Recurrent Neural Networks 2

CSE 849 Deep Learning
Spring 2025

Zijun Cui

Partial slides are adapted from CMU 11-785 Introduction to Deep Learning by Bhiksha Raj

Submission

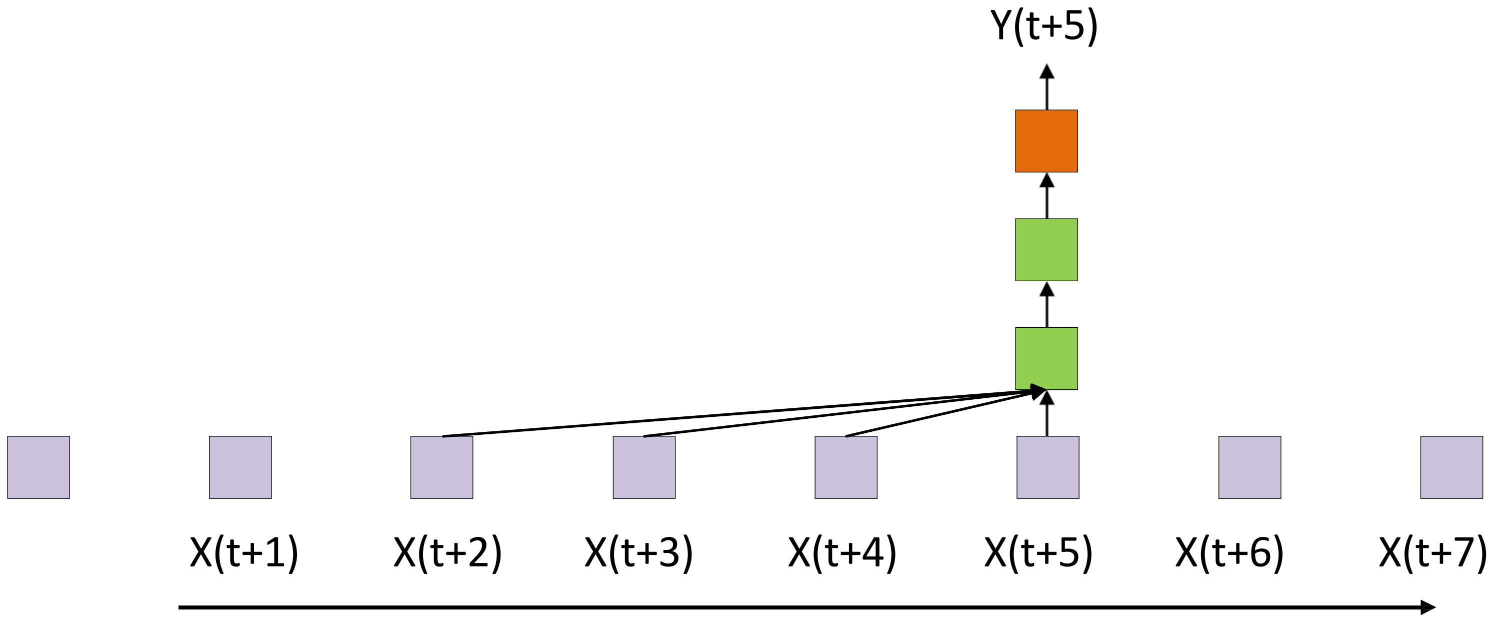
- Project 0: the submitted folder is empty
- Project 1: the submitted pdf file can’t be opened

- Be careful with your submission files to avoid such errors

Outline

- Stability
- Exploding/Vanishing gradients
- LSTM

The behavior of recurrence..

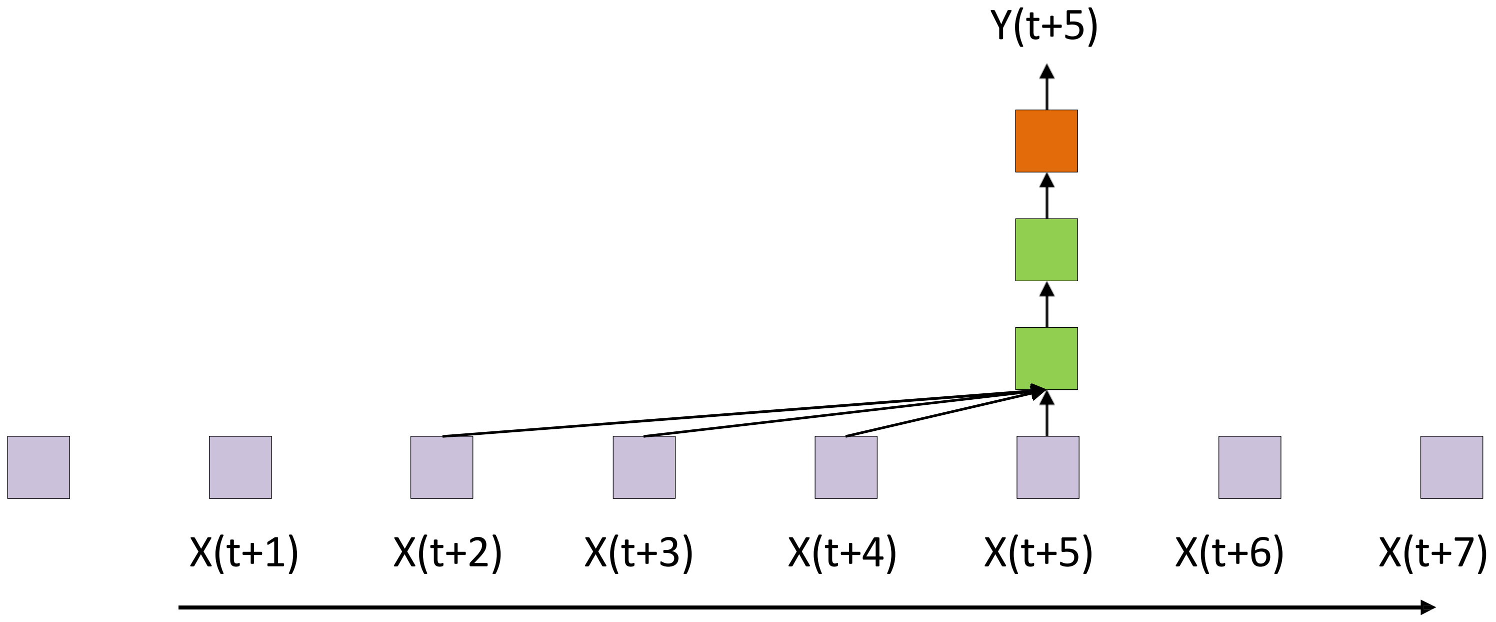


- Returning to an old model..

$$Y(t) = f(X(t-i), i = 0 \dots K)$$

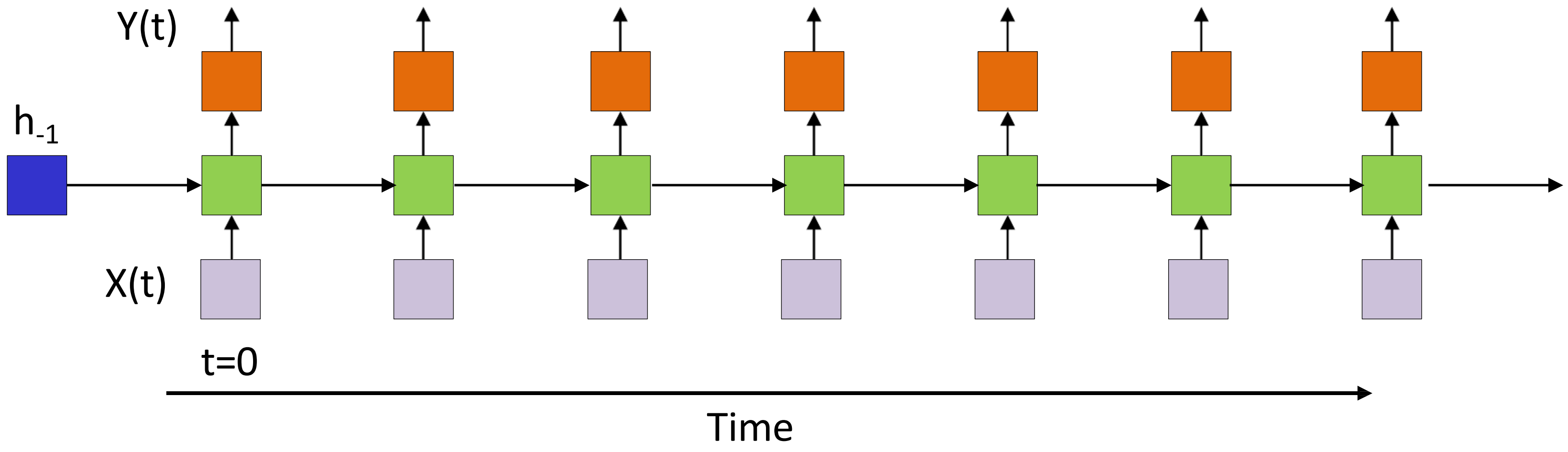
- When will the output "blow up"?

"BIBO" Stability



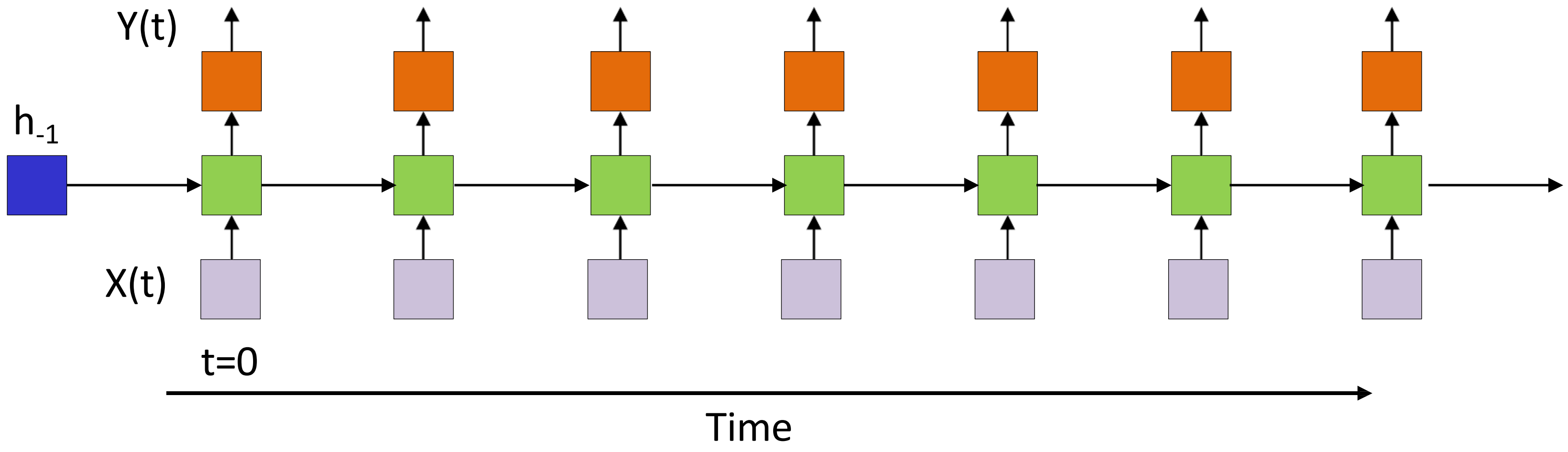
- Time-delay structures have bounded output if
  - The function $f()$ has bounded output for bounded input
    - Which is true of almost every activation function
  - $X(t)$ is bounded
- "Bounded Input Bounded Output" stability
  - This is a highly desirable characteristic

Is this BIBO?



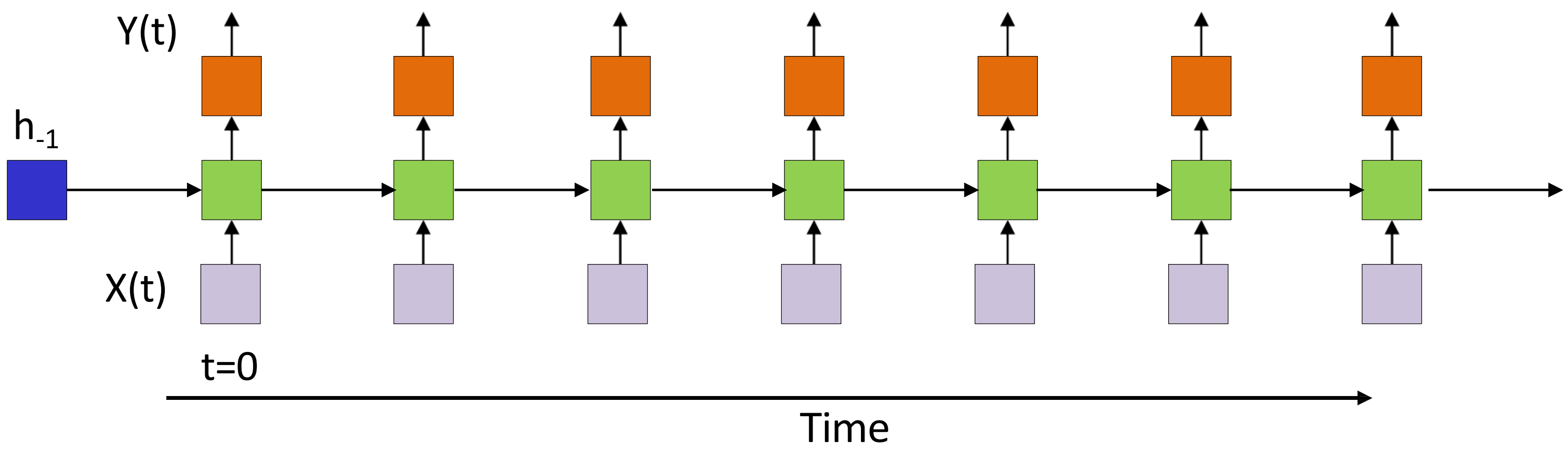
- Will this necessarily be BIBO?
  - Guaranteed if output and hidden activations are bounded
  - What if the activations are linear?

Analyzing recurrence



- Sufficient to analyze the behavior of the hidden layer h_t since it carries the relevant information
  - Will assume only a single hidden layer for simplicity

Streetlight effect



- Easier to analyze *linear* systems
  - Will attempt to extrapolate to non-linear systems subsequently

- All activations are identity functions

$h_k = W_h h_{k-1} + W_x x_k$ Using index "k" for time $h_{k-1} = W_h h_{k-2} + W_x x_{k-1}$

Linear systems

- $h_k = W_h h_{k-1} + W_x x_k$ Using index "k" for time $h_{k-1} = W_h h_{k-2} + W_x x_{k-1}$
- $h_k = W_h^2 h_{k-2} + W_h W_x x_{k-1} + W_x x_k$
- $h_k = W_h^{k+1} h_{-1} + W_h^k W_x x_0 + W_h^{k-1} W_x x_1 + W_h^{k-2} W_x x_2 + \cdots$

$H_k(1_{-1}) = W_h^{k+1}$: The response to the initial condition.

$H_k(1_t) = W_h^{k-t} W_x$: The response to an impulse at time t.

