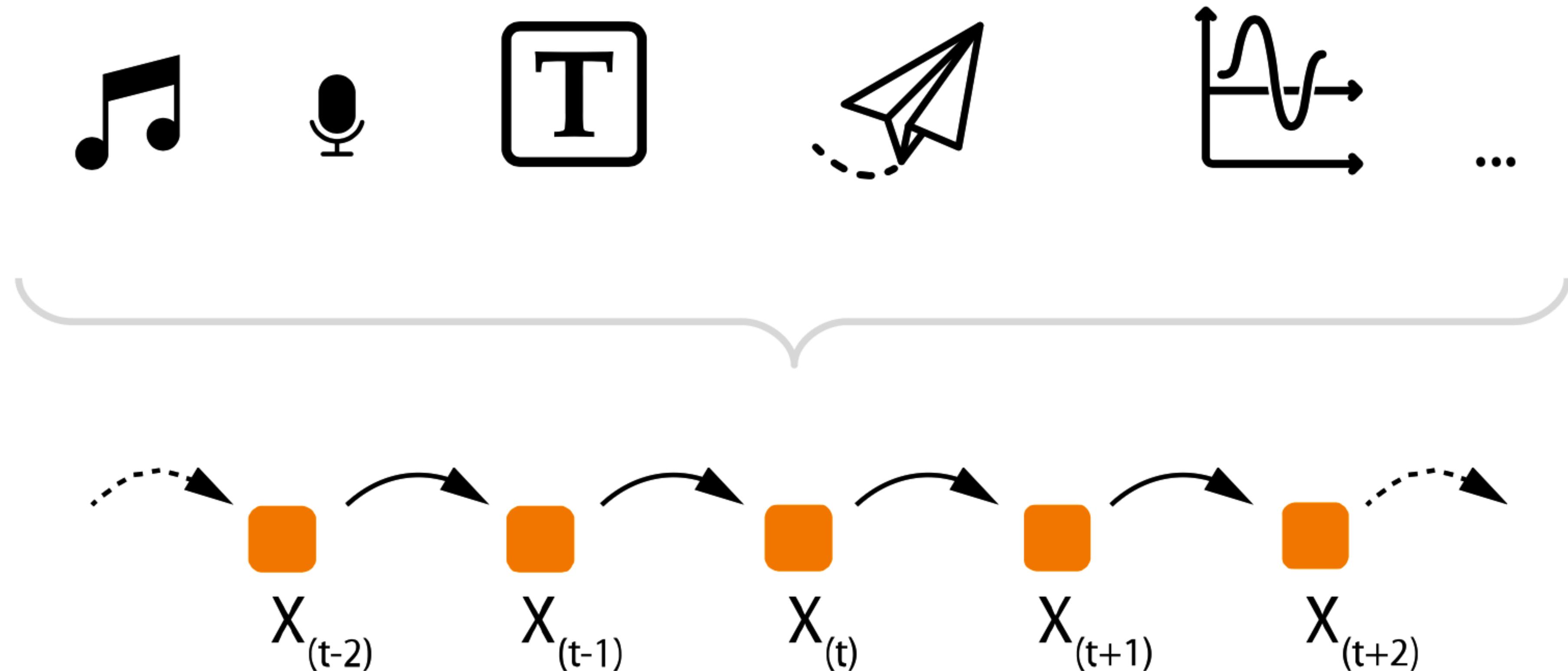


Deep Learning : Les Réseaux de Neurones Récurrents (RNN)

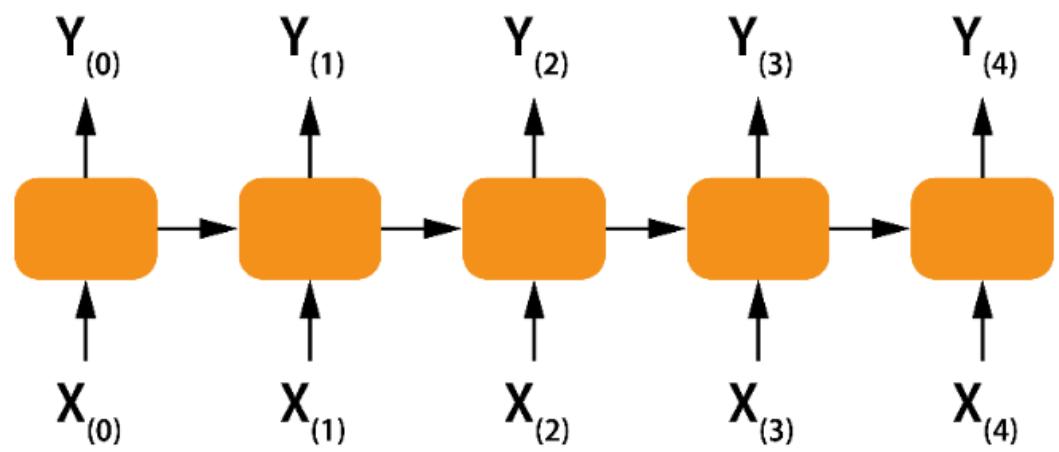
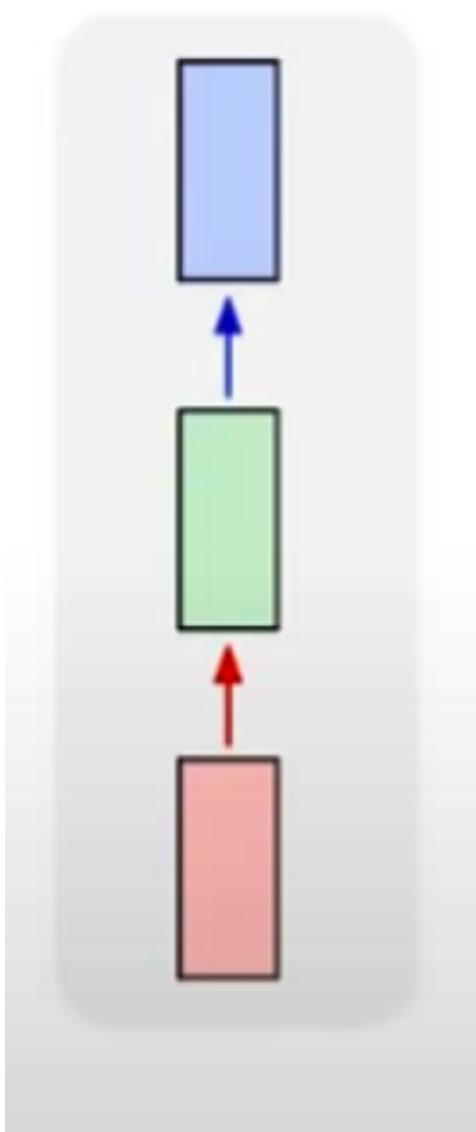
AL SAIDI IBTISSAM

Que se passerait-il si tout le monde n'était qu'une seule et immense séquence ?



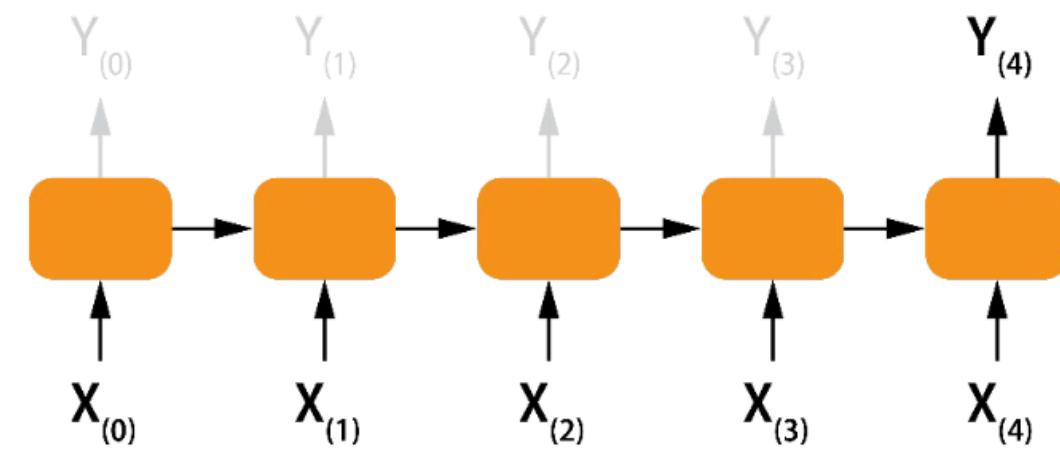
Réseau de neurone récurrent RNN

one to one



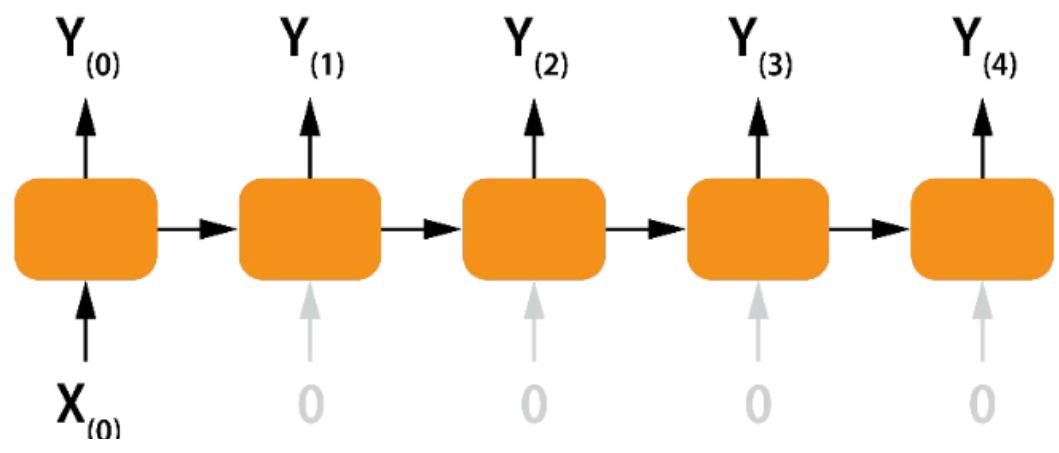
Many to many / Serie to serie

Exemple : Classification de
trames vidéos



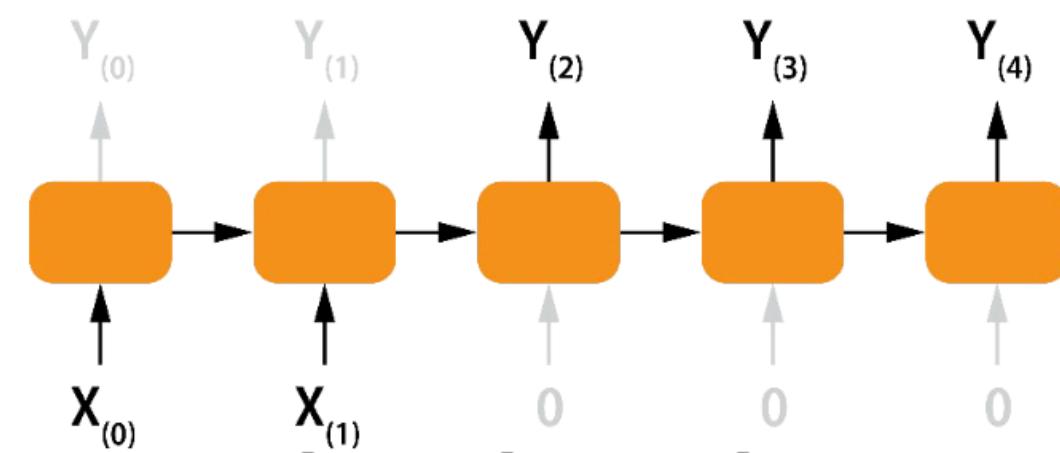
Many to one / Serie to vector

Exemple : Classification de
sentiment(texte)



