ANCHORMEN
data activators

SEQUENCES

Raoul Grouls

11-JAN-21

MOTIVATION FOR RNNs

SEQUENCES

Often, our data is a sequence. Think of things like:

- Sentences (sequence of words)
- Music (sequence of tones)
- EEG (sequence of electric pulses)

RECURRENT NETWORKS

A major problem with sequences is, that the history has to be taken into account to understand the data.

“Ik krijg mijn geld van de bank”
“Ik denk na over de aanschaf van een nieuwe bank”.

If we only look at the word itself, the model (and neither a human) can’t figure out the meaning without the **context** of the sequence!

MAKING NEURAL NETWORKS RECURRENT

SIMPLE RNN

Let's start with a simple neural network architecture, with one hidden layer.

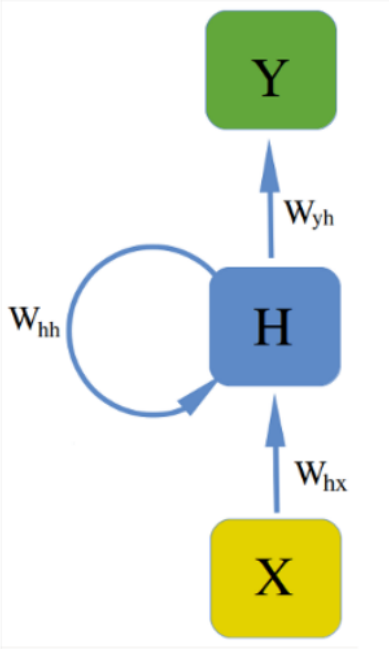
We can recognize the set of weights W_{hx} that transforms the X into the hidden state, and an additional set of weights W_{yh} that transforms the hidden state into the y.

How can we incorporate the history of the sequence into this?

SIMPLE RNN

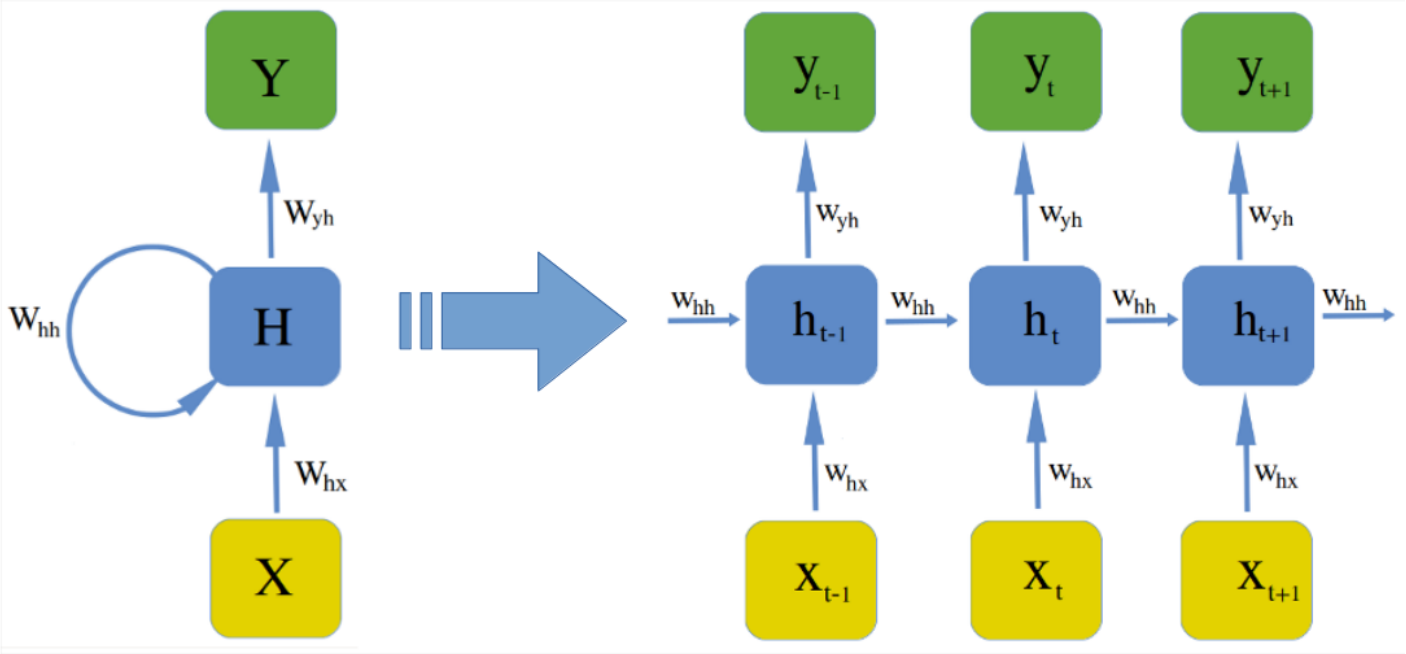
We can add a **loop**. This is why it is called a **recurrent** neural network.
We will define a 'hidden' state, that can be passed onto the next iteration.
We will try to learn weights W_{hh} that define how the hidden state should be passed on.

- do you understand why we need to learn weights for this?
- What would happen without the weights?



SIMPLE RNN

We will reuse the same weights for every iteration. If we unpack the network over time, we could visualize it like this:



While the math might look impressive at first, with all the subscripts, we can recognize the same basic linear formula we have been using over and over again since lesson 1:

$$y = ax + b$$

or, equivalent, but with a matrix notation:

$$y = WX + b$$

The subscripts like W_{hh} and W_{hx} to point to the fact that these are different sets of weights, but in essence there is nothing new.

