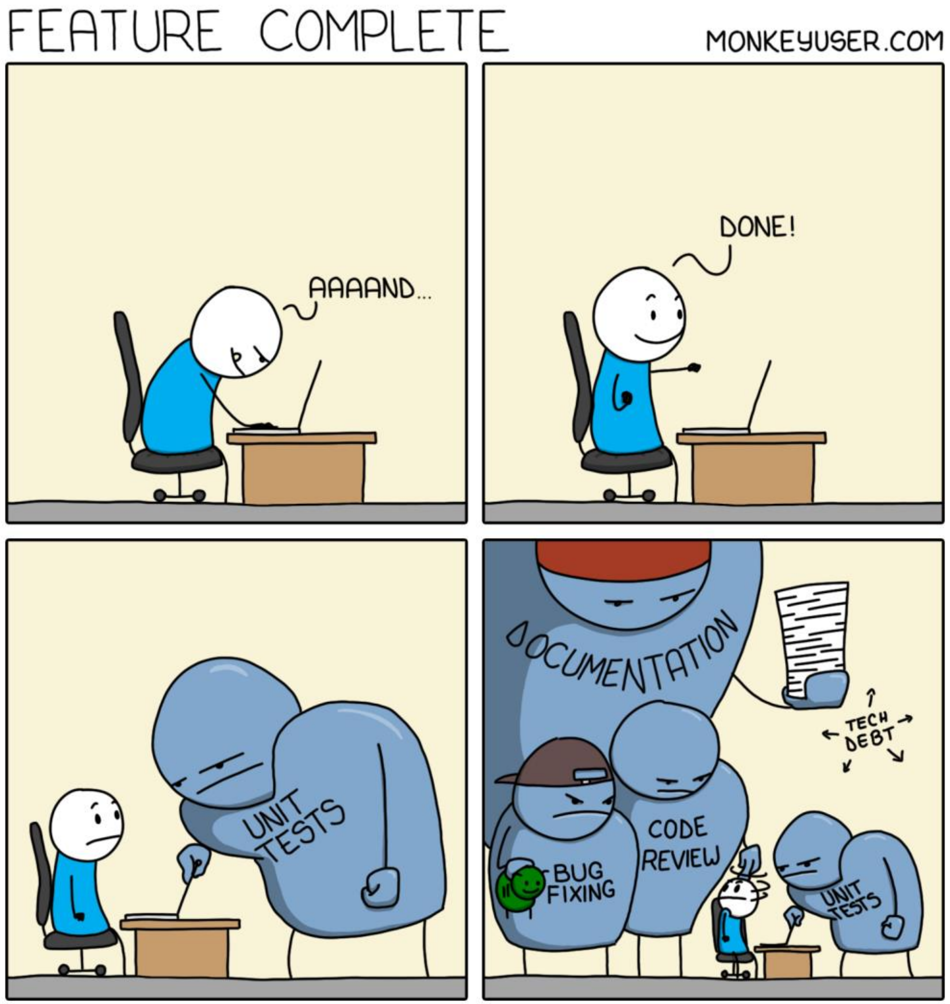
Long Short-Term Memory network – from Scratch

https://www.analyticsvidhya.com



<u>you should be fluent with:</u>

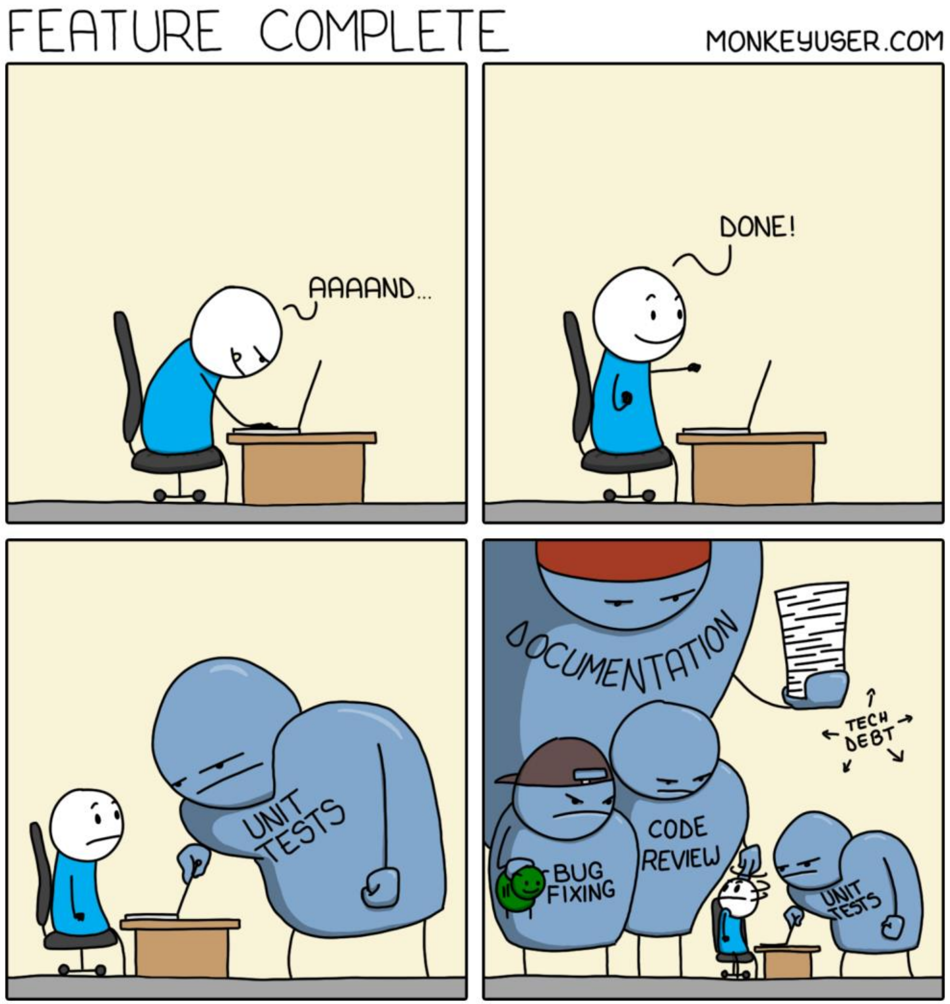
- *basic* ***OOP*** *(methods, classes, inheritance)*

- ***linear algebra*** *(dot product, inner product, outer product)*

- ***derivatives*** *(gradient)*

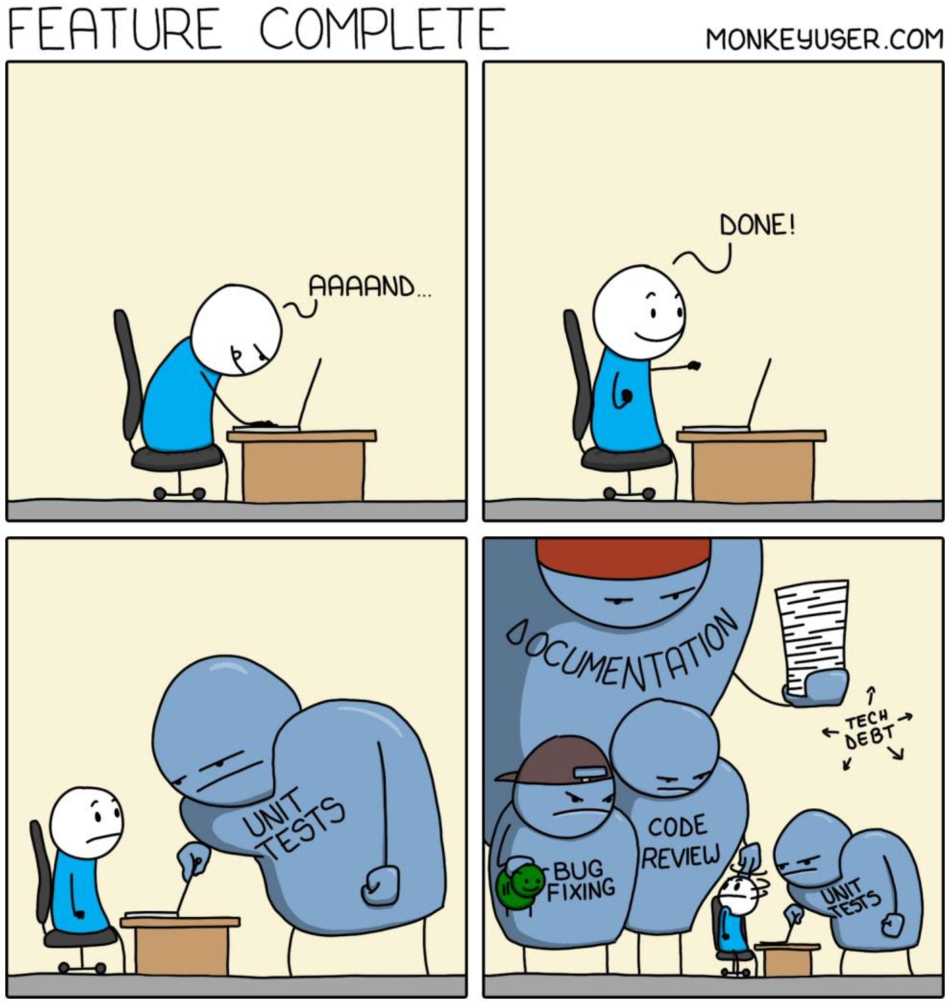
+

- ***Codes*** *from "RNN from Scratch"*



outline:

- *the idea*
- *from a classical RNN cell to a LSTM*
- ***Back**Propagation **T**hrough **T**ime*
- *full backpropagtion*
- *modifying the SGD optimizer*
- *running and testing the package*



outline:

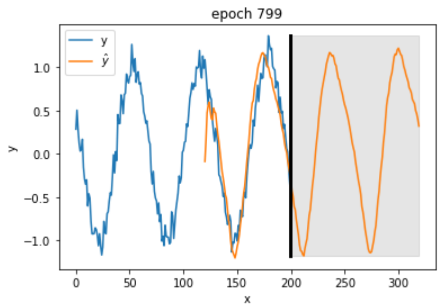
- ***the idea***
- *from a classical RNN cell to a LSTM*
- *BackPropagation Through Time*
- *full backpropagtion*
- *modifying the SGD optimizer*
- *running and testing the package*

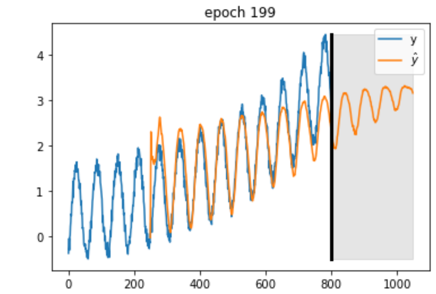
- time series analysis (prediction and forecasting)
- speech recognition
- anomaly detection

<u>new:</u>

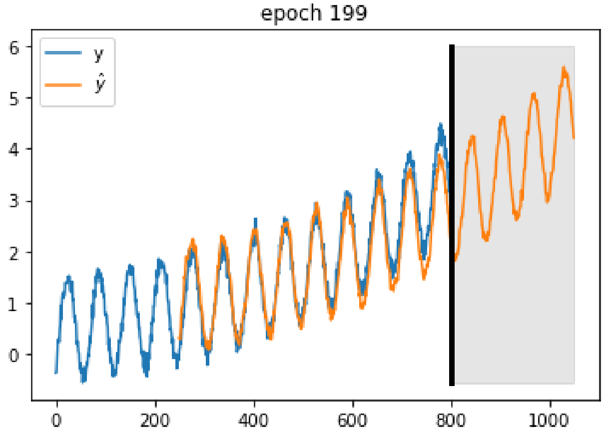
- **long-term** and **short-term** memory
- dealing with vanishing/exploding gradient
- invented 1997 by Sepp Hochreiter und Jürgen Schmidhuber

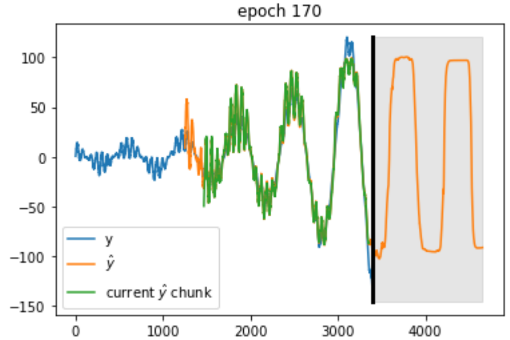
classical RNN





LSTM

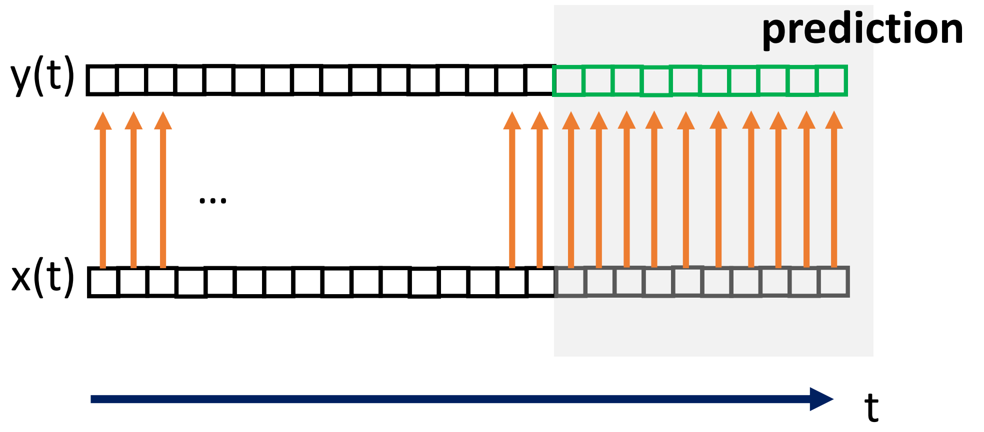


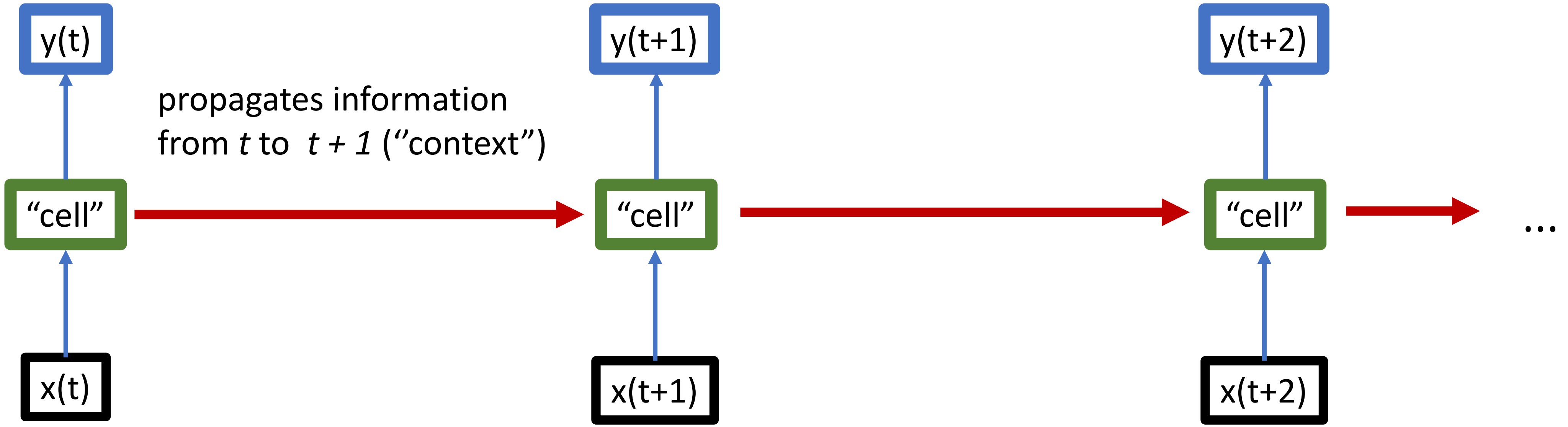


(adding more noise)

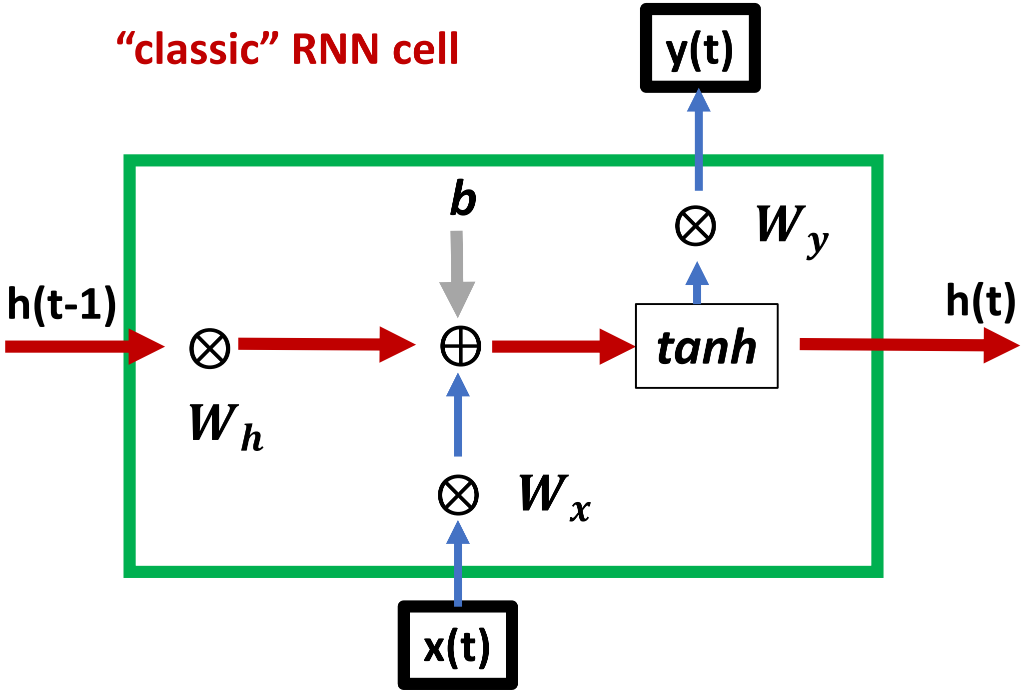
<u>Aim of the lecture:</u>

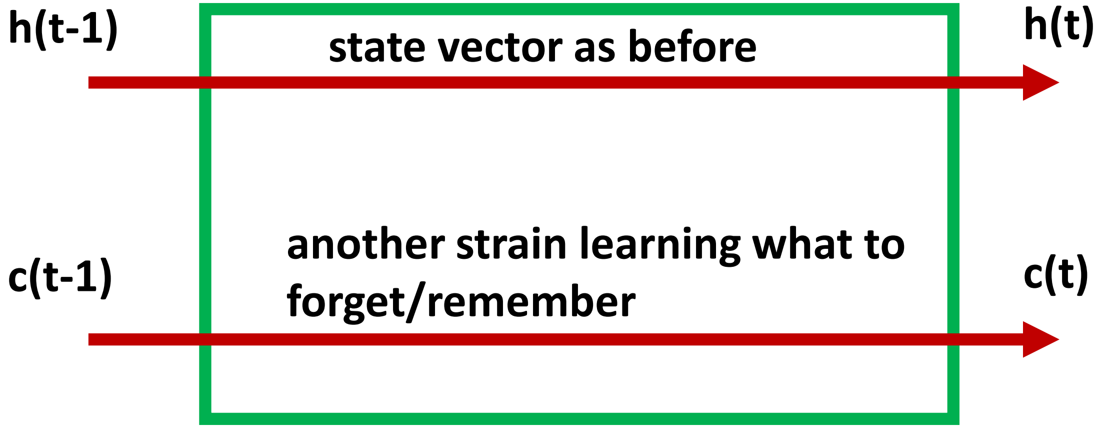
building a **many – to – many,**
one feature LSTM from our **previous RNN**

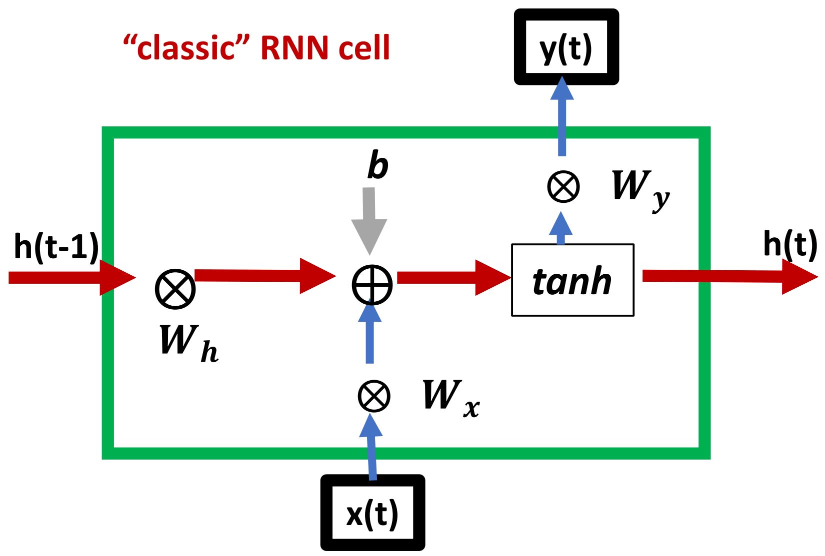

prediction
y(t)
...
x(t)
t

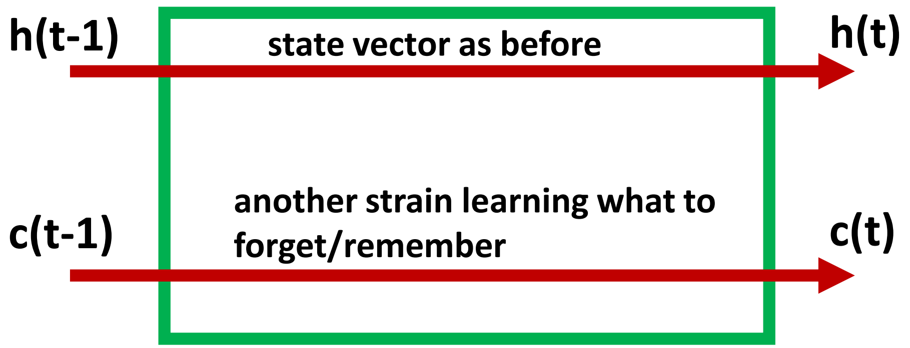

y(t)
y(t+1)
y(t+2)
propagates information
from t to t + 1 (”context”)
“cell”
“cell”
“cell”
...
x(t)
x(t+1)
x(t+2)

