Sequence Modeling for Data Mining

Marcel Pons Cloquells

May 21, 2021

1 Introduction

A great number of data mining problems involve handling data that is sequentially related, either having a temporal dimension such as in medical signals, or being made up of individual symbols which conform a sequence, such as in text and genomics data. In order to predict such kind of data, sequence prediction models use historical sequence information by taking previously observed phenomena or symbols as input, which conform the sequence and are commonly referred as *time steps*, and use them to learn and ultimately perform predictions for new unobserved symbols or sequences.

Nowadays, the state-of-the art models that are used for sequence prediction are based on neural networks (i.e. deep learning models) which have a better performance than other statistical models that take context into account but do not capture as effectively the sequential dependencies and relationships, like Hidden Markov Models.

When considering sequence modeling, the range of situations that involve temporal input and potentially sequential outputs is expanded, more precisely, in contrast to feed forward models that are suited to one-to-one problems (e.g. binary classification of spam/no spam given input features), situations of many-to-one, one-to-many, and many-to-many can be handled.

- Many-to-one: A model takes as input several time steps and generates a unique output. For example, a language processing problem known as sentiment analysis, which consists on predicting the sentiment or feeling (output) associated to a sequence of input words.
- One-to-many: The input does not follow a sequential structure, but the generated output does. An example of this situation is that of image captioning, which consists of producing an output sequence of words describing the content of an input static image.
- Many-to-many: Each step in the input has associated an output. Thereby a sequence is translated into another sequence. Examples for these situations are specially seen in machine translation.

In this article, *Recurrent Neural Networks* and *Long Short-Term Memory* networks will be introduced, which are the basis from which more complex models used nowadays are built, like Transformers. Among the many applications of these models, those of predicting the most likely next word to occur given an incomplete sentence will be used as an example.

First, a brief overview on neural networks is required in order to introduce the aforementioned models and some of the algorithms used, like *Backpropagation* and *Gradient descent*[1]. Furthermore, how these models are not suited for sequence prediction will be exposed.

2 Feed-forward neural network (overview)

Network Architecture

The fundamental block of neural networks is a single neuron, also known as a *perceptron*, which takes a set of inputs $x_1, x_2, \ldots, x_m$ that are multiplied by their corresponding weights $w_1, w_2, \ldots, w_m$, local to the neuron. All the input-weight multiplication results are added together and are passed through a

[1]The mathematical details of the algorithms and models will not be covered in depth, since the overview is only aimed to give an intuition of how they work.

non-linear activation function $g(\cdot)$ to produce the final output $\hat{y}$. Furthermore, a bias w_0 is considered in order to shift the activation function by adding a constant to the input.

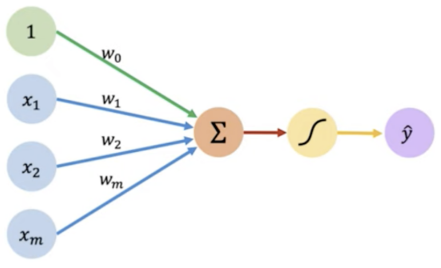

Figure 1: Representation of a perceptron.

So, a single artificial neuron like the one in Figure 1 is used to model functions $\mathbb{R}^m \mapsto \mathbb{R}$, where $\hat{y}$ is calculated by the dot product between the inputs and the weights, adding a bias and applying a non-linear function.

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i \cdot w_i\right) = g(w_0 + \boldsymbol{X}^T\boldsymbol{W}) \tag{1}$$

The weights $w_0, \ldots, w_m$ and bias are the parameters of the model. The learning task is to find suitable weights that allow neurons to model the data as accurately as possible.

A multi-layer network consists on placing layers in sequence, each one conformed by several artificial neurons. Every layer is fully-connected to the next one using as inputs the outputs of the preceding one. With regard to the terminology used to name the layers: the first one corresponds to the *input* layers, then the in-between layers are known as *hidden* layers[2] and the final one is the *output* layer. Each of these layers has its own set of weights $W^{(i)}$ and a bias term.