One to many / Vector to serie

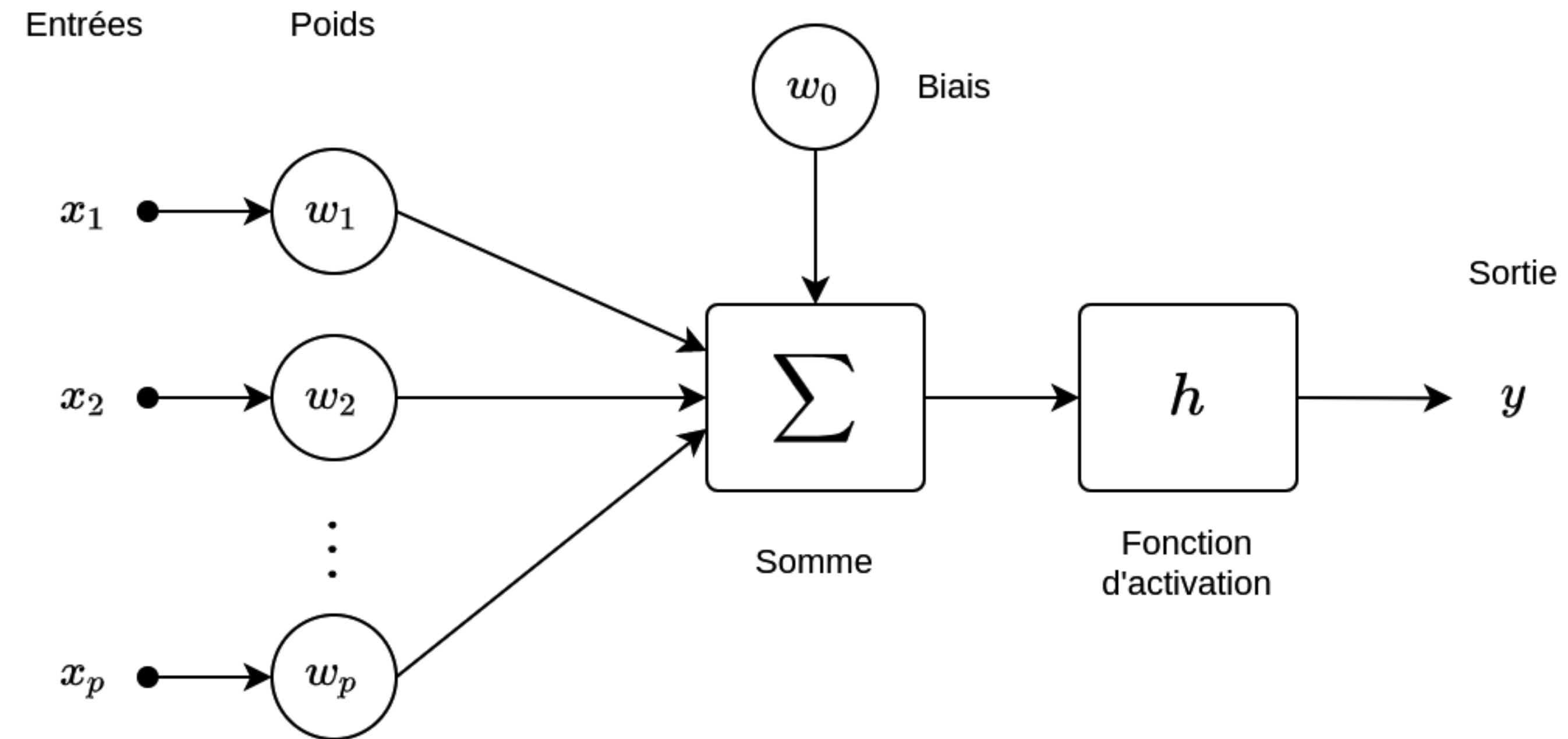
Exemple : Image captioning



Many to many / Encoder to decoder

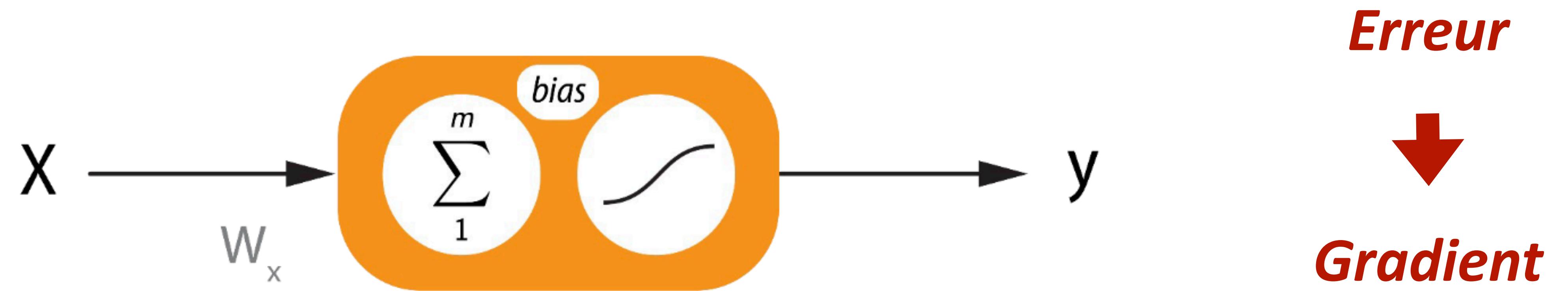
Exemple : Traduction, Réponse aux
questions (tailles entrée/sortie
variables)

Évolution des Neurones : la Forme Classique (forward propagation)



Réseau de neurone classique

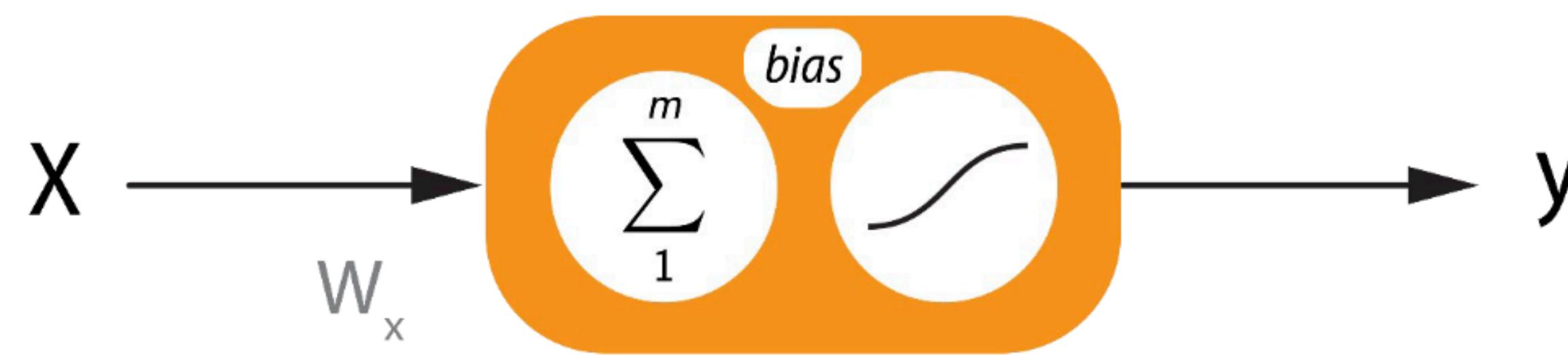
Évolution des Neurones : la Forme Classique (forward propagation)



$$y = \sigma(W_x^T \cdot X + b)$$

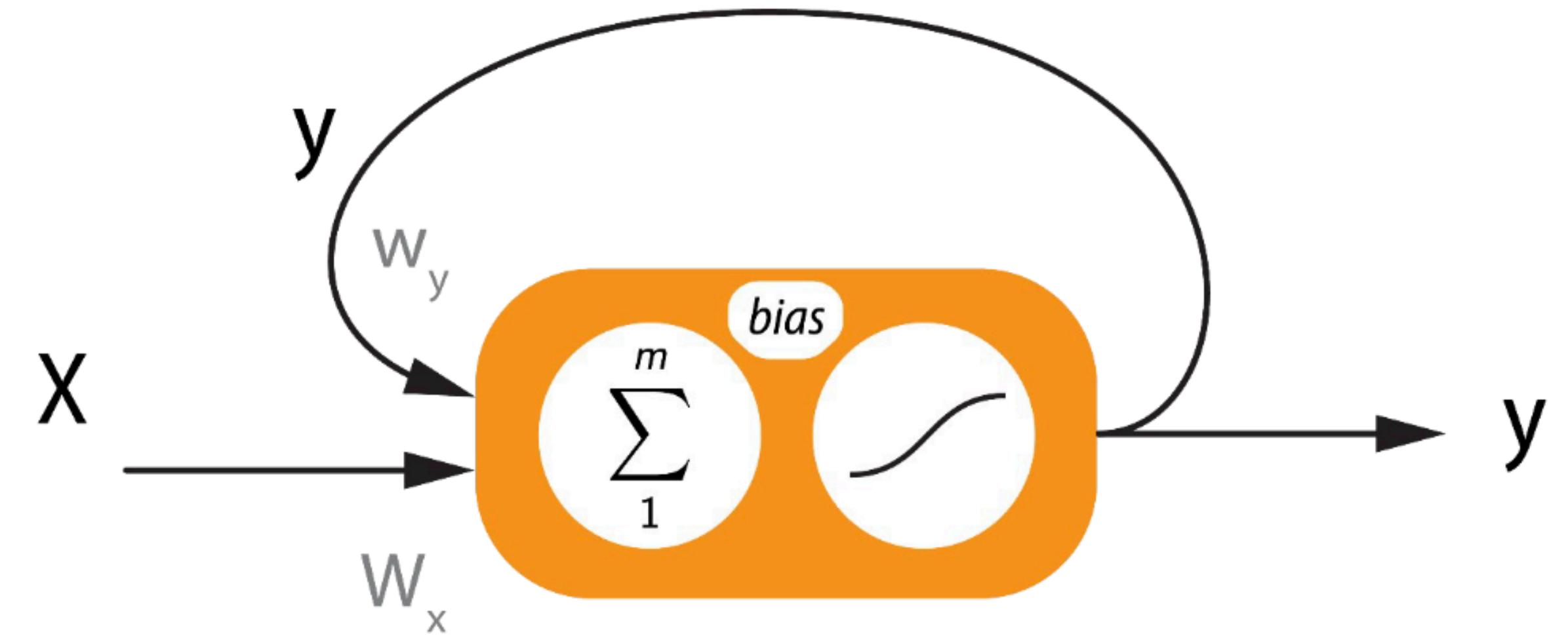
Réseau de neurone classique

Évolution des Neurones : De la Forme Classique à la Configuration Récurrente



$$y = \sigma(W_x^T \cdot X + b)$$

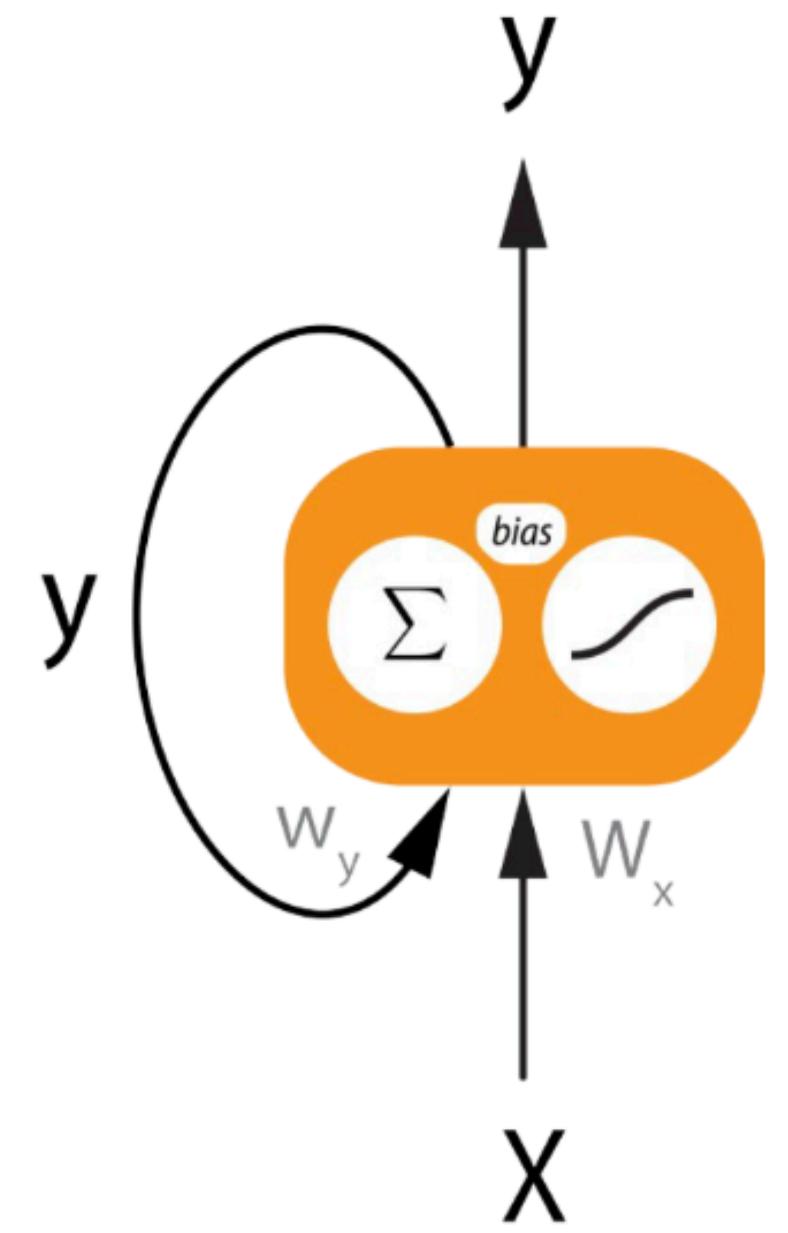
Réseau de neurone classique



$$y_{(t)} = \sigma(W_x^T \cdot X_{(t)} + w_y \cdot y_{(t-1)} + b)$$

Réseau de neurone récurrent

Neurone récurrent simple



$$y(t) = \sigma(W_x^T \cdot X_{(t)} + w_y \cdot y_{(t-1)} + b)$$

Avec :

W_y : est un scalaire

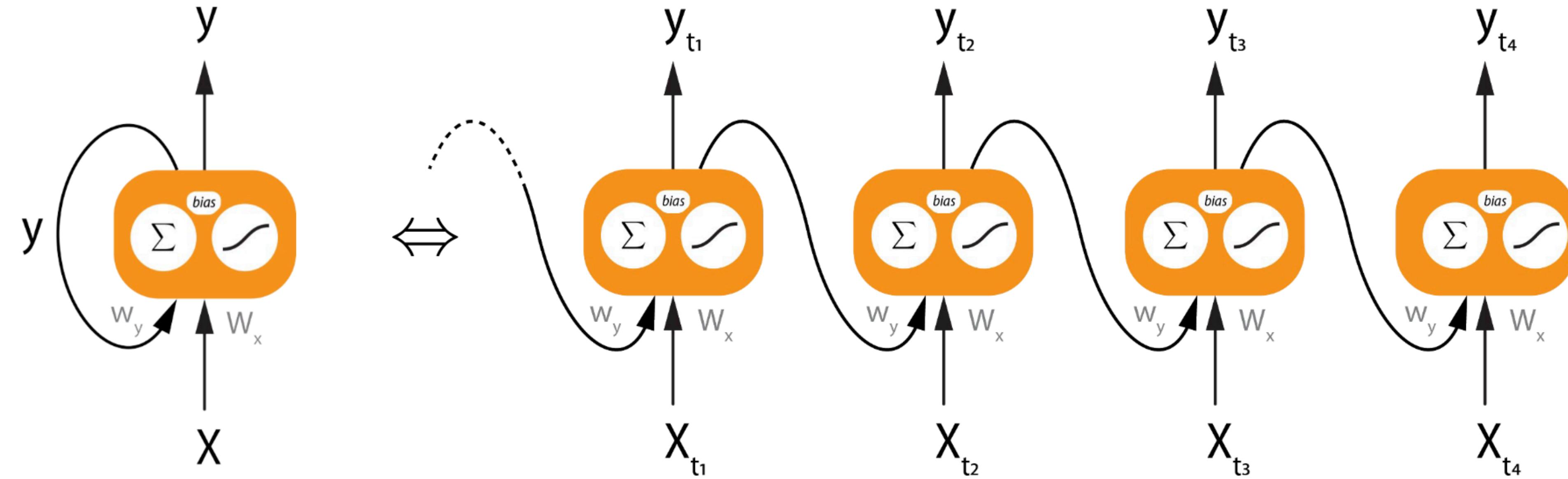
W_x : est un vecteur

b : est un scalaire

y : est un scalaire

Réseau de neurone récurrent

Neurone récurrent simple



Réseau de neurone récurrent

$$y_{(t)} = \sigma(W_x^T \cdot X_{(t)} + w_y \cdot y_{(t-1)} + b)$$

Avec :

