线性单元

- 给定特征$\boldsymbol{h}$，线性输出单元层产生一个向量$\hat{y} = \boldsymbol{W}^T\boldsymbol{h} + b$
- 线性输出层经常被用来产生条件高斯分布的均值

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$$

- 最大化其对数似然比时等价于最小化均方误差
- 对于所有输入，协方差矩阵必须被限定为正定矩阵
- 线性模型不会饱和

6.2 基于梯度的学习

用于Bernoulli 输出分布的sigmoid 单元

- 预测二值型变量y的值
- Bernoulli 分布仅需单个参数，但是预测值必须处在[0,1]之间
- 对于一个有效的条件概率分布，需要解决优化中梯度为0的问题
- 使用sigmoid 输出单元结合最大似然
- sigmoid 输出单元定义为：$\hat{y} = \sigma\left(\boldsymbol{w}^{\top}\boldsymbol{h} + b\right)$
- 两个部分：
  - 线性层计算$w^T h + b$
  - sigmoid 激活函数将其转化为概率
- 损失函数为softplus函数，不会收缩梯度

6.2 基于梯度的学习

用于Multinoulli输出分布的softmax单元

- 线性层预测了未归一化的对数概率： $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$
- 其中， $z_i = \log \hat{P}(y = i \mid \boldsymbol{x})$
- softmax 函数然后可以对z指数化和归一化来获得需要结果
- 最终形式为： $\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$
- $\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j)$

6.2 基于梯度的学习

其他的输出类型：

- 最大似然原则给如何为几乎任何种类的输出层设计一个好的代价函数提供了指导
- 一般的，定义一个条件分布$p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$，最大似然原则建议损失函数为-$logp(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$
- 例如，学习某个条件高斯分布的方差，使用精度β而不是方差来表示高斯分布
- 涉及到除法（除以方差）可能是数值不稳定的：
  - 梯度截断；
  - 启发式缩放梯度（Murray and Larochelle, 2014）

6.3 隐藏单元

- 如何选择隐藏单元的类型是前馈神经网络独有的问题
- 整流线性单元（ReLU）是默认选择
- 其他类型的隐藏单元
- 决定何时使用哪种类型的隐藏单元是困难的事
  - 不可能预先预测出哪种隐藏单元工作得最好
  - 先直觉认为某种隐藏单元可能表现良好，最后用验证集来评估

6.3 隐藏单元

整流线性单元及其扩展：

- 整流线性单元使用激活函数$g(z) = \max\{0, z\}$
- 优点：
  - 处于激活状态时，它的导数都能保持较大
  - 二阶导数几乎处处为0
- 一个缺陷：
  - 不能通过基于梯度的方法学习那些使它们激活为零的样本
- 扩展基于的原则：
  - 如果它们的行为更接近线性，那么模型更容易优化

6.3 隐藏单元

整流线性单元及其扩展：

- 三个扩展基于$g(z,\alpha)_i = \max\{0, z_i\} + \alpha_i \min\{0, z_i\}$
  - 绝对值整流：固定$\alpha_i = -1$
  - 渗漏整流线性单元：固定α_i一个类似0.01 的小值
  - 参数化整流线性单元或者PReLU 将α_i作为学习的参数
- maxout 单元：$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$
  - 可以学习具有多达k段的分段线性的凸函数
  - 学习激活函数本身而不仅仅是单元之间的关系
  - 要求更少的参数可以获得一些统计和计算上的优点
  - 抵抗灾难遗忘

6.3 隐藏单元

logistic sigmoid与双曲正切函数：

- logistic sigmoid 激活函数 $g(z) = \sigma(z)$
- 双曲正切激活函数 $g(z) = \tanh(z)$
- 两者之间的关系：$\tanh(z) = 2\sigma(2z) - 1$
- sigmoid 单元的广泛饱和性会使得基于梯度的学习变得非常困难
- 当必须要使用sigmoid 激活函数时，双曲正切激活函数通常要比 logistic sigmoid 函数表现更好

6.3 隐藏单元

其他隐藏单元：

- 都不常用
- 完全没有激活函数，或者单位函数
  - 线性隐藏单元因此提供了一种减少网络中参数数量的有效方法
- softmax 单元是另外一种经常用作输出的单元
- 径向基函数 $h_i = \exp\left(-\frac{1}{\sigma_i^2}||\boldsymbol{W}_{:,i} - \boldsymbol{x}||^2\right)$
- softplus函数 $g(a) = \zeta(a) = \log(1 + e^a)$
- 硬双曲正切函数 $g(a) = \max(-1, \min(1, a))$

