Lecture Notes: Part III

Neural Networks, Backpropagation

CS224n 是顶级院校斯坦福出品的深度学习与自然语言处理方向专业课程，核心内容覆盖 RNN、LSTM、CNN、transformer、bert、问答、摘要、文本生成、语言模型、阅读理解等前沿内容。

这组笔记介绍了单层和多层神经网络，以及如何将它们用于分类目的。然后我们讨论如何使用一种称为反向传播的分布式梯度下降技术来训练它们。我们将看到如何使用链式法则按顺序进行参数更新。在对神经网络进行严格的数学讨论之后，我们将讨论一些训练神经网络的实用技巧和技巧，包括:神经元单元(非线性)、梯度检查、Xavier 参数初始化、学习率、Adagrad 等。最后，我们将鼓励使用递归神经网络作为语言模型。

笔记核心词:

Neural networks , Forward omputation, Backward, propagation, Neuron Units, Max-margin Loss, Gradient checks, Xavier parameter initialization, Learning rates, Adagrad

课程**全部资料和信息**已整理发布，扫描下方**任意**二维码，均可获取！！

微信公众号 · 全套资料
回复 CS224n
底部菜单栏

Bilibili · 课程视频
视频简介
置顶评论

GitHub · 项目代码
阅读 ReadMe
点击超链接

1. Neural Networks: Foundations

在前面的讨论中认为，因为大部分数据是线性不可分的所以需要非线性分类器，不然的话线性分类器在这些数据上的表现是有限的。神经网络就是如右图所示的一类具有非线性决策分界的分类器。现在我们知道神经网络创建的决策边界，让我们看看这是如何创建的。

❐ 神经网络是受生物学启发的分类器，这就是为什么它们经常被称为“人工神经网络”，以区别于有机类。然而，在现实中，人类神经网络比人工神经网络更有能力、更复杂，因此通常最好不要在两者之间画太多的相似点。

I Notes info.

课件/Slides	Lecture 3, P16
视频/Video	Lecture –, --:--
GitHub · 代码	实时在线查阅文档
Bilibili · 视频	中英字幕课程视频

Stanford University X ShowMeAI

1.1 A Neuron

神经元是一个通用的计算单元，它接受 n 个输入并产生一个输出。不同的神经元根据它们不同的参数（一般认为是神经元的权值）会有不同的输出。对神经元来说一个常见的选择是 $sigmoid$，或者称为“二元逻辑回归”单元。这种神经元以 n 维的向量作为输入，然后计算出一个激活标量（输出）a。该神经元还与一个 n 维的权重向量 w 和一个偏置标量 b 相关联。这个神经元的输出是：

$$a = \frac{1}{1 + exp(-(w^T x + b))}$$

我们也可以把上面公式中的权值和偏置项结合在一起：

$$a = \frac{1}{1 + exp(-[w^T \quad x] \cdot [x \quad 1])}$$

这个公式可以以右图的形式可视化。

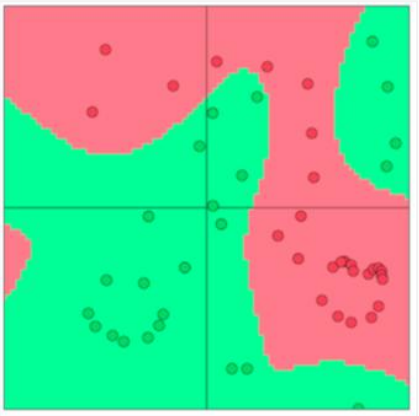
❐ 神经元：神经元是神经网络的基本组成部分。我们将看到，神经元可以是许多允许非线性在网络中累积的函数之一。

❐ → 图 2：这幅图像捕捉了在一个 sigmoid 神经元中，输入向量 x 是如何被缩放，求和，添加到一个偏置单元，然后传递给挤压 sigmoid 函数的。

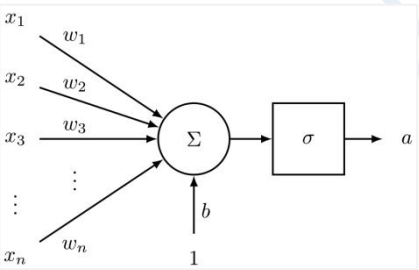
I.I Notes info.

课件/Slides	Lecture 3, P17
视频/Video	Lecture 3, 28:30
GitHub · 代码	实时在线查阅文档
Bilibili · 视频	中英字幕课程视频

Stanford University X ShowMeAI



❐ ↑图 1：我们在这里看到非线性决策边界如何很好地分离数据。这就是神经网络的威力。



1.2 A Single Layer of Neurons

我们将上述思想扩展到多个神经元，考虑输入 x 作为多个这样的神经元的输入，如右图 3 所示。

如果我们定义不同的神经元的权值为 $\{w^{(1)},\ldots,w^{(m)}\}$ 、偏置为 $\{b_1,\ldots,b_m\}$ 和相对应的激活输出为 $\{a_1,\ldots,a_m\}$：

$$a_1 = \frac{1}{1+\exp\left(-\left(w^{(1)T}x+b\right)\right)}$$
$$\vdots$$
$$a_m = \frac{1}{1+\exp\left(-\left(w^{(m)T}x+b\right)\right)}$$

让我们定义简化公式以便于更好地表达复杂的网络：

$$\sigma(z) = \begin{bmatrix} \frac{1}{1+\exp(z_1)} \\ \vdots \\ \frac{1}{1+\exp(z_m)} \end{bmatrix}$$

$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \in \mathbb{R}^m$$

$$W = \begin{bmatrix} -w^{(1)T}- \\ \vdots \\ -w^{(m)T}- \end{bmatrix} \in \mathbb{R}^{m\times n}$$

我们现在可以将缩放和偏差的输出写成：$z = Wx + b$

激活函数 sigmoid 可以变为如下形式：

$$\begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix} = \sigma(z) = \sigma(Wx+b)$$

