ecnológico
Monterrey

# NLP 2

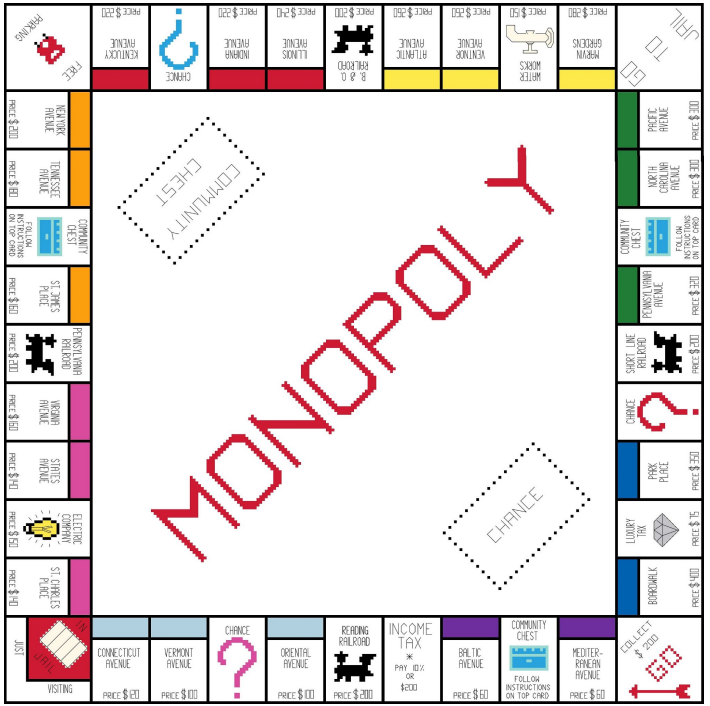
Inteligencia artificial avanzada para la ciencia de datos II Modulo 5 NLP 2

RNN

# Markov decision process (MDP)

A Markov Decision Process, or MDP is a mathematical framework we use to make **decisions in situations where there's some randomness involved.**

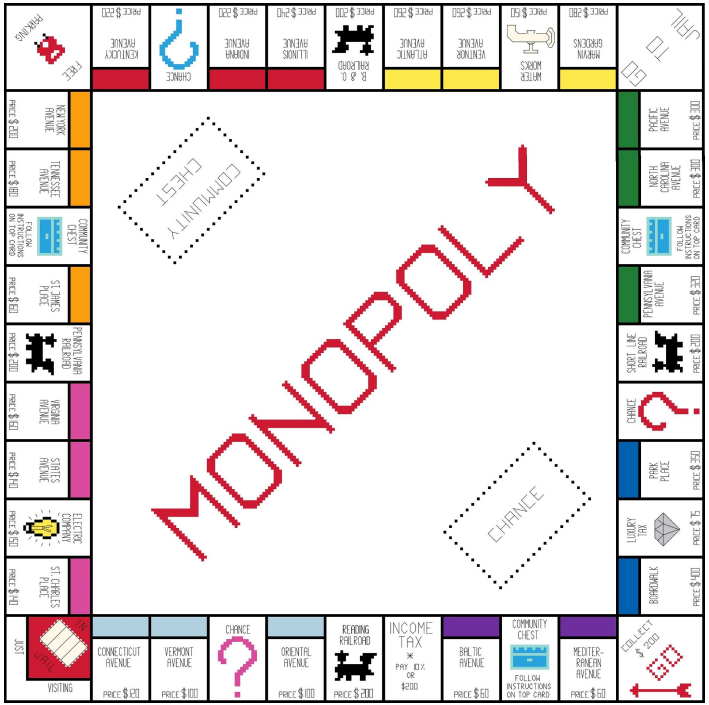
Imagine you're playing a game or trying to solve a problem, and at each step, you have multiple choices to make. However, you can't predict exactly what will happen next because there's an element of chance or randomness.



# Probability strategy

1. **State**: At any given moment, you're in a certain "state" or situation.
2. **Action**: You have a set of actions you can take in each state. These are like your choices or decisions.
3. **Reward**: After taking an action, you get some kind of "reward." This reward could be positive if you did something good or negative if you did something bad. It tells you how well you're doing.
4. **Transition Probability**: There's a chance that when you take an action in a state, you might end up in a different state. This is where the randomness comes in. The MDP tells you the probability of going from one state to another when you take a certain action.

