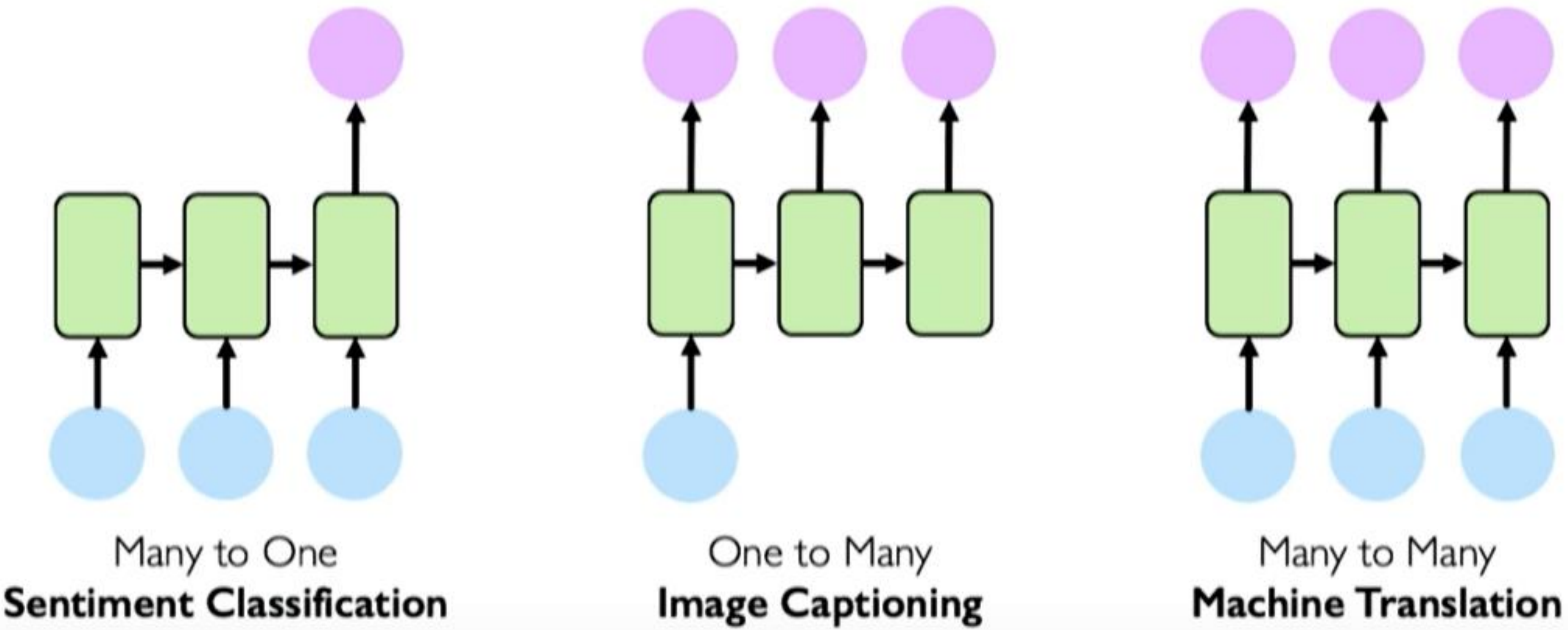
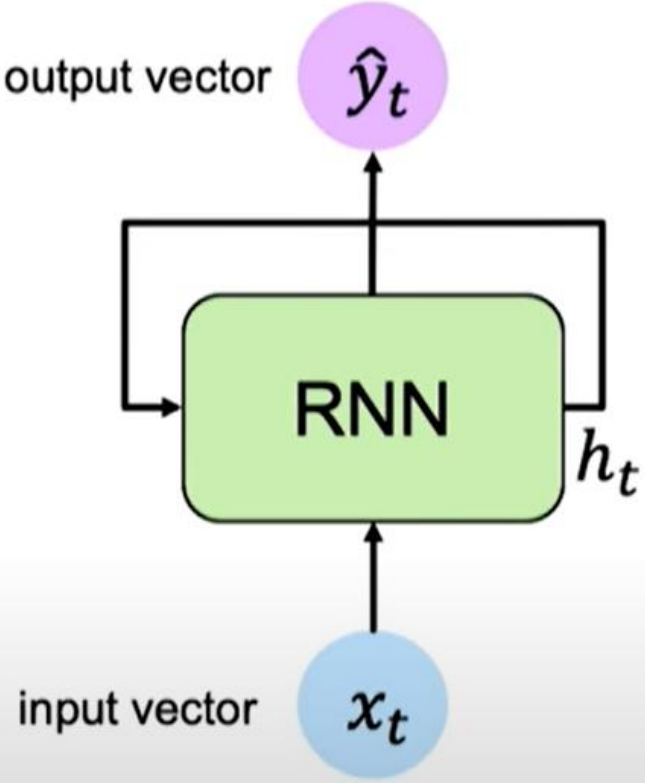
Les réseaux de neurones récurrents

Sonia Gharsalli

Applications





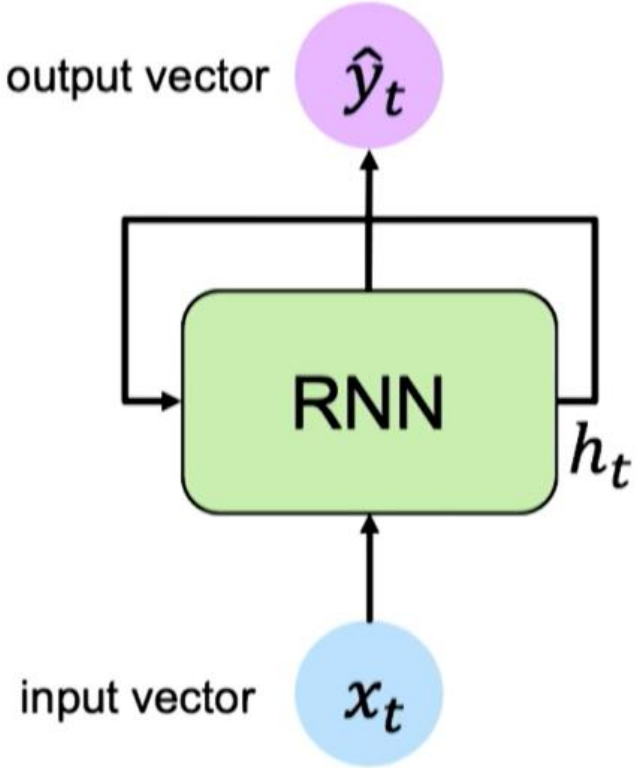
Apply a **recurrence relation** at every time step to process a sequence:

$$\underbrace{\boxed{h_t}}_{\text{cell state}} = \underbrace{\boxed{f_W}}_{\substack{\text{function}\\ \text{with weights}\\ \text{W}}}(\underbrace{\boxed{x_t}}_{\text{input}}, \underbrace{\boxed{h_{t-1}}}_{\text{old state}})$$

Note: the same function and set of parameters are used at every time step

code RNN

```
my_rnn = RNN()

hidden_state = [0,0,0]

sentence = [“i”, “love”, “NLP”]

for word in sentence:

        prediction,hidden_state = my_rnn(word, hidden_state)

next_word_prediction = prediction
```