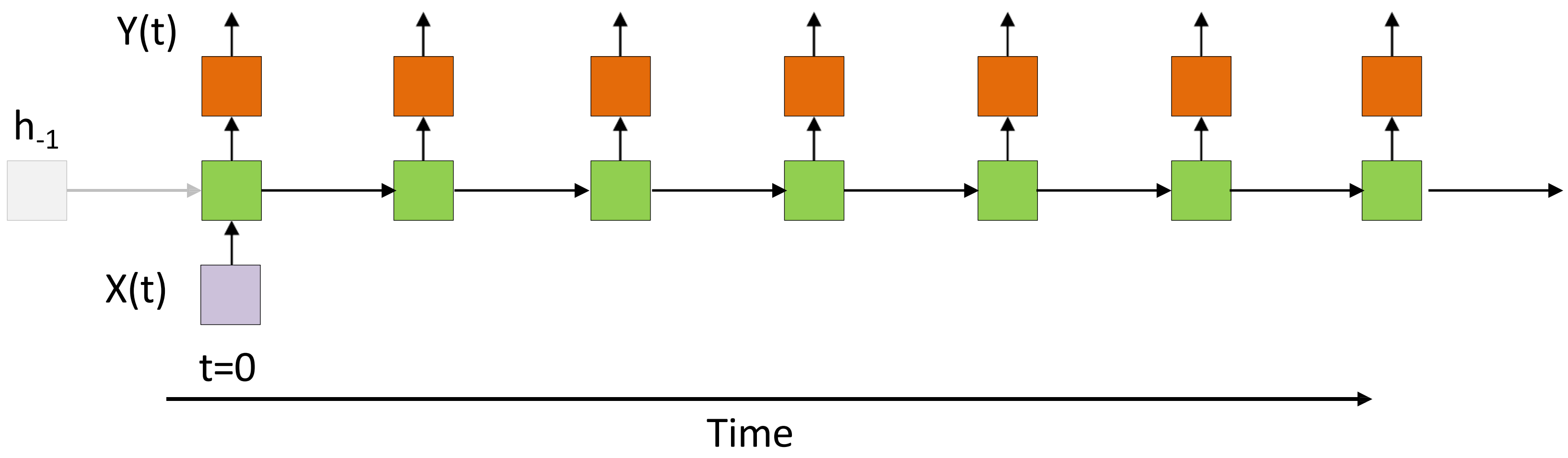
Using impulse response functions:

$$h_k = H_k(h_{-1}) + H_k(x_0) + H_k(x_1) + H_k(x_2) + \cdots$$

$$= h_{-1} H_k(1_{-1}) + x_0 H_k(1_0) + x_1 H_k(1_1) + x_2 H_k(1_2) + \cdots$$

- Where $H_k(1_t)$ is the hidden response at time k when the input is $[0\ 0\ 0\ \ldots 1\ 0\ ..\ 0]$ (where the 1 occurs in the t-th position) with 0 initial condition
  - The initial condition may be viewed as an input of h_{-1} at $t = -1$

Streetlight effect



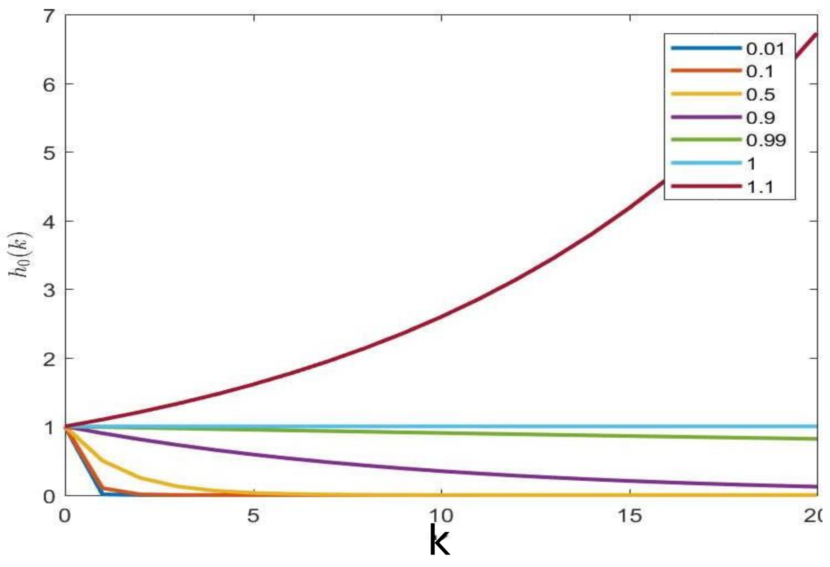
- Principle of superposition in linear systems:

$$h_k = h_{-1}H_k(1_{-1}) + x_0H_k(1_0) + x_1H_k(1_1) + x_2H_k(1_2) + \cdots$$

Linear recursions

- Consider simple, scalar, linear recursion (note change of notation)
  - $h(t) = wh(t-1) + cx(t)$
  - $h_0(t) = w^t cx(0)$

    - Response to a single input at 0



Linear recursions: Vector version

- Vector linear recursion (note change of notation)

$$h(t) = Wh(t-1) + Cx(t)$$

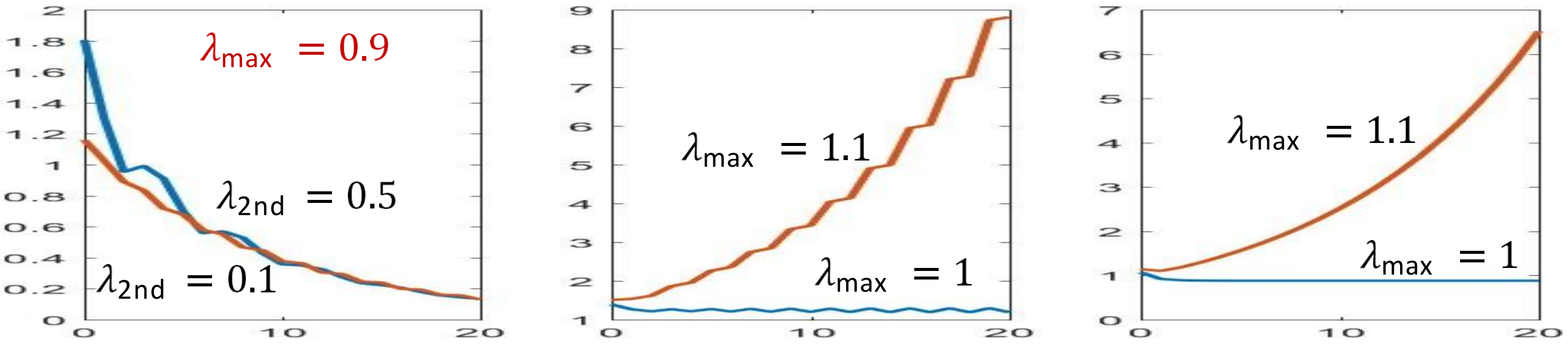
$$h_0(t) = W^t Cx(0)$$

Length of response vector to a single input at 0 is $|h_0(t)|$

- We can write $W = U\Lambda U^{-1}$
  - $Wu_i = \lambda_i u_i$
  - For any vector $x' = Cx$ we can write
    - $x' = a_1 u_1 + a_2 u_2 + \cdots + a_n u_n$
    - $Wx' = a_1 \lambda_1 u_1 + a_2 \lambda_2 u_2 + \cdots + a_n \lambda_n u_n$
    - $W^t x' = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots + a_n \lambda_n^t u_n$
  - $\lim_{t\to\infty} |W^t x'| = a_m \lambda_m^t u_m$ where $m = \operatorname{argmax}_j \lambda_j$

Linear recursions

- Vector linear recursion
  - $h(t) = Wh(t-1) + Cx(t)$
  - $h_0(t) = W^t cx(0)$
    - Response to a single input [1 1 1 1] at 0



Complex Eigenvalues

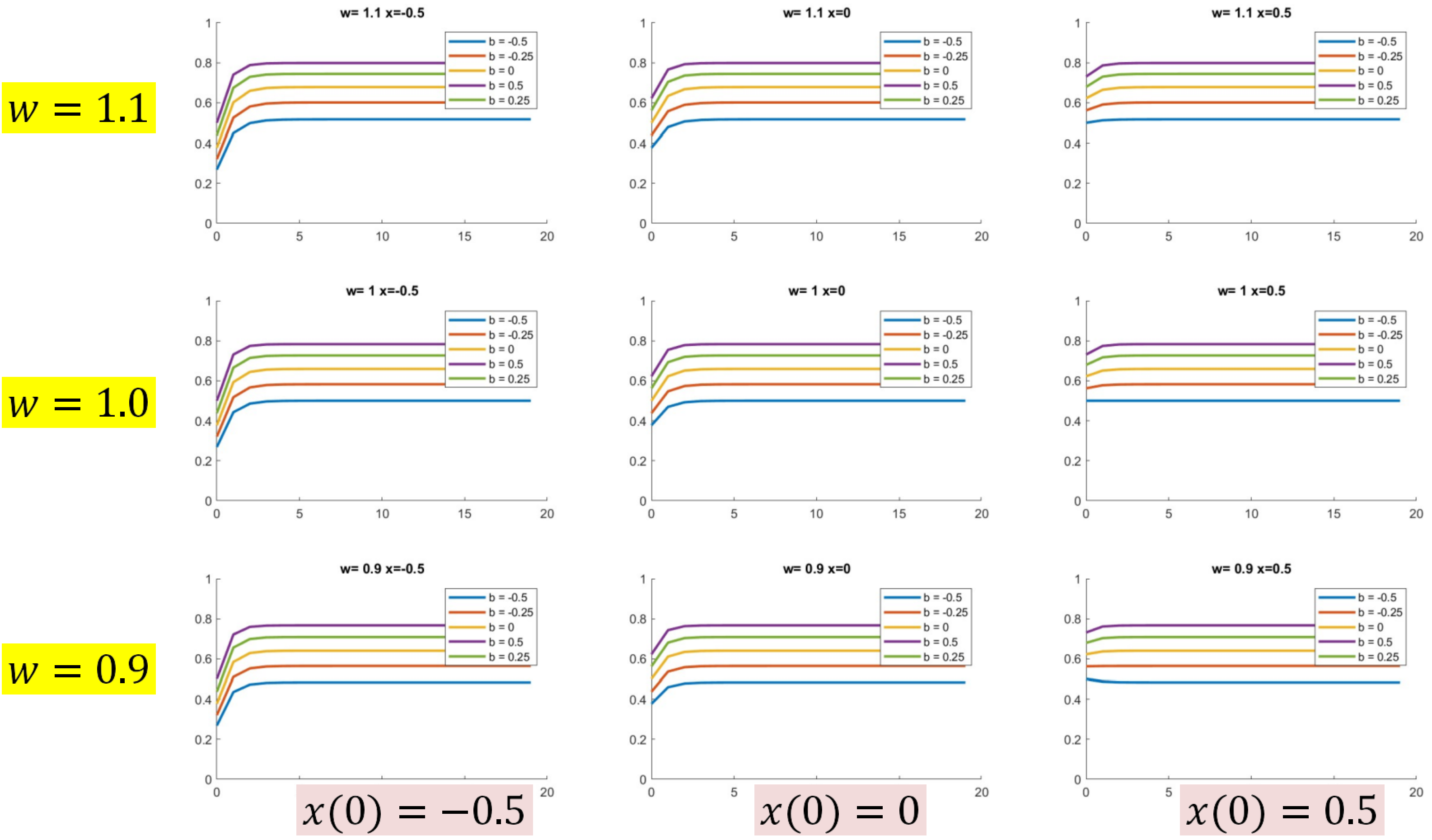
Lesson…

- In linear systems, long-term behavior depends entirely on the eigenvalues of the recurrent weights matrix
  - If the largest Eigen value is greater than 1, the system will “blow up”
  - If it is lesser than 1, the response will “vanish” very quickly
  - Complex Eigen values cause oscillatory response but with the same overall trends
    - Magnitudes greater than 1 will cause the system to blow up

- *The rate of blow up or vanishing depends only on the Eigen values and <u>not on the input</u>*

With non-linear activations: Sigmoid

$$h(t) = sigmoid(wh(t-1) + cx(t) + b)$$



- Scalar recurrence with sigmoid activation
- Final value depends only on b and w but not x

Scalar recurrence

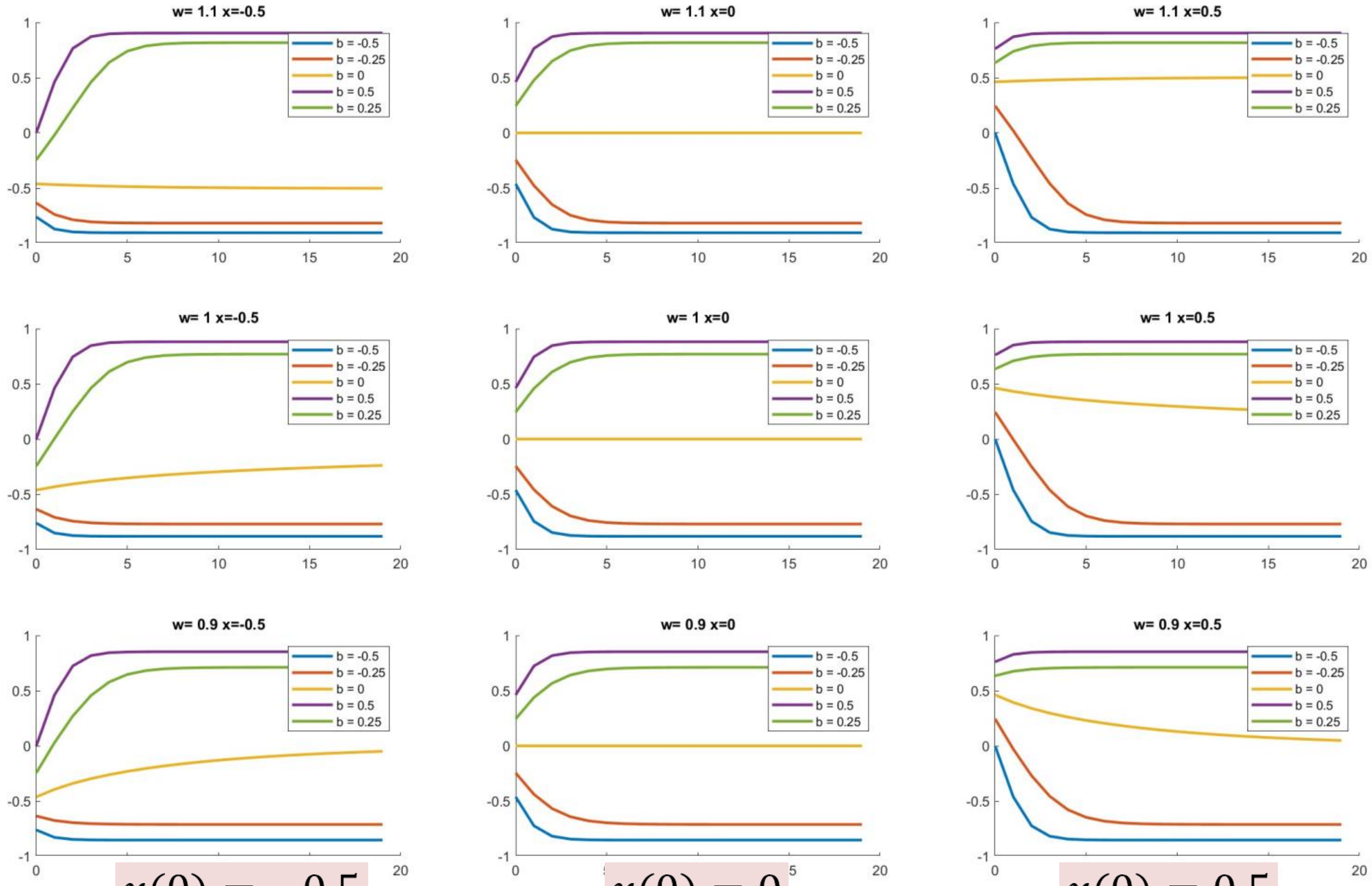
With non-linear activations: Tanh

$$h(t) = tanh(wh(t-1) + cx(t) + b)$$

$w = 1.1$

$w = 1.0$

$w = 0.9$



$x(0) = -0.5$ $x(0) = 0$ $x(0) = 0.5$

- Final value depends only on b and w, but not on x
- “Remembers” x value much longer than sigmoid