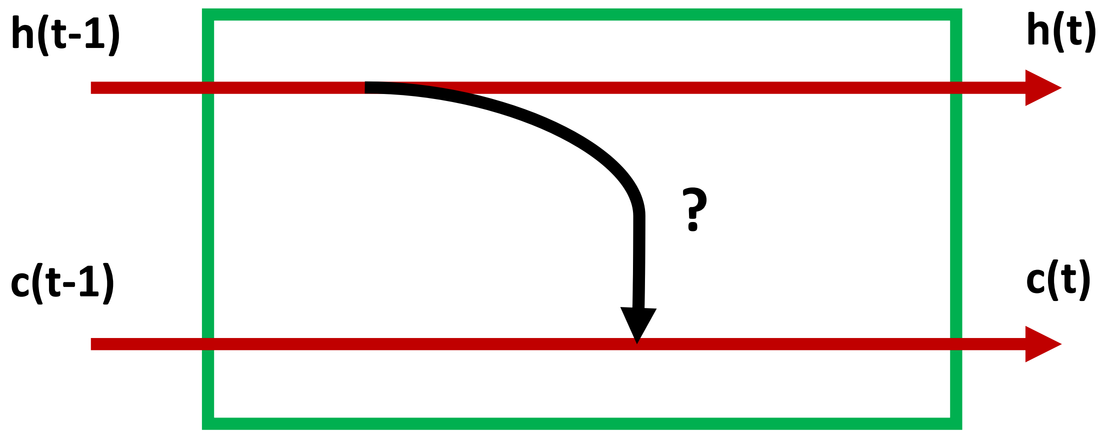
"classic" RNN cell

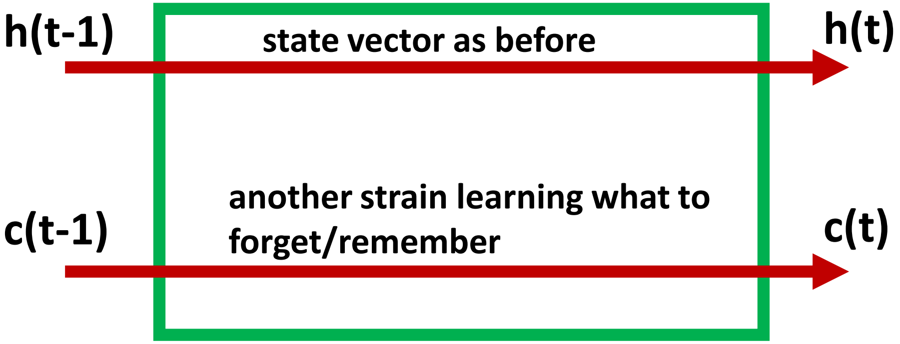


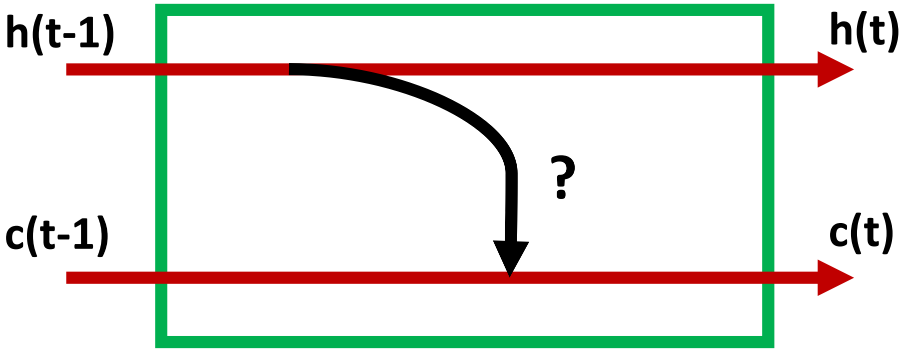



"classic" RNN cell
y(t)
b
W_y
h(t-1)
tanh
h(t)
W_h
W_x
x(t)

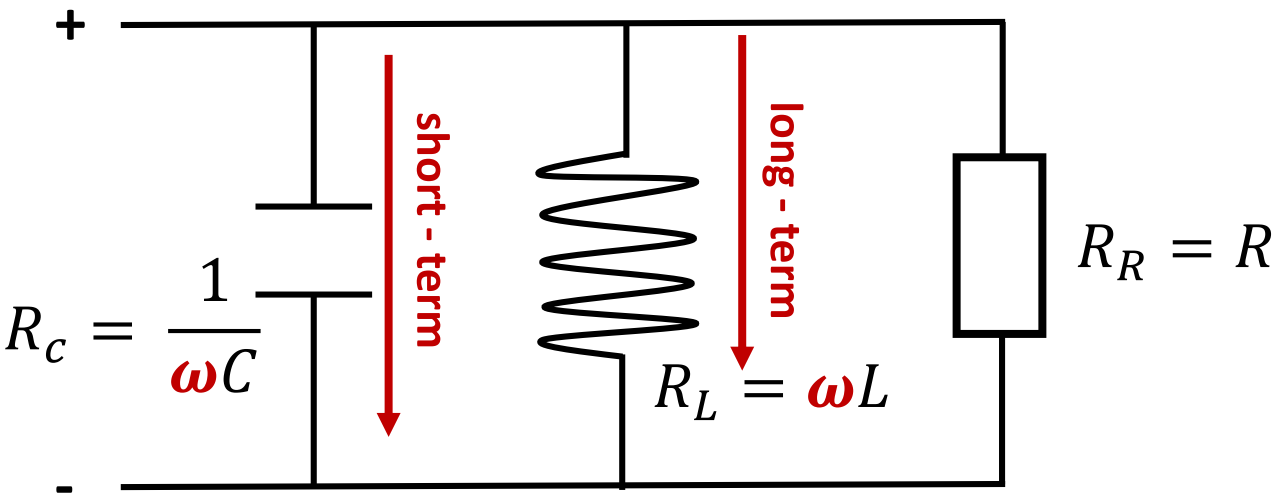

h(t-1)
state vector as before
h(t)
c(t-1)
another strain learning what to forget/remember
c(t)


h(t-1)
h(t)
?
c(t-1)
c(t)





electrical circuits:

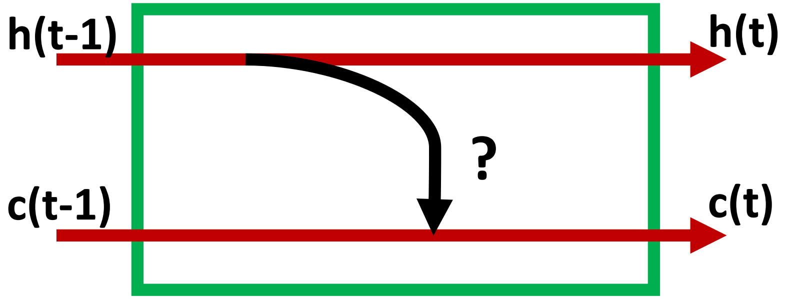


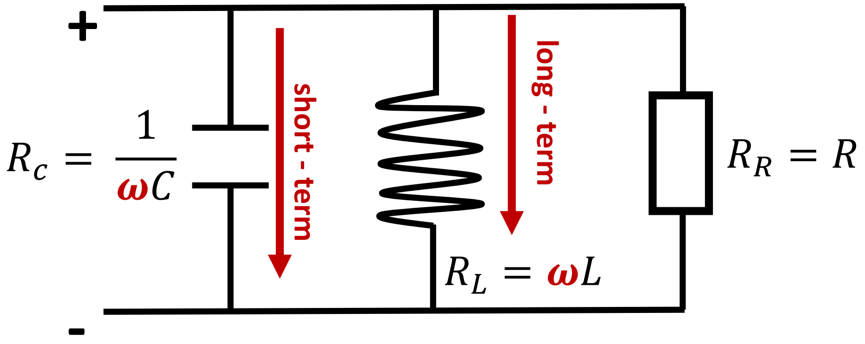
AC: $I(t) = I_0\, e^{i(\omega t + \varphi)}$

R_c: passes **short** -term changes

R_L: passes **long** -term changes

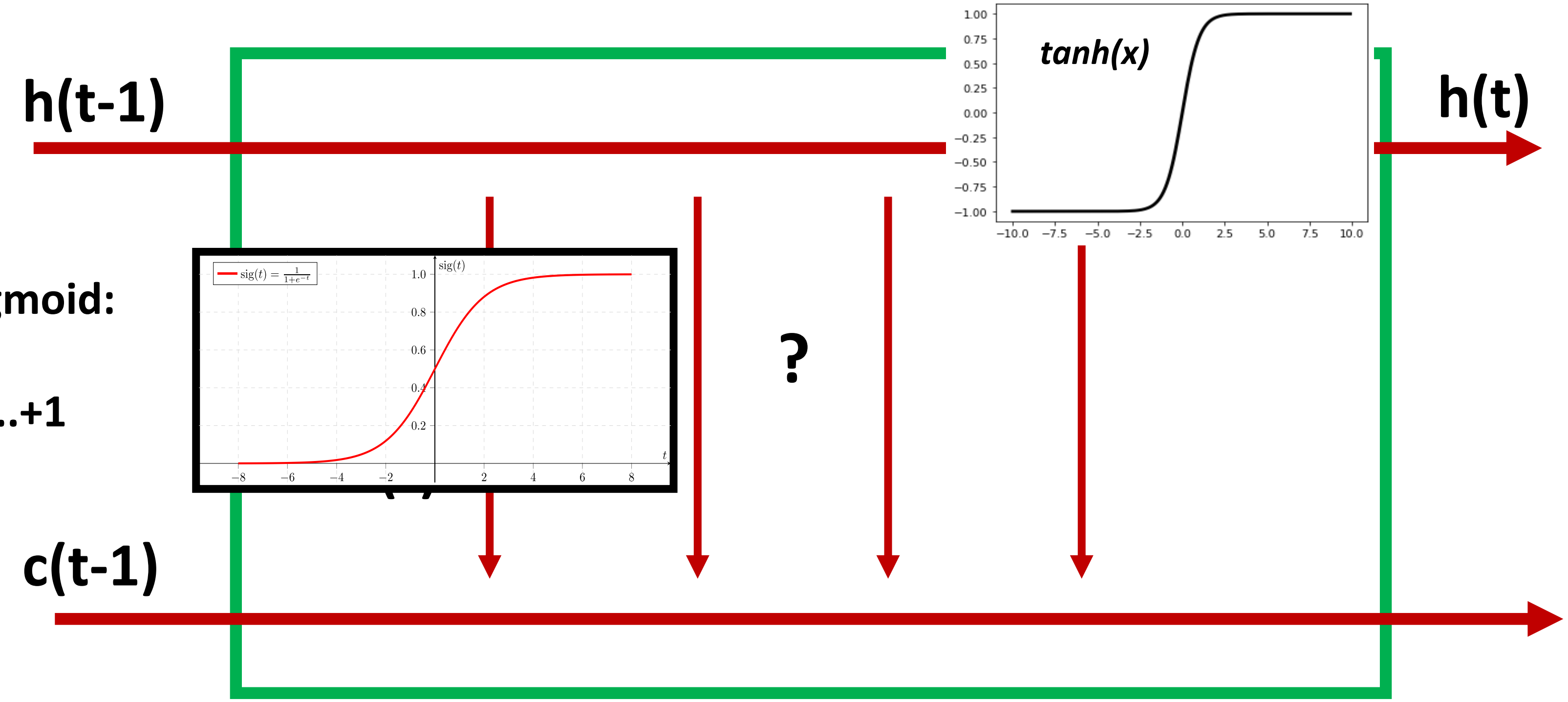
$$\frac{1}{R_{tot}} = \frac{1}{R_R} + \frac{1}{R_C} + \frac{1}{R_L}$$





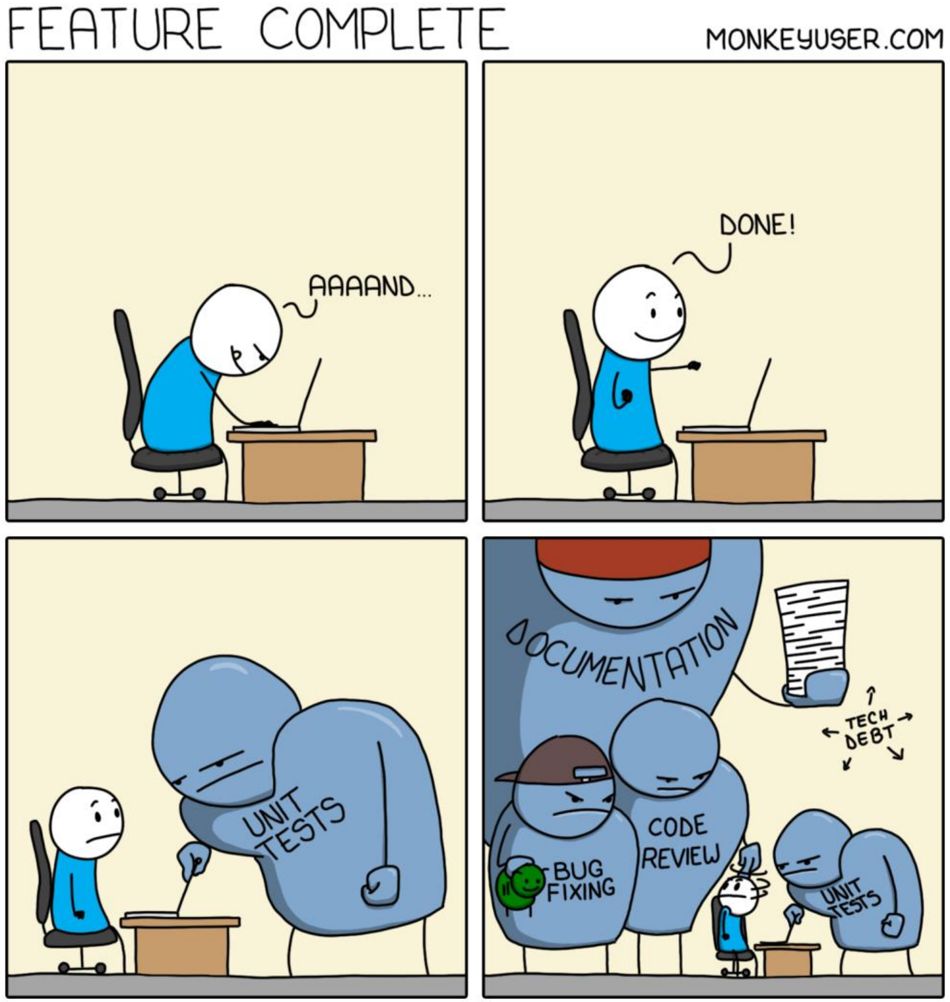
sigmoid:

0….+1



sigmoid:
learning whether information has to be passed on

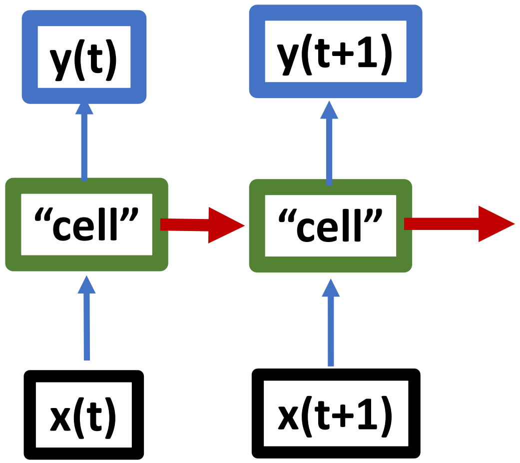
thanh:
state vector as for RNN

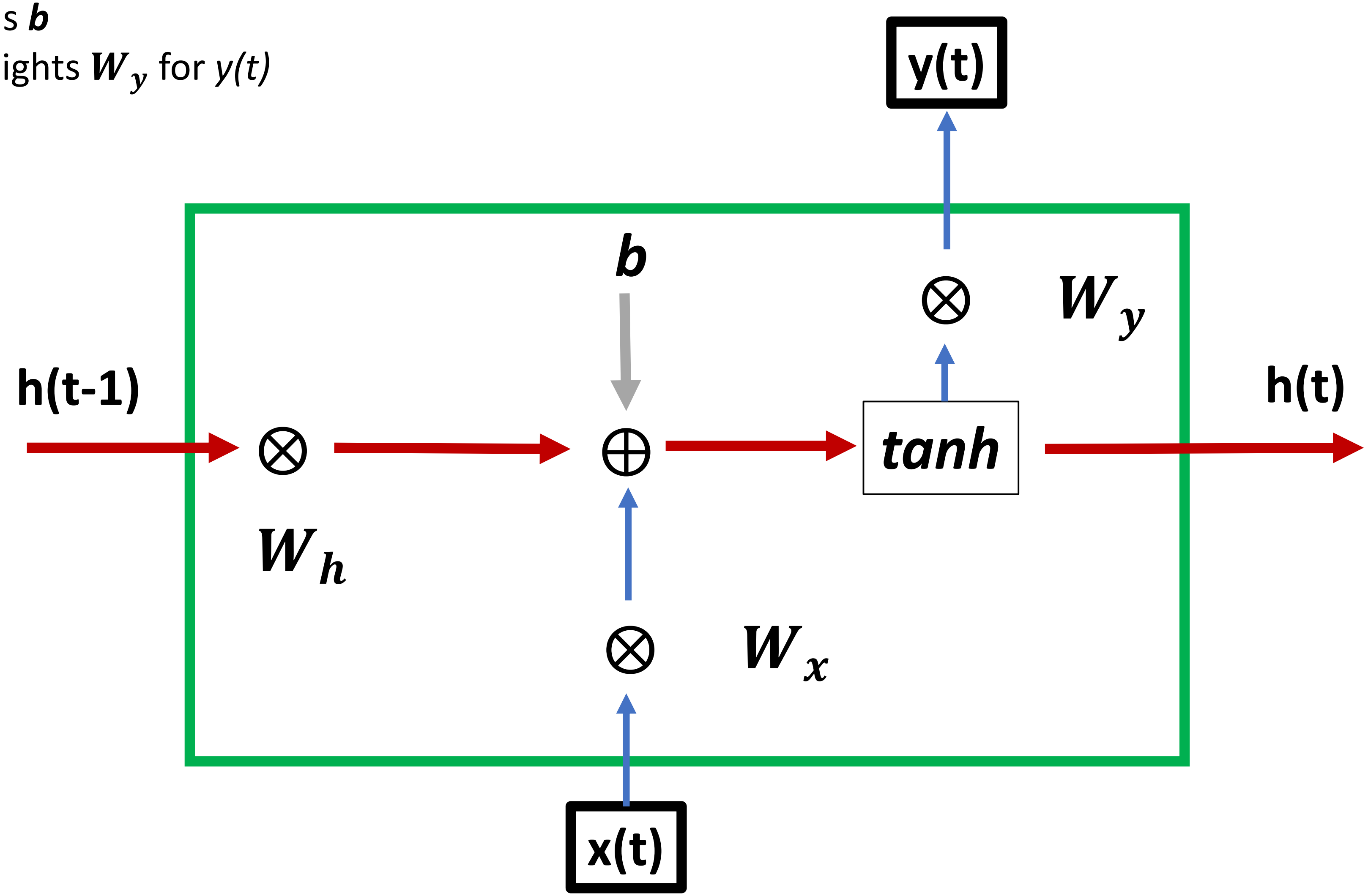


outline:

- *the idea*
- ***from a classical RNN cell to a LSTM***
- *BackPropagation Through Time*
- *full backpropagtion*
- *modifying the SGD optimizer*
- *running and testing the package*

