

04 Backprop and Neural Networks 笔记

0 前言

神经网络中的数学知识将在深度学习框架不能满足你的功能，需要自定义网络的时候用到，这还是很有用的。

1 神经网络: 基础

神经元是一系列具有非线性决策边界的分类器，如 Figure1 所示。

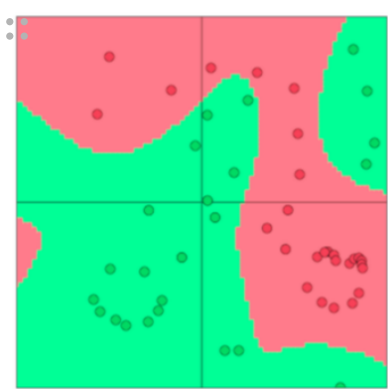


Figure 1: We see here how a non-linear decision boundary separates the data very well. This is the prowess of neural networks.

1.1 单个神经元

单个神经元指具有 n 个输入和 1 个输出的计算单元。使得不同神经元输出不同的是他们的参数(也称之为他们的权重)，最受欢迎的神经元选择之一是 sigmoid 单元或二元逻辑回归单元。计算单元输入 n 维向量 x 产生标量激活(输出) a ，神经元也通常与与 n 维权重向量 w 和偏置标量 b 相联系，这样的神经元的输出是：

$$a = \frac{1}{1 + \exp(-(wx + b))}$$

等价于：

$$a = \frac{1}{1 + \exp\left(-\begin{bmatrix} w^T & b \end{bmatrix} \cdot \begin{bmatrix} x & 1 \end{bmatrix}\right)}$$

公式可以用 Figure2 来可视化。

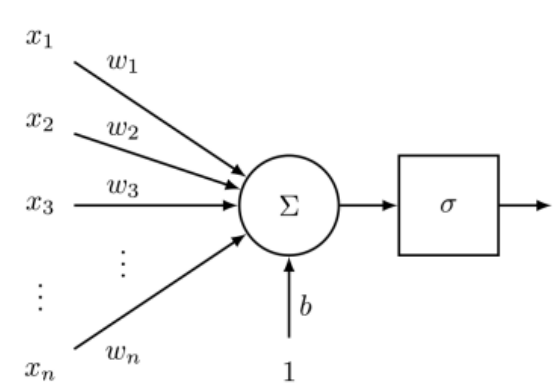


Figure 2: This image captures how in a sigmoid neuron, the input vector x is first scaled, summed, added to a bias unit, and then passed to the squashing sigmoid function

1.2 单层神经元

当输入 x 作为多个神经元的输入，就构成了单层神经元，如 Figure3 所示。

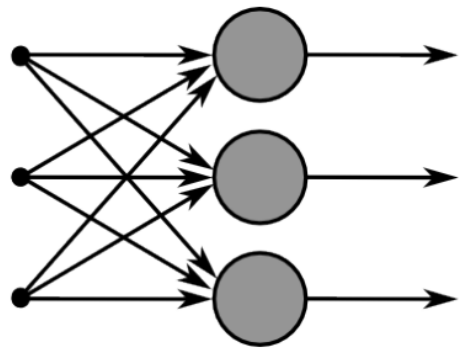


Figure 3: This image captures how multiple sigmoid units are stacked on the right, all of which receive the same input x .

如果我们把不同神经元的权重记为 $\{w^{(1)}, \dots, w^{(m)}\}$, 偏置记为 $\{b_1, \dots, b_m\}$, 那么各自的激活就是:

$$\begin{aligned} a_1 &= \frac{1}{1 + \exp(w^{(1)T}x + b_1)} \\ &\vdots \\ a_m &= \frac{1}{1 + \exp(w^{(m)T}x + b_m)} \end{aligned}$$

让我们定义以下抽象，以使符号简单且对更复杂的网络有用：

$$\begin{aligned} \sigma(z) &= \begin{bmatrix} \frac{1}{1+\exp(z_1)} \\ \vdots \\ \frac{1}{1+\exp(z_m)} \end{bmatrix} \\ b &= \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \in \mathbb{R}^m \\ W &= \begin{bmatrix} -w^{(1)T} \\ \dots \\ -w^{(m)T} \end{bmatrix} \in \mathbb{R}^{m \times n} \end{aligned}$$

现在可以把关于权重和偏置的输出记为：

$$z = Wx + b$$

sigmoid 函数的激活可以记为：

$$\begin{bmatrix} a^{(1)} \\ \vdots \\ a^{(m)} \end{bmatrix} = \sigma(z) = \sigma(Wx + b)$$

那么这些激活值是什么意思？这些激活可以被视为某些加权特征组合存在的指标。然后，我们可以结合使用这些激活来执行分类任务。

1.3 前馈计算(Feed-forward Computation)

前面我们看到一个输入向量 $x \in \mathbb{R}^n$ 如何被喂(fed)到一个单层的sigmoid单元，以生成激活 $a \in \mathbb{R}^m$.但这样做背后的直觉是什么? 让我们以下面的 NLP 中的命名实体识别(NER)问题为例：

”*Museums in Paris are amazing*”

这里我们想区分句子中间的"Paris"是否是命名实体，在这种情况下，我们倾向于不只是捕获词向量窗口中的当前单词，还包括单词之间的其他交互，以完成分类。比如，也许很重要，只有当"in"是第二个单词的时候，"Museums"才是第一个单词，这种非线性决策通常不会直接被输入直接喂到一个softmax函数的结构捕获到，而是需要对 1.2 节中提到的中间层进行评分(score).因此我们可以用另一个矩阵 $U \in R^{m \times 1}$ 从激活函数中为分类任务生成一个非归一化的分数，用到的激活函数为：

$$s = U^T a = U^T f(Wx + b)$$

其中 f 是激活函数.

维度分析：如果我们把每个单词用 4 维词向量表示，并且把 5 个单词的窗口(5-word window)作为输入(如同上面的例子)，那么输入是 $x \in \mathbb{R}^{20}$.如果我们在隐藏层中使用 8 个 sigmoid 单元并且从激活中生成 1 个分数来输出，那么 $W \in \mathbb{R}^{8 \times 20}$, $b \in 8$, $U \in \mathbb{R}^{8 \times 1}$, $s \in \mathbb{R}$.