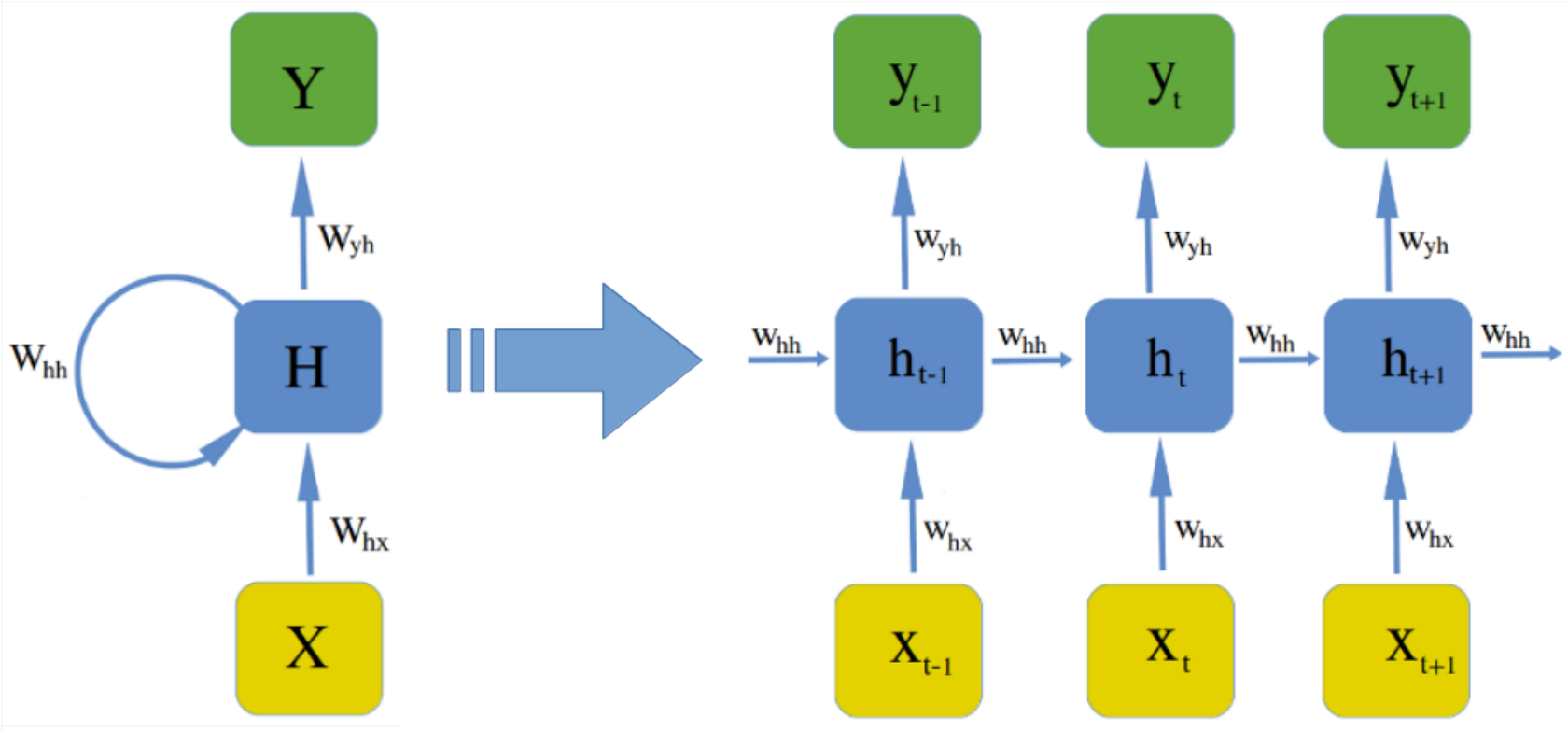
We will wrap the linear formula's in an activation function σ (for which have been using functions like a sigmoid, tanh or relu), which give us

$$y = \sigma(WX + b)$$

The formula for the hidden state can be written down like this:

$$h_t = \sigma(W_{hh}h_{t-1} + W_{hx}X_t + b_h)$$

The only difference here with our familiar notation seems to be that we sum the output of the hidden state h_{t-1} and the input X_t.



But this difference is just superficial. If we have the formula:

$$y = a_1 x_1 + a_2 x_2 + b$$

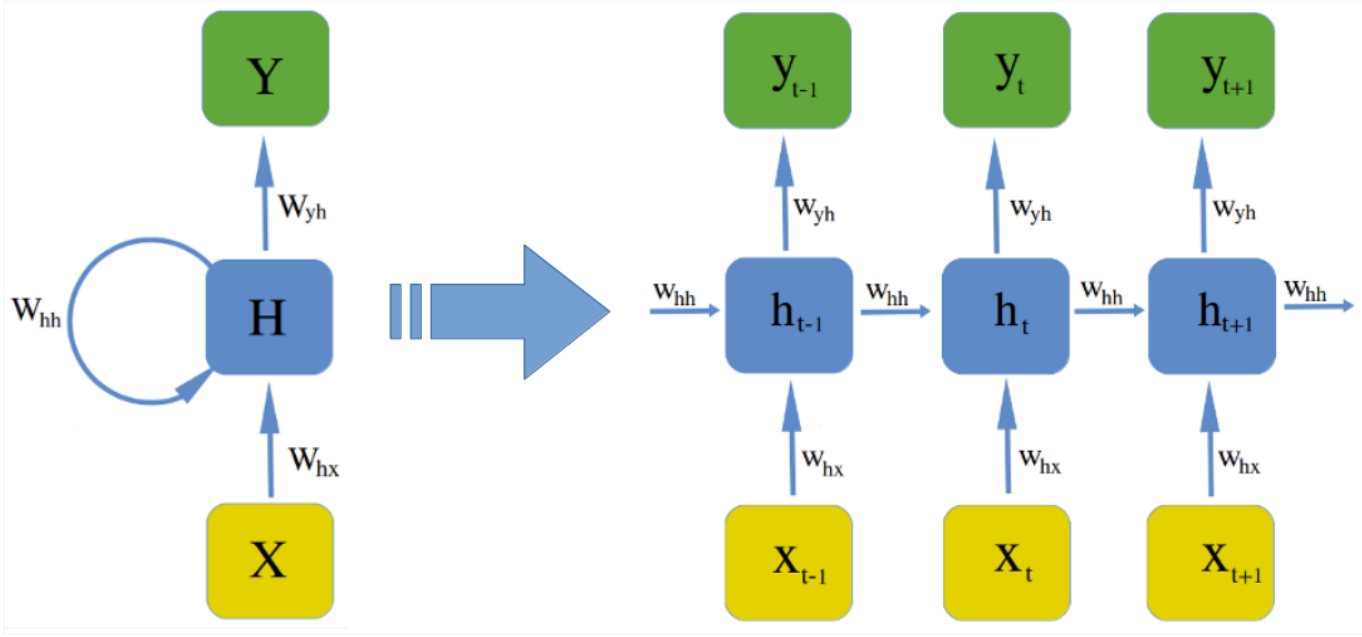
We can simplify that with a matrix notation:

$$y = AX + b$$

With $A = [a_1 , a_2]$ and $X = [x_1, x_2]$

Thus, we can

- concatenate the hidden state of the past timestep h_{t-1} and the current input X_t together as $[h_{t-1}, X_t]$
- stack both weights into one object $W_h = [W_{hh}, W_{hx}]$