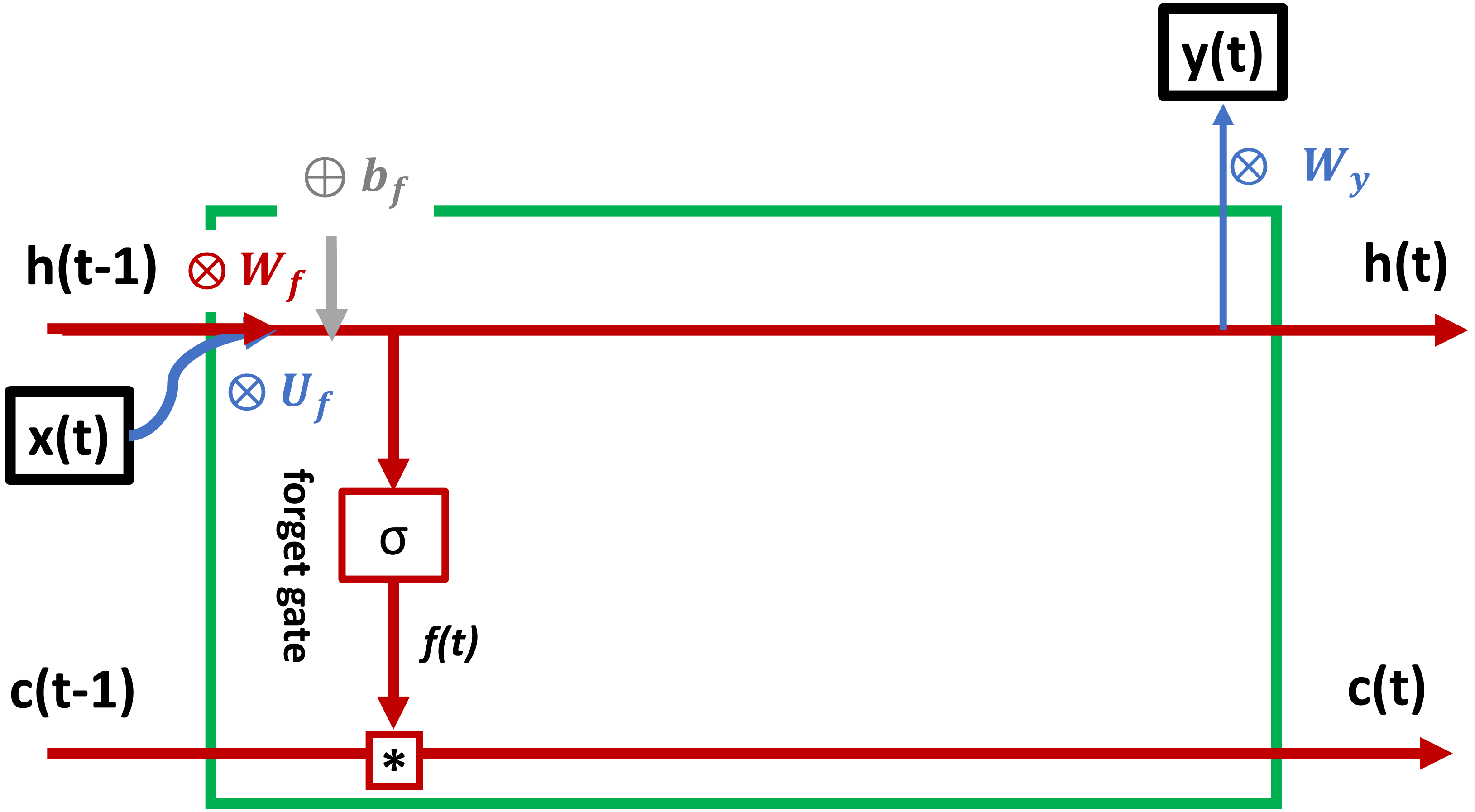
- **state vector** *h(t)*, i. e. the *"memory"* the system has along time
- weights $\boldsymbol{W_x}$ for *x(t)*
- weights $\boldsymbol{W_h}$ for *h(t)*
- bias $\boldsymbol{b}$
- weights $\boldsymbol{W_y}$ for *y(t)*





$$f(t) = \sigma\left(U_f \oplus x(t) + W_f \oplus h(t-1) + b_f\right)$$

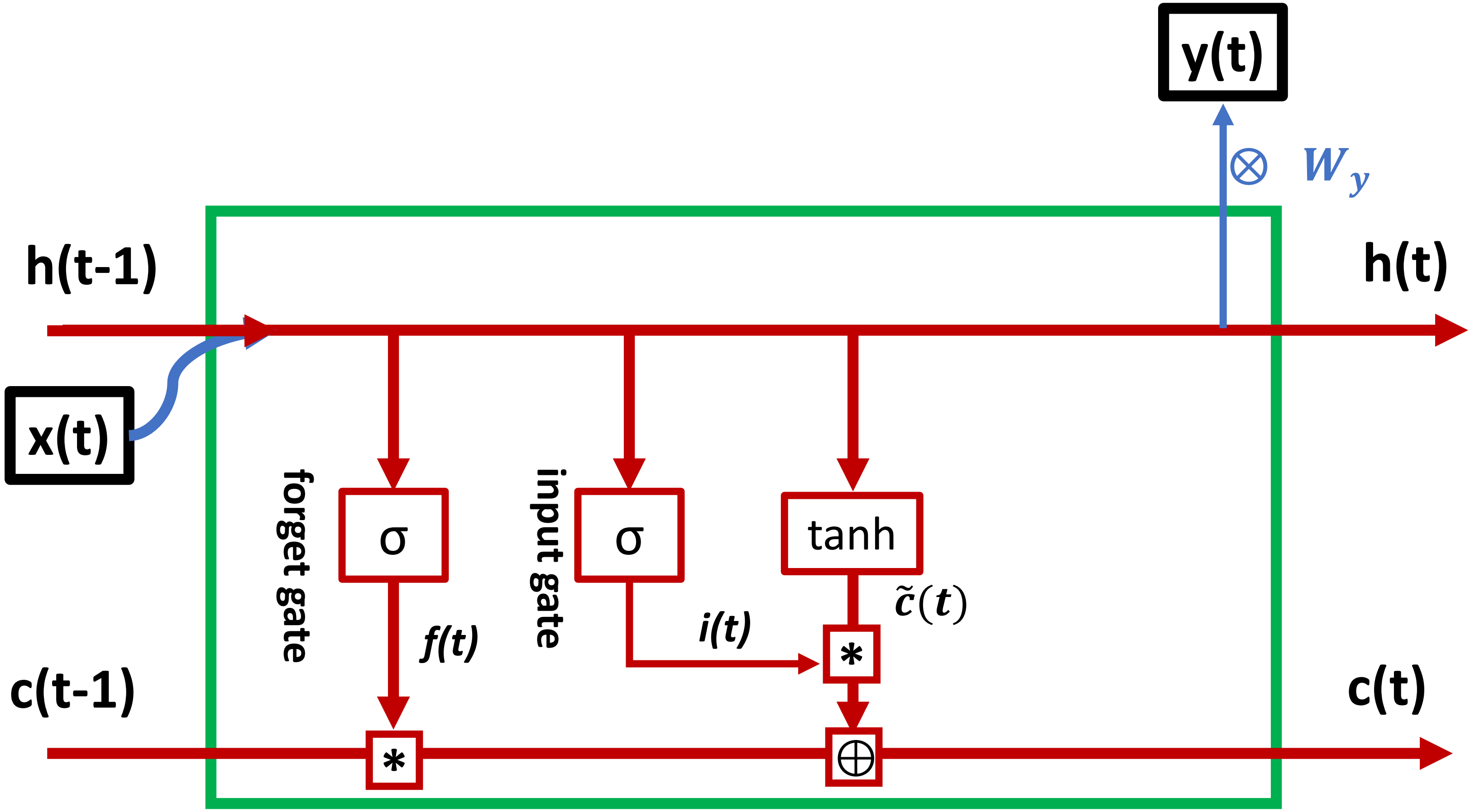
* element – wise multiplication



$$f(t) = \sigma\left(U_f \oplus x(t) + W_f \oplus h(t-1) + b_f\right)$$

$*$ element – wise multiplication

$$i(t) = \sigma\left(U_i \oplus x(t) + W_i \oplus h(t-1) + b_i\right) \qquad \tilde{c}(t) = tanh(U_g \oplus x(t) + W_g \oplus h(t-1) + b_g)$$



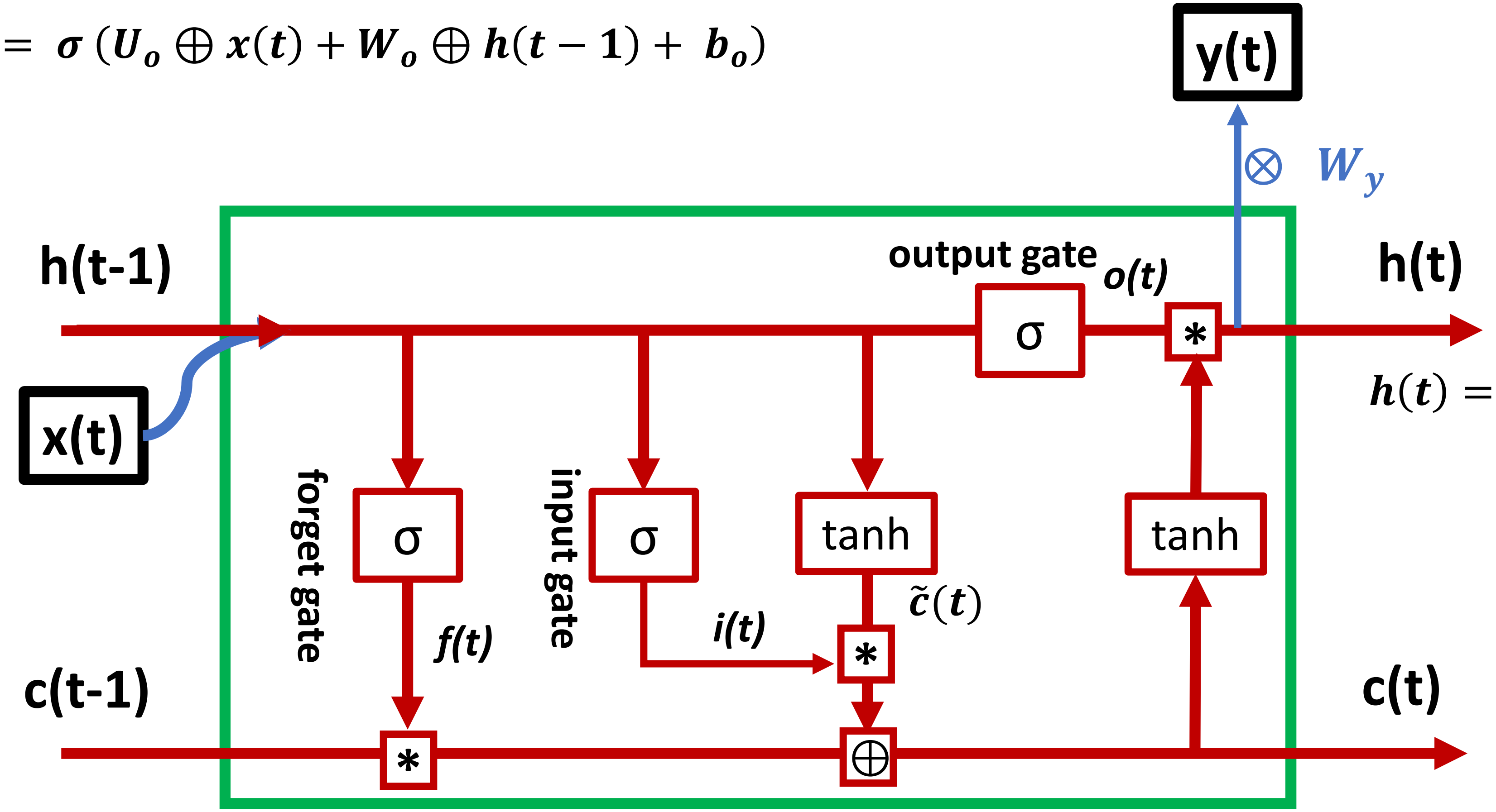
$$c(t) = f(t) * c(t-1) + i(t) * \tilde{c}(t)$$

$*$ element – wise multiplication

$$f(t) = \sigma\left(U_f \oplus x(t) + W_f \oplus h(t-1) + b_f\right)$$

$$i(t) = \sigma\left(U_i \oplus x(t) + W_i \oplus h(t-1) + b_i\right) \qquad \tilde{c}(t) = tanh(U_g \oplus x(t) + W_g \oplus h(t-1) + b_g)$$

$$o(t) = \sigma\left(U_o \oplus x(t) + W_o \oplus h(t-1) + b_o\right)$$



$$h(t) = tanh(c(t)) * o(t)$$

$$c(t) = f(t) * c(t-1) + i(t) * \tilde{c}(t)$$

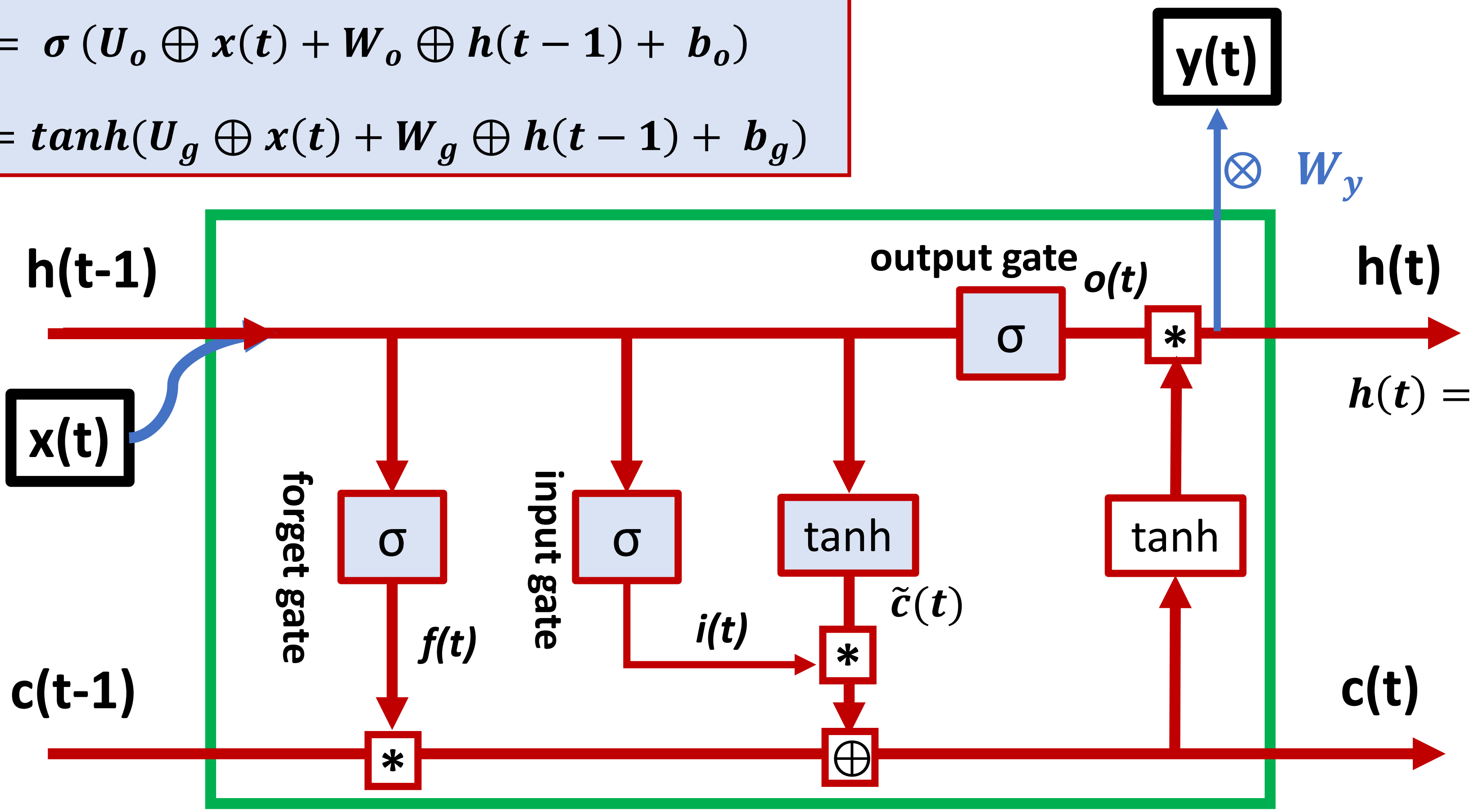
$$f(t) = \sigma\left(U_f \oplus x(t) + W_f \oplus h(t-1) + b_f\right)$$

$$i(t) = \sigma\left(U_i \oplus x(t) + W_i \oplus h(t-1) + b_i\right)$$

$$o(t) = \sigma\left(U_o \oplus x(t) + W_o \oplus h(t-1) + b_o\right)$$

$$\tilde{c}(t) = tanh\left(U_g \oplus x(t) + W_g \oplus h(t-1) + b_g\right)$$

$*$ element – wise multiplication



$$h(t) = tanh(c(t)) * o(t)$$

$$c(t) = f(t) * c(t-1) + i(t) * \tilde{c}(t)$$

$$f(t) = \sigma\left(U_f \oplus x(t) + W_f \oplus h(t-1) + b_f\right)$$

$$i(t) = \sigma\left(U_i \oplus x(t) + W_i \oplus h(t-1) + b_i\right)$$

$$o(t) = \sigma\left(U_o \oplus x(t) + W_o \oplus h(t-1) + b_o\right)$$

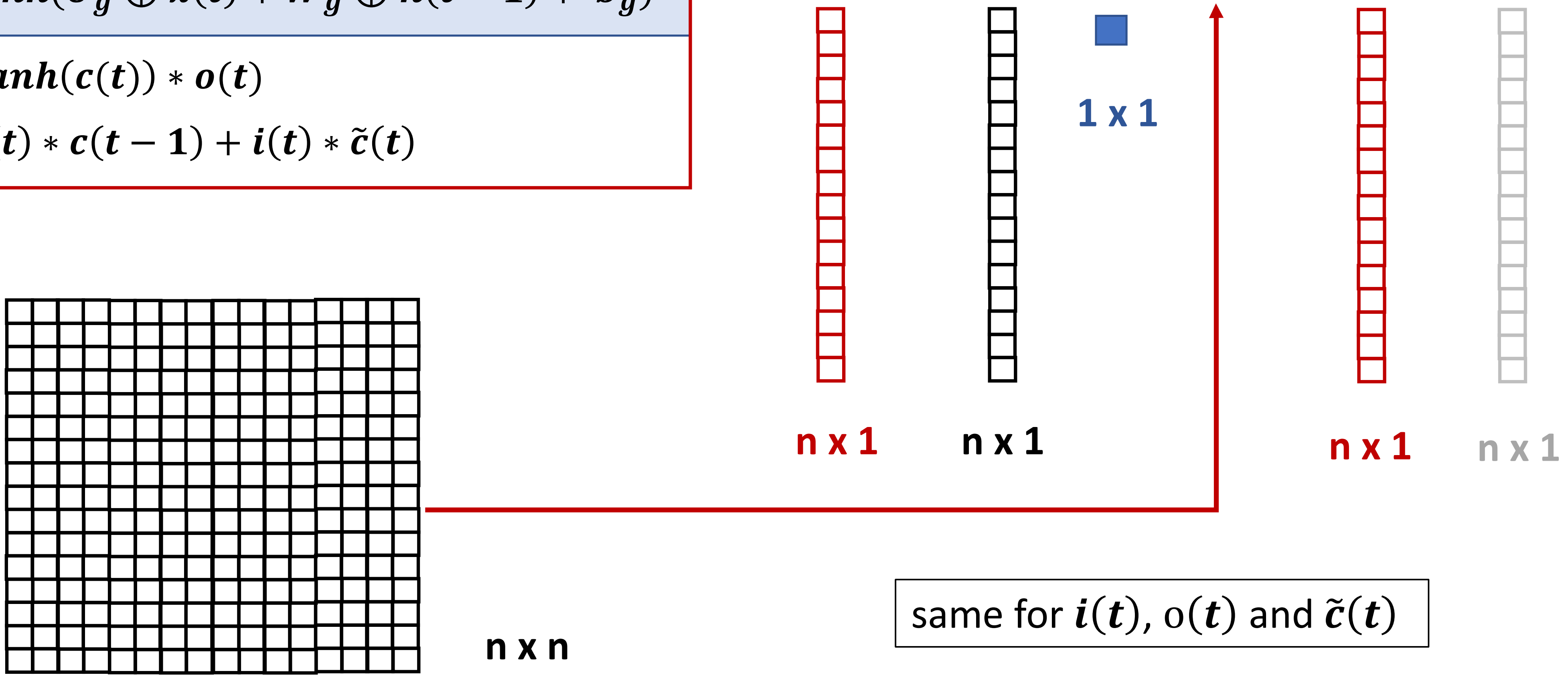
$$\tilde{c}(t) = tanh(U_g \oplus x(t) + W_g \oplus h(t-1) + b_g)$$

$$h(t) = tanh(c(t)) * o(t)$$

$$c(t) = f(t) * c(t-1) + i(t) * \tilde{c}(t)$$

* element – wise multiplication

$$f(t) = \sigma\left(U_f \oplus x(t) + \boxed{W_f} \oplus h(t-1) + b_f\right)$$

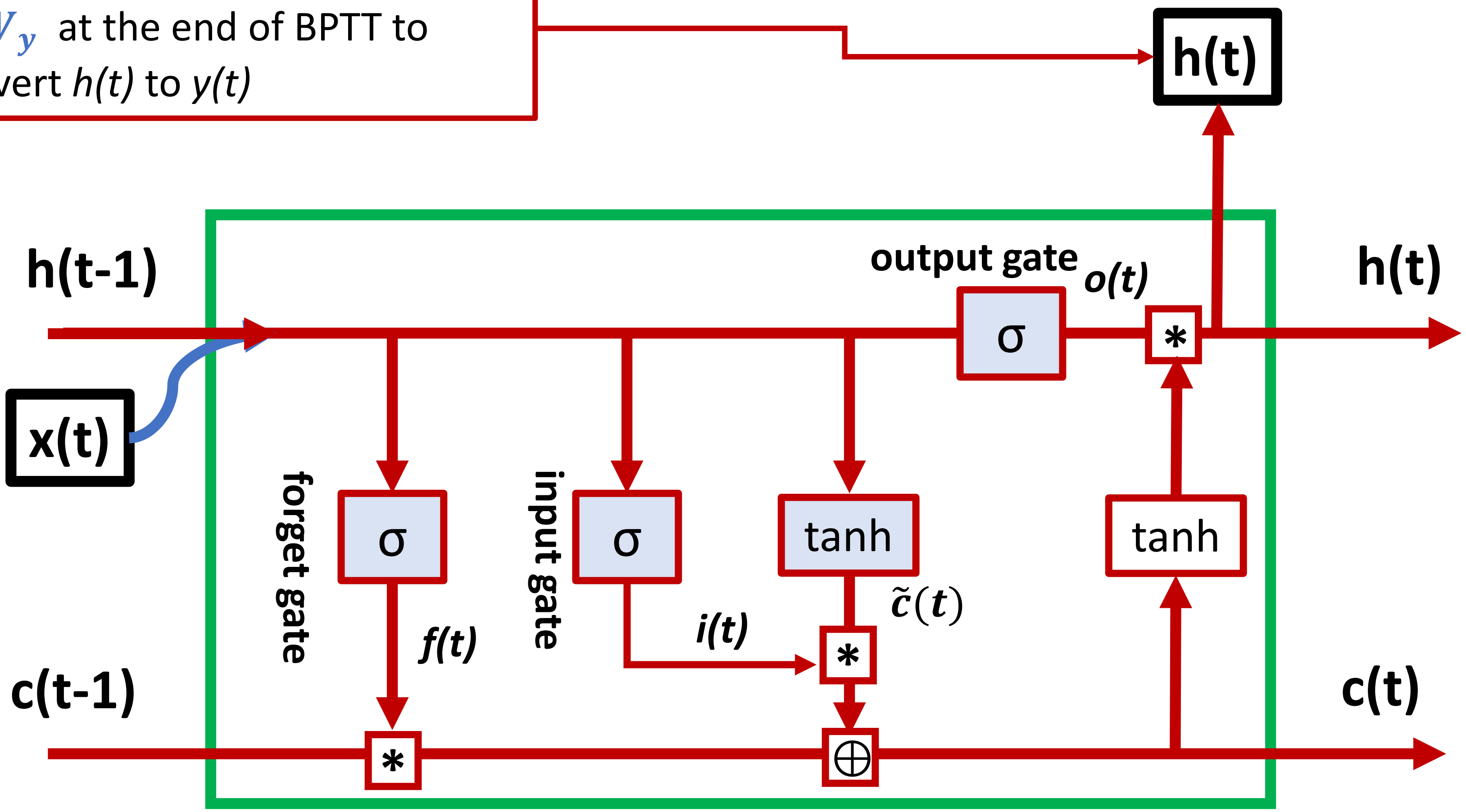


same for $i(t)$, $o(t)$ and $\tilde{c}(t)$

There is one more thing:

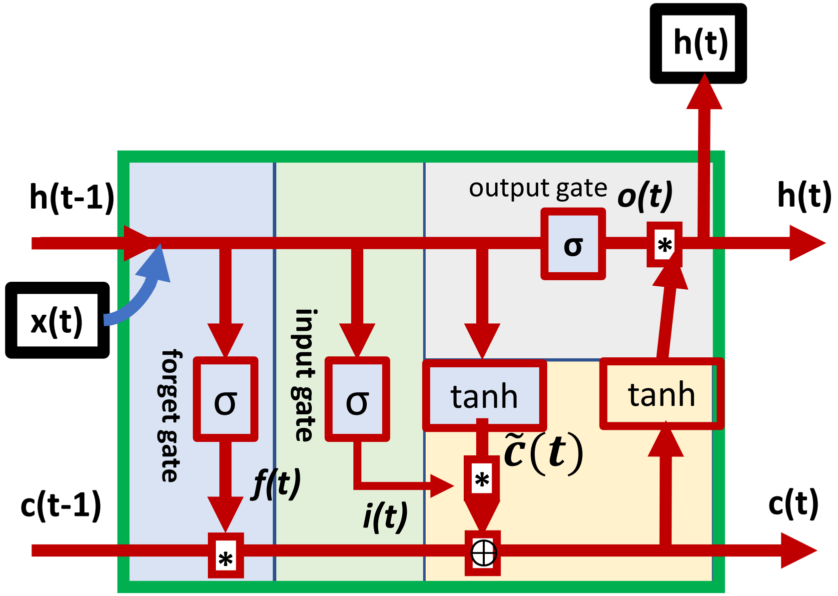
* element – wise multiplication

we will add a **dense layer** instead of W_y at the end of BPTT to convert *h(t)* to *y(t)*



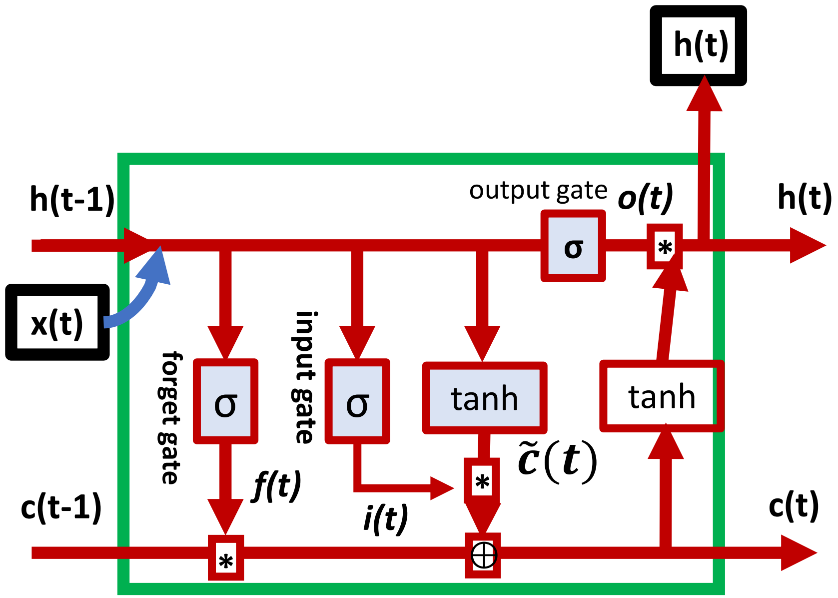
initializing learnables

```
class LSTM():

    def __init__(self, n_neurons):

        self.n_neurons = n_neurons

        self.Uf        = 0.1*np.random.randn(n_neurons, 1)
        self.bf        = 0.1*np.random.randn(n_neurons, 1)
        self.Wf        = 0.1*np.random.randn(n_neurons, n_neurons)

        self.Ui        = 0.1*np.random.randn(n_neurons, 1)
        self.bi        = 0.1*np.random.randn(n_neurons, 1)
        self.Wi        = 0.1*np.random.randn(n_neurons, n_neurons)

        self.Uo        = 0.1*np.random.randn(n_neurons, 1)
        self.bo        = 0.1*np.random.randn(n_neurons, 1)
        self.Wo        = 0.1*np.random.randn(n_neurons, n_neurons)

        self.Ug        = 0.1*np.random.randn(n_neurons, 1)
        self.bg        = 0.1*np.random.randn(n_neurons, 1)
        self.Wg        = 0.1*np.random.randn(n_neurons, n_neurons)
```



```
class LSTM():

        def __init__(self, n_neurons):
...
        def forward(self, X_t):

                T = max(X_t.shape)

                self.T   = T
                self.X_t = X_t


                n_neurons = self.n_neurons

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

                self.F          = [np.zeros((n_neurons,1)) for t in range(T)]
                self.O          = [np.zeros((n_neurons,1)) for t in range(T)]
                self.I          = [np.zeros((n_neurons,1)) for t in range(T)]
```

keeping track of the input/output of the gates



initializing dweights

```
def forward(self, X_t):
...
        self.I          = [np.zeros((n_neurons,1)) for t in range(T)]
```

forget gate

```
        self.dUf = 0.1*np.random.randn(n_neurons, 1)
        self.dbf = 0.1*np.random.randn(n_neurons, 1)
        self.dWf = 0.1*np.random.randn(n_neurons, n_neurons)
```

input gate

```
        self.dUi = 0.1*np.random.randn(n_neurons, 1)
        self.dbi = 0.1*np.random.randn(n_neurons, 1)
        self.dWi = 0.1*np.random.randn(n_neurons, n_neurons)
```

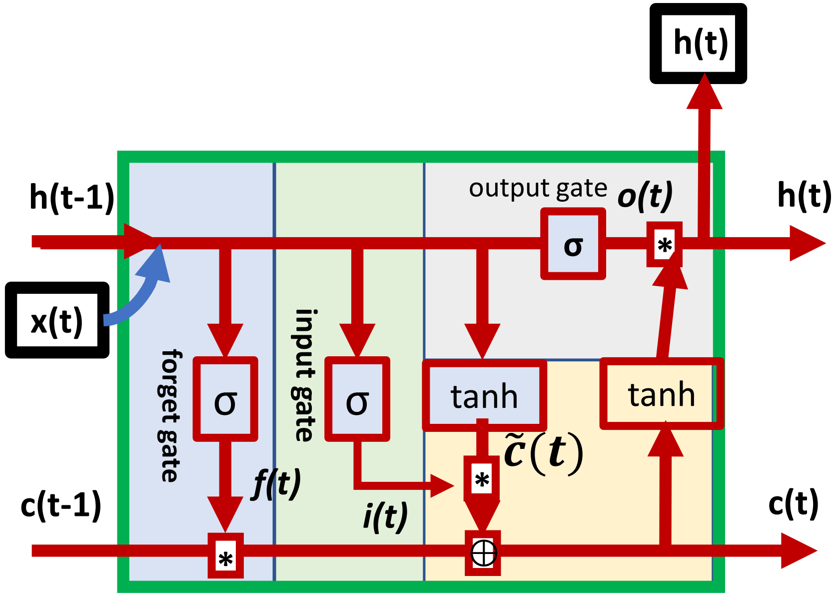
output gate

```
        self.dUo = 0.1*np.random.randn(n_neurons, 1)
        self.dbo = 0.1*np.random.randn(n_neurons, 1)
        self.dWo = 0.1*np.random.randn(n_neurons, n_neurons)
```

```
        self.dUg = 0.1*np.random.randn(n_neurons, 1)
        self.dbg = 0.1*np.random.randn(n_neurons, 1)
        self.dWg = 0.1*np.random.randn(n_neurons, n_neurons)
```

instances of activation functions for BPTT

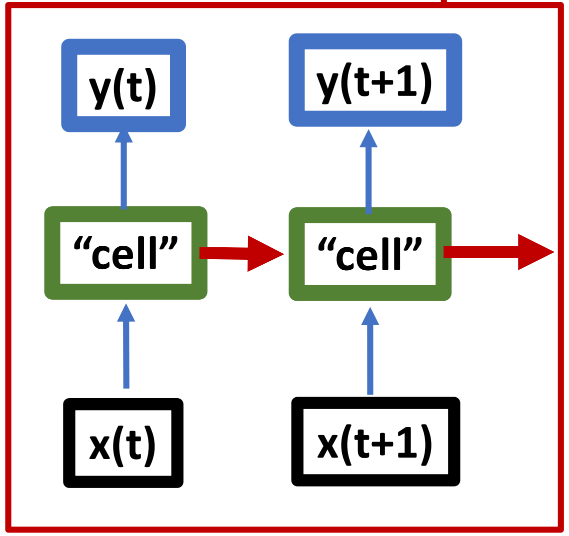
```
def forward(self, X_t):
...
        self.dWg = 0.1*np.random.randn(n_neurons, n_neurons)

        Sigmf    = [Sigmoid() for i in range(T)]
        Sigmi    = [Sigmoid() for i in range(T)]
        Sigmo    = [Sigmoid() for i in range(T)]

        Tanh1    = [Tanh()    for i in range(T)]
        Tanh2    = [Tanh()    for i in range(T)]
```



calling the LSTM cell

```
def forward(self, X_t):
...
        Tanh2     = [Tanh()           for i in range(T)]

        ht         = self.H[0]
        ct         = self.C[0]

        [H, C, Sigmf, Sigmi, Sigmo, Tanh1, Tanh2, F, O, I, C_tilde]\
                = self.LSTMCell(X_t, ht, ct, Sigmf, Sigmi, Sigmo, Tanh1, Tanh2,\
                    self.H, self.C, self.F, self.O, self.I, self.C_tilde)

        self.F       = F
        self.O       = O
        self.I       = I
        self.C_tilde = C_tilde

        self.H       = H
        self.C       = C

        self.Sigmf   = Sigmf
        self.Sigmi   = Sigmi
        self.Sigmo   = Sigmo
        self.Tanh1   = Tanh1
        self.Tanh2   = Tanh2
```

we want to keep track of the states and gates



```
def LSTMCell(self, X_t, ht, ct, Sigmf, Sigmi, Sigmo, Tanh1, Tanh2,\
                                            H, C, F, O, I, C_tilde):

    for t, xt in enumerate(X_t):

        xt = xt.reshape(1,1)
```

forget gate

```
        outf = np.dot(self.Uf, xt) + np.dot(self.Wf, ht) + self.bf
        Sigmf[t].forward(outf)
        ft   = Sigmf[t].output
```

$$f(t) = \sigma\left(U_f \oplus x(t) + W_f \oplus h(t-1) + b_f\right)$$

input gate

```
        outi = np.dot(self.Ui, xt) + np.dot(self.Wi, ht) + self.bi
        Sigmi[t].forward(outi)
        it   = Sigmi[t].output
```

$$i(t) = \sigma\left(U_i \oplus x(t) + W_i \oplus h(t-1) + b_i\right)$$

output gate

```
        outo = np.dot(self.Uo, xt) + np.dot(self.Wo, ht) + self.bo
        Sigmo[t].forward(outo)
        ot   = Sigmo[t].output
```

$$o(t) = \sigma\left(U_o \oplus x(t) + W_o \oplus h(t-1) + b_o\right)$$

```
def LSTMCell(self, X_t, ht, ct, Sigmf, Sigmi, Sigmo, Tanh1, Tanh2,\
                                                   H, C, F, O, I, C_tilde):

    for t, xt in enumerate(X_t):
    ...
            ot    = Sigmo[t].output

            outct_tilde = np.dot(self.Ug, xt) + np.dot(self.Wg, ht) + self.bg
            Tanh1[t].forward(outct_tilde)
            ct_tilde = Tanh1[t].output
```

$$\tilde{c}(t) = tanh(U_g \oplus x(t) + W_g \oplus h(t-1) + b_g)$$

```
            ct = np.multiply(ft, ct) + np.multiply(it, ct_tilde)
```

$$c(t) = f(t) * c(t-1) + i(t) * \tilde{c}(t)$$

```
            Tanh2[t].forward(ct)
            ht = np.multiply(Tanh2[t].output,ot)
```

$$h(t) = tanh(c(t)) * o(t)$$

```
def LSTMCell(self, X_t, ht, ct, Sigmf, Sigmi, Sigmo, Tanh1, Tanh2,\
                                                   H, C, F, O, I, C_tilde):

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

        Tanh2[t].forward(ct)
        ht = np.multiply(Tanh2[t].output,ot)

        H[t+1]       = ht
        C[t+1]       = ct
        C_tilde[t] = ct_tilde

        F[t]         = ft
        O[t]         = ot
        I[t]         = it

    return(H, C, Sigmf, Sigmi, Sigmo, Tanh1, Tanh2, F, O, I, C_tilde)
```

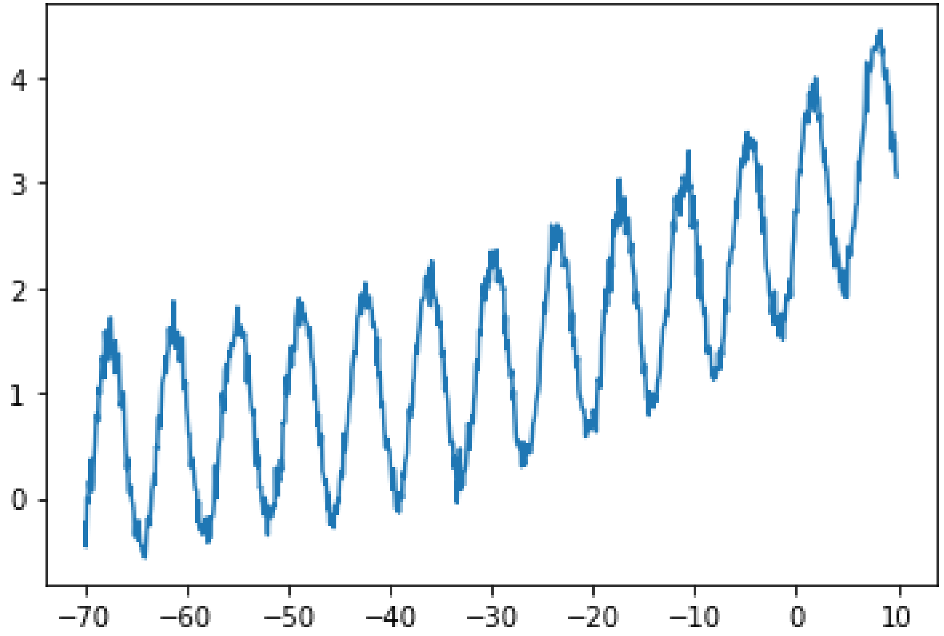
we want to keep track of the states and gates

let us test the code

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

X_t = np.arange(-70,10,0.1)
X_t = X_t.reshape(len(X_t),1)
Y_t = np.sin(X_t) + 0.1*np.random.randn(len(X_t),1) + np.exp((0.5*X_t + 20)*0.05)

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

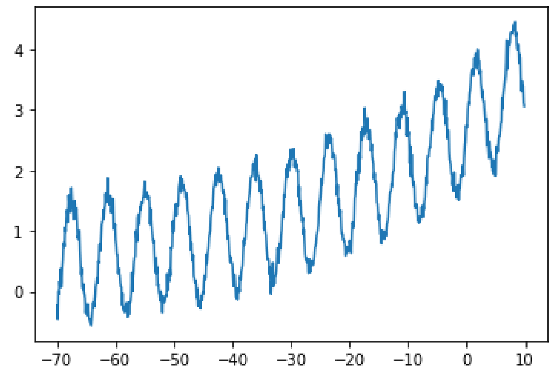
aim:

- prediction Y_t(t) → Y_t(t + dt)



let us test the code

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

X_t = np.arange(-70,10,0.1)
X_t = X_t.reshape(len(X_t),1)
Y_t = np.sin(X_t) + 0.1*np.random.randn(len(X_t),1) + np.exp((0.5*X_t + 20)*0.05)

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

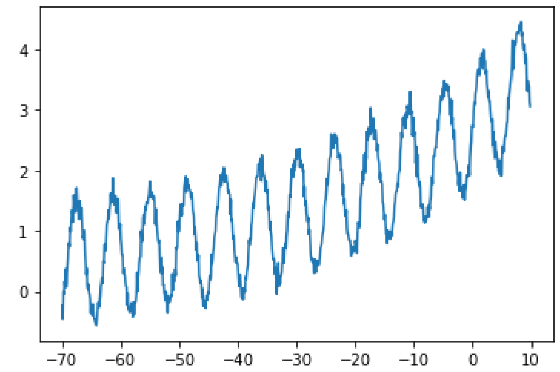


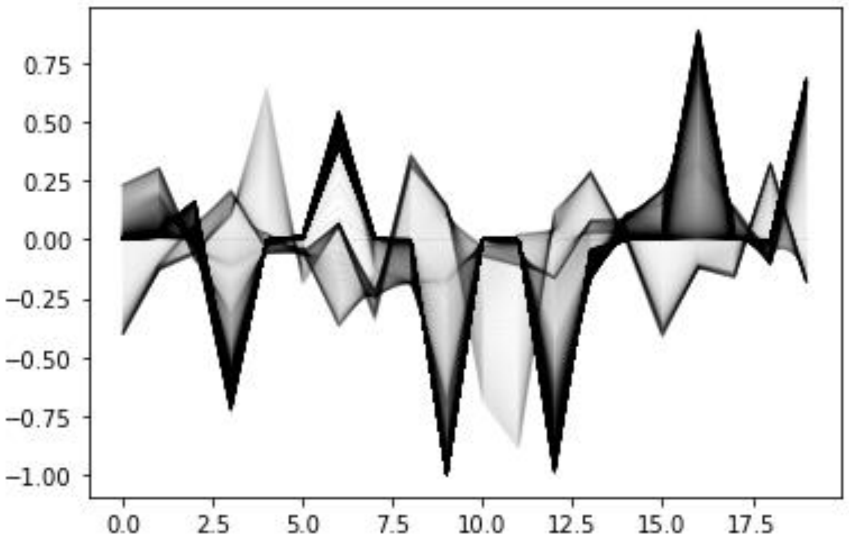

```
from LSTM import *

lstm = LSTM(n_neurons = 200)
lstm.forward(X_t)

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

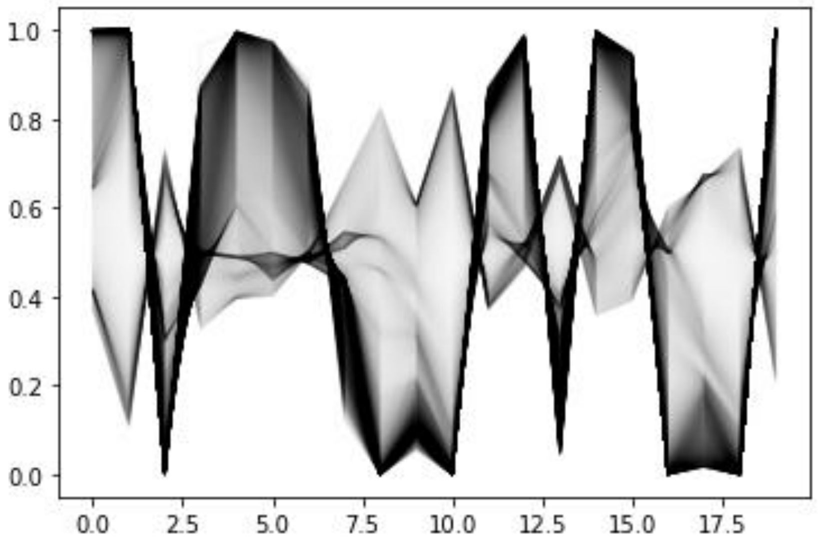
do the same for F, I and O

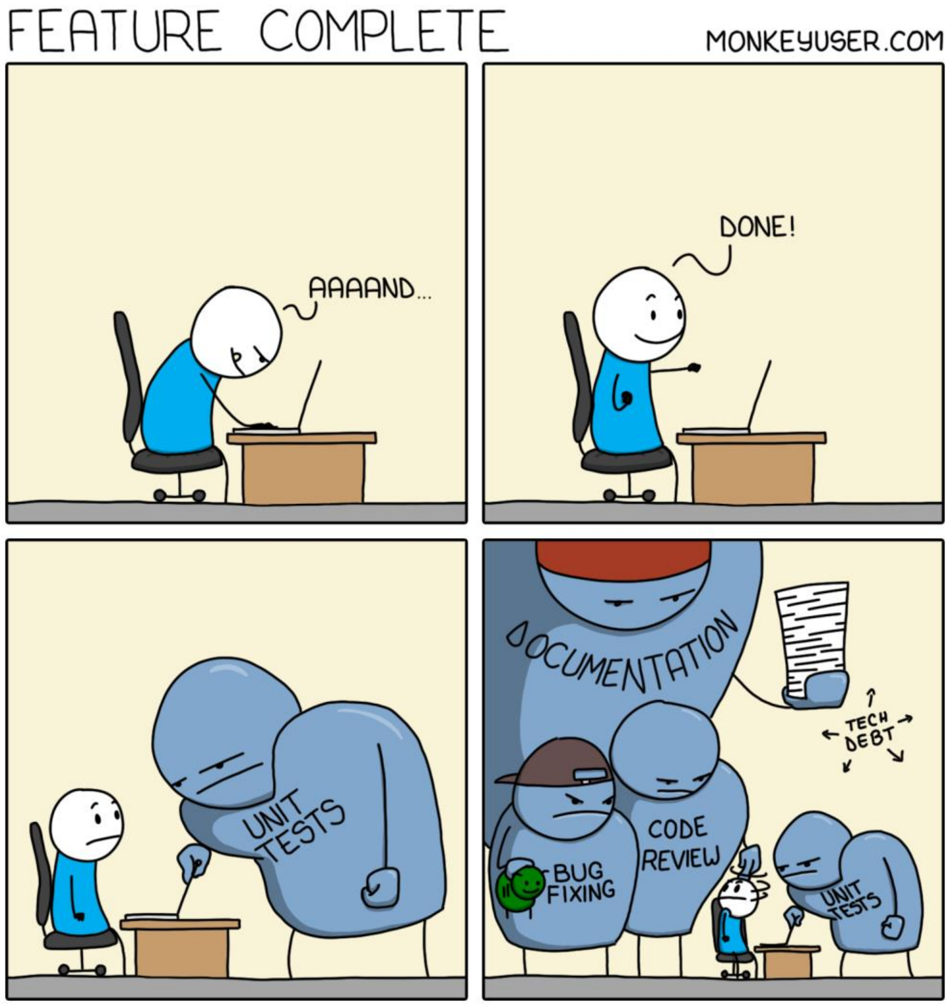
let us test the code