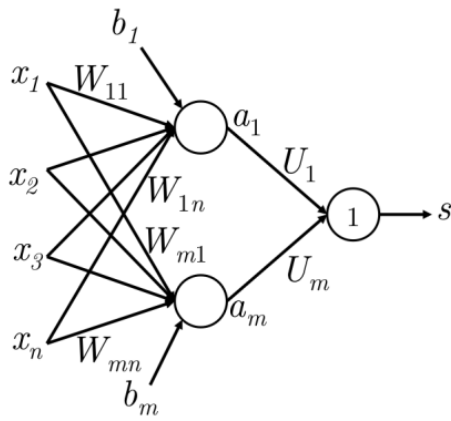


Figure 4: This image captures how a simple feed-forward network might compute its output

1.4 最大间隔目标函数(Maximum Margin Objective Function)

1.4.1 真正窗口和损坏窗口

例子：Not all museums in Paris are amazing

- 一个真正的窗口是"museums in Paris are amazing", 以 Paris 为中心的窗口和所有其他窗口都 “损坏” 了，因为它们的中心没有指定的实体位置
- “损坏” 窗口很容易找到，而且有很多：任何中心词没有在我们的语料库中明确标记为 NER 位置的窗口，比如"Not all museums in Paris"

1.4.2 最大间隔目标函数

和大多数机器学习模型一样，神经网络也需要一个优化目标，是我们希望分别最小化或最大化的 error 或 goodness 的度量。这里，我们将讨论一种流行的 error(误差)度量，称为最大间隔目标。使用这个目标的想法是为了确保标记为"ture"数据点计算出的分数高于标记为"false"数据点计算出的分数。

使用之前的例子，如果我们把对 “真” 标签窗口 *Museums in Paris are amazing* " 计算出的分数记为 s ，并且把对 “假” 标签窗口 *Not all museums in Paris* "计算出的分数记为 s_c (下标为 c，表示窗口已"corrupt").

"true" labeled window 这里翻译为 “真” 标签窗口

那么，我们的目标函数就是最大化($s - s_c$)或者最小化($s_c - s$). 但是为了确保只在 $s_c > s \Rightarrow s_c - s > 0$ 才计算 error, 我们需要修改目标函数.这么做背后的直觉是我们只关心 “真” 数据点比 “假” 数据点得分更高的情况，剩下的情况对我们来说并不重要。即我们想让 error 当 $s_c > s$ 时取($s_c - s$), 反之取 0. 因此，目标函数为：

$$\text{minimize } J = \max(s_c - s, 0)$$

"true" data point 这里翻译为 “真” 数据点

chapter04 **cs224n 疑问** 我理解真数据点比假数据点得分高是 $s - s_c > 0$ ，为什么给出的是 $s - s_c > 0$

然而，上面的优化目标是具有风险的，因为它没有试图创建一个安全的边界。我们想要 “真” 数据点得分比 “假” 数据点得分大于某个正的间隔 Δ . 换句话说，我们希望 error 当 $s - s_c < \Delta$ 就被开始计算 error, 而不仅仅是 $s - s_c < 0$. 因此，目标函数被修改为：

$$\text{minimize } J = \max(\Delta + s_c - s, 0)$$

取 Δ 为 1，其他不变参数必变，有关这方面的更多信息，请阅读函数和几何边界(functional and geometric margins)，这是支持向量机研究中经常涉及的主题。最终，我们定义了以下在所有训练窗口上对其优化的优化目标：

$$\text{minimize } J = \max(1 + s_c - s, 0)$$

其中 $s_c = U^T f(Wx_c + b)$ 且 $s = U^T f(Wx + b)$.

1.5 反向传播训练-元素的(Training with Backpropagation – Elemental)

这一节我们将讨论当 1.4 节中讨论的损失函数 J 为正时，如何训练模型中的不同参数。当损失函数为 0 时，无需对任何参数更新。由于我们通常采用梯度下降法(或 SGD 等变体)来更新参数，所以我们通常需要如下更新公式中任何参数的梯度信息：

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_{\theta^{(t)}} J$$

反向传播(backpropagation)是一种允许我们使用微分链式法则来计算模型前馈计算中使用的任何参数的损失梯度的技巧。为了更好的理解这一点，让我们来理解如Figure5所示我们将展示反向传播的简单网络。

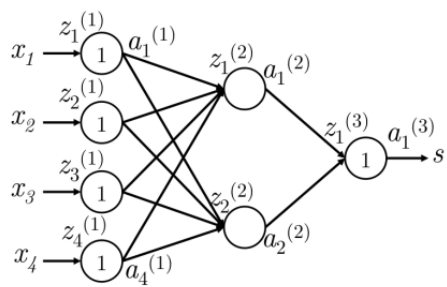


Figure 5: This is a 4-2-1 neural network

1.5.1 记号

这里我们使用了单层隐藏层和单个输出单元的神经网络。先介绍一些记号：

- x_i 是神经网络的输入
- s 是神经网络的输出
- 每一层(包括输入输出层)都有接收输入和产生输出的神经元。第k层的第j个神经元接收标量输入 $z_j^{(k)}$ 并产生标量输出 $a_j^{(k)}$
- 我们称在 $z_j^{(k)}$ 处计算的反向传播error为 $\delta_j^{(k)}$
- 第一层指的是输入层，而不是第一个隐藏层，对于输入层来说， $x_j = z_j^{(1)} = a_j^{(1)}$
- $W^{(k)}$ 是将第k层的输出映射到第(k+1) 层的输入的传递矩阵，因此，这个新的广义符号，从第1.3节的角度来满足 $W^{(1)} = W$ 以及 $W^{(2)} = U$

1.5.2 反向传播

假设损失函数 $J = (1 + s_c - s)$ 是正的且我们想要更新 $W_{14}^{(1)}$ (在Figure5和Figure6)，我们必须认识到 $W_{14}^{(1)}$ 只对 $z_1^{(2)}$ 起作用，因此对 $a_1^{(2)}$ 起作用。这一事实对于理解反向传播至关重要，即反向传播的梯度仅受其贡献的值影响。 $a_1^{(2)}$ 在随后的前向传播与 $W_1^{(2)}$ 相乘以计算得分。从最大边缘损失(max-margin loss)我们可以得出：

$$\frac{\partial J}{\partial s} = -\frac{\partial J}{\partial s_c} = -1$$

为了简单起见，我们将计算 $\frac{\partial s}{\partial W_{ij}^{(1)}}$ ：

$$\begin{aligned} \frac{\partial s}{\partial W_{ij}^{(1)}} &= \frac{\partial W_i^{(2)} a_i^{(2)}}{\partial W_{ij}^{(1)}} = \frac{\partial W_i^{(2)} a_i^{(2)}}{\partial W_{ij}^{(1)}} = W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}} \\ \Rightarrow W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}} &= W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\ &= W_i^{(2)} \frac{\partial f\left(z_i^{(2)}\right)}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\ &= W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\ &= W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial}{\partial W_{ij}^{(1)}}\left(b_i^{(1)}+a_1^{(1)} W_{i 1}^{(1)}+a_2^{(1)} W_{i 2}^{(1)}+a_3^{(1)} W_{i 3}^{(1)}+a_4^{(1)} W_{i 4}^{(1)}\right) \\ &= W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial}{\partial W_{ij}^{(1)}}\left(b_i^{(1)}+\sum_k a_k^{(1)} W_{i k}^{(1)}\right) \\ &= W_i^{(2)} f'\left(z_i^{(2)}\right) a_j^{(1)} \\ &= \delta_i^{(2)} \cdot a_j^{(1)} \end{aligned}$$