Output Vector

$$\hat{y}_t = \boldsymbol{W}_{\boldsymbol{hy}}^{\boldsymbol{T}} h_t$$

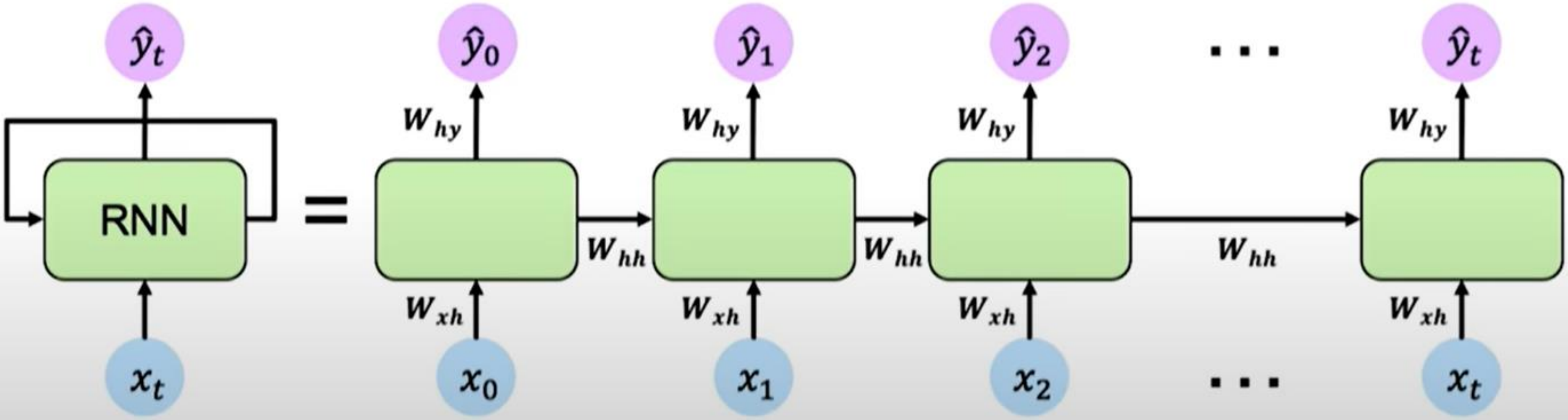
Update Hidden State

$$h_t = \tanh(\boldsymbol{W}_{\boldsymbol{hh}}^{\boldsymbol{T}} h_{t-1} + \boldsymbol{W}_{\boldsymbol{xh}}^{\boldsymbol{T}} x_t)$$

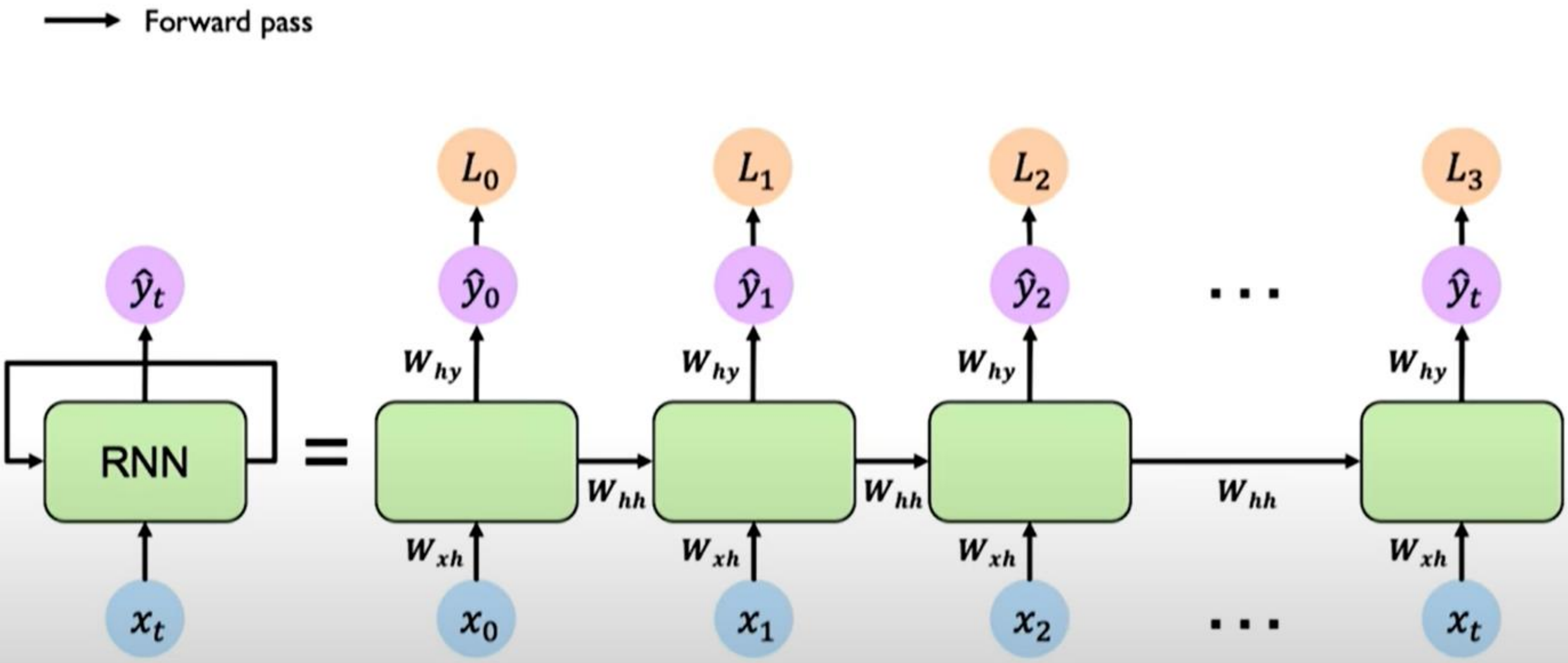
Input Vector

$$x_t$$

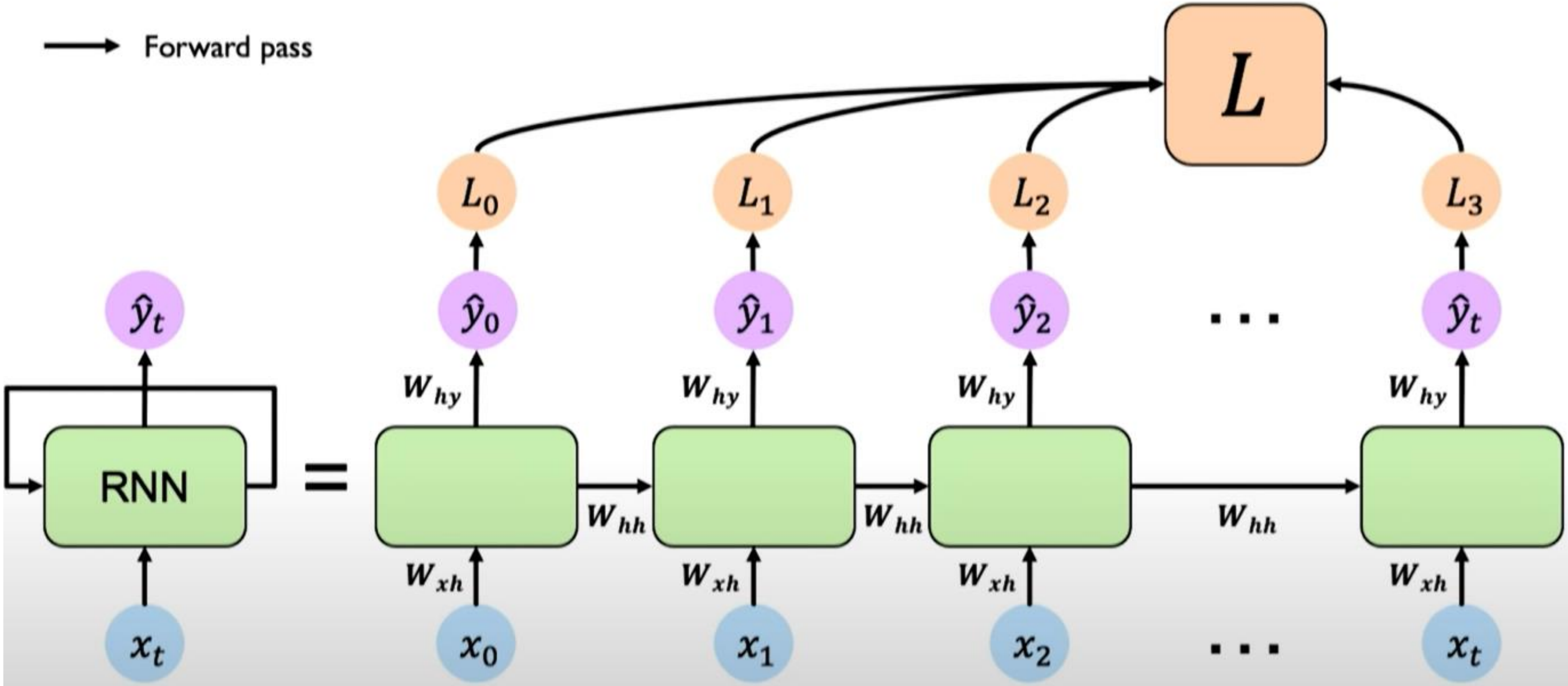
Mise à jour des états de RNN et de la sortie