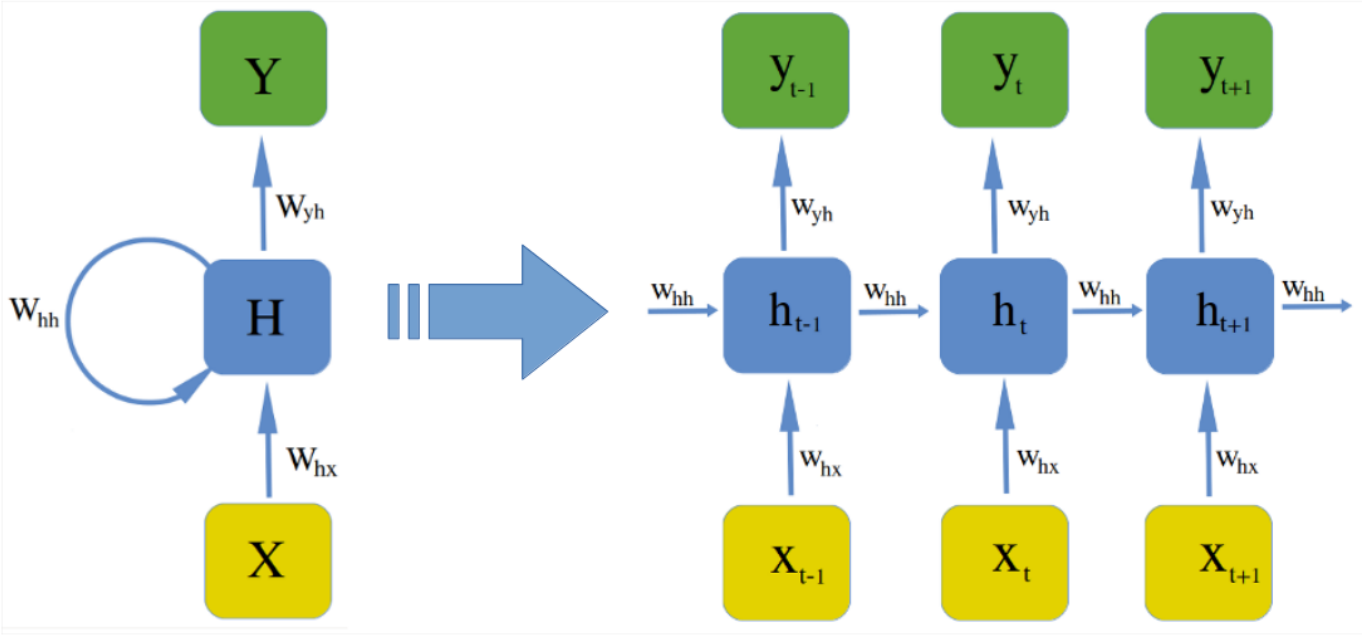
Instead of

$$h_t = \sigma(W_{hh}h_{t-1} + W_{hx}X_t + b_h)$$

We often write the equivalent,

$$h_t = \sigma(W_h[h_{t-1}, X_t] + b_h)$$

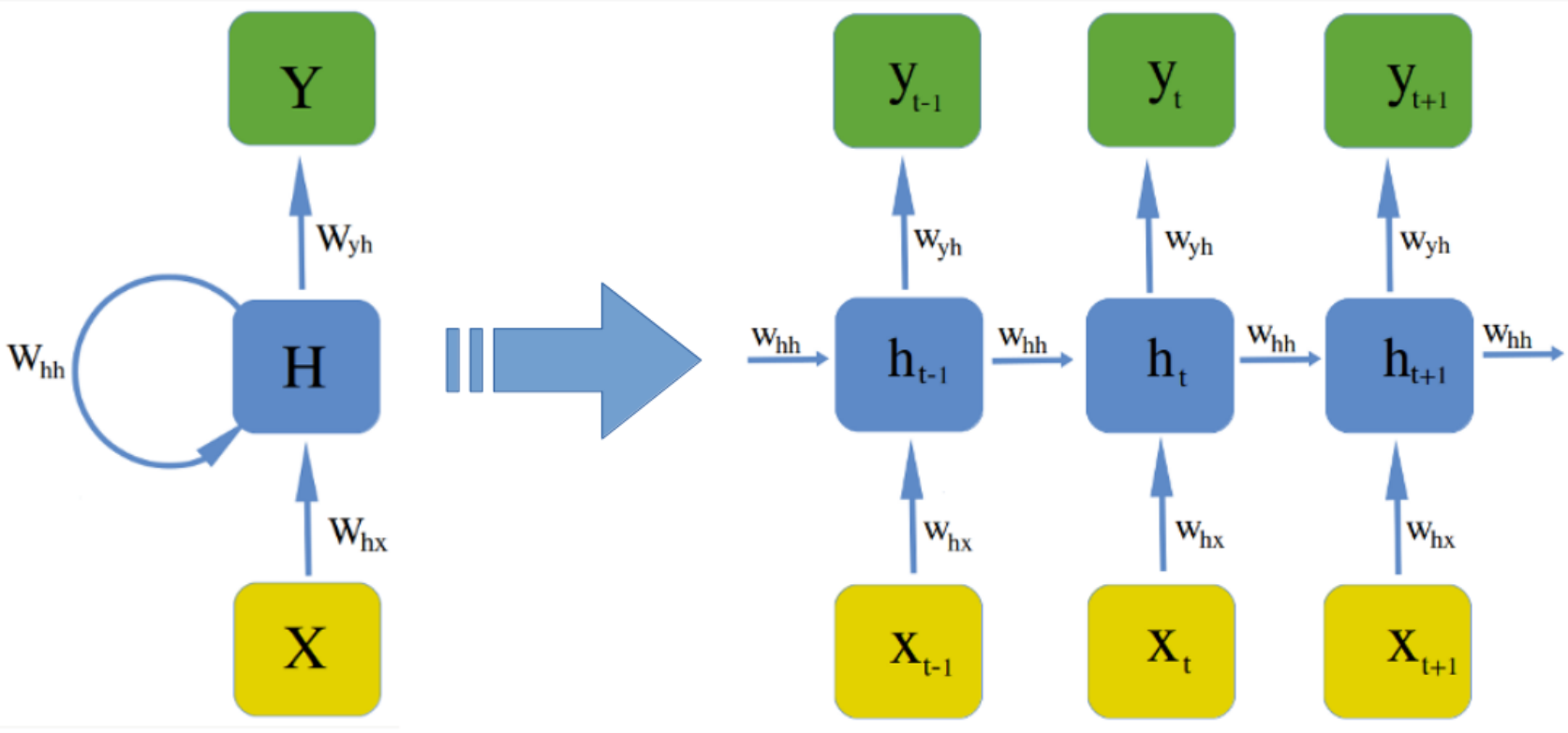
We can think of the hidden state as an **additional set of features**.



For the output, we have the same shape $y = WX + b$ that we have been using since lesson 1.

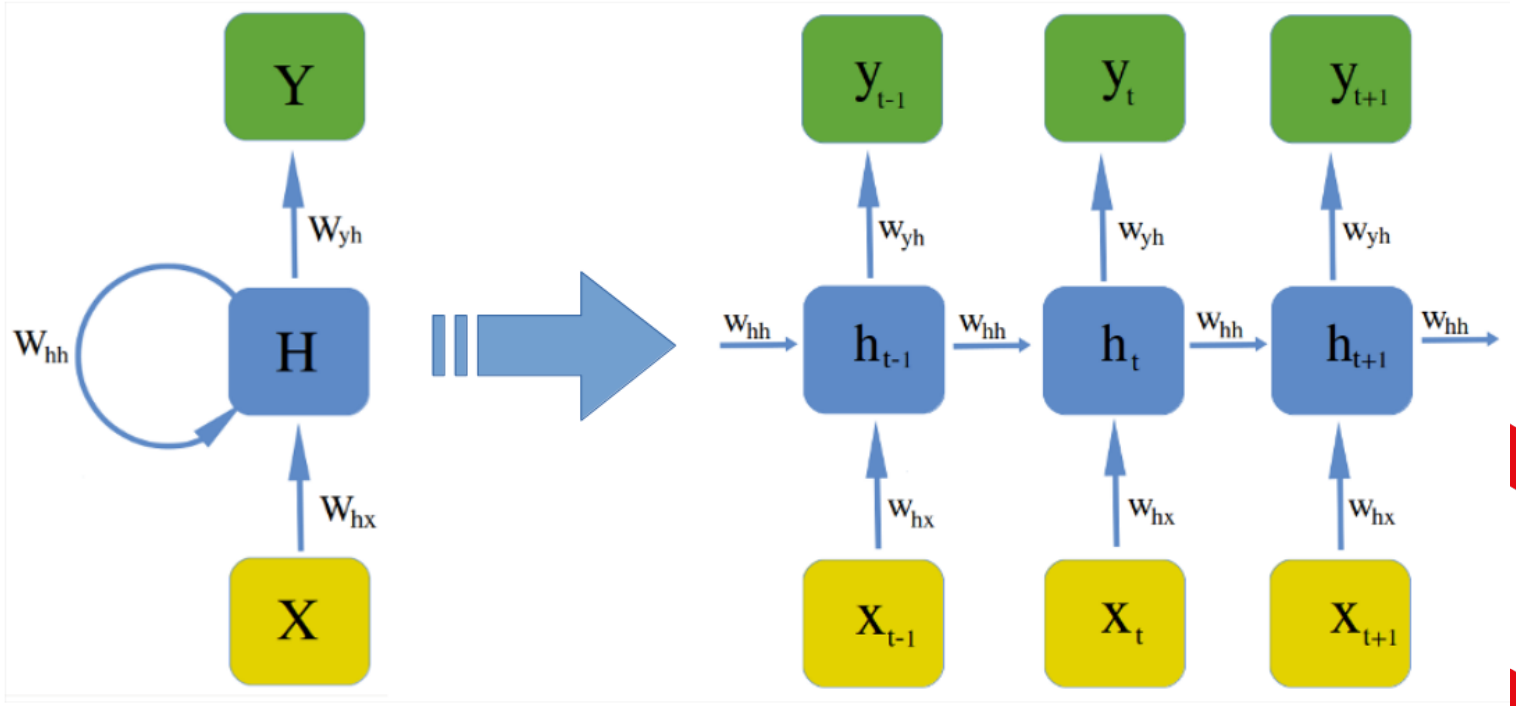
$$h_t = \sigma(W_h[h_{t-1}, X_t] + b_h)$$