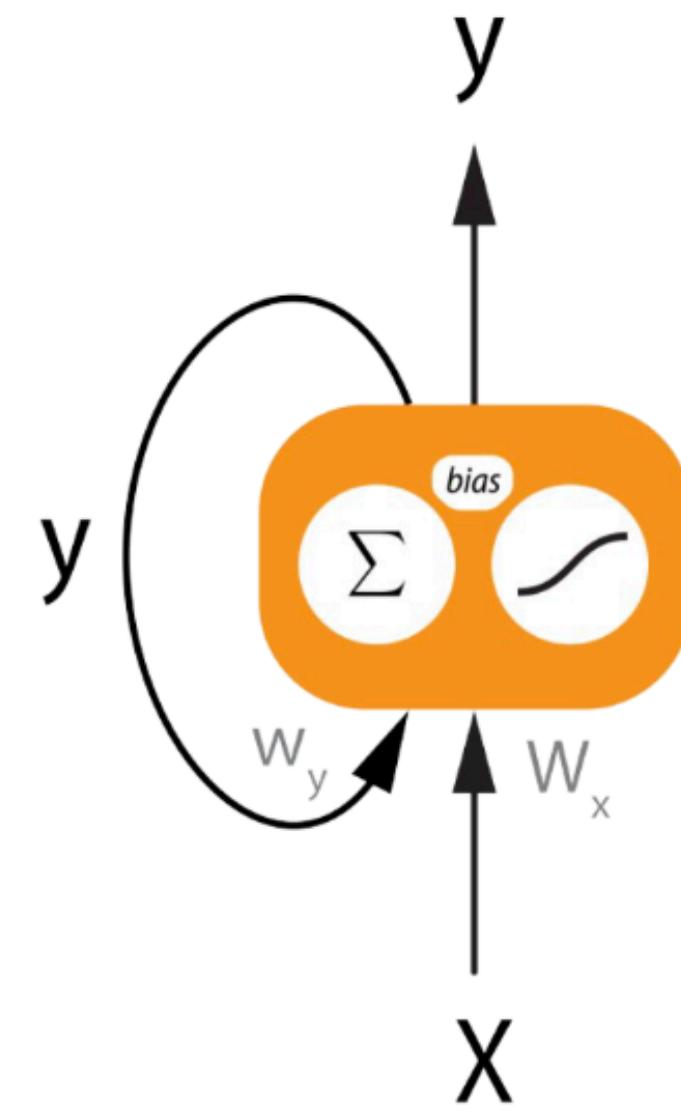
w_y : est un scalaire

W_x : est un vecteur

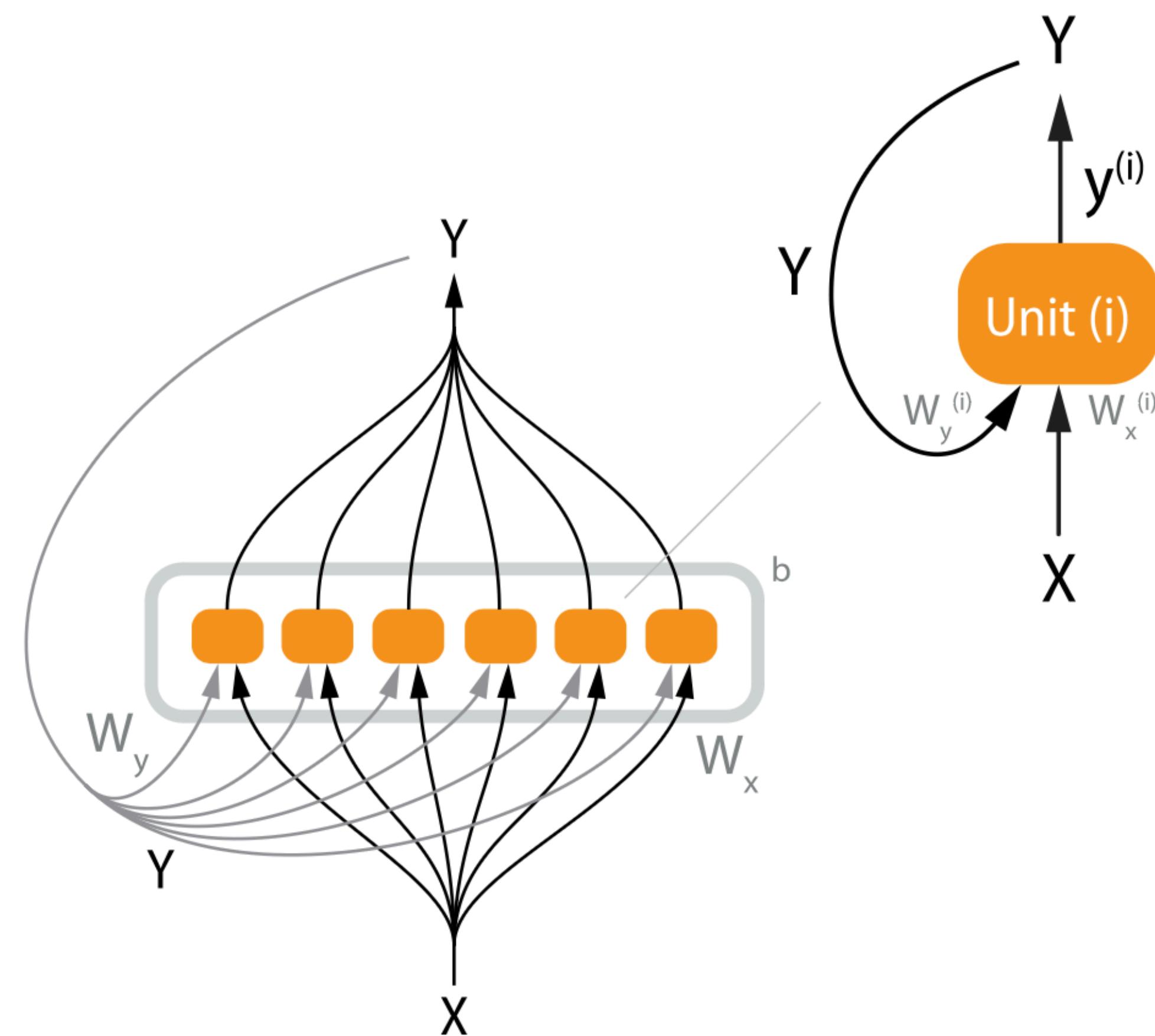
b : est un scalaire

y : est un scalaire

Couche / Cellule récurrente

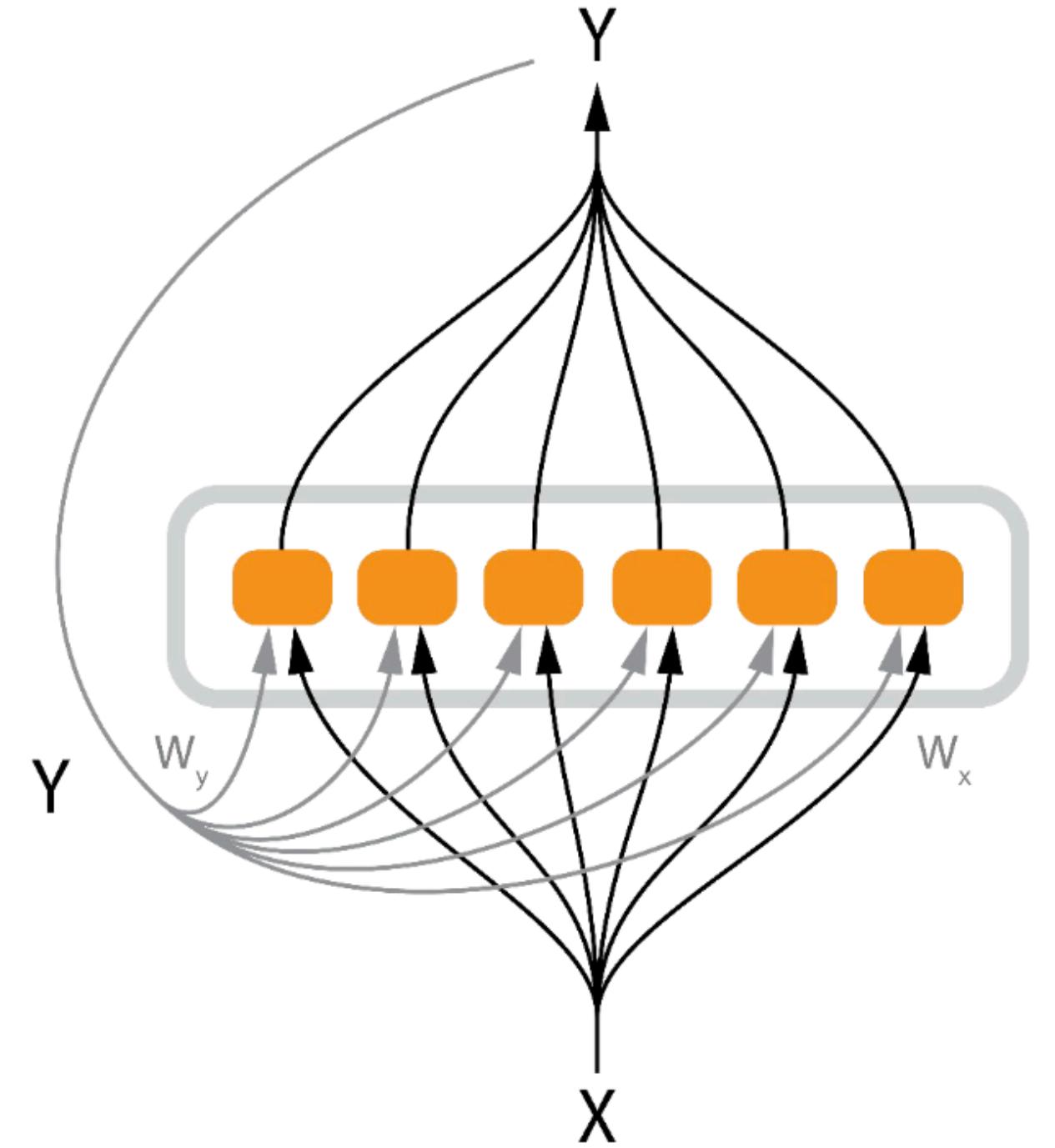


*Réseau de neurone
récurrent*



*Couche / Cellule
récurrente*

Couche / Cellule récurrente



$$Y_{(t)} = \Phi(W_x^T \cdot X_{(t)} + W_y^T \cdot Y_{(t-1)} + b)$$

Avec :

W_y : est un tensor

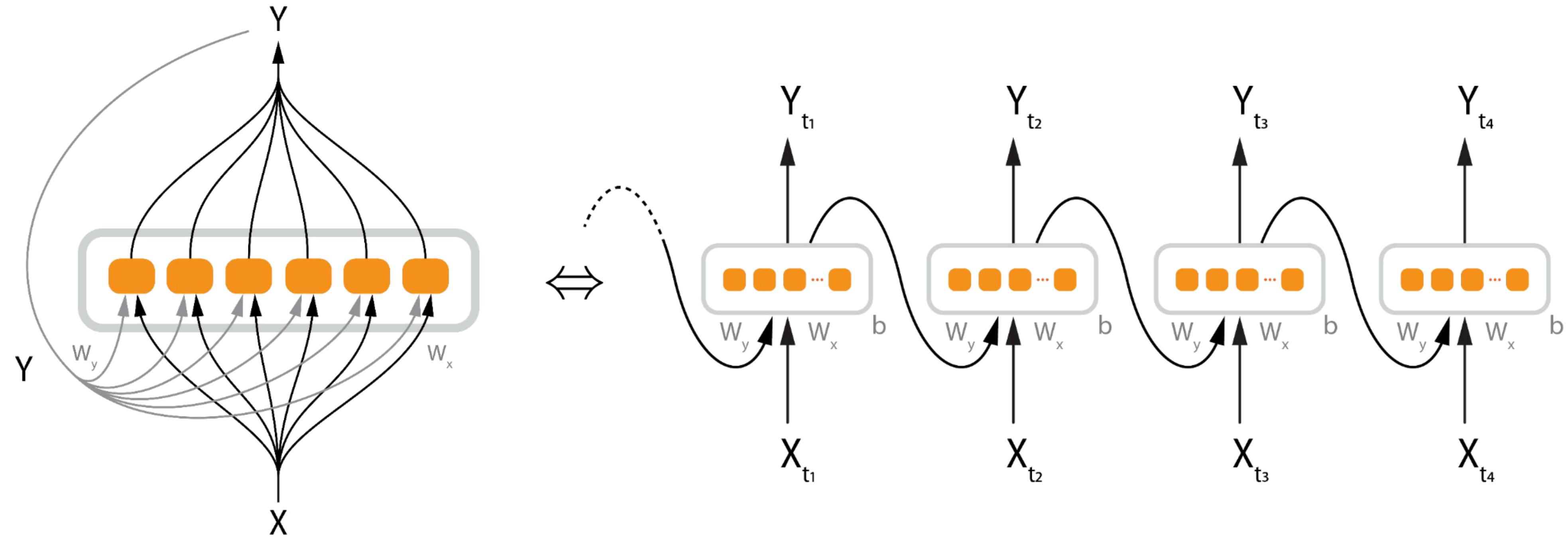
W_x : est un tensor

b : est un vecteur

y : est un vecteur

*Couche / Cellule
récurrente*

Couche / Cellule récurrente



**Couche / Cellule
récurrente**

$$Y_{(t)} = \phi(W_x^T \cdot X_{(t)} + W_y^T \cdot Y_{(t-1)} + b)$$

Avec :

