Machine Learning for Physicists Lecture 8

Summer 2017

University of Erlangen-Nuremberg
Florian Marquardt

Long short-term memory (LSTM)

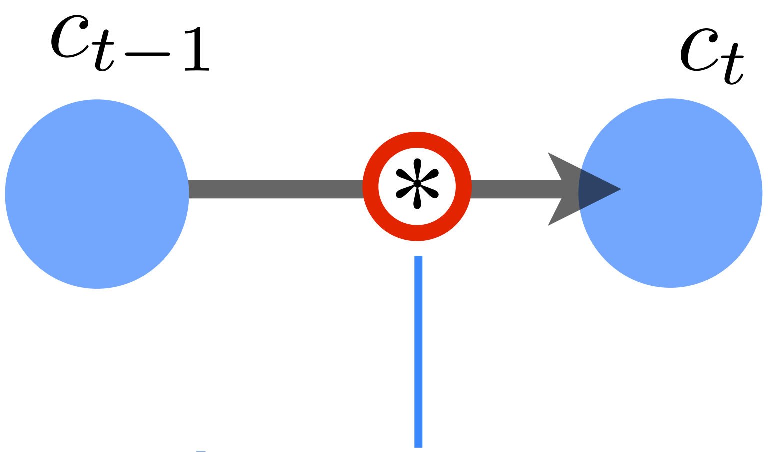
Why this name? "Long-term memory" would be the weights that are adapted during training and then stored forever. "Short-term memory" is the input-dependent memory we are talking about here. "Long short-term memory" tries to have long memory times in a robust way, for this short-term memory.

Sepp Hochreiter and Jürgen Schmidhuber, 1997

Main idea: determine read/write/delete operations of a memory cell via the network (through other neurons)
Most of the time, a memory neuron just sits there and is not used/changed!

LSTM: Forget gate (delete)

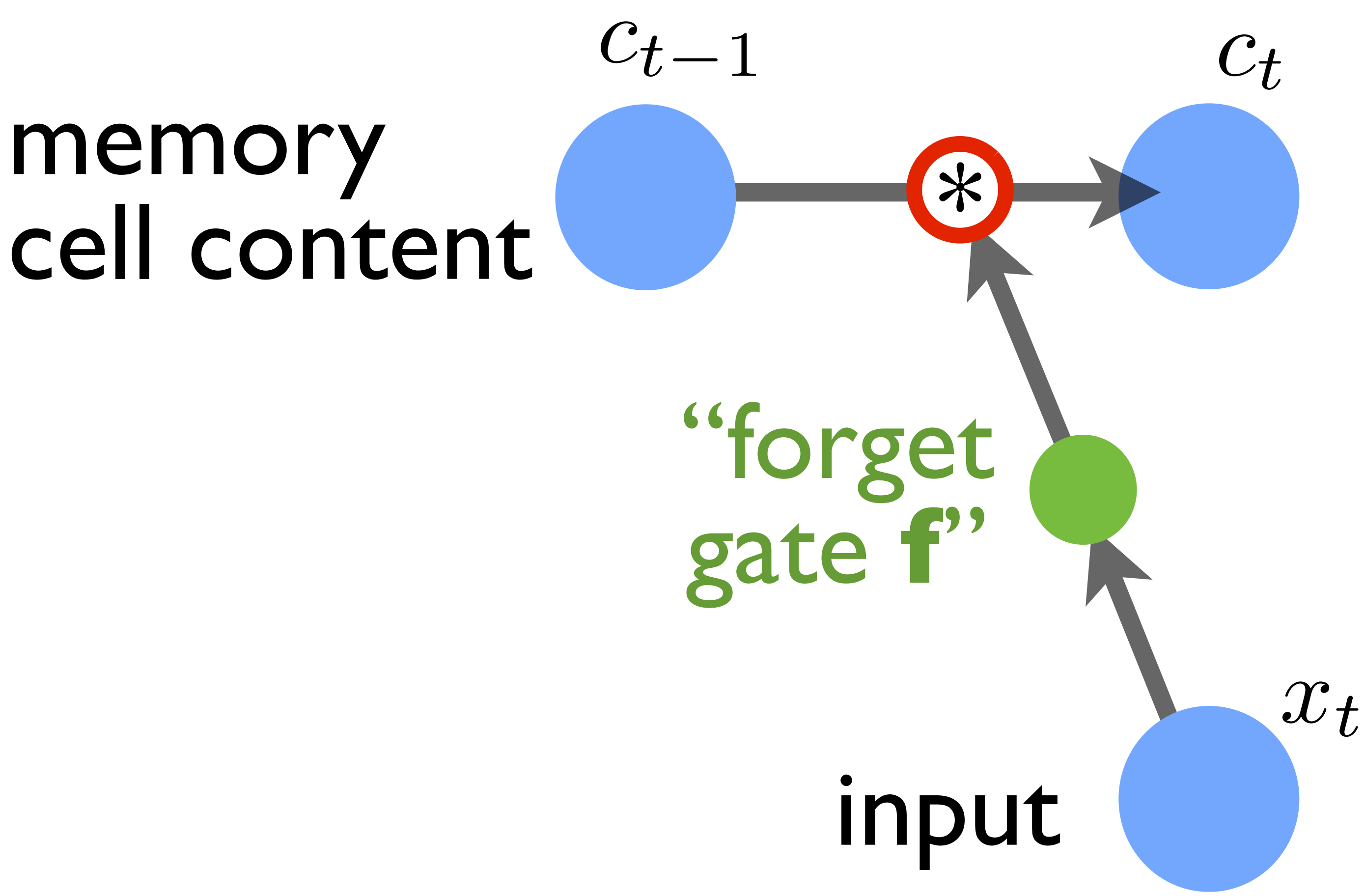
memory cell content



keep: $c_t = 1 * c_{t-1}$

delete: $c_t = 0 * c_{t-1}$

LSTM: Forget gate (delete)



