神经网络和语言模型

肖桐　朱靖波

xiaotong@mail.neu.edu.cn
zhujingbo@mail.neu.edu.cn

东北大学 自然语言处理实验室
http://www.nlplab.com

为什么要谈神经网络

- 近些年**深度学习（Deep Learning）**体现了巨大的潜力
  - 席卷了包括机器翻译在内的很多NLP任务
  - 已经成为了NLP中方法的新范式
  - 衍生出**神经机器翻译**等新一代方法（下一章内容）

为什么要谈神经网络

- 近些年**深度学习（Deep Learning）**体现了巨大的潜力
  - 席卷了包括机器翻译在内的很多NLP任务
  - 已经成为了NLP中方法的新范式
  - 衍生出**神经机器翻译**等新一代方法（下一章内容）

- 人工神经网络（**Artificial Neural Network**）是深度学习的实践基础

神经网络和深度学习的概念（1940s-1970s）

- **神经网络**最早出现在控制论中（Cybernetics），随后更多地在连接主义（Connectionism）中被提及
  - **最初的想法**：模拟大脑的生物学习机制进行计算机建模

图片引自《Deep Learning》

神经网络和深度学习的概念（1940s-1970s）

- **神经网络**最早出现在控制论中（Cybernetics），随后更多地在连接主义（Connectionism）中被提及
  - **最初的想法**：模拟大脑的生物学习机制进行计算机建模
  - 比如使用线性加权函数来描述输入**x**和结果**y**之间的联系

$$f(\mathbf{x}, \mathbf{w}) = x_1 \cdot w_1 + \ldots + x_n \cdot w_n$$

其中**w**是权重。这类模型也影响了随机梯度下降等现在机器学习方法的发展。

图片引自《Deep Learning》

神经网络和深度学习的概念（1940s-1970s）

- **神经网络**最早出现在控制论中（Cybernetics），随后更多地在连接主义（Connectionism）中被提及
  - **最初的想法**：模拟大脑的生物学习机制进行计算机建模
  - 比如使用线性加权函数来描述输入**x**和结果**y**之间的联系

$$f(\mathbf{x}, \mathbf{w}) = x_1 \cdot w_1 + ... + x_n \cdot w_n$$

    其中**w**是权重。这类模型也影响了随机梯度下降等现在机器学习方法的发展。
  - 这类方法的局限也很明显，无法描述非线性问题，如著名的异或函数（XOR）学习问题

图片引自《Deep Learning》

神经网络和深度学习的发展（1980s-1990s）

- 现在，生物学属性已经不是神经网络的唯一灵感来源。深度学习也进入了新的发展阶段。两类思潮影响巨大：

神经网络和深度学习的发展（1980s-1990s）

- 现在，生物学属性已经不是神经网络的唯一灵感来源。深度学习也进入了新的发展阶段。两类思潮影响巨大：
  - **连接主义（Connectionism）**。在认知学科中，早期的符号主义（Symbolicism）很难解释大脑如何使用神经元进行推理。连接主义的核心思想是："大量简单的计算单元连接到一起可以实现智能行为"。
    这也推动了反向传播等训练多层神经网络方法的应用，并发展了包括长短时记忆模型在内的经典建模方法。

神经网络和深度学习的发展（1980s-1990s）

- 现在，生物学属性已经不是神经网络的唯一灵感来源。深度学习也进入了新的发展阶段。两类思潮影响巨大：
  - **连接主义（Connectionism）**。在认知学科中，早期的符号主义（Symbolicism）很难解释大脑如何使用神经元进行推理。连接主义的核心思想是："大量简单的计算单元连接到一起可以实现智能行为"。
    这也推动了反向传播等训练多层神经网络方法的应用，并发展了包括长短时记忆模型在内的经典建模方法。
  - **分布式表示（Distributed representation）**：一个复杂系统的任何部分的输入都应该是多个特征共同表示的结果。比如，一个单词并非一个词条，而是由成百上千个特征共同描述出来，而每个特征都描述了这个词的"某个"方面。

神经网络和深度学习的发展（1980s-1990s）

- 现在，生物学属性已经不是神经网络的唯一灵感来源。深度学习也进入了新的发展阶段。两类思潮影响巨大：
  - **连接主义（Connectionism）**。在认知学科中，早期的符号主义（Symbolicism）很难解释大脑如何使用神经元进行推理。连接主义的核心思想是："大量简单的计算单元连接到一起可以实现智能行为"。
    这也推动了反向传播等训练多层神经网络方法的应用，并发展了包括长短时记忆模型在内的经典建模方法。
  - **分布式表示（Distributed representation）**：一个复杂系统的任何部分的输入都应该是多个特征共同表示的结果。比如，一个单词并非一个词条，而是由成百上千个特征共同描述出来，而每个特征都描述了这个词的"某个"方面。
- 遗憾的是，上世纪90年代后期，在很多应用中人们对神经网络方法期望过高，但是结果并没有达到预期。特别是，核方法、图模型等机器学习方法取得了很好的效果，神经网络研究进入又一次低谷。

第三次浪潮（2000s-now）

- **深度学习的爆发**源于2006年Hinton等人成功训练了一个深度信念网络（deep belief network）。之后，深度学习的浪潮逐步席卷了机器学习及人工智能应用领域，延续至今。现代深度学习的成功有三方面原因：

1. **模型和算法**的完善与改进
2. **并行计算能力**的提升使大规模实践变为了可能
3. 以Hinton等人为代表的学者的**坚持与持续投入**

第三次浪潮（2000s-now）

- **深度学习的爆发**源于2006年Hinton等人成功训练了一个深度信念网络（deep belief network）。之后，深度学习的浪潮逐步席卷了机器学习及人工智能应用领域，延续至今。现代深度学习的成功有三方面原因：
  1. **模型和算法**的完善与改进
  2. **并行计算能力**的提升使大规模实践变为了可能
  3. 以Hinton等人为代表的学者的**坚持与持续投入**
- **从应用的角度**，数据量的快速提升和模型容量的增加也为深度学习的成功提供了条件

第三次浪潮（2000s-now）

- **深度学习的爆发**源于2006年Hinton等人成功训练了一个深度信念网络（deep belief network）。之后，深度学习的浪潮逐步席卷了机器学习及人工智能应用领域，延续至今。现代深度学习的成功有三方面原因：
  1. **模型和算法**的完善与改进
  2. **并行计算能力**的提升使大规模实践变为了可能
  3. 以Hinton等人为代表的学者的**坚持与持续投入**
- **从应用的角度**，数据量的快速提升和模型容量的增加也为深度学习的成功提供了条件

端到端学习

- 深度神经网络给我们提供了一种机制，可以直接从学习输入到输出的关系，称之为端到端学习

端到端学习

- 深度神经网络给我们提供了一种机制，可以直接从学习输入到输出的关系，称之为端到端学习
  - **基于特征工程的方法**：需要大量人工定义的特征，这个过程往往会带来对问题的隐含假设

端到端学习

- 深度神经网络给我们提供了一种机制，可以直接从学习输入到输出的关系，称之为端到端学习
  - **基于特征工程的方法**：需要大量人工定义的特征，这个过程往往会带来对问题的隐含假设
  - **基于端到端学习的方法**：没有人工定义的特征，整个过程完全由神经网络建模

Feature Engineering + Machine Learning

VS.

Deep Learning (End-to-End Learning)

深度学习的表现 - 以语言建模为例

- 比如，在语言建模（LM）任务上，基于神经网络和深度学习的方法体现了巨大优势，在PTB数据上PPL值已经得到惊人的下降（PPL越低越好）
  - 传统n元语法模型面临数据稀疏等问题

模型	作者	年份	PPL
3-gram LM	Brown et al.	1992	178.0

深度学习的表现 - 以语言建模为例

- 比如，在语言建模（LM）任务上，基于神经网络和深度学习的方法体现了巨大优势，在PTB数据上PPL值已经得到惊人的下降（PPL越低越好）
  - 传统n元语法模型面临数据稀疏等问题
  - 神经语言模型可以更好地描述序列生成问题

模型	作者	年份	PPL
3-gram LM	Brown et al.	1992	178.0
Feed-forward Neural LM	Bengio et al.	2003	162.2
Recurrent NN-based LM	Mikolov et al.	2010	124.7
Recurrent NN-LDA	Mikolov et al.	2012	92.0
LSTM	Zaremba et al.	2014	78.4
RHN	Zilly et al.	2016	65.4
AWD-LSTM	Merity et al.	2018	58.8
GPT-2 (Transformer)	Radford et al.	2019	35.7

神经网络的基本单元 - 神经元

- 生物学上，神经元是神经系统的基本组成单元。很多人想象的神经网络应该是这样的

神经网络的基本单元 - 神经元

- 生物学上，神经元是神经系统的基本组成单元。很多人想象的神经网络应该是这样的

- 但我们这里说的是人工神经元，实际上是这样的 :)
  - 输入**x**经过**w**进行线性变化，之后加上偏移**b**，在经过激活函数f，最后得到**y** - 啥东东？？？

$$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$$

最简单的人工神经元模型 - 感知机（Perceptron）

- 感知机是人工神经元的一种实例。在上世纪50-60年代被提出后，对神经网络研究产生了深远影响。

最简单的人工神经元模型 - 感知机（Perceptron）

- 感知机是人工神经元的一种实例。在上世纪50-60年代被提出后，对神经网络研究产生了深远影响。
  - **输入**是若干个二值变量，$x_i = 0$ or 1

最简单的人工神经元模型 - 感知机（Perceptron）

- 感知机是人工神经元的一种实例。在上世纪50-60年代被提出后，对神经网络研究产生了深远影响。
  - **输入**是若干个二值变量，$x_i = 0$ or 1
  - 每一个输入变量对应一个**权重**w_i（实数）

最简单的人工神经元模型 - 感知机（Perceptron）

- 感知机是人工神经元的一种实例。在上世纪50-60年代被提出后，对神经网络研究产生了深远影响。
  - **输入**是若干个二值变量，$x_i = 0$ or 1
  - 每一个输入变量对应一个**权重**w_i（实数）
  - **输出**也是一个二值结果，$y = 0$ or 1。 判断的依据是，输入和加权和是否大于（或者小于）一个阈值σ：

$$y = \begin{cases} 0 & \sum_i w_i \cdot x_i < \sigma \\ 1 & \sum_i w_i \cdot x_i \geq \sigma \end{cases}$$

最简单的人工神经元模型 - 感知机（Perceptron）

- 感知机是人工神经元的一种实例。在上世纪50-60年代被提出后，对神经网络研究产生了深远影响。
  - **输入**是若干个二值变量，$x_i = 0$ or 1
  - 每一个输入变量对应一个**权重**w_i（实数）
  - **输出**也是一个二值结果，$y = 0$ or 1。 判断的依据是，输入和加权和是否大于（或者小于）一个阈值σ：

$$y = \begin{cases} 0 & \sum_i w_i \cdot x_i < \sigma \\ 1 & \sum_i w_i \cdot x_i \geq \sigma \end{cases}$$

一个例子

- 一个非常简单的例子。比如，有一个音乐会，你正在纠结是否去参加，有三个因素会影响你的决定
  - x_0：剧场是否离你足够近？
  - x_1：票价是否低于300元？
  - x_2：女朋友是否喜欢音乐会？

一个例子

- 一个非常简单的例子。比如，有一个音乐会，你正在纠结是否去参加，有三个因素会影响你的决定
  - x_0：剧场是否离你足够近？
  - x_1：票价是否低于300元？
  - x_2：女朋友是否喜欢音乐会？
- 如何决定？比如，女朋友很希望和你一起，但是剧场很远而且票价500元。如果这些因素对你的决策都是同等重要的，那么会有一个综合得分：

$$x_0 \cdot w_0 + x_1 \cdot w_1 + x_2 \cdot w_2 = 0 \cdot 1 + 0 \cdot 1 + 1 \cdot 1 = 1$$

一个例子

- 一个非常简单的例子。比如，有一个音乐会，你正在纠结是否去参加，有三个因素会影响你的决定
  - x_0：剧场是否离你足够近？
  - x_1：票价是否低于300元？
  - x_2：女朋友是否喜欢音乐会？
- 如何决定？比如，女朋友很希望和你一起，但是剧场很远而且票价500元。如果这些因素对你的决策都是同等重要的，那么会有一个综合得分：

$$x_0 \cdot w_0 + x_1 \cdot w_1 + x_2 \cdot w_2 = 0 \cdot 1 + 0 \cdot 1 + 1 \cdot 1 = 1$$

- 如果你不是十分纠结，能够接受不完美的事情，你可能会有$\sigma = 1$，于是

$$\sum_i x_i \cdot w_i \geq \sigma$$

那么，你会去参加音乐会

一个例子 - 权重

- 可以看出，实际上这个决策过程本质上就是一个感知机

一个例子 - 权重

- 可以看出，实际上这个决策过程本质上就是一个感知机
- 但是，人并不是完美的，往往对有些事情会更在意一些。如果你是**守财奴**，因此会对票价看的更重一些，这时你会用不均匀的权重计算每个因素的影响，比如：$w_0 = 0.5$，$w_1 = 2$，$w_2 = 0.5$

一个例子 - 权重

- 可以看出，实际上这个决策过程本质上就是一个感知机
- 但是，人并不是完美的，往往对有些事情会更在意一些。如果你是**守财奴**，因此会对票价看的更重一些，这时你会用不均匀的权重计算每个因素的影响，比如：$w_0 = 0.5$，$w_1 = 2$，$w_2 = 0.5$
- 女友很希望和你一起，但是剧场很远而且票价500元，会导致你选择不去看音乐会（女朋友都不要了，咋整）

$$\sum_i x_i \cdot w_i = 0 \cdot 0.5 + 0 \cdot 2 + 1 \cdot 0.5 = 0.5 < \sigma = 1$$

一个例子 - 输入形式

- 在遭受了女友一万点伤害之后，你意识到决策不应该只考虑非0即1的因素，应该把“程度”考虑进来：
  - x_0：10/距离
  - x_1：150/票价
  - x_2：女朋友是否喜欢？(这条不敢改)

一个例子 - 输入形式

- 在遭受了女友一万点伤害之后，你意识到决策不应该只考虑非0即1的因素，应该把“程度”考虑进来：
  - x_0：10/距离
  - x_1：150/票价
  - x_2：女朋友是否喜欢？(这条不敢改)
- 新模型中，x_0和x_1是连续变量，x_2是一个离散变量

一个例子 - 输入形式

- 在遭受了女友一万点伤害之后，你意识到决策不应该只考虑非0即1的因素，应该把“程度”考虑进来：
  - x_0：10/距离
  - x_1：150/票价
  - x_2：女朋友是否喜欢？(这条不敢改)
- 新模型中，x_0和x_1是连续变量，x_2是一个离散变量

- 女朋友很希望和你一起，但是剧场有20km远而且票价500元。于是有 $x_0 = 10/20 = 0.5$，$x_1 = 150/500 = 0.3$，$x_2 = 1$。综合来看$\sum_i x_i \cdot w_i \geq \sigma$，还是去听音乐会 :)

$$\sum_i x_i \cdot w_i = 0.5 \cdot 0.5 + 0.3 \cdot 2 + 1 \cdot 0.5 = 1.35 \geq \sigma = 1$$

一个例子 - 学习

- 一次成功的音乐会之后，你似乎掌握了真理：只要女朋友开心就好，为何不把这个因素的权重调大。最简单的方式是把w_0，w_1的权重都置0，同时令$w_3 > 0$

一个例子 - 学习

- 一次成功的音乐会之后，你似乎掌握了真理：只要女朋友开心就好，为何不把这个因素的权重调大。最简单的方式是把w_0，w_1的权重都置0，同时令$w_3 > 0$

一个例子 - 学习

- 一次成功的音乐会之后，你似乎掌握了真理：只要女朋友开心就好，为何不把这个因素的权重调大。最简单的方式是把w_0，w_1的权重都置0，同时令$w_3 > 0$
- 很快又有一场音乐会，距你1000公里，票价（不含路费）3000元，当然你女友是一直是喜欢音乐会的。根据新的决策模型，你义无反顾地决定去听这场音乐会

一个例子 - 学习

- 一次成功的音乐会之后，你似乎掌握了真理：只要女朋友开心就好，为何不把这个因素的权重调大。最简单的方式是把w_0，w_1的权重都置0，同时令$w_3 > 0$
- 很快又有一场音乐会，距你1000公里，票价（不含路费）3000元，当然你女友是一直是喜欢音乐会的。根据新的决策模型，你义无反顾地决定去听这场音乐会
- 之后，你女朋友又给了你1万点伤害，痛啊！！！
  - 结果你发现：女友既要浪漫，同时也爱财

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重

实验	1
结果	失败

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重

实验	1	2
结果	失败	成功

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重

实验	1	2	3
结果	失败	成功	失败

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重

实验	1	2	3	4
结果	失败	成功	失败	失败

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重

实验	1	2	3	4	...
结果	失败	成功	失败	失败	...