Scalar recurrence

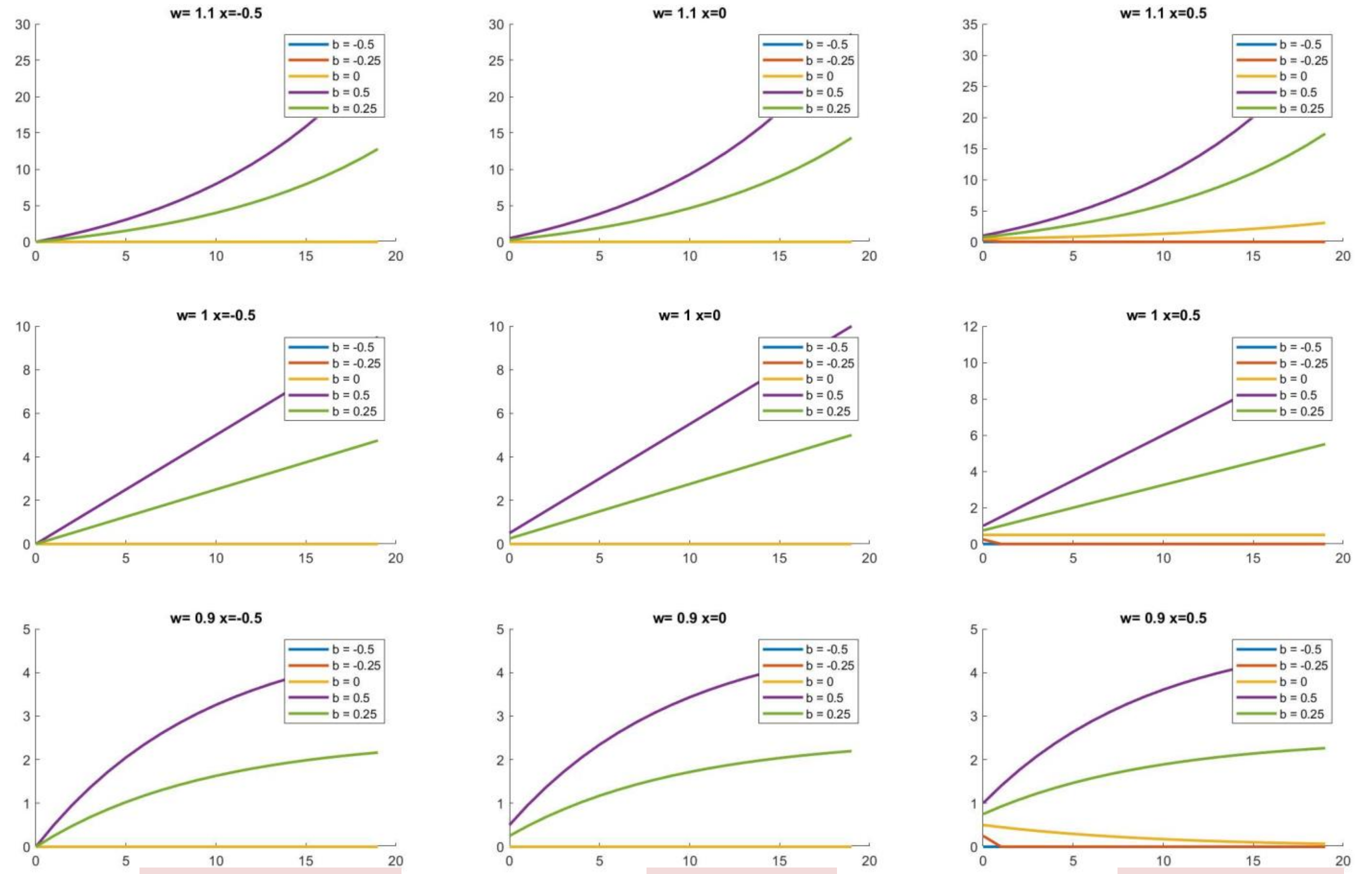
With non-linear activations: RELU

$$h(t) = relu(wh(t-1) + cx(t) + b)$$

$w = 1.1$

$w = 1.0$

$w = 0.9$



$x(0) = -0.5$ $x(0) = 0$ $x(0) = 0.5$

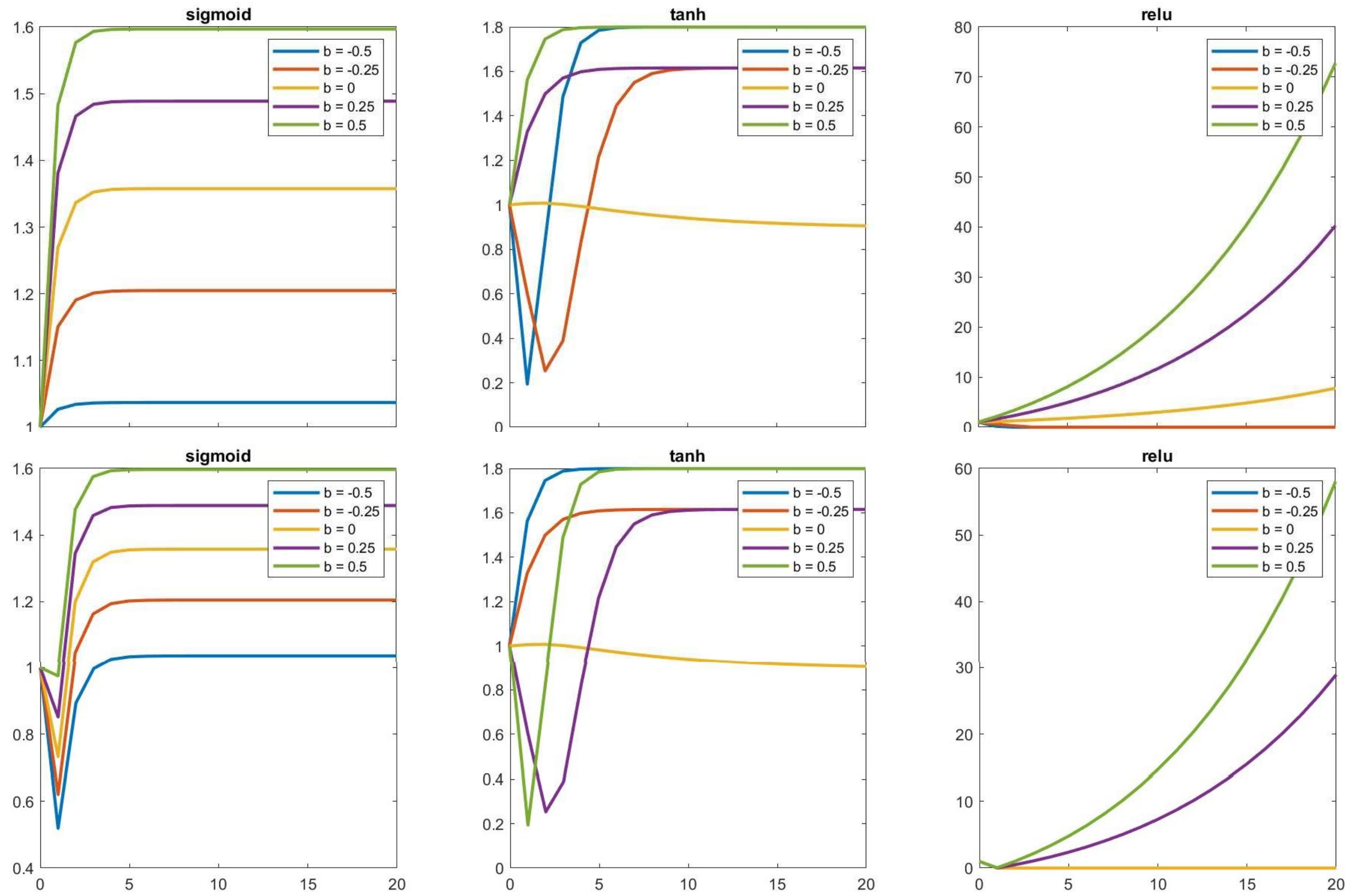
- Relu blows up if $w > 1$, for $x > 0$, and "dies" for $x < 0$
  - Unstable or useless

Scalar recurrence

Vector Process: Max eigenvalue 1.1

$$h(t) = f(Wh(t-1) + Cx(t) + b)$$

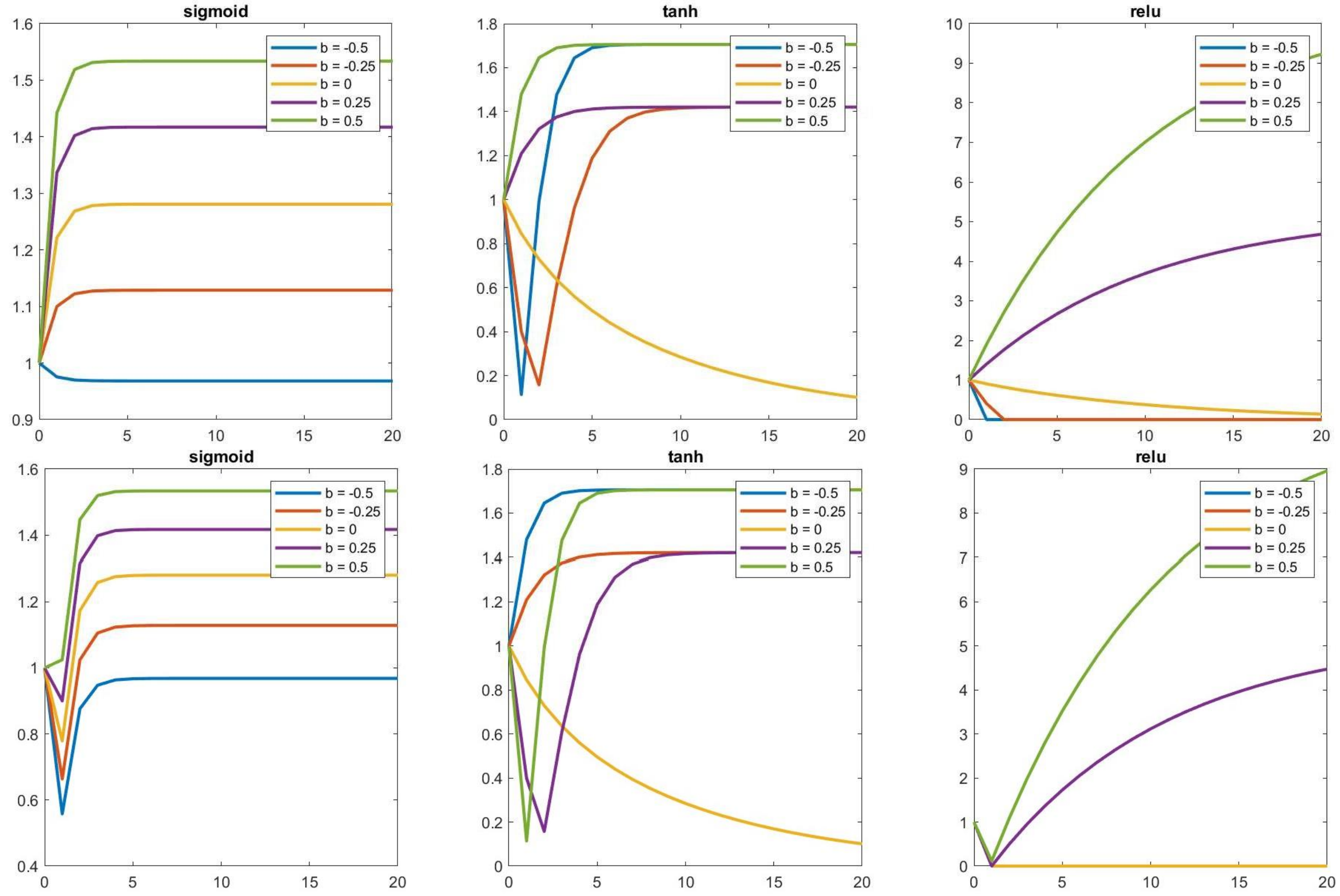
- **Initial x(0):** Top: $[1,1,1,\dots]$, Bottom: $[-1,-1,-1,\dots]$



Vector Process: Max eigenvalue 0.9

$$h(t) = f(Wh(t-1) + Cx(t) + b)$$

- **Initial x(0):** Top: $[1,1,1,\ldots]$, Bottom: $[-1,-1,-1,\ldots]$

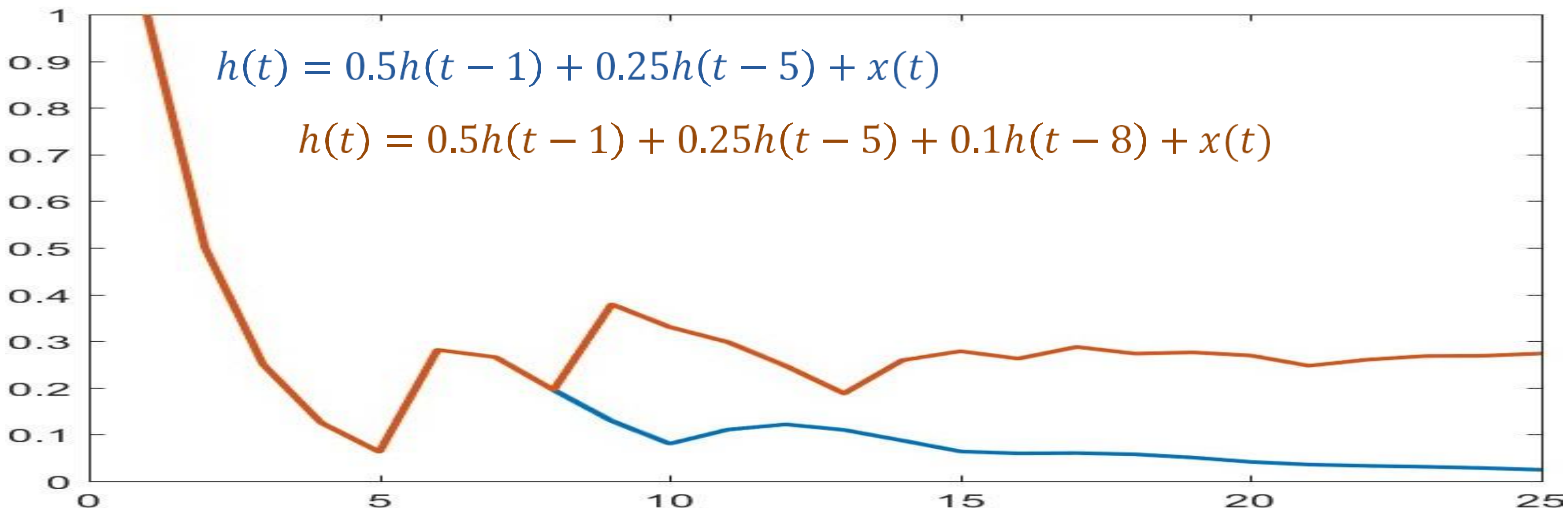


Stability Analysis

- Formal stability analysis considers convergence of “Lyapunov” functions
  - Alternately, Routh’s criterion and/or pole-zero analysis
  - Positive definite functions evaluated at h
  - Conclusions are similar: only the $\tanh$ activation gives us any reasonable behavior
    - And still has very short “memory”
- Lessons:
  - Bipolar activations (e.g. tanh) have the best memory behavior
  - Still sensitive to Eigenvalues of W and the bias
  - Best case memory is short
  - *Exponential memory behavior*
    - *“Forgets” in exponential manner*

How about deeper recursion

- Consider simple, scalar, linear recursion
  - Adding more “taps” adds more “modes” to memory in somewhat non-obvious ways



RNNs..