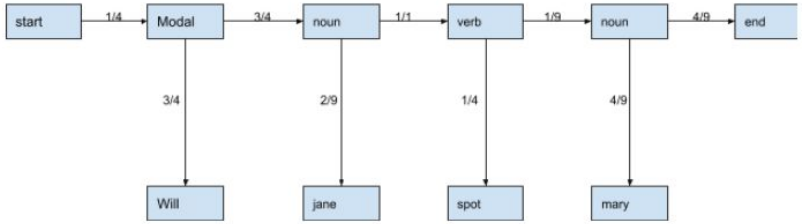
The goal of an MDP is to find a strategy or a set of rules that tells you which actions to take in each state to maximize the total rewards you'll get over time. This strategy is called a "policy."



# Text generation Markov

Predicting text is often considered a **Markovian problem** because it's based on the assumption that the future state (the next word or character) **depends primarily on the current state** (the words or characters that came before it) and not on the entire history of past states. This assumption simplifies the modeling and prediction process.

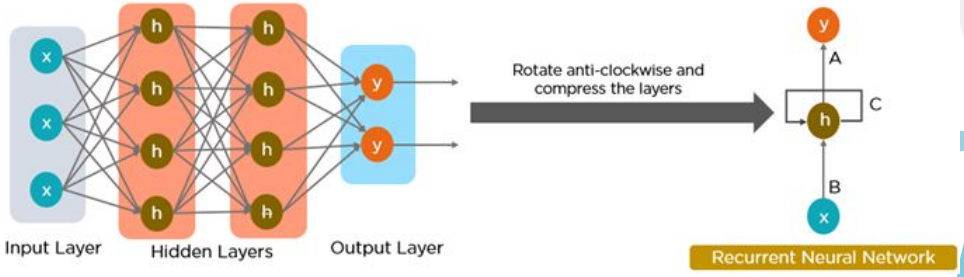
This assumption of "memorylessness" or the Markov property simplifies the problem significantly, is there a way to give **memory** to models?



# What is a RNN ?

A Recurrent Neural Network (**RNN**) is a type of artificial neural network designed for processing sequences of data.

Unlike traditional feedforward neural networks, which process data in a fixed and static manner, RNNs are capable of handling sequences of varying lengths by maintaining an internal hidden state that **captures information about previous elements in the sequence,** this is called the memory component.



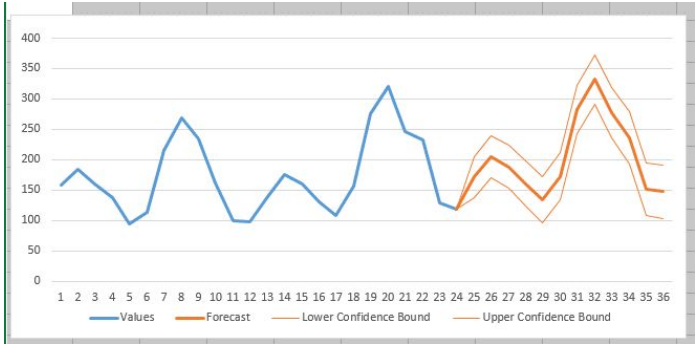
# Sequence length problem

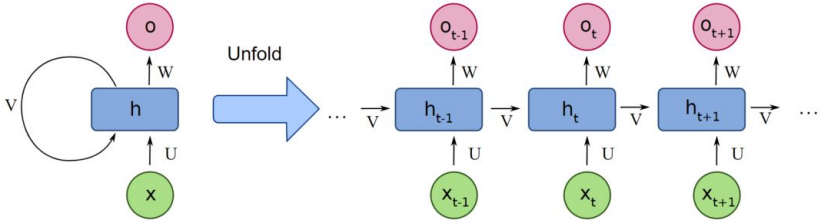
Some sequence models are traditionally used to forecast or predict in time series, for example looking the trade market uses a f**ix window size** of history to predict.

