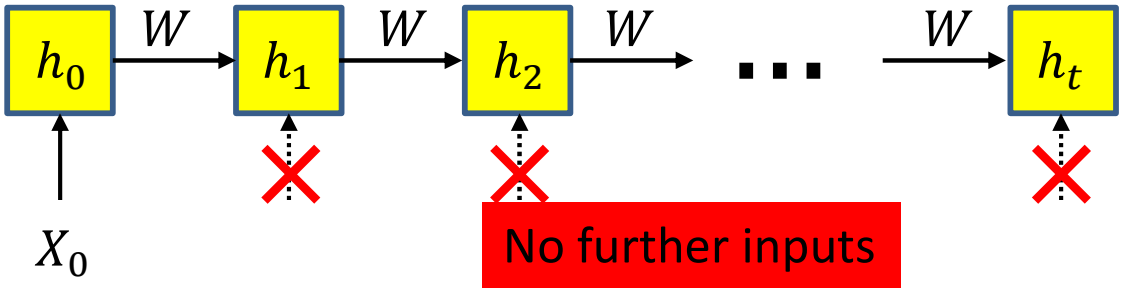
Deep Learning
Recurrent Networks: Modelling Language

Recap: Story so far

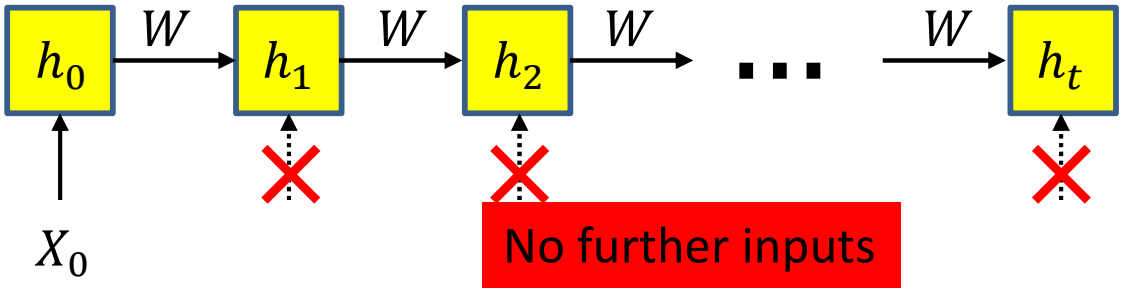
- Vanilla recurrent nets are poor at memorization
  - The memory behavior is dependent primarily on the recurrent weights and hidden state activations
- Recurrent (and Deep) networks also suffer from a "vanishing or exploding gradient" problem
  - The gradient of the error at the output gets concentrated into a small number of parameters in the earlier layers, and goes to zero for others
  - This too is a consequence of recurrent weights and hidden-state acivations

Recap: The vanishing memory – the weights matrix edition



- Response to a single input at $t = 0$, with recurrent weight matrix W and ***linear activation*** $h_t = W h_{t-1}$

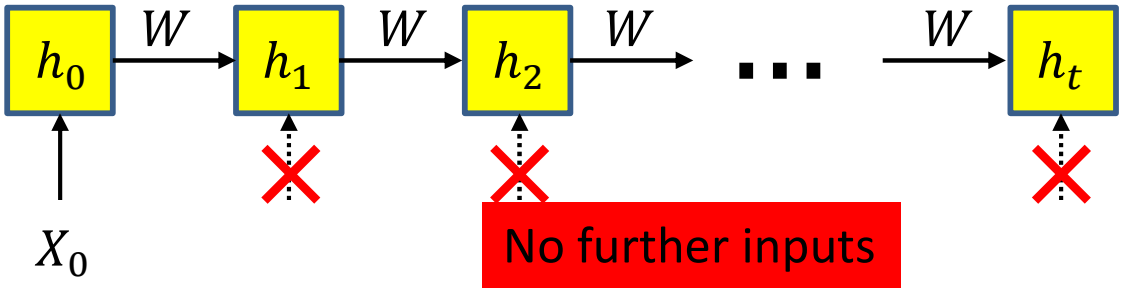
Recap: The vanishing memory – the weights matrix edition



- Response to a single input at $t = 0$, with recurrent weight matrix W and ***linear activation*** $h_t = W h_{t-1}$

$$h_t = W^t h_0$$

Recap: The vanishing memory – the weights matrix edition

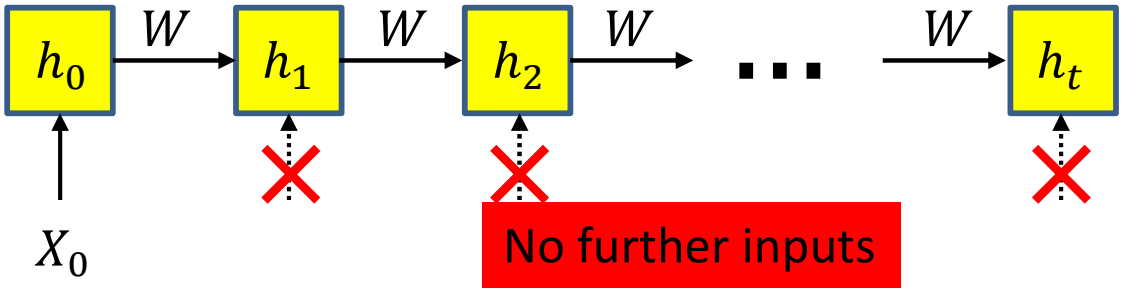


- Response to a single input at $t = 0$, with recurrent weight matrix W and ***linear activation*** $h_t = W h_{t-1}$

$$h_t = W^t h_0$$

  - Eigen decomposing $W = U \Lambda U^{-1}$

Recap: The vanishing memory – the weights matrix edition



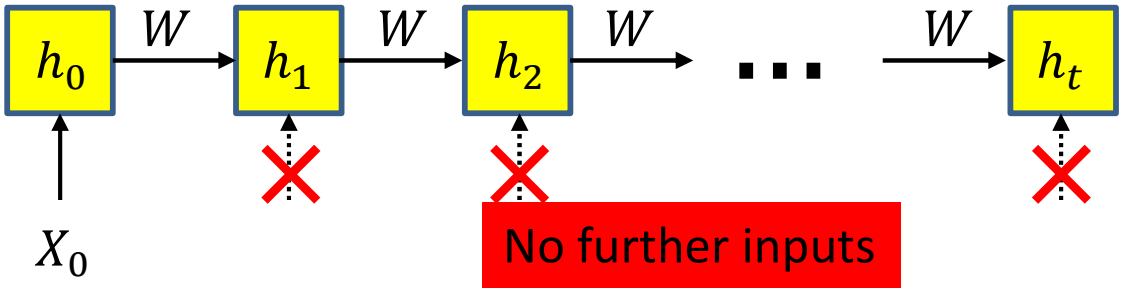
- Response to a single input at $t = 0$, with recurrent weight matrix W and ***linear activation*** $h_t = Wh_{t-1}$

$$h_t = W^t h_0$$

  - Eigen decomposing $W = U\Lambda U^{-1}$

$$h_0 = a_1 u_1 + a_2 u_2 + a_3 u_3 + \ldots$$

Recap: The vanishing memory – the weights matrix edition



- Response to a single input at $t = 0$, with recurrent weight matrix W and ***linear activation*** $h_t = Wh_{t-1}$

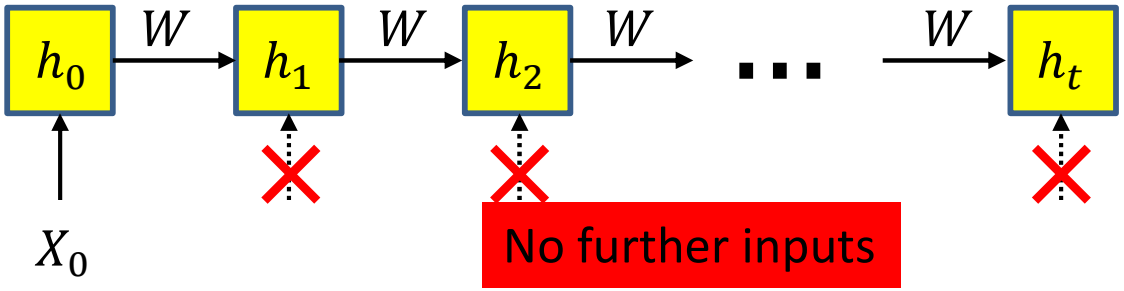
$$h_t = W^t h_0$$

  - Eigen decomposing $W = U\Lambda U^{-1}$

$$h_0 = a_1 u_1 + a_2 u_2 + a_3 u_3 + \ldots$$

$$h_t = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots$$

Recap: The vanishing memory – the weights matrix edition



- Response to a single input at $t = 0$, with recurrent weight matrix W and ***linear activation*** $h_t = Wh_{t-1}$

$$h_t = W^t h_0$$

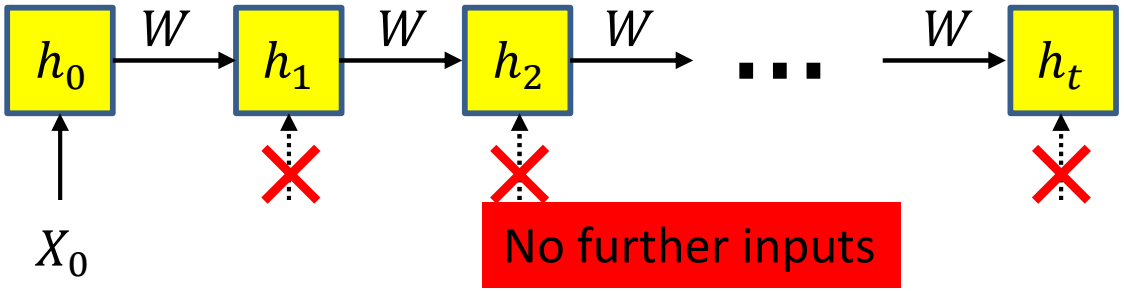
  - Eigen decomposing $W = U\Lambda U^{-1}$

$$h_0 = a_1 u_1 + a_2 u_2 + a_3 u_3 + \ldots$$

$$h_t = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots$$

$$= \lambda_1^t \left(a_1 u_1 + a_2 \left(\frac{\lambda_2}{\lambda_1}\right)^t u_2 + a_3 \left(\frac{\lambda_3}{\lambda_1}\right)^t u_3 + \ldots \right)$$

Recap: The vanishing memory – the weights matrix edition



- Response to a single input at $t = 0$, with recurrent weight matrix W and ***linear activation*** $h_t = Wh_{t-1}$

$$h_t = W^t h_0$$

  - Eigen decomposing $W = U\Lambda U^{-1}$

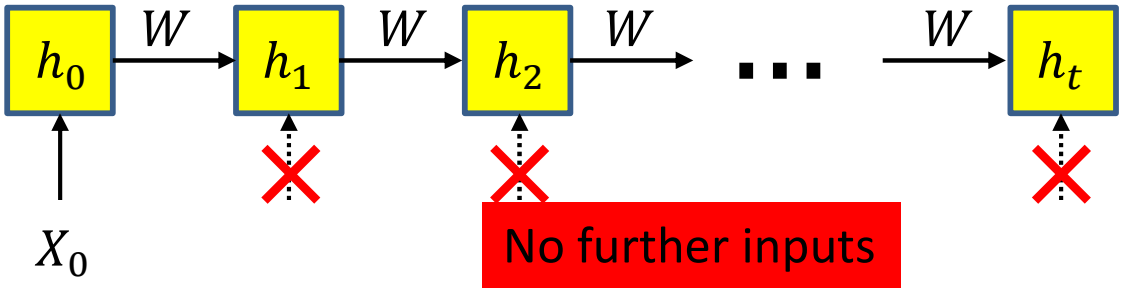
$$h_0 = a_1u_1 + a_2u_2 + a_3u_3 + \ldots$$

$$h_t = a_1\lambda_1^t u_1 + a_2\lambda_2^t u_2 + \cdots$$

$$= \lambda_1^t \left(a_1u_1 + a_2\left(\frac{\lambda_2}{\lambda_1}\right)^t u_2 + a_3\left(\frac{\lambda_3}{\lambda_1}\right)^t u_3 + \ldots \right)$$

$$h_t \approx \lambda_1^t a_1 u_1$$

Recap: The vanishing memory – the weights matrix edition



- Response to a single input at $t = 0$, with recurrent weight matrix W an

The long-term memory only depends on the largest Eigen value

In the worst case it grows or shrinks as λ_1^t

  - Eigen decomposing $W = U\Lambda U^{-1}$

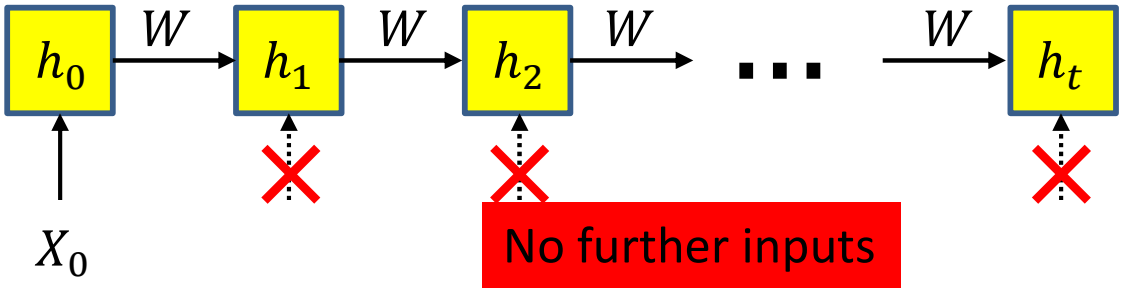
$$h_0 = a_1u_1 + a_2u_2 + a_3u_3 + \ldots$$

$$h_t = a_1\lambda_1^t u_1 + a_2\lambda_2^t u_2 + \cdots$$

$$= \lambda_1^t \left(a_1u_1 + a_2\left(\frac{\lambda_2}{\lambda_1}\right)^t u_2 + a_3\left(\frac{\lambda_3}{\lambda_1}\right)^t u_3 + \ldots \right)$$

$$h_t \approx \lambda_1^t a_1 u_1$$

Recap: The vanishing memory – the weights matrix edition



- Response to a single input at $t = 0$, with recurrent weight matrix W an

The long-term memory only depends on the largest Eigen value

In the worst case it grows or shrinks as λ_1^t

  - Eige

If the largest Eigenvalue is greater than 1, it blows up

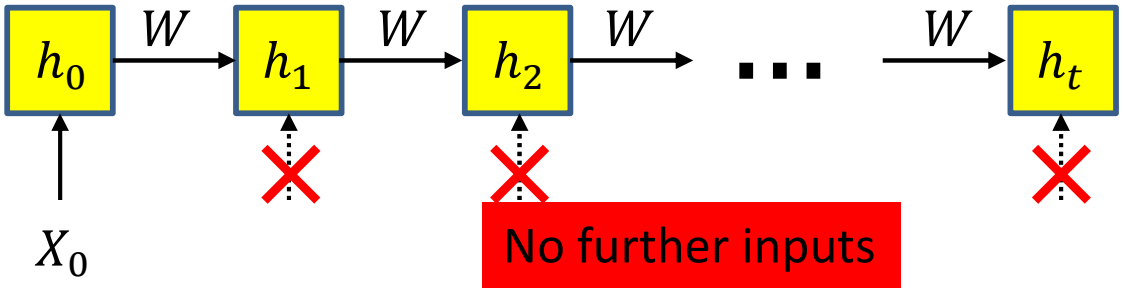
If the largest Eigenvalue is less than 1, it quickly shrinks to 0

Even if $\lambda_1 = 1$, it only "remembers" u_1

$$= \lambda_1^t \left(a_1 u_1 + a_2 \left(\frac{\lambda_2}{\lambda_1}\right)^t u_2 + a_3 \left(\frac{\lambda_3}{\lambda_1}\right)^t u_3 + \ldots \right)$$

$$h_t \approx \lambda_1^t a_1 u_1$$

Recap: The vanishing memory – the weights matrix edition



- Response to a single input at $t = 0$, with recurrent weight matrix W an
  - Eige

The long-term memory only depends on the largest Eigen value

In the worst case it grows or shrinks as λ_1^t

If the largest Eigenvalue is greater than 1, it blows up

If the largest Eigenvalue is less than 1, it quickly shrinks to 0

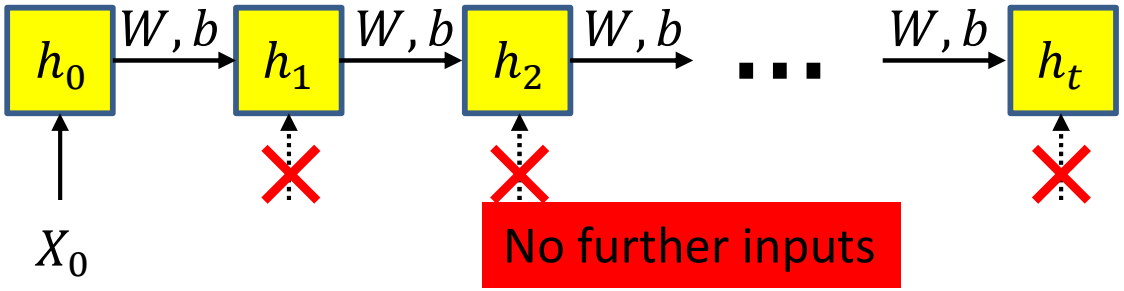
Even if $\lambda_1 = 1$, it only "remembers" u_1

What is "remembered" is a function of W and not particularly related to h_0 or the initial input x_0 (from which h_0 is computed)

$$h_t \approx \lambda_1^t a_1 u_1$$

FOR LINEAR RECURRENT ACTIVATION

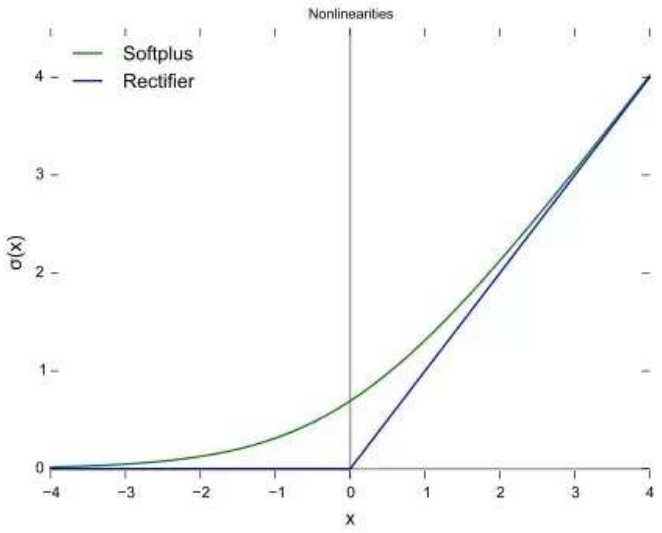
Recap: The vanishing memory – the activation edition



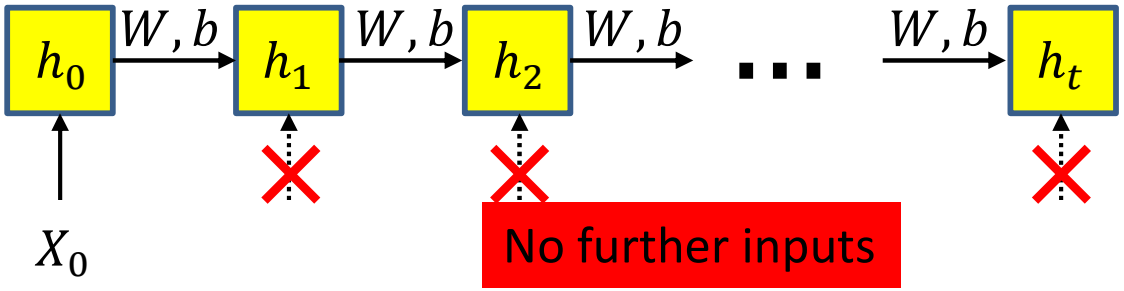
- Response to a single input at $t = 0$, with recurrent weight matrix W and ***nonlinear activation***

$$h_t = f(W h_{t-1} + b)$$

  - For ReLU/Softplus
    - For inputs in the linear region, h_t will grow or shrink with the largest Eigenvalue of W
    - For inputs in the negative region, the output will go to 0
    - Similar W dependent recurrence for ELU etc.