```
from LSTM import *

lstm = LSTM(n_neurons = 200)
lstm.forward(X_t)

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





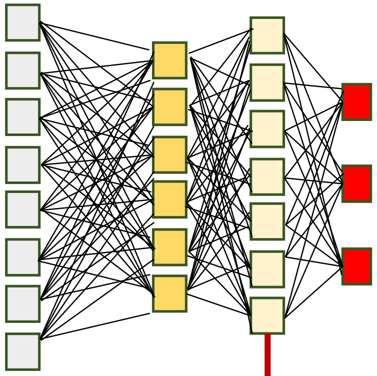




outline:

- the idea
- from a classical RNN cell to a LSTM
- **BackPropagation Through Time**
- full backpropagtion
- modifying the SGD optimizer
- running and testing the package

dense

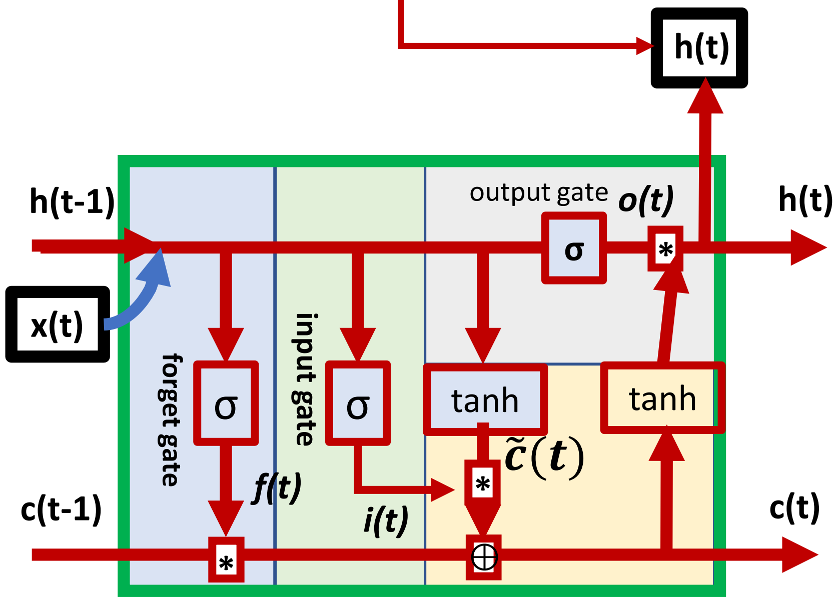


see lecture
ANN and CNN from Scratch

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

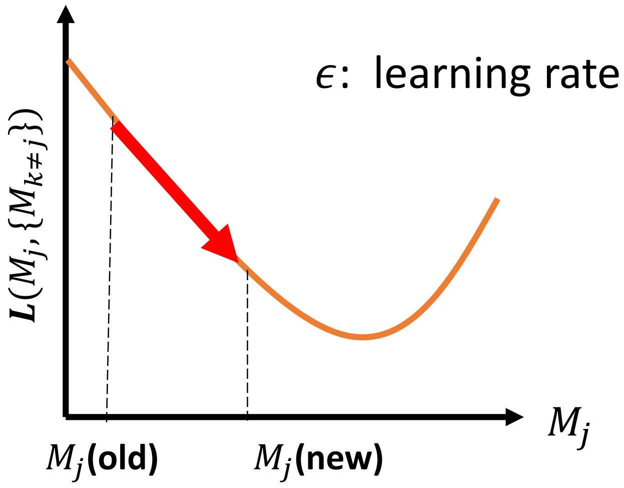
$$dY = dY(\boldsymbol{U_f}, \boldsymbol{W_f}, \boldsymbol{b_f}, \boldsymbol{U_i}, \boldsymbol{W_i}, \boldsymbol{b_i}, \boldsymbol{U_o}, \boldsymbol{W_o}, \boldsymbol{b_o}, \boldsymbol{U_g}, \boldsymbol{W_g}, \boldsymbol{b_g})$$

`dense.dinputs`



$$\Delta M_j = -\,\epsilon\,\frac{d\,L(M_j, \{M_{k\neq j}\})}{dM_j}$$

ϵ: learning rate



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, U_o, W_o, b_o, U_g, W_g, b_g)$$

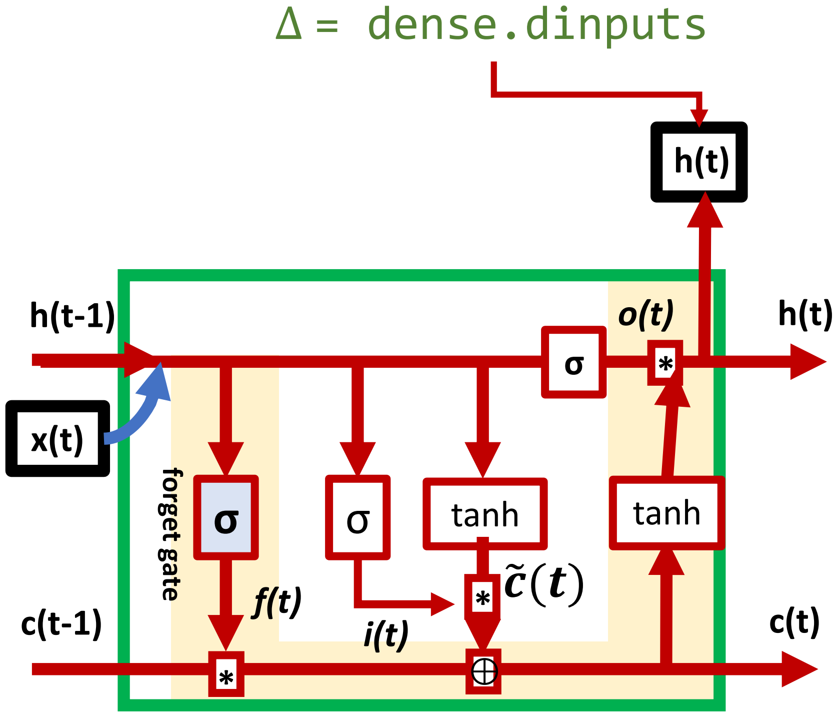
$$h(t) = \boxed{tanh(c(t)) * o(t)}$$

$$c(t) = \boxed{f(t) * c(t-1)} + i(t) * \tilde{c}(t)$$

$$f(t) = \boxed{\sigma\,(U_f \oplus x(t)} + W_f \oplus h(t-1) + b_f)$$

$$\frac{dh(t)}{dU_f} = \frac{\partial h(t)}{\partial tanh}\,\frac{\partial tanh}{\partial c(t)} * o(t)\quad \underbrace{\frac{\partial c(t)}{\partial f(t)}}_{= c(t-1)}\quad \frac{\partial f(t)}{\partial \sigma}\,\underbrace{\frac{\partial \sigma}{\partial U_f}}_{= x(t)}$$

```
dht      = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdft   = np.multiply(dhtdtanh,C[t-1])

Sigmf[t].backward(dctdft)
dsigmf = Sigmf[t].dinputs

dsigmfdUf = np.dot(dsigmf,xt)
dUf += dsigmfdUf
```



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, U_o, W_o, b_o, U_g, W_g, b_g)$$

$$h(t) = \boxed{tanh(c(t)) * o(t)}$$

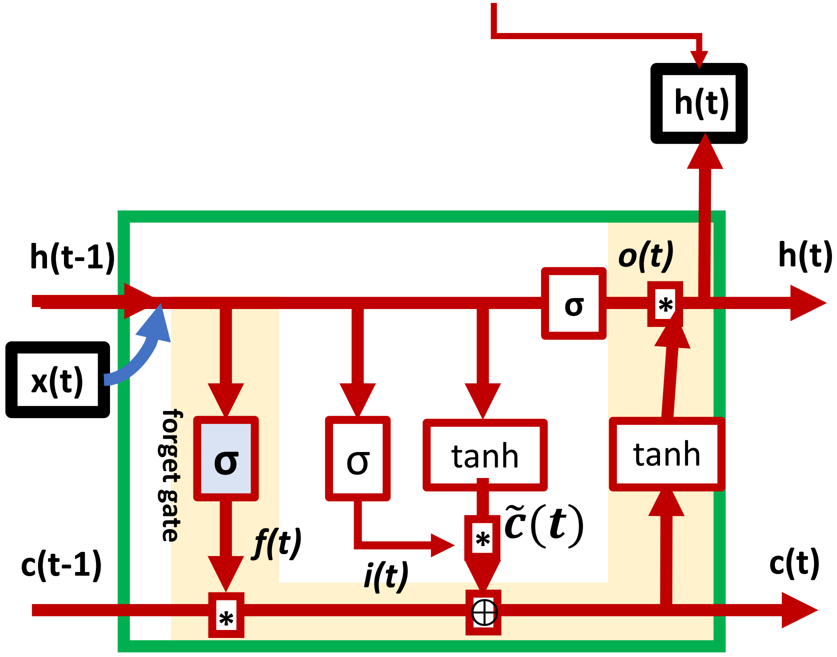
$$c(t) = \boxed{f(t) * c(t-1)} + i(t) * \tilde{c}(t)$$

$$f(t) = \sigma\left(U_f \oplus x(t) + \boxed{W_f \oplus h(t-1)} + b_f\right)$$

$$\frac{dh(t)}{dW_f} = \frac{\partial h(t)}{\partial tanh}\frac{\partial tanh}{\partial c(t)} * o(t)\;\underbrace{\frac{\partial c(t)}{\partial f(t)}}_{= c(t-1)}\;\frac{\partial f(t)}{\partial \sigma}\underbrace{\frac{\partial \sigma}{\partial W_f}}_{= h(t-1)}$$

```
dht      = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdft   = np.multiply(dhtdtanh,C[t-1])

Sigmf[t].backward(dctdft)
dsigmf = Sigmf[t].dinputs

dsigmfdWf = np.dot(dsigmf,H[t-1].T)
dWf += dsigmfdWf
```

$\Delta =$ dense.dinputs



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, U_o, W_o, b_o, U_g, W_g, b_g)$$

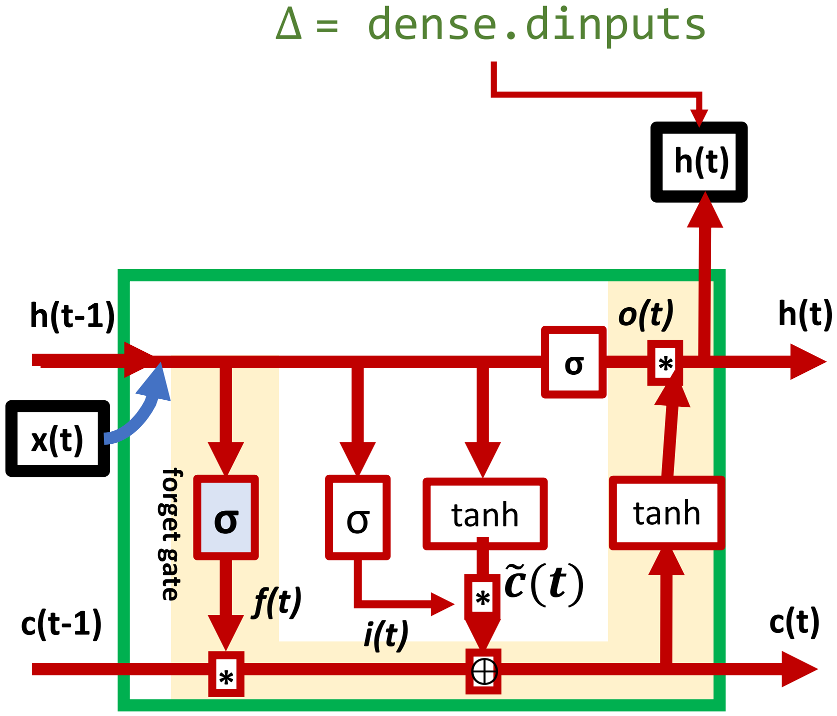
$$h(t) = \boxed{tanh(c(t)) * o(t)}$$

$$c(t) = \boxed{f(t) * c(t-1)} + i(t) * \tilde{c}(t)$$

$$f(t) = \sigma\left(U_f \oplus x(t) + W_f \oplus h(t-1) + \boxed{b_f}\right)$$

$$\frac{dh(t)}{db_f} = \frac{\partial h(t)}{\partial tanh}\frac{\partial tanh}{\partial c(t)} * o(t)\ \underbrace{\frac{\partial c(t)}{\partial f(t)}}_{= c(t-1)}\ \frac{\partial f(t)}{\partial \sigma}\ \underbrace{\frac{\partial \sigma}{\partial b_f}}_{= 1}$$

```
dht     = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdft   = np.multiply(dhtdtanh,C[t-1])

Sigmf[t].backward(dctdft)
dsigmf = Sigmf[t].dinputs

dbf += dsigmf
```



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, U_o, W_o, b_o, U_g, W_g, b_g)$$

$$h(t) = \boxed{\tanh(c(t)) * o(t)}$$

$$c(t) = f(t) * c(t-1) + \boxed{i(t) * \tilde{c}(t)}$$

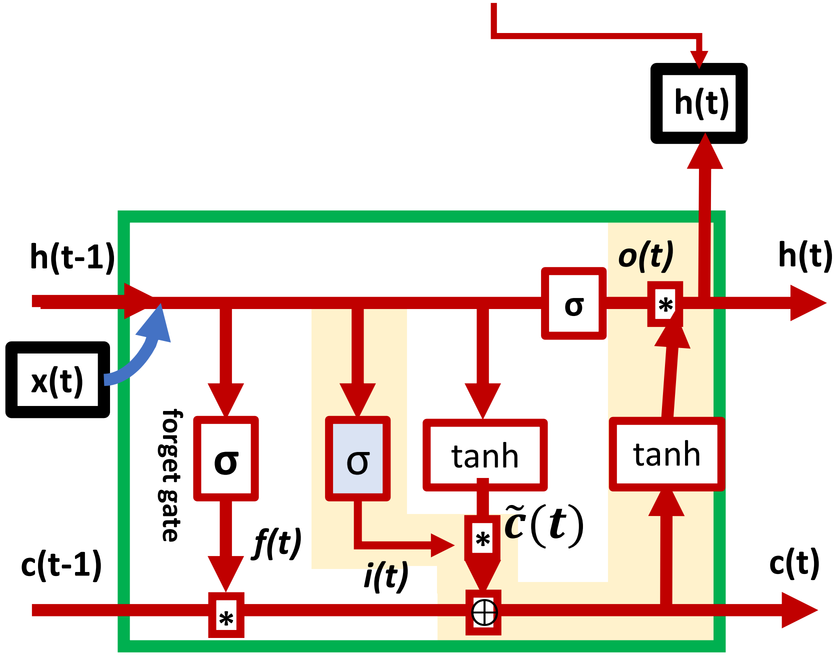
$$i(t) = \sigma\,\left(\boxed{U_i \oplus x(t)} + W_i \oplus h(t-1) + b_i\right)$$

$$\frac{dh(t)}{dU_i} = \frac{\partial h(t)}{\partial tanh}\frac{\partial tanh}{\partial c(t)} * o(t)\ \frac{\partial c(t)}{\partial i(t)} * \tilde{c}(t)\ \frac{\partial i(t)}{\partial \sigma}\ \frac{\partial \sigma}{\partial U_i}$$

$$= x(t)$$

```
dht      = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdit   = np.multiply(dhtdtanh,C_tilde[t])

Sigmi[t].backward(dctdit)
dsigmi = Sigmi[t].dinputs

dsigmidUi = np.dot(dsigmi,xt)
dUi += dsigmidUi
```

$\Delta =$ dense.dinputs



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, \boldsymbol{W_i}, b_i, U_o, W_o, b_o, U_g, W_g, b_g)$$

$$\boldsymbol{h(t)} = \boxed{\boldsymbol{tanh(c(t)) * o(t)}}$$

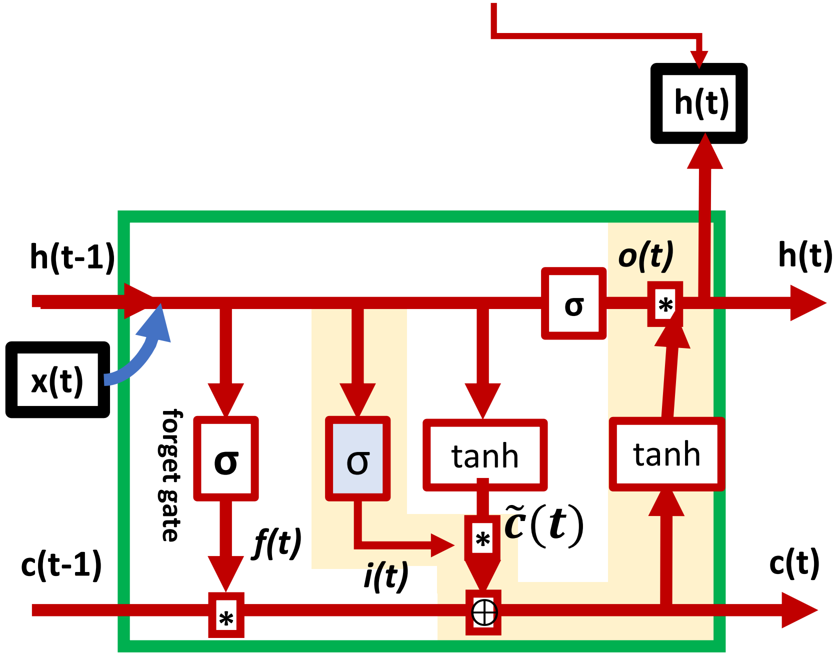
$$\mathbf{c}(\boldsymbol{t}) = \boldsymbol{f(t) * c(t-1)} + \boxed{\boldsymbol{i(t) * \tilde{c}(t)}}$$

$$\boldsymbol{i(t)} = \boldsymbol{\sigma}\,(\boldsymbol{U_i} \oplus \boldsymbol{x(t)} + \boxed{\boldsymbol{W_i} \oplus \boldsymbol{h(t-1)}} + \boldsymbol{b_i})$$

$$\frac{\boldsymbol{dh(t)}}{\boldsymbol{dW_i}} = \frac{\boldsymbol{\partial h(t)}}{\boldsymbol{\partial tanh}}\,\frac{\boldsymbol{\partial tanh}}{\boldsymbol{\partial c(t)}} * \boldsymbol{o(t)}\;\frac{\boldsymbol{\partial c(t)}}{\boldsymbol{\partial i(t)}} * \boldsymbol{\tilde{c}(t)}\;\frac{\boldsymbol{\partial i(t)}}{\boldsymbol{\partial \sigma}}\;\underbrace{\frac{\boldsymbol{\partial \sigma}}{\boldsymbol{\partial W_i}}}_{=\boldsymbol{h(t-1)}}$$

```
dht      = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdit   = np.multiply(dhtdtanh,C_tilde[t])

Sigmi[t].backward(dctdit)
dsigmi = Sigmi[t].dinputs

dsigmidWi = np.dot(dsigmi,H[t-1].T)
dWi += dsigmidWi
```

$\Delta =$ dense.dinputs



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, \mathbf{b_i}, U_o, W_o, b_o, U_g, W_g, b_g)$$

$$\boldsymbol{h(t)} = \boxed{\boldsymbol{tanh(c(t)) * o(t)}}$$

$$\mathbf{c(t) = f(t) * c(t-1)} + \boxed{\boldsymbol{i(t) * \tilde{c}(t)}}$$

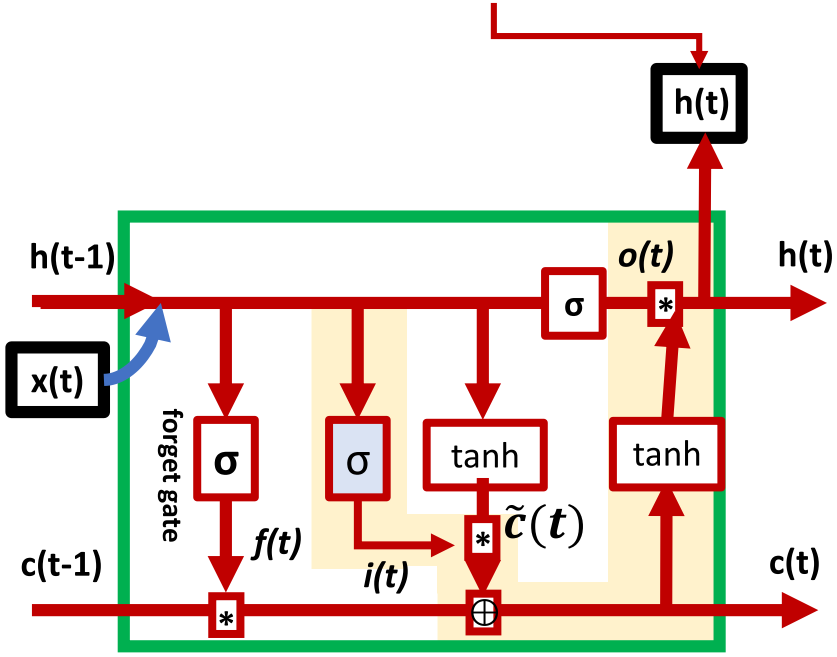
$$\boldsymbol{i(t) = \sigma\,(U_i \oplus x(t) + W_i \oplus h(t-1) +} \boxed{\boldsymbol{b_i}}\boldsymbol{)}$$

$$\frac{\boldsymbol{dh(t)}}{\boldsymbol{db_i}} = \frac{\boldsymbol{\partial h(t)}}{\boldsymbol{\partial tanh}}\frac{\boldsymbol{\partial tanh}}{\boldsymbol{\partial c(t)}} * \boldsymbol{o(t)}\ \frac{\boldsymbol{\partial c(t)}}{\boldsymbol{\partial i(t)}} * \boldsymbol{\tilde{c}(t)}\ \frac{\boldsymbol{\partial i(t)}}{\boldsymbol{\partial \sigma}}\ \frac{\boldsymbol{\partial \sigma}}{\boldsymbol{\partial b_i}}$$

$$= \mathbf{1}$$

```
dht     = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdit   = np.multiply(dhtdtanh,C_tilde[t])

Sigmi[t].backward(dctdit)
dsigmi = Sigmi[t].dinputs

dbi += dsigmi
```

