Deep Learning and Temporal Data Processing

3 - Recurrent Neural Networks

Andrea Palazzi

December 20, 2017

University of Modena and Reggio Emilia

Agenda

Introduction

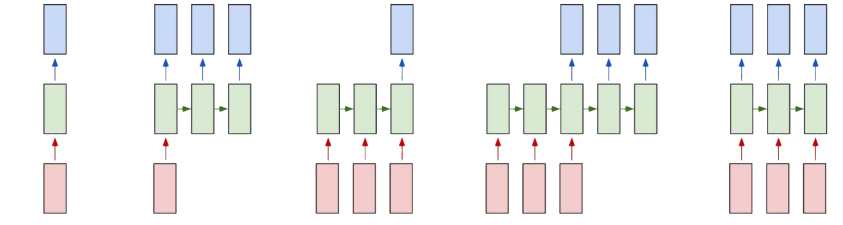
In **feedforward neural network** computation flows directly from input $\boldsymbol{x}$ through intermediate layers $\boldsymbol{h}$ to output $\boldsymbol{y}$.

Conversely, some networks topology feature feedback connections, in other words model outputs are fed back into the model itself.

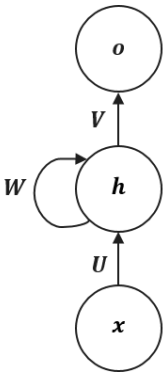
The term **recurrent neural networks** defines this family of models.

Recurrent neural networks (RNN) are **specialized for processing sequences**.
Similarly, we saw that convolutional neural networks feature specialized architecture for processing images.

RNNs boast a **much wider API with respect to feedforward neural networks**.
Indeed, these models can deal with *sequences* in the input, in the output or even both.



Vanilla RNN



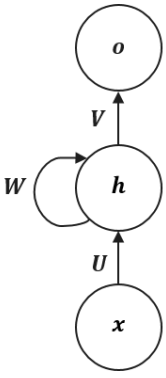
The vanilla RNN is provided with three sets of parameters:

- $\boldsymbol{U}$ maps inputs to the hidden state
- $\boldsymbol{W}$ parametrizes hidden state transition
- $\boldsymbol{V}$ maps hidden state to output

System dynamics is as simple as:

$$\begin{cases} \boldsymbol{h}^{(t)} = \phi(\boldsymbol{W}\,\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\,\boldsymbol{x}^{(t)}) \\ \boldsymbol{o}^{(t)} = \boldsymbol{V}\,\boldsymbol{h}^{(t)} \end{cases} \tag{1}$$

Intuition about Hidden State



The hidden state $\boldsymbol{h}^{(t)}$ can be intuitively viewed as a *lossy* summary of the sequence of past inputs fed to the network, in which are stored the main task-relevant aspects of the past sequence of inputs up to time t.

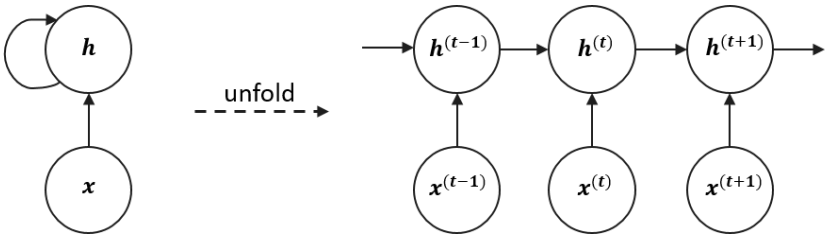
Since the an input sequence of arbitrary length $(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(t)})$ is mapped into a fixed size vector $\boldsymbol{h}^{(t)}$, this summary is necessarily lossy.

Training a RNN

Unfolding the Computational Graph

A recurrent computational graph can be unfolded into a sequential computational graph with a repetitive structure.

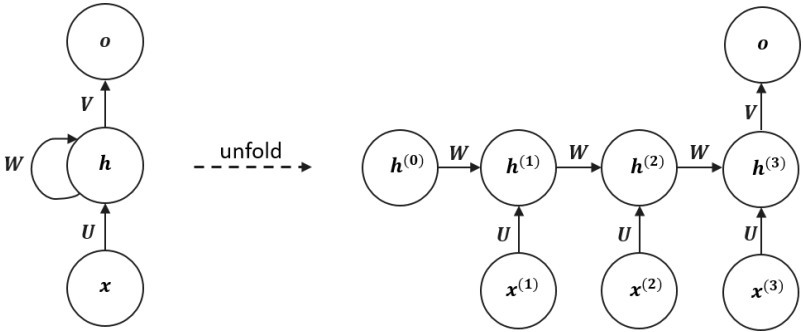
$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{t-1}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$



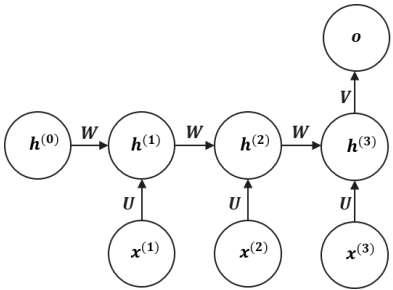
The capacity to **unfold a recurrent graph into a DAG** (Directed Acyclic Graph) allows to train recurrent neural network by means of standard backpropagation.