- Excellent models for time-series analysis tasks
  - Time-series prediction
  - Time-series classification
  - Sequence generation..
  - They can even simplify problems that are difficult for MLPs

- But the memory isn't all that great..
  - Also..

The vanishing gradient problem for deep networks

- A particular problem with training deep networks..
  - (Any deep network, not just recurrent nets)
  - The gradient of the error with respect to weights is unstable..

Training deep networks

- For

$$Div(X) = D\left(f_N\left(W_{N-1} f_{N-1}\left(W_{N-2} f_{N-2}\left(\ldots W_0 X\right)\right)\right)\right)$$

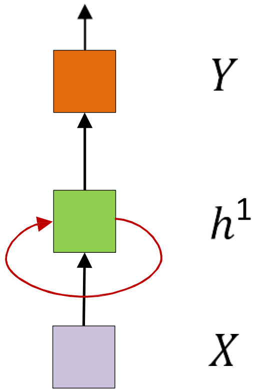
- We get:

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

- Where
  - $\nabla_{f_k} Div$ is the gradient $Div(X)$ of the error w.r.t the output of the kth layer of the network
  - ∇f_n is *jacobian* of $f_N()$ w.r.t. to its current input
  - All blue terms are matrices

Let's consider these Jacobians for an RNN

The Jacobian of the hidden layers for an RNN



$$h_i^{(1)}(t) = f_1\left(z_i^{(1)}(t)\right)$$

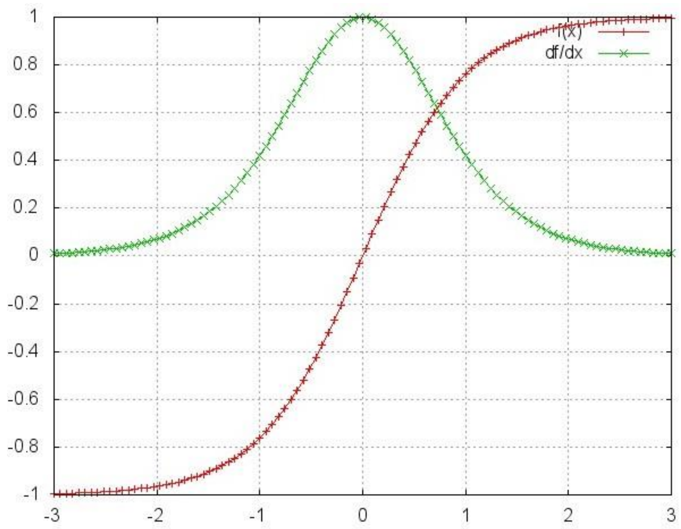
$$\nabla f_t(z_i) = \begin{bmatrix} f'_{t,1}(z_1) & 0 & \cdots & 0 \\ 0 & f'_{t,2}(z_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'_{t,N}(z_N) \end{bmatrix}$$

- $\nabla f_t()$ is the derivative of the output of the (layer of) hidden recurrent neurons with respect to their input
- For recurrent layers with scalar activations, this will be a diagonal matrix
  - The diagonals are the derivatives of the activation funcition
- There is a limit on how much multiplying a vector by the Jacobian will scale it
  - Bounded by the maximum value that the derivative will take

The derivative of the hidden state activation

$$\nabla f_t(z_i) = \begin{bmatrix} f'_{t,1}(z_1) & 0 & \cdots & 0 \\ 0 & f'_{t,2}(z_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'_{t,N}(z_N) \end{bmatrix}$$

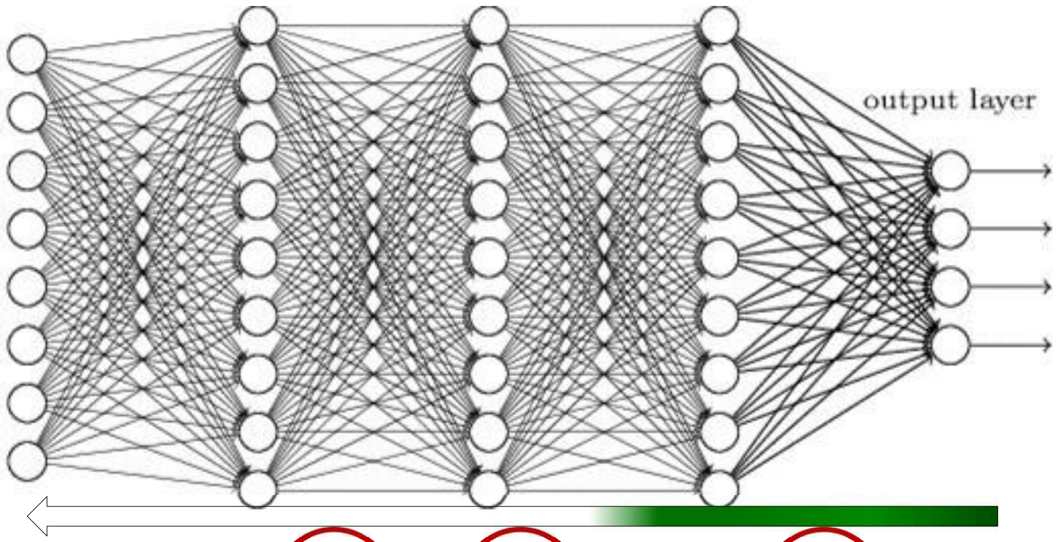
tanh()



- Most common activation functions, such as sigmoid, tanh() and RELU have derivatives that are always less than 1
- The most common activation for the hidden units in an RNN is the tanh()
  - The derivative of $\tanh()$is never greater than 1 (and mostly less than 1)

- **Multiplication by the Jacobian is always a *shrinking* operation**

Training deep networks



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

- As we go back in layers, the Jacobians of the activations constantly *shrink* the derivative
  - After a few layers the derivative of the divergence at any time is totally “forgotten”

What about the weights

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

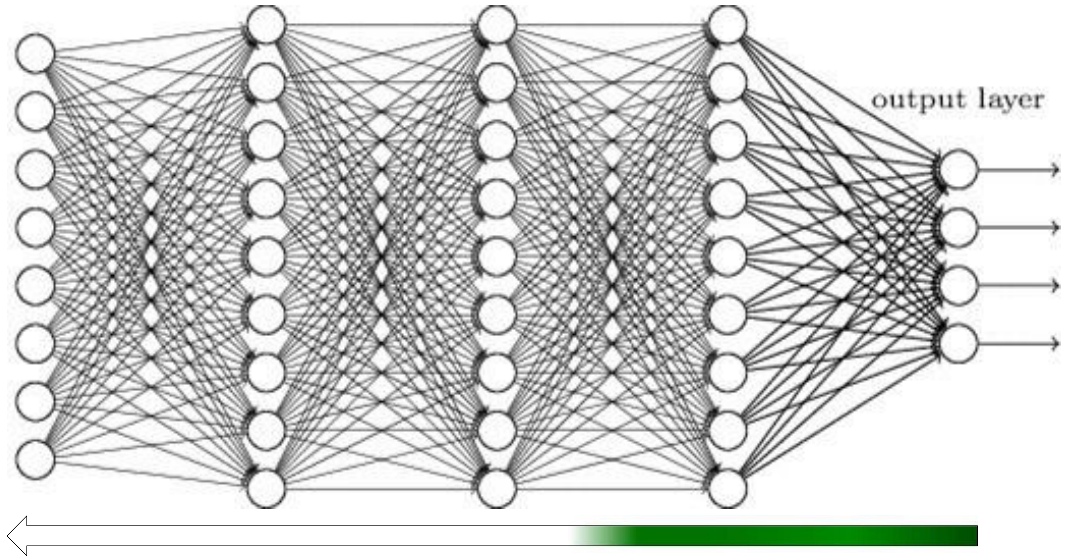
- The chain product for $\nabla_{f_k} Div$ will
  - *Expand* ∇D along directions in which the singular values of the weight matrices are greater than 1
  - *Shrink* ∇D in directions where the singular values are less than 1
  - Repeated multiplication by the weights matrix will result in **Exploding** or **vanishing** gradients

Exploding/Vanishing gradients

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

- Every blue term is a matrix
- ∇D is proportional to the actual error
  - Particularly for L_2 and KL divergence
- The chain product for $\nabla_{f_k} Div$ will
  - *Expand* ∇D in directions where each stage has singular values greater than 1
  - *Shrink* ∇D in directions where each stage has singular values less than 1

Gradient problems in deep networks

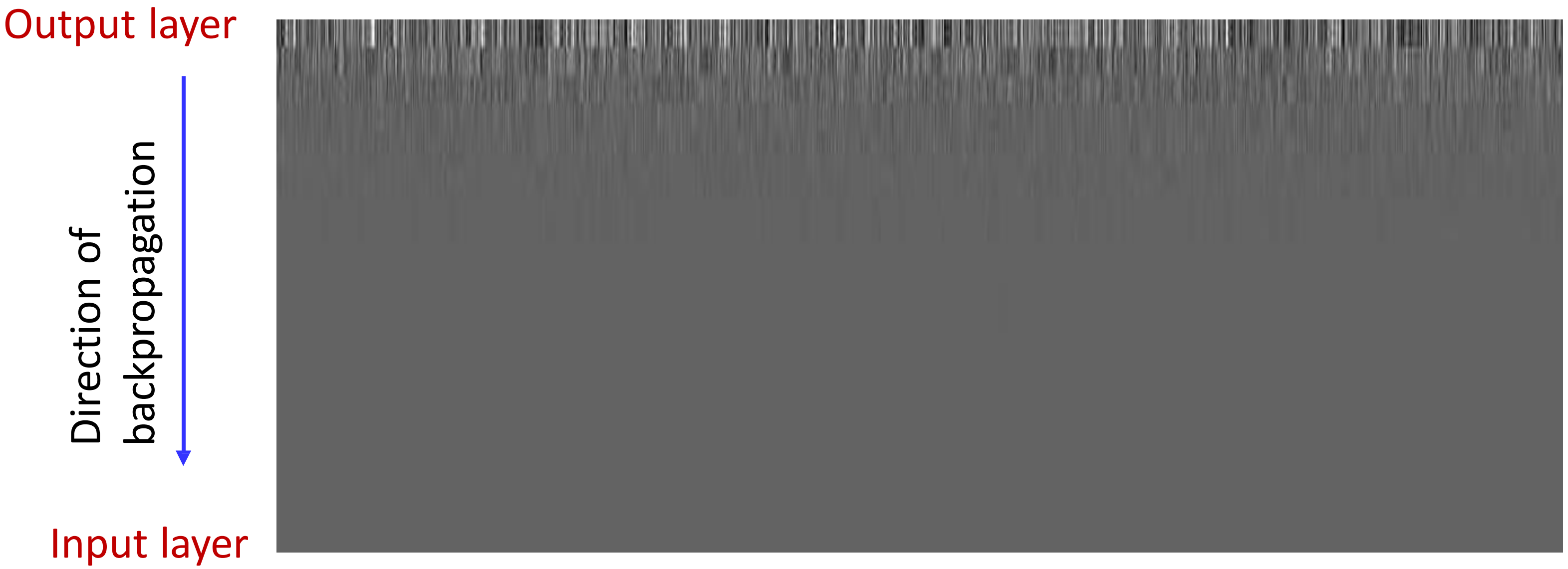


$$\nabla_{f_k} Div = \nabla D . \nabla f_N . W_N . \nabla f_{N-1} . W_{N-1} \ldots \nabla f_{k+1}\ W_{k+1}$$

- The gradients in the lower/earlier layers can *explode* or *vanish*
  - Resulting in insignificant or unstable gradient descent updates
  - Problem gets worse as network depth increases

Vanishing gradient examples..

ELU activation, Batch gradients



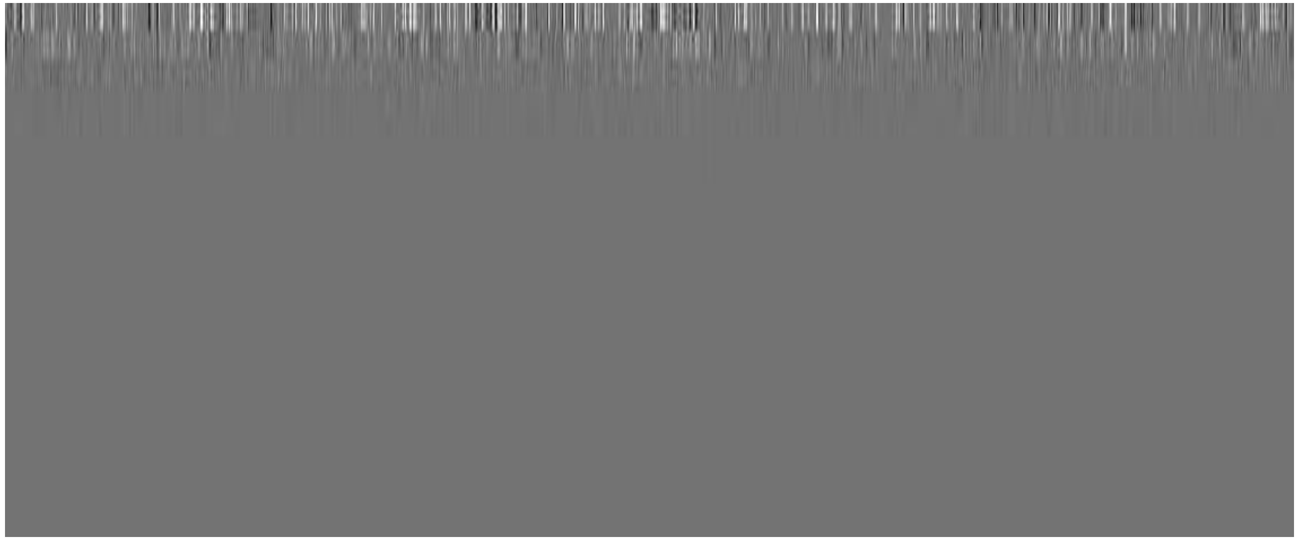
- 19 layer MNIST model
  - Different activations: Exponential linear units (ELU), RELU, sigmoid, tanh
  - Each layer is 1024 units wide Gradients
  - shown at initialization
    - Will actually *decrease* with additional training
- Figure shows $\log\left|\nabla_{W_{neuron}} Div\right|$ where W_{neuron} is the vector of incoming weights to each neuron
  - I.e. the gradient of the loss w.r.t. the entire set of weights to each neuron

Vanishing gradient examples..