注：公式第5行原notes中f前少写了一个偏导符号

我们从上面看到所求梯度简化为点积 $\delta_i^{(2)} \cdot a_j^{(1)}$,其中 $\delta_i^{(2)}$ 本质上是从第2层的第i个神经元反向传播的error， $a_j^{(1)}$ 是经由W缩放作为第2层第i个神经元的输入。

让我们以Figure6为例来更好地讨论反向传播的“错误共享/分布”解释，加入我们要更新 $W_{14}^{(1)}$ ：

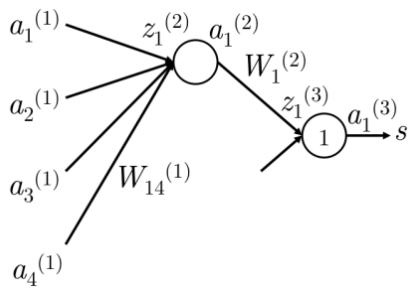


Figure 6: This subnetwork shows the relevant parts of the network required to update $W_{ij}^{(1)}$

1. 我们以 $a_1^{(3)}$ 的 1 信号反向传播开始。
2. 我们接着将 error 与从 $z_1^{(3)}$ 映射到 $a_1^{(3)}$ 的神经元的局部梯度相乘，在这个例子中梯度正好是 1，因此，error 仍然是 1。所以有 $\delta_1^{(3)} = 1$ 。
3. 此时，1 的 error 信号已经传递到 $z_1^{(3)}$ ，我们现在需要分配错误信号使得 error 的“错误共享”到达 $a_1^{(2)}$ 。
4. 总量是($z_1^{(3)}$ 处的 error 信号等于 $\delta_1^{(3)}$) $\times W_1^{(2)} = W_1^{(2)}$ ，因此，在 $a_1^{(2)}$ 处的 error 等于 $W_1^{(2)}$ 。
5. 与第 2 步做法相同，我们在将误差移动到把 $z_1^{(2)}$ 映射到 $a_1^{(2)}$ 的神经元上，通过把 $a_1^{(2)}$ 处的 error 信号与局部梯度相乘来完成这一操作，局部梯度是 $f'(z_1^{(2)})$ 。
6. 因此在 $z_1^{(2)}$ 处的 error 信号是 $f'(z_1^{(2)})W_1^{(2)}$ ，被称为 $\delta_1^{(2)}$ 。
7. 最后，我们需要将 error 的错误共享分配到 $W_{14}^{(1)}$ ，需要把 error 与它需要前向传播的 input 相乘，input 是 $a_4^{(1)}$ 。
8. 因此，对 $W_{14}^{(1)}$ 的梯度损失计算结果为 $a_4^{(1)}f'(z_1^{(2)})W_1^{(2)}$ 。

注意到使用此方法得到的结果与之前使用显式微分得到的结果完全相同。因此，我们可以使用微分链式法则或使用错误共享和分布式流方法计算网络中某个参数的误差梯度。这两种方法碰巧做了完全相同的事情，但多种方式考虑它们可能会有所帮助。

1.5.3 偏置更新

偏置项(如 $b_1^{(1)}$)和其他对神经元输入($z_1^{(2)}$)有贡献的权重在数学上是等价的，只是前向传播时其输入是 1(偏置与 1 相乘)。这样，第 k 层第 i 个神经元的偏置梯度是 $\delta_i^{(k)}$ 。比如，如果我们上面更新的是 $b_1^{(1)}$ 而不是 $W_{14}^{(1)}$ ，那么梯度就会是 $f'(z_1^{(2)})W_1^{(2)}$ 。

从 $\delta^{(k)}$ 到 $\delta^{(k-1)}$ 反向传播的一般步骤：

1. 我们有从 $z_i^{(k)}$ 反向传播的 error $\delta_i^{(k)}$ ，也即如 Figure7 所示的第 k 层的第 i 个神经元。
2. 我们通过 error 与路径上的权重 $W_{ij}^{(k-1)}$ 相乘来将 error 反向传播至 $a_j^{(k-1)}$ 。
3. 因此，在 $a_j^{(k-1)}$ 处接收的权重是 $\delta_i^{(k)}W_{ij}^{(k-1)}$ 。
4. 然而， $a_j^{(k-1)}$ 可能已经前向传播至如 Figure8 所示下一层的多个神经元，第 k 层的第 m 个神经元的 error 也应该使用完全相同的机制来反向传播。
5. 因此，在 $a_j^{(k-1)}$ 处接收的权重是 $\delta_i^{(k)}W_{ij}^{(k-1)} + \delta_m^{(k)}W_{mj}^{(k-1)}$ 。
6. 事实上，我们可以将其概括为 $\sum_i \delta_i^{(k)}W_{ij}^{(k-1)}$ 。
7. 现在有了在 $a_j^{(k-1)}$ 处的正确 error，我们通过把 error 和局部梯度 $f'(z_j^{(k-1)})$ 相乘来将 error 移动到第 k-1 层的第 j 个神经元。
8. 因此，到达 $z_j^{(k-1)}$ 的 error，记为 $\delta_j^{(k-1)}$ ，它等于 $f'(z_j^{(k-1)}) \sum_i \delta_i^{(k)}W_{ij}^{(k-1)}$ 。

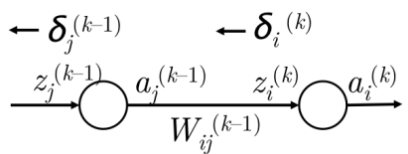


Figure 7: Propagating error from $\delta^{(k)}$ to $\delta^{(k-1)}$

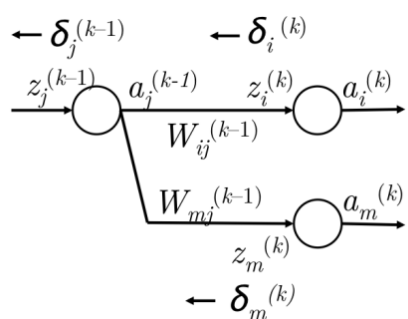


Figure 8: Propagating error from $\delta^{(k)}$ to $\delta^{(k-1)}$

1.6 反向传播训练-向量的(Training with Backpropagation – Vectorized)