$$y_t = \sigma(W_{yh}h_t + b_y)$$



We can summarize this intuitively as:

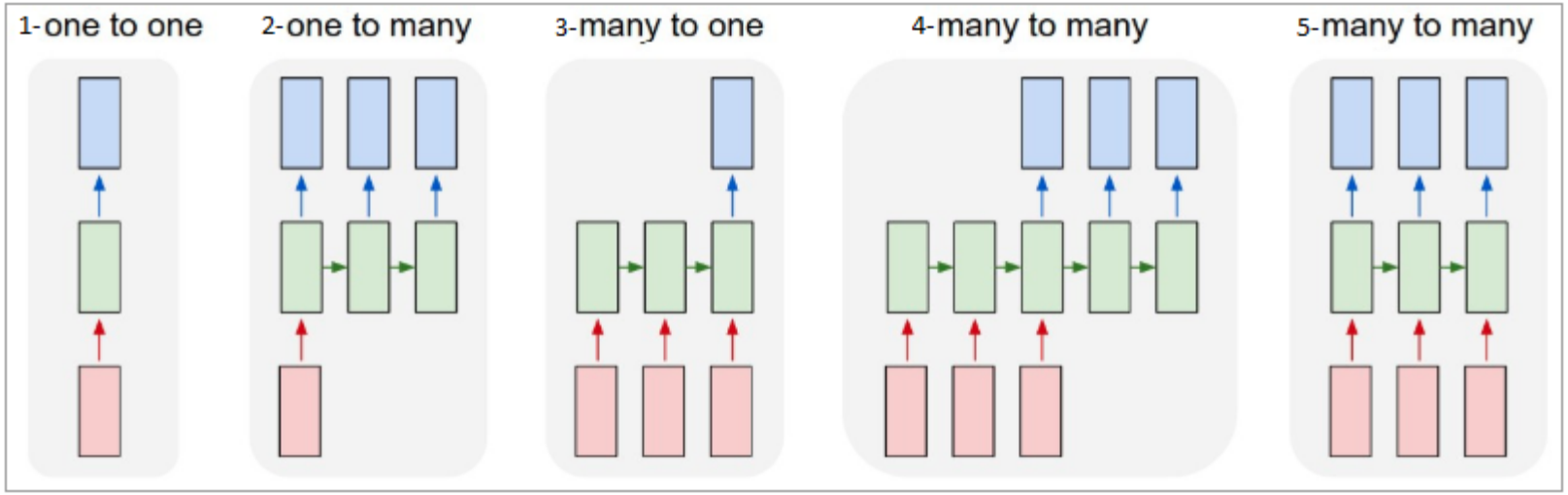
- We use a familiar dense network with one hidden layer
- The math is similiar to what we have been using for all neural networks: a linear function with an activation.
- To make it recurrent, we add the hidden state of the past as features (‘context’) to the input.
- This way, if we need the model to understand the meaning of ‘bank’, we have access to the context of the sequence to determine the proper meaning.



Shape of RNNs

Input of a sequence does not has to be equal to the length of the output.

- Text generation (one-to-many)
- Sentiment classification (many-to-one)
- Machine translation (many-to-many, encoder-decoder (type 4))
- Named entity recognition (many-to-many (type 5))



ADDING MEMORY

MOTIVATION: VANISHING GRADIENTS

Long term dependencies:

The man, which ate the... while… and… , **was** full

The men, which ate the…. while…. and…, **were** full

We get a vanishing gradients problem.

VANISHING GRADIENTS

Long term dependencies:

The man, which ate the... while… and… , **was** full

The men, which ate the…. while…. and…, **were** full

It is difficult for a simple RNN to capture very long term dependencies.

Solution: add **memory** and **forgetgates**.

GRU (SIMPLIFIED)

We will have to define two different states to create 'memory'.

C_t Memory Cell (similar to h_t from the RNN)

$\tilde{C}_t$ Candidate replacement for C_t: $\tanh(W_c[C_{t-1}, X_t] + b_c)$

We have an input (the hidden state) and we will transform that input as we have been doing before with a variantion on $y = \sigma(WX + b)$.

GRU (SIMPLIFIED)

The main idea we want to add to this, is:

Can we find a way to either

- *ignore* the new replacement $\tilde{C}_t$ (and keep remembering the past C_t)
- *Forget* the past C_t (and fully take the replacement $\tilde{C}_t$)
- *Something in between* (where we find a ratio between forgetting and remembering)

GRU (SIMPLIFIED)

We can achive this with a *forget gate* Γ_u (which is the greek capital gamma):

Γ_u Gate: $\sigma(W_u[C_{t-1}, X_t] + b_u)$

This gate will be an matrix of numbers between 0 and 1, because we will use a sigmoid activation.

0 will mean: keep the past, ignore the new state
1 will mean: forget the past, take the new state

Any number in between will mix the two extremes.

GRU (SIMPLIFIED)

C_t Memory Cell (similar to h_t from the RNN)

$\tilde{C}_t$ Candidate replacement for C_t: $\tanh(W_c[C_{t-1}, X_t] + b_c)$

Γ_u Gate: $\sigma(W_u[C_{t-1}, X_t] + b_u)$

$$C_t = \Gamma_u * \tilde{C}_t + (1 - \Gamma_u) * C_{t-1}$$

The gate has a sigmoid activation, so has a value between 0 and 1.
It acts like a mask or filter.

