深度学习中的循环神经网络

赵露君

2016年5月20日

摘要 让计算机具备智能是人们一直以来追求的目标，而近年来深度学习的发展让人们看到了希望。深度学习发源于机器学习中的神经网络，随着计算机计算能力的提升及数据量的增长，深度学习在计算机视觉与自然语言处理等领域取得了巨大的成功。而循环神经网络是其中一种适用于序列到序列学习的网络。本文首先回顾了深度学习的历史，接着简要介绍了基本的人工神经网络，然后开始讨论循环神经网络及其一种强有力的变种：长短期记忆网络，为了解决序列到序列中的长期依赖问题，我们讨论了记忆机制与注意力机制，最后介绍了一些应用。

关键字：人工智能，深度学习，神经网络，循环神经网络

1 简介

人工智能(Artificial Intelligence, AI)是学者们长期追求的目标。1950 年，Alan Turing 提出了**图灵测试**(Turing Test) [1]作为判断机器是否具有智能的标准。尽管有机器已经通过了图灵测试[1]，但我们离真正的智能还有一段距离。**机器学习**(Machine Learning)是一门人工智能的科学，该领域的主要研究对象是人工智能，特别是如何在经验学习中改善具体算法的性能[2]。**深度学习**(Deep Learning) [2, 3]发源于机器学习中的神经网络，在早期是指那些层数较多的神经网络，但随着近年来的快速发展，

[1]AI with 13-year-old boy's personality wins top prize at world's biggest Turing test, http://www.theverge.com/2012/6/27/3120135/eugene-goostman-ukrainian-boy-ai-turing-test

[2]来自维基百科：https://zh.wikipedia.org/wiki/%E6%9C%BA%E5%99%A8%E5%AD%A6%E4%B9%A0

其涵义有了进一步的扩展。由于计算机计算能力的提升及数据量的巨大增长，深度学习在一些领域取得了巨大的成功，**阿尔法围棋**(AlphaGo)[4]就是其中一个例子。

人工神经网络(Artificial Neural Network, ANN)，简称**神经网络**(Neural Network)，是指一种模仿大脑中神经系统的机器学习模型。每一层由一些**节点**(Node)构成，层与层之间的节点相互连接，从而构成了整个神经网络。神经网络最早的历史可以追溯到1958年 Rosenblatt 提出的**感知机**(Perceptron)[5]，感知机只有一个输入层和一个输出层，将输入值映射到0或1。由于感知机结构过于简单，甚至不能学会**异或**(XOR)函数，因而遭到了 Marvin Minsky 等人的质疑[6]。实际上，神经网络可以看成是**多层感知机**(Multilayer Perceptron)，但是当时一直找不到有效的学习算法，导致神经网络的研究陷入了停滞。直到1986年，由 David Rumelhart，Geoffrey Hinton 和 Ronald Williams 提出了**反向传播算法**(Backpropagation, BP)[7]，才解决了神经网络的学习问题，再一次引起了人们的注意。比如 Yann LeCun 等人用**卷积神经网络**(Convolutional Neural Network, CNN)将反向传播算法应用到了手写数字识别中[8]。由于当时的计算机计算能力不足，导致无法训练大规模的神经网络，而随着其他一些算法如**支持向量机**(Support Vector Machine)的兴起，神经网络一个新的寒冬开始来临。

循环神经网络(Recurrent Neural Network, RNN)是一种强有力的**序列模型**(Sequence Model)，可以将一个序列变为另一个序列。由于包含有自循环的隐藏层，循环神经网络因此得名。这个自循环的隐含层表征前一时间的输入也会对后面的输出产生影响，而在普通的神经网络中各个输入与输出之间是相互独立的。理论上，循环神经网络可以捕捉任意长时间的关联，但由于训练过程中的**梯度消失问题**(Vanishing Gradient Problem)[9–12]，很深的循环神经网络很难用反向传播算法训练，导致普通的循环神经网络难以捕捉到长期的依赖。**长短期记忆网络**(Long-Short Term Memory, LSTM)[13]是一种在 1997 年由 Hochreiter 和 Schmidhuber 提出的特殊的循环神经网络，被设计用来解决长期依赖的问题。循环神经网络在**自然语言处理**(Natural Language Processing, NLP)的各个任务上有着非常广泛的应用。

最近的深度学习浪潮开始于 2006 年，**加拿大高级研究院**(Canadian Institute for Advanced Research, CIFAR)的一些研究使深度神经网络再度进入人们的视野。由于**摩尔定律**(Moore's Law)，计算机的速度相对 90 年代有了数十倍的提升，同时具有大规模并行计算能力的**图形处理器**(Graphics Processing Unit, GPU)的出现，使得计算能力相对于双核 CPU 有了近 70 倍的提升[14]。Hinton 和他的两个

学生，Abdel-rahman Mohamed 和 George Dahl 利用 GPU 训练的**深度信念网络**(Deep Belief Network, DBN)首先在**语音识别**(Speech Recognition)上取得进展[15]。2011 年，Andrew Ng 和 Jeff Dean 开始了**谷歌大脑**(Google Brain)项目，利用谷歌庞大的计算资源来训练大规模深度神经网络。利用从 YouTube 上取来的 1000 万张图像，训练神经网络辨识猫[3]。深度学习的高潮在 2012 年到来。Alex Krizhevsky，Ilya Sutskever 与 Hinton 用卷积神经网络在**大规模视觉识别挑战赛**(Large Scale Visual Recognition Challenge, ILSVRC)上取得了高出第二名 10 个百分点的成绩[16]。在之后几年的 ILSVRC 中，深度学习一统天下。伴随着深度学习，越来越复杂的模型相继被提出。本文关注的**记忆机制**(Memory Mechanism)与**注意力机制**(Attention Mechanism)是为了解决循环神经网络中长期依赖的问题而提出的两种方法。

本文主要关注深度学习中的循环神经网络。我们将在文章的第二部分介绍基本的神经网络模型和算法，循环神经网络将会在第三部分介绍，接着会讨论被广泛使用的长短期记忆网络模型。为了解决更长时间的依赖问题，引入了记忆机制与注意力机制。最后介绍了一些应用。

2 神经网络

本节介绍的神经网络也可以称为**前馈神经网络**(Feedforward Neural Network)，以区别与后面要介绍的循环神经网络。在这种神经网络中，前一层只会向紧接着的后一层传播，而没有自循环的隐藏单元。