那么这些激活的作用是什么呢？可以把这些激活看作是一些加权特征组合存在的指标。然后我们可以使用这些激活的组合来执行分类任务。

1.3 Feed-forward Computation

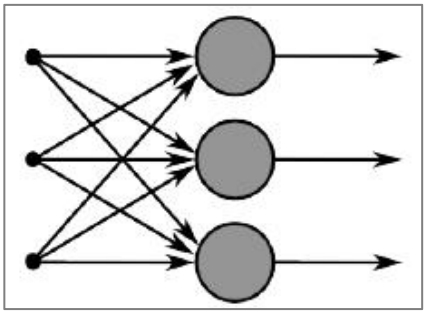
到目前为止我们知道一个输入向量 $x \in \mathbb{R}^n$ 可以经过一层 $sigmoid$ 单元的变换得到激活输出 $a \in \mathbb{R}^m$ 。但是这么做的直觉是什么呢？让我们考虑一个 NLP 中的命名实体识别问题为例：

Museums in Paris are amazing

1.2 Notes info.

课件/Slides	Lecture 3, P18
视频/Video	Lecture –, ––:00
GitHub • 代码	实时在线查阅文档
Bilibili • 视频	中英字幕课程视频

Stanford University X ShowMeAI



↑图 3：这幅图像捕捉了多个 sigmoid 单元如何堆叠在右侧，所有这些单元都接收相同的输入 x。

1.3 Notes info.

课件/Slides	Lecture 3, P20
视频/Video	Lecture 3, 33:00
GitHub • 代码	实时在线查阅文档
Bilibili • 视频	中英字幕课程视频

Stanford University X ShowMeAI

这里我们想判断中心词 $Paris$ 是不是以命名实体。在这种情况下，我们很可能不仅想要捕捉窗口中单词的单词向量，还想要捕捉单词之间的一些其他交互，以便进行分类。例如，可能只有 $Museums$ 是第一个单词和 in 是第二个单词的时候， $Paris$ 才是命名实体。这样的非线性决策通常不能被直接提供给 Softmax 函数的输入捕获，而是需要第 1.2 节中讨论的中间层进行评分。因此，我们可以使用另一个矩阵 $U \in \mathbb{R}^{m\times 1}$ 与激活输出计算得到未归一化的得分用于分类任务：

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

其中 f 是激活函数（例如 sigmoid 函数）。

1.4 Maximum Margin Objective Function

类似很多的机器学习模型，神经网络需要一个优化目标函数，一个我们想要最小化或最大化的误差。这里我们讨论一个常用的误差度量方法：**maximum margin objective** 最大间隔目标函数。使用这个目标函数的背后的思想是保证对“真”标签数据的计算得分要比“假”标签数据的计算得分要高。

回到前面的例子，如果我们令“真”标签窗口 $Museums\ in\ Paris\ are\ amazing$ 的计算得分为 s，令“假”标签窗口 Not all museums in Paris 的计算得分为 s_c（下标 c 表示这个这个窗口 corrupt ）

然后，我们对目标函数最大化 $(s - s_c)$ 或者最小化 $(s_c - s)$。然而，我们修改目标函数来保证误差仅在 $s_c > s \Rightarrow (s_c - s) > 0$ 才进行计算。这样做的直觉是，我们只关心“正确”数据点的得分高于“错误”数据点，其余的都不重要。因此，当 $s_c > s$ 则误差为 $(s_c - s)$，否则为 0。因此，我们的优化的目标函数现在为：

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

然而，上面的优化目标函数是有风险的，因为它不能创造一个安全的间隔。我们希望“真”数据要比“假”数据的得分大于某个正的间隔 Δ。换而言之，我们想要误差在 $(s - s_c < \Delta)$ 就开始计算，而不是当 $(s - s_c < 0)$ 时就计算。因此，我们修改优化目标函数为：

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

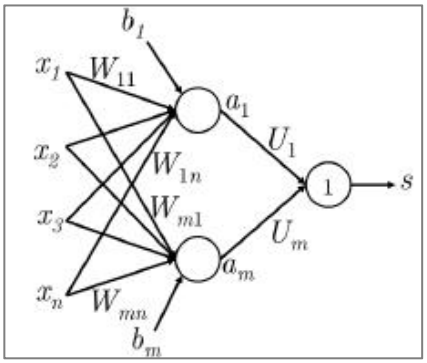
我们可以把这个间隔缩放使得 $\Delta = 1$，让其他参数在优化过程中自动进行调整，并且不会影响模型的表现。如果想更多地了解这

❐ 维度分析：
如果我们使用 4 维的词向量来表示每个单词并使用 5 个词的窗口，则输入是 $x \in \mathbb{R}^{20}$。
如果我们在隐藏层使用 8 个 sigmoid 单元和从激活函数中生成一个分数输出，其中 $W \in \mathbb{R}^{8\times 20}$，$b \in \mathbb{R}^8$，$U \in \mathbb{R}^{8\times 1}$，$s \in \mathbb{R}$。

1.4 Notes info.

课件/Slides	Lecture 3, P33
视频/Video	Lecture –, --:00
GitHub · 代码	实时在线查阅文档
Bilibili · 视频	中英字幕课程视频

Stanford University X ShowMeAI



❐ ↑图 4：这个图像捕捉了一个简单的前馈网络如何计算它的输出。

❐ 最大边际目标函数通常与支持向量机一起使用.

方面，可以去读一下 SVM 中的函数间隔和几何间隔中的相关内容。最后，我们定义在所有训练窗口上的优化目标函数为：

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

按照上面的公式有，$s_c = \mathbf{U}^T f(Wx_c + b)$ 和 $s = \mathbf{U}^T f(Wx + b)$。

1.5 Training with Backpropagation – Elemental

在这部分我们讨论当 1.4 节中讨论的损失函数 J 为正时，模型中不同参数时是如何训练的。如果损失为 0 时，那么不需要再更新参数。我们一般使用梯度下降（或者像 SGD 这样的变体）来更新参数，所以要知道在更新公式中需要的任意参数的梯度信息：