GRU (SIMPLIFIED)

C_t Memory Cell (similar to a_t from the RNN)

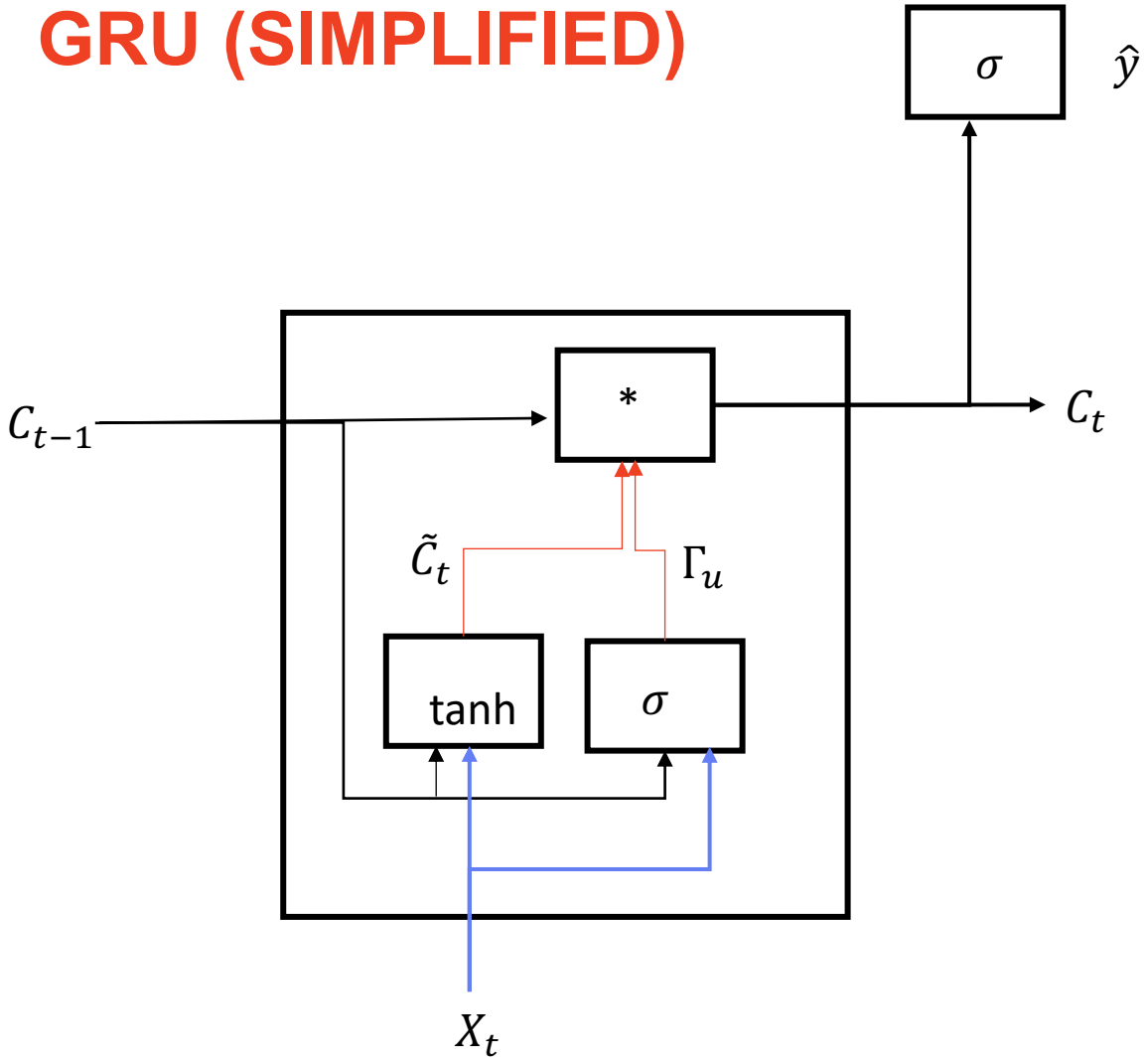
$\tilde{C}_t$ Candidate replacement for C_t: $\tanh(W_c[C_{t-1}, X_t] + b_c)$

Γ_u Gate: $\sigma(W_u[C_{t-1}, X_t] + b_u)$

$$C_t = \Gamma_u * \tilde{C}_t + (1 - \Gamma_u) * C_{t-1}$$

If $\Gamma_u = 1$, we take the new candidate. If $\Gamma_u = 0$, we don't update the old value.

GRU (SIMPLIFIED)



C_{t-1}	Memory Cell (similar to a_t from the RNN)
$\tilde{C}_t$	Candidate replacement for C_t: $\tanh(W_c[C_{t-1}, X_t] + b_c)$
Γ_u	Gate: $\sigma(W_u[C_{t-1}, X_t] + b_u)$
C_t	$\Gamma_u * \tilde{C}_t + (1 - \Gamma_u) * C_{t-1}$

GRU (FULL)

The full GRU architecture has two gates:

C_t Memory Cell (similar to a_t from the RNN)

$\tilde{C}_t$ Candidate replacement for C_t: $\tanh(W_c[\Gamma_r * C_{t-1}, X_t] + b_c)$

Γ_u Update gate: $\sigma(W_u[C_{t-1}, X_t] + b_u)$

Γ_r Relevance gate: $\sigma(W_r[C_{t-1}, X_t] + b_r)$

$$C_t = \Gamma_u * \tilde{C}_t + (1 - \Gamma_u) * C_{t-1}$$

LSTM

In time, the LSTM architecture came before the GRU.

The GRU was intended to be a simplification of the even more complex LSTM architecture, that uses three gates.

The GRU reuses one gate with the formula:

$$C_t = \Gamma_u * \tilde{C}_t + (1 - \Gamma_u) * C_{t-1}$$

The original LSTM architecture used two separate weights here, adding more complexity:

$$C_t = \Gamma_u * \tilde{C}_t + \Gamma_f * C_{t-1}$$

LSTM

GRU

$\tilde{C}_t$ $\tanh(W_c[\Gamma_r * C_{t-1}, X_t] + b_c)$

Γ_u Update gate: $\sigma(W_u[C_{t-1}, X_t] + b_u)$

Γ_r Relevance gate: $\sigma(W_r[C_{t-1}, X_t] + b_r)$

$C_t = \Gamma_u * \tilde{C}_t + (1 - \Gamma_u) * C_{t-1}$

$h_t = C_t$

LSTM

$\tilde{C}_t$ $\tanh(W_c[h_{t-1}, X_t] + b_c)$

Γ_u Update gate: $\sigma(W_u[h_{t-1}, X_t] + b_u)$

Γ_f Forget gate: $\sigma(W_f[h_{t-1}, X_t] + b_f)$

Γ_o Output gate: $\sigma(W_o[h_{t-1}, X_t] + b_o)$

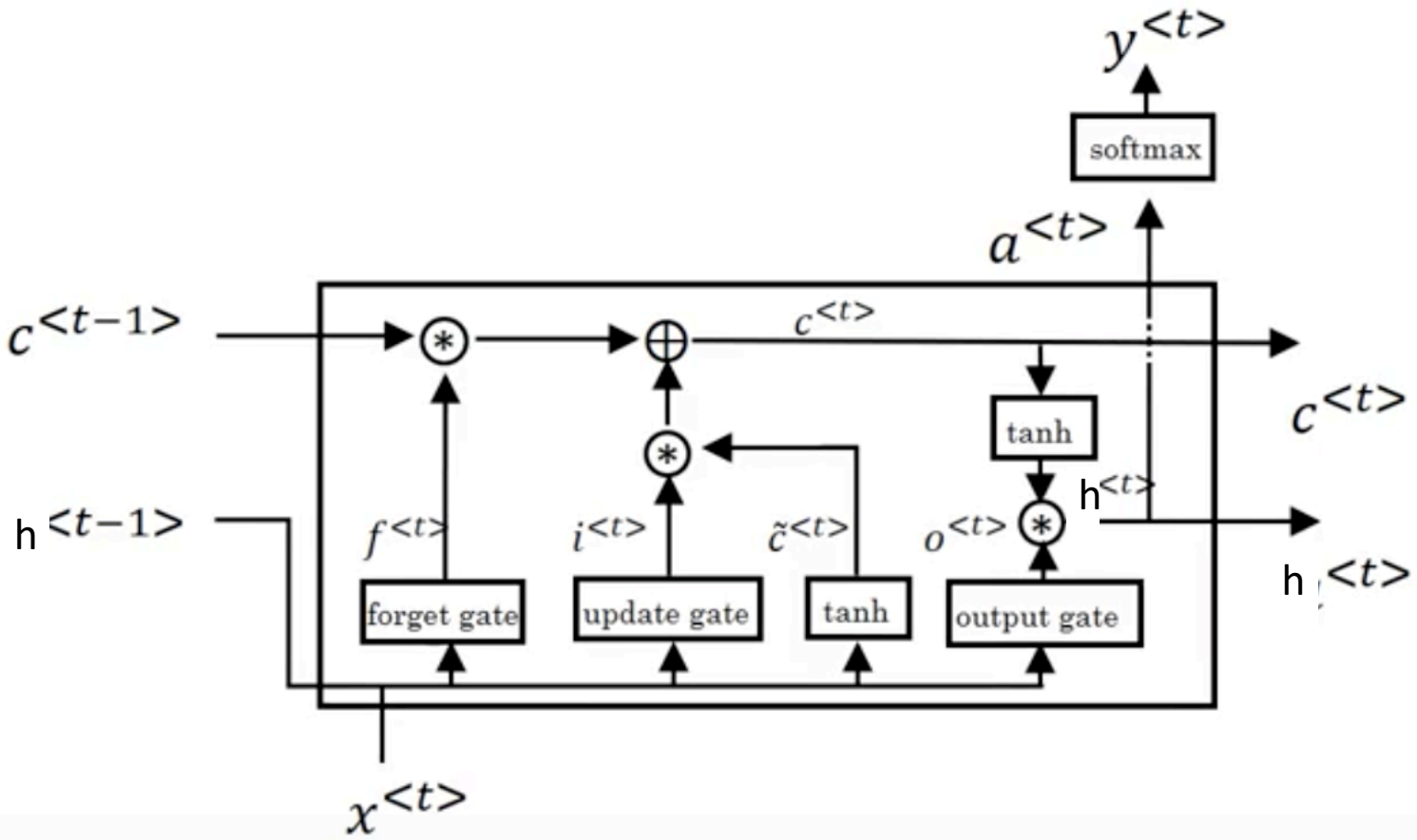
$C_t = \Gamma_u * \tilde{C}_t + \Gamma_f * C_{t-1}$

$h_t = \Gamma_o * C_t$

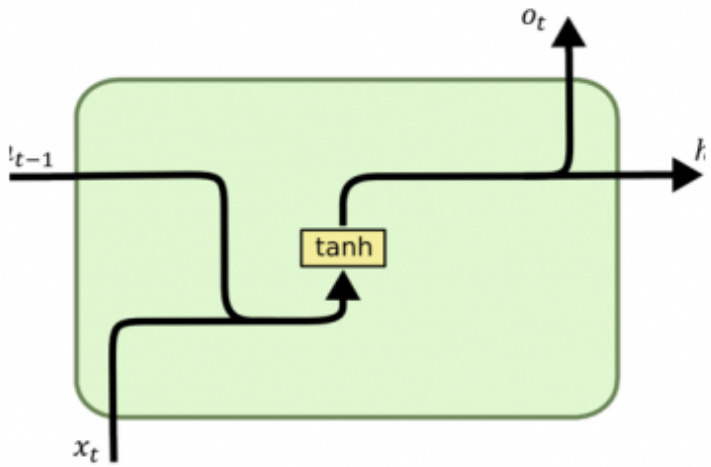
LSTM

$$C_t = \Gamma_u * \tilde{C}_t + \Gamma_f * C_{t-1}$$

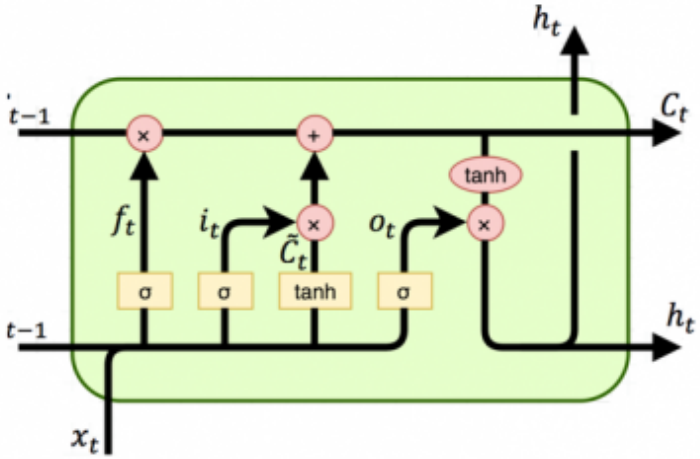
$$h_t = \Gamma_o * C_t$$



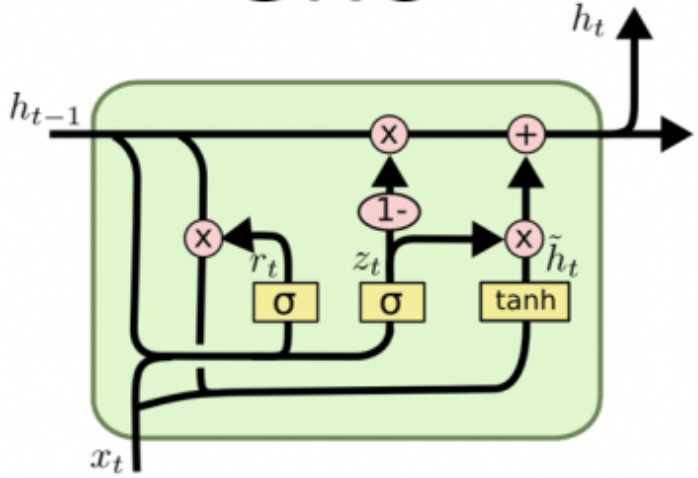
RNN


o_t
h_{t-1}
tanh
x_t

LSTM


h_t
C_t
$t-1$
f_t
i_t
$\tilde{C}_t$
o_t
σ
σ
tanh
σ
tanh
$t-1$
h_t
x_t

GRU


h_t
h_{t-1}
1-
r_t
z_t
$\tilde{h}_t$
σ
σ
tanh
x_t

SUMMARY

- All three architectures are recurrent
- The simple RNN does not has a mechanism for memory, but can suffer from vanishing gradients with long sequences.
- Both the GRU and LSTM have memory mechanisms, that can learn to keep something in memory for a long time, and forget it at a certain point in time.
- Because the GRU and LSTM are more complex, the will have much more weights to learn.