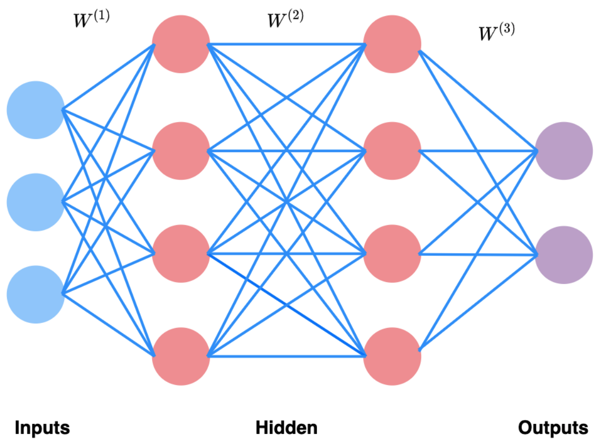

Figure 2: Schematic representation of a multi-layer network.

Learning Process

The *loss function* of a neural network defines in what extent wrong predictions occur, that is, the cost incurred from predictions that are far from the ground truth. The mean value of the loss for each individual observation is denominated *empirical loss* (2), which is the value to be minimized during the

[2]The name comes due to the fact that the states of these layers are typically unobserved and hidden to some extent.

process of learning.

$$J(\boldsymbol{W}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(x^{(i)}; \boldsymbol{W}), y^{(i)}) \tag{2}$$

Two loss functions that are frequently used are the *Softmax Cross-Entropy* (3) for classification problems and the *Mean Squared Error* (4) for regression problems.

$$J(\boldsymbol{W}) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \ log(f(x^{(i)}; \boldsymbol{W})) + (1 - y^{(i)}) \ log(1 - (f(x^{(i)}; \boldsymbol{W})) \tag{3}$$

$$J(\boldsymbol{W}) = \frac{1}{m} \sum_{i=1}^{n} (y^{(i)} - f(x^{(i)}; \boldsymbol{W}))^2 \tag{4}$$

The whole process of optimizing neural networks consists in finding the optimal weights and biases that minimize the loss function as much as possible. To that end, gradient descent methods are used to find the weights that make the empirical loss reach a local minima. All these methods originate from the *Gradient Descent* algorithm, which consists in iteratively finding, by using the partial derivatives (i.e. the gradient), the direction of steepest descent in the value of $J(W)$, moving down in that direction (with a predefined step size), updating W and repeating the process until convergence is reached. The size of the step in each iteration is called Learning Rate (η).

Algorithm 1 Gradient Descent Algorithm

```
1: Initialize weights randomly $\boldsymbol{W} \sim \mathcal{N}(0, \sigma^2)$
2: while not converged do
3:     Compute gradient $\frac{\partial J(W)}{\partial W}$
4:     Update weights $W \leftarrow W - \eta \cdot \frac{\partial J(W)}{\partial W}$
5: end while
6: return $\boldsymbol{W}^*$
```

In practise, the optimization of neural networks is arduous and computationally intensive. The loss functions are extremely non-convex, having many local minima and saddle regions, making the gradient descent not good enough due to the fact that it can get stuck in these regions. Consequently other gradient descent based algorithms with adaptive learning rate have emerged over the years, such as *Stochastic Gradient Descent (SGD)*, *Adam*, *RMSProp*, among others.

The way by means the neural networks are optimized has been introduced, where the gradients have the special role of determining how changes in weights affect the loss. But, how are the gradients for each vector of weights $W^{(i)}$ obtained? The *backpropagation* algorithm is the responsible of these computations, characterized by its extensive use of the chain rule for computing derivatives and its efficiency achieved by storing intermediate results in a dynamic programming manner.

This algorithm backpropagetes the information that comes from the loss function all the way to the first vector of weights, determining how every single weight in the network needs to change to impact $J(W)$. In a very simplified way and without dwelling into the derivative details, the algorithm can be explained in three steps that are repeated for each backward pass through the layers until the first one is reached. The first step consists on computing the error of the last layer (L), where a is the activation function and z is the sum of the dot products between the inputs and weights of the previous layer.

$$\delta^L = \frac{\partial J}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \tag{5}$$

Then, the error of L is backpropagated to the previous layer, $L - 1$ (6), obtaining the new error.

$$\delta^{L-1} = \frac{\partial J}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} = \delta^L \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{z^{L-1}} \tag{6}$$

δ^{L-1} is used to calculate the partial derivatives of J with respect to the bias b (7) and with respect to the weights w (8).

$$\frac{\partial J}{\partial b^{L-1}} = \delta^{L-1} \cdot \frac{\partial z^{L-1}}{\partial b^{L-1}} \tag{7}$$

$$\frac{\partial J}{\partial w^{L-1}} = \delta^{L-1} \cdot \frac{\partial z^{L-1}}{\partial w^{L-1}} \tag{8}$$

The error δ^{L-1} will also be backpropagated to the previous layer, where the same process will take place and repeat until the first layer is reached. In this way, the gradient vector composed by all the $\frac{\partial J}{\partial w^{(l)}}$ will be obtained (line 3 of the gradient descent pseudocode) and applied to the corresponding step of the optimization algorithm to update the weights.