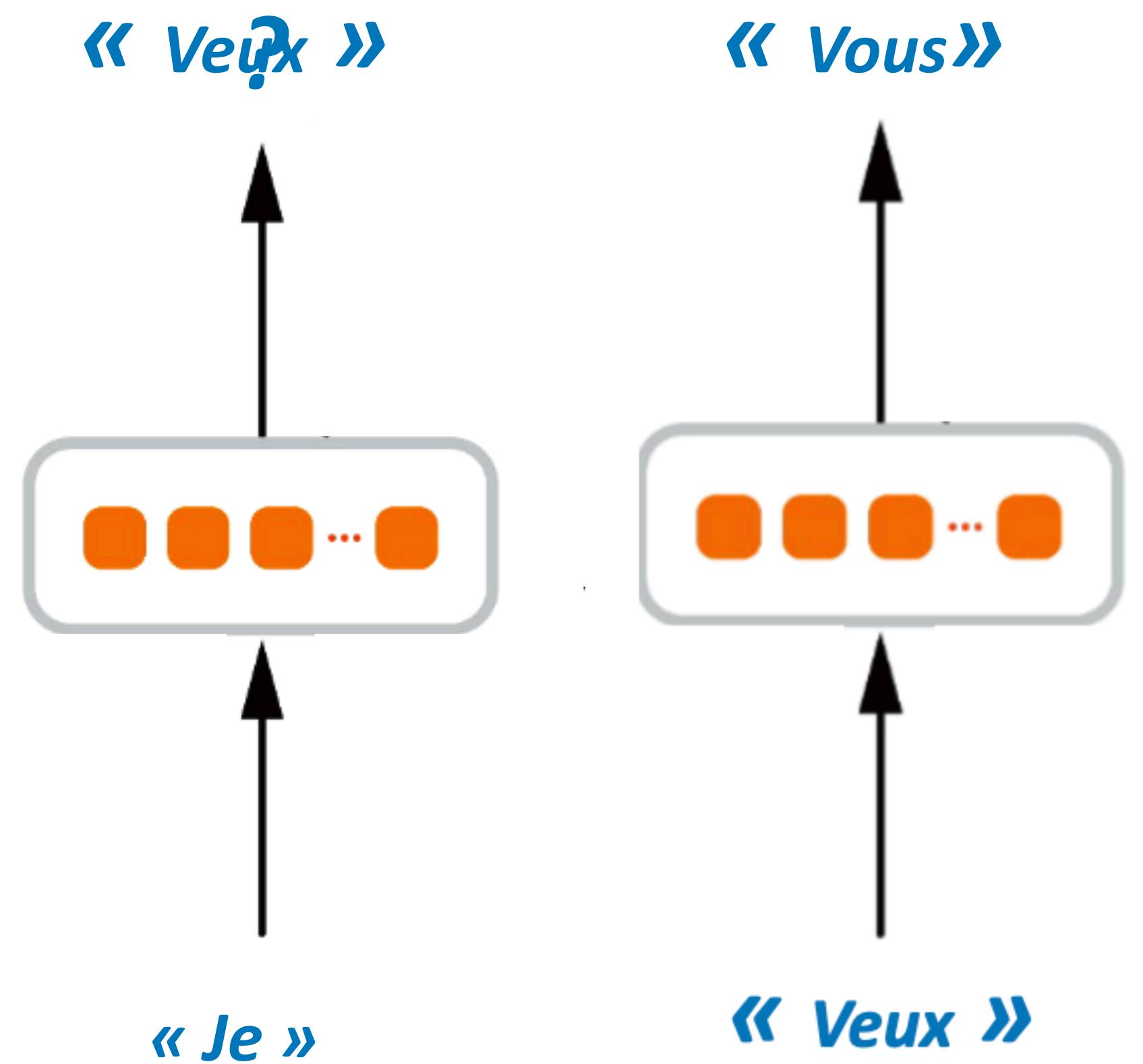
W_y : est un tensor

W_x : est un tensor

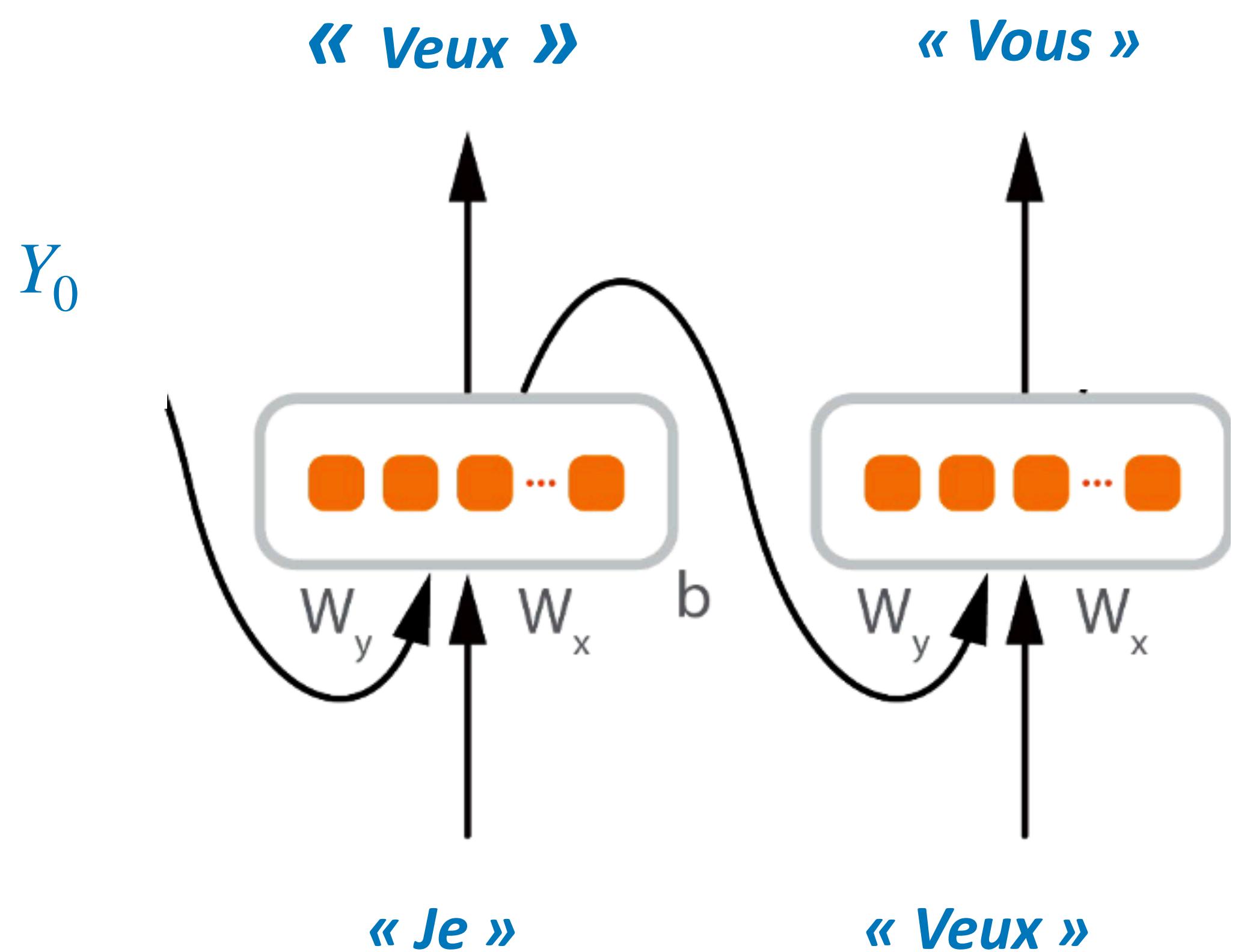
b : est un vecteur

y : est un vecteur

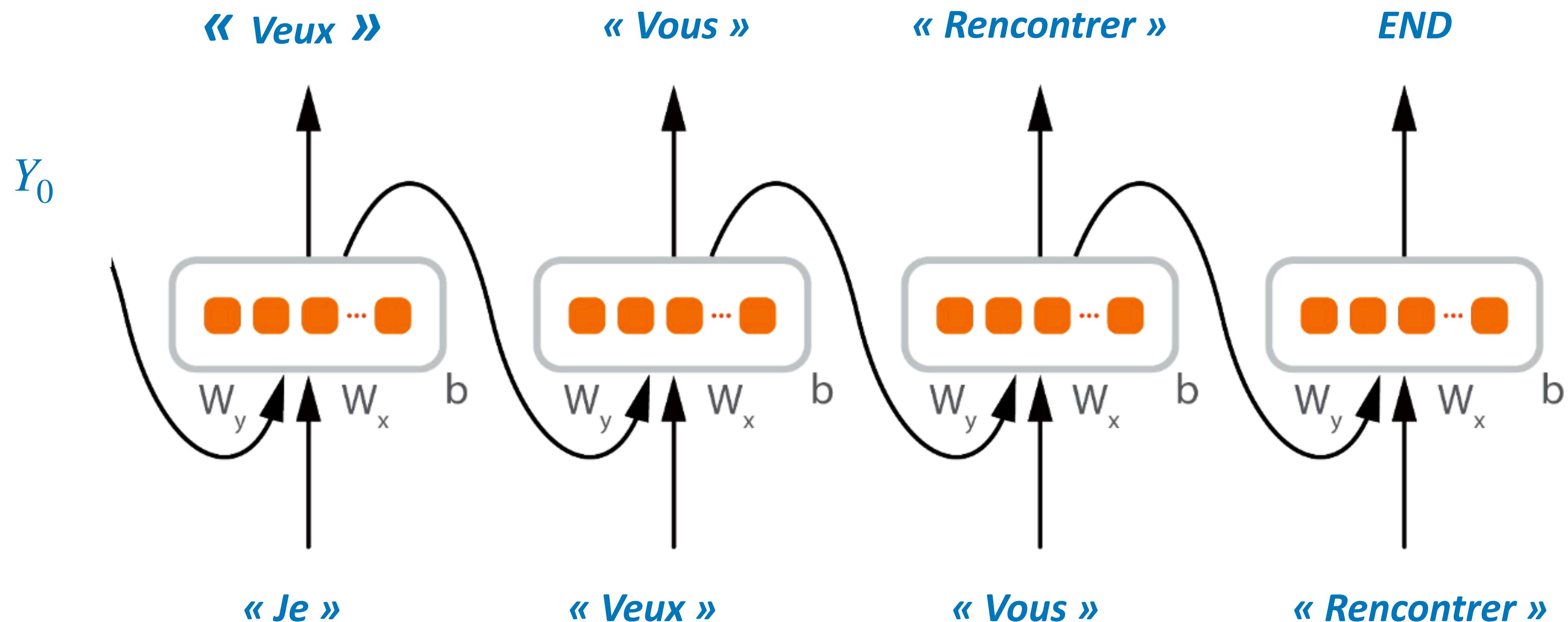
Couche / Cellule récurrente



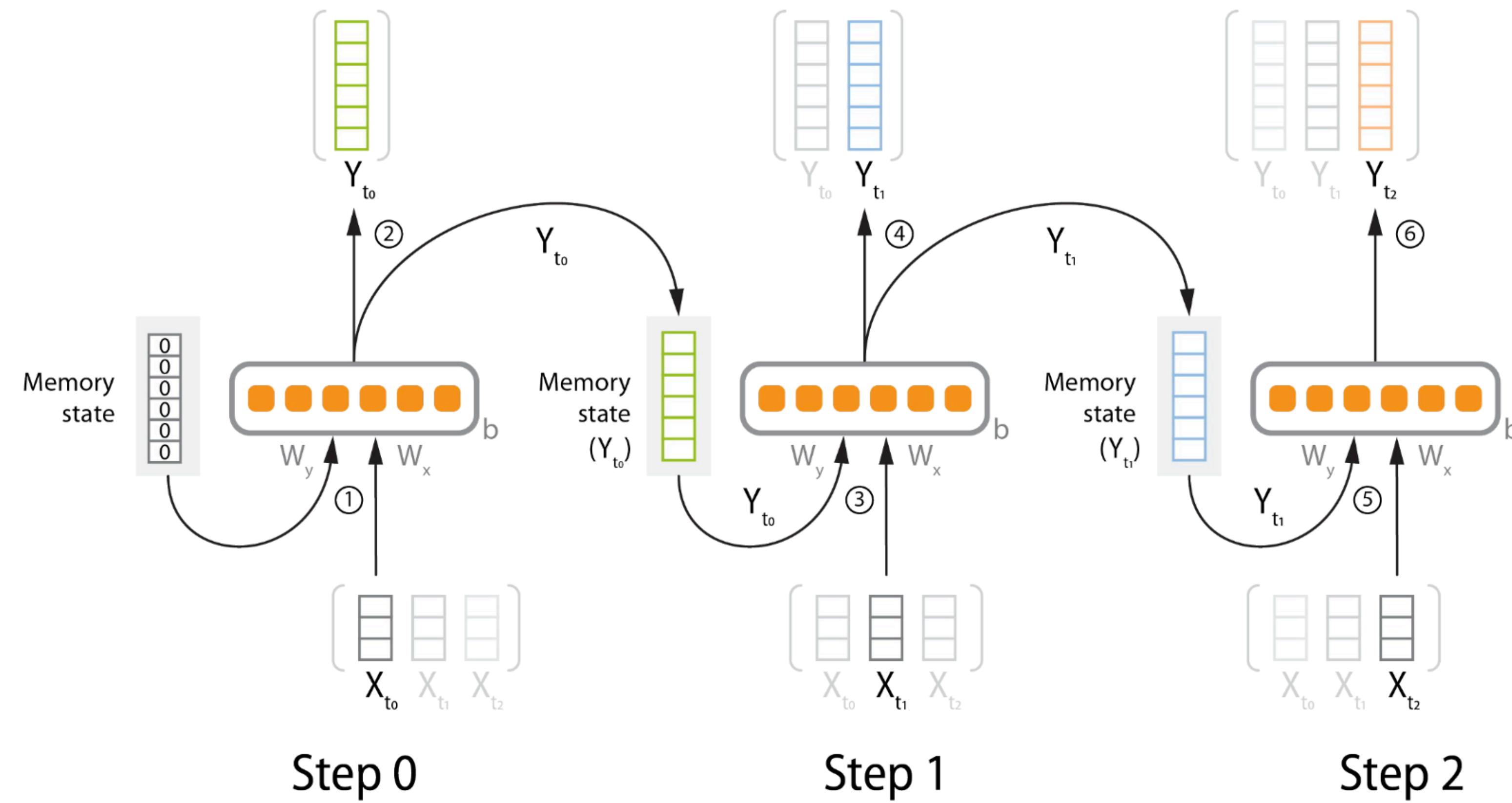
Couche / Cellule récurrente



Couche / Cellule récurrente



Couche / Cellule récurrente



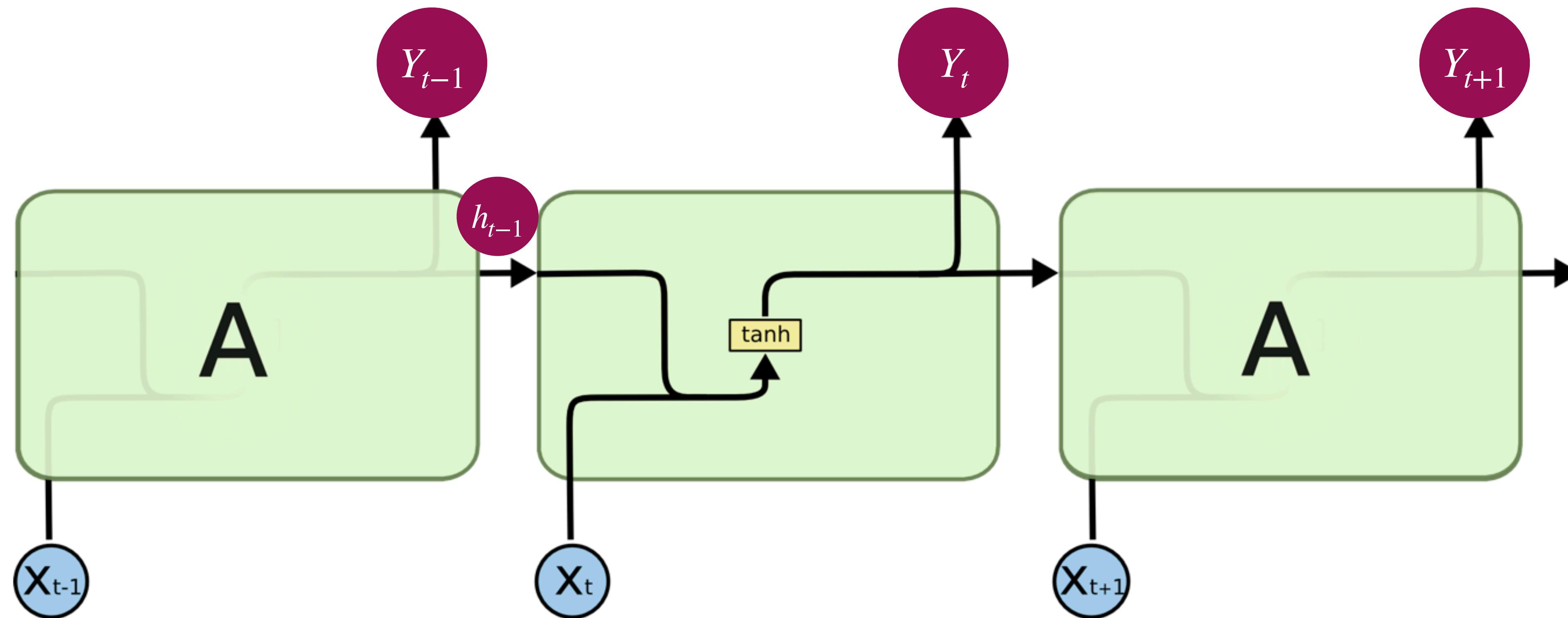
Avec :