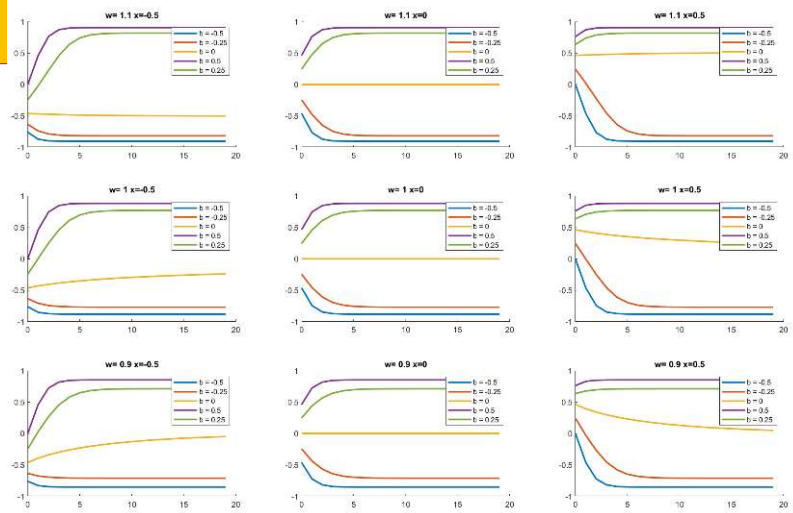


Recap: The vanishing memory – the activation edition



- Response to a single input at $t = 0$, with recurrent weight matrix W and ***nonlinear activation***

$$h_t = f(W h_{t-1} + b)$$

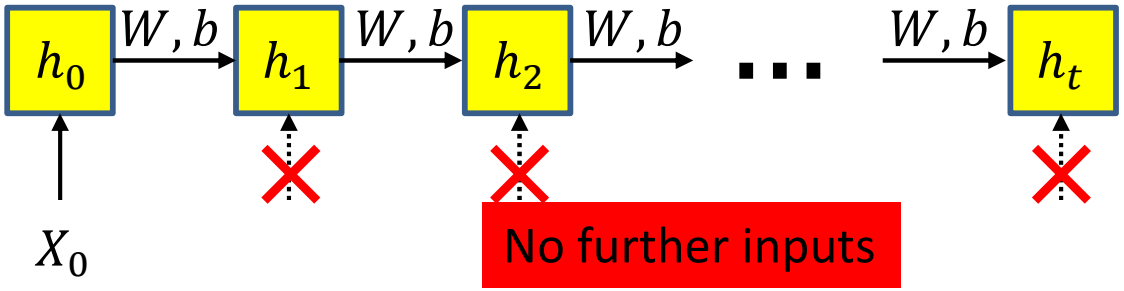


  - For general "saturating" activations the final "remembered" hidden state is just the solution to

$$h = f(Wh + b)$$

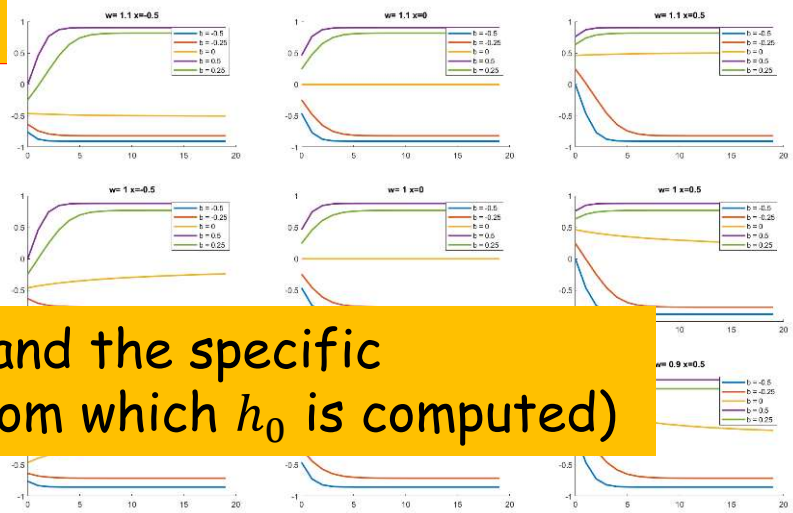
  - This is just a function of W and b and ignores h_0

Recap: The vanishing memory – the activation edition



- Response to a single input at $t = 0$, with recurrent weight matrix W and ***nonlinear activation***

$$h_t = f(W h_{t-1} + b)$$

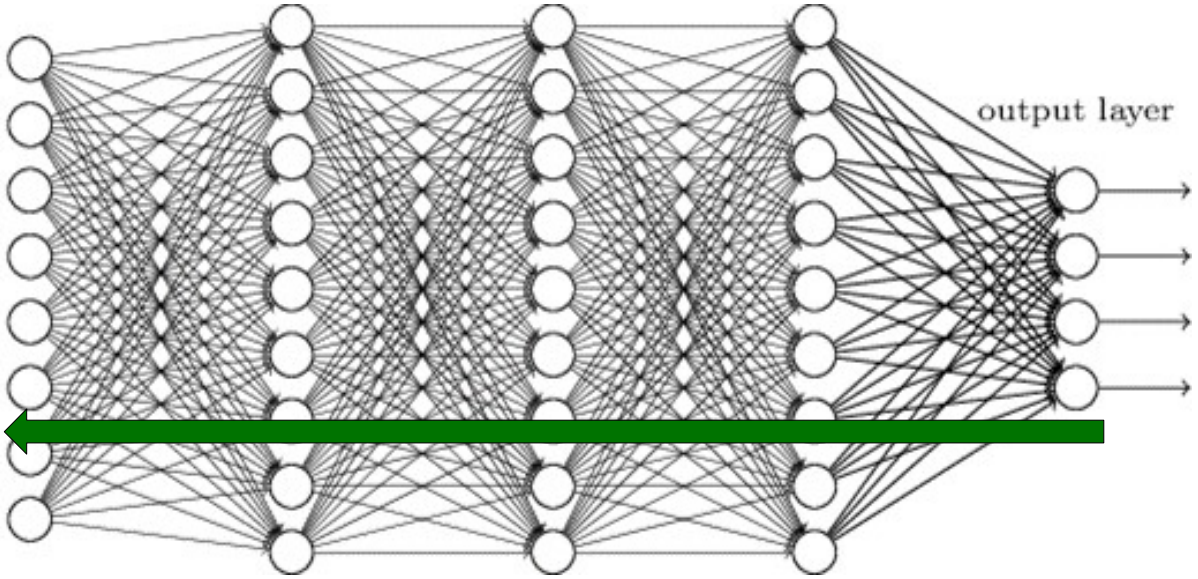


  - For general "saturating" activations the final "remembered" hidden state is just the solution to

What is "remembered" is a function of W, b and the specific activation, rather than the initial input x_0 (from which h_0 is computed)

  - This is just a function of W and b and ignores h_0

Recap 2: Exploding/Vanishing gradients in backpropagation



$$\nabla_{f_k} Div = \nabla D . \nabla f_N . W_N . \nabla f_{N-1} . W_{N-1} \ldots \nabla f_{k+1} W_{k+1}$$

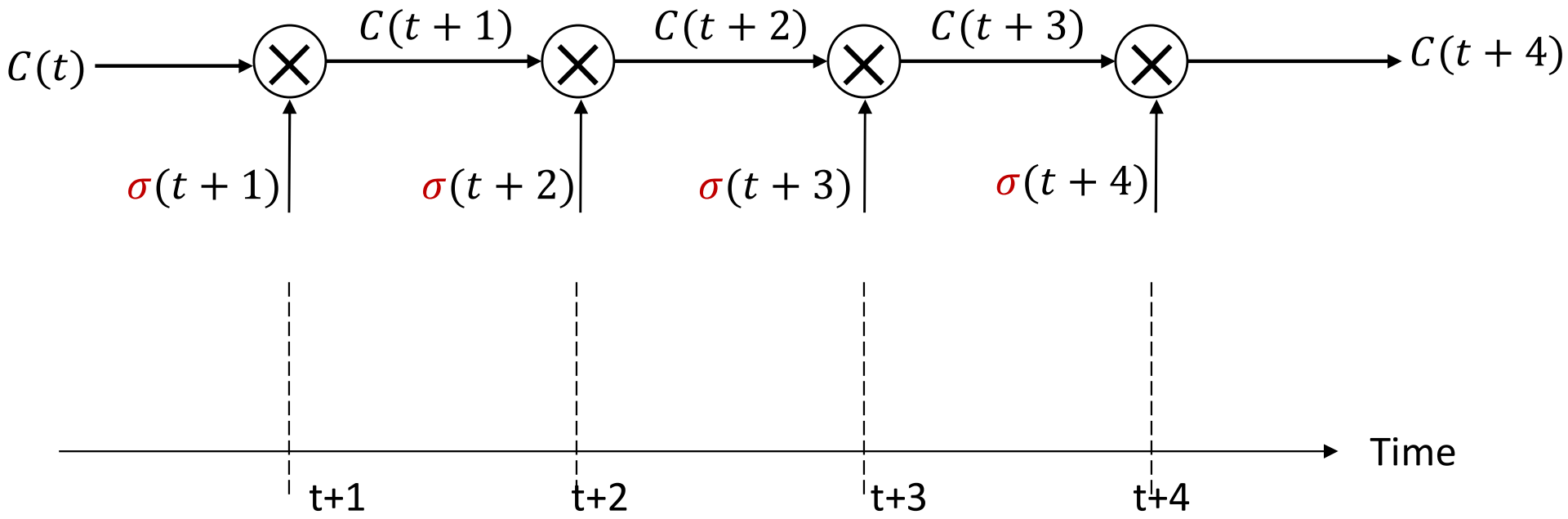
- The length of the loss gradient is modified by the Jacobians of the activations and the singular values of the weights matrix
  - Activations will always shrink (or at best maintain) the length of the gradient
  - Singular values of weight matrices are largely < 1
    - They will cause loss gradient to shrink in most, if not all directions
- This is also true for RNNs, which are just very deep networks

Parameters and activations are problems

- Both, the memory retention of the network, and its ability to *learn* to detect and remember patterns are heavily dominated by weights/biases and activations
  - Rather than what it is trying to "remember"

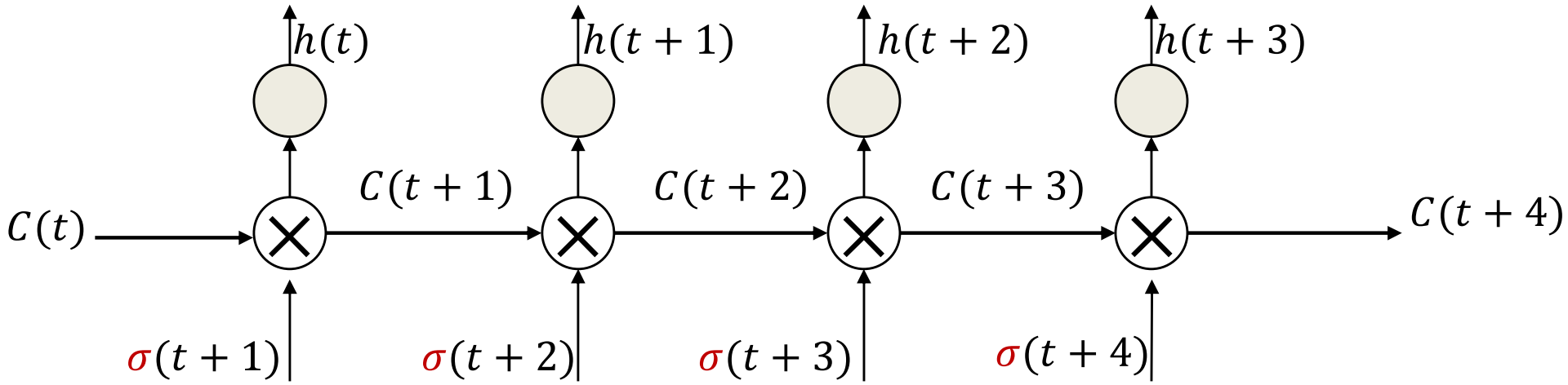
- Can we have a network that just "remembers" arbitrarily long, to be recalled on demand?
  - Not be directly dependent on vagaries of network parameters, but rather on input-based determination of *whether it must be remembered*

Enter – the constant error carousel



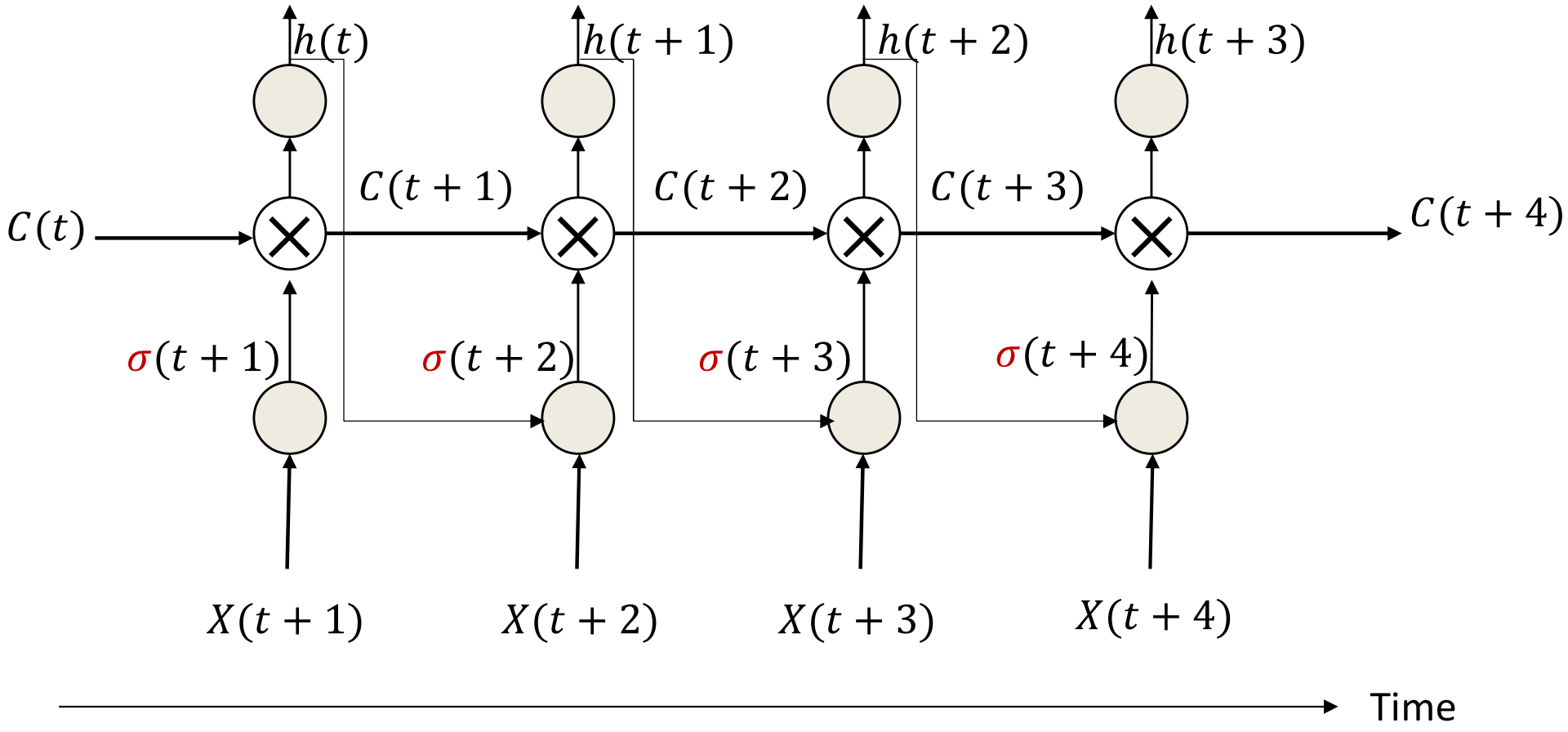
- History is carried through uncompressed
  - No weights, no nonlinearities
  - Only scaling is through the σ "gating" term that captures other triggers
  - E.g. "Have I seen Pattern2"?

Enter – the constant error carousel



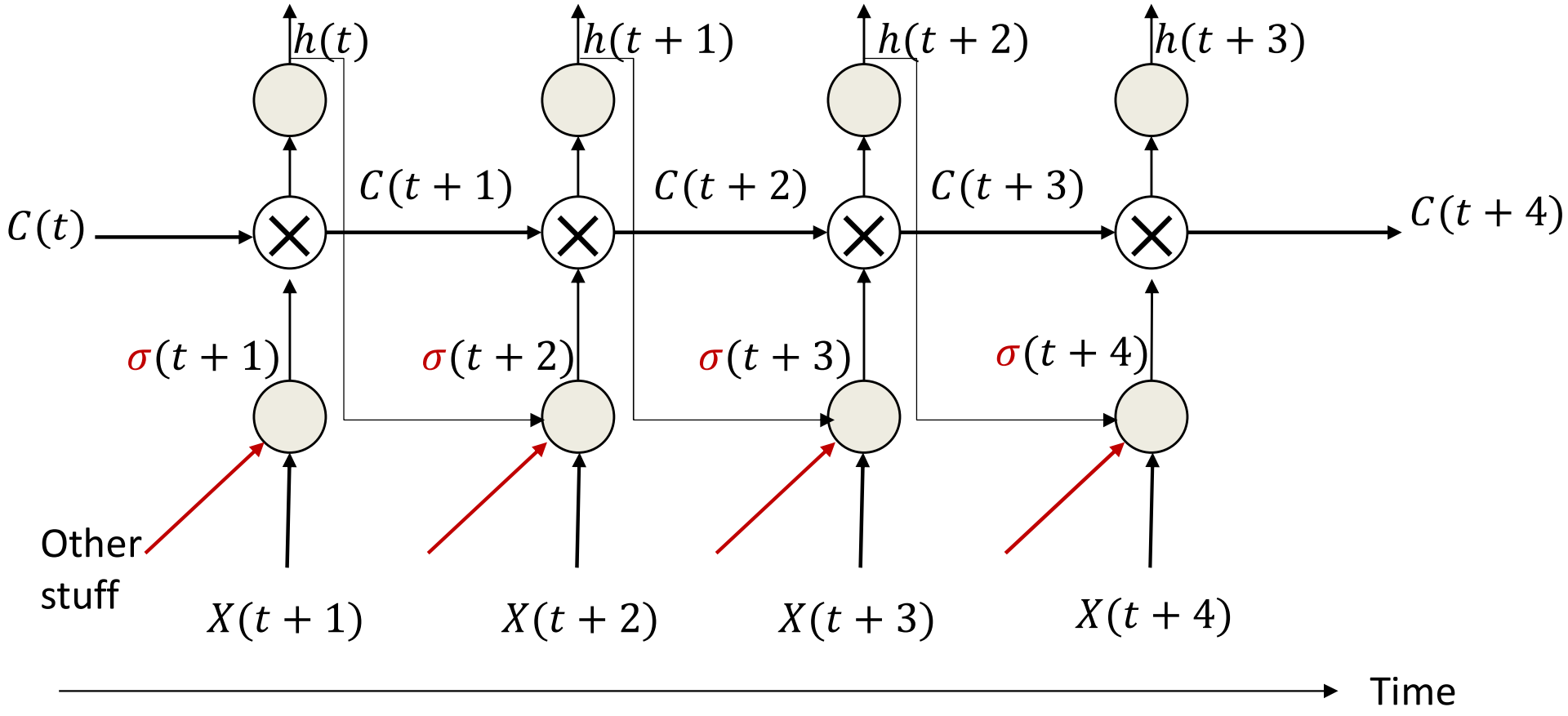
- Actual non-linear work is done by other portions of the network
  - Neurons that compute the workable state from the memory

