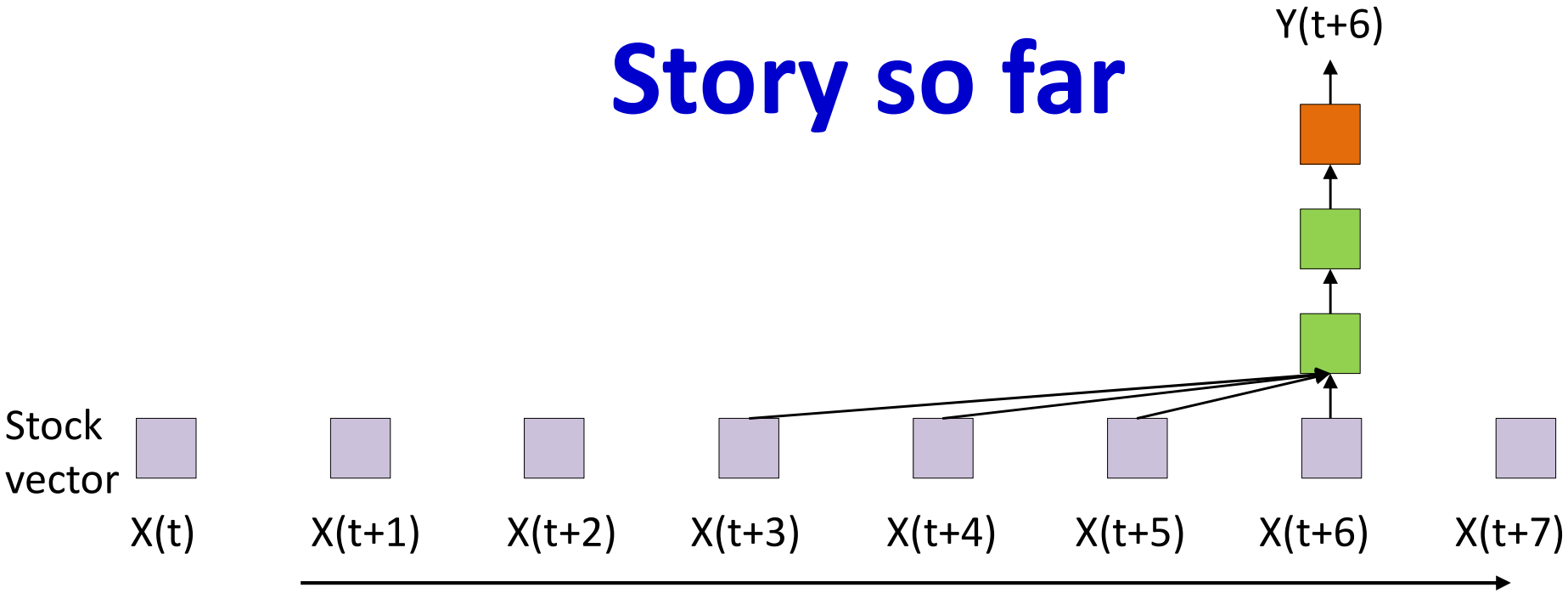
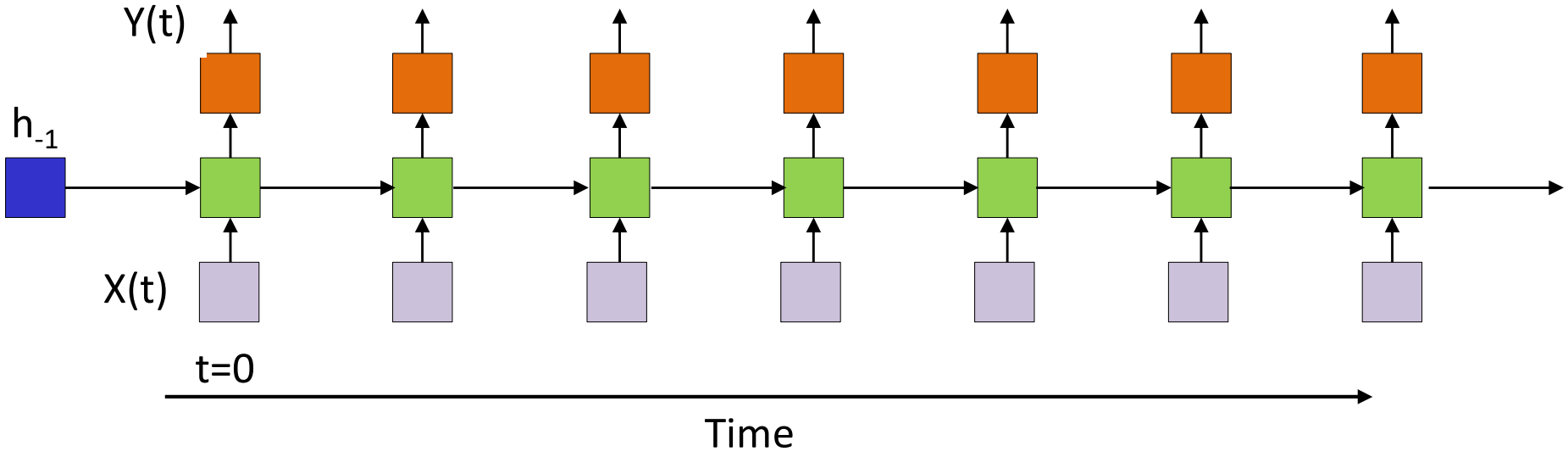
Deep Learning
Recurrent Networks: Stability analysis and LSTMs
Fall 2024

Story so far



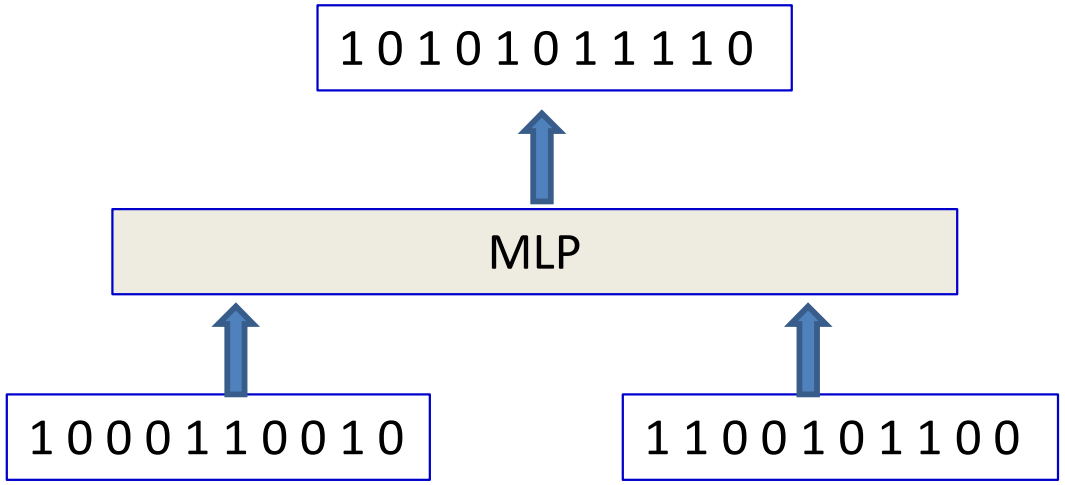
- ***Iterated structures*** are good for analyzing time series data with short-time dependence on the past
  - These are "***Time delay***" neural nets, AKA ***convnets***

Story so far



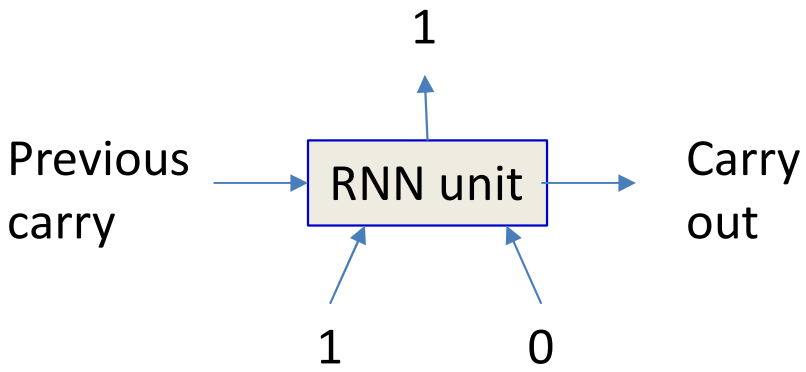
- Iterated structures are good for analyzing time series data with short-time dependence on the past
  - These are "Time delay" neural nets, AKA convnets
- ***Recurrent structures*** are good for analyzing time series data with ***long-term*** dependence on the past
  - These are ***recurrent*** neural networks

Recurrent structures can do what static structures cannot



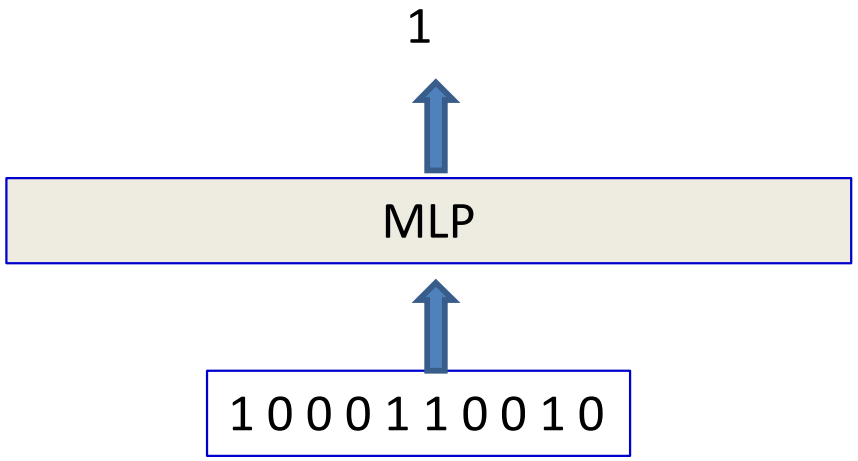
- The addition problem: Add two N-bit numbers to produce a N+1-bit number
  - Input is binary
  - Will require large number of training instances
    - Output must be specified for every pair of inputs
    - Weights that generalize will make errors
  - Network trained for N-bit numbers will not work for N+1 bit numbers

MLPs vs RNNs



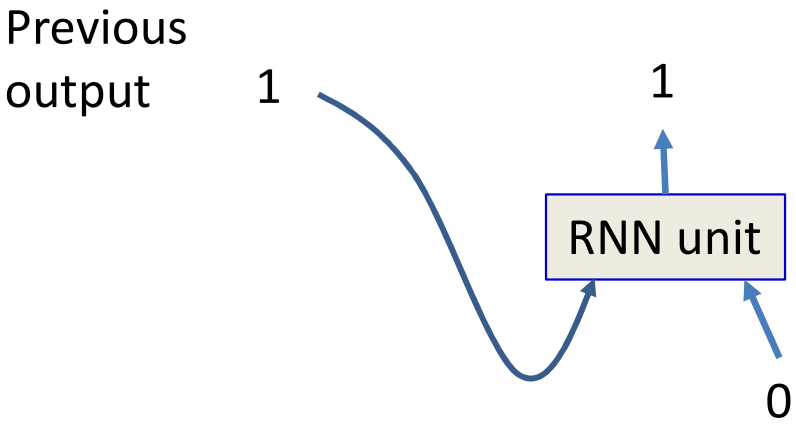
- The addition problem: Add two N-bit numbers to produce a N+1-bit number
- **RNN solution:** Very simple, can add two numbers of any size
- Needs very little training data

MLP: The parity problem



- Is the number of “ones” even or odd
- Network must be complex to capture all patterns
  - XOR network, quite complex
  - Fixed input size
- Needs a large amount of training data

RNN: The parity problem



- Trivial solution
  - Requires little training data
- Generalizes to input of any size

Poll 1

Some prediction and classification problems that require very large MLPs and a large amount of training data can be solved using small recurrent nets that only require small amounts of training data

- True
- False

Some problems that require large, complicated convolutional neural nets and large amounts of training data could also be solved using much smaller RNNs that only require small amounts of training data