一个例子 - 权重学习

- **痛定思痛**，你发现每个因素的权重需要准确地设置才能达到最好的决策效果
  - 如何确定最好的权重？
- **当然**，你是一个勇于实践的人
  - 方法很简单：不断地尝试，根据结果不断地调整权重
  - 在进行了很多次实验后，发现了相对好的一组权重

实验	1	2	3	4	...	10k
结果	失败	成功	失败	失败	...	成功

一个例子 - 总结

- 即便对于一个简单问题，如何设计一种合理方法的准确的进行决策并不简单。在上面这个模型中，还有一些问题需要回答

一个例子 - 总结

- 即便对于一个简单问题，如何设计一种合理方法的准确的进行决策并不简单。在上面这个模型中，还有一些问题需要回答
  - 对问题建模，即：定义输入$\{x_i\}$的形式

一个例子 - 总结

- 即便对于一个简单问题，如何设计一种合理方法的准确的进行决策并不简单。在上面这个模型中，还有一些问题需要回答
  - 对问题建模，即：定义输入$\{x_i\}$的形式
  - 设计有效的决策模型，即：定义y

一个例子 - 总结

- 即便对于一个简单问题，如何设计一种合理方法的准确的进行决策并不简单。在上面这个模型中，还有一些问题需要回答
  - 对问题建模，即：定义输入$\{x_i\}$的形式
  - 设计有效的决策模型，即：定义y
  - 决定模型所涉及的参数（如权重$\{w_i\}$）的最优值

一个例子 - 总结

- 即便对于一个简单问题，如何设计一种合理方法的准确的进行决策并不简单。在上面这个模型中，还有一些问题需要回答
  - 对问题建模，即：定义输入$\{x_i\}$的形式
  - 设计有效的决策模型，即：定义y
  - 决定模型所涉及的参数（如权重$\{w_i\}$）的最优值

- **当然**，后面的内容会涉及上面的问题，而且不止这些 :)

入门人工神经网络(深度学习)的三个基本问题

1. 人工神经网络的基本单元是什么，如何组合出更强大的模型？

2. 人工神经网络的数学描述是什么，如何编程实现这种数学模型？

3. 如何对模型中的参数进行学习，之后使用学习到的模型进行推断？

首先

人工神经网络的基本单元是什么，
如何组合出更强大的模型？

$$\mathbf{y} = ?(\mathbf{x})$$

预热 - 线性代数知识

- **矩阵**：我们用a表示一个标量(一个数)，用粗体**a**表示一个矩阵(或向量)，其中a_{ij}表示**a**第i行、第j列的元素

$$a = 5 \qquad \mathbf{a} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- **向量**：一种特殊的矩阵，只有一行或者一列，这里默认使用行向量，比如$\mathbf{a} = (a_1, a_2, a_3) = (10, 20, 30)$，**a**对应的列向量记为$\mathbf{a}^T$

预热 - 线性代数知识

- **矩阵**：我们用a表示一个标量(一个数)，用粗体**a**表示一个矩阵(或向量)，其中a_{ij}表示**a**第i行、第j列的元素

$$a = 5 \qquad \mathbf{a} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- **向量**：一种特殊的矩阵，只有一行或者一列，这里默认使用行向量，比如$\mathbf{a} = (a_1, a_2, a_3) = (10, 20, 30)$，**a**对应的列向量记为$\mathbf{a}^T$
- **代数运算**：矩阵可以按位进行+、-等代数运算，对于$\mathbf{a} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$，$\mathbf{b} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$，有$\mathbf{a} + \mathbf{b} = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$

预热 - 线性代数知识

- **矩阵**：我们用a表示一个标量(一个数)，用粗体**a**表示一个矩阵(或向量)，其中a_{ij}表示**a**第i行、第j列的元素

$$a = 5 \qquad \mathbf{a} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- **向量**：一种特殊的矩阵，只有一行或者一列，这里默认使用行向量，比如$\mathbf{a} = (a_1, a_2, a_3) = (10, 20, 30)$，**a**对应的列向量记为$\mathbf{a}^T$
- **代数运算**：矩阵可以按位进行+、-等代数运算，对于$\mathbf{a} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$，$\mathbf{b} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$，有$\mathbf{a} + \mathbf{b} = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$
- **矩阵的微分**：按位进行，对于矩阵**c**和标量x有

$$\frac{\partial \mathbf{c}}{\partial x} = \begin{pmatrix} \frac{\partial c_{11}}{\partial x} & \frac{\partial c_{12}}{\partial x} \\ \frac{\partial c_{21}}{\partial x} & \frac{\partial c_{22}}{\partial x} \end{pmatrix} \qquad \frac{\partial x}{\partial \mathbf{c}} = \begin{pmatrix} \frac{\partial x}{\partial c_{11}} & \frac{\partial x}{\partial c_{12}} \\ \frac{\partial x}{\partial c_{21}} & \frac{\partial x}{\partial c_{22}} \end{pmatrix}$$

预热 - 线性代数知识(续)

- 矩阵的乘法：对于$\mathbf{a} \in \mathbb{R}^{n\times k}$和$\mathbf{b} \in \mathbb{R}^{k\times m}$，用$\mathbf{c} = \mathbf{ab} \in \mathbb{R}^{n\times m}$表示$\mathbf{a}$和$\mathbf{b}$的矩阵乘法，其中

$$c_{pq} = \sum_{i=1}^{k} a_{pi} b_{iq}$$

对于方程$\begin{cases} 5x_1 + 2x_2 = y_1 \\ 3x_1 + x_2 = y_2 \end{cases}$，可以表示为$\mathbf{a}\mathbf{x}^T = \mathbf{y}^T$ 其中$\mathbf{a} = \begin{pmatrix} 5 & 2 \\ 3 & 1 \end{pmatrix}$，$\mathbf{x}^T = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$，$\mathbf{y}^T = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$

预热 - 线性代数知识(续)

- 矩阵的乘法：对于$\mathbf{a} \in \mathbb{R}^{n\times k}$和$\mathbf{b} \in \mathbb{R}^{k\times m}$，用$\mathbf{c} = \mathbf{ab} \in \mathbb{R}^{n\times m}$表示$\mathbf{a}$和$\mathbf{b}$的矩阵乘法，其中

$$c_{pq} = \sum_{i=1}^{k} a_{pi} b_{iq}$$

对于方程$\begin{cases} 5x_1 + 2x_2 = y_1 \\ 3x_1 + x_2 = y_2 \end{cases}$，可以表示为$\mathbf{a}\mathbf{x}^T = \mathbf{y}^T$ 其中$\mathbf{a} = \begin{pmatrix} 5 & 2 \\ 3 & 1 \end{pmatrix}$，$\mathbf{x}^T = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$，$\mathbf{y}^T = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$

- 其它
  - 单位矩阵：方阵$\mathbf{I}$，$I_{ij} = 1$当且仅当$i = j$，否则$I_{ij} = 0$
  - 转置：$\mathbf{a}$的转置记为$\mathbf{a}^T$，有$a^T_{ji} = a_{ij}$
  - 逆矩阵：方阵$\mathbf{a}$的逆矩阵记为$\mathbf{a}^{-1}$，有$\mathbf{a}\mathbf{a}^{-1} = \mathbf{a}^{-1}\mathbf{a} = \mathbf{I}$
  - 向量(矩阵)的范数：$||\mathbf{a}||_p = \left(\sum_i |a_i|^p\right)^{\frac{1}{p}}$

人工神经元即一个函数

- 神经元：

$$\mathbf{y} = f\left(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}\right)$$

人工神经元即一个函数

- 神经元：

$$\mathbf{y} = f\left(\overset{\text{输入}}{\mathbf{x}} \cdot \mathbf{w} + \mathbf{b}\right)$$

人工神经元即一个函数

- 神经元：

$$\mathbf{y} = f\left(\underbrace{\mathbf{x}}_{\text{输入}} \cdot \underbrace{\mathbf{w}}_{\text{参数(权重)}} + \mathbf{b}\right)$$

人工神经元即一个函数

- 神经元：

$$\mathbf{y} = f\left(\underbrace{\mathbf{x}}_{\text{输入}} \cdot \underbrace{\mathbf{w}}_{\text{参数(权重)}} + \underbrace{\mathbf{b}}_{\text{偏移}}\right)$$

人工神经元即一个函数

- 神经元：

$$\mathbf{y} = \underbrace{f}_{\text{激活函数}}\left(\overbrace{\mathbf{x}}^{\text{输入}} \cdot \underbrace{\mathbf{w}}_{\text{参数(权重)}} + \overbrace{\mathbf{b}}^{\text{偏移}}\right)$$

人工神经元即一个函数

- 神经元：

$$\underbrace{\mathbf{y}}_{\text{输出}} = \underbrace{f}_{\text{激活函数}} \left(\underbrace{\mathbf{x}}_{\text{输入}} \cdot \underbrace{\mathbf{w}}_{\text{参数(权重)}} + \underbrace{\mathbf{b}}_{\text{偏移}} \right)$$

人工神经元即一个函数

- 神经元：

$$\underbrace{\mathbf{y}}_{\text{输出}} = \underbrace{f}_{\text{激活函数}} \left(\underbrace{\mathbf{x}}_{\text{输入}} \cdot \underbrace{\mathbf{w}}_{\text{参数(权重)}} + \underbrace{\mathbf{b}}_{\text{偏移}} \right)$$

- 以感知机为例
  - 输入：$\mathbf{x} = (x_0, ..., x_n)$
  - 权重：$\mathbf{w} = (w_0, ..., w_n)$
  - 偏移：$\mathbf{b} = (-\sigma)$
  - 激活函数：$f(z) = 1$ 当$z \geq 0$, 其它情况$f(z) = 0$
  - 输出：$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} - \sigma)$

“层”的概念

- 对于一个问题（相同输入），可能会有多个输出，这时可以把多个相同的神经元并列起来，构成一“层”
  - 比如，天气预报需要同时预测湿度和温度

“层”的概念

- 对于一个问题（相同输入），可能会有多个输出，这时可以把多个相同的神经元并列起来，构成一“层”
  - 比如，天气预报需要同时预测湿度和温度

“层”的概念

- 对于一个问题（相同输入），可能会有多个输出，这时可以把多个相同的神经元并列起来，构成一“层”
  - 比如，天气预报需要同时预测湿度和温度

“层”的概念

- 对于一个问题（相同输入），可能会有多个输出，这时可以把多个相同的神经元并列起来，构成一“层”
  - 比如，天气预报需要同时预测湿度和温度

输入向量:
$\mathbf{x} = (x_0, x_1)$

“层”的概念

- 对于一个问题（相同输入），可能会有多个输出，这时可以把多个相同的神经元并列起来，构成一“层”
  - 比如，天气预报需要同时预测湿度和温度

输入向量:
$\mathbf{x} = (x_0, x_1)$

输出向量:
$\mathbf{y} = (y_0, y_1, y_2)$

“层”的概念

- 对于一个问题（相同输入），可能会有多个输出，这时可以把多个相同的神经元并列起来，构成一“层”
  - 比如，天气预报需要同时预测湿度和温度

一层神经元

输入向量:
$\mathbf{x} = (x_0, x_1)$

输出向量:
$\mathbf{y} = (y_0, y_1, y_2)$

“层”的概念

- 对于一个问题（相同输入），可能会有多个输出，这时可以把多个相同的神经元并列起来，构成一“层”
  - 比如，天气预报需要同时预测湿度和温度

一层神经元

输入向量:
$\mathbf{x} = (x_0, x_1)$

输出向量:
$\mathbf{y} = (y_0, y_1, y_2)$

参数(矩阵):$\mathbf{w} = \begin{pmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \end{pmatrix}$

“层”的概念

- 对于一个问题（相同输入），可能会有多个输出，这时可以把多个相同的神经元并列起来，构成一“层”
  - 比如，天气预报需要同时预测湿度和温度

一层神经元

输入向量:
$\mathbf{x} = (x_0, x_1)$

输出向量:
$\mathbf{y} = (y_0, y_1, y_2)$

参数(矩阵):$\mathbf{w} = \begin{pmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \end{pmatrix}$ 参数(向量):$\mathbf{b} = (b_0, b_1, b_2)$

神经网络：线性变换 + 激活函数

- 对于向量$\mathbf{x} \in \mathbb{R}^m$，一层神经网络首先把他经过线性变换映射到$\mathbb{R}^m$，之后经过激活函数变换成$\mathbf{y} \in \mathbb{R}^n$

$$\mathbf{y} = \underbrace{f}_{\text{激活函数}}\left(\underbrace{\mathbf{x} \cdot \mathbf{w} + \mathbf{b}}_{\text{线性变换}}\right)$$

线性变换

- 对于线性空间V，任意$\mathbf{a}$，$\mathbf{b} \in V$和数域中的任意α，线性变换$T(\cdot)$需满足

$$
\begin{aligned}
T(\mathbf{a}+\mathbf{b}) &= T(\mathbf{a})+T(\mathbf{b}) \\
T(\alpha\mathbf{a}) &= \alpha T(\mathbf{a})
\end{aligned}
$$

线性变换

- 对于线性空间V，任意$\mathbf{a}$，$\mathbf{b} \in V$和数域中的任意α，线性变换$T(\cdot)$需满足

$$
\begin{aligned}
T(\mathbf{a}+\mathbf{b}) &= T(\mathbf{a})+T(\mathbf{b}) \\
T(\alpha\mathbf{a}) &= \alpha T(\mathbf{a})
\end{aligned}
$$

- 线性变换的一种几何解释：

$$\mathbf{x} \cdot \mathbf{w} + \mathbf{b}$$

线性变换

- 对于线性空间V，任意$\mathbf{a}$，$\mathbf{b} \in V$和数域中的任意α，线性变换$T(\cdot)$需满足

$$
\begin{aligned}
T(\mathbf{a}+\mathbf{b}) &= T(\mathbf{a})+T(\mathbf{b}) \\
T(\alpha\mathbf{a}) &= \alpha T(\mathbf{a})
\end{aligned}
$$

- 线性变换的一种几何解释：

$$\mathbf{x} \cdot \mathbf{w} + \mathbf{b}$$

↑

旋转(rotation)、扩张(dilation)、挤压(squeeze)等

线性变换

- 对于线性空间V，任意$\mathbf{a}$，$\mathbf{b} \in V$和数域中的任意α，线性变换$T(\cdot)$需满足

$$
\begin{aligned}
T(\mathbf{a}+\mathbf{b}) &= T(\mathbf{a})+T(\mathbf{b}) \\
T(\alpha\mathbf{a}) &= \alpha T(\mathbf{a})
\end{aligned}
$$

- 线性变换的一种几何解释：

$$\mathbf{x} \cdot \mathbf{w} + \mathbf{b} \longleftarrow \text{平移(shift)}$$

$\uparrow$

旋转(rotation)、扩张(dilation)、挤压(squeeze)等

线性变换（续）

- 线性变换也适用于更加复杂的情况，这也给神经网络提供了拟合不同数据分布的能力
  - 比如，我们可以把三维图形投影到二维平面上
  - 再比如，我们也可以把二维平面上的图形映射到三维平面

$$\underbrace{\left\{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\right\}}_{5} \times \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \underbrace{\left\{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \cdots \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\right\}}_{5}$$

激活函数

- 激活函数更多地是为了解决实际问题中的非线性变换
  - 非线性部分提供了拟合任意函数的能力（稍后介绍）

我是一根筷子

我是一只蚯蚓

激活函数

- 激活函数更多地是为了解决实际问题中的非线性变换
  - 非线性部分提供了拟合任意函数的能力（稍后介绍）

我是一根筷子

我是一只蚯蚓

- 简单的非线性函数

Quadratic:

$y = \frac{1}{2}(x - 0.3)^2 + 0.2$

Exponential:

$y = 0.5 \cdot \exp(x)$

Absolute:

$y = |x - 0.3| + 0.1$

常用的激活函数

- 好多好多，列举不全 ...

$y = ln(1 + e^x)$

(a) softplus

$y = \frac{1}{1+e^{-x}}$

(b) sigmoid

$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

(c) tanh

$y = \max(0, x)$

(d) relu

$y = e^{-x^2}$

(e) gaussian

$y = x$

(f) identity

更多的层

- **单层神经网络**：线性变换 + 激活函数（非线性）
- 我们可以重复上面的过程，构建**多层神经网络**

更多的层

- **单层神经网络**：线性变换 + 激活函数（非线性）
- 我们可以重复上面的过程，构建**多层神经网络**

更多的层

- **单层神经网络**：线性变换 + 激活函数（非线性）
- 我们可以重复上面的过程，构建**多层神经网络**

更多的层

- **单层神经网络**：线性变换 + 激活函数（非线性）
- 我们可以重复上面的过程，构建**多层神经网络**

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

$w_2 = 100$

$b_2 = -6$

$w_2' = 0.7$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

$w_2 = 100$

$b_2 = -6$

$w_2' = -0.7$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

这是一个
step function

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1 + e^{-s})$

sum: $s = x_1 \cdot w + b$

这是一个
step function

函数的每一段都可
由step function
近似

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1+e^{-s})$

sum: $s = x_1 \cdot w + b$

多层神经网络可以逼近任意函数

- 以一个简单的三层网络为例（隐层激活函数：sigmoid）

sigmoid: $f(s) = 1/(1+e^{-s})$

sum: $s = x_1 \cdot w + b$

然后

人工神经网络的数学描述是什么，
如何编程实现这种数学模型？

如何描述神经网络 - 张量计算

- 对于神经网络，输入**x**和输出**y**的形式并不仅仅是向量

$$\mathbf{y} = f\left(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}\right)$$

如何描述神经网络 - 张量计算

- 对于神经网络，输入**x**和输出**y**的形式并不仅仅是向量

向量？矩阵？...