6.4 架构设计

两个关键问题：

- 神经网络应该具有多少单元，以及这些单元应该如何连接
- 如何将层与层之间连接起来

大多数神经网络架构将这些层布置成链式结构，其中每一层都是前一层的函数

- 第一层 $\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$
- 第二层 $\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$
- 以此类推

链式架构中，主要的架构考虑是选择网络的深度和每一层的宽度

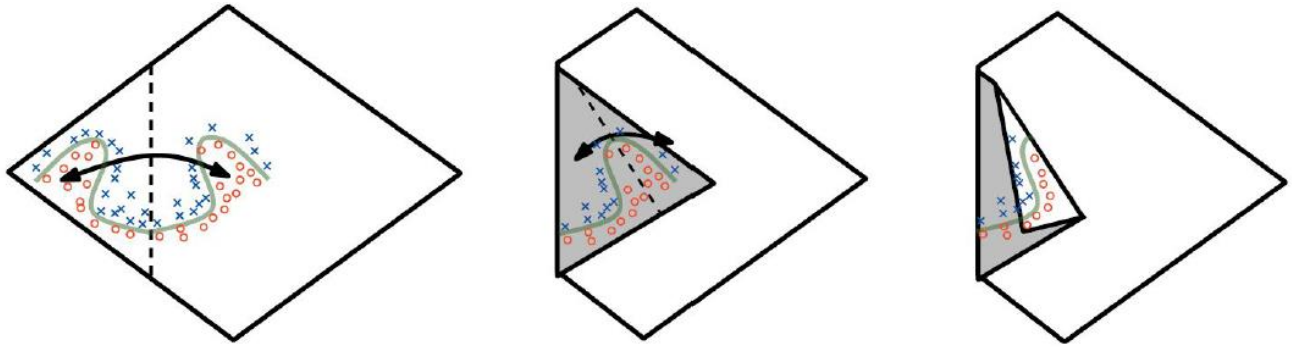
6.2 基于梯度的学习

万能近似性质和深度：

- 对于我们想要实现的目标，只需要知道定义在$\mathbb{R}^n$的有界闭集上的任意连续函数是Borel 可测的
  - 无论试图学习什么函数，一个大的MLP 一定能够表示这个函数
- 学习也可能因两个不同的原因而失败
  - 用于训练的优化算法可能找不到用于期望函数的参数值
  - 训练算法可能由于过拟合而选择了错误的函数
- 总之，具有单层的前馈网络足以表示任何函数，但是网络层可能大得不可实现，并且可能无法正确地学习和泛化

6.2 基于梯度的学习

万能近似性质和深度：



- 关于更深的整流网络具有指数优势的一个直观的几何解释
- 还可能出于统计原因来选择深度模型
  - 当我们选择一个特定的机器学习算法时，我们隐含地陈述了一些先验，这些先验是关于算法应该学得什么样的函数的
- 根据经验，更深的模型似乎确实在广泛的任务中泛化得更好

6.5 反向传播和其他的微分算法

前向传播

- 接收输入x并产生输出$\hat{y}$
- 训练过程中，前向传播可以持续向前直到产生一个代价函数$J(\theta)$

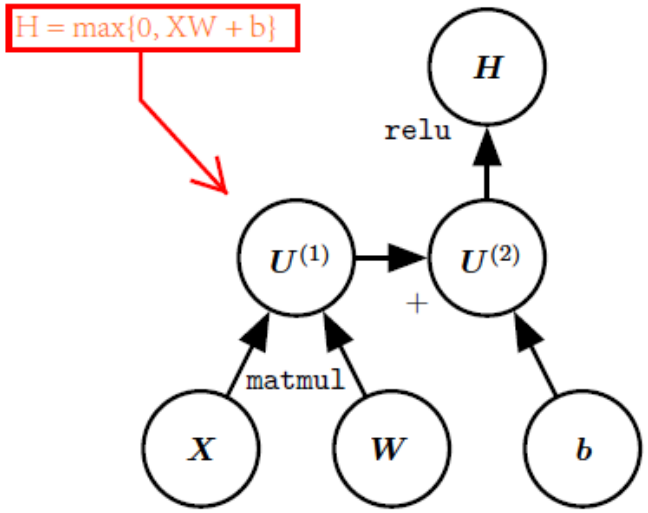
反向传播：

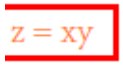
- 允许来自代价函数的信息通过网络向后流动，一边计算梯度
- 反向传播仅指用于计算梯度的方法