- True
- False

Poll 1

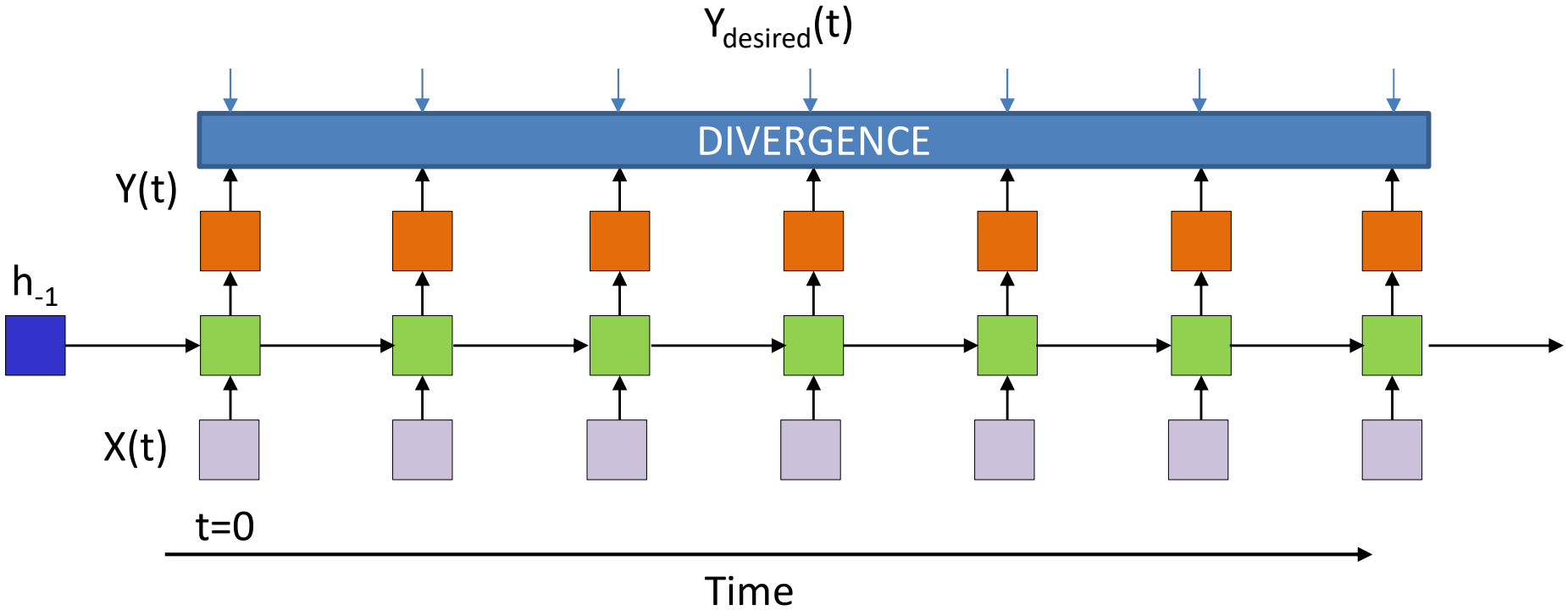
Some prediction and classification problems that require very large MLPs and a large amount of training data can be solved using small recurrent nets that only require small amounts of training data

- **True**
- False

Some problems that require large, complicated convolutional neural nets and large amounts of training data could also be solved using much smaller RNNs that only require small amounts of training data

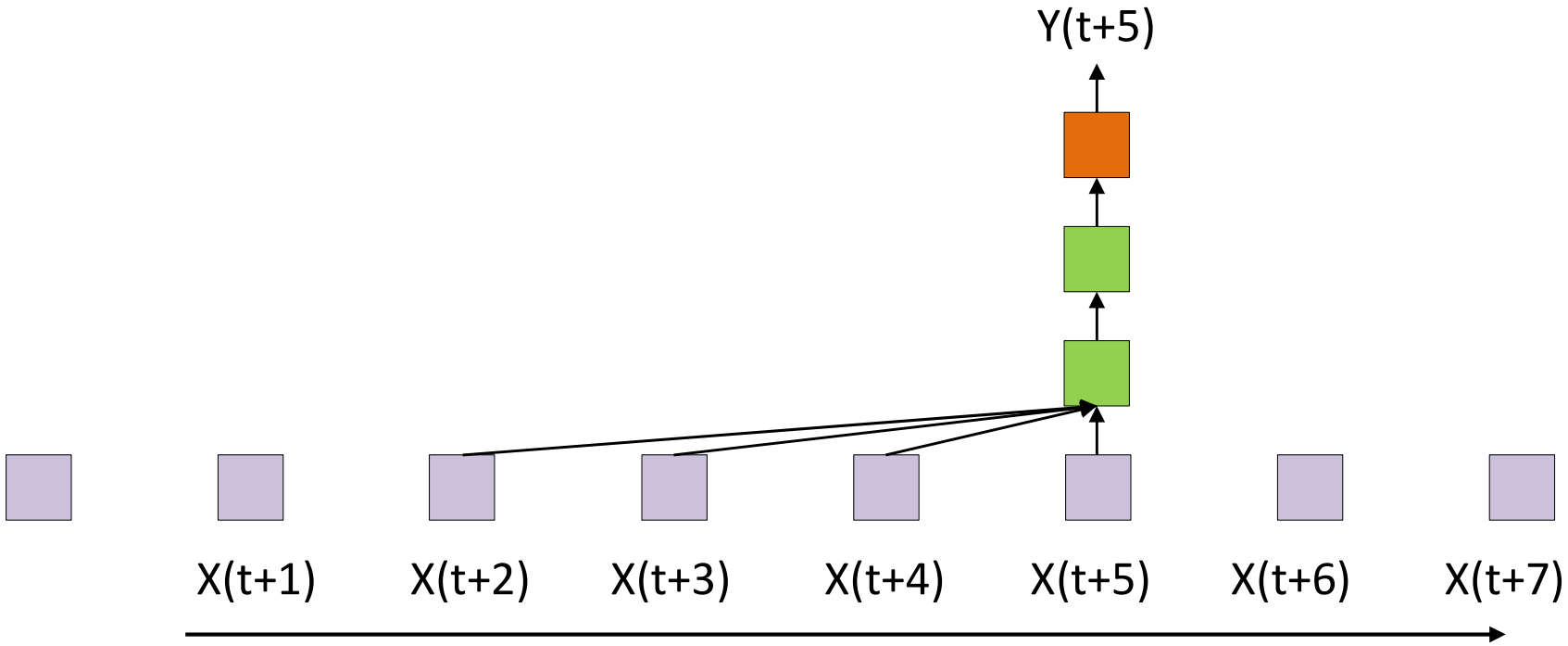
- **True**
- False

Story so far



- Recurrent structures can be trained by minimizing the divergence between the ***sequence*** of outputs and the ***sequence*** of desired outputs
  - Through gradient descent and backpropagation

The behavior of recurrence..

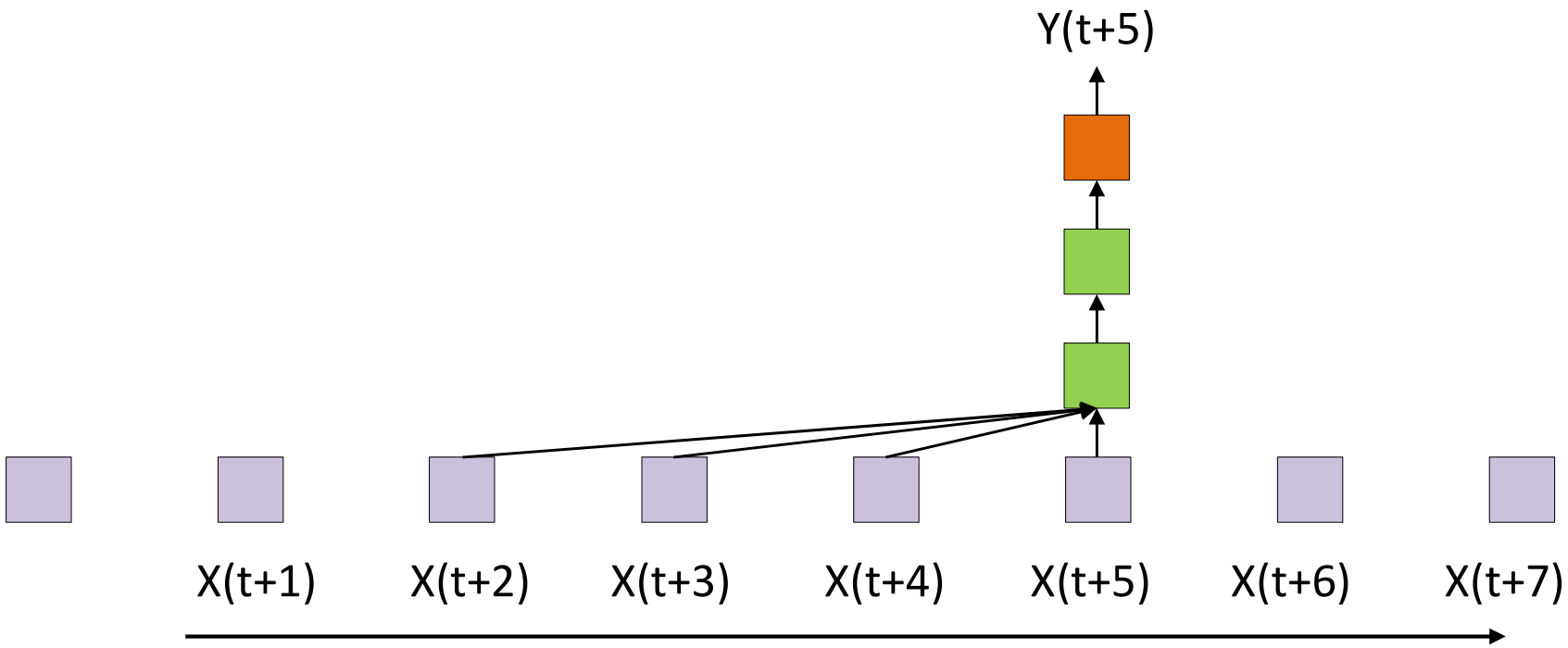


- Returning to an old model..

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

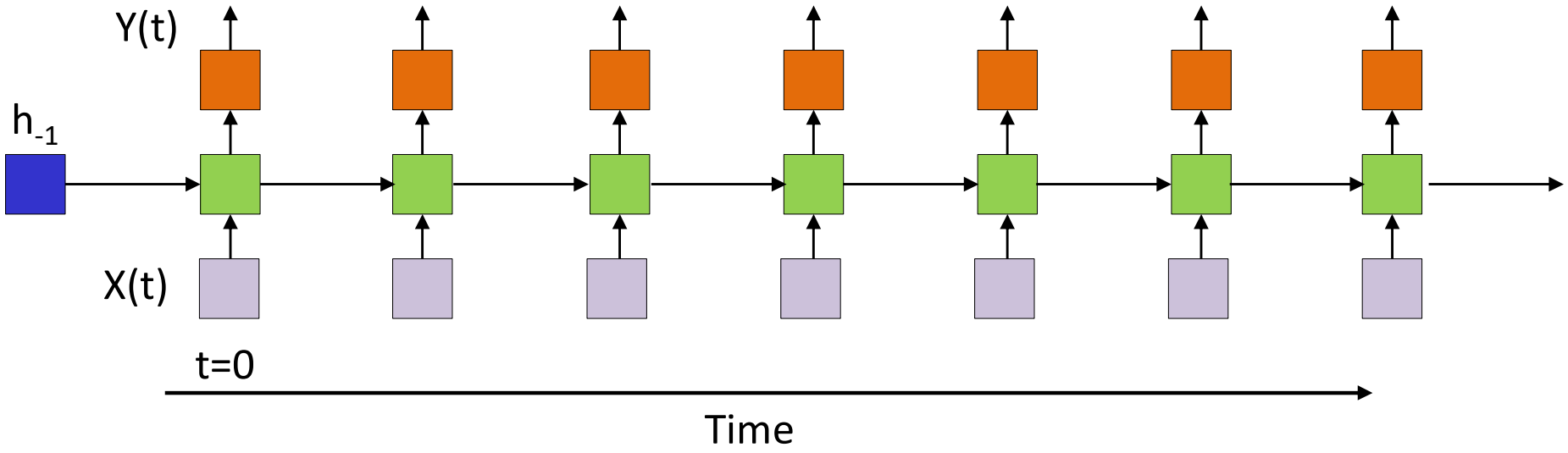
- When will the output “blow up”?