目前为止，我们讨论了对模型中的给定参数如何计算梯度。这里我们将泛化上面的方法，一次性更新权重矩阵和偏置向量。值得注意的是，这些只是上述方法的简单扩展，这些扩展将帮助构建直觉来理解矩阵层面 error 的传播方式。

对与给定参数 $W_{ij}^{(k)}$ ，我们定义 error 梯度是 $\delta_i^{(k+1)} \cdot a_j^{(k)}$ 。其中 $W^{(k)}$ 是将 $a^{(k)}$ 映射到 $z^{(k+1)}$ 的矩阵。因此我们确定整个矩阵 $W^{(k)}$ 的 error 梯度：

∇_{W^(k)} = [[δ₁^(k+1) a₁^(k) δ₁^(k+1) a₂^(k) ...
δ₂^(k+1) a₁^(k) δ₂^(k+1) a₂^(k) ...
⋮ ⋮ ⋱]] = δ^(k+1) a^{(k)T}

因此，梯度矩阵可以通过矩阵写成反向传播 error 向量和前向传播激活的外积。

现在，我们看一下如何计算 error 向量 $\delta^{(k)}$ 。1.5.3 节中的结论 $\delta_j^{(k)} = f' \left(z_j^{(k)} \right) \sum_i \delta_i^{(k+1)} W_{ij}^{(k)}$ 可以被用矩阵写成：

δ^(k) = f' (z^(k)) ∘ (W^{(k)T} δ^(k+1))

在上面的公式中 ∘ 运算符是表示向量之间对应元素的相乘(∘ : ℝ^N × ℝ^N → ℝ^N)

计算效率：在讨论了 element-wise 的更新和 vector-wise 的更新之后，必须意识到在如 MATLAB 和 Python(使用 Numpy/SciPy)之类的科学计算环境下， 向量化的实现会更快。因此，我们实践中应该使用向量化实现。更进一步，在反向传播中我们应该减少冗余计算。比如，注意到 $\delta^{(k)}$ 直接依赖于 $\delta^{(k+1)}$ ，那我们就应该提前把 $\delta^{(k+1)}$ 存下来。以此类推 $(k - 1) ... (1)$ 。这种递归过程使得反向传播是一个计算上可负担的过程。

2 神经网络: Tips and Tricks

前面讨论了神经网络的数学基础，现在要深入了解一些在实际使用神经网络时常用到的 tips 和 tricks。

2.1 梯度检验(Gradient Check)

上一节中，我们讨论了通过微积分(数学分析)方法在神经网络模型中对参数计算 error 梯度/更新的细节。现在介绍估算这些梯度的数值上(numerically)的技巧。尽管它对于训练模型来说计算效率太低以至于不能使用，但这个方法允许我们非常精确地估计任意参数的导数。因此，它可以作为一个有用的分析导数正确性的合理性检验。给定参数为 θ 且损失函数为 J 的模型， θ_i 处的数值上是梯度可以简单地由**中心差分公式(centered difference formula)**:

f'(θ) ≈ (J (θ⁽ⁱ⁺⁾) - J (θ⁽ⁱ⁻⁾)) / (2ε)

其中 ϵ 是一个很小的数(通常在 $1e^{-5}$ 左右)。 $J(\theta^{i+})$ 是当我们给参数 θ 的第 i 个元素添加 $+\epsilon$ 的扰动时，对给定输入在前向传播计算的 error。相似地，式子 $J(\theta^{i-})$ 是当我们给参数 θ 的第 i 个元素添加 $-\epsilon$ 的扰动时，对给定输入在前向传播计算的 error。因此，计算两次前向传播，我们可以对模型中任意给定参数估计梯度。我们注意到这个数值梯度的定义和导数的定义类似。在标量的情况下：

f'(x) ≈ (f(x + ε) - f(x)) / ε

当然，有一点不同，即上述的公式在计算梯度时只对 x 有正向扰动。虽然可以用这种方式定义梯度，但在实践中使用同时具有两个方向扰动的中心差分公式会更精确和稳定。背后的直觉是为了更好地逼近点周围的导数/斜率，我们需要检查点左右两边 f' 的行为。用泰勒定理也能说明中心差分公式 error 与 ϵ^2 成比例，这个值很小(ϵ 的高阶无穷小)，而且导数定义式更容易出错。

现在，你可能想问一个自然而然的问题，如果这个方法这么好，为什么不使用它来计算神经网络梯度而是使用反向传播呢？简单来说，正如前面暗示的，它的计算效率低。每当我们想计算一个元素的梯度的时候，需要在网络中做两次前向传播，这样是很耗费计算资源的。再者，很多大规模的神经网络含有几百万的参数，对每个参数都计算两次明显不是一个好的选择。同时在例如 *SGD* 这样的优化方法中，我们必须在数千次迭代中，每次迭代计算一次梯度，使用这样的方法很快会变得难以应付。这种低效性是我们只使用梯度检验来验证我们的分析梯度的正确性的原因。梯度检验的实现如下所示：

Python

Snippet 2.1

```
def eval_numerical_gradient(f, x):  
    """  
    a naive implementation of numerical gradient of f at x  
    - f should be a function that takes a single argument  
    - x is the point (numpy array) to evaluate the gradient  
    at  
    """  
    fx = f(x) # evaluate function value at original point
```



```
grad = np.zeros(x.shape)
h = 0.00001

# iterate over all indexes in x
it = np.nditer(x, flags=['multi_index'],
               op_flags=['readwrite'])
while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh_left = f(x) # evaluate f(x + h)
    x[ix] = old_value - h # decrement by h
    fxh_right = f(x) # evaluate f(x - h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh_left - fxh_right) / (2*h) # the slope
    it.iternext() # step to next dimension
return grad
```