Calculate “forget gate”:

$$f = \sigma(W^{(f)} x_t + b^{(f)})$$

sigmoid

(usually x,b,f are vectors, W the weight matrix)

Obtain new memory content:

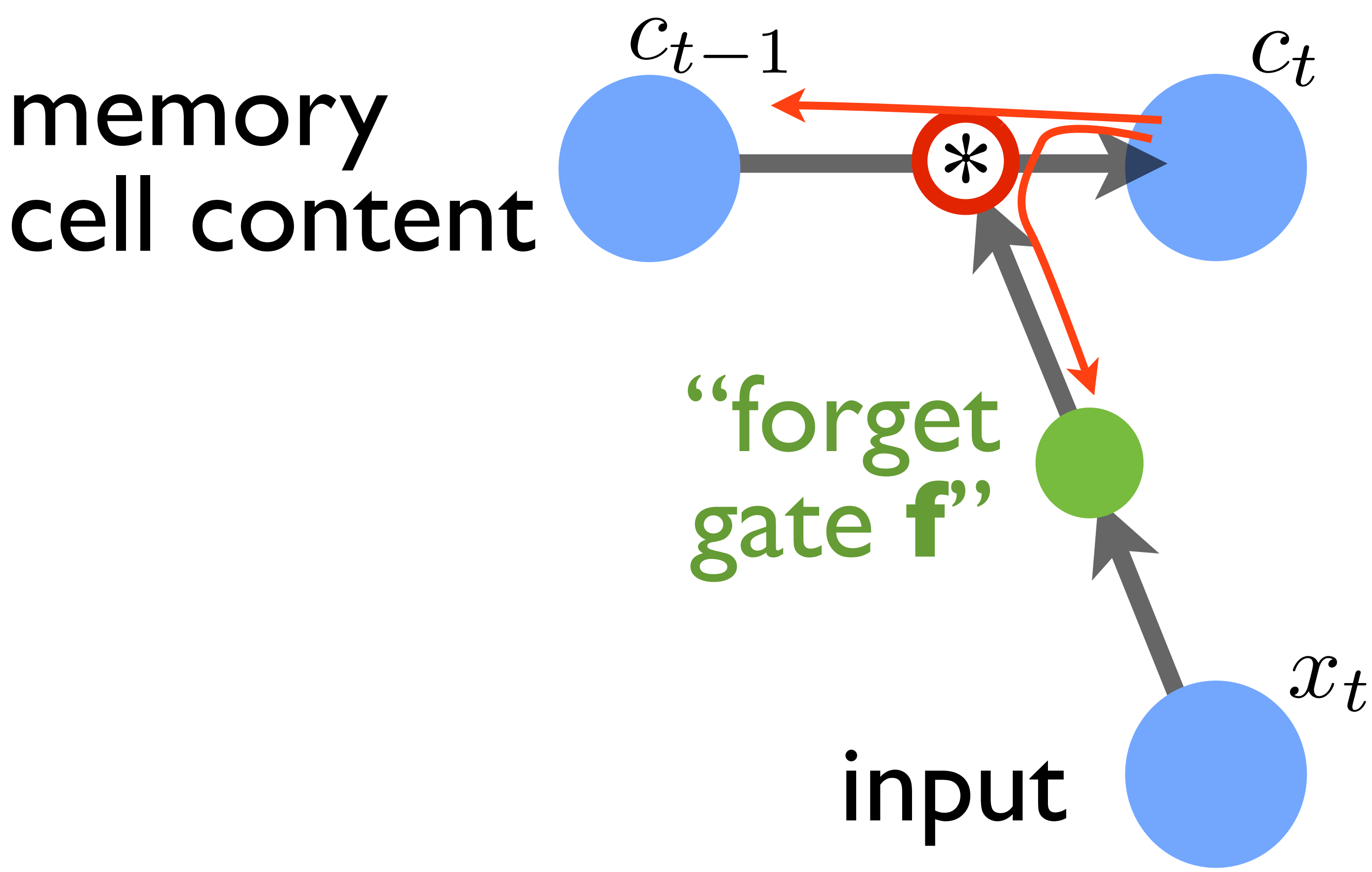
$$c_t = f * c_{t-1}$$

elementwise product

NEW: for the first time, we are **multiplying** neuron values!