前馈神经网络(以下简称神经网络)一般由**输入层**(Input Layer)、**隐藏层**(Hidden Layer)、**输出层**(Output Layer)构成。图 1 就是一个只有三层的神经网络。图中每一个节点称为**神经元**(Neuron)，在输入层共有 8 个节点，代表输入的是一个 8 维的向量。在输出层共有 10 个节点，代表输出的是一个 10 维的向量，如在手写数字识别[4]中，10 个节点可分别代表 $0, 1, 2, \ldots, 9$ 这 10 个数字。中间的隐藏层共有 15 个节点，代表一个 15 维的向量。如果没有隐藏层而只有输入层和输出层，模型等价于 Softmax 回归(Softmax Regression)。

[3]可见于纽约时报对该事件的报道：http://www.nytimes.com/2012/06/26/technology/in-a-big-network-of-computers-evidence-of-machine-learning.html

[4]标准的手写数字识别数据集 MINST 可见：http://yann.lecun.com/exdb/mnist/

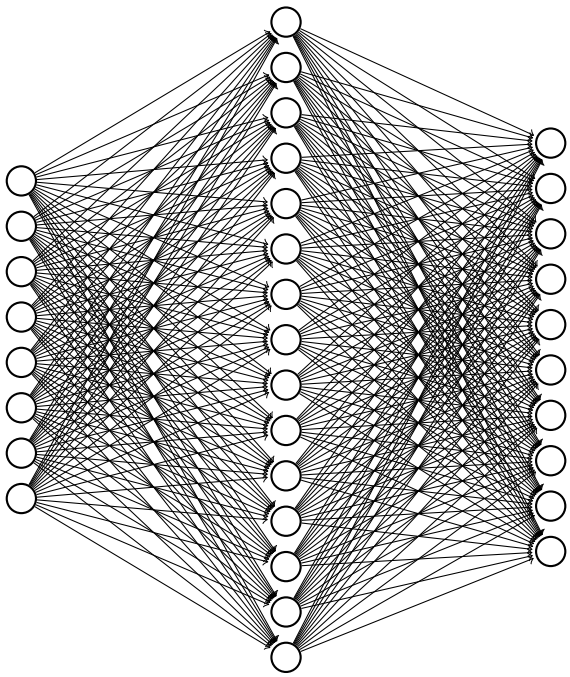

图 1：只有三层的前馈神经网络

一个非常值得一提的性质是，尽管只有三层的神经网络看似非常简单，但它的表现能力却非常强。只有三层的神经网络可以拟合任意函数，这称为神经网络的**普遍性**(Universality)。1989 年，George Cybenko 和 Kurt Hornik 等人分别独立证明了该结果[17, 18]，本文不会对此做深入讨论。另外，在 Nielsen 的书中提供了一个非常漂亮的可视化证明[19]。接下来简要介绍神经网络的模型和求解算法。

2.1 模型

首先，我们作一些约定。在本文中我们用小写字母表示标量，用粗体小写字母表示向量，用粗体大写字母表示矩阵，用字母加上符号“ˆ”表示预测值。

在图 1 给出的示例中只有一个隐藏层，而更一般的神经网络可以有多个隐藏层。我们用 $\boldsymbol{x}$ 表示输入向量，其中 $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, $\boldsymbol{x} \in \mathbb{R}^n$ 为 n 维向量，表示输入层有 n 个节点。用 L 表示神经网络的总层数，用 $\boldsymbol{a}^{(l)}$ 表示第 l 层向量。第一层即为输入层，故 $\boldsymbol{a}^{(1)} = \boldsymbol{x}$。用 $\hat{\boldsymbol{y}}$ 表示神经网络输出的预测值，其中 $\hat{\boldsymbol{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$, $\hat{\boldsymbol{y}} \in \mathbb{R}^m$ 为 m 维向量，表示输出层有 m 个节点，$\hat{\boldsymbol{y}} = \boldsymbol{a}^{(L)}$。

在神经网络的前馈计算中，后一层的向量由前一层的向量经一个线性变换加一个非线性的**激活函数**(Activation Function)得到，即：

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \cdot \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \quad (l = 2, \ldots, L) \tag{1}$$

$$\boldsymbol{a}^{(l)} = \boldsymbol{\sigma}(\boldsymbol{z}^{(l)}) \quad (l = 2, \ldots, L) \tag{2}$$

其中 σ 为 sigmoid 激活函数：

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

粗体的 $\boldsymbol{\sigma}$ 表示该函数是**元素级**(Element-wise)地应用于向量。例如：$\boldsymbol{\sigma}(\boldsymbol{x}) = (\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n))$。除了 sigmoid 函数外，另外两种常用的激活函数是**双曲正切**(Hyperbolic Tangent)函数 tanh：

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4}$$

与**整流线性单元**(Rectified Linear Unit)函数 ReLU：

$$\mathrm{ReLU}(x) = \max(0, x) \tag{5}$$

这三种函数的比较可见图 2。

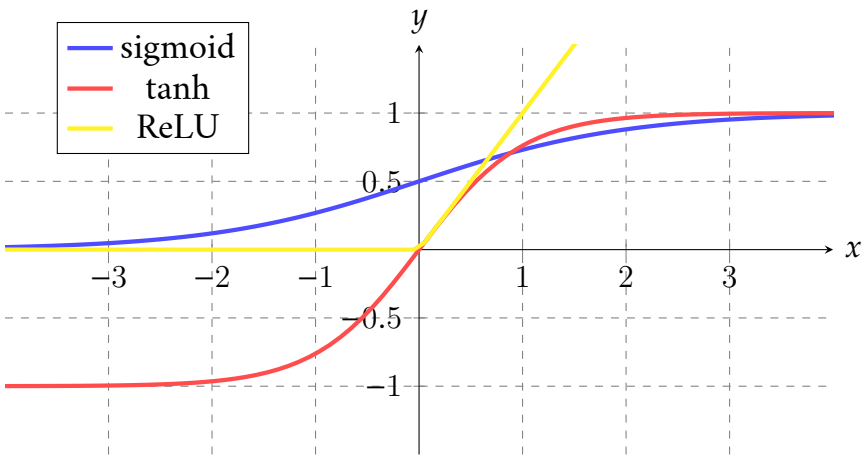

图 2: 几种激活函数

在每次前馈计算中，我们会引入两个参数：矩阵 $\boldsymbol{W}^{(l)}$ 和向量 $\boldsymbol{b}^{(l)}$。设第 l 层神经元个数为 $n^{(l)}$，则 $\boldsymbol{W}^{(l)} \in \mathbb{R}^{n^{(l)} \times n^{(l-1)}}$，$\boldsymbol{b}^{(l)} \in \mathbb{R}^{n^{(l-1)}}$。经过 $L-1$ 次前馈计算后，神经网络会输出预测值 $\hat{\boldsymbol{y}}$。接下来就是求解模型的参数使得预测值 $\hat{\boldsymbol{y}}$ 与真实值 $\boldsymbol{y}$ 尽可能接近。

