Abdelwahid Benslimane
wahid.benslimane@gmail.com

Recurrent Neural Networks (Vanilla RNN, LSTM, GRU)

Presentation and deep delving

Index

1) Brief history

Recurrent Neural Networks (RNNs) are a type of model conceptually similar to traditional artificial neural networks organized in successive layers of computational units called neurons. The interest of RNNs can be summarized by stating that operations occurring at time t-1 can influence those occurring at time t. They are particularly used in Natural Language Processing (NLP) but especially for tasks involving prediction from time series data. In general, they are used to process sequential data (sequences of words forming text, daily temperature trends, etc.).

The first recurrent neural networks were born in 1982, the year in which American physicist John Hopfield created the network of the same name. The Hopfield network is actually an associative memory that allows storing and recalling other models ([1]). The first RNN in the modern sense of the term, trained using gradient backpropagation, was presented in 1990 by Jeffrey Elman, a cognitive science researcher at the University of San Diego in California. This is what is now referred to as vanilla RNN. Then, Sepp Hochreiter and Jurgen Schmidhuber from the University of Munich introduced Long Short-Term Memory (LSTM) networks in 1997 ([2]). These were intended to solve the vanishing gradient problem by offering the ability to store both long-term and short-term information. The vanishing gradient problem, simply put, is a phenomenon where certain data's impact on model training weakens, resulting in reduced efficiency. This notably occurs when an RNN is trained on sequences that are too long (data distant from the beginning of the sequence causing this phenomenon). Finally, Gated Recurrent Units (GRUs) were introduced by Kyunghyun Cho et al. in 2014 ([3]). GRUs are a simplification (or generalization) of LSTMs that are found to be easier to train.

2) RNN, general case

RNNs in general are neural network architectures that use previous activations to compute the current activation. The output corresponding to time t+1 (h_{t+1}) actually results from an operation performed on the input at time t+1 as well as a value calculated at time t (h_t), which in turn results from an operation involving a result calculated at time t-1 (h_{t-1}). This historicity of information considered in prediction can theoretically be as long as desired because it directly depends on the length of the input sequence to the network, as illustrated in Fig. 1.

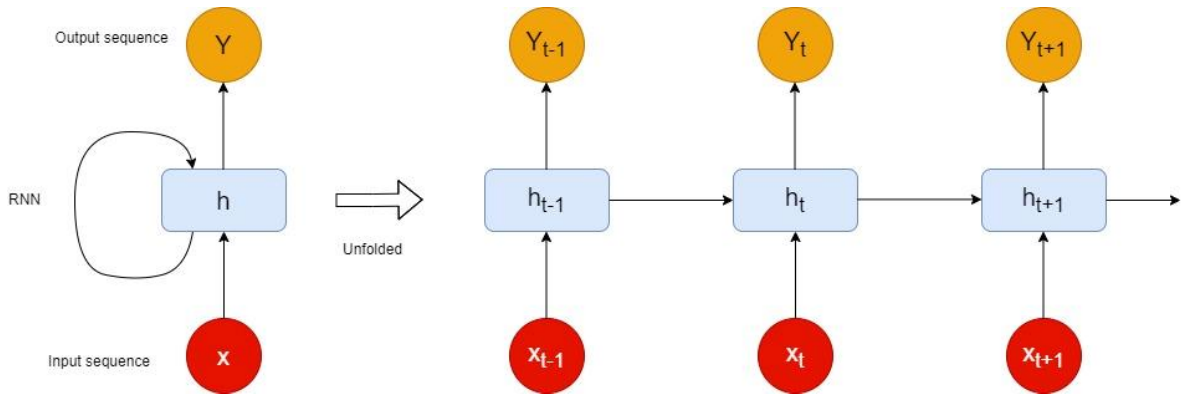

Fig. 1. Simplified structure of an RNN

RNNs can exhibit stacked hidden layers of arbitrary depth, with each layer possessing as many cells organized sequentially as the preceding layer, and consequently as many as the input data points of the model (each cell being directly linked to a data point in the input sequence of the model). This stacking of successive layers enables the discovery of complex recurrent patterns within time series, while the number of sequential cells within a layer allows for the consideration of varying lengths of historical information for prediction calculation. The final layer of an RNN is typically connected to a dense layer that produces the desired output result of the model.

