RECURRENT NEURAL NETWORKS TUTORIAL

1. General Working Principle

Before getting into the architecture of RNNs, general working principle regarding the networks that will be presented in this section, is given with the figure 1 below.

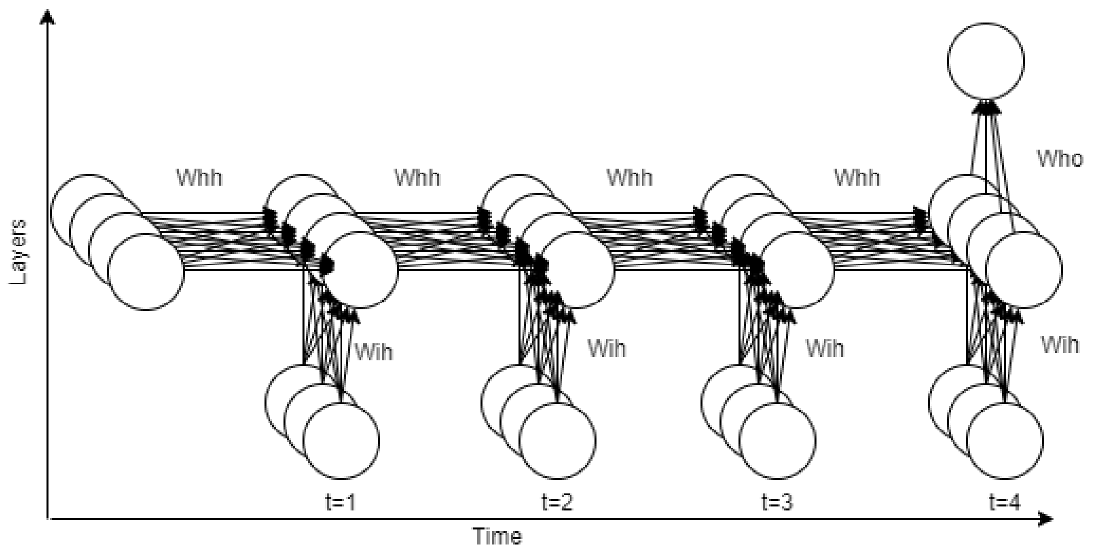

Figure 1 : General look of a Recurrent Neural Network unfolded through time using many-to-one architecture

The depicted network contains three layers as it can be seen by looking at the "Layers" axis. It has 3 neurons in input layer, 4 neurons in hidden layer and 1 neuron in output layer. The other axis "Time" shows the amount of time steps in the input data. As a solid example, assume that there exists a data set which contains 3 different variable. About the data set, it is also known that the value of 3rd variable has a complex dependency with other two variables. As a multivariate time series forecasting application, it is wanted to predict next value of the 3rd variable by taking the last 4 value of all three variables into account. So, each node in the input layer corresponds to a variable category and each tick in time axis corresponds to a time step.

In the forward pass, for each time step t, hidden layer values of that time step are calculated as a function of input layer at time step t and hidden layer at time step t-1. Another key point about this structure is the weights. As it is written in the figure 1, all the occurrence of the weights Wih, Whh and Who is identical. Thus, for this particular example there only exists 3 kind of weight matrices which are shared through the time. As an initial condition, hidden layer at the time step 0 is filled with zeros and input vectors start to be given into network from time step 1.

The example in the paragraph above can be seen as a typical “many-to-one” application. It means that it takes an input with more than one time steps and gives only one output at the end of calculations through time. There also exist different applications such as “one-to-many” or “many-to-many” as it can be seen in the figure 2.