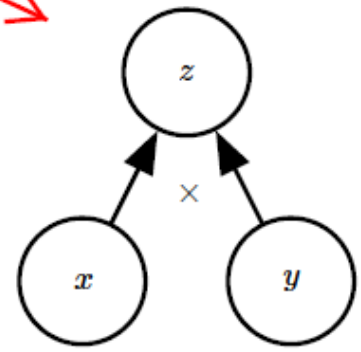
6.5 反向传播和其他的微分算法

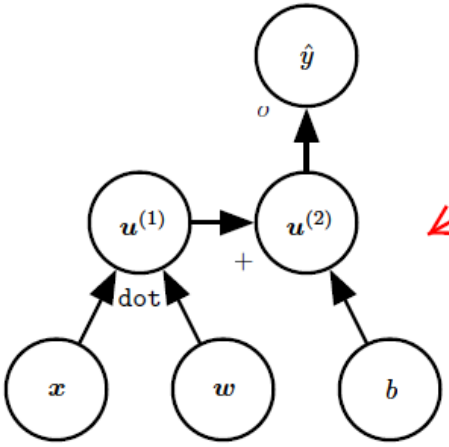
计算图

- 为了更精确地描述反向传播算法
- 使用图中的每一个节点来表示一个变量
- 操作是指一个或多个变量的简单函数

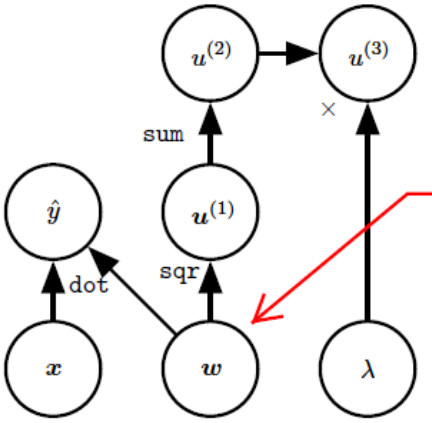








逻辑回归



对变量实施多个操作

6.5 反向传播和其他的微分算法

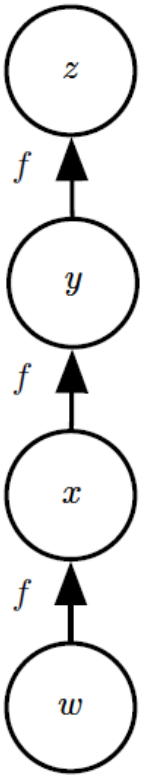
微积分中的链式法则

- 链式法则用于计算复合函数的导数
- $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$ 扩展到向量 $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$，等价于 $\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z$
- 应用于任意维度的张量，只是将Jacobian 乘以梯度
- $\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}.$

6.5 反向传播和其他的微分算法

递归地使用链式法则来实现反向传播

- 实际计算中，许多子表达式在梯度的整个表达式中重复若干次



$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$

6.5 反向传播和其他的微分算法

递归地使用链式法则来实现反向传播

- 前向传播算法

算法 6.1 计算将 n_i 个输入 $u^{(1)}$ 到 $u^{(n_i)}$ 映射到一个输出 $u^{(n)}$ 的程序。这定义了一个计算图，其中每个节点通过将函数 $f^{(i)}$ 应用到变量集合 $\mathbb{A}^{(i)}$ 上来计算 $u^{(i)}$ 的值，$\mathbb{A}^{(i)}$ 包含先前节点 $u^{(j)}$ 的值满足 $j < i$ 且 $j \in Pa(u^{(i)})$。计算图的输入是向量 $\boldsymbol{x}$，并且被分配给前 n_i 个节点 $u^{(1)}$ 到 $u^{(n_i)}$。计算图的输出可以从最后一个（输出）节点 $u^{(n)}$ 读出。