Each cell of an RNN can contain multiple neurons. The number of neurons is the same for all cells within a given layer, but this number may vary from one layer to another. As mentioned, cells are processed sequentially, while the processing of neurons within the same cell can be parallelized. It should be noted that the output of an RNN cell is a vector of variable length (the same for each cell within the same layer and dependent on the chosen architecture). Additionally, the input sequence of the RNN is a sequence of vector data points (each data point can be a vector of length 1). The diagram in Fig. 2 provides a macroscopic view of a deep RNN connected to a dense layer, showing the neurons contained within the cells.

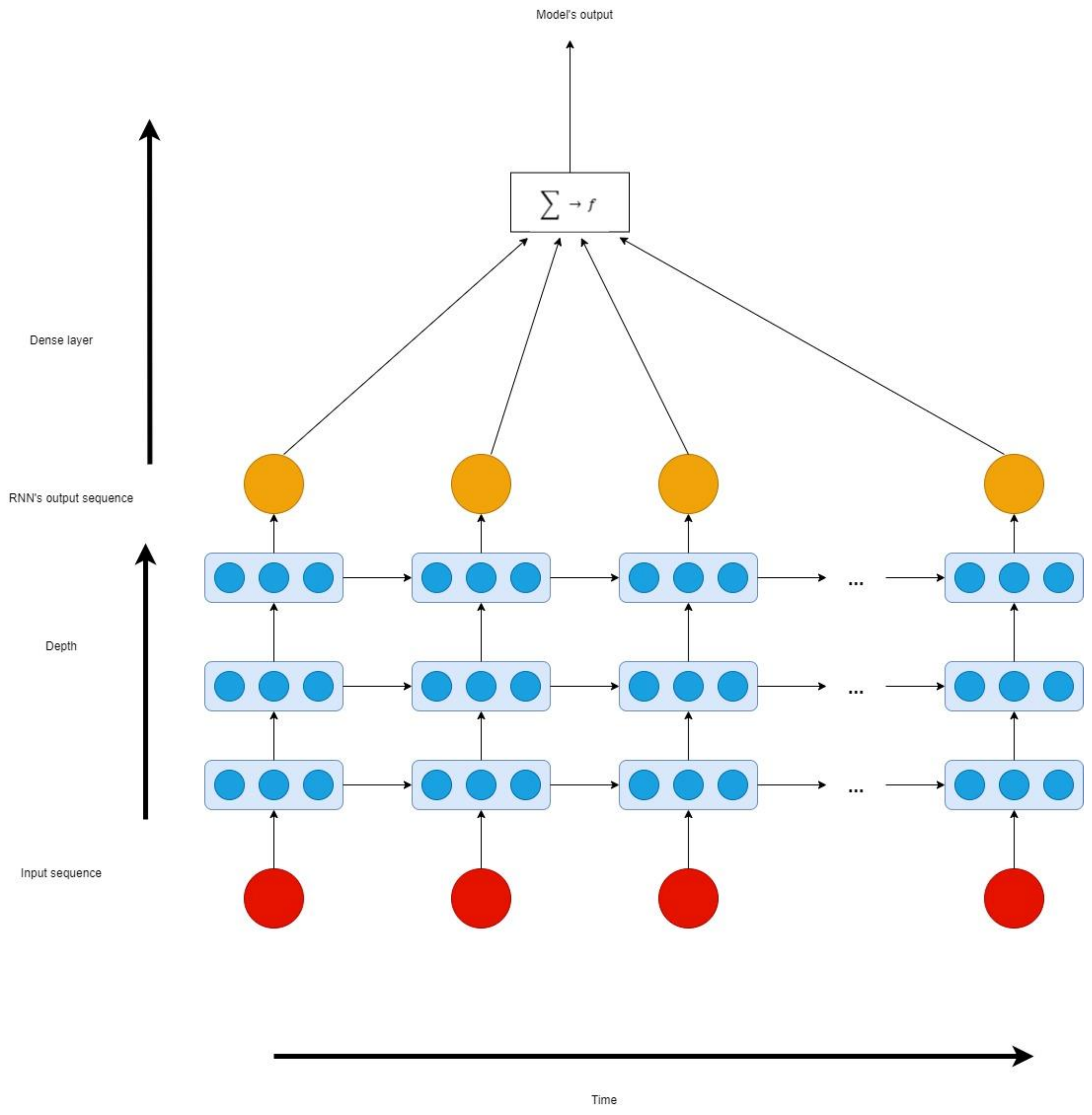

Fig. 2. Deep RNN connected to a dense layer

3) Types of sequences

Predictions made using an RNN can be of various types. Indeed, an RNN takes as input a sequence of data and produces as output a new sequence of data, but depending on the problem to which the RNN is applied, the process can fall into one of the following scenarios or the other:

- **One-to-many**

The input sequence has a length of 1, and the output sequence has a length greater than 1. A typical use case is image description. The image is first passed through another type of neural network, a Convolutional Neural Network (CNN), designed specifically to process images/photos. The CNN will output a vector that is essentially a condensed form of the information extracted from the image expressed in a latent space. This vector is the data that will be fed into the RNN to generate a sequence of words. Since the various types of architectures studied in this section can extend beyond simple RNNs, the term "model" will be used instead in the following diagrams.