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

反向传播是一种利用微分链式法则来计算模型上任意参数的损失梯度的方法。为了更进一步理解反向传播，我们先看右图的一个简单的网络。

这里我们使用只有单个隐藏层和单个输出单元的神经网络。现在让我们先建立一些符号定义：

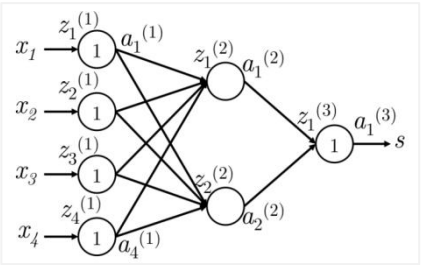
- x_i 是神经网络的输入
- s 是神经网络的输出
- 每层（包括输入和输出层）的神经元都接收一个输入和生成一个输出。第 k 层的第 j 个神经元接收标量输入 $z_j^{(k)}$ 和生成一个标量激活输出 $a_j^{(k)}$
- 我们把 $z_j^{(k)}$ 计算出的反向传播误差定义为 $\delta_j^{(k)}$
- 第 1 层是输入层，而不是第 1 个隐藏层。对输入层而言，$x_j^{(k)} = z_j^{(k)} = a_j^{(k)}$
- $W^{(k)}$ 是将第 k 层的输出映射到第 $k+1$ 层的输入的转移矩阵，因此将这个新的符号用在 Section 1.3 中的例子 $W^{(1)} = W$ 和 $W^{(2)} = U$。

现在开始反向传播：假设损失函数 $J = (1 + s_c - s)$ 为正值，我们想更新参数 $W_{14}^{(1)}$，我们看到 $W_{14}^{(1)}$ 只参与了 $z_1^{(2)}$ 和 $a_1^{(2)}$ 的计算。这点对于理解反向传播是非常重要的——反向传播的梯度只受它们所贡献的值的影响。$a_1^{(2)}$ 在随后的前向计算中和 $W_1^{(2)}$ 相乘计算得分。我们可以从最大间隔损失看到：

Notes info.

课件/Slides	Lecture 4, P12
视频/Video	Lecture 3, 57:00
GitHub · 代码	实时在线查阅文档
Bilibili · 视频	中英字幕课程视频

Stanford University X ShowMeAI



↑图 5：这是一个 4-2-1 神经网络，其中 k 层的神经元 j 接收输入 $z_j^{(k)}$，并产生激活输出 $a_j^{(k)}$。

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

为了简化我们只分析 $\frac{\partial s}{\partial W_{ij}^{(1)}}$。所以，

$$\frac{\partial s}{\partial W_{ij}^{(1)}} = \frac{\partial W^{(2)} a^{(2)}}{\partial W_{ij}^{(1)}} = \frac{\partial W_i^{(2)} a_i^{(2)}}{\partial W_{ij}^{(1)}} = W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}}$$

$$\Longrightarrow W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}} = W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} = W_i^{(2)} \frac{f\left(z_i^{(2)}\right)}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}}$$

$$= W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}}$$

$$= W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial}{\partial W_{ij}^{(1)}} \left(b_i^{(1)} + a_1^{(1)} W_{i1}^{(1)} + a_2^{(1)} W_{i2}^{(1)} + a_3^{(1)} W_{i3}^{(1)}\right.$$

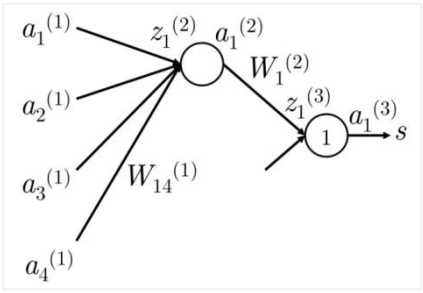
$$\left. + a_4^{(1)} W_{i4}^{(1)}\right) = W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial}{\partial W_{ij}^{(1)}} \left(b_i^{(1)} + \sum_k a_k^{(1)} W_{ik}^{(1)}\right)$$

$$= W_i^{(2)} f'\left(z_i^{(2)}\right) a_j^{(1)} = \delta_i^{(2)} a_j^{(1)}$$

其中， $a^{(1)}$ 指输入层的输入。我们可以看到梯度计算最后可以简化为 $\delta_i^{(2)} \cdot a_j^{(1)}$ ，其中 $\delta_i^{(2)}$ 本质上是第 2 层中第 i 个神经元反向传播的误差。 $a_j^{(1)}$ 与 W_{ij} 相乘的结果，输入第 2 层中第 i 个神经元中。

我们以右图为例，让我们从“误差共享/分配”的来阐释一下反向传播，现在我们要更新 $W_{14}^{(1)}$：

1. 我们从 $a_1^{(3)}$ 的 1 的误差信号开始反向传播。
2. 然后我们把误差与将 $z_1^{(3)}$ 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等于 1，则误差仍然为 1。所以有 $\delta_1^{(3)} = 1$。
3. 这里误差信号 1 已经到达 $z_1^{(3)}$。我们现在需要分配误差信号使得误差的“公平共享”到达 $a_1^{(2)}$。
4. 现在在 $a_1^{(2)}$ 的误差为 $\delta_1^{(3)} \times W_1^{(2)} = W_1^{(2)}$（在 $z_1^{(3)}$ 的误差信号为 $\delta_1^{(3)}$）。因此在 $a_1^{(2)}$ 的误差为 $W_1^{(2)}$。
5. 与第 2 步的做法相同，我们在将 $z_1^{(2)}$ 映射到 $a_1^{(2)}$ 的神经元上移动误差，将 $a_1^{(2)}$ 与局部梯度相乘，这里的局部梯度为 $f'(z_1^{(2)})$。
6. 因此在 $z_1^{(2)}$ 的误差是 $f'(z_1^{(2)}) W_1^{(2)}$，我们将其定义为 $\delta_1^{(2)}$。
7. 最后，我们通过将上面的误差与参与前向计算的 $a_4^{(1)}$ 相乘，把误差的“误差共享”分配到 $W_{14}^{(1)}$。



↑图 6：此子网显示了更新 $W_{ij}^{(1)}$ 所需的网络的相关部分。

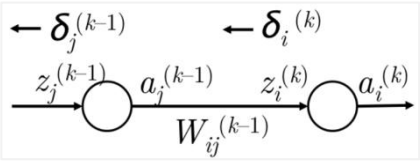
8. 所以，对于 $W_{14}^{(1)}$ 的梯度损失可以计算为 $a_4^{(1)} f'(z_1^{(2)}) W_1^{(2)}$ 。

