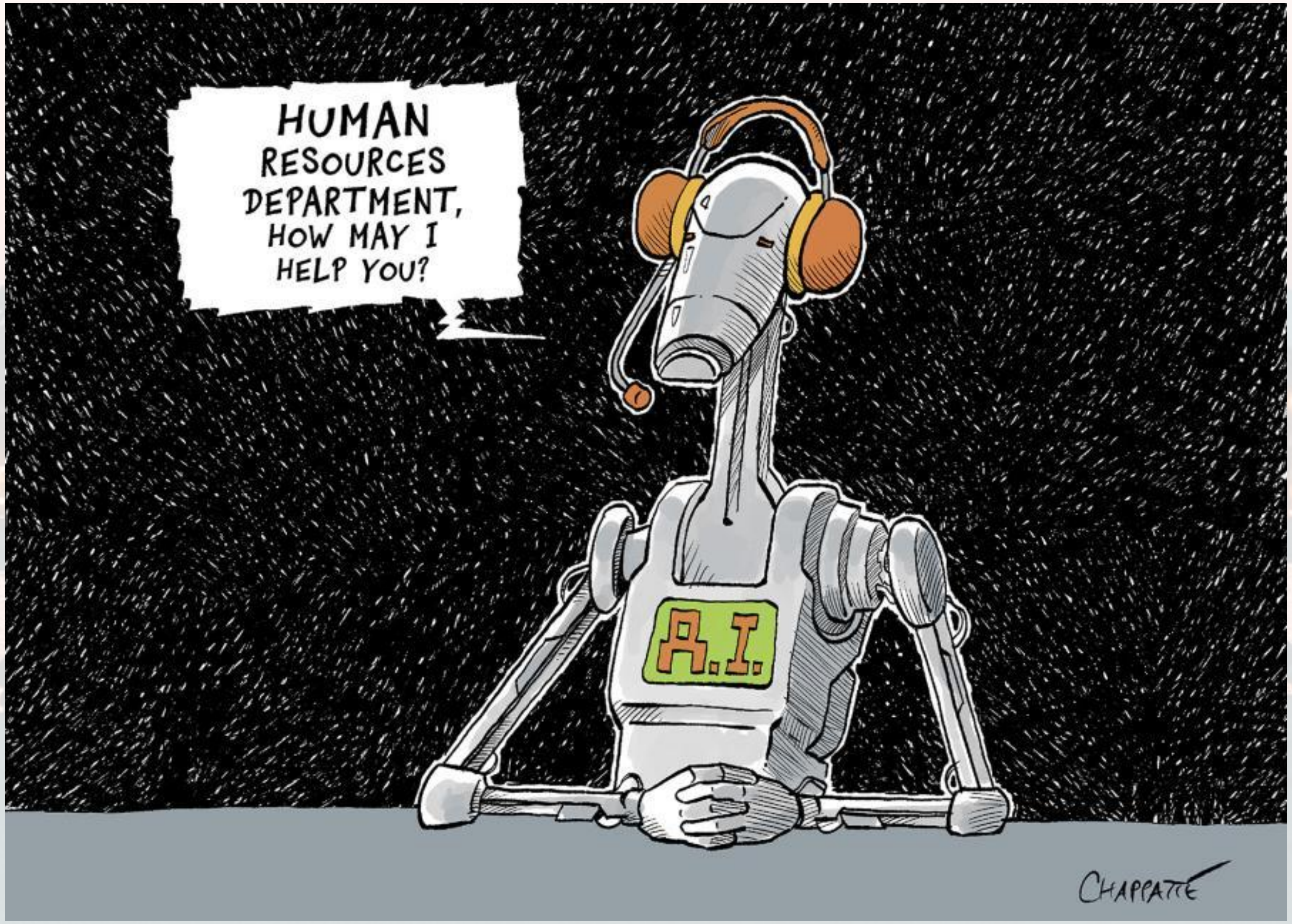
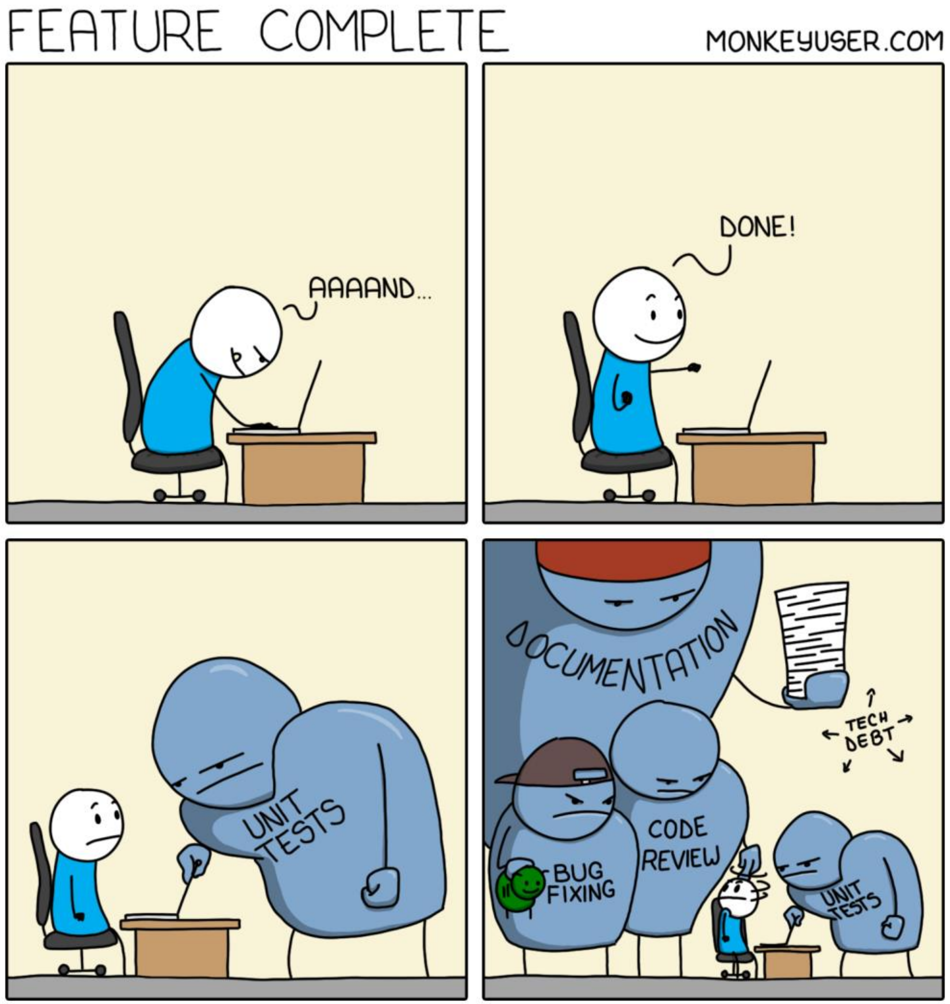
Recurrent Neural Networks – from Scratch



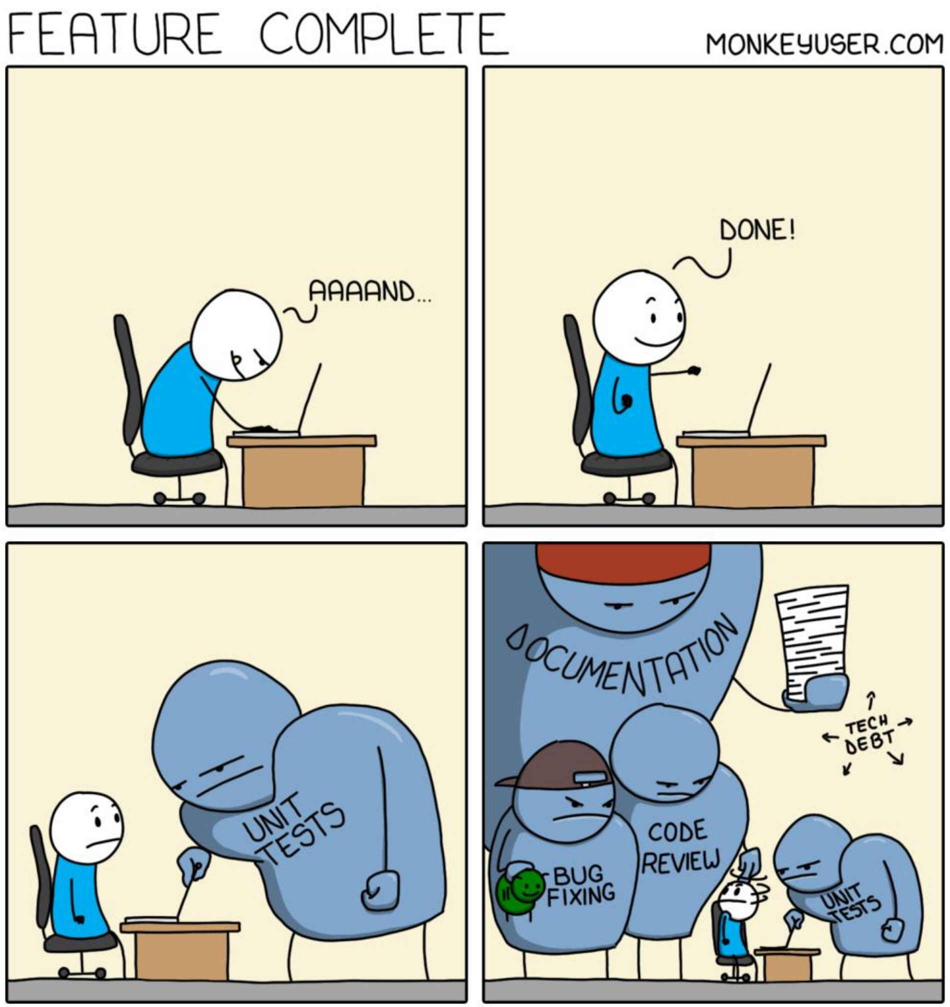


you should be fluent with:

- *basic* ***OOP*** *(methods, classes, inheritance)*
- ***linear algebra*** *(dot product, inner product, outer product)*
- ***derivatives*** *(gradient)*

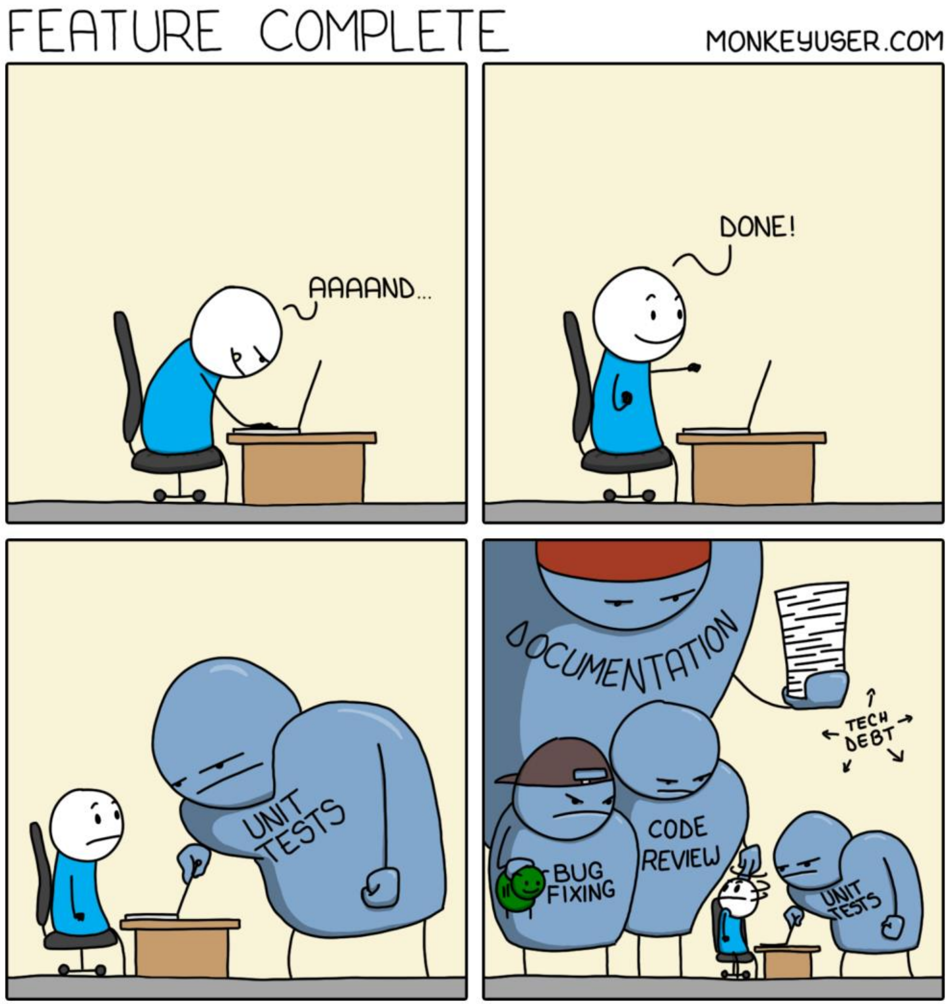
+

- ***Optimizer_SGD*** *from "ANN from Scratch"*



<u>outline:</u>

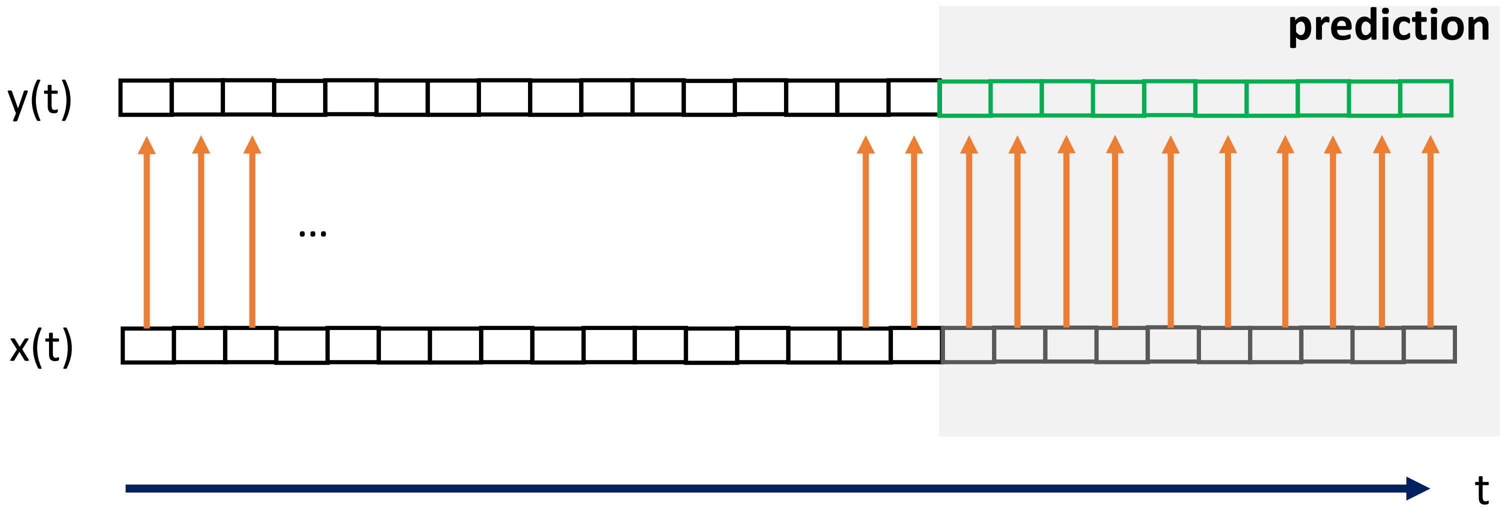
- *the idea*
- *the RNN cell*
- ***B**ack**P**ropagation **T**hrough **T**ime*
- *full backpropagtion*
- *creating an SGD optimizer*
- *creating a full package*

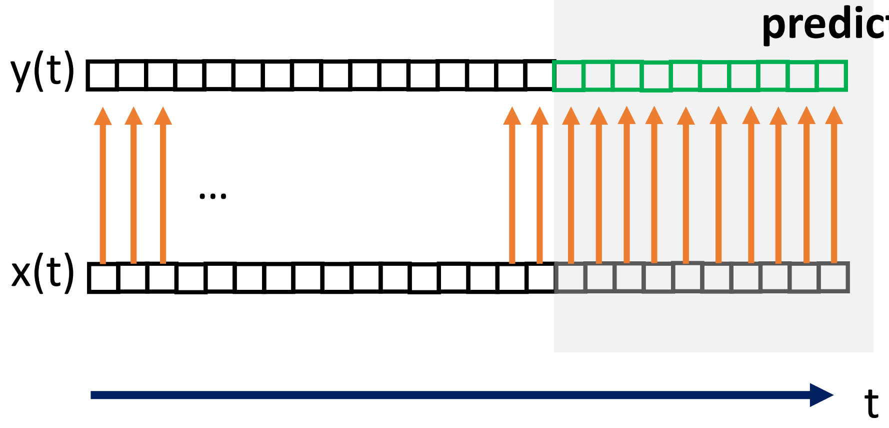


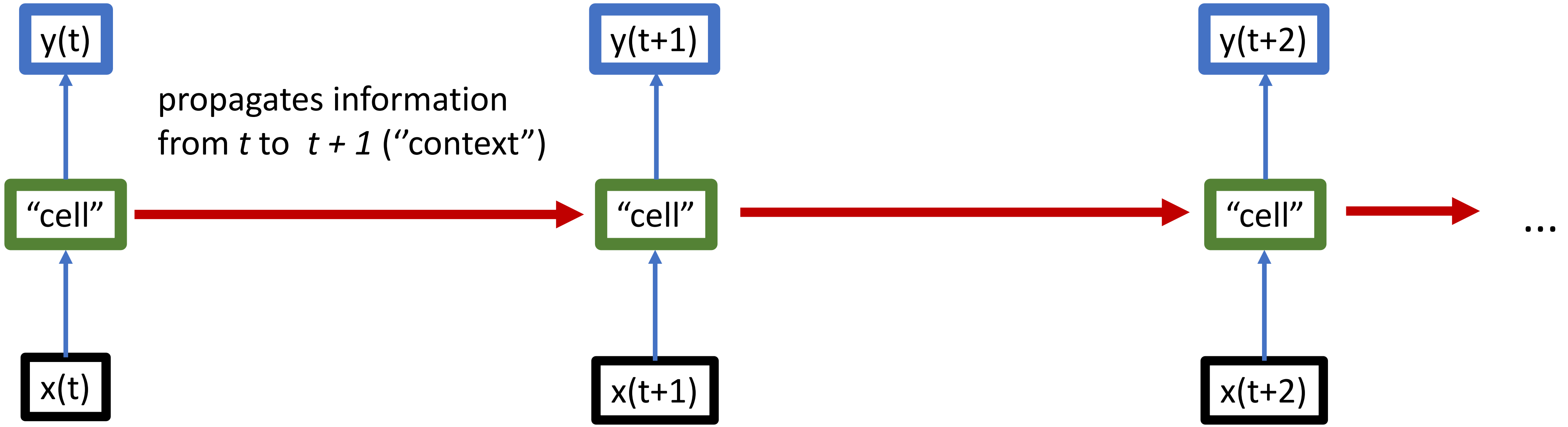
<u>outline:</u>

- *the idea*
- *the RNN cell*
- *BackPropagation Through Time*
- *full backpropagtion*
- *creating an SGD optimizer*
- *creating a full package*

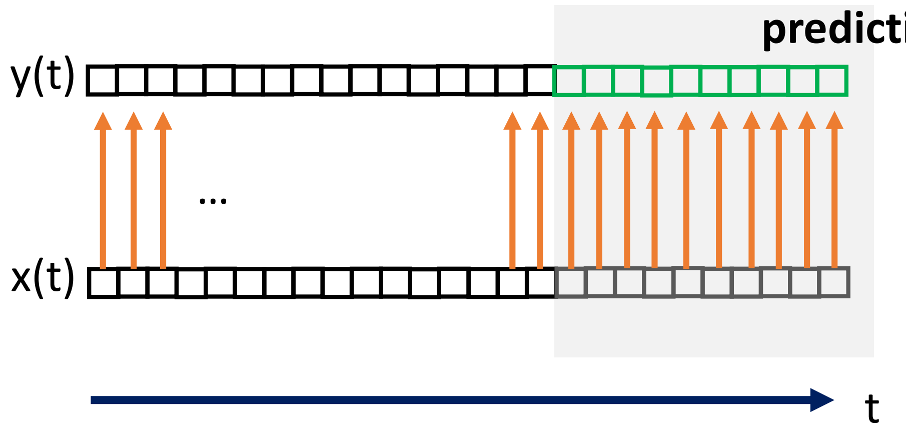
- time series analysis (prediction and forecasting)
- early speech recognition
- handwriting
- “precursor” of LSTMs
- invented by **Shun'ichi Amari** in 1972

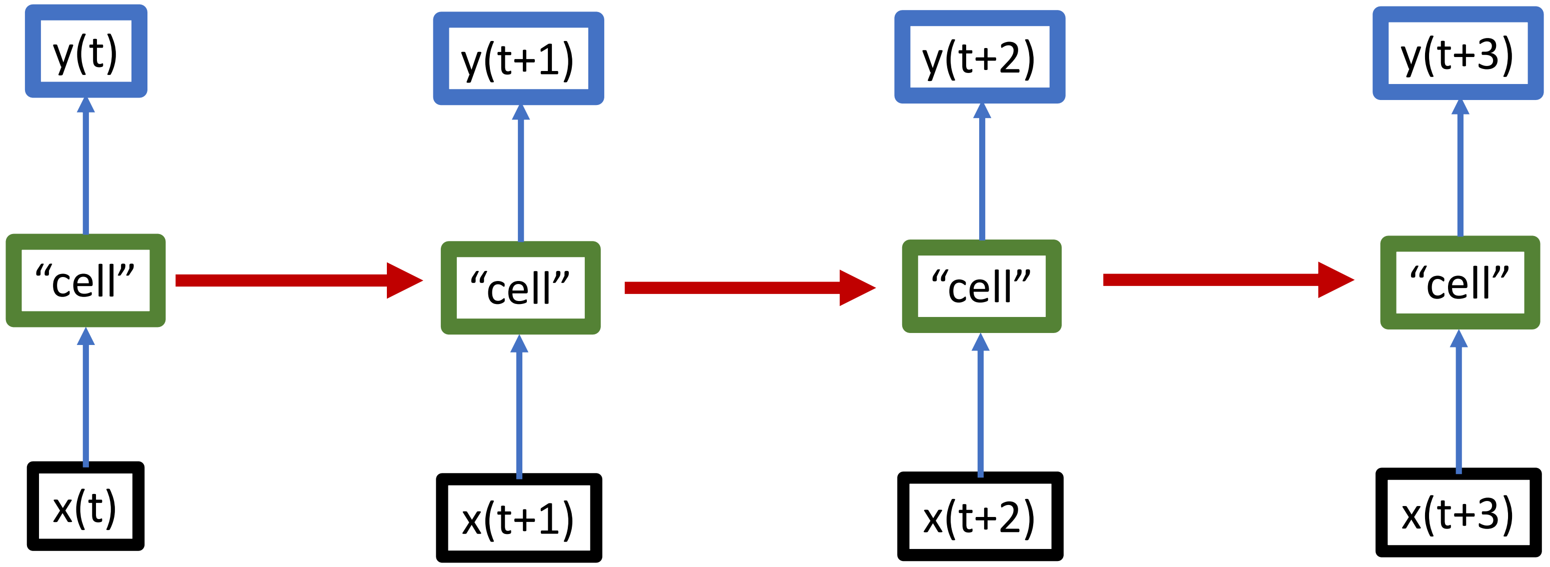


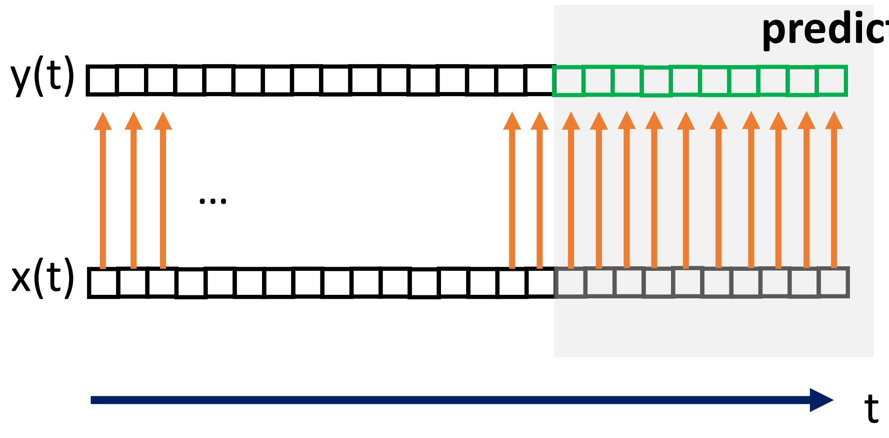

prediction
y(t)
...
x(t)
t


y(t)
y(t+1)
y(t+2)
propagates information
from *t* to *t + 1* ("context")
"cell"
"cell"
"cell"
...
x(t)
x(t+1)
x(t+2)