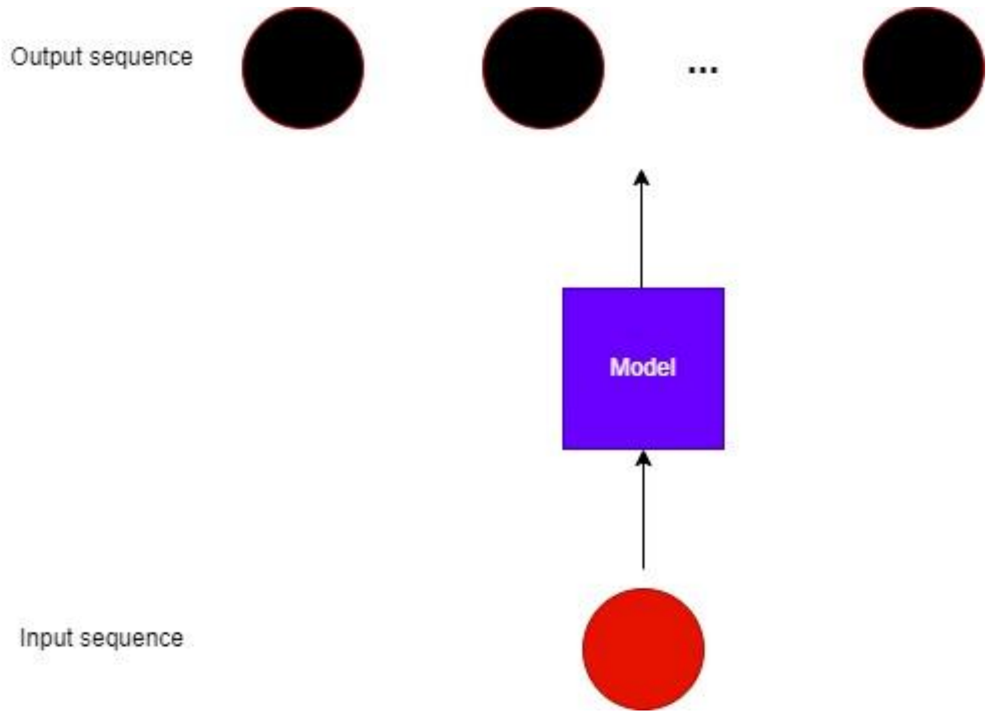

Fig. 3. One-to-many

- **Many-to-one**

The input sequence has a length greater than 1, while the output sequence of the model is of length 1. A typical use case is the classification of a textual or audio document, such as sentiment analysis in Natural Language Processing (NLP). Indeed, in the realm of NLP, we find the so-called seq2seq models composed of an encoder and a decoder, each primarily consisting of an RNN. As shown in Fig. 5, the encoder takes sequential data as input and produces as output a vector synthesizing the information contained in this sequence, which is then passed to the decoder to generate the desired output. In the case of a many-to-one architecture and sentiment analysis, the output is a label.