"BIBO" Stability



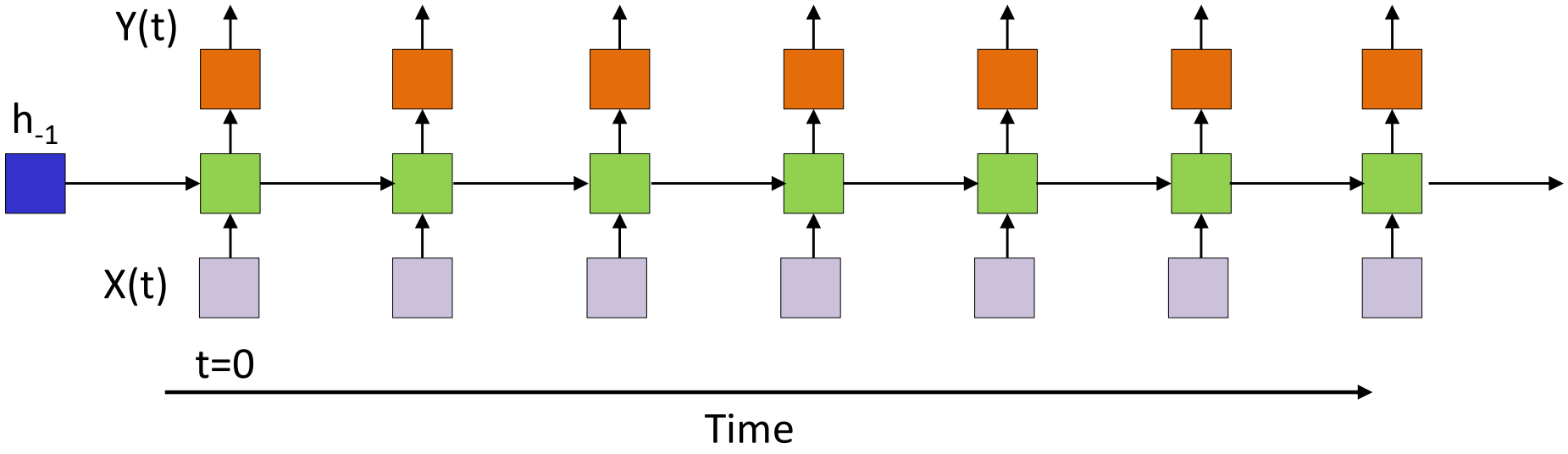
- Time-delay structures have bounded output if
  - The function $f()$ has bounded output for bounded input
    - Which is true of almost every activation function
  - $X(t)$ is bounded
- "Bounded Input Bounded Output" stability
  - This is a highly desirable characteristic

Is this BIBO?



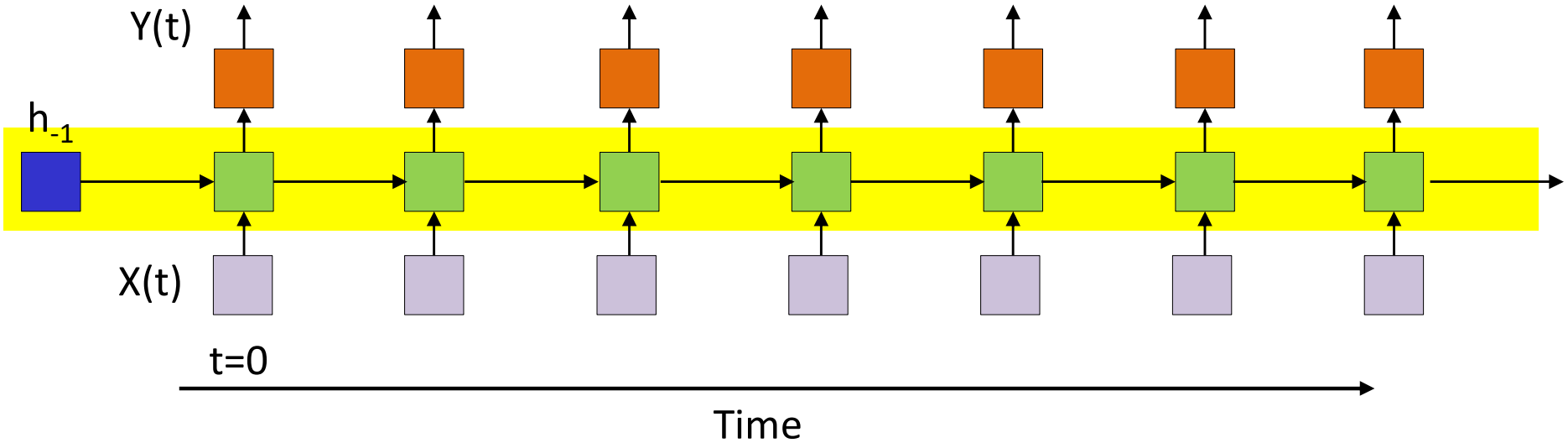
- Will this necessarily be BIBO?

Is this BIBO?



- Will this necessarily be BIBO?
  - Guaranteed if output and hidden activations are bounded
    - But will it *saturate* (and where)
  - What if the activations are linear?

Analyzing recurrence



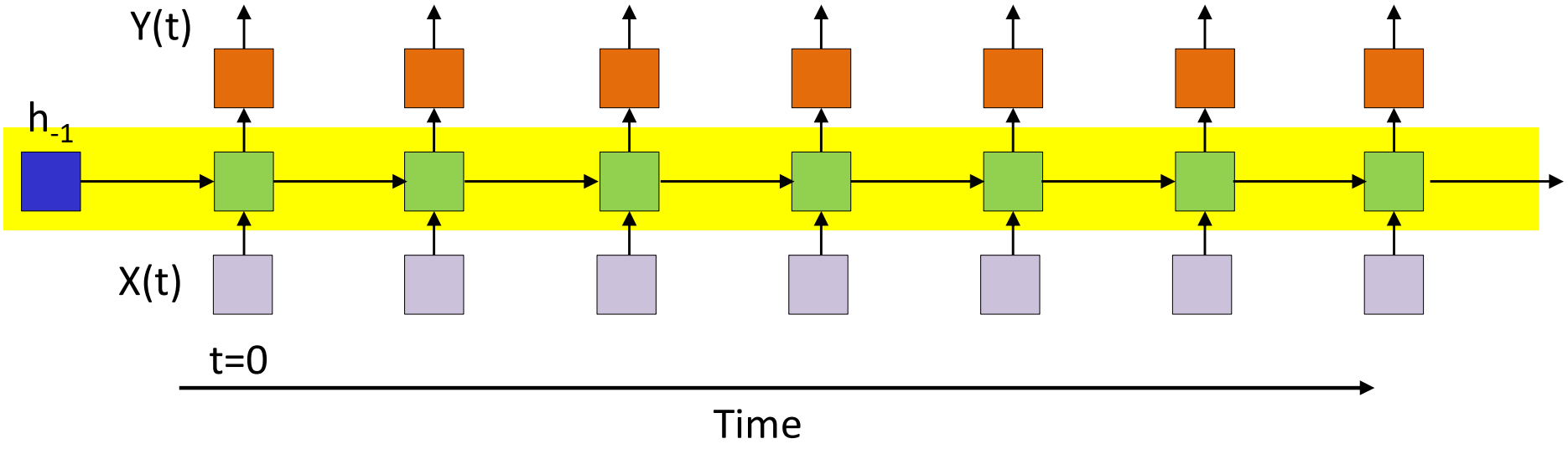
- Sufficient to analyze the behavior of the hidden layer h_t since it carries the relevant information
  - Will assume only a single hidden layer for simplicity

Analyzing Recursion

The streetlight effect is a type of observational bias where people only look for whatever they are searching by looking where it is easiest

"I'm searching for my keys."

Streetlight effect



- Easier to analyze *linear* systems
  - Will attempt to extrapolate to non-linear systems subsequently
- All activations are identity functions
  - $z_t = W_h h_{t-1} + W_x x_t, \qquad h_t = z_t$

Linear systems

- $h_k = W_h h_{k-1} + W_x x_k$
  - $h_{k-1} = W_h h_{k-2} + W_x x_{k-1}$

Using index "k" for time

Linear systems

- $h_k = W_h h_{k-1} + W_x x_k$
  - $h_{k-1} = W_h h_{k-2} + W_x x_{k-1}$
- $h_k = W_h^2 h_{k-2} + W_h W_x x_{k-1} + W_x x_k$

Using index "k" for time

Linear systems

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

Using index "k" for time

Linear systems

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

Using index "k" for time

Response at k to an input x0 at time 0, when there are no other inputs and zero initial condition

Linear systems

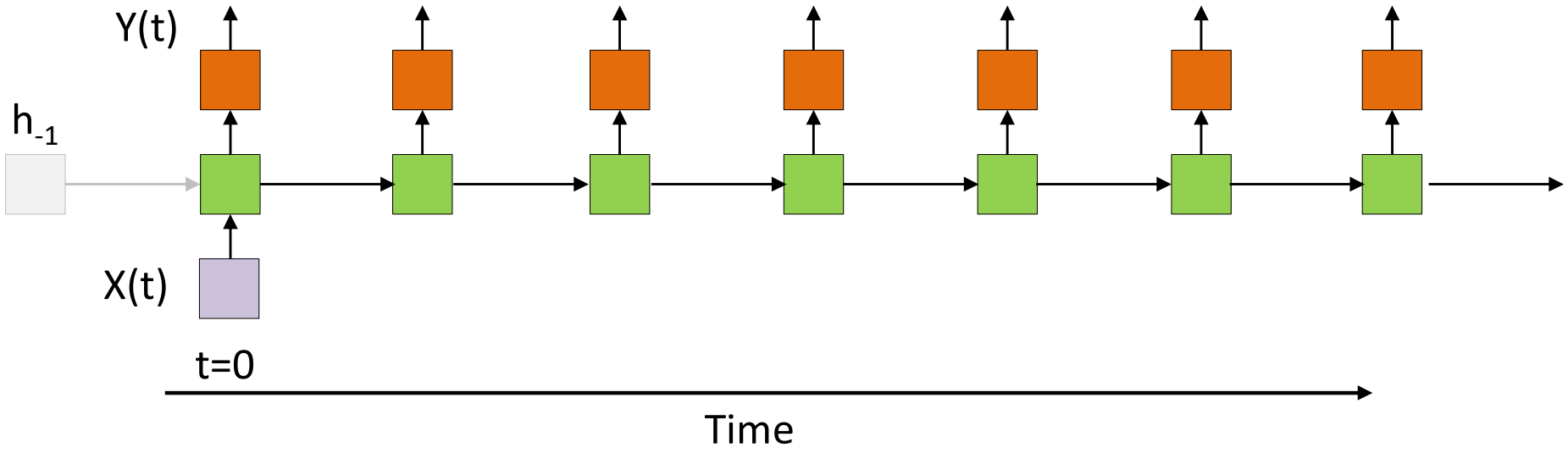
- $h_k = W_h h_{k-1} + W_x x_k$
  - $h_{k-1} = W_h h_{k-2} + W_x x_{k-1}$
- $h_k = W_h^2 h_{k-2} + W_h W_x x_{k-1} + W_x x_k$
- $h_k = W_h^{k+1} h_{-1} + W_h^k W_x x_0 + W_h^{k-1} W_x x_1 + W_h^{k-2} W_x x_2 + \cdots$
- $h_k = H_k(h_{-1}) + H_k(x_0) + H_k(x_1) + H_k(x_2) + \cdots$

Using index "k" for time

If this blows up, the entire system blows up

If this does not blow up, then responses to later inputs will also not blow up!

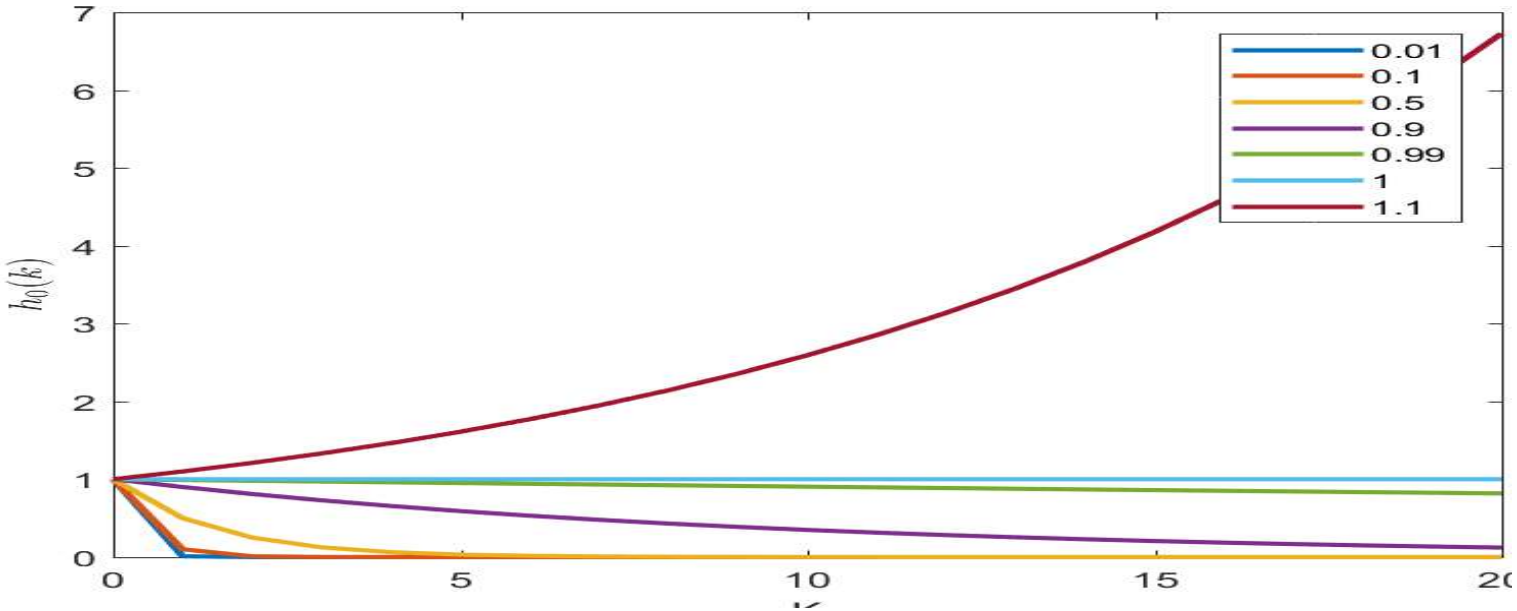
Streetlight effect



- Sufficient to analyze the response to a single input at $t = 0$
  - Principle of superposition in linear systems