Enter – the constant error carousel



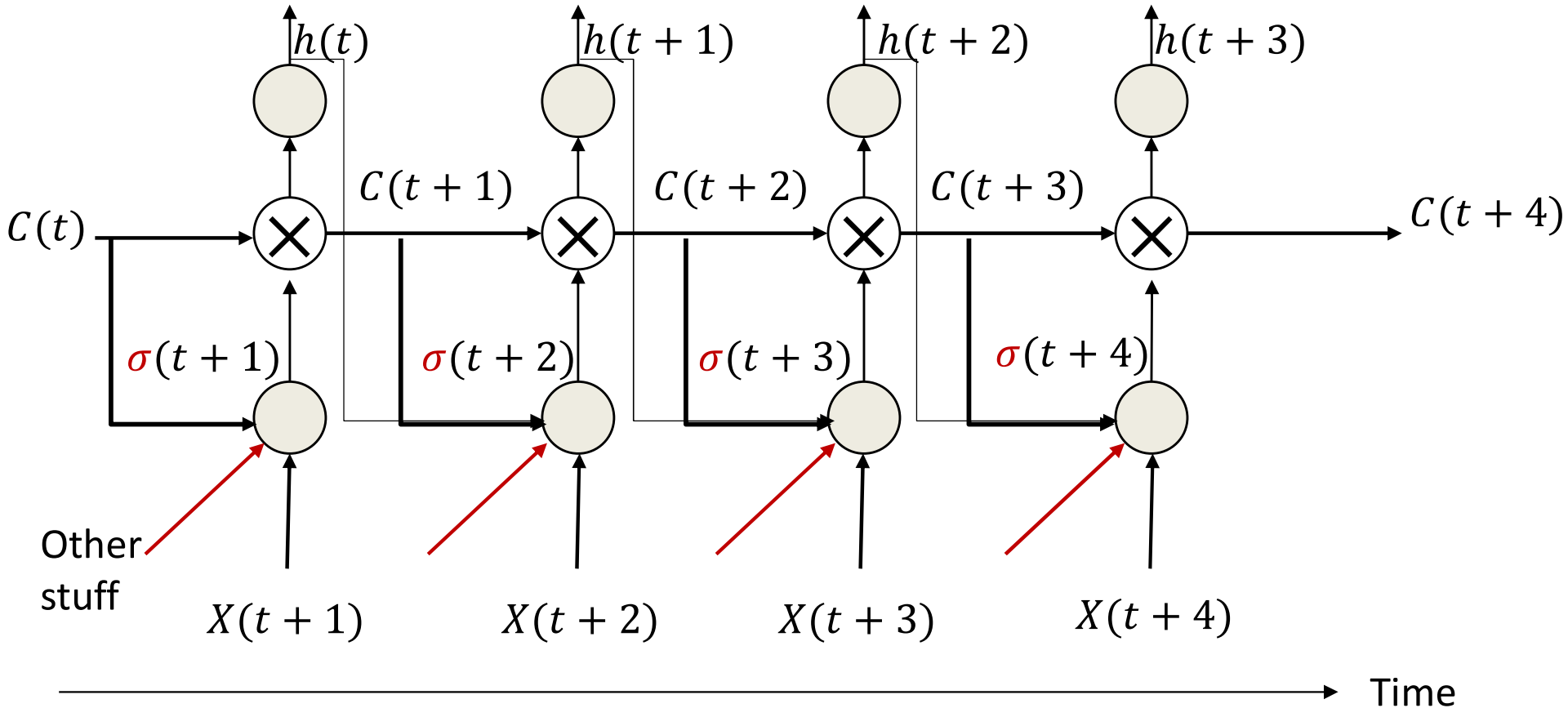
- The gate σ depends on current input, current hidden state…

Enter – the constant error carousel



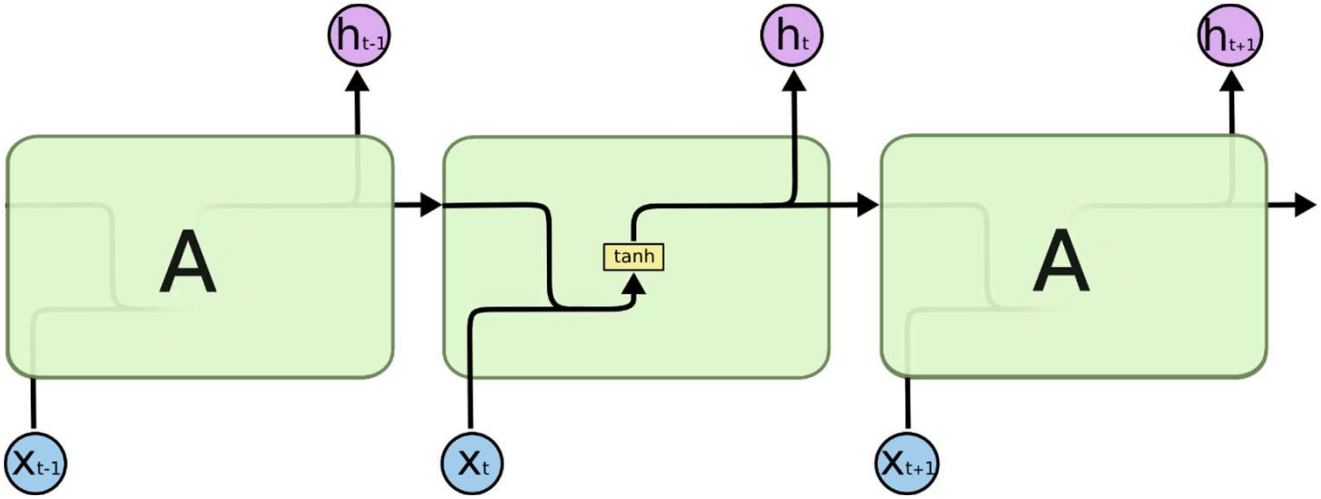
- The gate σ depends on current input, current hidden state… and other stuff…

Enter – the constant error carousel



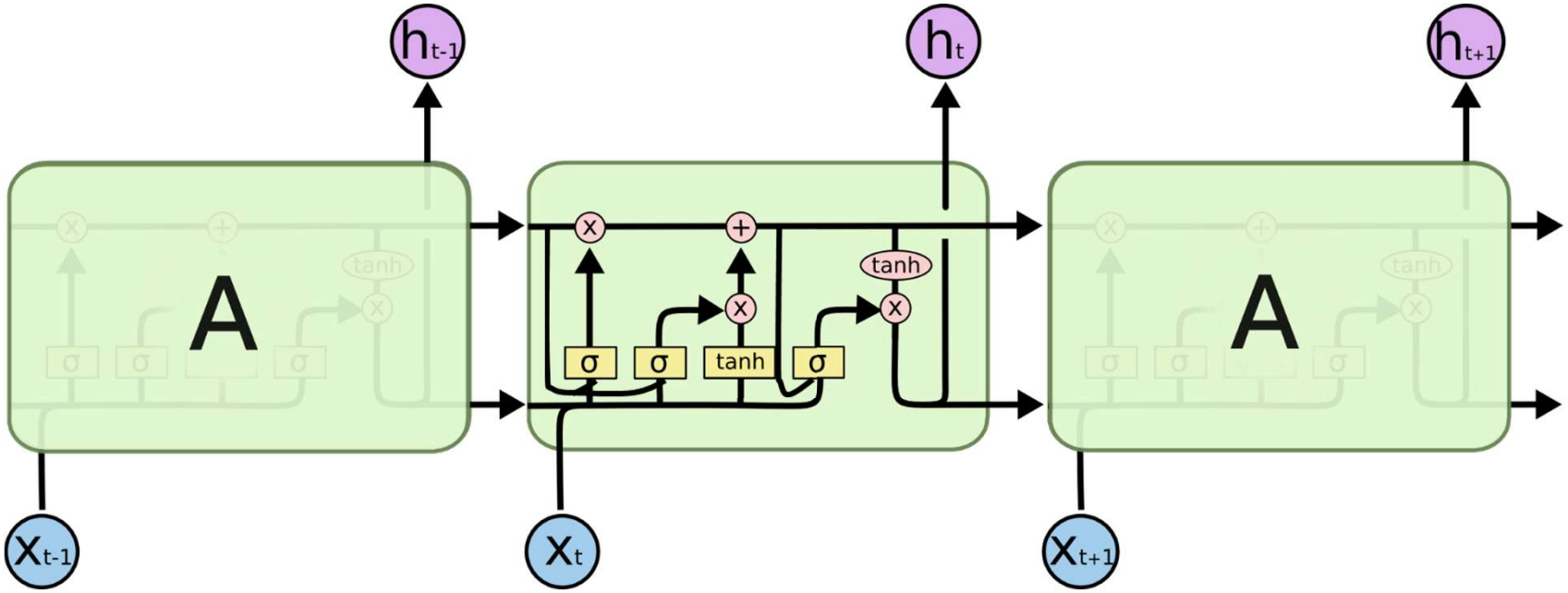
- The gate σ depends on current input, current hidden state… and other stuff…
- Including, obviously, what is currently in raw memory

Standard RNN



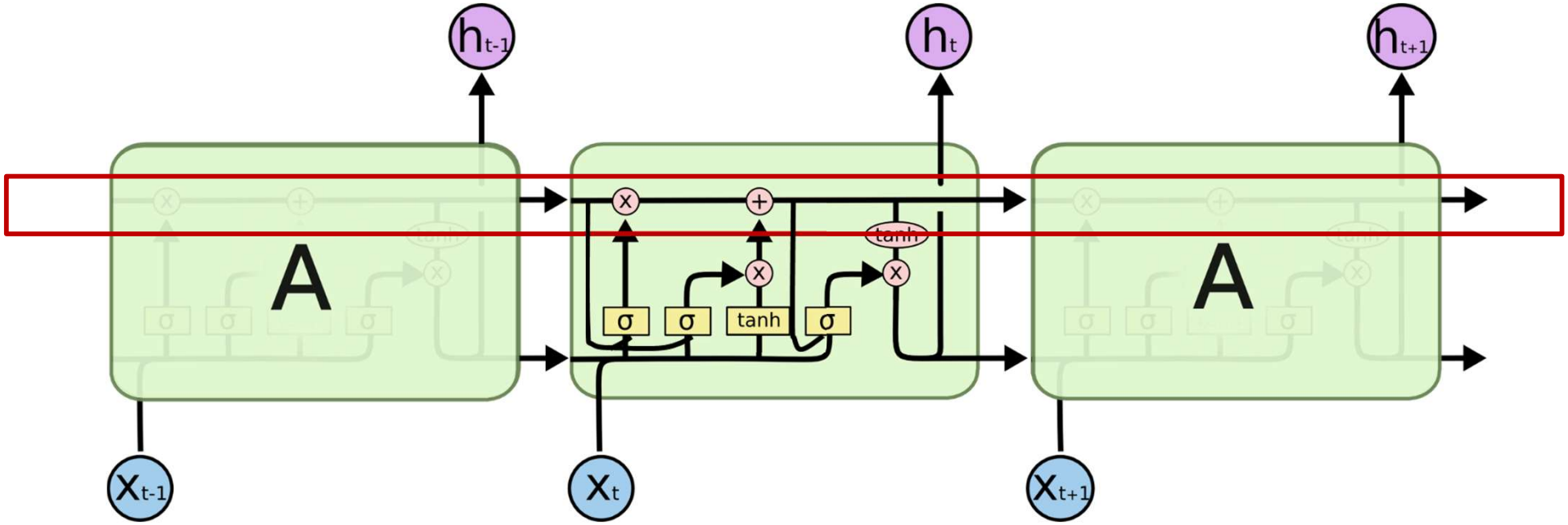
- Recurrent neurons receive past recurrent outputs and current input as inputs
- Processed through a tanh() activation function
  - As mentioned earlier, tanh() is the generally used activation for the hidden layer
- Current recurrent output passed to next higher layer and next time instant

Long Short-Term Memory



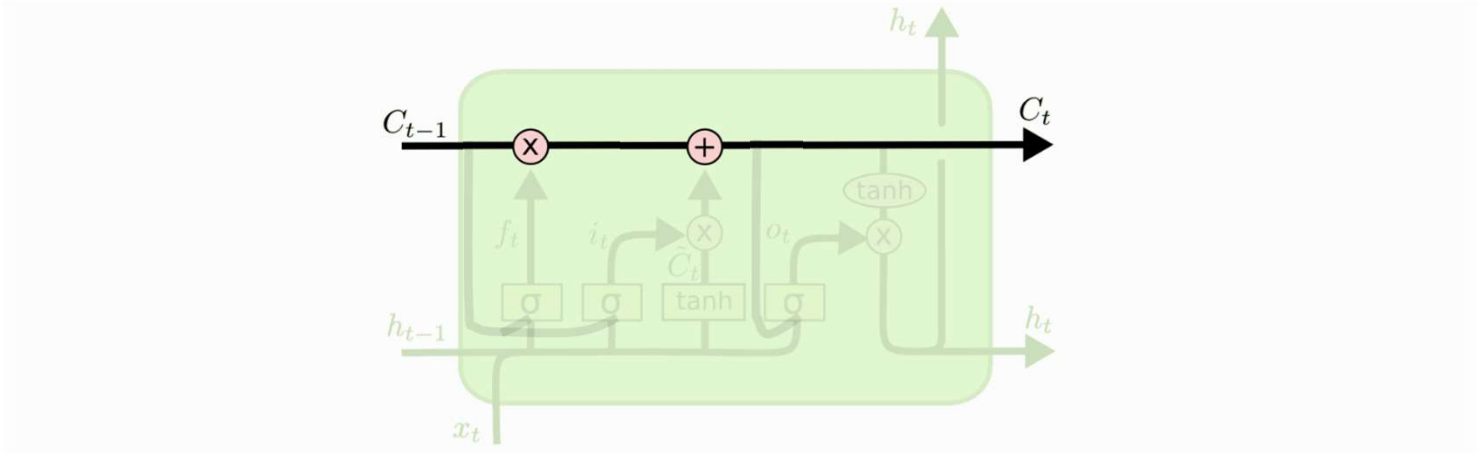
- The $\sigma()$ are *multiplicative gates* that decide if something is important or not
- Remember, every line actually represents a *vector*

LSTM: Constant Error Carousel



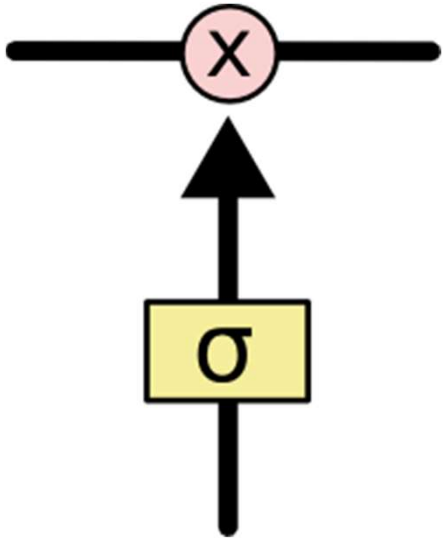
- Key component: a *remembered cell state*

LSTM: CEC



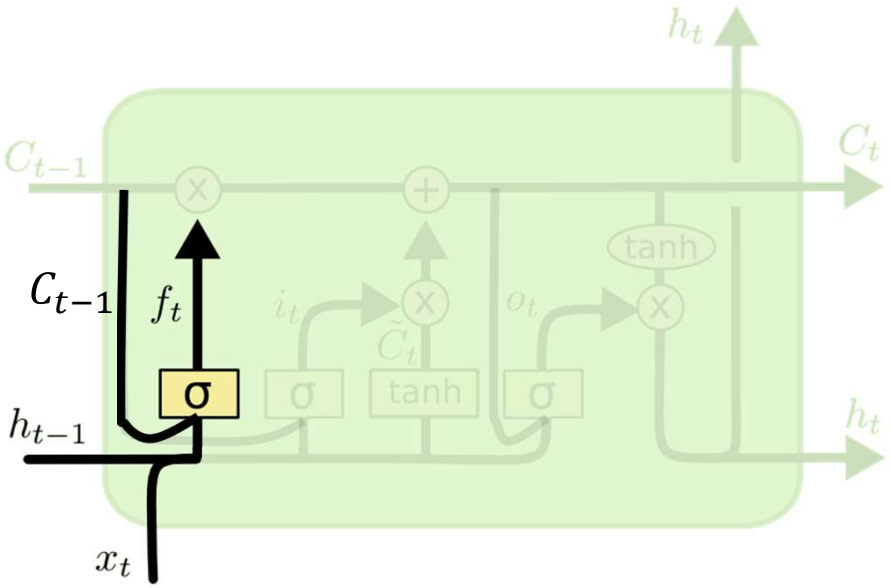
- C_t is the linear history carried by the *constant-error carousel*
- Carries information through, only affected by a gate
  - And *addition of history,* which too is gated..

LSTM: Gates



- Gates are simple sigmoidal units with outputs in the range (0,1)
- Controls how much of the information is to be let through

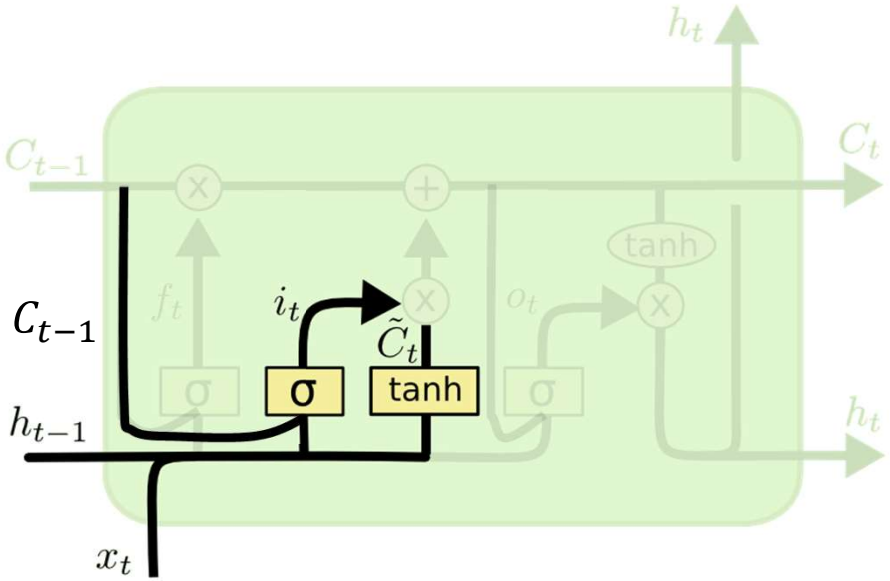
LSTM: Forget gate



$$f_t = \sigma\big(W_f(C_{t-1}, h_{t-1}, x_t) + b_f\big)$$

- The first gate determines whether to carry over the history or to forget it
  - More precisely, how much of the history to carry over
  - Also called the "forget" gate
  - Note, we're actually distinguishing between the cell memory C and the state h that is coming over time! They're related though

LSTM: Input gate

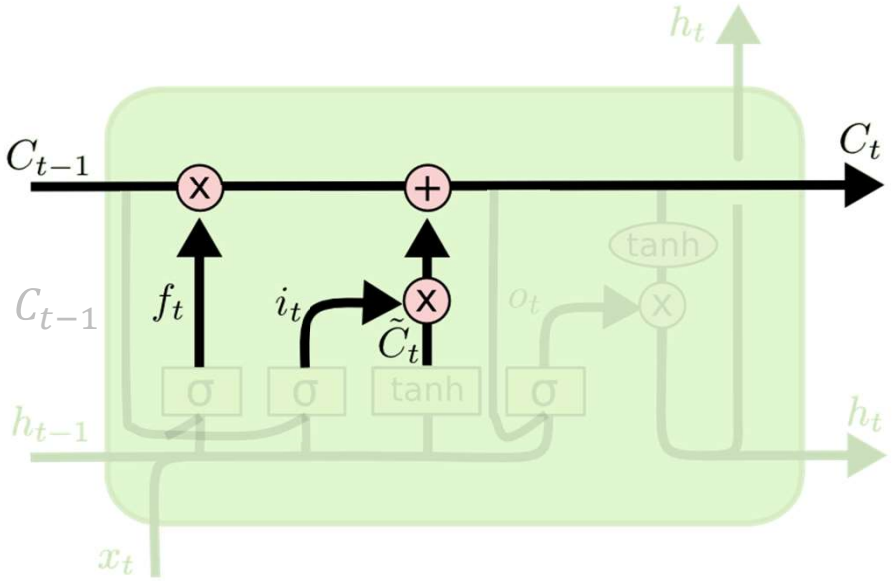


$$i_t = \sigma(W_i(C_{t-1}, h_{t-1}, x_t) + b_i)$$

$$\tilde{C}_t = \tanh(W_C(h_{t-1}, x_t) + b_C)$$

- The second input has two parts
  - A perceptron layer that determines if there's something new and interesting in the input
  - A gate that decides if its worth remembering