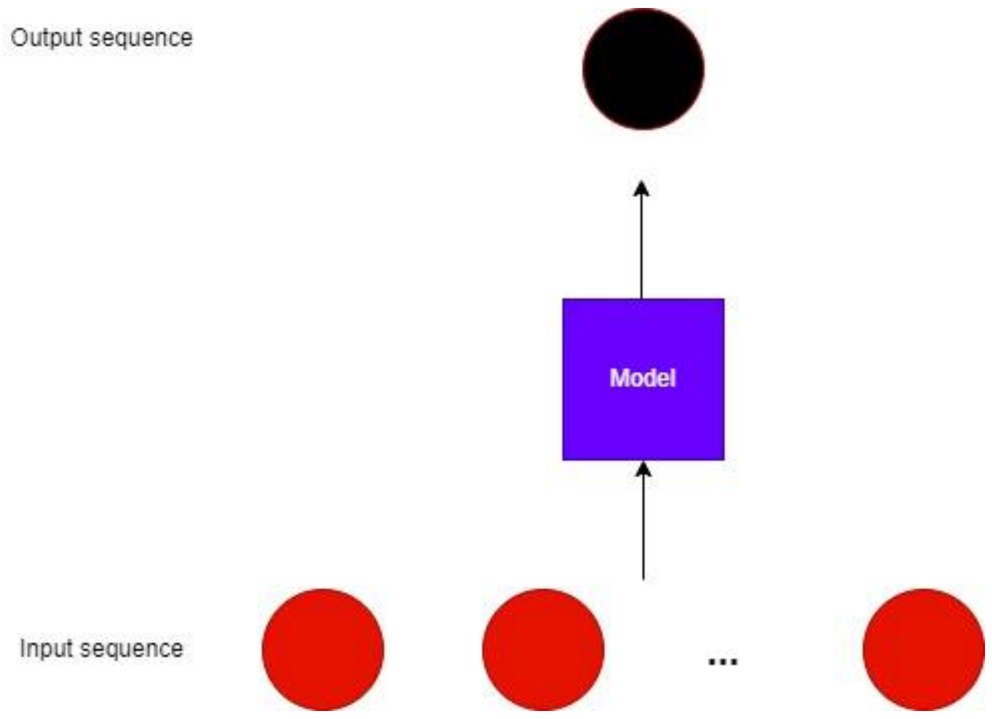

Fig. 4. Many-to-one

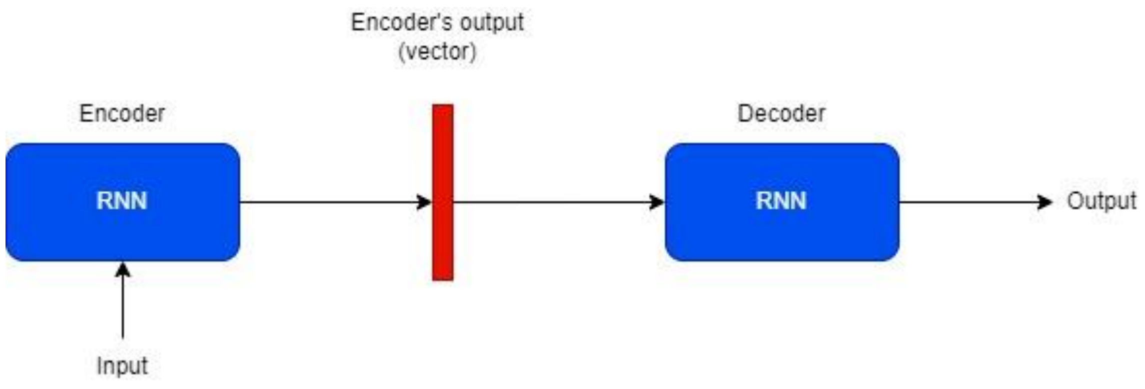

Fig. 5. Seq2seq model

- **Many-to-many**

Both the input sequence and the output sequence of the model are longer than 1. The respective lengths may be equal or different. A typical use case is the use of a conversational robot or a translation system based on the seq2seq model mentioned earlier. Naturally, one submits a question to a conversational robot or a sentence/text to be