But text as a sequence usually is **not fix length size** , for example:

**“Hello, my name is …”**
**“Hi, I’m …”**

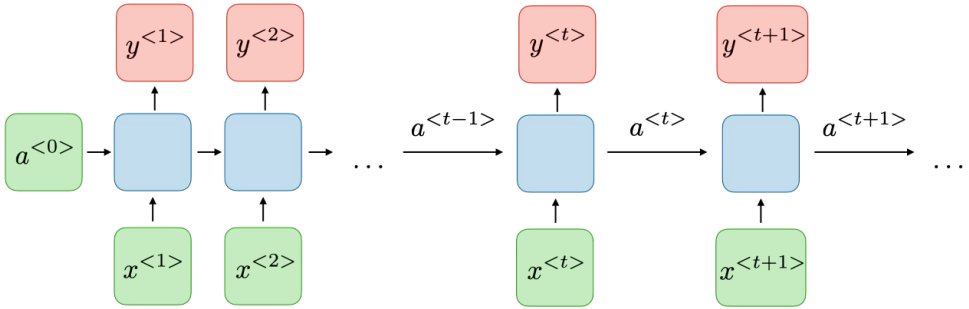
Must produce a similar prediction, RNN solves this by using a l**oop structure**, every token produces a new state (layer).



# RNN architecture

In a traditional architecture, sequence events **X** (tokens) are used as input (one by one) and will product an output **y** like in a FFN. But look to the **a** element, this is also an input in every iteration, and this is how we take into account the hole history of events.

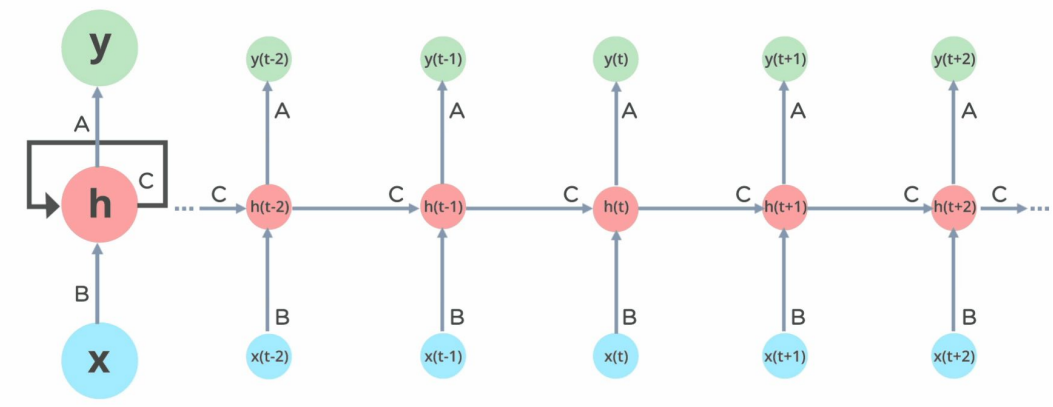



For each timestep $t$, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$\boxed{a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)} \quad \text{and} \quad \boxed{y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)}$$

where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and $g_1, g_2$ activation functions.

# How it works




Every event in the sequence will prepare context as future input and prepare an output in case this is the last event.

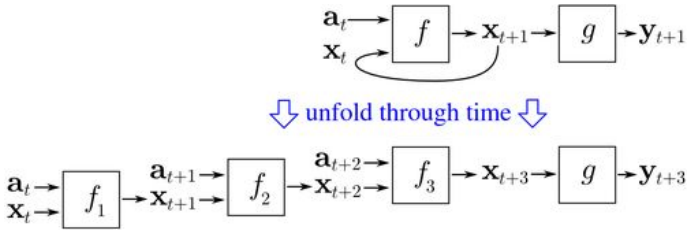
At the end the output is **y(N)** where **N** is the number of tokens in the event.

Also, there is not a log of events, in a discrete way RNN use **recurrency** and do not store anything in memory.

# Backpropagation Through Time (BPTT)

let's say you're using an RNN to process a sentence. You show it one word at a time and let it make predictions about what the next word might be. If it makes a mistake (like guessing the wrong word), you want to help it learn from that mistake.