2.2 优化

要求解模型参数，首先要定义**损失函数**(Loss Function)。在用神经网络做分类中，真实类别 $\boldsymbol{y}$ 一般用 One-hot 向量表示，即只有一个元素为 1，其余为 0 的向量。在这种情况下我们一般选用**交叉熵**(Cross Entropy)损失函数：

$$\mathcal{L}(\boldsymbol{W},\boldsymbol{b}) = \sum_{i=1}^{k} \mathcal{L}_i(\boldsymbol{W},\boldsymbol{b}) = \sum_{i=1}^{k} -\boldsymbol{y}_i^T \cdot \log(\hat{\boldsymbol{y}}_i) \tag{6}$$

这里 k 表示**数据集**(Data Set)大小，$\boldsymbol{y}_i$ 表示第 i 个数据的真实类别。

当使用交叉熵损失函数时，需要使模型的输出为一个**概率分布**(Probability Distribution)，故在最后一层我们将使用 Softmax 函数使之成为一个概率分布：

$$\begin{aligned} \boldsymbol{y} &= \mathbf{softmax}(\boldsymbol{x}) \\ y_i &= \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \end{aligned} \tag{7}$$

接下来即要最小化函数 $\mathcal{L}$，一般用**梯度下降法**(Gradient Descent)求解。即：

$$\boldsymbol{W}_t = \boldsymbol{W}_{t-1} - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \boldsymbol{W}_{t-1} - \alpha \sum_{i=1}^{k} \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{W}} \tag{8}$$

$$\boldsymbol{b}_t = \boldsymbol{b}_{t-1} - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = \boldsymbol{b}_{t-1} - \alpha \sum_{i=1}^{k} \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{b}} \tag{9}$$

这里，α 为**学习速率**(Learning Rate)。在实际应用中，由于数据集经常比较大，每次迭代需要在整个数据集上计算导数，导致计算代价太大，故一般采用**随机梯度下降**(Stochastic Gradient Descent, SGD)：

$$\boldsymbol{W}_t = \boldsymbol{W}_{t-1} - \alpha \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{W}} \tag{10}$$

$$\boldsymbol{b}_t = \boldsymbol{b}_{t-1} - \alpha \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{b}} \tag{11}$$

即每次只随机选一个数据计算导数。或是 Mini-batch **随机梯度下降**(Mini-batch Stochastic Gradient Descent)，即每次取一小部分计算导数。

近年来，也出现了很多随机梯度下降的改进算法，如 Adagrad[20]、Adadelta[21]、RMSProp[5]、Adam[22] 等。

[5]关于 RMSProp 没有正式的论文发表，首见于 Hinton 的网络公开课：http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

2.3 反向传播算法

接下来，就是要计算在随机梯度下降中出现的导数 $\frac{\partial \mathcal{L}_i}{\partial \boldsymbol{W}}$ 和 $\frac{\partial \mathcal{L}_i}{\partial \boldsymbol{b}}$。为了方便讨论，我们会去掉 $\mathcal{L}_i$ 中的下标 i，即变为：

$$\mathcal{L}(\boldsymbol{W}, \boldsymbol{b}) = -\boldsymbol{y}^T \cdot \log(\hat{\boldsymbol{y}}) \tag{12}$$

反向传播算法(以下简称 BP 算法)就是神经网络中计算导数 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}$ 和 $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}}$ 的算法。从数学上来看，BP 算法不过是函数求导中**链式法则**(Chain Rule)的应用。算法的意义在于给出了工程上易实现，且简单高效的算法。

首先我们引入一些记号：

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \cdot \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \quad (l = 2, \ldots, L) \tag{13}$$

$$\boldsymbol{\delta}^{(l)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(l)}} \quad (l = 2, \ldots, L) \tag{14}$$

这里，$\boldsymbol{\delta}^{(l)}$ 为**误差项**(Error)。

在 BP 算法中，首先会计算**输出误差**(Output Error) $\boldsymbol{\delta}^{(L)}$：

$$\begin{aligned} \boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(L)}} &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \cdot \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}^{(L)}} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}} \cdot \frac{\partial \boldsymbol{a}^{(L)}}{\partial \boldsymbol{z}^{(L)}} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}} \odot \boldsymbol{\sigma}'(\boldsymbol{z}^{(L)}) \quad (\text{需要 } \boldsymbol{a}^{(L)}, \boldsymbol{z}^{(L)}) \end{aligned} \tag{15}$$

这里，$\odot$ 为 Hadamard **乘积**(Hadamard Product)。

接着将**误差反向传播**(Backpropagate the Error)：

$$\boldsymbol{\delta}^{(l)} = ((\boldsymbol{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}) \odot \boldsymbol{\sigma}'(\boldsymbol{z}^{(l)}) \quad (\text{需要 } \boldsymbol{z}^{(l)}; l = L-1, L-2, \ldots, 2) \tag{16}$$

把每一层的误差计算出来后，就可以计算导数了：

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)} \quad (l = L, L-1, \ldots, 2) \tag{17}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \cdot (\boldsymbol{a}^{(l-1)})^T \quad (\text{需要 } \boldsymbol{a}^{(l-1)}; l = L, L-1, \ldots, 2) \tag{18}$$

在 BP 算法中，需要知道 $\boldsymbol{z}^{(l)}$ 和 $\boldsymbol{a}^{(l)}$ 的值，故每次反向传播的过程都首先需要一次前馈计算。BP 算法的完整描述参见算法 1。

算法 1 反向传播算法

1: **输入 x：** 把第一层向量 $a^{(1)}$ 设为 x。

2: **前馈计算：** 对 $l = 2, \ldots, L$，计算 $z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}$ 和 $a^{(l)} = \sigma(z^{(l)})$。

3: **输出误差：** 计算向量 $\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial a^{(L)}} \odot \sigma'(z^{(L)})$。

4: **误差反向传播：** 对 $l = L-1, L-2, \ldots, 2$，计算

$$\delta^{(l)} = ((W^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(z^{(l)})。$$

5: **输出：** 损失函数的梯度由 $\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}$ 和 $\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} \cdot (a^{(l-1)})^T$ 计算。

2.4 梯度消失问题

在神经网络中，一个使得神经网络层不可能很深的一个重要原因就是梯度消失问题。我们先来看一个简单的例子帮助更好的理解该问题。图 3 是一个最简单的深度神经网络，在该网络中所有层都只有一个神经元，共有三个隐藏层。注意现在的$w^{(l)}$和$b^{(l)}$均为标量。计算第二层的导数可知：