2.2 正则化

和很多机器学习的模型一样，神经网络很容易过拟合，即模型在训练集上表现完美，但是却不能泛化到没见过的数据上。一个常见的用于解决过拟合（“高方差问题”）的方法是引入 $L2$ 正则化。做法是在损失函数 J 上增加一个正则项，现在的损失函数：

$$J_R = J + \lambda \sum_{i=1}^L \left\| W^{(i)} \right\|_F$$

矩阵 U 的 Frobenius 范数：

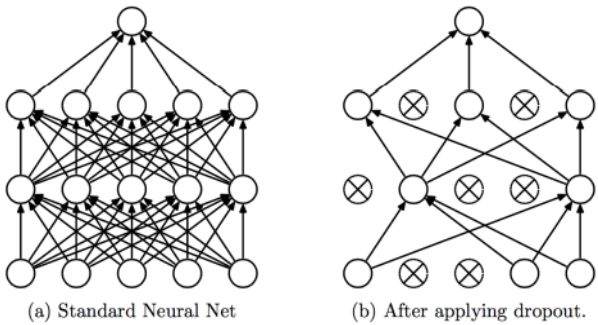
$$\|U\|_F = \sqrt{\sum_i \sum_j U_{ij}^2}$$

其中 $\left\| W^{(i)} \right\|_F$ 是矩阵 $W^{(i)}$ (神经网络的第*i*个矩阵)的Frobenius norm(Frobenius 范数, 简称F-范数)且 λ 是控制正则项相对原损失函数的权重的超参数。当我们尝试去最小化 J_R , 正则化本质上就是当优化损失函数的时候，惩罚数值太大的权值。由于Frobenius 范数的二次的性质（计算矩阵的元素的平方和）， $L2$ 正则化有效地降低了模型的灵活性并因此减少出现过拟合的可能性。引入这样一个约束可以用贝叶斯派的思想解释，这个正则项是对模型的参数加上一个先验分布，优化权值使其接近于0, 有多接近是取决于 λ 。选择一个合适的 λ 很重要，需要调参。 λ 太大会令很多权值都接近于0，模型无法在训练集上学习到有意义的东西，通常在训练、验证和测试集上的表现都非常差。 λ 值太小，会让模型仍旧出现过拟合的现象。需要注意的是，偏置项不会被正则化，以及不会计算入损失项中，尝试思考一下为什么(我理解是因为 $J = (1 + s_c - s)$, 又因为 $s_c = U^T f(Wx_c + b)$ 且 $s = U^T f(Wx + b)$, 化简后J中没有偏置 b 了)。

有时候我们会用到其他类型的正则项，例如 $L1$ 正则项，它的做法是将参数元素的绝对值全部加起来。但是，在实际中很少会用 $L1$ 正则项，因为它会导致权值参数稀疏。在下一部分，我们讨论 dropout，这是另外一种有效的正则化方法，通过在前向传播中随机丢弃神经元(将神经元设为 0)

2.3 Dropout

Dropout是一种很强大的正则化技巧，由 Srivastava et al.在论文 Dropout: A Simple Way to Prevent Neural Networks from Overfittin 中首次提出。这个想法简单有效，训练时，在每一次前向/反向传播中随机"drop"一些概率是($1 - p$)的神经元的子集(也就是说，保留概率是 p 的神经元)。在测试阶段，我们将使用全部的神经元来进行预测。结果是网络能从数据中学到更多有意义的信息，更少出现过拟合以及通常在任务上表现更好。这种技巧如此有效的一个直观原因是，dropout是一次以指数形式训练许多较小的网络，并并其预测值取平均。



Dropout applied to an artificial neural network. Image credits to Srivastava et al

在实践中，我们dropout的方式是我们取每个神经元层的输出 h , 保留概率为 p 的神经元，其余设为0。然后，反向传播时，我们只通过前向传播时保持"活着"(keep alive)的神经元传递梯度。最后，在测试阶段，我们使用神经网络中全部的神经元进行前向传播计算。然而，一个关键的微妙之处在于，为了dropout有效工作，测试阶段的神经元的预期输出应与训练阶段大致相同，否则输出的大小可能会很不同，并且神经网络的表现也不好定义。因此，我们通常必须在测试阶段将每个神经元的输出除以某个值， 这留给读者作为练习来确定这个值应该是多少，以便在训练和测试期间的预期输出相等。(个人认为，如果保留概率为 p 的神经元，是否被 dropout 服从伯努利分布，分布的期望就是np。那么就是整体输出的期望是原来的 p , 所以相应的，测试阶段要除以 $(1/p)$)。

2.4 神经元单元(Neuron Units)

到目前为止，我们已经讨论了包含非线性 sigmoidal 神经元的神经网络。但是在很多应用中，使用其他激活函数可以设计更好的神经网络。下面列出一些常见的激活函数和激活函数的梯度定义，它们可以和前面讨论过的 sigmoidal 函数互相替换。

2.4.1 Sigmoid

Sigmoid是我们讨论过的常用选择，激活函数 σ 为

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

其中 $\sigma(z) \in (0, 1)$ 。

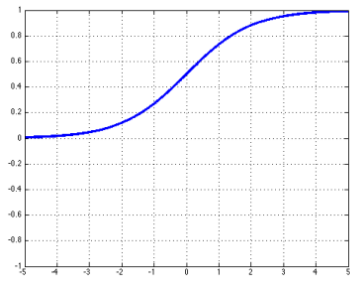


Figure 9: The response of a sigmoid nonlinearity

$\sigma(z)$ 的梯度为：

$$\sigma'(z) = \frac{-\exp(-z)}{1 + \exp(-z)} = \sigma(z)(1 - \sigma(z))$$

2.4.2 Tanh

Tanh 函数是 sigmoid 函数的一种替代，Tanh 函数在实践中通常收敛得更快。tanh 和 sigmoid 的主要区别在于，tanh 的输出范围为 -1 到 1 ，而 sigmoid 的输出范围为 0 到 1 。

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} = 2\sigma(2z) - 1$$

其中 $\tanh(z) \in (-1, 1)$ 。

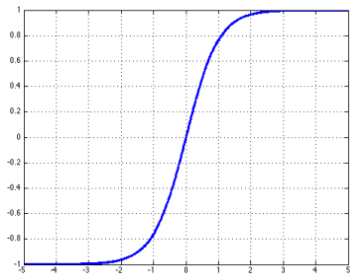


Figure 10: The response of a tanh nonlinearity

$\tanh(z)$ 的梯度为：

$$\tanh'(z) = 1 - \left(\frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \right)^2 = 1 - \tanh^2(z)$$

2.4.3 Hard tanh

Hard-tanh 函数有时优于 tanh 函数，因为它计算量更小。然而，当 z 的量级大于 1 时它会饱和。Hard-tanh 的激活函数为：

$$\text{hardtanh}(z) = \begin{cases} -1 & : z < -1 \\ z & : -1 \leq z \leq 1 \\ 1 & : z > 1 \end{cases}$$