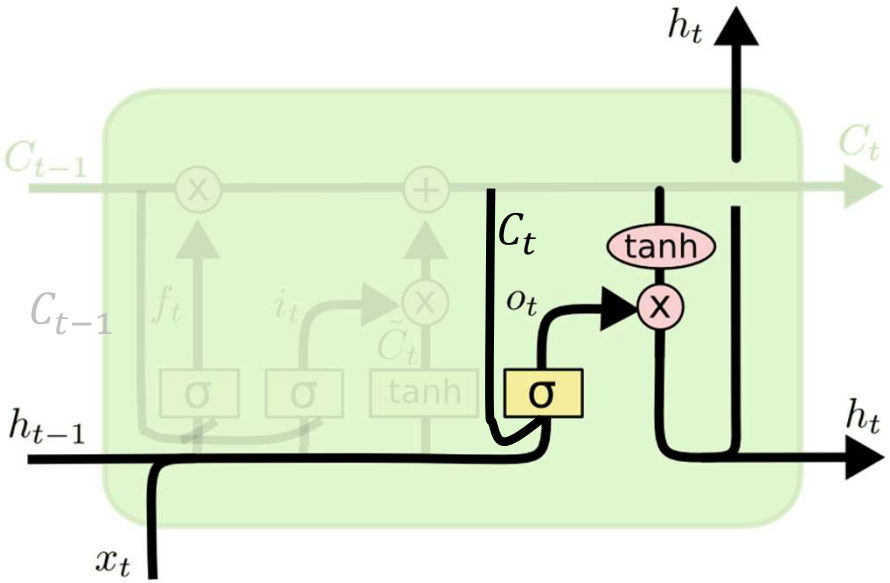
LSTM: Memory cell update



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- The second input has two parts
  - A perceptron layer that determines if there's something interesting in the input
  - A gate that decides if its worth remembering
  - **If so its added to the current memory cell**

LSTM: Output and Output gate

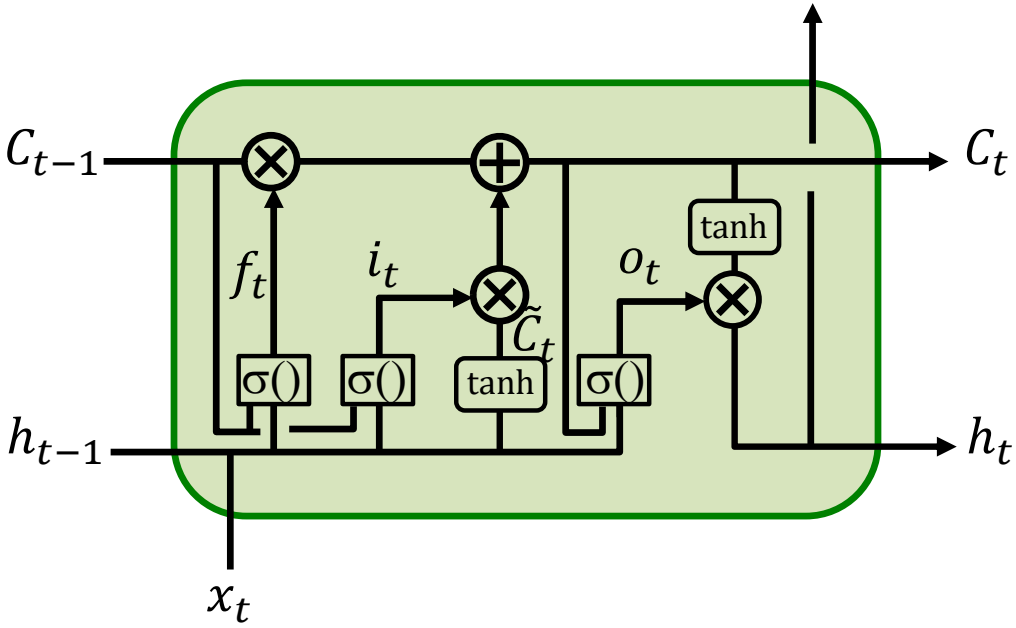


$$o_t = \sigma(W_o(C_t, h_{t-1}, x_t) + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

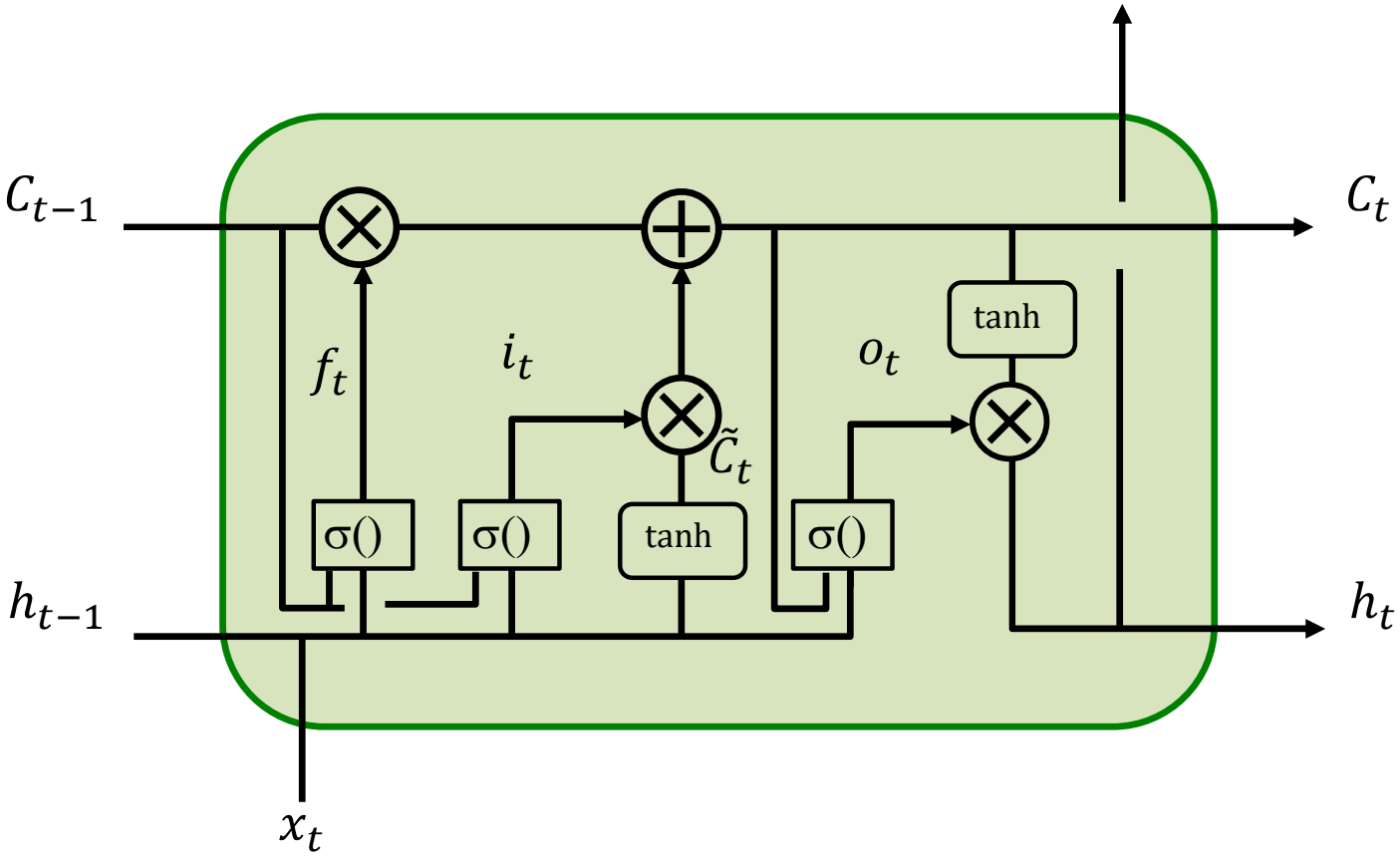
- The *output* of the cell
  - Simply compress it with tanh to make it lie between 1 and -1
    - Note that this compression no longer affects our ability to *carry* memory forward
  - Controlled by an *output* gate
    - To decide if the memory contents are worth reporting at *this* time

The complete LSTM unit



- With all components..

LSTM computation: Forward



- Forward rules:

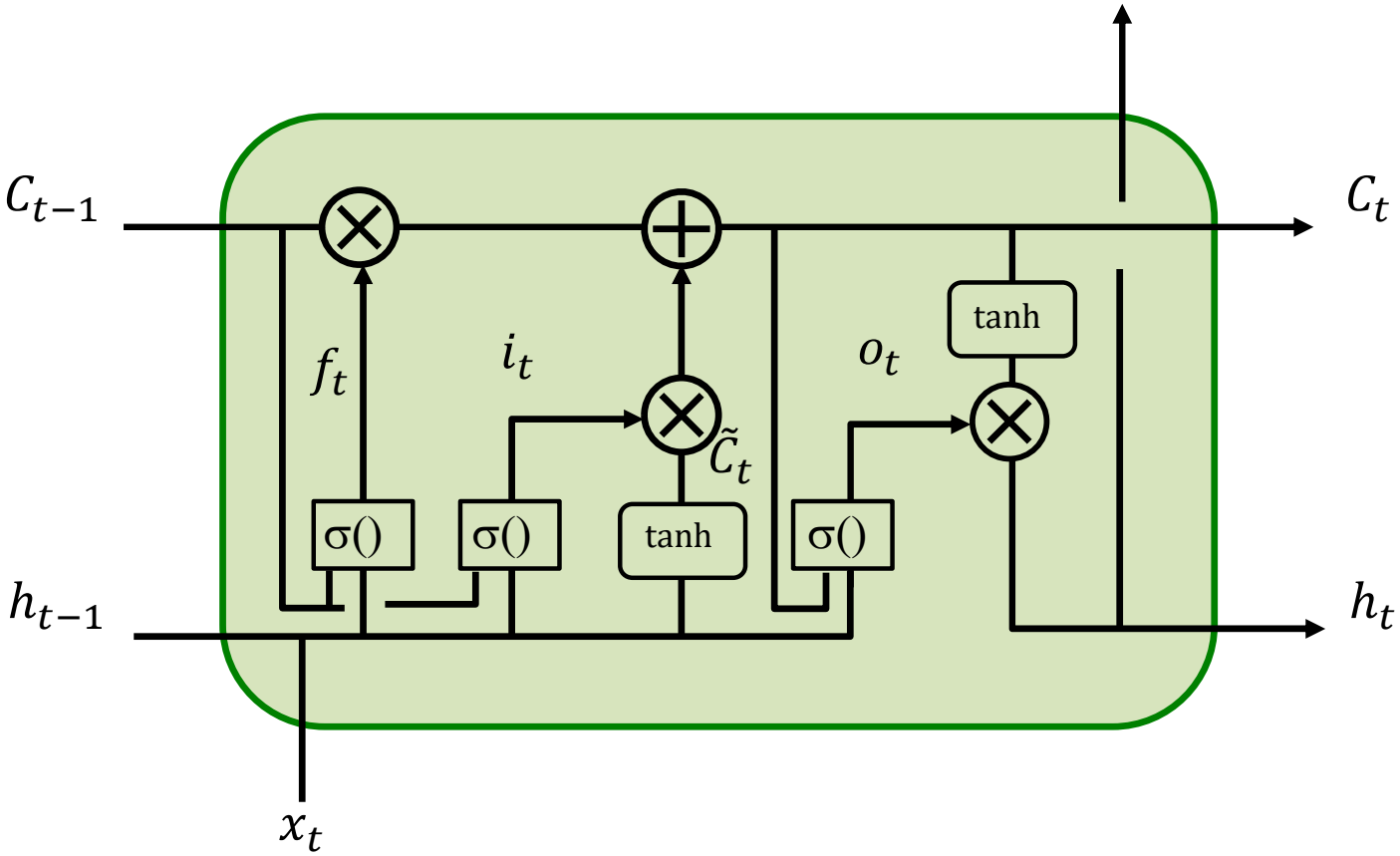
Gates

$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Variables

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM computation: Forward



- Forward rules:

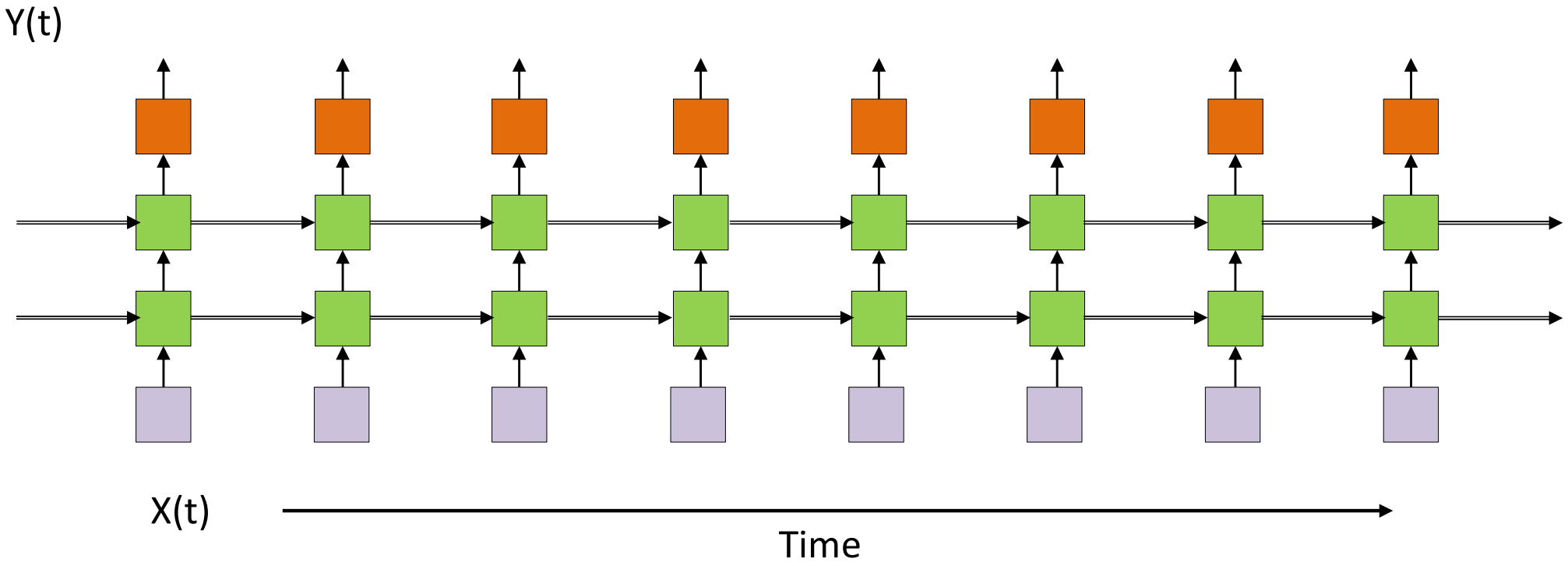
Gates

$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

Variables

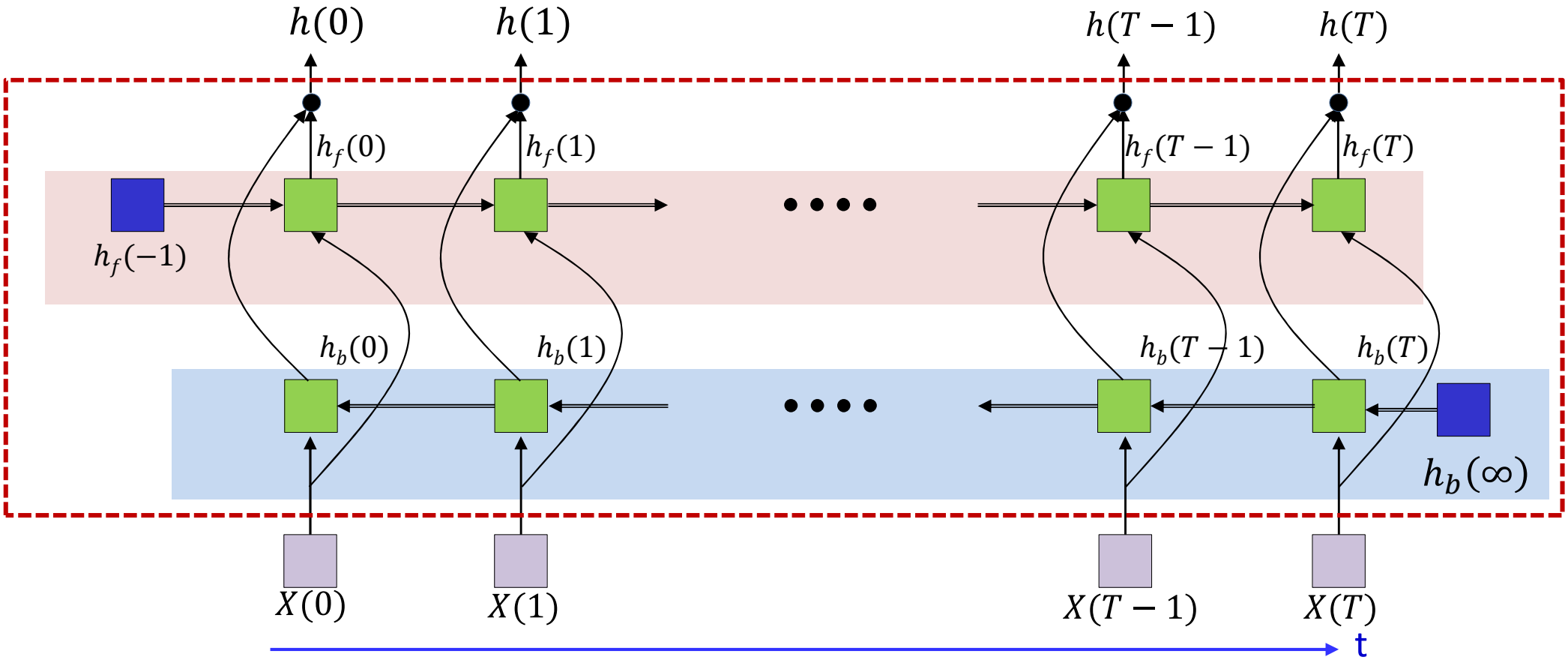
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$h_t = o_t * \tanh(C_t)$$

LSTM architectures example



- Each green box is now an (layer of) LSTM cell(s)
  - Keep in mind each box is an *array* of units
  - For LSTMs the horizontal arrows carry both $C(t)$ and $h(t)$

Bidirectional LSTM



- Like the BRNN, but now the hidden nodes are LSTM units.
  - Or layers of LSTM units

Poll 1

- @ , @

Poll 1

Select all that are true

- **Conventional RNNs lose their memory because of the properties of their recurrent weights matrices and activations**
- **LSTMs eliminate this problem by eliminating recurrent weights and activations**
- LSTMs have no mechanism for forgetting patterns

In an LSTM the forget gate and the input pattern detector are separate because (select all that are true)

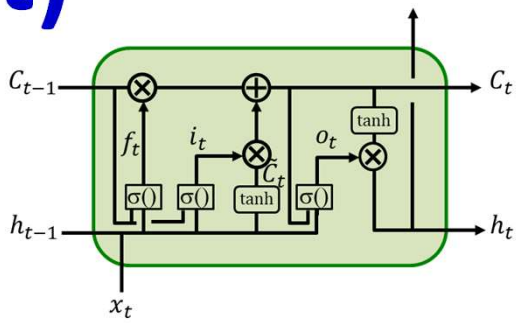
- **The patterns that inform the LSTM that a stored memory may be forgotten could be different from the patterns that are stored**
- A “forget” signal from the forget gate will also result in a “decrement” signal from the input pattern detector

Notes on the pseudocode

`Class LSTM_cell`

- We will assume an object-oriented program
- Each LSTM unit is assumed to be an “LSTM cell”
- There’s a new copy of the LSTM cell at each time, at each layer
- LSTM cells retain local variables that are not relevant to the computation outside the cell
  - These are static and retain their value once computed, unless overwritten

