Deep Learning
Recurrent Networks: Stability analysis and LSTMs

Recap: Story so far

- Recurrent networks retain information from the infinite past in principle
- In practice, they are poor at memorization
  - The hidden outputs can blow up, or shrink to zero depending on the Eigen values of the recurrent weights matrix
  - The memory is also a function of the activation of the hidden units
    - Tanh activations are the most effective at retaining memory, but even they don't hold it very long
- 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

Exploding/Vanishing gradients

$$h = f_N\left(\underline{W_N f_{N-1}}\left(\underline{W_{N-2} f_{N-1}}\left(\dots \underline{W_1} X\right)\right)\right)$$

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

- The memory retention of the network depends on the behavior of the underlined terms
  - Which in turn depends on the parameters W 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*

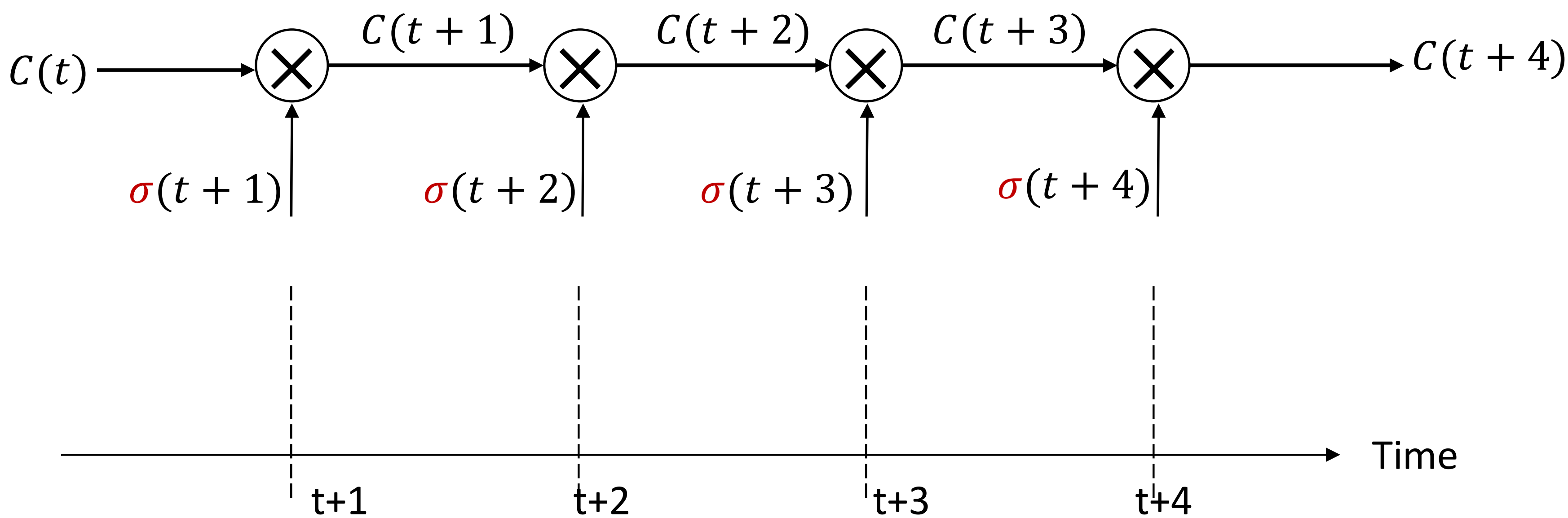
Exploding/Vanishing gradients

$$h = f_N\left(\underline{W_N f_{N-1}}\left(\underline{W_{N-2} f_{N-1}}\left(\ldots W_1 X\right)\right)\right)$$

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

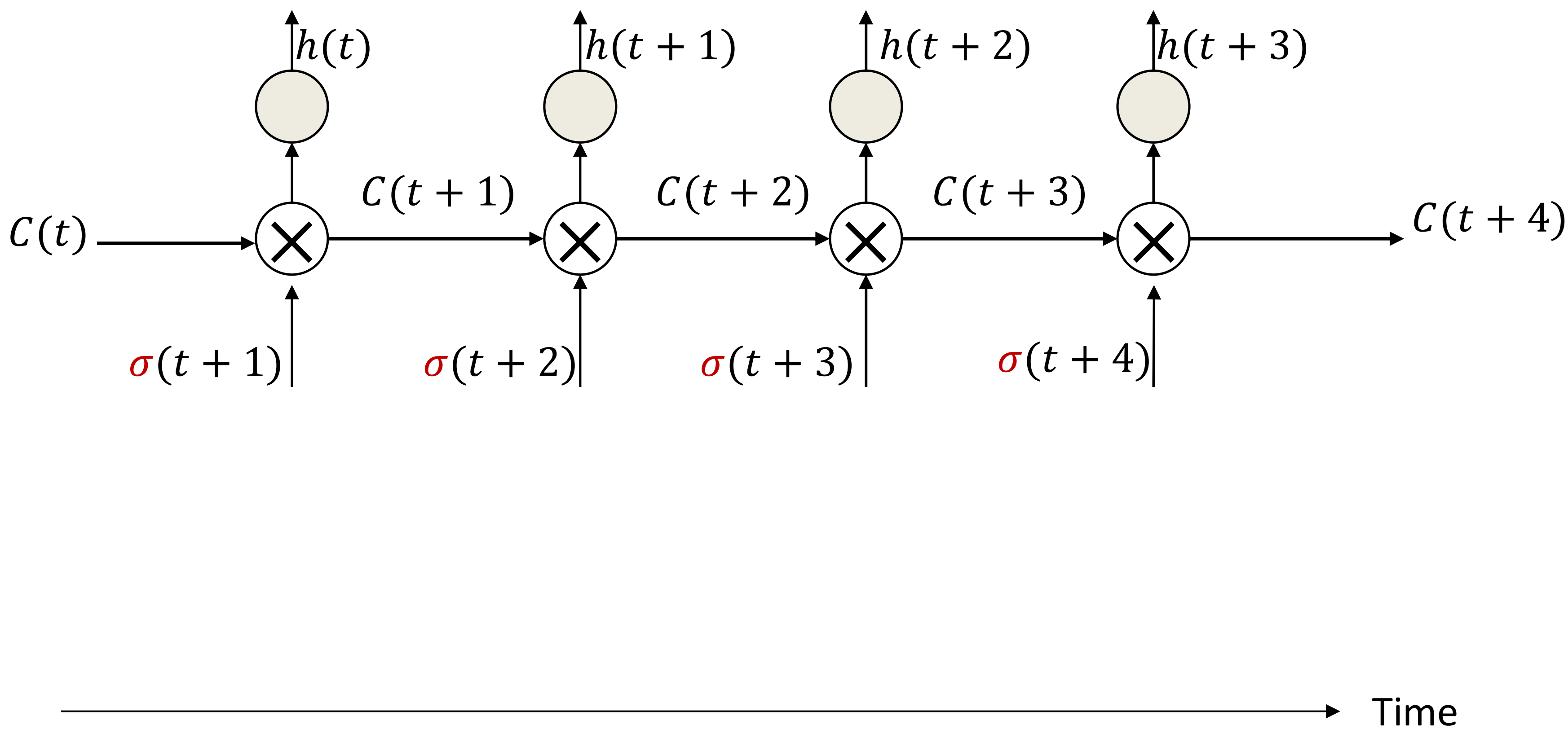
- Replace this with something that doesn't fade or blow up?
- Network that "retains" *useful* memory arbitrarily long, to be recalled on demand?
  - Input-based determination of *whether it must be remembered*
  - **Retain memories until a switch *based on the input* flags them as ok to forget**
    - Or remember less
  - $Memory(k) \approx C(x_0).\sigma_1(x_1).\sigma_2(x_2).\ldots\sigma_k(x_k)$

Enter – the constant error carousel



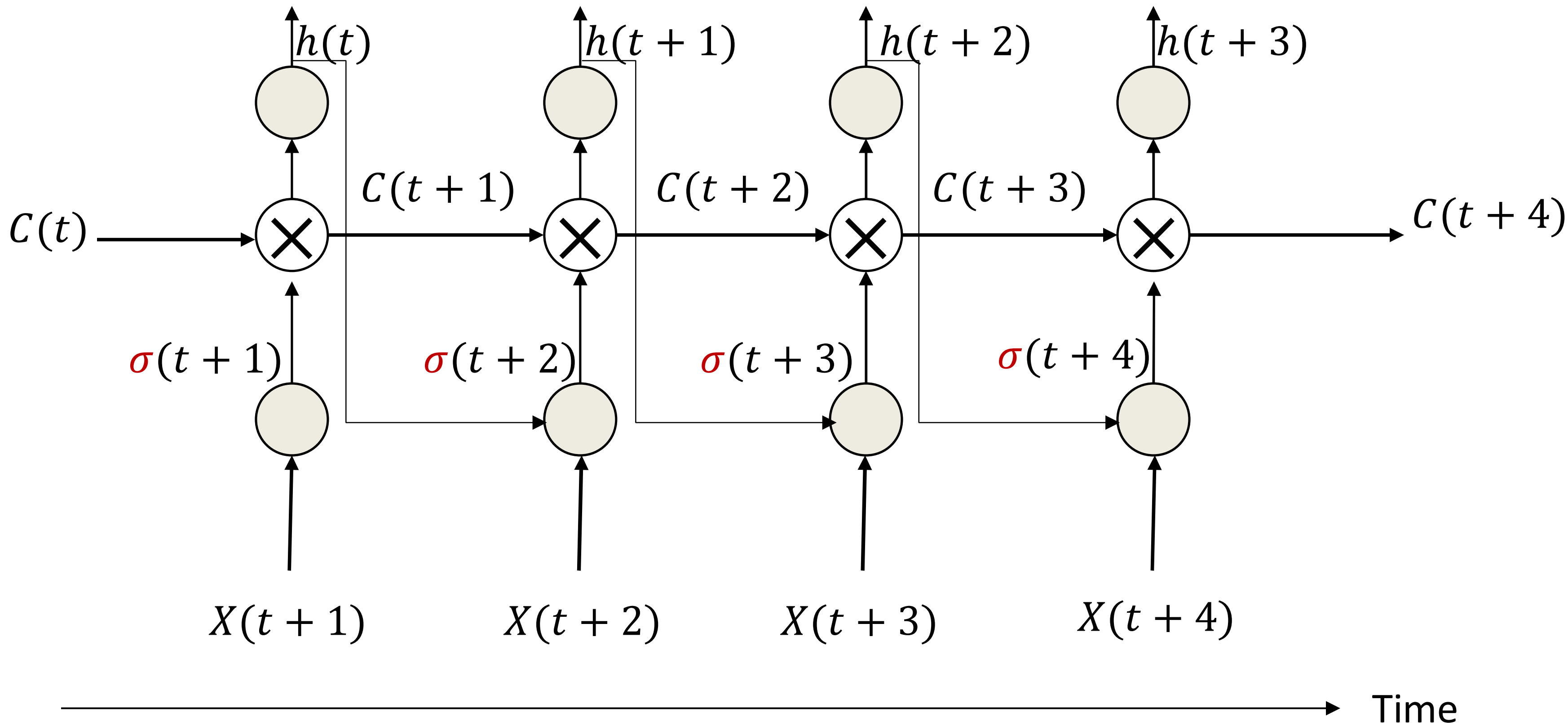
- History is carried through uncompressed
  - No weights, no nonlinearities
  - Only scaling is through the σ "gating" term that captures other triggers
  - E.g. "Have I seen Pattern2"?

Enter – the constant error carousel



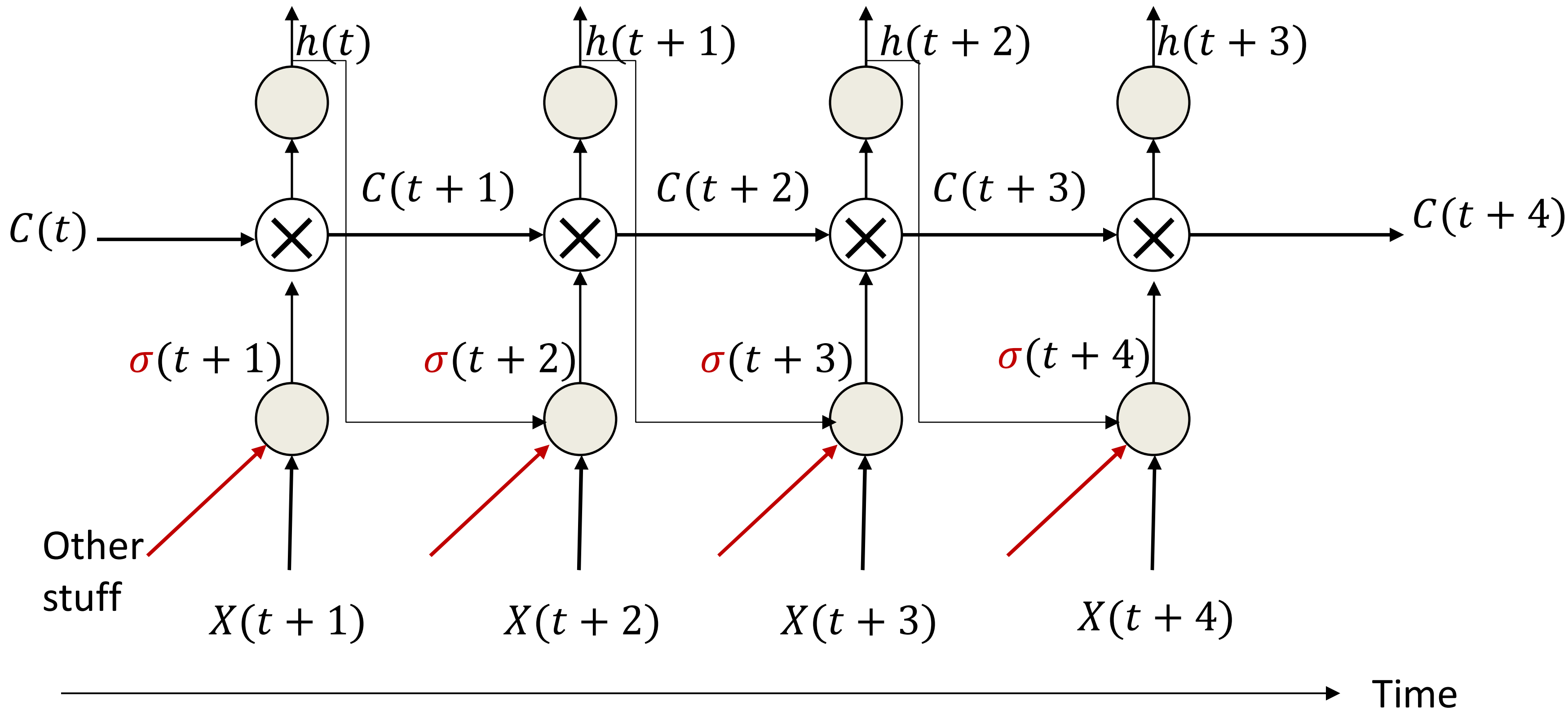
- Actual non-linear work is done by other portions of the network
  - Neurons that compute the workable state from the memory

Enter – the constant error carousel



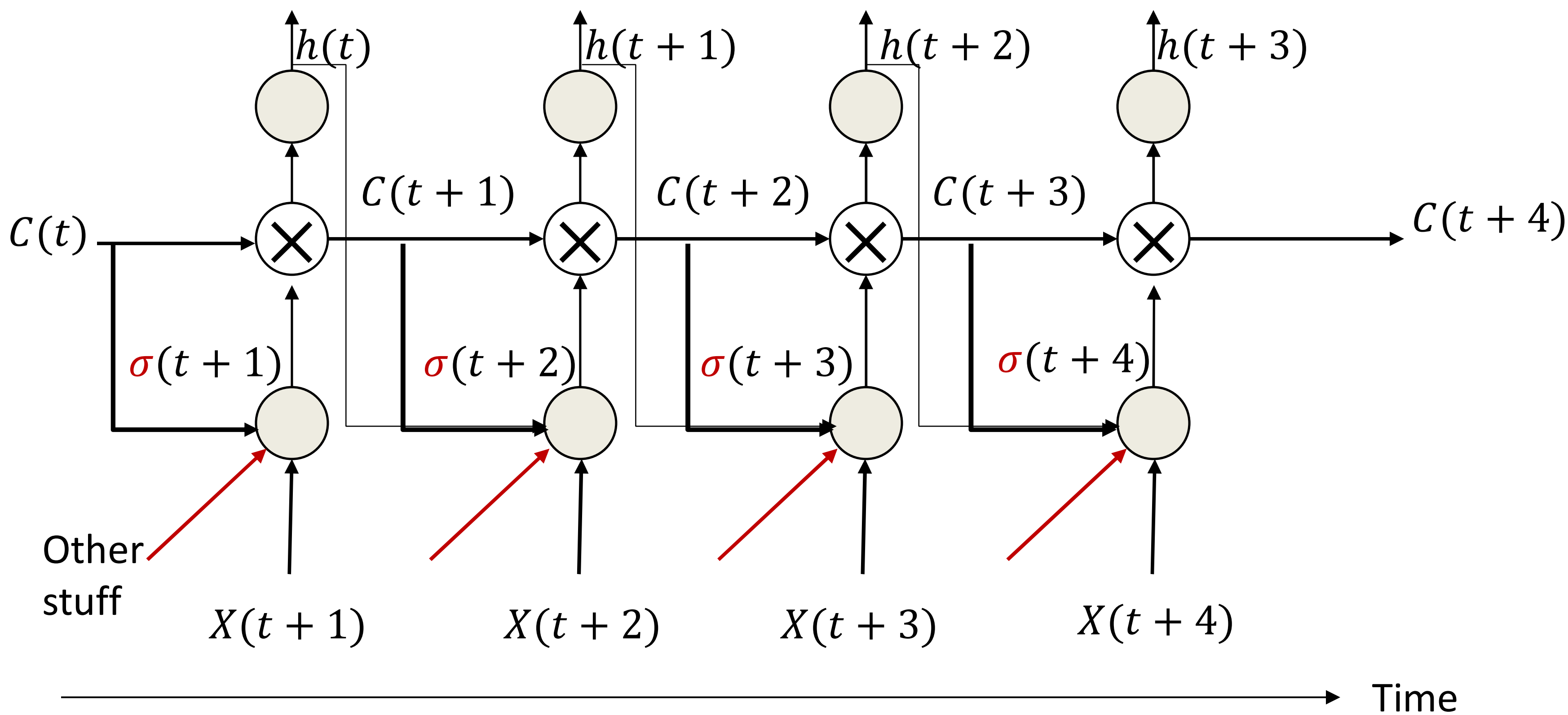
- The gate σ depends on current input, current hidden state…

Enter – the constant error carousel



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

Enter – the constant error carousel



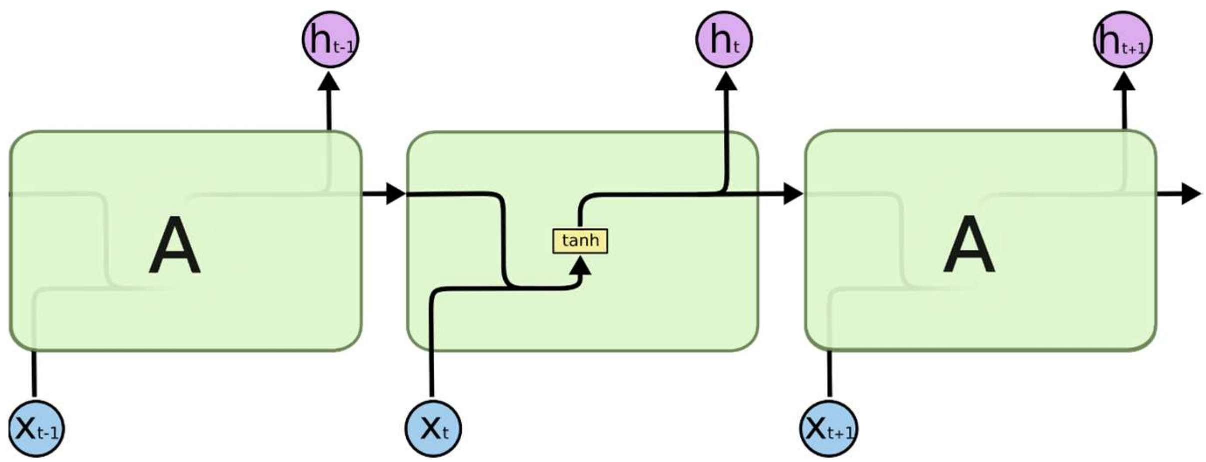
- The gate σ depends on current input, current hidden state… and other stuff…
- Including, obviously, what is currently in raw memory

Enter the *LSTM*

- *Long Short-Term Memory*
- Explicitly latch information to prevent decay / blowup

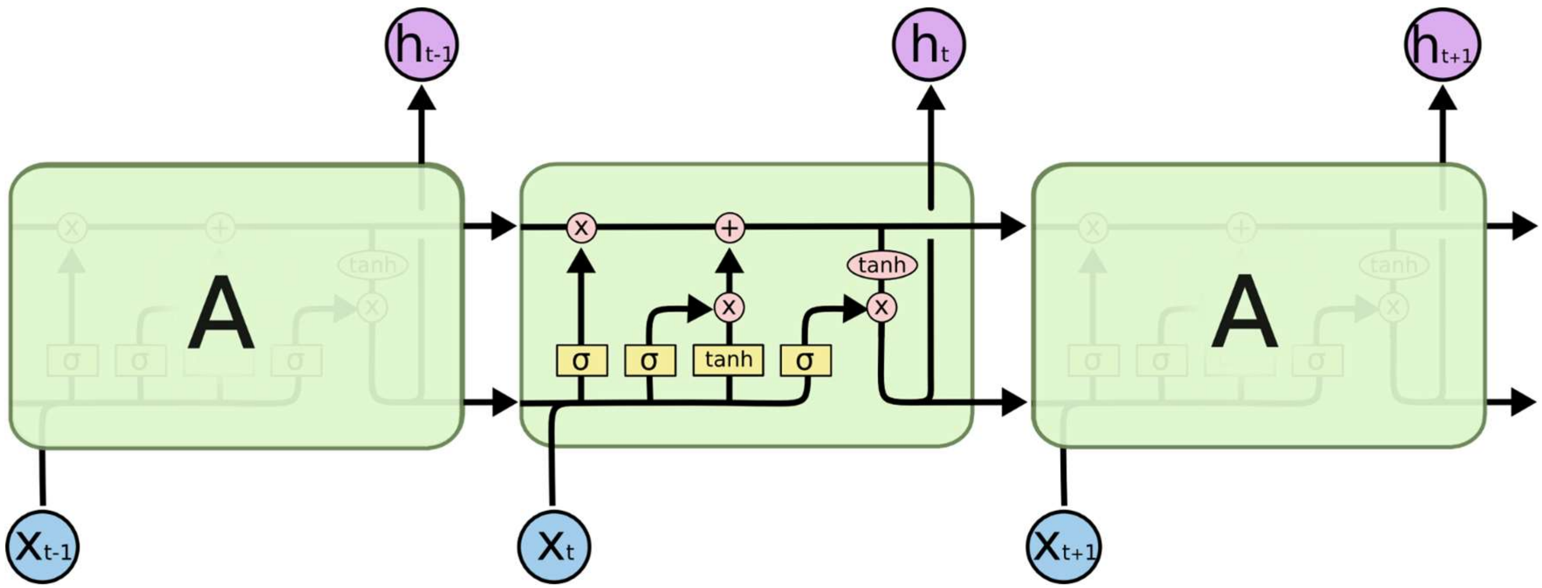
- Following notes borrow liberally from
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Standard RNN



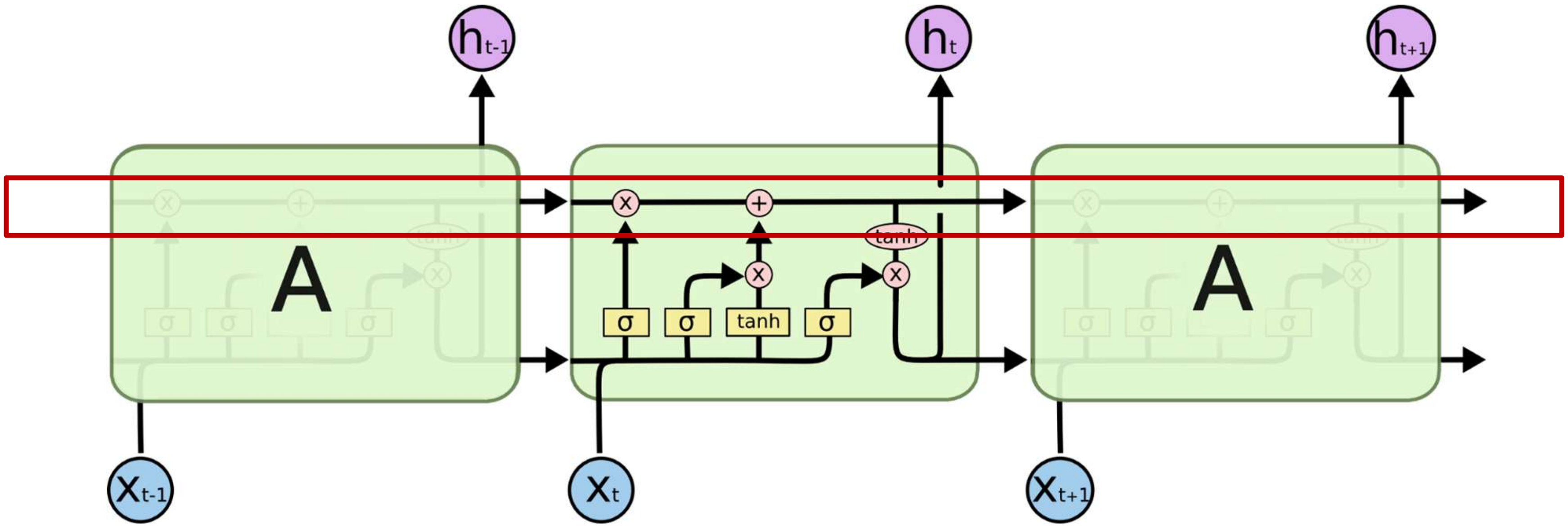
- Recurrent neurons receive past recurrent outputs and current input as inputs
- Processed through a tanh() activation function
  - As mentioned earlier, tanh() is the generally used activation for the hidden layer
- Current recurrent output passed to next higher layer and next time instant