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \sigma'(z^{(2)}) \times w^{(3)} \times \sigma'(z^{(3)}) \times w^{(4)} \times \sigma'(z^{(4)}) \times w^{(5)} \times \sigma'(z^{(5)}) \times \frac{\partial \mathcal{L}}{\partial a^{(5)}} \tag{19}$$

由 sigmoid 函数的性质可知：$\sigma' \leq \frac{1}{4}$。通常，我们会用均值为 0，方差为 1 的 Gaussian **分布**(Gaussian Distribution)初始化$w^{(l)}$和$b^{(l)}$，会使用**正则化**(Regularization)的技巧来防止过拟合。这样，$w^{(l)}$ 和 $b^{(l)}$ 的值都不会太大。如果 $w^{(l)}$ 的值小于 1，由于在计算导数的过程中，每一层会比后一层多乘以 $\sigma'(z^{(l-1)}) \times w^{(l)} < \frac{1}{4}$，故导数会随着反向传播而指数级缩小。这样会导致用随机梯度下降来优化时，前面几层的参数学习的过程极慢。

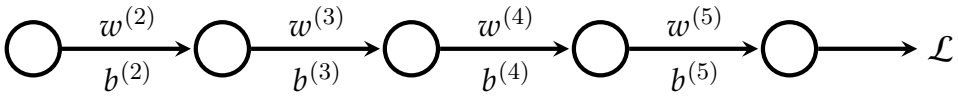

图 3: 最简单的深度神经网络

Xavier Glort 和 Yoshua Bengio 发现[23]，不同的激活函数对性能有很大的影响，sigmoid 函数是造成梯度消失的一个很重要的原因。同时几组团队发现[24–26]，十分简单的 ReLU 函数却是最好的。ReLU 的导数为 1 或 0 可以有效的避免梯

度消失问题。与此同时，可以用 GPU 来加速训练过程。在循环神经网络仍然存在梯度消失的问题，在下一节中还有关于该问题的进一步讨论。

3 循环神经网络

传统的神经网络的一个很大弊端是，当前的输出只与当前的输入有关。然而在实际中，事物之间常常具有某种时间的关联性。比如说，当前发生的事件会受到之前很长一段时间的影响。但我们读到一个词的时候，单单凭这一个词我们无法理解它的意思[6]，而是需要根据**上下文**(Context)来理解它的意思。

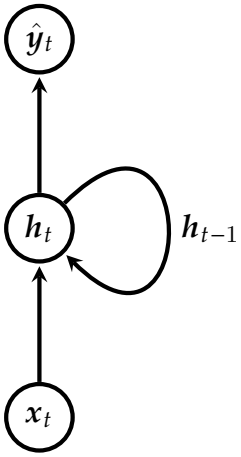

图 4: 循环神经网络

循环神经网络(以下简称 RNN，需要注意的是，应该与另一种 RNN(Recursive Neural Network, **递归神经网络**)相区别，本文提到的 RNN 都是指循环神经网络)则是处理该问题的一种强有力的模型。图 4 则是一个简单 RNN 结构图。在图 4 中，每一个节点表示一个向量，$\boldsymbol{x}_t$ 是输入，$\hat{\boldsymbol{y}}_t$ 是输出，$\boldsymbol{h}_t$ 是一个自循环的隐藏层单元，允许之前的信息得以保留，并对下一时刻的输出产生影响。

3.1 模型

如果我们把图 4 沿时间展开的话，就得到了图 5。我们用 $\mathbf{x}$ 表示输入序列，其中 $\mathbf{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$, $\boldsymbol{x}_t \in \mathbb{R}^n$ 是该序列在 t 时刻的一个 n 维向量，T 是该序列长度。

[6]**词的一词多义现象。**

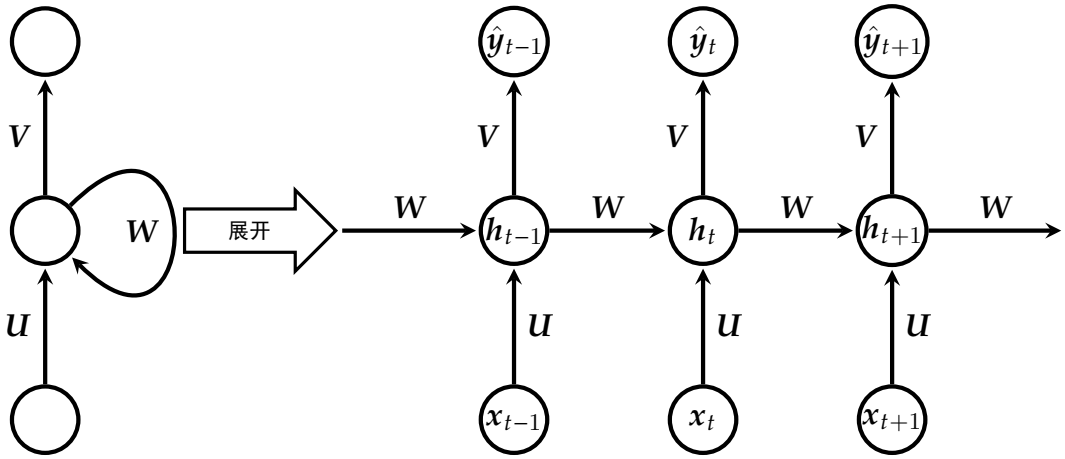

图 5: 展开的循环神经网络

用$\mathbf{h}$来表示隐藏层单元序列，其中$\mathbf{h} = (\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_T)$, $\boldsymbol{h}_t \in \mathbb{R}^k$，在$\boldsymbol{h}_t$中保存了前$t$时刻的信息。用$\hat{\mathbf{y}}$来表示 RNN 的输出序列，其中$\hat{\mathbf{y}} = (\hat{\boldsymbol{y}}_1, \hat{\boldsymbol{y}}_2, \ldots, \hat{\boldsymbol{y}}_T)$, $\hat{\boldsymbol{y}}_t \in \mathbb{R}^m$。

在 RNN 的前馈计算中，首先需要初始化一个隐藏层单元状态$\boldsymbol{h}_0$。之后的$\boldsymbol{h}_t$由前一时刻的$\boldsymbol{h}_{t-1}$和当前的输入$\boldsymbol{x}_t$计算得到：

$$\boldsymbol{h}_t = \mathbf{tanh}(\boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{W}\boldsymbol{h}_{t-1}) \quad (t = 1, 2, \ldots, T) \tag{20}$$

输出值$\hat{\boldsymbol{y}}_t$由$\boldsymbol{h}_t$计算得到：

$$\hat{\boldsymbol{y}}_t = \mathbf{softmax}(\boldsymbol{V}\boldsymbol{h}_t) \quad (t = 1, 2, \ldots, T) \tag{21}$$

