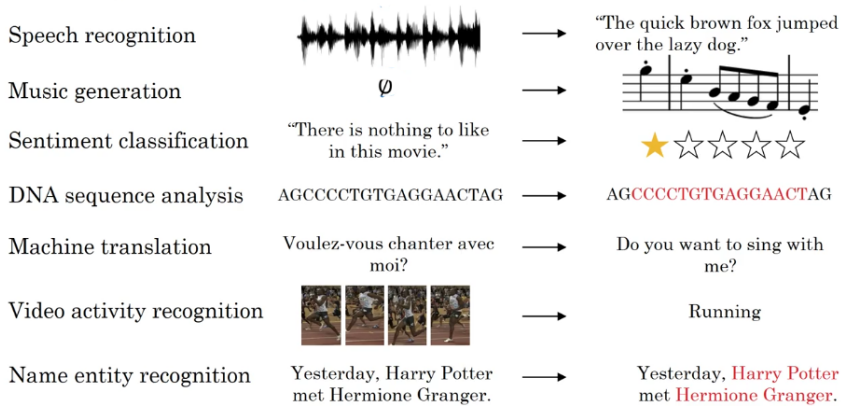
Notes for Lab 5: Short Intro to RNNs

Centrale Supelec - DTY

Sequences

Sequences are 1D data that come in form of a sequence, like words and speech.



Let's take for an example *Named Entity Recognition.* Where we want to determine the names present is the input phrases (persons names, companies names...), the inputs are the words (each one is indexed in the input sequence) with a binary ouput to tell if the word is a name or not.

x:	Harry	Potter	and	Hermione	Granfer	invented	a	new	spell.
	x<1>	x<2>	x<3>	x<4>	x<5>	x<6>	x<7>	x<8>	x<9>
y:	1	1	0	1	1	0	0	0	0

Notations The t-th element of sequence of the input i is indexed as: $X^{(i)<t>}$, and the length of each input i is written as: $T_x^{(i)}$. Same for the ouputs, the t-th element of sequence of the ouptut i is $Y^{(i)<t>}$, and the output sequence length is $T_y^{(i)}$.

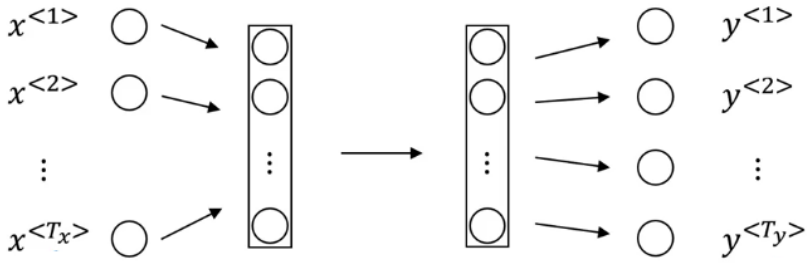
So, in summary:

- Superscript $[l]$ denotes an object associated with the l^{th} layer.
  Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.
- Superscript (i) denotes an object associated with the i^{th} example.
  Example: $x^{(i)}$ is the i^{th} training example input.
- Superscript $\langle t \rangle$ denotes an object at the t^{th} time-step.
  Example: $x^{\langle t \rangle}$ is the input x at the t^{th} time-step. $x^{(i)\langle t \rangle}$ is the input at the t^{th} timestep of example i.
- Lowerscript i denotes the i^{th} entry of a vector.
  Example: $a_i^{[l]}$ denotes the i^{th} entry of the activations in layer l.

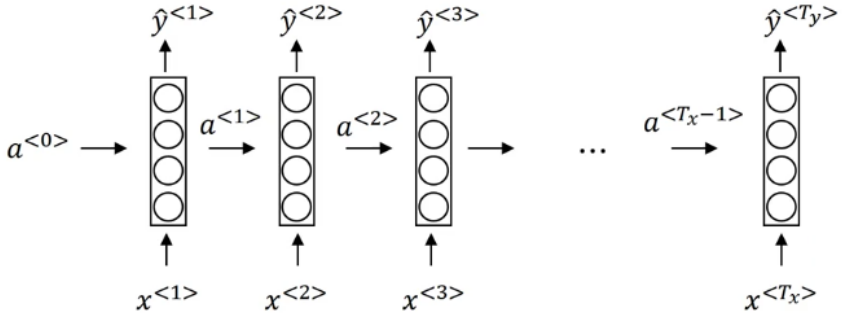
Recurrent neural nets

We could use a standard network, taking as input T_x words and outputing T_y outputs, but the problems with such approach:

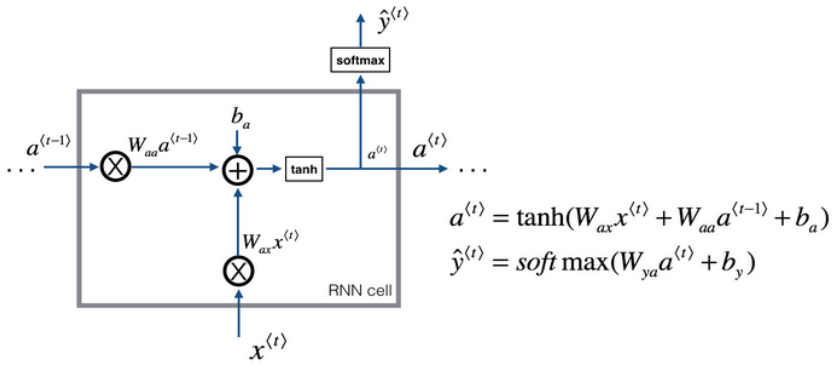
- Inputs and ouputs can be of different lengths in different examples.
- Doesn't share features learned across different positions of text (the same word reappering in a different position means the word is also a person's name).
- The length of the inputs is quite big, depending on the size of the dictionary, so the network will have an enormous number of parameters.



To take into account the nature of sequence data, RNNs use recurrent connections to take as inputs, in addition to the current word in the input sequence, the state at the previous time step. With two weight matrices, W_{ax} (input $\rightarrow$ hidden state) and W_{aa} (previous hidden state $\rightarrow$ current one) and W_{ya} (hidden state $\rightarrow$ output), we generally start the previous hidden state at t = 0 with zeros.



A Recurrent neural network can also be seen as the repetition of a single cell. The following figure describes the operations for a single time-step of an RNN cell.



Side note Sometimes, the network might also need to take into account not only the previous words but also the next words in the sequence for better predictions, per example; "He said, Teddy Roosevelt was a great president", to be able to predict Teddy as a name, having both the previous words and Roosevelt will given the network a better change to classify it as a name. This implemented in Bidirectionnal RNNs.

Forward propagation

$$a^{<t>} = g\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right)$$
$$\hat{y}^{<t>} = g\left(W_{ya}a^{<t>} + b_y\right)$$

To simplify the notation:

$$a^{<t>} = g\left(w_a\left[a^{(t-1)}, x^{(t)}\right] + b_a\right)$$
$$\hat{y}^{<t>} = g\left(W_y a^{<t>} + b_y\right)$$

For a 10,000 input vector and a hidden state of size 100, W_{ax} is of size (10,000 x 100) and W_{aa} is of size (100 x 100), we denote by $W_a = [W_{ax}|W_{ax}]$, a matrix of size (100 x 10,100) where we stack both matrices W_{aa} and W_{ax}. and $[a_{t-1}, x_t]$ as a vector of size 10,100.

Back propagation through time (BPTT)

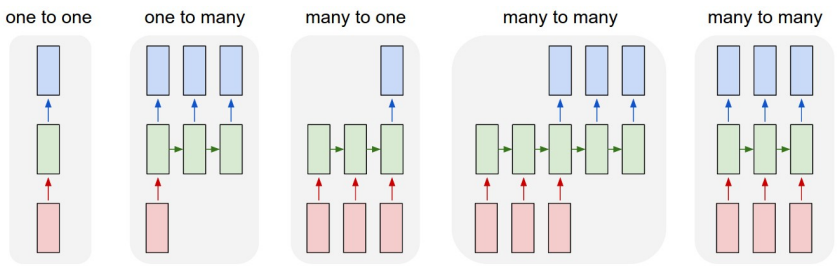
In general :

- First, present the input pattern and propagate it through the network to get the output.
- Then compare the predicted output to the expected output and calculate the error.
- Then calculate the derivates of the error with respect to the network weights
- Try to adjust the weights so that the error is minimum.