向量 e.g., $(1, 3)$

$$\mathbf{y} = f\left(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}\right)$$

矩阵 e.g., $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

如何描述神经网络 - 张量计算

- 对于神经网络，输入**x**和输出**y**的形式并不仅仅是向量

向量？矩阵？...　　向量 e.g., $(1, 3)$

$$\mathbf{y} = f\left(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}\right)$$

矩阵 e.g., $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

- **x**和**y**实际上是一个叫tensor的东西，即**张量**。比如，

如何描述神经网络 - 张量计算

- 对于神经网络，输入**x**和输出**y**的形式并不仅仅是向量

向量？矩阵？...　　向量 e.g., $(1,3)$

$$\mathbf{y} = f\left(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}\right)$$

矩阵 e.g., $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

- **x**和**y**实际上是一个叫tensor的东西，即张量。比如，

$$\mathbf{x} = (1,3)$$

如何描述神经网络 - 张量计算

- 对于神经网络，输入**x**和输出**y**的形式并不仅仅是向量

向量？矩阵？...　　向量 e.g., $(1,3)$

$$\mathbf{y} = f\left(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}\right)$$

矩阵 e.g., $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

- **x**和**y**实际上是一个叫tensor的东西，即张量。比如，

$$\mathbf{x} = (1,3) \qquad \mathbf{x} = \begin{pmatrix} -1 & 3 \\ 0.2 & 2 \end{pmatrix}$$

如何描述神经网络 - 张量计算

- 对于神经网络，输入**x**和输出**y**的形式并不仅仅是向量

向量？矩阵？...　　向量 e.g., $(1,3)$

$$\mathbf{y} = f\left(\mathbf{x} \cdot \mathbf{w} + \mathbf{b}\right)$$

矩阵 e.g., $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

- **x**和**y**实际上是一个叫tensor的东西，即张量。比如，

$$\mathbf{x} = (1,3) \quad \mathbf{x} = \begin{pmatrix} -1 & 3 \\ 0.2 & 2 \end{pmatrix} \quad \text{啥？}\mathbf{x} = \begin{pmatrix} \begin{pmatrix} -1 & 3 \\ 0.2 & 2 \end{pmatrix} \\ \begin{pmatrix} -1 & 3 \\ 0.2 & 2 \end{pmatrix} \end{pmatrix}$$

张量是什么

- 深度学习中，张量被“简单”地定义为多维数组
  - 张量的阶（rank）表示有多少个独立的方向，每个方向可以由多个维度表示

张量是什么

- 深度学习中，张量被“简单”地定义为多维数组
  - 张量的阶（rank）表示有多少个独立的方向，每个方向可以由多个维度表示

3

标量

scalar

(rank=0)

张量是什么

- 深度学习中，张量被“简单”地定义为多维数组
  - 张量的阶（rank）表示有多少个独立的方向，每个方向可以由多个维度表示

$$3 \qquad \begin{pmatrix} 2 \\ .3 \\ -8 \\ .2 \end{pmatrix}$$

标量	向量
scalar	vector
(rank=0)	(rank=1)

张量是什么

- 深度学习中，张量被“简单”地定义为多维数组
  - 张量的阶（rank）表示有多少个独立的方向，每个方向可以由多个维度表示

$$3 \qquad \begin{pmatrix} 2 \\ .3 \\ -8 \\ .2 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 9 \\ 1 & 0 & 0 \\ 1 & -4 & 7 \end{pmatrix}$$

标量	向量	矩阵
scalar	vector	matrix
(rank=0)	(rank=1)	(rank=2)

张量是什么

- 深度学习中，张量被“简单”地定义为多维数组
  - 张量的阶（rank）表示有多少个独立的方向，每个方向可以由多个维度表示

$$3 \qquad \begin{pmatrix} 2 \\ .3 \\ -8 \\ .2 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 9 \\ 1 & 0 & 0 \\ 1 & -4 & 7 \end{pmatrix}$$

标量	向量	矩阵	3阶张量
scalar	vector	matrix	tensor
(rank=0)	(rank=1)	(rank=2)	(rank=3)

事实上，张量不是简单的多维数组 - 别慌，了解下 :)

- 非常负责任的说，张量不是向量和矩阵的简单扩展，甚至说，多维数组也不是张量所必须的表达形式

事实上，张量不是简单的多维数组 - 别慌，了解下 :)

- **非常负责任的说**，张量不是向量和矩阵的简单扩展，甚至说，多维数组也不是张量所必须的表达形式
- 严格意义上，张量是：

1. **看不懂的定义**：由若干坐标系改变时满足一定坐标转化关系的抽象对象，它是一个不随参照系的坐标变换而变化的几何量（几何定义）

事实上，张量不是简单的多维数组 - 别慌，了解下 :)

- **非常负责任的说**，张量不是向量和矩阵的简单扩展，甚至说，多维数组也不是张量所必须的表达形式
- 严格意义上，张量是：
  1. **看不懂的定义**：由若干坐标系改变时满足一定坐标转化关系的抽象对象，它是一个不随参照系的坐标变换而变化的几何量（几何定义）
  2. **还是看不懂的定义**：若干向量和协向量通过张量乘法定义的量（代数定义）

事实上，张量不是简单的多维数组 - 别慌，了解下 :)

- **非常负责任的说**，张量不是向量和矩阵的简单扩展，甚至说，多维数组也不是张量所必须的表达形式
- 严格意义上，张量是：

1. **看不懂的定义**：由若干坐标系改变时满足一定坐标转化关系的抽象对象，它是一个不随参照系的坐标变换而变化的几何量（几何定义）
2. **还是看不懂的定义**：若干向量和协向量通过张量乘法定义的量（代数定义）
3. **还可以解释的定义**：张量是多重线性函数，是定义在一些向量空间和笛卡儿积上的多重线性映射
   - 张量记为$T(v_0, ..., v_r)$，其中输入是r个向量$\{v_0, ..., v_r\}$
   - 多重线性是指，对于每个输入，函数都是线性的，比如，对于一个v_i，我们有

$$T(v_0, ..., v_i + c \cdot u, ..., v_r) = T(v_0, ..., v_i, ..., v_r) + c \cdot T(v_0, ..., u, ..., v_r)$$

其中，c为任意数。这个性质非常重要，它可以推导出前面的其它定义。

张量是一个“量”，不是“矩阵”

- 再理解一下，
  - 如果一个物理量，在物体的某个位置上只是一个单值，它就是标量，比如密度
  - 如果它在同一个位置、从多个的方向上看，有不同的值，而且这个数恰好用矩阵乘观察方向来算出来，就是张量(rank>1)，比如应力张量

张量是一个“量”，不是“矩阵”

- 再理解一下，
  - 如果一个物理量，在物体的某个位置上只是一个单值，它就是标量，比如密度
  - 如果它在同一个位置、从多个的方向上看，有不同的值，而且这个数恰好用矩阵乘观察方向来算出来，就是张量(rank>1)，比如应力张量

$T(v,u)$是一个三维空间(x,y,z)上的2阶张量，其中v和u是两个向量

张量是一个“量”，不是“矩阵”

- 再理解一下，
  - 如果一个物理量，在物体的某个位置上只是一个单值，它就是标量，比如密度
  - 如果它在同一个位置、从多个的方向上看，有不同的值，而且这个数恰好用矩阵乘观察方向来算出来，就是张量(rank>1)，比如应力张量

$T(v,u)$是一个三维空间(x,y,z)上的2阶张量，其中v和u是两个向量

方向$v=(a,b,c)$

张量是一个“量”，不是“矩阵”

- 再理解一下，
  - 如果一个物理量，在物体的某个位置上只是一个单值，它就是标量，比如密度
  - 如果它在同一个位置、从多个的方向上看，有不同的值，而且这个数恰好用矩阵乘观察方向来算出来，就是张量(rank>1)，比如应力张量

$T(v,u)$是一个三维空间(x,y,z)上的2阶张量，其中v和u是两个向量

方向$v=(a,b,c)$

方向$u=(a',b',c')$

张量是一个“量”，不是“矩阵”

- 再理解一下，
  - 如果一个物理量，在物体的某个位置上只是一个单值，它就是标量，比如密度
  - 如果它在同一个位置、从多个的方向上看，有不同的值，而且这个数恰好用矩阵乘观察方向来算出来，就是张量(rank>1)，比如应力张量

$T(v,u)$是一个三维空间(x,y,z)上的2阶张量，其中v和u是两个向量

$$T(v,u) = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}^T \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{pmatrix} \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix}$$

方向$v = (a,b,c)$

方向$u = (a',b',c')$

张量是一个“量”，不是“矩阵”

- 再理解一下，
  - 如果一个物理量，在物体的某个位置上只是一个单值，它就是标量，比如密度
  - 如果它在同一个位置、从多个的方向上看，有不同的值，而且这个数恰好用矩阵乘观察方向来算出来，就是张量(rank>1)，比如应力张量

$T(v,u)$是一个三维空间(x,y,z)上的2阶张量，其中v和u是两个向量

$$T(v,u) = \underbrace{\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}^T}_{v\text{在基向量上的投影}} \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{pmatrix} \underbrace{\begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix}}_{u\text{在基向量上的投影}}$$

方向$v = (a,b,c)$

方向$u = (a',b',c')$

张量是一个“量”，不是“矩阵”

- 再理解一下，
  - 如果一个物理量，在物体的某个位置上只是一个单值，它就是标量，比如密度
  - 如果它在同一个位置、从多个的方向上看，有不同的值，而且这个数恰好用矩阵乘观察方向来算出来，就是张量(rank>1)，比如应力张量

方向$v = (a, b, c)$

方向$u = (a', b', c')$

$T(v, u)$是一个三维空间(x, y, z)上的2阶张量，其中v和u是两个向量

v在基向量上的投影 ↓ ↓ u在基向量上的投影

$$T(v,u) = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}^T \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{pmatrix} \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix}$$

↑

张量在3×3个方向上的分量，恰巧用“矩阵”表示，记为$[T]$，想象一下坐标系的旋转

“矩阵”是“张量”的扩展：在神经网络中使用张量

- 但是前面的可以忽略 - 在这里，“张量就是多维数组”
  - 向量、矩阵都可以看做数学上的“张量”的扩展

“矩阵”是“张量”的扩展：在神经网络中使用张量

- 但是前面的可以忽略 - 在这里，“张量就是多维数组”
  - 向量、矩阵都可以看做数学上的“张量”的扩展
- 张量$T(1:3)$表示一个向量，有三个元素

物理存储：

1	2	3

“矩阵”是“张量”的扩展：在神经网络中使用张量

- 但是前面的可以忽略 - 在这里，“张量就是多维数组”
  - 向量、矩阵都可以看做数学上的“张量”的扩展
- 张量$T(1:3)$表示一个向量，有三个元素

  物理存储：

1	2	3

- 张量$T(1:2,1:3)$表示一个3×2的矩阵

  物理存储：

1	2	3	4	5	6

“矩阵”是“张量”的扩展：在神经网络中使用张量

- 但是前面的可以忽略 - 在这里，“张量就是多维数组”
  - 向量、矩阵都可以看做数学上的“张量”的扩展
- 张量$T(1:3)$表示一个向量，有三个元素

  物理存储：

1	2	3

- 张量$T(1:2,1:3)$表示一个3×2的矩阵

  物理存储：

1	2	3	4	5	6

- 张量$T(1:2,1:2,1:3)$表示一个三阶张量，大小是$3\times 2\times 2$

  物理存储：

1	2	3	4	5	6	7	8	9	10	11	12

$\underbrace{\qquad\qquad}_{3\times 2 \text{ sub-tensor}}$ $\underbrace{\qquad\qquad}_{3\times 2 \text{ sub-tensor}}$

“矩阵”是“张量”的扩展：在神经网络中使用张量

- 但是前面的可以忽略 - 在这里，“张量就是多维数组”
  - 向量、矩阵都可以看做数学上的“张量”的扩展
- 张量$T(1:3)$表示一个向量，有三个元素

  物理存储：

1	2	3

- 张量$T(1:2,1:3)$表示一个3×2的矩阵

  物理存储：

1	2	3	4	5	6

- 张量$T(1:2,1:2,1:3)$表示一个三阶张量，大小是$3\times 2\times 2$

  物理存储：

1	2	3	4	5	6	7	8	9	10	11	12

$\underbrace{\qquad\qquad\qquad}_{3\times 2 \text{ sub-tensor}}$ $\underbrace{\qquad\qquad\qquad}_{3\times 2 \text{ sub-tensor}}$

- 高阶张量：数组！数组！数组！
  - 和C++、Python中的多维数组一模一样

张量的矩阵乘法

- 对于神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$，$\mathbf{x} \cdot \mathbf{w}$或$\mathbf{x} \times \mathbf{w}$是线性变换，其中$\mathbf{x}$是输入张量，$\mathbf{w}$是一个矩阵
  - $\mathbf{x} \cdot \mathbf{w}$表示的是矩阵乘法（或记为$\times$）
  - 注意，这里不是张量乘法，因为张量乘法还有其它定义
  - $\mathbf{w}$是$n \times m$的矩阵，$\mathbf{x}$的形状是$\ldots \times n$，即$\mathbf{x}$的第一维度需要和$\mathbf{w}$的行数大小相等

$$\mathbf{x}(1:4, 1:4, 1:4) \times \mathbf{w}(1:4, 1:2) = \mathbf{s}(1:4, 1:4, 1:2)$$

张量的矩阵乘法

- 对于神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$，$\mathbf{x} \cdot \mathbf{w}$或$\mathbf{x} \times \mathbf{w}$是线性变换，其中$\mathbf{x}$是输入张量，$\mathbf{w}$是一个矩阵
  - $\mathbf{x} \cdot \mathbf{w}$表示的是矩阵乘法（或记为$\times$）
  - 注意，这里不是张量乘法，因为张量乘法还有其它定义
  - $\mathbf{w}$是$n \times m$的矩阵，$\mathbf{x}$的形状是$\ldots \times n$，即$\mathbf{x}$的第一维度需要和$\mathbf{w}$的行数大小相等

$$\mathbf{x}(1:4, 1:4, 1:4) \times \mathbf{w}(1:4, 1:2) = \mathbf{s}(1:4, 1:4, 1:2)$$

$\mathbf{x}$

-1	0
0	-1
1	1
0	0

$\mathbf{w}$

张量的矩阵乘法

- 对于神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$，$\mathbf{x} \cdot \mathbf{w}$或$\mathbf{x} \times \mathbf{w}$是线性变换，其中$\mathbf{x}$是输入张量，$\mathbf{w}$是一个矩阵
  - $\mathbf{x} \cdot \mathbf{w}$表示的是矩阵乘法（或记为$\times$）
  - 注意，这里不是张量乘法，因为张量乘法还有其它定义
  - $\mathbf{w}$是$n \times m$的矩阵，$\mathbf{x}$的形状是$\ldots \times n$，即$\mathbf{x}$的第一维度需要和$\mathbf{w}$的行数大小相等

$$\mathbf{x}(1:4, 1:4, 1:4) \times \mathbf{w}(1:4, 1:2) = \mathbf{s}(1:4, 1:4, 1:2)$$

张量的矩阵乘法

- 对于神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$，$\mathbf{x} \cdot \mathbf{w}$或$\mathbf{x} \times \mathbf{w}$是线性变换，其中$\mathbf{x}$是输入张量，$\mathbf{w}$是一个矩阵
  - $\mathbf{x} \cdot \mathbf{w}$表示的是矩阵乘法（或记为$\times$）
  - 注意，这里不是张量乘法，因为张量乘法还有其它定义
  - $\mathbf{w}$是$n \times m$的矩阵，$\mathbf{x}$的形状是$\ldots \times n$，即$\mathbf{x}$的第一维度需要和$\mathbf{w}$的行数大小相等

$$\mathbf{x}(1:4, 1:4, 1:4) \times \mathbf{w}(1:4, 1:2) = \mathbf{s}(1:4, 1:4, 1:2)$$

张量的矩阵乘法

- 对于神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$，$\mathbf{x} \cdot \mathbf{w}$或$\mathbf{x} \times \mathbf{w}$是线性变换，其中$\mathbf{x}$是输入张量，$\mathbf{w}$是一个矩阵
  - $\mathbf{x} \cdot \mathbf{w}$表示的是矩阵乘法（或记为$\times$）
  - 注意，这里不是张量乘法，因为张量乘法还有其它定义
  - $\mathbf{w}$是$n \times m$的矩阵，$\mathbf{x}$的形状是$\ldots \times n$，即$\mathbf{x}$的第一维度需要和$\mathbf{w}$的行数大小相等

$$\mathbf{x}(1:4, 1:4, 1:4) \times \mathbf{w}(1:4, 1:2) = \mathbf{s}(1:4, 1:4, 1:2)$$

张量的矩阵乘法

- 对于神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$，$\mathbf{x} \cdot \mathbf{w}$或$\mathbf{x} \times \mathbf{w}$是线性变换，其中$\mathbf{x}$是输入张量，$\mathbf{w}$是一个矩阵
  - $\mathbf{x} \cdot \mathbf{w}$表示的是矩阵乘法（或记为$\times$）
  - 注意，这里不是张量乘法，因为张量乘法还有其它定义
  - $\mathbf{w}$是$n \times m$的矩阵，$\mathbf{x}$的形状是$\ldots \times n$，即$\mathbf{x}$的第一维度需要和$\mathbf{w}$的行数大小相等

$$\mathbf{x}(1:4, 1:4, 1:4) \times \mathbf{w}(1:4, 1:2) = \mathbf{s}(1:4, 1:4, 1:2)$$

张量的单元操作

- 神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$也包括一些张量的单元操作（element-wise operation）
  - 加法：$\mathbf{s} + \mathbf{b}$，其中$\mathbf{s} = \mathbf{x} \cdot \mathbf{w}$
  - 激活函数：$f(\cdot)$

张量的单元操作

- 神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$也包括一些张量的单元操作（element-wise operation）
  - 加法：$\mathbf{s} + \mathbf{b}$，其中$\mathbf{s} = \mathbf{x} \cdot \mathbf{w}$
  - 激活函数：$f(\cdot)$
- 单元加就是对张量中的每个位置都进行加法

张量的单元操作

- 神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$也包括一些张量的单元操作（element-wise operation）
  - 加法：$\mathbf{s} + \mathbf{b}$，其中$\mathbf{s} = \mathbf{x} \cdot \mathbf{w}$
  - 激活函数：$f(\cdot)$
- 单元加就是对张量中的每个位置都进行加法
  - 扩展：加法的**广播**，重复利用一个张量进行加法，并不要求两个张量形状相同

s

1	2	3	4
5	6	7	8

$+$

b

1	1	1	1

$=$

s+b

2	3	4	5
6	7	8	9

张量的单元操作

- 神经网络$\mathbf{y} = f(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$也包括一些张量的单元操作（element-wise operation）
  - 加法：$\mathbf{s} + \mathbf{b}$，其中$\mathbf{s} = \mathbf{x} \cdot \mathbf{w}$
  - 激活函数：$f(\cdot)$
- **单元加**就是对张量中的每个位置都进行加法
  - 扩展：加法的**广播**，重复利用一个张量进行加法，并不要求两个张量形状相同

$$\overset{\mathbf{s}}{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}} + \overset{\mathbf{b}}{\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}} = \overset{\mathbf{s+b}}{\begin{bmatrix} 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix}}$$