LSTM: Forget gate (delete)

Backpropagation



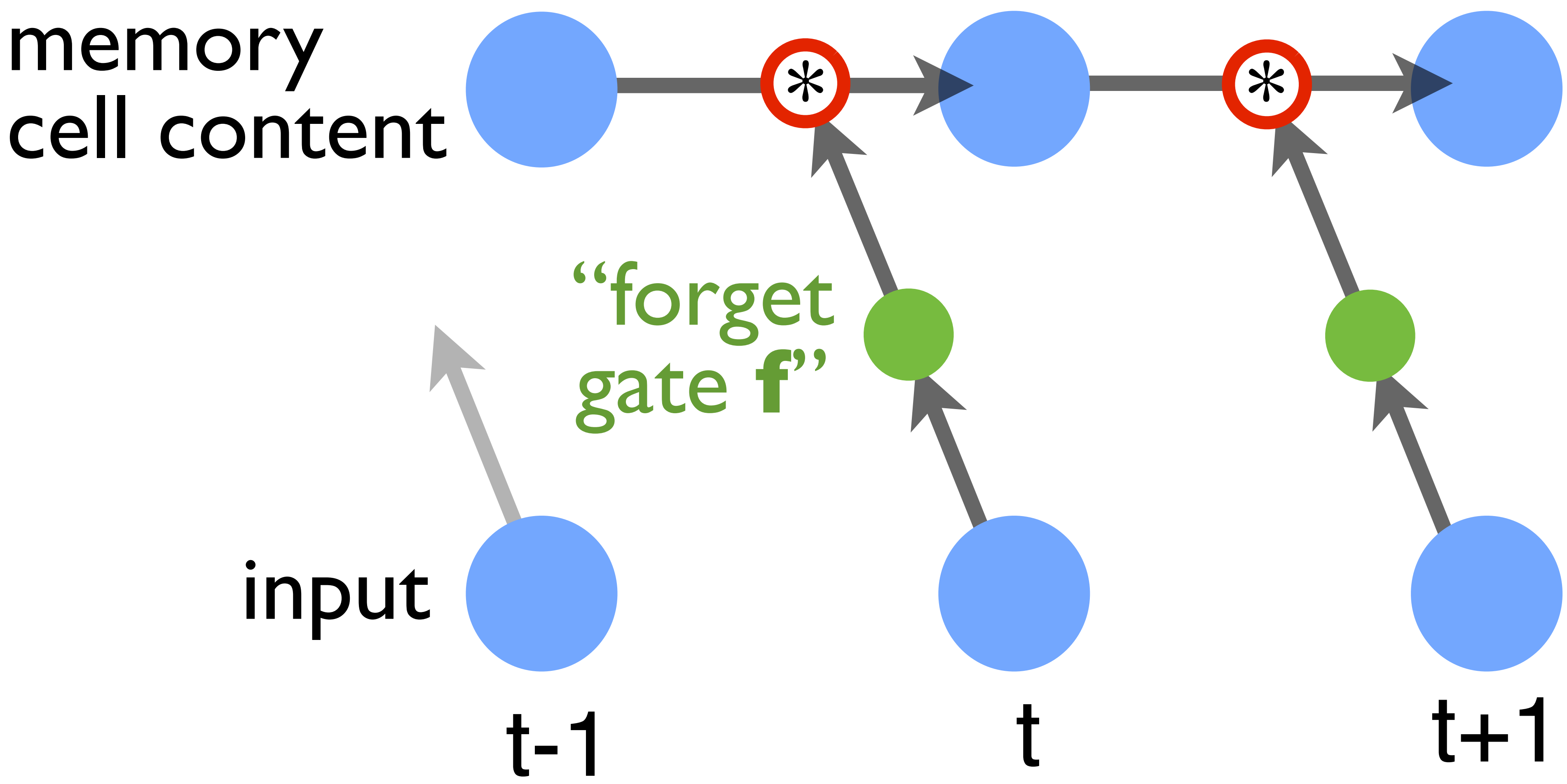
The multiplication * splits the error backpropagation into two branches

product rule:

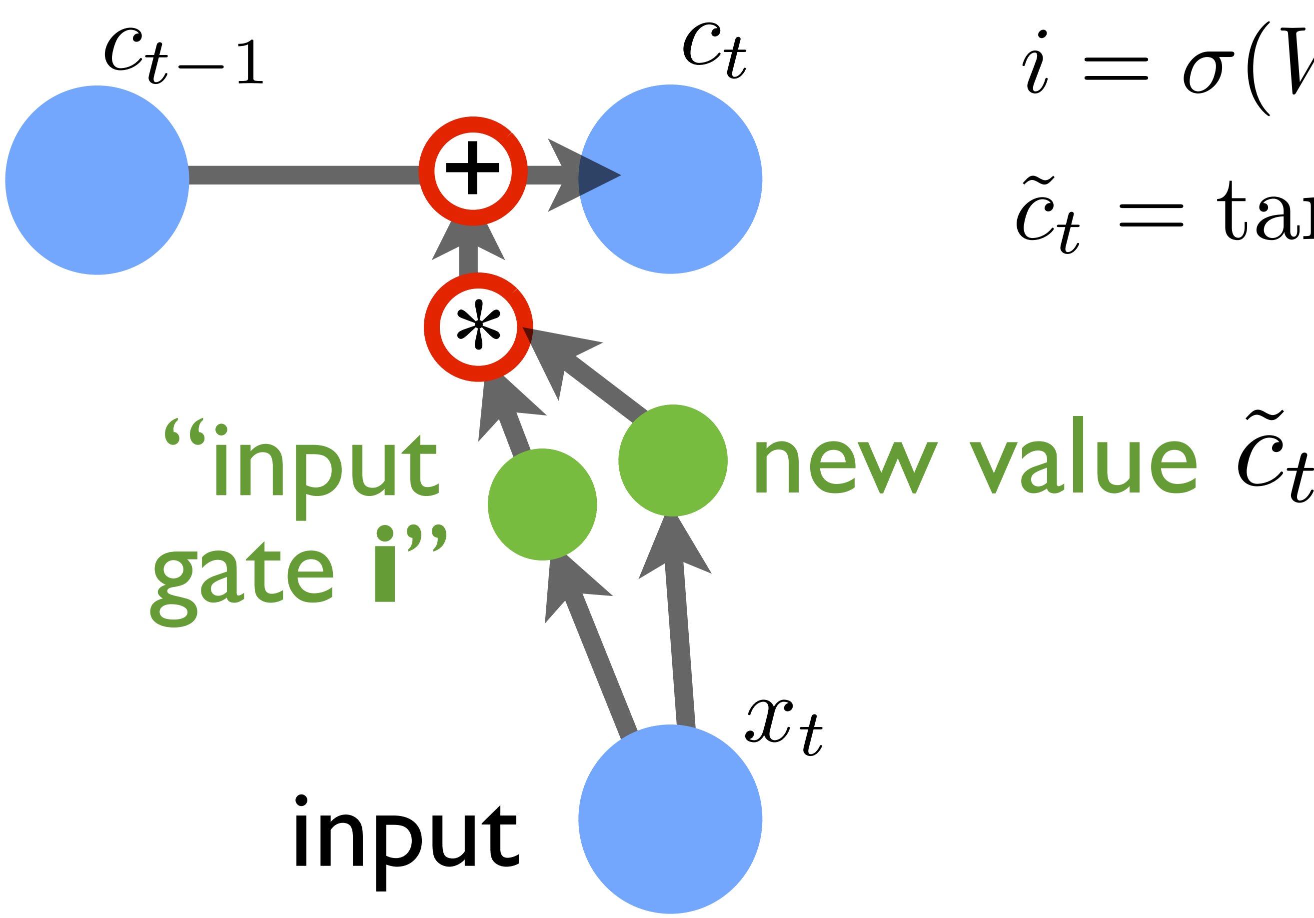
$$\frac{\partial f_j c_{t-1,j}}{\partial w_*} = \frac{\partial f_j}{\partial w_*} c_{t-1,j} + f_j \frac{\partial c_{t-1,j}}{\partial w_*}$$

(Note: if time is not specified, we are referring to t)

LSTM: Forget gate (delete)



LSTM: Write new memory value



$$i = \sigma(W^{(i)} x_t + b^{(i)})$$

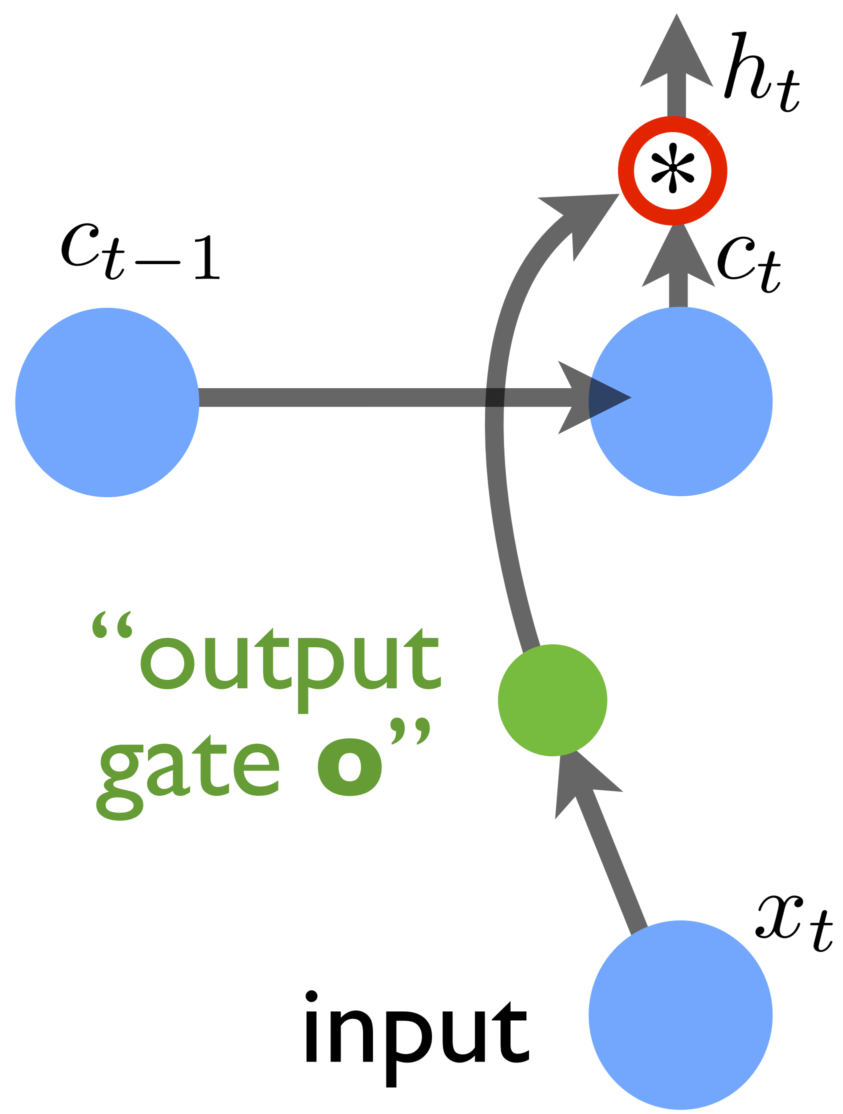
$$\tilde{c}_t = \tanh(W^{(c)} x_t + b^{(c)})$$