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 $W = U\Lambda U^{-1} = U \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \ddots \end{bmatrix} U^{-1}$

$$W^t = U\Lambda^t U^{-1} = U \begin{bmatrix} \lambda_1^t & 0 & 0 \\ 0 & \lambda_2^t & 0 \\ 0 & 0 & \ddots \end{bmatrix} U^{-1} = \lambda_1^k U \begin{bmatrix} 1 & 0 & 0 \\ 0 & \left(\frac{\lambda_2}{\lambda_1}\right)^t & 0 \\ 0 & 0 & \ddots \end{bmatrix} U^{-1}$$

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 $|h_0(t)|$

For any input, for large t the length of the hidden vector will expand or contract according to the t –th power of the largest eigen value λ_1 of the *recurrent* weight matrix

$$W^t = U\Lambda^t U^{-1} = U\begin{bmatrix}\lambda_1^t & 0 & 0\\ 0 & \lambda_2^t & 0\\ 0 & 0 & \ddots\end{bmatrix}U^{-1} = \lambda_1^k U\begin{bmatrix}1 & 0 & 0\\ 0 & \left(\frac{\lambda_2}{\lambda_1}\right)^t & 0\\ 0 & 0 & \ddots\end{bmatrix}U^{-1}$$

Linear recursions: Vector version

- Vector linear recursion (note change of notation)

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

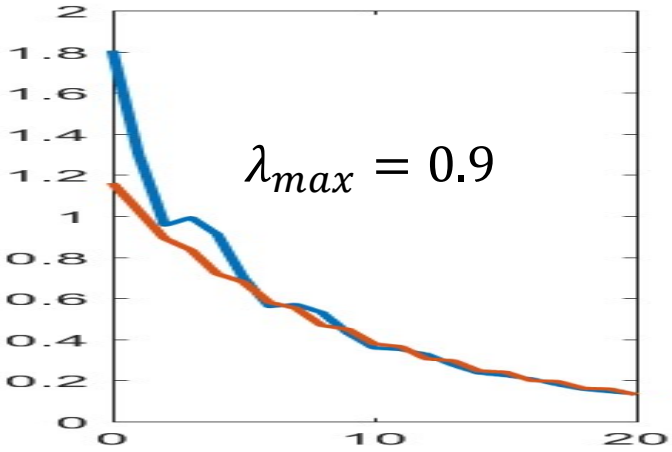
If $|\lambda_1| > 1$ it will blow up, otherwise it will contract and shrink to 0 rapidly

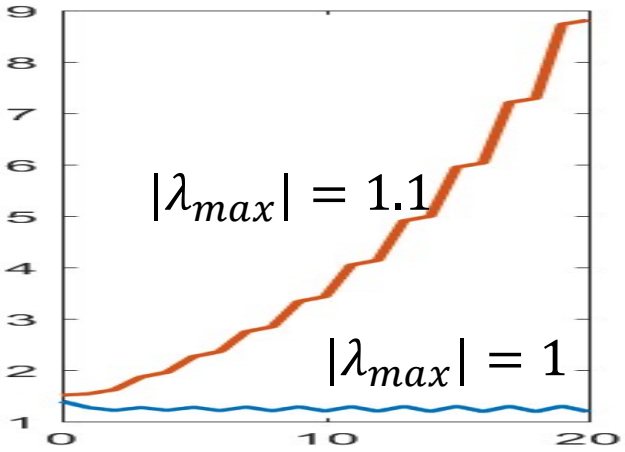
For any input, for large t the length of the hidden vector will expand or contract according to the t –th power of the largest eigen value of the *recurrent* weight matrix

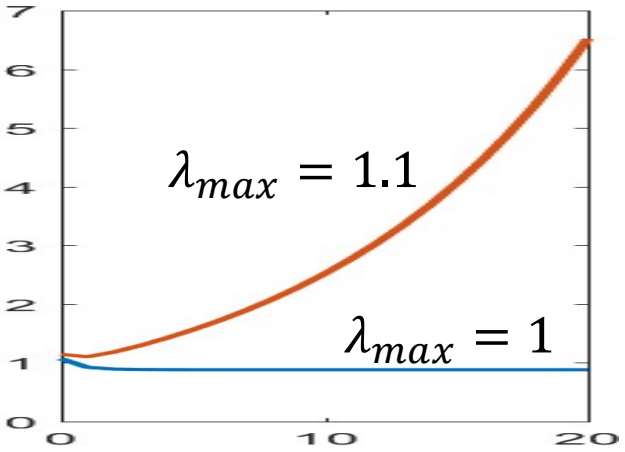
$$W^t = U\Lambda^t U^{-1} = U\begin{bmatrix} \lambda_1^t & 0 & 0 \\ 0 & \lambda_2^t & 0 \\ 0 & 0 & \ddots \end{bmatrix} U^{-1} = \lambda_1^k U \begin{bmatrix} 1 & 0 & 0 \\ 0 & \left(\frac{\lambda_2}{\lambda_1}\right)^t & 0 \\ 0 & 0 & \ddots \end{bmatrix} U^{-1}$$

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

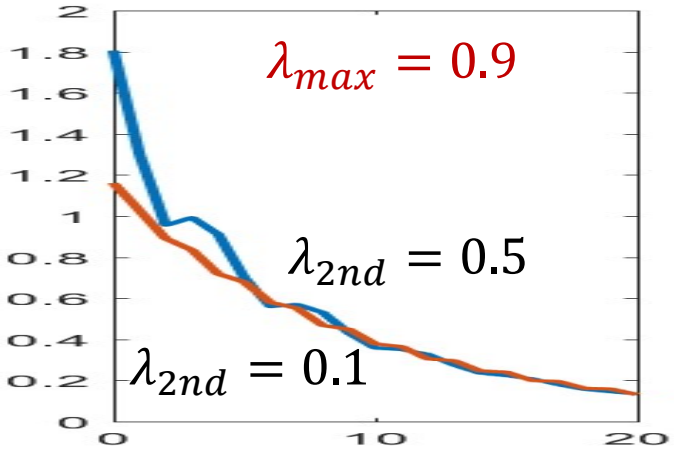


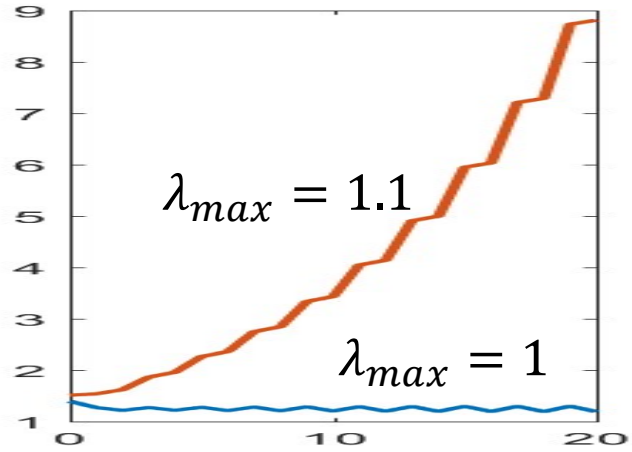


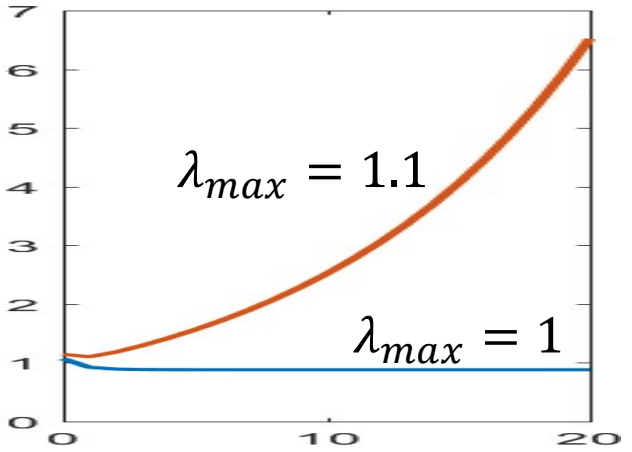


Linear recursions

- 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…

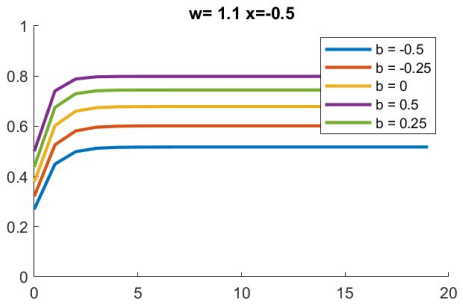
- In linear systems, long-term behavior depends entirely on the eigenvalues of the recurrent weights matrix
  - If the largest Eigen value is greater than 1, the system will "blow up"
  - If it is lesser than 1, the response will "vanish" very quickly
  - Complex Eigen values cause oscillatory response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

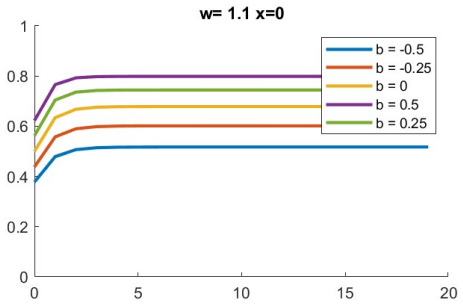
- *The rate of blow up or vanishing depends only on the Eigen values and not on the input*

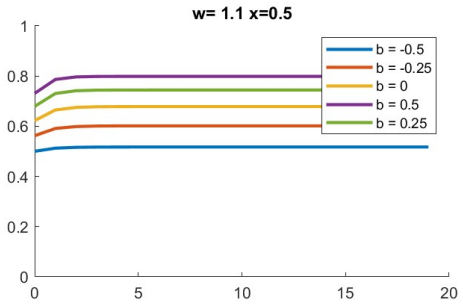
With non-linear activations: Sigmoid

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

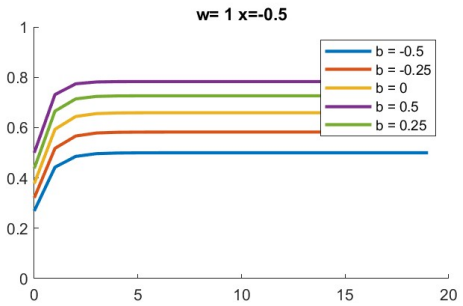
$w = 1.1$

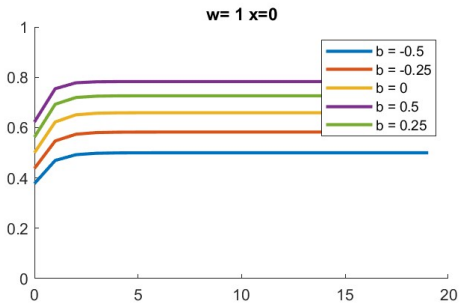


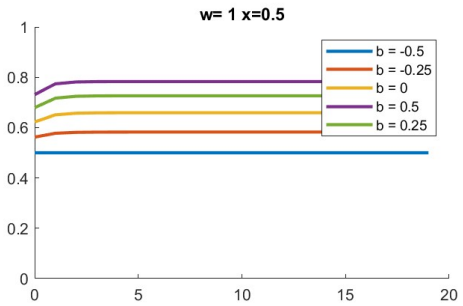




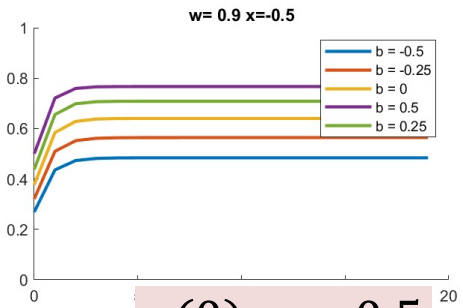
$w = 1.0$

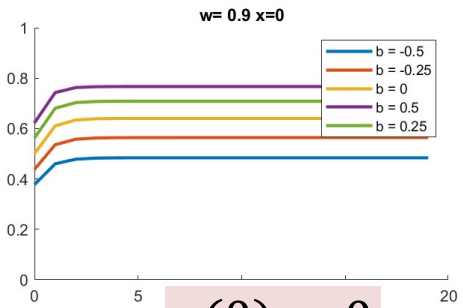


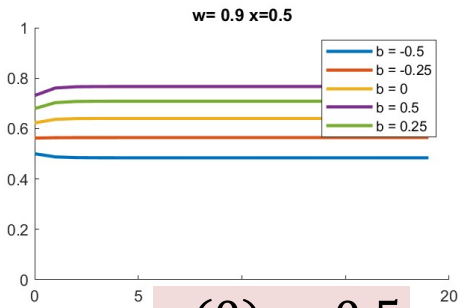




$w = 0.9$







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

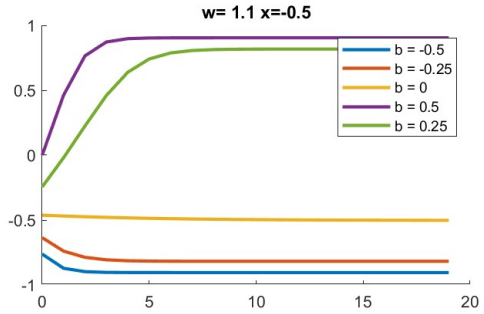
- Scalar recurrence with sigmoid activation
- Final value depends only on b and w but not x