Long Short-Term Memory



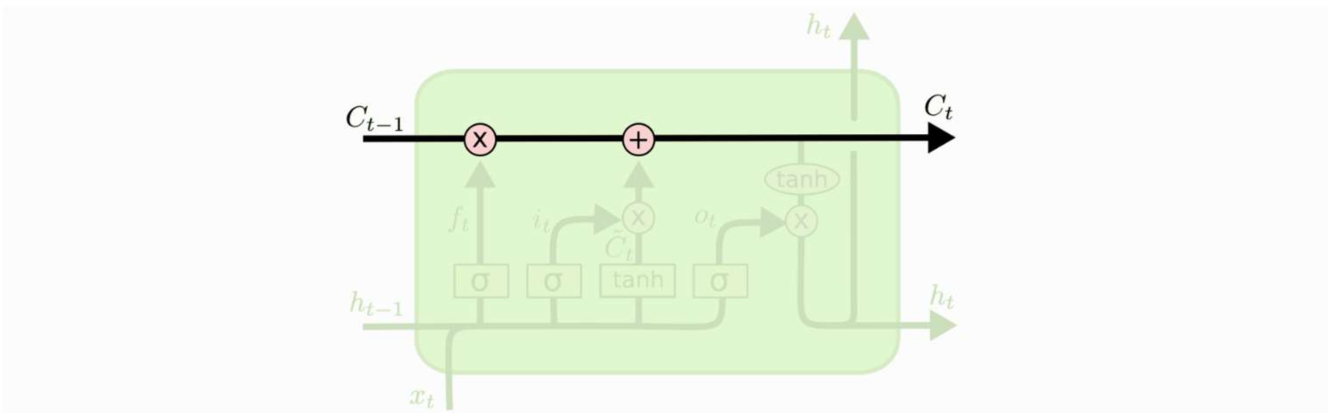
- The $\sigma()$ are *multiplicative gates* that decide if something is important or not
- Remember, every line actually represents a *vector*

LSTM: Constant Error Carousel



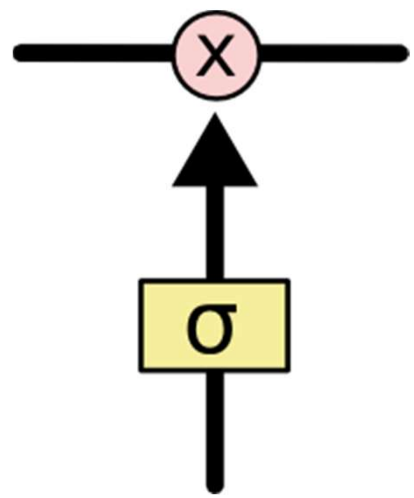
- Key component: a *remembered cell state*

LSTM: CEC



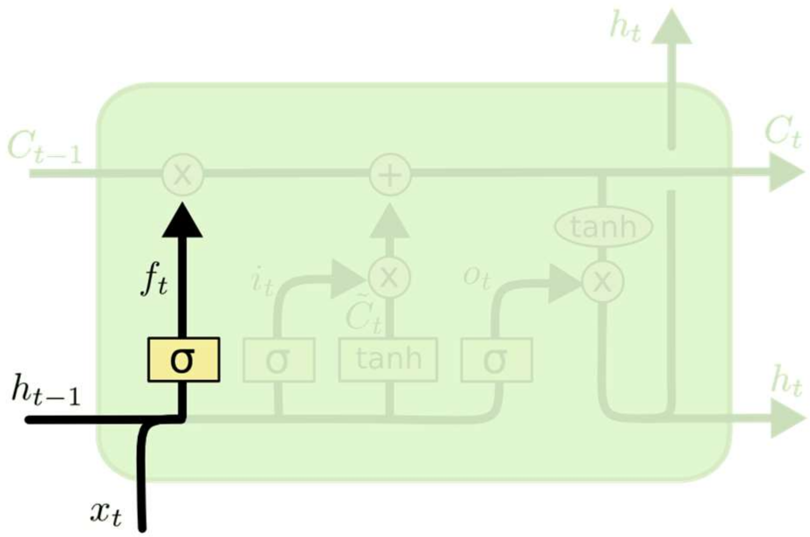
- C_t is the linear history carried by the *constant-error carousel*
- Carries information through, only affected by a gate
  - And *addition of history,* which too is gated..

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

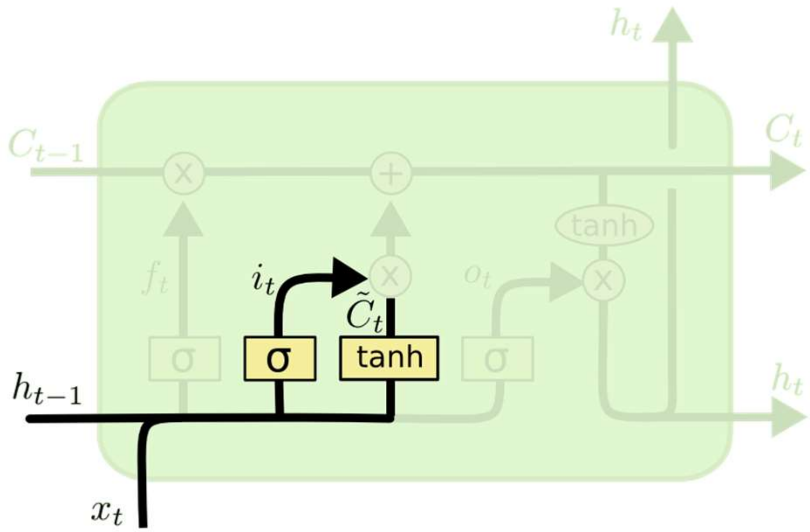
LSTM: Forget gate



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- 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

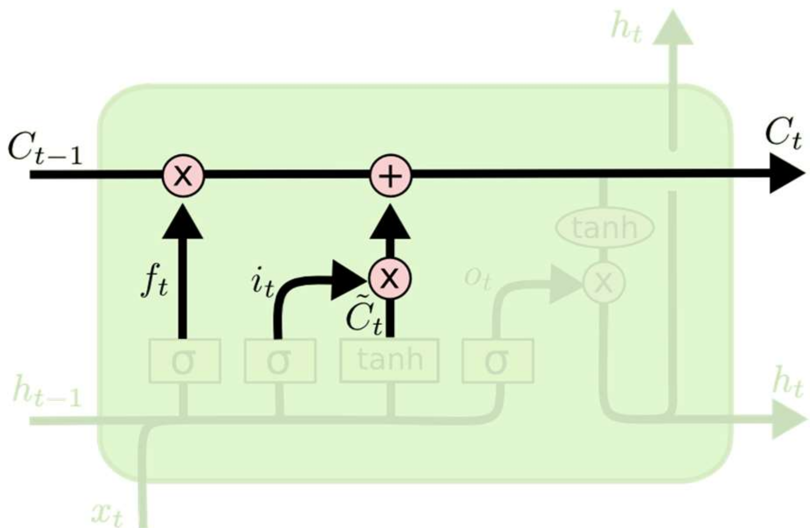


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

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

- 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

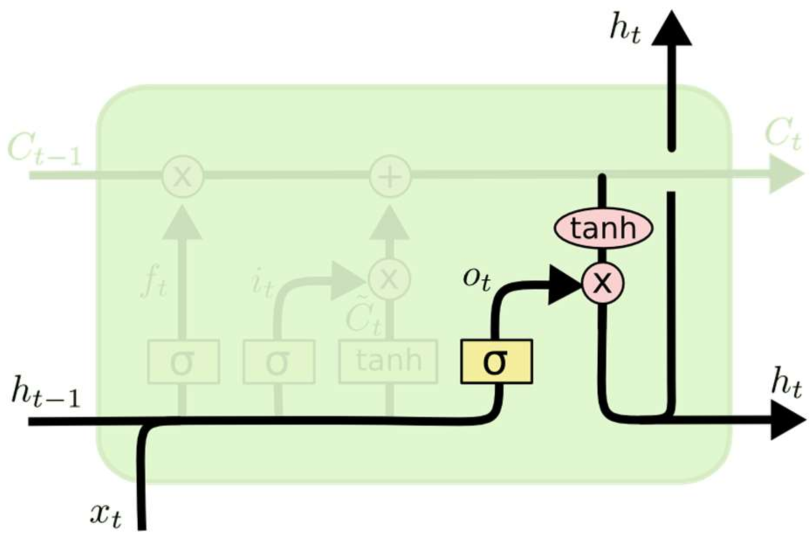
LSTM: Memory cell update



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

- 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

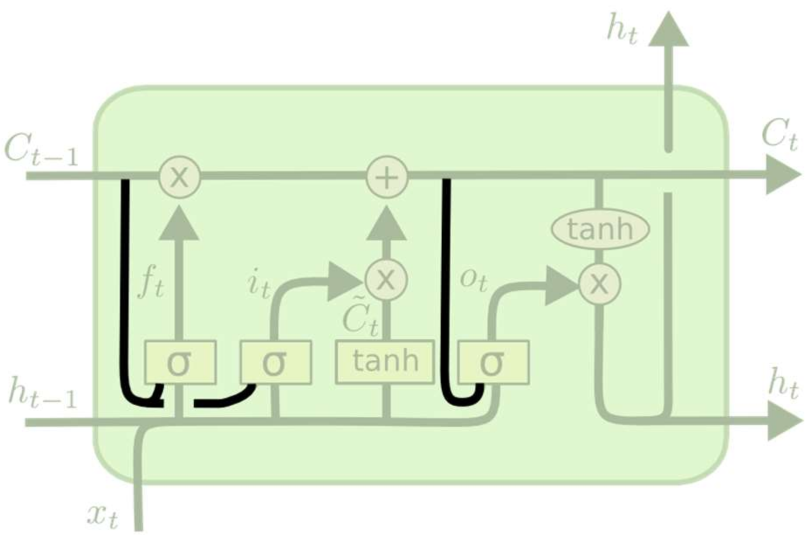


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

- 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

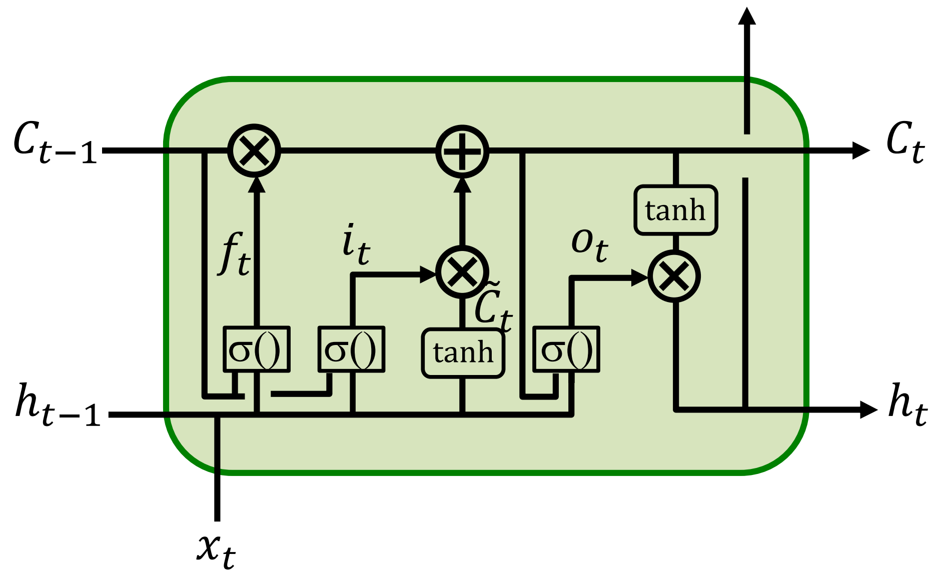
LSTM: The "Peephole" Connection



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

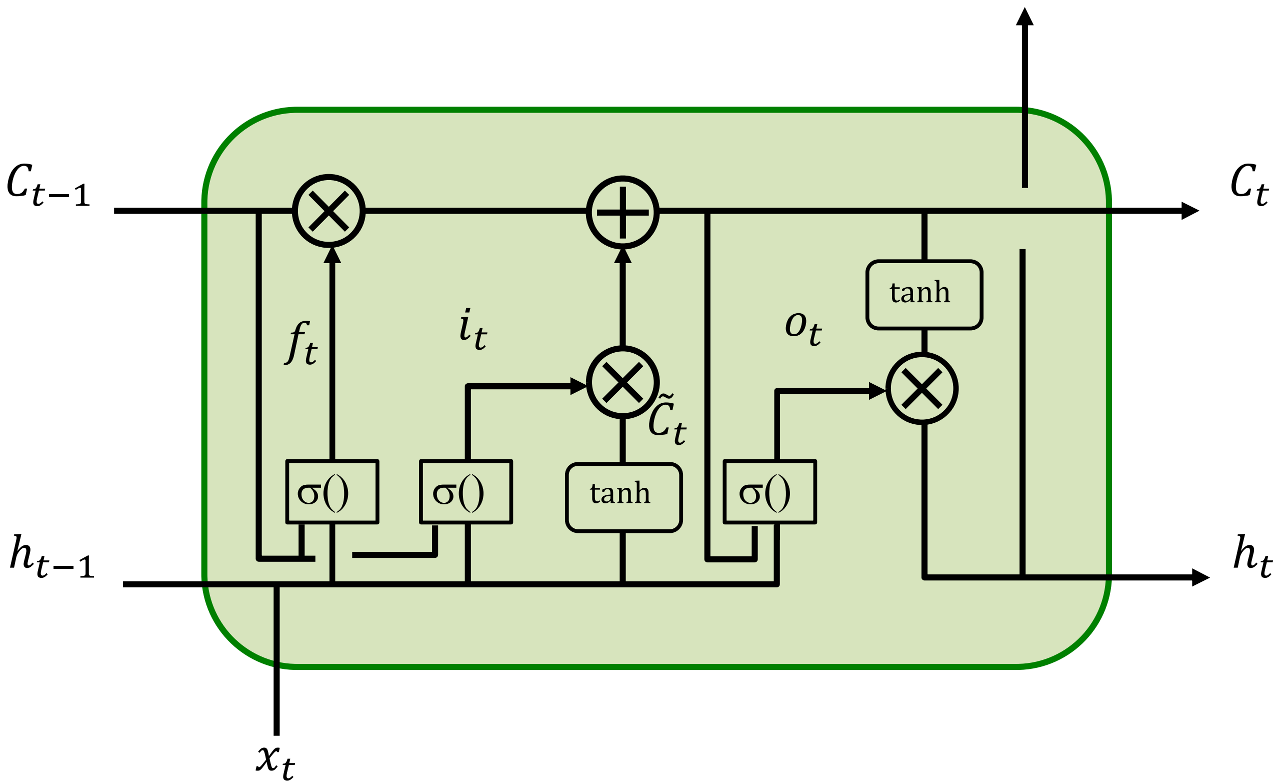
- The raw memory is informative by itself and can also be input
  - Note, we're using both C and h

The complete LSTM unit



- With input, output, and forget gates and the peephole connection..

LSTM computation: Forward



- Forward rules:

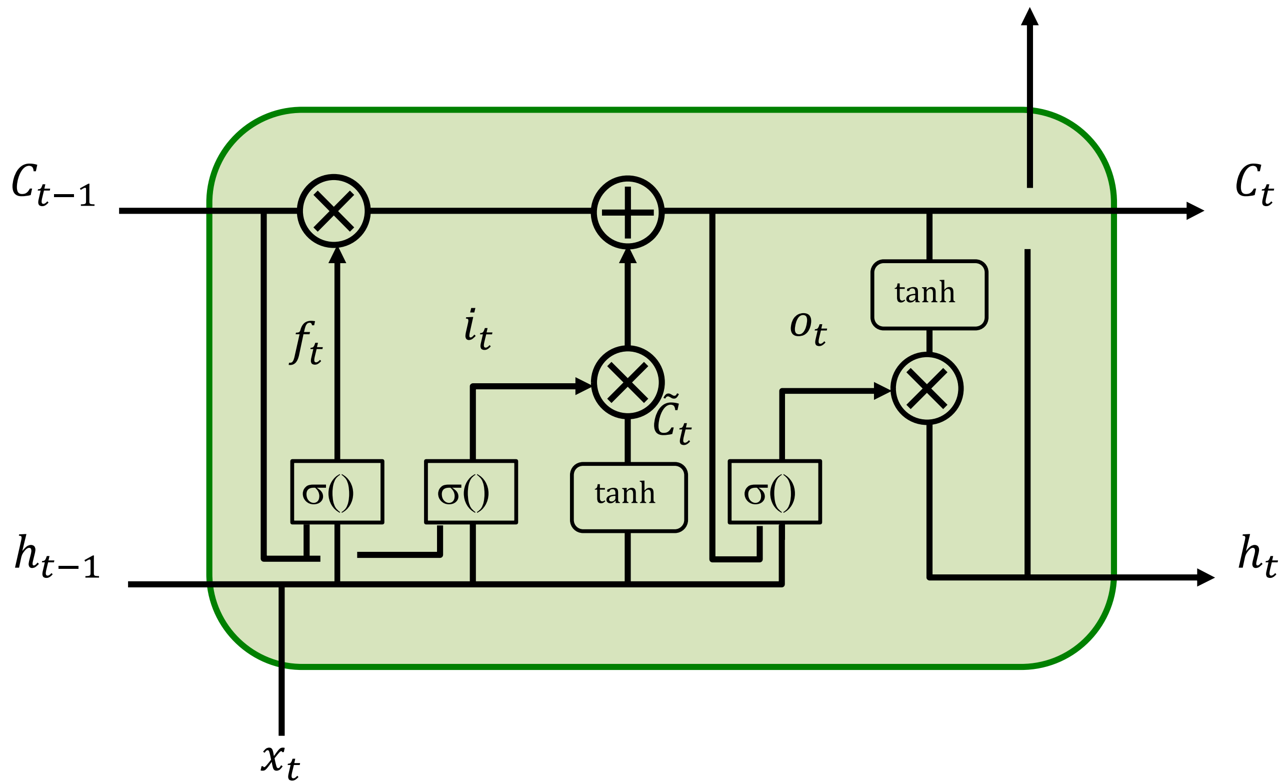
Gates

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

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

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

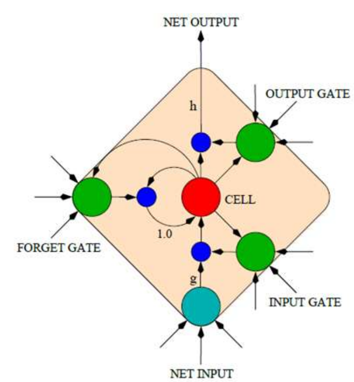
Variables

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

LSTM Equations

- $i = \sigma(x_t U^i + s_{t-1} W^i)$
- $f = \sigma(x_t U^f + s_{t-1} W^f)$
- $o = \sigma(x_t U^o + s_{t-1} W^o)$
- $g = \tanh(x_t U^g + s_{t-1} W^g)$
- $c_t = c_{t-1} \circ f + g \circ i$
- $s_t = \tanh(c_t) \circ o$
- $y = softmax(V s_t)$

- i: **input gate, how much of the new information will be let through the memory cell.**
- f: **forget gate, responsible for information should be thrown away from memory cell.**
- o: **output gate, how much of the information will be passed to expose to the next time step.**
- g: **self-recurrent which is equal to standard RNN**
- c_t: **internal memory of the memory cell**
- s_t: **hidden state**
- **y: final output**



LSTM Memory Cell

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 zf, zi, zc, zo, f, i, o, Ci
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 zf, zi, zc, zo, f, i, o, Ci
function [Co, ho] = LSTM_cell.forward(C,h,x, [W,b])
    zf = WfcC + Wfhh + Wfxx + bf
    f = sigmoid(zf) # forget gate

    zi = WicC + Wihh + Wixx + bi
    i = sigmoid(zi) # input gate

    zc = WccC + Wchh + Wcxx + bc
    Ci = tanh(zc)  # Detecting input pattern

    Co = f∘C + i∘Ci # "∘" is component-wise multiply

    zo = WocCo + Wohh + Woxx + bo
    o = sigmoid(zo) # output gate

    ho = o∘tanh(Co) # "∘" is component-wise multiply

    return Co,ho
```

LSTM network forward

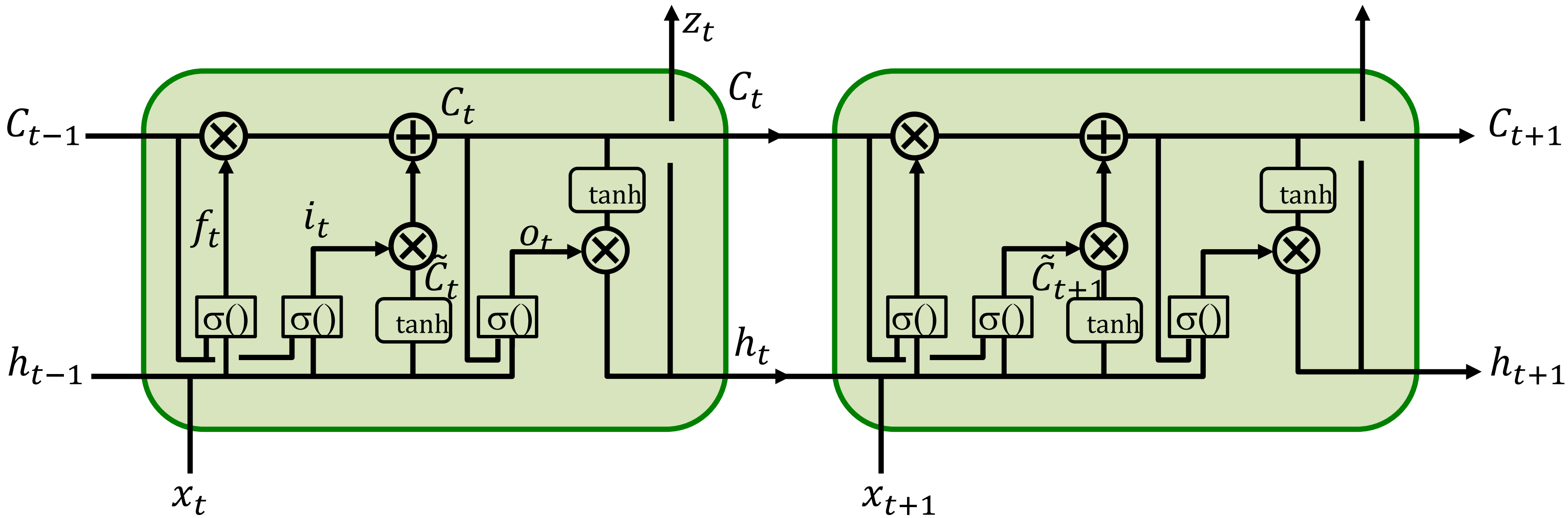
```
# Assuming h(-1,*) is known and C(-1,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
#              for the lth hidden layer
# Wo and bo are output layer weights and biases