Incompatibility with sequence data

The feed-forward model seen so far is not well suited for handling ordered sequential data because it does not have a notion of time or sequence. The inputs and outputs present in these models are static or fixed. For this reason, if the models take a sequence over and over again for each time step, treating them isolatedly, the relationship that is inherent to sequence data between input time steps is lost. Consequently, the output of a step is not going to depend on previous steps.

3 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks achieve the maintenance of the inherent relationship in the sequences relating the computations that the neurons are doing in a particular time step from prior steps as well as the input at the current time step, thereby having a sense of forward looking by passing the information to future steps. In this way, these models can capture dependencies and context in a sequence and use this information to make predictions, such as what is the most likely word to occur in a sentence like *the sky is cloudy, it is likely to . . .*

The way by means the computations are linked between steps comes from the pass of a *cell state* h_t from one step to the other, capturing in this way some notion of memory. Besides, a key feature of RNNs is that they use the same activation function and the same set of parameter weights at each time step of processing the sequence (i.e. no different $W^{(i)}$ are used). It is important to remark that this set of weights is not maintained constant over the pass through the network, but that it is changing over the course of the sequence while being applied to each individual time step. This relation can be depicted as a cycle that shows the concept of a recurrent relation, which unrolled represents a neural network with t time steps (Figure 3).

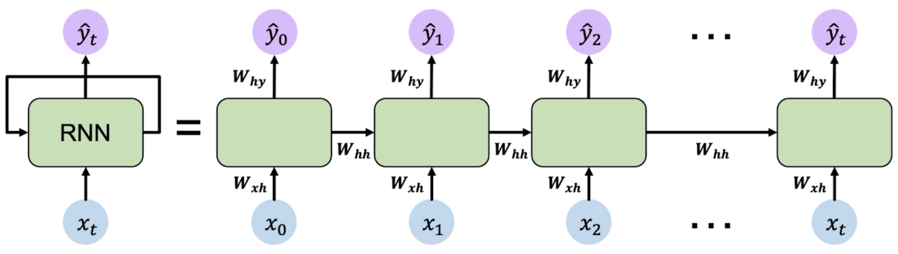

Figure 3: Schematic representation of a Recurrent Neural Network. *Source:* [1].

The cell state is computed by a function that takes the current time step input x_t as well as the prior cell state of the preceding time step h_{t-1} (9). The activation functions used are not exclusively of this type of networks but they are standard ones used in other neural networks, such as the *sigmoid* and *ReLU* functions.

$$h_t = g(W_{hh}^T \cdot h_{t-1} + W_{xh}^T \cdot x_t) \tag{9}$$

Learning Process

For the purpose of training a RNN, sequential training data $x^{(1)}, \ldots, x^{(t)}$ is passed to the model, which during the initial forward pass through the network computes an output distribution $\hat{y}^{(t)}$ for every time step taking into account the inputs given so far. For instance and following the scenario of the word prediction, the network would predict the probability distribution of every word given the words computed so far.

The outputs for each individual time step have their corresponding loss function determining the extend by which the prediction resembles the ground truth[3]. The empirical loss $J(\theta)$ that is used to train the model and ultimately assess the final performance is obtained by the addition of these individual loss functions[4].