RELU activation, Batch gradients

Output layer

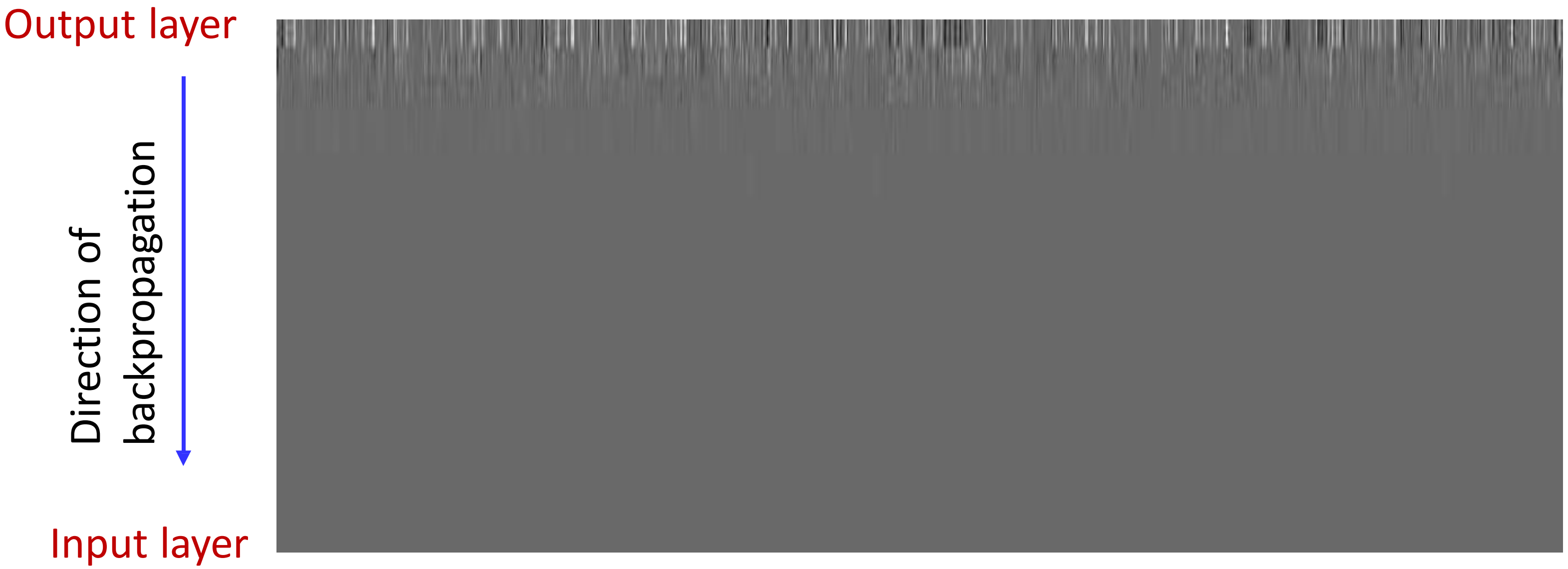
Direction of backpropagation



Input layer

Vanishing gradient examples..

Sigmoid activation, Batch gradients

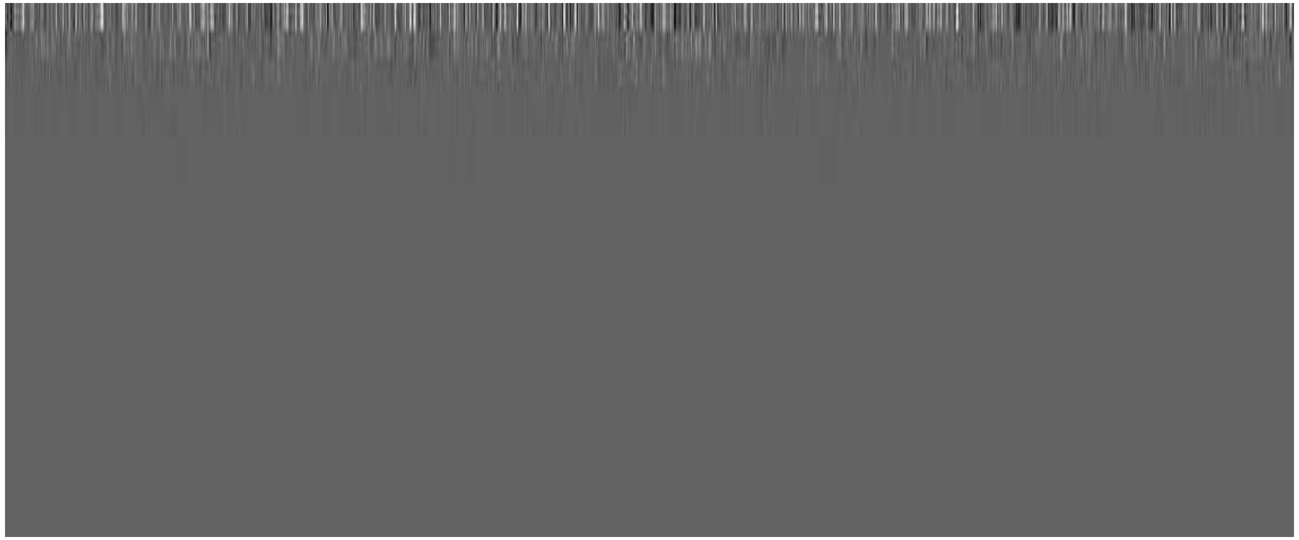


Vanishing gradient examples..

Tanh activation, Batch gradients

Output layer

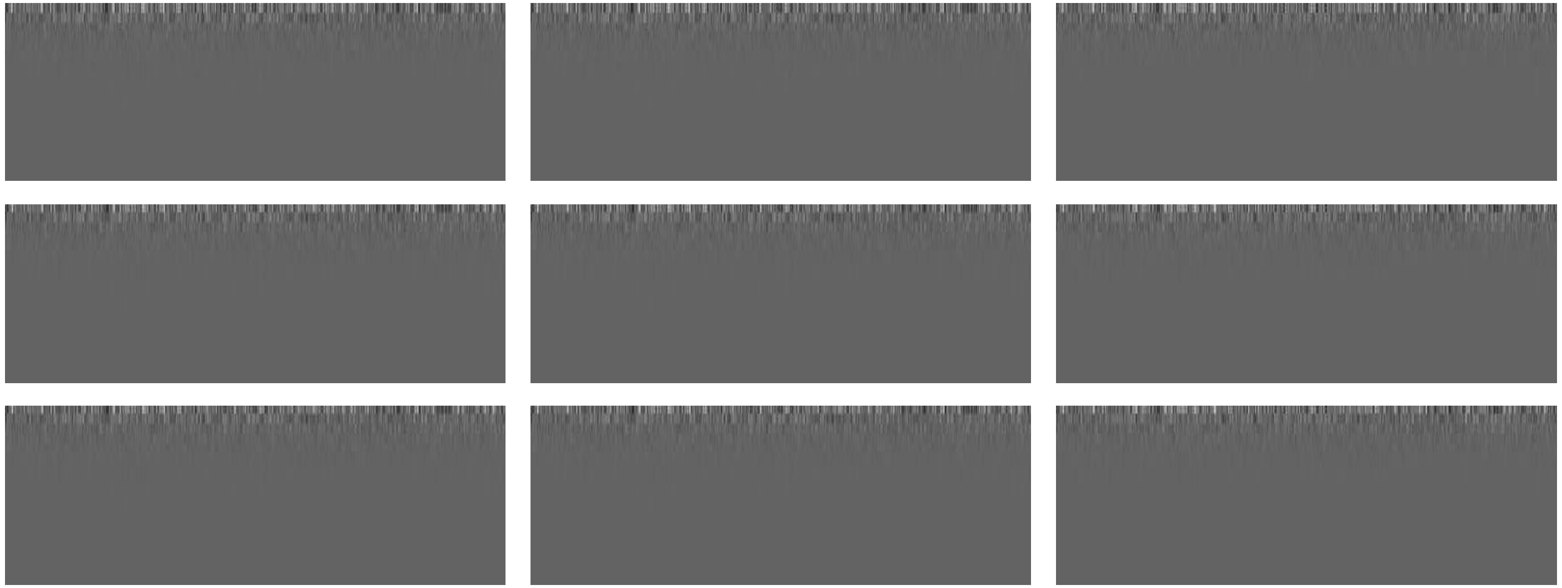
Direction of backpropagation



Input layer

Vanishing gradient examples..

ELU activation, Individual instances



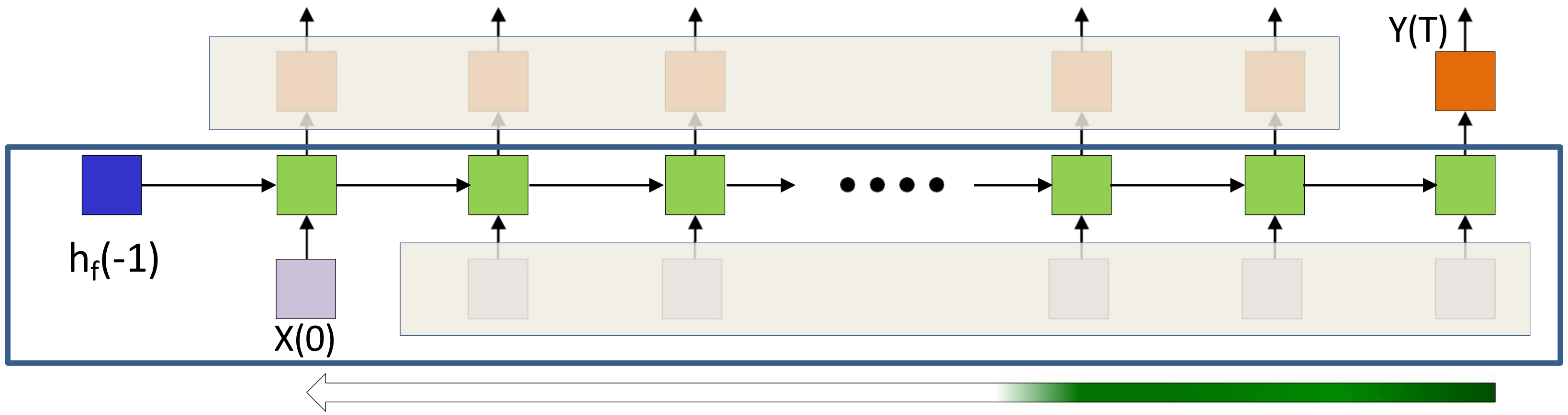
- 19 layer MNIST model
  - Different activations: Exponential linear units, RELU, sigmoid, tanh
  - Each layer is 1024 units wide
  - Gradients shown at initialization
    - Will actually *decrease* with additional training

Vanishing gradients

- ELU activations maintain gradients longest
- But in all cases gradients effectively vanish after about 10 layers!
  - Your results may vary

- Both batch gradients and gradients for individual instances disappear
  - In reality a tiny number will actually blow up.

Recurrent nets are very deep nets



- The relation between $X(0)$ and $Y(T)$ is one of a very deep network
  - Gradients from errors at $\mathrm{t} = T$ will vanish by the time they're propagated to $t = 0$

- Stuff gets forgotten in the forward pass too
  - Each weights matrix and activation can shrink components of the input

The long-term dependency problem

1

PATTERN1 [...............................] PATTERN 2

Jane had a quick lunch in the bistro. Then *she*..

- Any other pattern of any length can happen between pattern 1 and pattern 2
  - RNN will “forget” pattern 1 if intermediate stuff is too long
  - “Jane” -> the next pronoun referring to her will be “she”
- Must know to “remember” for extended periods of time and “recall” when necessary
  - Need a way to “remember” stuff