注意我们使用这个方法得到的结果是和之前微分的方法的结果是完全一样的。因此，计算网络中的相应参数的梯度误差既可以使用链式法则也可以使用误差共享和分配的方法——这两个方法能得到相同结果，但是多种方式考虑它们可能是有帮助的。

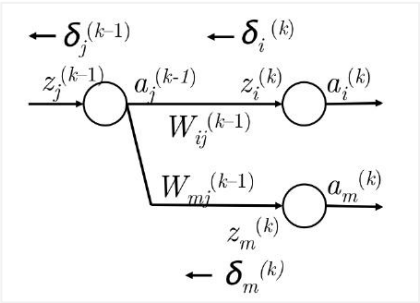
偏置更新：偏置项（例如 $b_1^{(1)}$ ）和其他权值在数学形式是等价的，只是在计算下一层神经 $z_1^{(2)}$ 元输入时相乘的值是常量 1 。因此在第 k 层的第 i 个神经元的偏置的梯度时 $\delta_i^{(k)}$ 。例如在上面的例子中，我们更新的是 $b_1^{(1)}$ 而不是 $W_{14}^{(1)}$ ，那么这个梯度为 $f'(z_1^{(2)}) W_1^{(2)}$ 。

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

- 我们有从 $z_i^{(k)}$ 向后传播的误差 $\delta_i^{(k)}$ ，如图 7 所示
- 我们通过把 $\delta_i^{(k)}$ 与路径上的权值 $W_{ij}^{(k-1)}$ 相乘，将这个误差反向传播到 $a_j^{(k-1)}$ 。
- 因此在 $a_j^{(k-1)}$ 接收的误差是 $\delta_i^{(k)} W_{ij}^{(k-1)}$ 。
- 然而， $a_j^{(k-1)}$ 在前向计算可能出右图的情况，会参与下一层中的多个神经元的计算。那么第 k 层的第 m 个神经元的误差也要使用上一步方法将误差反向传播到 $a_j^{(k-1)}$ 上。
- 因此现在在 $a_j^{(k-1)}$ 接收的误差是 $\delta_i^{(k)} W_{ij}^{(k-1)} + \delta_m^{(k)} W_{mj}^{(k-1)}$ 。
- 实际上，我们可以把上面误差和简化为 $\sum_i \delta_i^{(k)} W_{ij}^{(k-1)}$ 。
- 现在我们有在 $a_j^{(k-1)}$ 正确的误差，然后将其与局部梯度 $f'(z_j^{(k-1)})$ 相乘，把误差信息反向传到第 $k-1$ 层的第 j 个神经元上。
- 因此到达 $z_j^{(k-1)}$ 的误差为 $f'(z_j^{(k-1)}) \sum_i \delta_i^{(k)} W_{ij}^{(k-1)}$ 。



❐ ↑图 7：从 $\delta^{(k)}$ 到 $\delta^{(k-1)}$ 的传播误差



❐ ↑图 8：从 $\delta^{(k)}$ 到 $\delta^{(k-1)}$ 的传播误差

1.6 Training with Backpropagation – Vectorized

到目前为止，我们讨论了对模型中的给定参数计算梯度的方法。这里会一般泛化上面的方法，让我们可以直接一次过更新权值矩阵和偏置向量。注意这只是对上面模型的简单地扩展，这将有助于更好理解在矩阵-向量级别上进行误差反向传播的方法。

对更定的参数 $W_{ij}^{(k)}$ ，我们知道它的误差梯度是 $\delta_j^{(k+1)} \cdot a_j^{(k)}$ 。其中 $W^{(k)}$ 是将 $a^{(k)}$ 映射到 $z^{(k+1)}$ 的矩阵。因此我们可以确定整个矩阵 $W^{(k)}$ 的梯度误差为：

1.6 Notes info.

课件/Slides	Lecture –, P–
视频/Video	Lecture3, 64:00
GitHub • 代码	实时在线查阅文档
Bilibili • 视频	中英字幕课程视频

Stanford University X ShowMeAI

$$\nabla_{W^k} = \begin{bmatrix} \delta_1^{(k+1)} a_1^{(k)} & \delta_1^{(k+1)} a_2^{(k)} & \cdots \\ \delta_2^{(k+1)} a_1^{(k)} & \delta_2^{(k+1)} a_2^{(k)} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \delta^{(k+1)} a_1^{(k)T}$$

❐ 误差按以下方式从（k+1）层传播到（k）层：

因此我们可以将整个矩阵形式的梯度写为在矩阵中的反向传播的误差向量和前向激活输出的**外积**。

$$\delta_i^{(k)} = f'(z^{(k)}) \circ \left(W^{(k)T} \delta^{(k+1)}\right)$$

当然，这是假设在正向传播中，信号$z^{(k)}$首先经过激活神经元 f 产生激活 $a^{(k)}$，然后通过传递矩阵 $W^{(k)}$ 线性组合产生 $z^{(k+1)}$ 。

现在我们来看看如何能够计算误差向量 $\delta^{(k+1)}$ 。我们从上面的例子中有， $\delta_i^{(k)} = f'(z_j^{(k)}) \sum_i \delta_i^{(k+1)} W_{ij}^{(k)}$ 。可以简单地改写为矩阵的形式：

$$\delta_i^{(k)} = f'(z^{(k)}) \circ \left(W^{(k)T} \delta^{(k+1)}\right)$$

在上面的公式中 $\circ$ 运算符是表示向量之间对应元素的相乘（ $\mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^N$ ）。

计算效率：在探索了 element-wise 的更新和 vector-wise 的更新之后，必须认识到在科学计算环境中，如 MATLAB 或 Python（使用 Numpy / Scipy 库），向量化运算的计算效率是非常高的。因此在实际中应该使用向量化运算。

此外，我们也要减少反向传播中的多余的计算——例如，注意到 $\delta^{(k)}$ 是直接依赖在 $\delta^{(k+1)}$ 上。所以我们要保证使用 $\delta^{(k+1)}$ 更新 $W^{(k)}$ 时，要保存 $\delta^{(k+1)}$ 用于后面 $\delta^{(k)}$ 的计算-然后计算 $(k-1)\ldots(1)$ 层的时候重复上述的步骤。这样的递归过程是使得反向传播成为计算上可负担的过程。