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

模型的参数

父节点

单个或者小批量实例的代价函数

6.5 反向传播和其他的微分算法

递归地使用链式法则来实现反向传播

- 反向传播算法

算法 6.2 反向传播算法的简化版本，用于计算 $u^{(n)}$ 关于图中变量的导数。这个示例旨在通过演示所有变量都是标量的简化情况来进一步理解反向传播算法，这里我们希望计算关于 $u^{(1)}, \ldots, u^{(n_i)}$ 的导数。这个简化版本计算了关于图中所有节点的导数。假定与每条边相关联的偏导数计算需要恒定的时间的话，该算法的计算成本与图中边的数量成比例。这与前向传播的计算次数具有相同的阶。每个 $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ 是 $u^{(i)}$ 的父节点 $u^{(j)}$ 的函数，从而将前向图的节点链接到反向传播图中添加的节点。

运行前向传播（对于此例是算法 6.1）获得网络的激活。

初始化 `grad_table`，用于存储计算好的导数的数据结构。`grad_table`$[u^{(i)}]$ 将存储 $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ 计算好的值。

自己对自己的导数为1

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

下一行使用存储的值计算 $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$：

`grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

6.5 反向传播和其他的微分算法

全连接MLP 中的反向传播计算

- 前向传播

算法 6.3 典型深度神经网络中的前向传播和代价函数的计算。损失函数 $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ 取决于输出 $\hat{\boldsymbol{y}}$ 和目标 $\boldsymbol{y}$（参考第 6.2.1.1 节中损失函数的示例）。为了获得总代价 J，损失函数可以加上正则项 $\Omega(\theta)$，其中 θ 包含所有参数（权重和偏置）。算法 6.4 说明了如何计算 J 关于参数 $\boldsymbol{W}$ 和 $\boldsymbol{b}$ 的梯度。为简单起见，该演示仅使用单个输入样本 $\boldsymbol{x}$。实际应用应该使用小批量。请参考第 6.5.7 节以获得更加真实的演示。

Require: 网络深度，l

Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$，模型的权重矩阵

Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$，模型的偏置参数

Require: $\boldsymbol{x}$，程序的输入

Require: $\boldsymbol{y}$，目标输出

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$

for $k = 1, \ldots, l$ **do**

$\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$

$\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$

end for

$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$

$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

本层的偏置值和权重矩

上一层的输出

相应的激活函数

本层的输出

6.5 反向传播和其他的微分算法

全连接MLP 中的反向传播计算

- 反向传播

算法 6.4 深度神经网络中算法6.3的反向计算，它不止使用了输入 $\boldsymbol{x}$ 和目标 $\boldsymbol{y}$。该计算对于每一层 k 都产生了对激活 $\boldsymbol{a}^{(k)}$ 的梯度，从输出层开始向后计算一直到第一个隐藏层。这些梯度可以看作是对每层的输出应如何调整以减小误差的指导，根据这些梯度可以获得对每层参数的梯度。权重和偏置上的梯度可以立即用作随机梯度更新的一部分（梯度算出后即可执行更新），或者与其他基于梯度的优化方法一起使用。

在前向计算完成后，计算顶层的梯度：

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

将关于层输出的梯度转换为非线性激活输入前的梯度（如果 f 是逐元素的，则逐元素地相乘）：

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

计算关于权重和偏置的梯度（如果需要的话，还要包括正则项）：

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

关于下一更低层的隐藏层传播梯度：

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

6.5 反向传播和其他的微分算法

符号到符号的导数

- 代数表达式和计算图都对符号或不具有特定值的变量进行操作
- 符号到数值的微分
  - 采用计算图和一组用于图的输入的数值，然后返回在这些输入值处梯度的一组数值
  - Torch (Collobert et al., 2011b)和Caffe (Jia, 2013)
- 符号到符号的微分
  - 采用计算图以及添加一些额外的节点到计算图中，这些额外的节点提供了我们所需导数的符号描述
  - Theano (Bergstra *et al.*, 2010b; Bastien*et al.*, 2012b)和Tensorlow (Abadi *et al.*, 2015)