- La forme de Wx est : (nombre d'unités, taille de x)
- La forme de Wy est : (nombre d'unités, nombre d'unités)
- La forme de y est : (nombre d'unités,)

Couche / Cellule récurrente

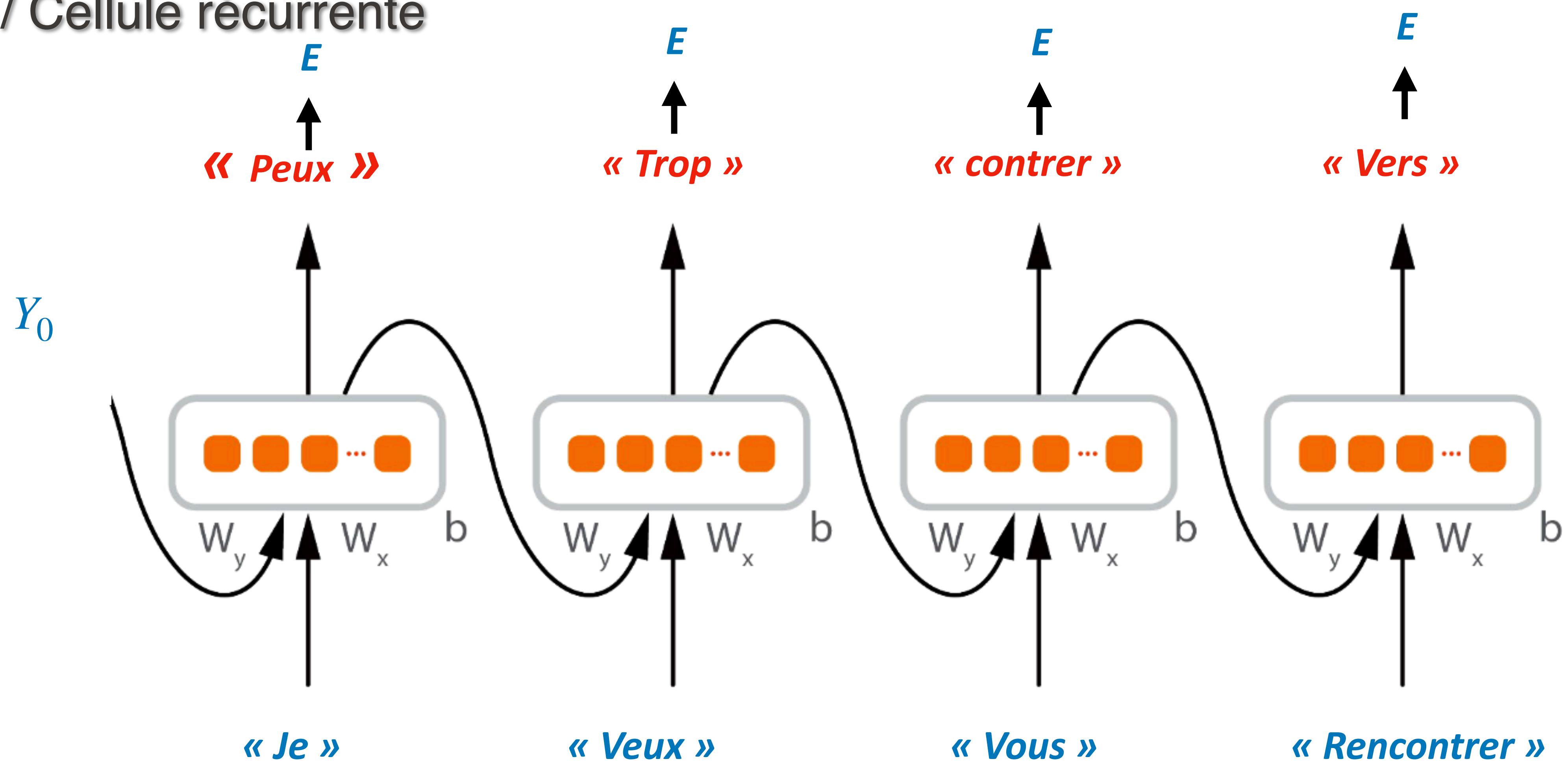
$$Y_{(t)} = \phi (W_x^T \cdot X_{(t)} + W_y^T \cdot Y_{(t-1)} + b)$$

Couche / Cellule récurrente



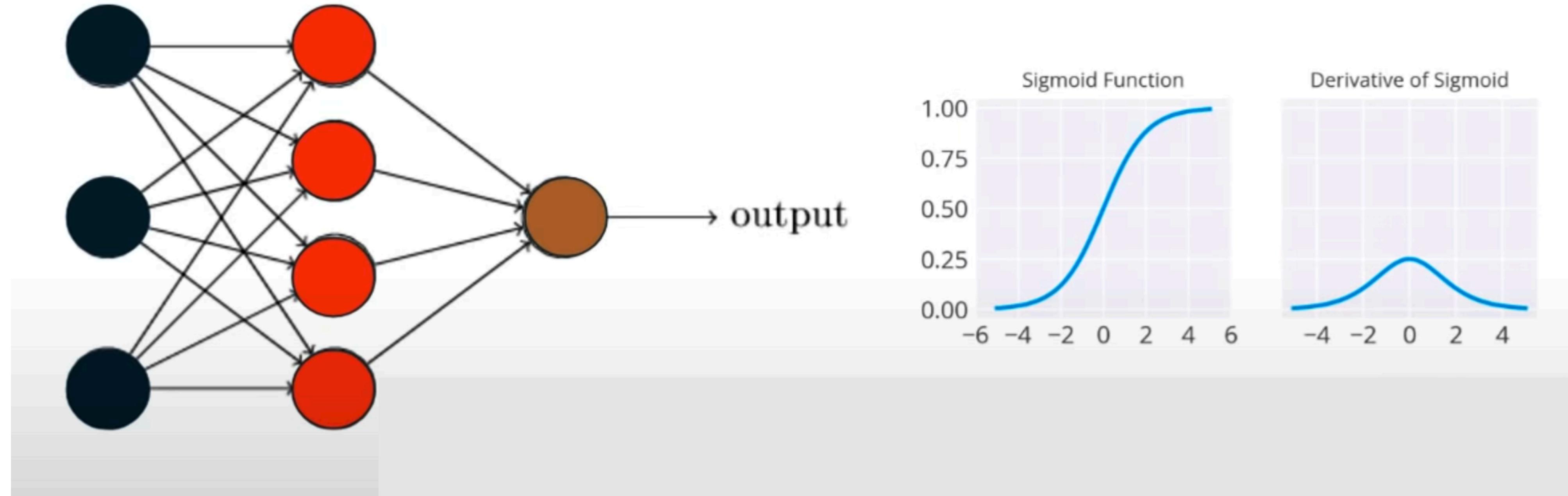
$$Y_{(t)} = \tanh(W_{xY}^T X_{(t)} + W_{hY}^T h_{(t-1)} + b_Y)$$

Couche / Cellule récurrente



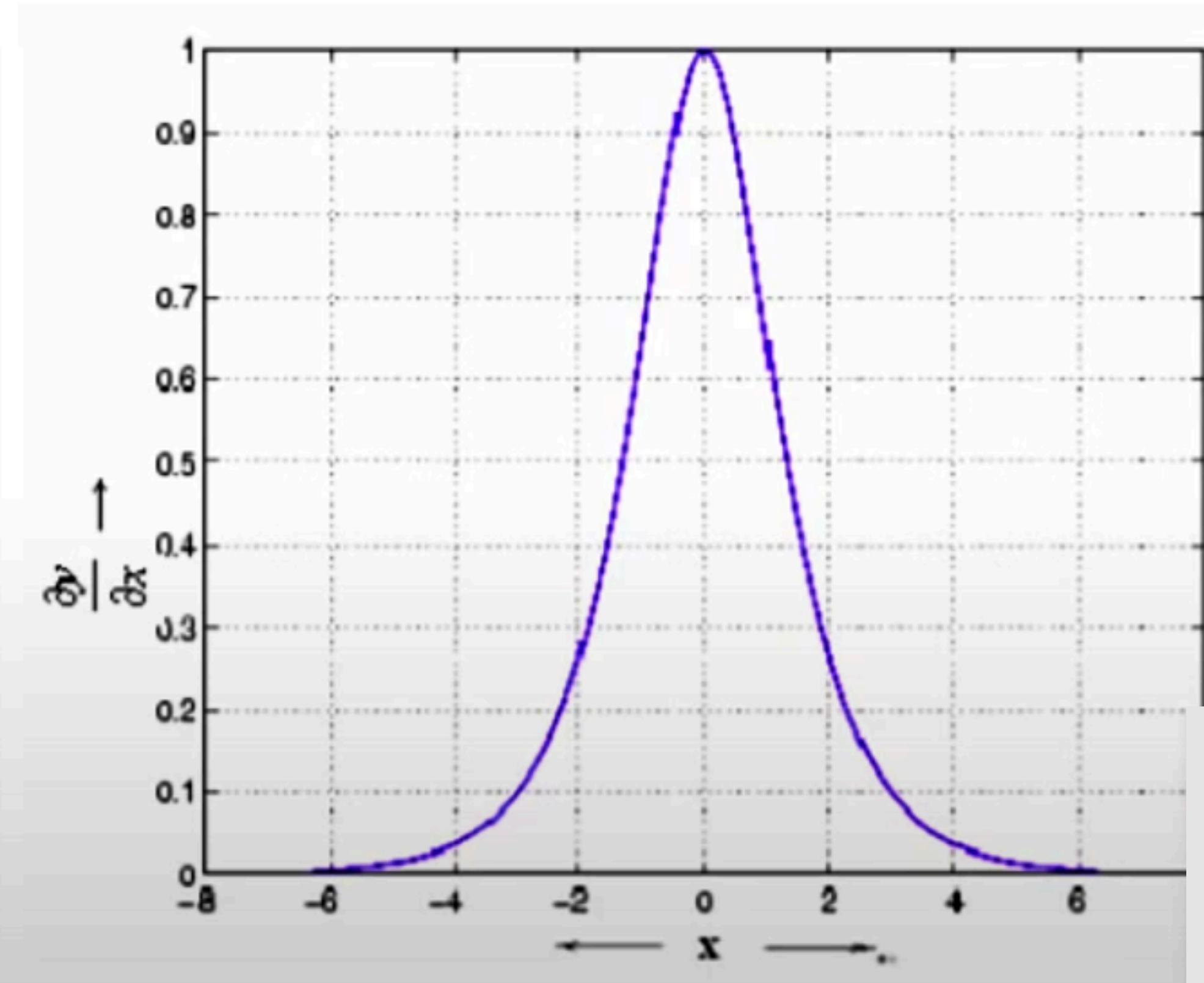
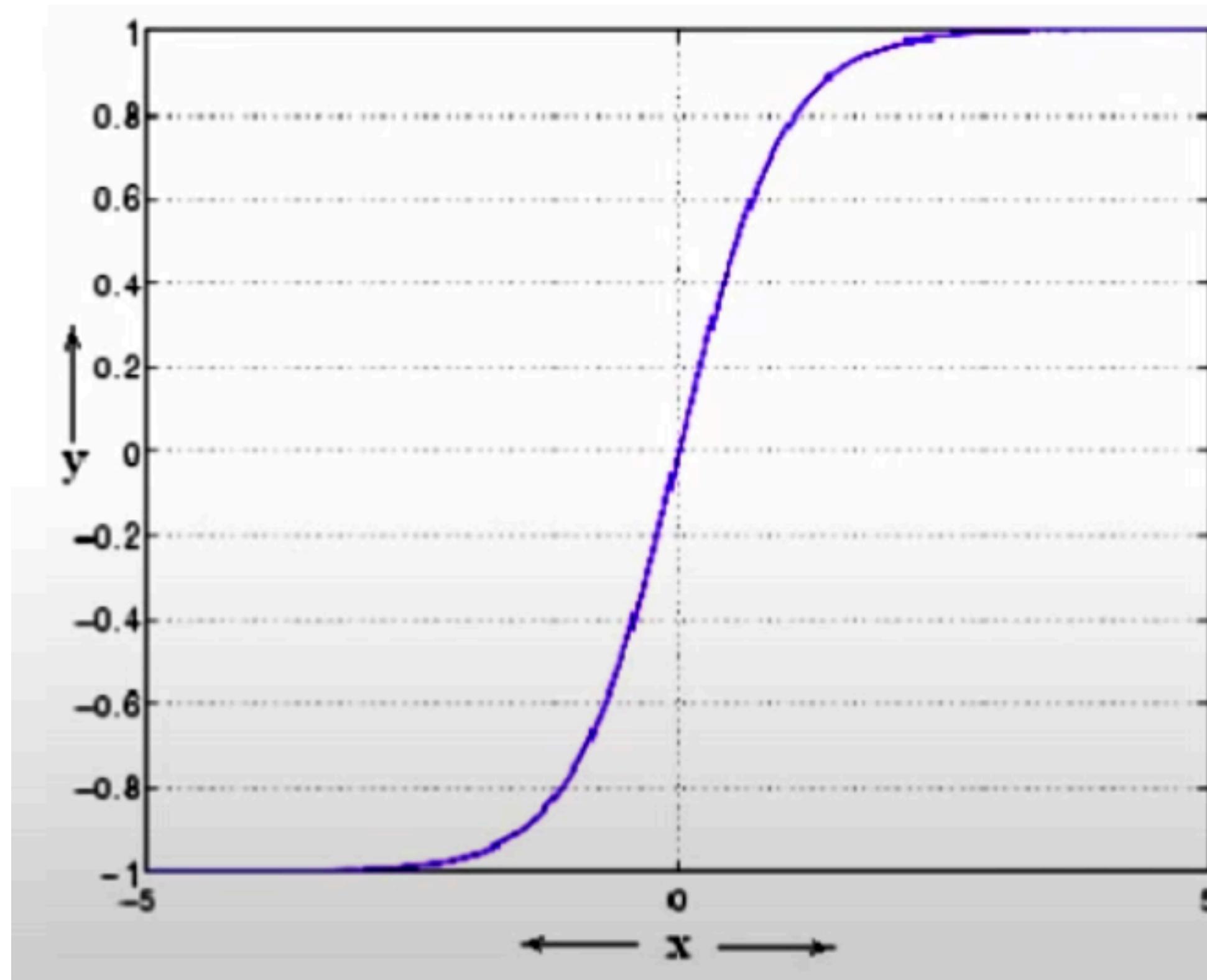
Disparition du gradient (the gradient vanishing problem)

- Lorsque l'entrée est trop grande (ou trop petite), la dérivée de la fonction logistique tend vers 0.