both delete and write together:

$$c_t = f * c_{t-1} + i * \tilde{c}_t$$

forget   new value

LSTM: Read (output) memory value



$$o = \sigma(W^{(o)} x_t + b^{(o)})$$

$$h_t = o * \tanh(c_t)$$

LSTM: exploit previous memory output 'h'

make f,i,o etc. at time t depend on output 'h' calculated in previous time step!

(otherwise: 'h' could only be used in higher layers, but not to control memory access in present layer)

$$f = \sigma(W^{(f)} x_t + U^{(f)} h_{t-1} + b^{(f)})$$

...and likewise for every other quantity!

Thus, result of readout can actually influence subsequent operations (e.g.: readout of some selected other memory cell!)

Sometimes, o is even made to depend on c_t

LSTM: backpropagation through time is OK

As long as memory content is not read or written, the backpropagation gradient is trivial:

$$c_t = c_{t-1} = c_{t-2} = \dots$$

$$\frac{\partial c_t}{\partial w_*} = \frac{\partial c_{t-1}}{\partial w_*} = \frac{\partial c_{t-2}}{\partial w_*} = \dots$$

(deviation vector multiplied by 1)

During those 'silent' time-intervals: No explosion or vanishing gradient!

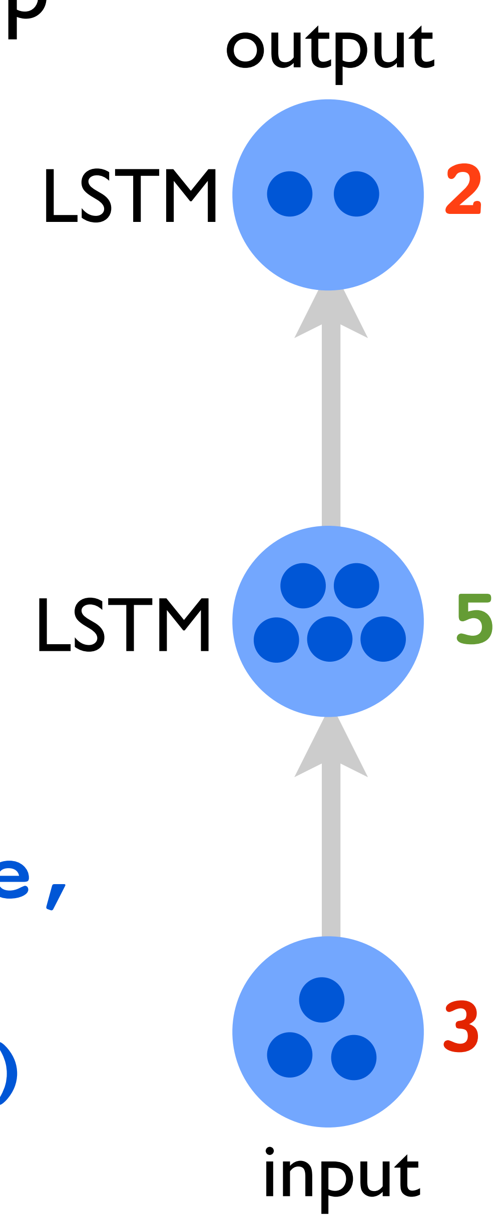
Adding an LSTM layer with 10 memory cells:

Each of those cells has the full structure, with **f**,**i**,**o** gates and the memory content **c**, and the output **h**.

```
rnn.add(LSTM(10, return_sequences=True))
```

whether to return the full time sequence of outputs, or only the output at the final time