这样，我们就得到了整个输出：$\hat{\mathbf{y}} = (\hat{\boldsymbol{y}}_1, \hat{\boldsymbol{y}}_2, \ldots, \hat{\boldsymbol{y}}_T)$。

需要注意的是，这里的激活函数 **tanh** 也可由其他的激活函数替换。在不同的时刻，$\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}$三个参数是共享的。

3.2 优化

与神经网络中类似，在 RNN 中也会使用交叉熵损失函数：

$$\mathcal{L}(\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}) = \sum_{i=1}^{d} \mathcal{L}_i(\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}) = \sum_{i=1}^{d} \sum_{t=1}^{T} -(\boldsymbol{y}_t^{(i)})^T \cdot \mathbf{log}(\hat{\boldsymbol{y}}_t^{(i)}) \tag{22}$$

这里d表示数据集大小，$\mathbf{y}^{(i)} = (\boldsymbol{y}_1^{(i)}, \boldsymbol{y}_2^{(i)}, \ldots, \boldsymbol{y}_T^{(i)})$表示第$i$个序列的$T$个真实类别。

在神经网络中用到的优化算法，都可以用在 RNN 中。

3.3 沿时间反向传播算法

与神经网络类似，在 RNN 中也需要计算导数 $\frac{\partial \mathcal{L}_i}{\partial \boldsymbol{U}}$, $\frac{\partial \mathcal{L}_i}{\partial \boldsymbol{W}}$ 和 $\frac{\partial \mathcal{L}_i}{\partial \boldsymbol{V}}$。RNN 中计算导数的算法叫**沿时间反向传播算法**(Backpropagation Through Time, BPTT) [27]。为了方便讨论，我们会去掉 $\mathcal{L}_i$ 中的下标 i，即变为：

$$\mathcal{L}(\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}) = \sum_{t=1}^{T} -(\boldsymbol{y}_t)^T \cdot \log(\hat{\boldsymbol{y}}_t) \tag{23}$$

首先，我们引入一些记号：

$$\mathcal{L}_t = -(\boldsymbol{y}_t)^T \cdot \log(\hat{\boldsymbol{y}}_t) \quad (t = 1, 2, \ldots, T) \tag{24}$$

$$\boldsymbol{s}_t = \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{W}\boldsymbol{h}_{t-1} \quad (t = 1, 2, \ldots, T) \tag{25}$$

$$\boldsymbol{z}_t = \boldsymbol{V}\boldsymbol{h}_t \quad (t = 1, 2, \ldots, T) \tag{26}$$

需要注意的是，本小节中的 $\mathcal{L}_t$ 跟上一小节中的 $\mathcal{L}_i$ 含义已经不一样了。

计算 $\boldsymbol{V}$ 的导数相对简单：

$$\begin{aligned} \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{V}} &= \frac{\partial \mathcal{L}_t}{\partial \hat{\boldsymbol{y}}_t} \cdot \frac{\partial \hat{\boldsymbol{y}}_t}{\partial \boldsymbol{V}} = \frac{\partial \mathcal{L}_t}{\partial \hat{\boldsymbol{y}}_t} \cdot \frac{\partial \hat{\boldsymbol{y}}_t}{\partial \boldsymbol{z}_t} \cdot \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{V}} \\ &= \left(\frac{\partial \mathcal{L}_t}{\partial \hat{\boldsymbol{y}}_t} \cdot \frac{\partial \hat{\boldsymbol{y}}_t}{\partial \boldsymbol{z}_t} \right) \cdot \boldsymbol{h}_t^T \quad (\text{需要 } \hat{\boldsymbol{y}}_t, \boldsymbol{h}_t; t = 1, 2, \ldots, T) \end{aligned} \tag{27}$$

接下来计算 $\boldsymbol{W}$ 的导数：

$$\begin{aligned} \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{W}} &= \frac{\partial \mathcal{L}_t}{\partial \hat{\boldsymbol{y}}_t} \cdot \frac{\partial \hat{\boldsymbol{y}}_t}{\partial \boldsymbol{z}_t} \cdot \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{h}_t} \cdot \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}} \\ &= \left(\frac{\partial \mathcal{L}_t}{\partial \hat{\boldsymbol{y}}_t} \cdot \frac{\partial \hat{\boldsymbol{y}}_t}{\partial \boldsymbol{z}_t} \right)^T \cdot \boldsymbol{V} \cdot \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}} \quad (\text{需要 } \hat{\boldsymbol{y}}_t; t = 1, 2, \ldots, T) \end{aligned} \tag{28}$$

会发现还需要计算 $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}}$：

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{s}_t} \cdot \left(\frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{W}} + \boldsymbol{W} \cdot \frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{W}} \right) \quad (\text{需要 } \boldsymbol{h}_t, \boldsymbol{h}_{t-1}; t = 2, 3, \ldots, T) \tag{29}$$

这是 $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}}$ 的一个递推式，故我们首先要计算 $\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}}$：

$$\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{s}_1} \cdot \frac{\partial \boldsymbol{s}_1}{\partial \boldsymbol{W}} \quad (\text{需要 } \boldsymbol{h}_1, \boldsymbol{h}_0) \tag{30}$$

计算 $\boldsymbol{U}$ 的导数的方式与 $\boldsymbol{W}$ 类似，不再赘述。

在 BPTT 算法中，需要知道 $\boldsymbol{h}_t, \hat{\boldsymbol{y}}_t$ 的值，故每次反向传播的过程都首先需要一次前馈计算。

3.4 扩展

有时候，我们不仅要考虑之前的输入对当前的影响，也要考虑之后的输入对当前的影响[7]。**双向循环神经网络**(Bidirectional Recurrent Neural Network, Bi-RNN) [28]就是为了处理这种情况而提出的一种对 RNN 的扩展。图 6 展示了 Bi-RNN 的结构。

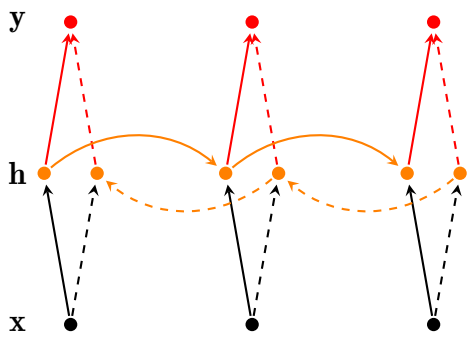

图 6: 双向循环神经网络

在 Bi-RNN 中，由前向计算和后向计算组成：

$$\overrightarrow{\boldsymbol{h}}_t = \tanh(\overrightarrow{\boldsymbol{U}}\boldsymbol{x}_t + \overrightarrow{\boldsymbol{W}}\overrightarrow{\boldsymbol{h}}_{t-1}) \quad (t = 1, 2, \ldots, T) \tag{31}$$

$$\overleftarrow{\boldsymbol{h}}_t = \tanh(\overleftarrow{\boldsymbol{U}}\boldsymbol{x}_t + \overleftarrow{\boldsymbol{W}}\overleftarrow{\boldsymbol{h}}_{t+1}) \quad (t = T, T-1, \ldots, 1) \tag{32}$$