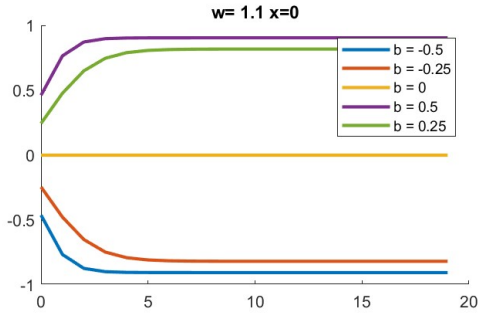
Scalar recurrence

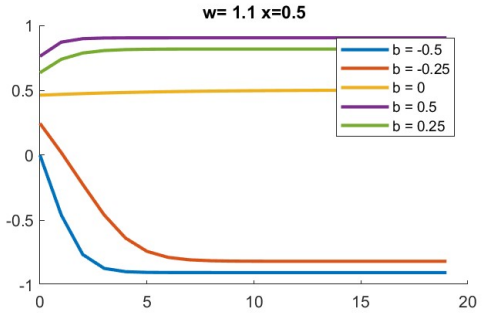
With non-linear activations: Tanh

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

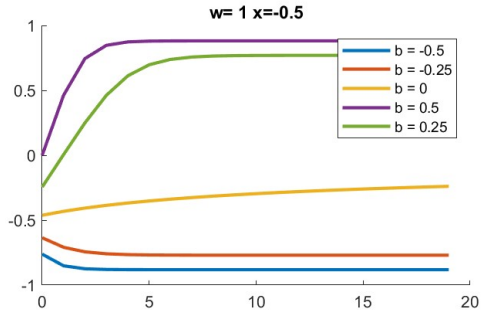
$w = 1.1$

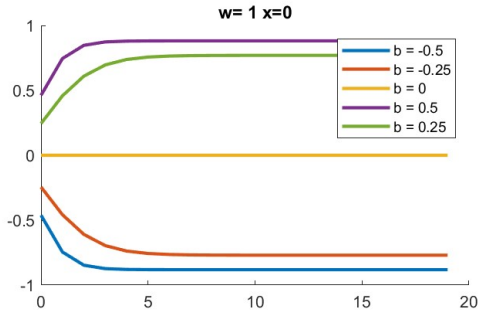


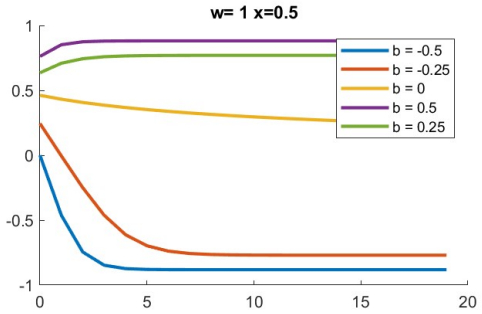




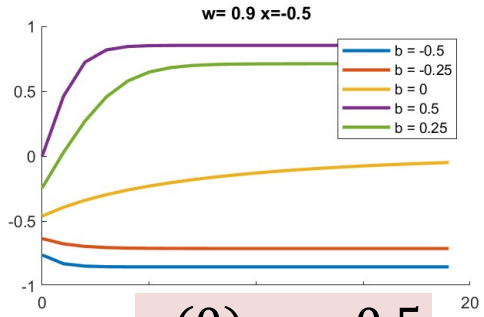
$w = 1.0$

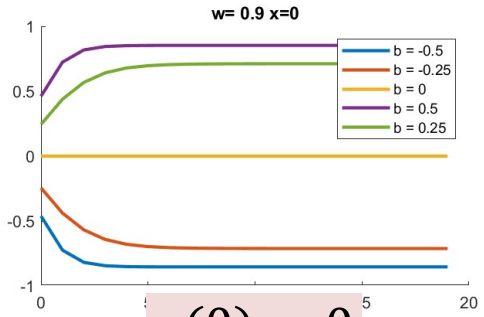


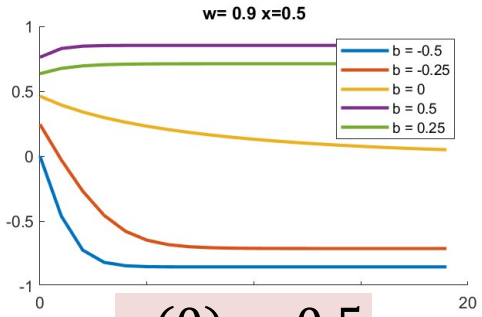




$w = 0.9$







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

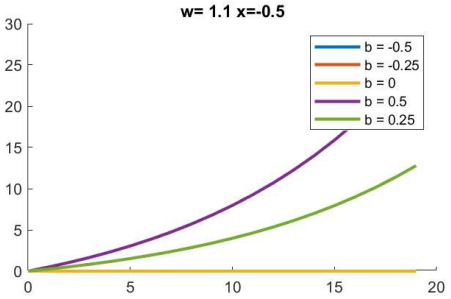
- Final value depends only on b and w, but not on x
- "Remembers" x value much longer than sigmoid

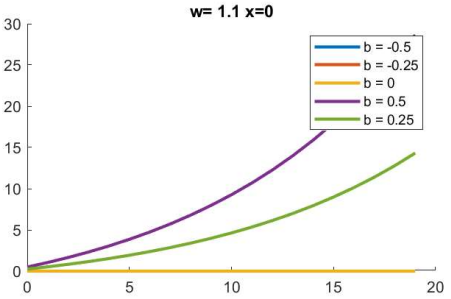
Scalar recurrence

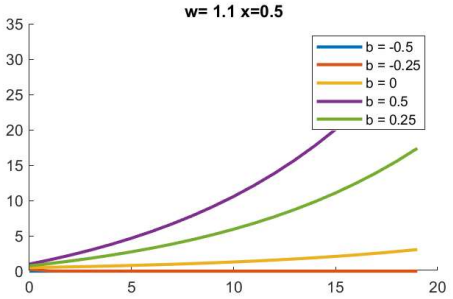
With non-linear activations: RELU

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

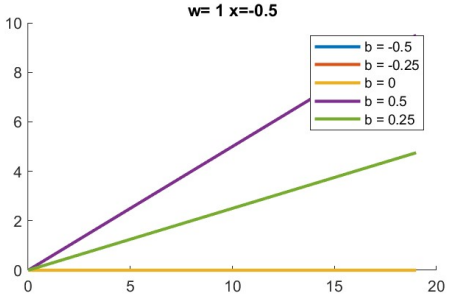
$w = 1.1$

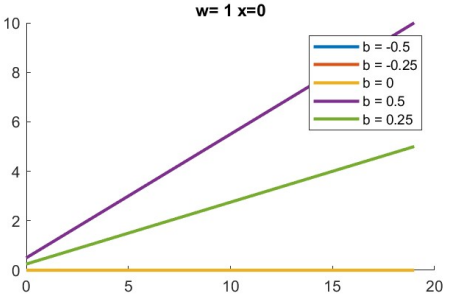


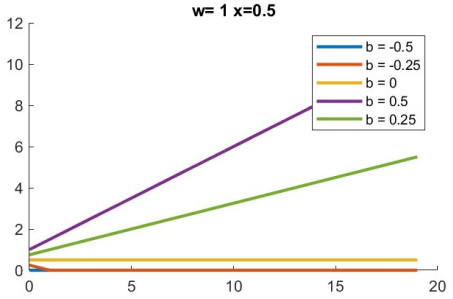




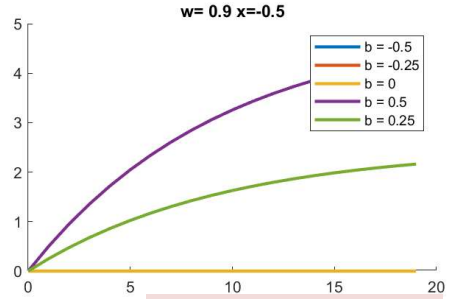
$w = 1.0$

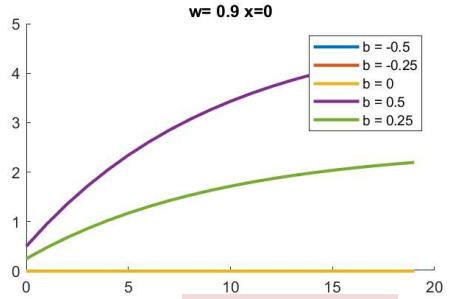


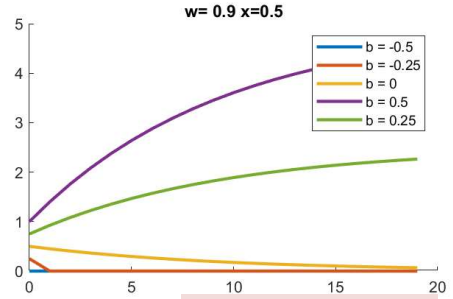




$w = 0.9$







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

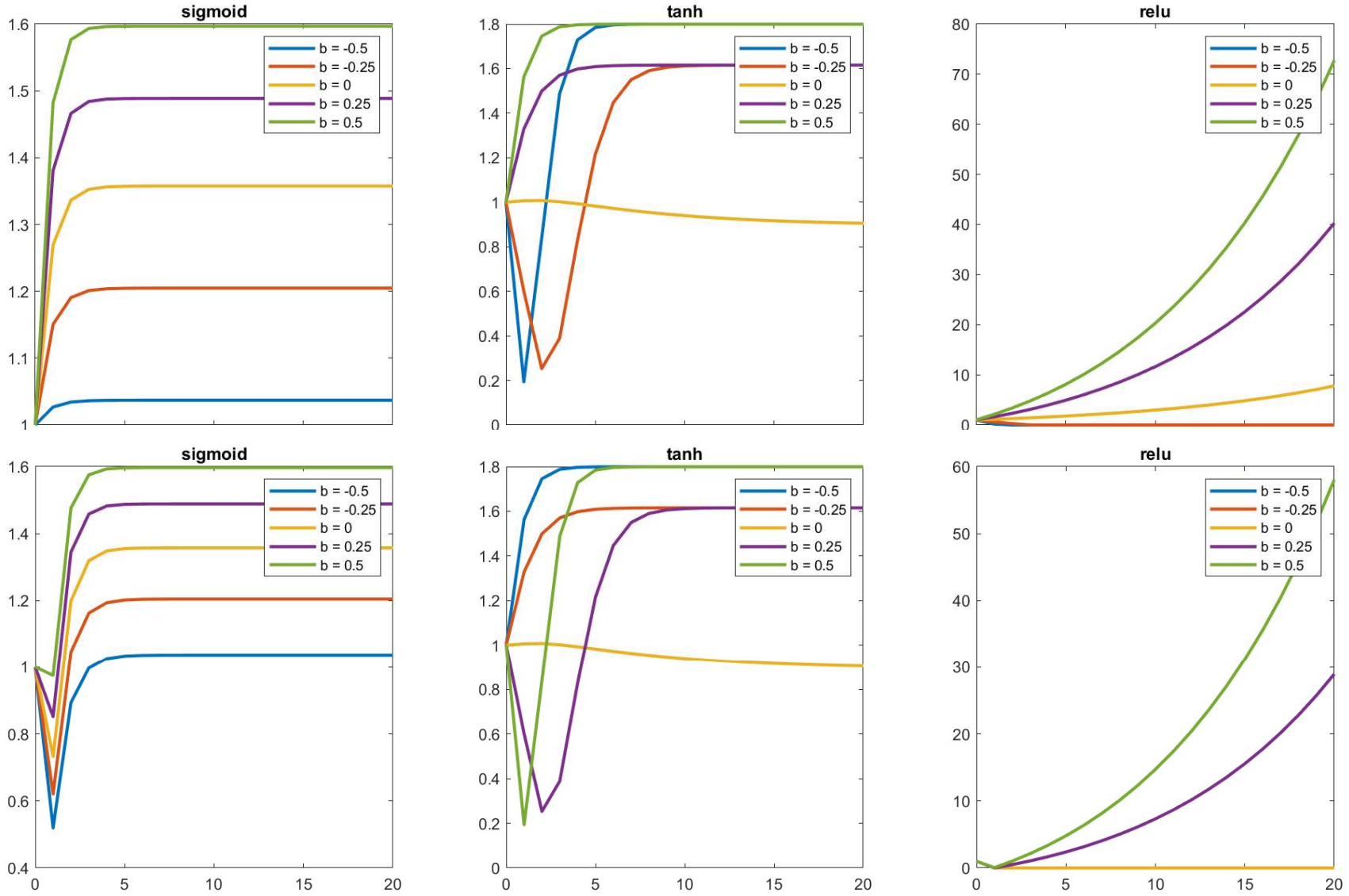
- Relu blows up if $w > 1$, for $x > 0$, and "dies" for $x < 0$
  - Unstable or useless