On calcule la perte à chaque instant (t0, t1, t2, …)

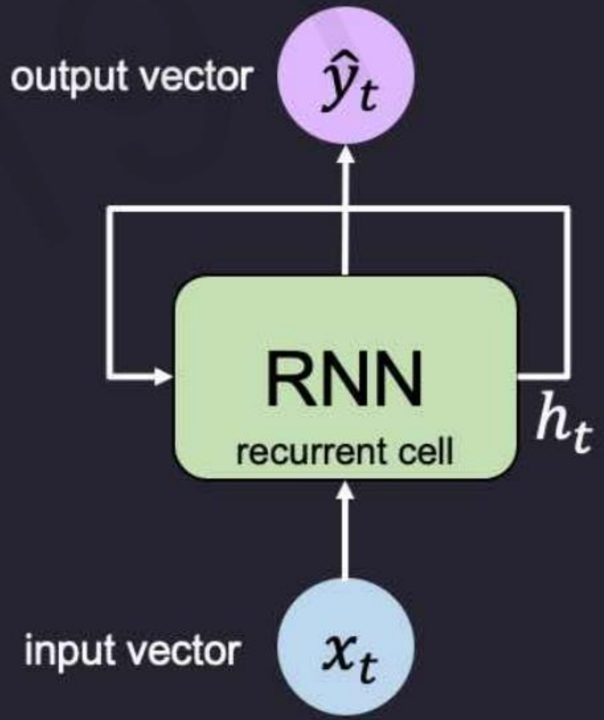


On somme toutes les pertes pour avoir un coût totale



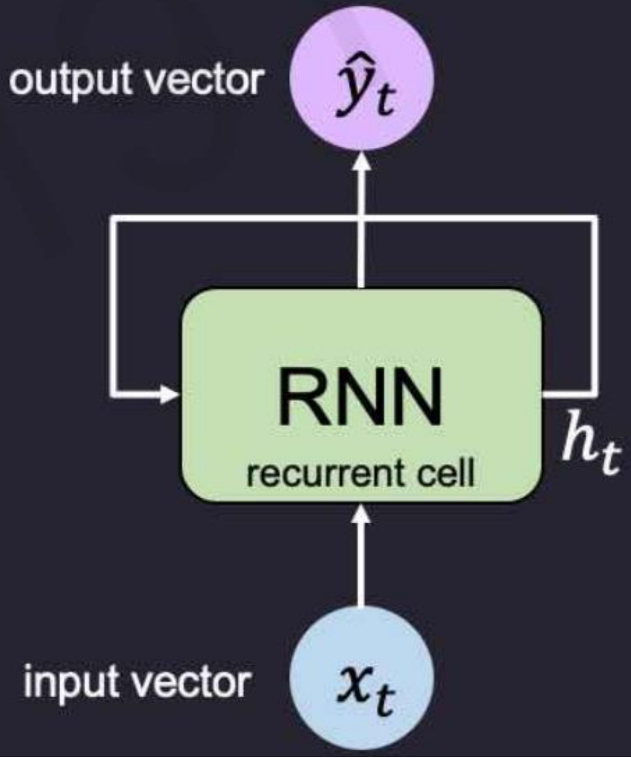
RNNs from Scratch

```python
class MyRNNCell(tf.keras.layers.Layer):
  def __init__(self, rnn_units, input_dim, output_dim):
    super(MyRNNCell, self).__init__()

    # Initialize weight matrices
    self.W_xh = self.add_weight([rnn_units, input_dim])
    self.W_hh = self.add_weight([rnn_units, rnn_units])
    self.W_hy = self.add_weight([output_dim, rnn_units])

    # Initialize hidden state to zeros
    self.h = tf.zeros([rnn_units, 1])

  def call(self, x):
    # Update the hidden state
    self.h = tf.math.tanh( self.W_hh * self.h + self.W_xh * x )

    # Compute the output
    output = self.W_hy * self.h

    # Return the current output and hidden state
    return output, self.h
```