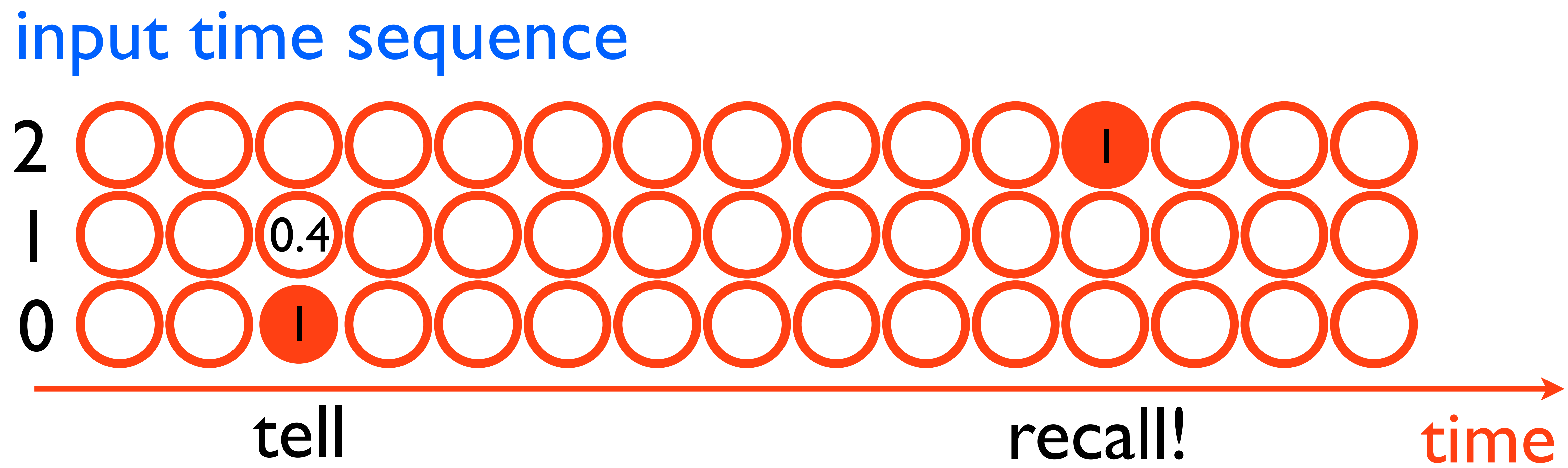
Two LSTM layers (input > LSTM > LSTM=output), taking an input of 3 neuron values for each time step and producing a time sequence with 2 neuron values for each time step

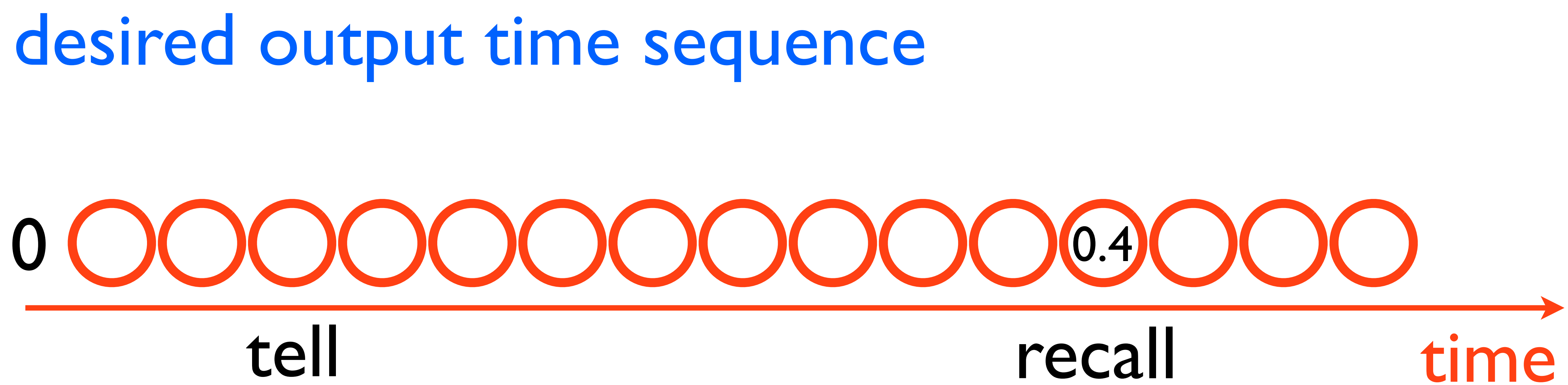



```
def init_memory_net():
    global rnn, batchsize, timesteps
     rnn = Sequential()
    # note: batch_input_shape is
(batchsize,timesteps,data_dim)
    rnn.add(LSTM(5, batch_input_shape=(None,
   timesteps, 3), return_sequences=True))
    rnn.add(LSTM(2, return_sequences=True))
    rnn.compile(loss='mean_squared_error',
   optimizer='adam', metrics=['accuracy'])
```