6.5 反向传播和其他的微分算法

符号到符号的导数

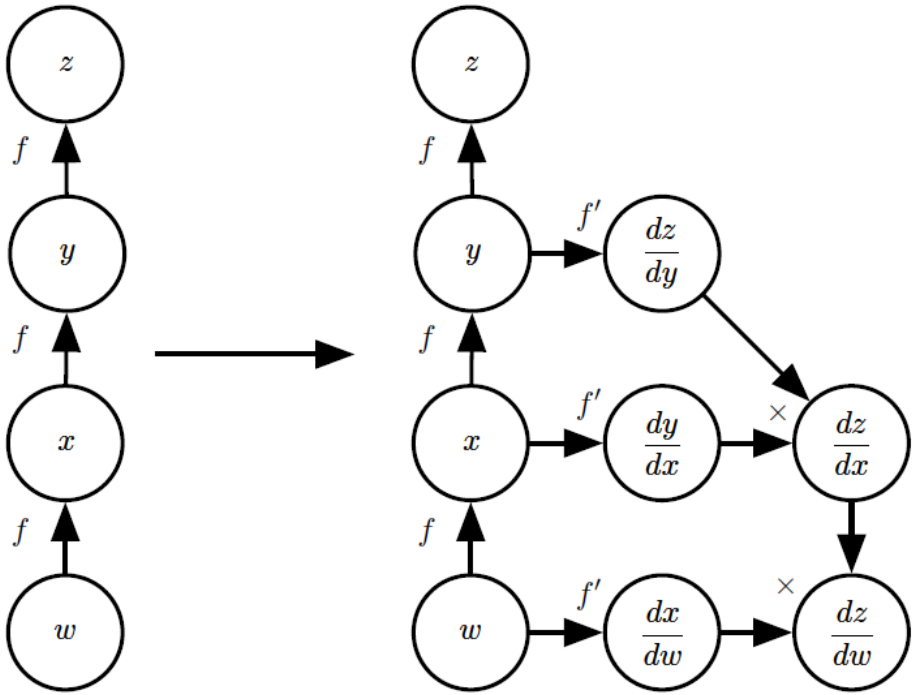


图 6.10: 使用符号到符号的方法计算导数的示例。在这种方法中，反向传播算法不需要访问任何实际的特定数值。相反，它将节点添加到计算图中来描述如何计算这些导数。通用图形求值引擎可以在随后计算任何特定数值的导数。*(左)* 在这个例子中，我们从表示 $z = f(f(f(w)))$ 的图开始。*(右)* 我们运行反向传播算法，指导它构造表达式 $\frac{dz}{dw}$ 对应的图。在这个例子中，我们不解释反向传播算法如何工作。我们的目的只是说明想要的结果是什么：符号描述的导数的计算图。

6.5 反向传播和其他的微分算法

一般化的反向传播

- 最外围的框架

算法 6.5 反向传播算法最外围的骨架。这部分做简单的设置和清理工作。大多数重要的工作发生在算法 6.6 的子程序 build_grad 中。

Require: $\mathbb{T}$，需要计算梯度的目标变量集

Require: $\mathcal{G}$，计算图

Require: z，要微分的变量

令 $\mathcal{G}'$ 为 $\mathcal{G}$ 剪枝后的计算图，其中仅包括 z 的祖先以及 $\mathbb{T}$ 中节点的后代。

初始化 grad_table，它是关联张量和对应导数的数据结构。

grad_table$[z] \leftarrow 1$

for $\mathbf{V}$ in $\mathbb{T}$ **do**

  build_grad$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \text{grad_table})$

end for

Return grad_table restricted to $\mathbb{T}$

6.5 反向传播和其他的微分算法

算法 6.6 反向传播算法的内循环子程序 `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$，由算法 6.5 中定义的反向传播算法调用。

Require: $\mathbf{V}$，应该被加到 $\mathcal{G}$ 和 `grad_table` 的变量。

Require: $\mathcal{G}$，要修改的图。

Require: $\mathcal{G}'$，根据参与梯度的节点 $\mathcal{G}$ 的受限图。

Require: `grad_table`，将节点映射到对应梯度的数据结构。

返回：需要计算梯度的目标变量集T

```
if 𝕍 is in grad_table then
    Return grad_table[V]
end if
```