- 类似的，我们可以做减法、乘法，也包括激活函数。这也被称作函数的向量化（vectorization）

$$\mathrm{Relu}\left(\begin{pmatrix} 2 \\ -.3 \end{pmatrix}\right) = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

如何实现？- 开源张量计算框架

- 实现神经网络的开源系统很多，一个简单好用的工具包NumPy `https://numpy.org/`
  - Python接口，多维数组的定义使用方便
  - 提供了张量表示和使用的范式

如何实现？- 开源张量计算框架

- 实现神经网络的开源系统很多，一个简单好用的工具包NumPy `https://numpy.org/`
  - Python接口，多维数组的定义使用方便
  - 提供了张量表示和使用的范式
- 最近，很火的两个框架：TensorFlow和PyTorch
  - Google和Facebook出品，质量有保证
  - 功能强大，接口丰富
  - 可以进行大规模部署和应用
  - 大量可参考的实例

TensorFlow PyTorch

如何实现？- 开源张量计算框架

- 实现神经网络的开源系统很多，一个简单好用的工具包NumPy `https://numpy.org/`
  - Python接口，多维数组的定义使用方便
  - 提供了张量表示和使用的范式
- 最近，很火的两个框架：TensorFlow和PyTorch
  - Google和Facebook出品，质量有保证
  - 功能强大，接口丰富
  - 可以进行大规模部署和应用
  - 大量可参考的实例

TensorFlow PyTorch

- 还有其它还在更新的优秀框架：CNTK、MXNet、PaddlePaddle、Keras、Chainer、dl4j、NiuTensor等

NiuTensor

- 这里使用我们自研的NiuTensor工具包进行教学 http://www.niutrans.com/opensource/niutensor/index.html
  - 简单小巧，易于修改
  - C++语言编写，代码高度优化
  - 同时支持CPU和GPU设备
  - 丰富的张量计算接口

使用NiuTensor

- NiuTensor的使用很简单，下面是一个C++例子

```
#include "source/tensor/XTensor.h"          // 引用XTensor定义的头文件
using namespace nts;                        // 引用nts命名空间

int main(int argc, const char ** argv){
   XTensor tensor;                          // 声明张量tensor
   InitTensor2D(&tensor, 2, 2, X_FLOAT);    // 定义张量为2*2的矩阵
   tensor.SetDataRand();                    // [0,1]均匀分布初始化张量
   tensor.Dump(stdout);                     // 输出张量内容
   return 0;
}
```

使用NiuTensor

- NiuTensor的使用很简单，下面是一个C++例子

```
#include "source/tensor/XTensor.h"          // 引用XTensor定义的头文件
using namespace nts;                        // 引用nts命名空间

int main(int argc, const char ** argv){
   XTensor tensor;                          // 声明张量tensor
   InitTensor2D(&tensor, 2, 2, X_FLOAT);    // 定义张量为2*2的矩阵
   tensor.SetDataRand();                    // [0,1]均匀分布初始化张量
   tensor.Dump(stdout);                     // 输出张量内容
   return 0;
}
```

- 运行这个程序会显示张量每个元素的值
  - 二阶张量(order=2)，形状是2×2 (dimsize=2,2)，数据类型是单精度浮点(dtype=X_FLOAT)，非稀疏(dense=1.00)

```
order=2 dimsize=2,2 dtype=X_FLOAT dense=1.000000
3.605762e-001 2.992340e-001 1.393780e-001 7.301248e-001
```

定义XTensor

- 张量由类XTensor表示，利用InitTensor定义，参数：
  - 指向XTensor类型变量的指针
  - 张量的阶
  - 各个方向维度的大小（与传统多维数组约定一样）
  - 张量的数据类型等（有缺省值）

```
XTensor tensor;                          // 声明张量tensor
int sizes[6] = {2,3,4,2,3,4};            // 张量的形状为2*3*4*2*3*4
InitTensor(&tensor, 6, sizes, X_FLOAT);  // 定义形状为sizes的6阶张量
```

定义XTensor

- 张量由类XTensor表示，利用InitTensor定义，参数：
  - 指向XTensor类型变量的指针
  - 张量的阶
  - 各个方向维度的大小（与传统多维数组约定一样）
  - 张量的数据类型等（有缺省值）

```
XTensor tensor;                          // 声明张量tensor
int sizes[6] = {2,3,4,2,3,4};            // 张量的形状为2*3*4*2*3*4
InitTensor(&tensor, 6, sizes, X_FLOAT);  // 定义形状为sizes的6阶张量
```

- 更简便的定义方式

```
XTensor a, b, c;                         // 声明张量tensor
InitTensor1D(&a, 10, X_INT);             // 10维的整数型向量
InitTensor1D(&b, 10);                    // 10维的向量，缺省类型(浮点)
InitTensor4D(&c, 10, 20, 30, 40);        // 10*20*30*40的4阶张量(浮点)
```

定义XTensor

- 张量由类XTensor表示，利用InitTensor定义，参数：
  - 指向XTensor类型变量的指针
  - 张量的阶
  - 各个方向维度的大小（与传统多维数组约定一样）
  - 张量的数据类型等（有缺省值）

```
XTensor tensor;                          // 声明张量tensor
int sizes[6] = {2,3,4,2,3,4};            // 张量的形状为2*3*4*2*3*4
InitTensor(&tensor, 6, sizes, X_FLOAT);  // 定义形状为sizes的6阶张量
```

- 更简便的定义方式

```
XTensor a, b, c;                         // 声明张量tensor
InitTensor1D(&a, 10, X_INT);             // 10维的整数型向量
InitTensor1D(&b, 10);                    // 10维的向量，缺省类型(浮点)
InitTensor4D(&c, 10, 20, 30, 40);        // 10*20*30*40的4阶张量(浮点)
```

- 直接在GPU上定义张量

```
XTensor tensorGPU;                       // 声明张量tensor
InitTensor2D(&tensorGPU, 10, 20, \       // 在编号为0的GPU上定义张量
             X_FLOAT, 0);
```

代数运算

- 各种单元算子（1阶运算）+、-、*、\、Log、Exp、Power、Absolute等，还有Sigmoid、Softmax等激活函数

```
XTensor a, b, c, d, e;                  // 声明张量tensor
InitTensor3D(&a, 2, 3, 4);              // a为2*3*4的3阶张量
InitTensor3D(&b, 2, 3, 4);              // b为2*3*4的3阶张量
InitTensor3D(&c, 2, 3, 4);              // c为2*3*4的3阶张量
a.SetDataRand();                        // 随机初始化a
b.SetDataRand();                        // 随机初始化b
c.SetDataRand();                        // 随机初始化c
d = a + b * c;                          // d被赋值为 a + b * c
d = ((a + b) * d - b / c ) * d;         // d可以被嵌套使用
e = Sigmoid(d);                         // d经过激活函数Sigmoid赋值给e
```

代数运算

- 各种单元算子（1阶运算）+、-、*、\、Log、Exp、Power、Absolute等，还有Sigmoid、Softmax等激活函数

```
XTensor a, b, c, d, e;                  // 声明张量tensor
InitTensor3D(&a, 2, 3, 4);              // a为2*3*4的3阶张量
InitTensor3D(&b, 2, 3, 4);              // b为2*3*4的3阶张量
InitTensor3D(&c, 2, 3, 4);              // c为2*3*4的3阶张量
a.SetDataRand();                        // 随机初始化a
b.SetDataRand();                        // 随机初始化b
c.SetDataRand();                        // 随机初始化c
d = a + b * c;                          // d被赋值为 a + b * c
d = ((a + b) * d - b / c ) * d;         // d可以被嵌套使用
e = Sigmoid(d);                         // d经过激活函数Sigmoid赋值给e
```

- 高阶运算，最常用的是矩阵乘法(MMul)

```
XTensor a, b, c;                        // 声明张量tensor
InitTensor4D(&a, 2, 2, 3, 4);           // a为2*2*3*4的4阶张量
InitTensor2D(&b, 4, 5);                 // b为4*5的矩阵
a.SetDataRand();                        // 随机初始化a
b.SetDataRand();                        // 随机初始化b
c = MMul(a, b);                         // 矩阵乘的结果为2*2*3*5的4阶张量
```

其它常用函数

- 其它函数，列不全，可以参考网站上的详细说明

函数	描述
a.Reshape(o, s)	把a变换为阶为o、形状为s的张量
a.Get(pos)	取张量中位置为pos的元素
a.Set(v, pos)	把张量中位置为pos的元素的值设为v
a.Dump(file)	把张量存到file中，file为文件句柄
a.Read(file)	从file中读取张量，file为文件句柄
Power(a, p)	计算指数a^p
Linear(a, s, b)	计算 a * s + b，s和b都是一个数
CopyValues(a)	构建a的一个拷贝
ReduceMax(a, d)	对a沿着方向d进行规约，得到最大值
ReduceSum(a, d)	对a沿着方向d进行规约，得到和
Concatenate(a, b, d)	把两个张量a和b沿d方向级联
Merge(a, d)	对张量a沿d方向合并
Split(a, d, n)	对张量a沿d方向分裂成n份
Sigmoid(a)	对a进行Sigmoid变换
Softmax(a)	对a进行Softmax变换，沿最后一个方向
HardTanH(a)	对a进行hard tanh变换(双曲正切的近似)
Relu(a)	对a进行Relu变换

构建神经网络

- 可以很方便的构建一个单层网络

```
XTensor x, y, w, b;
InitTensor3D(&x, 3, 4, 5);
InitTensor2D(&w, 5, 3);
InitTensor1D(&b, 3);
...
y = Sigmoid(MMul(x, w) + b);
```

构建神经网络

- 可以很方便的构建一个单层网络

```
XTensor x, y, w, b;
InitTensor3D(&x, 3, 4, 5);
InitTensor2D(&w, 5, 3);
InitTensor1D(&b, 3);
...
y = Sigmoid(MMul(x, w) + b);
```

- 一个多层网络

```
XTensor x, y, h1, h2;
XTensor w1, b1, w2, w3;
InitTensor3D(&x, 3, 4, 5);
InitTensor2D(&w1, 5, 3);
InitTensor1D(&b1, 3);
InitTensor2D(&w2, 3, 6);
InitTensor2D(&w3, 6, 4);
...
h1 = Sigmoid(MMul(x, w1) + b1);
h2 = HandTanH(MMul(h1, w2));
y = Relu(MMul(h2, w3));
```

更复杂一点的例子

- 任何网络都可以构建，比如RNN、Transformer 等

```
XTensor x[3], y[3], r, wh;
XTensor h1, h2, w1, b1, h3, h4;
XList splits;
...
for(unsigned i = 0; i < 3; i++){
    r = Concatenate(x[i] + r) * wh;
    splits.Add(&r);
}
```

更复杂一点的例子

- 任何网络都可以构建，比如RNN、Transformer 等

```
XTensor x[3], y[3], r, wh;
XTensor h1, h2, w1, b1, h3, h4;
XList splits;
...
for(unsigned i = 0; i < 3; i++){
    r = Concatenate(x[i] + r) * wh;
    splits.Add(&r);
}

h1 = Merge(splits, 0);
h2 = Relu(h1 * w1 + b1);
h3 = h1 + h2;
h4 = Softmax(h3);
```

更复杂一点的例子

- 任何网络都可以构建，比如RNN、Transformer 等

```
XTensor x[3], y[3], r, wh;
XTensor h1, h2, w1, b1, h3, h4;
XList splits;
...
for(unsigned i = 0; i < 3; i++){
    r = Concatenate(x[i] + r) * wh;
    splits.Add(&r);
}

h1 = Merge(splits, 0);
h2 = Relu(h1 * w1 + b1);
h3 = h1 + h2;
h4 = Softmax(h3);

Split(h4, splits, 0);

for(unsigned i = 0; i < 3; i++){
    y[i] = *(XTensor*)splits.Get(i);
    y[i].Dump(stdout);
}
```

还有一个问题

如何对模型中的参数进行学习，
之后使用学习到的模型进行推断？

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = ?$$

神经网络 = 函数表达式

- 所有的神经网络都可以看做由变量和函数组成的表达式

$$\mathbf{y} = \mathbf{x} + \mathbf{b}$$

$$\mathbf{y} = \mathrm{Relu}(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$$

$$\mathbf{y} = (\mathrm{Relu}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}) + \mathbf{x}) \cdot \mathbf{w}_2$$

$$\mathbf{y} = \mathrm{Sigmoid}(\mathrm{Relu}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) + \mathbf{x}) \cdot \mathbf{w}_2 + \mathbf{b}_2$$

神经网络 = 函数表达式

- 所有的神经网络都可以看做由变量和函数组成的表达式

$$\mathbf{y} = \mathbf{x} + \mathbf{b}$$

$$\mathbf{y} = \mathrm{Relu}(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$$

$$\mathbf{y} = (\mathrm{Relu}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}) + \mathbf{x}) \cdot \mathbf{w}_2$$

$$\mathbf{y} = \mathrm{Sigmoid}(\mathrm{Relu}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) + \mathbf{x}) \cdot \mathbf{w}_2 + \mathbf{b}_2$$

输入变量 - 由用户指定

神经网络 = 函数表达式

- 所有的神经网络都可以看做由变量和函数组成的表达式

$$\mathbf{y} = \mathbf{x} + \mathbf{b}$$

$$\mathbf{y} = \mathrm{Relu}(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$$

$$\mathbf{y} = (\mathrm{Relu}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}) + \mathbf{x}) \cdot \mathbf{w}_2$$

$$\mathbf{y} = \mathrm{Sigmoid}(\mathrm{Relu}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) + \mathbf{x}) \cdot \mathbf{w}_2 + \mathbf{b}_2$$

输入变量 - 由用户指定

模型参数 - 怎么设置？？？

神经网络 = 函数表达式

- 所有的神经网络都可以看做由变量和函数组成的表达式

$$\mathbf{y} = \mathbf{x} + \mathbf{b}$$

$$\mathbf{y} = \mathrm{Relu}(\mathbf{x} \cdot \mathbf{w} + \mathbf{b})$$

$$\mathbf{y} = (\mathrm{Relu}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}) + \mathbf{x}) \cdot \mathbf{w}_2$$

$$\mathbf{y} = \mathrm{Sigmoid}(\mathrm{Relu}(\mathbf{x} \cdot \mathbf{w}_1 + \mathbf{b}_1) + \mathbf{x}) \cdot \mathbf{w}_2 + \mathbf{b}_2$$

输入变量 - 由用户指定

模型参数 - 怎么设置？？？

问题来了：
如何确定w和b，使x与y对应得更好？

目标函数和损失函数

- 这是一个典型的优化问题，有两个基本问题
  1. 优化的目标是什么？
  2. 如何调整参数**w**和**b**达成目标？

目标函数和损失函数

- 这是一个典型的优化问题，有两个基本问题
  1. 优化的目标是什么？
  2. 如何调整参数**w**和**b**达成目标？
- **定义目标**：对于给定**x**，什么样的**y**是好的
  - 假设：多个输入样本$\{\mathbf{x}_1, ..., \mathbf{x}_n\}$，每个$\mathbf{x}_i$都对应正确答案$\tilde{\mathbf{y}}_i$
  - 对于一个神经网络$\mathbf{y} = f(\mathbf{x})$，每个$\mathbf{x}_i$也会有一个输出$\mathbf{y}_i$
  - 如果可以度量答案$\tilde{\mathbf{y}}_i$和网络输出$\mathbf{y}_i$之间的偏差，进而调整网络参数减小这种偏差，就可以得到更好的模型

目标函数和损失函数

- 这是一个典型的优化问题，有两个基本问题
  1. 优化的目标是什么？
  2. 如何调整参数**w**和**b**达成目标？
- **定义目标**：对于给定**x**，什么样的**y**是好的
  - 假设：多个输入样本$\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$，每个$\mathbf{x}_i$都对应正确答案$\tilde{\mathbf{y}}_i$
  - 对于一个神经网络$\mathbf{y} = f(\mathbf{x})$，每个$\mathbf{x}_i$也会有一个输出$\mathbf{y}_i$
  - 如果可以度量答案$\tilde{\mathbf{y}}_i$和网络输出$\mathbf{y}_i$之间的偏差，进而调整网络参数减小这种偏差，就可以得到更好的模型

目标函数和损失函数

- 这是一个典型的优化问题，有两个基本问题
  1. 优化的目标是什么？
  2. 如何调整参数**w**和**b**达成目标？
- **定义目标**：对于给定**x**，什么样的**y**是好的
  - 假设：多个输入样本$\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$，每个$\mathbf{x}_i$都对应正确答案$\tilde{\mathbf{y}}_i$
  - 对于一个神经网络$\mathbf{y} = f(\mathbf{x})$，每个$\mathbf{x}_i$也会有一个输出$\mathbf{y}_i$
  - 如果可以度量答案$\tilde{\mathbf{y}}_i$和网络输出$\mathbf{y}_i$之间的偏差，进而调整网络参数减小这种偏差，就可以得到更好的模型

目标函数和损失函数

- 这是一个典型的优化问题，有两个基本问题
  1. 优化的目标是什么？
  2. 如何调整参数**w**和**b**达成目标？
- **定义目标**：对于给定**x**，什么样的**y**是好的
  - 假设：多个输入样本$\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$，每个$\mathbf{x}_i$都对应正确答案$\tilde{\mathbf{y}}_i$
  - 对于一个神经网络$\mathbf{y} = f(\mathbf{x})$，每个$\mathbf{x}_i$也会有一个输出$\mathbf{y}_i$
  - 如果可以度量答案$\tilde{\mathbf{y}}_i$和网络输出$\mathbf{y}_i$之间的偏差，进而调整网络参数减小这种偏差，就可以得到更好的模型