Example: A network for recall

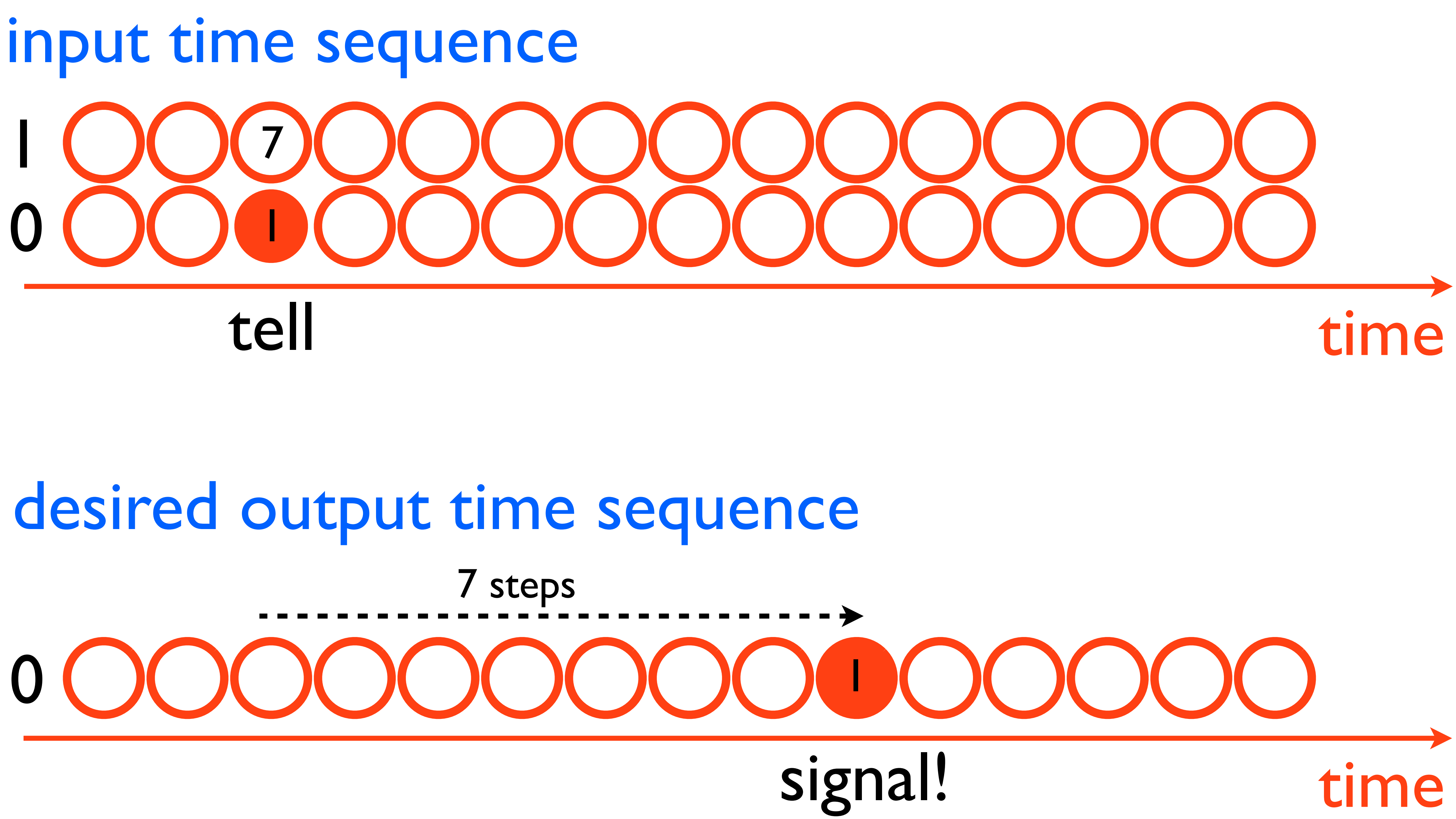
(see code on website)