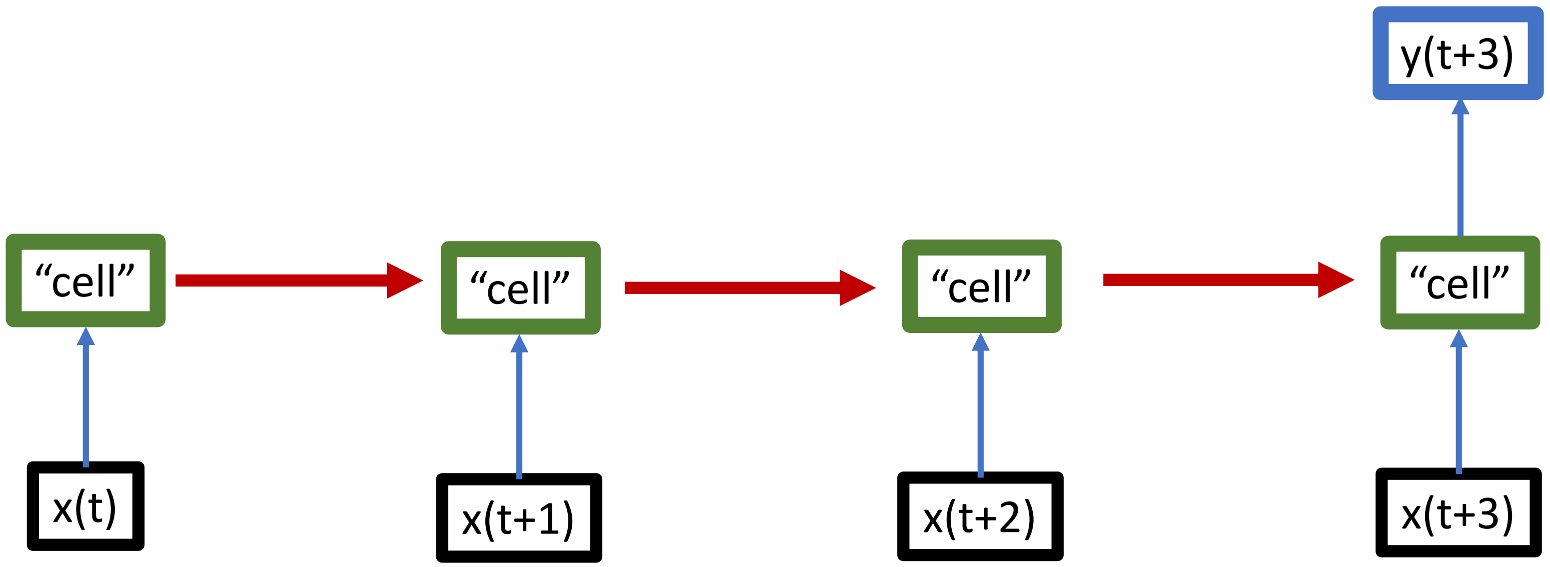
“many to many”

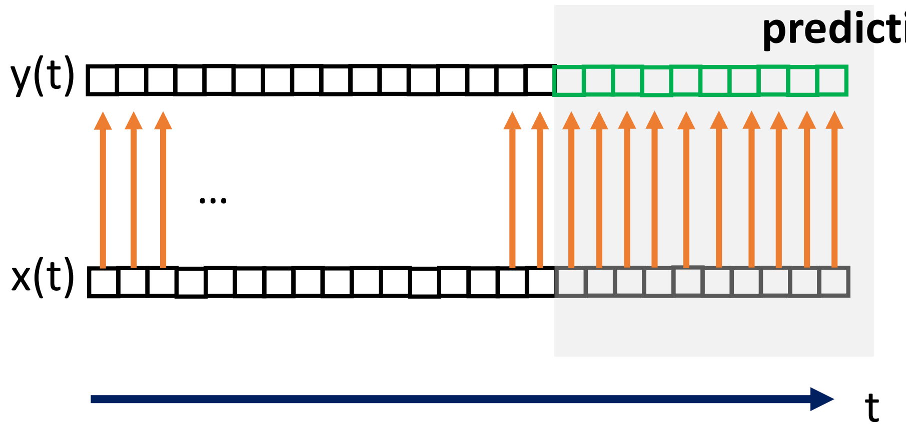

prediction
y(t)
...
x(t)
t


y(t)
y(t+1)
y(t+2)
y(t+3)
“cell”
“cell”
“cell”
“cell”
x(t)
x(t+1)
x(t+2)
x(t+3)

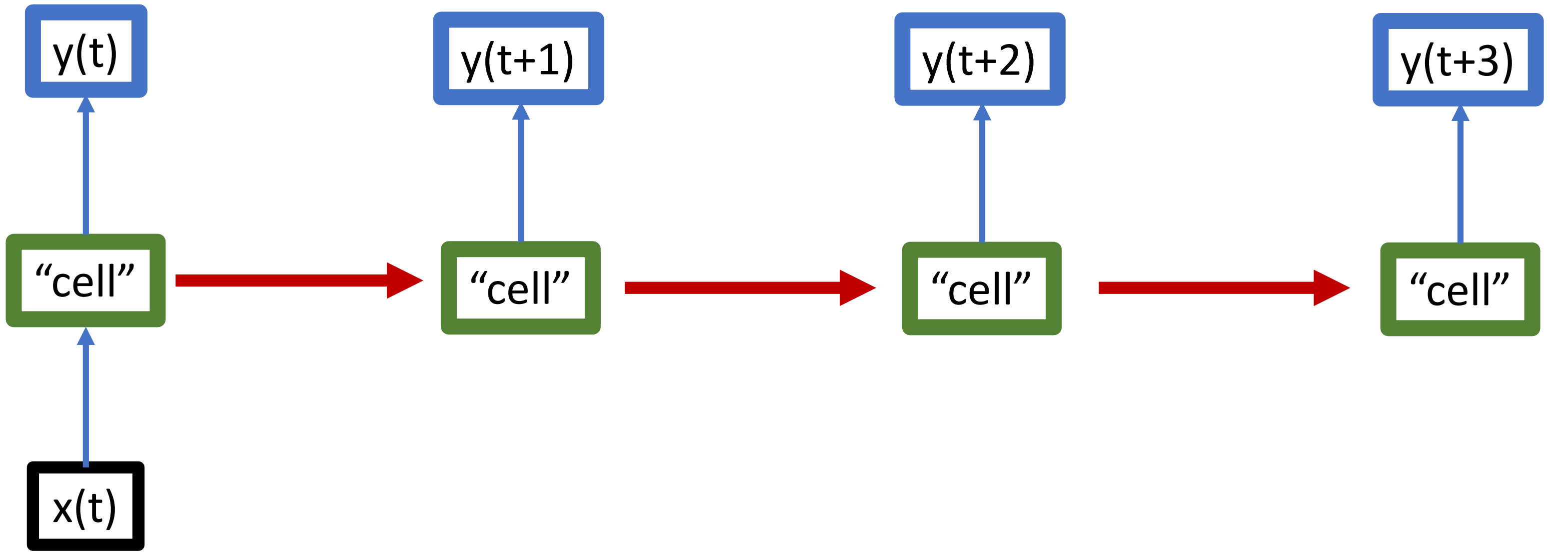


“many to one”





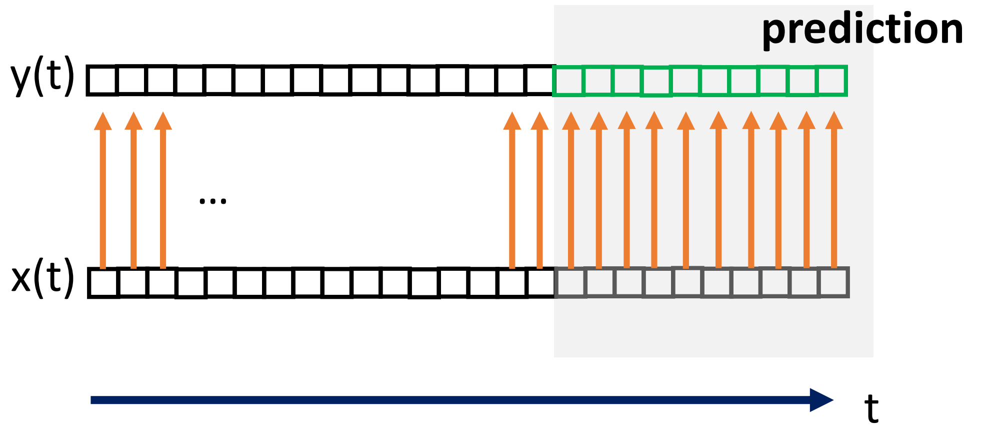
“one to many”

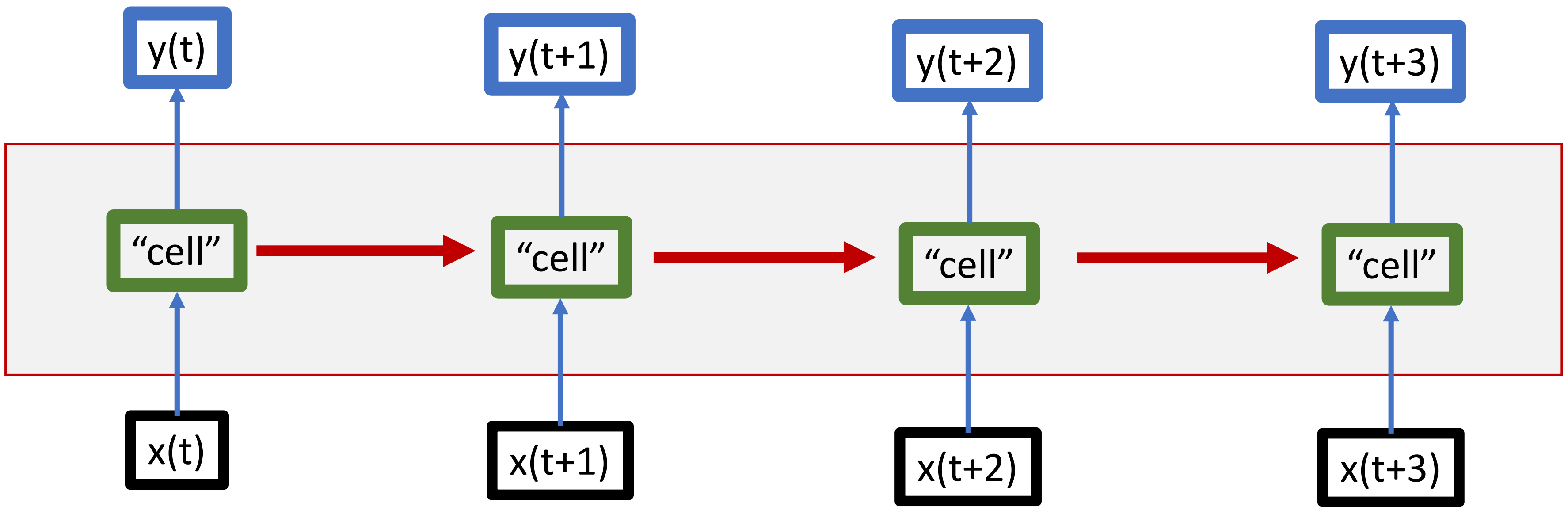


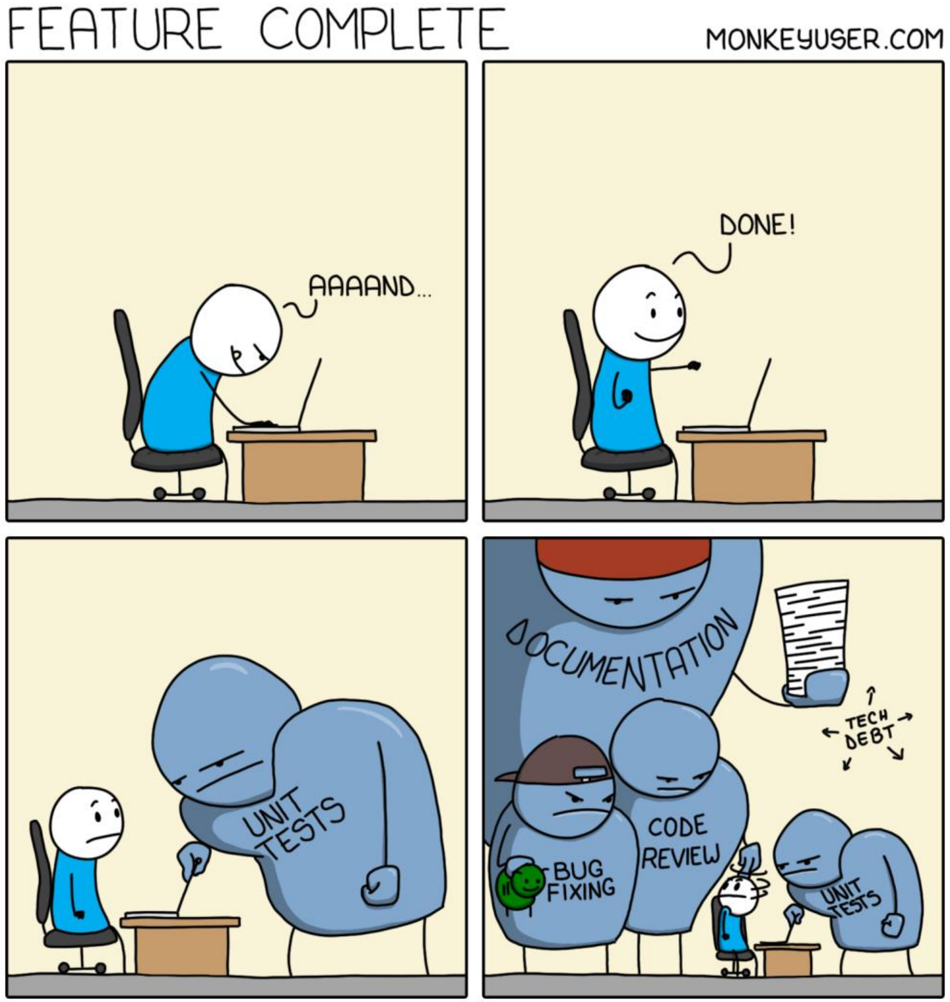
Applying the **identical** cell **recursively**!

→ easy to implement
→ direction (arrow of time, see later)

→ exploding/vanishing gradients
→ has a “short memory”



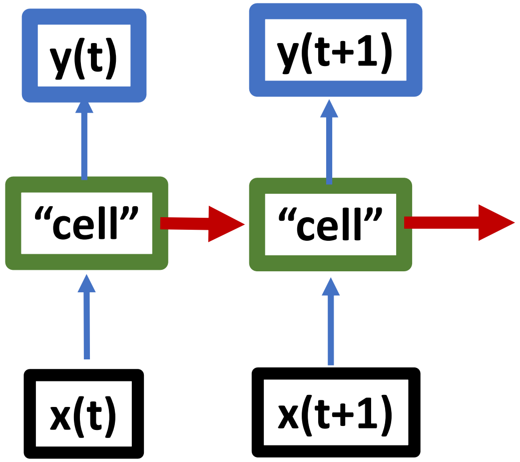


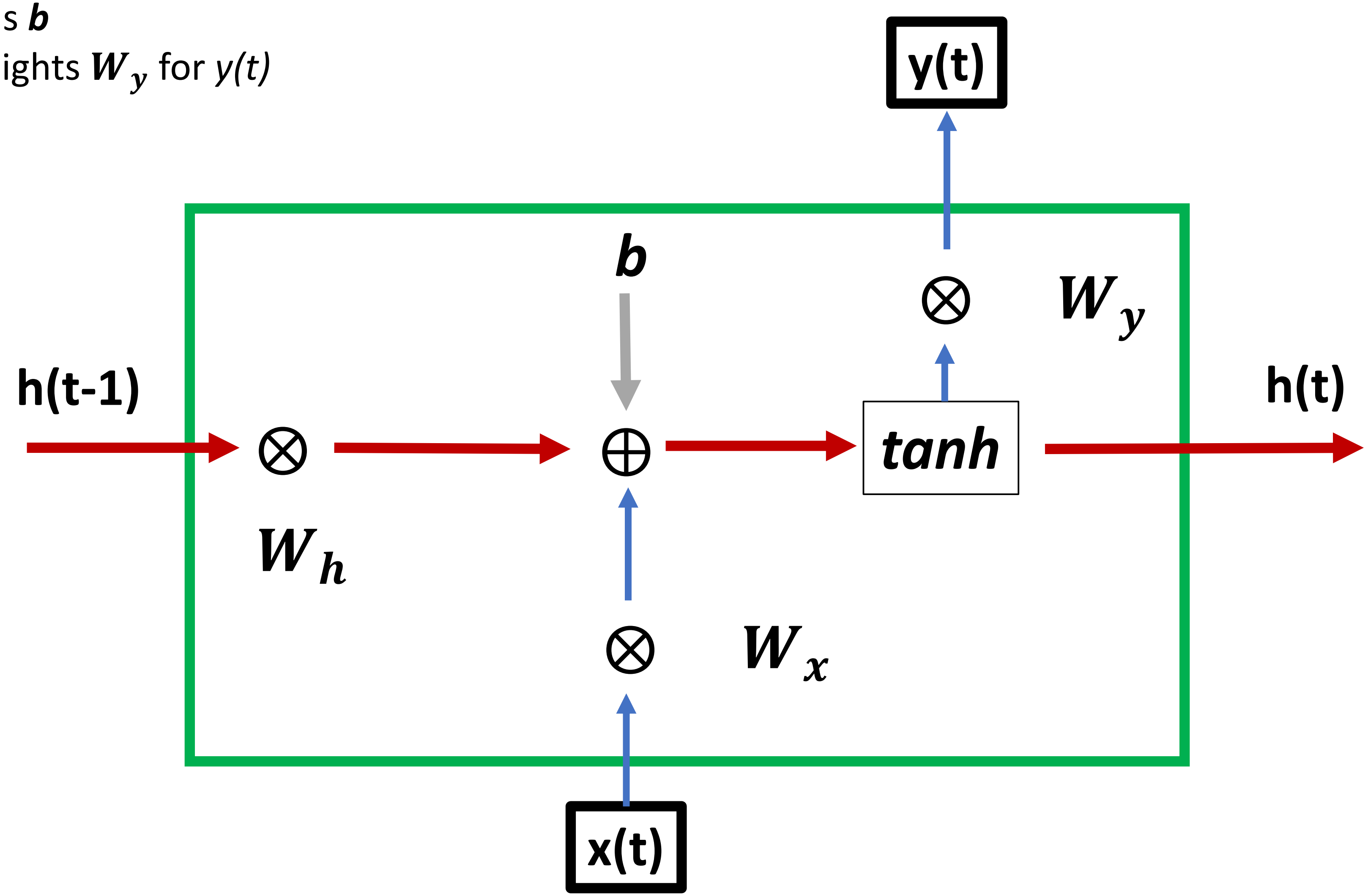


outline:

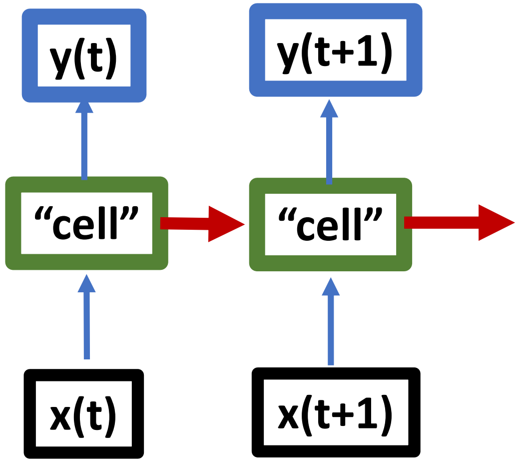
- *the idea*
- *the RNN cell*
- *BackPropagation Through Time*
- *full backpropagtion*
- *creating an SGD optimizer*
- *creating a full package*

- **state vector** *h(t)*, i. e. the *"memory"* the system has along time
- weights W_x for *x(t)*
- weights W_h for *h(t)*
- bias b
- weights W_y for *y(t)*