目标函数和损失函数

- 这是一个典型的优化问题，有两个基本问题
  1. 优化的目标是什么？
  2. 如何调整参数**w**和**b**达成目标？
- **定义目标**：对于给定**x**，什么样的**y**是好的
  - 假设：多个输入样本$\{\mathbf{x}_1, ..., \mathbf{x}_n\}$，每个$\mathbf{x}_i$都对应正确答案$\tilde{\mathbf{y}}_i$
  - 对于一个神经网络$\mathbf{y} = f(\mathbf{x})$，每个$\mathbf{x}_i$也会有一个输出$\mathbf{y}_i$
  - 如果可以度量答案$\tilde{\mathbf{y}}_i$和网络输出$\mathbf{y}_i$之间的偏差，进而调整网络参数减小这种偏差，就可以得到更好的模型

- 这个过程就是参数优化/训练，而$\tilde{\mathbf{y}}_i$和$\mathbf{y}_i$之间偏差的度量就是一种损失函数，也称作训练的目标函数，而优化的目标就是**最小化损失函数**

常见的损失函数

- 损失函数记为$Loss(\tilde{\mathbf{y}}_i, \mathbf{y}_i)$，简记为$L$，以下是常用的定义

名称	定义	NiuTensor实现(yh表示$\tilde{\mathbf{y}}_i$)	应用
0-1	$L = \begin{cases} 0 & \tilde{\mathbf{y}}_i = \mathbf{y}_i \\ 1 & \tilde{\mathbf{y}}_i \neq \mathbf{y}_i \end{cases}$	`L = Sign(Absolute(yh - y))`	感知机
Hinge	$L = \max(0, 1 - \tilde{\mathbf{y}}_i \cdot \mathbf{y}_i)$	`L = Max(0, 1 - yh * y))`	SVM
绝对值	$L = \lvert\tilde{\mathbf{y}}_i - \mathbf{y}_i\rvert$	`L = Absolute(yh - y)`	回归
Logistic	$L = \log(1 + \tilde{\mathbf{y}}_i \cdot \mathbf{y}_i)$	`L = Log(1 + yh * y)`	回归
平方	$L = (\tilde{\mathbf{y}}_i - \mathbf{y}_i)^2$	`L = Power(yh - y, 2)`	回归
指数	$L = \exp(-\tilde{\mathbf{y}}_i \cdot \mathbf{y}_i)$	`L = Exp(Negate(yh * y))`	AdaBoost
交叉熵	$L = -\sum_k \mathbf{y}_i^{[k]} \log \tilde{\mathbf{y}}_i^{[k]}$ $\mathbf{y}_i^{[k]}$: $\mathbf{y}_i$的第k维	`L = CrossEntropy(y, yh)`	多分类

- 注意：
  - 损失函数可以根据问题不同进行选择，没有固定要求
  - 有些损失函数对网络输出有约束，比如交叉熵要求$\tilde{\mathbf{y}}_i$和$\mathbf{y}_i$都是概率分布

参数优化

- 对于第i个输入样本$(\mathbf{x}_i, \tilde{\mathbf{y}}_i)$，如果把损失函数看做参数$\mathbf{w}$的函数（把$\mathbf{b}$也作为一种$\mathbf{w}$），记为$L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w})$，则参数学习可以被描述为：

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w})$$

$\hat{\mathbf{w}}$表示在训练集上使得损失的平均值达到最小的参数。$\frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w})$被称作代价函数(cost function)，它是损失函数均值期望的估计。

参数优化

- 对于第i个输入样本$(\mathbf{x}_i, \tilde{\mathbf{y}}_i)$，如果把损失函数看做参数$\mathbf{w}$的函数（把$\mathbf{b}$也作为一种$\mathbf{w}$），记为$L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w})$，则参数学习可以被描述为：

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w})$$

  $\hat{\mathbf{w}}$表示在训练集上使得损失的平均值达到最小的参数。$\frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w})$被称作代价函数(cost function)，它是损失函数均值期望的估计。

- 核心问题：**求解$\arg\min$，即找到代价函数最小值点**
  - 这是非常常见的问题，回忆一下第三章的IBM模型，当时使用的是EM算法
  - 但是这里并不是一个生成模型
  - 需要一种更加通用的求解方法

梯度下降（Gradient Descent）

- 如果把目标函数看做是参数$\mathbf{w}$的函数，记为$J(\mathbf{w})$。优化目标是：找到使$J(\mathbf{w})$达到最小的$\mathbf{w}$
- 注意，$\mathbf{w}$可能包含几亿个实数，不能是SMT中MERT之类的调参方法。这里可以考虑一种更加适合大量实数参数的优化方法，其核心思想是梯度下降：

梯度下降（Gradient Descent）

- 如果把目标函数看做是参数$\mathbf{w}$的函数，记为$J(\mathbf{w})$。优化目标是：找到使$J(\mathbf{w})$达到最小的$\mathbf{w}$
- 注意，$\mathbf{w}$可能包含几亿个实数，不能是SMT中MERT之类的调参方法。这里可以考虑一种更加适合大量实数参数的优化方法，其核心思想是梯度下降：
  - 如果$J(\mathbf{w})$对于$\mathbf{w}$可微分，$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$表示$J$在$\mathbf{w}$处变化最大的方向
  - $\mathbf{w}$沿着梯度方向更新，新的$\mathbf{w}$可以使函数更接近极值

梯度下降的不同实现方式

- **梯度下降**：我们可以沿着梯度方向更新**w**一小步，之后得到更好的**w**，之后重新计算梯度，不断重复上述过程

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \cdot \frac{\partial J(\mathbf{w}_t)}{\partial \mathbf{w}_t}$$

其中t表示更新的步数，α是一个参数，表示更新步幅的大小。α的设置需要根据任务进行调整。而$J(\mathbf{w}_t)$的形式决定了具体的算法具体的实现。

梯度下降的不同实现方式

- **梯度下降**：我们可以沿着梯度方向更新**w**一小步，之后得到更好的**w**，之后重新计算梯度，不断重复上述过程

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \cdot \frac{\partial J(\mathbf{w}_t)}{\partial \mathbf{w}_t}$$

其中t表示更新的步数，α是一个参数，表示更新步幅的大小。α的设置需要根据任务进行调整。而$J(\mathbf{w}_t)$的形式决定了具体的算法具体的实现。

- **批量梯度下降(Batch Gradient Descent)**：

$$J(\mathbf{w}_t) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w}_t)$$

这种方法训练稳定，但是由于每次更新需要对所有训练样本进行遍历，效率低（比如n很大），大规模数据上很少使用

梯度下降的不同实现方式(续)

- 随机梯度下降(**Stochastic Gradient Descent**)：

$$J(\mathbf{w}_t) = L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w}_t)$$

大名鼎鼎的SGD，所有机器学习的课程里几乎都有介绍。每次随机选取一个样本进行梯度计算和参数更新，更新的计算代价低，而且适用于利用少量样本进行在线学习(online learning)，不过方法收敛慢

梯度下降的不同实现方式(续)

- **随机梯度下降(Stochastic Gradient Descent)**：

$$J(\mathbf{w}_t) = L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w}_t)$$

大名鼎鼎的SGD，所有机器学习的课程里几乎都有介绍。每次随机选取一个样本进行梯度计算和参数更新，更新的计算代价低，而且适用于利用少量样本进行在线学习(online learning)，不过方法收敛慢

- **小批量梯度下降(Mini-batch Gradient Descent)**：

$$J(\mathbf{w}_t) = \frac{1}{m} \sum_{i=j}^{j+m-1} L(\mathbf{x}_i, \tilde{\mathbf{y}}_i; \mathbf{w}_t)$$

每次随机使用若干样本进行参数更新(数量不会特别大)，算是一种折中方案，当今最常用的方法之一

一些改进

- **变种和改进**：提高基于梯度的方法的收敛速度、训练稳定性等，可以google一下
  - Momentum, Adagrad, Adadelta, RMSprop, Adam, AdaMax, Nadam, AMSGrad等等
  - `http://ruder.io/optimizing-gradient-descent`

一些改进

- **变种和改进**：提高基于梯度的方法的收敛速度、训练稳定性等，可以google一下
  - Momentum, Adagrad, Adadelta, RMSprop, Adam, AdaMax, Nadam, AMSGrad等等
  - `http://ruder.io/optimizing-gradient-descent`
- **并行化**：大规模数据处理需要分布式计算，梯度更新的策略需要设计
  - **同步更新**：所有计算节点完成计算后，统一汇总并更新参数。效果稳定，但是并行度低
  - **异步更新**：每个节点可以随时更新。并行度高，但是由于节点间参数可能不同步，方法不十分稳定

一些改进

- **变种和改进**：提高基于梯度的方法的收敛速度、训练稳定性等，可以google一下
  - Momentum, Adagrad, Adadelta, RMSprop, Adam, AdaMax, Nadam, AMSGrad等等
  - `http://ruder.io/optimizing-gradient-descent`
- **并行化**：大规模数据处理需要分布式计算，梯度更新的策略需要设计
  - **同步更新**：所有计算节点完成计算后，统一汇总并更新参数。效果稳定，但是并行度低
  - **异步更新**：每个节点可以随时更新。并行度高，但是由于节点间参数可能不同步，方法不十分稳定
- **其它**
  - 深度网络梯度消失和爆炸的问题，使用梯度裁剪、残差连接等
  - 引入正则化因子，可以对外部知识建模，比如引入噪声让训练更稳定

如何计算梯度? - 数值微分

- 还有一个核心问题：如何计算梯度

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = ?$$

如何计算梯度? - 数值微分

- 还有一个核心问题：如何计算梯度

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = ?$$

- 数值微分 - 简单粗暴的方法

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \lim_{\Delta \mathbf{w} \to 0} \frac{L(\mathbf{w} + \Delta \mathbf{w}) - L(\mathbf{w} - \Delta \mathbf{w})}{2\Delta \mathbf{w}}$$

最基本的微分公式，我们可以将$\mathbf{w}$变化一点儿（用$\Delta\mathbf{w}$表示），之后看$L(\cdot)$的变化

如何计算梯度？- 数值微分

- 还有一个核心问题：如何计算梯度

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = ?$$

- 数值微分 - 简单粗暴的方法

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \lim_{\Delta \mathbf{w} \to 0} \frac{L(\mathbf{w} + \Delta \mathbf{w}) - L(\mathbf{w} - \Delta \mathbf{w})}{2\Delta \mathbf{w}}$$

最基本的微分公式，我们可以将$\mathbf{w}$变化一点儿（用$\Delta\mathbf{w}$表示），之后看$L(\cdot)$的变化

  - 优点很明显：方法真的非常简单，易于实现
  - 缺点也很明显：效率太低，对于复杂网络、参数量稍微大一些的模型基本上无法使用

如何计算梯度？- 符号微分

- **符号微分**：类似于手写出微分表达式，最后带入变量的值，得到微分结果。比如，对于如下表达式

$$L(\mathbf{w}) = \mathbf{x} \cdot \mathbf{w} + 2\mathbf{w}^2$$

如何计算梯度？- 符号微分

- **符号微分**：类似于手写出微分表达式，最后带入变量的值，得到微分结果。比如，对于如下表达式

$$L(\mathbf{w}) = \mathbf{x} \cdot \mathbf{w} + 2\mathbf{w}^2$$

可以手动推导出微分表达式

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{x} + 4\mathbf{w}$$

如何计算梯度？- 符号微分

- **符号微分**：类似于手写出微分表达式，最后带入变量的值，得到微分结果。比如，对于如下表达式

$$L(\mathbf{w}) = \mathbf{x} \cdot \mathbf{w} + 2\mathbf{w}^2$$

可以手动推导出微分表达式

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{x} + 4\mathbf{w}$$

最后，带入$\mathbf{x} = \begin{pmatrix} 2 \\ -3 \end{pmatrix}$和$\mathbf{w} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$，得到微分结果

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \begin{pmatrix} 2 \\ -3 \end{pmatrix} + 4\begin{pmatrix} -1 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 1 \end{pmatrix}$$

符号微分的膨胀问题

- **Expression Swell**：深层函数的微分表达式会非常复杂
  - 表达式冗长不易存储和管理
  - 真正需要的是微分的结果值，而不是微分表达式

函数	微分表达式	化简的微分表达式
x	1	1
$x(x+1)$	$(x+1)+x$	$2x+1$
$x(x+1)$ (x^2+x+1)	$(x+1)(x^2+x+1)$ $+x(x^2+x+1)$ $+x(x+1)(2x+1)$	$4x^3+6x^2$ $+4x+1$
(x^2+x) (x^2+x+1) $(x^4+2x^3$ $+2x^2+x+1)$	$(2x+1)(x^2+x+1)$ $(x^4+2x^3+2x^2+x+1)$ $+(2x+1)(x^2+x)$ $(x^4+2x^3+2x^2+x+1)$ $+(x^2+x)(x^2+x+1)$ $(4x^3+6x^2+4x+1)$	$8x^7+28x^6$ $+48x^5+50x^4$ $+36x^3+18x^2$ $+6x+1$

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步

1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
2. **反向计算**：从神经网络的输出，逆向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步

1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
2. **反向计算**：从神经网络的输出，逆向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步

1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
2. **反向计算**：从神经网络的输出，逐向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步
  1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
  2. **反向计算**：从神经网络的输出，逐向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步

1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
2. **反向计算**：从神经网络的输出，逆向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步
  1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
  2. **反向计算**：从神经网络的输出，逐向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步

1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
2. **反向计算**：从神经网络的输出，逐向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步

1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
2. **反向计算**：从神经网络的输出，逆向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步

1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
2. **反向计算**：从神经网络的输出，逆向逐层计算每层网络输入（输出）所对应的微分

如何计算梯度？- 自动微分

- **自动微分**：复杂的微分变成简单的步骤，这些步骤完全自动化，而且容易进行存储、计算。这可以用一种反向模式进行描述（也就是反向传播思想），包括两步
  1. **前向计算**：从神经网络的输入，逐层计算每层网络的输出值，这也是神经网络的标准使用方式
  2. **反向计算**：从神经网络的输出，逆向逐层计算每层网络输入（输出）所对应的微分

- 自动微分可以用计算图实现(TensorFlow、 NiuTensor 等)，不过计算图超出了课程的范围，建议大家自行学习

符号说明

- 以一个K层神经网络为例重新明确一下符号
  - 这里假设每层神经网络中都不含偏置项（不含**b**）

符号说明

- 以一个K层神经网络为例重新明确一下符号
  - 这里假设每层神经网络中都不含偏置项（不含**b**）

符号说明

- 以一个K层神经网络为例重新明确一下符号
  - 这里假设每层神经网络中都不含偏置项（不含**b**）

h_i^k：第k层，第i个神经元的输出

$\mathbf{h}^k$：第k层的输出

$\mathbf{s}^k$：第k层的线性变换$\mathbf{s}^k = \mathbf{h}^{k-1}\mathbf{w}^k$

f^k：第k层的激活函数$\mathbf{h}^k = f^k(\mathbf{s}^k)$

符号说明

- 以一个K层神经网络为例重新明确一下符号
  - 这里假设每层神经网络中都不含偏置项（不含**b**）

h_i^k：第k层，第i个神经元的输出

$\mathbf{h}^k$：第k层的输出

$\mathbf{s}^k$：第k层的线性变换$\mathbf{s}^k = \mathbf{h}^{k-1}\mathbf{w}^k$

f^k：第k层的激活函数$\mathbf{h}^k = f^k(\mathbf{s}^k)$

$\mathbf{h}^K$：网络最后的输出

符号说明

- 以一个K层神经网络为例重新明确一下符号
  - 这里假设每层神经网络中都不含偏置项（不含**b**）

h_i^k：第k层，第i个神经元的输出

$\mathbf{h}^k$：第k层的输出

$\mathbf{s}^k$：第k层的线性变换$\mathbf{s}^k = \mathbf{h}^{k-1}\mathbf{w}^k$

f^k：第k层的激活函数$\mathbf{h}^k = f^k(\mathbf{s}^k)$

$\mathbf{h}^K$：网络最后的输出

$w_{j,i}^k$：第$k-1$层神经元j与第k层神经元i的连接权重

$\mathbf{w}^k$：第$k-1$层与第k层的连接权重

符号说明

- 以一个K层神经网络为例重新明确一下符号
  - 这里假设每层神经网络中都不含偏置项（不含**b**）

h_i^k：第k层，第i个神经元的输出

$\mathbf{h}^k$：第k层的输出

$\mathbf{s}^k$：第k层的线性变换$\mathbf{s}^k = \mathbf{h}^{k-1}\mathbf{w}^k$

f^k：第k层的激活函数$\mathbf{h}^k = f^k(\mathbf{s}^k)$

$\mathbf{h}^K$：网络最后的输出

$w_{j,i}^k$：第$k-1$层神经元j与第k层神经元i的连接权重

$\mathbf{w}^k$：第$k-1$层与第k层的连接权重

$$\text{对于第}k\text{层}: \mathbf{h}^k = f^k(\mathbf{s}^k) = f^k(\sum_j h_j^{k-1} w_{j,i}^k) = f^k(\mathbf{h}^{k-1}\mathbf{w}^k)$$

反向传播 - 输出层

- 输出层(两个阶段)

反向传播 - 输出层

- 输出层(两个阶段)

反向传播 - 输出层

- 输出层(两个阶段)

反向传播 - 输出层

- 输出层(两个阶段)

- 反向传播从输出向输入传播梯度，因此我们先考虑阶段二

反向传播 - 输出层

- 输出层(两个阶段)