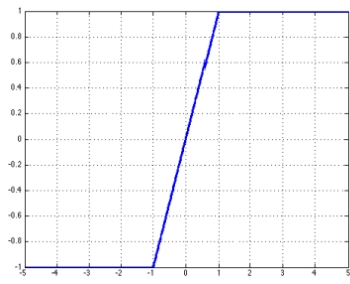


Figure 11: The response of a hard tanh nonlinearity

导数也可以用分段函数来表示：

$$\text{hardtanh}'(z) = \begin{cases} 1 & : -1 \leq z \leq 1 \\ 0 & : \text{otherwise} \end{cases}$$

2.4.4 Soft sign

soft sign 函数是另一种非线性函数，可被视为 tanh 的替代品，因为它不像硬剪裁函数那样容易饱和：

$$\text{softsign}(z) = \frac{z}{1 + |z|}$$

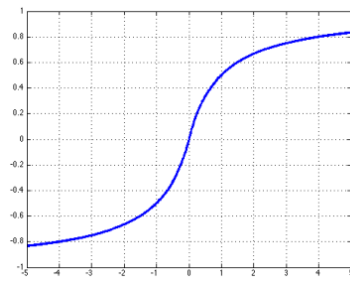


Figure 12: The response of a soft sign nonlinearity

导数：

$$\text{softsign}'(z) = \frac{\text{sgn}(z)}{(1 + |z|)^2}$$

其中，sgn 是 signum 函数，根据 z 的符号返回 ±1。

2.4.5 ReLU

ReLU (Rectified Linear Unit) 函数是一种常用的激活选择，因为它即使对于较大的 z 值也不会饱和，并且在计算机视觉应用中取得了很大的成功：

$$\text{rect}(z) = \max(z, 0)$$

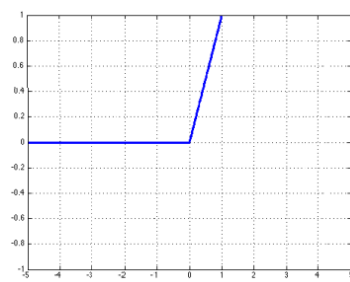


Figure 13: The response of a ReLU nonlinearity

导数是分段函数：

$$\text{rect}'(z) = \begin{cases} 1 & : z > 0 \\ 0 & : \text{otherwise} \end{cases}$$

2.4.6 Leaky ReLU

传统的 ReLU 单元在设计上不会传播非正 z 的任何 error，而 Leaky ReLU 修改了这一点，从而即使 z 为负也允许小 error 反向传播：

$$\text{leaky}(z) = \max(z, k \cdot z)$$

其中 $0 < k < 1$ 。

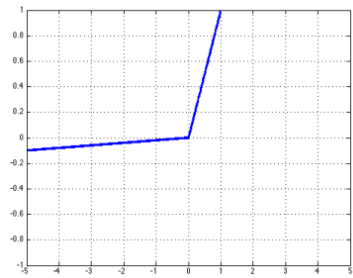


Figure 14: The response of a leaky ReLU nonlinearity

因此，导数可以表示为：

$$\text{leaky}'(z) = \begin{cases} 1 & : z > 0 \\ k & : \text{otherwise} \end{cases}$$

2.5 数据处理

与一般的机器学习模型一样，确保模型在手头的任务上获得合理性能的关键步骤是对数据执行基本的预处理。下面概述了一些常见技术。

2.5.1 均值减法(Mean Subtraction)

给定一组输入数据 X ，通常通过从 X 中减去 X 的平均特征向量来将数据变成零中心(zero-center)。重要的一点是，在实践中，平均值仅在整个训练集上计算，并且在训练集、验证集和测试集中减去该平均值。

2.5.2 归一化(Normalization)

另一种常用的技术（可能不如 mean subtraction 常用）是缩放每个输入特征维度，使其具有相似的大小范围。这很有用，因为输入特性通常以不同的“单位”进行度量，但我们通常希望最初将所有特性视为同等重要。实现这一点的方法是简单地将特征除以在它们各自在整个训练集上计算的的标准差。

2.5.3 白化(Whitening)

白化不像 mean detracton 和 normalization 那样常用，它本质上是将数据转换为具有单位协方差矩阵的数据，即特征变得不相关且方差为 1。步骤如下：

- 首先，对数据进行均值减法，得到 X' 。
- 然后，对 X' 进行SVD分解得到矩阵 U 、 S 、 V 。
- 接着，计算 UX' 将 X' 投影到 U 的列所定义的基上。
- 最后，将结果的每个维度除以相应 S 中的奇异值(singular value)，以适当地缩放数据（如果奇异值为零，我们就用一个很小的数字来代替）。

2.6 参数初始化

让神经网络实现最佳性能的关键一步是用合理的方式来初始化参数。一个好的起始方法是一般将权值初始化为正态分布在 0 附近的小随机数，实践中效果还不错。然而，在 Understanding the difficulty of training deep feedforward neural networks (2010)这篇论文中, Xavier et al 研究了对不同权重和偏置初始化方案 training dynamics 的影响，实验结果表明，对于 sigmoid 和 tanh 激活单元，当矩阵 $W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$ 的权重以如下的均匀分布随机初始化时，收敛速度更快，错误率更低。

$$W \sim U \left[-\sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}} \right]$$

其中 $n^{(l)}$ 是 W (fan-in)的输入单元数， $n^{(l+1)}$ 是 W (fan-out)的输出单元数。在这个参数初始化方案中，偏置单元初始化为 0。这种方法试图保持跨层之间的激活方差以及反向传播的梯度方差。如果没有这样的初始化，梯度方差（作为信息的代理）通常会随着层间的反向传播而减小。

fan-in: n.输入端数；扇入
fan-out: n.输出端数；扇出

chapter04 cs224n 疑问 梯度方差的是什么意思，反向传播中它减小有什么影响

2.7 学习策略(Learning Strategies)

训练期间模型参数更新的速率/幅度可以使用学习率进行控制。在最简单的梯度下降公式中， α 是学习率：

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \nabla_{\theta} J_t(\theta)$$

你可能会认为，为了快速收敛，我们应该将 α 设置为更大的值，但更大的收敛速度并不能保证更快收敛。事实上，对于非常大的学习率，我们可能会遇到损失函数实际上会发散，因为参数更新会导致模型超出凸极小值，如 Figure15 所示。在非凸模型中（我们使用的大多数模型），学习率高的结果是不可预测的，但损失函数发散的可能性非常高。