Because the gradient conceptually flows backward though time instead of though layers, this algorithm is usually referred to as **backpropagation through time (BPTT)**.

Let's make an example for an simple architecture which processes sequences of length $\tau = 3$ and produces an output at the end of the sequence.



Now, given a differentiable loss on final output $L(\boldsymbol{y}, \boldsymbol{o})$ let's compute the derivative of the objective L with respect to the weights $\boldsymbol{V}$, $\boldsymbol{W}$ and $\boldsymbol{U}$ and see what happens.



- $\frac{\partial L}{\partial \boldsymbol{V}} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{V}}$
- $\frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^{(3)}} \sum_{k=0}^{3} \left(\frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}} \right)$
- $\frac{\partial L}{\partial \boldsymbol{U}} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^{(3)}} \sum_{k=0}^{3} \left(\frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{U}} \right)$

Consideration: while $\frac{\partial L}{\partial \boldsymbol{V}}$ depends only on current state, $\frac{\partial L}{\partial \boldsymbol{W}}$ and $\frac{\partial L}{\partial \boldsymbol{U}}$ depend on all previous sequence states.

Looking closer, we see that terms $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ must be themselves computed through the chain rule. For example, we can obtain $\frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}}$.

Turns out [3] that when using *tanh* and *sigmoid* activations, the 2-norm of these Jacobian matrices is upper bounded by 1 and $1/4$ respectively. Thus, we can easily end up multiplying smaller and smaller numbers until the gradients become zero.

This problem is known as **vanishing gradient problem**, and causes serious trouble when trying to learn long-term dependencies in the input sequences, because contributions of "far-away" steps become zero.

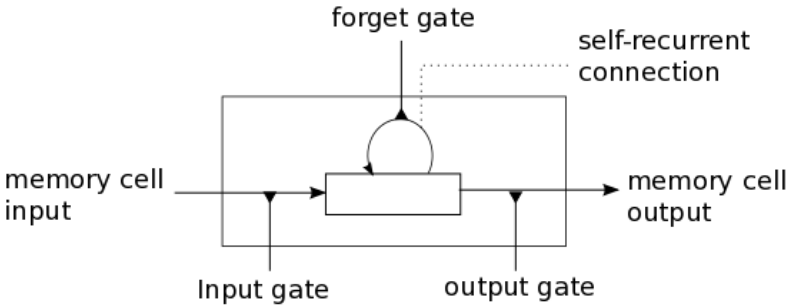
Depending on the network parameters and choice of activation functions, the opposite problem can arise. This is called **exploding gradient problem** and happens when gradients become bigger and bigger until numerical problems destroy the optimization process.

Both issues can be mitigated through proper weight initialization, accurate choice of activation functions and gradient clipping.

Note: these problems can also happen in deep feedforward networks: however, they are more common in recurrent architectures because these models are usually very deep (actually as deep as the length of the input sequence).

Advanced RNN Architectures

Long Short-Term Memory (LSTM) and **Gated Recurrent Unit (GRU)** are more complex recurrent architectures that have been proposed [2, 1] to overcome the issues in the gradient flow and to ease the learning of long-term dependencies thanks to the introduction of **learnable gating mechanisms**.



Let's see **how update equations look like for a LSTM model**. Please notice that LSTM framework, notation is usually slightly different than form vanilla RNN.

$$\begin{cases} \boldsymbol{i} = \sigma(\boldsymbol{x}^{(t)}\boldsymbol{U}_i + \boldsymbol{s}^{(t-1)}\boldsymbol{W}_i) \\ \boldsymbol{f} = \sigma(\boldsymbol{x}^{(t)}\boldsymbol{U}_f + \boldsymbol{s}^{(t-1)}\boldsymbol{W}_f) \\ \boldsymbol{o} = \sigma(\boldsymbol{x}^{(t)}\boldsymbol{U}_o + \boldsymbol{s}^{(t-1)}\boldsymbol{W}_o) \\ \boldsymbol{g} = \tanh(\boldsymbol{x}^{(t)}\boldsymbol{U}_g + \boldsymbol{s}^{(t-1)}\boldsymbol{W}_g) \\ \boldsymbol{c}^{(t)} = \boldsymbol{c}^{(t-1)} \odot \boldsymbol{f} + \boldsymbol{g} \odot \boldsymbol{i} \\ \boldsymbol{s}^{(t)} = \tanh(\boldsymbol{c}^{(t)}) \odot \boldsymbol{o} \end{cases} \tag{2}$$