The Backpropagation Through Time is the application of Backpropagation training algorithm which is applied to the sequence data like the time series. It is applied to the recurrent neural network. The recurrent neural network is shown one input each timestep and predicts the corresponding output. So, we can say that BTPP works by unrolling all input timesteps. Each timestep has one input time step, one output time step and one copy of the network. Then the errors are calculated and accumulated for each timestep. The network is then rolled back to update the weights.

Types of RNNs

In most cases, the size of the input sequence T_x is different than the ouput sequence T_y, and depending on T_x and T_y we can end up with different types of RNN structures:



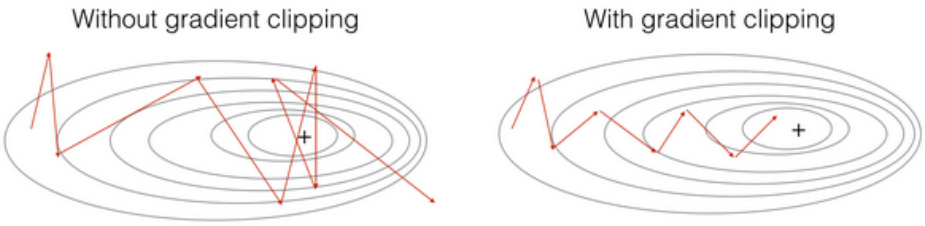
- Many to many: both the input and output are sequences and Tx = Ty.
- Many to many: both the input and output are sequences but this time with different lengths, like Machine translation, with two distinct part, an encoder and a decoder.
- Many to one: In sentiment analysis for example, where we input a sequence and ouput a sentiment, with a prediction as integer between [0, 5] on how positive or negative the input text is.
- One to one: ordinary nets, like feed forward nets.
- One to many: Like music generation, where we input one note and the network starts generating music notes, and reuse the outputs as net inputs to generate further new music.

Vanishing and exploding gradients

One of the problem with vanilla RNNs is the vanashing and exploding gradients. In language we can have very long dependencies, like in the sentence "the *cat/cats* with .. ate ..... *was / were*", depending on cat being plural or singular, the network needs to ouput the correct prediction later in the sequence.

To learn such dependecies, the gradient from (was/were) needs to backpropagate over large number of steps to affect the earlier layers and modify how the RNN do computation in these layers. And with the problem of vanashing and exploding gradients, this becomes very unlikely, so in RNNs we only have local dependencies, where each word only depends on a limited number of words proceding it.

Gradient clipping: For exploding gradient we can apply gradient clipping. The overall loop structure usually consists of a forward pass, a cost computation, a backward pass, and a parameter update. Before updating the parameters, we perform gradient clipping when needed to make sure that the gradients are not "exploding," meaning taking on overly large values, but for vanashing gradients is much harder to solve.



Gated Recurrent Unit (GRU)

One solution to the vanashing gradient problem is to use GRU unit instead. GRUs will have a new cell C called memory cell, providing a bit of memory to remeber if the car was singular or plural, $c^{<t>}$ will output an activation $a^{<t>}$, at every time step. We'll consider writing $\tilde{C}^{<t>}$ to the memory cell, and the important idea of GRU, is having a gate Γ_u between zero and one (zero or one most of the time), and the gate will decide if we update the memory cell with the candidate $\tilde{C}^{<t>}$ or not.

We also have another gate Γ_r which tells us how relative is the sate of the memory cell to compute the new condidate for the memory cell.

$$
\begin{aligned}
\tilde{c}^{<t>} &= \tanh\left(W_c\left[\Gamma_r * c^{<t-1>}, x^{<t>}\right] + b_c\right) \\
\Gamma_u &= \sigma\left(W_u\left[c^{<t-1>}, x^{<t>}\right] + b_u\right) \\
\Gamma_r &= \sigma\left(W_r\left[c^{<t-1>}, x^{<t>}\right] + b_r\right) \\
c^{<t>} &= \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>} \\
a^{<t>} &= c^{<t>}
\end{aligned}
$$

The cell memory cell and the gate have the same dimension as the state of the hidden layer (element wise multiplication in the updating step). In our example, after the network output was/were depending on the state of cat, we can then update the state of the memory cell.

We can also use LSTMs, for more details please refer to this post Understanding LSTM Networks