Scalar recurrence

Vector Process: Max eigenvalue 1.1

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

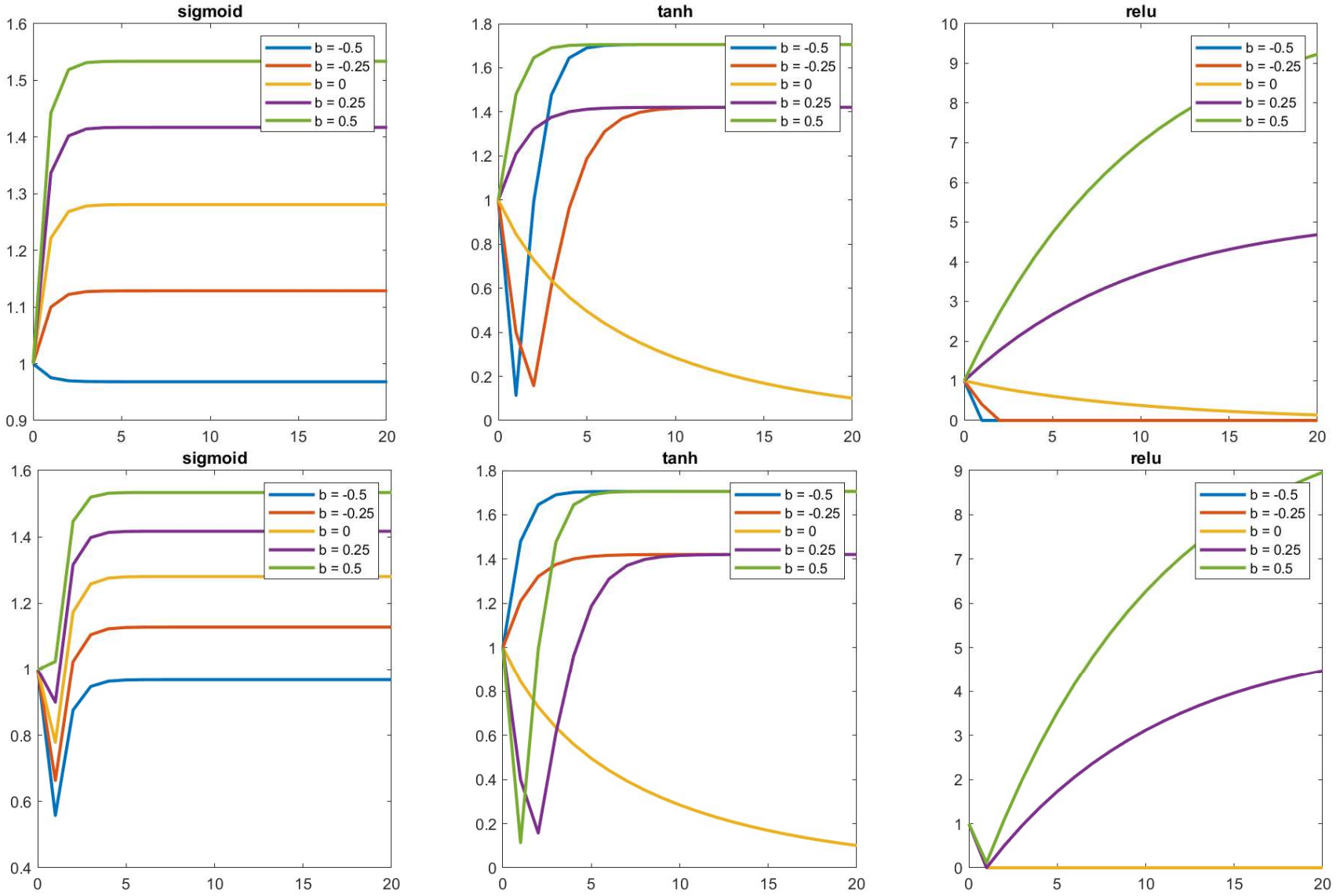
- **Initial x(0):** Top: $[1,1,1,\ldots]$, Bottom: $[-1,-1,-1,\ldots]$



Vector Process: Max eigenvalue 0.9

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

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



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*

Poll 2

Select all that are true about how long (how many time steps) an RNN can retain some memory of an input pattern

- It depends on the weights of the recurrent layers
- It depends on the bias of the recurrent layers
- It depends on the activation function used in the recurrent layers
- It depends on the actual input being "remembered"

Select all that are true about *what* an RNN remembers about an input pattern

- It depends on the weights of the recurrent layers
- It depends on the bias of the recurrent layers
- It depends on the activation function used in the recurrent layers
- It depends on the actual input being "remembered"

Poll 2

Select all that are true about how long (how many time steps) an RNN can retain some memory of an input pattern

- **It depends on the weights of the recurrent layers**
- **It depends on the bias of the recurrent layers**
- **It depends on the activation function used in the recurrent layers**
- It depends on the actual input being “remembered”

Select all that are true about *what* an RNN remembers about an input pattern

- **It depends on the weights of the recurrent layers**
- **It depends on the bias of the recurrent layers**
- **It depends on the activation function used in the recurrent layers**
- It depends on the actual input being “remembered”

Story so far

- Recurrent networks retain information from the infinite past in principle
- In practice, they tend to blow up or forget
  - If the largest Eigen value of the recurrent weights matrix is greater than 1, the network response may blow up
  - If it's less than one, the response dies down very quickly
- The "memory" of the network also depends on the parameters (and activation) of the hidden units
  - Sigmoid activations saturate and the network becomes unable to retain new information
  - RELU activations blow up or vanish rapidly
  - Tanh activations are the slightly more effective at storing memory
    - But still, for not very long

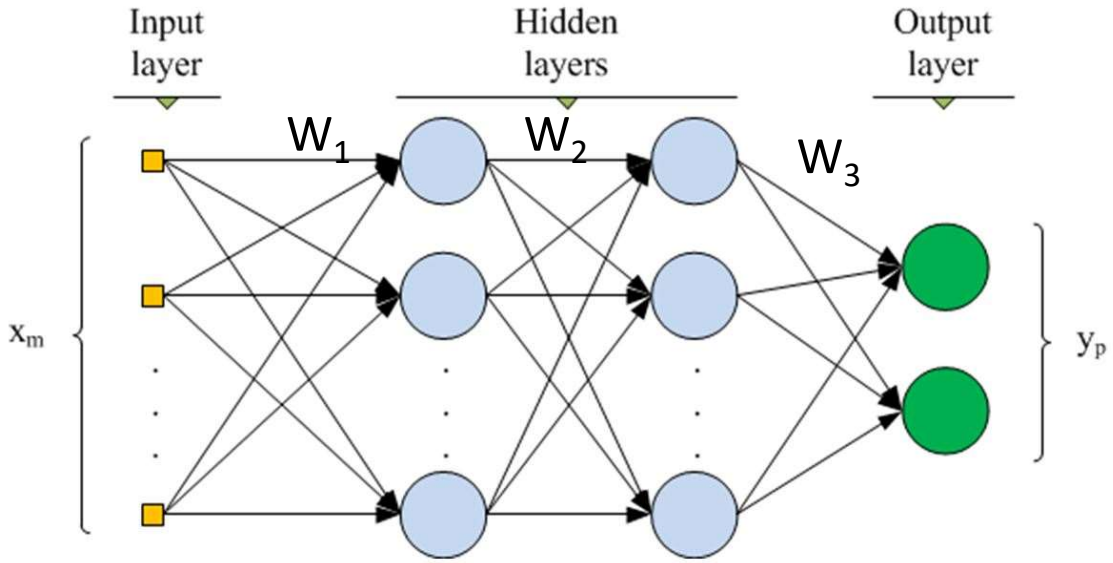
RNNs..

- 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..

Some useful preliminary math: The problem with training deep networks



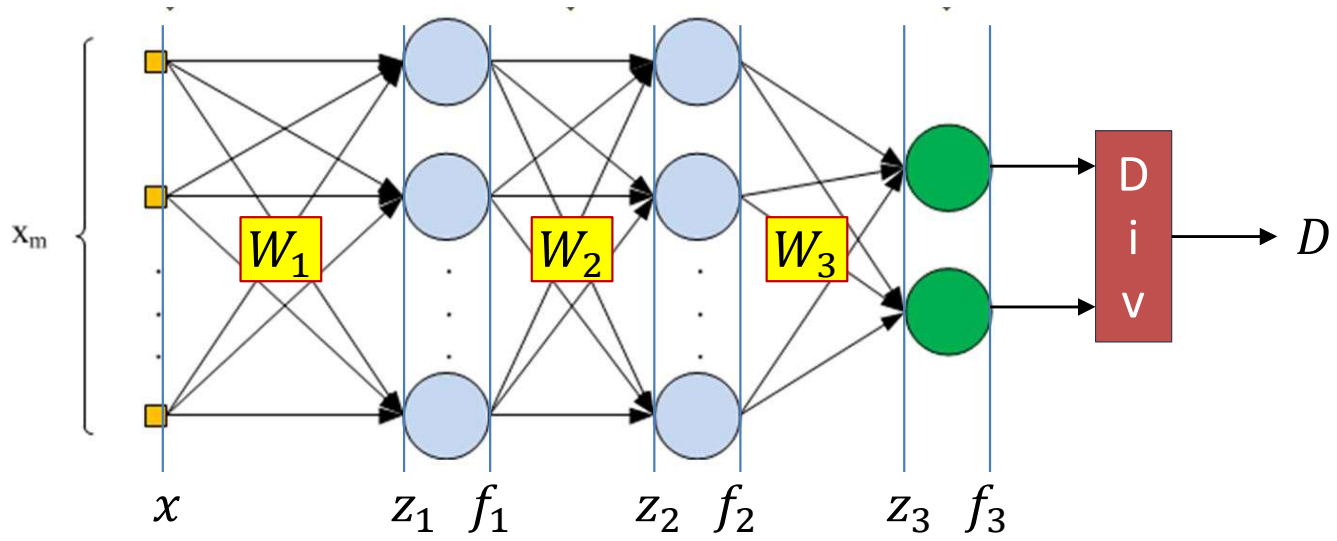
- A multilayer perceptron is a nested function

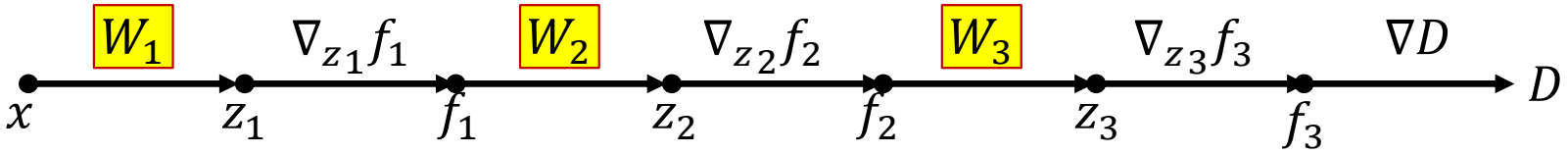
$$Y = f_N\left(W_N f_{N-1}\left(W_{N-1} f_{N-2}(\ldots W_1 X)\right)\right)$$

- W_k is the weights *matrix* at the k[th] layer
- The *error* for X can be written as

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

Training deep networks





- Recall the influence diagram
  - From the chain rule (right multiplying derivatives):

$$\nabla_{f_1} D = \nabla D . \nabla_{z_3} f_3 . W_3 . \nabla_{z_2} f_2 . W_2$$

Training deep networks

- For

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

- 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
    - Needed to compute the gradient of the error w.r.t W_{k-1}
  - ∇f_n is *jacobian* of $f_N()$ w.r.t. to its current input
    - (dropping the subscript for brevity)
  - All blue terms are matrices
  - All function derivatives are w.r.t. the (entire, affine) argument of the function