translated to the translation system, and the output will similarly be a sequence of words.

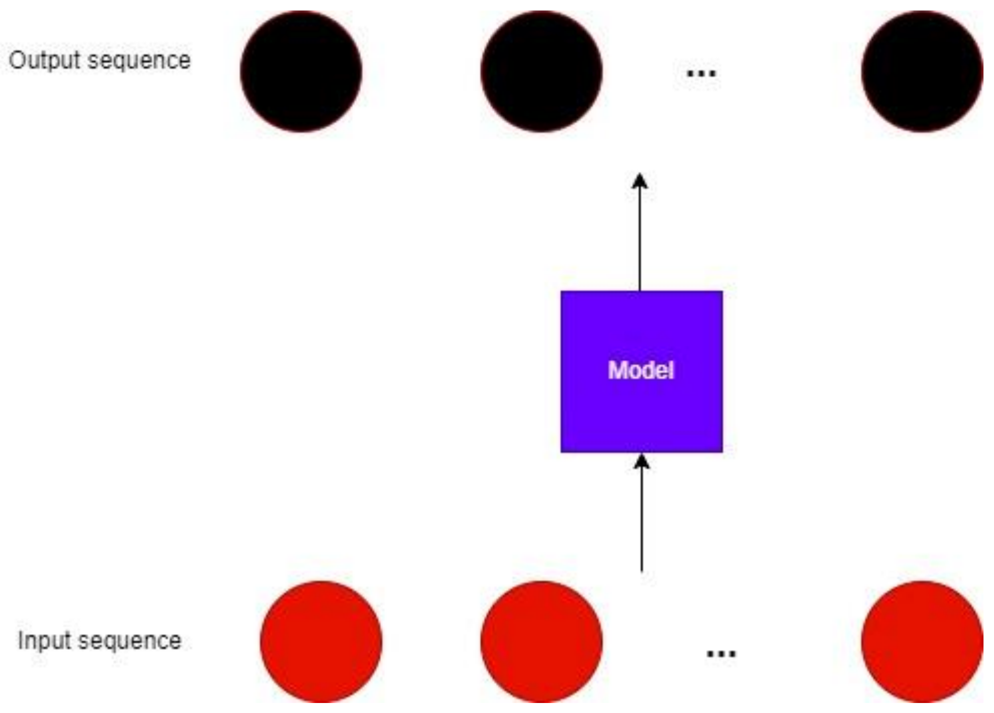

Fig. 6. Many-to-many

4) State of art

As mentioned in section 1), there are three types of cells composing RNNs, which, classified in chronological order of appearance, are vanilla RNN, LSTM, and GRU. They represent the state of the art concerning recurrent neural networks. Below is a description of their respective characteristics and architectures.