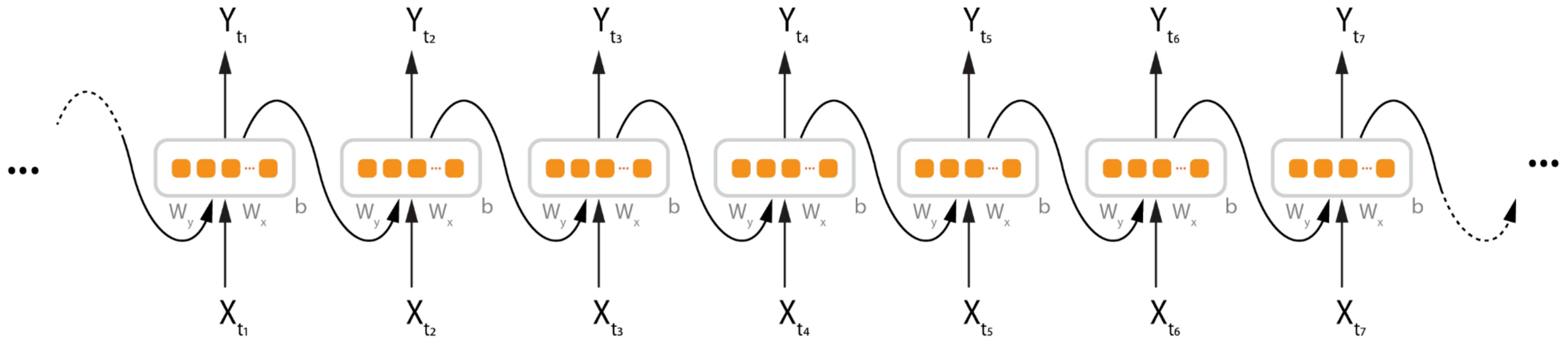


- Durant la rétro-propagation, le gradient a la tendance à diminuer progressivement. Cela implique que les neurones situés dans les couches basses apprennent à un rythme beaucoup plus lent que ceux dans les couches supérieures, ce qui est appelé le phénomène de 'vanishing gradient'.

Disparition du gradient (the gradient vanishing problem)



Une des limites du RNN



Convergence lente,
Mémoire courte,
Gradients qui disparaissent / explosent

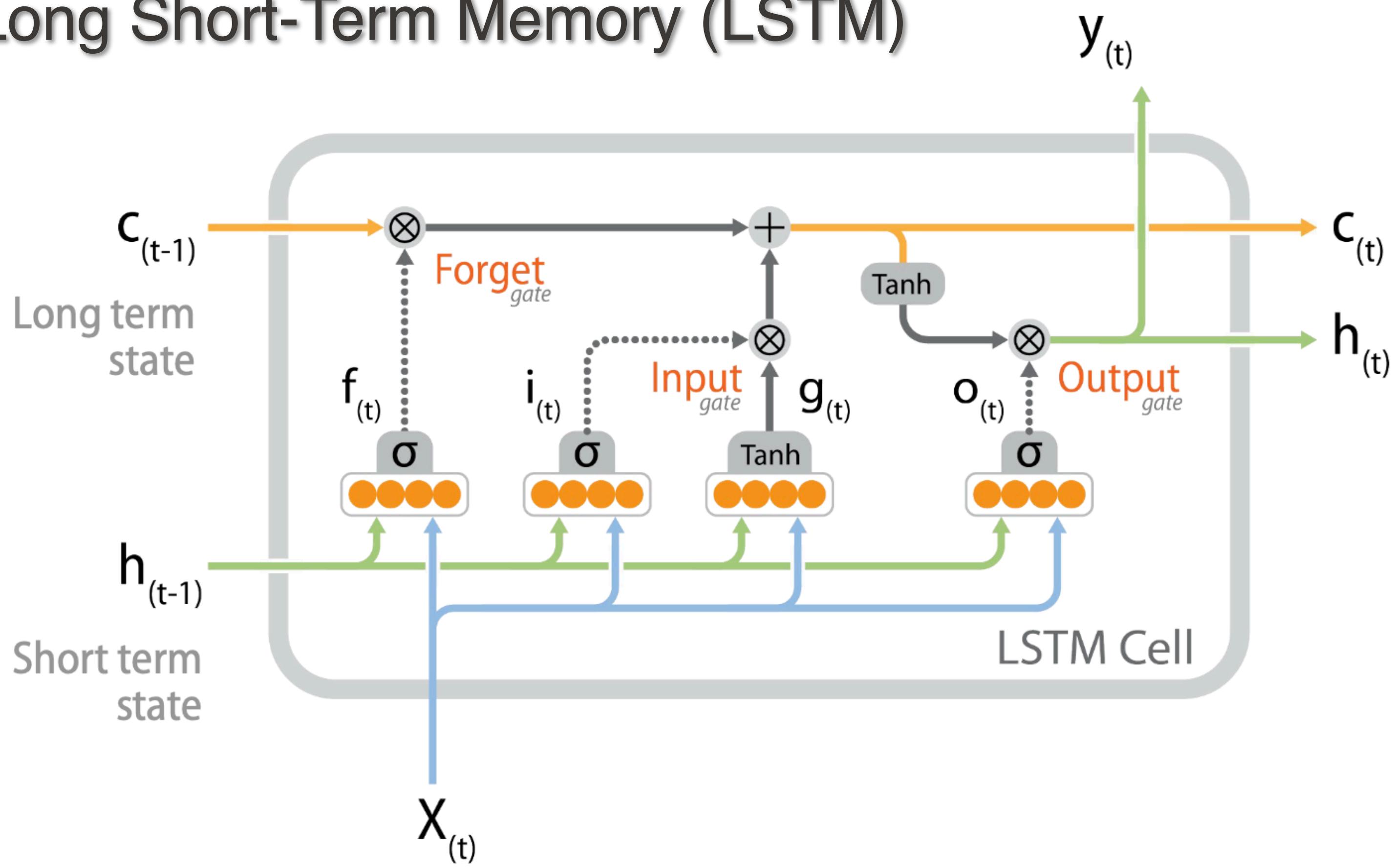
=>

En général, ça ne fonctionne pas



Long short-term memory (LSTM)

Long Short-Term Memory (LSTM)



$$f_{(t)} = \sigma(W_{xf}^T X_{(t)} + W_{hf}^T h_{(t-1)} + b_f)$$

$$i_{(t)} = \sigma(W_{xi}^T X_{(t)} + W_{hi}^T h_{(t-1)} + b_i)$$

$$g_{(t)} = \tanh(W_{xg}^T X_{(t)} + W_{hg}^T h_{(t-1)} + b_g)$$

$$o_{(t)} = \sigma(W_{xo}^T X_{(t)} + W_{ho}^T h_{(t-1)} + b_o)$$

$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)}$$

$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})$$

Avec :

$X_{(t)} \in \mathbb{R}^d$ Vecteur d'entrée

$f_{(t)} \in \mathbb{R}^h$ Vecteur d'activation de la forget gate's

$i_{(t)} \in \mathbb{R}^h$ Vecteur d'activation de la input gate's

$o_{(t)} \in \mathbb{R}^h$ Vecteur d'activation de la output gate's

$g_{(t)} \in \mathbb{R}^h$ vecteur d'entrée actuel

$h_{(t)}, y_{(t)} \in \mathbb{R}^h$ hidden state ou vecteur de sortie

$c_{(t)} \in \mathbb{R}^h$ Vecteur d'état de la cellule

\otimes hadamard product

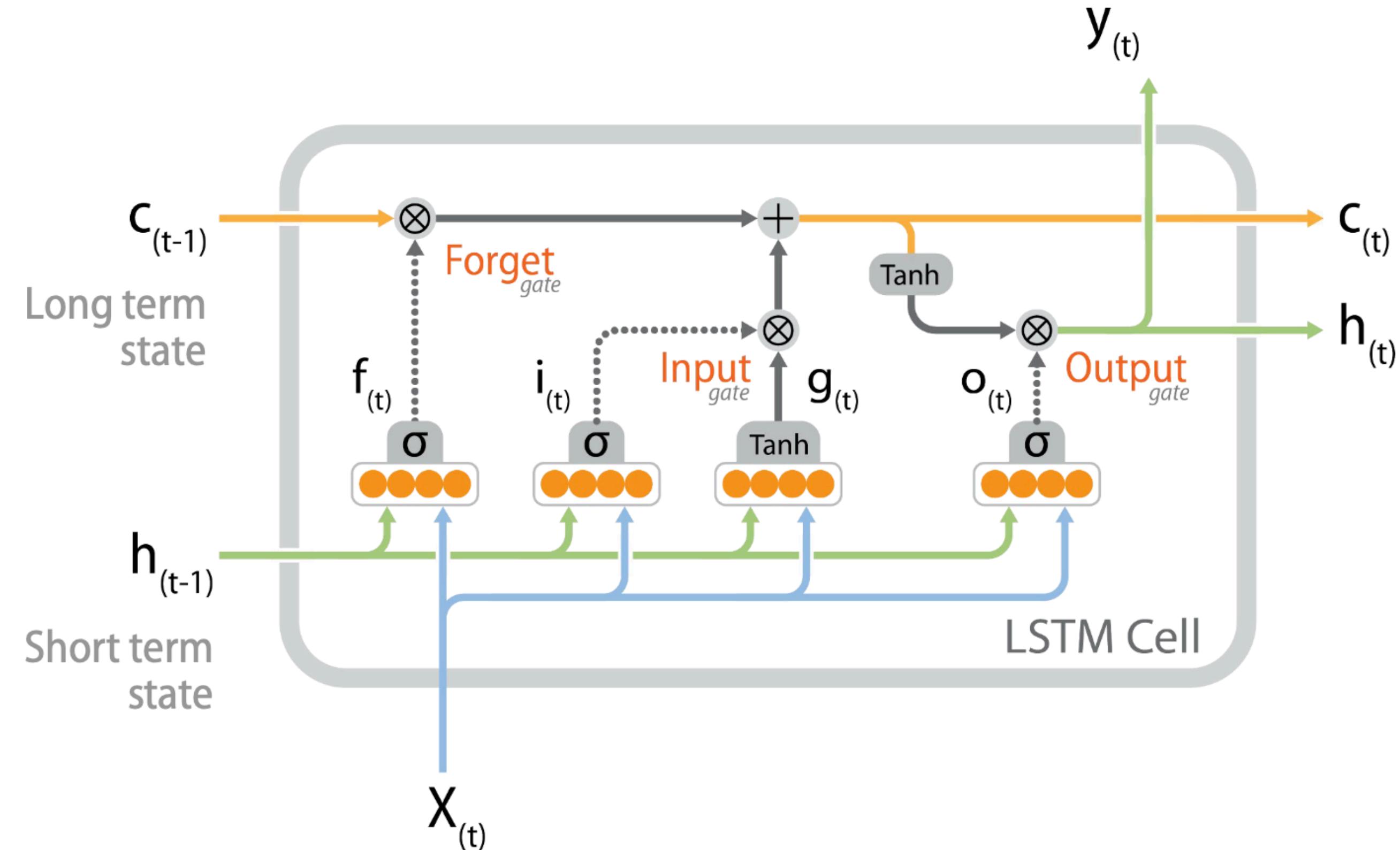
σ fonction sigmoid

W_y matrice de poids

b vecteur biais

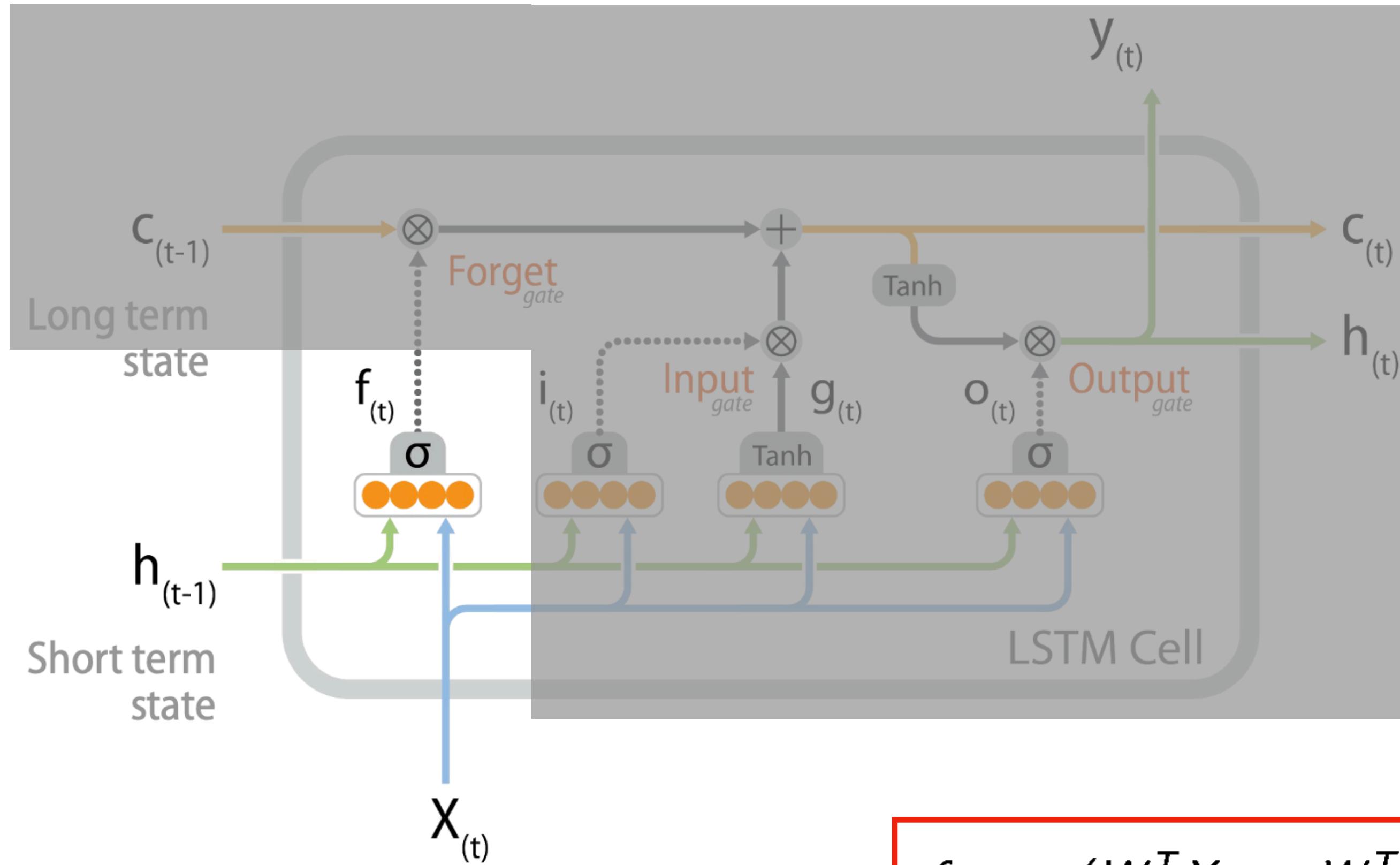
Long Short-Term Memory (LSTM)