Training deep networks

- For

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

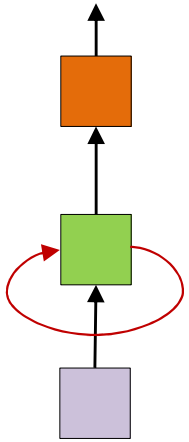
- We get:

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

- 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
    - Needed to compute the gradient of the error w.r.t W_k
  - ∇f_n is *jacobian* of $f_N()$ w.r.t. to its current input
  - All blue terms are matrices

Lets consider these Jacobians for an RNN (or more generally for any network)

The Jacobian of the hidden layers for an RNN



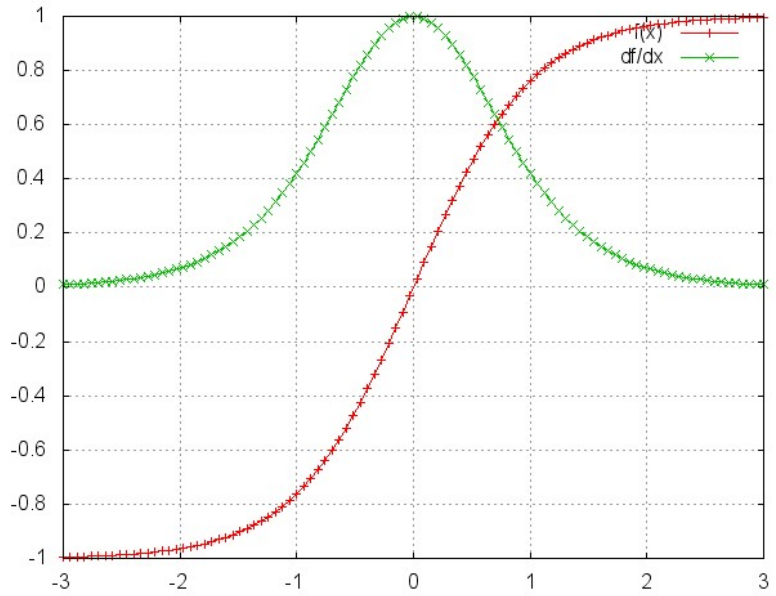
$$\nabla f_t(z_i) = \begin{bmatrix} f'_{t,1}(z_1) & 0 & \cdots & 0 \\ 0 & f'_{t,2}(z_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'_{t,N}(z_N) \end{bmatrix}$$

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

- $\nabla f_t()$ 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 function
- 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

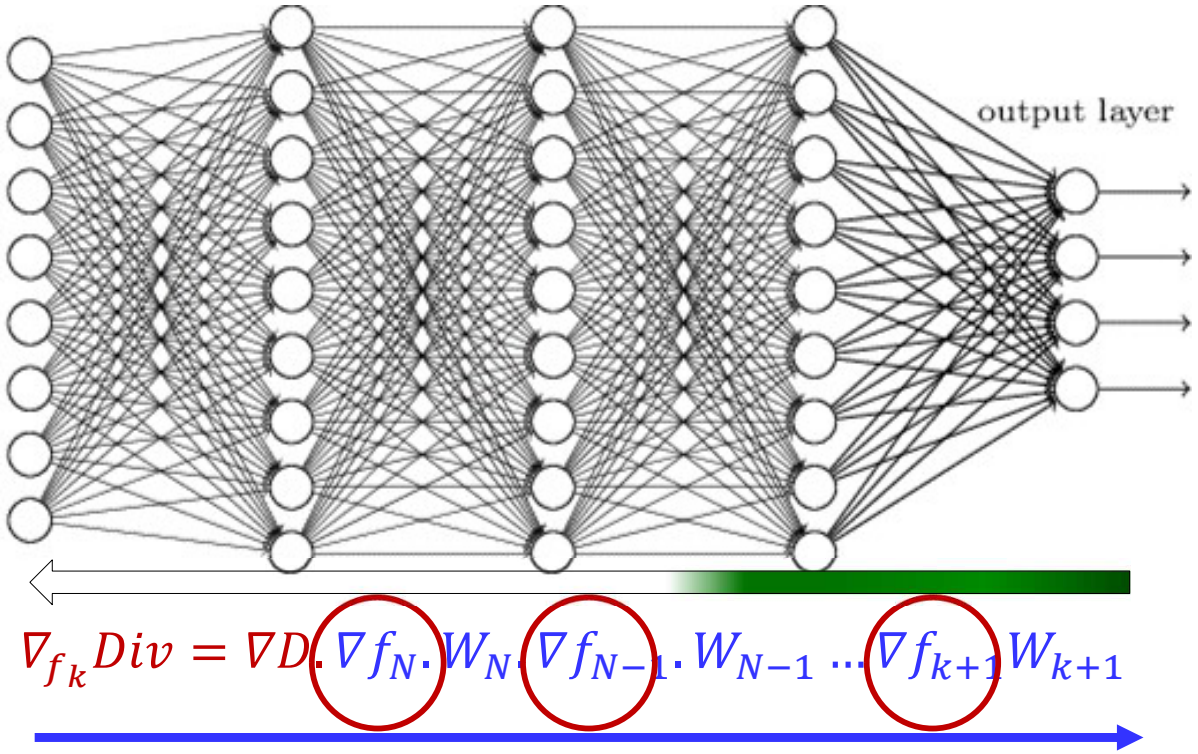
$$\nabla f_t(z_i) = \begin{bmatrix} f'_{t,1}(z_1) & 0 & \cdots & 0 \\ 0 & f'_{t,2}(z_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'_{t,N}(z_N) \end{bmatrix}$$



- 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



- 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

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

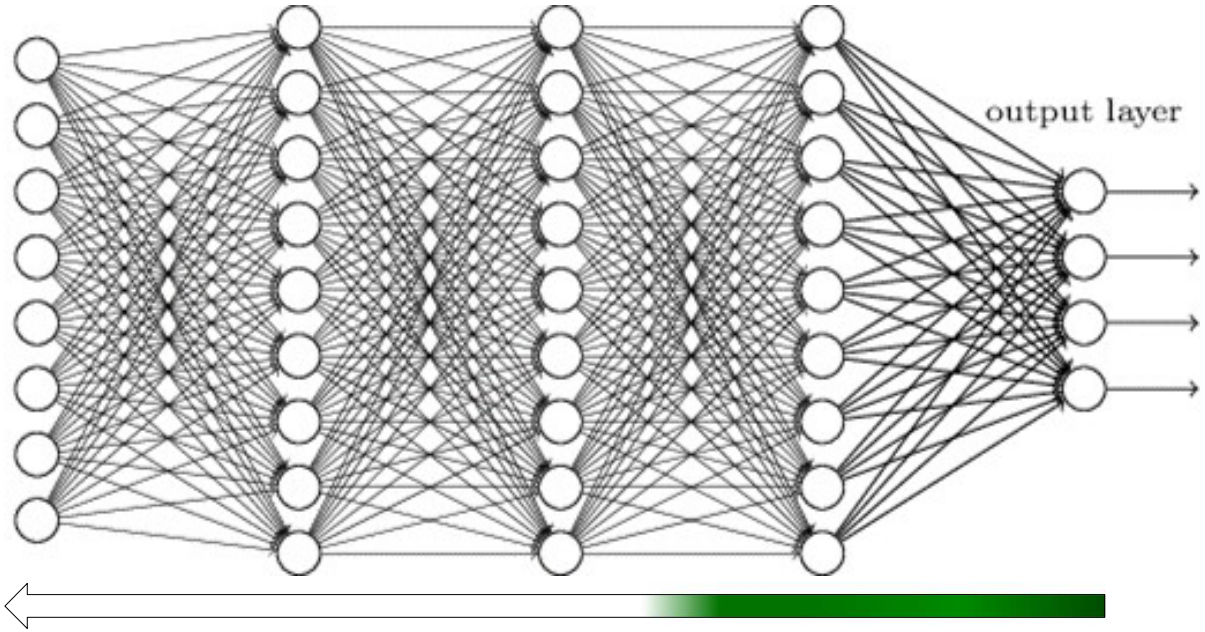
- In a single-layer RNN, the weight matrices are identical
  - The conclusion below holds for any deep network, though
- The chain product for $\nabla_{f_k} Div$ will
  - E*xpand* ∇D along directions in which the singular values of the weight matrices are greater than 1
  - S*hrink* ∇D in directions where the singular values are less than 1
  - Repeated multiplication by the weights matrix will result in **Exploding** or **vanishing** gradients

Exploding/Vanishing gradients

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

- 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
  - E*xpand* ∇D in directions where each stage has singular values greater than 1
  - S*hrink* ∇D in directions where each stage has singular values less than 1

Gradient problems in deep networks

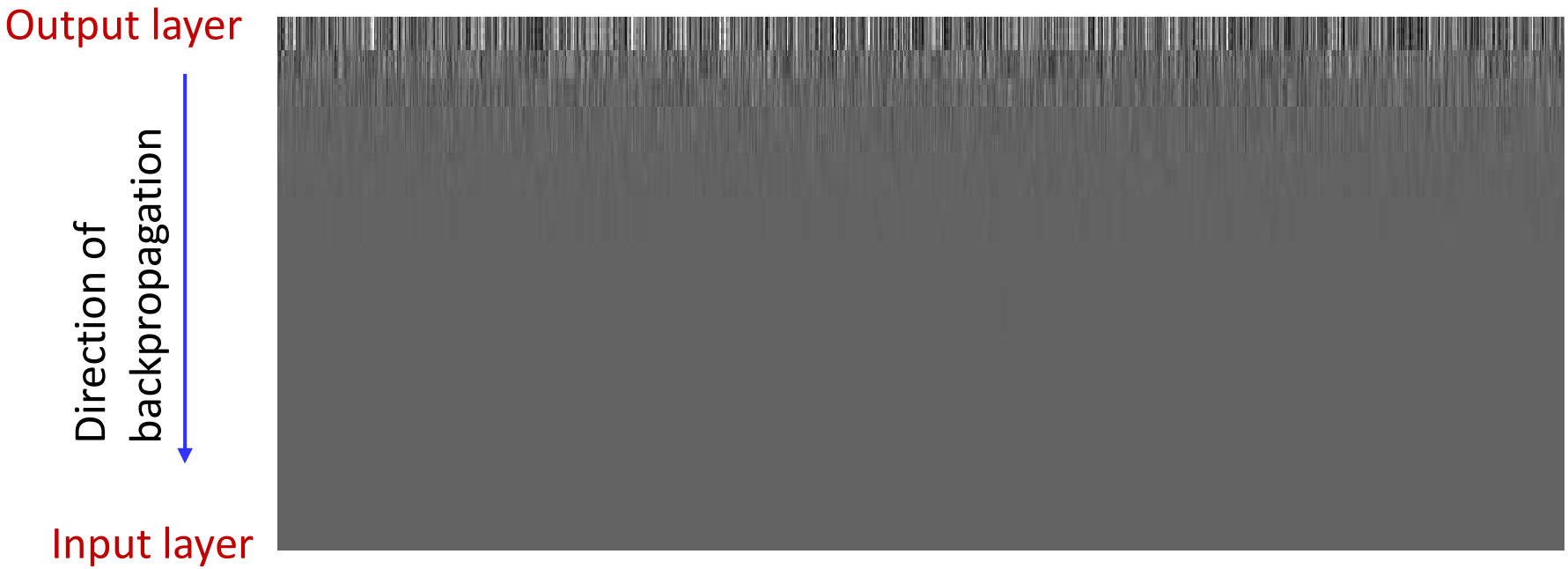


$$\nabla_{f_k} Div = \nabla D . \nabla f_N . W_N . \nabla f_{N-1} . W_{N-1} \ldots \nabla f_{k+1} W_{k+1}$$

- 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, 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