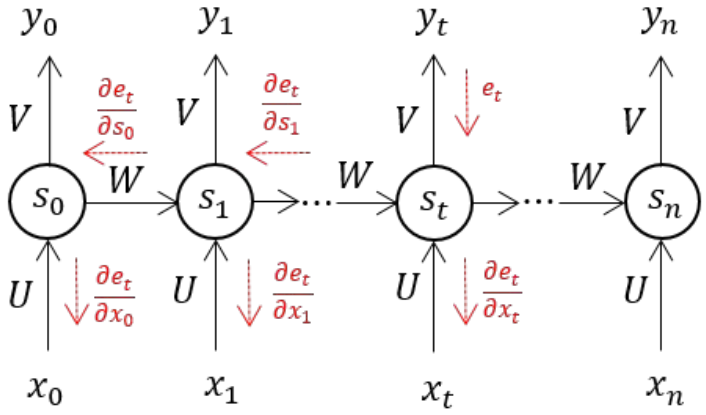
**Breaking Down Time Steps**: BPTT breaks down this sequence into individual time steps. It's like looking at each word in the sentence one after the other. At each time step, the RNN makes a prediction based on the current word and what it "remembers" from the past words.



# Backpropagation Through Time (BPTT)

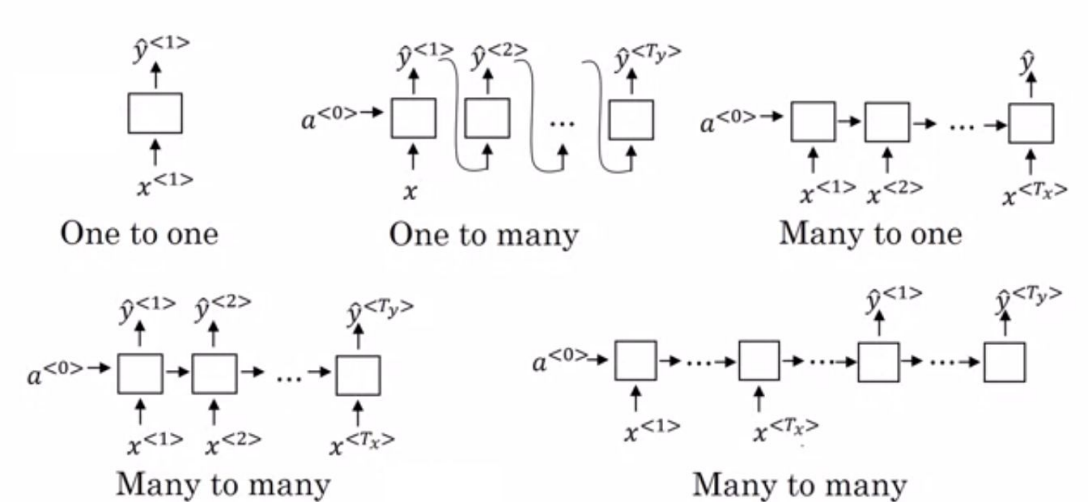
When the RNN makes a mistake (like guessing the wrong word), BPTT calculates how much that mistake contributed to the overall error. It's a bit like figuring out how wrong the RNN was for that specific word.

**Learning from Mistakes**: After looking at all the words in the sentence, BPTT adds up all these little mistakes and tells the RNN how much it needs to adjust its "knowledge" (the internal parameters) so that it makes better predictions in the future.



# RNN variations

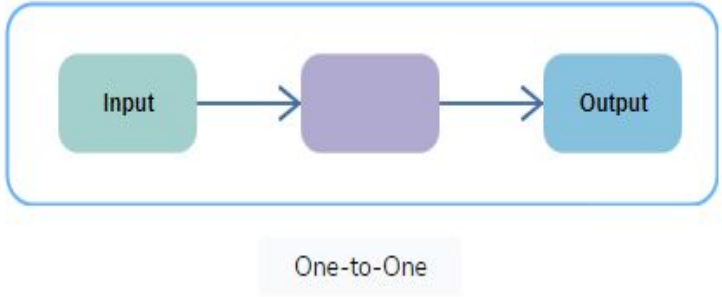
RNN where designed to solve a sequence prediction problem (guess next word), however there are some variations form original one.