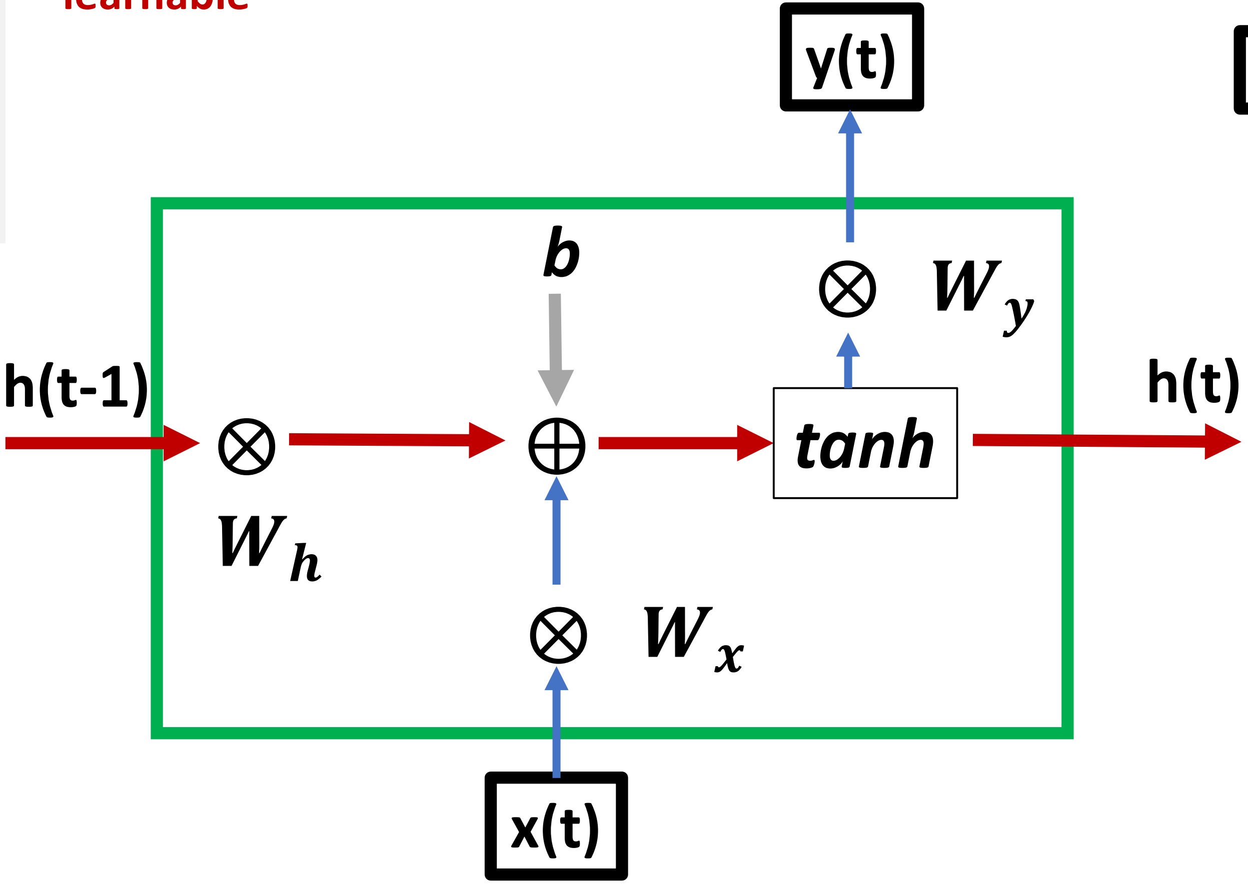


- **state vector** *h(t)*, i. e. the *"memory"* the system has along time



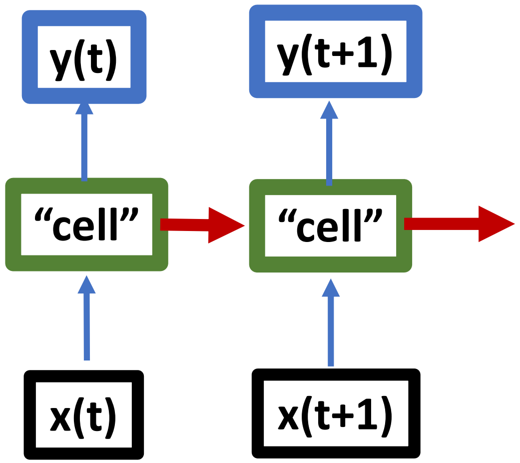
- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*

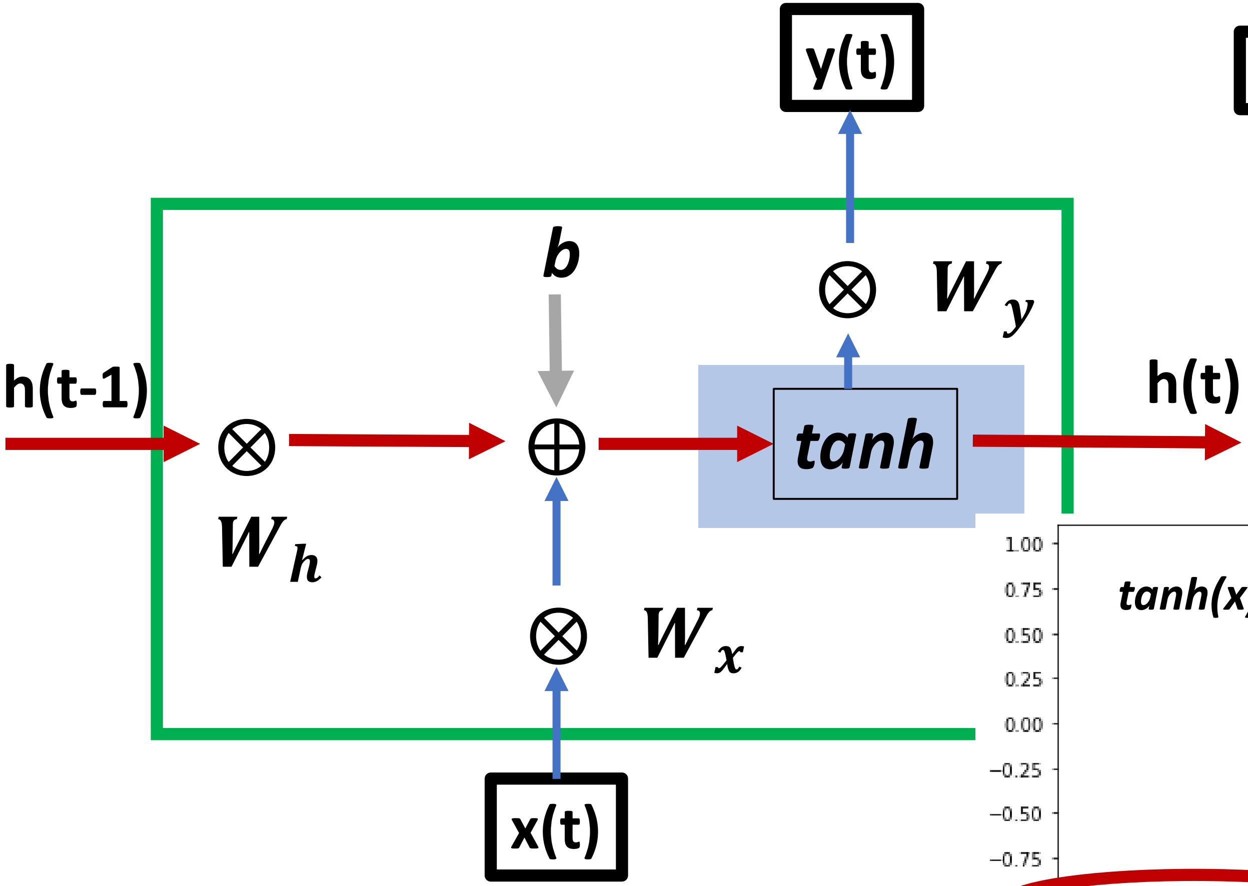
learnable

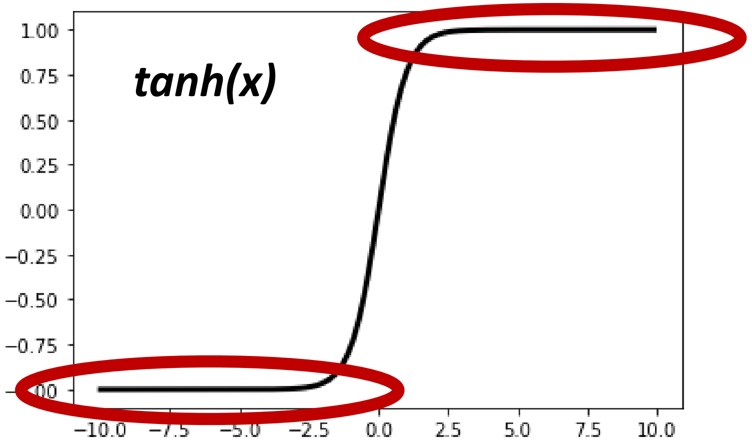


- **state vector** *h(t)*, i. e. the *"memory"* the system has along time

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *x(t)*
- bias ***b***
- weights $\boldsymbol{W_y}$ for *y(t)*







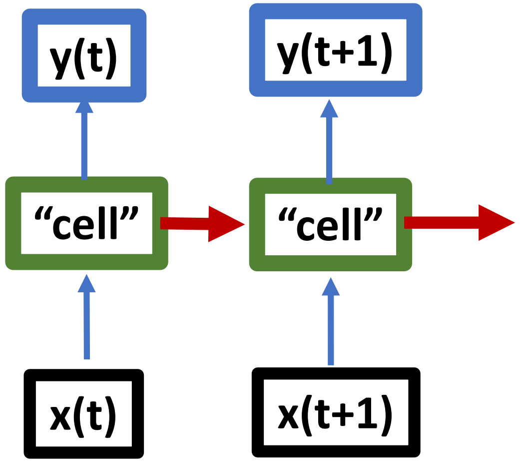
for large absolute values:

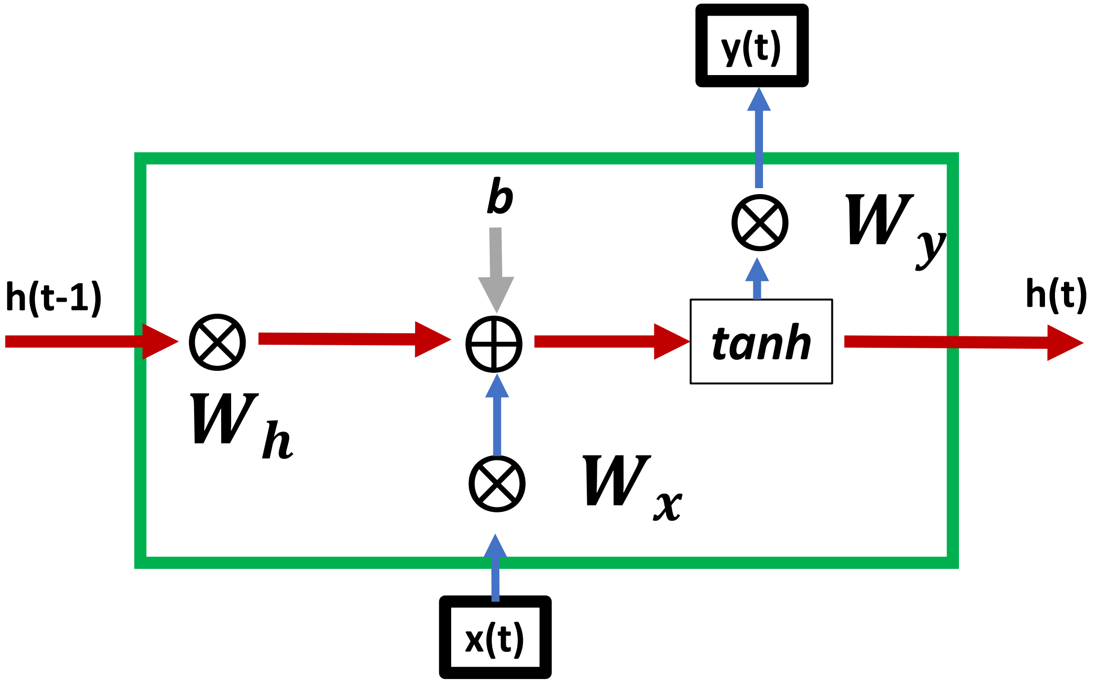
$$\frac{d\ tanh(x)}{dx} \approx 0$$

(see later backpropagation)

- **state vector** *h(t)*, i. e. the *“memory”* the system has along time

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *x(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*

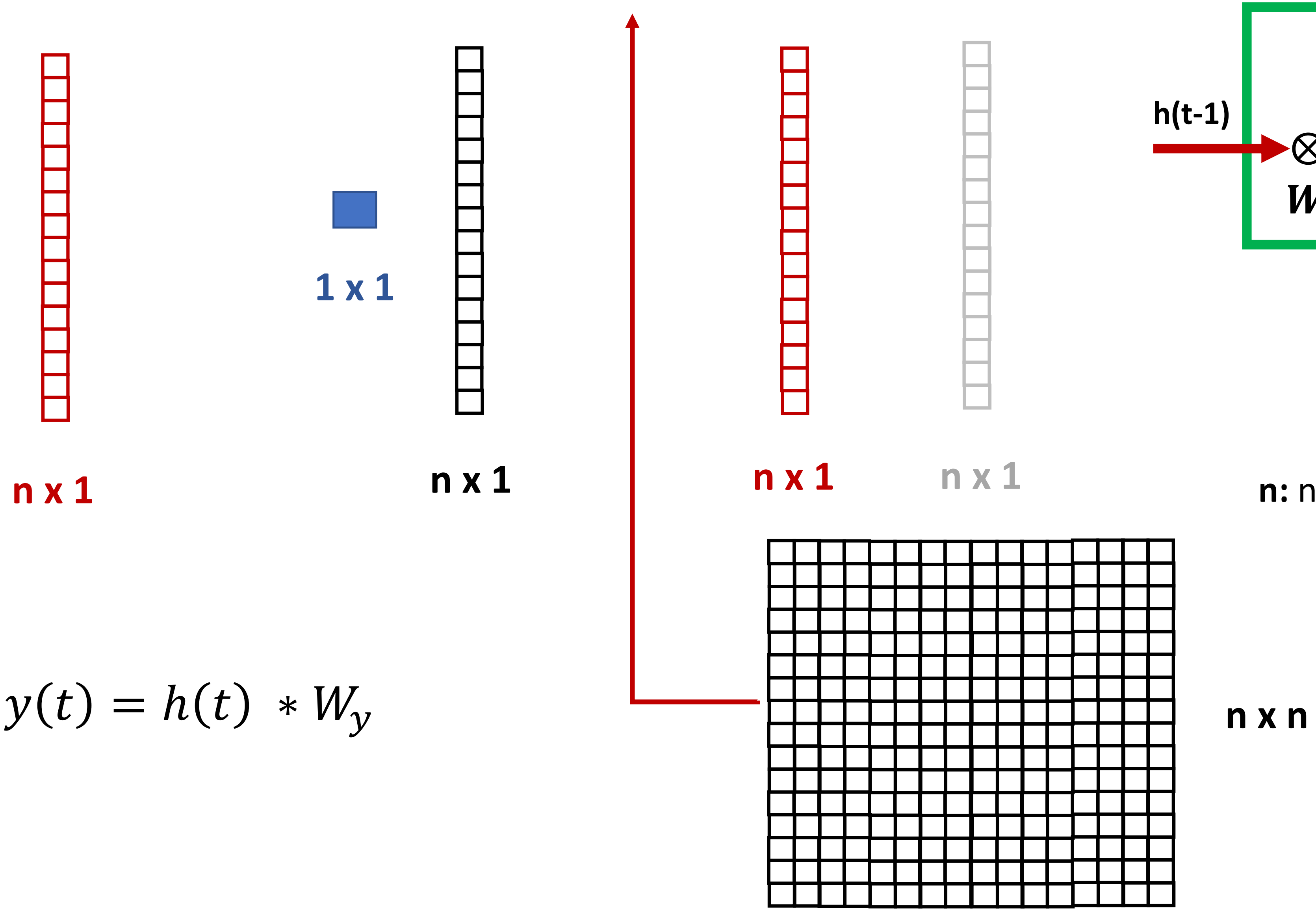


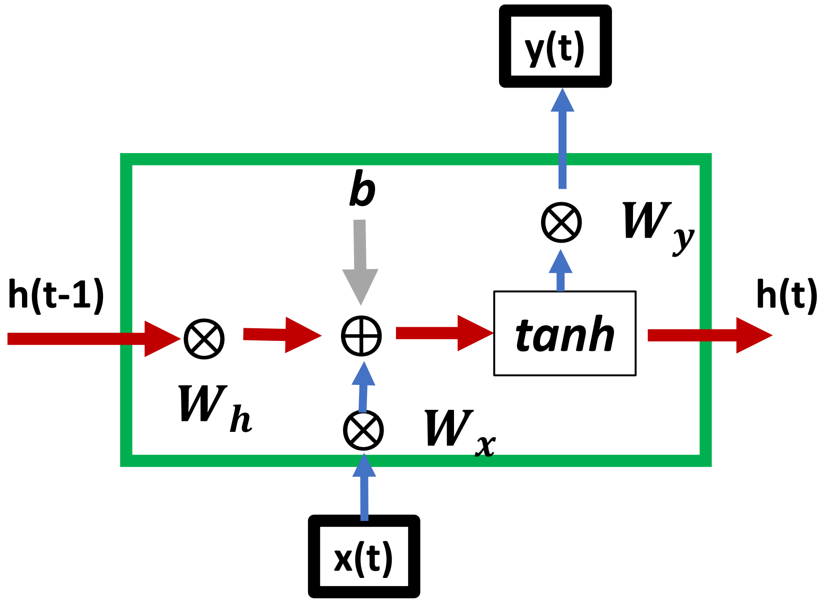


$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

$$y(t) = h(t) * W_y$$

$$h(t) = \tanh[\, x(t) * W_x + \boxed{W_h} * h(t-1) + b \,]$$



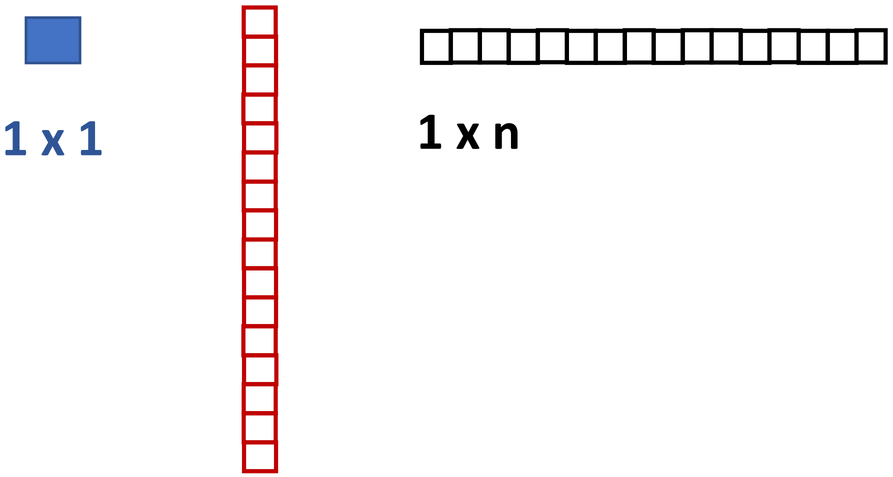


n: number of states/ neurons

$$y(t) = h(t) * W_y$$

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

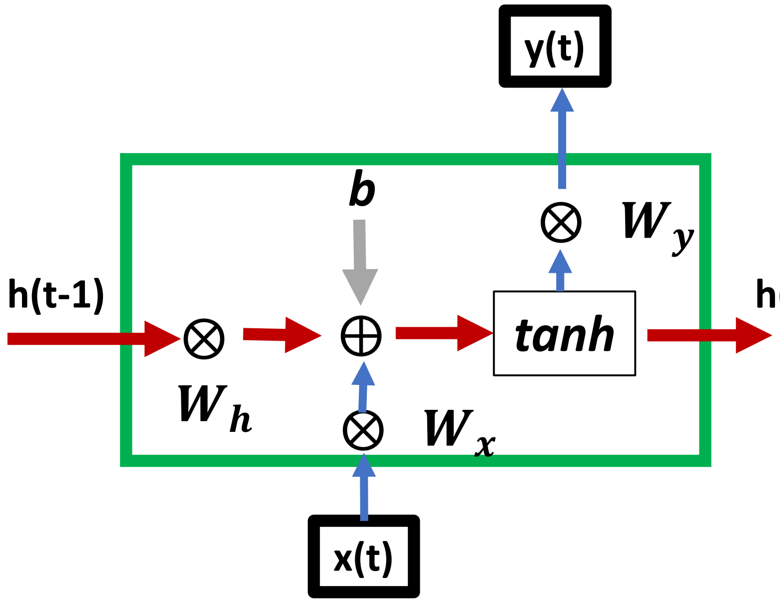
$$y(t) = h(t) * W_y$$



1 x 1

n x 1

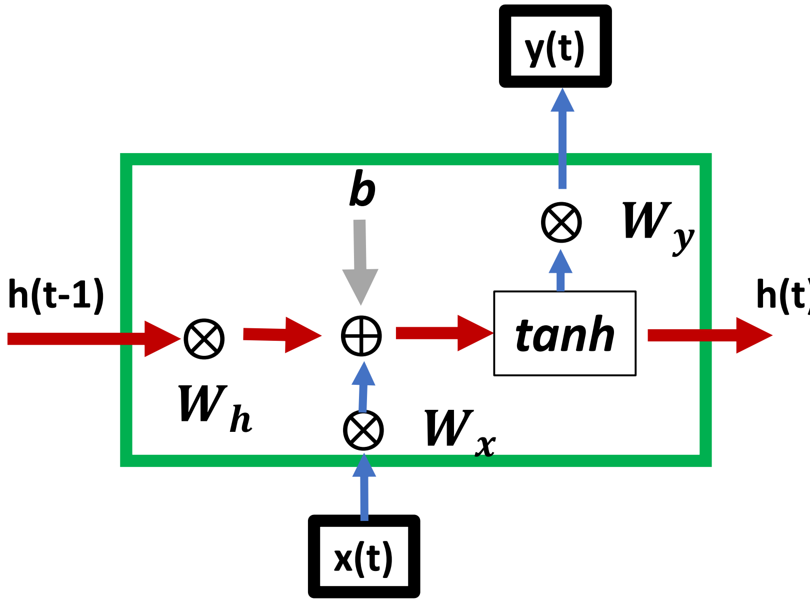
1 x n



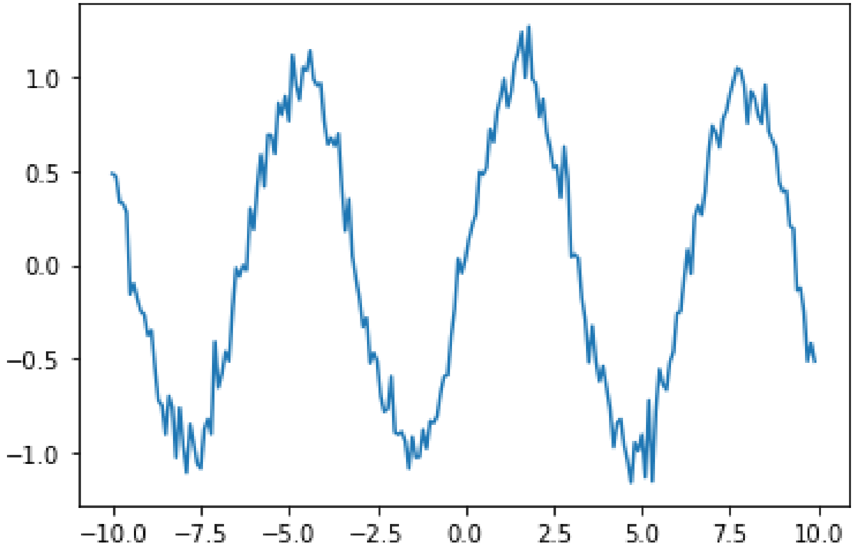
n: number of states/ neurons

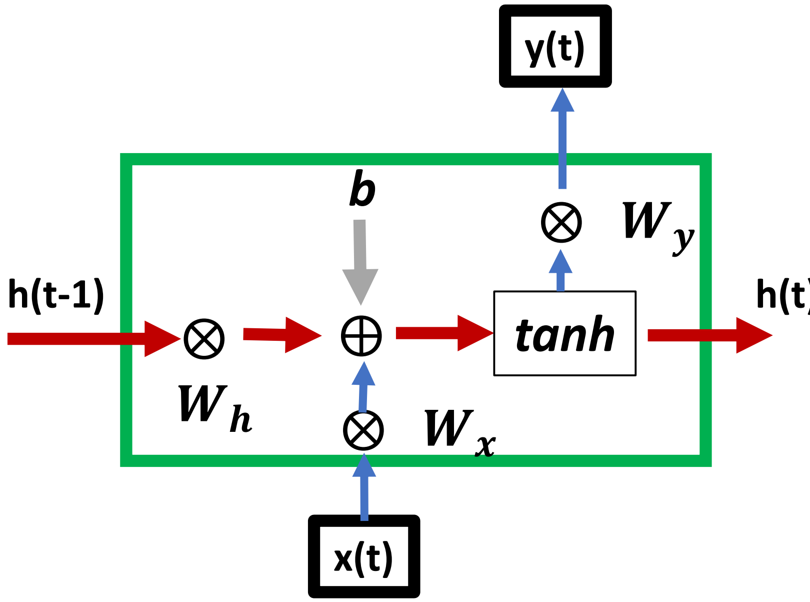
We are ready to build the first cell now!