$\Delta =$ dense.dinputs



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, \boldsymbol{U_o}, W_o, b_o, U_g, W_g, b_g)$$

$$\boldsymbol{h(t)} = \boxed{\boldsymbol{tanh(c(t)) * o(t)}}$$

$$\boldsymbol{o(t)} = \boldsymbol{\sigma}\,\boxed{(\boldsymbol{U_o} \oplus \boldsymbol{x(t)}} + \boldsymbol{W_o} \oplus \boldsymbol{h(t-1)} + \boldsymbol{b_o})$$

$$\frac{\boldsymbol{dh(t)}}{\boldsymbol{dU_o}} = \frac{\boldsymbol{\partial h(t)}}{\boldsymbol{\partial o(t)}} * \boldsymbol{tanh(c(t))}\ \frac{\boldsymbol{\partial o(t)}}{\boldsymbol{\partial\sigma}}\ \frac{\boldsymbol{\partial\sigma}}{\boldsymbol{\partial U_o}}$$

$$= \boldsymbol{x(t)}$$

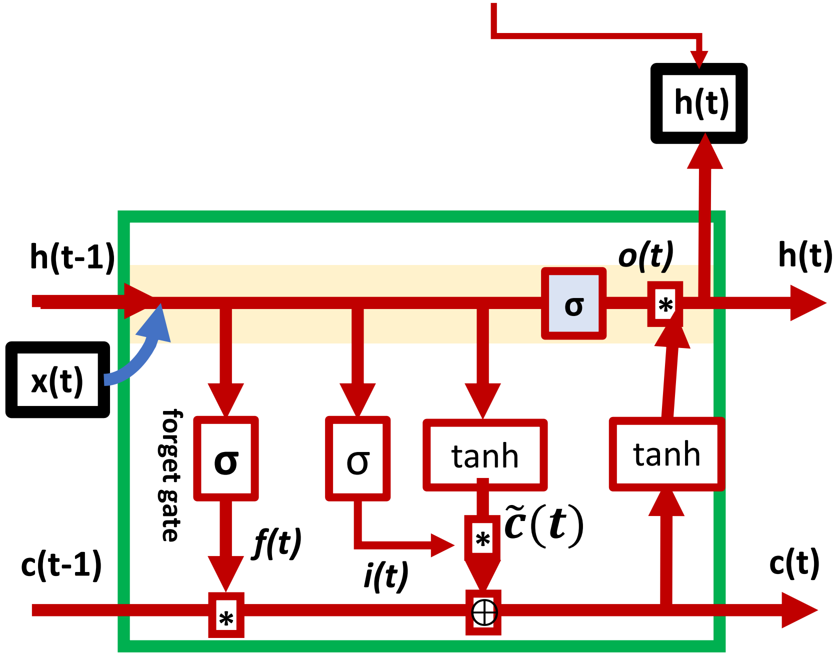
```
dht     = dvalues[-1].reshape(self.n_neurons,1)
```

```
Sigmo[t].backward(np.multiply(dht, Tanh2[t].output))
dsigmo = Sigmo[t].dinputs
```

```
dsigmodUo = np.dot(dsigmo,xt)
```

```
dUo += dsigmodUo
```

$\Delta =$ dense.dinputs



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, U_o, \boldsymbol{W_o}, b_o, U_g, W_g, b_g)$$

$$\boldsymbol{h(t)} = \boxed{\boldsymbol{tanh(c(t)) * o(t)}}$$

$$\boldsymbol{o(t)} = \boldsymbol{\sigma}\,(\boldsymbol{U_o} \oplus \boldsymbol{x(t)} + \boxed{\boldsymbol{W_o} \oplus \boldsymbol{h(t-1)}} + \boldsymbol{b_o})$$

$$\frac{\boldsymbol{dh(t)}}{\boldsymbol{dW_o}} = \frac{\boldsymbol{\partial h(t)}}{\boldsymbol{\partial o(t)}} * \boldsymbol{tanh(c(t))}\, \frac{\boldsymbol{\partial o(t)}}{\boldsymbol{\partial \sigma}} \frac{\boldsymbol{\partial \sigma}}{\boldsymbol{\partial W_o}}$$

$$= \boldsymbol{h(t-1)}$$

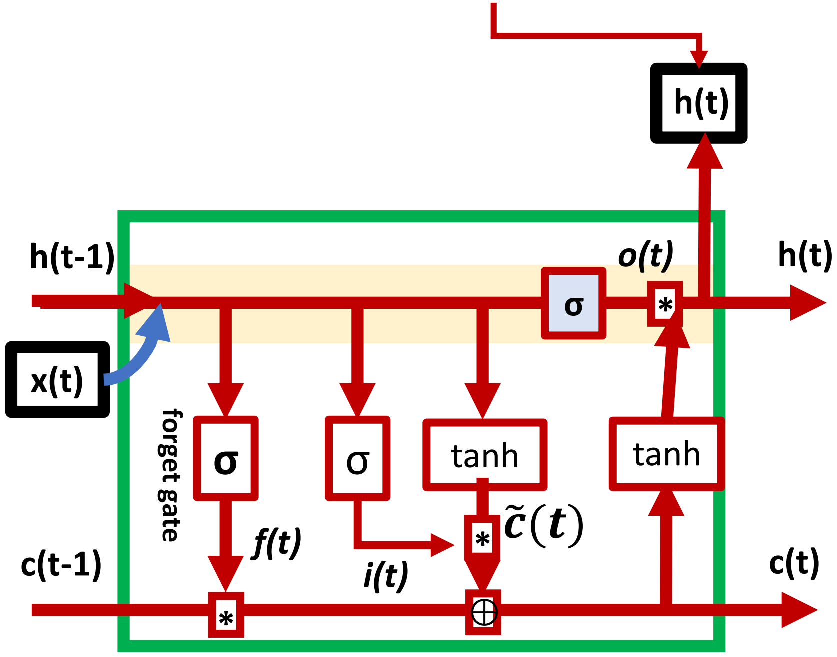
```
dht     = dvalues[-1].reshape(self.n_neurons,1)
```

```
Sigmo[t].backward(np.multiply(dht, Tanh2[t].output))
dsigmo = Sigmo[t].dinputs
```

```
dsigmodWo = np.dot(dsigmo,H[t-1].T)
```

```
dWo += dsigmodWo
```

$\Delta =$ dense.dinputs



$$\Delta = \Delta\left(U_f, W_f, b_f, U_i, W_i, b_i, U_o, W_o, \boldsymbol{b_o}, U_g, W_g, b_g\right)$$

$$\boldsymbol{h(t)} = \boxed{\boldsymbol{tanh(c(t)) * o(t)}}$$

$$\boldsymbol{o(t)} = \boldsymbol{\sigma}\left(\boldsymbol{U_o} \oplus \boldsymbol{x(t)} + \boldsymbol{W_o} \oplus \boldsymbol{h(t-1)} + \boxed{\boldsymbol{b_o}}\right)$$

$$\frac{\boldsymbol{dh(t)}}{\boldsymbol{db_o}} = \frac{\boldsymbol{\partial h(t)}}{\boldsymbol{\partial o(t)}} * \boldsymbol{tanh(c(t))} \frac{\boldsymbol{\partial o(t)}}{\boldsymbol{\partial \sigma}} \frac{\boldsymbol{\partial \sigma}}{\boldsymbol{\partial b_o}}$$

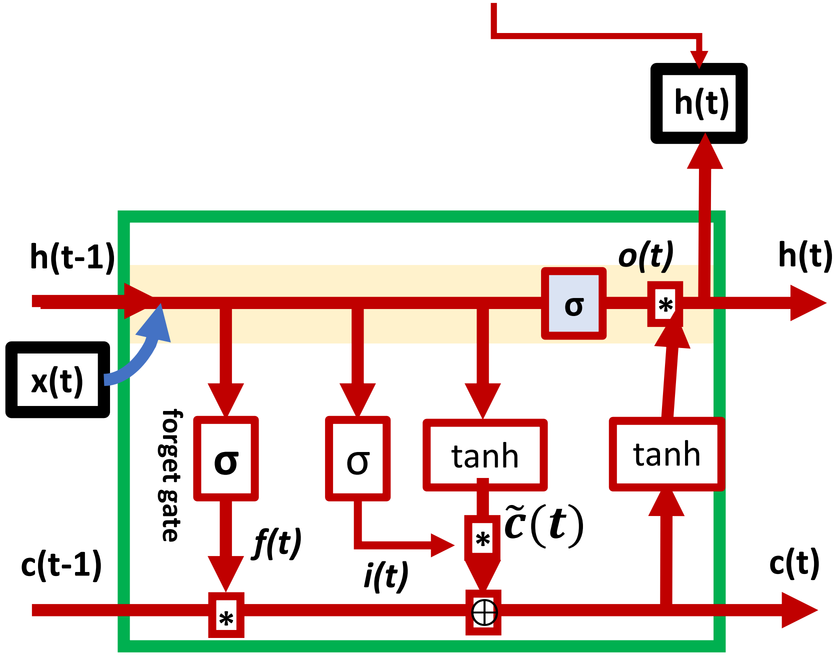
$$= \mathbf{1}$$

```
dht     = dvalues[-1].reshape(self.n_neurons,1)
```

$\Delta =$ dense.dinputs

```
Sigmo[t].backward(np.multiply(dht, Tanh2[t].output))
dsigmo = Sigmo[t].dinputs
```

```
dbo += dsigmo
```



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, U_o, W_o, b_o, U_g, W_g, b_g)$$

$$h(t) = \boxed{tanh(c(t)) * o(t)}$$

$$c(t) = f(t) * c(t-1) + \boxed{i(t) * \tilde{c}(t)}$$

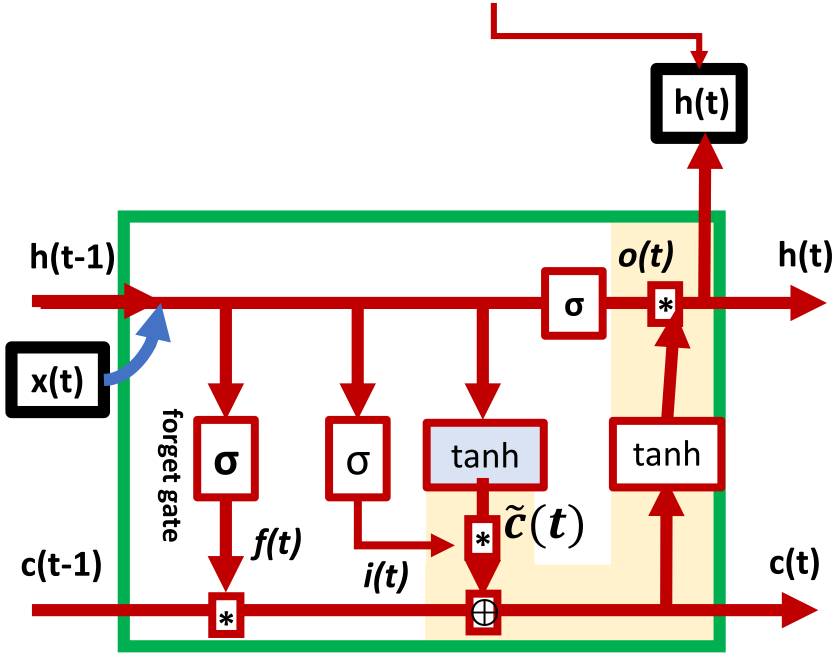
$$\tilde{c}(t) = tanh(\boxed{U_g \oplus x(t)} + W_g \oplus h(t-1) + b_g)$$

$$\frac{dh(t)}{dU_g} = \frac{\partial h(t)}{\partial tanh}\frac{\partial tanh}{\partial c(t)} * o(t)\ \frac{\partial c(t)}{\partial \tilde{c}(t)} * i(t)\ \frac{\partial \tilde{c}(t)}{\partial tanh}\frac{\partial tanh}{\partial U_g}$$

$$= x(t)$$

```
dht      = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdct_tilde = np.multiply(dhtdtanh,I[t])

Tanh1[t].backward(dctdct_tilde)
dtanh1 = Tanh1[t].dinputs

dtanh1dUg = np.dot(dtanh1,xt)
dUg += dtanh1dUg
```

$\Delta =$ dense.dinputs



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, U_o, W_o, b_o, U_g, \boldsymbol{W_g}, b_g)$$

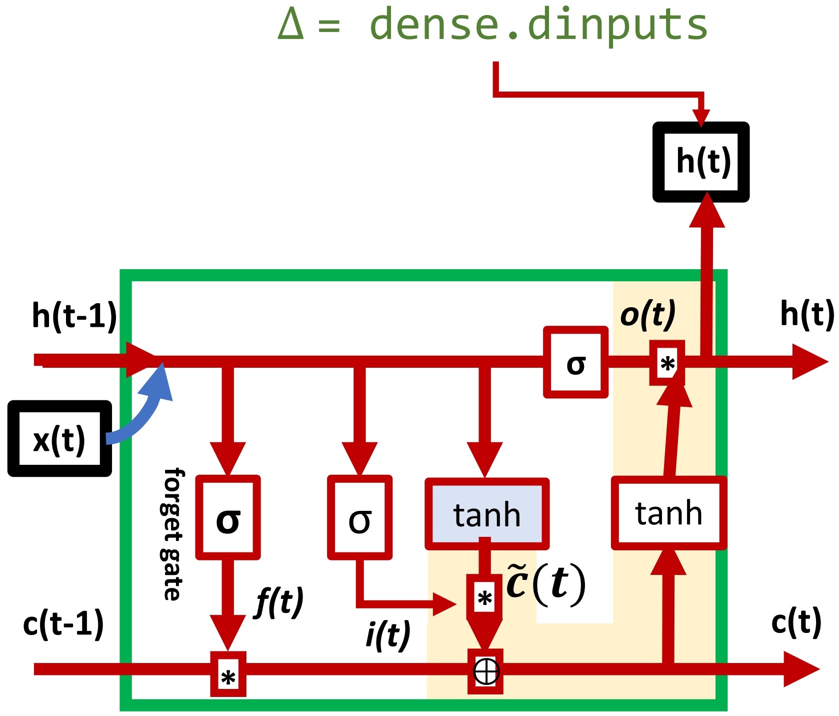
$$\boldsymbol{h(t)} = \boxed{\boldsymbol{tanh(c(t)) * o(t)}}$$

$$\mathbf{c(t)} = \boldsymbol{f(t) * c(t-1)} + \boxed{\boldsymbol{i(t) * \tilde{c}(t)}}$$

$$\boldsymbol{\tilde{c}(t)} = \boldsymbol{tanh(U_g \oplus x(t)} + \boxed{\boldsymbol{W_g \oplus h(t-1)}} + \boldsymbol{b_g)}$$

$$\frac{\boldsymbol{dh(t)}}{\boldsymbol{dW_g}} = \frac{\boldsymbol{\partial h(t)}}{\boldsymbol{\partial tanh}}\,\frac{\boldsymbol{\partial tanh}}{\boldsymbol{\partial c(t)}} * \boldsymbol{o(t)}\;\frac{\boldsymbol{\partial c(t)}}{\boldsymbol{\partial \tilde{c}(t)}} * \boldsymbol{i(t)}\;\frac{\boldsymbol{\partial \tilde{c}(t)}}{\boldsymbol{\partial tanh}}\,\underbrace{\frac{\boldsymbol{\partial tanh}}{\boldsymbol{\partial W_g}}}_{= \boldsymbol{h(t-1)}}$$

```
dht     = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdct_tilde = np.multiply(dhtdtanh,I[t])

Tanh1[t].backward(dctdct_tilde)
dtanh1 = Tanh1[t].dinputs

dtanh1dWg = np.dot(dtanh1,H[t-1].T)
dWg += dtanh1dWg
```



$$\Delta = \Delta\,(U_f, W_f, b_f, U_i, W_i, b_i, U_o, W_o, b_o, U_g, W_g, \boldsymbol{b_g})$$

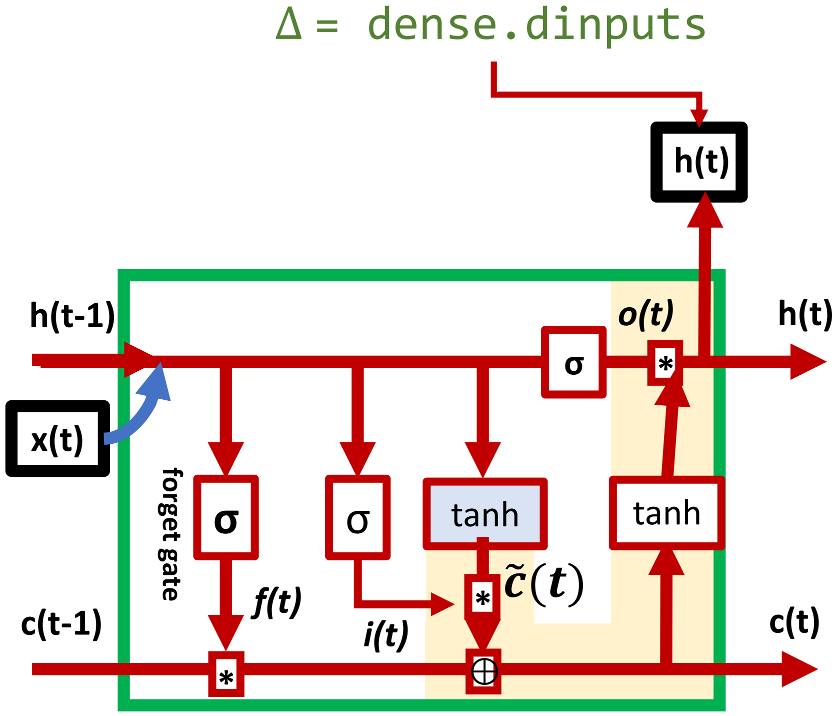
$$\boldsymbol{h(t)} = \boxed{\boldsymbol{tanh(c(t)) * o(t)}}$$

$$\mathbf{c}(\boldsymbol{t}) = \boldsymbol{f(t) * c(t-1)} + \boxed{\boldsymbol{i(t) * \tilde{c}(t)}}$$

$$\boldsymbol{\tilde{c}(t) = tanh(U_g \oplus x(t)} + \boxed{\boldsymbol{W_g \oplus h(t-1)}} + \boldsymbol{b_g)}$$

$$\frac{\boldsymbol{dh(t)}}{\boldsymbol{db_g}} = \frac{\boldsymbol{\partial h(t)}}{\boldsymbol{\partial tanh}} \frac{\boldsymbol{\partial tanh}}{\boldsymbol{\partial c(t)}} * \boldsymbol{o(t)}\ \frac{\boldsymbol{\partial c(t)}}{\boldsymbol{\partial \tilde{c}(t)}} * \boldsymbol{i(t)}\ \frac{\boldsymbol{\partial \tilde{c}(t)}}{\boldsymbol{\partial tanh}} \underbrace{\frac{\boldsymbol{\partial tanh}}{\boldsymbol{\partial b_g}}}_{=\,\mathbf{1}}$$

```
dht      = dvalues[-1].reshape(self.n_neurons,1)

Tanh2[t].backward(dht)
dtanh2 = Tanh2[t].dinputs

dhtdtanh = np.multiply(O[t], dtanh2)

dctdct_tilde = np.multiply(dhtdtanh,I[t])

Tanh1[t].backward(dctdct_tilde)
dtanh1 = Tanh1[t].dinputs

dbg += dtanh1
```



We finally need *dh(t)* for the previous cell

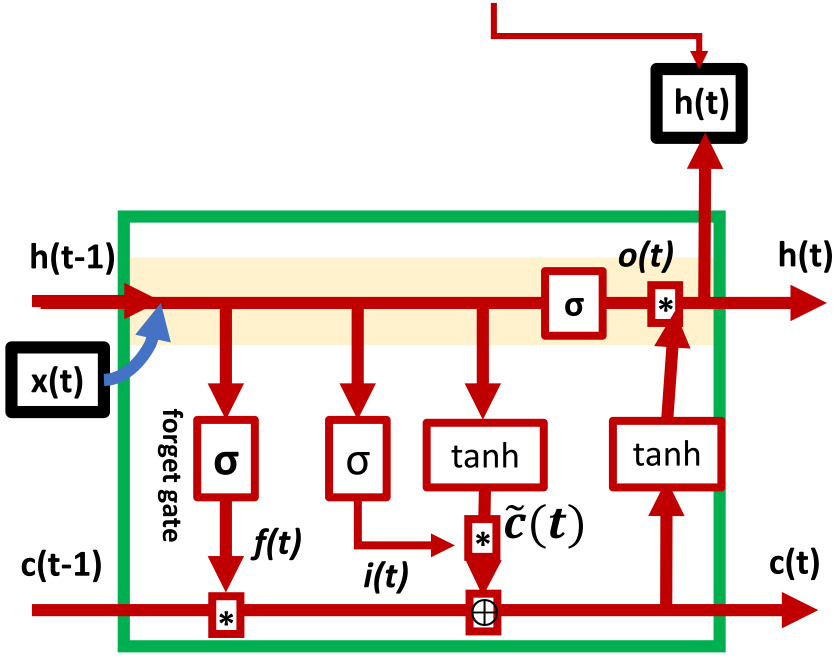
$$h(t) = tanh(c(t)) * o(t)$$
$$c(t) = f(t) * c(t-1) + i(t) * \tilde{c}(t)$$

$$f(t) = \sigma\left(U_f \oplus x(t) + W_f \oplus h(t-1) + b_f\right)$$
$$\tilde{c}(t) = tanh\left(U_g \oplus x(t) + W_g \oplus h(t-1) + b_g\right)$$
$$i(t) = \sigma\left(U_i \oplus x(t) + W_i \oplus h(t-1) + b_i\right)$$
$$o(t) = \sigma\left(U_o \oplus x(t) + W_o \oplus h(t-1) + b_o\right)$$

$$\frac{dh(t)}{dh(t-1)} =$$

```
dht =   np.dot(Wf, dsigmf) + np.dot(Wi, dsigmi) +\
        np.dot(Wo, dsigmo) + np.dot(Wg, dtanh1) +\
        dvalues[t-1,:].reshape(self.n_neurons, 1)
```