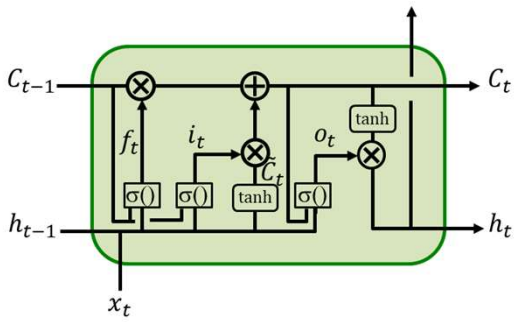
LSTM cell (single unit) Definitions



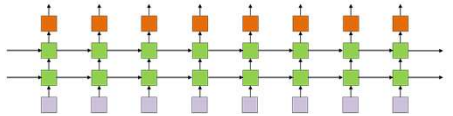

```
# Input:
#     C : previous value of CEC
#     h : previous hidden state value ("output" of cell)
#     x:  Current input
# [W,b]: The set of all model parameters for the cell
#       These include all weights and biases
# Output
#     C : Next value of CEC
#     h : Next value of h
# In the function:  sigmoid(x) = 1/(1+exp(-x))
#                   performed component-wise

# Static local variables to the cell
static local z_f, z_i, z_c, z_o, f, i, o, C_i
function [C,h] = LSTM_cell.forward(C,h,x,[W,b])
    code on next slide
```

LSTM cell forward

```
# Continuing from previous slide
# Note: [W,h] is a set of parameters, whose individual elements are
#       shown in red within the code.  These are passed in

# Static local variables which aren't required outside this cell
static local z_f, z_i, z_c, z_o, f, i, o, C_i
function [C_o, h_o] = LSTM_cell.forward(C,h,x, [W,b])
    z_f = W_fc C + W_fh h + W_fx x + b_f
    f = sigmoid(z_f) # forget gate

    z_i = W_ic C + W_ih h + W_ix x + b_i
    i = sigmoid(z_i) # input gate

    z_c = W_ch h + W_cx x + b_c
    C_i = tanh(z_c)  # Detecting input pattern

    C_o = f∘C + i∘C_i # "∘" is component-wise multiply

    z_o = W_oc C_o + W_oh h + W_ox x + b_o
    o = sigmoid(z_o) # output gate

    h_o = o∘tanh(C_o) # "∘" is component-wise multiply

    return C_o,h_o
```



LSTM network forward



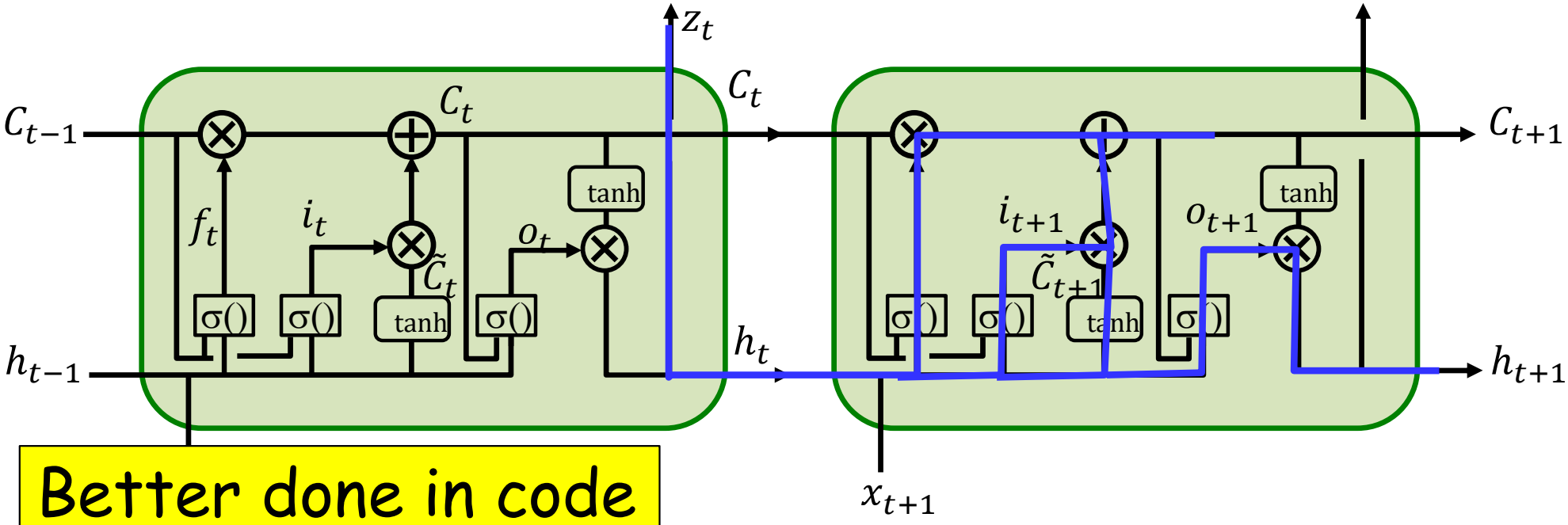
```
# Assuming h(-1,*) is known and C(-1,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
#                for the l^th hidden layer
# W_o and b_o are output layer weights and biases

for t = 0:T-1  # Including both ends of the index
    h(t,0) = x(t) # Vectors. Initialize h(0) to input
    for l = 1:L  # hidden layers operate at time t
        [C(t,l),h(t,l)] = LSTM_cell(t,l).forward(…
                …C(t-1,l),h(t-1,l),h(t,l-1)[W{l},b{l}])
    z_o(t) = W_o h(t,L) + b_o
    Y(t) = softmax( z_o(t) )
```

Training the LSTM

- Identical to training regular RNNs with one difference
  - Commonality: Define a sequence divergence and backpropagate its derivative through time

- Difference: Instead of backpropagating gradients through an RNN unit, we will backpropagate through an LSTM cell

Backpropagation rules: Backward



Better done in code

$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ tanh'(.) + tanh(.) \circ \sigma'(.) W_{Co}) +$$
$$\nabla_{C_{t+1}} Div \circ \left(f_{t+1} + C_t \circ \sigma'(.) W_{Cf} + \tilde{C}_{t+1} \circ \sigma'(.) W_{Ci} \ldots\right)$$

$$\nabla_{h_t} Div = \nabla_{z_t} Div \nabla_{h_t} z_t + \nabla_{C_{t+1}} Div \circ \left(C_t \circ \sigma'(.) W_{hf} + \tilde{C}_{t+1} \circ \sigma'(.) W_{hi}\right) +$$
$$\nabla_{C_{t+1}} Div \circ o_{t+1} \circ tanh'(.) W_{hC} + \nabla_{h_{t+1}} Div \circ tanh(.) \circ \sigma'(.) W_{ho}$$

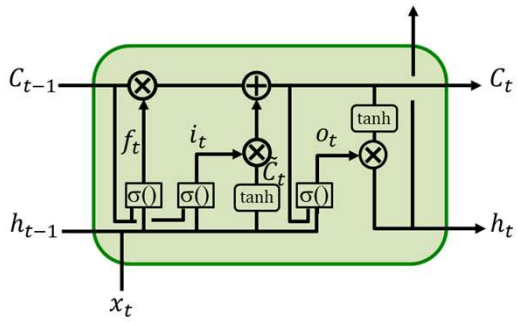
Notes on the backward pseudocode

`Class LSTM_cell`

- We first provide backward computation *within a cell*
- For the backward code, we will assume the static variables computed during the forward are still available
- The following slides first show the forward code for reference
- Subsequently we will give you the backward, and explicitly indicate *which* of the forward equations each backward equation refers to
  - *The backward code for a cell is long (but simple) and extends over multiple slides*

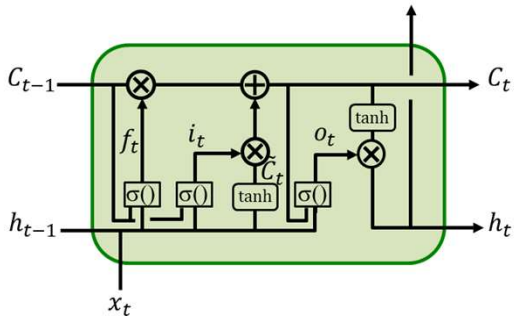
LSTM cell forward (for reference)

```
# Continuing from previous slide
# Note: [W,h] is a set of parameters, whose individual elements are
#       shown in red within the code.  These are passed in

# Static local variables which aren't required outside this cell
static local z_f, z_i, z_c, z_o, f, i, o, C_i
function [C_o, h_o] = LSTM_cell.forward(C,h,x, [W,b])
    z_f = W_fc C + W_fh h + W_fx x + b_f
    f = sigmoid(z_f) # forget gate

    z_i = W_ic C + W_ih h + W_ix x + b_i
    i = sigmoid(z_i) # input gate

    z_c = W_ch h + W_cx x + b_c
    C_i = tanh(z_c)  # Detecting input pattern

    C_o = f∘C + i∘C_i # "∘" is component-wise multiply

    z_o = W_oc C_o + W_oh h + W_ox x + b_o
    o = sigmoid(z_o) # output gate

    h_o = o∘tanh(C_o) # "∘" is component-wise multiply

    return C_o,h_o
```



LSTM cell backward

```
# Static local variables carried over from forward
static local z_f, z_i, z_c, z_o, f, i, o, C_i
function [dC,dh,dx,d[W, b]]=LSTM_cell.backward(dC_o, dh_o, C, h, C_o, h_o, x, [W,b])
    # First invert h_o = o∘tanh(C)
    do = dh_o ∘ tanh(C_o)^T
    d tanhC_o = dh_o ∘ o^T
    dC_o += dtanhC_o ∘ (1-tanh^2(C_o))^T #(1-tanh^2) is the derivative of tanh

    # Next invert o = sigmoid(z_o)
    dz_o = do ∘ sigmoid(z_o)^T ∘ (1-sigmoid(z_o))^T # do x derivative of sigmoid(z_o)

    # Next invert z_o = W_oc C_o + W_oh h + W_ox x + b_o
    dC_o += dz_o W_oc  #  Note – this is a regular matrix multiply
    dh = dz_o W_oh
    dx = dz_o W_ox

    dW_oc = C_o dz_o  # Note – this multiplies a column vector by a row vector
    dW_oh = h dz_o
    dW_ox = x dz_o
    db_o = dz_o

    # Next invert C_o = f∘C + i∘C_i
    dC = dC_o ∘ f
    dC_i = dC_o ∘ i
    di = dC_o ∘ C_i
    df = dC_o ∘ C
```



LSTM cell backward (continued)

$\# \text{ Next invert } C_i = \tanh(z_c)$

$dz_c = dC_i \circ (1-\tanh^2(z_c))^T$

$\# \text{ Next invert } W_{ch}h + W_{cx}x + b_c$

$dh \mathrel{+}= dz_c\, W_{ch}$

$dx \mathrel{+}= dz_c\, W_{cx}$

$dW_{ch} = h\, dz_c$

$dW_{cx} = x\, dz_c$

$db_c = dz_c$

$\# \text{ Next invert } i = \text{sigmoid}(z_i)$

$dz_i = di \circ \text{sigmoid}(z_i)^T \circ (1-\text{sigmoid}(z_i))^T$

$\# \text{ Next invert } z_i = W_{ic}C + W_{ih}h + W_{ix}x + b_i$

$dC \mathrel{+}= dz_i\, W_{ic}$

$dh \mathrel{+}= dz_i\, W_{ih}$

$dx \mathrel{+}= dz_i\, W_{ix}$

$dW_{ic} = C\, dz_i$

$dW_{ih} = h\, dz_i$

$dW_{ix} = x\, dz_i$

$db_i = dz_i$

LSTM cell backward (continued)

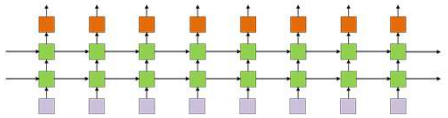
```
# Next invert f = sigmoid(z_f)
dz_f = df ∘ sigmoid(z_f)^T ∘ (1-sigmoid(z_f))^T

# Finally invert z_f = W_fc C + W_fh h + W_fx x + b_f
dC += dz_f W_fc
dh += dz_f W_fh
dx += dz_f W_fx

dW_fc = C dz_f
dW_fh = h dz_f
dW_fx = x dz_f
db_f = dz_f

return dC, dh, dx, d[W, b]
# d[W,b] is shorthand for the complete set
  of weight and bias derivatives
```

LSTM network forward (for reference)

```
# Assuming h(-1,*) is known and C(-1,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
#                for the l^th hidden layer
# W_o and b_o are output layer weights and biases

for t = 0:T-1  # Including both ends of the index
    h(t,0) = x(t) # Vectors. Initialize h(0) to input
    for l = 1:L  # hidden layers operate at time t
        [C(t,l),h(t,l)] = LSTM_cell(t,l).forward(…
                …C(t-1,l),h(t-1,l),h(t,l-1)[W{l},b{l}])
    z_o(t) = W_o h(t,L) + b_o
    Y(t) = softmax( z_o(t) )
```

LSTM network backward

```
# Assuming h(-1,*) is known and C(-1,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
#              for the l^th hidden layer
# W_o and b_o are output layer weights and biases
# Y is the output of the network
# Assuming dW_o and db_o and d[W{l} b{l}] (for all l) are
#          all initialized to 0 at the start of the computation

for t = T-1:0  # Including both ends of the index
    dz_o  = dY(t) ∘ Softmax_Jacobian(zo(t))
    dW_o += h(t,L) dz_o(t)
    dh(t,L) = dz_o(t)W_o
    db_o += dz_o(t)

    for l = L-1:0
        [dC(t,l),dh(t,l),dx(t,l),d[W, b]] = …
              … LSTM_cell(t,l).backward(…
              … dC(t+1,l), dh(t+1,l)+dx(t,l+1), C(t-1,l), h(t-1,l), …
              … C(t,l), h(t,l), h(t,l-1), [W(l),b(l)])
        d[W{l} b{l}] += d[W,b]
```



Poll 2

- @ , @

Poll 2

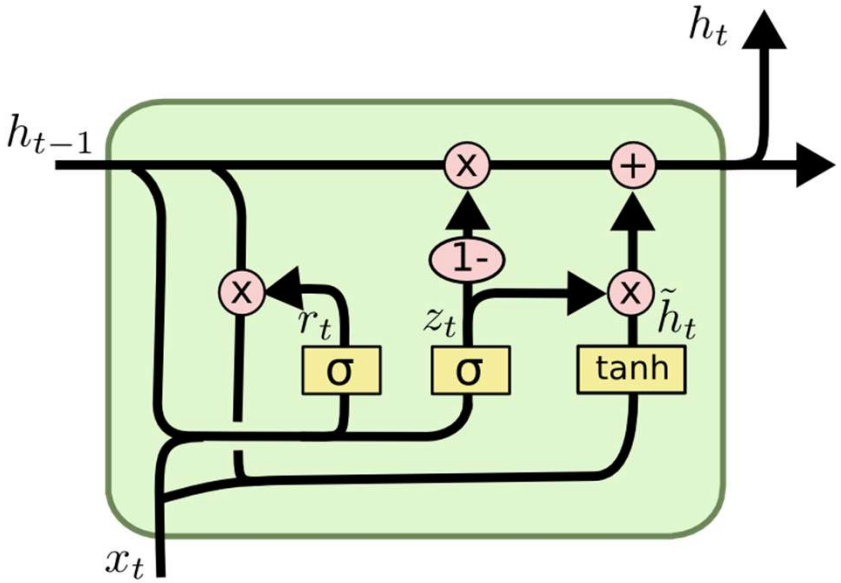
Backward computation in an LSTM can be performed by computing the derivatives of all forward operations in reverse order (T/F)

- **True**
- False

This necessarily requires computation of complicated derivative formulae (T/F)

- **True**
- **False**

Gated Recurrent Units: Lets simplify the LSTM



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

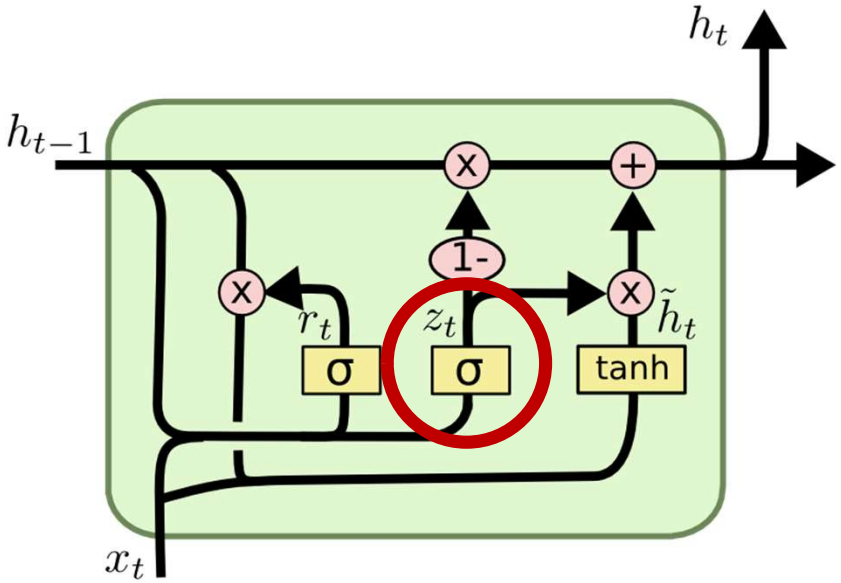
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- Simplified LSTM which addresses some of your concerns of *why*

Gated Recurrent Units: Lets simplify the LSTM



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

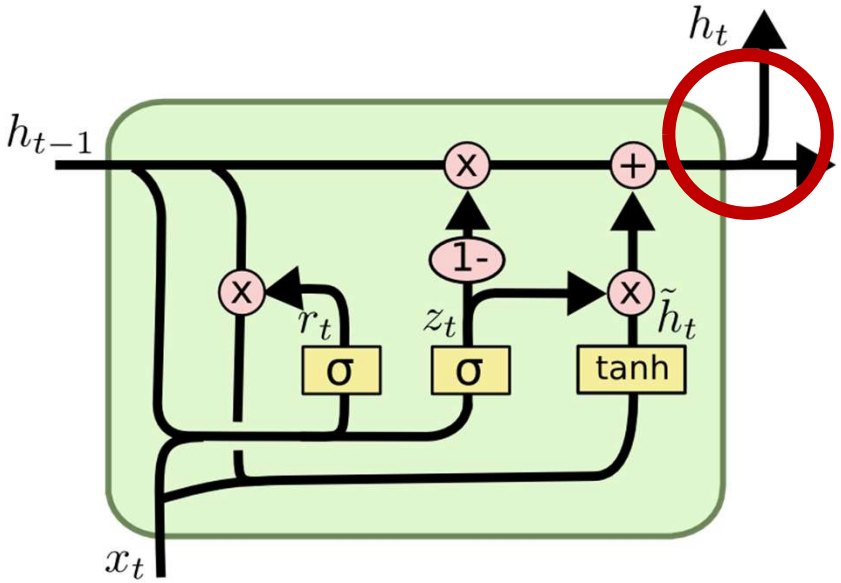
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- Combine forget and input gates
  - In new input is to be remembered, then this means old memory is to be forgotten
    - Why compute twice?