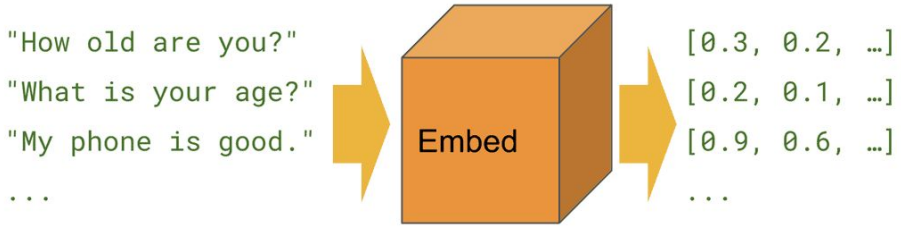


# One to one

Although you might find in literature as type of RNN, this is not recurrent, it only takes one input and produces an output like traditional NN.

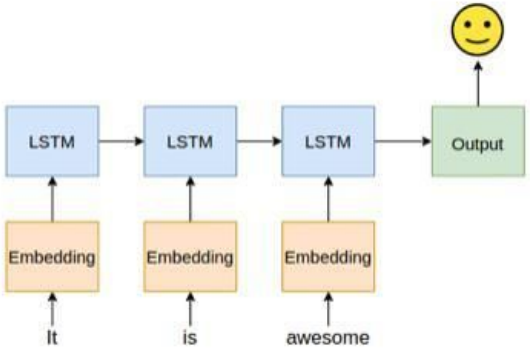
For example, a token classification model or a sentence embedding encoder.

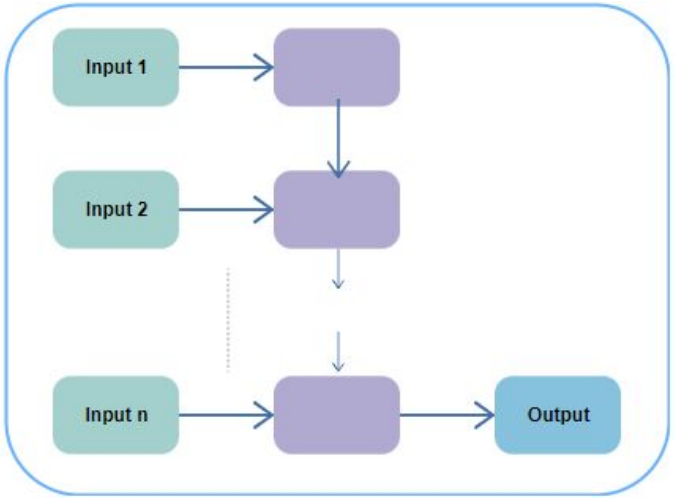


# Many to one

A many-to-one RNN, multiple inputs (a sequence) are used to produce a single output.
This is common in sentiment analysis, where a sequence of words (a sentence) is processed to predict the sentiment of the entire sentence.