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}])
    zo(t) = Woh(t,L) + bo
    Y(t) = softmax( zo(t) )
```

Training the LSTM

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

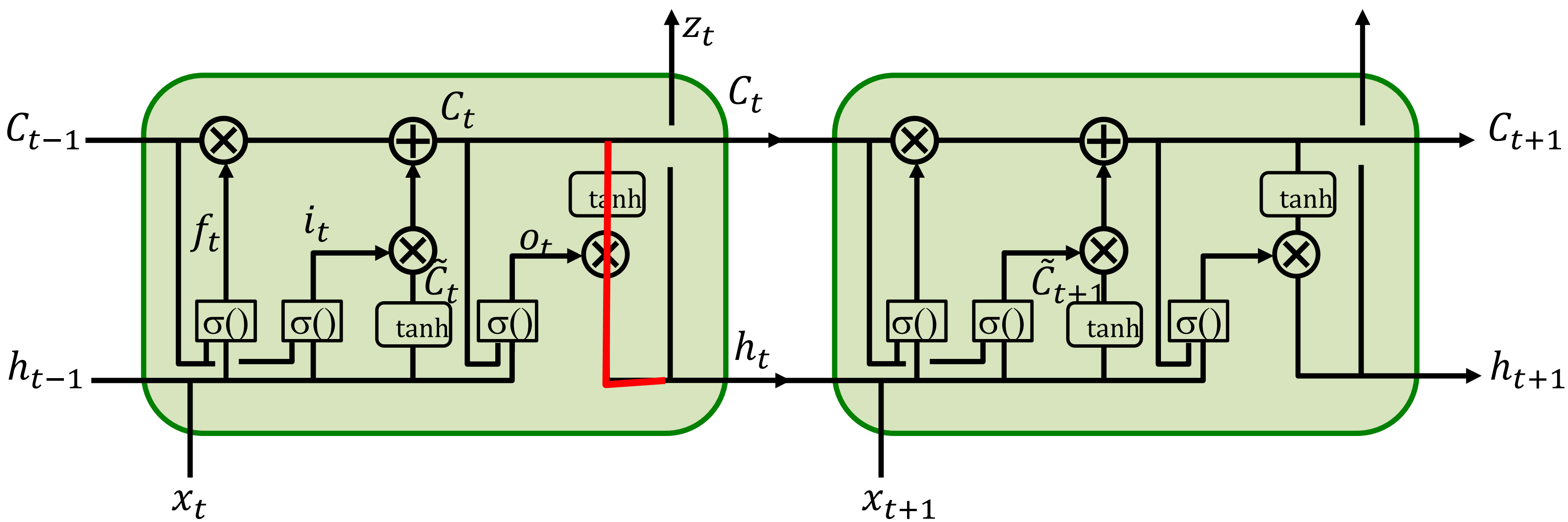
- Difference: Instead of backpropagating gradients through an RNN unit, we will backpropagate through an LSTM cell

Backpropagation rules: Backward



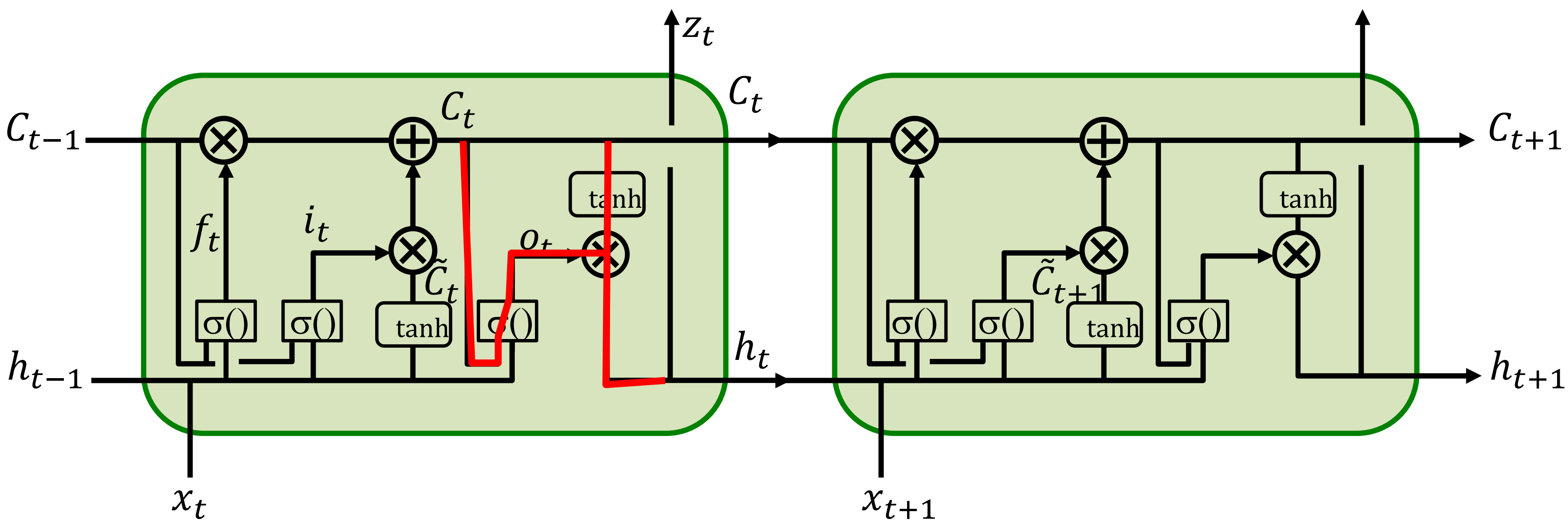
$$\nabla_{C_t} Div =$$

Backpropagation rules: Backward



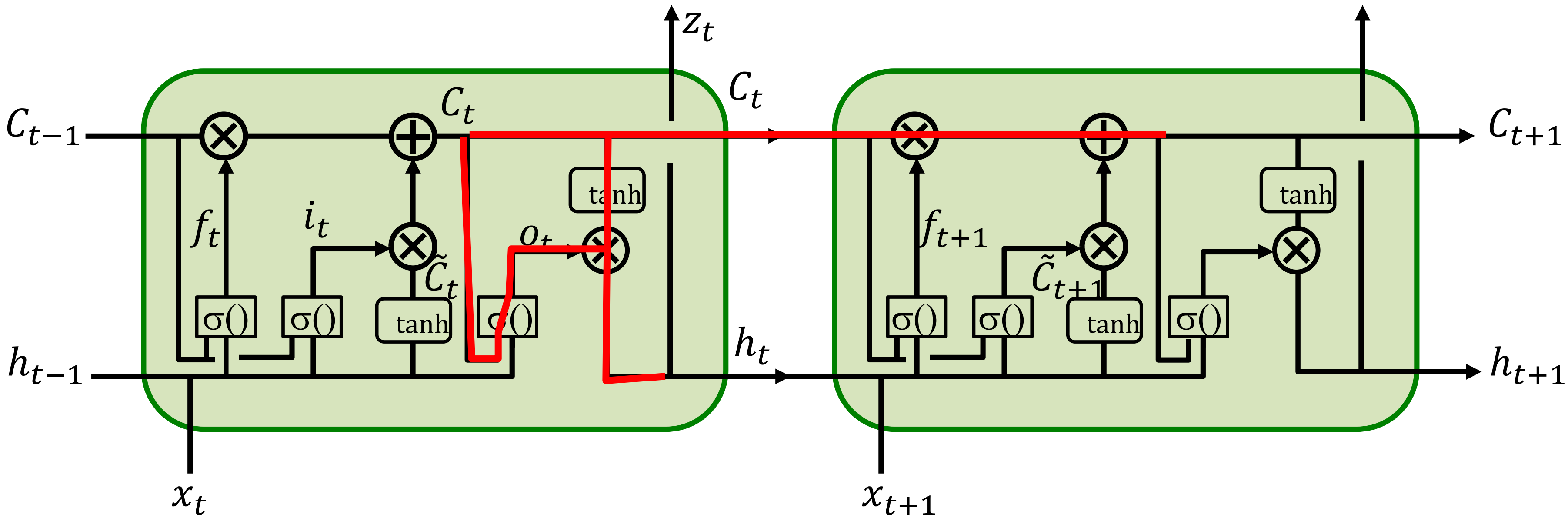
$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ o_t \circ tanh'(.)$$

Backpropagation rules: Backward



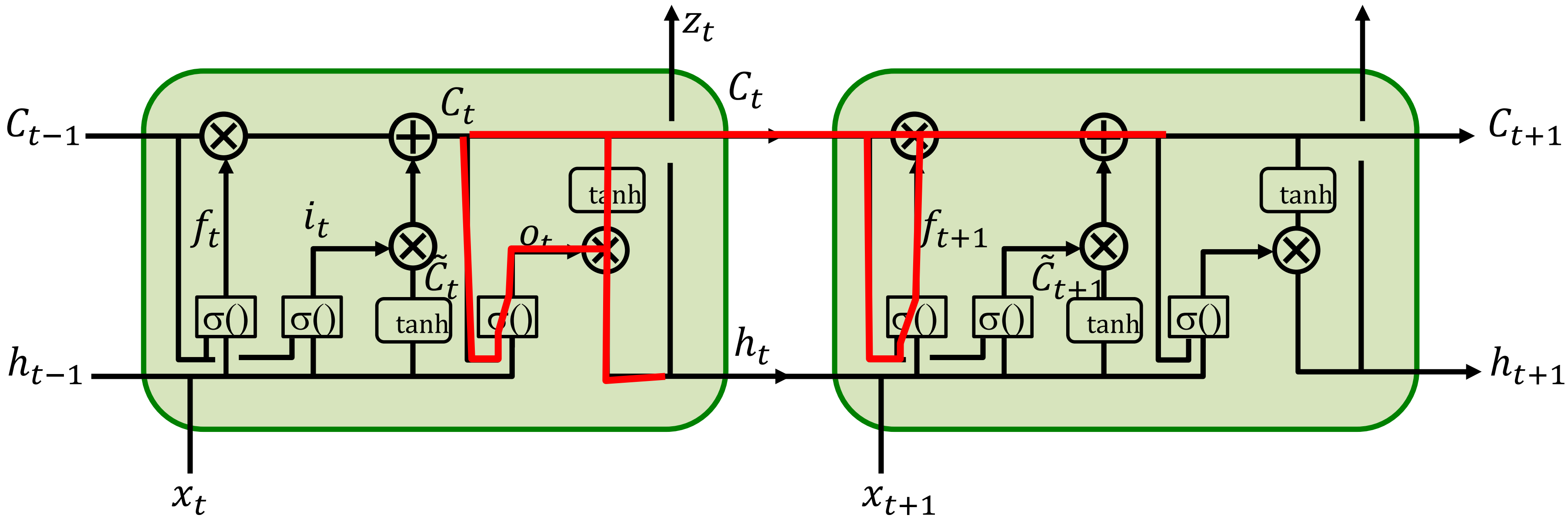
$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ tanh'(.) + tanh(.) \circ \sigma'(.) W_{Co})$$

Backpropagation rules: Backward



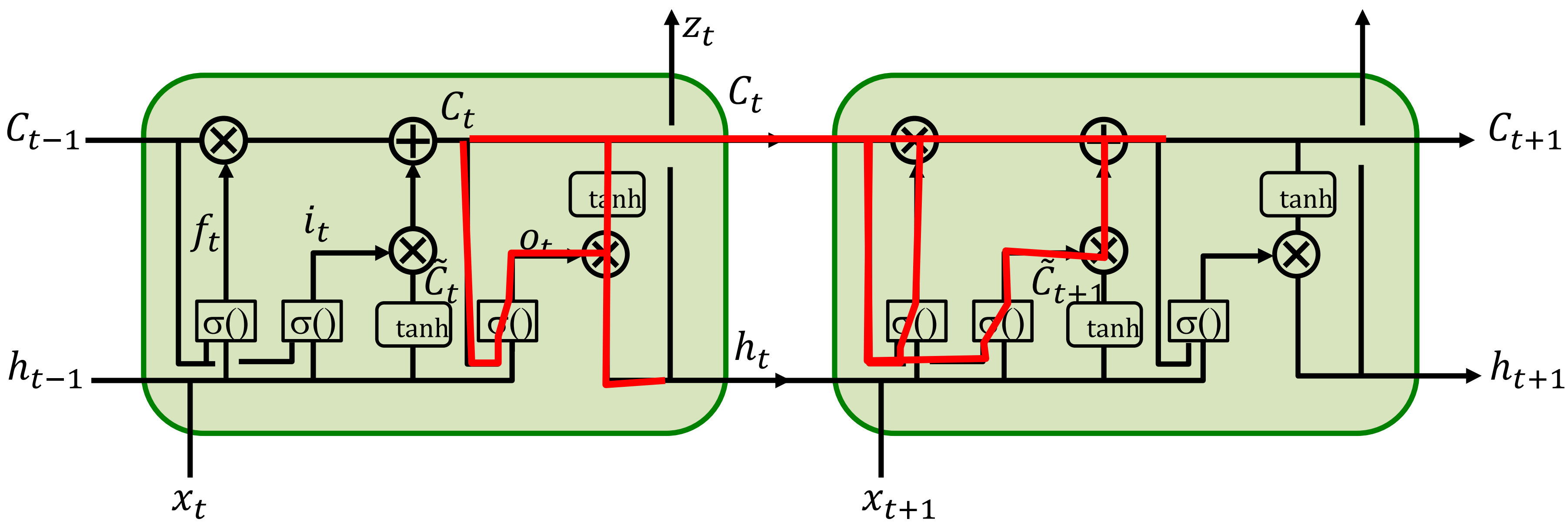
$\nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ tanh'(.) + tanh(.) \circ \sigma'(.) W_{Co}) +$
$\nabla_{C_{t+1}} Div \circ f_{t+1} +$

Backpropagation rules: Backward



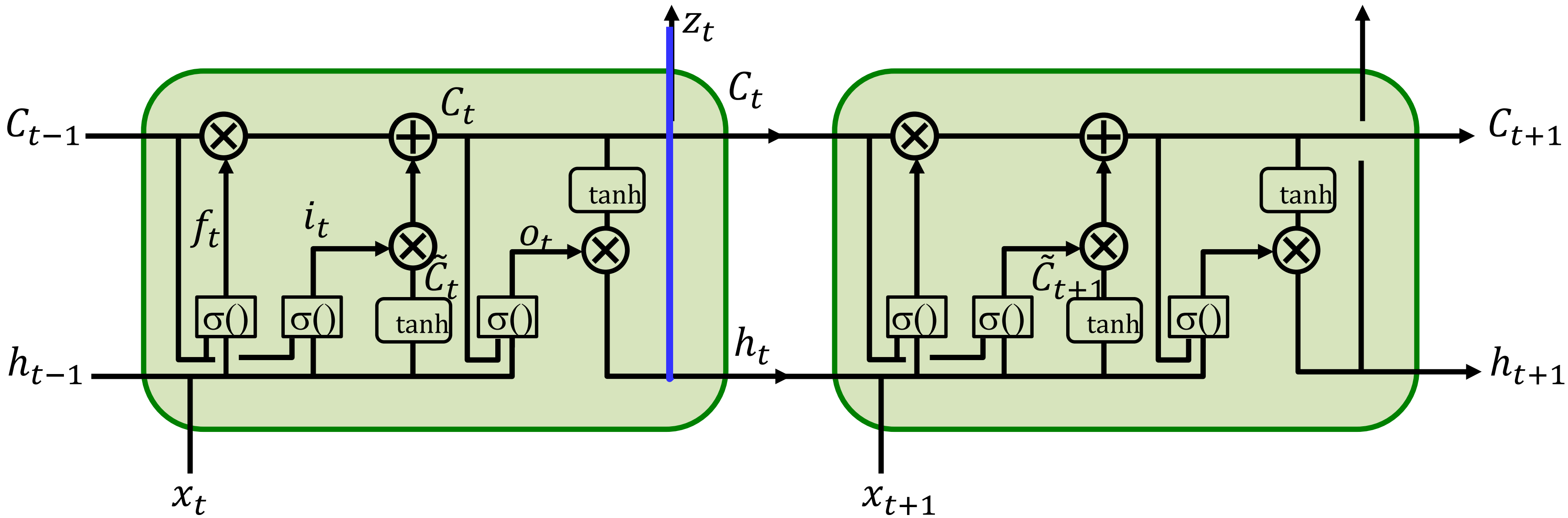
$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ tanh'(.) + tanh(.) \circ \sigma'(.) W_{Co}) +$$
$$\nabla_{C_{t+1}} Div \circ \left(f_{t+1} + C_t \circ \sigma'(.) W_{Cf}\right)$$

Backpropagation rules: Backward



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

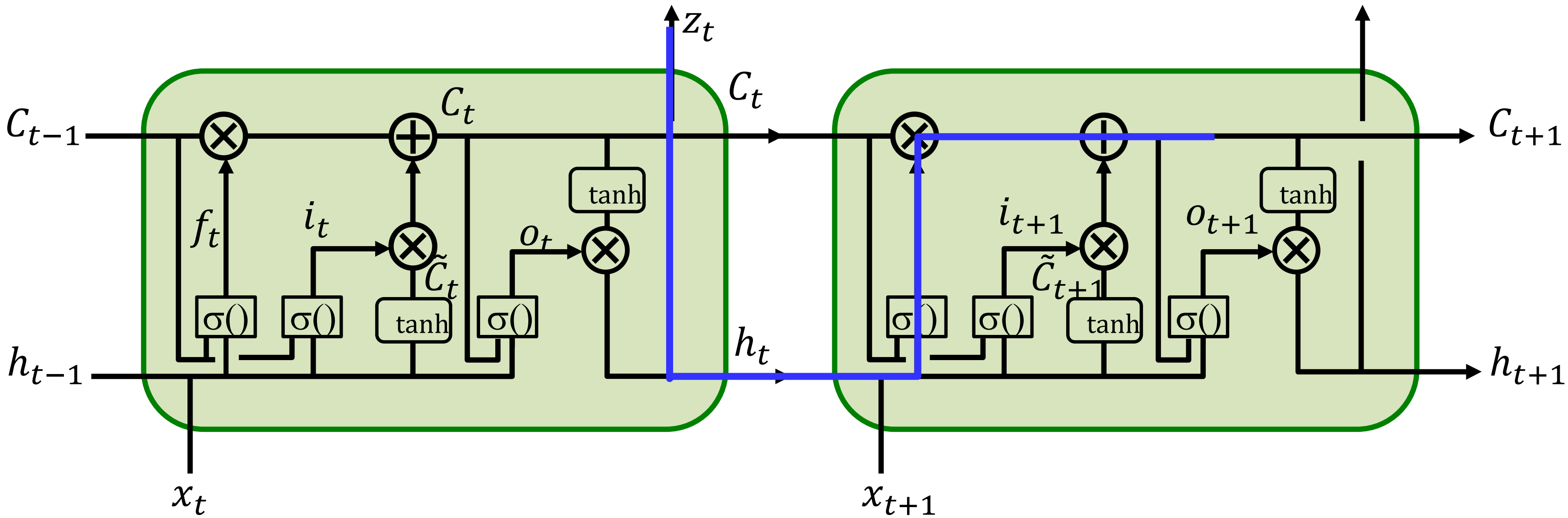
Backpropagation rules: Backward



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

$$\nabla_{h_t} Div = \nabla_{z_t} Div \nabla_{h_t} z_t$$

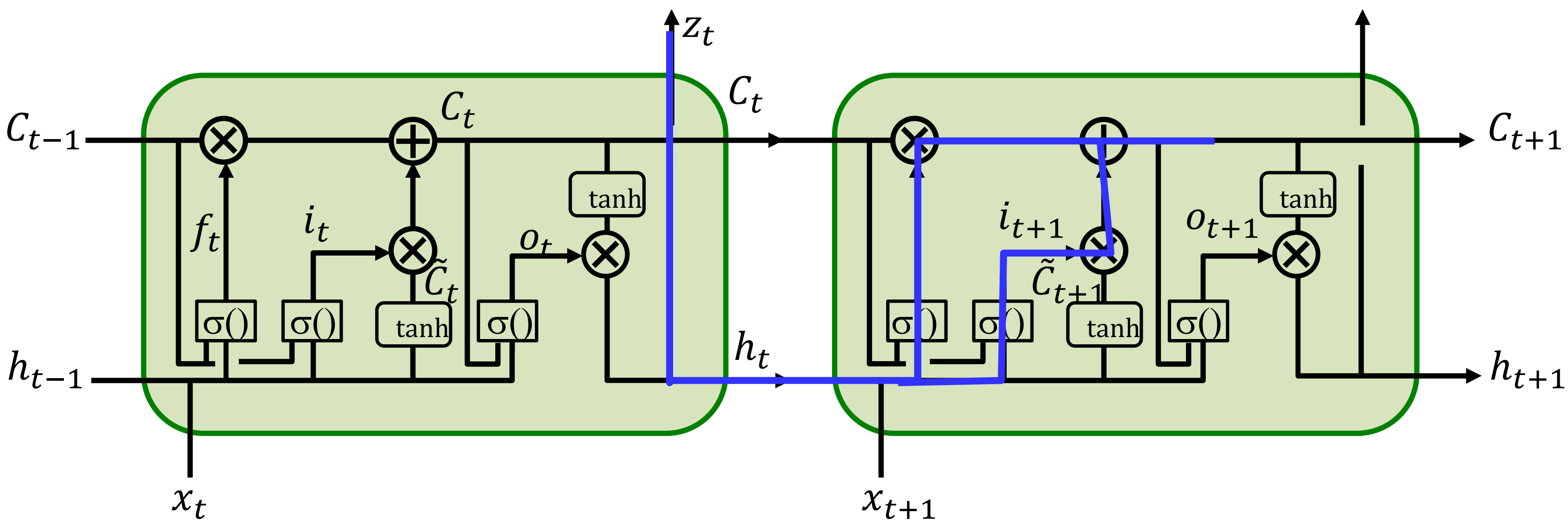
Backpropagation rules: Backward



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

$$\nabla_{h_t} Div = \nabla_{z_t} Div \nabla_{h_t} z_t + \nabla_{C_{t+1}} Div \circ C_t \circ \sigma'(.)W_{hf}$$

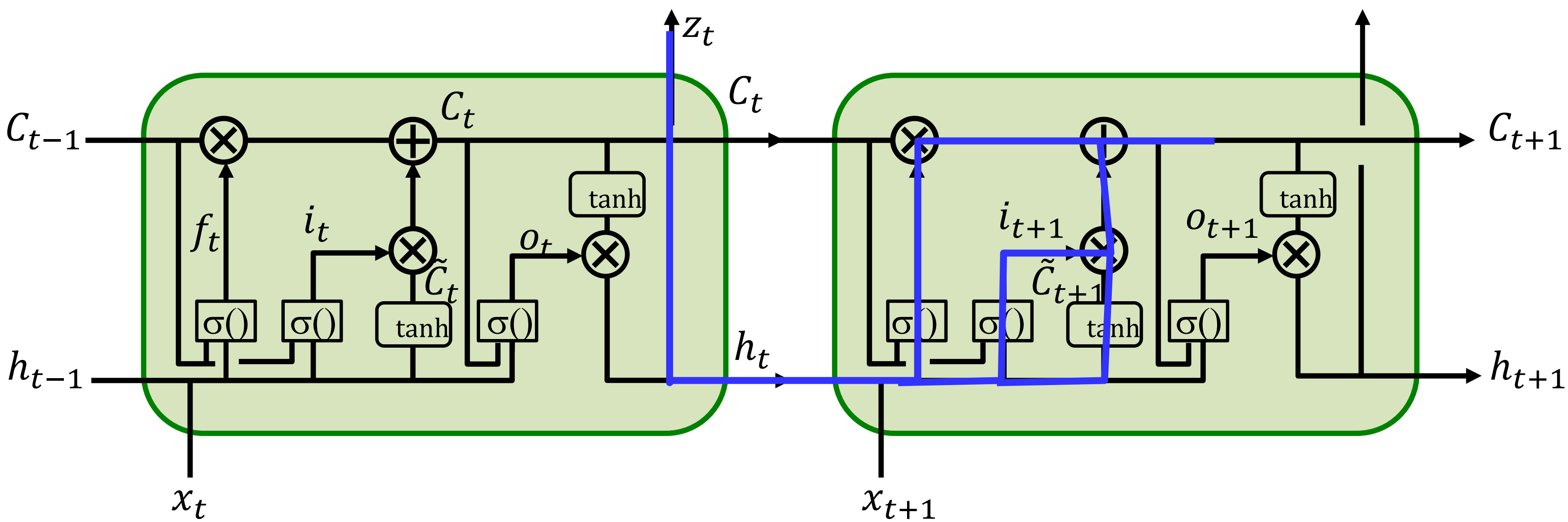
Backpropagation rules: Backward



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

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

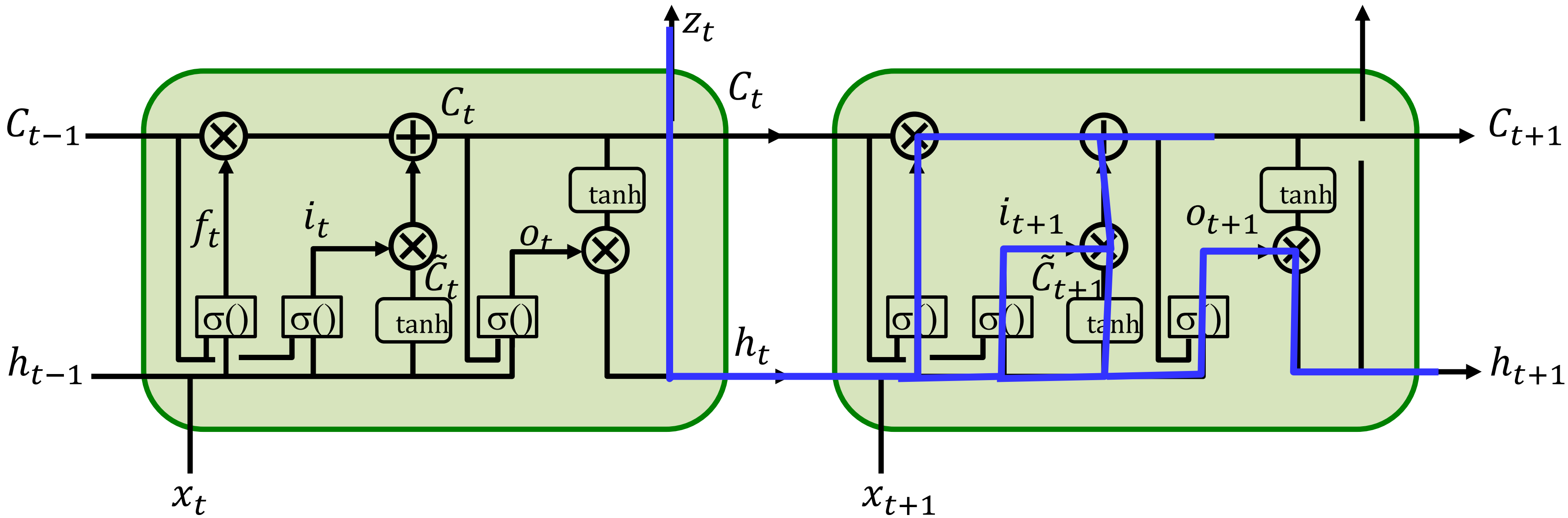
Backpropagation rules: Backward



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

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

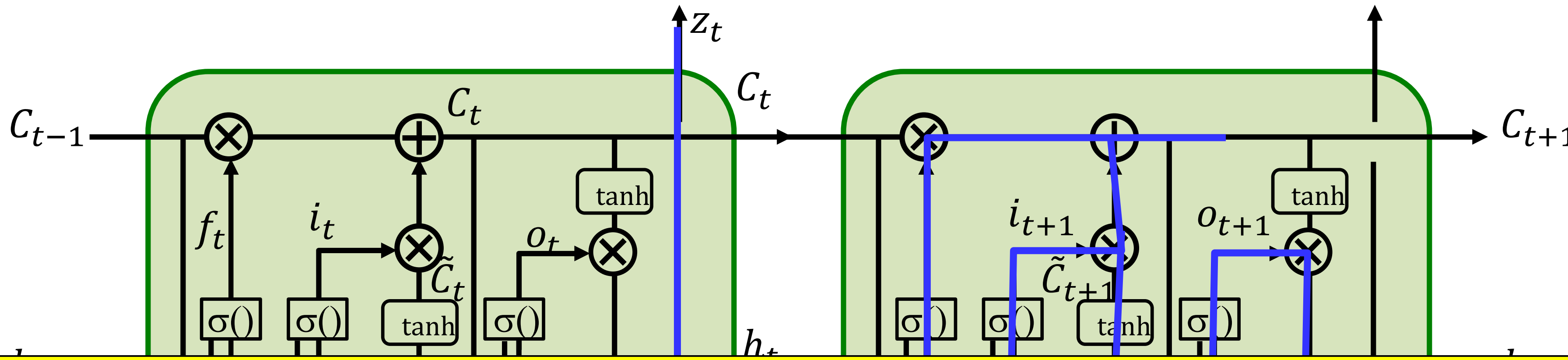
Backpropagation rules: Backward



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

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

Backpropagation rules: Backward



Not explicitly deriving the derivatives w.r.t weights;
Left as an exercise

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

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

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_cc 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 zf, zi, zc, zo, f, i, o, Ci
function [dC,dh,dx,d[W, b]]=LSTM_cell.backward(dCo, dho, C, h, Co, ho, x, [W,b])
    # First invert ho = o∘tanh(C)
    do = dho ∘ tanh(Co)^T
    d tanhCo = dho ∘ o
    dCo += dtanhCo ∘ (1-tanh²(Co))^T #(1-tanh²) is the derivative of tanh

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

    # Next invert zo = WocCo + Wohh + Woxx + bo
    dCo += dzoWoc  #  Note - this is a regular matrix multiply
    dh = dzo Woh
    dx = dzo Wox

    dWoc = Codzo  # Note - this multiplies a column vector by a row vector
    dWoh = h dzo
    dWox = x dzo
    dbo = dzo

    # Next invert Co = f∘C + i∘Ci
    dC = dCo ∘ f
    dCi = dCo ∘ i
    di = dCo ∘ Ci
    df = dCo ∘ C
```

LSTM cell backward (continued)

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

Next invert $z_c = W_{cc}C + W_{ch}h + W_{cx}x + b_c$
dC += $dz_c W_{cc}$
dh += $dz_c\ W_{ch}$
dx += $dz_c\ W_{cx}$