2. Neural Networks: Tips and Tricks

2.1 Gradient Check

在上一部分中，我们详细地讨论了如何用基于微积分的方法计算神经网络中的参数的误差梯度 / 更新。这里我们介绍一种用数值近似这些梯度的方法——虽然在计算上的低效不能直接用于训练神经网络，这种方法可以非常准确地估计任何参数的导数；因此，它可以作为对导数的正确性的有用的检查。给定一个模型的参数向量 θ 和损失函数 J，围绕 θ_i 的数值梯度由 central difference formula 得出：

$$f'(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}$$

其中 ϵ 是一个很小的值（一般约为 $1e^{-5}$）。当我们使用 $+\epsilon$ 扰动参数 θ 的第 i 个元素时，就可以在前向传播上计算误差 $J(\theta^{(i+)})$。相似地，当我们使用 $-\epsilon$ 扰动参数 θ 的第 i 个元素时，就可以在前向传播上计算误差 $J(\theta^{(i-)})$。因此，计算两次前向传播，我们可以估计在模型中任意给定参数的梯度。我们注意到数值梯度的定义和导数的定义很相似，其中，在标量的情况下：

$$f'(\theta) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

当然，还是有一点不同——上面的定义仅仅在正向扰动 x 计算梯度。虽然是可以用这种方式定义数值梯度，但在实际中使用 central difference formula 常常可以更准确和更稳定，因为我们在两个方向都对参数扰动。为了更好地逼近一个点附近的导数/斜率，我们需要在该点的左边和右边检查函数 f' 的行为。也可以使用泰勒定理来表示 central difference formula 有 ϵ^2 比例误差，这相当小，而导数定义更容易出错。

现在你可能会产生疑问，如果这个方法这么准确，为什么我们不用它而不是用反向传播来计算神经网络的梯度？

这是因为效率的问题——每当我们想计算一个元素的梯度，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 这样的优化技术中，我们需要通过数千次的迭代来计算梯度，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：

2.1 Notes info.

课件/Slides	Lecture 4, P47
视频/Video	Lecture 4, 57:30
GitHub • 代码	实时在线查阅文档
Bilibili • 视频	中英字幕课程视频