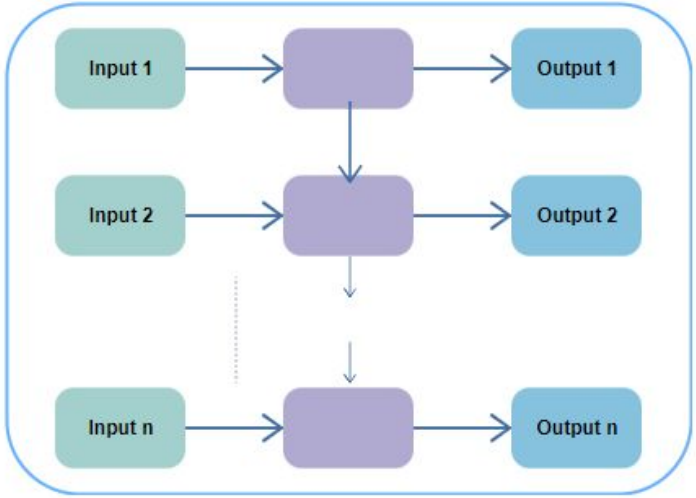



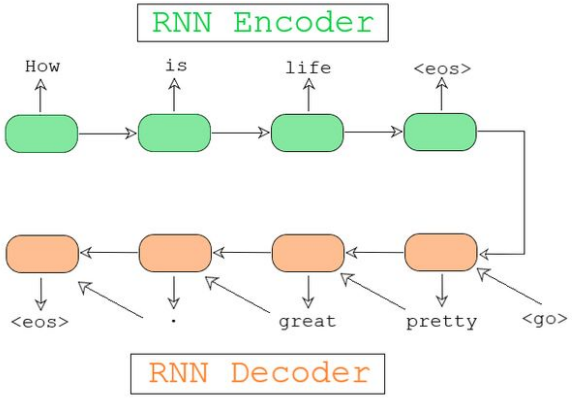

Many-to-One

# Many to Many (N:N)

In a many-to-many RNN, there is a sequence of inputs and a sequence of outputs, and they are aligned in time.

This architecture is used for tasks like sequence-to-sequence translation, where a sequence of words in one language is translated into a sequence of words in another language.

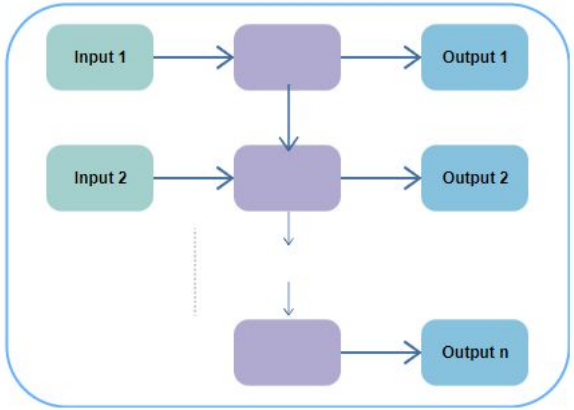


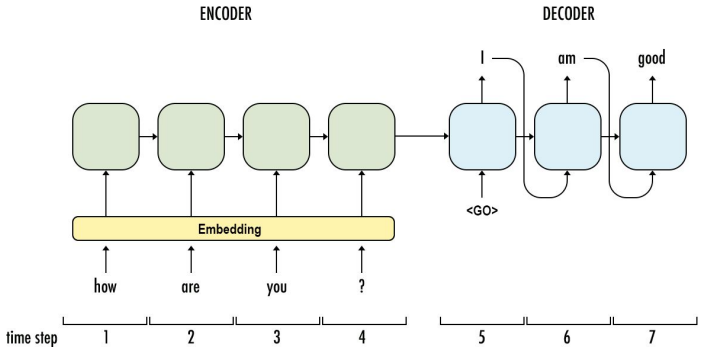



# Many to Many (N:M)

In this variation of many-to-many, the input and output sequences can have different lengths, and they may not be aligned in time.

This is useful for tasks like speech recognition, where the length of the spoken input may not match the length of the recognized text output.