- 反向传播从输出向输入传播梯度，因此我们先考虑阶段二。令$\pi^k = \frac{\partial L}{\partial \mathbf{s}^k}$表示损失$L$在第$k$层激活函数输入处的梯度，利用链式法有

$$
\begin{aligned}
\pi^K &= \frac{\partial L}{\partial \mathbf{s}^K} \\
&= \frac{\partial L}{\partial \mathbf{h}^K} \cdot \frac{\partial \mathbf{h}^K}{\partial \mathbf{s}^K} \\
&= \frac{\partial L}{\partial \mathbf{h}^K} \cdot \frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}
\end{aligned}
$$

反向传播 - 输出层($\mathbf{s}^K$处的梯度)

$$\pi^K = \frac{\partial L}{\partial \mathbf{h}^K} \cdot \frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$$

反向传播 - 输出层($\mathbf{s}^K$处的梯度)

$$\pi^K = \boxed{\frac{\partial L}{\partial \mathbf{h}^K}} \cdot \frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$$

- $\boxed{\frac{\partial L}{\partial \mathbf{h}^K}}$ 表示损失L相对网络输出的变化率，比如，对于$L = \frac{1}{2}||\tilde{\mathbf{y}} - \mathbf{h}^K||^2$，有$\frac{\partial L}{\partial \mathbf{h}^K} = \tilde{\mathbf{y}} - \mathbf{h}^K$

反向传播 - 输出层($\mathbf{s}^K$处的梯度)

$$\pi^K = \frac{\partial L}{\partial \mathbf{h}^K} \cdot \frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$$

- $\frac{\partial L}{\partial \mathbf{h}^K}$ 表示损失L相对网络输出的变化率，比如，对于$L = \frac{1}{2}||\tilde{\mathbf{y}} - \mathbf{h}^K||^2$，有$\frac{\partial L}{\partial \mathbf{h}^K} = \tilde{\mathbf{y}} - \mathbf{h}^K$
- $\frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$ 表示激活函数相对于它自己的输入的变化率，比如，对于$f(\mathbf{s}) = \frac{1}{1+\exp(-\mathbf{s})}$，有$\frac{\partial f(\mathbf{s})}{\partial \mathbf{s}} = f(\mathbf{s})(1 - f(\mathbf{s}))$

反向传播 - 输出层($\mathbf{s}^K$处的梯度)

$$\pi^K = \frac{\partial L}{\partial \mathbf{h}^K} \cdot \frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$$

- $\frac{\partial L}{\partial \mathbf{h}^K}$ 表示损失L相对网络输出的变化率，比如，对于$L = \frac{1}{2}||\tilde{\mathbf{y}} - \mathbf{h}^K||^2$，有$\frac{\partial L}{\partial \mathbf{h}^K} = \tilde{\mathbf{y}} - \mathbf{h}^K$
- $\frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$ 表示激活函数相对于它自己的输入的变化率，比如，对于$f(\mathbf{s}) = \frac{1}{1+\exp(-\mathbf{s})}$，有$\frac{\partial f(\mathbf{s})}{\partial \mathbf{s}} = f(\mathbf{s})(1 - f(\mathbf{s}))$
- 这个结果符合直觉，在s^K出的梯度相当于在损失函数微分($\frac{\partial L}{\partial \mathbf{h}^K}$)和激活函数微分($\frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$) 的乘积

反向传播 - 输出层($\mathbf{s}^K$处的梯度)

$$\pi^K = \frac{\partial L}{\partial \mathbf{h}^K} \cdot \frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$$

- $\frac{\partial L}{\partial \mathbf{h}^K}$ 表示损失L相对网络输出的变化率，比如，对于$L = \frac{1}{2}||\tilde{\mathbf{y}} - \mathbf{h}^K||^2$，有$\frac{\partial L}{\partial \mathbf{h}^K} = \tilde{\mathbf{y}} - \mathbf{h}^K$
- $\frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$ 表示激活函数相对于它自己的输入的变化率，比如，对于$f(\mathbf{s}) = \frac{1}{1+\exp(-\mathbf{s})}$，有$\frac{\partial f(\mathbf{s})}{\partial \mathbf{s}} = f(\mathbf{s})(1 - f(\mathbf{s}))$
- 这个结果符合直觉，在s^K出的梯度相当于在损失函数微分($\frac{\partial L}{\partial \mathbf{h}^K}$)和激活函数微分($\frac{\partial f^K(\mathbf{s}^K)}{\partial \mathbf{s}^K}$) 的乘积，注意这里所有操作都是单元级，比如张量按单元乘法

反向传播 - 输出层($\mathbf{h}^{K-1}$处的梯度)

- 已经得到$\mathbf{s}^K$处的梯度

反向传播 - 输出层($\mathbf{h}^{K-1}$处的梯度)

- 已经得到$\mathbf{s}^K$处的梯度，下面求解两个问题
  1. 计算损失L对于第K层参数矩阵$\mathbf{w}^K$的梯度，$\frac{\partial L}{\partial \mathbf{w}^K}$
  2. 计算损失L对于第K层输入$\mathbf{h}^{K-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{K-1}}$

反向传播 - 输出层($\mathbf{h}^{K-1}$处的梯度)

- 已经得到$\mathbf{s}^K$处的梯度，下面求解两个问题
  1. 计算损失L对于第K层参数矩阵$\mathbf{w}^K$的梯度，$\frac{\partial L}{\partial \mathbf{w}^K}$
  2. 计算损失L对于第K层输入$\mathbf{h}^{K-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{K-1}}$

- 由于$\mathbf{s}^K = \mathbf{h}^{K-1}\mathbf{w}^K$，而且$\pi^K = \frac{\partial L}{\partial \mathbf{s}^K}$已经求解，可以得到(需要一些数学分析和线性代数的知识，推导一下!)：

$$\frac{\partial L}{\partial \mathbf{w}^K} = [\mathbf{h}^{K-1}]^T \pi^K$$
$$\frac{\partial L}{\partial \mathbf{h}^{K-1}} = \pi^K [\mathbf{w}^K]^T$$

这里，$[\mathbf{A}]^T$表示$\mathbf{A}$的转置，$\pi^K[\mathbf{w}^K]^T$表示张量π^K矩阵乘$\mathbf{w}^K$的转置

反向传播 - 隐层

- 对于任意隐层k，$\mathbf{h}^k = f^k(\mathbf{s}^k) = f^k(\mathbf{h}^{k-1}\mathbf{w}^k)$。给定：隐层输出处的梯度$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}$，需要

1. 计算损失L对于第k层参数矩阵$\mathbf{w}^k$的梯度，$\frac{\partial L}{\partial \mathbf{w}^k}$
2. 计算损失L对于第k层输入$\mathbf{h}^{k-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{k-1}}$

反向传播 - 隐层

- 对于任意隐层k，$\mathbf{h}^k = f^k(\mathbf{s}^k) = f^k(\mathbf{h}^{k-1}\mathbf{w}^k)$。给定：隐层输出处的梯度$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}$，需要
  1. 计算损失L对于第k层参数矩阵$\mathbf{w}^k$的梯度，$\frac{\partial L}{\partial \mathbf{w}^k}$
  2. 计算损失L对于第k层输入$\mathbf{h}^{k-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{k-1}}$
- 直接套用上一页的方法，可以将$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}\frac{\partial f^k(\mathbf{s}^k)}{\partial \mathbf{s}^k}$反向传播

$$
\begin{aligned}
\frac{\partial L}{\partial \mathbf{w}^k} &= [\mathbf{h}^{k-1}]^T \pi^k \\
\frac{\partial L}{\partial \mathbf{h}^{k-1}} &= \pi^k [\mathbf{w}^k]^T
\end{aligned}
$$

反向传播 - 隐层

- 对于任意隐层k，$\mathbf{h}^k = f^k(\mathbf{s}^k) = f^k(\mathbf{h}^{k-1}\mathbf{w}^k)$。给定：隐层输出处的梯度$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}$，需要
  1. 计算损失L对于第k层参数矩阵$\mathbf{w}^k$的梯度，$\frac{\partial L}{\partial \mathbf{w}^k}$
  2. 计算损失L对于第k层输入$\mathbf{h}^{k-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{k-1}}$
- 直接套用上一页的方法，可以将$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}\frac{\partial f^k(\mathbf{s}^k)}{\partial \mathbf{s}^k}$反向传播

$$
\begin{aligned}
\frac{\partial L}{\partial \mathbf{w}^k} &= [\mathbf{h}^{k-1}]^T \pi^k \\
\frac{\partial L}{\partial \mathbf{h}^{k-1}} &= \pi^k [\mathbf{w}^k]^T
\end{aligned}
$$

$$
\cdots \longrightarrow \boxed{\mathbf{h}^{k-1}} \xrightarrow[\mathbf{s}^k = \mathbf{h}^{k-1}\mathbf{w}^k]{} \boxed{\mathbf{s}^k} \xrightarrow[\mathbf{h}^k = f(\mathbf{s}^k)]{} \boxed{\mathbf{h}^k} \longrightarrow \cdots
$$

反向传播 - 隐层

- 对于任意隐层k，$\mathbf{h}^k = f^k(\mathbf{s}^k) = f^k(\mathbf{h}^{k-1}\mathbf{w}^k)$。给定：隐层输出处的梯度$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}$，需要
  1. 计算损失L对于第k层参数矩阵$\mathbf{w}^k$的梯度，$\frac{\partial L}{\partial \mathbf{w}^k}$
  2. 计算损失L对于第k层输入$\mathbf{h}^{k-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{k-1}}$
- 直接套用上一页的方法，可以将$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}\frac{\partial f^k(\mathbf{s}^k)}{\partial \mathbf{s}^k}$反向传播

$$
\begin{aligned}
\frac{\partial L}{\partial \mathbf{w}^k} &= [\mathbf{h}^{k-1}]^T \pi^k \\
\frac{\partial L}{\partial \mathbf{h}^{k-1}} &= \pi^k [\mathbf{w}^k]^T
\end{aligned}
$$

反向传播 - 隐层

- 对于任意隐层k，$\mathbf{h}^k = f^k(\mathbf{s}^k) = f^k(\mathbf{h}^{k-1}\mathbf{w}^k)$。给定：隐层输出处的梯度$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}$，需要
  1. 计算损失L对于第k层参数矩阵$\mathbf{w}^k$的梯度，$\frac{\partial L}{\partial \mathbf{w}^k}$
  2. 计算损失L对于第k层输入$\mathbf{h}^{k-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{k-1}}$
- 直接套用上一页的方法，可以将$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}\frac{\partial f^k(\mathbf{s}^k)}{\partial \mathbf{s}^k}$反向传播

$$\frac{\partial L}{\partial \mathbf{w}^k} = [\mathbf{h}^{k-1}]^T \pi^k$$

$$\frac{\partial L}{\partial \mathbf{h}^{k-1}} = \pi^k [\mathbf{w}^k]^T$$

反向传播 - 隐层

- 对于任意隐层k，$\mathbf{h}^k = f^k(\mathbf{s}^k) = f^k(\mathbf{h}^{k-1}\mathbf{w}^k)$。给定：隐层输出处的梯度$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}$，需要
  1. 计算损失L对于第k层参数矩阵$\mathbf{w}^k$的梯度，$\frac{\partial L}{\partial \mathbf{w}^k}$
  2. 计算损失L对于第k层输入$\mathbf{h}^{k-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{k-1}}$
- 直接套用上一页的方法，可以将$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}\frac{\partial f^k(\mathbf{s}^k)}{\partial \mathbf{s}^k}$反向传播

$$
\begin{aligned}
\frac{\partial L}{\partial \mathbf{w}^k} &= [\mathbf{h}^{k-1}]^T \pi^k \\
\frac{\partial L}{\partial \mathbf{h}^{k-1}} &= \pi^k [\mathbf{w}^k]^T
\end{aligned}
$$

反向传播 - 隐层

- 对于任意隐层k，$\mathbf{h}^k = f^k(\mathbf{s}^k) = f^k(\mathbf{h}^{k-1}\mathbf{w}^k)$。给定：隐层输出处的梯度$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}$，需要
  1. 计算损失L对于第k层参数矩阵$\mathbf{w}^k$的梯度，$\frac{\partial L}{\partial \mathbf{w}^k}$
  2. 计算损失L对于第k层输入$\mathbf{h}^{k-1}$的梯度，$\frac{\partial L}{\partial \mathbf{h}^{k-1}}$
- 直接套用上一页的方法，可以将$\pi^k = \frac{\partial L}{\partial \mathbf{h}^k}\frac{\partial f^k(\mathbf{s}^k)}{\partial \mathbf{s}^k}$反向传播

$$\frac{\partial L}{\partial \mathbf{w}^k} = [\mathbf{h}^{k-1}]^T \pi^k$$
$$\frac{\partial L}{\partial \mathbf{h}^{k-1}} = \pi^k [\mathbf{w}^k]^T$$

反向传播的实现

- 对于一个多层神经网络很容易实现反向传播

```
XTensor x, y, gold, h[5], w[5], s[5];
XTensor dh[5], dw[5], ds[5];
... // 前向过程
h[0] = x;
y = h[4];
```

反向传播的实现

- 对于一个多层神经网络很容易实现反向传播

```
XTensor x, y, gold, h[5], w[5], s[5];
XTensor dh[5], dw[5], ds[5];
... // 前向过程
h[0] = x;
y = h[4];

CrossEntropyBackward(dh[4], y, gold);
SoftmaxBackward(y, s[4], dh[4], ds[4]);
MMul(h[3], X_TRANS, ds[4], X_NOTRANS, dw[4]);
MMul(ds[4], X_NOTRANS, w[4], X_RANS, dh[3]);
```

反向传播的实现

- 对于一个多层神经网络很容易实现反向传播

```
XTensor x, y, gold, h[5], w[5], s[5];
XTensor dh[5], dw[5], ds[5];
... // 前向过程
h[0] = x;
y = h[4];

CrossEntropyBackward(dh[4], y, gold);
SoftmaxBackward(y, s[4], dh[4], ds[4]);
MMul(h[3], X_TRANS, ds[4], X_NOTRANS, dw[4]);
MMul(ds[4], X_NOTRANS, w[4], X_RANS, dh[3]);

dh[2] = dh[3];
ReluBackward(h[2], s[2], dh[2], ds[2]);
MMul(h[1], X_TRANS, ds[2], X_NOTRANS, dw[2]);
MMul(ds[2], X_NOTRANS, w[2], X_TRANS, dh[2]);
```

反向传播的实现

- 对于一个多层神经网络很容易实现反向传播

```
XTensor x, y, gold, h[5], w[5], s[5];
XTensor dh[5], dw[5], ds[5];
... // 前向过程
h[0] = x;
y = h[4];

CrossEntropyBackward(dh[4], y, gold);
SoftmaxBackward(y, s[4], dh[4], ds[4]);
MMul(h[3], X_TRANS, ds[4], X_NOTRANS, dw[4]);
MMul(ds[4], X_NOTRANS, w[4], X_RANS, dh[3]);

dh[2] = dh[3];
ReluBackward(h[2], s[2], dh[2], ds[2]);
MMul(h[1], X_TRANS, ds[2], X_NOTRANS, dw[2]);
MMul(ds[2], X_NOTRANS, w[2], X_TRANS, dh[2]);

dh[1] = dh[1] + dh[3];
```

反向传播的实现

- 对于一个多层神经网络很容易实现反向传播

```
XTensor x, y, gold, h[5], w[5], s[5];
XTensor dh[5], dw[5], ds[5];
... // 前向过程
h[0] = x;
y = h[4];

CrossEntropyBackward(dh[4], y, gold);
SoftmaxBackward(y, s[4], dh[4], ds[4]);
MMul(h[3], X_TRANS, ds[4], X_NOTRANS, dw[4]);
MMul(ds[4], X_NOTRANS, w[4], X_RANS, dh[3]);

dh[2] = dh[3];
ReluBackward(h[2], s[2], dh[2], ds[2]);
MMul(h[1], X_TRANS, ds[2], X_NOTRANS, dw[2]);
MMul(ds[2], X_NOTRANS, w[2], X_TRANS, dh[2]);

dh[1] = dh[1] + dh[3];
  ... // 继续反向传播

for(unsigned i = 0; i < 5; i++){
    ... // 通过dw[i]访问参数的梯度
}
```

更简单的实现

- 幸运的是，现在几乎所有的主流深度学习框架都实现了自动微分，一个函数可以搞定

```
XTensor x, loss, gold, h[5], w[5], b[5];
...

h[1] = Relu(MMul(x, w[1]) + b[1]);
h[2] = Relu(MMul(h[1], w[2]) + b[2]);
h[3] = HardTanH(h[2]);
h[4] = Softmax(MMul(h[3], w[3]));
loss = CrossEntropy(h[4], gold);

XNet net;
net.Backward(loss); //一行代码实现自动微分

for(unsigned i = 0; i < 5; i++){
    ... // 通过w[i].grad访问参数的梯度
}
```

- 其它优秀的自动微分实现也可以参考TensorFlow、PyTorch等工具

前向计算及其它问题

- 前向计算实际上就是网络构建的过程，有两种常用方式
  - **动态图**(如PyTorch、NiuTensor)：写完函数表达式，前向计算即完成，易于调试
  - **静态图**(如TensorFlow)：函数表达式完成后，并不能得到前向计算结果，需要显性调用一个Forward函数，但是计算图可以进行深度优化，执行效率较高

前向计算及其它问题

- 前向计算实际上就是网络构建的过程，有两种常用方式
  - **动态图**(如PyTorch、NiuTensor)：写完函数表达式，前向计算即完成，易于调试
  - **静态图**(如TensorFlow)：函数表达式完成后，并不能得到前向计算结果，需要显性调用一个Forward函数，但是计算图可以进行深度优化，执行效率较高
- 其它一些深度学习系统实现的问题，值得关注，不过这些都超出了本课程的范围
  - **分布式训练**：对于复杂模型的海量数据训练，需要利用多个设备（多机、多卡）同时训练
  - **低精度计算**：为了提高效率可以采用半精度或者定点数进行计算
  - **模型压缩**：减少冗余，可以压缩模型，使得模型易于存储同时提高系统运行效率
  - **训练方法和超参选择**：不同任务往往需要不同的训练策略，包括超参设置，坑很多，需要积累经验