Δ = dense.dinputs



```
def backward(self, dvalues):

        T       = self.T
        H       = self.H
        C       = self.C

        O       = self.O
        I       = self.I
        C_tilde = self.C_tilde

        X_t     = self.X_t

        Sigmf   = self.Sigmf
        Sigmi   = self.Sigmi
        Sigmo   = self.Sigmo
        Tanh1   = self.Tanh1
        Tanh2   = self.Tanh2
```

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

        ...
        Tanh2    = self.Tanh2

        dht      = dvalues[-1,:].reshape(self.n_neurons,1)

       for t in reversed(range(T)):

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

            Tanh2[t].backward(dht)
            dtanh2 = Tanh2[t].dinputs

            dhtdtanh = np.multiply(O[t], dtanh2)

            dctdft       = np.multiply(dhtdtanh,C[t-1])
            dctdit       = np.multiply(dhtdtanh,C_tilde[t])
            dctdct_tilde = np.multiply(dhtdtanh,I[t])
```

actual BPTT

```python
def backward(self, dvalues):
        ...
        dht      = dvalues[-1].reshape(self.n_neurons,1)
```

actual BPTT

```python
        for t in reversed(range(T)):
            ...
            dctdct_tilde = np.multiply(dhtdtanh,I[t])

            Tanh1[t].backward(dctdct_tilde)
            dtanh1 = Tanh1[t].dinputs

            Sigmf[t].backward(dctdft)
            dsigmf = Sigmf[t].dinputs

            Sigmi[t].backward(dctdit)
            dsigmi = Sigmi[t].dinputs

            Sigmo[t].backward(np.multiply(dht, Tanh2[t].output))
            dsigmo = Sigmo[t].dinputs
```

```
def backward(self, dvalues):
        ...
        dht      = dvalues[-1].reshape(self.n_neurons,1)
```

actual BPTT

```
for t in reversed(range(T)):
        ...
        dsigmo = Sigmo[t].dinputs

        dsigmfdUf = np.dot(dsigmf,xt)
        dsigmfdWf = np.dot(dsigmf,H[t-1].T)

        self.dUf += dsigmfdUf
        self.dWf += dsigmfdWf
        self.dbf += dsigmf

        dsigmidUi = np.dot(dsigmi,xt)
        dsigmidWi = np.dot(dsigmi,H[t-1].T)

        self.dUi += dsigmidUi
        self.dWi += dsigmidWi
        self.dbi += dsigmi
```

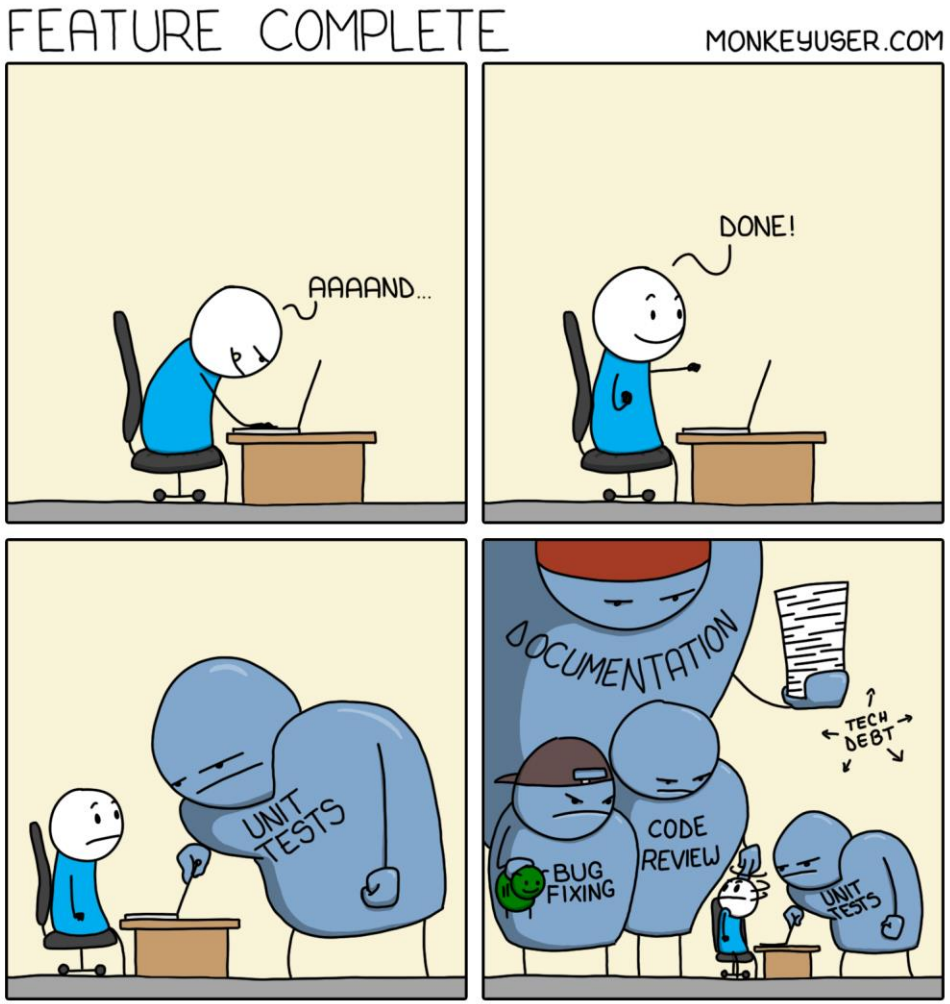
```
def backward(self, dvalues):
        ...
        dht      = dvalues[-1].reshape(self.n_neurons,1)
```

actual BPTT

```
        for t in reversed(range(T)):
                ...
                self.dbi += dsigmi

                dsigmodUo = np.dot(dsigmo,xt)
                dsigmodWo = np.dot(dsigmo,H[t-1].T)

                self.dUo += dsigmodUo
                self.dWo += dsigmodWo
                self.dbo += dsigmo

                dtanh1dUg = np.dot(dtanh1,xt)
                dtanh1dWg = np.dot(dtanh1,H[t-1].T)

                self.dUg += dtanh1dUg
                self.dWg += dtanh1dWg
                self.dbg += dtanh1
```

```
def backward(self, dvalues):
        ...
        dht      = dvalues[-1].reshape(self.n_neurons,1)

        for t in reversed(range(T)):
                ...
                self.dbg += dtanh1

                dht = np.dot(self.Wf, dsigmf) + np.dot(self.Wi, dsigmi) +\
                      np.dot(self.Wo, dsigmo) + np.dot(self.Wg, dtanh1) +\
                      dvalues[t-1,:].reshape(self.n_neurons, 1)

    self.H   = H
```

actual BPTT



outline:

- the idea
- from a classical RNN cell to a LSTM
- BackPropagation Through Time
- ***full backpropagtion***
- modifying the SGD optimizer
- running and testing the package

We want to test the backpropagation before we are going to modify our optimizer

```
from LSTM import *

n_neurons = 200

lstm = LSTM(n_neurons)
lstm.forward(X_t)

T              = max(X_t.shape)
dense1         = Layer_Dense(n_neurons, T)
dense2         = Layer_Dense(T, 1)

lr = 1e-5
Monitor = np.zeros((100))
```

adding a dense layer for

h(t) → Y_hat(t)

We want to test the backpropagation before we are going to modify our optimizer

```
lr = 1e-5
Monitor = np.zeros((100))

for i in range(100):

        lstm.forward(X_t)

        H = np.array(lstm.H)
        H = H.reshape((H.shape[0],H.shape[1]))

        dense1.forward(H[1:,:])
        dense2.forward(dense1.output)

        Y_hat = dense2.output

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

        Monitor[i] = L
```

We want to test the backpropagation before we are going to modify our optimizer

```
for i in range(100):
...
       Monitor[i] = L

       dense2.backward(dY)
       dense1.backward(dense2.dinputs)

       lstm.backward(dense1.dinputs)

       dense1.weights -= lr*dense1.dweights
       dense2.weights -= lr*dense2.dweights

       dense1.biases -= lr*dense1.dbiases
       dense2.biases -= lr*dense2.dbiases
```

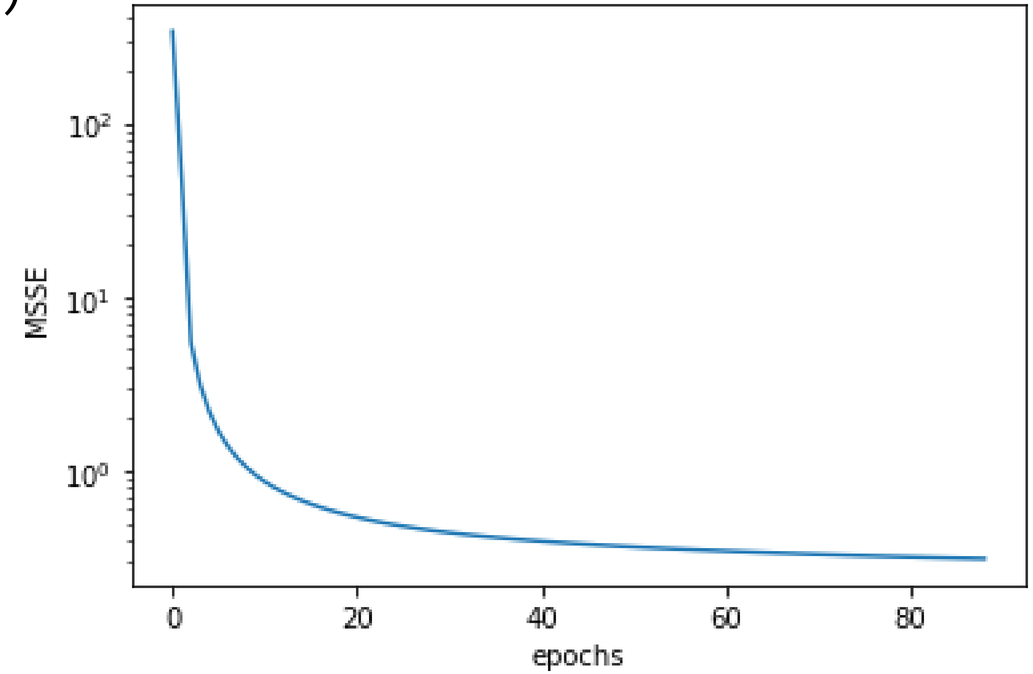
We want to test the backpropagation before we are going to modify our optimizer

```
for i in range(100):
...
        dense2.biases -= lr*dense2.dbiases

        lstm.Uf -= lr*lstm.dUf
        lstm.Ui -= lr*lstm.dUi
        lstm.Uo -= lr*lstm.dUo
        lstm.Ug -= lr*lstm.dUg

        lstm.Wf -= lr*lstm.dWf
        lstm.Wi -= lr*lstm.dWi
        lstm.Wo -= lr*lstm.dWo
        lstm.Wg -= lr*lstm.dWg

        lstm.bf -= lr*lstm.dbf
        lstm.bi -= lr*lstm.dbi
        lstm.bo -= lr*lstm.dbo
        lstm.bg -= lr*lstm.dbg

        print(f'current MSSE = {L:.3f}')
```

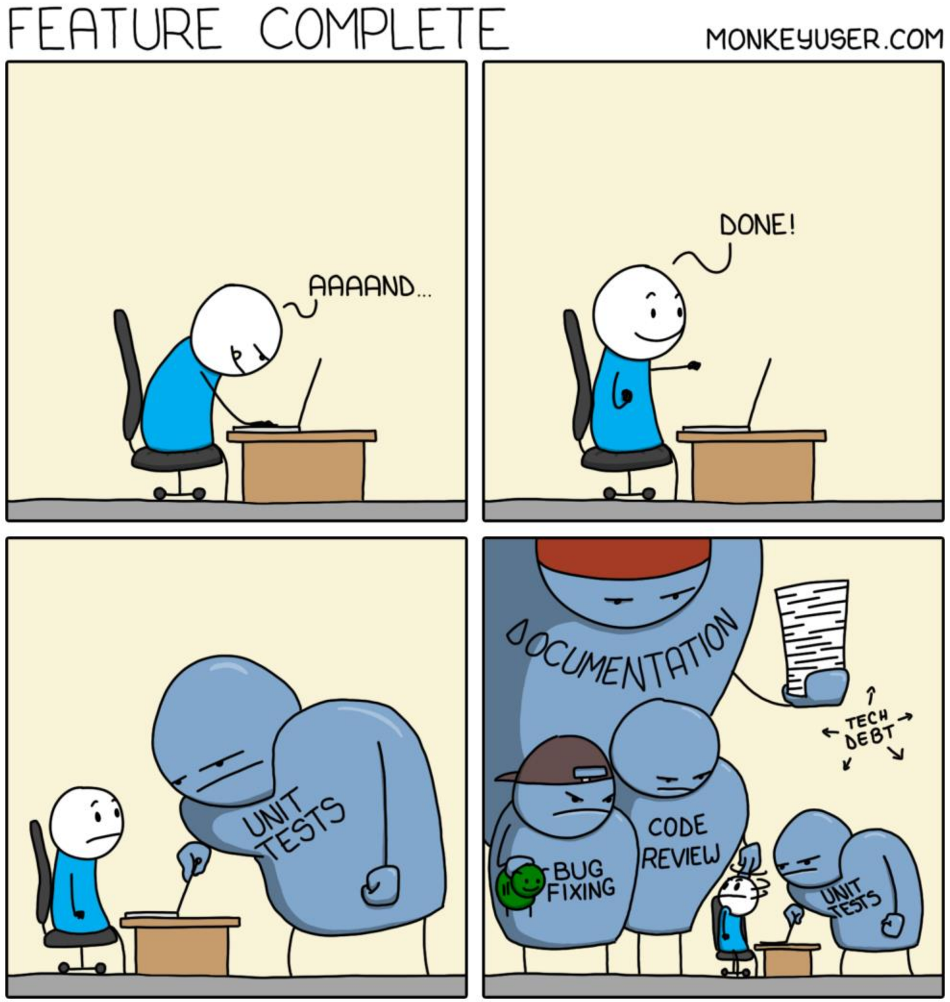
We want to test the backpropagation before we are going to modify our optimizer

```
for i in range(100):
...
        print(f'current MSSE = {L:.3f}’)
```

```
plt.plot(range(100),Monitor)
plt.xlabel('epochs')
plt.ylabel('MSSE')
```




```
current MSSE = 337.707
current MSSE = 47.685
current MSSE = 5.465
current MSSE = 3.140
current MSSE = 2.186
current MSSE = 1.688
current MSSE = 1.388
current MSSE = 1.191
current MSSE = 1.052
current MSSE = 0.949
current MSSE = 0.870
current MSSE = 0.807
current MSSE = 0.756
current MSSE = 0.713
current MSSE = 0.677
current MSSE = 0.647
current MSSE = 0.620
current MSSE = 0.597
current MSSE = 0.576
```



outline:

- *the idea*
- *from a classical RNN cell to a LSTM*
- *BackPropagation Through Time*
- *full backpropagtion*
- ***modifying the SGD optimizer***
- *running and testing the package*

We want to use **two** optimizer: `Optimizer_SGD` from the previous lecture(s) and…

```
class Optimizer_SGD_LSTM:
        ...

    def update_params(self, layer):

        if self.momentum:

            if not hasattr(layer, 'Uf_momentums'):
                layer.Uf_momentums = np.zeros_like(layer.Uf)
                layer.Ui_momentums = np.zeros_like(layer.Ui)
                layer.Uo_momentums = np.zeros_like(layer.Uo)
                layer.Ug_momentums = np.zeros_like(layer.Ug)

                layer.Wf_momentums = np.zeros_like(layer.Wf)
                layer.Wi_momentums = np.zeros_like(layer.Wi)
                layer.Wo_momentums = np.zeros_like(layer.Wo)
                layer.Wg_momentums = np.zeros_like(layer.Wg)

                layer.bf_momentums = np.zeros_like(layer.bf)
                layer.bi_momentums = np.zeros_like(layer.bi)
                layer.bo_momentums = np.zeros_like(layer.bo)
                layer.bg_momentums = np.zeros_like(layer.bg)
```

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

note: try

`dir(lstm)`

as alternative

We want to use **two** optimizer: Optimizer_SGD from the previous lecture(s) and…

```
class Optimizer_SGD_LSTM:
        ...

    def update_params(self, layer):

            if self.momentum:

                if not hasattr(layer, 'Uf_momentums’):
                ...

             Uf_updates = self.momentum * layer.Uf_momentums - \
                          self.current_learning_rate * layer.dUf
             layer.Uf_momentums = Uf_updates
```

for more details see my *“ANN from Scratch”* lecture

do this for all learnables of the LSTM

We want to use **two** optimizer: Optimizer_SGD from the previous lecture(s) and…

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

```
class Optimizer_SGD_LSTM:
      ...

  def update_params(self, layer):

        if self.momentum:

            if not hasattr(layer, 'Uf_momentums'):
            ...

        Uf_updates = self.momentum * layer.Uf_momentums - \
                         self.current_learning_rate * layer.dUf
        layer.Uf_momentums = Uf_updates

            ...

        else:

          Uf_updates = -self.current_learning_rate * layer.dUf
          Ui_updates = -self.current_learning_rate * layer.dUi
          Uo_updates = -self.current_learning_rate * layer.dUo
          Ug_updates = -self.current_learning_rate * layer.dUg
```

do this for all learnables of the LSTM

We want to use **two** optimizer: Optimizer_SGD from the previous lecture(s) and…

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

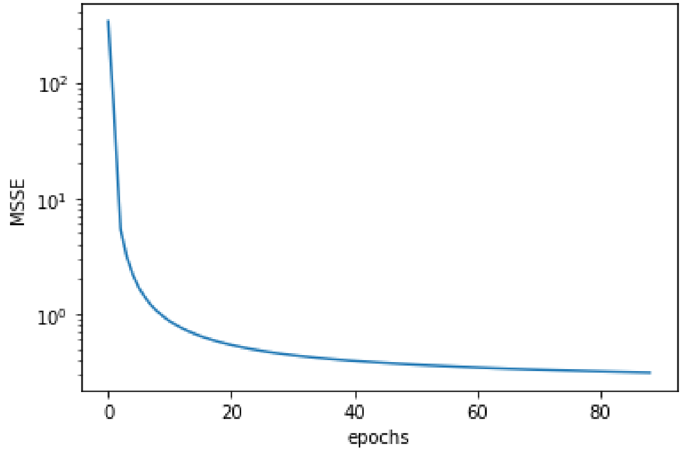
```
class Optimizer_SGD_LSTM:
        ...

    def update_params(self, layer):
            ...
            else:

                Uf_updates = -self.current_learning_rate * layer.dUf
                Ui_updates = -self.current_learning_rate * layer.dUi
                Uo_updates = -self.current_learning_rate * layer.dUo
                Ug_updates = -self.current_learning_rate * layer.dUg
                ...

            layer.Uf += Uf_updates
            layer.Ui += Ui_updates
            layer.Uo += Uo_updates
            layer.Ug += Ug_updates
                ...
```

```
lstm.Uf -= lr*lstm.dUf
lstm.Ui -= lr*lstm.dUi
lstm.Uo -= lr*lstm.dUo
lstm.Ug -= lr*lstm.dUg
```

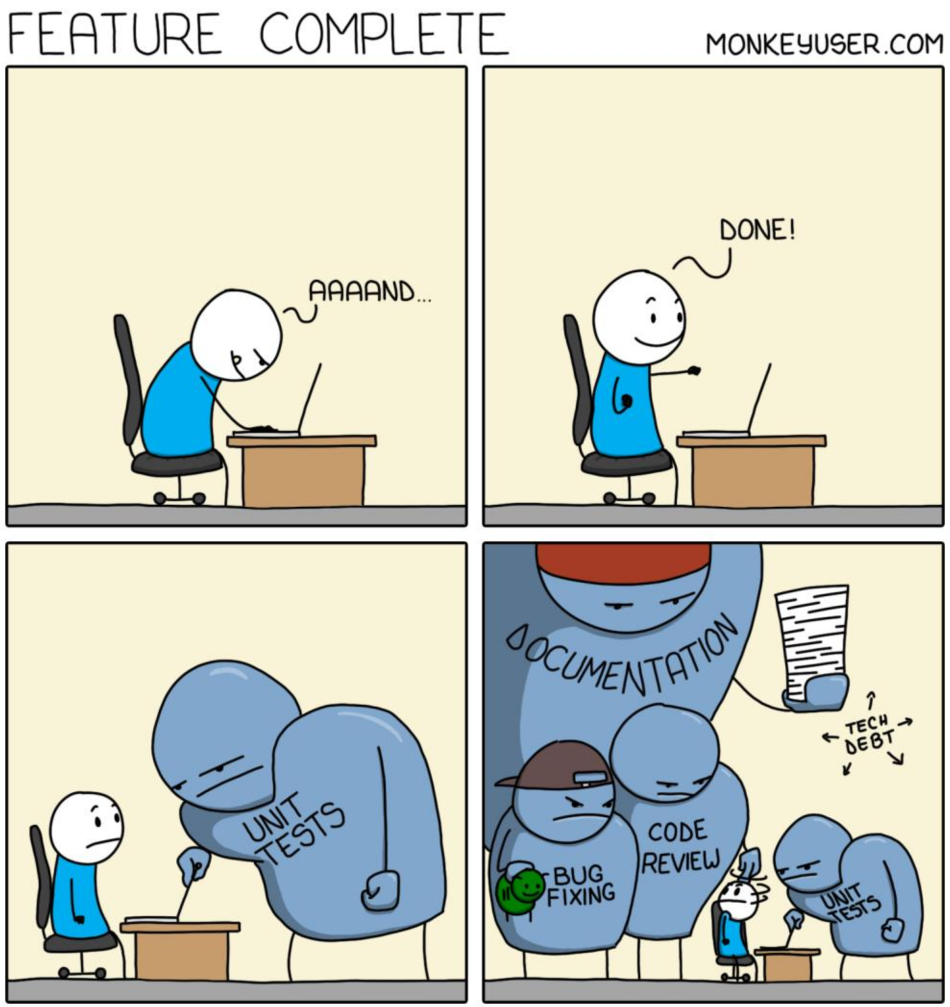
do this for all learnables of the LSTM