输出值 $\hat{\boldsymbol{y}}_t$ 由 $\overrightarrow{\boldsymbol{h}}_t$ 和 $\overleftarrow{\boldsymbol{h}}_t$ 计算得到：

$$\hat{\boldsymbol{y}}_t = \mathbf{softmax}(\boldsymbol{V}[\overrightarrow{\boldsymbol{h}}_t; \overleftarrow{\boldsymbol{h}}_t]) \quad (t = 1, 2, \ldots, T) \tag{33}$$

除此之外，还可以增加隐藏层，使之成为**深度双向循环神经网络**(Deep Bidirectional Recurrent Neural Network, Deep Bi-RNN)。图 7 展示了 Deep Bi-RNN 的结构。

假设有 L 个隐藏层，第一个隐藏层：

$$\overrightarrow{\boldsymbol{h}}_t^{(1)} = \tanh(\overrightarrow{\boldsymbol{U}}^{(1)}\boldsymbol{x}_t + \overrightarrow{\boldsymbol{W}}^{(1)}\overrightarrow{\boldsymbol{h}}_{t-1}^{(1)}) \quad (t = 1, 2, \ldots, T) \tag{34}$$

$$\overleftarrow{\boldsymbol{h}}_t^{(1)} = \tanh(\overleftarrow{\boldsymbol{U}}^{(1)}\boldsymbol{x}_t + \overleftarrow{\boldsymbol{W}}^{(1)}\overleftarrow{\boldsymbol{h}}_{t+1}^{(1)}) \quad (t = T, T-1, \ldots, 1) \tag{35}$$

[7]所谓上下文就是既要考虑之前的，也要考虑之后的。

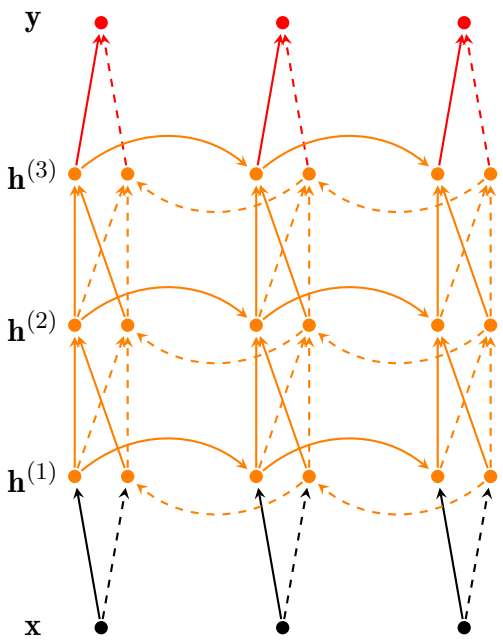

图 7: 深度双向循环神经网络

第 l 个隐藏层($l = 2, 3, \ldots, L$):

$$\overrightarrow{\boldsymbol{h}}_t^{(l)} = \tanh(\overrightarrow{\boldsymbol{U}}^{(l)}[\overrightarrow{\boldsymbol{h}}_t^{(l-1)}; \overleftarrow{\boldsymbol{h}}_t^{(l-1)}] + \overrightarrow{\boldsymbol{W}}^{(l)}\overrightarrow{\boldsymbol{h}}_{t-1}^{(l)}) \quad (t = 1, 2, \ldots, T) \tag{36}$$

$$\overleftarrow{\boldsymbol{h}}_t^{(l)} = \tanh(\overleftarrow{\boldsymbol{U}}^{(l)}[\overrightarrow{\boldsymbol{h}}_t^{(l-1)}; \overleftarrow{\boldsymbol{h}}_t^{(l-1)}] + \overleftarrow{\boldsymbol{W}}^{(l)}\overleftarrow{\boldsymbol{h}}_{t+1}^{(l)}) \quad (t = T, T-1, \ldots, 1) \tag{37}$$

输出层:

$$\hat{\boldsymbol{y}}_t = \mathrm{softmax}(\boldsymbol{V}[\overrightarrow{\boldsymbol{h}}_t^{(L)}; \overleftarrow{\boldsymbol{h}}_t^{(L)}]) \quad (t = 1, 2, \ldots, T) \tag{38}$$

3.5 梯度消失问题

在 RNN 中，仍然存在梯度消失问题[29]。为了便于分析，与之前稍有不同，我们假设隐藏层状态为:

$$\boldsymbol{h}_t = \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{W}\sigma(\boldsymbol{h}_{t-1}) + \boldsymbol{b} \quad (t = 1, 2, \ldots, T) \tag{39}$$

记 θ 代表所有参数 $\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{b}$，损失函数 $\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t(\boldsymbol{h}_t)$，则:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \theta} \tag{40}$$

$$\frac{\partial \mathcal{L}_t}{\partial \theta} = \sum_{k=1}^{t} \left(\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} \frac{\partial^+ \boldsymbol{h}_k}{\partial \theta} \right) \tag{41}$$

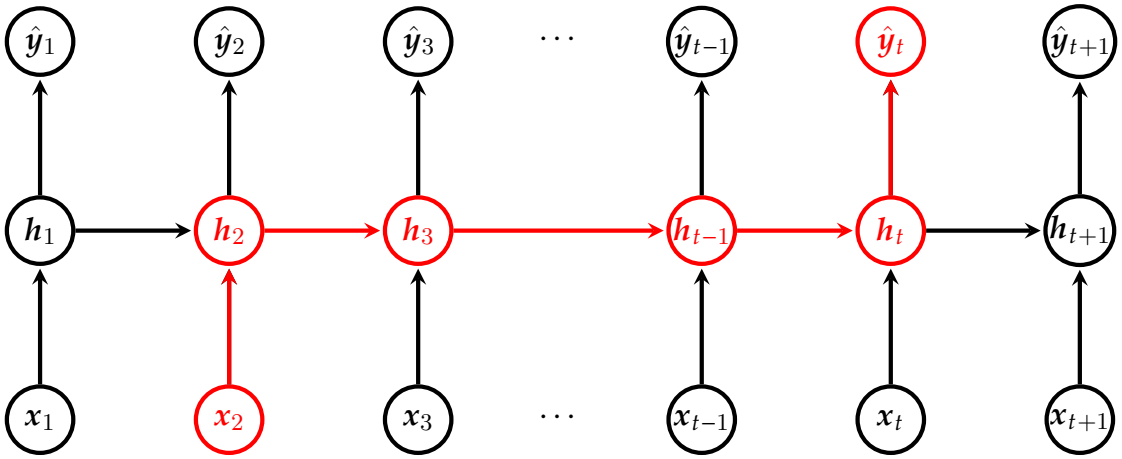

图 8：循环神经网络中的长期依赖

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}} = \prod_{i=k+1}^{t} \boldsymbol{W}^T \text{diag}(\sigma'(\boldsymbol{h}_{i-1})) \tag{42}$$

