Scalable Machine Learning and Deep Learning - Review Questions 4

Deadline: December 1, 2019

Anna Martignano, Daniele Montesi

May 4, 2020

1. What's the vanishing problem in RNN?

   In Deep Learning, vanishing gradient problem is exhibited whenever a network is made of multiple layers and their activaction function is always outputting a value ¡ 1 than, whenever multiplied on every layer, the gradient **vanishes** becoming very small. In RNN the vanishing problems is present as the number of layers of the network is dependent on the **length** of the output/input vector. Suppose we have to predict the probability of a word "was" or the word "were" in a sentence having a previous sentence: "The cat, whose .... , xxx full". Which word should be inserted at the place of "xxx" space?.

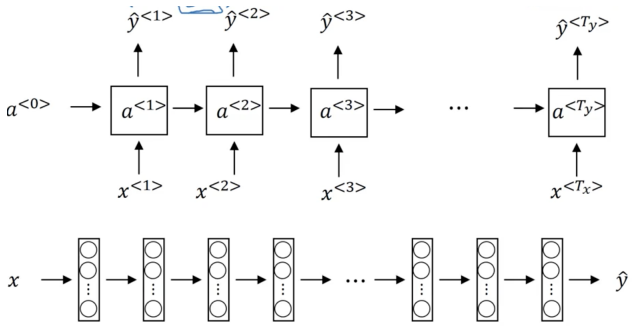


Figure 1: RNN associated to multiple input sequence (courtesy of deeplearning.ai)

   Due to the long lenght of the sentence, the network needs to save a state accounting for words very far from the ones to be predicted. This is a problem in RNN. When performing backward-pass of the backpropagation the weights associated to the first sentence sequence will have a very small contribution of the latest words, since in the middle there are many other words. In other words, when unfolded, each one of the single values to input/output sequence constitute a layer, turning the network into a case analogous to a deep feed-forward neural network. The network is able to "Store" the memory of the inputs on the weights w only for close states. As solution, we can implement a LSTM to take into account the loosing-memory problem.

---

2. Explain the impact of different gates in LSTM?

   Long-Short-Term-Memory technique (or LSTM) helps the Recurrent Neural Network to avoid the problem of Vanishing Gradient after the network unfolding. Their principle stands into the processing of the neuron state computed by the neurons of a given step t (C^t), re-weighting it through many **gates**. IN LSTM there are 2 states: long term state (C^t) and short term state ((h^{t-1}). The short-term state is made up of multiple activation functions constituting the **gates**. Those gates models also the output h^t.

In LSTM there are 4 gates:

- **Forget Gate**: decides what information has to be forgotten. Applied a **sigmoid** over the h^{t-1} and x^t. In the extreme cases, if forget gate outputs 1: keep all, if outputs 0: forget all.
- **Input Gate**: decides what of the input should be stored in the cell state C^t
- **Candidate Generation Gate**: generate a new vector candidate applying a *sigmoid*. This candidate is then weighted based on the Input gate's output and added to the cell state C^{t-1} to make the state C^t
- **Output Gate**: decides the output h^t based on the previous gate's outputs. In particular, it multiplies the *sigmoid* (over the input weights) and the *tanh* over the cell state C^t

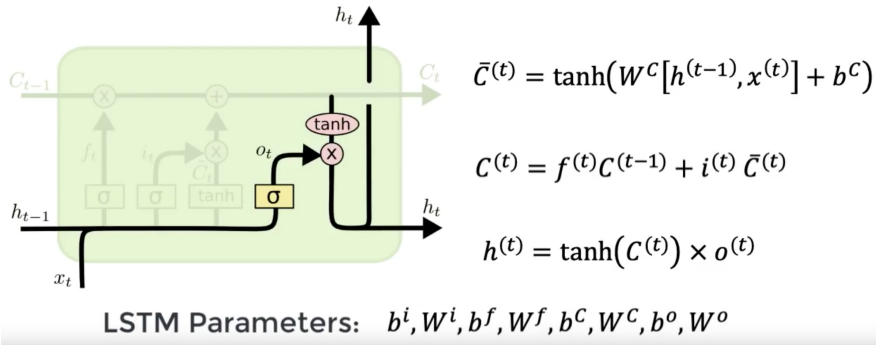
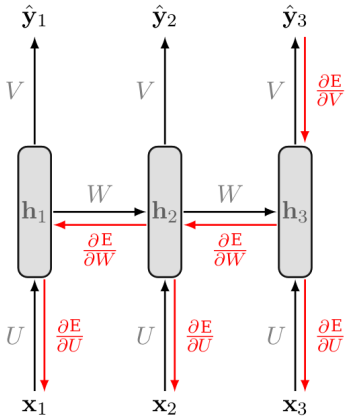


Figure 2: LSTM gates and parameters (from the slides)

LSTM adds many parameters to the network due to the multiple layers each cell introduces. However, the network is prevented by suffering of vanishing/exploding gradient.