CONVOLUTIONS

1D CONVOLUTIONS

Timeseries with 1 feature

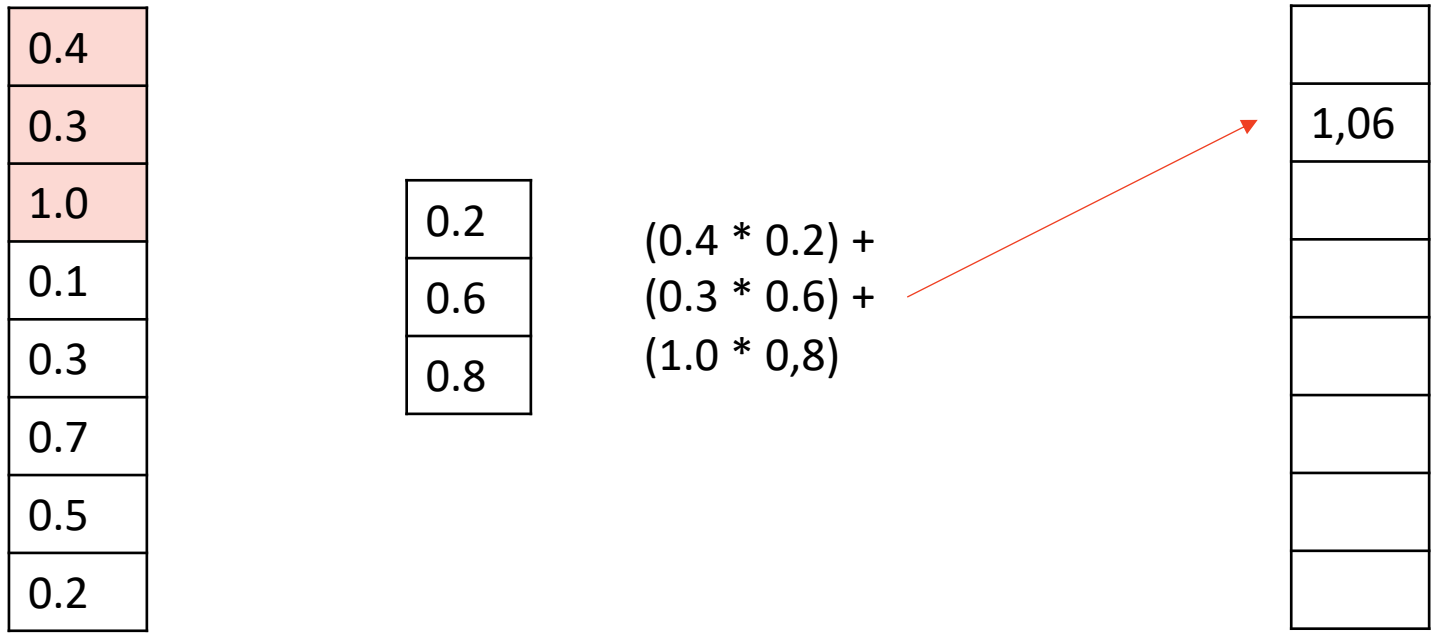
0.4
0.3
1.0
0.1
0.3
0.7
0.5
0.2

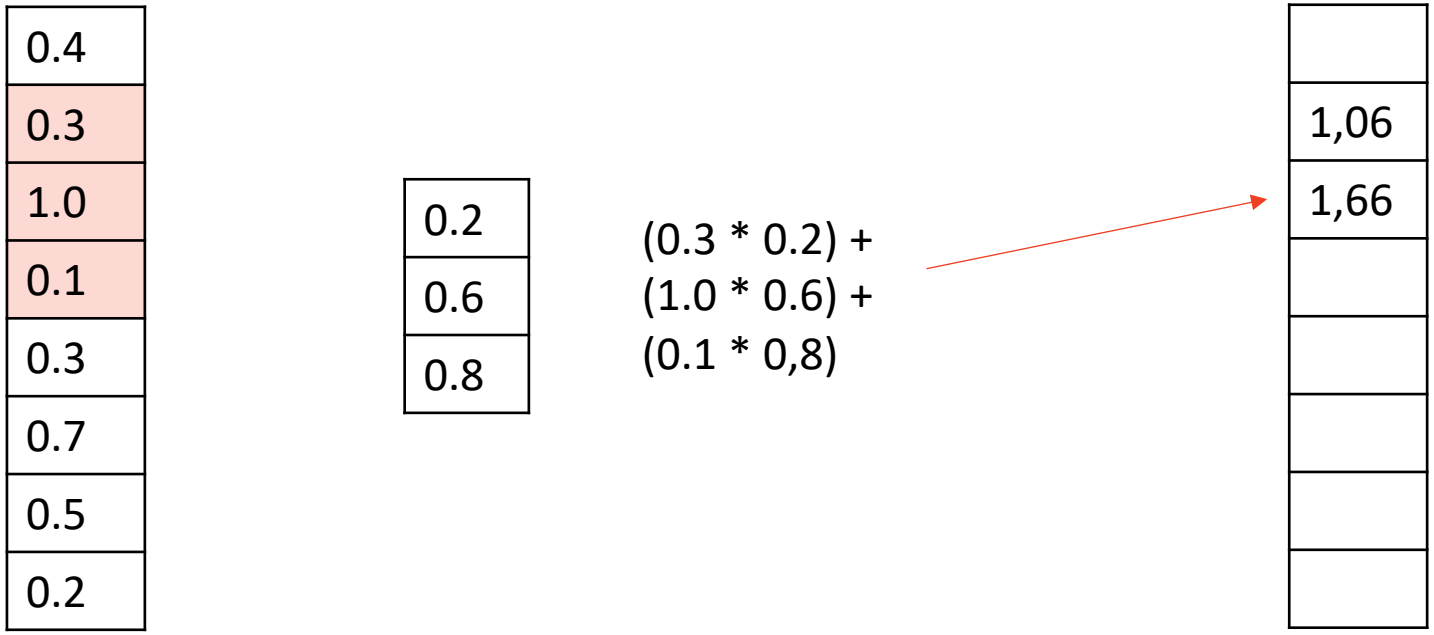
1D Kernel with size 3

0.2
0.6
0.8

In essence, this works the same as a 2D convolution.

We just miss a dimension.


0.4
0.3
1.0
0.1
0.3
0.7
0.5
0.2
0.2
0.6
0.8
(0.4 * 0.2) +
(0.3 * 0.6) +
(1.0 * 0,8)
1,06