由于 $\sigma'(x) \leq \frac{1}{4}$，故 $\left\|\text{diag}(\sigma'(\boldsymbol{h}_k))\right\| \leq \gamma \in \mathbb{R}$。设 λ_1 是 $\boldsymbol{W}$ 的最大**奇异值**(Singular Value)，接下来证明：如果 $\lambda_1 < \frac{1}{\gamma}$，那么梯度消失问题就会出现。首先容易知道：

$$\forall k, \left\|\frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k}\right\| \leq \left\|\boldsymbol{W}^T\right\| \left\|\text{diag}(\sigma'(\boldsymbol{h}_k))\right\| < \frac{1}{\gamma}\gamma < 1 \tag{43}$$

设 $\eta \in \mathbb{R}$ 满足：$\forall k, \left\|\frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k}\right\| \leq \eta < 1$。那么：

$$\left\|\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}\left(\prod_{i=k}^{t-1} \frac{\partial \boldsymbol{h}_{i+1}}{\partial \boldsymbol{h}_i}\right)\right\| \leq \eta^{t-k}\left\|\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}\right\| \tag{44}$$

由于 $\eta < 1$，故梯度消失问题出现。

在神经网络中用来解决梯度消失问题的办法仍然适用于 RNN，在下文中我们将看到另一种解决该问题的方法。

4 长短期记忆网络

虽然在理论上，传统的 RNN 可以捕捉到长期的依赖，但由于上文中提到的梯度消失问题，使得当 T 很大时，RNN 的训练十分困难。在实际问题中，却又常常会需要这种长期依赖(图 8)，比如在**机器翻译**(Machine Translation, MT)，在一种语言中的前几个词翻译成另一种语言后却在最后面。

长短期记忆网络(下文简称 LSTM) 就是为了解决这个问题而提出的一种模型。LSTM 在 1997 年首次提出，随着深度学习的浪潮再度进入人们的视野，在自然语言处理的各项任务上有着广泛的应用，并且衍生了诸多**变体**(Variants)。实际中使用的 RNN 一般都是 LSTM。

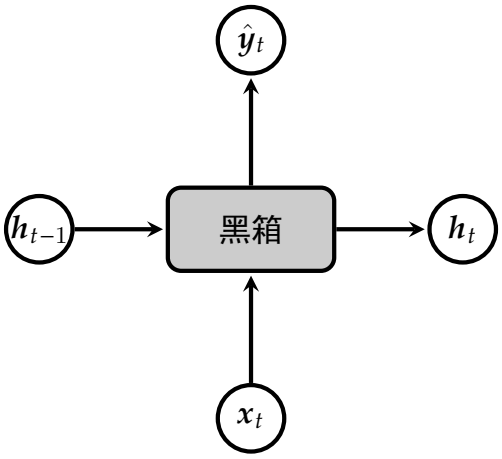

图 9: 循环神经网络“黑箱”模型

为了更好的理解 LSTM，我们先从另一个角度来审视 RNN。如图 9 所示，在每一时刻，给定当前输入 $\boldsymbol{x}_t$ 和上一时刻隐藏层状态 $\boldsymbol{h}_{t-1}$，经过一定计算，输出 $\hat{\boldsymbol{y}}_t$ 以及更新过的隐藏层状态 $\boldsymbol{h}_{t-1}$。可以把这一计算过程看成“黑箱”，而不去关心具体的计算过程，无非是进行一些线性或非线性变换。

LSTM 与传统 RNN 所不同的正是其中的计算过程。

4.1 模型

接下来我们来仔细看看，在 LSTM 里，“黑箱”里的具体计算过程究竟是怎样的。我们仍然用 $\mathbf{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$, $\boldsymbol{x}_t \in \mathbb{R}^n$ 表示输入序列，用 $\mathbf{h} = (\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_T)$, $\boldsymbol{h}_t \in \mathbb{R}^k$ 来表示隐藏层单元序列，用 $\hat{\mathbf{y}} = (\hat{\boldsymbol{y}}_1, \hat{\boldsymbol{y}}_2, \ldots, \hat{\boldsymbol{y}}_T)$, $\hat{\boldsymbol{y}}_t \in \mathbb{R}^m$ 来表示输出序列。在 LSTM 里，还包含新增的一组**记忆单元**(Memory Unit)序列 $\mathbf{c} = (\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_T)$, $\boldsymbol{c}_t \in \mathbb{R}^k$，单独用来存储信息。LSTM 创造性地引入了**门机制**(Gating Mechanism)，通过三个门来分别控制当前的信息是否要进入记忆单元，记忆单元中的哪些信息需要忘记掉，以及在输出时取出有用的信息。

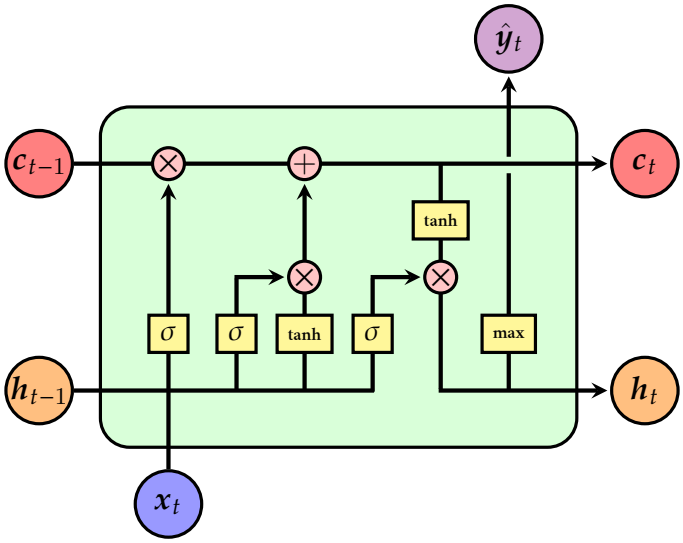

图 10：长短期记忆网络

图 10 展示了 LSTM 里“黑箱”的具体计算过程[8]，相比传统的 RNN 要复杂不少，我们一步步来看。

首先是**遗忘门**(Forget Gate)。在 t 时刻，遗忘门用来把之前储存在 $\boldsymbol{c}_{t-1}$ 里，但之后再也用不着的信息“遗忘”掉：