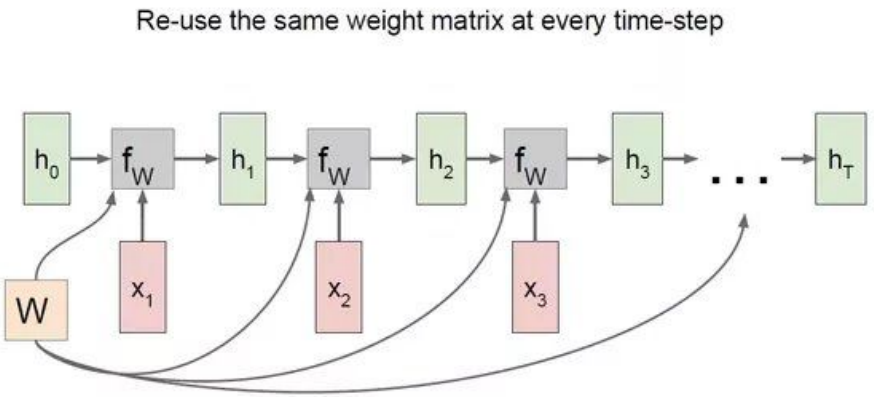


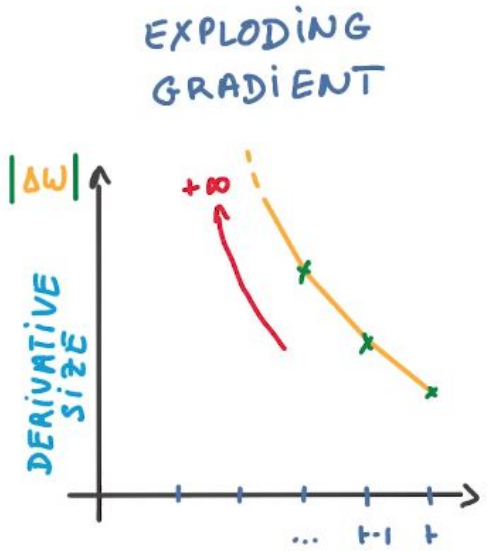



# Pros vs Cons

| Benefit | Limitations |
| --- | --- |
| • Processes sequential data<br>• Can memorize and store previous results<br>• Takes into account both the current and the previous results in the computation of new results<br>• Regardless of the increasing size of the input, the model size remains fixed<br>• It shares weights to other units across time | • The computation time is slow as it is recurrent.<br>• Cannot process future data in computation of current data.<br>• Training is complicated.<br>• **Exploding Gradient**: An exponential increase in model weights occur due to an accumulation of large gradient errors.<br>• **Vanishing Gradient:** The gradients become too small and unable to make significant changes in the model weights. |

# Exploding Gradient

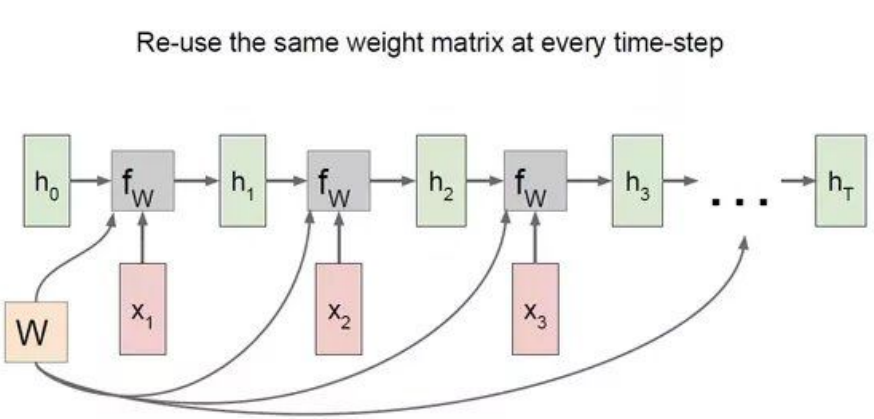


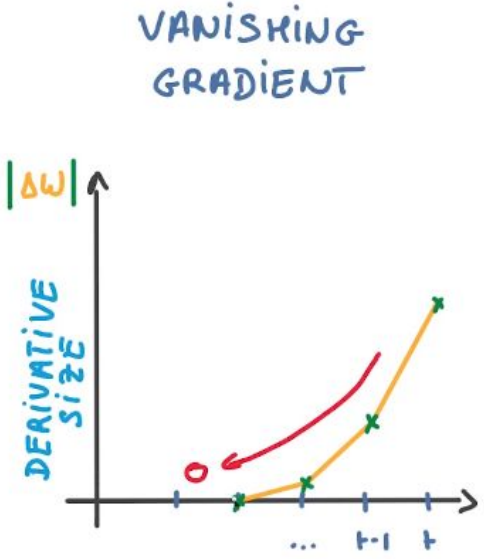




As we have discuss RNN have memory from past events and reuses weights in the network, if the weight has a tendency of going up (values greater than 1), outputs will increase always !!!

**Solution: never let W goes greater than 1**

# Vanishing Gradient




As an opposite version if weight tendency decreases (values lower than 1) outputs will decrease.

**Solution: LSTM model .. but we will review it next session ...**

# Thanks

Do you have any questions?

emmanuel.paez@tec.mx
Slack #module-5-nlp-1