---

3. Assume the error of the following network is $\mathtt{E} = \mathtt{E}^{(1)} + \mathtt{E}^{(2)}$, then compute the $\frac{\partial \mathtt{E}}{\partial \mathtt{u}}$.



$$\frac{\partial E}{\partial u} = \sum_t \frac{\partial J^{(t)}}{\partial u} = \frac{\partial J^{(1)}}{\partial u} + \frac{\partial J^{(2)}}{\partial u} \tag{1}$$

$$\frac{\partial J^{(1)}}{\partial u} = \frac{\partial J^{(1)}}{\partial \hat{y}^{(1)}} \frac{\partial \hat{y}^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial u} \tag{2}$$

$$\frac{\partial J^{(2)}}{\partial u} = \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial u} + \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial u} \tag{3}$$

More in general, the formula to compute the derivatives of $\frac{\partial J}{\partial u}$ or $\frac{\partial J}{\partial w}$ is the following:

$$\frac{\partial J^{(t)}}{\partial u} = \sum_{k=1}^{t} \frac{\partial J^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial z^{(t)}} \frac{\partial z^{(t)}}{\partial h^{(t)}} \left(\prod_{j=k+1}^{t} \frac{\partial h^{(j)}}{\partial s^{(j)}} \frac{\partial s^{(j)}}{\partial h^{(j-1)}} \right) \frac{\partial h^{(k)}}{\partial s^{(k)}} \frac{\partial s^{(k)}}{\partial u} \tag{4}$$

---

4. Assume we have a stacked autoencoder with three hidden layers $\mathtt{h_1}$, $\mathtt{h_2}$, and $\mathtt{h_3}$, in which each layer applies the following functions respectively, $\mathbf{h_1} = \mathtt{f_1}(\mathbf{x})$, $\mathbf{h_2} = \mathtt{f_2}(\mathbf{h_1})$, and $\mathbf{h_3} = \mathtt{f_3}(\mathbf{h_2})$, and the output of the network will be $\mathbf{y} = \mathtt{f_4}(\mathbf{h_3})$. Do you think if it is a good autoencoder if it generates $\mathtt{f_4}(\mathtt{f_3}(\mathtt{f_2}(\mathtt{f_1}(\mathbf{x})))) = \mathbf{x}$ for all input instances $\mathbf{x}$. How can we improve it?

   In this case, the autoencoder $\mathtt{f_4}(\mathtt{f_3}(\mathtt{f_2}(\mathtt{f_1}(\mathbf{x})))) = \mathbf{x}$ simply succeeds in learning how to copy perfectly the input, without enriching the input representation.
   One way to obtain meaningful features from the autoencoder is to force it to have a smaller dimension than x. So, we obtain an **undercomplete** autoencoder whose code dimension is less than the input dimension able to capture the most salient features of the training data instead of simply reproducing them exactly.
   An alternative to undercomplete autoencoder is to regularize the autoencoder by using a loss function that encourages the model to have other properties besides the ability to copy its input to its output. These other properties include:

   - sparsity of the representation
   - smallness of the derivative of the representation
   - robustness to noise or to missing inputs

   A **regularized** autoencoder is able to capture meaningful insights about the data representation even if the autoencoder is **nonlinear** and **overcomplete**.

---

5. How does Gibbs sampling work? When do we need to use Gibbs sampling?

   Gibbs sampling is a Markov chain Monte Carlo (MCMC) algorithm used when the direct sampling from a multivariate probability distribution is not possible or too difficult.
   The underlying idea is to generate posterior samples by **sweeping** through each variable (or block of variables) to sample from its conditional distribution with the remaining variables fixed to their current values.
   If we consider the random variables X_1, X_2, and X_3, we start by setting these variables to their initial values $x_1^{(0)}$, $x_2^{(0)}$, and $x_3^{(0)}$ (often values sampled from a prior distribution q). At iteration i, we sample:

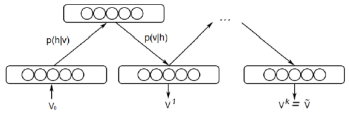
   - $x_1^{(i)} p(X_1 = x_1 | X_2 = x_2^{(i1)}, X_3 = x_3^{(i1)})$
   - $x_2^{(i)} p(X_2 = x_2 | X_1 = x_1^{(i1)}, X_3 = x_3^{(i1)})$
   - $x_3^{(i)} p(X_3 = x_3 | X_1 = x_1^{(i1)}, X_2 = x_2^{(i1)})$

   The process terminates when the sample values have an *approximate* distribution (under a certain threshold) as if they were sampled from the true posterior joint distribution.
   The Gibbs sampling can be used in the context of **Restricted Boltzmann Machine**, i.e. a generative stochastic artificial neural network that can learn a probability distribution over its set of inputs.
   Specifically for RBMs, **sweeping** is achieved by the formula:

   - $a(h_j) = \sum_{i=1}^{I} h_i \times v_i$
   - $P(h_j|v) = sigmoid(a(h_i))$

Where I is the total number of visible neurons, j is a particular hidden neuron. Each hidden neuron h_j is set to 1 with the probability $P(h_j|v)$. The inverse process is repeated to determine $P(v_i|h)$. It is much easier to sample from the conditional distribution and trying to derive an approximate joint distribution, instead that computing the joint distribution itself.

Here Gibbs sampling can be used to compute a chain of conditional probabilities as shown in the picture.



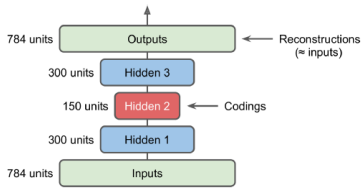
The process needs to be repeated k times till the values convergence and here it is described:

- Given an input vector v, compute the conditional probability $p(h|v)$.
- Knowing the hidden values h, we use $p(v|h)$ for prediction of new input values v.

---

6. How do you tie weights in a stacked autoencoder? What is the point of doing so?

The common architecture of stacked autoencoder is symmetrical respect to the *coding layer*, i.e. the central hidden layer which is the most efficient data representation of the entire stack.



A best practice in case of symmetrical autoencoder is to tie the weights of the decoder layers to the weights of the encoder layers. That is to say, we enforce the weights on encoder and decoder as equal, which is one of the PCA properties we would like to achieve. PCA and Autoencoders share architectural similarities, but their properties must be in someway enforced by the external.

Moreover, with tied weights we halve the number of weights which has the beneficial effects of:

- Reducing the risk of overfitting
- Increasing the training speed