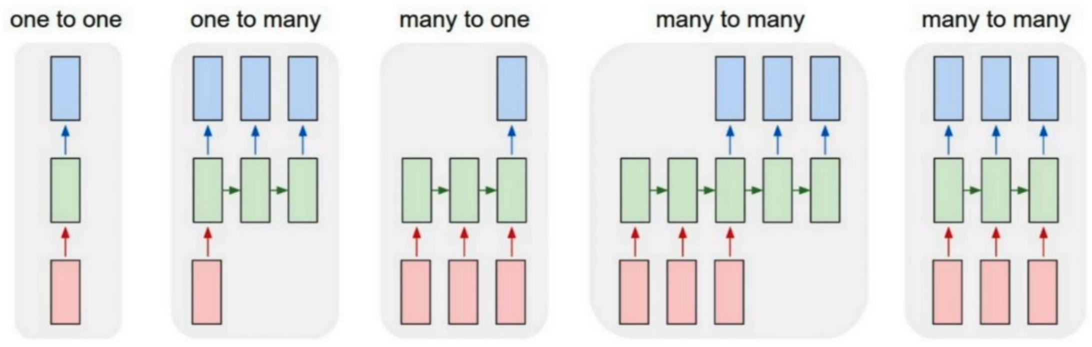

Figure 2 : Different working schemes of Recurrent Neural Networks [1].

2. Simple (Vanilla) RNN Mathematical Model

In the formal definition of a standard Recurrent Neural Networks for a sequence of input vectors $x_1, x_1, x_1, \ldots, x_T$, the networks creates a sequence of hidden layer states $h_1, h_1, h_1, \ldots, h_T$ as a function of previous hidden state vector and present input vector, Finally a sequence of output vectors $y_1, y_1, y_1, \ldots, y_T$ are produced. In the feedforward calculation, the equations 1 and 2, are performed for each time step t, where the expressions $W_{xh}, W_{hh},$ and W_{hy} denote weight matrices, b_h and b_y denote biases and σ denotes activation function. It is also a useful practice to define an initial bias for the place of the term $W_{hh}h_0$ which returns zero.

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

$$y_t = W_{hy}h_t + b_y \tag{2}$$

The illustrated version of the of the equations 1 and 2 are given with the figure 3.

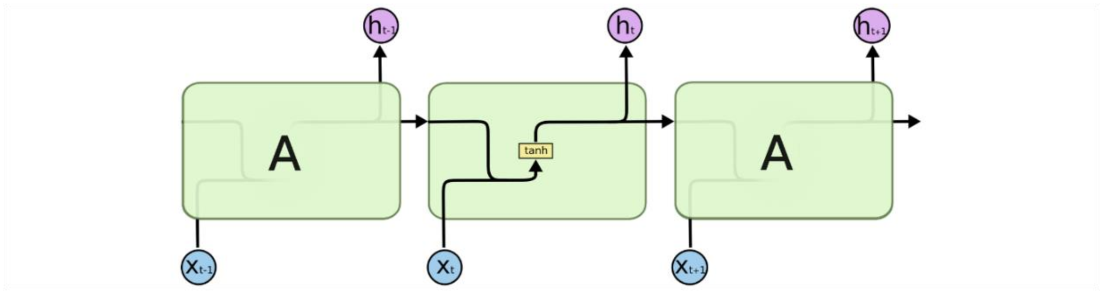

Figure 3 : Illustration of Simple RNN [2].

3. Backpropagation

After the forward calculation phase, total error regarding the output vectors is calculated according to the equation:

$$\mathcal{E} = \sum_{1 \leq t \leq T} \mathcal{E}_y \tag{3}$$

where each error value for each output is calculated according to loss function $\mathcal{E}_y = \mathcal{L}(y_t)$. In order to optimize weights according to calculated error, backpropagation through time (BPTT) algorithm is used with the optimizer stochastic gradient descent. For the sake of simplicity all of the weight matrices and biases will be denoted with symbol θ for the following mathematical formulas:

$$\mathcal{E} = \sum_{1 \leq t \leq T} \mathcal{E}_y \tag{4}$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial y_t} \frac{\partial y_t}{\partial h_k} \frac{\partial h_k}{\partial \theta}\right) \tag{5}$$

$$\frac{\partial y_t}{\partial h_k} = \prod_{t \ge i > k} \frac{\partial h_i}{\partial h_{i-1}} = W_{hh}^T diag(\sigma'(h_{i-1})) \tag{6}$$

As it is shown, the term $\frac{\partial \mathcal{E}_t}{\partial \theta}$ is actually a sum of all the matrices $\frac{\partial \mathcal{E}_t}{\partial \theta}$ at each time step k for each time step t, where $T > t > k$. In other words, as the network opened up through time for an output vector at time step t, it is visible that all the equations of the previous time steps which contain the term θ, have an effect on that particular output vector. In order to optimize this effect, derivatives of these terms with respect to that particular output vector have to be taken into account with a sum operation.