Exploding/Vanishing gradients

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

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

- 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*

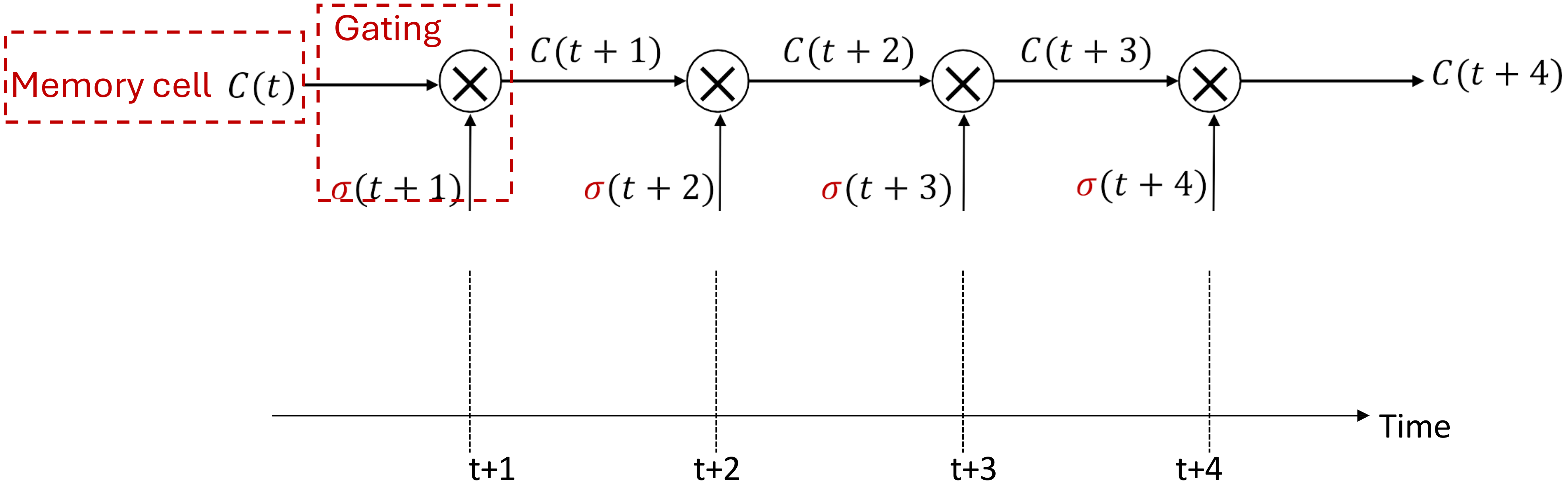
Exploding/Vanishing gradients

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

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

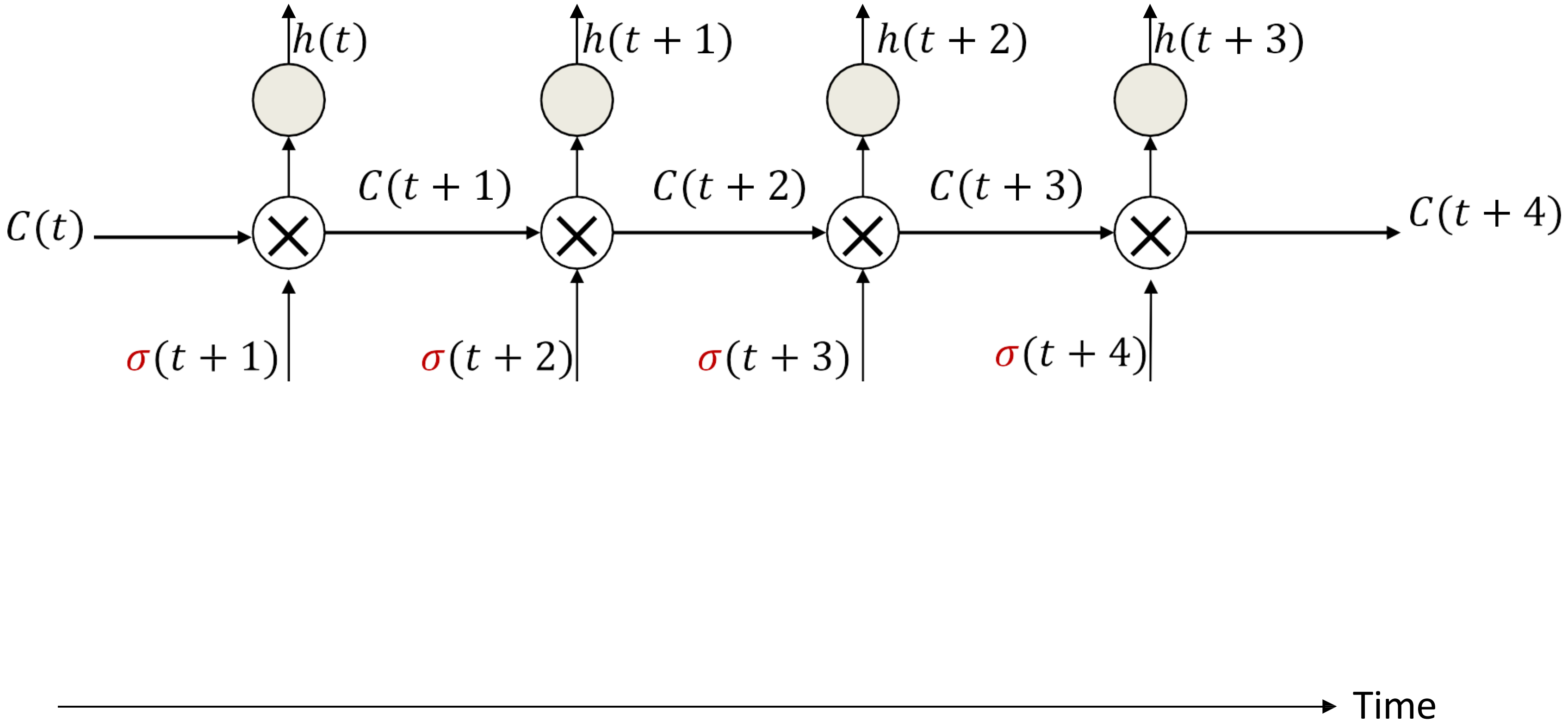
- Replace this with something that doesn't fade or blow up?
- Network that "retains" *useful* memory arbitrarily long, to be recalled on demand?
  - Input-based determination of *whether it must be remembered*
  - **Retain memories until a switch *based on the input* flags them as ok to forget**
   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

Constant Error Carousel



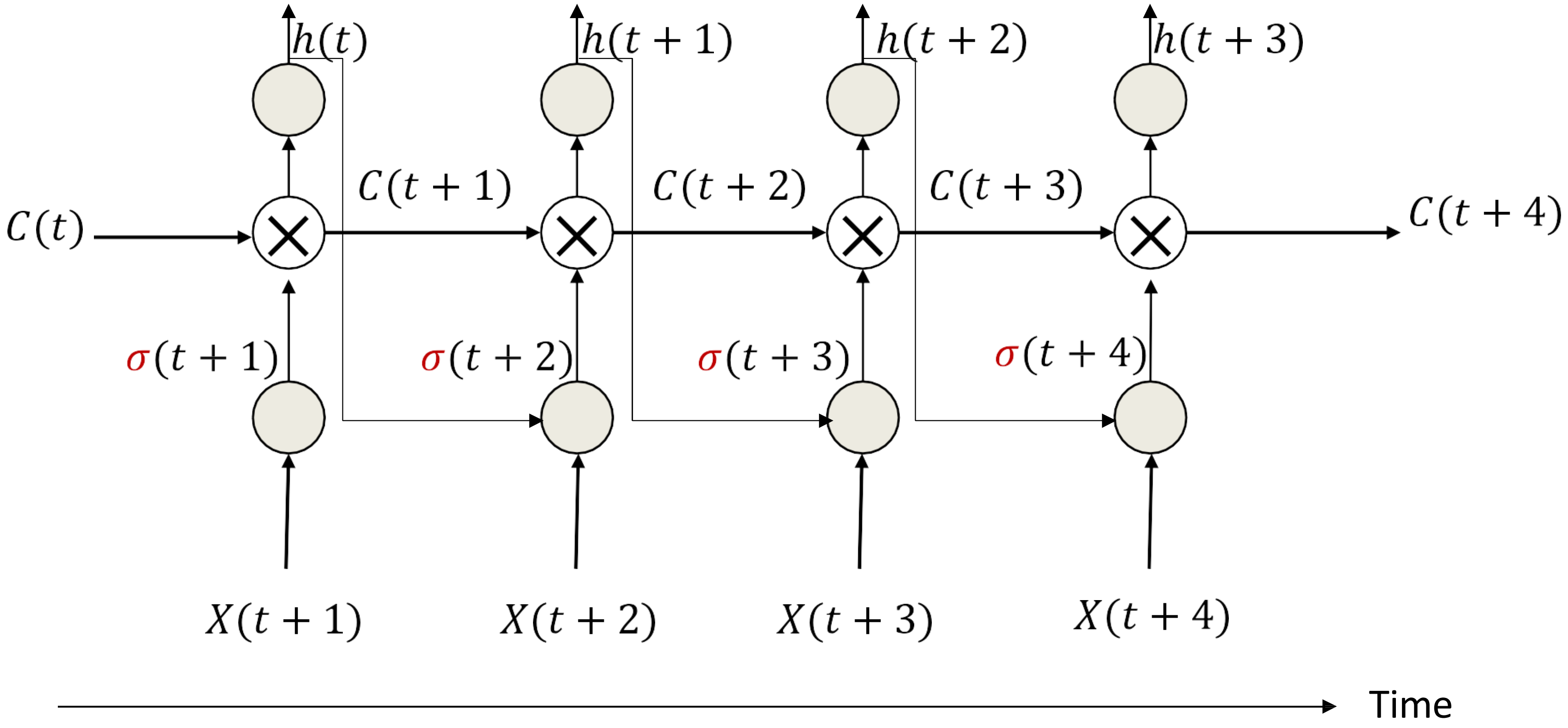
- History is carried through uncompressed
  - No weights, no nonlinearities – enable long-term information retention
  - Only scaling is through the s “gating” term that captures other triggers
  - E.g. “Have I seen Pattern2”?

Constant Error Carousel



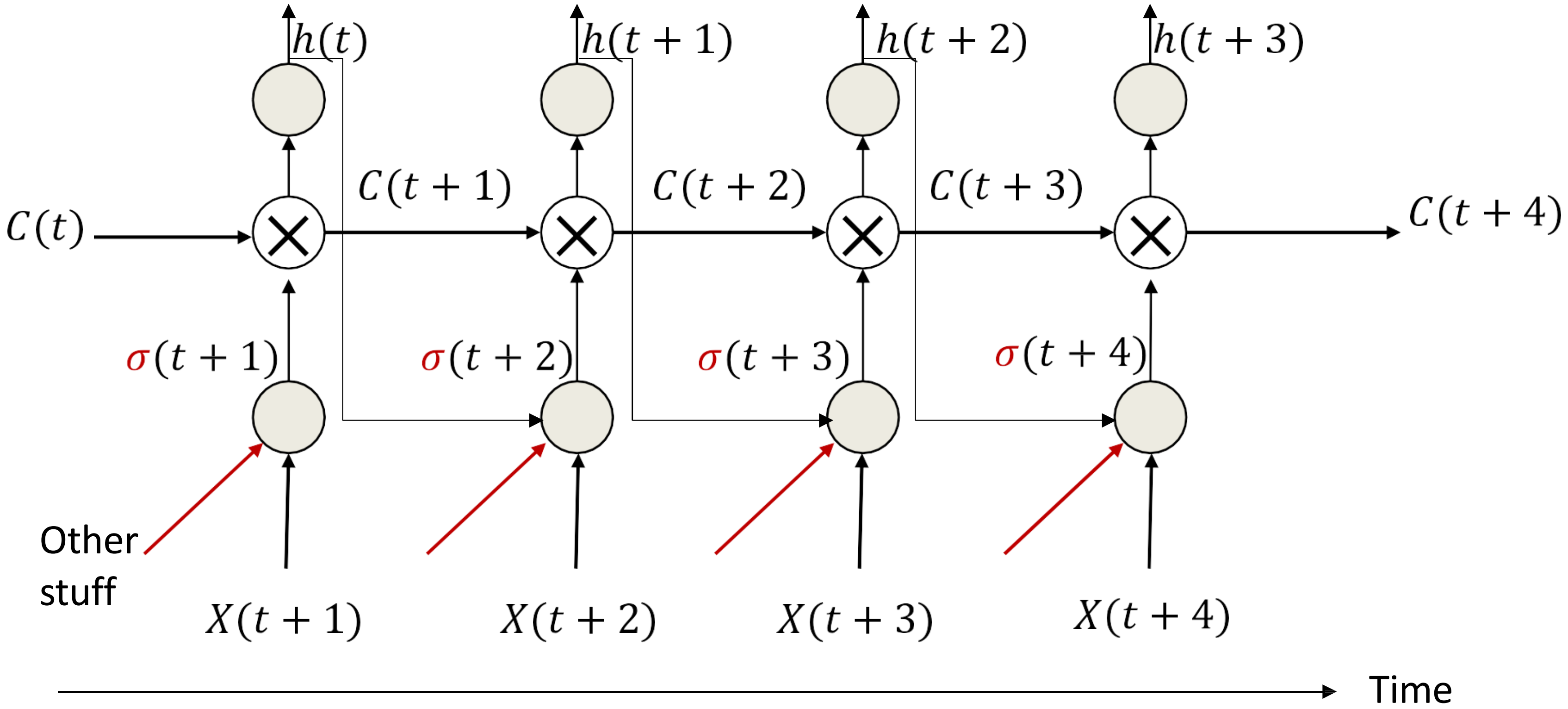
- Actual non-linear work is done by other portions of the network
  - Neurons that compute the workable state from the memory

Constant Error Carousel



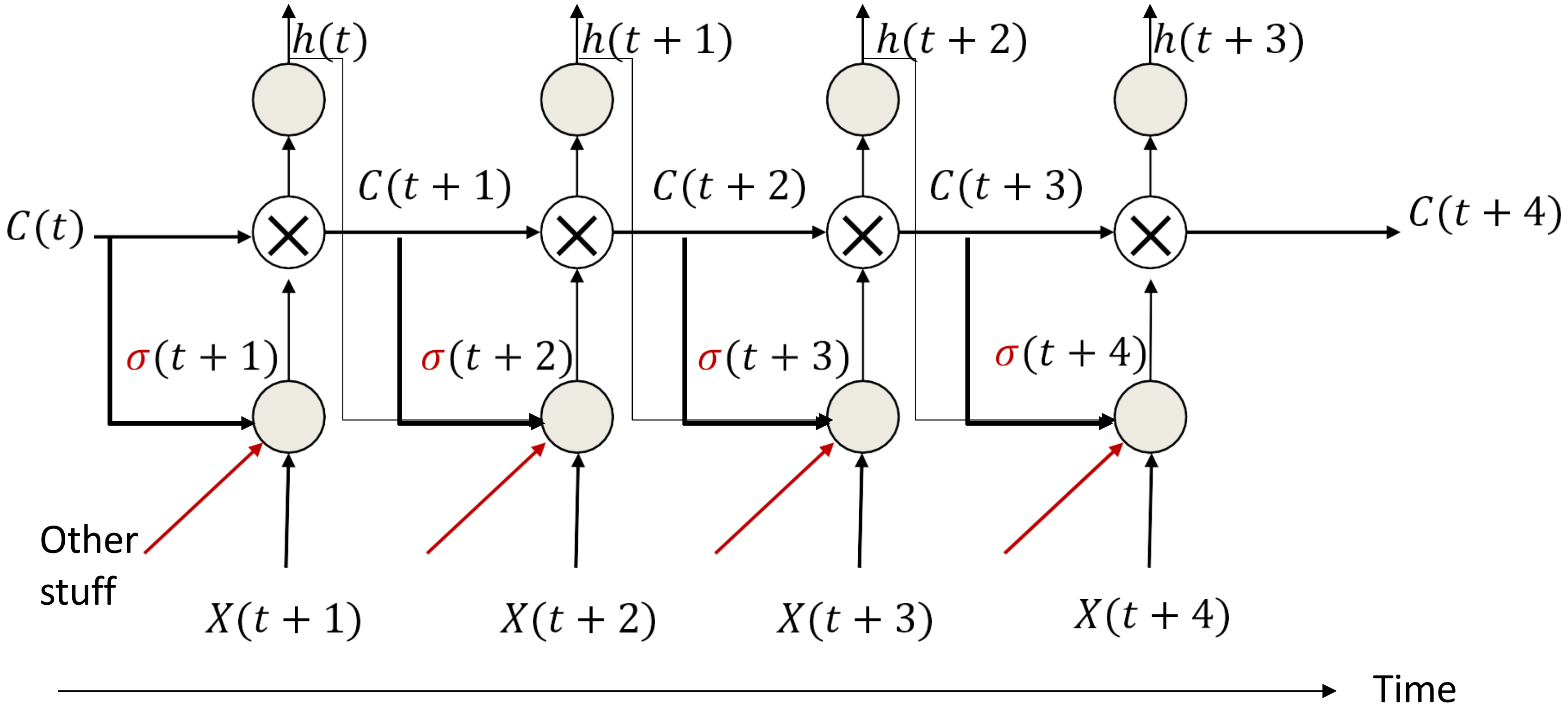
- The gate s depends on current input, current hidden state…

Constant Error Carousel



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

Constant Error Carousel



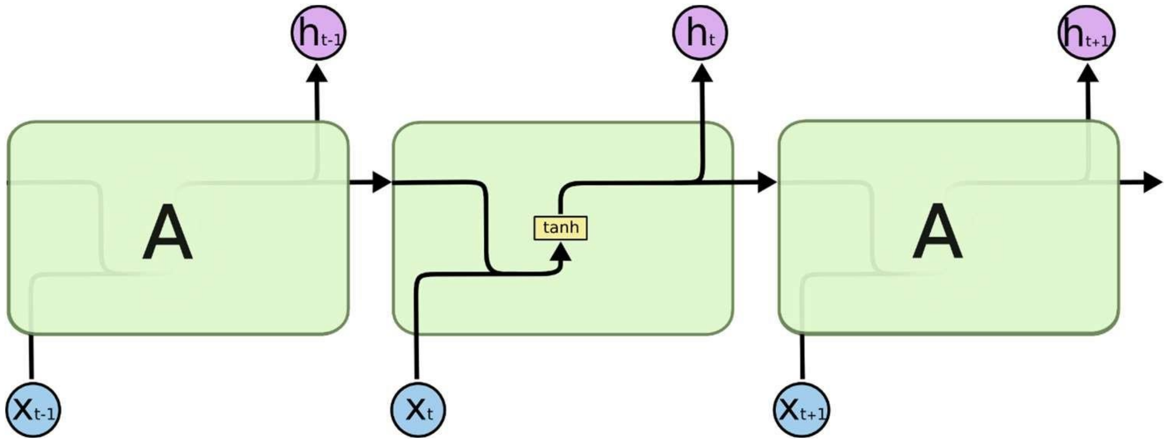
- The gate s depends on current input, current hidden state… and other stuff…
- Including, obviously, what is currently in raw memory