Aim: training an RNN on noisy periodic data




```
import numpy as np
import matplotlib.pyplot as plt

X_t = np.arange(-10, 10, 0.1)
X_t = X_t.reshape(len(X_t), 1)
Y_t = np.sin(X_t) + 0.1*np.random.randn(len(X_t), 1)

plt.plot(X_t, Y_t)
plt.show()
```





We are ready to build the first cell now!

Aim: training an RNN on noisy periodic data

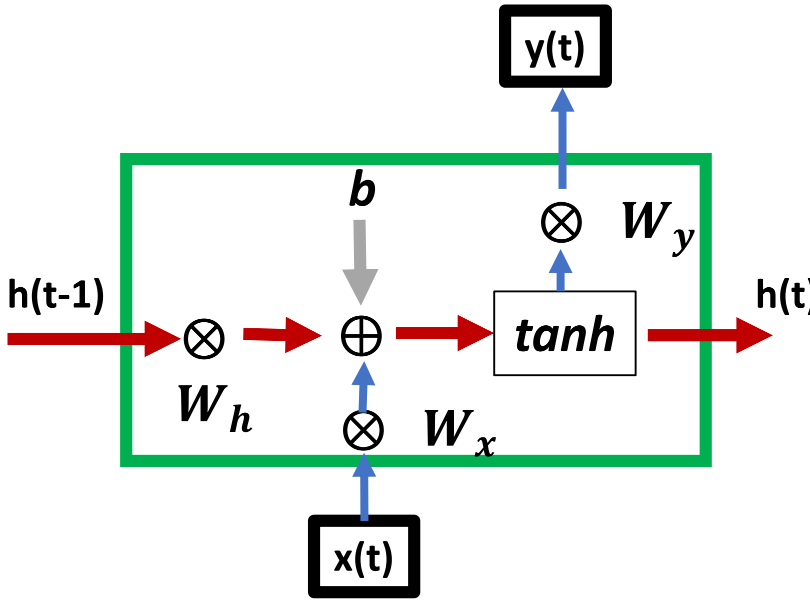
```
import numpy as np

class RNN():

    def __init__(self, X_t, n_neurons):

        self.T          = max(X_t.shape)
        self.X_t        = X_t
        self.Y_hat      = np.zeros((self.T, 1))

        self.n_neurons = n_neurons

        self.Wx         = 0.1*np.random.randn(n_neurons, 1)
        self.Wh         = 0.1*np.random.randn(n_neurons, n_neurons)
        self.Wy         = 0.1*np.random.randn(1, n_neurons)
        self.biases     = 0.1*np.random.randn(n_neurons, 1)

        self.H          = [np.zeros((n_neurons, 1)) for t in range(self.T + 1)]
```

initializing a vector for the prediction of y, $\hat{y}$

we also want to keep track of the state vector

<u>We are ready to build the first cell now!</u>

Aim: training an RNN on noisy periodic data

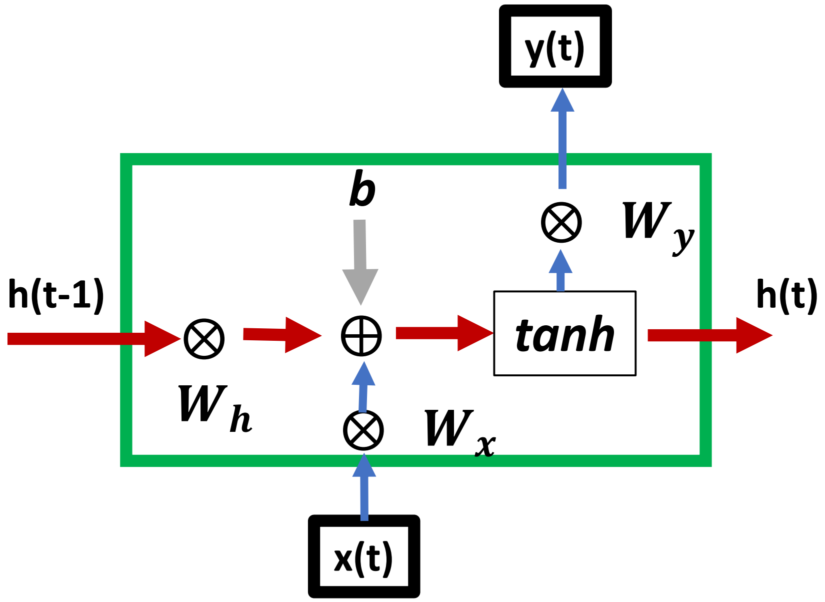



```
class RNN():

    def __init__(self, X_t, n_neurons):

        ...

        self.H = [np.zeros((n_neurons, 1)) for t in range(self.T + 1)]

    def forward(self, xt, ht_1):

        out     = np.dot(self.Wx, xt) + np.dot(self.Wh, ht_1) + self.biases
        ht      = np.tanh(out)
        y_hat_t = np.dot(self.Wy, ht)

        return ht, y_hat_t, out
```

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

let us run a test:

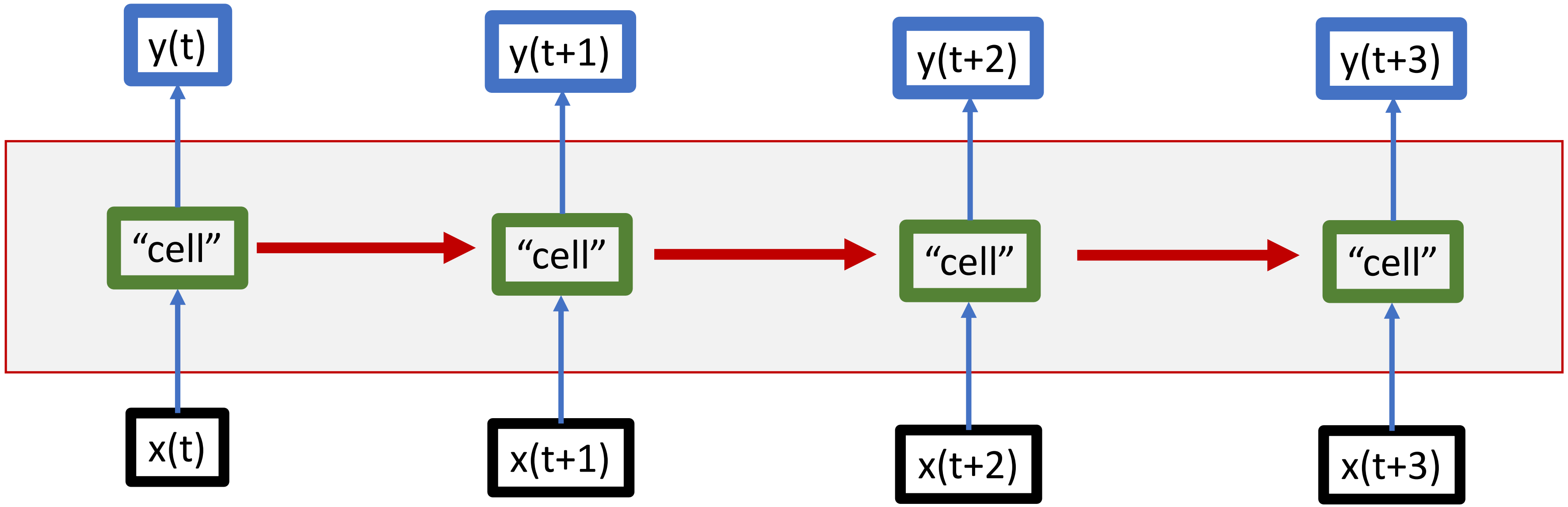
```
n_neurons = 500

from RNN import *

rnn   = RNN(X_t, n_neurons)

Y_hat = rnn.Y_hat
H     = rnn.H
T     = rnn.T
ht    = H[0]

for t, xt in enumerate(X_t):

    xt = xt.reshape(1, 1)

    [ht, y_hat_t, out] = rnn.forward(xt, ht)

    H[t+1]   = ht
    Y_hat[t] = y_hat_t
```



initial state vector

we apply the cell recursively over all *t* we apply the cell **recursively** over all *t*

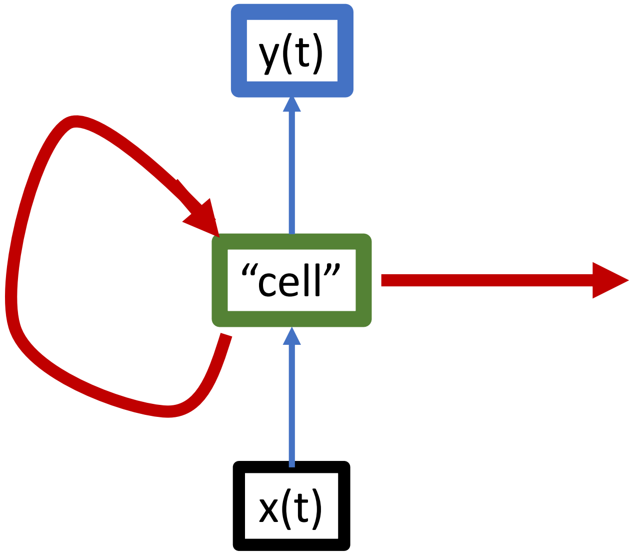
```
for t, xt in enumerate(X_t):

    xt = xt.reshape(1, 1)

    [ht, y_hat_t, out] = rnn.forward(xt, ht)

    H[t+1]    = ht
    Y_hat[t] = y_hat_t
```



we apply the cell **recursively** over all *t*

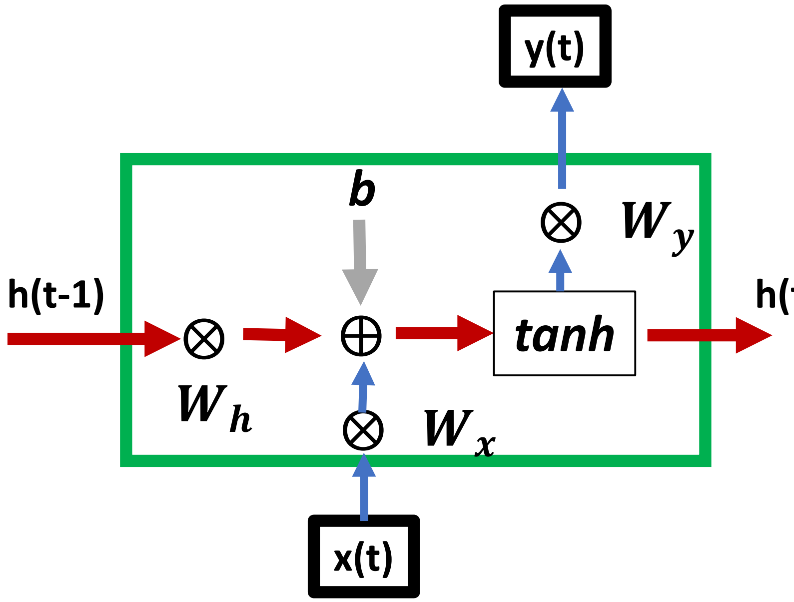
```
for t, xt in enumerate(X_t):

    xt = xt.reshape(1, 1)

    [ht, y_hat_t, out] = rnn.forward(xt, ht)

    H[t+1]   = ht
    Y_hat[t] = y_hat_t
```

loop over *X_t*



let us run a test:

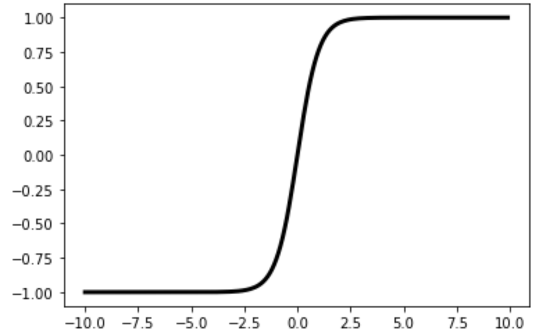
```
n_neurons = 500

from RNN import *

rnn   = RNN(X_t, n_neurons)

Y_hat = rnn.Y_hat
H     = rnn.H
T     = rnn.T
ht    = H[0]
```

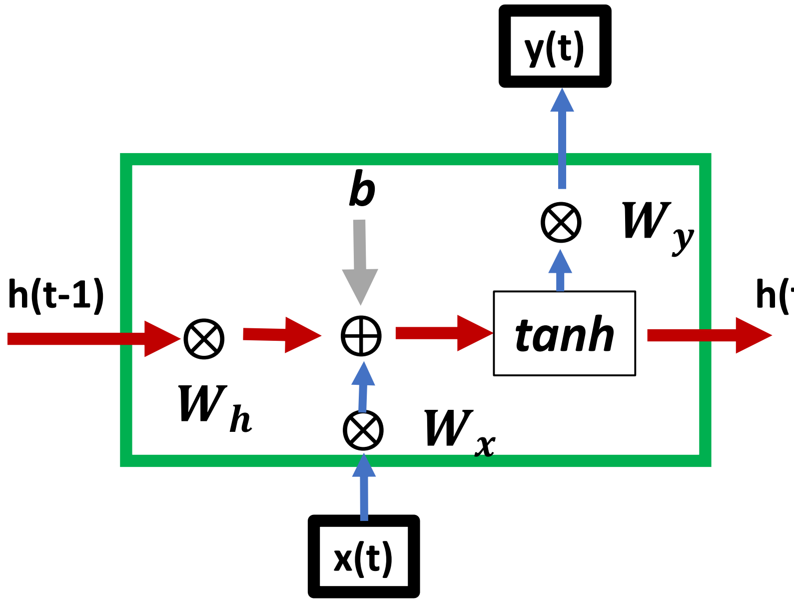



```
for t, xt in enumerate(X_t):

    xt = xt.reshape(1, 1)

    [ht, y_hat_t, out] = rnn.forward(xt, ht)

    H[t+1]   = ht
    Y_hat[t] = y_hat_t
```

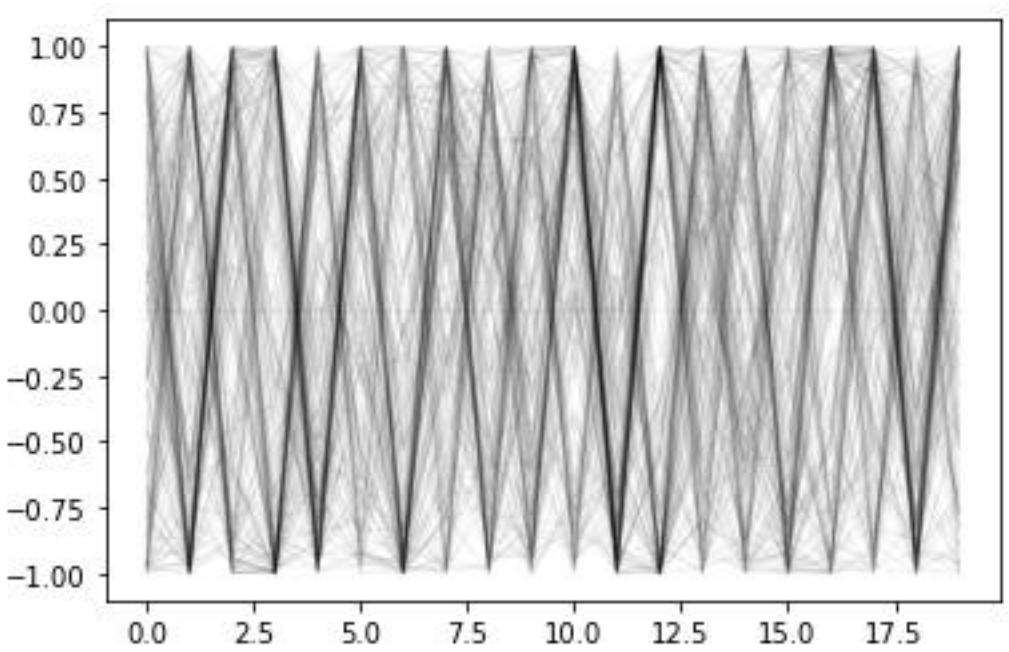
let us run a test:

```
n_neurons = 500

from RNN import *

rnn    = RNN(X_t, n_neurons)

...
```





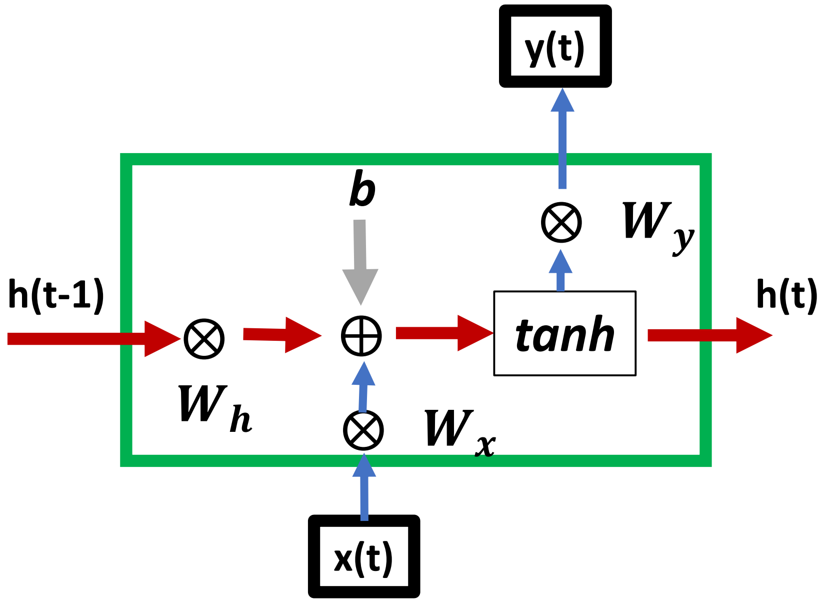

```
for h in H:
        plt.plot(np.arange(20), h[0:20],\
        'k-', linewidth = 1, alpha = 0.05)
```

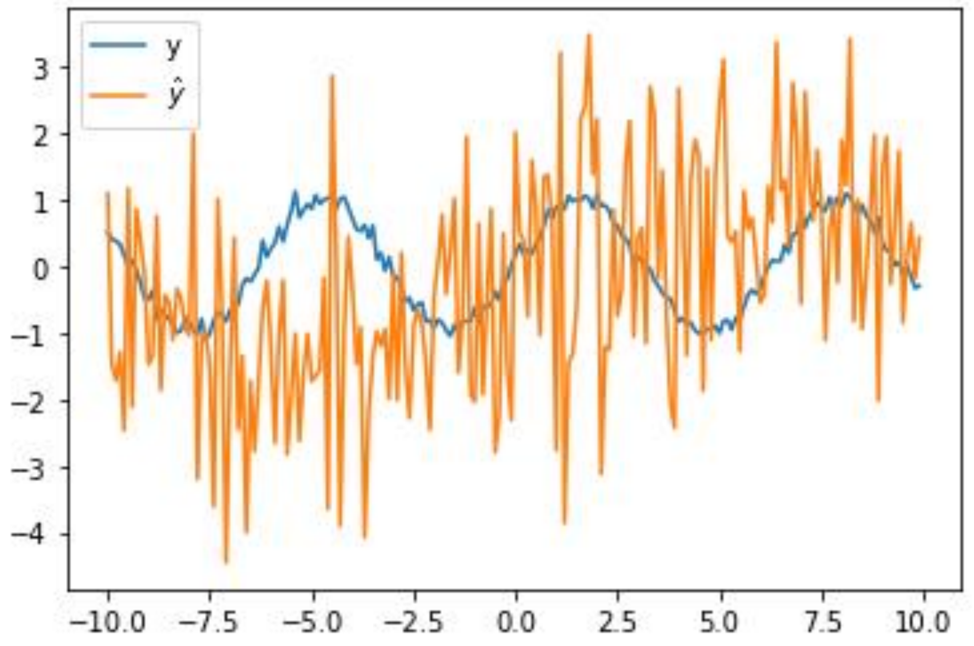


defined states -1 or +1

let us run a test:

```
n_neurons = 500

from RNN import *

rnn   = RNN(X_t, n_neurons)


...


plt.plot(X_t, Y_t)
plt.plot(X_t, Y_hat)
plt.legend(['y', '$\hat{y}$'])
plt.show()
```





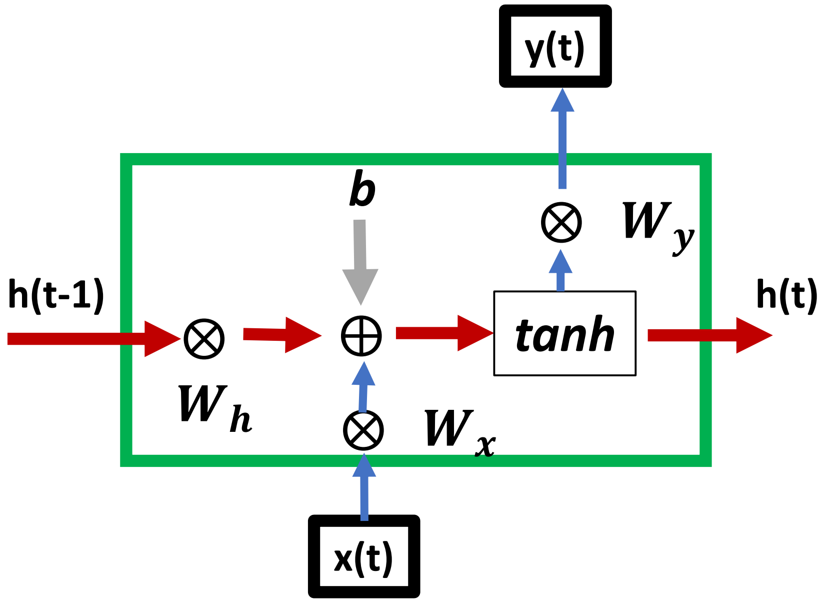
let us run a test:

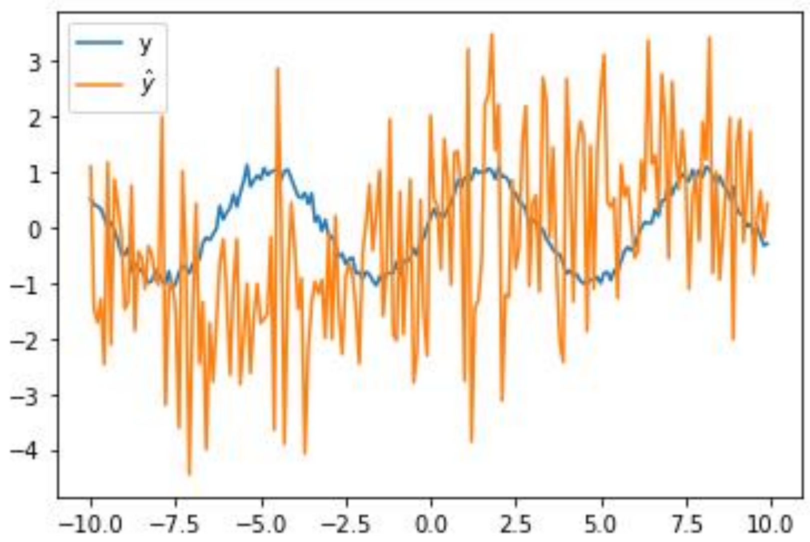
```
n_neurons = 500

from RNN import *

rnn    = RNN(X_t, n_neurons)

...

plt.plot(X_t, Y_t)
plt.plot(X_t, Y_hat)
plt.legend(['y', '$\hat{y}$'])
plt.show()
```