Gated Recurrent Units: Lets simplify the LSTM



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

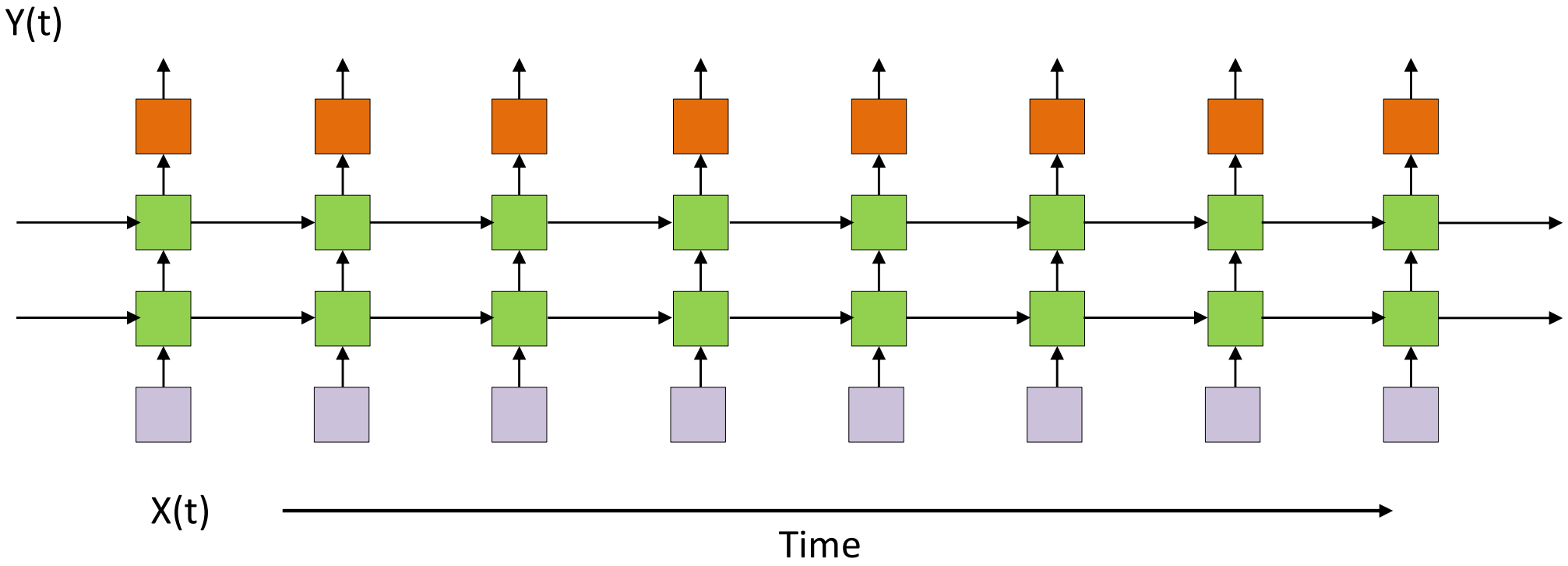
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- Don't bother to separately maintain compressed and regular memories
  - Pointless computation!
  - Redundant representation

GRU architectures example



- Each green box is now a (layer of) GRU cell(s)
  - Keep in mind each box is an *array* of units

Poll 3

- @, @

Poll 3

GRUs are simplifications of the LSTM that use the principle that if a pattern triggers forgetting of a pattern, it cannot also trigger increment of the memory

- **True**
- False

Like LSTMs, GRUs retain separate lines for the store memory and the hidden state

- True
- **False**

Story so far

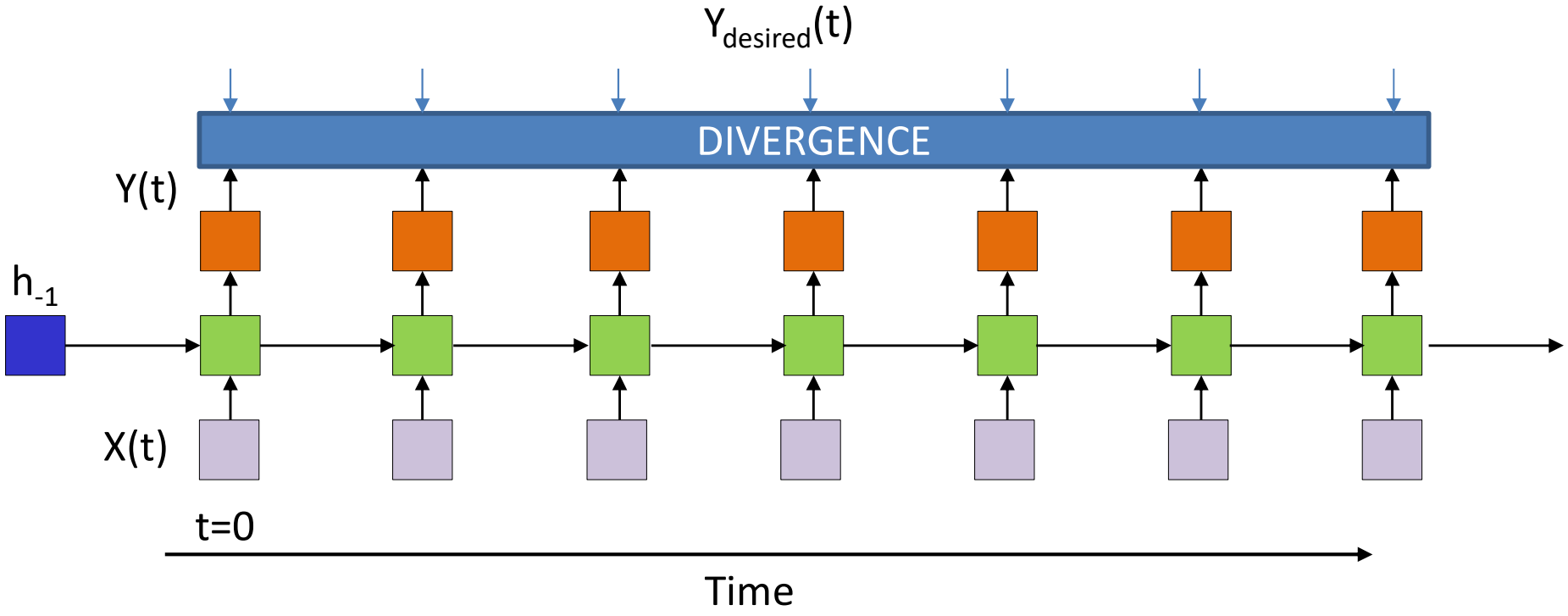
- Recurrent networks are poor at memorization
  - Memory can explode or vanish depending on the weights and activation
- They also suffer from the vanishing gradient problem during training
  - Error at any time cannot affect parameter updates in the too-distant past
  - E.g. seeing a "close bracket" cannot affect its ability to predict an "open bracket" if it happened too long ago in the input

- LSTMs are an alternative formalism where memory is made more directly dependent on the input, rather than network parameters/structure
  - Through a "Constant Error Carousel" memory structure with no weights or activations, but instead direct switching and "increment/decrement" from pattern recognizers
  - Do not suffer from a vanishing gradient problem but ***do* suffer from *exploding* gradient issue**

Significant issues

- The Divergence
- How to use these nets..
- This and more in the remaining lecture(s)

Key Issue



- How do we define the divergence
- Also: how do we compute the outputs..

But first – a brief detour…

Which open source project?

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECON
  return segtable;
}
```

Related math. What is it talking about?

Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

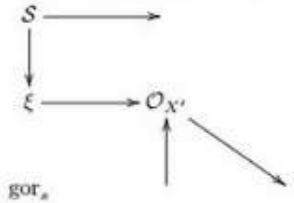
be a morphism of algebraic spaces over S and Y.

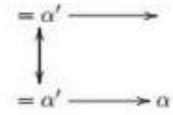
Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram





$\mathrm{Spec}(K_\psi)$ Mor_{Sets} $\mathrm{d}(\mathcal{O}_{X_{X/k}}, \mathcal{G})$

is a limit. Then $\mathcal{G}$ is a finite type and assume S is a flat and $\mathcal{F}$ and $\mathcal{G}$ is a finite type f_*. This is of finite type diagrams, and

- the composition of $\mathcal{G}$ is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

□

Proof. We have see that $X = \mathrm{Spec}(R)$ and $\mathcal{F}$ is a finite type representable by algebraic space. The property $\mathcal{F}$ is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U. □

Proof. This is clear that $\mathcal{G}$ is a finite presentation, see Lemmas ??.
A *reduced above* we conclude that U is an open covering of $\mathcal{C}$. The functor $\mathcal{F}$ is a "field

$$\mathcal{O}_{X,x} \longrightarrow \mathcal{F}_{\overline{x}} \quad \text{-}1(\mathcal{O}_{X_{\acute{e}tale}}) \longrightarrow \mathcal{O}_{X_t}^{-1}\mathcal{O}_{X_\lambda}(\mathcal{O}_{X_\eta}^{\overline{w}})$$

is an isomorphism of covering of $\mathcal{O}_{X_t}$. If $\mathcal{F}$ is the unique element of $\mathcal{F}$ such that X is an isomorphism.
The property $\mathcal{F}$ is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme $\mathcal{O}_X$-algebra with $\mathcal{F}$ are opens of finite type over S.
If $\mathcal{F}$ is a scheme theoretic image points. □

If $\mathcal{F}$ is a finite direct sum $\mathcal{O}_{X_\lambda}$ is a closed immersion, see Lemma ??. This is a sequence of $\mathcal{F}$ is a similar morphism.

And a Wikipedia page explaining it all