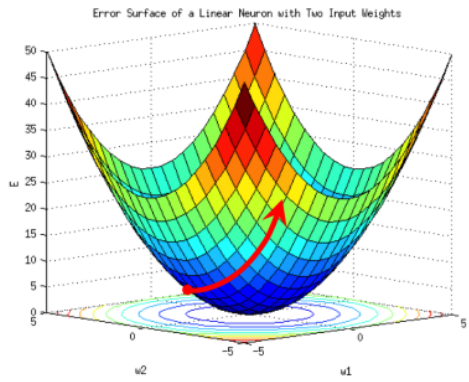


Figure 15: Here we see that updating parameter w2 with a large learning rate can lead to divergence of the error

避免损失函数发散的一个简答的解决方法是使用一个很小的学习率，让我们可以谨慎地在参数空间中移动。当然，如果我们使用了一个太小的学习率，损失函数可能不会在合理的时间内收敛，或者会困在局部最优点。因此，与任何其他超参数一样，学习率必须有效地调整。

由于深度学习系统中最消耗计算资源的是训练阶段，因此一些研究已在尝试改善这种简单的设置学习率的方法。例如， Ronan Collobert通过神经元($n^{(l)}$)的平方根倒数来缩放权重 W_{ij} (其中 $W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$)的学习率。

还有其他已经被证明有效的技术，其中一个叫 **annealing**，在几次迭代之后，学习率以某种方式减小，这种方法保证以一个高的的学习率开始训练宁且快速逼近最小值；当越来越接近最小值时，开始降低学习率，以便在更细微的范围内找到最优值。一个常见的实现 annealing 的方法是在每 n 次学习迭代后，通过因子 x 来减小学习率 α 。指数衰减也很常见，关于迭代次数 t 的学习率 α 是 $\alpha(t) = \alpha_0 e^{-kt}$ ，其中 α_0 是初始学习率，k是超参。另一个方法是允许学习率随时间衰减，比如：

$$\alpha(t) = \frac{\alpha_0 \tau}{\max(t, \tau)}$$

在上面的方案中， α_0 是可调节的参数，代表初始学习率。 τ 也是可调节参数学习率应该何时衰减。在实践中，这个方法效果很好。在下面的章节，我们将讨论不需要手动设置学习率的其他自适应梯度下降(adaptive gradient descent)方法。

2.8 动量更新(Momentum Updates)

动量方法，灵感来自于物理学中的对动力学研究，是梯度下降方法的一种变体，尝试使用更新的“速度”的一种更有效的更新方案。动量更新的伪代码如下所示：

Python

Snippet 2.2

```
# Computes a standard momentum update
# on parameters x
v = mu*v - alpha*grad_x
x += v
```

2.9 自适应优化方法

daGrad是标准随机梯度下降（SGD）的一种实现，但有一个关键区别：每个参数的学习率是不同的。每个参数的学习率取决于该参数的梯度更新历史。参数的更新历史越稀少，就使用越大的学习率加快更新。换句话说，之前很少被更新的参数倾向于现在以更大的学习率更新。公式如下：

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^t g_{\tau,i}^2}} g_{t,i}$$

其中 $g_{t,i} = \frac{\partial}{\partial \theta_i^t} J_t(\theta)$ 。

在这个技巧中，我们看到如果梯度的历史 RMS 很低，那么学习率会非常高。这个技巧的一个简单的实现如下所示：

Python

Snippet 2.3

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / np.sqrt(cache + 1e-8)
```

其他常见的自适应方法有 RMSProp 和 Adam，其更新规则如下所示(Andrej Karpathy 提供)：



Python

Snippet 2.4