- **Vanilla RNN**

A vanilla RNN cell can be modelled as shown in Fig. 7.

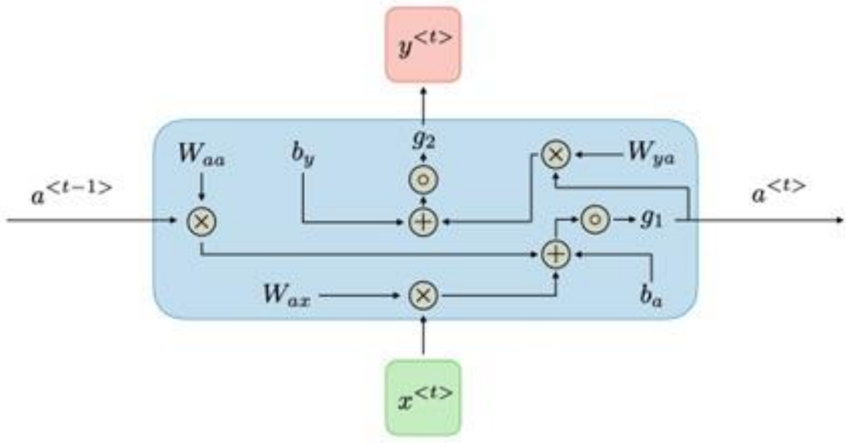

Fig. 7. Vanilla RNN
Source: https://stanford.edu/~shervine/l/fr/teaching/cs-230/pense-bete-reseaux-neurones-recurrents

W_{aa} is the weight matrix of the connections between the neurons of cell h_{t-1} corresponding to time step t-1, and those of the current cell h_t corresponding to the current time step.

W_{ax} is the weight matrix of the connections between the input $x^{<t>}$ corresponding to the current time step and the neurons of the current cell h_t. It is possible that the data in the sequence are vectors, as in the case of multivariate prediction.

W_{ya} is the weight matrix of the connections between the neurons of cell h_t and those of the cell corresponding to the same time step in the next layer.

Parameters b_a and b_y are respectively vectors of size a (number of neurons in the current cell) and size y (number of neurons in a cell of the next layer) representing biases.

Finally, g_1 and g_2 are activation functions.

The activations are obtained as follows:

$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$

$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$

As in any context of deep neural networks with an increase in the number of parameters, gradient backpropagation, which is the mechanism facilitating the computation of the gradient of the cost function with respect to the network parameters (this gradient that is then used for convergence towards optimal parameters with the gradient descent), almost inevitably leads to phenomena of gradient explosion or vanishing gradient, which in turn result in a suboptimal configuration of the model at the end of training. It is to address these issues that variants of vanilla RNNs, such as LSTMs and GRUs, have emerged.

I provide an explanation of the gradient backpropagation in the following document: https://github.com/abenslimaneakawahid/ML-and-maths-theory/blob/main/gradient_backpropagation.pdf, and of the gradient descent in this one: https://github.com/abenslimaneakawahid/iterative-methods/blob/main/Gradient_desc_opt_var_step_size.pdf.

- **LSTM**

The goal of LSTM-type architectures is, at the level of each cell, to detect the relevance of information passing between cells in the same layer and to perform sorting through different gates, which act as filters. Additionally, LSTM cells are defined by an internal state $c^{<t>}$, in addition to the external state $a^{<t>}$ already present in basic RNNs. The gates are actually vectors of size R (number of neurons in the cell + dimension of the current input) whose coefficients range from 0 to 1 and are of the form: $\Gamma = \sigma(W.[a^{<t-1>}, x^{<t>}] + b)$, where σ is a sigmoid function, and W and b are specific coefficients (model parameters) independent of time.