```
Naturalism and decision for the majority of Arab countries' capitalide was grounded
by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated
with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal
in the [[Protestant Immineners]], which could be said to be directly in Cantonese
Communication, which followed a ceremony and set inspired prison, training. The
emperor travelled back to [[Antioch, Perth, October 25|21]] to note, the Kingdom
of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known
in western [[Scotland]], near Italy to the conquest of India with the conflict.
Copyright was the succession of independence in the slop of Syrian influence that
was a famous German movement based on a more popular servicious, non-doctrinal
and sexual power post. Many governments recognize the military housing of the
[[Civil Liberalization and Infantry Resolution 265 National Party in Hungary]],
that is sympathetic to be to the [[Punjab Resolution]]
(PJS)[http://www.humah.yahoo.com/guardian.
cfm/7754800786d17551963s89.htm Official economics Adjoint for the Nazism, Montgomery
was swear to advance to the resources for those Socialism's rule,
was starting to signing a major tripad of aid exile.]]
```

The unreasonable effectiveness of recurrent neural networks..

- All previous examples were *generated* blindly by a *recurrent* neural network..
  - With simple architectures

- http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Modern text generation is a lot more sophisticated that that

- One of the many sages of the time, the Bodhisattva Bodhisattva Sakyamuni (1575-1611) was a popular religious figure in India and around the world. This Bodhisattva Buddha was said to have passed his life peacefully and joyfully, without passion and anger. For over twenty years he lived as a lay man and dedicated himself toward the welfare, prosperity, and welfare of others. Among the many spiritual and philosophical teachings he wrote, three are most important; the first, titled the "Three Treatises of Avalokiteśvara"; the second, the teachings of the "Ten Questions;" and the third, "The Eightfold Path of Discipline.“
  - Entirely randomly generated

Brief detour: Language models

- Modelling language using recurrent nets

- More generally language models and embeddings..

Language modelling using RNNs

Four score and seven years ???

A B R A H A M L I N C O L ??

- Problem: Given a sequence of words (or characters) predict the next one

Language modelling: Representing words

- Represent words as one-hot vectors
  - Pre-specify a vocabulary of N words in fixed (e.g. lexical) order
    - *E.g. [A AARDVARK AARON ABACK ABACUS… ZZYP]*
  - Represent each word by an N-dimensional vector with N-1 zeros and a single 1 (in the position of the word in the ordered list of words)
    - E.g. "AARDVARK" → [0 1 0 0 0 …]
    - E.g. "AARON" → [0 0 1 0 0 0 …]

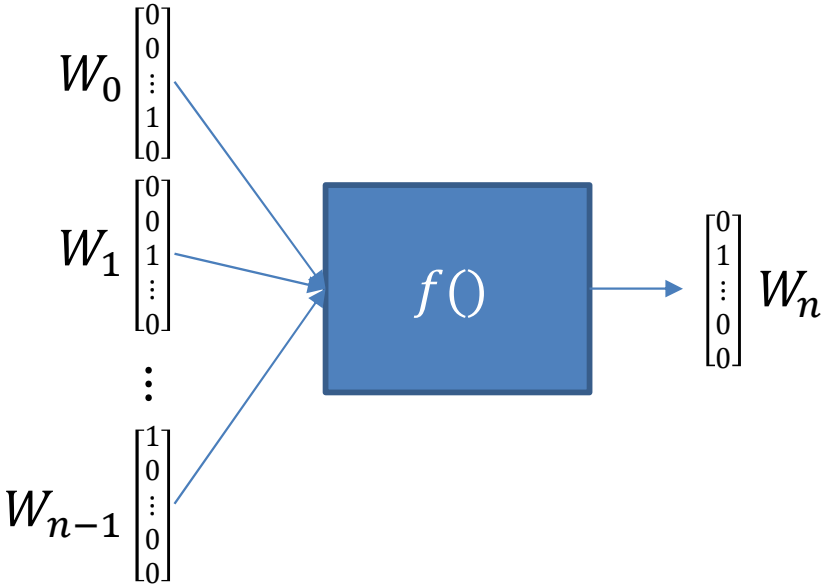
- Characters can be similarly represented
  - English will require about 100 characters, to include both cases, special characters such as commas, hyphens, apostrophes, etc., and the space character

Predicting words

Four score and seven years ???

$W_n = f(W_0, \ldots, W_{n-1})$

Nx1 one-hot vectors



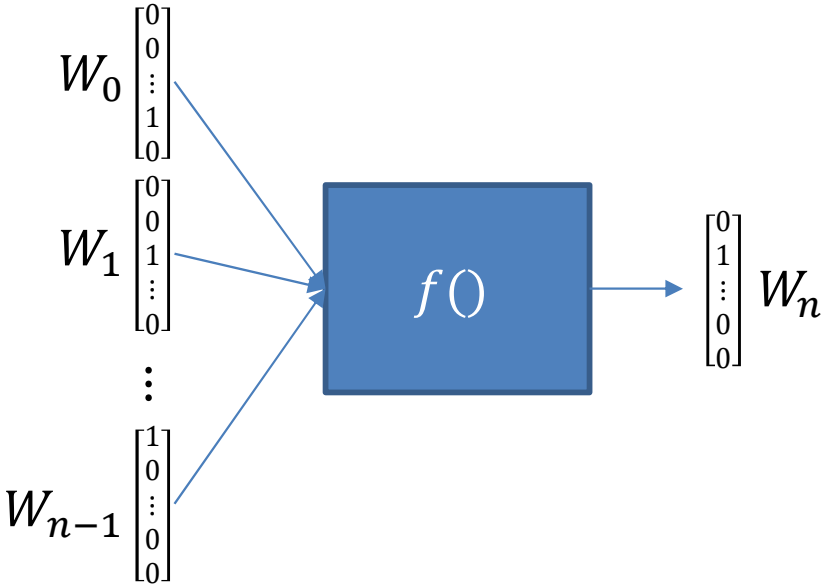
- Given one-hot representations of $W_0 \ldots W_{n-1}$, predict W_n

Predicting words

Four score and seven years ???

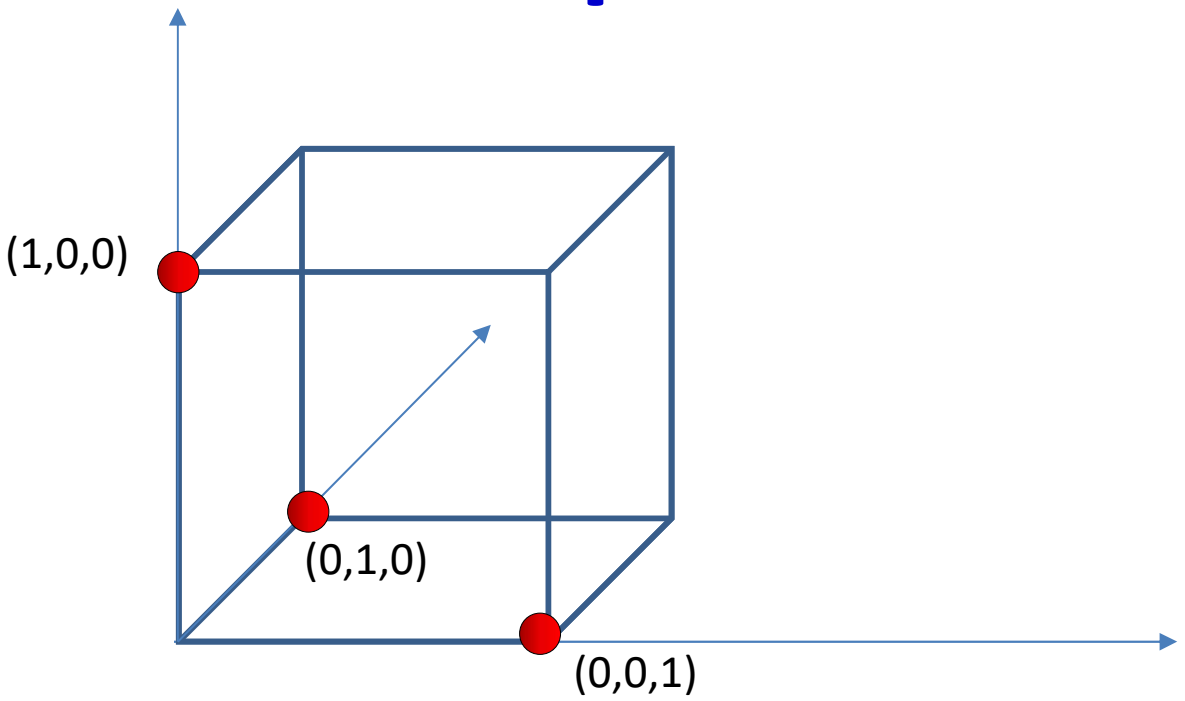
$W_n = f(W_0, \ldots, W_{n-1})$

Nx1 one-hot vectors



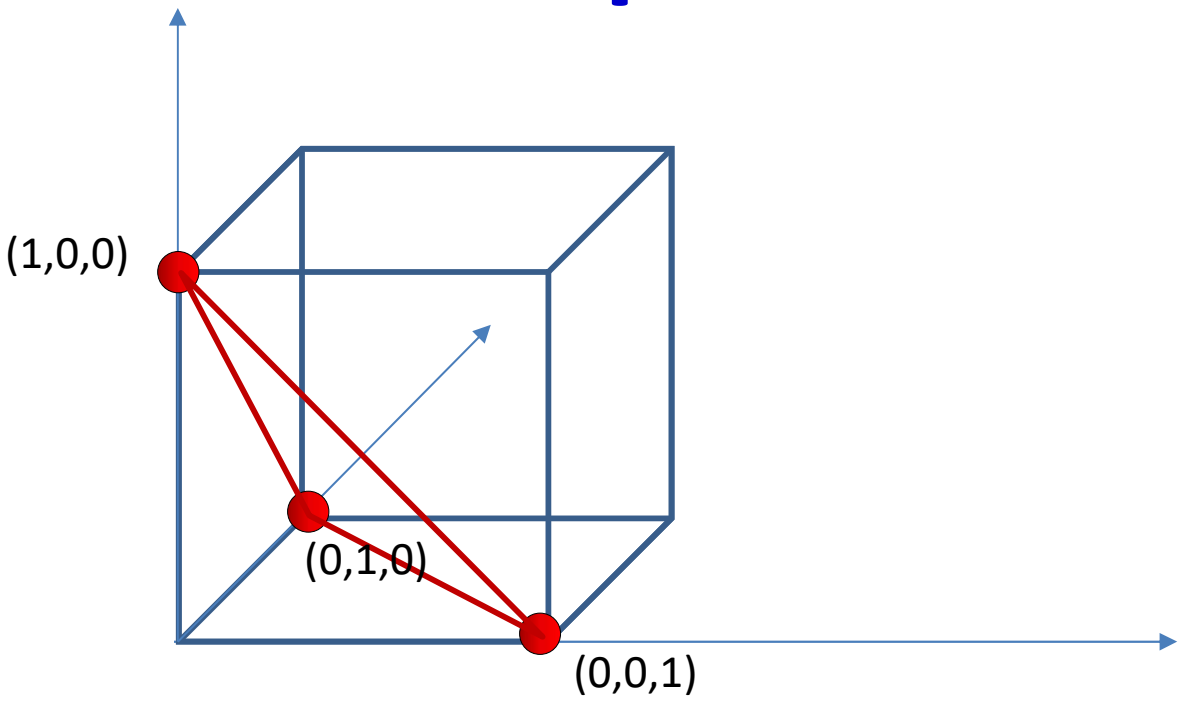
- Given one-hot representations of $W_0 \ldots W_{n-1}$, predict W_n
- **Dimensionality problem:** All inputs $W_0 \ldots W_{n-1}$ are both very high-dimensional and very sparse

The one-hot representation



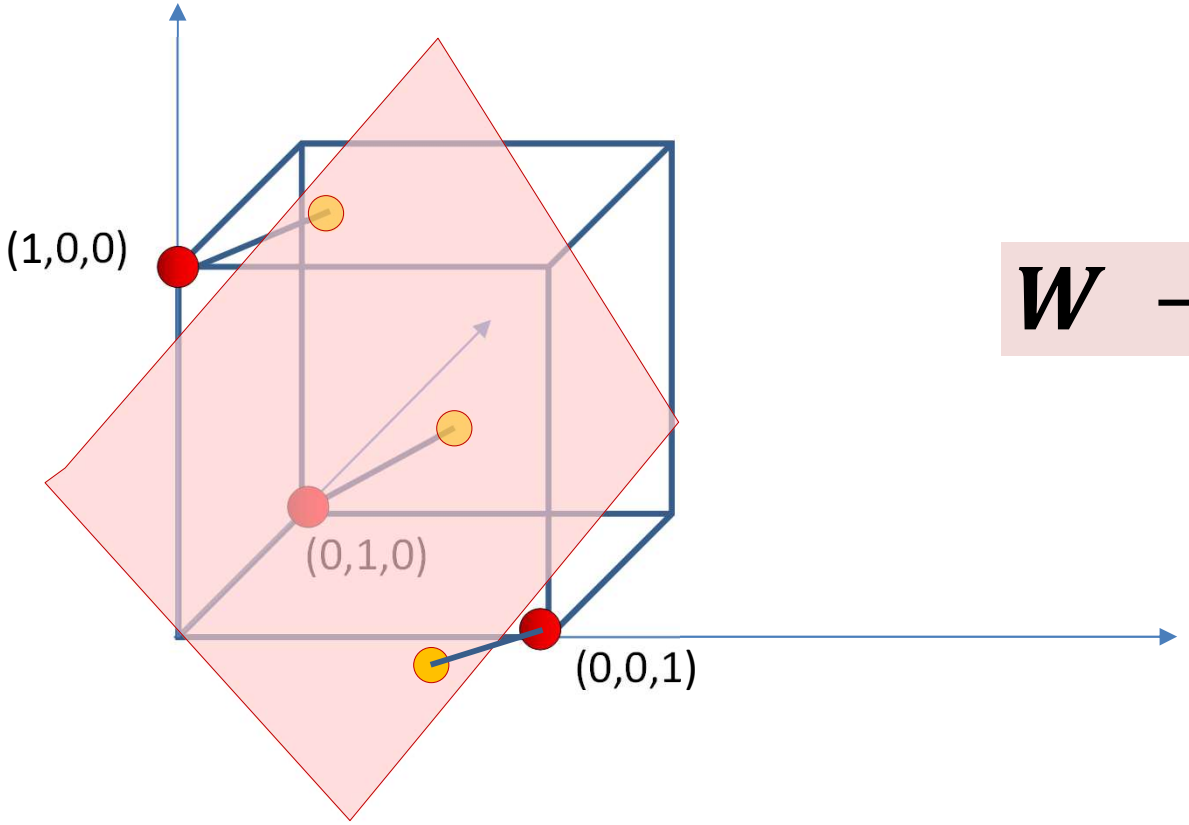
- The one hot representation uses only N corners of the 2^N corners of a unit cube
  - Actual volume of space used = 0
    - $(1, \varepsilon, \delta)$ has no meaning except for $\varepsilon = \delta = 0$
  - Density of points: $\mathcal{O}\left(\frac{N}{r^N}\right)$
- This is a tremendously inefficient use of dimensions

Why one-hot representation



- The one-hot representation makes no assumptions about the relative importance of words
  - All word vectors are the same length
- It makes no assumptions about the relationships between words
  - The distance between every pair of words is the same

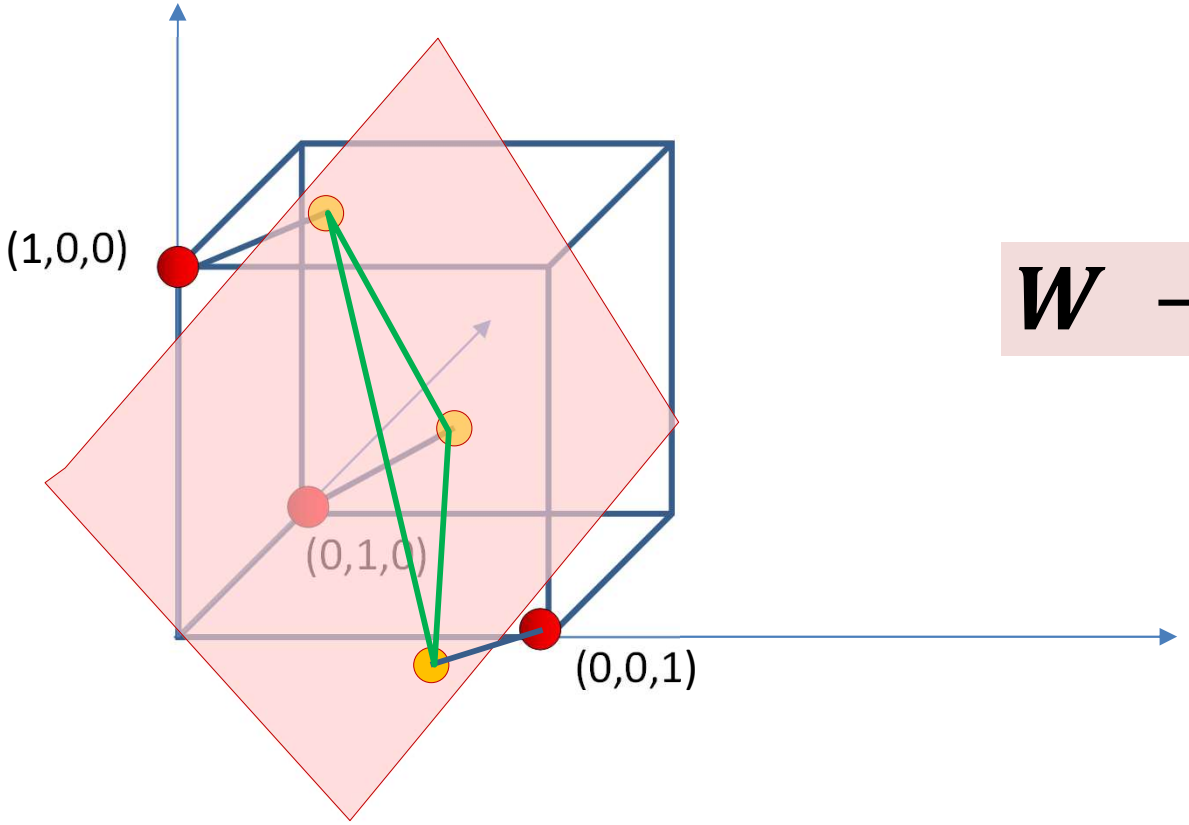
Solution to dimensionality problem



$$W \rightarrow PW$$

- Project the points onto a lower-dimensional subspace
  - Or more generally, a linear transform into a lower-dimensional subspace
  - The volume used is still 0, but density can go up by many orders of magnitude
    - Density of points: $\mathcal{O}\left(\frac{N}{r^M}\right)$