$dW_{cc} = C\ dz_c$
$dW_{ch} = h\ dz_c$
$dW_{cx} = x\ dz_c$
$db_c = dz_c$

Next invert i = sigmoid(z_i)
$dz_i = di \circ \mathrm{sigmoid}(z_i)^T \circ (1-\mathrm{sigmoid}(z_i))^T$

Next invert $z_i = W_{ic}C + W_{ih}h + W_{ix}x + b_i$
dC += $dz_i\ W_{ic}$
dh += $dz_i\ W_{ih}$
dx += $dz_i\ W_{ix}$

$dW_{ic} = C\ dz_i$
$dW_{ih} = h\ dz_i$
$dW_{ix} = x\ dz_i$
$db_i = dz_i$

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
```

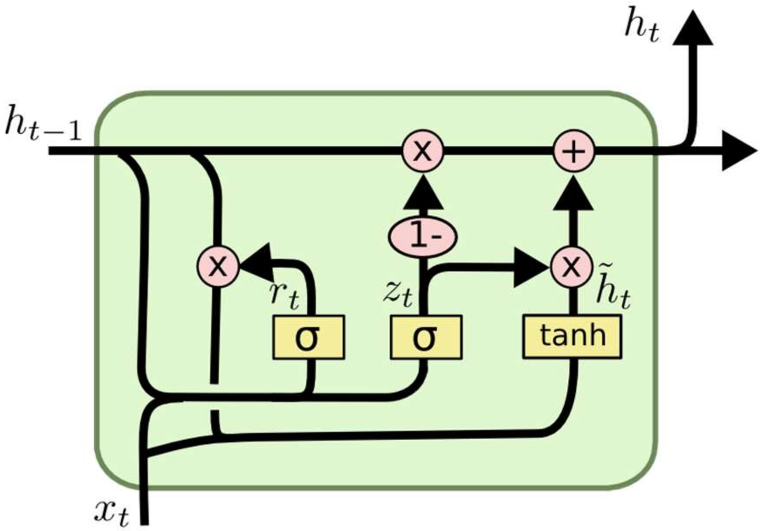
LSTM network forward (for reference)