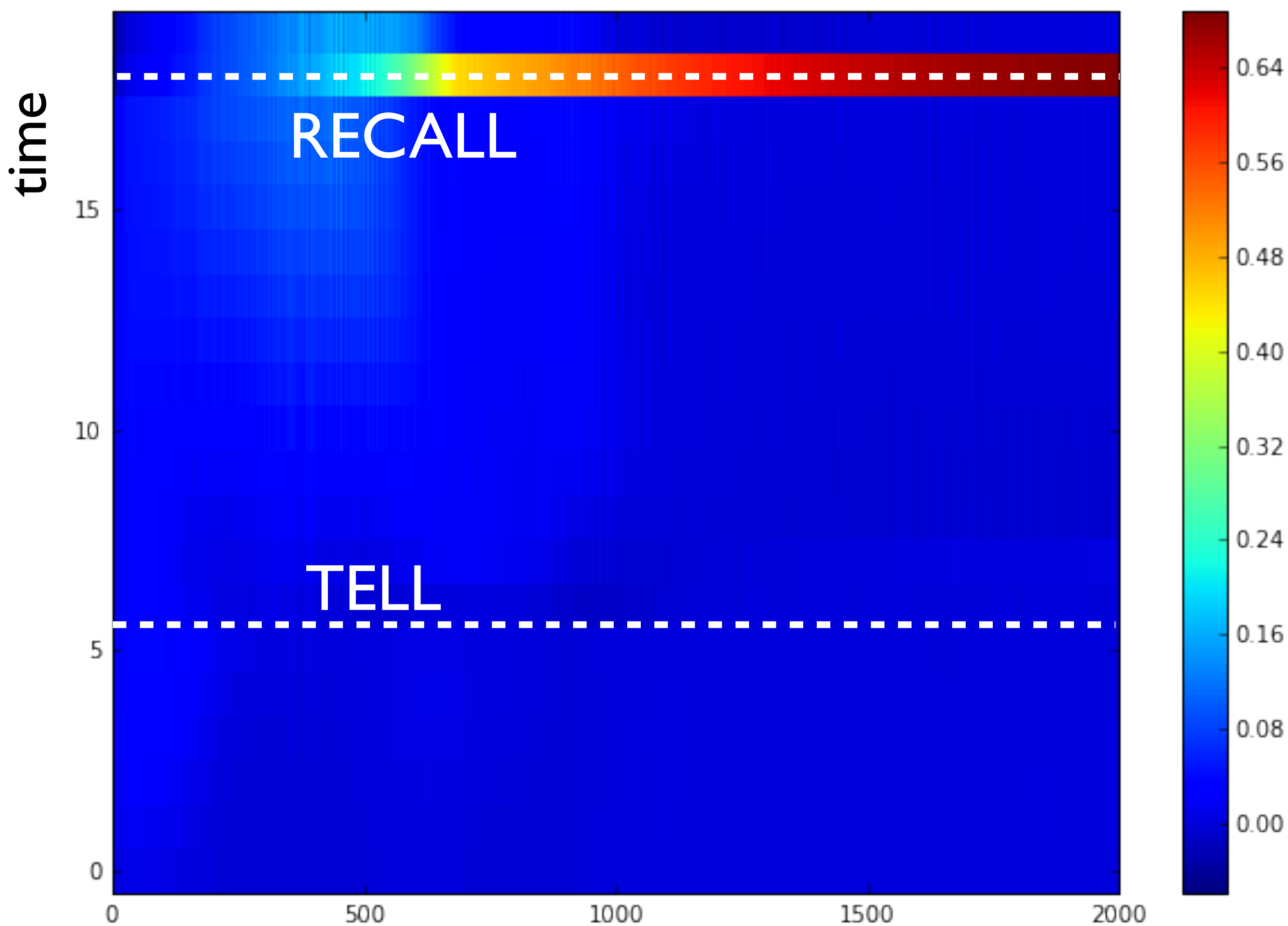


Example: A network that counts down

(see code on website)

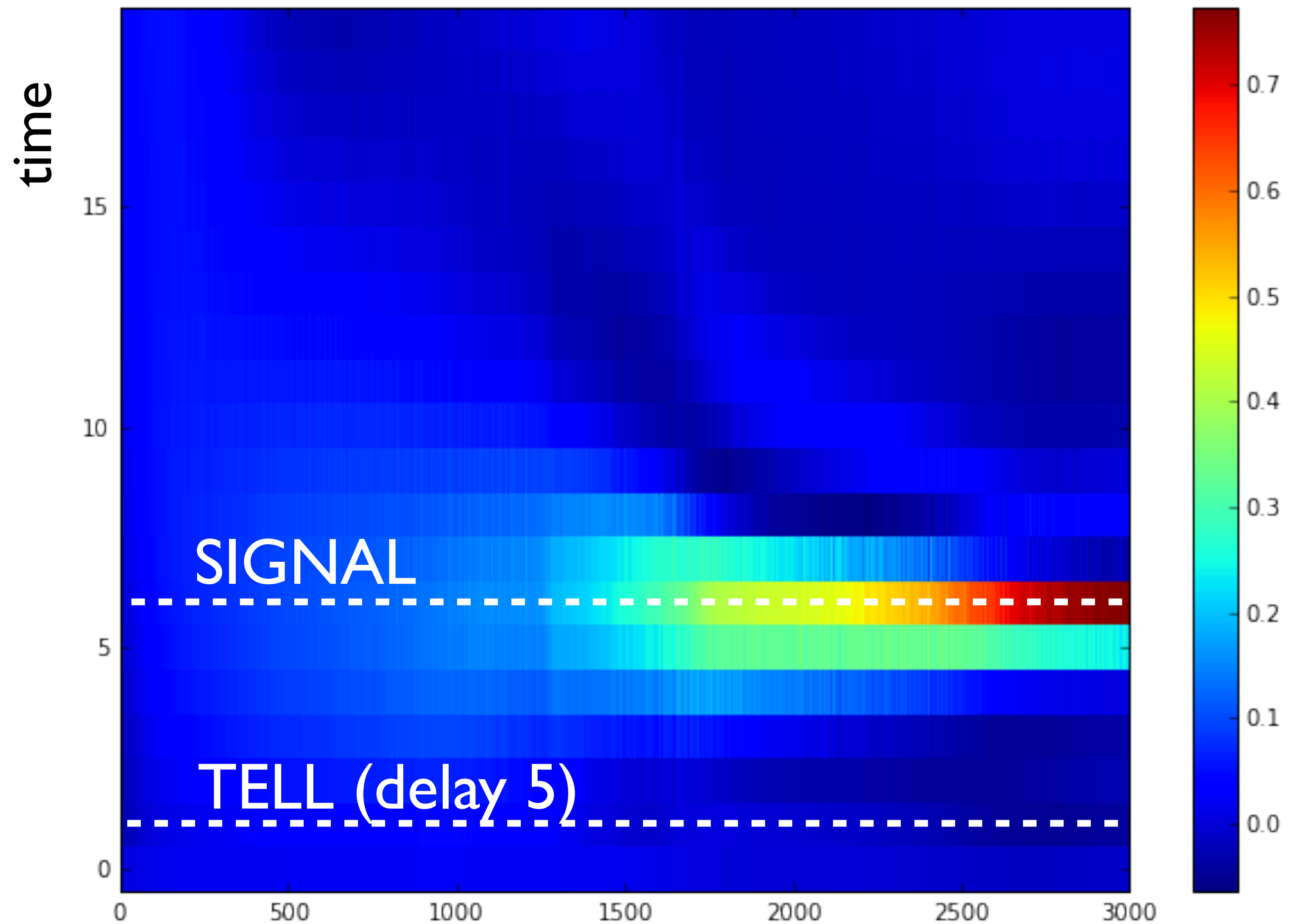


Output of the recall network, evolving during training (for a fixed input sequence)



Learning episode (batch of 20 for each episode)

Output of the countdown network, evolving during training (for a fixed input sequence)



Learning episode (batch of 20 for each episode)