```
# Update rule for RMS prop
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

```
Snippet 2.5
# Update rule for Adam
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

RMSProp是AdaGrad的一个变体，它利用平方梯度的移动平局值。实际上，和AdaGrad不一样，它的更新不会单调变小。Adam 更新规则又是RMSProp的一个变体，但是加上了动量更新。更详细的内容请阅读参考。

 An overview of gradient descent optimization.pdf  643.58 KB

chapter04 cs224n 疑问 优化理论这块不是很熟悉，需要花时间看数学公式

3 计算图

3.1 公式推导

一个简单的3层神经网络

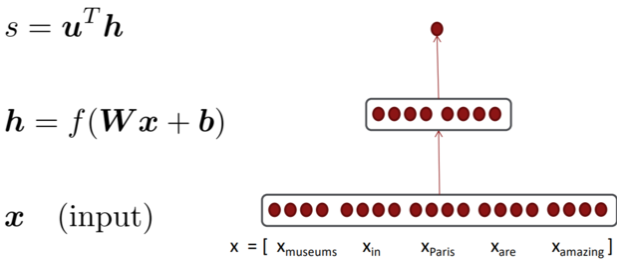


Figure15

求解偏导及

$$\begin{aligned} s &= \mathbf{u}^T \mathbf{h} \\ \mathbf{h} &= f(\mathbf{z}) \\ \mathbf{z} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\ \mathbf{x} &\text{ (input)} \end{aligned}$$

$$\begin{aligned} \frac{\partial s}{\partial \mathbf{b}} &= \frac{\partial s}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{b}} \\ &\downarrow \qquad \downarrow \qquad \downarrow \\ &= \mathbf{u}^T \text{diag}(f'(\mathbf{z})) \mathbf{I} \\ &= \mathbf{u}^T \circ f'(\mathbf{z}) \end{aligned}$$

Useful Jacobians from previous slide

$$\begin{aligned} \frac{\partial}{\partial \mathbf{u}}(\mathbf{u}^T \mathbf{h}) &= \mathbf{h}^T \\ \frac{\partial}{\partial \mathbf{z}}(f(\mathbf{z})) &= \text{diag}(f'(\mathbf{z})) \\ \frac{\partial}{\partial \mathbf{b}}(\mathbf{W}\mathbf{x} + \mathbf{b}) &= \mathbf{I} \end{aligned}$$

Figure16

chapter04 cs224n 疑问 不明白为什么是对角阵 diag，以及如何化简矩阵元素相乘

链式法则

- Suppose we now want to compute $\frac{\partial s}{\partial \mathbf{W}}$
 - Using the chain rule again:

$$\begin{aligned} \frac{\partial s}{\partial \mathbf{W}} &= \boldsymbol{\delta} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \\ \frac{\partial s}{\partial \mathbf{b}} &= \boldsymbol{\delta} \frac{\partial \mathbf{z}}{\partial \mathbf{b}} = \boldsymbol{\delta} \\ \boldsymbol{\delta} &= \frac{\partial s}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \mathbf{u}^T \circ f'(\mathbf{z}) \end{aligned}$$

Figure17

3.2 计算图原理

在深度学习框架（TensorFlow, PyTorch 等）中，梯度不是依赖人们手动计算的，而是通过计算图（Computational Graphs）来自动求导的。比如对于一个简单的神经网络：

$$\begin{aligned} s &= \mathbf{u}^T \mathbf{h} \\ \mathbf{h} &= f(\mathbf{z}) \\ \mathbf{z} &= \mathbf{W}\mathbf{x} + \mathbf{b} \end{aligned}$$

前向传播用计算图来表示就是：

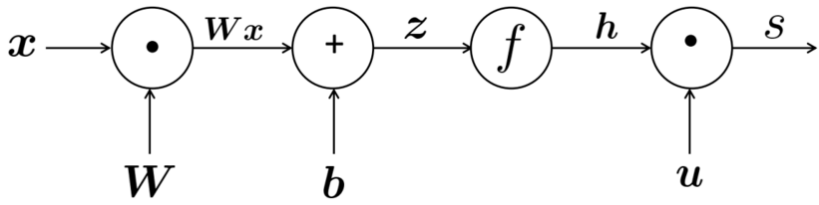


Figure18 Computation Graphs 1

反向传播也是沿着计算图的路径传播，即：

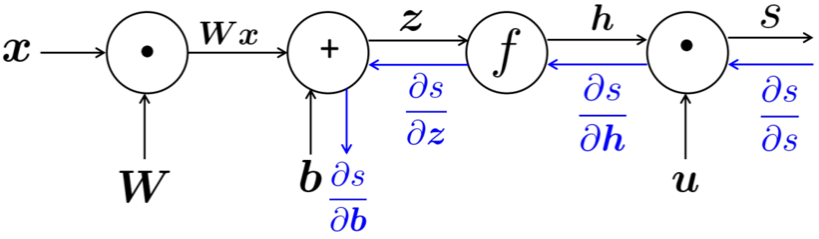


Figure19 Computation Graphs 2

对于单个节点来说，反向传播的梯度等于上一个节点的梯度与当前节点梯度的乘积，下图中 $h = f(z)$

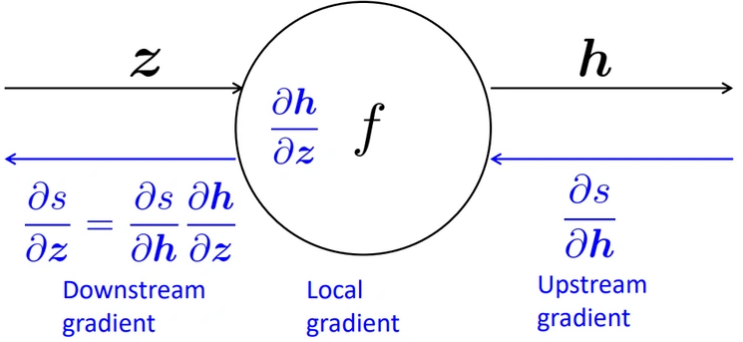


Figure20 Computation Graphs 3

多个输入意味着多个当前节点梯度，下图中 $Z = Wx$

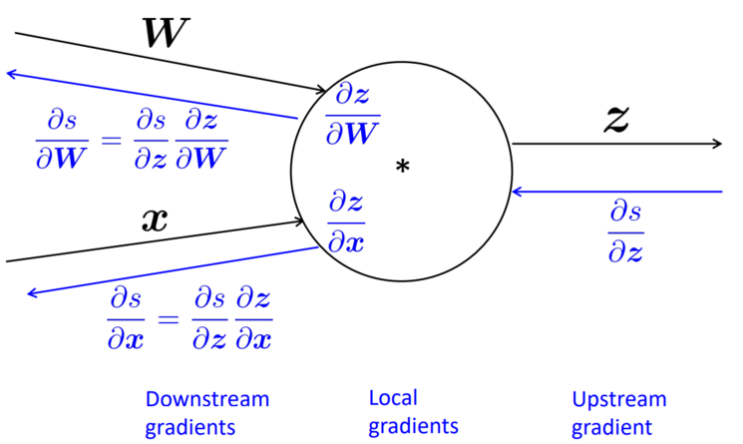


Figure21 Computation Graphs 4

3.2 计算图实现

概览:

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Figure22 Computation Graphs 5

前向/后向 API:

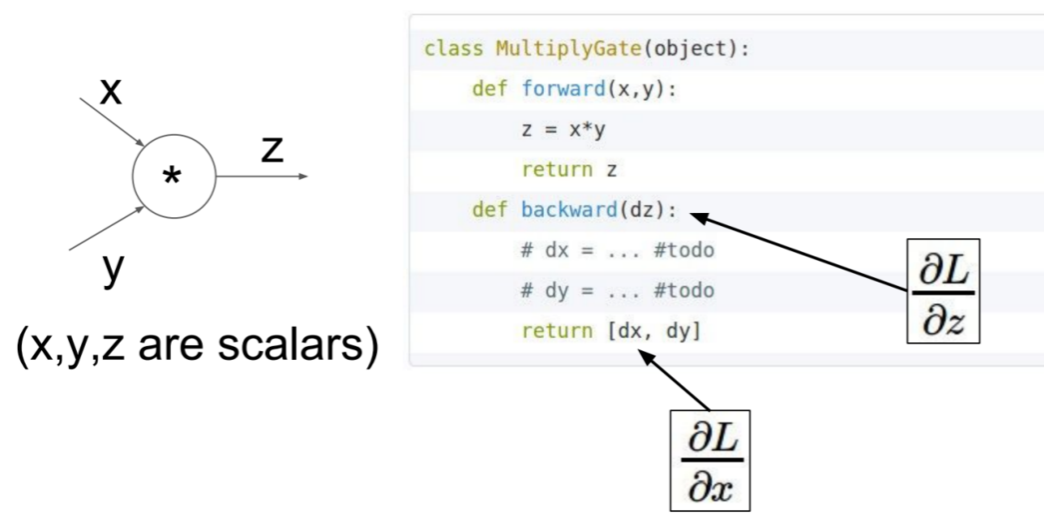


Figure23 Computation Graphs 6

4 引用

1. cs224n winter2021 notes
2. cs224n winter2021 slides
3. cs224n winter2019 视频
4. <https://zhuanlan.zhihu.com/p/61272294>
5. <https://zhuanlan.zhihu.com/p/382219345>
6. <https://www.zhihu.com/question/61751133/answer/794717140>
7. <https://www.zhihu.com/question/61751133/answer/243909675>
8. <https://ruder.io/optimizing-gradient-descent/>