```
# 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 lth hidden layer
# Wo and bo 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}])
    zo(t) = Woh(t,L) + bo
    Y(t) = softmax( zo(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 lth hidden layer
# Wo and bo are output layer weights and biases
# Y is the output of the network
# Assuming dWo and dbo 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
    dzo  = dY(t) ∘ Softmax_Jacobian(zo(t))
    dWo += h(t,L) dzo(t)
    dh(t,L) = dzo(t)Wo
    dbo += dzo(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]
```

Gated Recurrent Units: Lets simplify the LSTM



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

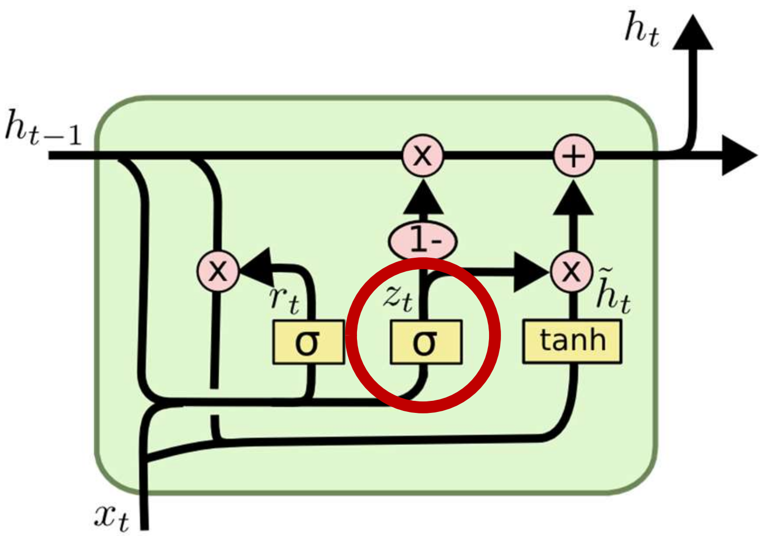
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

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

Gated Recurrent Units: Lets simplify the LSTM



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

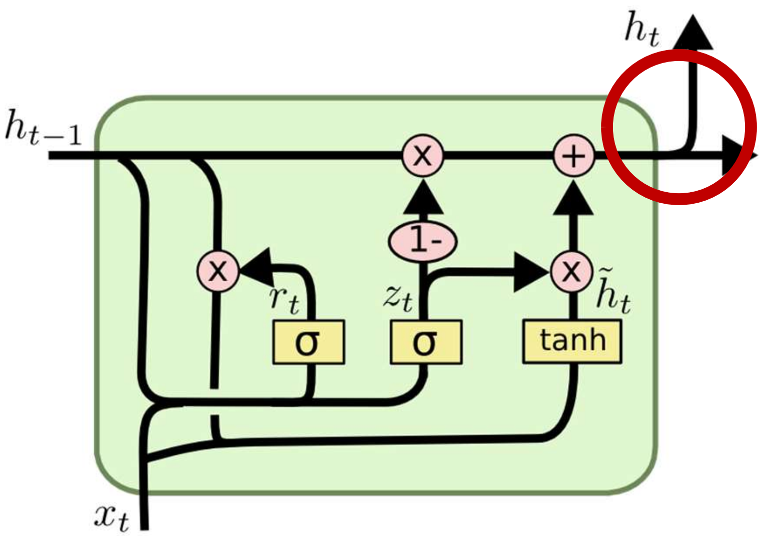
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

- 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



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

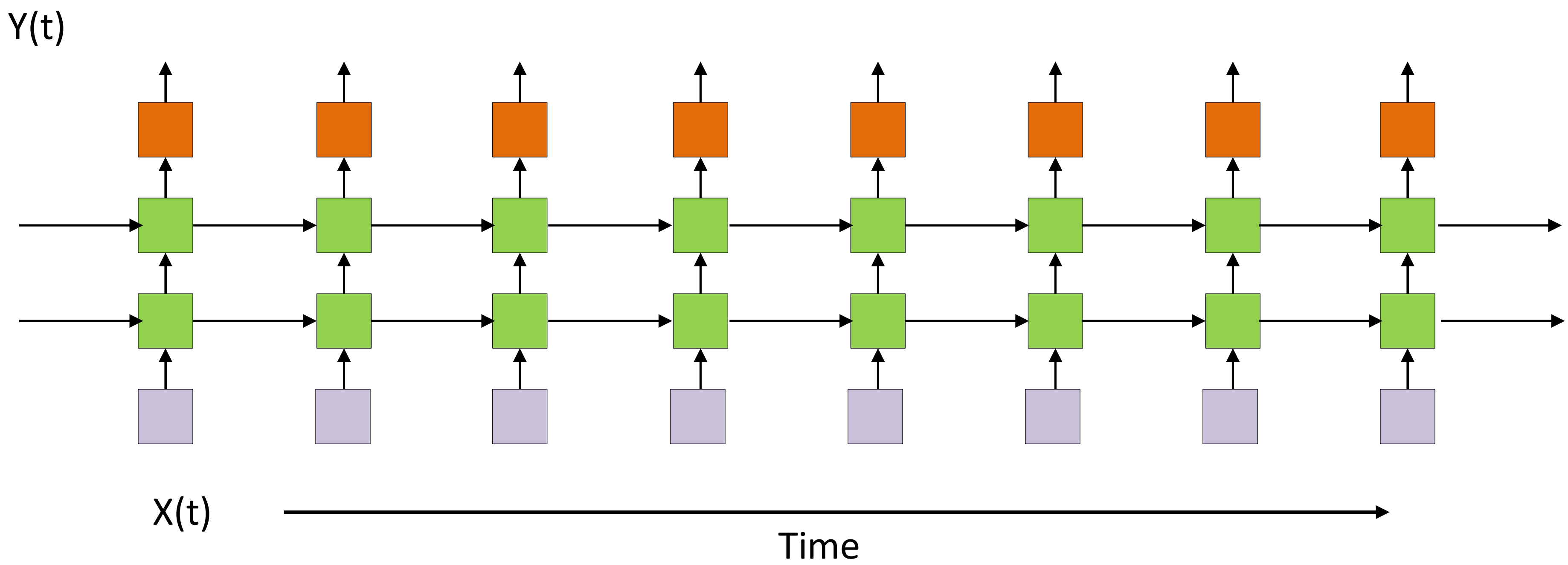
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

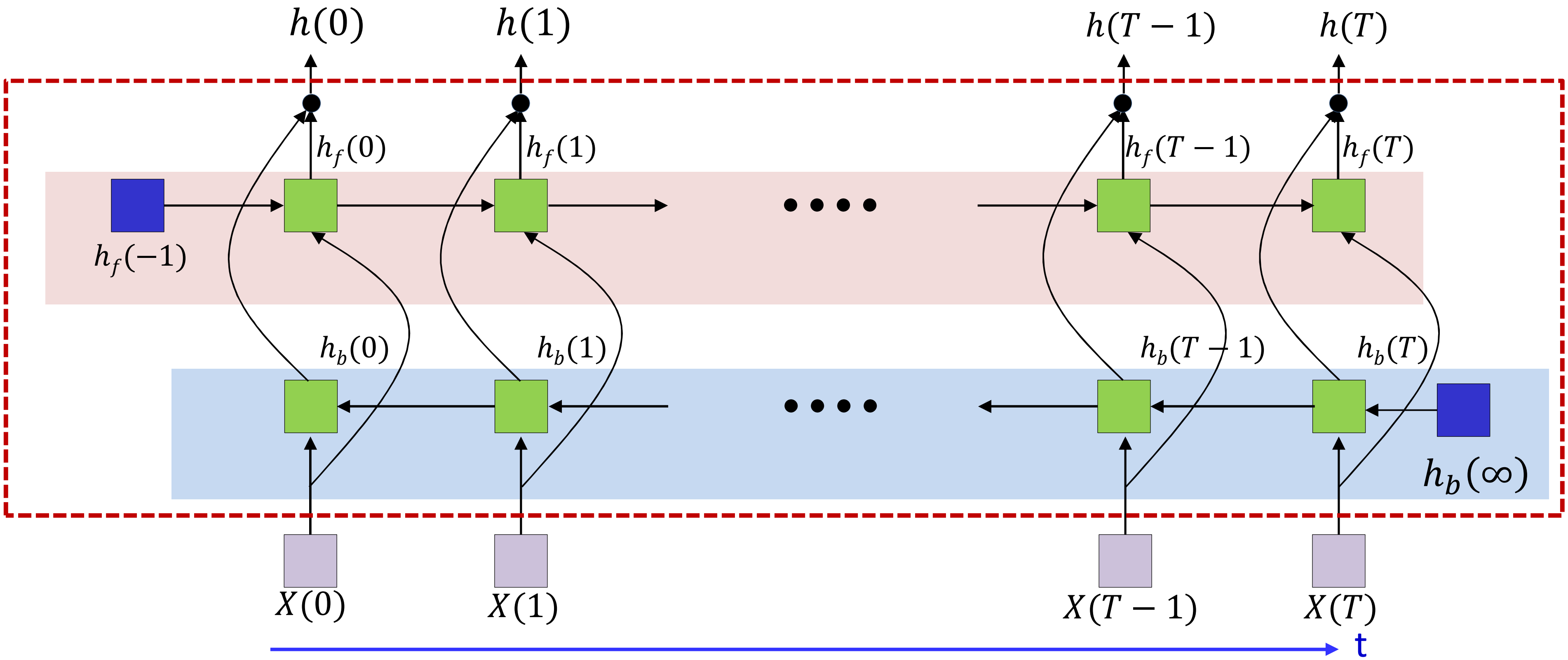
- Don’t bother to separately maintain compressed and regular memories
  - Pointless computation!
  - Redundant representation

LSTM architectures example



- Each green box is now a (layer of) LSTM or GRU 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

Story so far

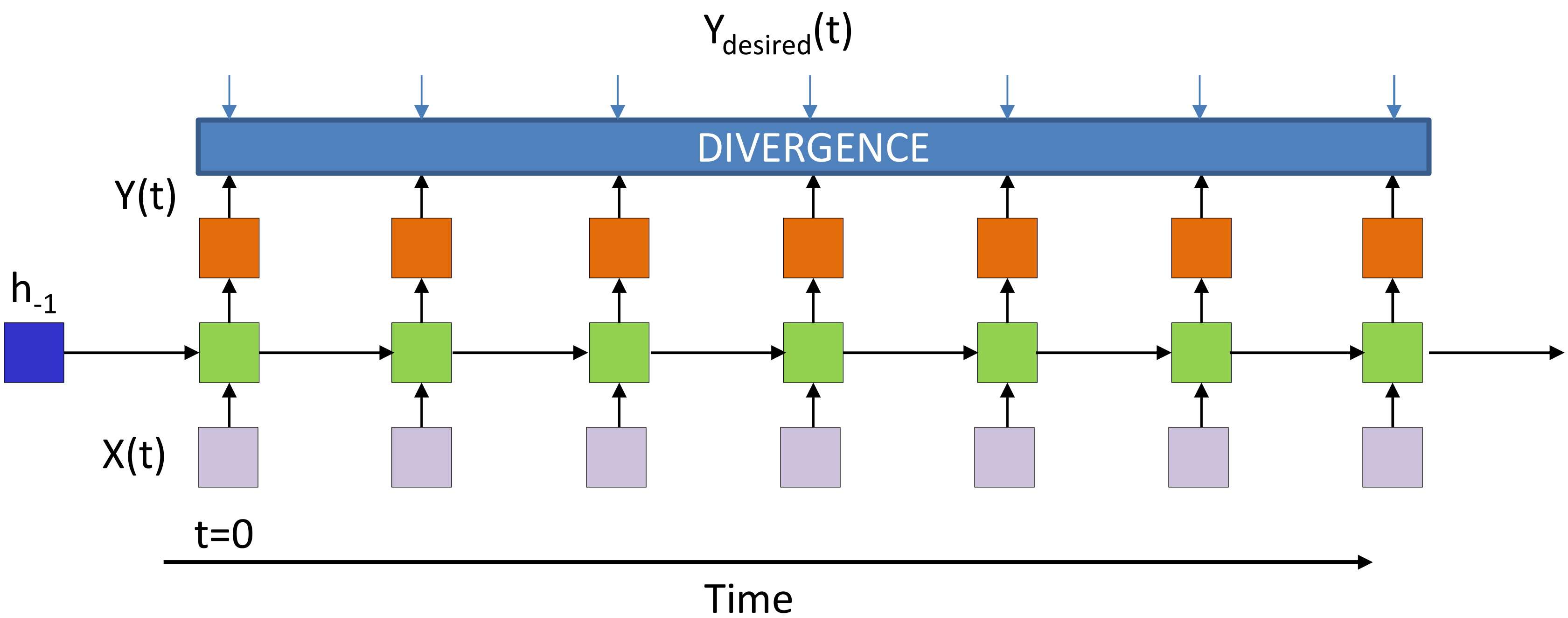
- Recurrent networks are poor at memorization
  - Memory can explode or vanish depending on the weights and activation
- They also suffer from the vanishing gradient problem during training
  - Error at any time cannot affect parameter updates in the too-distant past
  - E.g. seeing a "close bracket" 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..

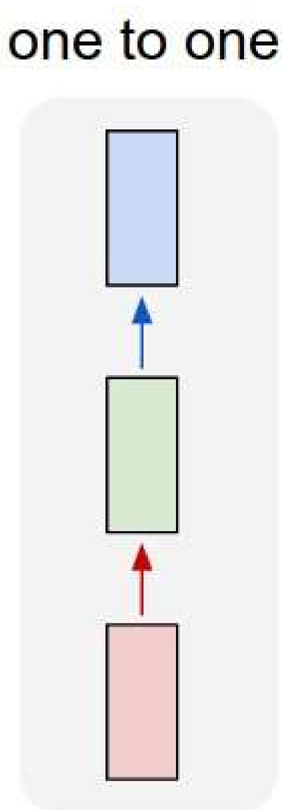
What follows in this series on recurrent nets

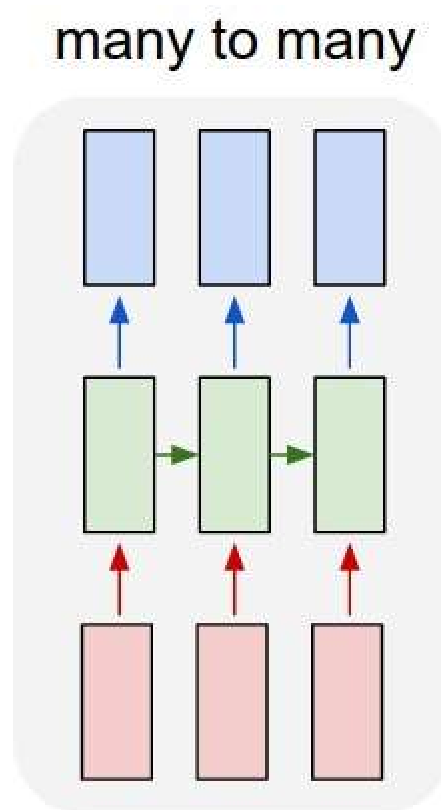
- Architectures: How to train recurrent networks of different architectures

- Synchrony: How to train recurrent networks when
  - The target output is time-synchronous with the input
  - The target output is order-synchronous, but not time synchronous
  - Applies to only some types of nets

- How to make predictions/inference with such networks

Variants of recurrent nets

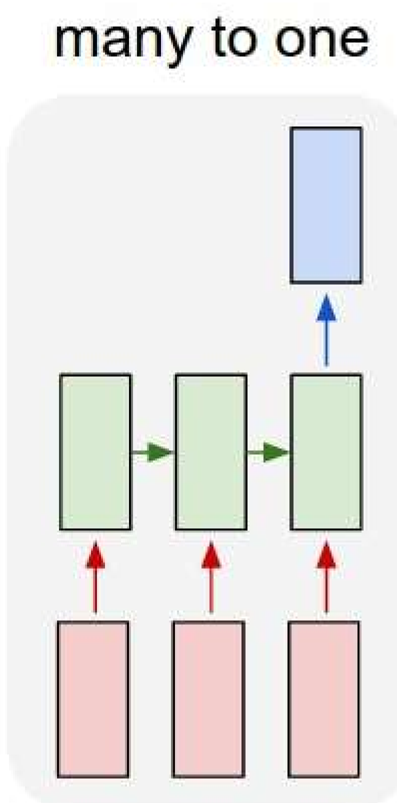


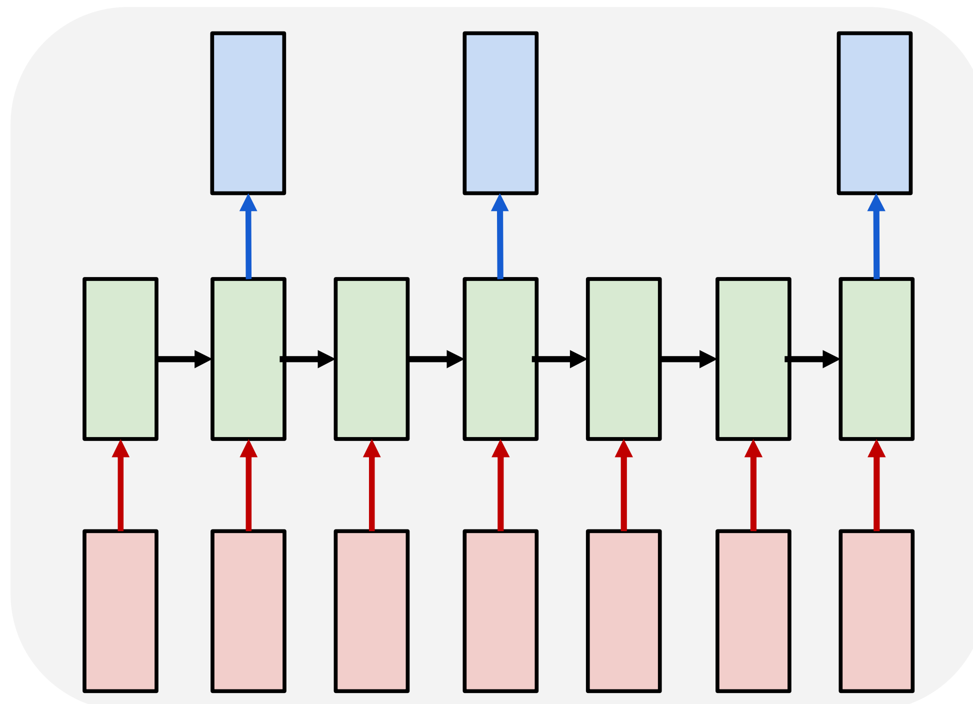


Images from Karpathy

- Conventional MLP
- Time-synchronous outputs
  - E.g. part of speech tagging

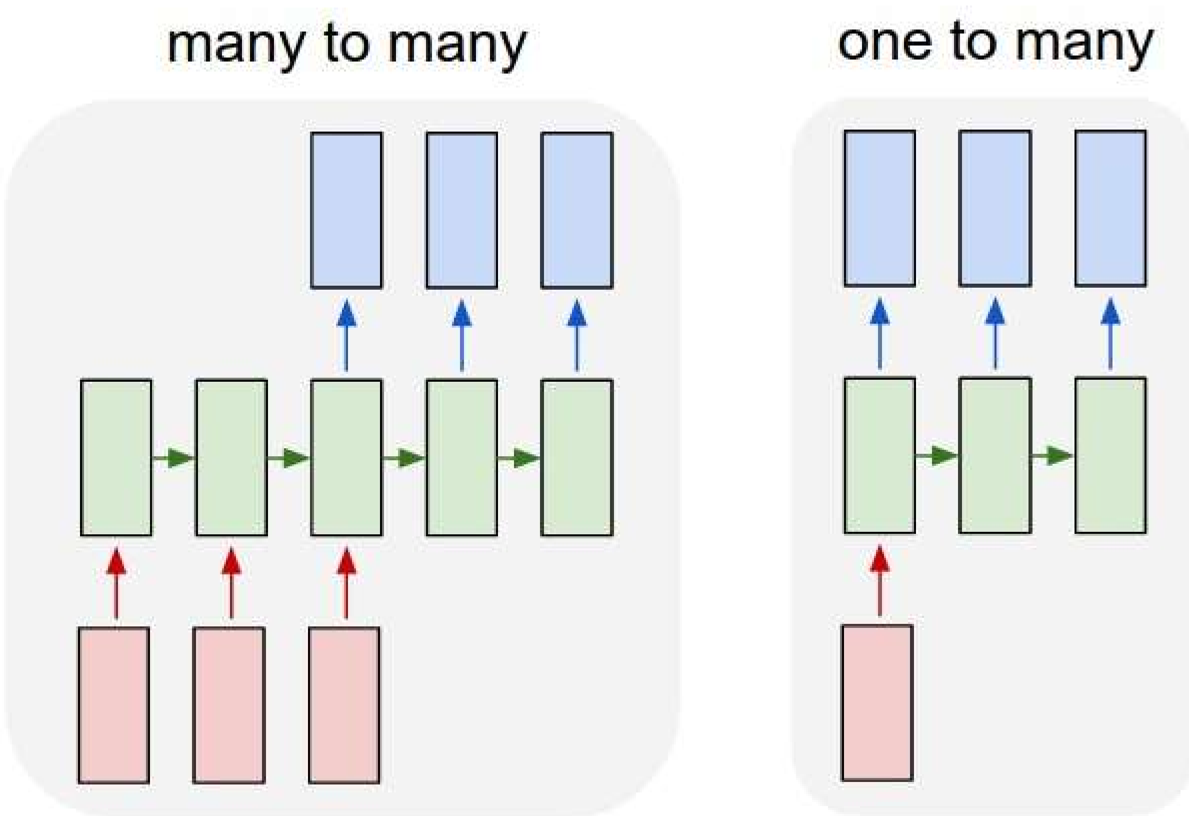
Variants of recurrent nets





- Sequence classification: Classifying a full input sequence
  - E.g isolated word/phrase recognition
- Order synchronous , time asynchronous sequence-to-sequence generation
  - E.g. speech recognition
  - Exact location of output is unknown a priori

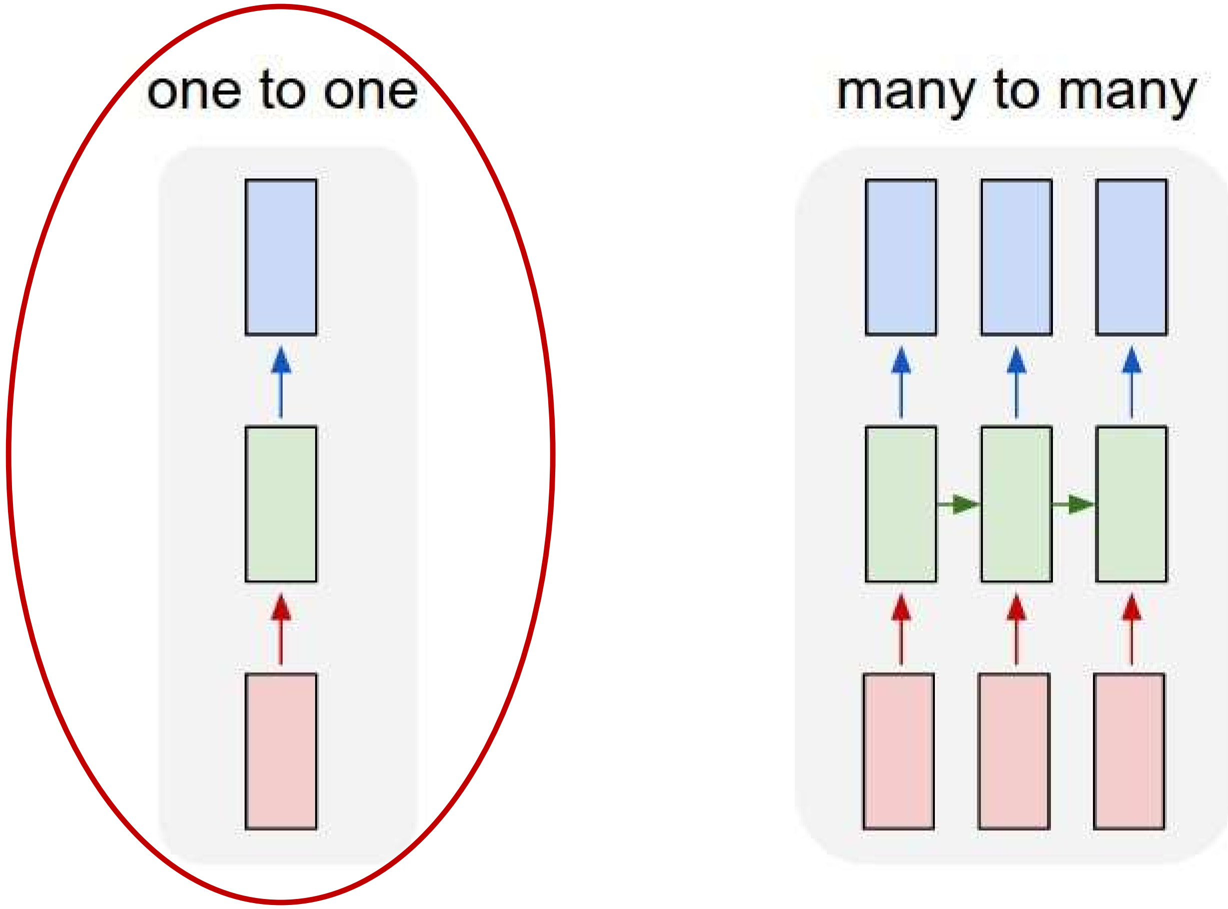
More variants



Images from Karpathy

- A posteriori sequence to sequence: Generate output sequence after processing input
  - E.g. language translation
- Single-input a posteriori sequence generation
  - E.g. captioning an image

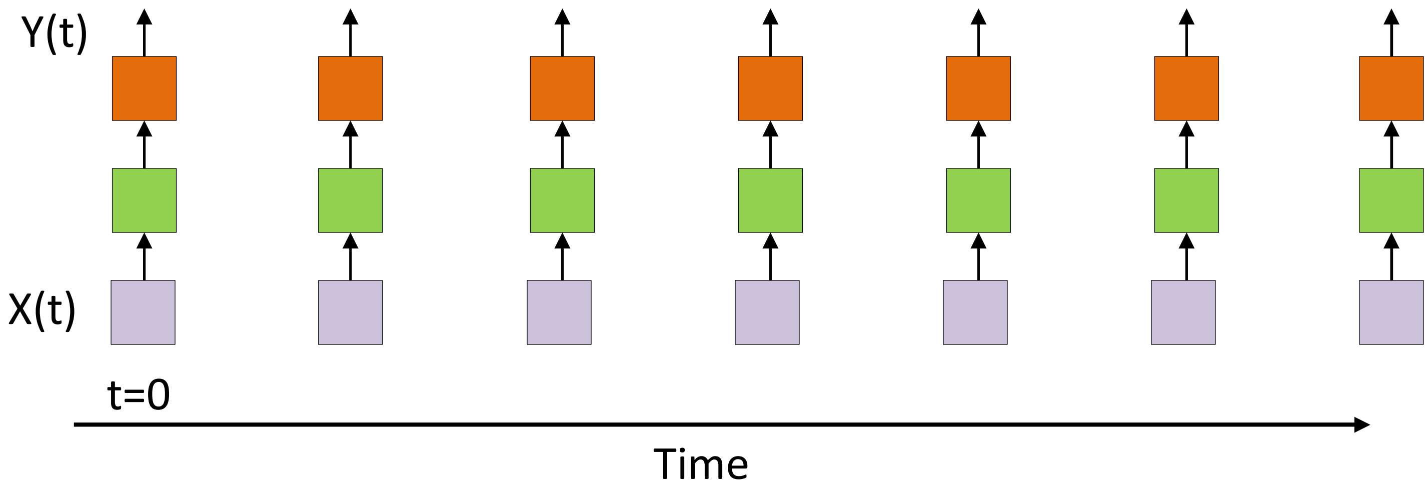
Variants of recurrent nets



Images from Karpathy

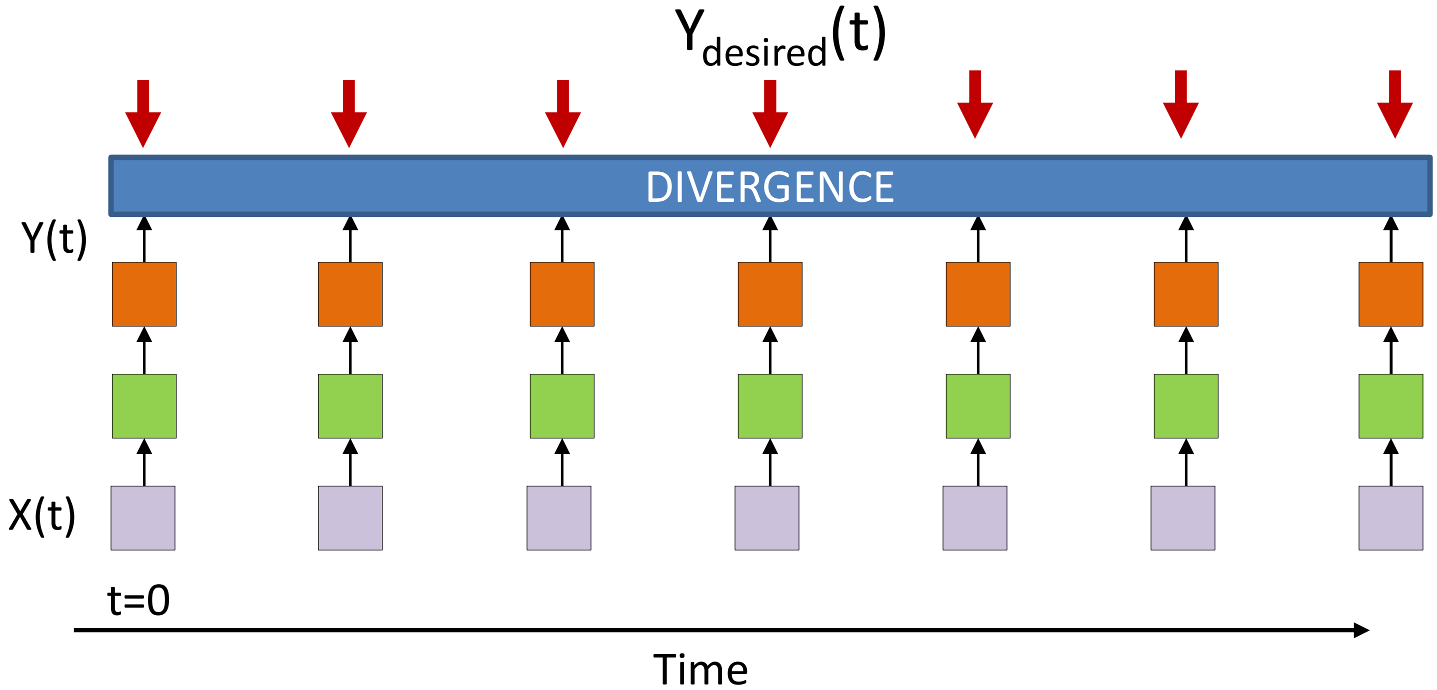
- Conventional MLP
- Time-synchronous outputs
  - E.g. part of speech tagging

Regular MLP for processing sequences



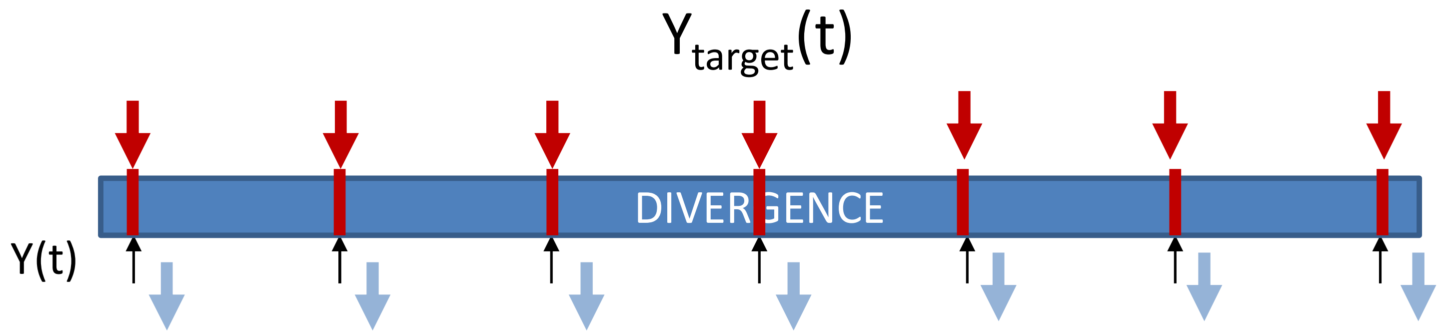
- No recurrence in model
  - Exactly as many outputs as inputs
  - Every input produces a unique output