4. Appearance of vanishing gradient

During the backpropagation phase, in the equation 5 the term $\frac{\partial y_t}{\partial h_k}$ enables the calculated error to be carried along the time steps (from t to k). In 1994 it is revealed that as the distance between t and k grows, the magnitude of the term $\frac{\partial y_t}{\partial h_k}$ decay exponentially [3]. This situation which is called "Vanishing Gradient" decrease the efficiency of learning the long-term dependencies. In [3], it is concluded that under the assumption of the use of hyperbolic activation functions, either network would not be robust to input noise or would not be efficiently trainable by gradient descent with required long-term dependencies. As an indirect outcome it is also noted that, deep neural networks without recurrences may also be a subject to this problem, due to the fact that an unfolded recurrent neural network is not so different than a deep feedforward neural network with shared weights.

As a solution to this problem different optimization techniques have been offered such as "Simulated Annealing" in [4], "Time-Weighted Pseudo-Newton Optimization" in [3] and [5] or

“Discrete Error Propagation” in [3]. Beside these solutions, today an alternative practice for hidden layers is to use ReLu activation function which is an abbreviation for “Rectified Linear Unit”, instead of using sigmoid or tangent hyperbolic activation functions that map the magnitude of the output between certain intervals. The problem with sigmoid and tangent hyperbolic functions is that they squeeze the output to an interval. Thus, they are only sensitive for the values in this interval. Otherwise output gets saturated if it is out of the range which is $(-1,1)$ for tangent hyperbolic and $(0,1)$ for sigmoid. In [6], it is stated that due to its linear features no gradient vanishing effect depending on the non-linear natures of tanh and sigmoid. Furthermore, the computational cost of training a network with ReLu is much lower than sigmoid or tanh functions. In contrast to these advantages on preventing vanishing gradient, ReLu brings its own problems, such as not having an upper limit which causes exploding gradient problem or another a phenomenon called “Dying ReLu” which happens frequently when a neuron with ReLU become inactive and returns constantly zero no matter what the inputs are [7]. This insensitivity to the inputs leads erroneous results and insufficient convergence. Many alternative versions of ReLu such as ELU, LReLu or SReLu have been proposed to overcome “Dying ReLu” or other possible problems that may occur [8].

REFERENCES

[1] **Url-1** <http://karpathy.github.io/2015/05/21/rnn-effectiveness/> date retrieved 21.05.2015.

[2] **Url-2** <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> date retrieved 27.08.2015.

[3] **Elman, J.L.** (1990). Finding Structure in Time. *Cogn. Sci., 14*, 179-211.

[4] **Corana, Angelo & Marchesi, Michele & Martini, Claudio & Ridella, Sandro.** (1987). Minimizing Multimodal Functions Of Continuous-Variables with Simulated Annealing Algorithm. ACM Transactions on Mathematical Software. 13. 262-280. 10.1145/29380.29864.

[5] **Becker, Suzanna & Lecun, Yann.** (1989). Improving the Convergence of Back-Propagation Learning with Second-Order Methods.

[6] **Glorot, X., Bordes, A. & Bengio, Y..** *(2011). Deep Sparse Rectifier Neural Networks. Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, in PMLR 15:315-323*

[7] **Lu, L., Shin, Y., Su, Y., & Karniadakis, G.E.** (2019). Dying ReLU and Initialization: Theory and Numerical Examples. *ArXiv, abs/1903.06733*.

[8] **Clevert, D., Unterthiner, T., & Hochreiter, S.** (2016). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *CoRR, abs/1511.07289*.