

Deep Learning and Temporal Data Processing

3 - Recurrent Neural Networks

Andrea Palazzi

June 21th, 2017

University of Modena and Reggio Emilia

Introduction

Vanilla RNN

Training a RNN

Credits

Introduction

test [1]

In **feedforward neural network** computation flows directly from input x through intermediate layers h to output y .

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

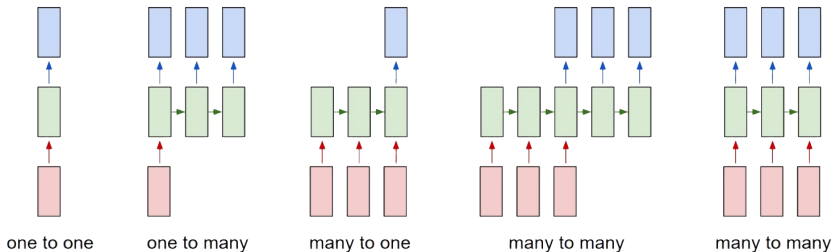
The term **recurrent neural networks** defines this family of models.

Recurrent neural networks (RNN) are **specialized for processing sequences**.

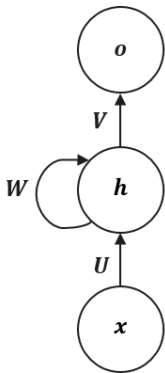
Similarly, we saw that convolutional neural networks feature specialized architecture for processing images.

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

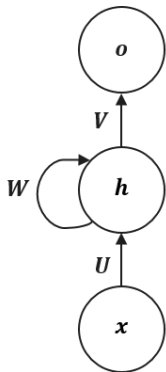


The vanilla RNN is provided with three sets of parameters:

- U maps inputs to the hidden state
- W parametrizes hidden state transition
- V maps hidden state to output

System dynamics is as simple as:

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



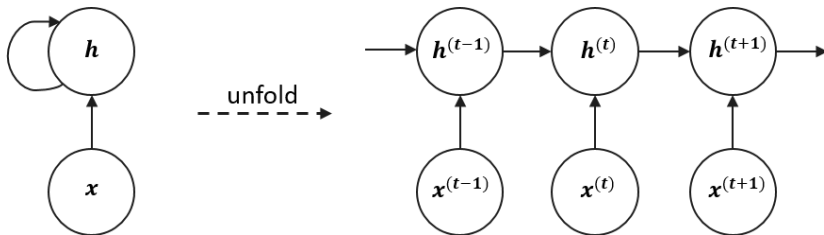
The hidden state $\mathbf{h}^{(t)}$ can be intuitively viewed as a *lossy* summary of the sequence of past inputs fed to the network, in which are stored the main task-relevant aspects of the past sequence of inputs up to time t .

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

Training a RNN

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

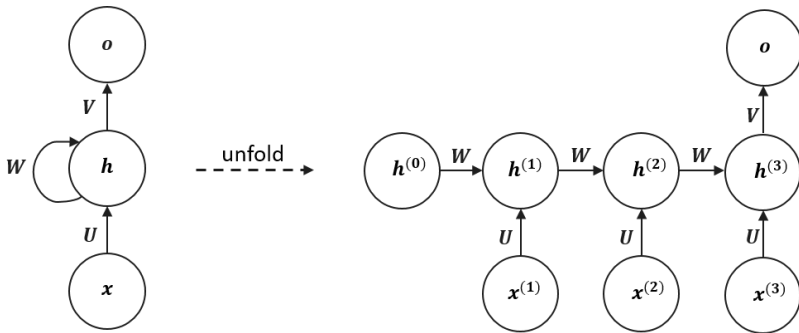
$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta)$$



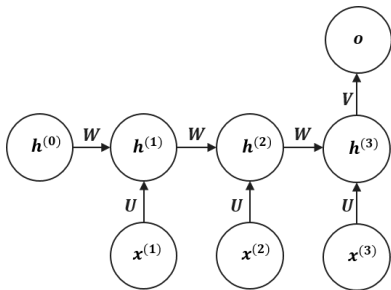
The capacity to **unfold a recurrent graph into a DAG** (Directed Acyclic Graph) allows to train recurrent neural network by means of standard backpropagation.

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 $L(\mathbf{y}, \mathbf{o})$ let's compute the derivative of the objective L with respect to the weights \mathbf{V} , \mathbf{W} and \mathbf{U} and see what happens.



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

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

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

Turns out [2] 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).

Vanishing and exploding gradient problem.

Credits

These slides heavily borrow from the following Stanford course:

- <http://cs231n.stanford.edu/>

if you want to deepen your knowledge of these concepts, I'd really suggest you to start from here!

Also, nice convolution animations are taken from here:

- https://github.com/vdumoulin/conv_arithmetic

- [1] G. Cybenko.

Approximation by superpositions of a sigmoidal function.

Mathematics of Control, Signals, and Systems (MCSS), 2(4):303–314, 1989.

- [2] 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.