  - The output at time t is unrelated to the output at $t' \neq t$

Learning in a Regular MLP



- No recurrence
  - Exactly as many outputs as inputs
    - **One to one correspondence between desired output and actual output**
  - The output at time t is unrelated to the output at $t' \neq t$.

Regular MLP



- Gradient backpropagated at each time

$$\nabla_{Y(t)} Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big)$$

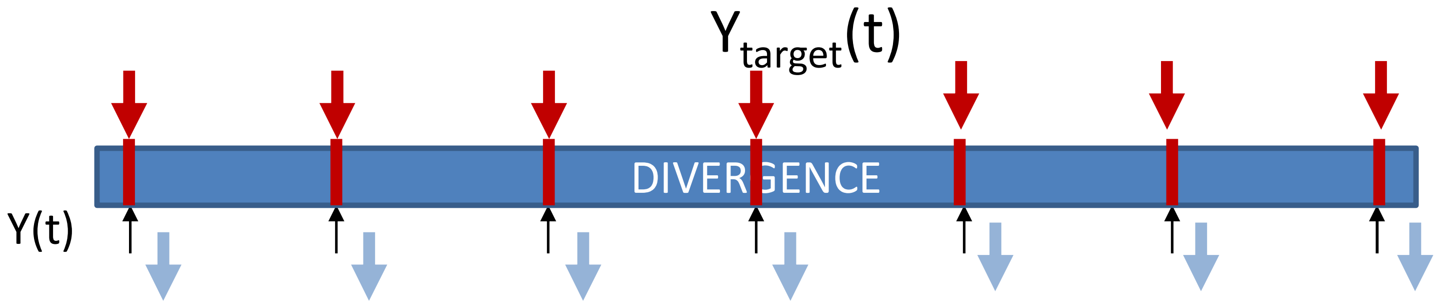
- Common assumption:

$$Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big) = \sum_t w_t Div\big(Y_{target}(t), Y(t)\big)$$

$$\nabla_{Y(t)} Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big) = w_t \nabla_{Y(t)} Div\big(Y_{target}(t), Y(t)\big)$$

  - w_t is typically set to 1.0
  - This is further backpropagated to update weights etc

Regular MLP



- Gradient backpropagated at each time

$$\nabla_{Y(t)} Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big)$$

- Common assumption:

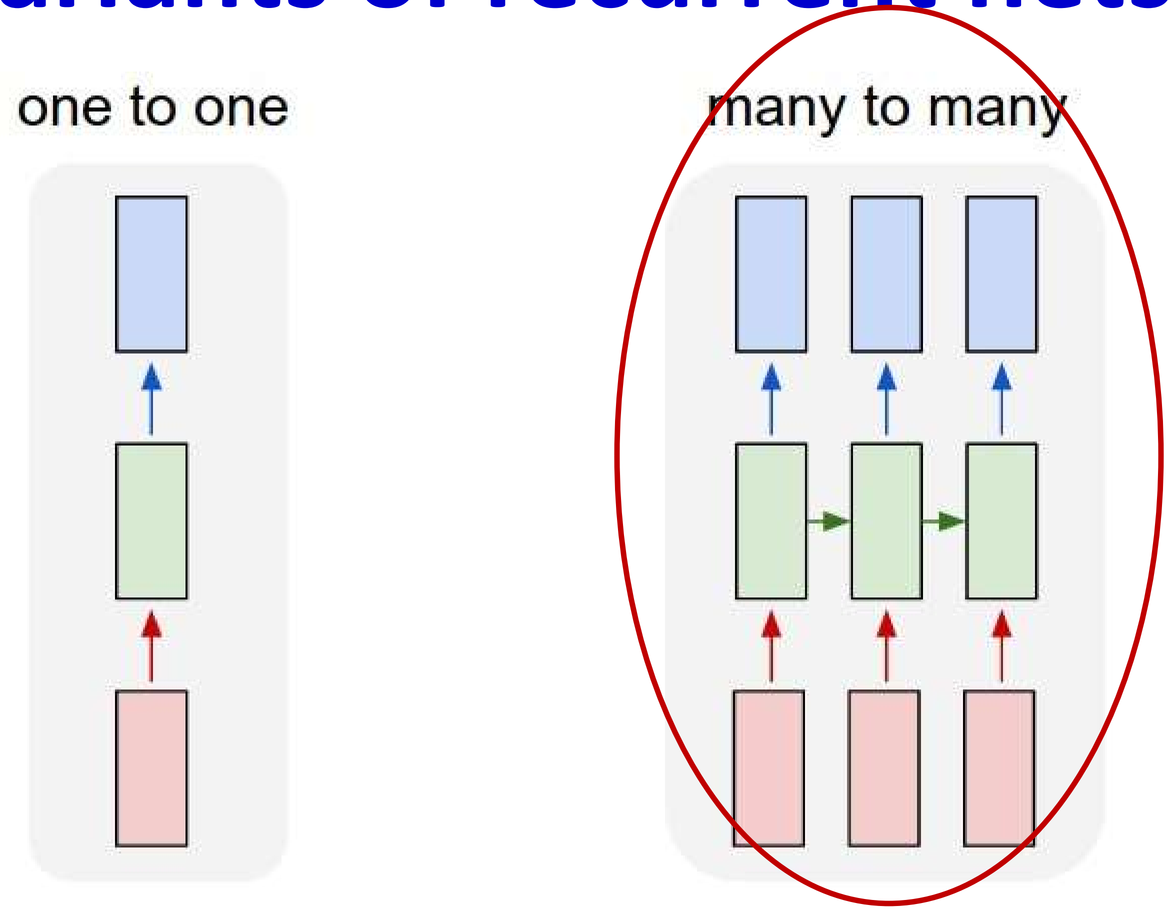
$$Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big) = \sum_t Div\big(Y_{target}(t), Y(t)\big)$$

$$\nabla_{Y(t)} Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big) = \nabla_{Y(t)} Div\big(Y_{target}(t), Y(t)\big)$$

  – This is further backpropagated to update weights etc

Typical Divergence for classification: $Div\big(Y_{target}(t), Y(t)\big) = KL(Y_{target}(t), Y(t))$

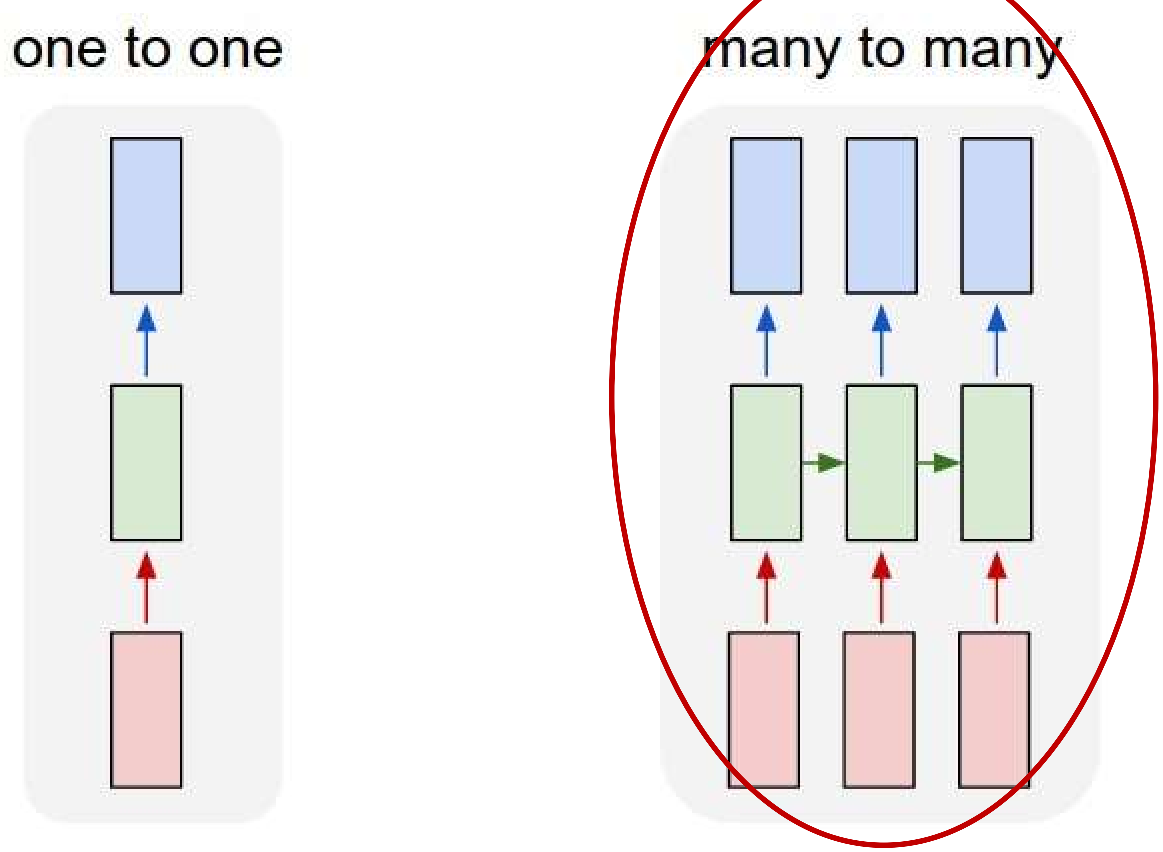
Variants of recurrent nets



Images from Karpathy

- Conventional MLP
- Time-synchronous outputs
  - E.g. part of speech tagging

Variants of recurrent nets

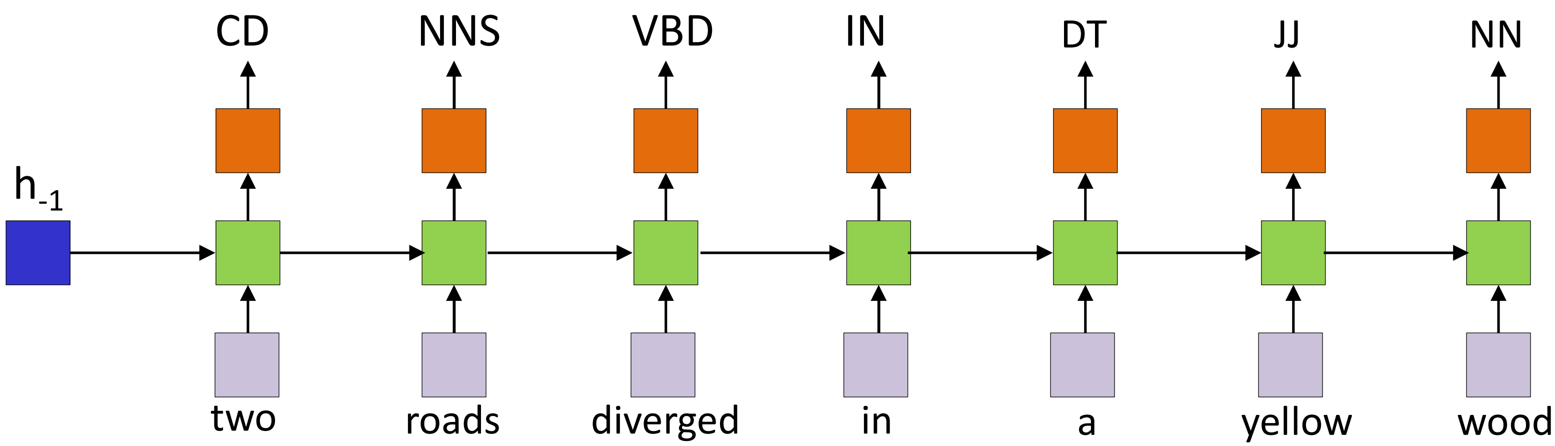


Images from Karpathy

With a brief detour into modelling language

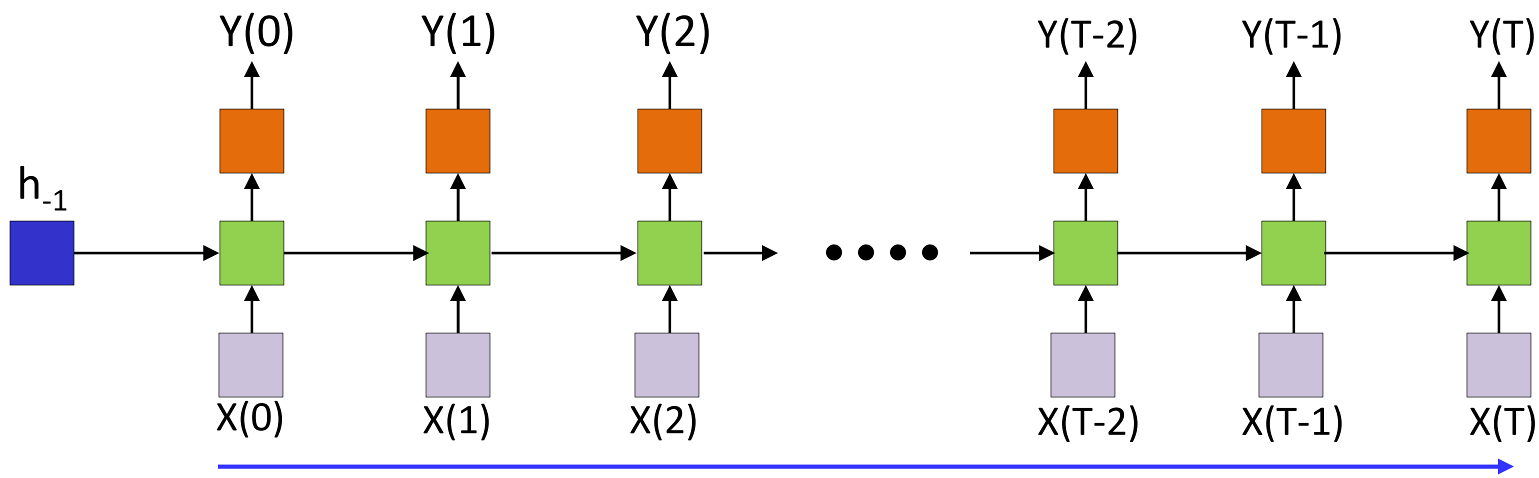
- Conventional MLP
- Time-synchronous outputs
  - E.g. part of speech tagging

Time synchronous network



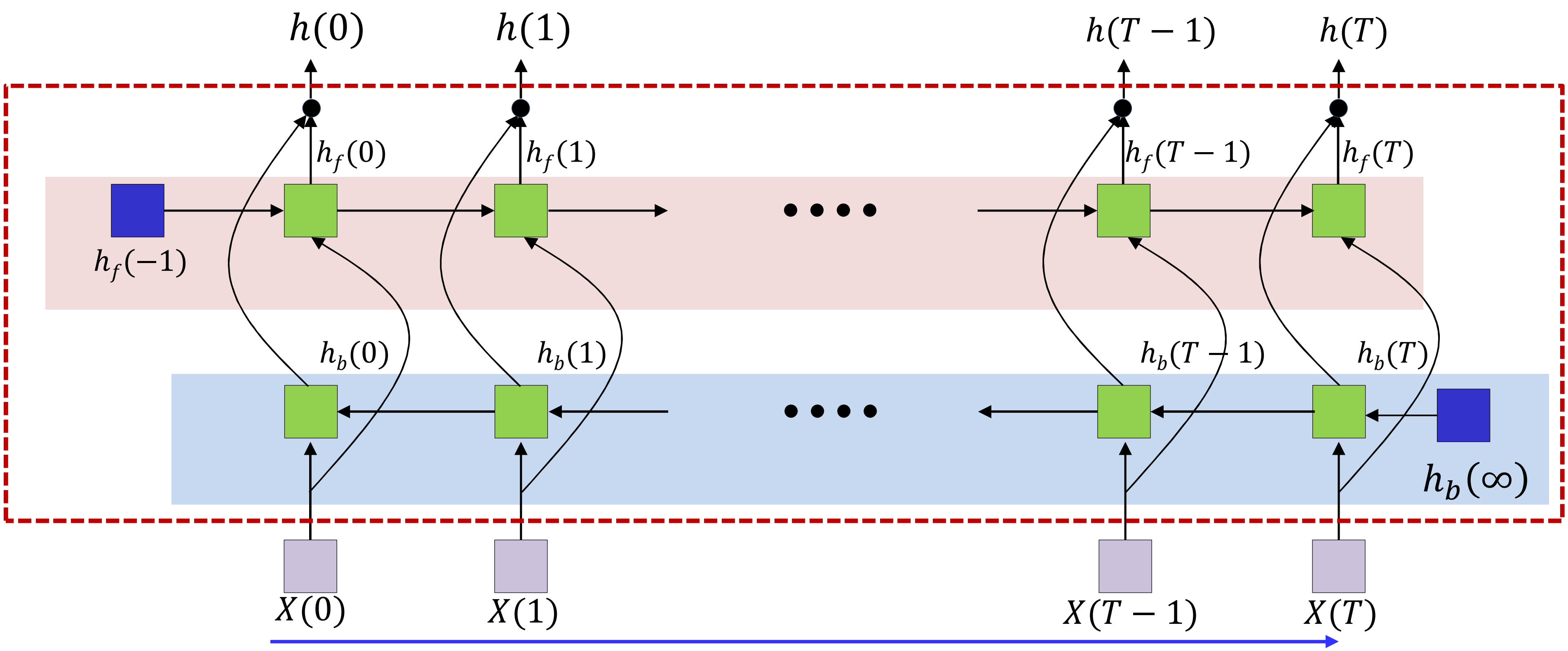
- Network produces one output for each input
  - With one-to-one correspondence
  - E.g. Assigning grammar tags to words
    - May require a bidirectional network to consider both past and future words in the sentence

Time-synchronous networks: Inference



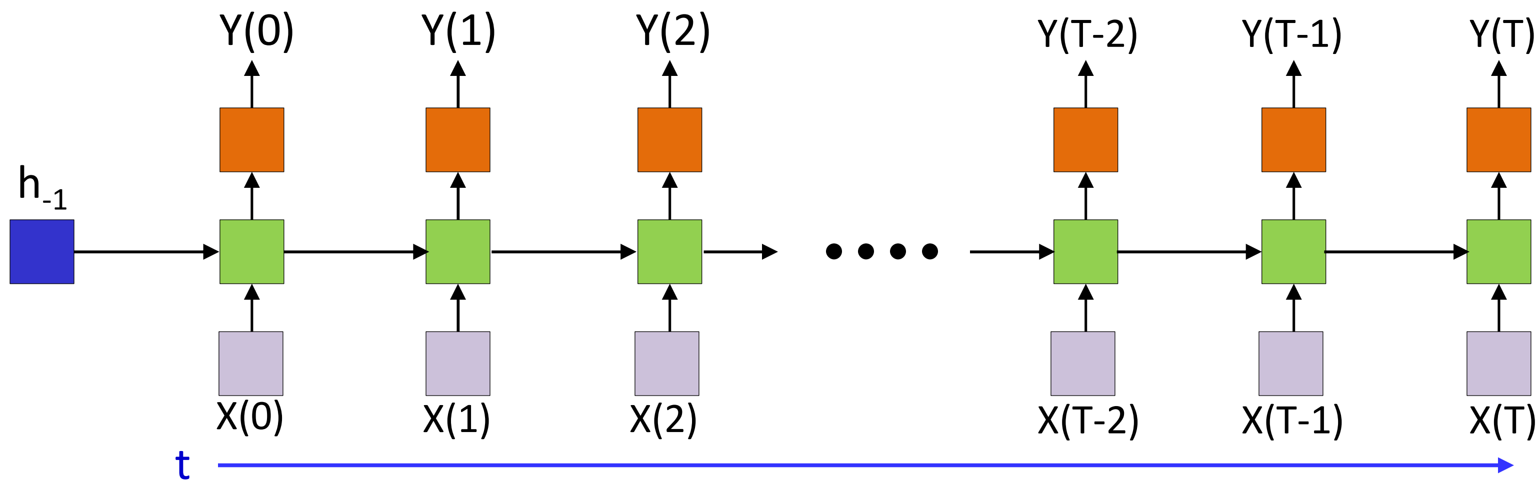
- One sided network: Process input left to right and produce output after each input

Time-synchronous networks: Inference



- **For bidirectional networks:**
  - Process input left to right using forward net
  - Process it right to left using backward net
  - The combined outputs are time-synchronous, one per input time, and are passed up to the next layer
- Rest of the lecture(s) will not specifically consider bidirectional nets, but the discussion generalizes

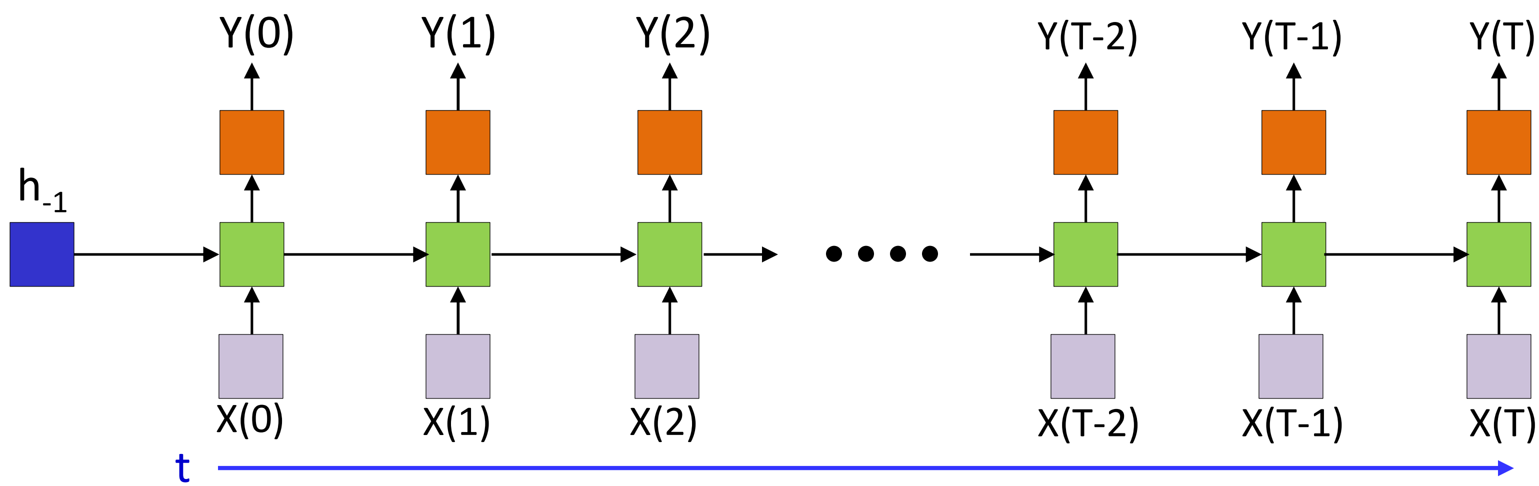
How do we *train* the network



- Back propagation through time (BPTT)

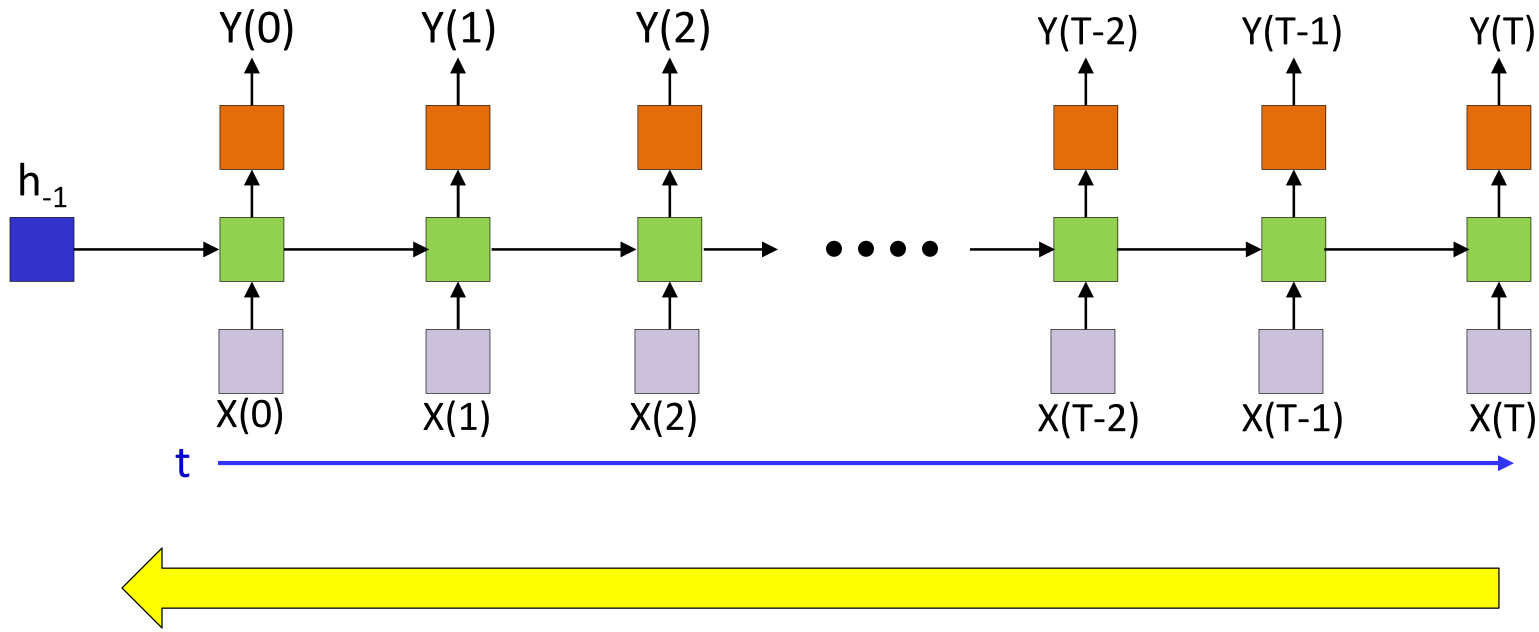
- Given a collection of *sequence* training instances comprising input sequences and output sequences of equal length, with one-to-one correspondence
  - $(\mathbf{X}_i, \mathbf{D}_i)$, where
  - $\mathbf{X}_i = X_{i,0}, \dots, X_{i,T}$
  - $\mathbf{D}_i = D_{i,0}, \dots, D_{i,T}$

Training: Forward pass



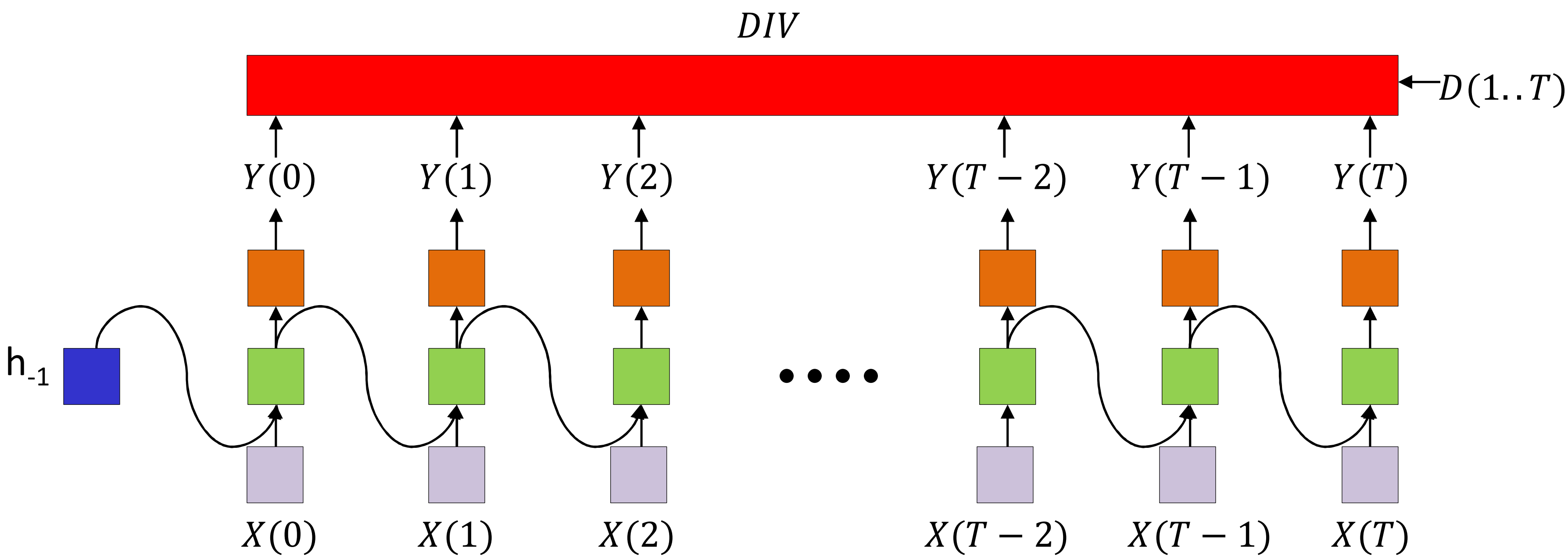
- For each training input:
- Forward pass: pass the entire data sequence through the network, generate outputs

Training: Computing gradients



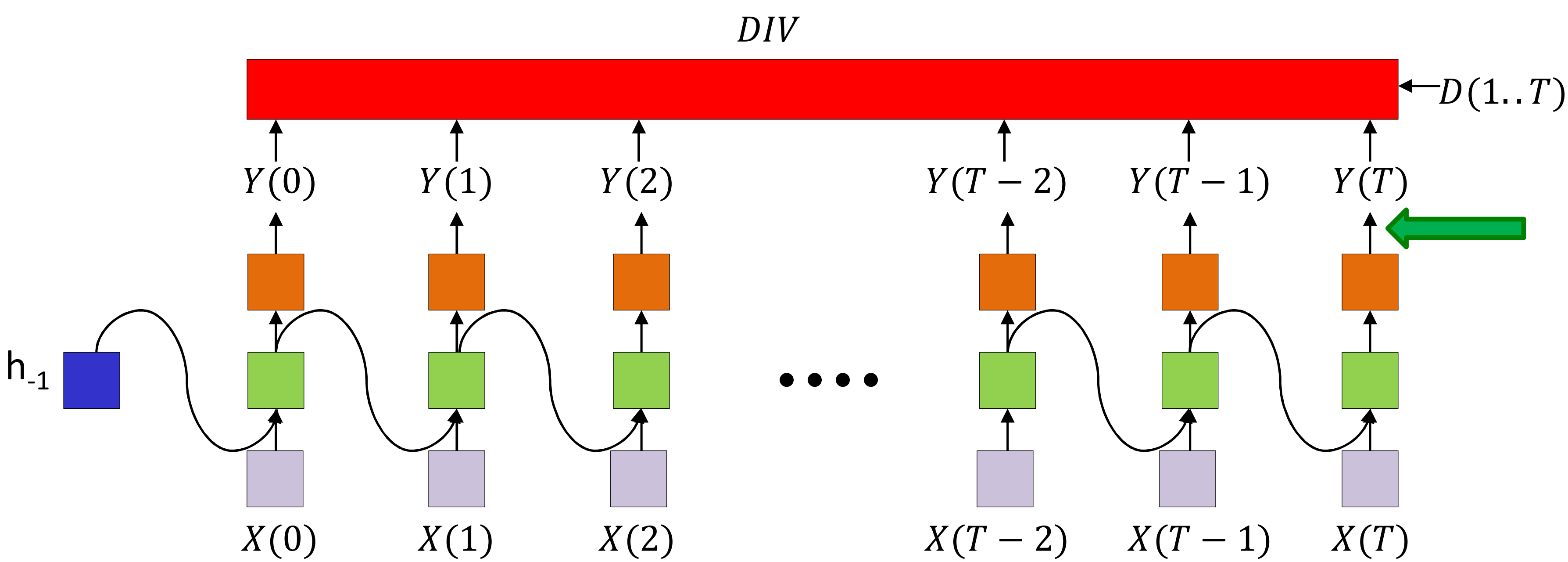
- For each training input:
- Backward pass: Compute divergence gradients via backpropagation
  - *Back Propagation Through Time*

Back Propagation Through Time



- The divergence computed is between the *sequence of outputs* by the network and the *desired sequence of outputs*
- This is *not* just the sum of the divergences at individual times