进入正题

如何将神经元网络应用到**NLP**？
- 语言模型的神经网络建模

自然语言处理遇到神经网络

- 神经网络方法给自然语言处理(NLP)带来了新的思路

传统基于统计的方法	深度学习方法
基于离散空间的表示模型	基于连续空间的表示模型
NLP问题的隐含结构假设	无隐含结构假设，端到端学习
特征工程为主	无显性特征，但需要设计网络
特征、规则的存储耗资源	模型存储相对小，但计算慢

自然语言处理遇到神经网络

- 神经网络方法给自然语言处理(NLP)带来了新的思路

传统基于统计的方法	深度学习方法
基于离散空间的表示模型	基于连续空间的表示模型
NLP问题的隐含结构假设	无隐含结构假设，端到端学习
特征工程为主	无显性特征，但需要设计网络
特征、规则的存储耗资源	模型存储相对小，但计算慢

- 语言模型任务也可以使用深度学习方法(效果非常好)
  - 语言模型要回答的问题是如何评价一个词串的好坏
  - 可以回忆一下第二章提到的n元语法模型

$$\mathbf{P}(w_0 w_1 ... w_m) = ?$$

自然语言处理遇到神经网络

- 神经网络方法给自然语言处理(NLP)带来了新的思路

传统基于统计的方法	深度学习方法
基于离散空间的表示模型	基于连续空间的表示模型
NLP问题的隐含结构假设	无隐含结构假设，端到端学习
特征工程为主	无显性特征，但需要设计网络
特征、规则的存储耗资源	模型存储相对小，但计算慢

- 语言模型任务也可以使用深度学习方法(效果非常好)
  - 语言模型要回答的问题是如何评价一个词串的好坏
  - 可以回忆一下第二章提到的n元语法模型

$$\mathbf{P}(w_0 w_1 ... w_m) = ?$$

如何对词串的生成概率进行建模？

n-gram语言模型

- 链式法则

$$
\begin{aligned}
\mathrm{P}(w_1 w_2 ... w_m) &= \mathrm{P}(w_1)\mathrm{P}(w_2|w_1)\mathrm{P}(w_3|w_1 w_2)... \\
&\quad \mathrm{P}(w_m|w_1 ... w_{m-1})
\end{aligned}
$$

n-gram语言模型

- 链式法则

$$\begin{aligned} \mathrm{P}(w_1 w_2 \ldots w_m) &= \mathrm{P}(w_1)\mathrm{P}(w_2|w_1)\mathrm{P}(w_3|w_1 w_2)\ldots \\ &\quad \mathrm{P}(w_m|w_1 \ldots w_{m-1}) \end{aligned}$$

- 传统n-gram语言模型：当前词仅依赖于前面$n-1$个词

$$\begin{aligned} \mathrm{P}(w_1 w_2 \ldots w_m) &= \mathrm{P}(w_1)\mathrm{P}(w_2|w_1)\mathrm{P}(w_3|w_1 w_2)\ldots \\ &\quad \mathrm{P}(w_m|\underbrace{w_{m-n+1}\ldots w_{m-1}}_{\text{前面}n-1\text{个词}}) \end{aligned}$$

其中

$$\mathrm{P}(w_m|w_{m-n+1}\ldots w_{m-1}) = \frac{\mathrm{count}(w_{m-n+1}\ldots w_m)}{\mathrm{count}(w_{m-n+1}\ldots w_{m-1})}$$

$\mathrm{count}(\cdot)$表示在训练数据上统计的频次

n-gram生成概率的神经网络建模

- 传统的n-gram语言模型实际上就是一个查询表，用$w_{m-n+1}...w_m$查询n-gram概率$\mathrm{P}(w_m|w_{m-n+1}...w_{m-1})$
  - 这张表本质上是一种$w_{m-n+1}...w_m$的离散表示
  - 随着n的增大，数据稀疏问题会非常严重，因为绝大多数n-gram是没见过的
  - 因为要维护n-gram的索引，存储消耗大

n-gram生成概率的神经网络建模

- 传统的n-gram语言模型实际上就是一个查询表，用$w_{m-n+1}...w_m$查询n-gram概率$\mathrm{P}(w_m|w_{m-n+1}...w_{m-1})$
  - 这张表本质上是一种$w_{m-n+1}...w_m$的离散表示
  - 随着n的增大，数据稀疏问题会非常严重，因为绝大多数n-gram是没见过的
  - 因为要维护n-gram的索引，存储消耗大
- 另一种思路是直接对$\mathrm{P}(w_m|w_{m-n+1}...w_{m-1})$进行连续空间建模，即定义函数$g$，对于任意的$w_{m-n+1}...w_m$有

$$g(w_{m-n+1}...w_m) \approx \mathrm{P}(w_m|w_{m-n+1}...w_{m-1})$$

n-gram生成概率的神经网络建模

- 传统的n-gram语言模型实际上就是一个查询表，用$w_{m-n+1}...w_m$查询n-gram概率$\mathrm{P}(w_m|w_{m-n+1}...w_{m-1})$
  - 这张表本质上是一种$w_{m-n+1}...w_m$的离散表示
  - 随着n的增大，数据稀疏问题会非常严重，因为绝大多数n-gram是没见过的
  - 因为要维护n-gram的索引，存储消耗大
- 另一种思路是直接对$\mathrm{P}(w_m|w_{m-n+1}...w_{m-1})$进行连续空间建模，即定义函数$g$，对于任意的$w_{m-n+1}...w_m$有

$$g(w_{m-n+1}...w_m) \approx \mathrm{P}(w_m|w_{m-n+1}...w_{m-1})$$

- 最具代表性的方法是前馈神经网络(FNN)语言模型
  - 经典中的经典，对现代神经语言模型的设计产生深远影响

A Neural Probabilistic Language Model
Bengio et al., 2003, Journal of Machine Learning Research 3: 1137-1155

前馈神经网络语言模型(Bengio et al., 2003)

- 以4-gram语言模型为例

前馈神经网络语言模型(Bengio et al., 2003)

- 以4-gram语言模型为例

One-hot表示

每个词用一个词汇表大小的0-1向量表示，仅一位为1，其余为0，比如：

$$(0, 0, \underset{\uparrow}{1}, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

词表中第3个词

前馈神经网络语言模型(Bengio et al., 2003)

- 以4-gram语言模型为例

词的分布式表示

词的0-1表示乘一个矩阵**C**，这里可以把**C**看做一个查询表

One-hot表示

每个词用一个词汇表大小的0-1向量表示，仅一位为1，其余为0，比如：

$$(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

↑ 词表中第3个词

前馈神经网络语言模型(Bengio et al., 2003)

- 以4-gram语言模型为例

One-hot表示

每个词用一个词汇表大小的0-1向量表示，仅一位为1，其余为0，比如：

$$(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

↑ 词表中第3个词

词的分布式表示

词的0-1表示乘一个矩阵**C**，这里可以把**C**看做一个查询表

$$w_{i-1} \qquad\qquad \mathbf{C}$$

$$(0, 0, 1, \ldots) \times \begin{pmatrix} 0 & 1 & 3 \\ .2 & -1 & .3 \\ 1 & 7 & .3 \\ \ldots & & \end{pmatrix}$$

在把**C**中索引到的行输出(i.e., e_{i-1})

前馈神经网络语言模型(Bengio et al., 2003)

- 以4-gram语言模型为例

One-hot表示

每个词用一个词汇表大小的0-1向量表示，仅一位为1，其余为0，比如：

$$(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

↑ 词表中第3个词

多层神经网络

$[e_0, e_1, e_2]$表示把三个向量级联在一起，之后经过两层网络，最后通过Softmax输出

注意，$h_0\mathbf{U}$得到所有词的表示(向量)，Softmax确保输出词汇表上的一个分布

词的分布式表示

词的0-1表示乘一个矩阵**C**，这里可以把**C**看做一个查询表

$$\underset{(0,0,1,\ldots)}{w_{i-1}} \times \overset{\mathbf{C}}{\begin{pmatrix} 0 & 1 & 3 \\ .2 & -1 & .3 \\ 1 & 7 & .3 \\ \ldots & & \end{pmatrix}}$$

在把**C**中索引到的行输出(i.e., e_{i-1})

前馈神经网络语言模型(Bengio et al., 2003)

- 以4-gram语言模型为例

One-hot表示

每个词用一个词汇表大小的0-1向量表示，仅一位为1，其余为0，比如：

$$(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

↑ 词表中第3个词

底层向上层的直接连接(可选)

多层神经网络

$[e_0, e_1, e_2]$表示把三个向量级联在一起，之后经过两层网络，最后通过Softmax输出

注意，$h_0\mathbf{U}$得到所有词的表示(向量)，Softmax确保输出词汇表上的一个分布

词的分布式表示

词的0-1表示乘一个矩阵**C**，这里可以把**C**看做一个查询表

$$w_{i-1} \qquad\qquad \mathbf{C}$$

$$(0, 0, 1, \ldots) \times \begin{pmatrix} 0 & 1 & 3 \\ .2 & -1 & .3 \\ 1 & 7 & .3 \\ \ldots & & \end{pmatrix}$$

在把**C**中索引到的行输出(i.e., e_{i-1})

前馈神经网络语言模型(FNN LM)的实现

- 实现非常简单，几行代码
  - 细节1：做batching时可以把$w[i]$进行扩展，比如放入多个词
  - 细节2：TanH一般会用HardTanH实现，因为TanH容易溢出

```
XTensor w[3], e[3], h0, y;
XTensor C, H, d, U;
...

for(unsigned i = 0; i < 3; i++)
    e[i] = MMul(w[i], C);
e01 = Concatenate(e[0], e[1], -1);
e = Concatenate(e01, e[2], -1);

h0 = TanH(MMul(e, H) + d);
y = Softmax(MMul(h0, U));

for(unsigned k = 0; k < size; k++){
    ... // y的第k元素表示 P(w|...)
    ... // w为词汇表里第k个词
}
```

注：size表示词汇表大小

神经语言建模的意义

- Bengio el al. (2003)中有待讨论的问题

1. 神经网络每一层究竟学到了什么
   词汇、句法？还是其它一些知识？如何解释？
2. 网络的层数变多会怎样 - 10层、20层、100层的网络
   # of layers: 10 $\rightarrow$ 20 $\rightarrow$ 100 $\rightarrow$ 1000
3. 超参(比如隐藏层大小)如何选择 - 不同任务的最优设置
   单词的分布式表示维度多大好？
   隐层多大好？
   激活函数如何选择？
   ...

神经语言建模的意义

- Bengio el al. (2003)中有待讨论的问题
  1. 神经网络每一层究竟学到了什么
     词汇、句法？还是其它一些知识？如何解释？
  2. 网络的层数变多会怎样 - 10层、20层、100层的网络
     # of layers: 10 → 20 → 100 → 1000
  3. 超参(比如隐藏层大小)如何选择 - 不同任务的最优设置
     单词的分布式表示维度多大好？
     隐层多大好？
     激活函数如何选择？
     ...
- 从FNN LM得到的启发
  - 重新定义词是什么 - 非词典里的一项，而是一个实数向量
  - 多层神经网络可以很好的表示单词之间的(短距离)依赖
  - n-gram的生成概率可以使用连续空间函数描述，缓解数据稀疏问题，模型并不需要记录完整的n-gram

循环神经网络(Recurrent Neural Networks)

- FNN LM固然有效，但是和传统的n-gram LM一样，需要依赖有限上下文假设

循环神经网络(Recurrent Neural Networks)

- FNN LM固然有效，但是和传统的n-gram LM一样，需要依赖有限上下文假设

- 能否直接对原始问题建模，即定义函数g，对于任意的$w_1 \ldots w_m$有

$$g(w_1 \ldots w_m) \approx \mathrm{P}(w_m | w_1 \ldots w_{m-1})$$

循环神经网络(Recurrent Neural Networks)

- FNN LM固然有效，但是和传统的n-gram LM一样，需要依赖有限上下文假设

- 能否直接对原始问题建模，即定义函数g，对于任意的$w_1 \ldots w_m$有

$$g(w_1 \ldots w_m) \approx \mathrm{P}(w_m | w_1 \ldots w_{m-1})$$

- **循环神经网络(RNNs)**可以很好的解决上述问题，因此也被成功的应用于语言建模任务
  - 它假设每个词的生成都依赖已经生成的所有词
  - 对于不同位置的词的生成概率都可以用同一个函数描述

Recurrent Neural Network Based Language Model Mikolov et al., 2010, In Proc. of Interspeech, 1045-1048

循环单元

- 有输入序列$(\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_t, ...)$，其中$\mathbf{x}_t$表示序列中第$t$个元素，也被称作时刻$t$输入。它所对应的输出序列是$(\mathbf{y}_0, \mathbf{y}_1, ..., \mathbf{y}_t, ...)$。 在循环神经网络中，每个时刻的输出都可以用同一个循环单元来描述。

循环单元

- 有输入序列$(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_t, \ldots)$，其中$\mathbf{x}_t$表示序列中第$t$个元素，也被称作时刻$t$输入。它所对应的输出序列是$(\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_t, \ldots)$。在循环神经网络中，每个时刻的输出都可以用同一个循环单元来描述。对于语言模型，一种简单的结构：

$$
\begin{aligned}
\mathbf{y}_t &= \mathrm{Softmax}(\mathbf{h}_t \mathbf{V}) \\
\mathbf{h}_t &= \mathrm{TanH}(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W})
\end{aligned}
$$

$\mathbf{h}_t$: t时刻的隐层状态
$\mathbf{h}_{t-1}$: $t-1$时刻的隐层状态
$\mathbf{V}, \mathbf{U}, \mathbf{W}$: 参数

循环单元

- 有输入序列$(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_t, \ldots)$，其中$\mathbf{x}_t$表示序列中第$t$个元素，也被称作时刻$t$输入。它所对应的输出序列是$(\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_t, \ldots)$。在循环神经网络中，每个时刻的输出都可以用同一个循环单元来描述。对于语言模型，一种简单的结构：

$$
\begin{aligned}
\mathbf{y}_t &= \mathrm{Softmax}(\mathbf{h}_t \mathbf{V}) \\
\mathbf{h}_t &= \mathrm{TanH}(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W})
\end{aligned}
$$

$\mathbf{h}_t$: t时刻的隐层状态
$\mathbf{h}_{t-1}$: $t-1$时刻的隐层状态
$\mathbf{V}, \mathbf{U}, \mathbf{W}$: 参数

- **如何体现循环？**t时刻的状态是$t-1$时刻状态的函数，这个过程可以不断被执行

循环神经网络的“记忆”

- 循环神经网络可以记忆任意长度的历史，因此可以非常适合处理不定长的序列，比如自然语言句子
  - 注意：$\mathbf{h}_{t-1}$可以被传递到后续状态

$$\mathbf{h}_t = \mathrm{TanH}(\mathbf{x}_t\mathbf{U} + \mathbf{h}_{t-1}\mathbf{W})$$

循环神经网络的“记忆”

- 循环神经网络可以记忆任意长度的历史，因此可以非常适合处理不定长的序列，比如自然语言句子
  - 注意：$\mathbf{h}_{t-1}$可以被传递到后续状态

$$
\begin{aligned}
\mathbf{h}_t &= \mathrm{TanH}(\mathbf{x}_t\mathbf{U} + \mathbf{h}_{t-1}\mathbf{W}) \\
\mathbf{h}_{t+1} &= \mathrm{TanH}(\mathbf{x}_{t+1}\mathbf{U} + \mathbf{h}_t\mathbf{W}) \\
&= \mathrm{TanH}(\mathbf{x}_{t+1}\mathbf{U} + \mathrm{TanH}(\mathbf{x}_t\mathbf{U} + \mathbf{h}_{t-1}\mathbf{W})\mathbf{W})
\end{aligned}
$$

循环神经网络的“记忆”

- 循环神经网络可以记忆任意长度的历史，因此可以非常适合处理不定长的序列，比如自然语言句子
  - 注意：$\mathbf{h}_{t-1}$可以被传递到后续状态

$$
\begin{aligned}
\mathbf{h}_t &= \mathrm{TanH}(\mathbf{x}_t\mathbf{U} + \mathbf{h}_{t-1}\mathbf{W}) \\
\mathbf{h}_{t+1} &= \mathrm{TanH}(\mathbf{x}_{t+1}\mathbf{U} + \mathbf{h}_t\mathbf{W}) \\
&= \mathrm{TanH}(\mathbf{x}_{t+1}\mathbf{U} + \mathrm{TanH}(\mathbf{x}_t\mathbf{U} + \mathbf{h}_{t-1}\mathbf{W})\mathbf{W}) \\
\mathbf{h}_{t+2} &= \mathrm{TanH}(\mathbf{x}_{t+2}\mathbf{U} + \mathbf{h}_{t+1}\mathbf{W}) \\
&= \mathrm{TanH}(\mathbf{x}_{t+2}\mathbf{U} + \\
&\quad \mathrm{TanH}(\mathbf{x}_{t+1}\mathbf{U} + \mathrm{TanH}(\mathbf{x}_t\mathbf{U} + \mathbf{h}_{t-1}\mathbf{W})\mathbf{W})\mathbf{W})
\end{aligned}
$$

基于循环神经网络的语言模型(RNN LM)

- 循环神经网络可以被直接用于语言模型

基于循环神经网络的语言模型(RNN LM)

- 循环神经网络可以被直接用于语言模型
  - 与FNN LM类似，首先把词从one-hot表示转换成分布式表示

基于循环神经网络的语言模型(RNN LM)