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \tag{45}$$

接着是**输入门**(Input Gate)。输入门用来控制把当前的哪些信息储存到记忆单元中：

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \tag{46}$$

当然我们还需要当前的输入产生信息：

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \tag{47}$$

接下来我们就可以通过遗忘门和输入门的控制来更新记忆单元：

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \tag{48}$$

记忆更新完成后，通过**输出门**(Output Gate)把对当前判断有用的信息取出来，更新隐藏层状态：

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \tag{49}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) \tag{50}$$

[8]为了保持美观，softmax 在图中被表示成了 max。

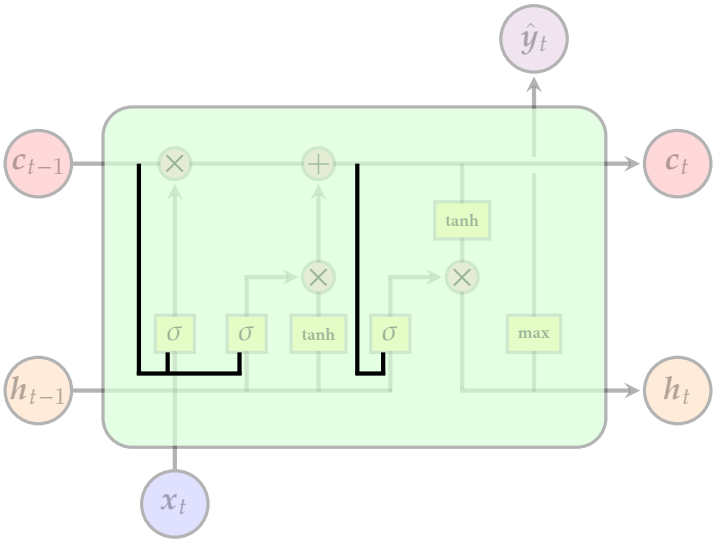

图 11: 长短期记忆网络变体

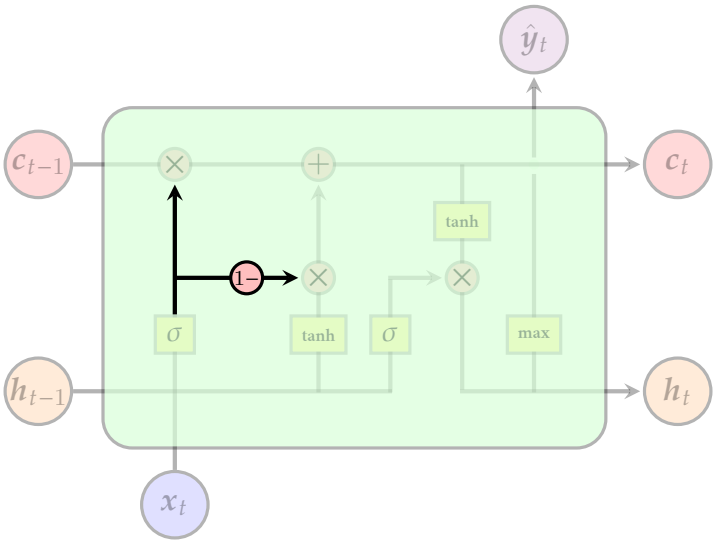

图 12: 长短期记忆网络变体2

然后就可以输出了：

$$\hat{\boldsymbol{y}}_t = \mathbf{softmax}(\boldsymbol{W}_{hy}\boldsymbol{h}_t) \tag{51}$$

这样，我们就完成了“黑箱”里的整个计算过程。

4.2 变体

LSTM 衍生了诸多变体，这里介绍三种。

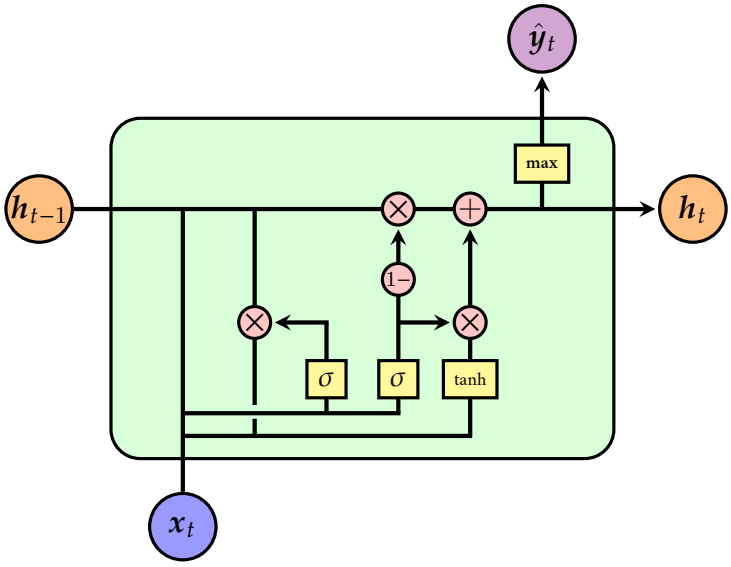

图 13: 门循环单元

5 应用

参考文献

[1] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433--460, 1950.

[2] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436--444, 2015.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484--489, 2016.

[5] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[6] Marvin Minsky and Seymour Papert. Perceptrons. 1969.

[7] David E Rumelhart, Geoffrey E Hintont, and Ronald J Williams. Learning representations by back-propagating errors. *NATURE*, 323:9, 1986.

[8] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541--551, 1989.

[9] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Master's thesis, Institut fur Informatik, Technische Universitat, Munchen*, 1991.

[10] Yoshua Bengio, Paolo Frasconi, and Patrice Simard. The problem of learning long-term dependencies in recurrent networks. In *Neural Networks, 1993., IEEE International Conference on*, pages 1183--1188. IEEE, 1993.

[11] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157--166, 1994.

[12] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735--1780, 1997.

[14] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873--880. ACM, 2009.

[15] Abdel-rahman Mohamed, George Dahl, and Geoffrey Hinton. Deep belief networks for phone recognition. In *Nips workshop on deep learning for speech recognition and related applications*, volume 1, page 39, 2009.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097--1105, 2012.

[17] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303--314, 1989.

[18] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359--366, 1989.

[19] Michael A. Nielsen. Neural networks and deep learning. Determination Press, 2015.

[20] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121--2159, 2011.

[21] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[22] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249--256, 2010.

[24] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807--814, 2010.

[25] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315--323, 2011.

[26] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, page 1, 2013.

[27] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550--1560, 1990.

[28] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45(11):2673--2681, 1997.

[29] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2012.