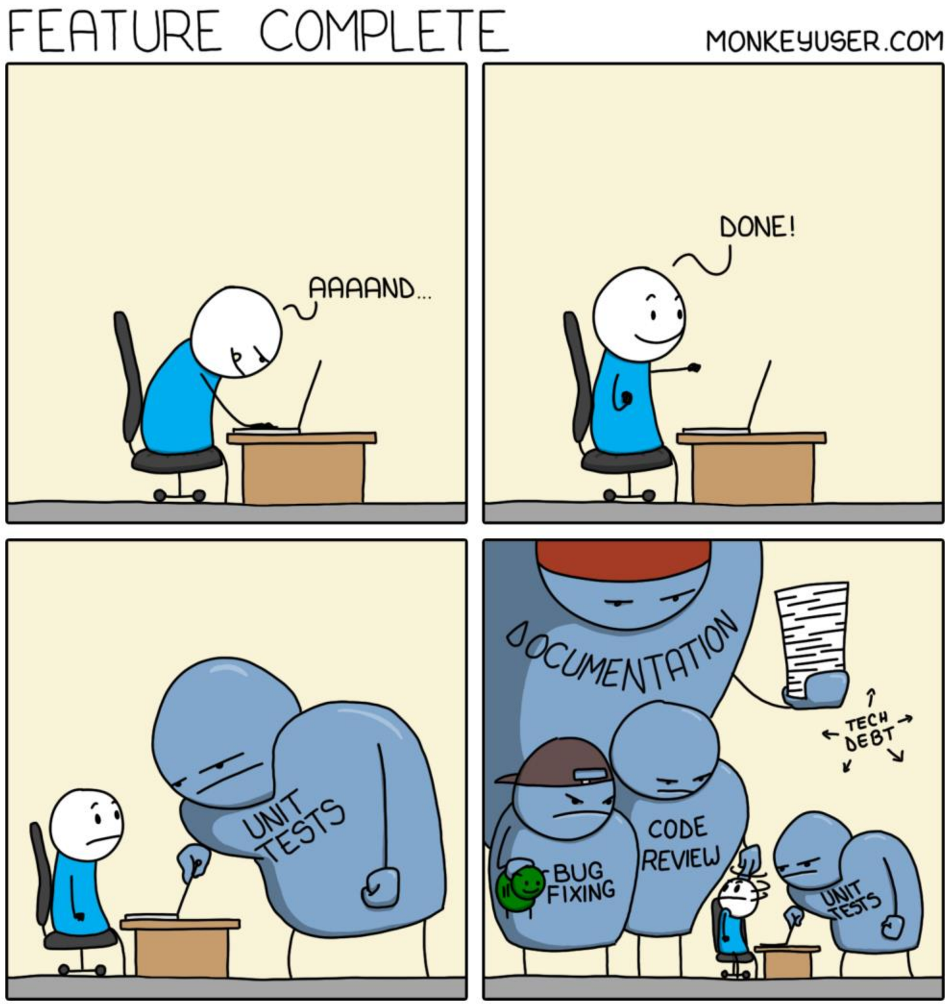
We can therefore define the **loss *L*** as MSSE:

```
dY          = Y_hat - Y_t

L           = 0.5*np.dot(dY.T,dY)/T
```

$$\boxed{L = \frac{1}{2}\sum_{t=1}^{T}[\,\hat{y}(t) - y(t)\,]^2}$$

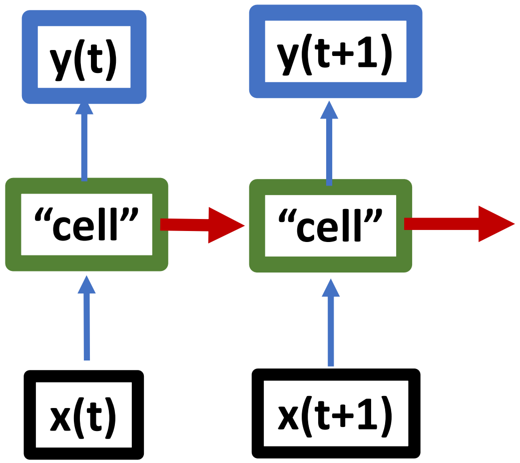






outline:

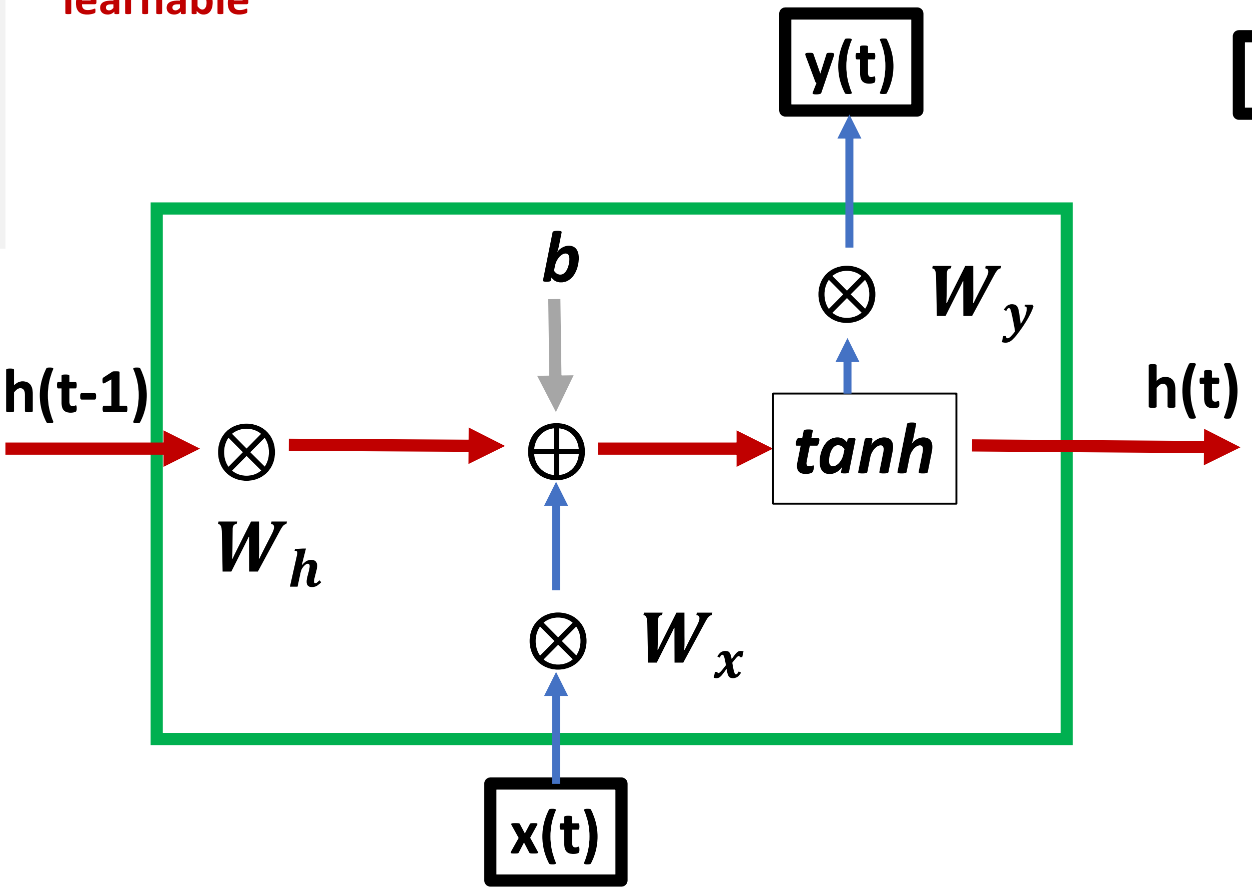
- the idea
- the RNN cell
- ***BackPropagation Through Time***
- *full backpropagtion*
- *creating an SGD optimizer*
- *creating a full package*



- **state vector** *h(t)*, i. e. the *"memory"* the system has along time

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*

learnable



- weights W_x for *x(t)*
- weights W_h for *h(t)*
- bias ***b***
- weights W_y for *y(t)*

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

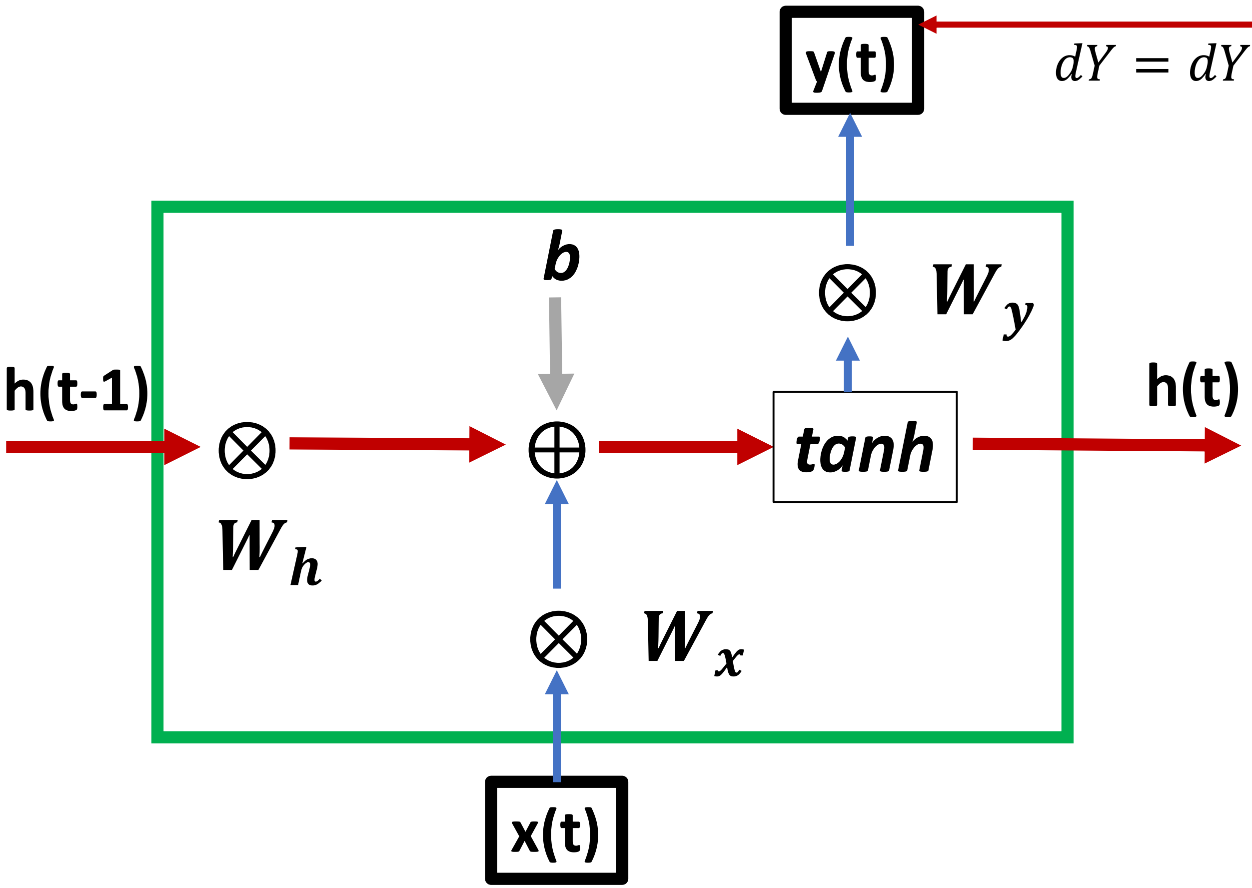
$$y(t) = h(t) * W_y$$

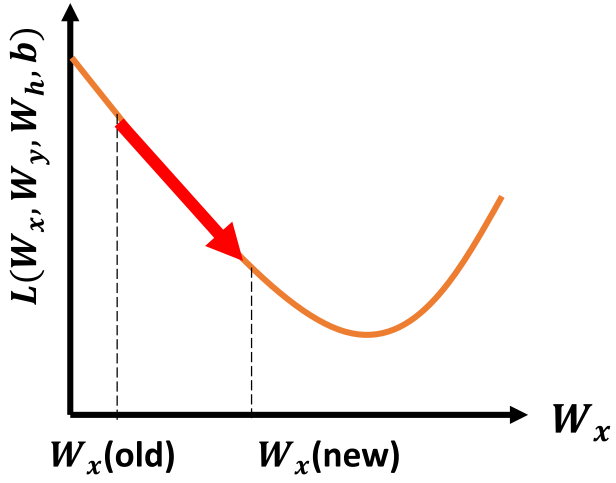
$$L = \frac{1}{2} \sum_{t=1}^{T} [\, \boxed{\hat{y}(t) - y(t)} \,]^2$$

$$dY = dY(W_x, W_y, W_h, b)$$

ϵ: learning rate

$$W_x(\text{new}) = W_x(\text{old}) - \epsilon \frac{d\, L(W_x, W_y, W_h, b)}{dW_x}$$





- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias ***b***
- weights $\boldsymbol{W_y}$ for *y(t)*

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

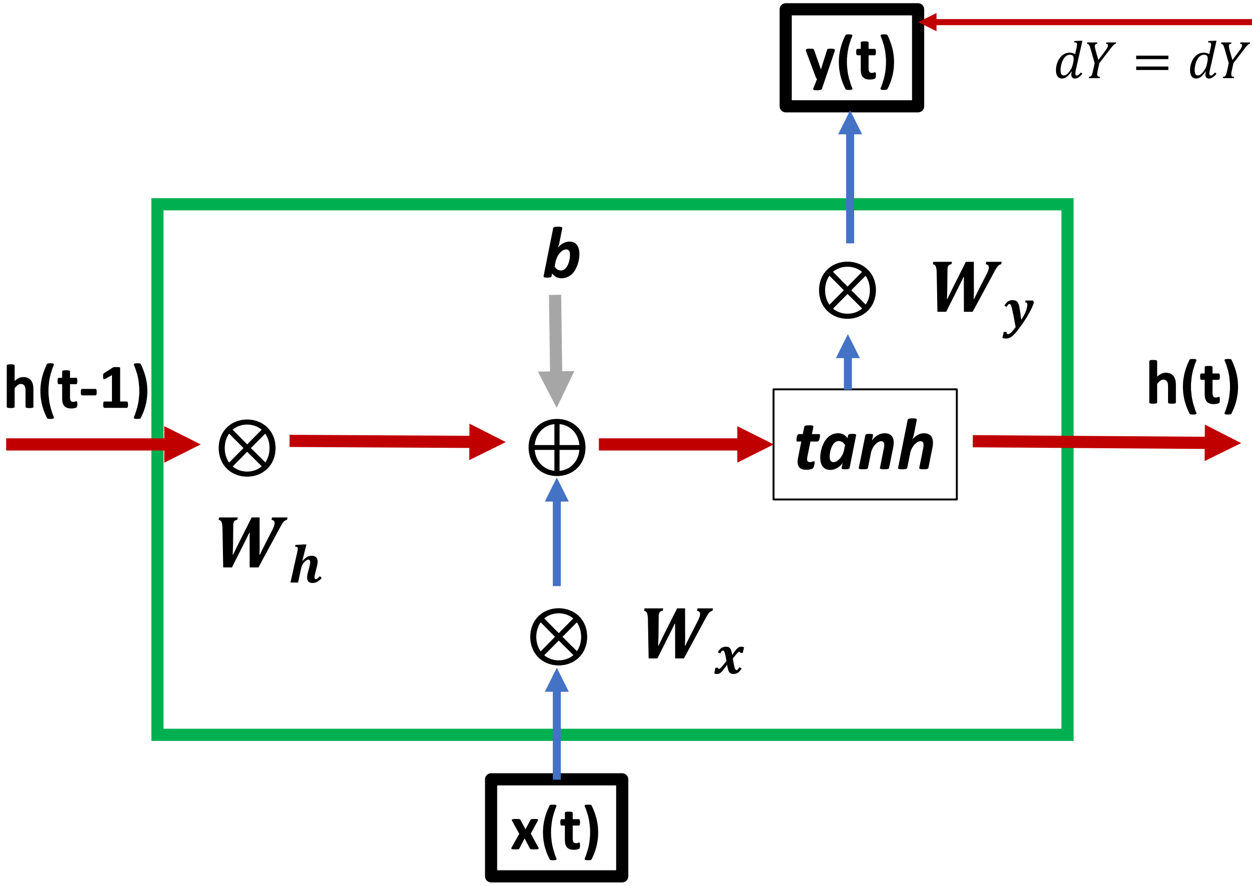
$$y(t) = h(t) * W_y$$

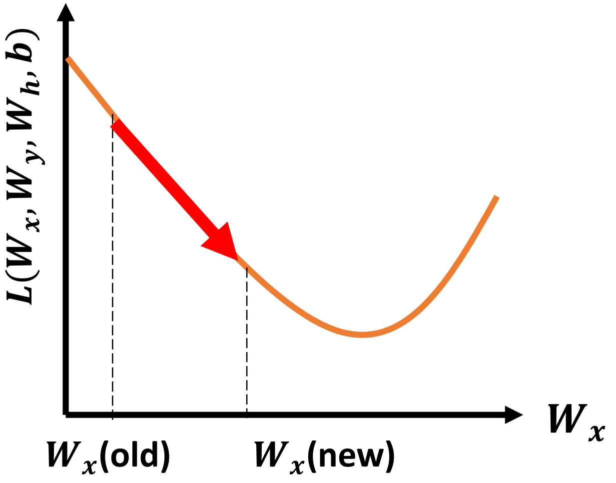
$$L = \frac{1}{2} \sum_{t=1}^{T} \boxed{[\hat{y}(t) - y(t)]}^2$$

$$dY = dY(W_x, W_y, W_h, b)$$

ϵ: learning rate

$$dW_x = -\,\epsilon \, \frac{d\, L(W_x, W_y, W_h, b)}{dW_x}$$





- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias ***b***
- weights $\boldsymbol{W_y}$ for *y(t)*

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b\,]$$

$$y(t) = h(t) * W_y$$

$$L = \frac{1}{2}\sum_{t=1}^{T} [\,\boxed{\hat{y}(t) - y(t)}\,]^2$$

ϵ: learning rate

dY

$$dW_x = -\,\epsilon \frac{d\,L(W_x, W_y, W_h, b)}{dW_x} = -\,\epsilon \boxed{\frac{dL}{dy}} \frac{dy}{dh(t)} \frac{dh(t)}{d\tanh} \frac{d\tanh}{dW_x}$$

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

$$\boxed{y(t) = h(t) * W_y}$$

ϵ: learning rate

dY W_y

$$dW_x = -\epsilon \frac{d\, L(W_x, W_y, W_h, b)}{dW_x} = -\epsilon \frac{dL}{dy} \boxed{\frac{dy}{dh(t)}} \frac{dh(t)}{d\tanh} \frac{d\tanh}{dW_x}$$

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*

$$\boxed{h(t) = \tanh[\,x(t) * W_x + W_h * h(t-1) + b\,]}$$

$$y(t) = h(t) * W_y$$

ϵ: learning rate

$$dW_x = -\,\epsilon\,\frac{d\,L(W_x, W_y, W_h, b)}{dW_x} = -\,\epsilon\,\underbrace{\frac{dL}{dy}}_{dY}\,\underbrace{\frac{dy}{dh(t)}}_{W_y}\,\underbrace{\boxed{\frac{dh(t)}{d\tanh}}}_{1\,-\,tanh^2}\,\frac{dtanh}{dW_x}$$

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*

$$h(t) = \tanh[\boxed{x(t) * W_x} + W_h * h(t-1) + b]$$

$$y(t) = h(t) * W_y$$

ϵ: learning rate

$dY \quad W_y \quad 1 - tanh^2 \quad x(t)$

$$dW_x = -\epsilon \frac{d\,L(W_x, W_y, W_h, b)}{dW_x} = -\epsilon \frac{dL}{dy}\frac{dy}{dh(t)}\frac{dh(t)}{d\tanh}\boxed{\frac{dtanh}{dW_x}}$$

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

$$\boxed{y(t) = h(t) * W_y}$$

ϵ: learning rate

$$dW_x = -\epsilon \frac{d\,L(W_x, W_y, W_h, b)}{dW_x} = -\epsilon \frac{dL}{dy}\frac{dy}{dh(t)}\frac{dh(t)}{d\tanh}\frac{dtanh}{dW_x} = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$

$$dW_y = -\epsilon \frac{d\,L(W_x, W_y, W_h, b)}{dW_y} = -\epsilon \frac{dL}{dy}\frac{dy}{dW_y} = -\epsilon * dY * h(t)$$

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias ***b***
- weights $\boldsymbol{W_y}$ for *y(t)*

$$h(t) = \tanh[\, x(t) * W_x + \boxed{W_h * h(t-1)} + b\,]$$

$$y(t) = h(t) * W_y$$

ϵ: learning rate

$$dW_x = -\epsilon \frac{d\, L(W_x, W_y, W_h, b)}{dW_x} = -\epsilon \frac{dL}{dy} \frac{dy}{dh(t)} \frac{dh(t)}{d \tanh} \frac{dtanh}{dW_x} = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$

$$dW_y = -\epsilon \frac{d\, L(W_x, W_y, W_h, b)}{dW_y} = -\epsilon \frac{dL}{dy} \frac{dy}{dW_y} = -\epsilon * dY * h(t)$$

$$dW_h = -\epsilon \frac{d\, L(W_x, W_y, W_h, b)}{dW_h} = -\epsilon \frac{dL}{dy} \frac{dy}{dh(t)} \frac{dh(t)}{d \tanh} \frac{dtanh}{dW_h} = -\epsilon * dY * W_y * (1 - tanh^2) * \boxed{h(t-1)}$$

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias ***b***
- weights $\boldsymbol{W_y}$ for *y(t)*

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) \boxed{+\, b\,]}$$

$$y(t) = h(t) * W_y$$

ϵ: learning rate

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$

$$dW_y = -\epsilon * dY * h(t)$$

$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$

$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

$$y(t) = h(t) * W_y$$

ϵ: learning rate

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$

$$dW_y = -\epsilon * dY * h(t)$$

$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$

$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$

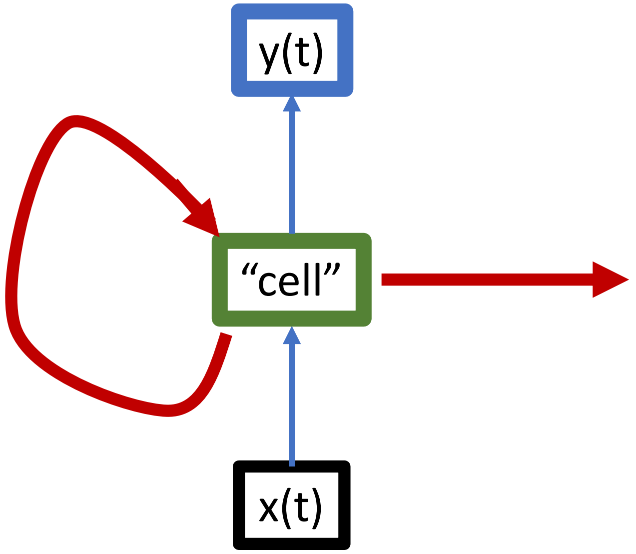
$$dW_y = -\epsilon * dY * h(t)$$

$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$

$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

```
for t, xt in enumerate(X_t):

    xt = xt.reshape(1, 1)

    [ht, y_hat_t, out] = rnn.forward(xt, ht)

    H[t+1]    = ht
    Y_hat[t] = y_hat_t
```

loop over *X_t*



this has to be done **reversely over time**

→ different *dy(t)* for **each *t***
→ derivative for **each tanh at time *t***
→ they **all** contribute to dW_x, dW_y, dW_h and *db*