- 循环神经网络可以被直接用于语言模型
  - 与FNN LM类似，首先把词从one-hot表示转换成分布式表示
  - t时刻预测$\mathrm{P}(x_{t+1}|x_1...x_t)$

基于循环神经网络的语言模型(RNN LM)

- 循环神经网络可以被直接用于语言模型
  - 与FNN LM类似，首先把词从one-hot表示转换成分布式表示
  - t时刻预测$\mathrm{P}(x_{t+1}|x_1 \ldots x_t)$
  - 可以叠加更多的层

进一步的问题

- **循环单元设计**：循环单元就是一个函数，入读当前时刻的输入和上一时刻的状态，生成当前时刻的状态

$$\mathbf{h}_t = g(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta)$$

很多种方式设计$g(\cdot)$，如著名的LSTM、GRU等

进一步的问题

- **循环单元设计**：循环单元就是一个函数，入读当前时刻的输入和上一时刻的状态，生成当前时刻的状态

$$\mathbf{h}_t = g(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta)$$

  很多种方式设计$g(\cdot)$，如著名的LSTM、GRU等
- **梯度消失/爆炸**：随着序列变长，在反向传播时循环神经网络会产生更多的局部梯度相乘计算，这会导致梯度消失/爆炸问题

$$\underbrace{0.2 \times 0.3 \times \ldots \times 0.2 \times 0.1}_{100\text{项}} \approx 0$$

  - 可以考虑梯度裁剪，限制梯度的大小
  - 也可以引入short-cut connection，如残差网络
- **训练**：有了自动微分，这不是个大问题 :)

自注意力机制(Self-Attention)

- RNN LM效果很好，但是当序列过长,词汇之间信息传递路径过长，容易出现梯度消失、梯度爆炸的问题。

自注意力机制(Self-Attention)

- RNN LM效果很好，但是当序列过长,词汇之间信息传递路径过长，容易出现梯度消失、梯度爆炸的问题。

- 能否将不同位置之间的词汇间信息传递的距离拉近为1？

自注意力机制(Self-Attention)

- RNN LM效果很好，但是当序列过长,词汇之间信息传递路径过长，容易出现梯度消失、梯度爆炸的问题。

- 能否将不同位置之间的词汇间信息传递的距离拉近为1？

- **自注意力机制(Self-Attention)**可以很好的解决长距离依赖问题，在长距离语言建模任务取得了很好的效果
  - 更充分的表示序列不同位置之间的复杂关系
  - 并行训练，提高效率

Attention Is All You Need
Vaswani et al., 2017, In Proc. of Neural Information Processing Systems, 6000-6010

Transformer语言模型(Vaswani et al., 2017)

- 一个简单的例子

Transformer语言模型(Vaswani et al., 2017)

- 一个简单的例子

词的分布式表示

前面已经介绍过！

基于One-hot表示获得

新加入位置向量PE

Transformer语言模型(Vaswani et al., 2017)

- 一个简单的例子

词的分布式表示

前面已经介绍过！

基于One-hot表示获得

新加入位置向量PE

自注意力机制

计算词汇之间的相关度

多头自注意力机制

后面将会介绍

Transformer语言模型(Vaswani et al., 2017)

- 一个简单的例子

词的分布式表示

前面已经介绍过！

基于One-hot表示获得

新加入位置向量PE

自注意力机制

计算词汇之间的相关度

多头自注意力机制

后面将会介绍

前馈神经网络和输出层

双层全连接网络

激活函数为Relu

最后通过Softmax输出

Transformer语言模型(Vaswani et al., 2017)

- 多头注意力机制

多头注意力

$$MultiHead(Q, K, V) = Concat(head_1, ...head_n)W^0$$

把输入压缩成多个维度较小的输出，分别做自注意力

再把结果级联，经过线性变换得到最终输出

Transformer语言模型(Vaswani et al., 2017)

- 多头注意力机制

多头注意力

$MultiHead(Q,K,V) = Concat(head_1, ...head_n)W^0$

把输入压缩成多个维度较小的输出，分别做自注意力

再把结果级联，经过线性变换得到最终输出

基于点乘的自注意力

$head_i = softmax(\frac{QK^T}{\sqrt{d_k}})V$

计算得到位置向量的加权和

Q,K,V都是相同的

语言模型评价

- 语言模型的评价指标 - 困惑度(Perplexity, PPL)
  - 语言模型预测一个语言样本的能力
  - 困惑度越低，建模的效果越好

$$\mathrm{PPL}(w_1 ... w_m) = \mathrm{P}(w_1 ... w_m)^{-1/m}$$

语言模型评价

- 语言模型的评价指标 - 困惑度(Perplexity, PPL)
  - 语言模型预测一个语言样本的能力
  - 困惑度越低，建模的效果越好

$$\mathrm{PPL}(w_1 ... w_m) = \mathrm{P}(w_1 ... w_m)^{-1/m}$$

- Penn Treebank(PTB)上的评价结果

模型	作者	年份	PPL
FNN LM	Bengio et al.	2003	162.2
RNN LM	Mikolov et al.	2010	124.7
RNN-LDA LM	Mikolov et al.	2012	92.0
RNN(LSTM) LM	Zaremba et al.	2014	78.4
RHN	Zilly et al.	2016	65.4
RNN(AWD-LSTM) LM	Merity et al.	2018	58.8
GPT-2 (Transformer)	Radford et al.	2019	35.7

单词的表示

- 如何表示一个单词？
  - **One-hot**: 假如有一个词典V，里面包含10k个单词，并进行编号。每个单词都可以表示为10k维的one-hot向量，仅在编号那个维度为1，其它为0

$$
\begin{array}{r c c}
 & \text{桌子} & \text{椅子} \\
\text{你}_1 & \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \cdots \\ 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \cdots \\ 0 \end{bmatrix} \\
\text{桌子}_2 & & \\
\text{他}_3 & & \\
\text{椅子}_4 & & \\
\text{我们}_5 & & \\
\cdots & & \\
\text{你好}_{10k} & &
\end{array}
$$

单词的one-hot表示

单词的表示

- 如何表示一个单词？
  - **One-hot**: 假如有一个词典V，里面包含10k个单词，并进行编号。每个单词都可以表示为10k维的one-hot向量，仅在编号那个维度为1，其它为0
  - **Distributed**: 类似于神经语言模型，每个单词可以被表示为一个实数向量，每一维都对应一种“属性” - 词嵌入

$$
\begin{array}{r} \\ 你_1 \\ 桌子_2 \\ 他_3 \\ 椅子_4 \\ 我们_5 \\ \dots \\ 你好_{10k} \end{array}
\begin{array}{c} 桌子 \\ \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} \end{array}
\begin{array}{c} 椅子 \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{bmatrix} \end{array}
$$

单词的one-hot表示

$$
\begin{array}{r} \\ 属性_1 \\ 属性_2 \\ 属性_3 \\ \dots \\ 属性_{512} \end{array}
\begin{array}{c} 桌子 \\ \begin{bmatrix} .1 \\ -1 \\ 2 \\ \dots \\ 0 \end{bmatrix} \end{array}
\begin{array}{c} 椅子 \\ \begin{bmatrix} 1 \\ 2 \\ .2 \\ \dots \\ -1 \end{bmatrix} \end{array}
$$

单词的分布式表示(词嵌入)

单词的表示

- 如何表示一个单词？
  - **One-hot**: 假如有一个词典V，里面包含10k个单词，并进行编号。每个单词都可以表示为10k维的one-hot向量，仅在编号那个维度为1，其它为0
  - **Distributed**: 类似于神经语言模型，每个单词可以被表示为一个实数向量，每一维都对应一种“属性” - 词嵌入

$\text{cosine}(\text{‘桌子’}, \text{‘椅子’}) = 0$

$$
\begin{array}{r} \\ \text{你}_1 \\ \text{桌子}_2 \\ \text{他}_3 \\ \text{椅子}_4 \\ \text{我们}_5 \\ \dots \\ \text{你好}_{10k} \end{array}
\begin{array}{c} \text{桌子} \\ \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} \end{array}
\begin{array}{c} \text{椅子} \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{bmatrix} \end{array}
$$

单词的one-hot表示

$\text{cosine}(\text{‘桌子’}, \text{‘椅子’}) = 0.5$

$$
\begin{array}{r} \\ \text{属性}_1 \\ \text{属性}_2 \\ \text{属性}_3 \\ \dots \\ \text{属性}_{512} \end{array}
\begin{array}{c} \text{桌子} \\ \begin{bmatrix} .1 \\ -1 \\ 2 \\ \dots \\ 0 \end{bmatrix} \end{array}
\begin{array}{c} \text{椅子} \\ \begin{bmatrix} 1 \\ 2 \\ .2 \\ \dots \\ -1 \end{bmatrix} \end{array}
$$

单词的分布式表示(词嵌入)

为什么需要分布式表示？

- 一个自然的问题：分布式表示中每一维都是什么意思
  - 可以把每一维都理解为一个属性，比如：性别、身高等
  - 但是，模型更多的是把一个维度看做是事物的一种“刻画”，是一种统计意义上的“语义”，而非人工归纳的属性

为什么需要分布式表示？

- 一个自然的问题：分布式表示中每一维都是什么意思
  - 可以把每一维都理解为一个属性，比如：性别、身高等
  - 但是，模型更多的是把一个维度看做是事物的一种“刻画”，是一种统计意义上的“语义”，而非人工归纳的属性
- 那这种方法有什么好处？
  - 更容易刻画词语之间的相似性
  - 连续空间表示模型可以更准确的刻画客观事物，而不是非零即一的判断
- 预测下一个词任务
  - 分布式表示很容易指导“桌子”和“椅子”是相似的
  - 即使“椅子”没在这个句型中出现过，系统仍然可以通过它和“桌子”的相似性进行预测

屋里 要 摆放 一个 -----	预测下个词
屋里 要 摆放 一个 桌子	见过
屋里 要 摆放 一个 椅子	没见过，但是仍然是合理预测

分布式表示的可视化

- 一个著名的例子：国王 → 王后

$$\overrightarrow{\text{国王}} - \overrightarrow{\text{男人}} + \overrightarrow{\text{女人}} = \overrightarrow{\text{王后}}$$

这里，$\overrightarrow{\text{word}}$表示单词的分布式向量表示

- 更多的词的可视化：相似的词聚在一起

神经语言模型中的词嵌入

- 在神经语言模型中，需要把词表示成它的分布式表示

w的分布式表示

e=(8,.2,-1,.9,...,1)

↑

$$e = w\ \mathbf{C}$$

↑

单词w

w=(0,0,1,0,...,0)

神经语言模型中的词嵌入

- 在神经语言模型中，需要把词表示成它的分布式表示
  - 其中**C**是词嵌入矩阵，每一行对应一个词的分布式表示

神经语言模型中的词嵌入

- 在神经语言模型中，需要把词表示成它的分布式表示
  - 其中**C**是词嵌入矩阵，每一行对应一个词的分布式表示
  - **C**可以用语言模型训练，也可以利用其它模型训练，固定词嵌入，让语言模型专注长片段的学习

神经语言模型中的词嵌入

- 在神经语言模型中，需要把词表示成它的分布式表示
  - 其中**C**是词嵌入矩阵，每一行对应一个词的分布式表示
  - **C**可以用语言模型训练，也可以利用其它模型训练，固定词嵌入，让语言模型专注长片段的学习

- 词嵌入如何学习得到？
  - 可以和语言模型的其它部分一起训练，不过速度较慢
  - 也可以考虑使用效率更高的外部模型，如word2vec、Glove等，这样可以使用更大规模的数据

不仅如“词”

- 词嵌入已经成为诸多NLP系统的标配，当然也衍生出各种花式玩法，甚至有“embed everything”的口号，但是词嵌入也有问题
  - 每个词都对应唯一的向量表示，但是对于一词多义现象，词义需要通过上下文进行区分。一个著名的例子：

Jobs was the CEO of apple.
He finally ate the apple.

不仅如“词”

- 词嵌入已经成为诸多NLP系统的标配，当然也衍生出各种花式玩法，甚至有“embed everything”的口号，但是词嵌入也有问题
  - 每个词都对应唯一的向量表示，但是对于一词多义现象，词义需要通过上下文进行区分。一个著名的例子：

    Jobs was the CEO of apple.
    He finally ate the apple.
- 引入上下文信息
  - 上述问题引发了新的思考：不能简单地考虑词的表示，应同时考虑其上下文信息
  - 对于句子中的一个词(或者位置)，同时表示词和上下文

表示更长的片段 - 上下文表示模型

- 在语言模型中已经包含了每个位置的上下文表示信息
  - 以RNN LM为例，位置i的隐层输出就是一种$w_1 ... w_i$的表示

表示更长的片段 - 上下文表示模型

- 在语言模型中已经包含了每个位置的上下文表示信息
  - 以RNN LM为例，位置i的隐层输出就是一种$w_1 ... w_i$的表示

表示更长的片段 - 上下文表示模型

- 在语言模型中已经包含了每个位置的上下文表示信息
  - 以RNN LM为例，位置i的隐层输出就是一种$w_1 ... w_i$的表示

更强的表示模型 - ELMO

- **ELMO**(Embedding from Language Models)可以说是掀起了基于语言模型的预训练的热潮
  - 仍然使用RNN结构，不过循环单元换成了LSTM
  - 同时考虑自左向右和自右向左的建模方式，同时表示一个词左端和右端的上下文
  - 融合所有层的输出，送给下游应用，提供了更丰富的信息

更强的表示模型 - GPT

- **GPT**(Generative Pre-Training)也是一种基于语言模型的表示模型
  - 架构换成了Transformer，特征抽取能力更强
  - 基于Pre-training + Fine-tuning的框架，预训练作为下游系统部件的参数初始值，因此可以更好的适应目标任务

更强的表示模型 - BERT

- **BERT**(Bidirectional Encoder Representations from Transformers)是最近非常火爆的表示模型
  - 仍然基于Transformer但是考虑了左右两端的上下文(可以对比GPT)
  - 使用了Mask方法来增加训练得到模型的健壮性，这个方法几乎成为了预训练表示模型的新范式

预训练

- 语言模型可以使用大量无标注数据进行训练，得到的模型可以被直接用于下游系统，以序列到序列任务为例

Decoder (目标任务正常训练)
Encoder (语言模型预先训练)

预训练

- 语言模型可以使用大量无标注数据进行训练，得到的模型可以被直接用于下游系统，以序列到序列任务为例

Decoder (目标任务正常训练)
Encoder (语言模型预先训练)

- 衍生出了非常火爆的范式：大规模语言模型pre-training + 目标任务fine-tuning
  - 许多NLP任务都可以被描述为语言建模，在外部训练得到的语言模型作为模块放入目标系统中(参数初始化)

(a) standard method　　(b) pre-training + fine-tuning

预训练带来的新思路

- 预训练模型刷榜各种任务的同时，引发了一些思考：预训练究竟给我们带来了什么？
  - 有标注数据量有限，预训练提供使用超大规模数据的方法
  - 从大规模无标注数据中学习通用知识，提升泛化能力
  - 神经网络复杂且不容易训练，预训练可以使模型关注优质解的高密度区域

预训练带来的新思路

- 预训练模型刷榜各种任务的同时，引发了一些思考：预训练究竟给我们带来了什么？
  - 有标注数据量有限，预训练提供使用超大规模数据的方法
  - 从大规模无标注数据中学习通用知识，提升泛化能力
  - 神经网络复杂且不容易训练，预训练可以使模型关注优质解的高密度区域

预训练带来的新思路

- 预训练模型刷榜各种任务的同时，引发了一些思考：预训练究竟给我们带来了什么？
  - 有标注数据量有限，预训练提供使用超大规模数据的方法
  - 从大规模无标注数据中学习通用知识，提升泛化能力
  - 神经网络复杂且不容易训练，预训练可以使模型关注优质解的高密度区域

预训练带来的新思路

- 预训练模型刷榜各种任务的同时，引发了一些思考：预训练究竟给我们带来了什么？
  - 有标注数据量有限，预训练提供使用超大规模数据的方法
  - 从大规模无标注数据中学习通用知识，提升泛化能力
  - 神经网络复杂且不容易训练，预训练可以使模型关注优质解的高密度区域

预训练带来的新思路

- 预训练模型刷榜各种任务的同时，引发了一些思考：预训练究竟给我们带来了什么？
  - 有标注数据量有限，预训练提供使用超大规模数据的方法
  - 从大规模无标注数据中学习通用知识，提升泛化能力
  - 神经网络复杂且不容易训练，预训练可以使模型关注优质解的高密度区域

- 机器翻译中的预训练
  - 机器翻译中预训练还没有屠榜，一方面由于很多机器翻译任务训练数据量并不小，另一方面也反应出翻译的双语建模对预训练也提出了新的要求

总结 - 长出一口气

- 讲了很多，累呀累，再整理一下主要观点
  - 神经网络没有那么复杂，入门并不难
  - 简单的网络结构可以组合成强大的模型
  - 语言模型可以用神经网络实现，效果很好，最近出现的预训练等范式证明了神经语言模型的潜力

总结 - 长出一口气

- 讲了很多，累呀累，再整理一下主要观点
  - 神经网络没有那么复杂，入门并不难
  - 简单的网络结构可以组合成强大的模型
  - 语言模型可以用神经网络实现，效果很好，最近出现的预训练等范式证明了神经语言模型的潜力
- 仍然有很多问题需要讨论
  - 常见的神经网络结构(面向NLP)
    google一下LSTM、GRU、CNN
  - 深层模型和训练方法。深度学习如何体现“深”？
    深层网络可以带来什么？
    如何有效的训练深层模型？
  - 如何把神经网络用于包括机器翻译在内的其它NLP任务？
    比如encoder-decoder框架
  - 深度学习的实践技巧
    “炼金术”了解下，因为不同任务调参和模型设计都有技巧
    ...

又结束一章内容

内容很多，开个了个头
学习深度学习技术需要实践和经验的积累！