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} \mathcal{L}^{(t)}(\theta) \tag{10}$$

Recurrent Neural Networks minimize $J(\theta)$ in a similar way as feed forward models, however, since these models are composed on various time steps to capture the sequence information, the gradients of the loss $J(\theta)$ with respect to the parameters are propagated through each individual time step as well as across them, all the way from the last time step back to the beginning. Due to these characteristics, this variant is called *Backpropagation through time* (BTT). As it is mentioned above, in RNNs the matrix of weights W is shared between all time steps, therefore the gradient in each step is performed upon this repeated matrix. Due to this fact, BTT adds the gradients as it backpropagates over the time steps:

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial W_h} \right|_{(i)} \tag{11}$$

Figure 4 illustrates the process of backpropagation through time:

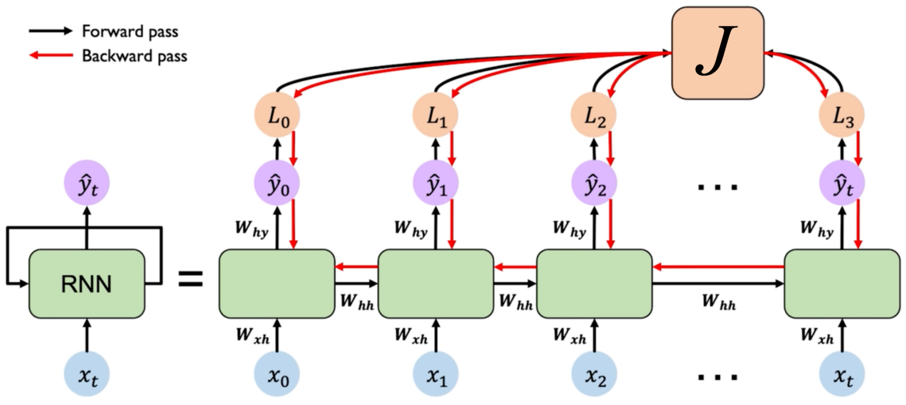

Figure 4: Representation of BTT. *Source:* [1].

[3]Here we consider this scenario, but situations in which only the output of the final time step is used to calculate the loss are also typical.

[4]In our example of classification of the next word, cross-entropy loss function is used to measure the loss between the predicted probability distribution $\hat{y}^{(t)}$ and the true next word $y^{(t)}$.

The problem that arises with BTT is that computing the gradient with respect to the initial cell state involves so many factors of W and also repeated computations of the gradient with respect to W.

This could lead to two problems. The first one, usually less common, is called *exploding gradients*, which arises when the gradients are large and become larger due to the chain of multiplications across the network, eventually being infeasible to optimize[5]. *Gradient clipping* is one way to deal with this issue, which involves scaling back the values of particularly large gradients by some factor, thereby mitigating somehow the problem.

The other problem, more frequently noticed in sequence modeling, are *vanishing gradients*, characteristic when the chain of multiplications takes place between small gradients, generating increasingly smaller values as going back through the network. These gradients make the process of propagating errors from the loss function back to the distant past extremely difficult, making the longer term dependencies between time steps hardly appreciable and, in contrast, biasing the weights and parameters to capture shorter term dependencies. For example, due to this issue, RNNs encounter difficulty in accurately predicting the next word in a sentence like the one below because of not capturing the longer term dependencies between the 7^{th} step and the target word *project* at the end:

> *When she tried to print her project, she found that the printer was out of toner. She went to the copy shop to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her ___*

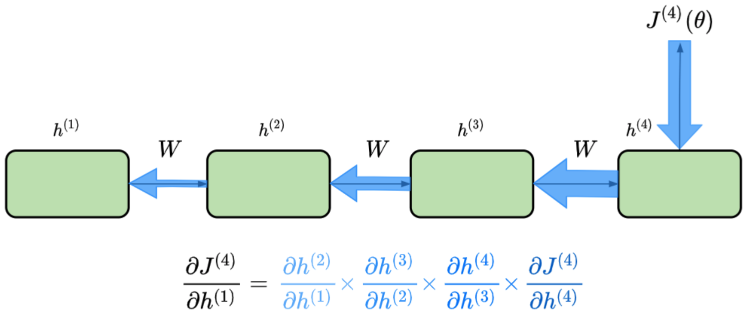
$J^{(4)}(\theta)$
$h^{(1)}$
$h^{(2)}$
$h^{(3)}$
$h^{(4)}$
W
W
W
$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$

Figure 5: Schematic representation of the vanishing gradient problem, where each partial derivative is smaller as going backwards through the time steps.

There are several ways to deal with vanishing gradients. The most robust and common one consists in introducing a more complex recurrent unit, called *gated cells*, used specially by *LSTM* networks.