返回计算图g'中V的子节点

$i \leftarrow 1$

```
for C in get_consumers(V, G') do
    op ← get_operation(C)
    D ← build_grad(C, G, G', grad_table)
    G^(i) ← op.bprop(get_inputs(C, G'), V, D)
    i ← i + 1
end for
G ← ∑_i G^(i)
grad_table[V] = G
```

op存储计算图中流入C的边

反向传播，计算梯度，计算得到对于梯度D的输出

插入 $\mathbf{G}$ 和将其生成到 $\mathcal{G}$ 中的操作

Return $\mathbf{G}$

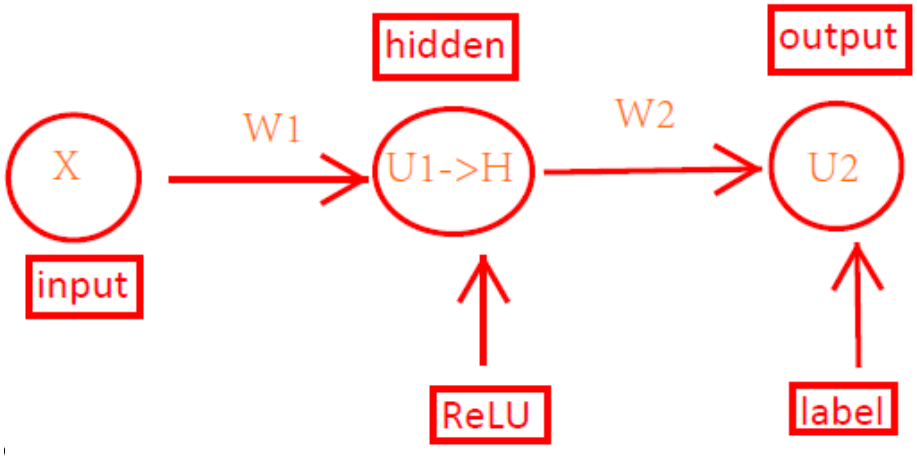
6.5 反向传播和其他的微分算法

实例：用于MLP 训练的反向传播

- 模型：具有单个隐藏层的MLP
- 方法：小批量随机梯度下降算法，反向传播算法用于计算单个小批量上的代价的梯度
- 代价函数：交叉熵，包含正则项

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right)$$

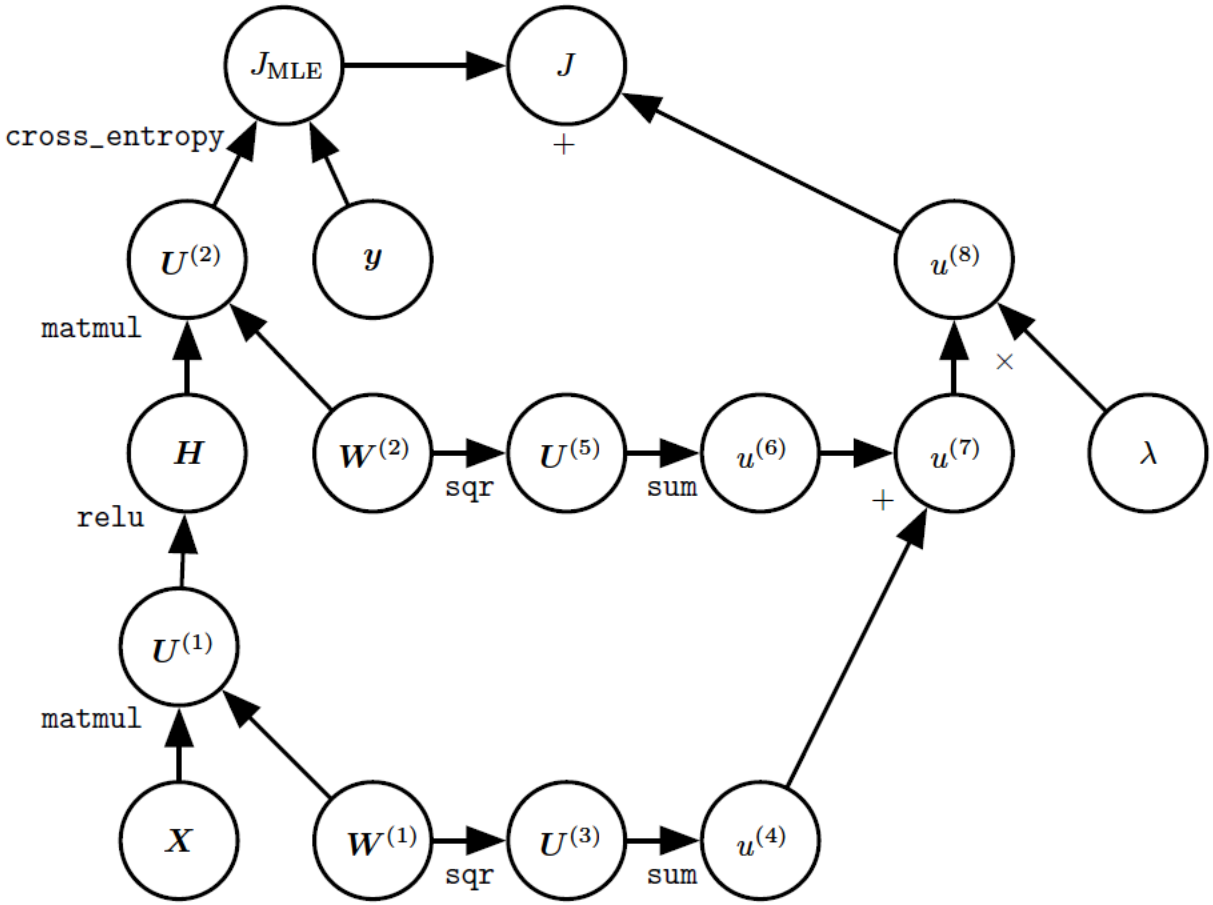
- 隐藏单元：ReLU



6.5 反向传播和其他的微分算法

实例：用于MLP 训练的反向传播

- 代价函数的计算图



6.5 反向传播和其他的微分算法

实例：用于MLP 训练的反向传播

- 代价函数
  - 计算 $\nabla_{\boldsymbol{W}^{(1)}} J$ 和 $\nabla_{\boldsymbol{W}^{(2)}} J$
  - 两种不同的路径，一条通过交叉熵代价，另一条通过权重衰减代价
  - 权重衰减代价总是对$W^{(i)}$ 上的梯度贡献$2\lambda W^{(i)}$
  - 另一条
    - 令$\boldsymbol{G}$是对未归一化对数概率$U^{(2)}$的梯度