RNN Implementation in TensorFlow

```
tf.keras.layers.SimpleRNN(rnn_units)
```

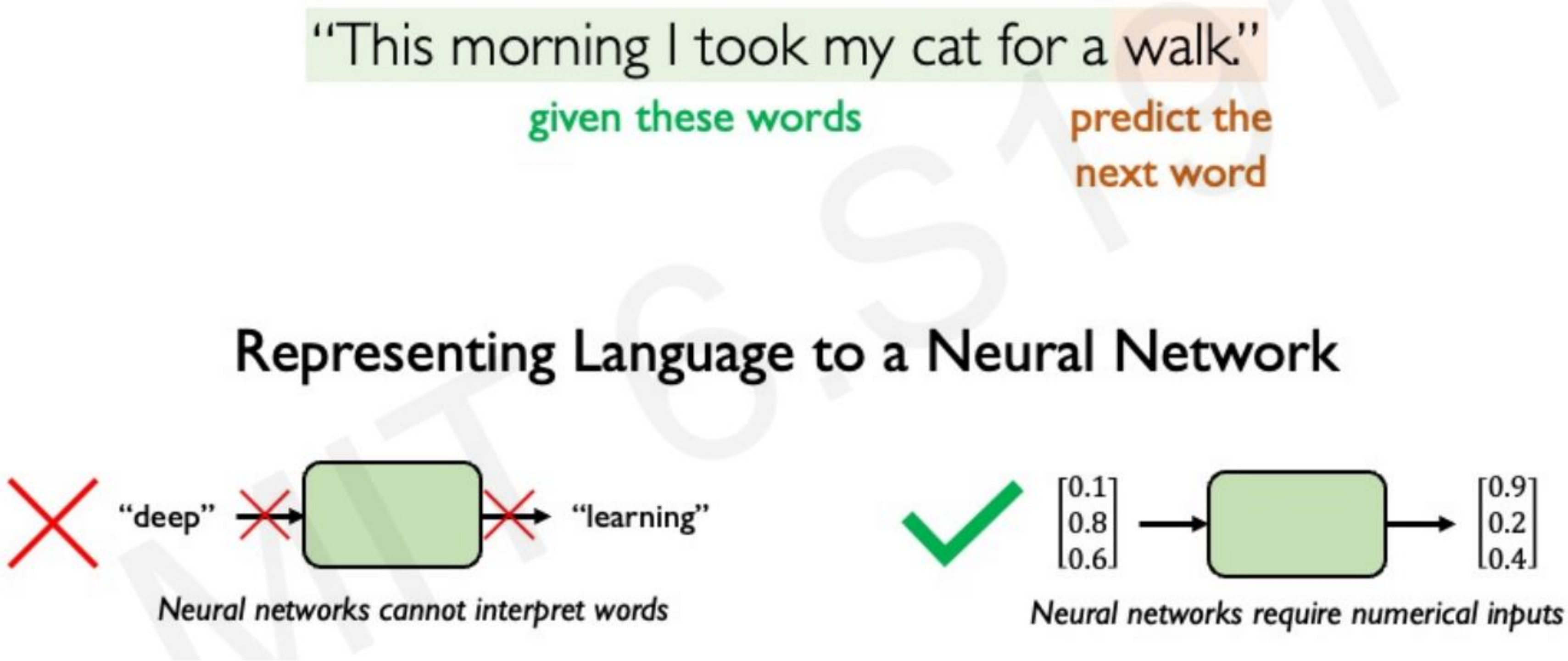


Les critères de conception

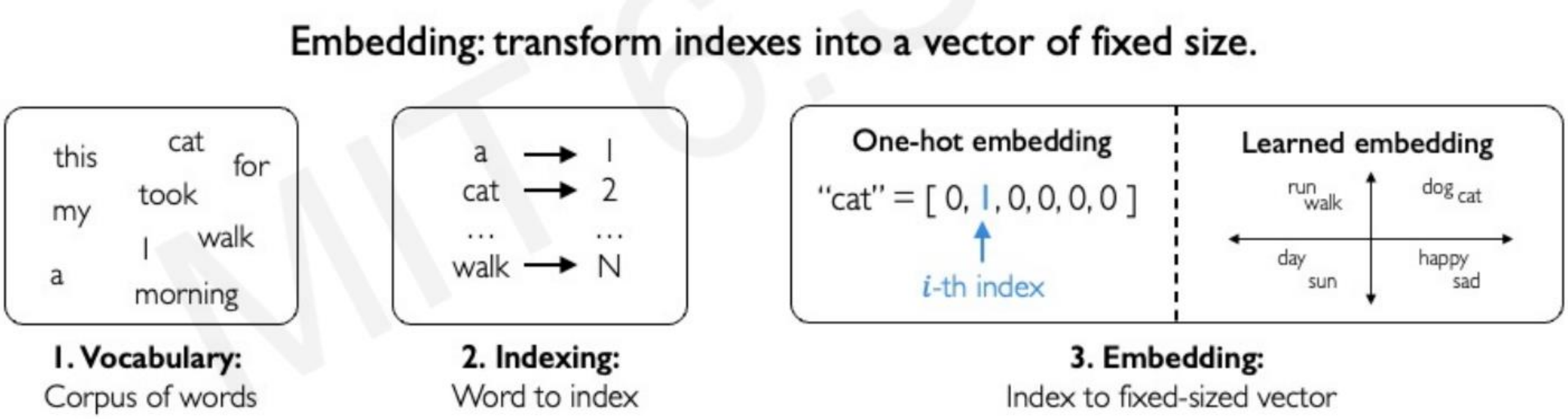
Les modèles de séquence doivent :

- Traiter les séquences de longueur variable.
- Prendre en considération les dépendances qui apparaissent loin dans le texte.
- Détecter l’ordre d’apparition des termes

Problème de prédiction du terme suivant:



Représentation vectorielle des termes:



Traiter des séquences de longueur variable

The food was great

vs.

We visited a restaurant for lunch

vs.

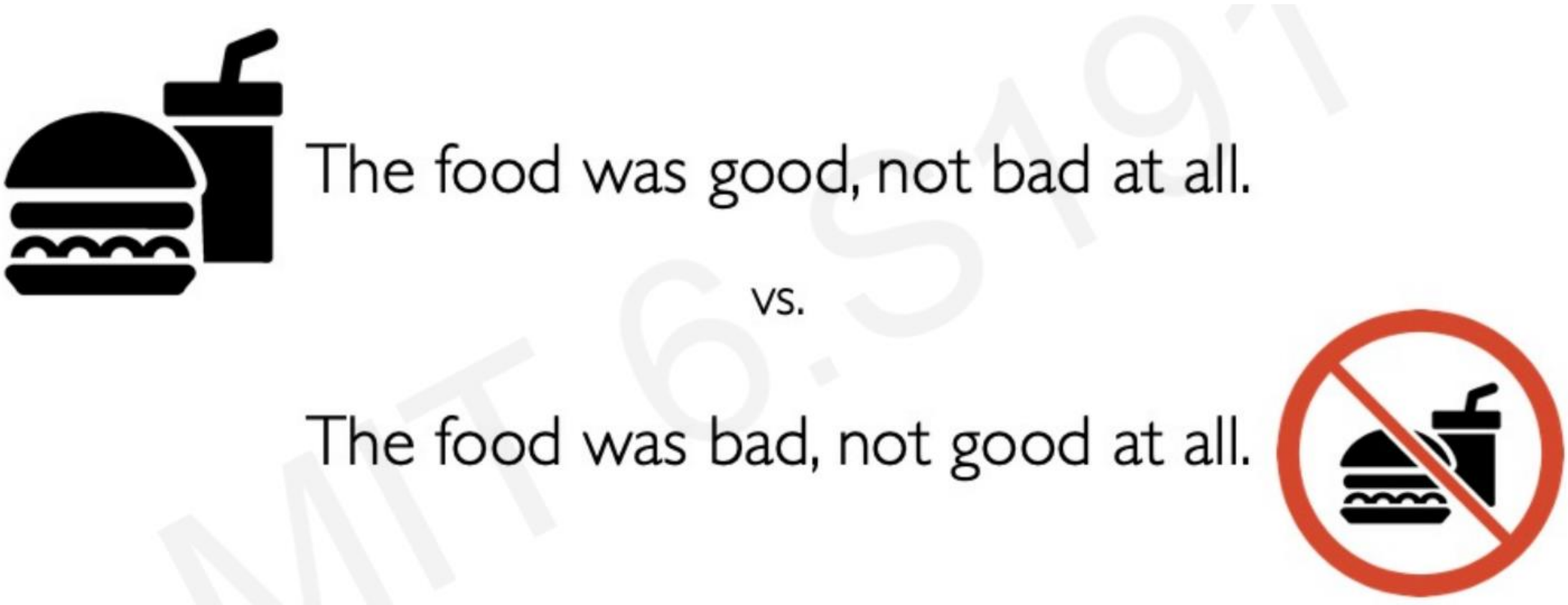
We were hungry but cleaned the house before eating