The gates are as follows:

- Forget gate Γ_f

  $\Gamma_f = \sigma(W_f \cdot [a^{<t-1>}, x^{<t>}] + b_f)$

- Input gate Γr:

  $\Gamma_r = \sigma(W_r \cdot [a^{<t-1>}, x^{<t>}] + b_r)$

- Output gate Γ_o

  $\Gamma_o = \sigma(W_o \cdot [a^{<t-1>}, x^{<t>}] + b_o)$

- Update gate Γ_u

  $\Gamma_u = \sigma(W_u \cdot [a^{<t-1>}, x^{<t>}] + b_u)$

An intermediate internal state c̃ is obtained from the input gate, which acts as a filter on the information transmitted by the previous cell and the current input $x^{<t>}$. In fact, a Hadamard product is applied between Γ_r and $a^{<t-1>}$, which tends to eliminate coefficients of $a^{<t-1>}$ close to 0. The rest of the operation is explained below:

$\tilde{c} = \tanh(W_{c.}[\Gamma_r * a^{<t-1>}, x^{<t>]}]+ b_c)$ with W_c and b_c as specific parameters independent of time.

To obtain the final internal state of the current cell, we apply the gate Γ_u to c̃ and the gate Γ_f to the internal state of the previous cell $c^{<t-1>,}$ and then add the results: $c^{<t>} = \Gamma_u * \tilde{c} + \Gamma_f * c^{<t-1>}$. Finally, the external state $a^{<t>}$ is obtained by taking the Hadamard product between the gate Γ_o and $c^{<t>}$:

$a^{<t>} = \Gamma_o * c^{<t>}$.

The diagram in Fig. 8 is a schematic representation of an LSTM cell.

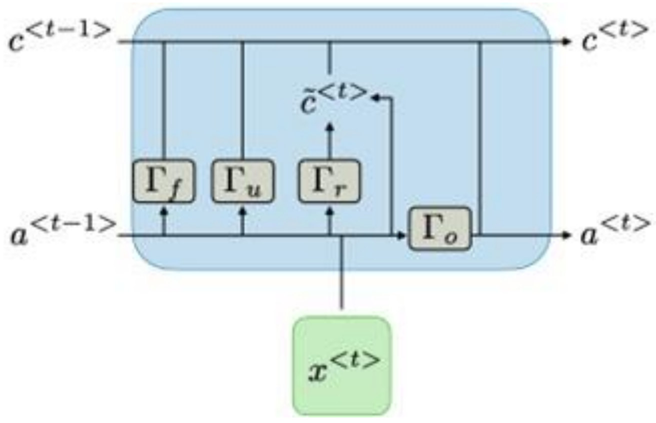

Fig. 8. LSTM
Source: https://stanford.edu/~shervine/l/fr/teaching/cs-230/pense-bete-reseaux-neurones-recurrents

- **GRU**

A GRU cell, depicted in Fig. 9, is a simplification of an LSTM cell. Indeed, the internal state and the external state are merged, $a^{<t>} = {}^{c<t>}$, and the output gate is obtained from the update gate: $\Gamma_f = 1 - \Gamma_u$. The internal state is thus obtained as follows: $c^{<t>} = \Gamma_u * \tilde{c} + (1 - \Gamma_u) * c^{<t-1>}$.

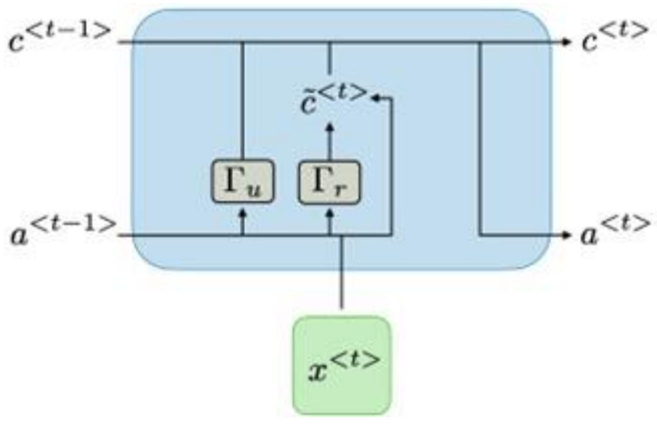

Fig. 9. GRU
Source: https://stanford.edu/~shervine/l/fr/teaching/cs-230/pense-bete-reseaux-neurones-recurrents

An advantage offered by GRUs compared to LSTMs is that they are less complex and therefore easier to implement and train.