4 Long Short-Term Memory RNNs

LSTMs have a chain like structure as seen so far, but the recurrent unit is slightly more complex than in RNNs, containing different interacting layers which are defined by standard neural network operations (i.e. non-linear functions and point wise operations). These layers, typically called *gates*, control which information passes through the network and which not. Moreover, apart from a hidden state h_t, on each step t there is also a cell state c_t, which stores long term information. The cell state goes through the sequence as a flow of information, being effectively and selectively modified in each time step by the aforementioned gates.

The gates dynamically select, depending on the context, which information is erased, written or read by controlling in what extent they are opened, closed or somewhere in-between. Each gate undertakes a role

[5]If the gradients are considerably big, the update step in gradient descent algorithms is large, eventually involving reaching bad parameter configurations.

in the processing of the information and cell state, where the flow and storage in each time step can be divided in 3 steps:

1. The first step is to determine what part of the older information is irrelevant and forget it. This decision is achieved by taking the previous history state h_{t-1} and the time step input x_t and passing them though a sigmoid gate σ, which outputs a number between 0 and 1 modulating how much should be passed in or kept out (12).

$$f_t = \sigma(W_f \cdot [h_{h-1}, x_t] + b_f) \tag{12}$$

The old cell state (c_{t-1}) is multiplied by f_t, forgetting the information that was considered irrelevant.

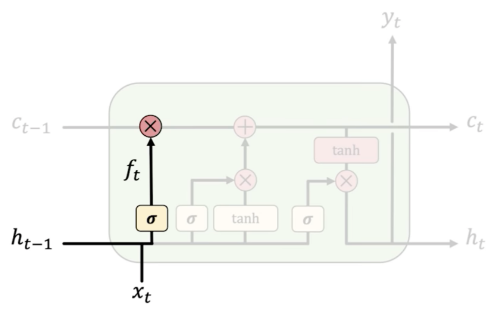

Figure 6: Forget gate schema. *Source:* [1].

2. The next step is to determine what part of the previous information is relevant and store it into the i_t cell state. Another sigmoid gate σ filters the relevant information and an hyperbolic tangent (*tanh*) gate creates a vector of new candidate values $\tilde{C}_t$ that is multiplied by i_t.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{13}$$
$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{14}$$

The old cell state c_{t-1} is updated into a new cell state c_t by adding the updating cell state ($i_t \otimes \tilde{C}_t$).

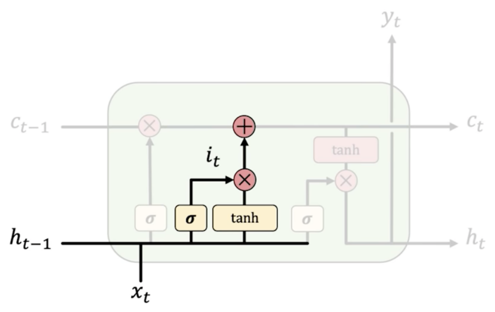

Figure 7: Update gate schema. *Source:* [1].

3. An interacting layer, composed by a sigmoid σ and a *tanh*, controls what information encoded in the cell state c_t is ultimately outputted and sent to the network as input in the following time step. This operation controls both the value of the output y_t as well as the cell state that is passed on, time step to time step, in the form of h_t.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{15}$$
$$h_t = o_t \otimes tanh(C_t) \tag{16}$$

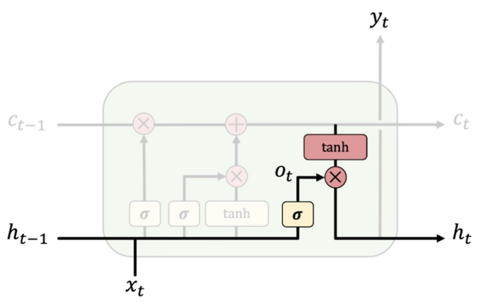

Figure 8: Output gate schema. *Source:* [1].

With the mechanisms seen so far, LSTM networks effectively capture longer term dependencies and also help to overcome the vanishing gradients problem. All of these gating mechanisms work to allow the uninterrupted flow of gradient computations over time, first controlling the gradient values at each time step by means of the initial forget gate and second by balancing the gradient values during backpropagation achieved by maintaining the separate cell state c_t across the gradient computations.

5 References

[1] MIT Introduction to deep learning. http://introtodeeplearning.com/. Accessed: 2021-05-01.

[2] Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2021-05-09.

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

[4] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.