Stanford University X ShowMeAI

❐ 梯度检查是比较分析梯度和数值梯度的好方法．分析梯度应接近，数值梯度可使用以下公式计算：

$$f'(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}$$

$J(\theta^{(i+)})$和 $J(\theta^{(i-)})$可以使用两个正向过程进行评估．

```python
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
  """

  f(x) = 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)
    # restore to previous value (very important!)
    x[ix] = old_value

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

2.2 Regularization

和很多机器学习的模型一样，神经网络很容易过拟合，这令到模型在训练集上能获得近乎完美的表现，但是却不能泛化到测试集上。一个常见的用于解决过拟合（“高方差问题”）的方法是使用 L2 正则化。我们只需要在损失函数 J 上增加一个正则项，现在的损失函数如下：

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

在上面的公式中，$||W^{(i)}||_F$ 是矩阵 $W^{(i)}$（在神经网络中的第 i 个权值矩阵）的 Frobenius 范数，λ 是超参数控制损失函数中的权值的大小。

$$||U||_F = \sqrt{\sum_i \sum_l U_{il}^2}$$

Notes info.

课件/Slides	Lecture 4, P50
视频/Video	Lecture 4, 61:30
GitHub · 代码	实时在线查阅文档
Bilibili · 视频	中英字幕课程视频

Stanford University X ShowMeAI

❐ 矩阵 U 的 Frobenius 范数的定义

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（让权值的数值分配更加均衡，防止出现部分权值特别大的情况）。由于 Frobenius 范数的二次的性质（计算矩阵的元素的平方和），L2 正则项有效地降低了模型的灵活性和因此减少出现过拟合的可能性。增加这样一个约束可以使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 的值。选择一个合适的 λ 值是很重要的，并且需要通过超参数调整来选择。λ 的值太大会令很多权值都接近于 0，则模型就不能在训练集上学习到有意义的东西，经常在训练、验证和测试集上的表现都非常差。λ 的值太小，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，不会计算入损失项中——尝试去思考一下为什么。

❐ 为何在损失项中不计算偏置项?

偏置项在模型中仅仅是偏移的关系，使用少量的数据就能拟合到这项，而且从经验上来说，偏置值的大小对模型表现没有很显著的影响，因此不需要正则化偏置项.

[深度学习里面的偏置为什么不加正则?]

(https://www.zhihu.com/question/66894061)

实际上，有时也会使用其他类型的正则化，例如 L1 正则化，它对参数元素的绝对值（而不是平方）求和。然而，这在实践中不太常用，因为它会导致参数权重稀疏。在下一节中，我们将讨论 dropout，它有效地作为另一种正则化形式，通过在前向传递中随机丢弃（即设置为零）神经元。

下面摘录已有的三条回答

- 首先正则化主要是为了防止过拟合，而过拟合一般表现为模型对于输入的微小改变产生了输出的较大差异，这主要是由于有些参数 w 过大的关系，通过对 ||W|| 进行惩罚，可以缓解这种问题。而如果对||b||进行惩罚，其实是没有作用的，因为在对输出结果的贡献中，参数 b 对于输入的改变是不敏感的，不管输入改变是大还是小，参数 b 的贡献就只是加个偏置而已。

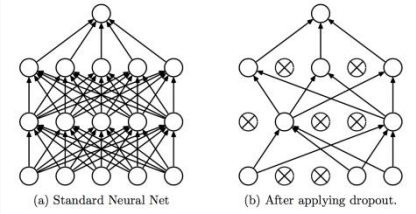
  举个例子，如果你在训练集中，w 和 b 都表现得很好，但是在测试集上发生了过拟合，b 是不背这个锅的，因为它对于所有的数据都是一视同仁的（都只是给它们加个偏置），要背锅的是 w，因为它会对不同的数据产生不一样的加权。或者说，模型对于输入的微小改变产生了输出的较大差异，这是因为模型的“曲率”太大，而模型的曲率是由 w 决定的，b 不贡献曲率（对输入进行求导，b 是直接约掉的）。

- 从贝叶斯的角度来讲，正则化项通常都包含一定的先验信息，神经网络倾向于较小的权重以便更好地泛化，但是对偏置就没有这样一致的先验知识。另外，很多神经网络更倾向于区分方向信息（对应于权重），而不是位置信息（对应于偏置），所以对偏置加正则化项对控制过拟合的作用是有限的，相反很可能会因为不恰当的正则强度影响神经网络找到最优点。

- 过拟合会使得模型对异常点很敏感，即准确插入异常点，导致拟合函数中的曲率很大（即函数曲线的切线斜率非常高），而偏置对模型的曲率没有贡献（对多项式模型进行求导，为 W 的线性加和），所以正则化他们也没有什么意义。

- 有时候我们会用到其他类型的正则项，例如 L1 正则项，它将参数元素的绝对值全部加起来–然而，在实际中很少会用 L1 正则项，因为会令权值参数变得稀疏。在下一部分，我们讨论 dropout，这是另外一种有效的正则化方法，通过在前向传播过程随机将神经元设为 0

- Dropout 实际上是通过在每次迭代中忽略它们的权值来实现“冻结”部分 unit。这些“冻结”的 unit 不是把它们设为 0，而是对于该迭代，网络假定它们为 0。“冻结”的 unit 不会为此次迭代更新。



❐ ↑Dropout applied to an artificial neural network. Image credits to Srivastava et al.【Dropout 应用于人工神经网络，图像来源于 Srivastava 等人。】

2.3 Dropout

Dropout 是一个非常强大的正则化技术，是 Srivastava 在论文《Dropout: A Simple Way to Prevent Neural Networks from Overfitting》中首次提出，右图展示了 dropout 如何应用在神经网络上。

这个想法是简单而有效的——训练过程中，在每次的前向 / 反向传播中我们按照一定概率 $(1-p)$ 随机地“ drop ”一些神经元子集（或者等价的，我们保持一定概率 p 的神经元是激活的）。然后，在测试阶段，我们将使用全部的神经元来进行预测。使用 Dropout 神经网络一般能从数据中学到更多有意义的信息，更少出现过拟合和通常在现今的任务上获得更高的整体表现。这种技术应该如此有效的一个直观原因是， dropout 本质上作的是一次以指数形式训练许多较小的网络，并对其预测进行平均。

实际上，我们使用 dropout 的方式是我们取每个神经元层的输出 h，并保持概率 p 的神经元是激活的，否则将神经元设置为 0。然后，在反向传播中我们仅对在前向传播中激活的神经元回传梯度。最后，在测试过程，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，有一个关键的微妙之处，为了使 dropout 有效地工作，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有很大的不同，网络的表现已经不再明确了。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（该值为 p）。

以下源于 《神经网络与深度学习》[https://nndl.github.io/] P190

- 目的：缓解过拟合问题，一定程度上达到正则化的效果
- 效果：减少下层节点对其的依赖，迫使网络去学习更加鲁棒的特征

集成学习的解释

- 每做一次丢弃，相当于从原始的网络中采样得到一个子网络。如果一个神经网络有 n 个神经元，那么总共可以采样出 2^n 个子网络。每次迭代都相当于训练一个不同的子网络，这些子网络都共享原始网络的参数。那么，最终的网络可以近似看作是集成了指数级个不同网络的组合模型。

贝叶斯学习的解释

- 丢弃法也可以解释为一种贝叶斯学习的近似。用 $y = f(x, \theta)$ 来表示要学习的神经网络，贝叶斯学习是假设参数 θ 为随机向量，并且先验分布为 $q(\theta)$，贝叶斯方法的预测为

$$\begin{aligned}\mathbb{E}_{q(\theta)}[y] &= \int_q f(\mathbf{x}, \theta) q(\theta) d\theta \\ &\approx \frac{1}{M} \sum_{m=1}^{M} f(\mathbf{x}, \theta_m)\end{aligned}$$

- 其中 $f(x, \theta_m)$ 为第 m 次应用丢弃方法后的网络，其参数 θ_m 为对全部参数 θ 的一次采样。

RNN 中的变分 Dropout Variational Dropout

- Dropout 一般是针对神经元进行随机丢弃，但是也可以扩展到对神经元之间 的连接进行随机丢弃，或每一层进行随机丢弃。
- 在 RNN 中，不能直接对每个时刻的隐状态进行随机 丢弃，这样会损害循环网络在时间维度上记忆能力。
- 一种简单的方法是对非时间维度的连接（即非循环连接）进行随机丢失。如图所示，虚线边表示进行随机丢弃，不同的颜色表示不同的丢弃掩码。

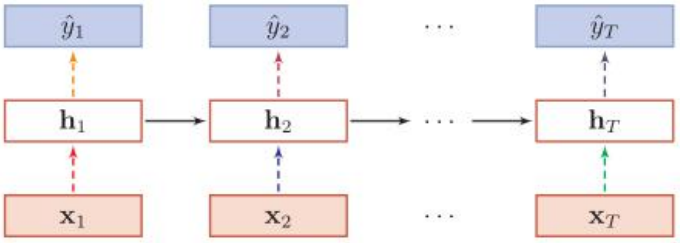


图 7.11 针对非循环连接的丢弃法

- 然而根据贝叶斯学习的解释，丢弃法是一种对参数θ的采样。每次采样的参数需要在每个时刻保持不变。因此，在对循环神经网络上使用丢弃法时，需要对参数矩阵的每个元素进行随机丢弃，并在所有时刻都使用相同的丢弃掩码。这种方法称为变分丢弃法（Variational Dropout）。图 7.12 给出了变分丢弃法的示例，相同颜色表示使用相同的丢弃掩码。

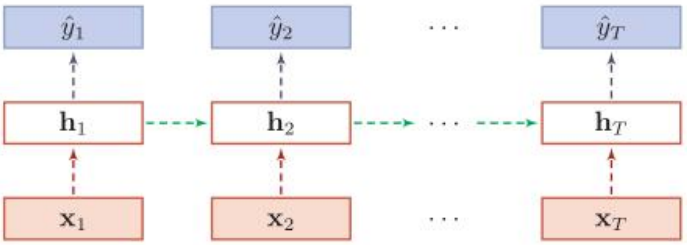


图 7.12 变分丢弃法

2.4 Neuron Units

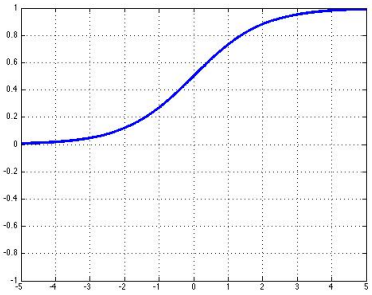
到目前为止，我们讨论了含有 sigmoidal neurons 的非线性分类的神经网络。但是在很多应用中，使用其他激活函数可以设计更好的神经网络。下面列出一些常见的激活函数和激活函数的梯度定义，它们可以和前面讨论过的 sigmoidal 函数互相替换。

Sigmoid：这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 $\sigma(z) \in (0,1)$

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



❐ ↑Figure 9: The response of a sigmoid nonlinearity【图 9:sigmoid 非线性的响应】

