RNN and LSTM

Matteo Lugli

January 3, 2024

1 Recurrent Neural Networks

At each time step, to compute the next *hidden state* h_t, the neural network considers bot h_{t-1} and x_t. If needed, it computes the output y_t and computes the loss. So we end up with a loss function value for each one of the timesteps in our *sequence*, that has a certain *sequence lenght*. In the below figure, since we unrolled the recursive loop 3 times, we can say that the sequence lenght is 3. This means that the network will take in input batches of 3 elements from our dataset: for example, it could take sequences of temperatures measured in 3 consecutive days (1 temperature for each day).

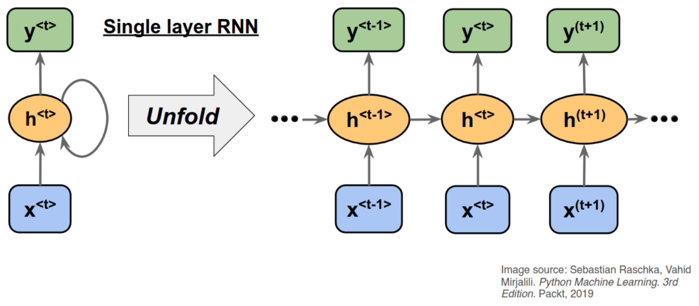

Figure 1: RNN architecture

Equations 1 and 2 describe how to compute the next hidden state and the output. Consider that we use different weights matrices applied on the hidden state and on the input, but in some papers you'll find them collapsed in one single matrix W.

$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h) \tag{1}$$

$$y_t = \sigma(W_{yh}h_t + b_y) \tag{2}$$

As we said before, at each timestep we compute the loss, so we end up with L_{t-1},L_t, L_{t+1}. To compute the overall loss, we sum all of the "partial losses" together:

$$L = \sum_{t=1}^{T} L_t \tag{3}$$

Kind of easy up to here. The tricky part is how to train such networks: turns out that we need to go "back in time" to perform the backpropagation correctly. That's why the variant of the standard algorithm used is called *backpropagation through time*.

2 Backpropagation through time

Let's write the formula for the derivative of the loss with respect to the weights W_{hh}. I'll use the "top" notation for time.

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial h^{(t)}} \cdot \frac{\partial h^{(t)}}{\partial W_{hh}} \tag{4}$$

Take a closer look at the third term of the above equation. It describes how the hidden state at time t varies on a small change of the weight matrix W_{hh}. If we think about it, when changing a little the weight matrix we also affect the computation of the hidden states before h_t, which we still have to consider when computing the gradient. This happens because we use the same weight matrix W_{hh} for all of the unrolls! Let's re-write the formula considering all of the previous terms:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial W_{hh}} \tag{5}$$

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \sum_{k=1}^{t} \frac{\partial L^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{+(k)}}{\partial W_{hh}} \tag{6}$$

Where $\frac{\partial h^{+(k)}}{\partial W_{hh}}$ indicates the immediate partial derivative of the state $h^{(k)}$ with respect to the weights (where h^{k-1} is taken has a constant). Now we can bring out $\frac{\partial L^{(t)}}{\partial h^{(t)}}$ from the summation, because it does not depend on k.

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \frac{\partial L^{(t)}}{\partial h^{(t)}} \sum_{k=1}^{t} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{+(k)}}{\partial W_{HRH}} \tag{7}$$

$$= \frac{\partial L^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \sum_{k=1}^{t} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{+(k)}}{\partial W_{hh}} \tag{8}$$

So our final formulation is the following:

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial h^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial h^{(t)}}{\partial h^{(k)}} \cdot \frac{\partial h^{(k)}}{\partial W_{hh}} \right) \tag{9}$$

Let's analyze what we wrote once again. There is one term that turns out to be suspicious: $\frac{\partial h^{(t)}}{\partial h^{(k)}}$. This can actually be computed with the chain rule once again, and it looks something like this:

$$\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial h^{(i)}}{\partial h^{(i-1)}} = \prod_{i=k+1}^{t} \sigma'(W_{hh}h^{(i-1)} + W_{hx}x^{(i)} + b_h)W_{hh} \tag{10}$$

This means that if we have very long sequences of elements, we end up multiplying the weight matrix many times to make just one backpropagation flow. This is what usually causes **exploding gradient** (this one can be simply solved by using gradient clipping) and **vanishing gradient** (needs a different type of architecture) problems: if W contains really small elements, the gradient vanishes to basically zero, on the other hand it might explode with big numbers. It has been proved that if the **spectral radious** of W is > 1, then the gradient explodes, if it is < 1, then it vanishes. This makes a vanilla neural network basically impossibile to train, because gradient cannot reach the oldest timesteps.

Many papers have been published on this topic: they state that training "vanilla" recurrent neural networks is really hard because of this problem. To adress the issue, more complex architectures have been proposed, like *Long Short Term Memory* networks (LSTM).

3 RNN architecture advantages

- They can process input of any length;
- Model size does not increase for bigger inputs;
- Same weights are applied at each timestep, so they are efficient and symmetric;

4 LSTM Architecture

The lstm architecture was first designed to adress the issue of vanishing and exploding gradient. In recurrent neural networks, it is really hard to learn long term dependencies because of these two problems. One thing that used to be done before lstm was to backpropagate only through a limited amount of timesteps, trying to get around the problem of vanishing and exploding gradient. A smart way to improve this approach is to let the recurrent cell learn by itself a **gating** logic that could preserve long term dependencies and block or allow the flow of the gradient during backpropagation.

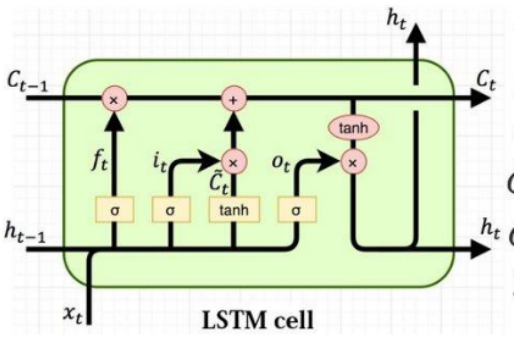

Figure 2: LSTM architecture

$$\begin{cases} i = \sigma(x^{(t)}U_i + h^{(t-1)}W_i) \\ f = \sigma(x^{(t)}U_f + h^{(t-1)}W_f) \\ o = \sigma(x^{(t)}U_o + h^{(t-1)}W_o) \\ g = \tanh(x^{(t)}U_g + h^{(t-1)}W_g) \end{cases} \tag{11}$$

$$\begin{cases} c^{(t)} = f \odot c^{(t-1)} + g \odot i \\ h^{(t)} = \mathrm{tanh} c^{(t)} \odot o \end{cases} \tag{12}$$

4.1 Ifog

- i is the **input** gate: it specifies which elements of the cell state should be overwritten. It uses σ;
- f is the **forget** gate: it specifies which elements of the cell state should be resetted (forgot). It uses σ;
- o is the **output** gate: it specifies which elements of the cell state should be exposed to the outside world. It uses σ;
- g (called $\widetilde{C}_t$ in the above figure) is the **gate gate**: it computes the new candidate elements for the new cell state. It uses tanh.

It is easier to think about the "sigma" gates as if they should return a binary value, either zero or one: they can express which one of the elements of the cell state should be mantained, overwritten, or exposed to the outside. As you can see from 12 each one of the gates has its own weight matrix, that are changed and fine tuned during training to avoid the problems of RNNs.