    - 第一步，将$H^{\top}$ $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)\top}$
    - 紧接着，计算
    - 然后，将$X^{\top}G'$加到$W^{(1)}$的梯度上
  - 梯度下降算法或者其他优化算法所要做的就是使用这些梯度来更新参数

6.5 反向传播和其他的微分算法

复杂化

- 反向传播经常涉及将许多张量加在一起
- 反向传播的现实实现还需要处理各种数据类型
- 跟踪一些操作具有未定义的梯度的情况并且确定用户请求的梯度是否是未定义的

6.5 反向传播和其他的微分算法

深度学习界以外的微分

- 反向传播算法是一种称为反向模式累加的更广泛类型的技术的特殊情况
- 前向模式累加计算已经被提出用于循环神经网络梯度的实时计算
- 前向模式和后向模式的关系类似于左乘和右乘一系列矩阵之间的关系
- 以不同的顺序来计算链式法则的子表达式

6.5 反向传播和其他的微分算法

高阶微分

- 在深度学习的相关领域，很少会计算标量函数的单个二阶导数
- 通常对Hessian 矩阵的性质比较感兴趣，然而完整的Hessian 矩阵甚至不能表示
- 典型的，使用Krylov 方法，而不是不是显式地计算Hessian 矩阵
- Krylov 方法是用于执行各种操作的一组迭代技术
  - 近似求解矩阵的逆
  - 近似矩阵的特征值或特征向量
- 计算Hessian 矩阵H和一个任意向量$\boldsymbol{v}$间的乘积即可
  - 典型的$\boldsymbol{v}$是$\boldsymbol{e}^{(i)}$，是$e_i^{(i)} = 1$并且其他元素都为0的one-hot向量

6.6 历史小记

现代前馈网络的核心思想自20 世纪80 年代以来没有发生重大变化，神经网络性能的大部分改进可归因于两个因素：

- 较大的数据集减少了统计泛化对神经网络的挑战的程度
- 神经网络由于更强大的计算机和更好的软件基础设施已经变得更大

前馈网络还有许多未实现的潜力

- 未来优化算法和模型设计的进步将进一步提高它们的性能