```
optimizerLSTM = Optimizer_SGD_LSTM(learning_rate = 1e-5)
optimizer     = Optimizer_SGD(learning_rate = 1e-5)
Monitor = np.zeros((100))

for i in range(100):

        lstm.forward(X_t)

        H = np.array(lstm.H)
        H = H.reshape((H.shape[0],H.shape[1]))

        dense1.forward(H[1:,:])
        dense2.forward(dense1.output)

        ...
```

```
...
dense2.forward(dense1.output)
...
optimizer_lstm.pre_update_params()
optimizer.pre_update_params()

optimizer.update_params(dense1)
optimizer.update_params(dense2)

optimizer_lstm.update_params(lstm)

optimizer_lstm.post_update_params()
optimizer.post_update_params()
```



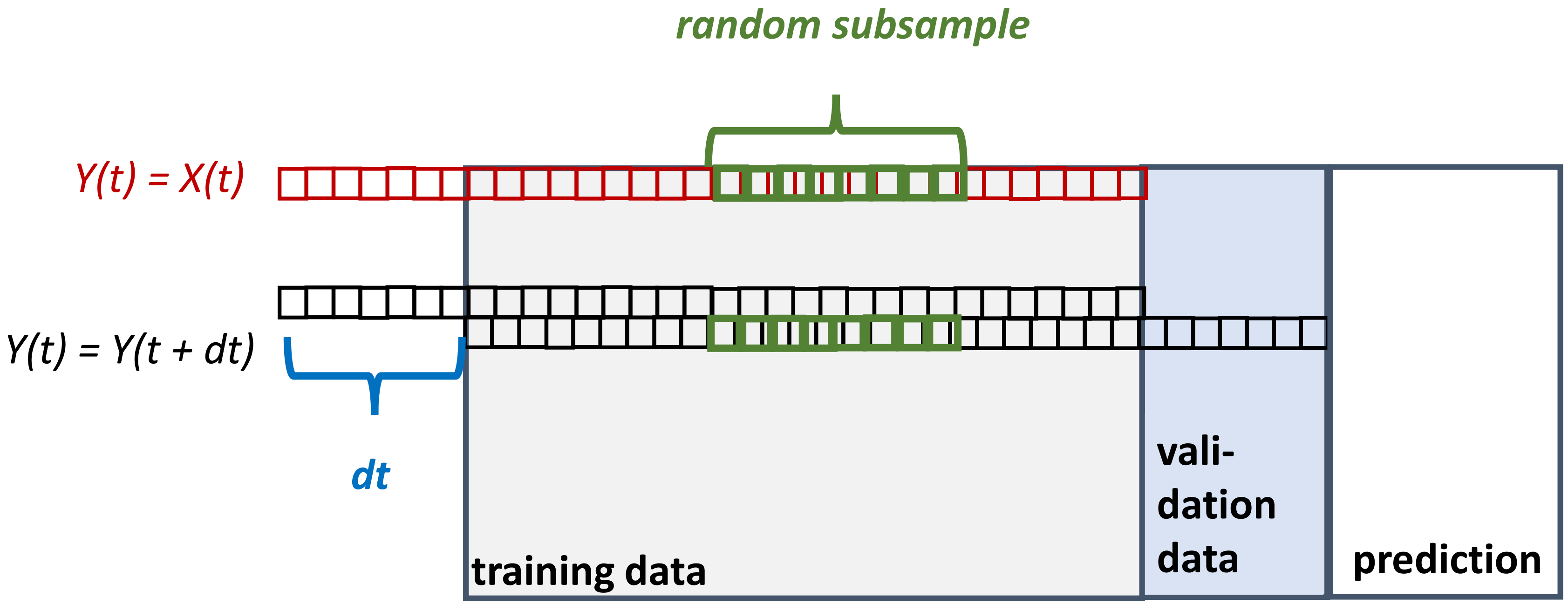

```
current MSSE = 337.707
current MSSE = 47.685
current MSSE = 5.465
current MSSE = 3.140
current MSSE = 2.186
current MSSE = 1.688
current MSSE = 1.388
current MSSE = 1.191
current MSSE = 1.052
current MSSE = 0.949
current MSSE = 0.870
current MSSE = 0.807
current MSSE = 0.756
current MSSE = 0.713
current MSSE = 0.677
current MSSE = 0.647
current MSSE = 0.620
current MSSE = 0.597
current MSSE = 0.576
```



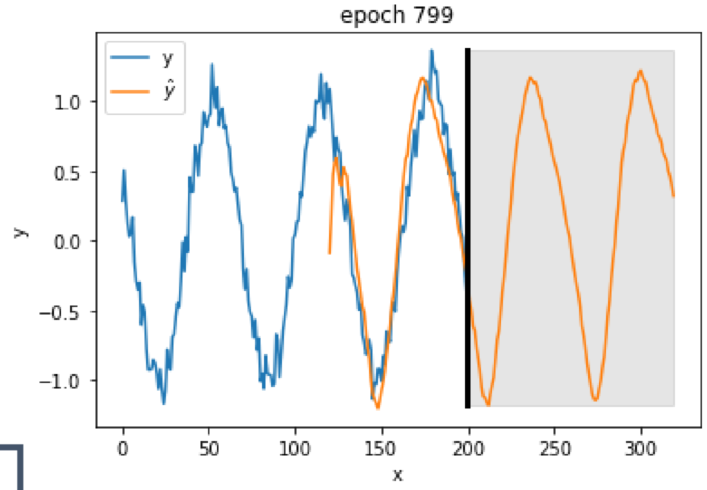
outline:

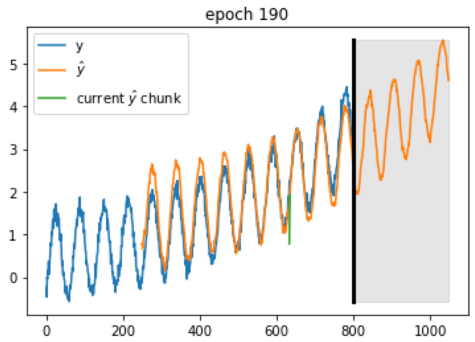
- *the idea*
- *from a classical RNN cell to a LSTM*
- *BackPropagation Through Time*
- *full backpropagtion*
- *modifying the SGD optimizer*
- ***running and testing the package***

now: *Y(t)* → *Y(t + dt)*



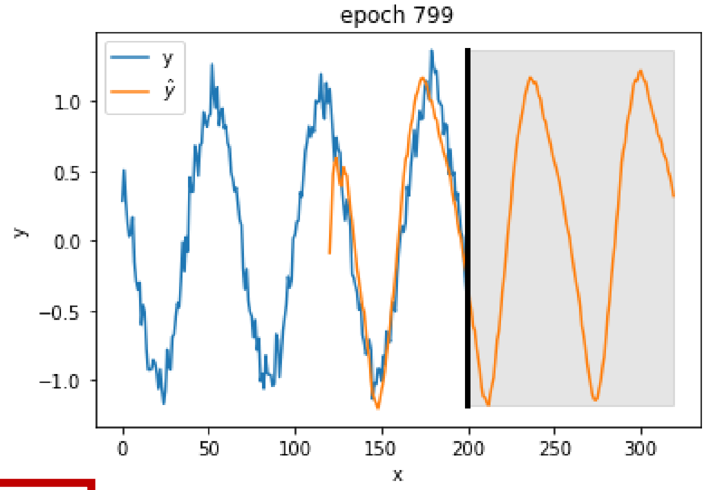
classical RNN



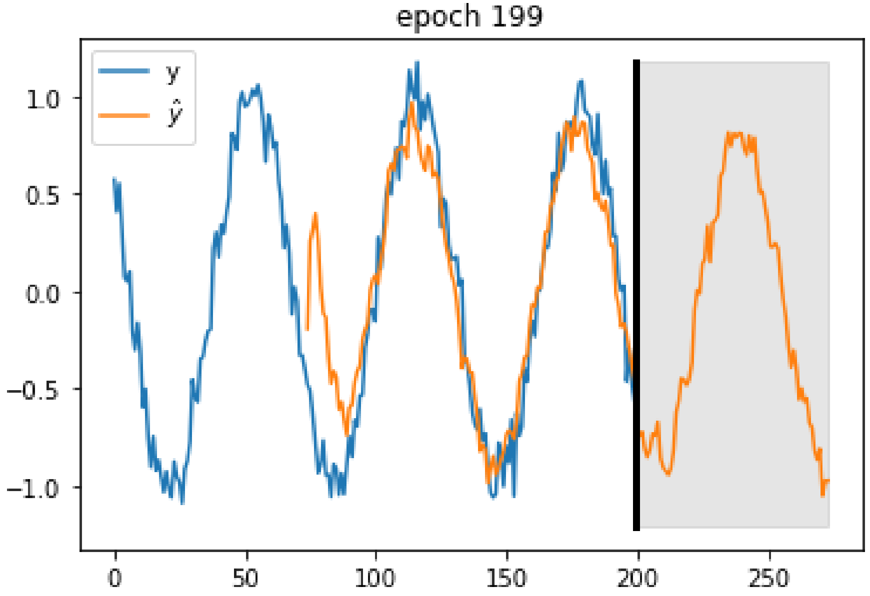


now: $Y(t) \rightarrow Y(t + dt)$

classical RNN



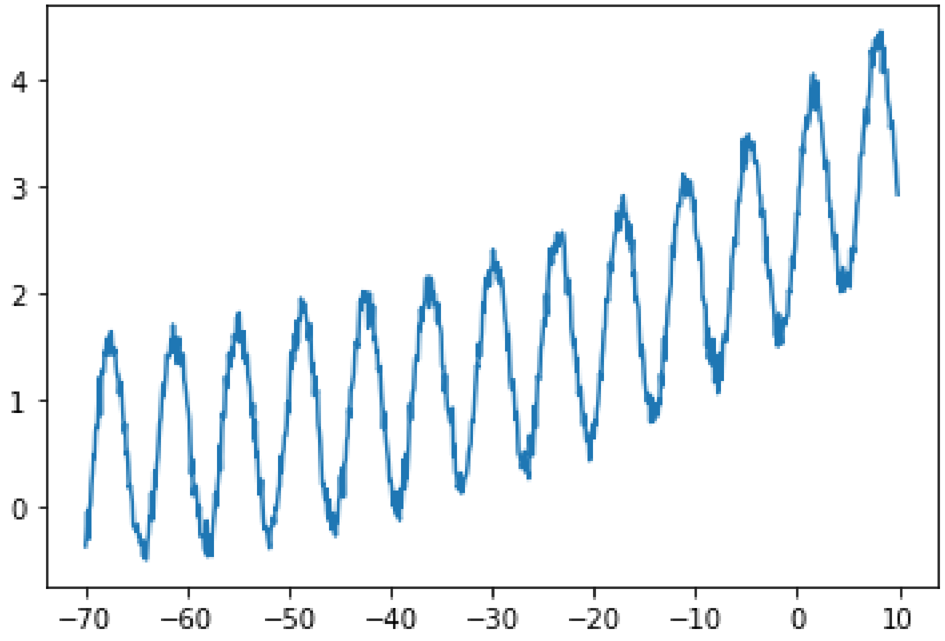

```
dt    = 75
[lstm, dense1, dense2] = RunMyLSTM(Y_t, Y_t, n_neurons = 200,\
                               n_epoch = 200, plot_each = 10, dt = dt,\
                               momentum = 0.8, decay = 0.01,\
                               learning_rate = 1e-3)
```



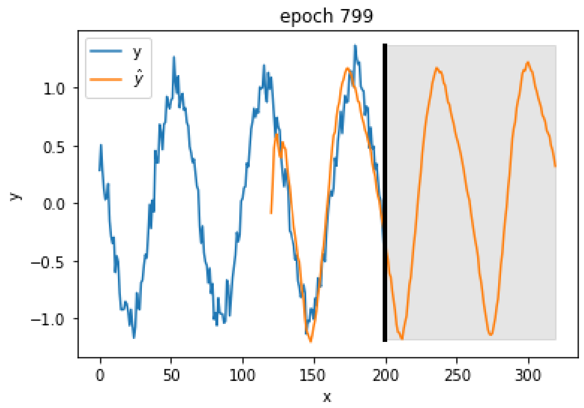
That is a lot better (and faster!) than the classical RNN!

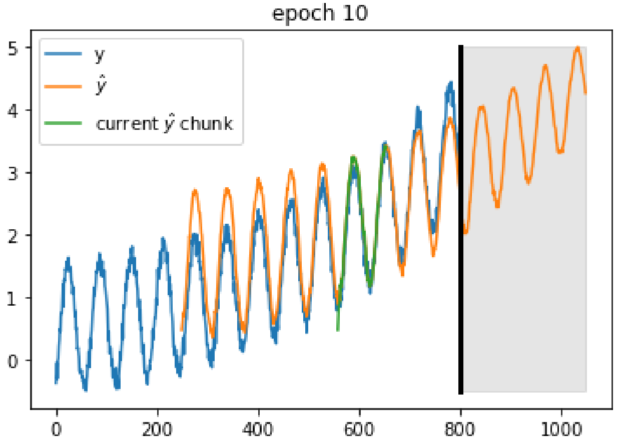
now: *Y(t)* → *Y(t + dt)*

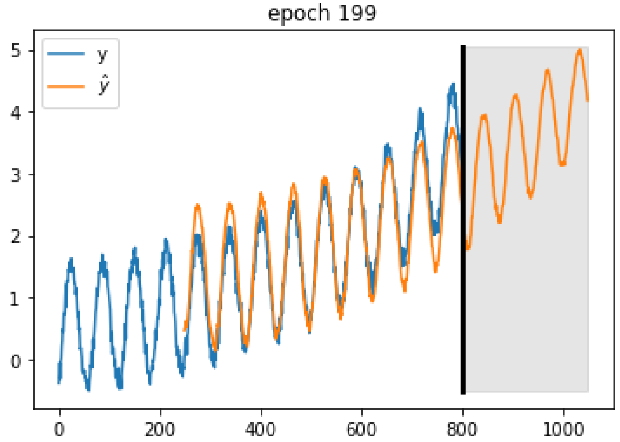
Let's challenge the LSTM and include long-term changes:



classical RNN





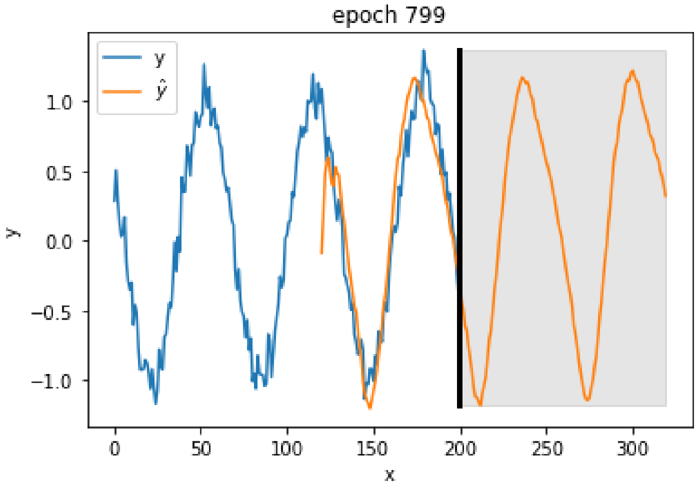



```
dt    = 250
[lstm, dense1, dense2] = RunMyLSTM(Y_t, Y_t, n_neurons = 200,\
                                   n_epoch = 200, plot_each = 10, dt = dt,\
                                   momentum = 0.8, decay = 0.01,\
                                   learning_rate = 1e-5)
```

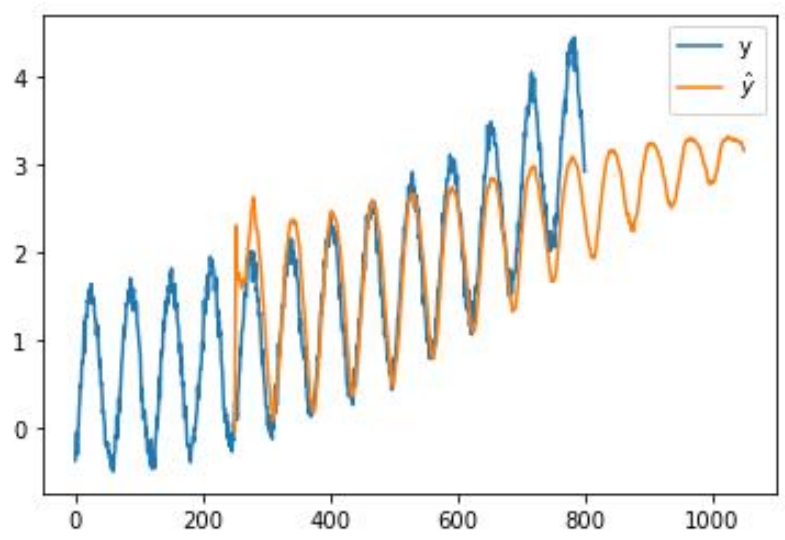
now: $Y(t)$ → $Y(t + dt)$

Let's do the same thing with the **RNN** now:

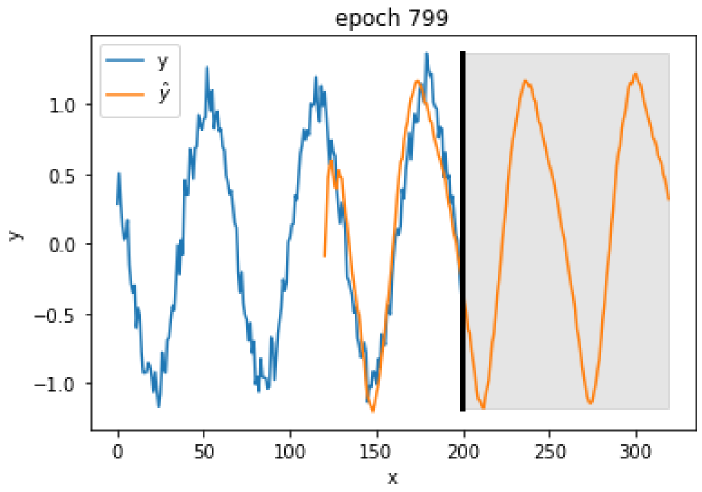
classical RNN

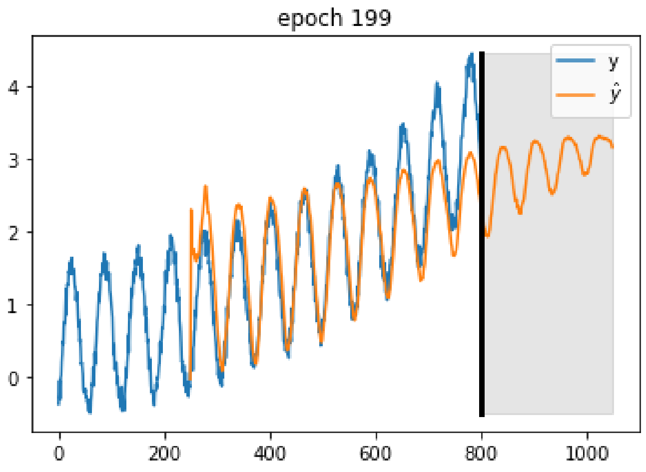



```
dt  = 250
rnn = RunMyRNN(Y_t, Y_t, n_neurons = 200,\
                                   n_epoch = 200, plot_each = 10, dt = dt,\
                                   momentum = 0.8, decay = 0.01,\
                                   learning_rate = 1e-5)

Y_hat      = ApplyMyRNN(Y_t, rnn)

X_plot     = np.arange(0, len(Y_t))
X_plot_hat = np.arange(0, len(Y_hat)) + dt

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

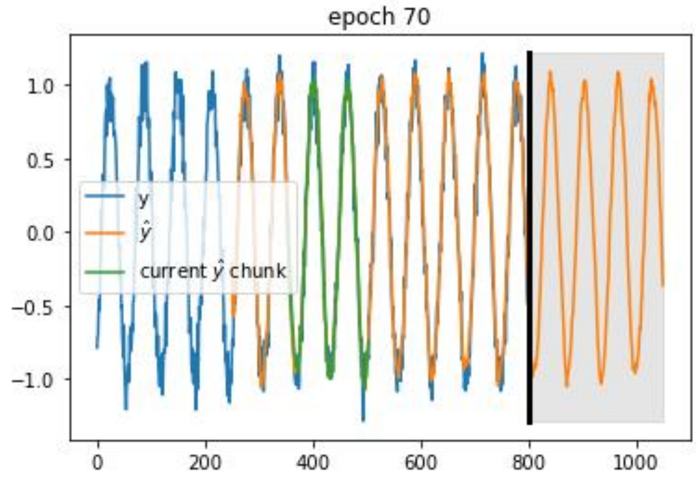


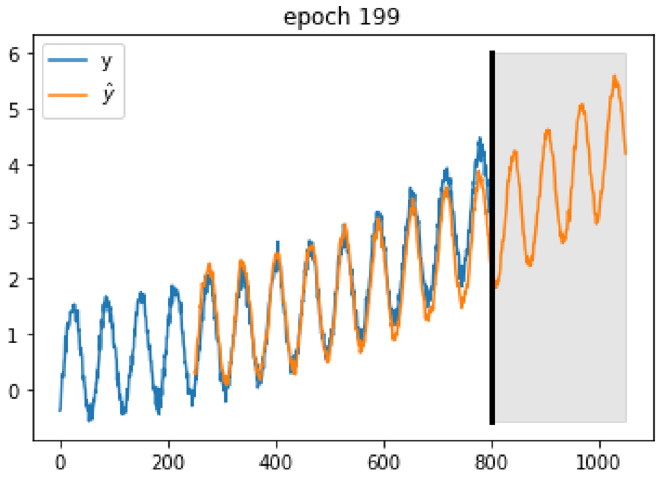
classical RNN

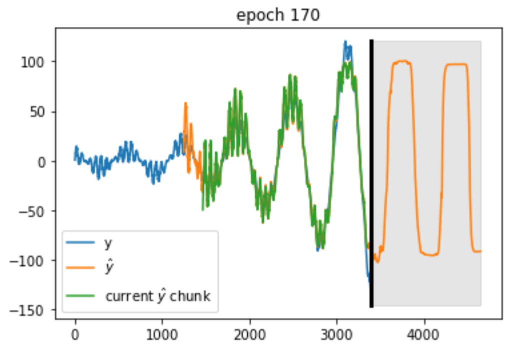




LSTM







Thank you very much for your attention!