  - Unless we explicitly define it that way

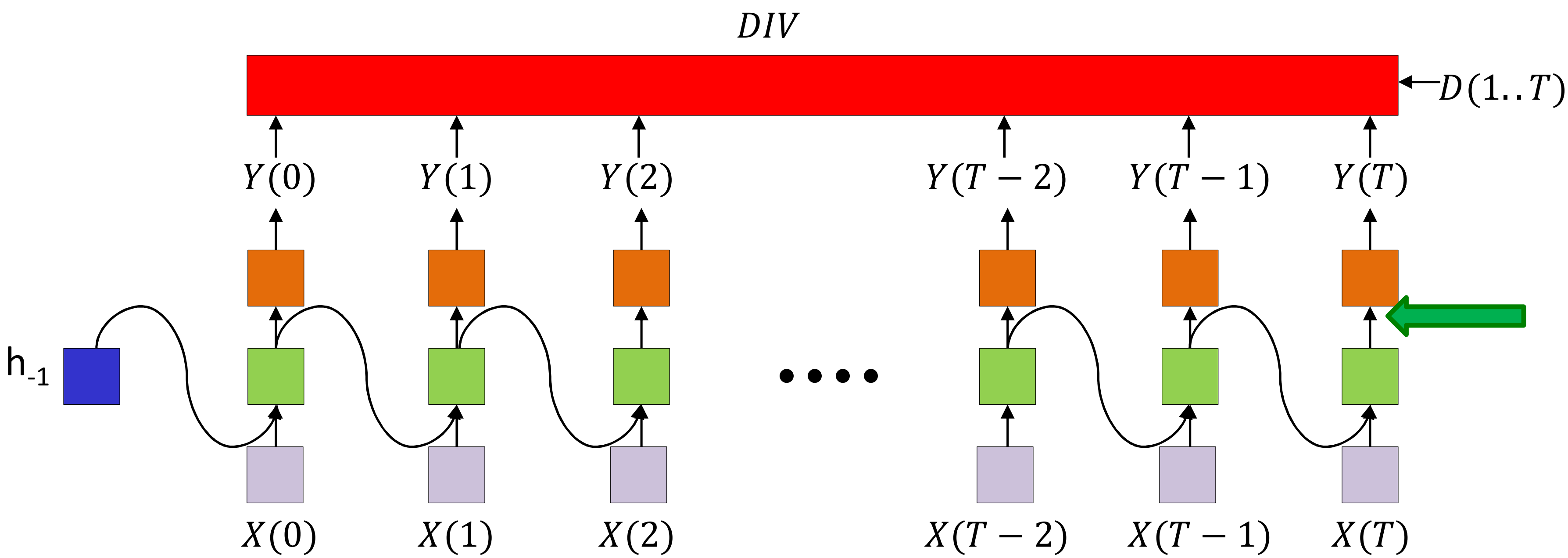
Back Propagation Through Time



First step of backprop: Compute $\nabla_{Y(t)}DIV$ for all t

The rest of backprop continues from there

Back Propagation Through Time

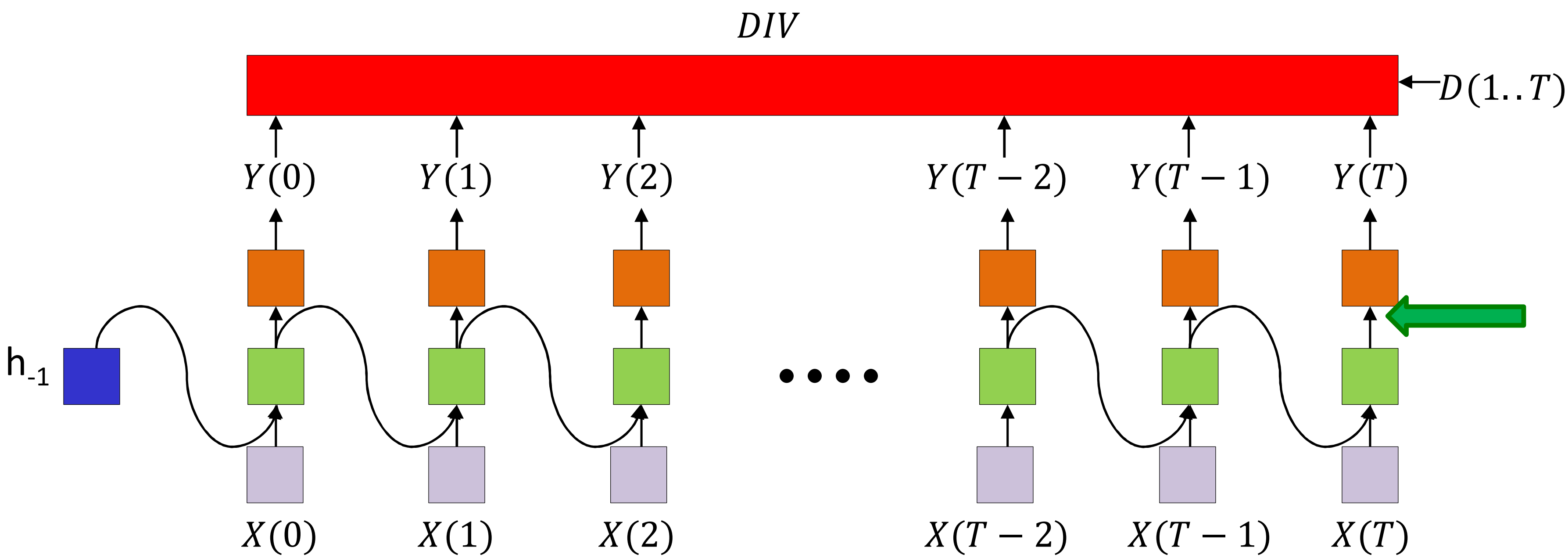


First step of backprop: Compute $\nabla_{Y(t)}DIV$ for all t

$$\nabla_{Z^{(1)}(t)}DIV = \nabla_{Y(t)}DIV \, \nabla_{Z(t)}Y(t)$$

And so on!

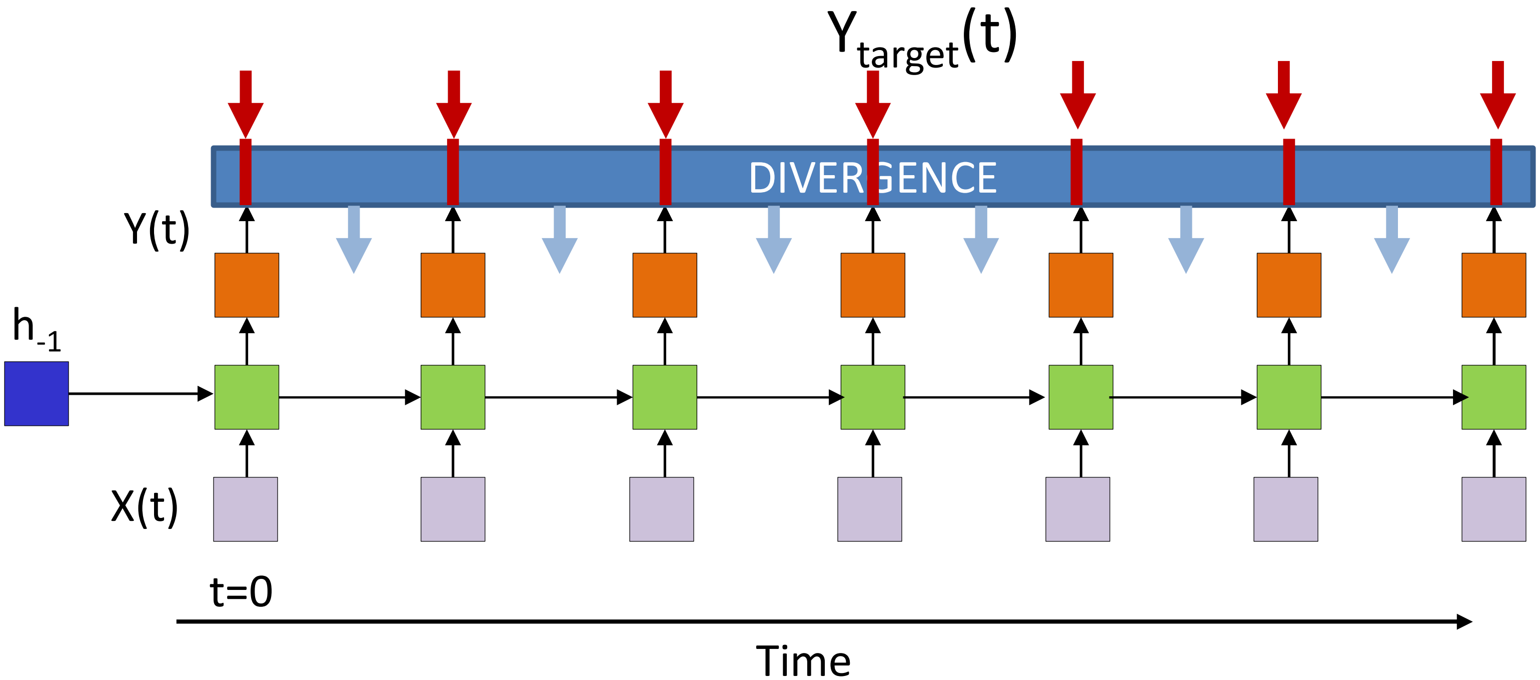
Back Propagation Through Time



First step of backprop: Compute $\nabla_{Y(t)} DIV$ for all t

- The key component is the computation of this derivative!!
- This depends on the definition of "DIV"

Time-synchronous recurrence

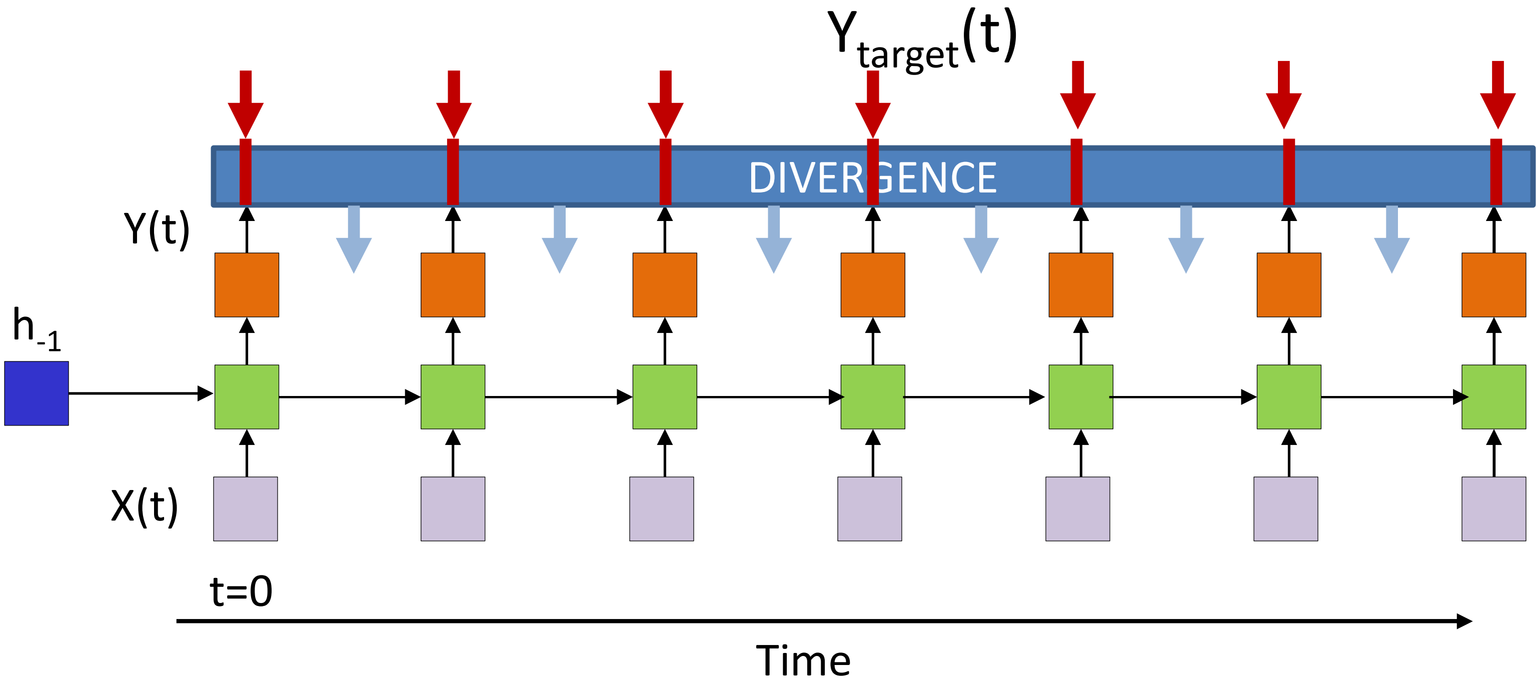


- Usual assumption: ***Sequence divergence is the sum of the divergence at individual instants***

$$Div(Y_{target}(1 \dots T), Y(1 \dots T)) = \sum_t Div(Y_{target}(t), Y(t))$$

$$\nabla_{Y(t)} Div(Y_{target}(1 \dots T), Y(1 \dots T)) = \nabla_{Y(t)} Div(Y_{target}(t), Y(t))$$

Time-synchronous recurrence



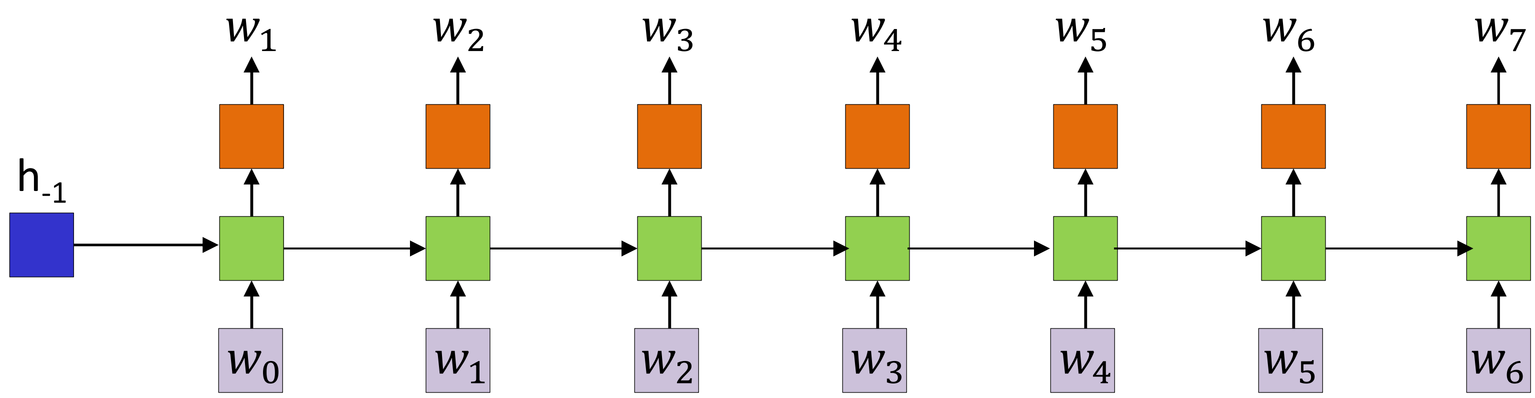
- Usual assumption: ***Sequence divergence is the sum of the divergence at individual instants***

$$Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right) = \sum_t Div\left(Y_{target}(t), Y(t)\right)$$

$$\nabla_{Y(t)} Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right) = \nabla_{Y(t)} Div\left(Y_{target}(t), Y(t)\right)$$

Typical Divergence for classification: $Div\left(Y_{target}(t), Y(t)\right) = KL(Y_{target}(t), Y(t))$

Simple recurrence example: Text Modelling



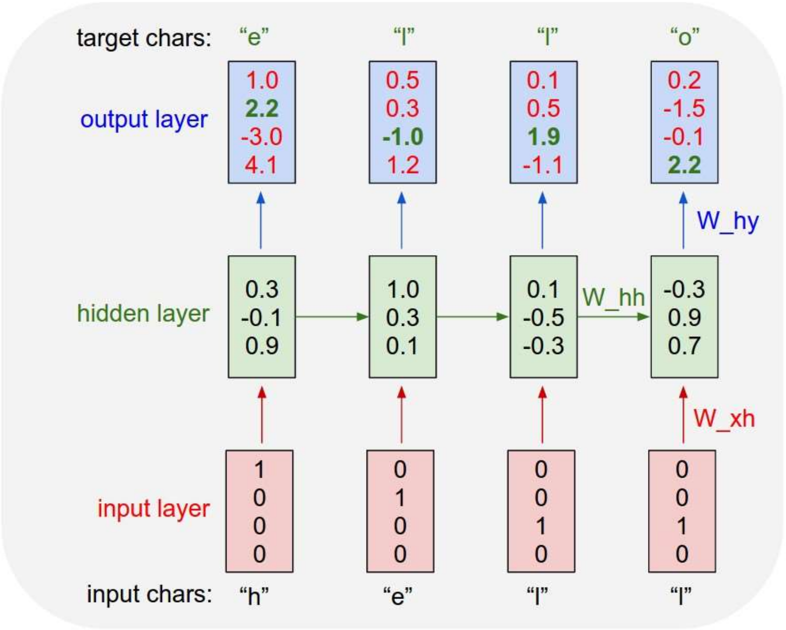
- Learn a model that can predict the next character given a sequence of characters
    - L I N C O L ?
  - Or, at a higher level, words
    - TO BE OR NOT TO ???
- After observing inputs $w_0 \ldots w_k$ it predicts w_{k+1}

Simple recurrence example: Text Modelling

Figure from Andrej Karpathy.

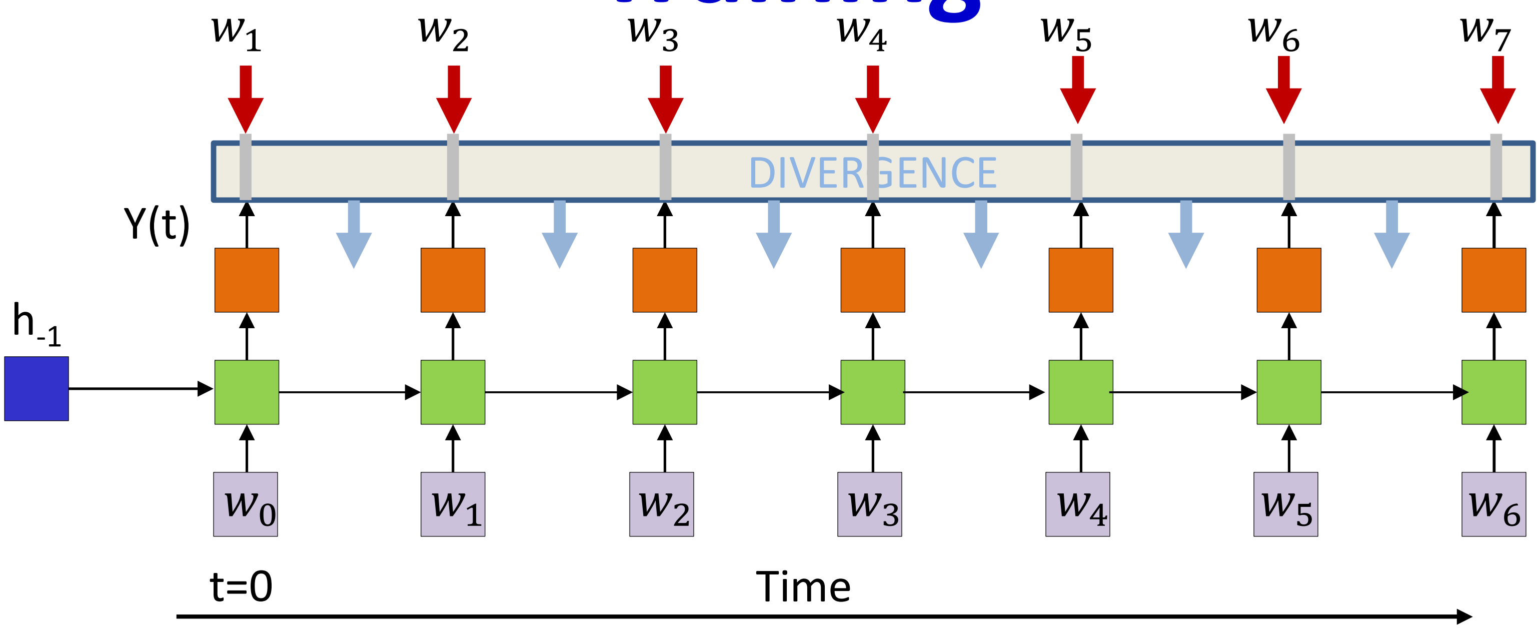
Input: Sequence of characters (presented as one-hot vectors).

Target output after observing “h e l l” is “o”



- Input presented as one-hot vectors
  - Actually “embeddings” of one-hot vectors
- Output: probability distribution over characters
  - Must ideally peak at the target character

Training



- Input: symbols as one-hot vectors
  - Dimensionality of the vector is the size of the "vocabulary"
- Output: Probability distribution over symbols

$$Y(t,i) = P(V_i|w_0 \dots w_{t-1})$$

  - V_i is the i-th symbol in the vocabulary
- Divergence

The probability assigned to the correct next word

$$Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big) = \sum_t KL\big(Y_{target}(t), Y(t)\big) = -\sum_t \log Y(t, w_{t+1})$$

Brief detour: Language models

- Modelling language using time-synchronous 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 …]

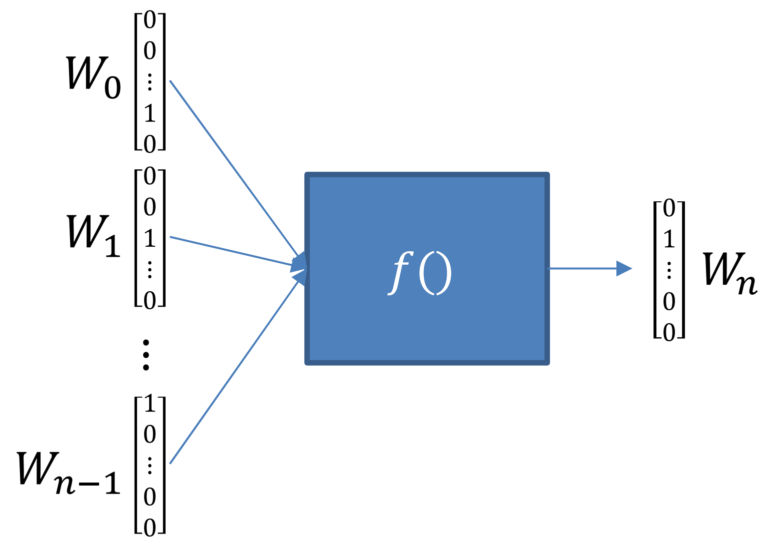
- Characters can be similarly represented
  - English will require about 100 characters, to include both cases, special characters such as commas, hyphens, apostrophes, etc., and the space character

Predicting words

Four score and seven years ???

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

Nx1 one-hot vectors



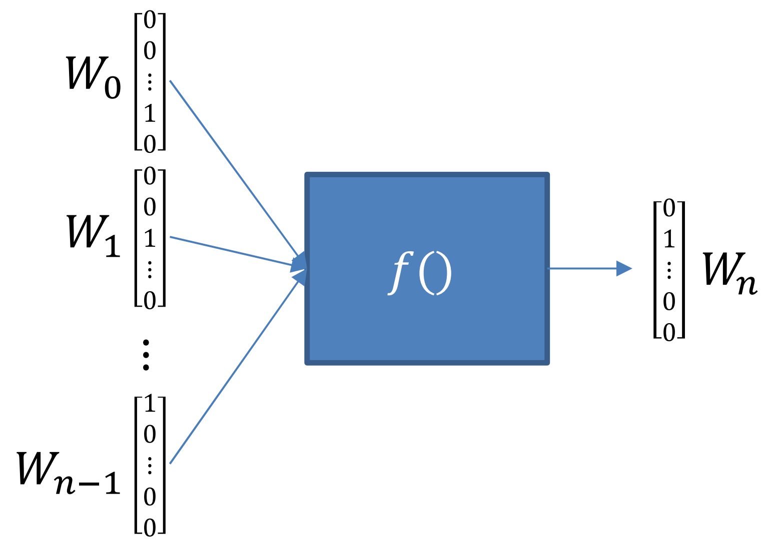
- Given one-hot representations of $W_0 \ldots W_{n-1}$, predict W_n

Predicting words

Four score and seven years ???

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

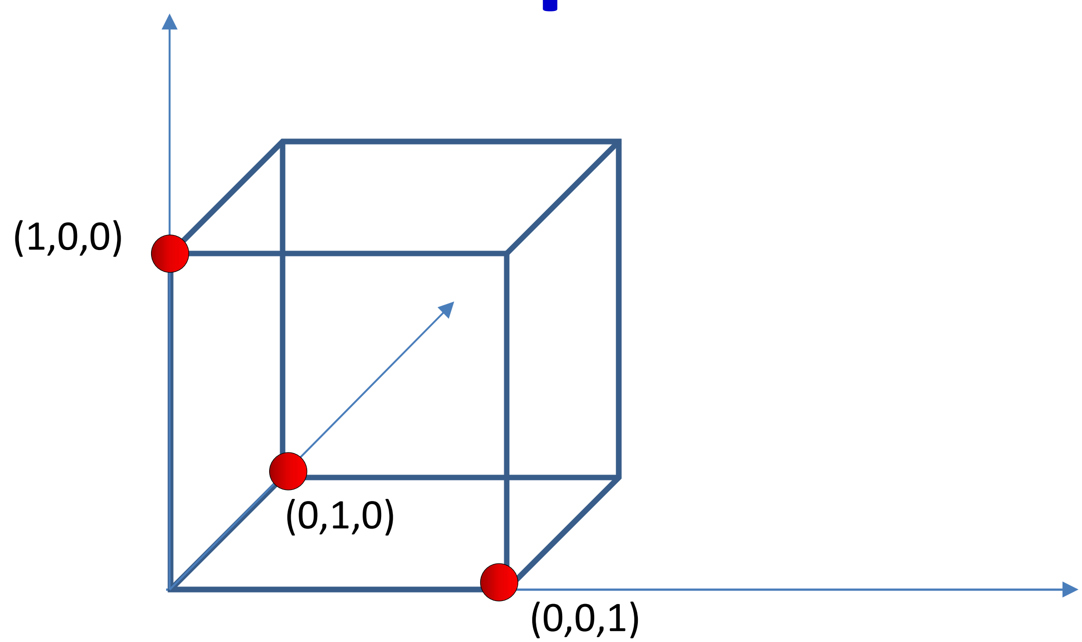
Nx1 one-hot vectors



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

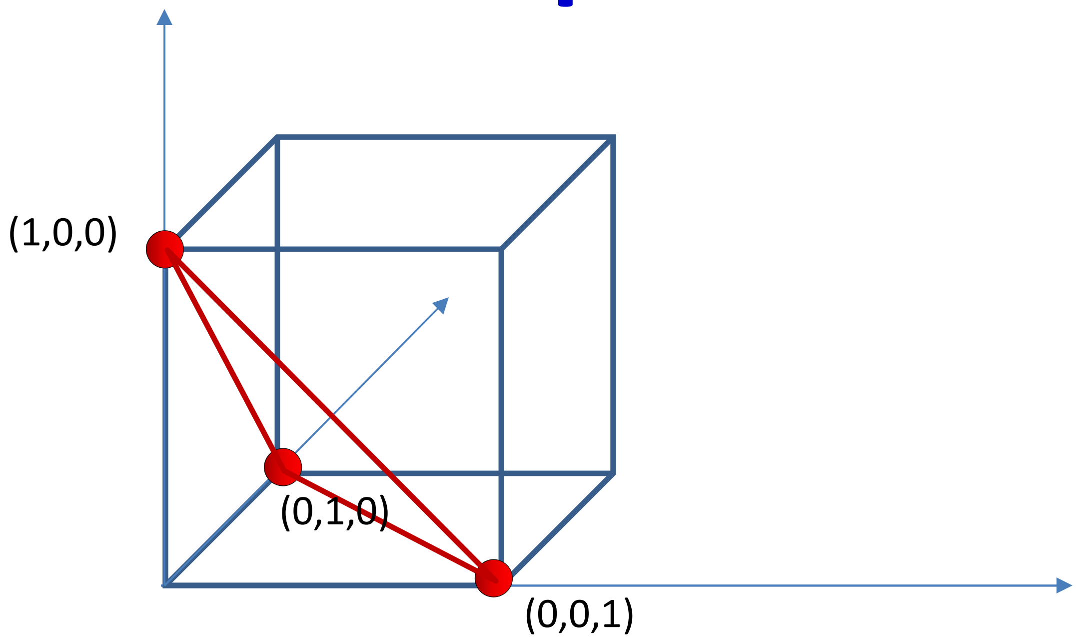
- **Dimensionality problem:** All inputs $W_0 \ldots W_{n-1}$ are both very high-dimensional and very sparse