- **Forget Gate** : Capacité à supprimer l'information inutile.
- **Input Gate** : Capacité à intégrer de nouvelles informations pertinentes.
- **Output Gate** : État actuel de la cellule à l'instant t.



Long Short-Term Memory (LSTM) : Forget Gate

- **Forget Gate** : Capacité à supprimer l'information inutile.



$$f_{(t)} = \sigma(W_{xf}^T X_{(t)} + W_{hf}^T h_{(t-1)} + b_f)$$

```
>>> c_prev.shape
```

```
(5,)
```

```
>>> h_prev.shape
```

```
(5,)
```

```
>>> x.shape
```

```
(4,)
```

```
>>> x_h_prev = np.hstack((x, h_prev))
```

```
>>> x_h_prev.shape
```

```
(9,)
```

```
>>> Wf.shape
```

```
(9, 5)
```

```
>>> bf.shape
```

```
(5,)
```

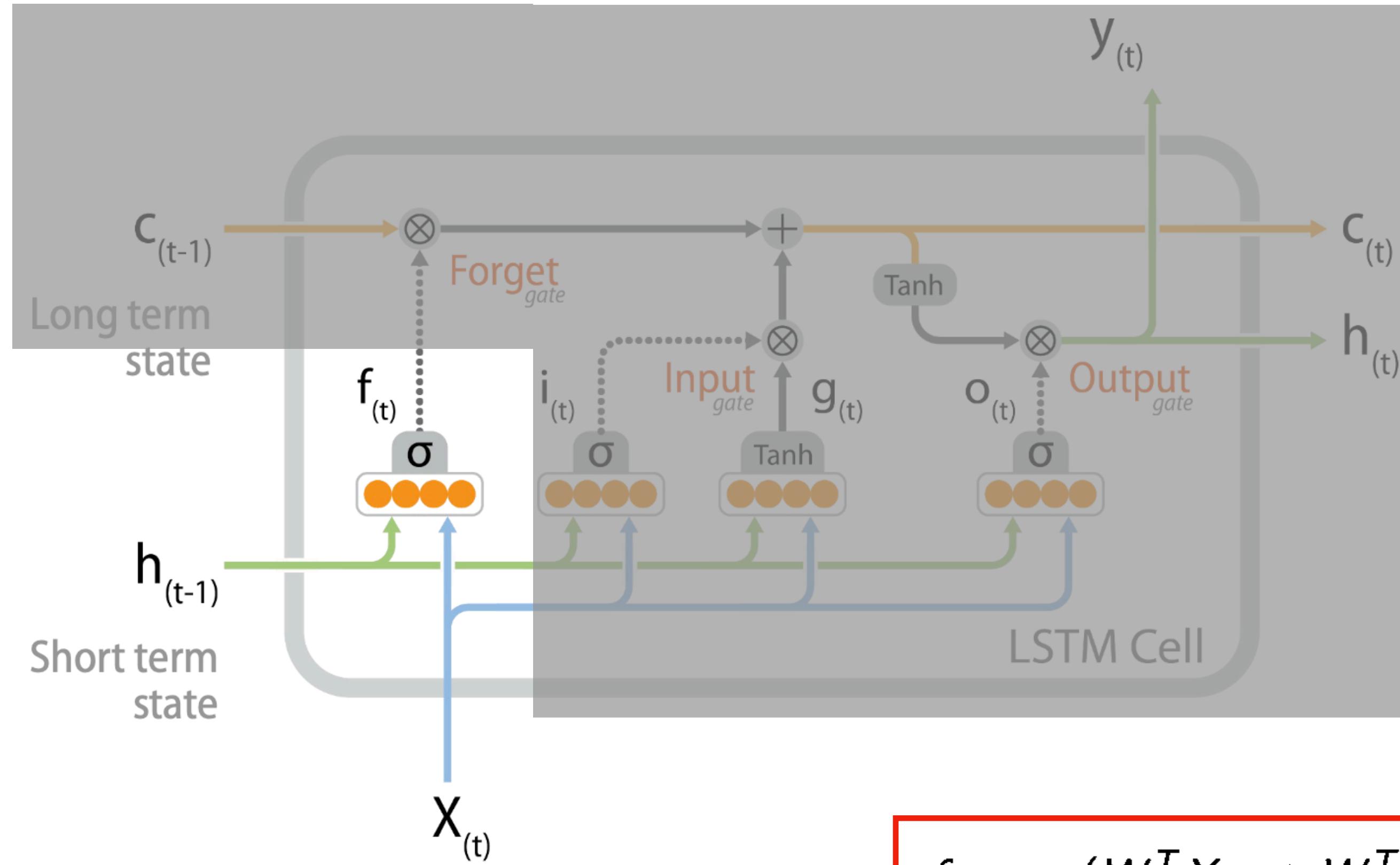
```
>>> ft = sigmoid(np.dot(x_h_prev, Wf) + bf)
```

```
>>> ft.shape
```

```
(5,)
```

Long Short-Term Memory (LSTM) : Forget Gate

- **Forget Gate** : Capacité à supprimer l'information inutile.

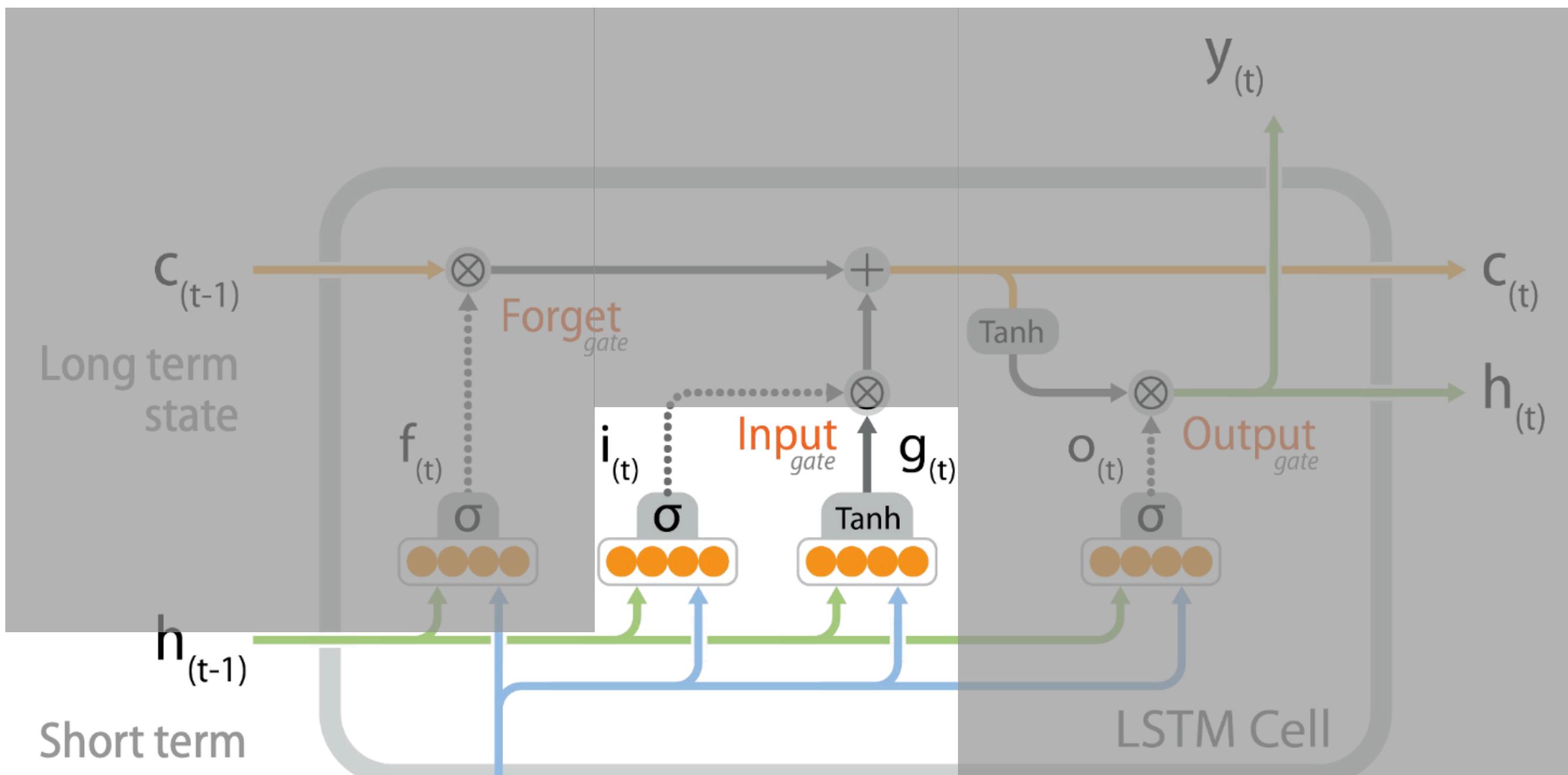


$$f_{(t)} = \sigma(W_{xf}^T X_{(t)} + W_{hf}^T h_{(t-1)} + b_f)$$

```
>>> I_t
array([0.00605241, 0.02419927, 0.12958965, 0.83141943,
0.5440948 ])
>>> c_prev
array([ 0.38000574, 1.13691447, 1.57618308, -1.01247179,
1.02257568])
>>> c_prev_forget = ft*c_prev
>>> c_prev_forget.shape
(5,)
>>> c_prev_forget.shape
(5,)
>>>
```

Long Short-Term Memory (LSTM) : Input Gate

- **Input Gate** : Capacité à intégrer de nouvelles informations pertinentes.



$$i_{(t)} = \sigma(W_{xi}^T X_{(t)} + W_{hi}^T h_{(t-1)} + b_i)$$

$$g_{(t)} = \tanh(W_{xg}^T X_{(t)} + W_{hg}^T h_{(t-1)} + b_g)$$

```
>>> Ct = np.tanh(np.dot(x_h_prev, Wc) + bc)
>>> Ct
array([-0.17806474, -0.99993564, 0.99164565, 0.92774236,
-0.99527522])
>>> Ct.shape
(5,)
```

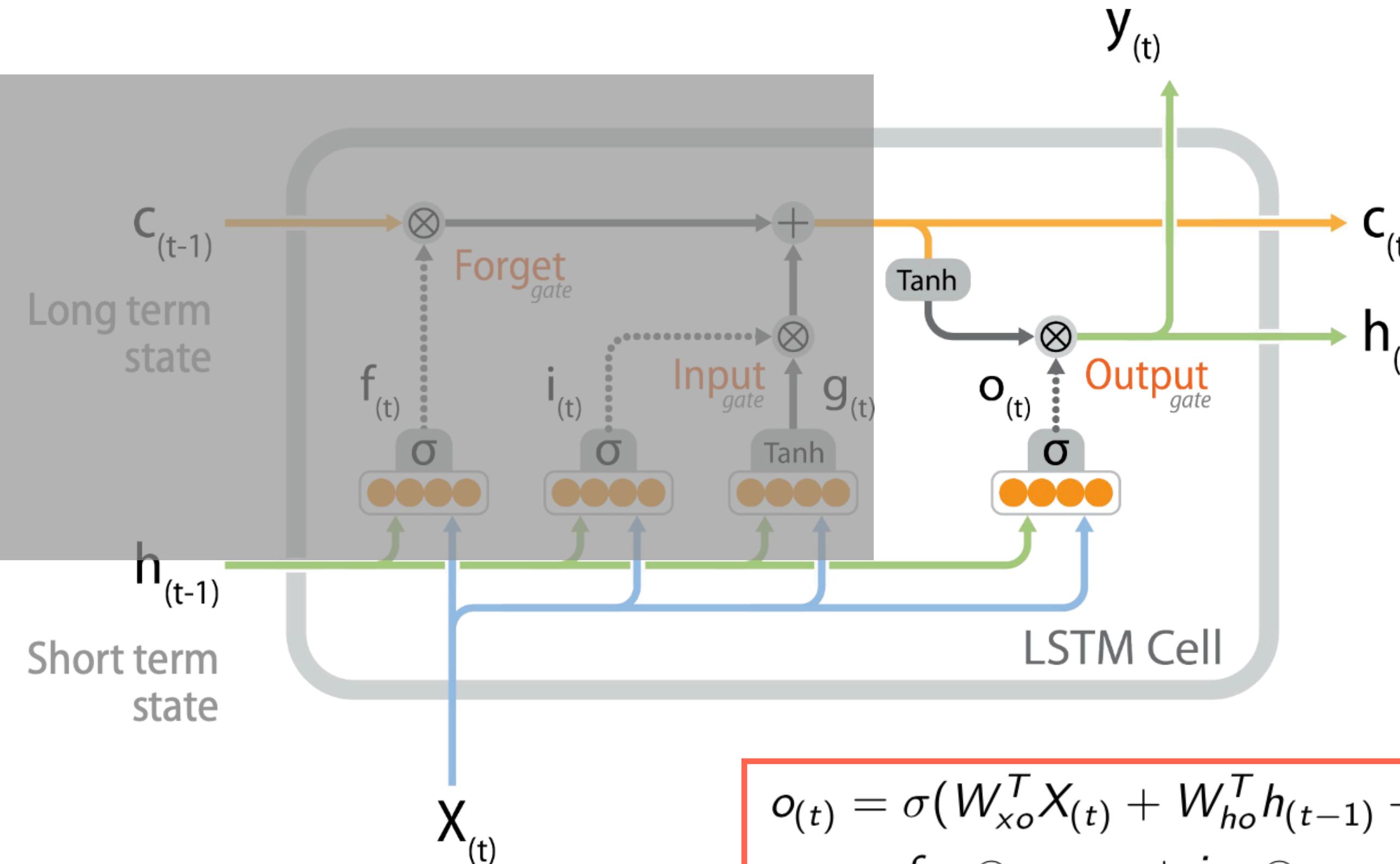
```
>>> it = sigmoid(np.dot(x_h_prev, Wi) + bi)
>>> it
array([0.00798643, 0.92300084, 0.22905397, 0.27818745,
0.96195338])
>>> it.shape
(5,)
```

```
>>> new_c = it*Ct
>>> new_c
array([-0.0014221 , -0.92294144, 0.22714037, 0.25808628,
-0.95740836])
>>> new_c.shape
(5,)
```

```
>>> c = c_prev*ft + it*Ct
>>> c.shape
(5,)
>>> c
```