Enter the *LSTM*

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

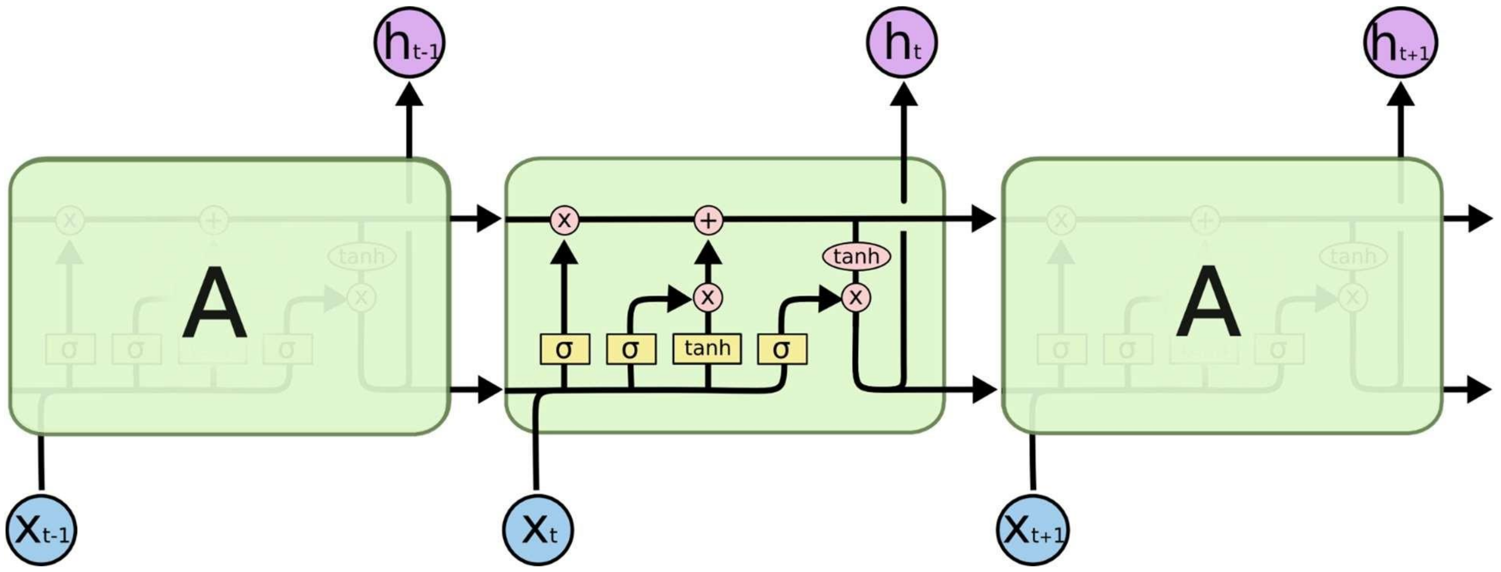
Following slides are based on http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Standard RNN



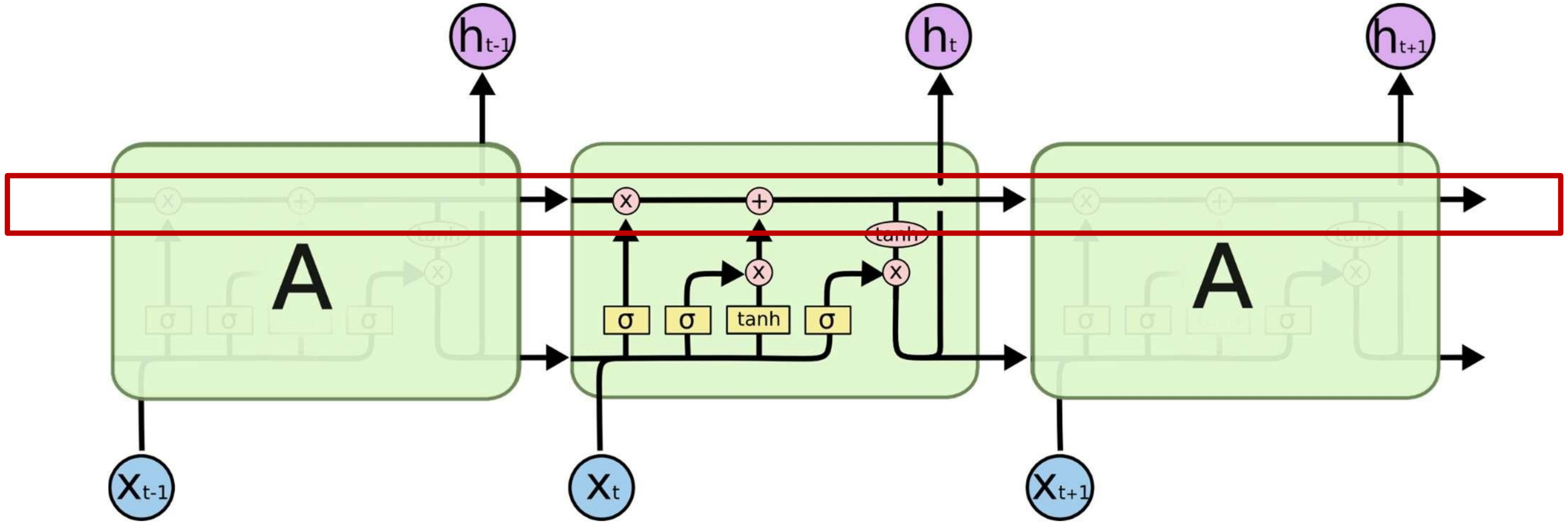
- Recurrent neurons receive past recurrent outputs and current input as inputs
- Processed through a tanh() activation function
  - As mentioned earlier, tanh() is the generally used activation for the hidden layer
- Current recurrent output passed to next higher layer and next time instant

Long Short-Term Memory



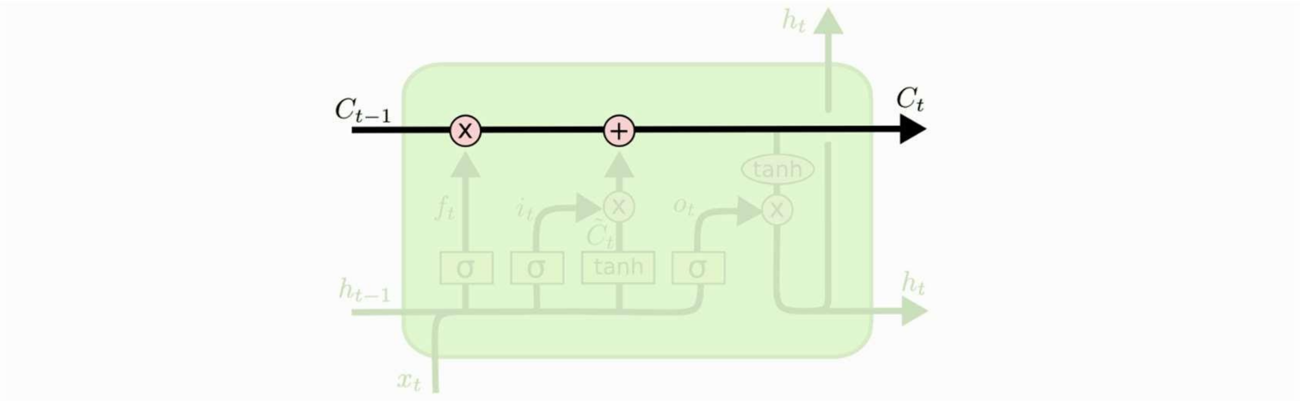
- The $\sigma()$ are *multiplicative gates* that decide if something is important or not
- Remember, every line actually represents a *vector*

LSTM: Constant Error Carousel



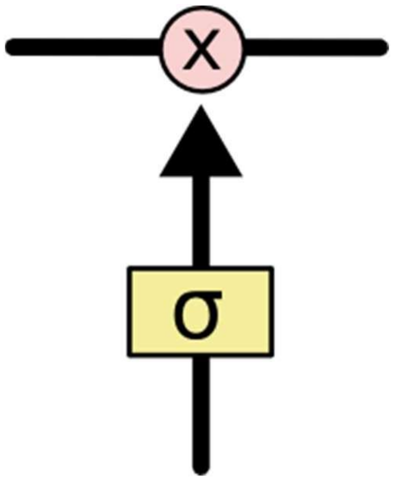
- Key component: a *remembered cell state*

LSTM: CEC



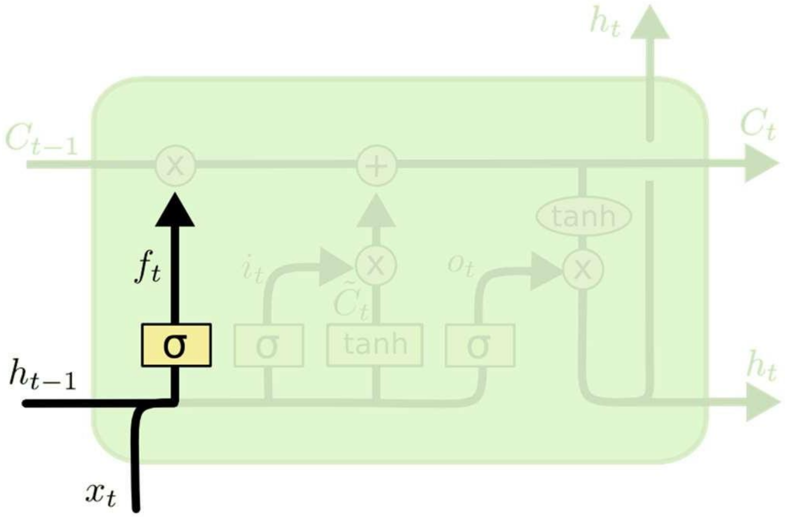
- C_t is the linear history carried by the *constant-error carousel*
- Carries information through, only affected by a gate
  – And *addition of history,* which too is gated..