→ we need to write a **reversed loop** for the backpropagation part

→ **B**ack**P**ropagation **T**hrough **T**ime (**BPTT**)

We need to restructure the code slightly:

1) creating a class for the activation function (tanh)
→ we can call **different instances during the loop** and **keep track of the derivatives**

2) defining the forward part of the cell as own method
→ makes the code more readable

We need to restructure the code slightly:

1) creating a class for the activation function (tanh)

→ we can call **different instances during the loop** and **keep track of the derivatives**

2) defining the forward part of the cell as own method

→ makes the code more readable

$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$

```
class Tanh:

    def forward(self, inputs):
        self.output = np.tanh(inputs)
        self.inputs = inputs

    def backward(self, dvalues):
        deriv        = 1 - self.output**2
        self.dinputs = np.multiply(deriv, dvalues)
```

$dh(t)$

$(1 - tanh^2)$

$(1 - tanh^2) * dh(t)$

We need to restructure the code slightly:

1) creating a class for the activation function (tanh)
→ we can call **different instances during the loop** and **keep track of the derivatives**

2) defining the forward part of the cell as own method
→ makes the code more readable

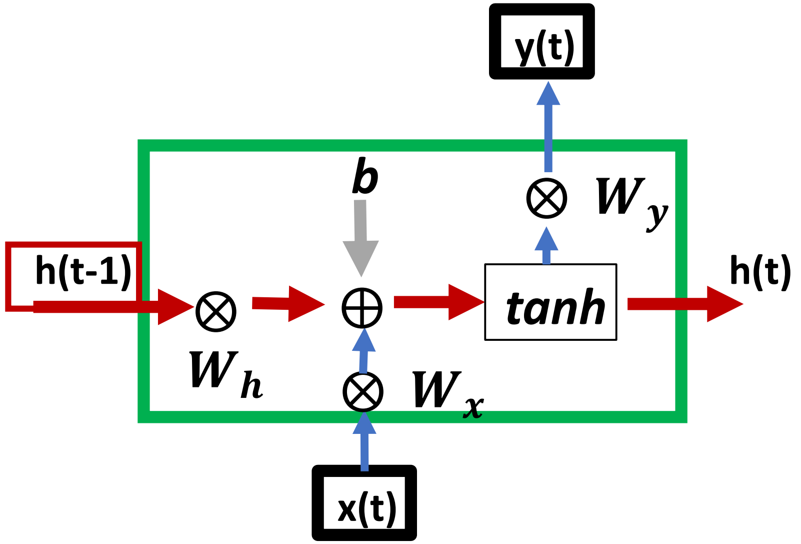
2) defining the forward part of the cell as own method
→ makes the code more readable

we want to be more flexible concerning the activation (see also later)

```
class RNN():

    def __init__(self, X_t, n_neurons, Activation):

        ...

        self.H = [np.zeros((n_neurons, 1)) for t in range(self.T + 1)]

        self.Activation = Activation

    def forward(self, xt, ht_1):
```

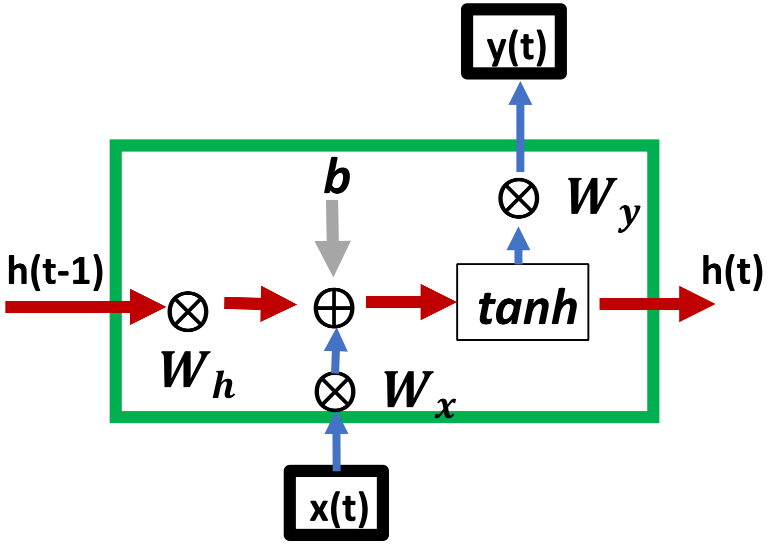
2) defining the forward part of the cell as own method

→ makes the code more readable




```python
def forward(self):

    #initializing dweights
    self.dWx        = np.zeros((self.n_neurons, 1))
    self.dWh        = np.zeros((self.n_neurons, self.n_neurons))
    self.dWy        = np.zeros((1, self.n_neurons))
    self.dbiases    = np.zeros((self.n_neurons, 1))

    #extracting variables we are going to need
    X_t             = self.X_t
    H               = self.H
    Y_hat           = self.Y_hat
    ht              = H[0]# initial state vector
```

We need to store the outer derivative $(1 - tanh^2)$ for every time point *t:*




```
self.output = np.tanh(inputs)
…
deriv        = 1 - self.output**2
```

```
def forward(self):

...
    ht          = H[0]# initial state vector
    Activation  = self.Activation

    ACT         = [Activation for i in range(self.T)]
```

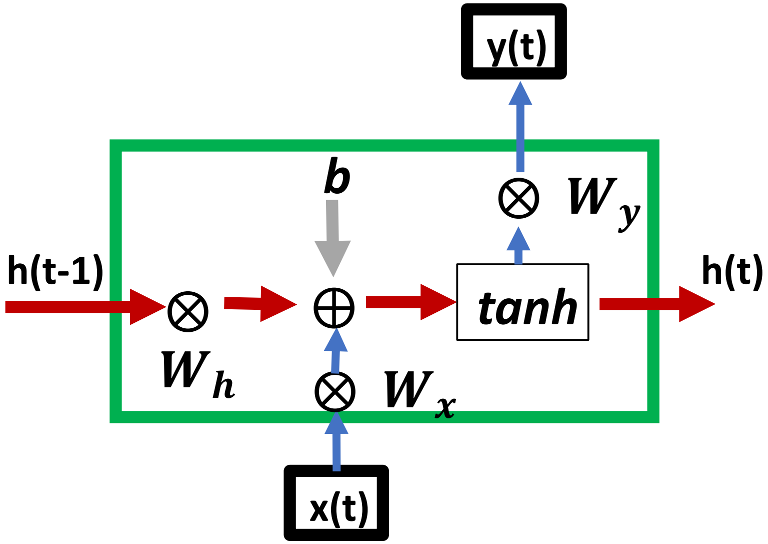
storing instances of an activation function in a list:
makes the code even more flexible

1) creating a class for the activation function (tanh)

→ we can call **different instances during the loop** and **keep track of the derivatives**

2) defining the forward part of the cell as **own method in the RNN class**

→ makes the code more readable



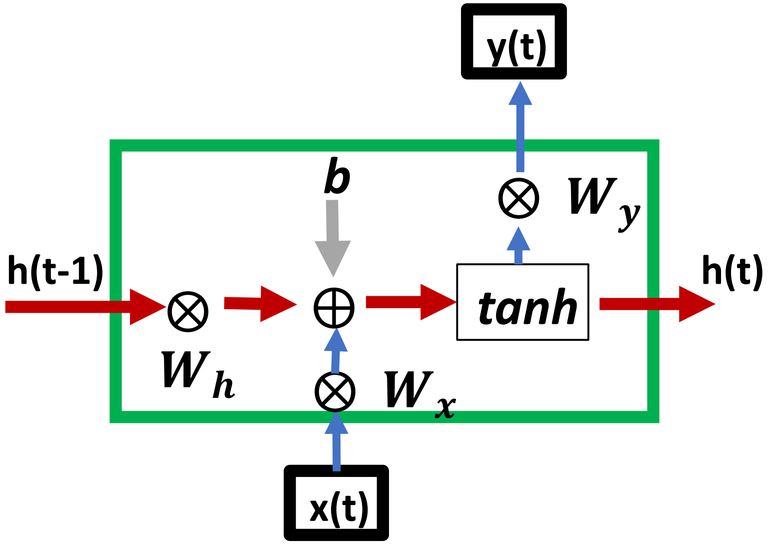

```
def RNNCell(self, X_t, ht, ACT, H, Y_hat):

    for t, xt in enumerate(X_t):

        xt       = xt.reshape(1, 1)
        out      = np.dot(self.Wx, xt) + np.dot(self.Wh, ht) + self.biases

        ACT[t].forward(out)
        ht   = ACT[t].output

        y_hat_t  = np.dot(self.Wy, ht)

        H[t+1]   = ht
        Y_hat[t] = y_hat_t

    return(ACT,H,Y_hat)
```

1) creating a class for the activation function (tanh)

→ we can call **different instances during the loop** and **keep track of the derivatives**

2) defining the forward part of the cell as **own method in the RNN class**

→ makes the code more readable




```
def forward(self):

...
        ht              = H[0]# initial state vector
        Activation      = self.Activation
        ACT             = [Activation for i in range(self.T)]


        [ACT,H,Y_hat]   = self.RNNCell(X_t, ht, ACT, H, Y_hat)

        self.Y_hat      = Y_hat
        self.H          = H
        self.ACT        = ACT
```

The code so far:

```
class Tanh:

    def forward(self, inputs):
        self.output = np.tanh(inputs)
        self.inputs = inputs

    def backward(self, dvalues):
        deriv        = 1 - self.output**2
        self.dinputs = np.multiply(deriv, dvalues)
```

The code so far:

```
import numpy as np

class RNN():

    def __init__(self, X_t, n_neurons, Activation):

        self.T              = max(X_t.shape)
        self.X_t            = X_t

        self.Y_hat          = np.zeros((self.T, 1))

        self.n_neurons      = n_neurons

        self.Wx             = 0.1*np.random.randn(n_neurons, 1)
        self.Wh             = 0.1*np.random.randn(n_neurons, n_neurons)
        self.Wy             = 0.1*np.random.randn(1, n_neurons)
        self.biases         = 0.1*np.random.randn(n_neurons, 1)

        self.H        = [np.zeros((self.n_neurons,1)) for t in range(self.T + 1)]
        self.Activation     = Activation
```

The code so far:

```
class RNN():

...
        self.Activation      = Activation

  def forward(self):

        self.dWx        = np.zeros((self.n_neurons, 1))
        self.dWh        = np.zeros((self.n_neurons, self.n_neurons))
        self.dWy        = np.zeros((1, self.n_neurons))
        self.dbiases    = np.zeros((self.n_neurons, 1))

        X_t             = self.X_t
        H               = self.H
        Y_hat           = self.Y_hat
        ht              = H[0]# initial state vector
        Activation      = self.Activation

        ACT             = [Activation for i in range(self.T)]

        [ACT,H,Y_hat]   = self.RNNCell(X_t, ht, ACT, H, Y_hat)
```

The code so far:

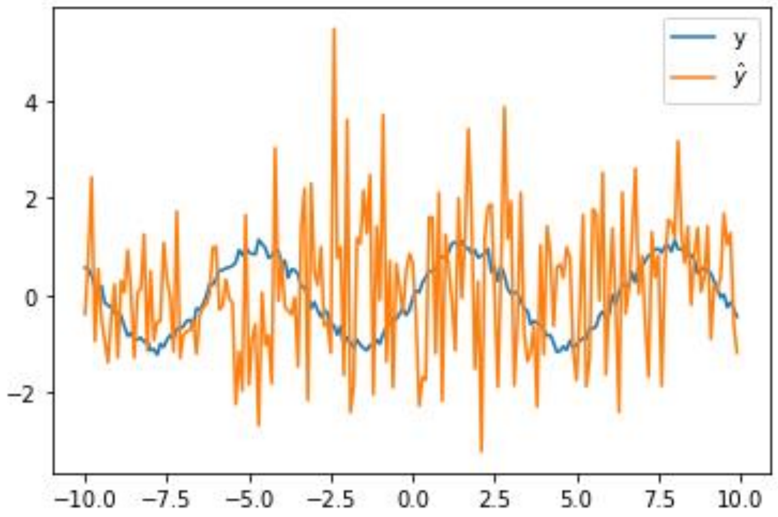
```
class RNN():
...
        self.Activation      = Activation

  def forward(self):

...

        ACT              = [Activation for i in range(self.T)]

        [ACT,H,Y_hat]    = self.RNNCell(X_t, ht, ACT, H, Y_hat)

        self.Y_hat       = Y_hat
        self.H           = H
        self.ACT         = ACT
```

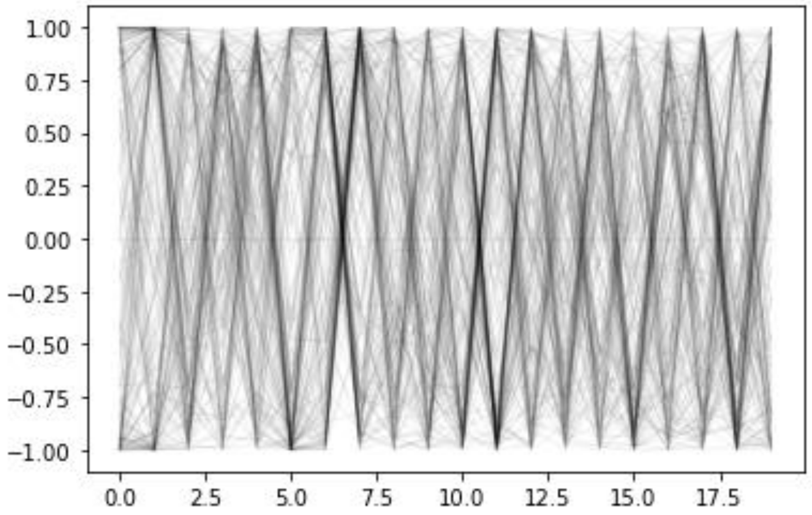
The code so far:

```
class RNN():
...

  def forward(self):

...
        self.ACT       = ACT

  def RNNCell(self, X_t, ht, ACT, H, Y_hat):

        for t, xt in enumerate(X_t):

              xt        = xt.reshape(1, 1)
              out       = np.dot(self.Wx, xt) + np.dot(self.Wh, ht) + self.biases

              ACT[t].forward(out)
              ht   = ACT[t].output

              y_hat_t   = np.dot(self.Wy, ht)
              H[t+1]    = ht
              Y_hat[t] = y_hat_t

        return(ACT,H,Y_hat)
```

Let us test run the code:

```
from RNN import *

rnn   = RNN(X_t, n_neurons, Tanh())
rnn.forward()

Y_hat = rnn.Y_hat
H     = rnn.H
T     = rnn.T

dY         = Y_hat - Y_t
L          = 0.5*np.dot(dY.T,dY)/T

plt.plot(X_t, Y_t)
plt.plot(X_t, Y_hat)
plt.legend(['y', '$\hat{y}$'])
plt.show()

for h in H:
    plt.plot(np.arange(20), h[0:20], 'k-', linewidth = 1, alpha = 0.05)
```





Let us test run the code:

```
from RNN import *

rnn    = RNN(X_t, n_neurons, Tanh())
rnn.forward()

Y_hat = rnn.Y_hat
H     = rnn.H
T     = rnn.T

dY          = Y_hat - Y_t
L           = 0.5*np.dot(dY.T,dY)/T

plt.plot(X_t, Y_t)
plt.plot(X_t, Y_hat)
plt.legend(['y', '$\hat{y}$'])
plt.show()

for h in H:
    plt.plot(np.arange(20), h[0:20], 'k-', linewidth = 1, alpha = 0.05)
```

```
rnn.ACT[0].
    backward
    forward
    inputs
    output
```

We have T instances, each contains the $(1 - tanh^2) * dh(t)$ part for each *t* already!

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

```
class RNN():

...

  def backward(self, dinputs):

      T        = self.T
      H        = self.H
      X_t      = self.X_t

      ACT      = self.ACT

      dWx      = self.dWx
      dWy      = self.dWy
      dWh      = self.dWh
      dbiases  = self.dbiases

      Wy       = self.Wy
      Wh       = self.Wh
```

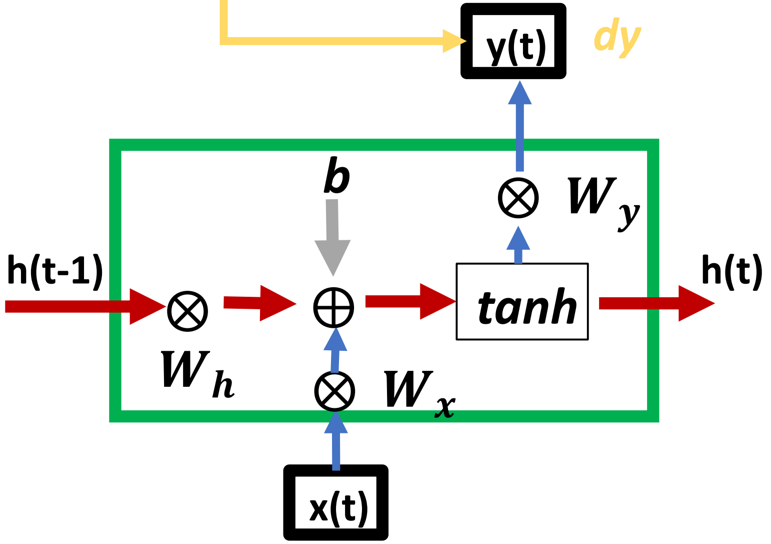
will be dY

```python
def backward(self, dinputs):  dy

...
        Wh        = self.Wh
```

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

```python
dinputs[-1].reshape(1,1)
```

$$\frac{dL}{dW_i} = -\epsilon \boxed{\frac{dL}{dy}} \frac{dy}{dh(t)} \frac{dh(t)}{d\tanh} \frac{dtanh}{dW_i}$$

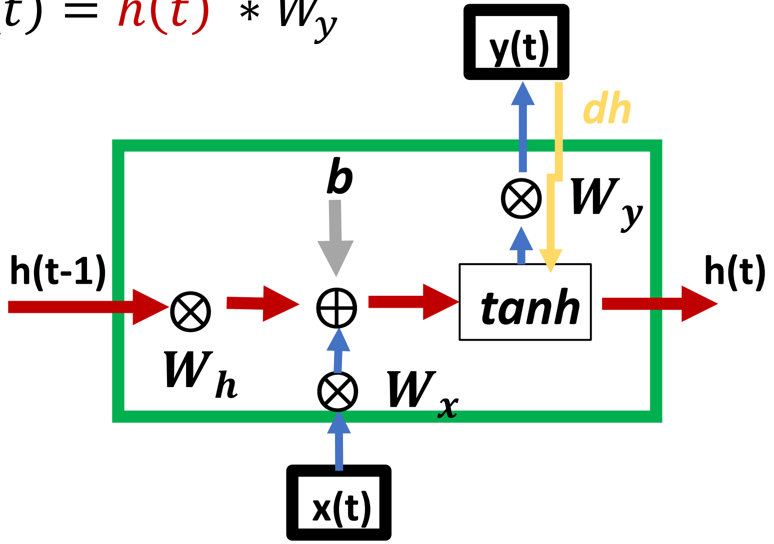


```python
def backward(self, dinputs):

...
        Wh        = self.Wh

        dht       = np.dot(Wy.T, dinputs[-1].reshape(1,1))
```

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * h(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

$$\frac{dL}{dW_i} = -\epsilon \frac{dL}{dy} \boxed{\frac{dy}{dh(t)}} \frac{dh(t)}{d\tanh} \frac{dtanh}{dW_i}$$

$$y(t) = h(t) * W_y$$

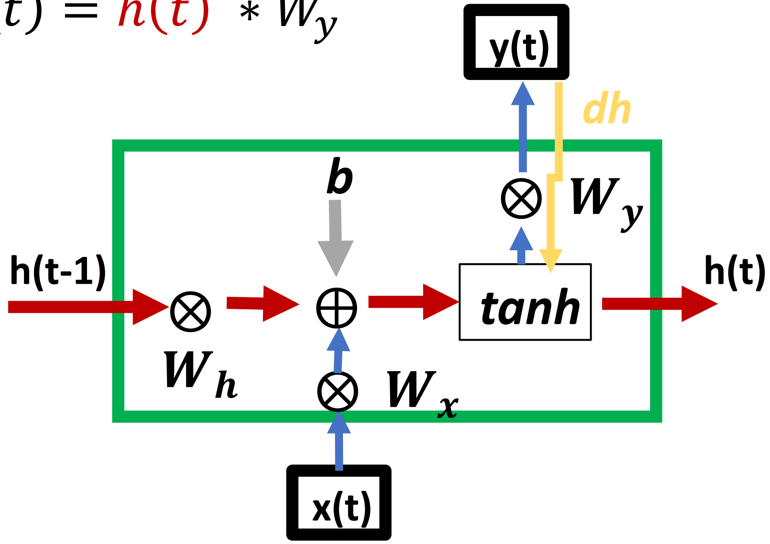


```
def backward(self, dinputs):

...
        Wh      = self.Wh

        dht     = np.dot(Wy.T, dinputs[-1].reshape(1,1))
```

$$dW_x = -\epsilon * \boxed{dY * W_y} * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * \boxed{dY * W_y} * (1 - tanh^2) * \mathrm{h}(t-1)$$
$$db = -\epsilon * \boxed{dY * W_y} * (1 - tanh^2)$$

$$\frac{dL}{dW_i} = -\epsilon \boxed{\frac{dL}{dy}\frac{dy}{dh(t)}} \frac{dh(t)}{d\tanh} \frac{dtanh}{dW_i}$$

$$y(t) = h(t) * W_y$$

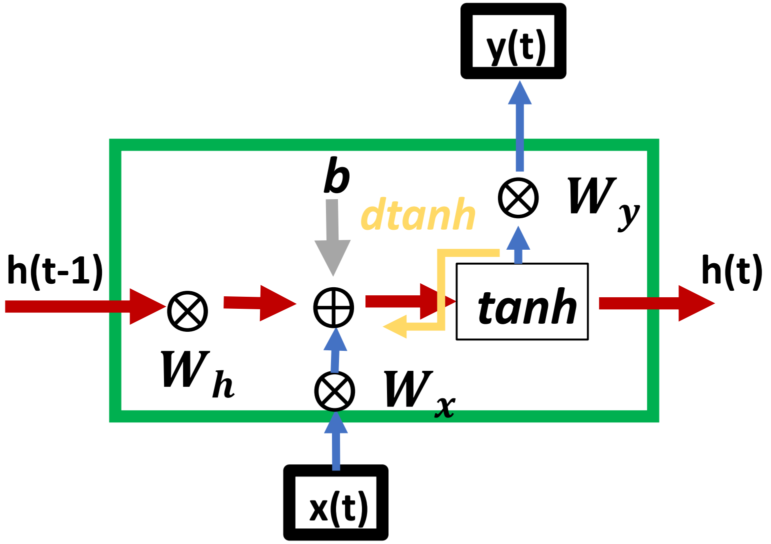


```
def backward(self, dinputs):

...
        Wh        = self.Wh

        dht       = np.dot(Wy.T, dinputs[-1].reshape(1,1))


        for t in reversed(range(T)):

            dy = dinputs[t].reshape(1,1)
            xt = X_t[t].reshape(1,1)

            ACT[t].backward(dht)
            dtanh = ACT[t].dinputs
```