The one-hot representation



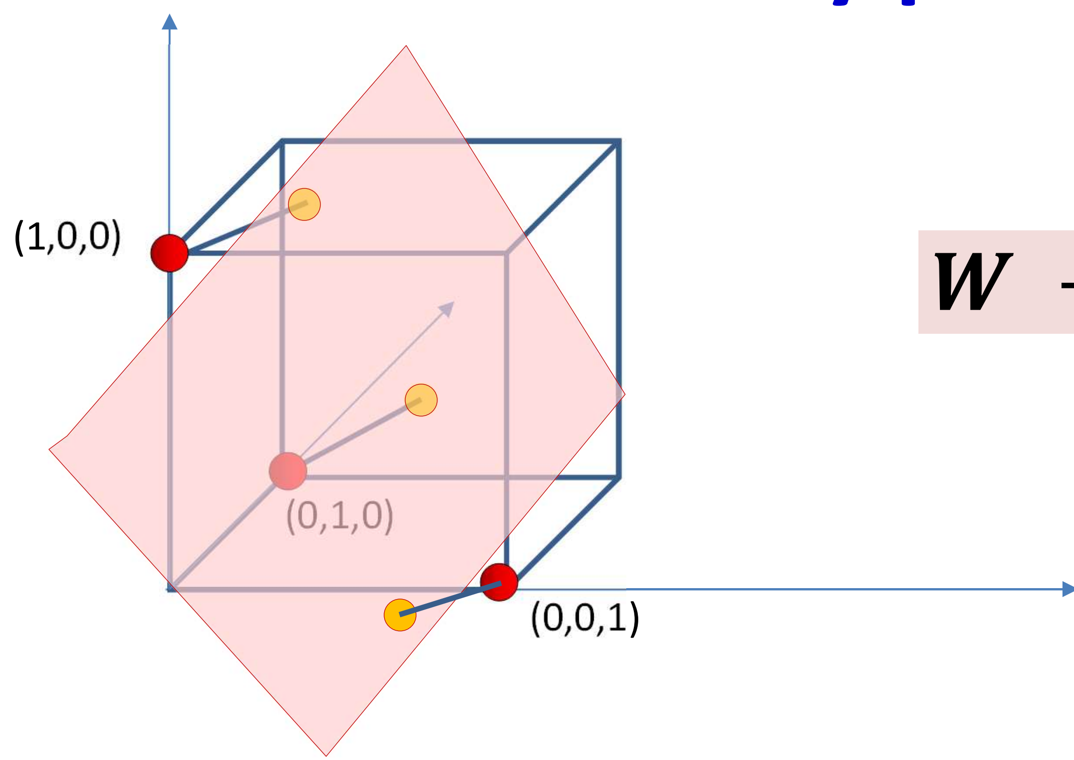
- The one hot representation uses only N corners of the 2^N corners of a unit cube
  - Actual volume of space used = 0
    - $(1, \varepsilon, \delta)$ has no meaning except for $\varepsilon = \delta = 0$
  - Density of points: $\mathcal{O}\left(\frac{N}{r^N}\right)$
- This is a tremendously inefficient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

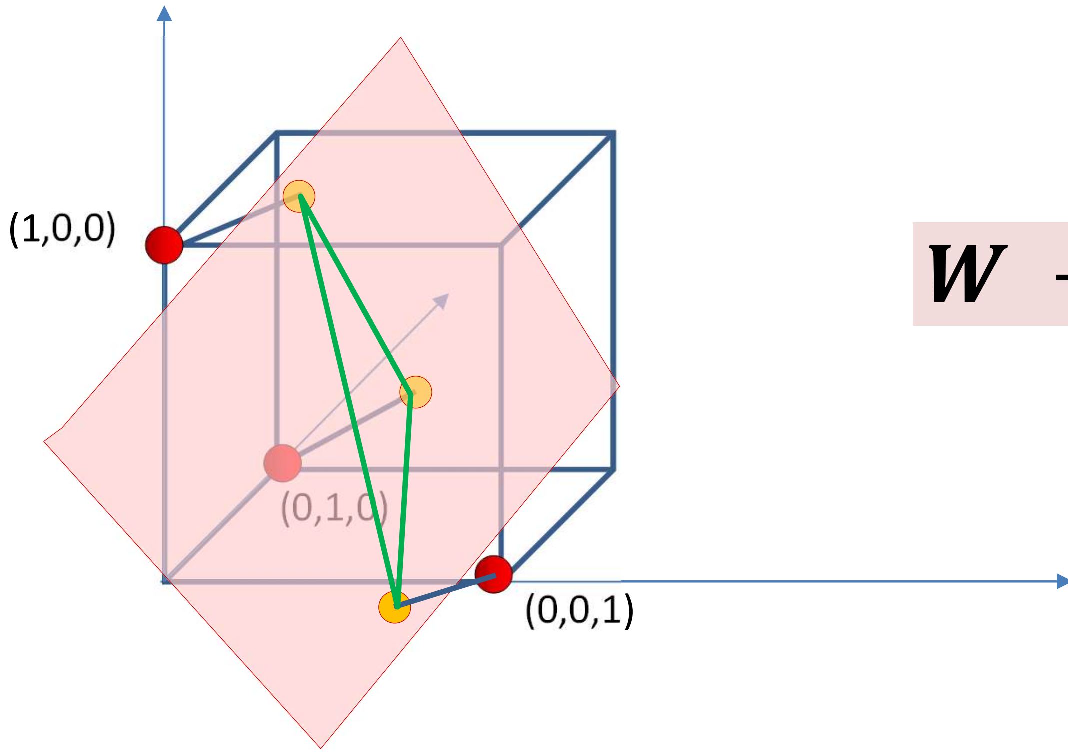
Solution to dimensionality problem



$$\boldsymbol{W} \rightarrow \boldsymbol{PW}$$

- 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



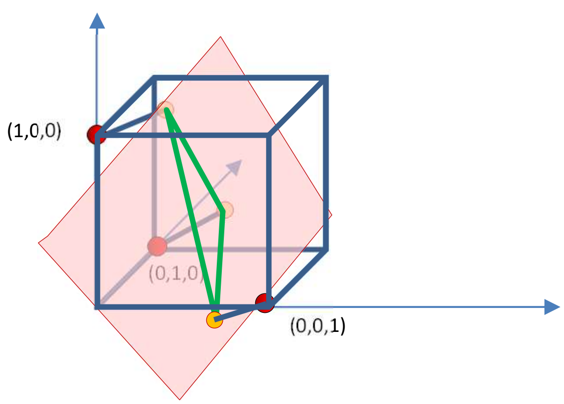
$$W \rightarrow PW$$

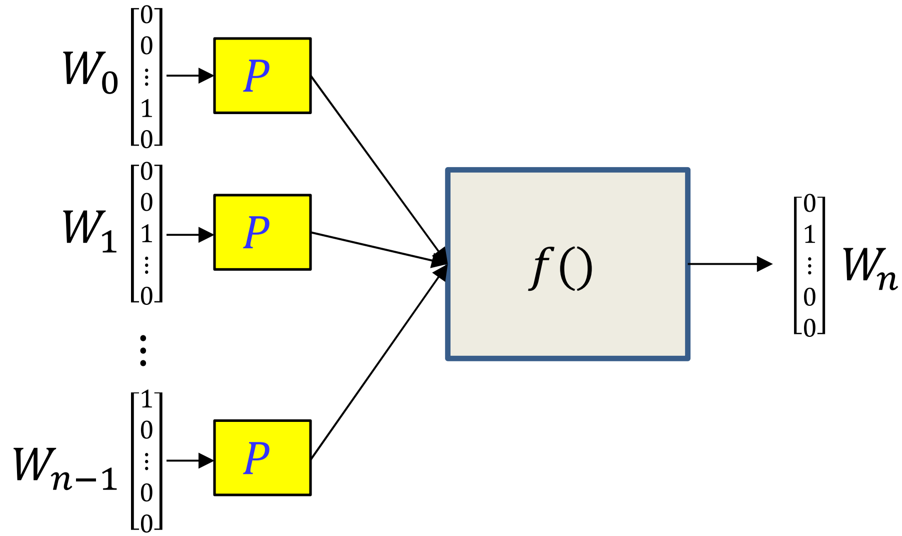
- Project the points onto a lower-dimensional subspace
  - Or more generally, a linear transform into a lower-dimensional subspace
  - The volume used is still 0, but density can go up by many orders of magnitude
    - Density of points: $\mathcal{O}\left(\frac{N}{r^M}\right)$
  - If properly learned,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

The *Projected* word vectors

Four score and seven years ???

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

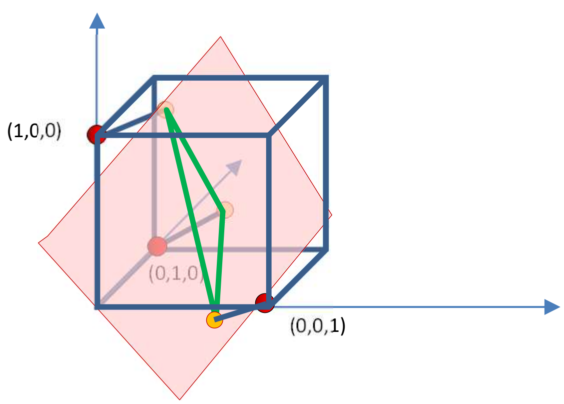


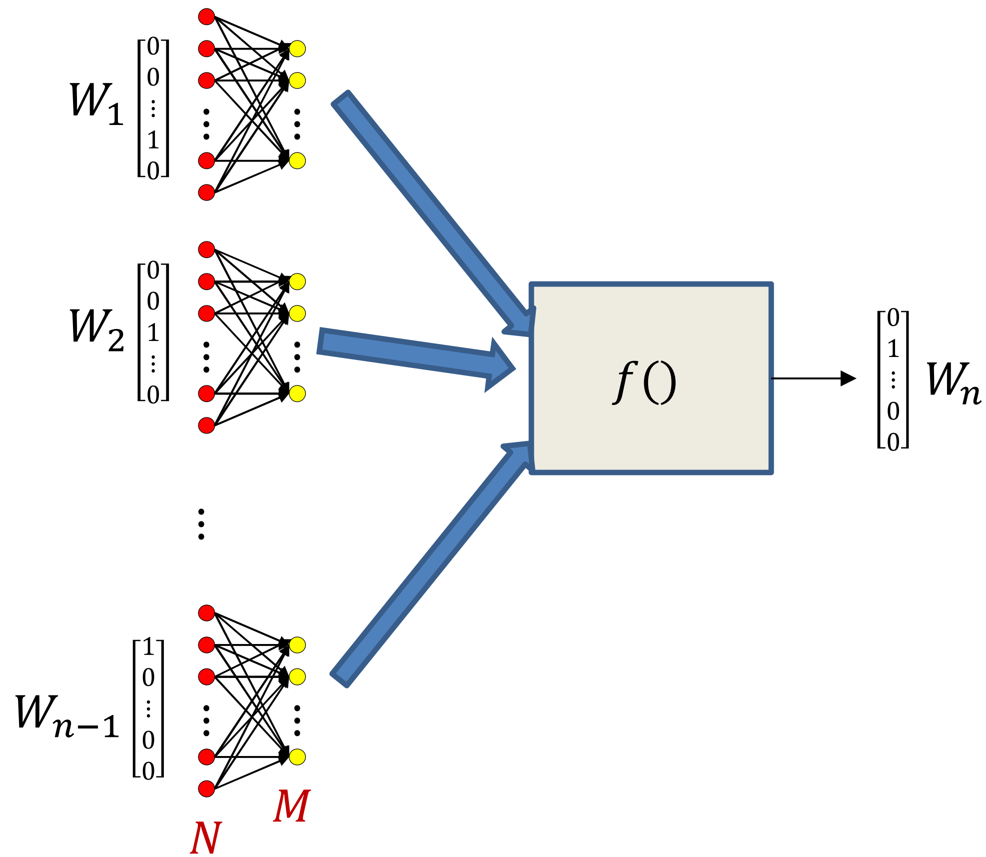


- *Project* the N-dimensional one-hot word vectors into a lower-dimensional space
  - Replace every one-hot vector W_i by PW_i
  - P is an $M \times N$ matrix
  - PW_i is now an M-dimensional vector
  - *Learn P* using an appropriate objective
    - Distances in the projected space will reflect relationships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

"Projection"

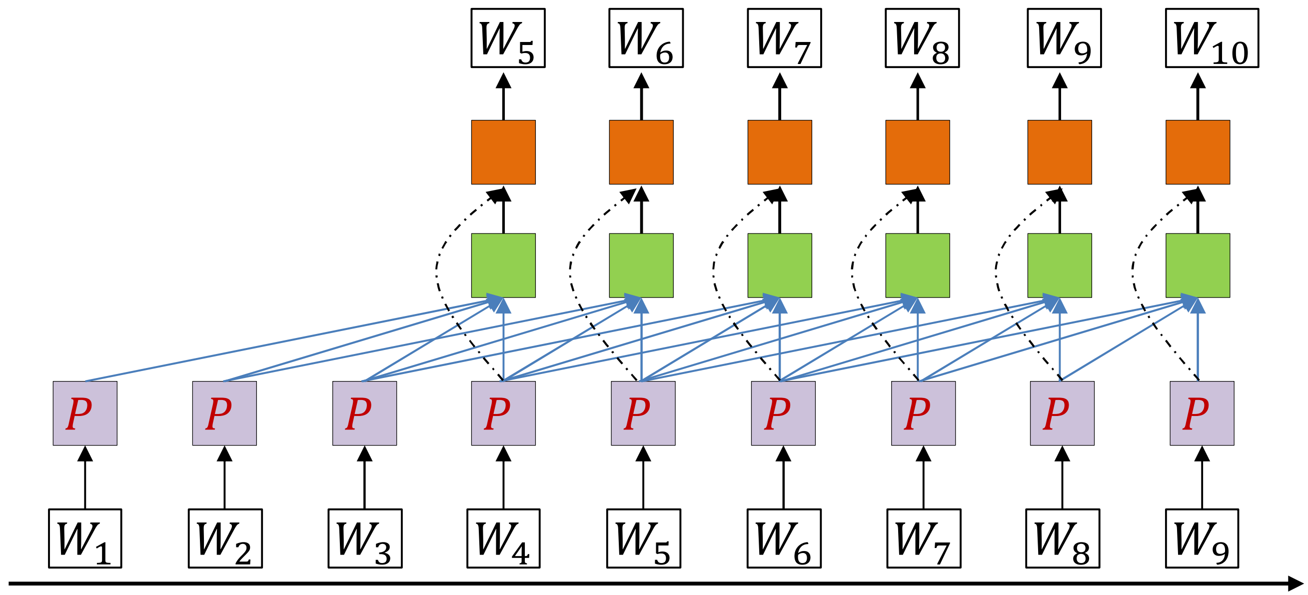
$$W_n = f(PW_1, PW_2, \dots, PW_{n-1})$$





- 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

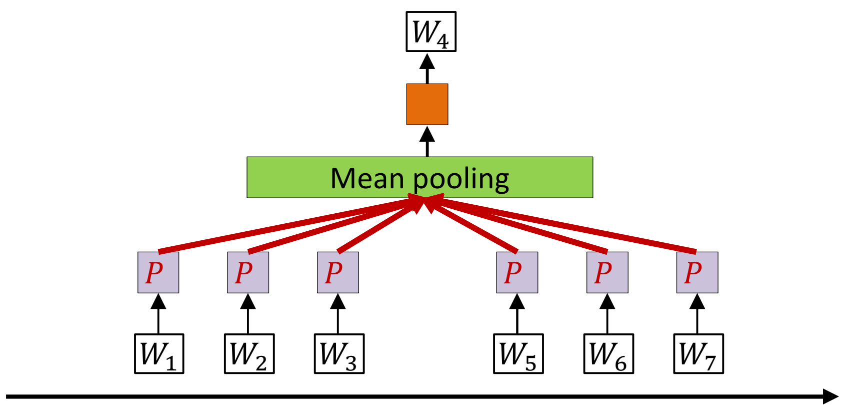
Predicting words: The TDNN model

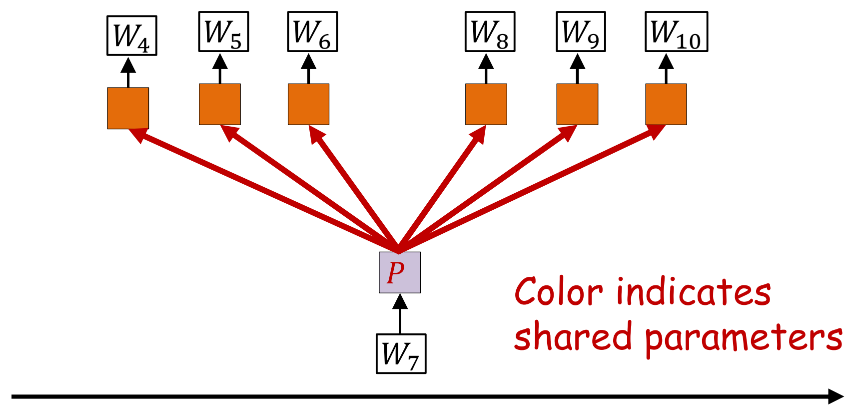


- Predict each word based on the past N words
  - "A neural probabilistic language model", Bengio et al. 2003
  - Hidden layer has Tanh() activation, output is softmax

- 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

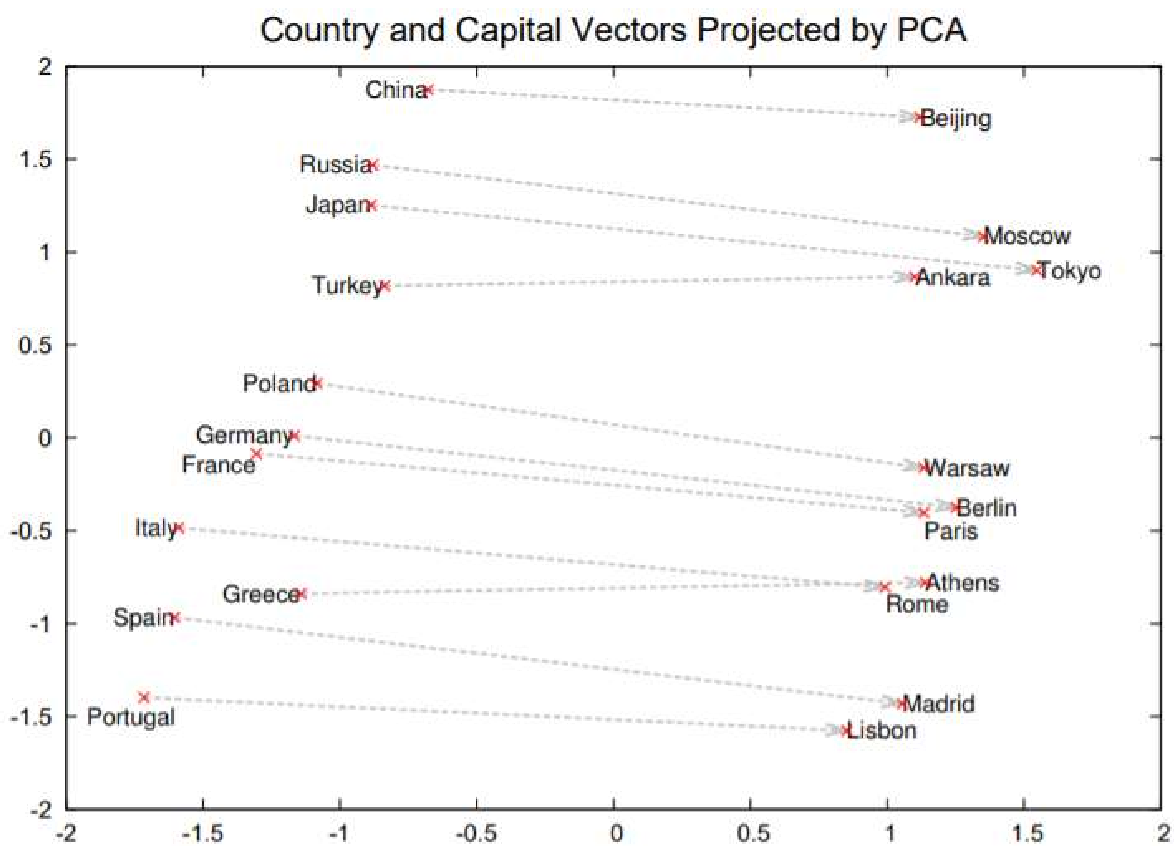
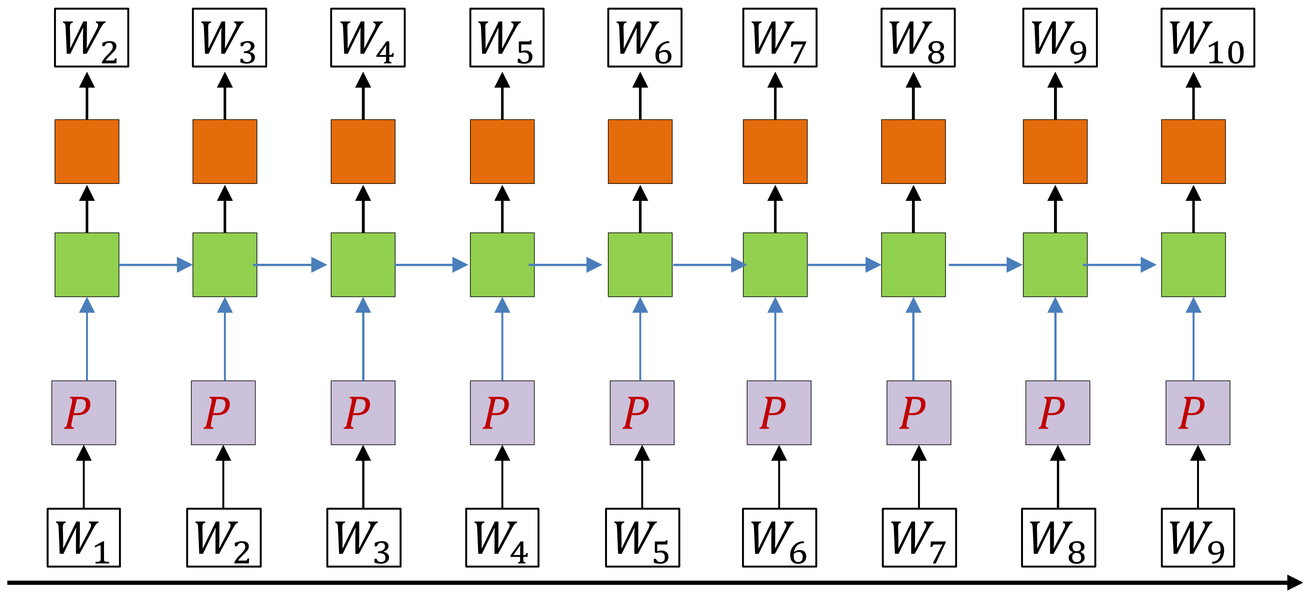


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised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

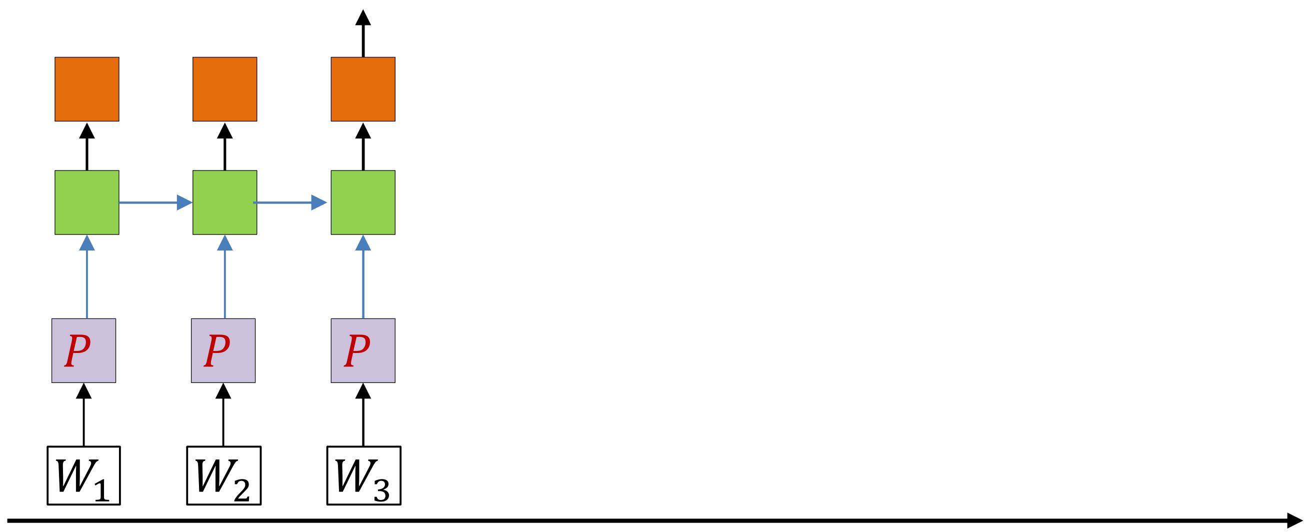
- From Mikolov et al., 2013, “Distributed Representations of Words and Phrases and their Compositionality”

Modelling language



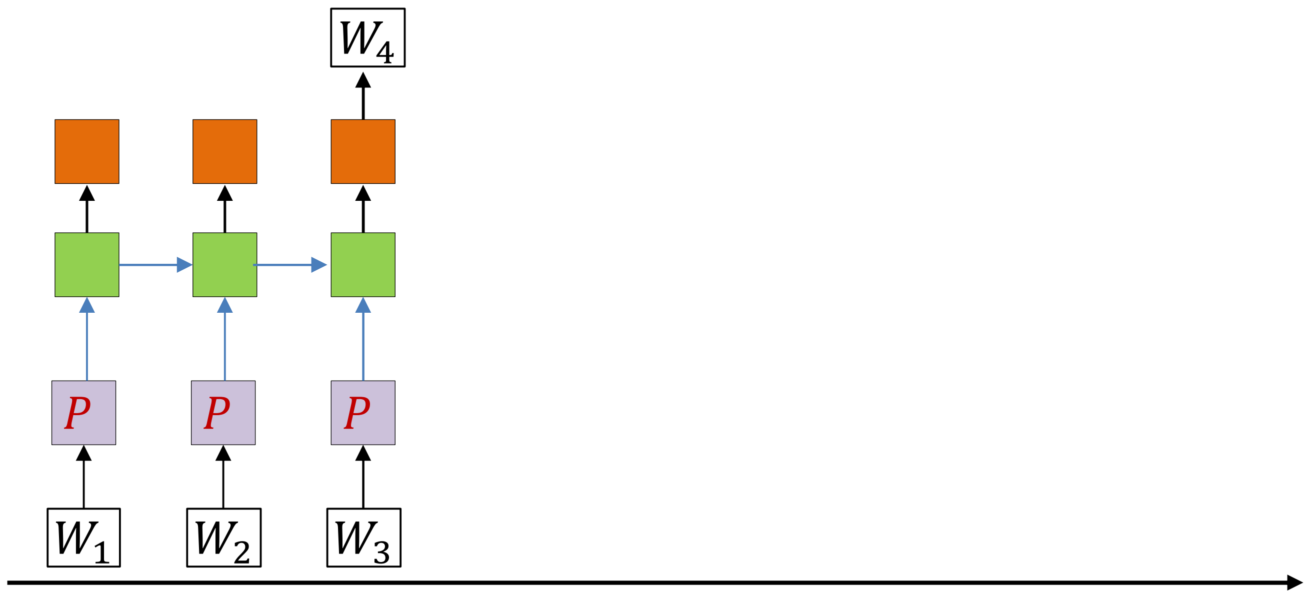
- The hidden units are (one or more layers of) LSTM units
- Trained via backpropagation from a lot of text
  - No explicit labels in the training data: at each time the next word is the label.

Generating Language: Synthesis



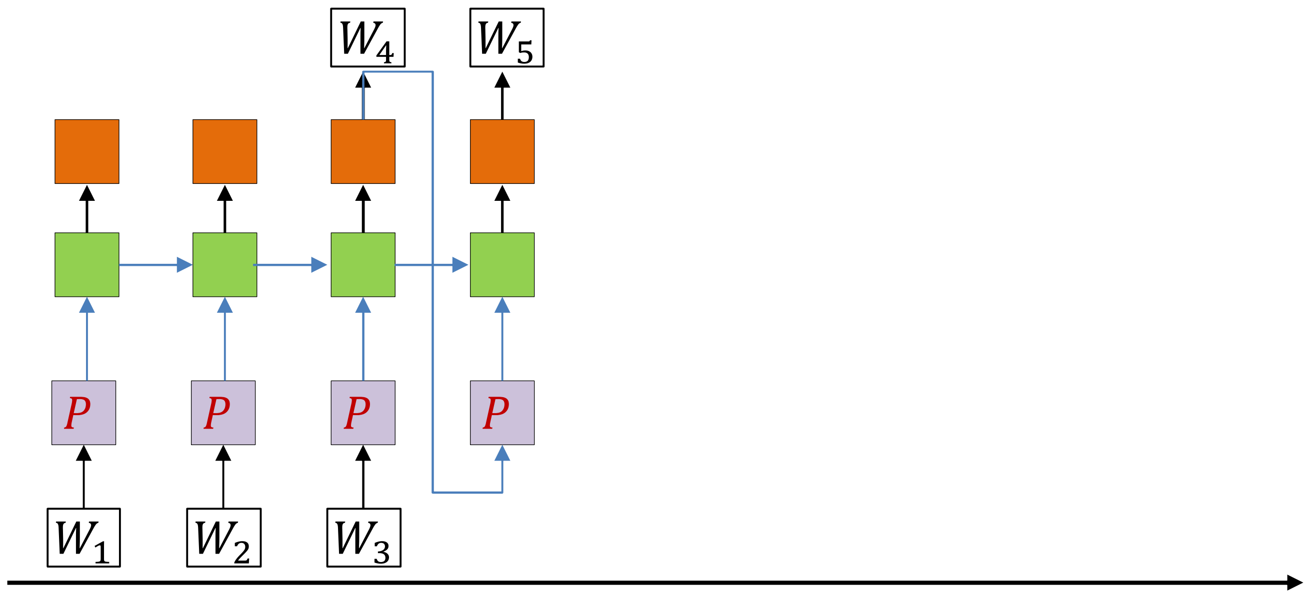
- On trained model : Provide the first few words
  - One-hot vectors
- After the last input word, the network generates a probability distribution over words
  - Outputs an N-valued probability distribution rather than a one-hot vector

Generating Language: Synthesis



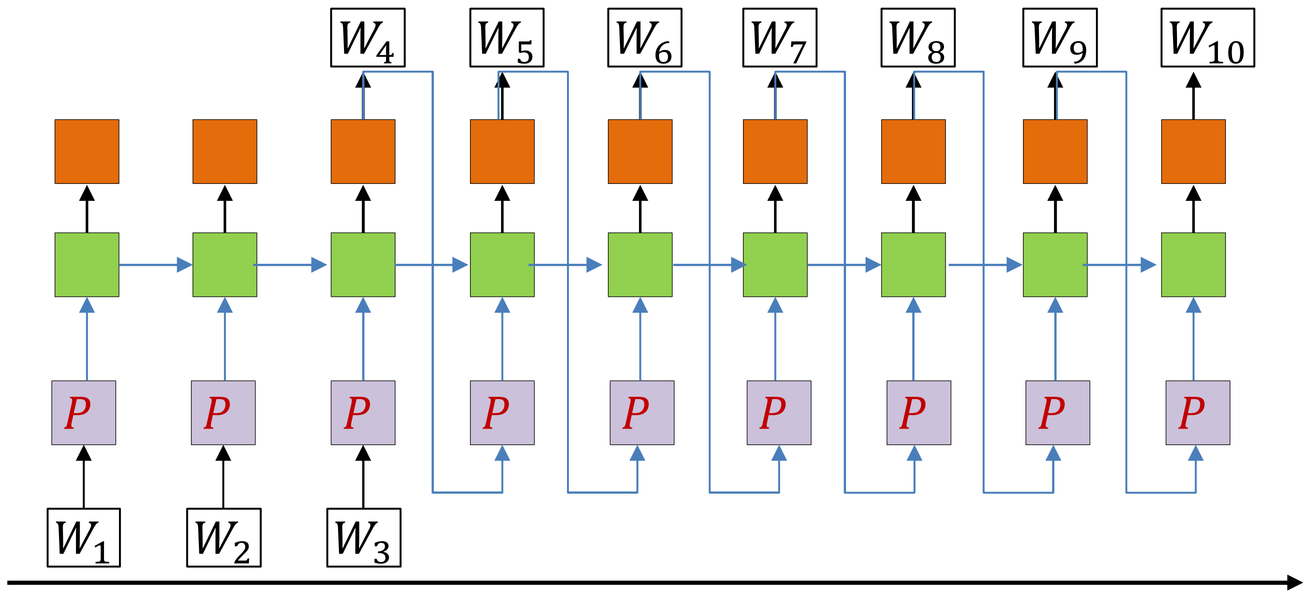
- On trained model : Provide the first few words
  - One-hot vectors
- After the last input word, the network generates a probability distribution over words
  - Outputs an N-valued probability distribution rather than a one-hot vector
- Draw a word from the distribution
  - And set it as the next word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

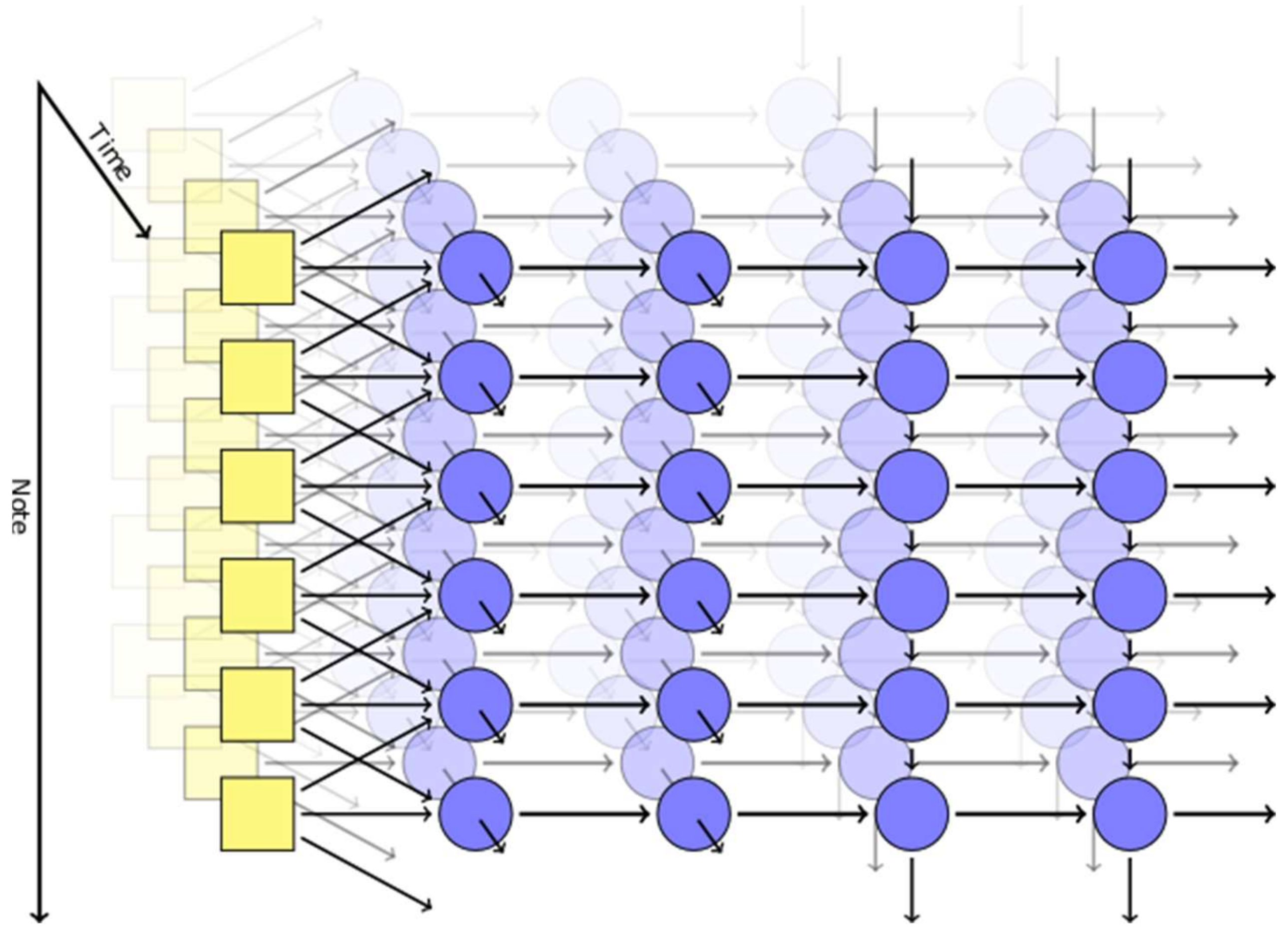
Which open source project?

```
/*
 * 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