Solution to dimensionality problem



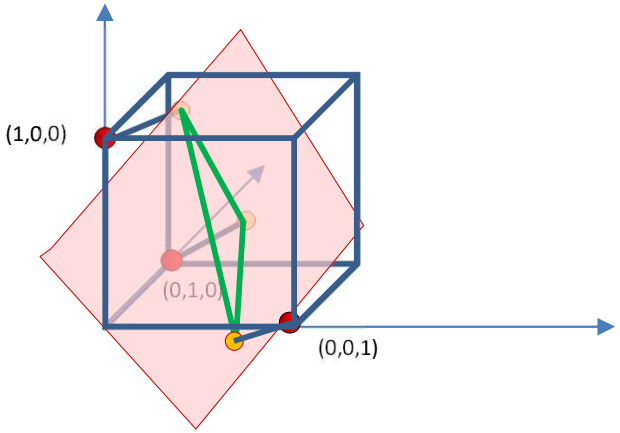
$$W \rightarrow PW$$

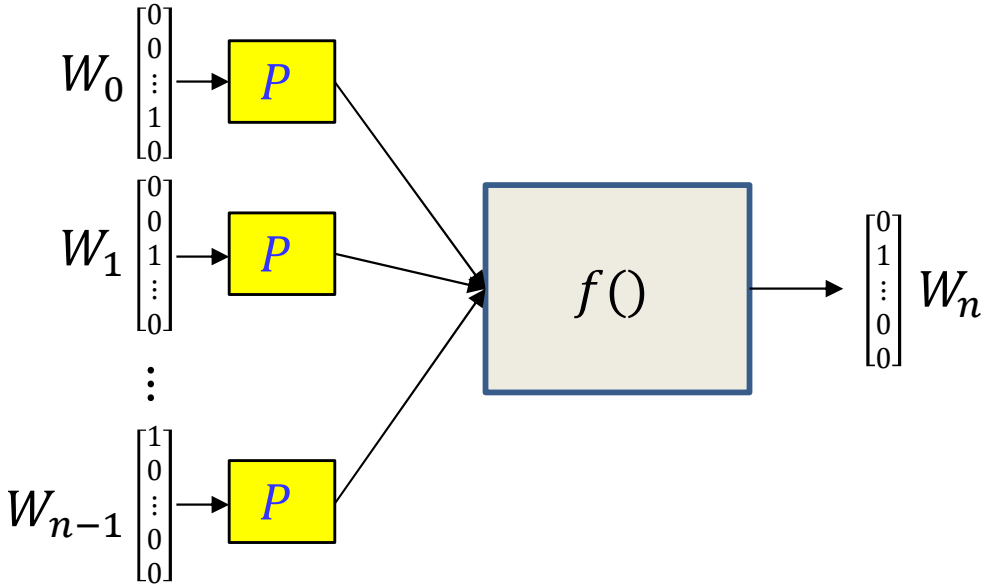
- Project the points onto a lower-dimensional subspace
  - Or more generally, a linear transform into a lower-dimensional subspace
  - The volume used is still 0, but density can go up by many orders of magnitude
    - Density of points: $\mathcal{O}\left(\frac{N}{r^M}\right)$
  - If properly learned, the distances between projected points will capture semantic relations between the words

The *Projected* word vectors

Four score and seven years ???

$$W_n = f(PW_0, PW_2, \ldots, PW_{n-1})$$

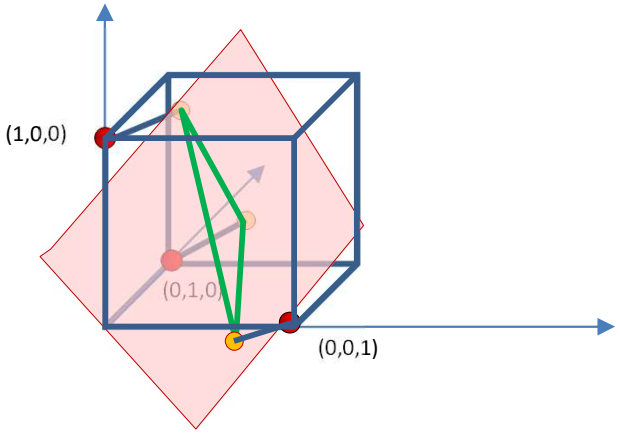


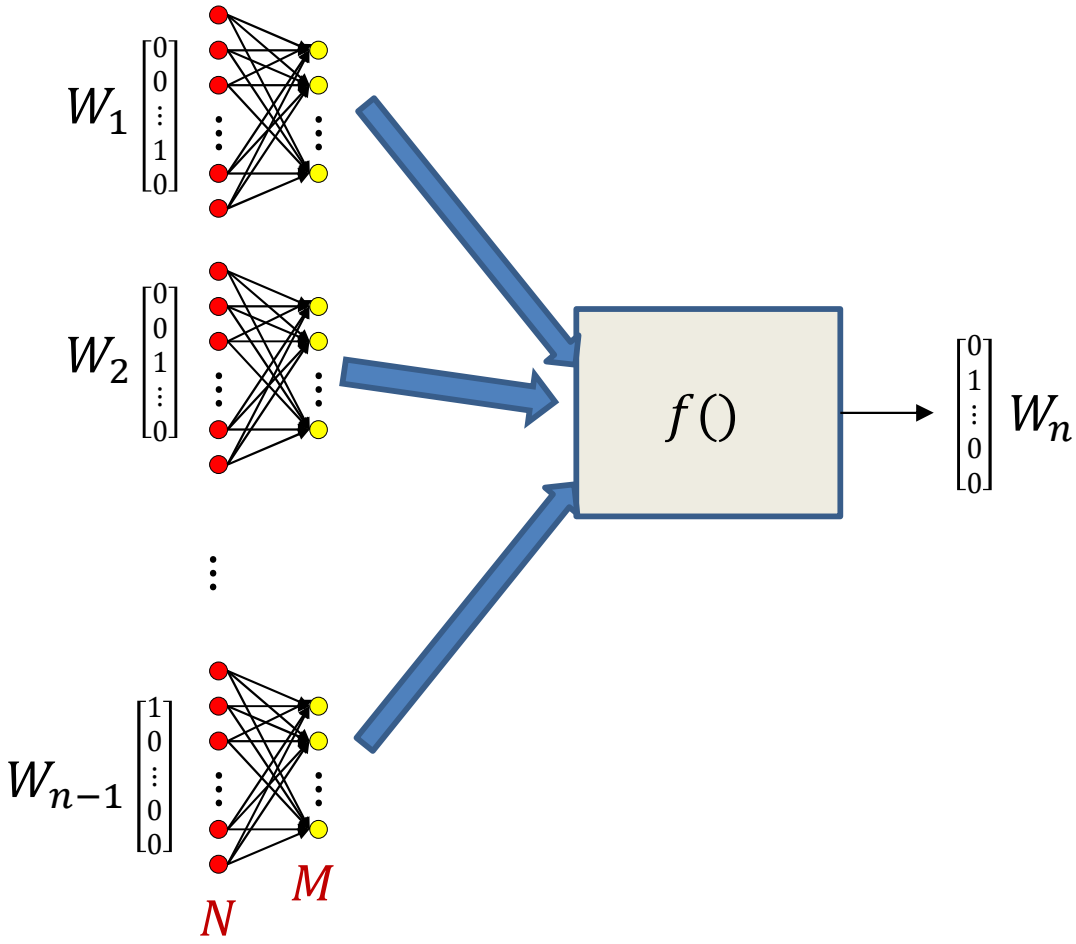


- *Project* the N-dimensional one-hot word vectors into a lower-dimensional space
  - Replace every one-hot vector W_i by PW_i
  - P is an $M \times N$ matrix
  - PW_i is now an M-dimensional vector
  - *Learn P* using an appropriate objective
    - Distances in the projected space will reflect relationships imposed by the objective

"Projection"

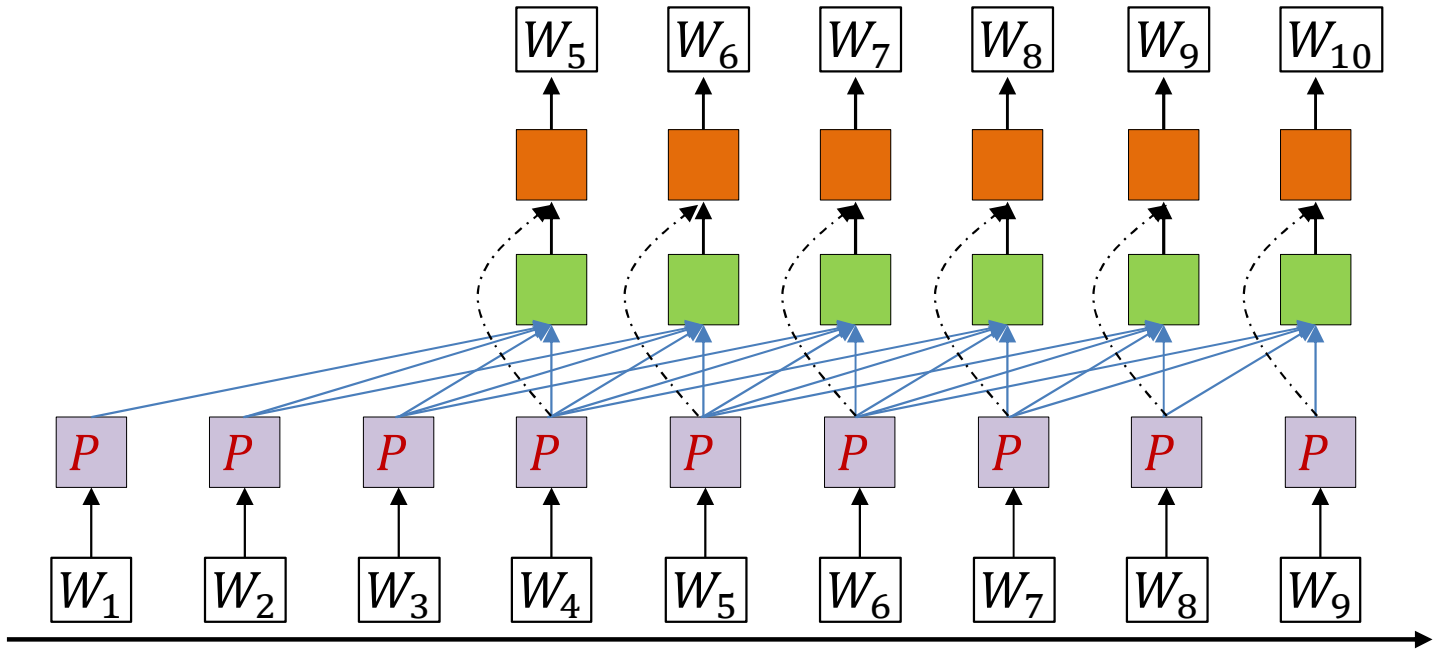
$$W_n = f(PW_1, PW_2, \ldots, PW_{n-1})$$





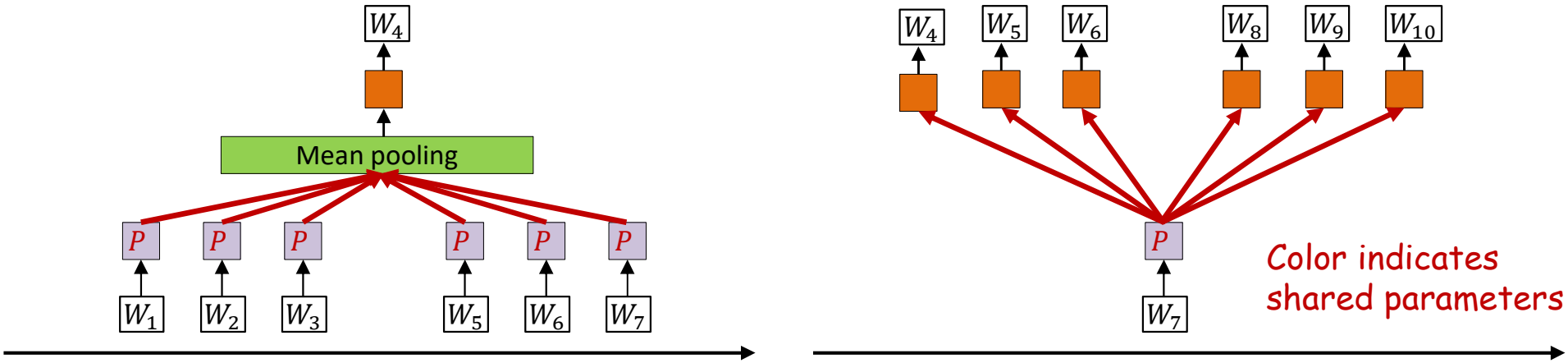
- *P* is a simple linear transform
- A single transform can be implemented as a layer of M neurons with linear activation
- The transforms that apply to the individual inputs are all M-neuron linear-activation subnets with tied weights

Predicting words: The TDNN model



- Predict each word based on the past N words
  - "A neural probabilistic language model", Bengio et al. 2003
  - Hidden layer has Tanh() activation, output is softmax
- One of the outcomes of learning this model is that we also learn low-dimensional representations PW of words

Alternative models to learn projections



- Soft bag of words: Predict word based on words in immediate context
  - Without considering specific position
- Skip-grams: Predict adjacent words based on current word
- More on these in a future recitation?

Embeddings: Examples

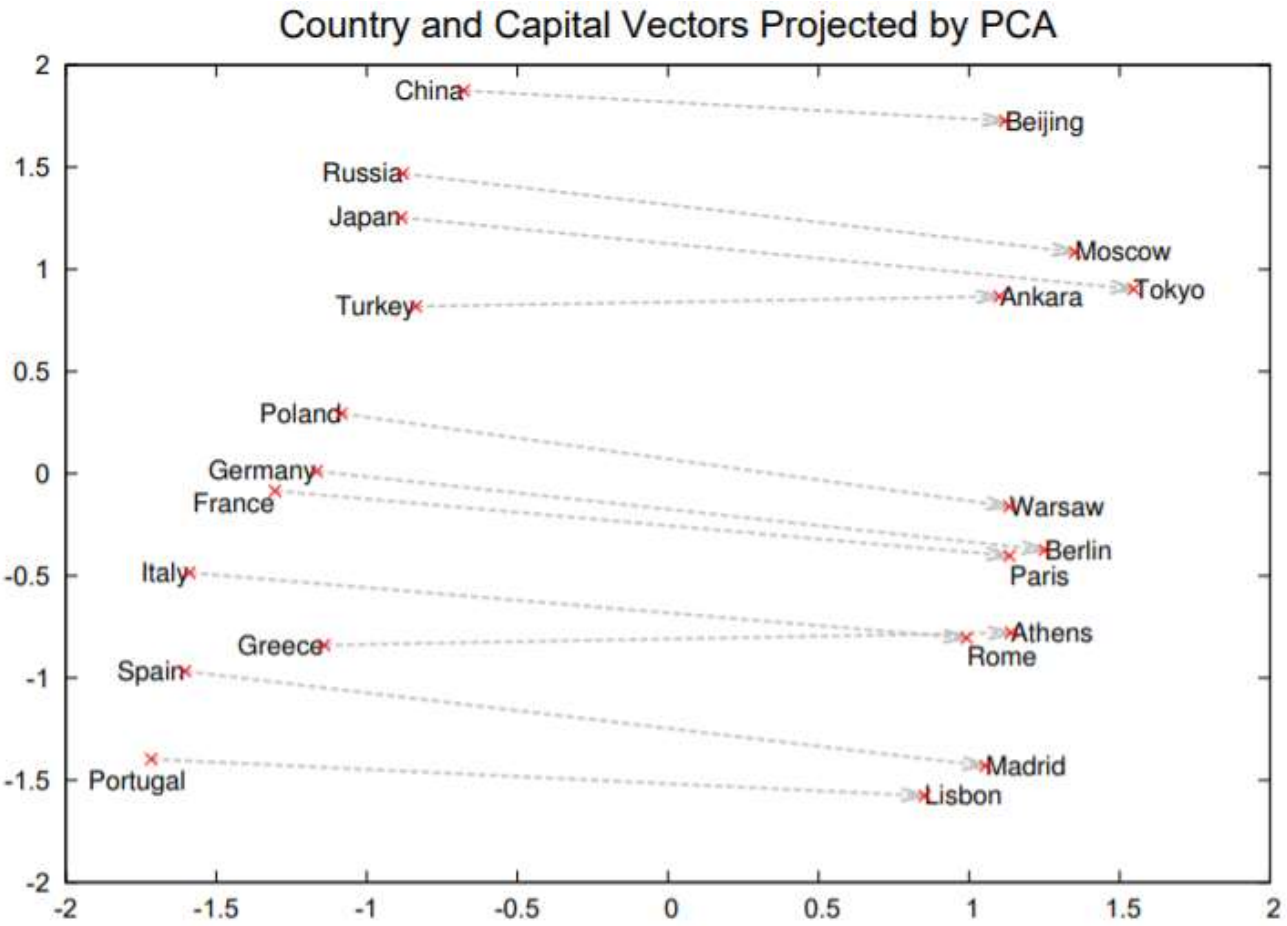


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

- From Mikolov et al., 2013, "Distributed Representations of Words and Phrases and their Compositionality"

Poll 4

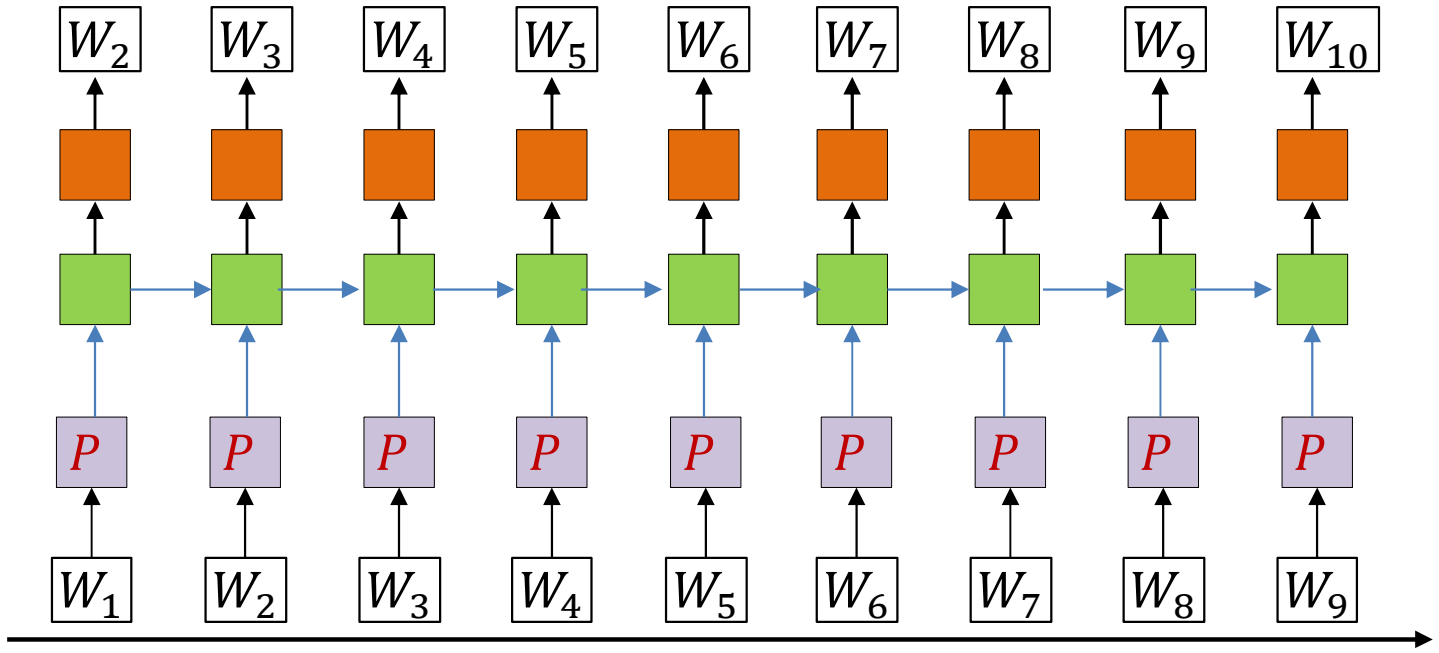
- @

Poll 4

Select all that are true

- **The distance between any two non-identical one-hot vectors is the same**
- **Words are represented as one-hot embeddings because these do not impose any a priori assumption about which words are closer than others**
- **Word embeddings derived from language models are lower-dimensional real-valued representations where the distance between words is a meaningful representation of their closeness**
- Low dimensional word embeddings enable you to find representations for words that were not part of your training vocabulary

Modelling language

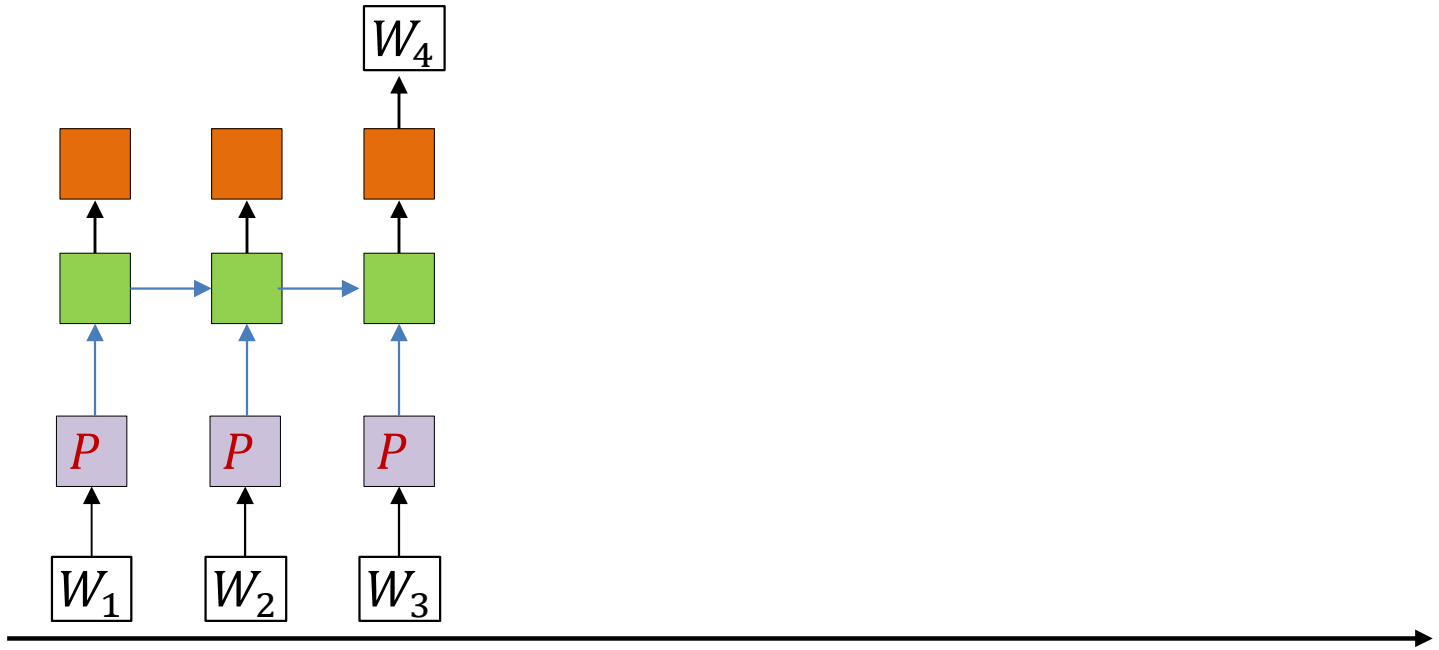


- The hidden units are (one or more layers of) LSTM units
- Trained via backpropagation from a lot of text
  - No explicit labels in the training data: at each time the next word is the label.

Generating Language: Synthesis

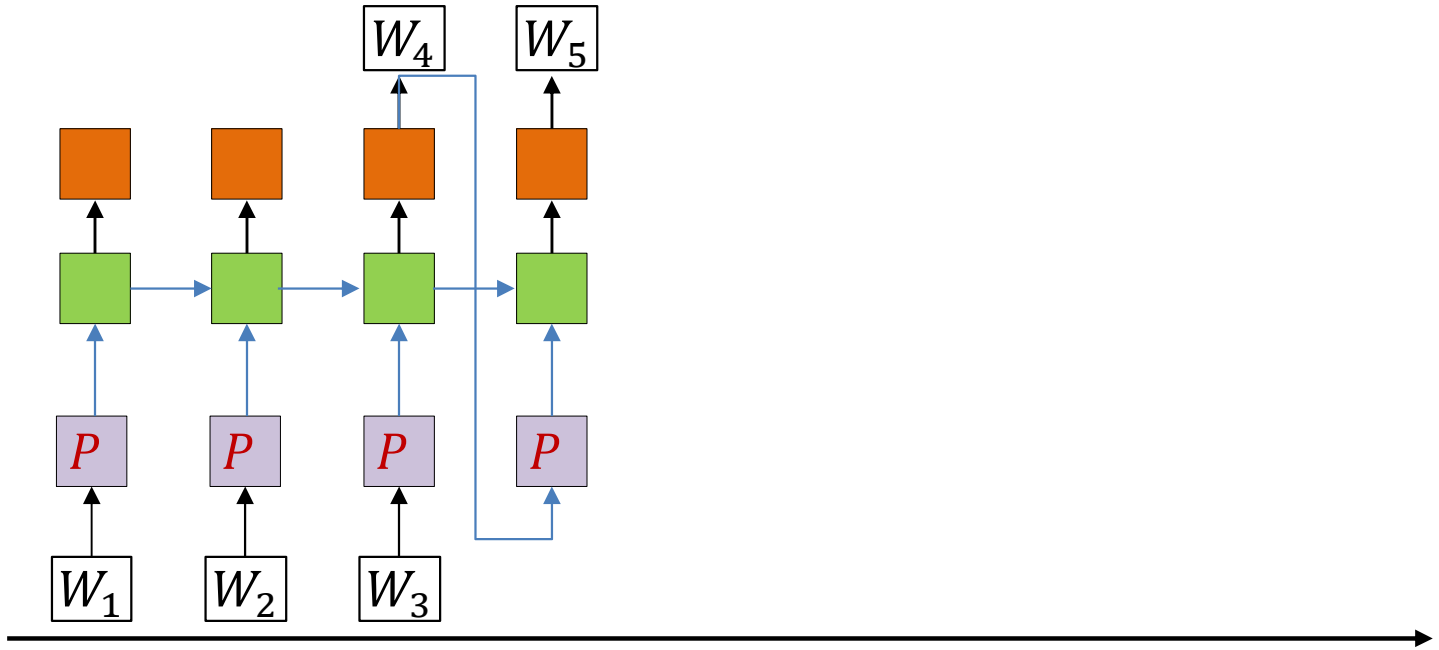
- On trained model : Provide the first few words
  - One-hot vectors
- After the last input word, the network generates a probability distribution over words
  - Outputs an N-valued probability distribution rather than a one-hot vector

Generating Language: Synthesis



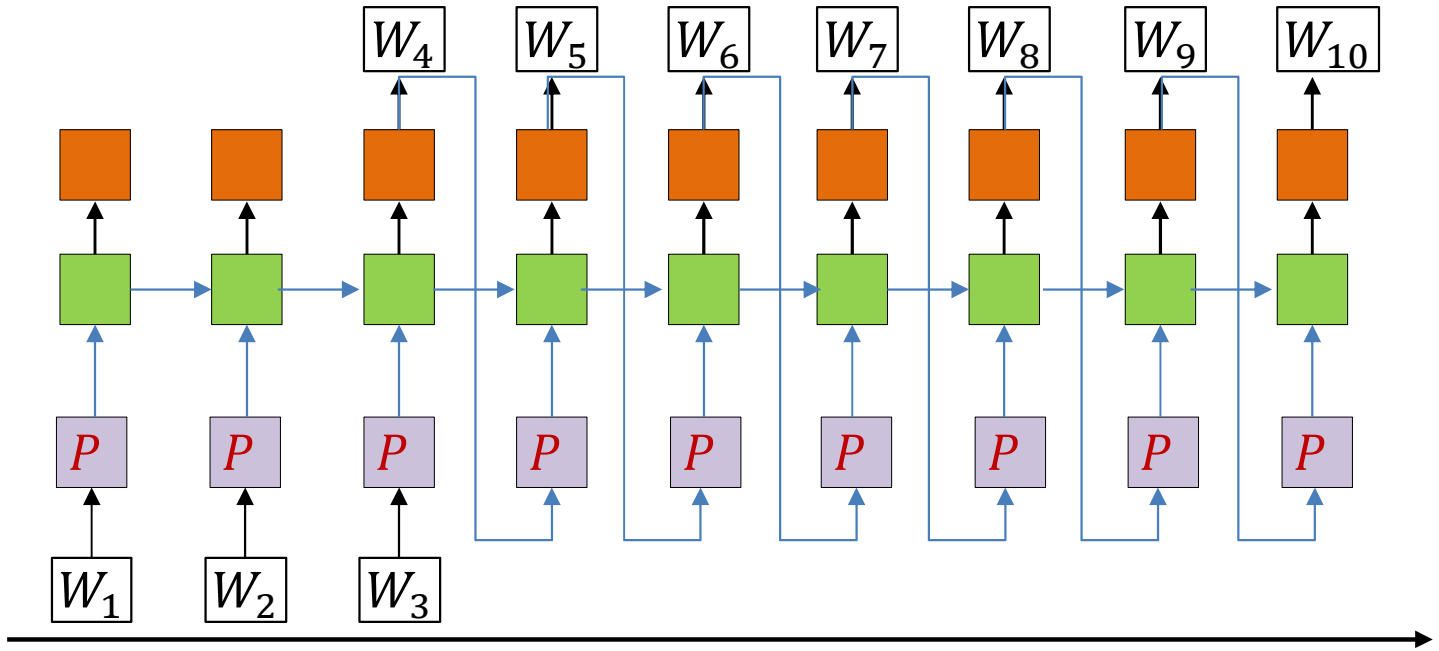
- On trained model : Provide the first few words
  - One-hot vectors
- After the last input word, the network generates a probability distribution over words
  - Outputs an N-valued probability distribution rather than a one-hot vector
- Draw a word from the distribution
  - And set it as the next word in the series

Generating Language: Synthesis



- Feed the drawn word as the next word in the series
  - And draw the next word from the output probability distribution

Generating Language: Synthesis



- Feed the drawn word as the next word in the series
  - And draw the next word from the output probability distribution
- Continue this process until we terminate generation
  - In some cases, e.g. generating programs, there may be a natural termination

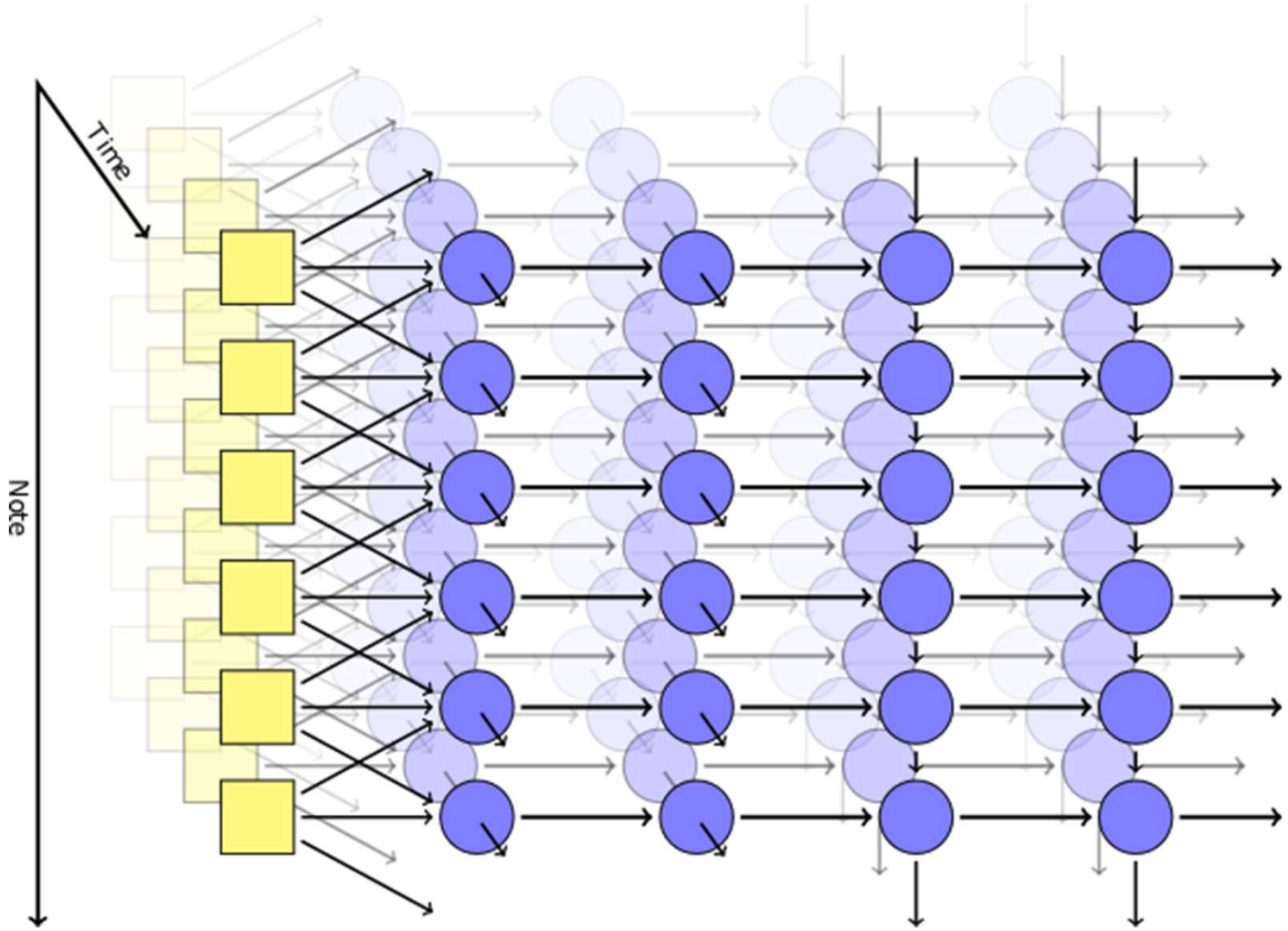
Which open source project?

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECON
  return segtable;
}
```

Trained on linux source code

Actually uses a *character-level* model (predicts character sequences)

Composing music with RNN



http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/

Returning to our problem

- Divergences are harder to define in other scenarios..
- … next class