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * h(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

$$\frac{dL}{dW_i} = -\epsilon \frac{dL}{dy} \frac{dy}{dh(t)} \boxed{\frac{dh(t)}{d\tanh}} \frac{dtanh}{dW_i}$$

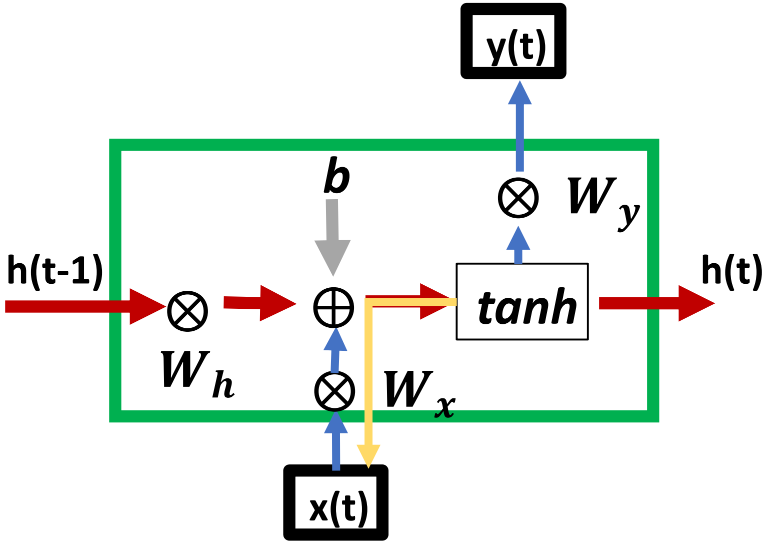
$$h(t) = \tanh[\, x(t) * W_x + W_h * h(t-1) + b \,]$$



```
def backward(self, dinputs):

...
        Wh      = self.Wh

        dht     = np.dot(Wy.T, dinputs[-1].reshape(1,1))


        for t in reversed(range(T)):

            dy = dinputs[t].reshape(1,1)
            xt = X_t[t].reshape(1,1)

            ACT[t].backward(dht)
            dtanh = ACT[t].dinputs

           dWx      += np.dot(dtanh,xt)
```

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * \boxed{x(t)}$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

$$\frac{dL}{dW_i} = -\epsilon \frac{dL}{dy}\frac{dy}{dh(t)}\frac{dh(t)}{d\tanh}\frac{dtanh}{dW_i}$$



```python
def backward(self, dinputs):

...
        Wh      = self.Wh

        dht     = np.dot(Wy.T, dinputs[-1].reshape(1,1))


        for t in reversed(range(T)):

            dy = dinputs[t].reshape(1,1)
            xt = X_t[t].reshape(1,1)

            ACT[t].backward(dht)
            dtanh = ACT[t].dinputs

            dWx      += np.dot(dtanh,xt)
            dWy      += np.dot(H[t+1],dy).T
```

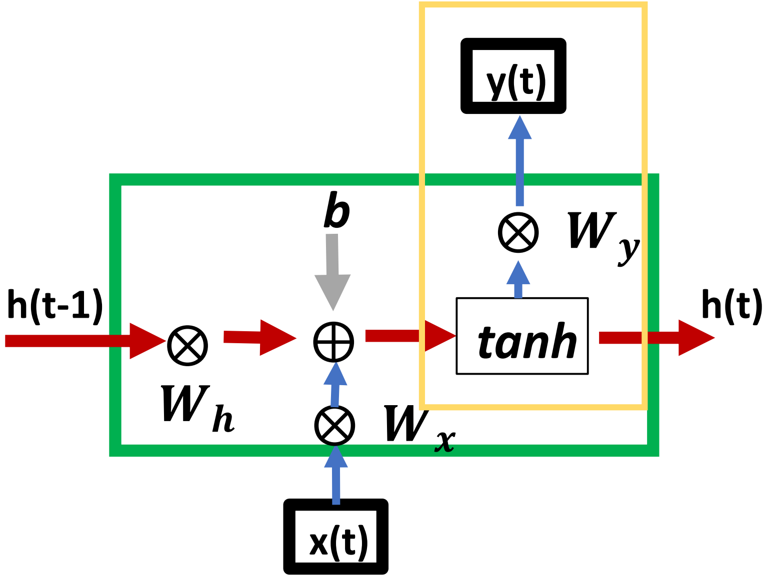
$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$

$$dW_y = -\epsilon * dY * h(t)$$

$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$

$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

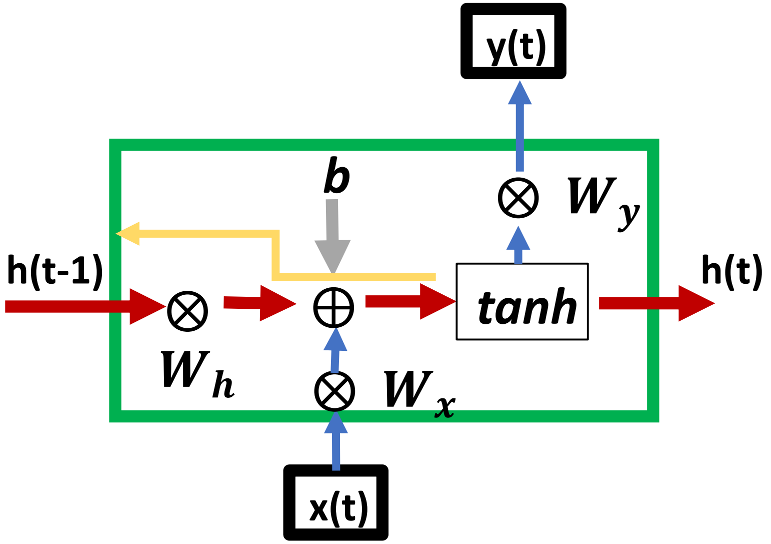
$$\frac{dL}{dW_i} = -\epsilon \frac{dL}{dy} \frac{dy}{dh(t)} \frac{dh(t)}{d \tanh} \frac{dtanh}{dW_i}$$



```
def backward(self, dinputs):

...
        Wh      = self.Wh

        dht     = np.dot(Wy.T, dinputs[-1].reshape(1,1))


        for t in reversed(range(T)):

            dy = dinputs[t].reshape(1,1)
            xt = X_t[t].reshape(1,1)

            ACT[t].backward(dht)
            dtanh = ACT[t].dinputs

            dWx     += np.dot(dtanh,xt)
            dWy     += np.dot(H[t+1],dy).T
            dWh     += np.dot(H[t],dtanh.T)
```

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \boxed{\mathrm{h}(t-1)}$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

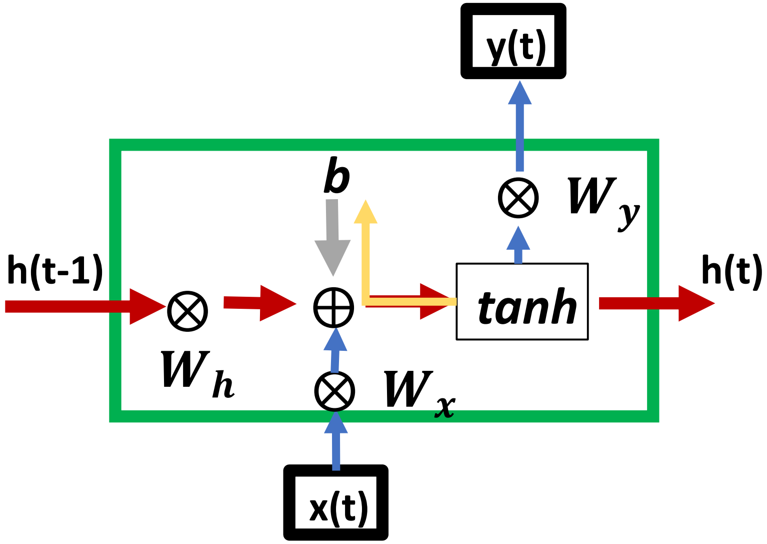
$$\frac{dL}{dW_i} = -\epsilon \frac{dL}{dy}\frac{dy}{dh(t)}\frac{dh(t)}{d\tanh}\frac{dtanh}{dW_i}$$



```
def backward(self, dinputs):

...
        Wh      = self.Wh

        dht     = np.dot(Wy.T, dinputs[-1].reshape(1,1))


        for t in reversed(range(T)):

            dy = dinputs[t].reshape(1,1)
            xt = X_t[t].reshape(1,1)

            ACT[t].backward(dht)
            dtanh = ACT[t].dinputs

           dWx     += np.dot(dtanh,xt)
           dWy     += np.dot(H[t+1],dy).T
           dWh     += np.dot(H[t],dtanh.T)
           dbiases += dtanh
```

$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

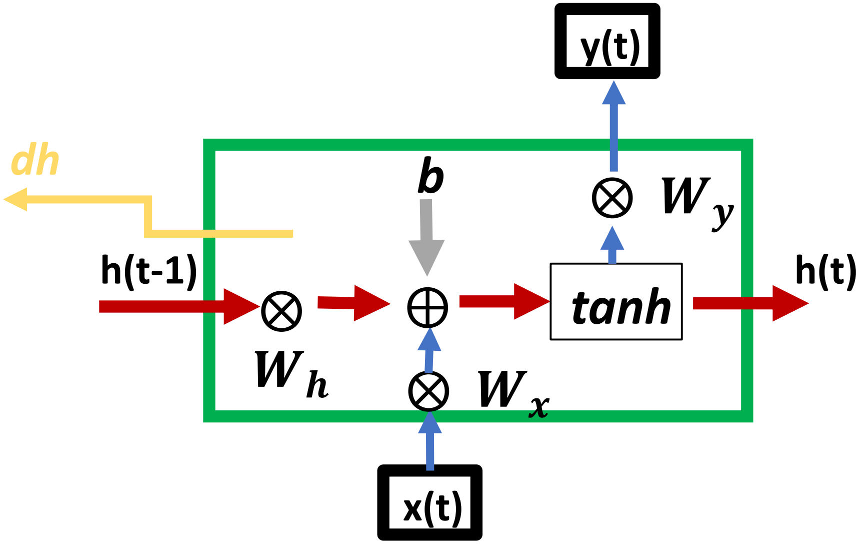
$$\frac{dL}{dW_i} = -\epsilon \frac{dL}{dy} \frac{dy}{dh(t)} \frac{dh(t)}{d\tanh} \frac{dtanh}{dW_i}$$



$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

```
def backward(self, dinputs):

...
        Wh      = self.Wh

        dht     = np.dot(Wy.T, dinputs[-1].reshape(1,1))


        for t in reversed(range(T)):

            dy = dinputs[t].reshape(1,1)
            xt = X_t[t].reshape(1,1)

            ACT[t].backward(dht)
            dtanh = ACT[t].dinputs

            dWx     += np.dot(dtanh,xt)
            dWy     += np.dot(H[t+1],dy).T
            dWh     += np.dot(H[t],dtanh.T)
            dbiases += dtanh

            dht = np.dot(Wh, dtanh)
```

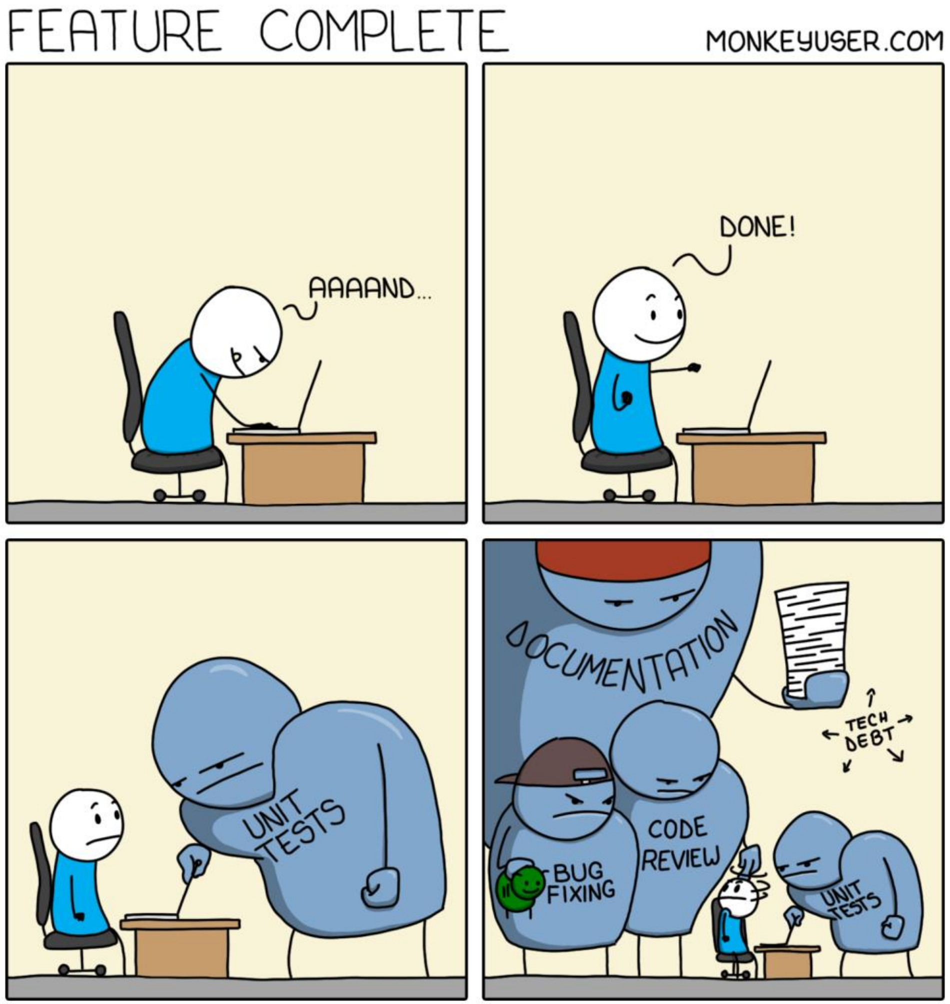
$$h(t) = \tanh[\, x(t) * W_x + W_h * \boxed{h(t-1)} + b\,]$$



$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * \mathrm{h}(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

```python
def backward(self, dinputs):

...
        Wh      = self.Wh

        dht     = np.dot(Wy.T, dinputs[-1].reshape(1,1))


        for t in reversed(range(T)):

...

            dht = np.dot(Wh, dtanh)

            self.dWx      = dWx
            self.dWy      = dWy
            self.dWh      = dWh
            self.dbiases = dbiases

            self.H        = H
```

```
def backward(self, dinputs):

        T       = self.T
        H       = self.H
        X_t     = self.X_t

        ACT     = self.ACT

        dWx     = self.dWx
        dWy     = self.dWy
        dWh     = self.dWh
        Wy      = self.Wy
        Wh      = self.Wh

        dht     = np.dot(Wy.T,dinputs[-1].reshape(1,1))

        dbiases = self.dbiases

        for t in reversed(range(T)):

                dy = dinputs[t].reshape(1,1)
                xt = X_t[t].reshape(1,1)

                ACT[t].backward(dht)
                dtanh = ACT[t].dinputs

                dWx     += np.dot(dtanh,xt)
                dWy     += np.dot(H[t+1],dy).T
                dWh     += np.dot(H[t],dtanh.T)
                dbiases += dtanh

                dht = np.dot(Wh, dtanh)

        self.dWx     = dWx
        self.dWy     = dWy
        self.dWh     = dWh
        self.dbiases = dbiases

        self.H       = H
```



outline:

- the idea
- the RNN cell
- BackPropagation Through Time
- **full backpropagtion**
- creating an SGD optimizer
- creating a full package

The RNN is now ready and it should be able to learn!

```
from RNN import *

n_neurons = 500
rnn       = RNN(X_t, n_neurons, Tanh())
T         = rnn.T
n_epoch   = 20
e         = 1e-5
```

learning rate

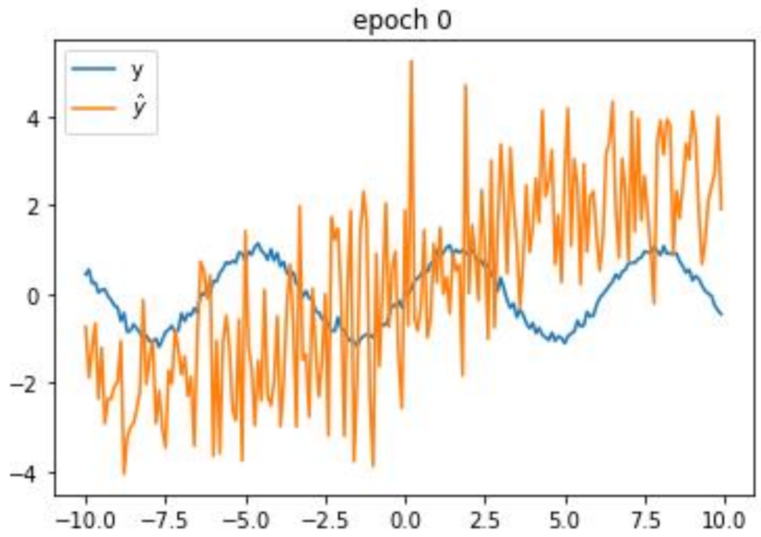
$$dW_x = -\epsilon * dY * W_y * (1 - tanh^2) * x(t)$$
$$dW_y = -\epsilon * dY * h(t)$$
$$dW_h = -\epsilon * dY * W_y * (1 - tanh^2) * h(t-1)$$
$$db = -\epsilon * dY * W_y * (1 - tanh^2)$$

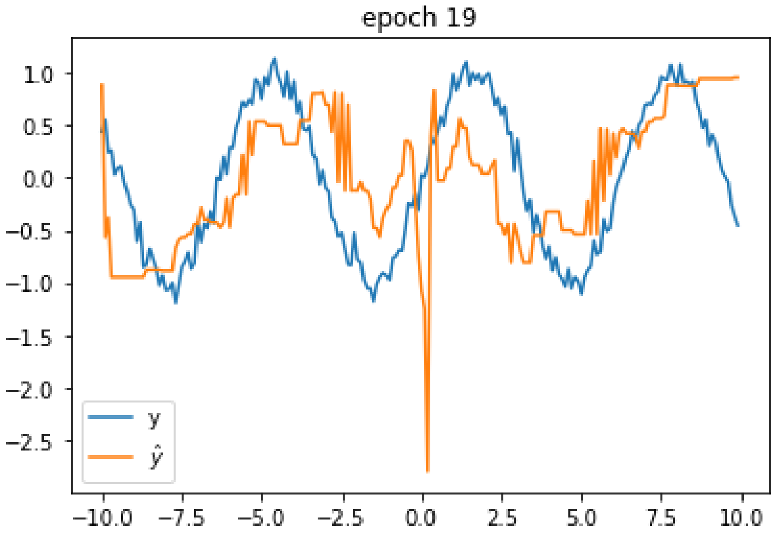
```
for n in range(n_epoch):

    rnn.forward()

    Y_hat        = rnn.Y_hat
    dY           = Y_hat - Y_t
    L            = 0.5*np.dot(dY.T,dY)/T
    print(float(L))

    rnn.backward(dY)

    rnn.Wx     -= e* rnn.dWx
    rnn.Wy     -= e* rnn.dWy
    rnn.Wh     -= e* rnn.dWh
    rnn.biases -= e* rnn.dbiases

    plt.plot(X_t, Y_t)
    plt.plot(X_t, Y_hat)
    plt.legend(['y', '$\hat{y}$'])
    plt.title('epoch ' + str(n))
    plt.show()
```

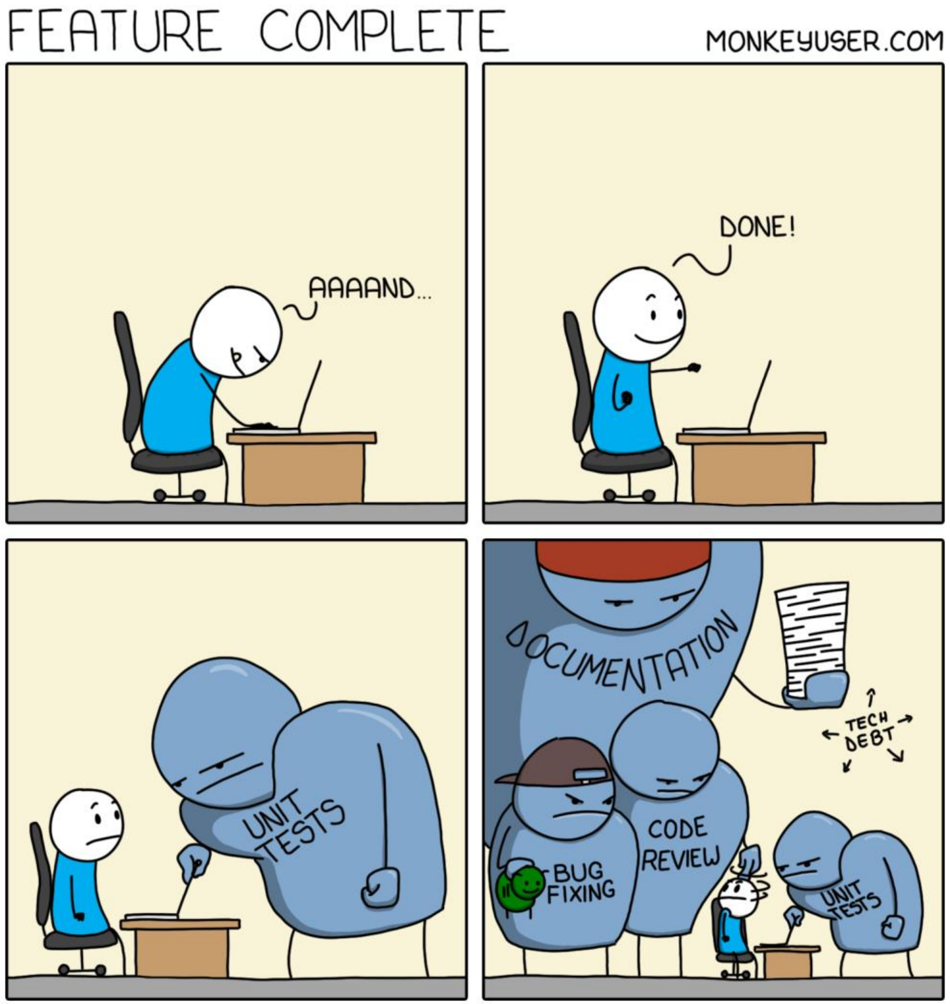
```
from RNN import *

...

for n in range(n_epoch):

    rnn.forward()

    Y_hat= rnn.Y_hat
    dY          = Y_hat - Y_t
    L           = 0.5*np.dot(dY.T,dY)/T
    print(float(L))

    rnn.backward(dY)

    rnn.Wx      -= e* rnn.dWx
    rnn.Wy      -= e* rnn.dWy
    rnn.Wh      -= e* rnn.dWh
    rnn.biases  -= e* rnn.dbiases

    plt.plot(X_t, Y_t)
    plt.plot(X_t, Y_hat)
    plt.legend(['y', '$\hat{y}$'])
    plt.title('epoch ' + str(n))
    plt.show()
```

L

```
2.4421365086946225
3.643908836529324
0.4021025933602323
0.3648829976898011
0.3484465922481833
0.33344522087248
0.31965178295937685
0.3069639694033091
0.29528875247232894
0.284540967482768
0.27464264441702246
0.2655223998379806
0.2571148802216734
0.249360252339423
0.2422037367127193
0.23559518050029948
0.22948866648462166
0.2238421551065919
0.21861715675530824
0.21377843175565578
```







outline:

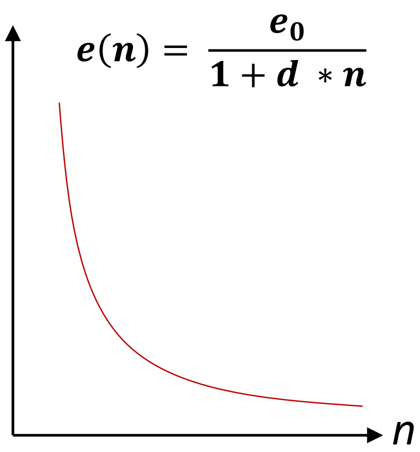
- *the idea*
- *the RNN cell*
- *BackPropagation Through Time*
- *full backpropagtion*
- ***creating an SGD optimizer***
- *creating a full package*

- the optimizer will be pretty similar to the one from *"ANN from Scratch"*
- with momentum and learning rate decay

```
class Optimizer_SGD:

    def __init__(self, learning_rate = 0.001, decay = 0, momentum = 0):

        self.learning_rate         = learning_rate
        self.current_learning_rate = learning_rate
        self.decay                 = decay
        self.iterations            = 0
        self.momentum              = momentum
```

for more details see my *"ANN from Scratch"* lecture ☺

```
class Optimizer_SGD:

        def __init__(self, learning_rate = 0.001, decay = 0, momentum = 0):

        ...
                self.momentum                   = momentum


        def pre_update_params(self):
                if self.decay:
                        self.current_learning_rate = self.learning_rate * \
                            (1/ (1 + self.decay*self.iterations))
```



e:	learning rate
d:	decay
n:	epochs/iterations

for more details see my *"ANN from Scratch"* lecture

```
class Optimizer_SGD:

        def __init__(self, learning_rate = 0.001, decay = 0, momentum = 0):

        ...
                self.momentum                   = momentum

    def pre_update_params(self):
            if self.decay:
                    self.current_learning_rate = self.learning_rate * \
                        (1/ (1 + self.decay*self.iterations))


    def post_update_params(self):
            self.iterations += 1
```

Just counting the iterations/epochs for current learning rate

```
class Optimizer_SGD:
        ...
    def update_params(self, layer):

        if self.momentum:

            if not hasattr(layer, 'Wx_momentums'):
                layer.Wx_momentums      = np.zeros_like(layer.Wx)
                layer.Wy_momentums      = np.zeros_like(layer.Wy)
                layer.Wh_momentums      = np.zeros_like(layer.Wh)
                layer.bias_momentums    = np.zeros_like(layer.biases)
```

for more details see my *"ANN from Scratch"* lecture

for more details see my *"ANN from Scratch"* lecture

```python
class Optimizer_SGD:
        ...
    def update_params(self, layer):

            if self.momentum:

                if not hasattr(layer, 'Wx_momentums'):
                    layer.Wx_momentums      = np.zeros_like(layer.Wx)
                    layer.Wy_momentums      = np.zeros_like(layer.Wy)
                    layer.Wh_momentums      = np.zeros_like(layer.Wh)
                    layer.bias_momentums    = np.zeros_like(layer.biases)

                Wx_updates = self.momentum * layer.Wx_momentums - \
                    self.current_learning_rate * layer.dWx
                layer.Wx_momentums = Wx_updates

                Wy_updates = self.momentum * layer.Wy_momentums - \
                    self.current_learning_rate * layer.dWy
                layer.Wy_momentums = Wy_updates

                Wh_updates = self.momentum * layer.Wh_momentums - \
                    self.current_learning_rate * layer.dWh
                    layer.Wh_momentums = Wh_updates

                bias_updates = self.momentum * layer.bias_momentums - \
                    self.current_learning_rate * layer.dbiases
                    layer.bias_momentums = bias_updates
```

```
class Optimizer_SGD:
        ...
    def update_params(self, layer):

        if self.momentum:

            if not hasattr(layer, 'Wx_momentums'):
                layer.Wx_momentums     = np.zeros_like(layer.Wx)

                ...

            bias_updates = self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
                layer.bias_momentums = bias_updates

        else:

            Wx_updates      = -self.current_learning_rate * layer.dWx
            Wy_updates      = -self.current_learning_rate * layer.dWy
            Wh_updates      = -self.current_learning_rate * layer.dWh
            bias_updates    = -self.current_learning_rate * layer.dbiases

        layer.Wx       += Wx_updates
        layer.Wy       += Wy_updates
        layer.Wh       += Wh_updates
        layer.biases   += bias_updates
```

for more details see my *"ANN from Scratch"* lecture

```
rnn.Wx -= e* rnn.dWx
rnn.Wy -= e* rnn.dWy
rnn.Wh -= e* rnn.dWh
```

```
from RNN import *

n_neurons = 500
rnn       = RNN(X_t, n_neurons, Tanh())
optimizer = Optimizer_SGD(learning_rate = 1e-5, momentum = 0.95)
T         = rnn.T
n_epoch   = 200
Monitor   = np.zeros((n_epoch,1))


for n in range(n_epoch):

    rnn.forward()

    Y_hat     = rnn.Y_hat
    dY        = Y_hat - Y_t
    L         = 0.5*np.dot(dY.T,dY)/T
```

we want to keep track of the loss function

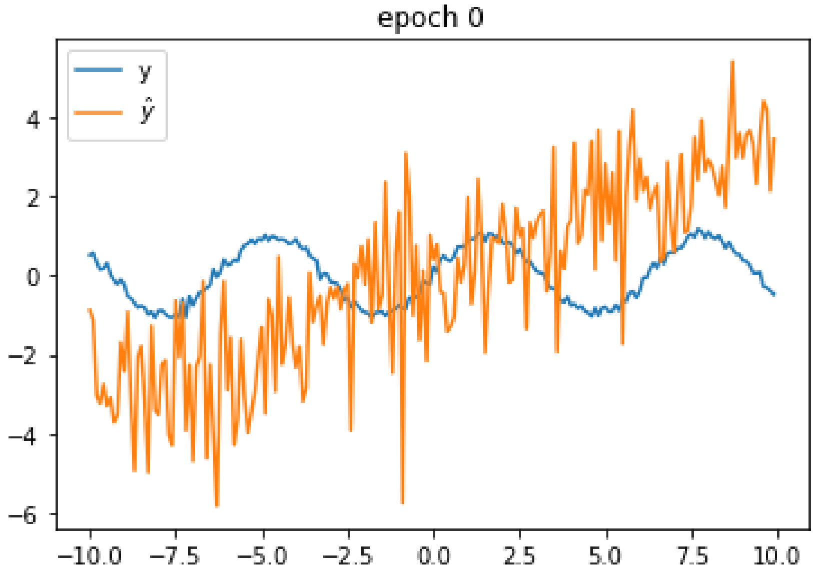
```
from RNN import *

n_neurons = 500
rnn       = RNN(X_t, n_neurons, Tanh())
optimizer = Optimizer_SGD(learning_rate = 1e-5, momentum = 0.95)
T         = rnn.T
n_epoch   = 200
Monitor   = np.zeros((n_epoch,1))

for n in range(n_epoch):

    rnn.forward()

    Y_hat     = rnn.Y_hat
    dY        = Y_hat - Y_t
    L         = 0.5*np.dot(dY.T,dY)/T

    Monitor[n] = L

    rnn.backward(dY)

    optimizer.pre_update_params()
    optimizer.update_params(rnn)
    optimizer.post_update_params()
```

we want to keep track of the loss function

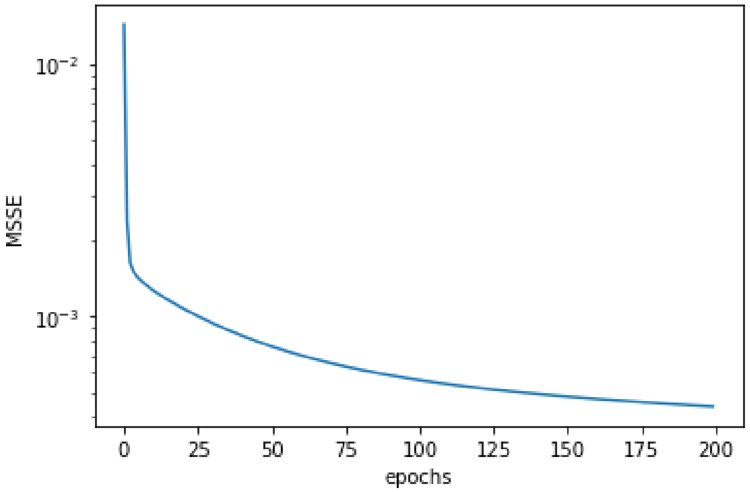
updating W_x, W_y, W_h, biases and learning rate

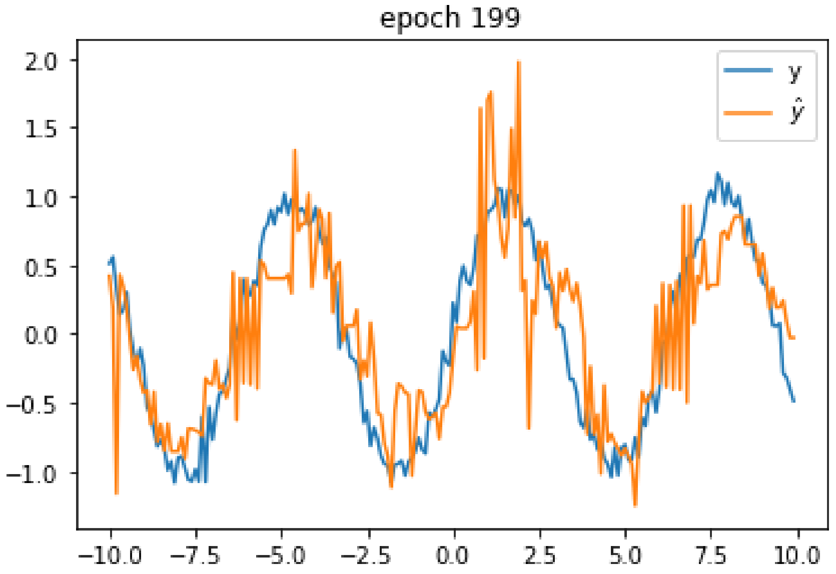
```python
from RNN import *

n_neurons = 500
rnn       = RNN(X_t, n_neurons, Tanh())
optimizer = Optimizer_SGD(learning_rate = 1e-5, momentum = 0.95)
T         = rnn.T
n_epoch   = 200
Monitor   = np.zeros((n_epoch,1))

for n in range(n_epoch):

...
    optimizer.post_update_params()

    r = n/50
    if r - np.ceil(r) == 0:
        plt.plot(X_t, Y_t)
        plt.plot(X_t, Y_hat)
        plt.legend(['y', '$\hat{y}$'])
        plt.title('epoch ' + str(n))
        plt.show()
```

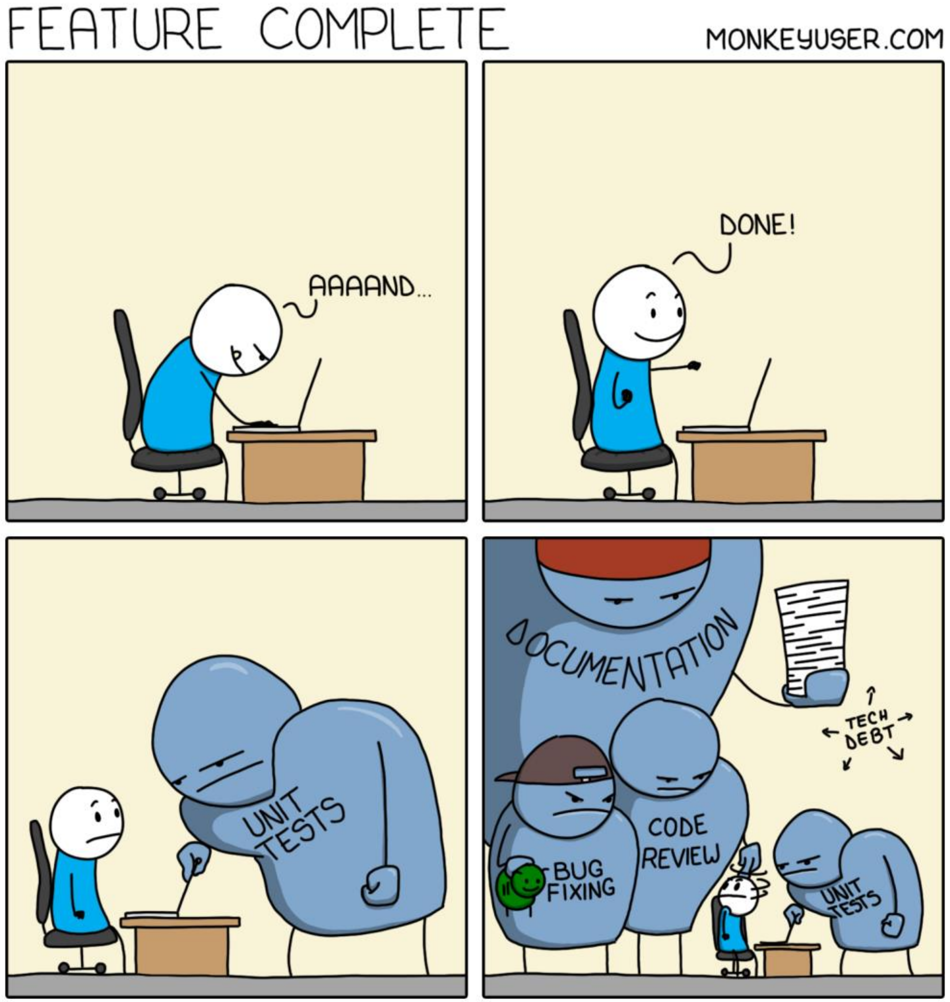
```python
for n in range(n_epoch):

...
     optimizer.post_update_params()

    r = n/50
    if r - np.ceil(r) == 0:
          plt.plot(X_t, Y_t)
          plt.plot(X_t, Y_hat)
          plt.legend(['y', '$\hat{y}$'])
          plt.title('epoch ' + str(n))
          plt.show()
```




```python
plt.plot(range(n_epoch), Monitor)
plt.xlabel('epochs')
plt.ylabel('MSSE')
plt.yscale('log')
plt.show()
```







outline:

- *the idea*
- *the RNN cell*
- *BackPropagation Through Time*
- *full backpropagtion*
- *creating an SGD optimizer*
- *creating a full package*

The RNN works now: → creating a function for training and applying the RNN

```
import matplotlib.pyplot as plt

def RunMyRNN(X_t, Y_t, Activation, n_epoch = 1000, n_neurons = 800,
             learning_rate = 1e-5, decay = 0, momentum = 0.95):

    rnn       = RNN(X_t, n_neurons, Activation)
    optimizer = Optimizer_SGD(learning_rate, decay, momentum)
    T         = rnn.T
    Monitor   = np.zeros((n_epoch,1))

    print("RNN is running...")
```

it is nice for the user to see that the code is running

The RNN works now: → creating a function for **training** and applying the RNN

```
def RunMyRNN(X_t, Y_t, Activation, n_epoch = 1000, n_neurons = 800,
             learning_rate = 1e-5, decay = 0, momentum = 0.95):
...

    print("RNN is running...")

    for n in range(n_epoch):

        rnn.forward()

        dY = rnn.Y_hat - Y_t
        L  = 0.5*np.dot(dY.T,dY)/T

        rnn.backward(dY)


        optimizer.pre_update_params()
        optimizer.update_params(rnn)
        optimizer.post_update_params()

        Monitor[n] = L
```

we just copy/paste the lines from the test runs

The RNN works now: → creating a function for **training** and applying the RNN

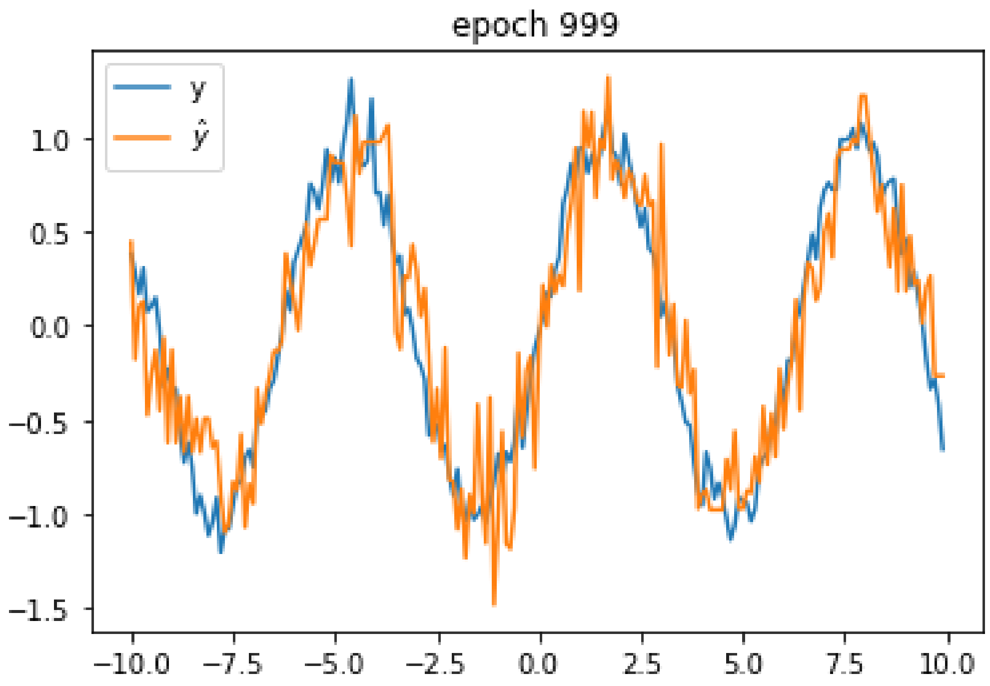
```
...
        Monitor[n] = L

        r = n/100
        if r - np.ceil(r) == 0:
                plt.plot(X_t, Y_t)
                plt.plot(X_t, Y_hat)
                plt.legend(['y', '$\hat{y}$'])
                plt.title('epoch ' + str(n))
                plt.show()


        plt.plot(range(n_epoch), Monitor/T)
        plt.xlabel('epochs’)
        plt.ylabel('MSSE’)
        plt.yscale('log’)
        plt.show()
```

The RNN works now: → creating a function for **training** and applying the RNN

```
...
        plt.show()

        L = float(L)

        print(f'Done! MSSE = {L:.3f}')

        return(rnn)
```

we will need the weights for the application of the trained RNN to new data

```
In [57]: from RNN import *

In [58]: RunMyRNN(X_t,Y_t, Tanh())
RNN is running...
Done! MSSE = 0.040
```



The RNN works now: → creating a function for training and **applying** the RNN

```
def ApplyMyRNN(X_t, rnn):

        T     = max(X_t.shape)
        Y_hat = np.zeros((T, 1))

        H     = rnn.H
        ht    = H[0]
        H     = [np.zeros((rnn.n_neurons,1)) for t in range(T+1)]


        [_,_,Y_hat]  = rnn.RNNCell(X_t, ht, rnn.ACT, H, Y_hat)


        plt.plot(X_t, Y_hat)
        plt.legend('$\hat{y}$')
        plt.show()

        return(Y_hat)
```

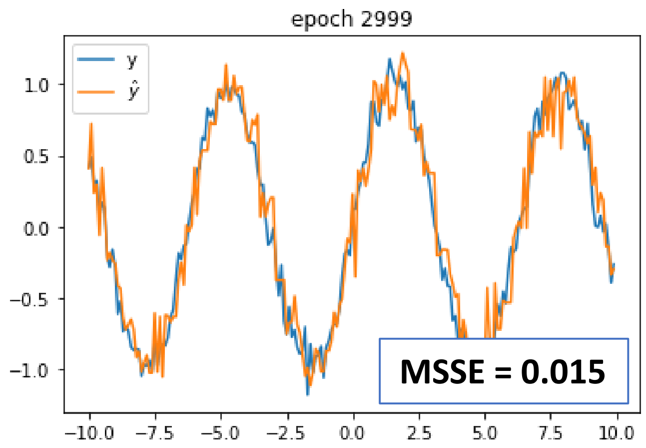
X_t is now the **new** data set and rnn the trained network

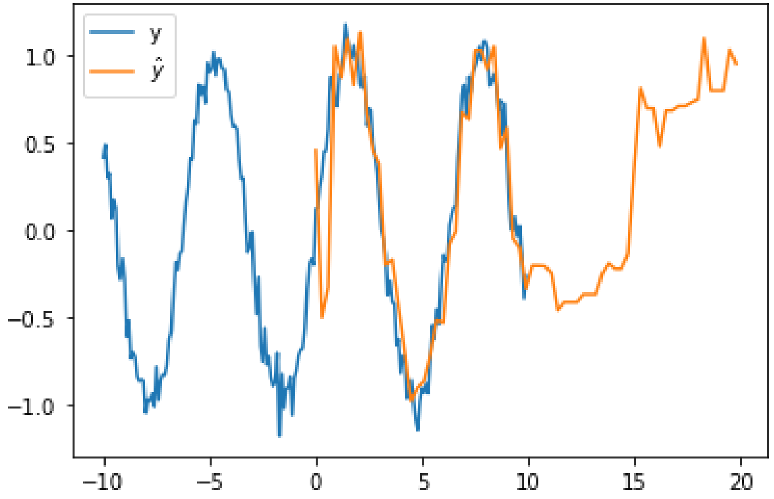
We only need the forward part

The RNN works now: → creating a function for training and **applying** the RNN

```
from RNN import *

rnn = RunMyRNN(X_t,Y_t, Tanh(), n_epoch = 3000)

X_new = np.arange(0, 20, 0.3)
X_new = X_new.reshape(len(X_new), 1)

Y_hat = ApplyMyRNN(X_new,rnn)


plt.plot(X_t, Y_t)
plt.plot(X_new, Y_hat)
plt.legend(['y', '$\hat{y}$'])
plt.title('epoch ' + str(n))
plt.show()
```

let us try a X_new with different spacing and different length than X_t

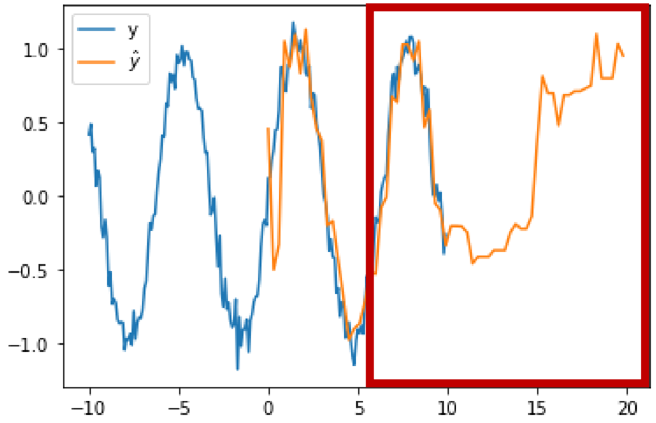




summary and final words:

+ simple, can be implemented easily
+ help to understand more complex architectures

- due to BPTT: RNNs suffer from exploding/vanishing gradient
  → `np.clip`
- training often fails in particular for long sequences
- slow, can't be parallelized
- other prediction methods outperformed RNNs at the time
- short memory (need "context")
  → solved by LSTMs, self – attention



Thank you very much for your attention!