The paper [4] provides interesting insights into the comparison between LSTM and GRU cells. The dataset used in the study consists of sequences of numerical symbols, each representing an encoded string according to a method described in the paper, details of which are not pertinent to discuss here. The exercise involves predicting the next symbol based on the input sequence. However, it is important to specify that the sequences are characterized by a certain complexity that can vary from one to another and corresponds to their compressibility. The more compressible a sequence can be summarized, the less complex it is considered, and vice versa.

The researchers, Roberto Cahuantzi, Xinye Chen, and Stefan Güttel, note: "Generally, GRUs outperformed LSTMs for low-complexity strings while LSTMs performed better on high-complexity strings. The latter finding is consistent with experiences in language modeling (typically involving very complex strings), where LSTM was also found to perform better than GRUs since it is better at capturing long-term dependencies."

In summary, LSTMs prove to be more effective on sequences with high complexity, while GRUs outperform them when the complexity is lower. By extension, this makes LSTMs more effective for natural language processing tasks.

5) Alternative approaches

It may be interesting to note that there are more traditional statistical approaches for relatively simple time series prediction, while RNNs can be used when time series are more complex/less linear and making predictions is no longer feasible with traditional methods. Among these methods, we can mention ARMA (auto-regressive moving average), ARIMA (auto-regressive integrated moving average), and SARIMA (seasonal auto-regressive integrated moving average).

The ARMA method is only suitable for stationary time series. A series is said to be stationary if its basic statistical characteristics, such as variance, mean, or covariance, are independent of time. When the series is not stationary, the ARIMA method can be used. It uses a specific operator to remove trends and make the series stationary.

In addition to trend, a time series may also exhibit seasonal patterns. For example, this is the case if we observe the daily number of flu virus contractions. Such a series will not be stationary and will show a periodic nature. To remove seasonal effects, the SARIMA method is used, which applies an additional operation to remove the seasonality phenomenon.

Also, for indicative purposes regarding natural language processing, a new type of highly efficient model emerged in 2017. This is the Transformer, a model developed by researchers from Google and the University of Toronto, introduced in a paper titled "Attention is all you need" ([5]). The Transformer no longer relies on RNNs but on the mechanism of self-attention to capture relevant information from a sequence of words. In simple terms, self-attention is a method that evaluates the asymmetric relationships between different words in a text. As an example, the famous tool ChatGPT is based on Transformer technology. The Transformer, due to its very high complexity (sometimes involving billions of parameters), is extremely resource-intensive (CPU, RAM, etc.) and data-intensive to be trained, and RNN-based models are still used in cases where these resources are lacking (e.g., in embedded systems).

I explain the Transformer model in the following document:
https://github.com/abenslimaneakawahid/generative-AI/blob/main/GenAI_simplified.pdf.

Bibliographies

1. Hopfield, J. J. (1982). Biophysics Neural networks and physical systems with emergent collective computational abilities. Proc. Natl. Acad. Sci. USA (vol. 79), 2554-2558.
2. Hochreiter, S., Schmidhuber, J. (1997). Long Short-term Memory. Neural Computation 9 (8), 1735-1780.
3. Kyunghyun, C., Van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. arXiv: 1406.1078 [cs.CL].
4. Cahuantzi, R., Chen, X., Güttel, S. (2023). A comparison of LSTM and GRU networks for learning symbolic sequences. arXiv:2107.02248v3 [cs.LG].
5. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., Polosukhin, I. (2017). Attention Is All You Need. arXiv:1706.03762 [cs.CL].