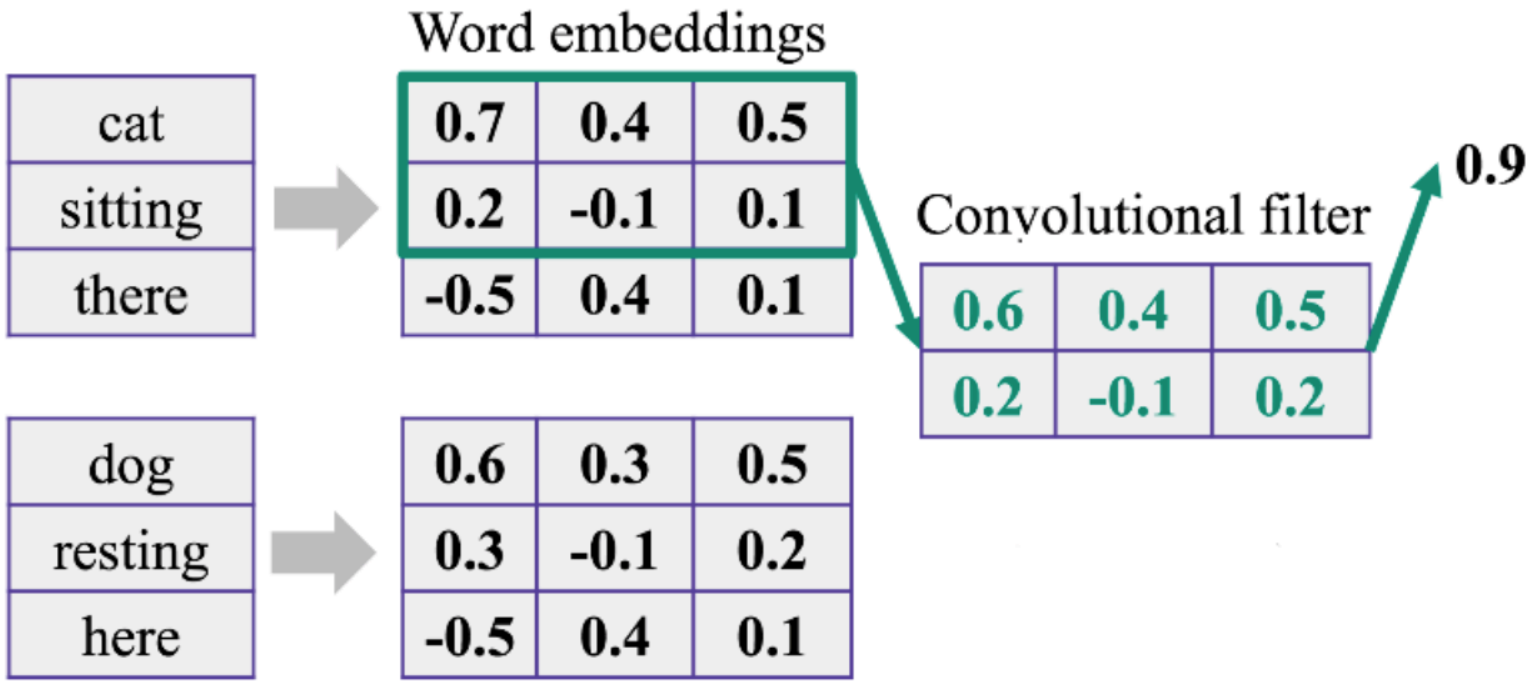

0.4
0.3
1.0
0.1
0.3
0.7
0.5
0.2
0.2
0.6
0.8
(0.3 * 0.2) +
(1.0 * 0.6) +
(0.1 * 0,8)
1,06
1,66

0.4
0.3
1.0
0.1
0.3
0.7
0.5
0.2

0.2
0.6
0.8

1.06
1.66
0.5
0.76
1.78
0,6

CONVOLUTIONS



This can be extended to timeseries with more features, eg 3 features in this example.

0,7*0,6+0,4*0,4+0,5*0,5+0,2*0,2+-0,1*-0,1+0,1*0,2 = 0,9

CONVOLUTIONS

Intuitive, you can think of convolutions in the same way we used it with images.

It is a way to extract features from a timeseries, where we don't specify the features ourselves but let the machine learn the features.

Examples of features are:

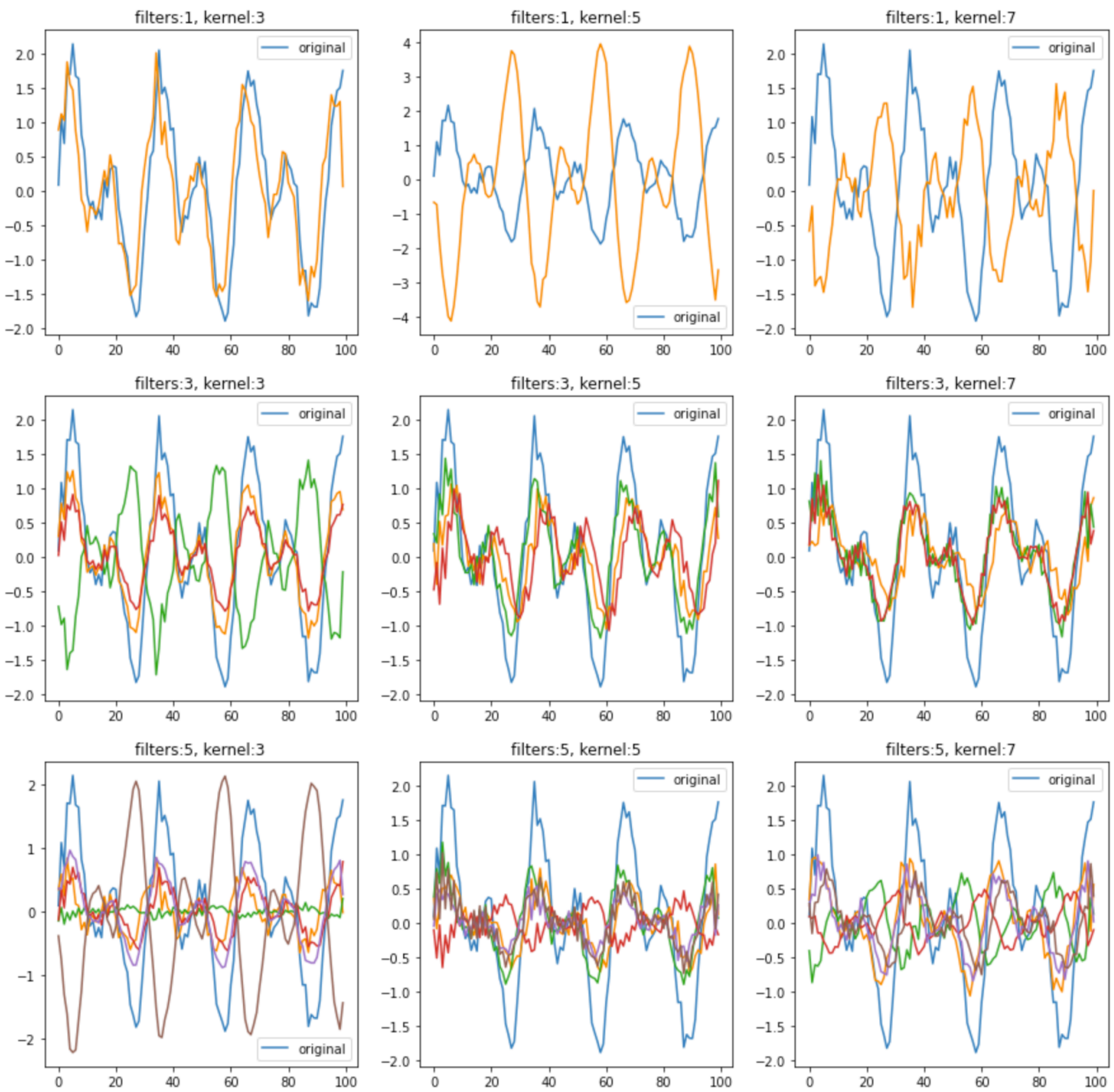
- Removing noise (eg, moving average)
- Detecting fast rising peaks
- Detecting sudden drops
- Etc.

CONVOLUTIONS

This is an example of random filters.

You can vary the amount of filters, and the size of the kernel.

Note how some filters increase small variations, while others smoothen the variations.



EVALUATION – CROSSVALIDATON

We saw this before in the lesson on crossvalidation.

Note how we can't use standard crossvalidation!

We will have to create what is called a 'windowed' dataset, where the test data never 'leaks' future information.