Long Short-Term Memory (LSTM) : Output Gate

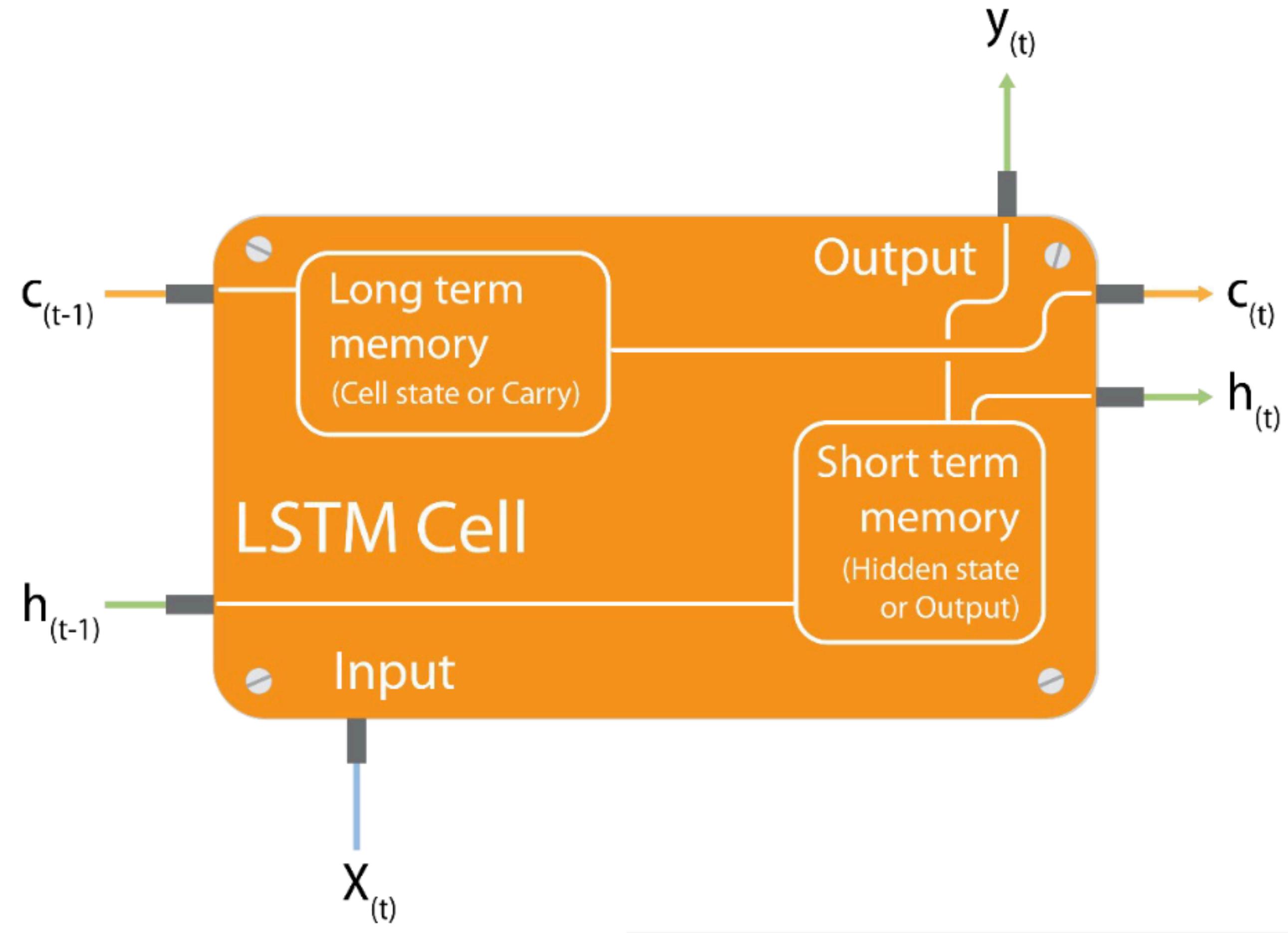
- **Output Gate** : État actuel de la cellule à l'instant t.



$$\begin{aligned} o_{(t)} &= \sigma(W_{xo}^T X_{(t)} + W_{ho}^T h_{(t-1)} + b_o) \\ c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\ y_{(t)} &= h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)}) \end{aligned}$$

```
>>> ot = sigmoid(np.dot(x_h_prev, Wo) + bo)
>>> ot.shape
(5,)
>>> h = ot * np.tanh(c)
>>> h.shape
(5,)
```

Long Short-Term Memory (LSTM)

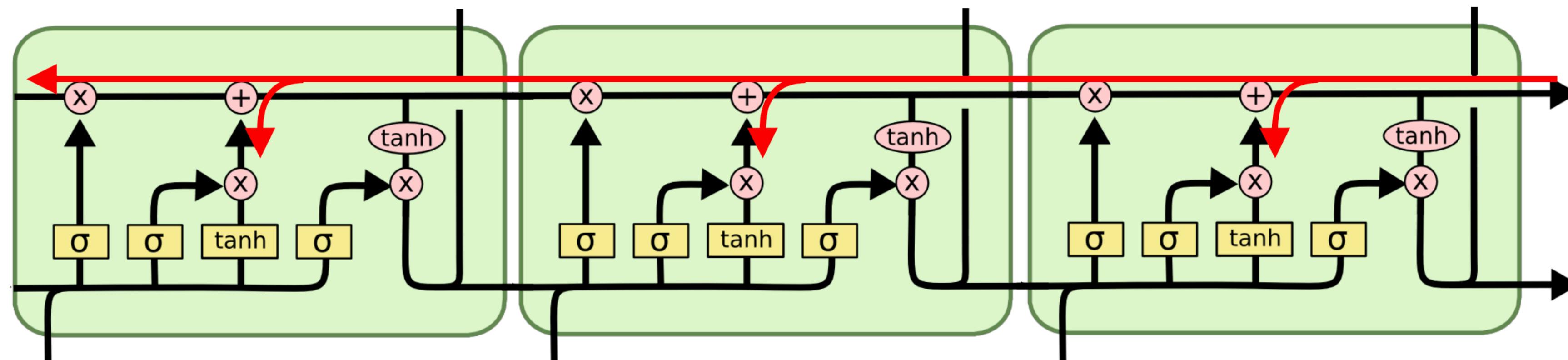


- La cellule c_t est affectée lentement (**slow state**)
- L'état h_t est affecté plus rapidement (**fast state**)

Long Short-Term Memory (LSTM) : Flot du gradient

Le **gradient** se propage de manière plus efficace que dans les RNN, notamment à travers la cellule, en prenant pour exemple ResNet.

Contrairement aux **RNN**, où le gradient doit traverser un *tanh*

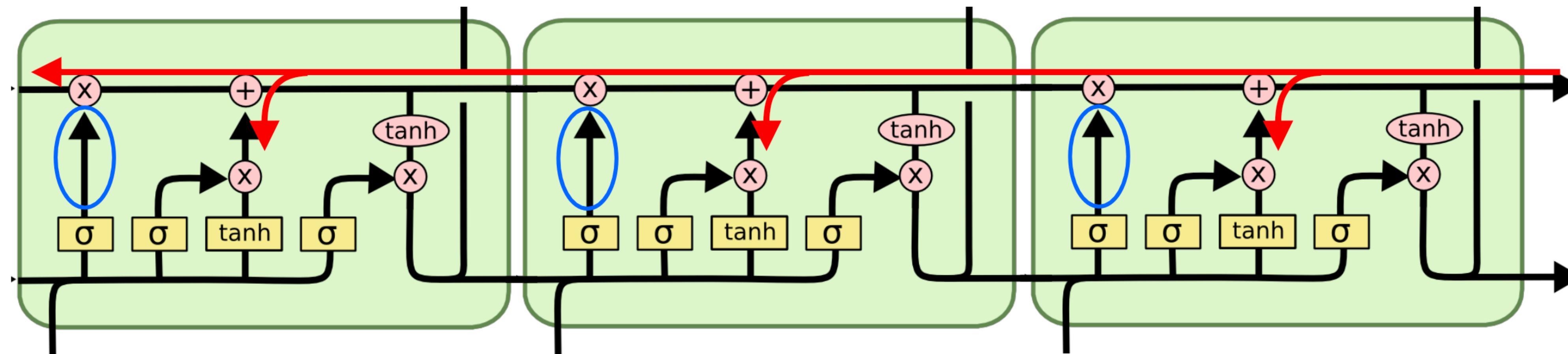


En tenant compte également du fait que la porte d'oubli (forget gate) ait des valeurs d'entrée proches de 1.

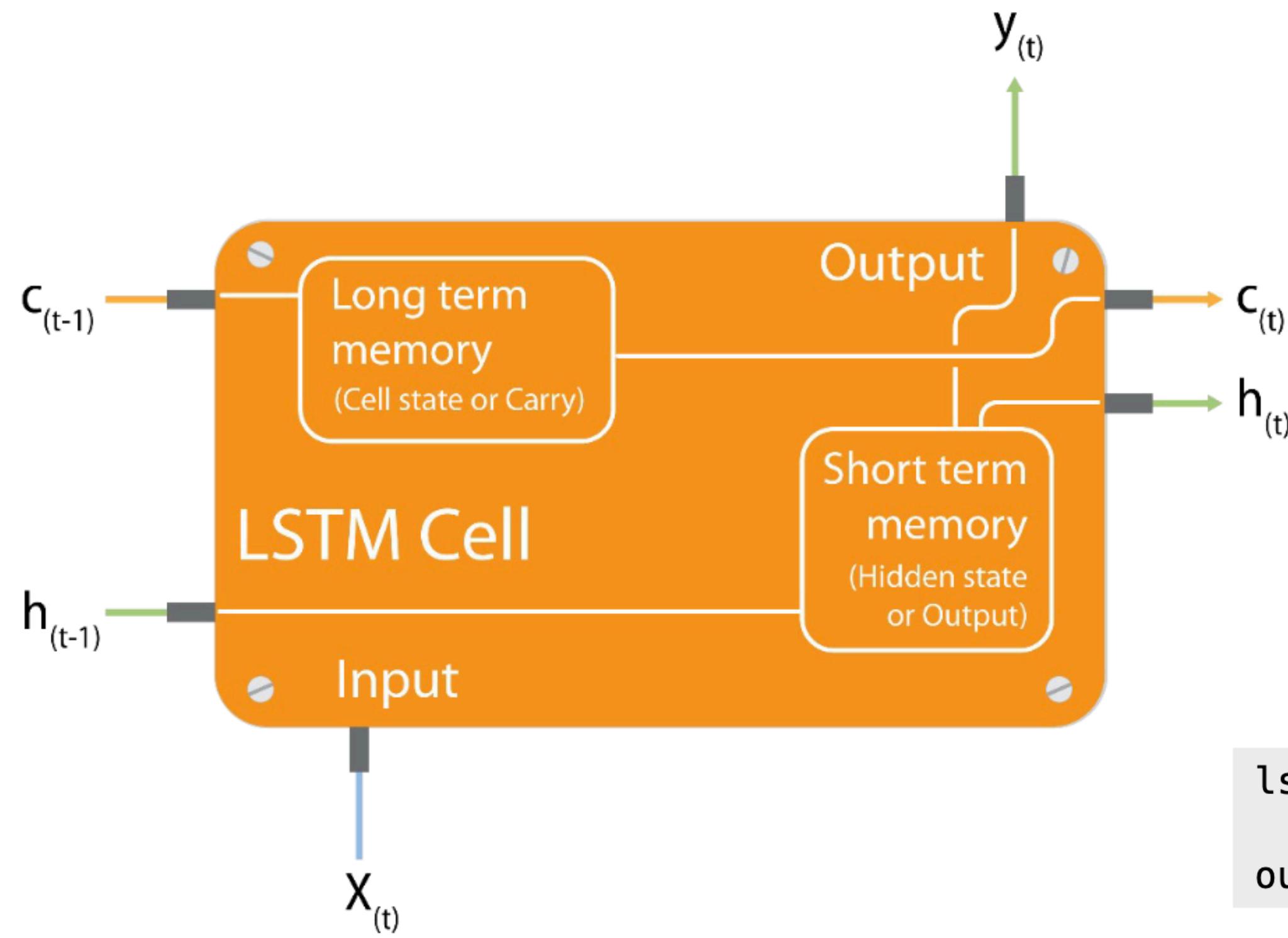
Long Short-Term Memory (LSTM) : Flot du gradient

Le **gradient** est multiplié par la sortie du **forget gate**

- qui adopte des valeurs différentes à chaque itération
- ce qui évite le problème du **gradient**



Long Short-Term Memory (LSTM)



```
inputs = tf.random.normal([32, 20, 8])
lstm = tf.keras.layers.LSTM(16)
output = lstm(inputs)
```

Serie to vector

Inputs shape is : (32, 20, 8)
Output shape is : (32, 16)

```
lstm = tf.keras.layers.LSTM(18, return_sequences=True, return_state=True)
output, memory_state, carry_state = lstm(inputs)
```

Serie to serie

Output shape : (32, 20, 18)
Memory state : (32, 18)
Carry state : (32, 18)