Traiter les dépendances

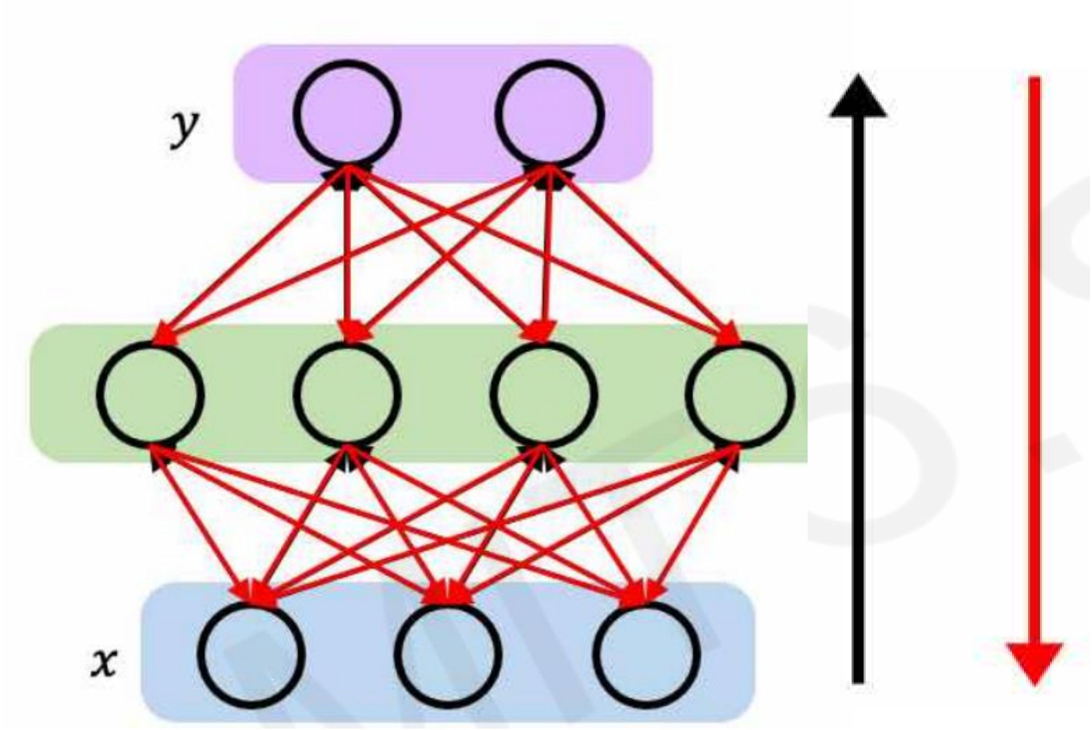
"**France** is where I grew up, but I now live in Boston. I speak fluent ___."

We need information from **the distant past** to accurately predict the correct word.

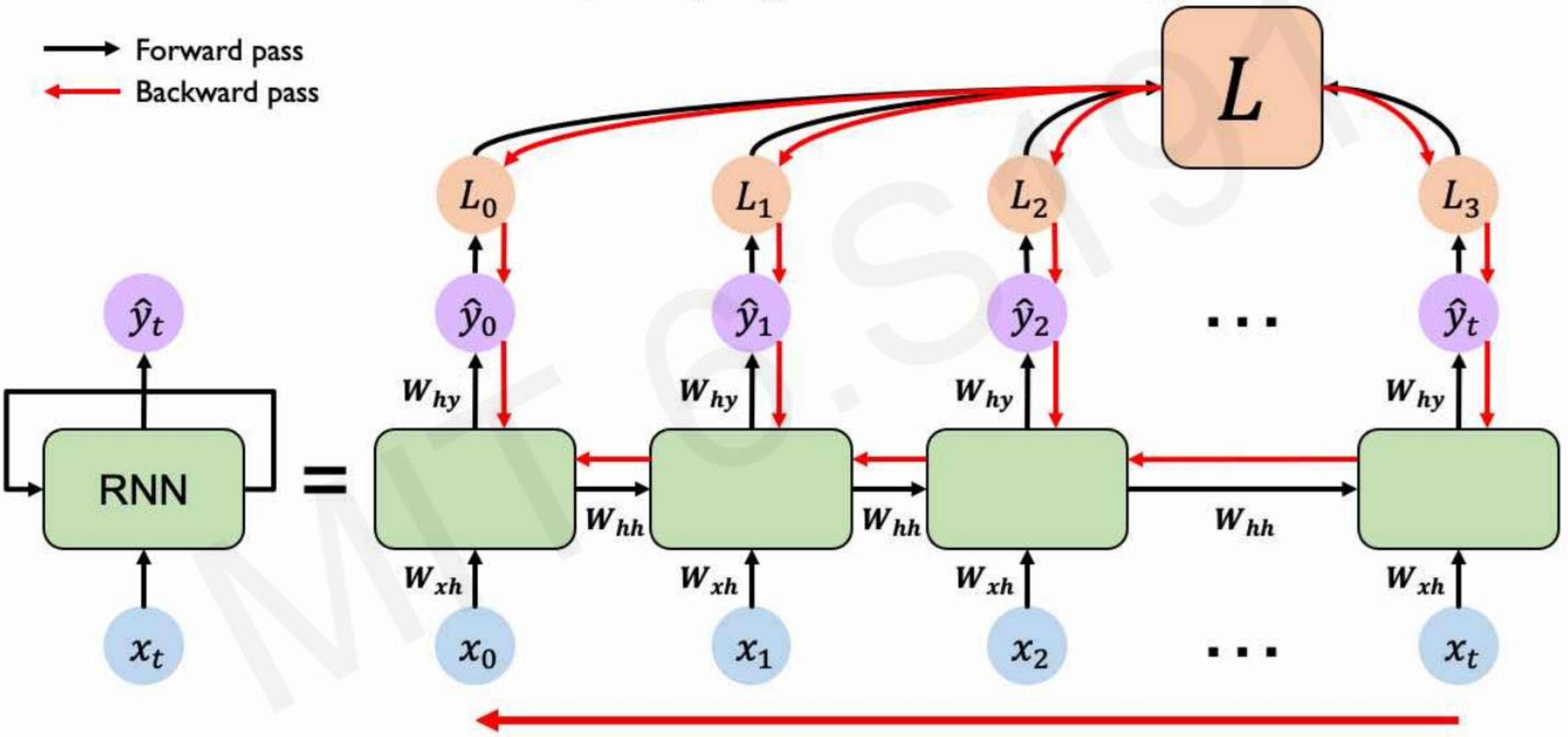
Détecter l’ordre d’apparition des termes



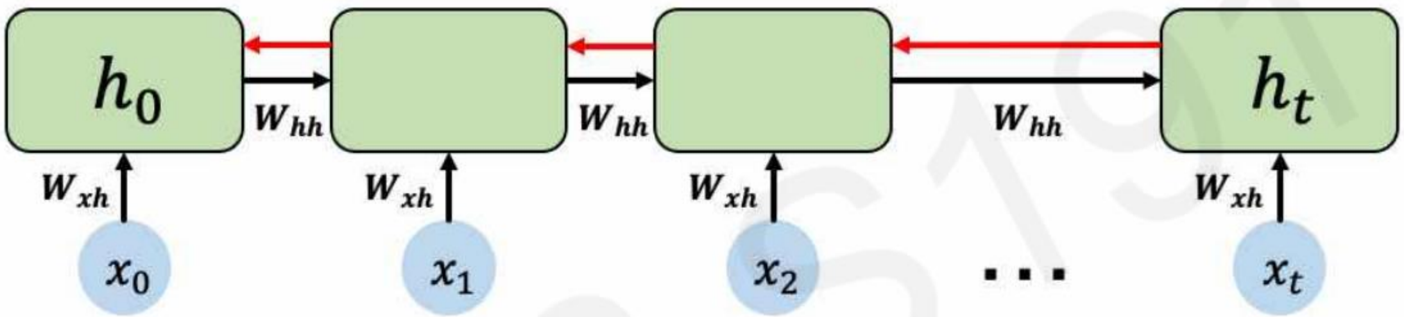
Backpropagation through time (BPTT)