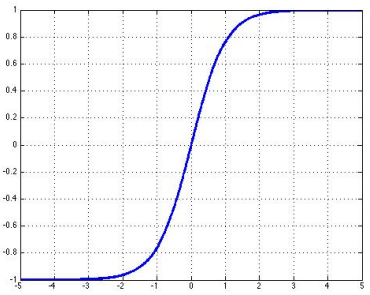
Tanh：tan$h$ 函数是 sigmoid 函数之外的另一个选择，在实际中它能更快地收敛。tan$h$ 和 sigmoid 的主要不同在于 tan$h$ 的输出范围在 −1 到 1，而 sigmoid 的输出范围在 0 到 1。

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

其中 $tanh(z) \in (-1,1)$。

$tanh(z)$ 的梯度为：

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



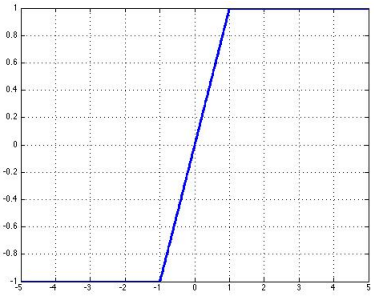
❐ ↑Figure 10: The response of a tanh nonlinearity【图 10:tanh 非线性的响应】

Hard tanh：有时候 hard tan$h$ 函数有时比 tan$h$ 函数的选择更为优先，因为它的计算量更小。然而当 z 的值大于 1 时，函数的数值会饱和（如右图所示会恒等于 1）。hard tan$h$ 激活函数为：

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

hard tan$h$ 这个函数的微分也可以用分段函数的形式表示：

$$hardtanh(z) \quad = \quad \begin{cases} 1 & : -1 \le z \le 1 \\ 0 & : otherwise \end{cases}$$



❐ ↑Figure 11: The response of a hard tanh nonlinearity【图 11：hard tanh 非线性的响应】

Soft sign：soft sign 函数是另外一种非线性激活函数，它可以是 tan$h$ 的另外一种选择，因为它和 hard clipped functions 一样不会过早地饱和：

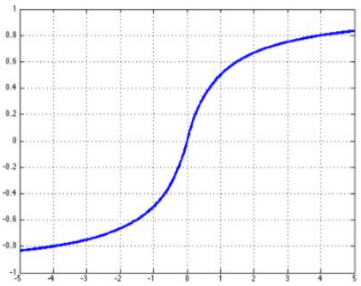
$$softsign(z) = \frac{z}{1 + |z|}$$

soft sign 函数的微分表达式为：

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

其中 sgn 是符号函数，根据 z 的符号返回 1 或者 -1。

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



❐ ↑Figure 12: The response of a soft sign nonlinearity【图 12：软符号非线性的响应】

ReLU：ReLU（Rectified Linear Unit）函数是激活函数中的一个常见的选择，当 z 的值特别大的时候它也不会饱和。在计算机视觉应用中取得了很大的成功：

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

ReLU 函数的微分是一个分段函数：

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

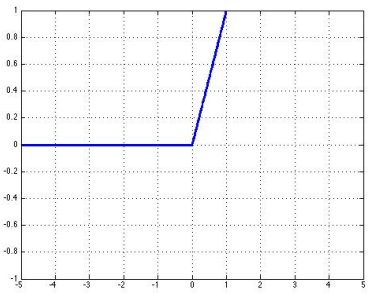
Leaky ReLU：传统的 ReLU 单元当 z 的值小于 0 时，是不会反向传播误差 leaky ReLU 改善了这一点，当 z 的值小于 0 时，仍然会有一个很小的误差反向传播回去。

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

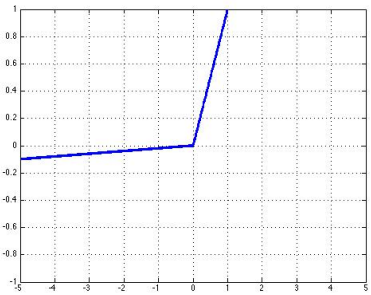
其中 $0 < k < 1$

leaky ReLU 函数的微分是一个分段函数：

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



↑Figure 13: The response of a ReLU nonlinearity【图 13:ReLU 非线性的响应】



↑Figure 14: The response of a leaky ReLU nonlinearity【图 14：泄漏 ReLU 非线性的响应】

2.5 Data Preprocessing

与机器学习模型的一般情况一样，确保模型在当前任务上获得合理性能的一个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的技术。

Mean Subtraction

给定一组输入数据 X，一般把 X 中的值减去 X 的平均特征向量来使数据零中心化。在实践中很重要的一点是，只计算训练集的平均值，而且在训练集，验证集和测试集都是减去同一平均值。

Normalization

另外一个常见的技术（虽然没有 mean Subtraction 常用）是将每个输入特征维度缩小，让每个输入特征维度具有相似的幅度范围。这是很有用的，因此不同的输入特征是用不同“单位”度量，但是最初的时候我们经常认为所有的特征同样重要。实现方法是将特征除以它们各自在训练集中计算的标准差。

Whitening

相比上述的两个方法，whitening 没有那么常用，它本质上是数据经过转换后，特征之间相关性较低，所有特征具有相同的方差（协方差阵为 I）。首先对数据进行 Mean Subtraction 处

理，得到 X' 。然后我们对 X' 进行奇异值分解得到矩阵 U，S，V，计算 UX' 将 X' 投影到由 U 的列定义的基上。我们最后将结果的每个维度除以 S 中的相应奇异值，从而适当地缩放我们的数据（如果其中有奇异值为 0，我们就除以一个很小的值代替）。

2.6 Parameter Initialization

让神经网络实现最佳性能的关键一步是以合理的方式初始化参数。一个好的起始方法是将权值初始化为通常分布在 0 附近的很小的随机数–在实践中效果还不错。在论文 Understanding the difficulty of training deep feedforward neural networks (2010), Xavier 研究不同权值和偏置初始化方案对训练动力（ training dynamics ）的影响。实验结果表明，对于 sigmoid 和 tanh 激活单元，当一个权值矩阵 $W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$ 以如下的均匀分布的方式随机初始化，能够实现更快的收敛和得到更低的误差：

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

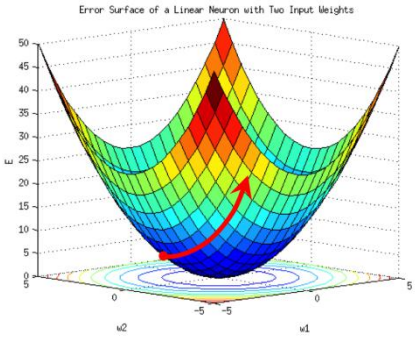
其中 $n^{(l)}$ 是 $W(\text{fan}-\text{in})$ 的输入单元数， $n^{(l+1)}$ 是 $W(\text{fan}-\text{out})$ 的输出单元数。在这个参数初始化方案中，偏置单元是初始化为 0 。这种方法是尝试保持跨层之间的激活方差以及反向传播梯度方差。如果没有这样的初始化，梯度方差（当中含有纠正信息）通常随着跨层的反向传播而衰减。

2.7 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：

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

你可能会认为如果要更快地收敛，我们应该对 α 取一个较大的值——然而，在更快的收敛速度下并不能保证更快的收敛。实际上，如果学习率非常高，我们可能会遇到损失函数难以收敛的情况，因为参数更新幅度过大，会导致模型越过凸优化的极小值点，如右图所示。在非凸模型中（我们很多时候遇到的模型都是非凸），高学习率的结果是难以预测的，但是损失函数难以收敛的可能性是非常高的。



↑ Figure 15: Here we see that updating parameter w2 with a large learning rate can lead to divergence of the error.【图 15：在这里，我们看到用大的学习率更新参数 w2 会导致错误的发散。】

避免损失函数难以收敛的一个简答的解决方法是使用一个很小的学习率，让模型谨慎地在参数空间中迭代——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。

深度学习系统中最消耗计算资源的是训练阶段，一些研究已在尝试提升设置学习率的新方法。例如， Ronan Collobert 通过取 $\text{fan}-\text{in}$ 的神经元 $(n^{(l)})$ 的平方根的倒数来缩放权值 W_{ij}（$W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$）的学习率。

还有其他已经被证明有效的技术-这个方法叫 annealing 退火，在多次迭代之后，学习率以以下方式降低：保证以一个高的的学习率开始训练和快速逼近最小值；当越来越接近最小值时，开始降低学习率，让我们可以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 次的迭代学习后，通过一个因子 x 来降低学习率 α 。

指数衰减也是很常见的方法，在 t 次迭代后学习率变为 $\alpha(t) = \alpha_0 e^{-kt}$，其中 α_0 是初始的学习率和 k 是超参数。

还有另外一种方法是允许学习率随着时间减少：

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

在上述的方案中， α_0 是一个可调的参数，代表起始的学习率。τ 也是一个可调参数，表示学习率应该在该时间点开始减少。在实际中，这个方法是很有效的。在下一部分我们讨论另外一种不需要手动设定学习率的自适应梯度下降的方法。

2.8 Momentum Updates

动量方法，灵感来自于物理学中的对动力学的研究，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：

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

2.9 Adaptive Optimization Methods

AdaGrad 是标准的随机梯度下降(SGD)的一种实现，但是有一点关键的不同：对每个参数学习率是不同的。每个参数的学习率取决于每个参数梯度更新的历史，参数的历史更新越小，就使用更大的学习率加快更新。即，过去没有更新太大的参数，现在更有可能有更高的学习率。

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i} \text{ where } g_{t,i} = \frac{\partial}{\partial \theta_i^t} J_t(\theta)$$

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