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

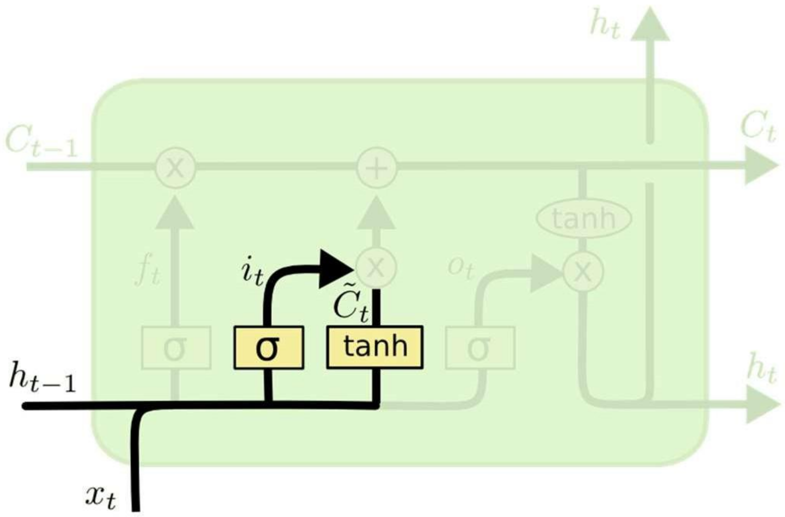
LSTM: Forget gate



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

- 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

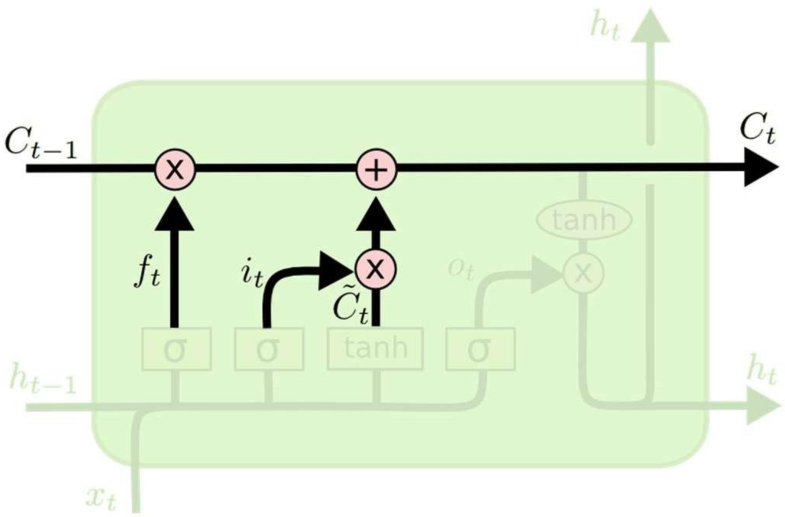


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

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

- 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

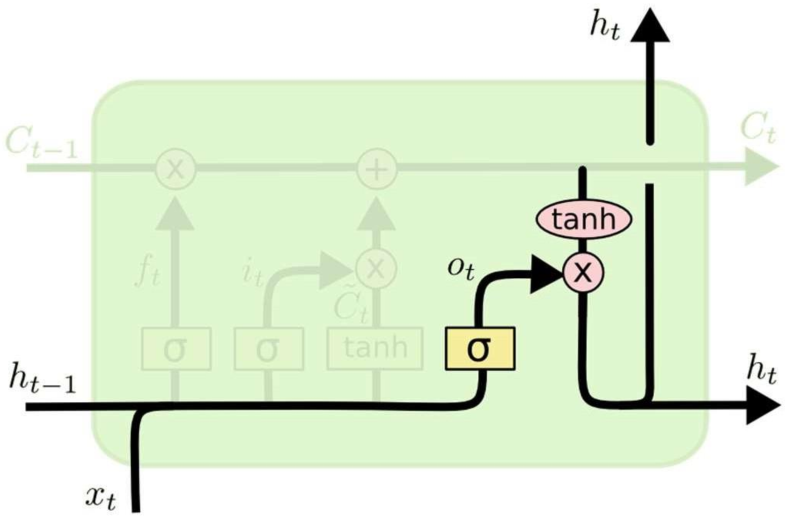
LSTM: Memory cell update



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

- 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

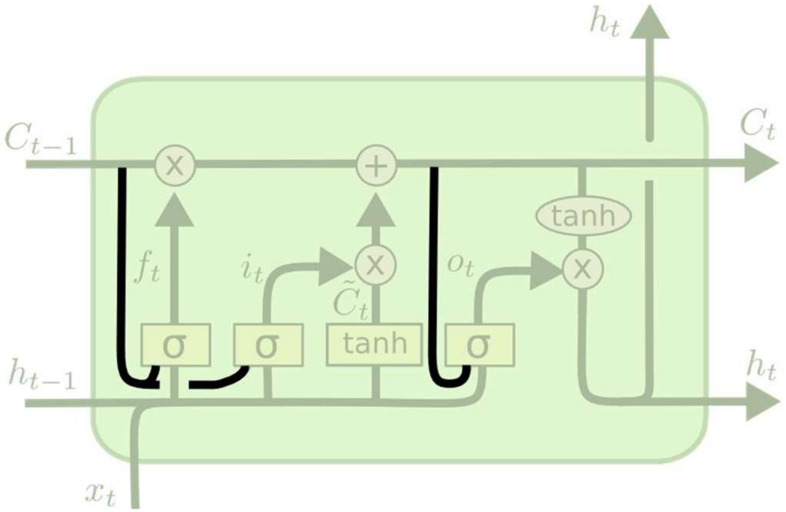


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

- 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

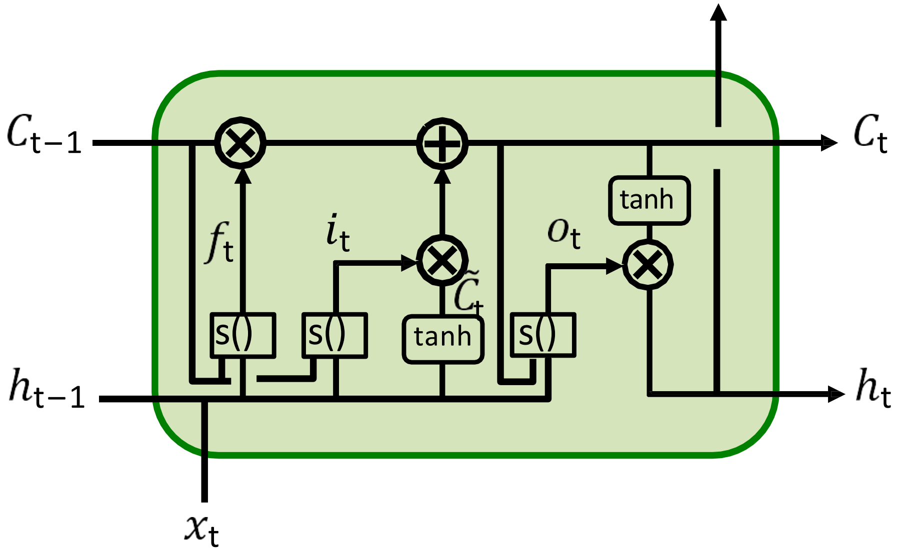
LSTM: The "Peephole" Connection



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

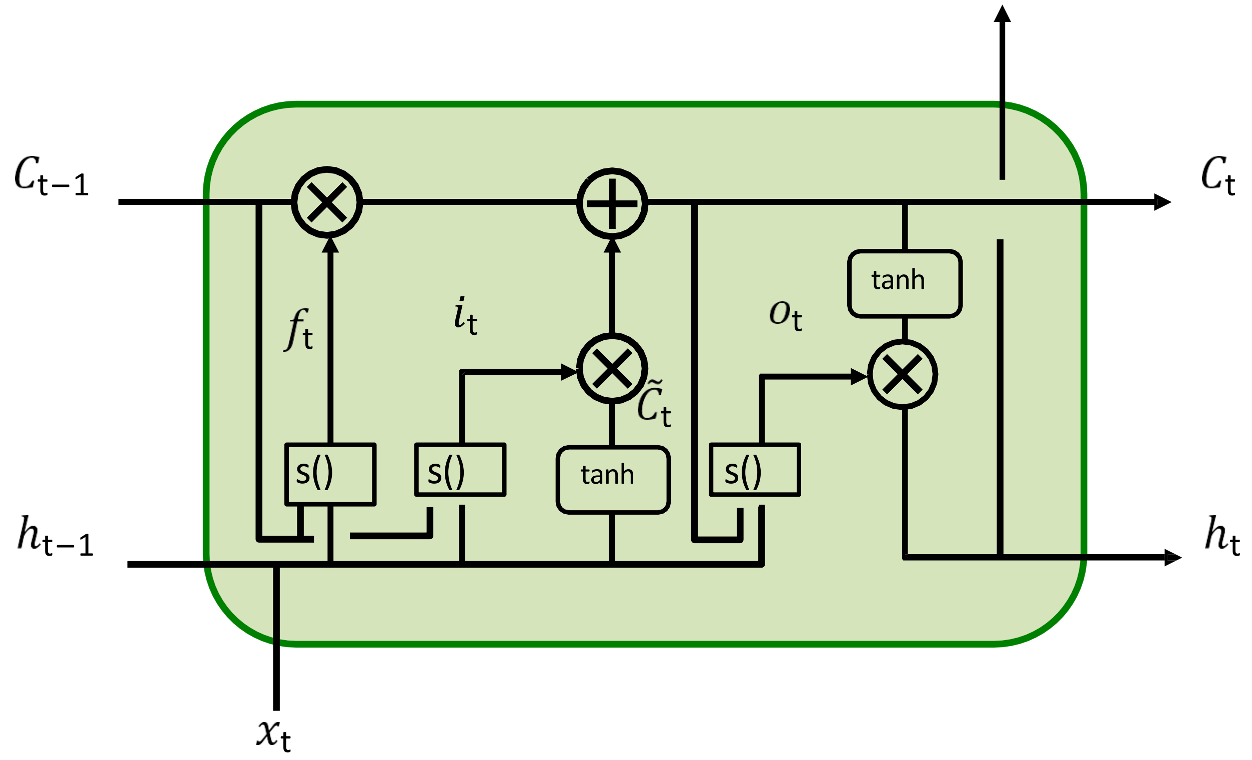
- The raw memory is informative by itself and can also be input
  - Note, we're using both C and h

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 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 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
```

Assuming a peephole connection into the tanh

LSTM network forward

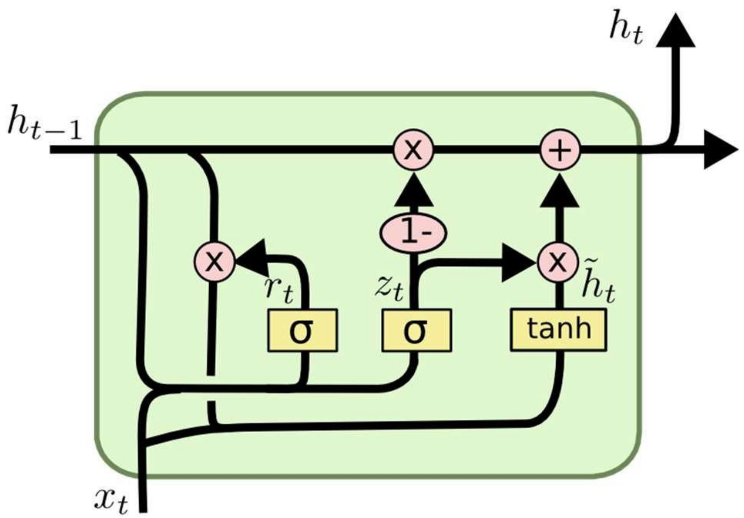
```
# Assuming h(-1,*) is known and C(-1,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
#                  for the lth hidden layer
# Wo and bo are output layer weights and biases

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

Gated Recurrent Units: Let's simplify the LSTM



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

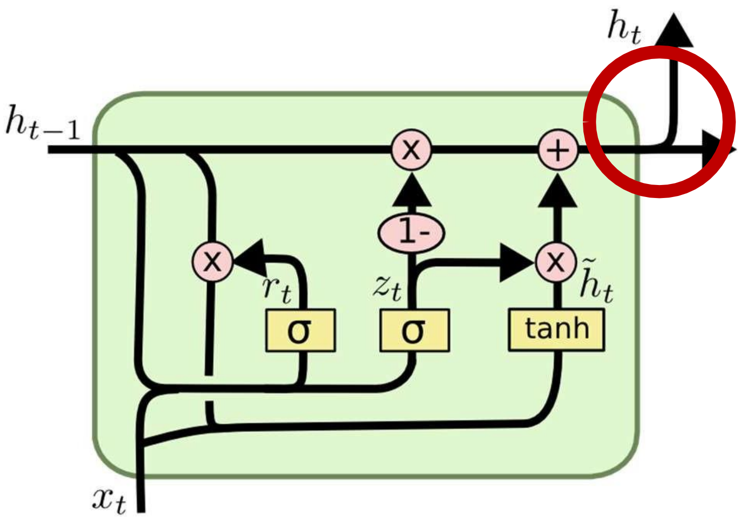
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

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

Gated Recurrent Units: Lets simplify the LSTM



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

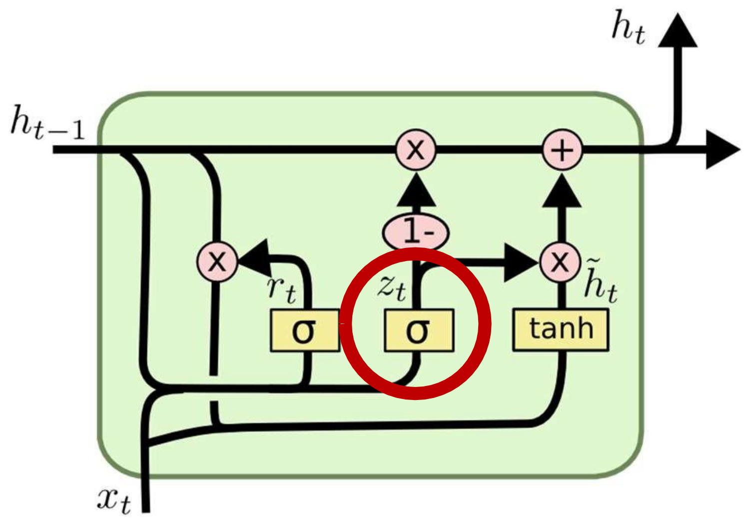
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

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

Gated Recurrent Units: Lets simplify the LSTM



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

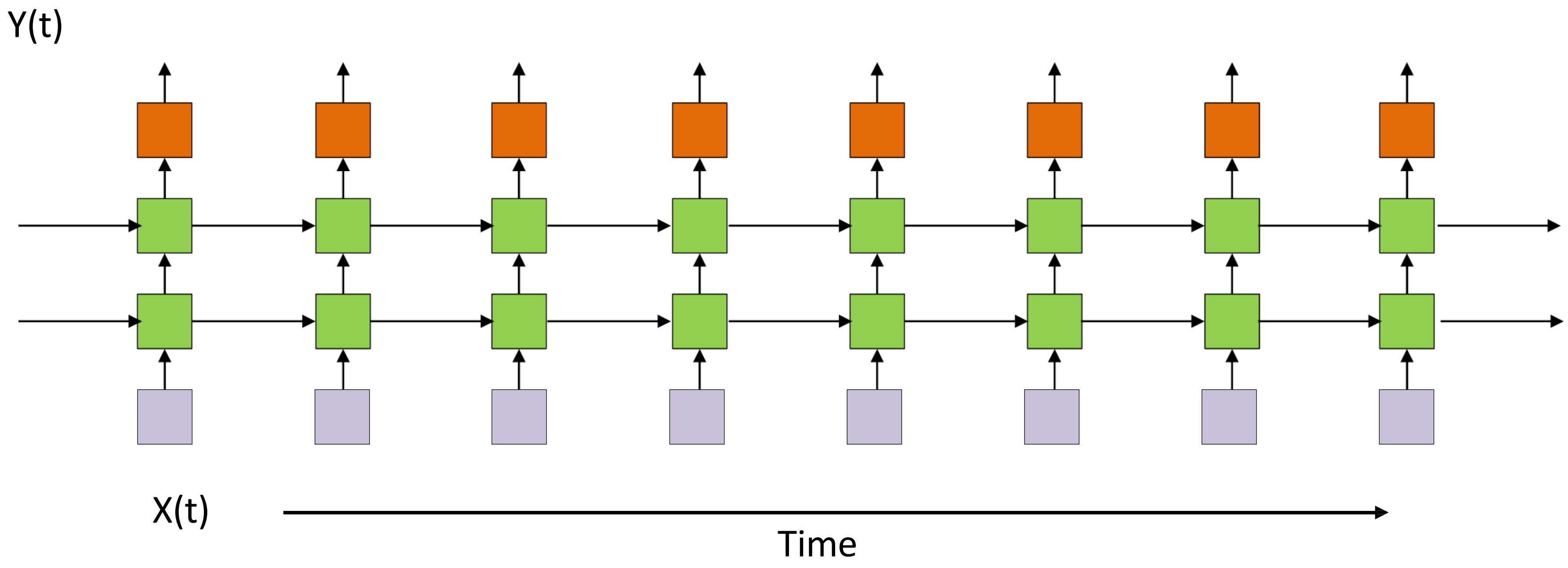
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

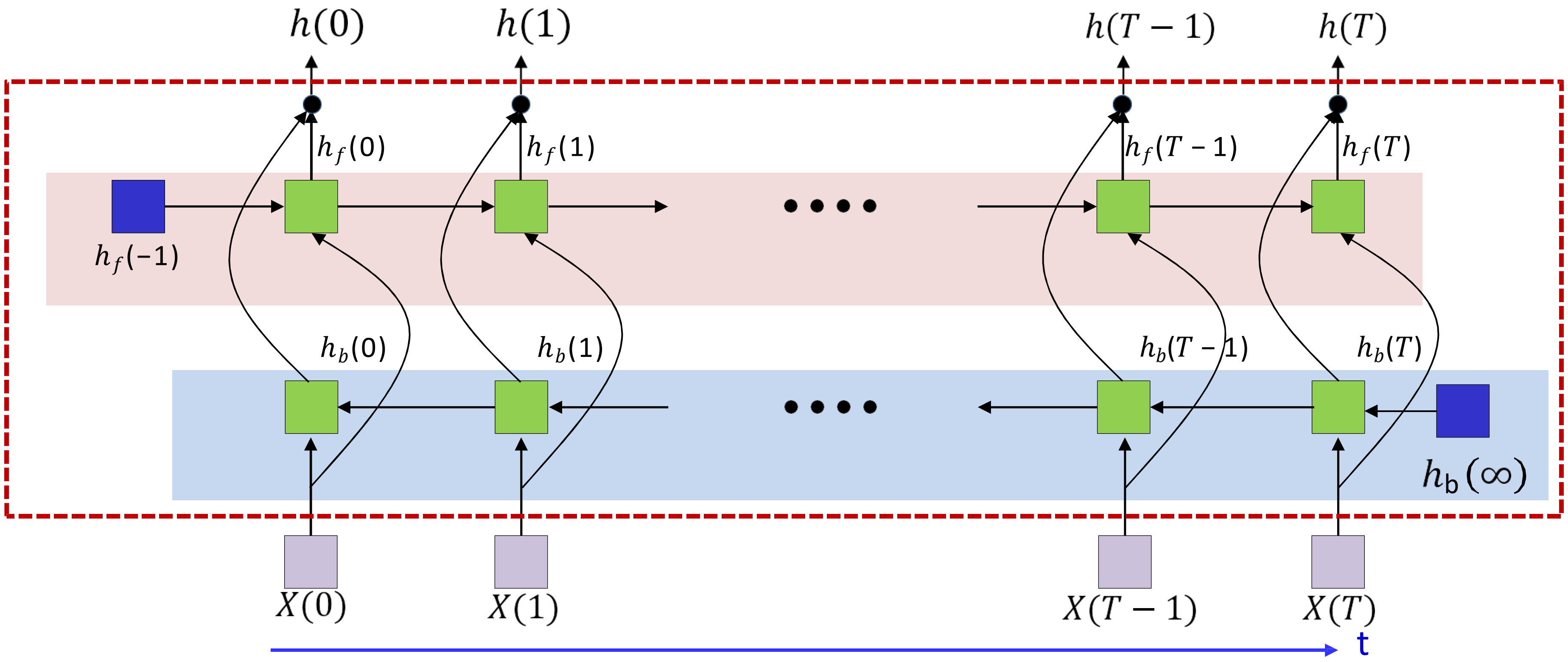
- Combine forget and input gates
  - If new input is to be remembered, then this means old memory is to be forgotten

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