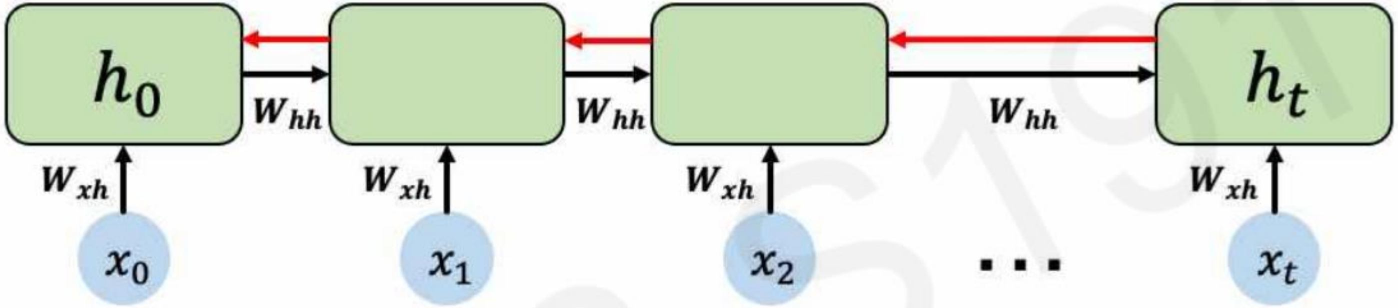
1) On calcule la fonction de perte pour chaque sortie
2) On met à jour les paramètres de façon à minimiser la perte



Forward pass
Backward pass
L
L_0
L_1
L_2
L_3
$\hat{y}_t$
$\hat{y}_0$
$\hat{y}_1$
$\hat{y}_2$
...
$\hat{y}_t$
W_{hy}
W_{hy}
W_{hy}
W_{hy}
RNN
=
W_{hh}
W_{hh}
W_{hh}
W_{xh}
W_{xh}
W_{xh}
W_{xh}
x_t
x_0
x_1
x_2
...
x_t



Computing the gradient wrt h_0 involves **many factors of** W_{hh} + **repeated gradient computation**!



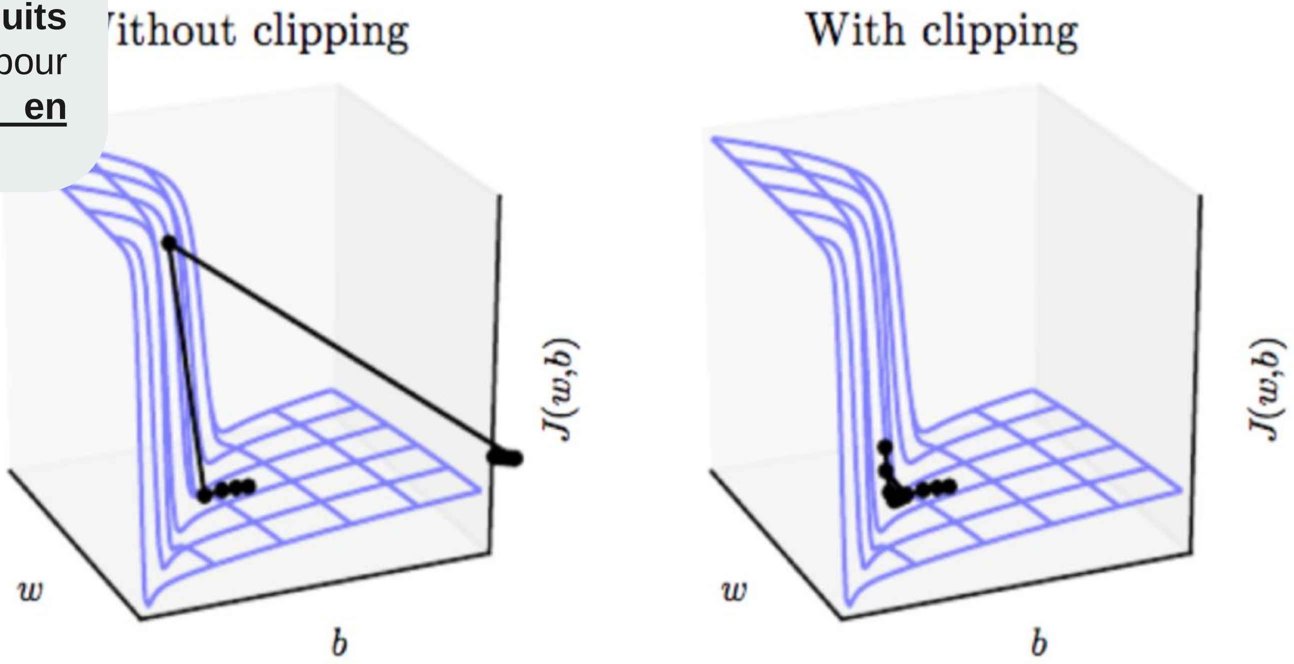
Computing the gradient wrt h_0 involves **many factors of W_{hh} + repeated gradient computation**!

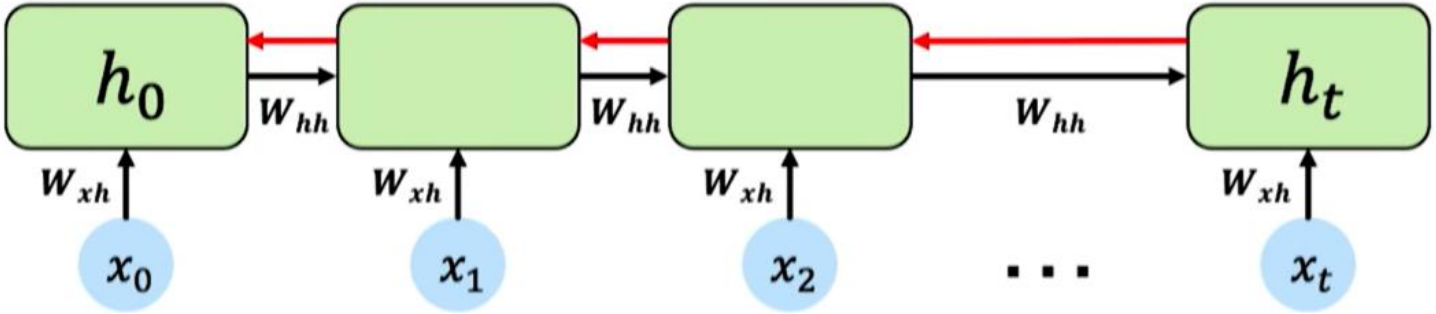
Many values > 1:
exploding gradients

Gradient clipping to
scale big gradients

Le **clipping des gradients** consiste à limiter la norme des gradients pendant l'entraînement à un seuil. Si la norme des gradients dépasse ce seuil, ils sont **réduits proportionnellement** pour respecter cette limite, **sans en modifier la direction**.