```
# 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：

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

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

- RMSProp 是利用平方梯度的移动平局值，是 AdaGrad 的一个变体——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。

2.10 More reference

如果希望了解以上的梯度优化算法的具体细节，可以阅读这篇文章：An overview of gradient descent optimization algorithms）。

机器学习	深度学习	自然语言处理	计算机视觉	知识图谱
Machine Learning	Deep Learning	Natural Language Processing	Computer Vision	Knowledge Graphs
Stanford · CS229	Stanford · CS230	Stanford · CS224n	Stanford · CS231n	Stanford · CS520

 # 系列内容 Awesome AI Courses Notes Cheatsheets

图机器学习	深度强化学习	自动驾驶
Machine Learning with Graphs	Deep Reinforcement Learning	Deep Learning for Self-Driving Cars
Stanford · CS224W	UCBerkeley · CS285	MIT · 6.S094
...	...	...

是 ShowMeAI 资料库的分支系列，覆盖最具知名度的 TOP20+门 AI 课程，旨在为读者和学习者提供一整套高品质中文学习笔记和速查表。

斯坦福大学(Stanford University) Natural Language Processing with Deep Learning (CS224n) 课程，是本系列的第三门产出。

课程版本为 2019 Winter，核心深度内容(transformer、bert、问答、摘要、文本生成等)在当前(2021 年)工业界和研究界依旧是前沿的方法。最新版课程的笔记生产已在规划中，也敬请期待。

笔记内容经由深度加工整合，以 5 个部分构建起完整的“CS224n 内容世界”，并依托 GitHub 创建了汇总页。快扫描二维码，跳转进入吧！有任何建议和反馈，也欢迎通过下方渠道和我们联络 (*￣3￣)~