Here $\odot$ denotes element-wise multiplication. If this sounds complicated, in the following slides we'll parse each of these equations to make a sense out of it.

$$\begin{cases} \boldsymbol{i} = \sigma(\boldsymbol{x}^{(t)}\boldsymbol{U}_i + \boldsymbol{s}^{(t-1)}\boldsymbol{W}_i) \\ \boldsymbol{f} = \sigma(\boldsymbol{x}^{(t)}\boldsymbol{U}_f + \boldsymbol{s}^{(t-1)}\boldsymbol{W}_f) \\ \boldsymbol{o} = \sigma(\boldsymbol{x}^{(t)}\boldsymbol{U}_o + \boldsymbol{s}^{(t-1)}\boldsymbol{W}_o) \\ \dots \end{cases} \tag{3}$$

These are the input, forget and output **gates** respectively. Each gate has the same dimension of the hidden state. Gates are multiplied element-wise with other functions of LSTM thus acting as **continuous, differentiable switches** thanks to the sigmoid activation function.

Input and *output gates* controls how much of the input let through and how much of the internal state exposing to the external, respectively. Conversely, *forget gate* controls how much memory form previous time step must be overwritten.

$$\begin{cases} \dots \\ \boldsymbol{g} = \tanh(\boldsymbol{x}^{(t)}\boldsymbol{U}_g + \boldsymbol{s}^{(t-1)}\boldsymbol{W}_g) \\ \dots \end{cases} \tag{4}$$

This equation computes what could intuitively be described a **candidate state**. Indeed, the equation is pretty much the same that we saw for vanilla RNN architecture. Nonetheless, the amount of influence of $\boldsymbol{g}$ on the LSTM memory cell is controlled by input gate $\boldsymbol{i}$.

$$\begin{cases} \dots \\ \boldsymbol{c}^{(t)} = \boldsymbol{c}^{(t-1)} \odot \boldsymbol{f} + \boldsymbol{g} \odot \boldsymbol{i} \\ \boldsymbol{s}^{(t)} = \tanh(\boldsymbol{c}^{(t)}) \odot \boldsymbol{o} \end{cases} \tag{5}$$

First equation computes the **update for memory cell** $\boldsymbol{c}$. The *forget gate* controls how much of memory from previous steps $\boldsymbol{c}^{(t)}$ must be kept. *Input gate* supervises the amount of newly computed state $\boldsymbol{g}$ that has to flow into the memory.

Eventually, last equation compute the **output hidden state** from the current memory. *Output gate* regulates the amount of information to be exposed to successive layers.

Credits

These slides heavily borrow from a number of awesome sources. I'm really grateful to all the people who take the time to share their knowledge on this subject with others.

In particular:

- Stanford CS231n Convolutional Neural Networks for Visual Recognition
  http://cs231n.stanford.edu/
- Stanford CS20SI TensorFlow for Deep Learning Research
  http://web.stanford.edu/class/cs20si/syllabus.html
- Deep Learning Book (GoodFellow, Bengio, Courville)
  http://www.deeplearningbook.org/

Credits ii

- Marc'Aurelio Ranzato, "Large-Scale Visual Recognition with Deep Learning"
  www.cs.toronto.edu/~ranzato/publications/ranzato_cvpr13.pdf
- Convolution arithmetic animations
  https://github.com/vdumoulin/conv_arithmetic
- Andrej Karphathy personal blog
  http://karpathy.github.io/
- WildML blog on AI, DL and NLP
  http://www.wildml.com/
- Michael Nielsen Deep Learning online book
  http://neuralnetworksanddeeplearning.com/

[1] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio.
Learning phrase representations using rnn encoder-decoder for statistical machine translation.
arXiv preprint arXiv:1406.1078, 2014.

[2] S. Hochreiter and J. Schmidhuber.
Long short-term memory.
Neural computation, 9(8):1735–1780, 1997.

[3] R. Pascanu, T. Mikolov, and Y. Bengio.
On the difficulty of training recurrent neural networks.
In *International Conference on Machine Learning*, pages 1310–1318, 2013.