- **Tensorflow:** **tf.clip_by_global_norm**





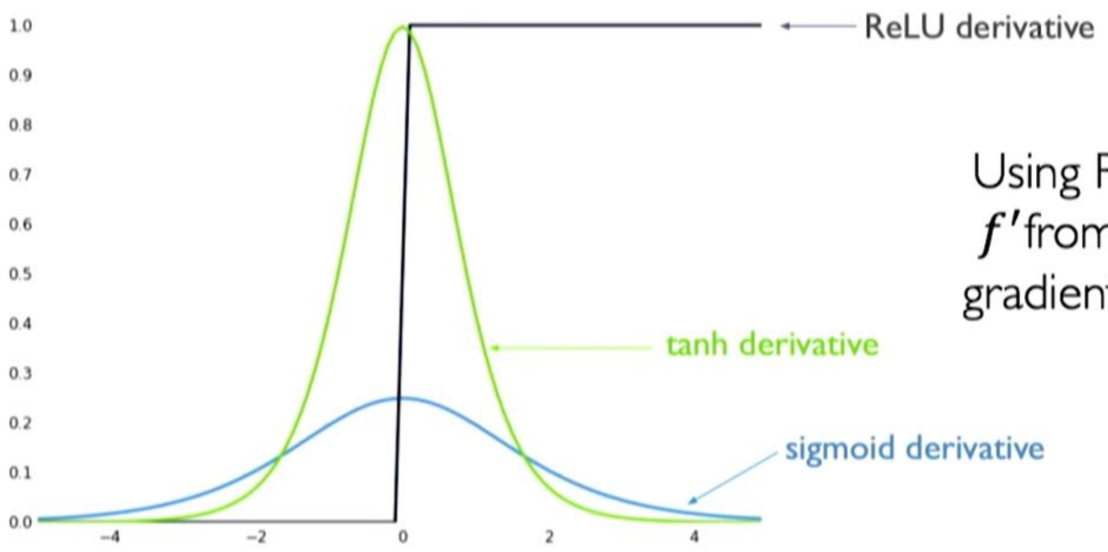
Computing the gradient wrt h_0 involves **many factors of W_{hh} + repeated gradient computation!**

exploding gradients

Many values < 1:
vanishing gradients

1. Activation function
2. Weight initialization
3. Network architecture

Choix de la fonction d'activation



Using ReLU prevents f' from shrinking the gradients when $x > 0$

Initialisation des poids et des biais autrement

Initialize **weights** to identity matrix

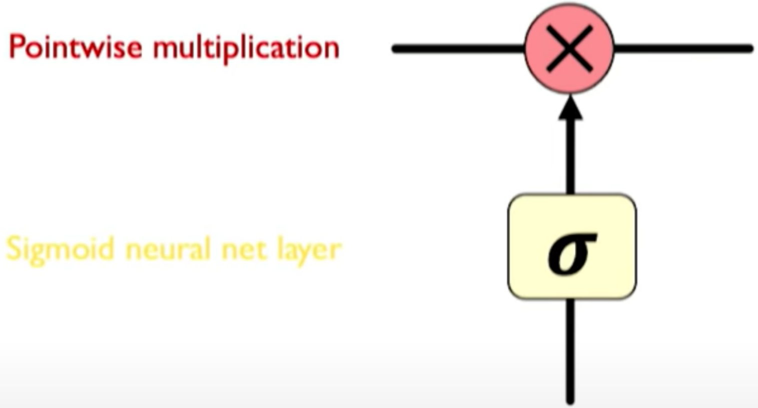
Initialize **biases** to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

Ajout de la cellule Gate

Idea: use **gates** to selectively **add** or **remove** information within **each recurrent unit with**



Gates optionally let information through the cell

Long Short Term Memory (LSTMs) networks rely on a gated cell to track information throughout many time steps.

LSTM and GRU : les cellules d’état

LSTM : Long Short Term Memory

LSTM contrôle le passage du flux d'information

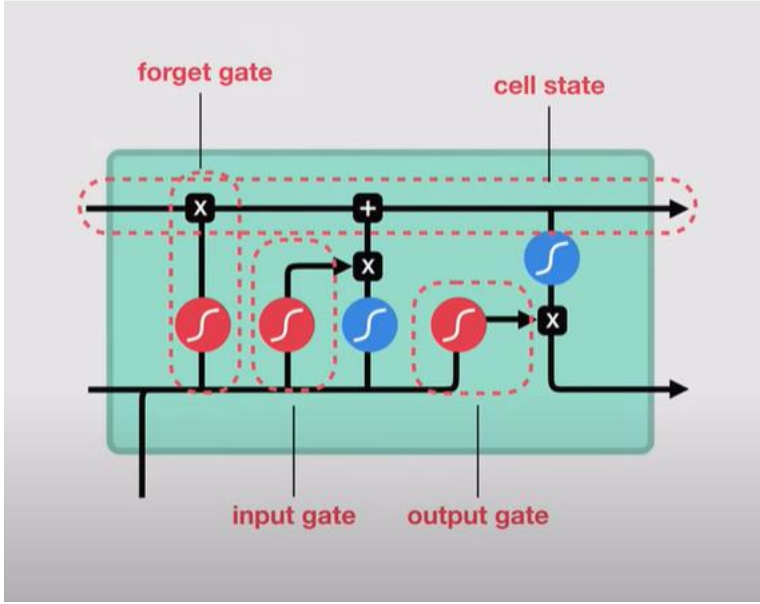
=> Elle peut capter une dépendance qui vient dans le début du texte et ceci grâce à plusieurs étapes:

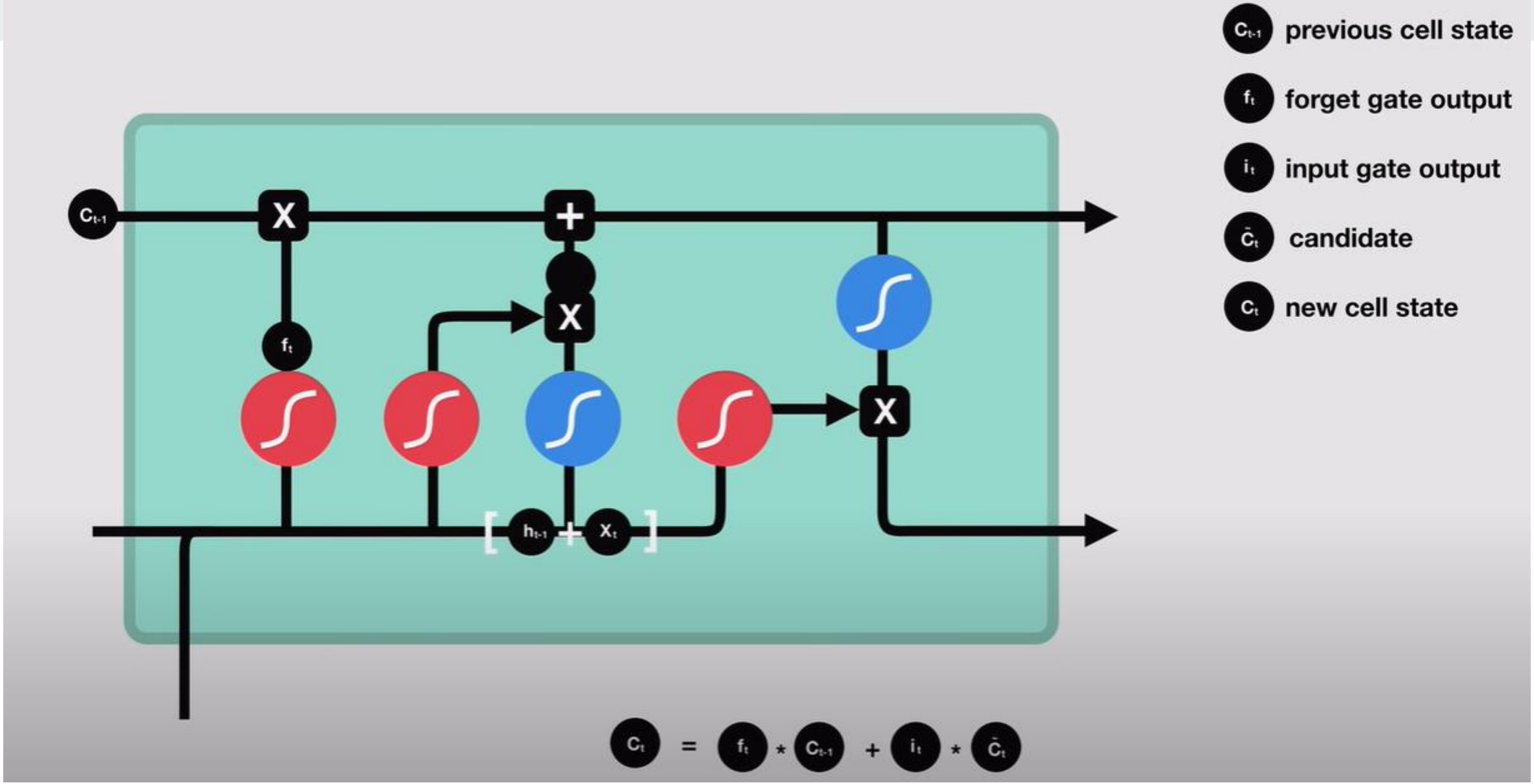
1) forget
2) input
3) Output

 Fonction sigmoid

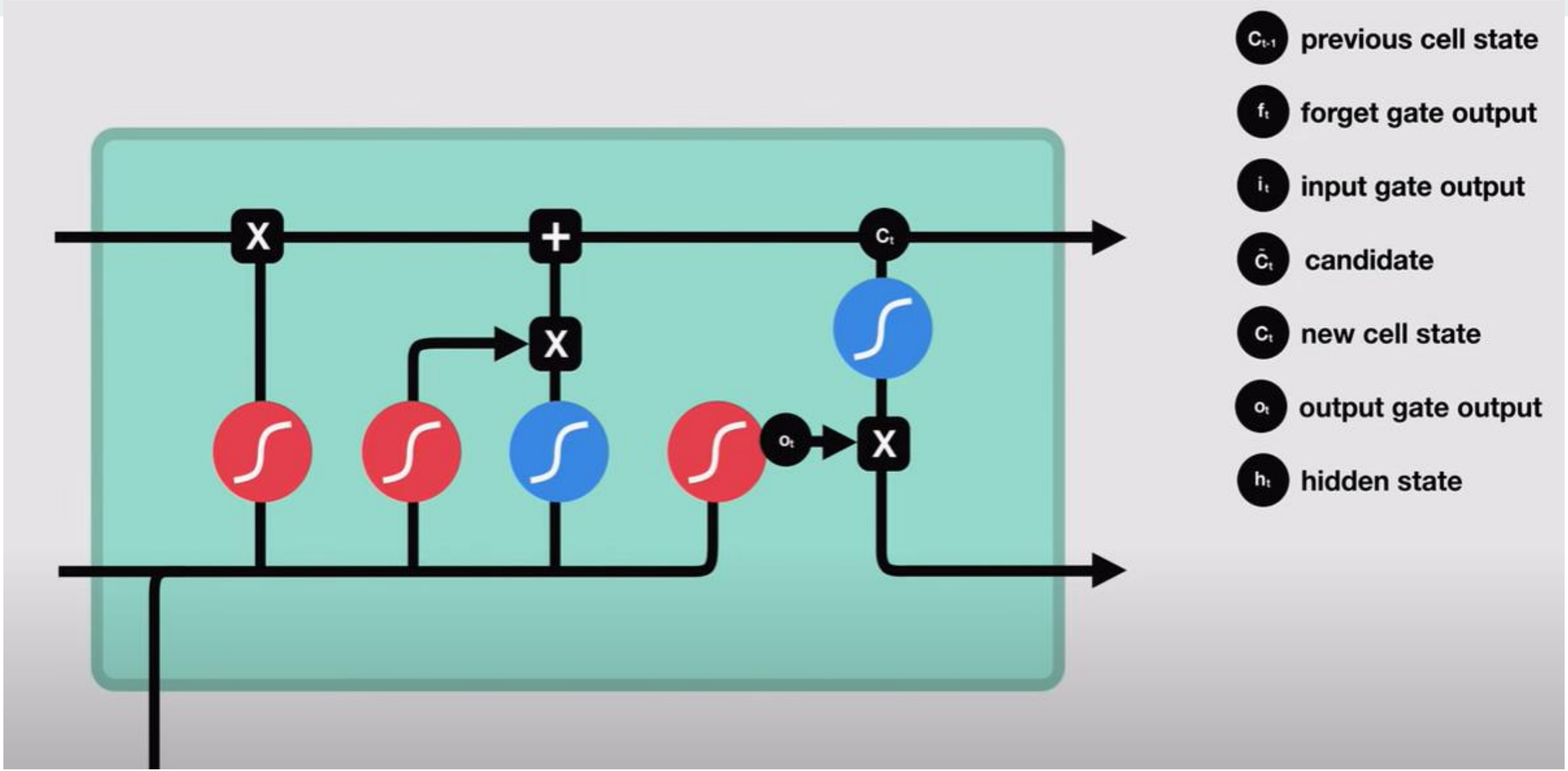
 Fonction tanh

```
tf.keras.layers.LSTM(num_units)
```





c_{t-1} previous cell state
f_t forget gate output
i_t input gate output
$\tilde{c}_t$ candidate
c_t new cell state
c_{t-1}
f_t
h_{t-1} + x_t
$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$


X
+
C_t
X
o_t
X
C_{t-1} previous cell state
f_t forget gate output
i_t input gate output
$\tilde{C}_t$ candidate
C_t new cell state
o_t output gate output
h_t hidden state

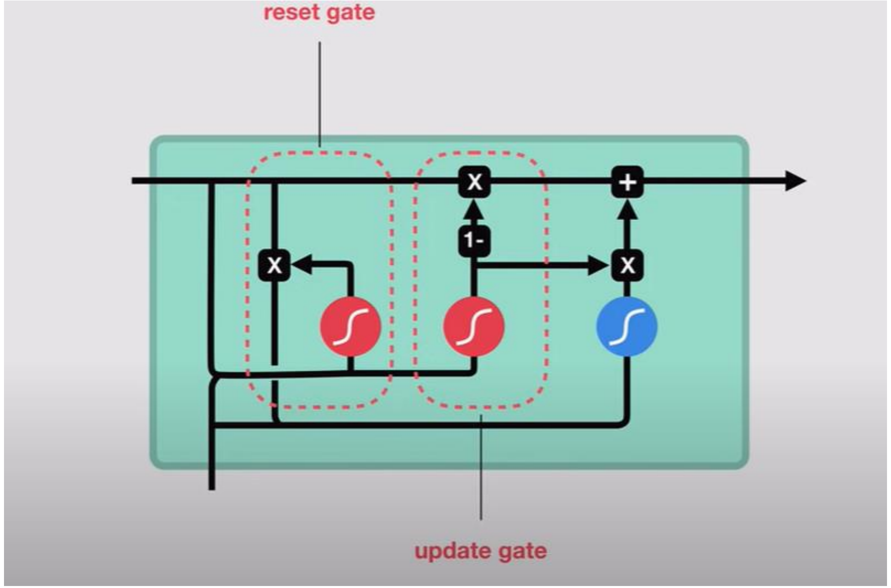
LSTM from Scratch

```
def LSTMCELL(prev_ct, prev_ht, input):
    combine = prev_ht + input
    ft = forget_layer(combine)
    candidate = candidate_layer(combine)
    it = input_layer(combine)
    Ct = prev_ct * ft + candidate * it
    ot = output_layer(combine)
    ht = ot * tanh(Ct)
    return ht, Ct



ct = [0, 0, 0]
ht = [0, 0, 0]

for input in inputs:
    ct, ht = LSTMCELL(ct, ht, input)
```

GRU


Fonction sigmoid


Fonction tanh



Limitations

- Pas assez longue mémoire pour les dépendances lointaines dans le texte
- Des algorithmes très long lors de l’apprentissage => difficulté de faire un codage en parallèle