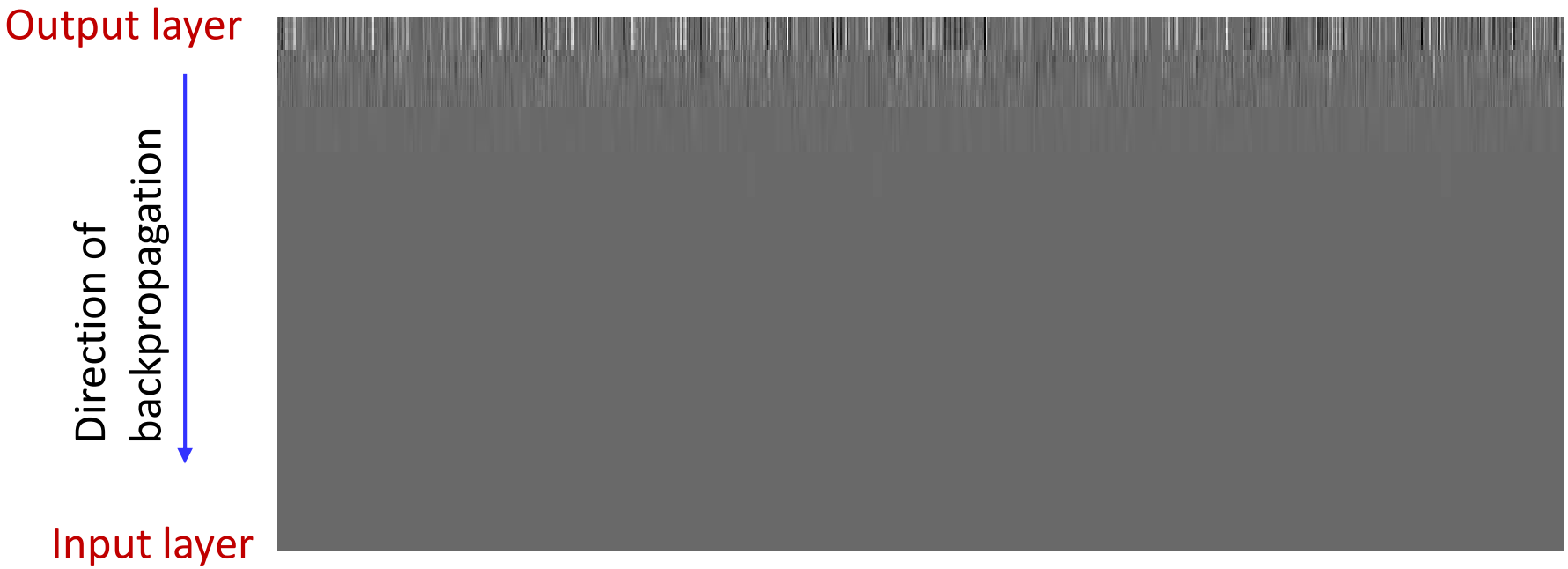
Vanishing gradient examples..

RELU activation, Batch gradients

- 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
- 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..

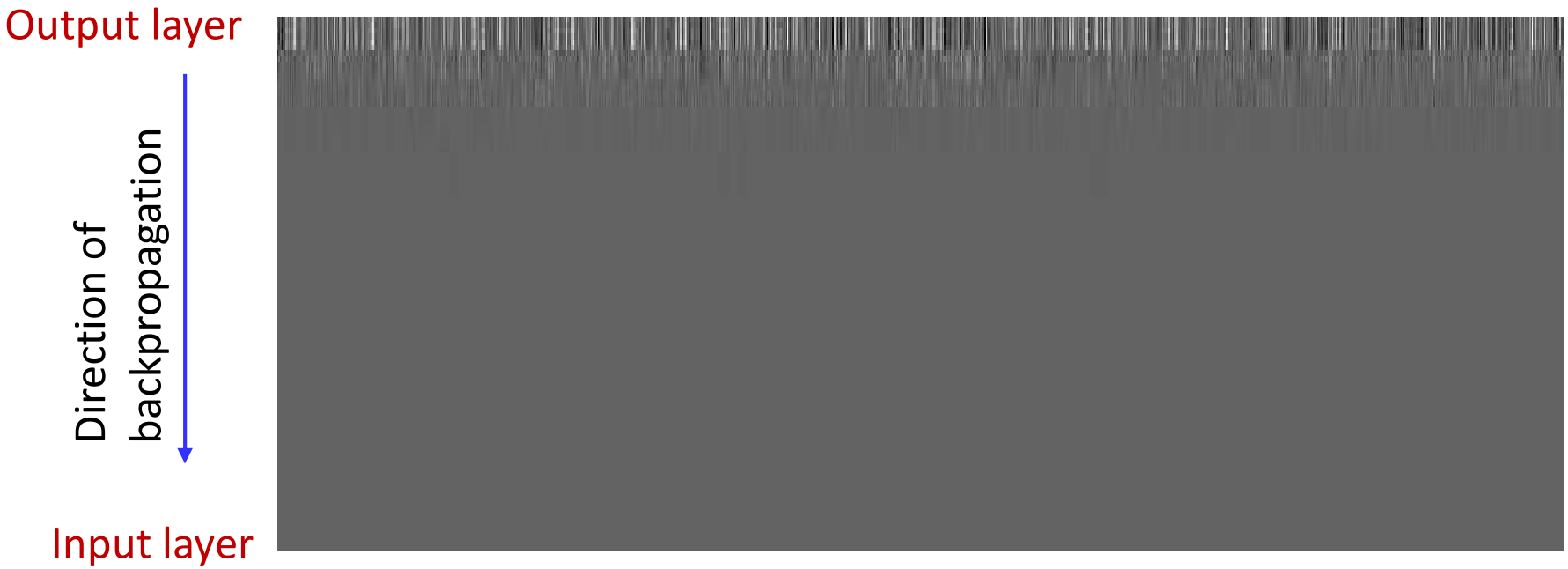
Sigmoid activation, Batch gradients



- 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
- 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..

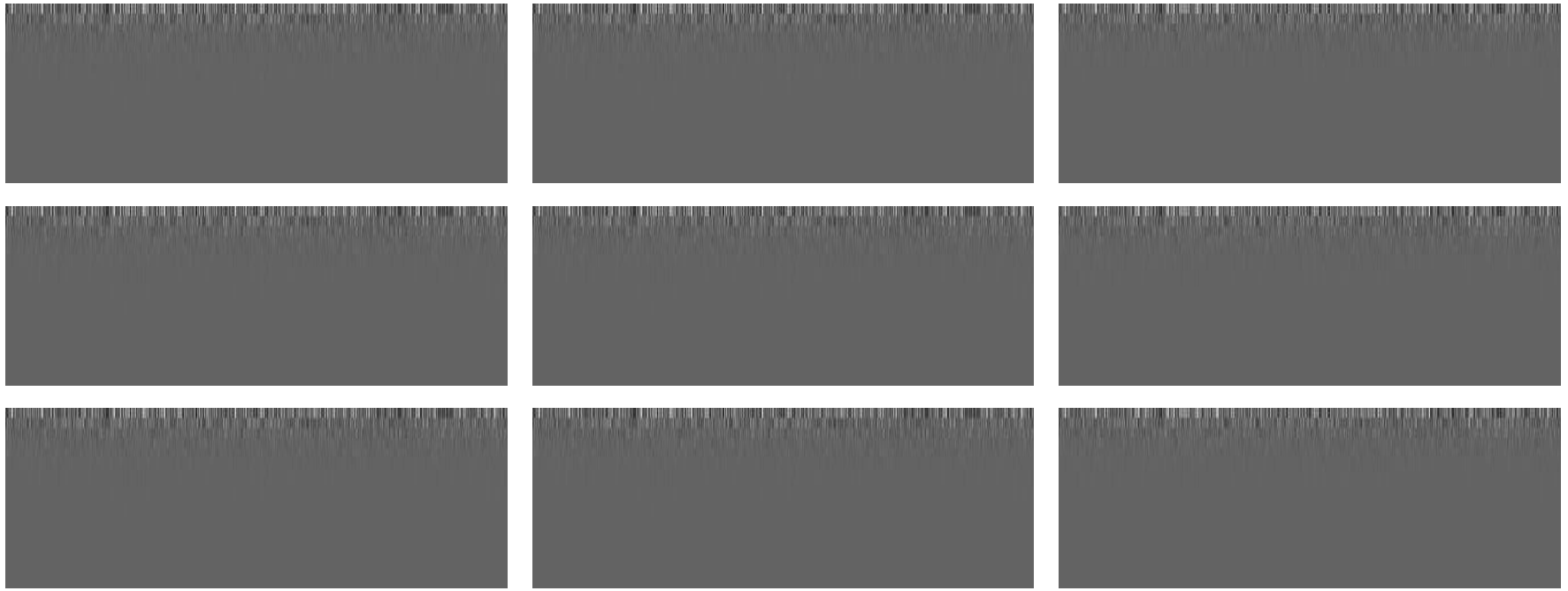
Tanh activation, Batch gradients



- 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
- 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..

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
- 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 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.

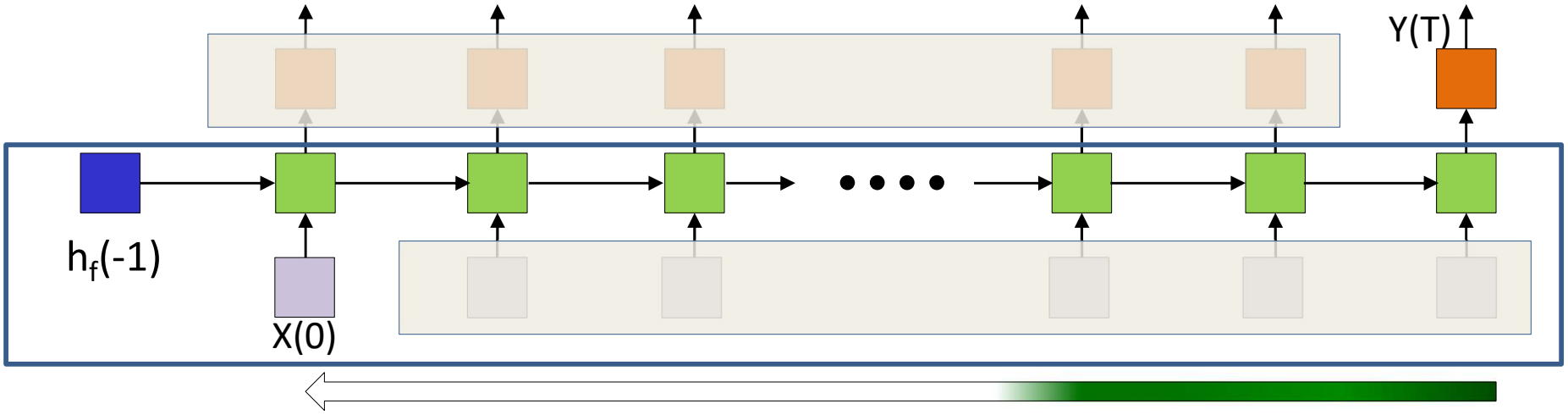
Story so far

- 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

- 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

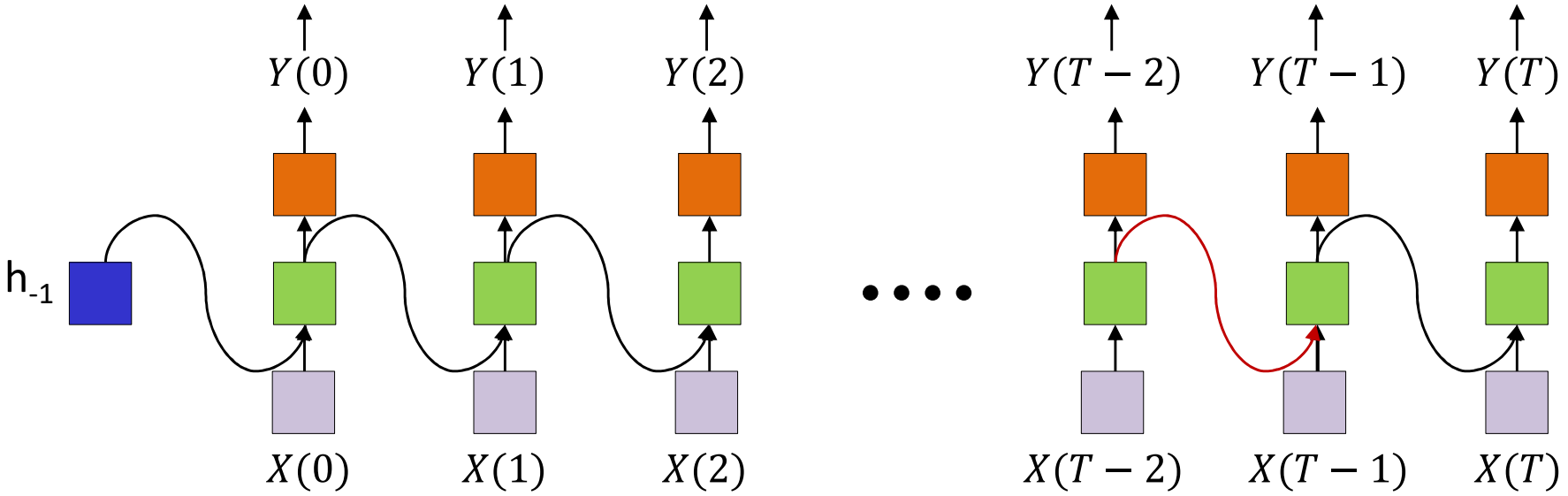
Recurrent nets are very deep nets

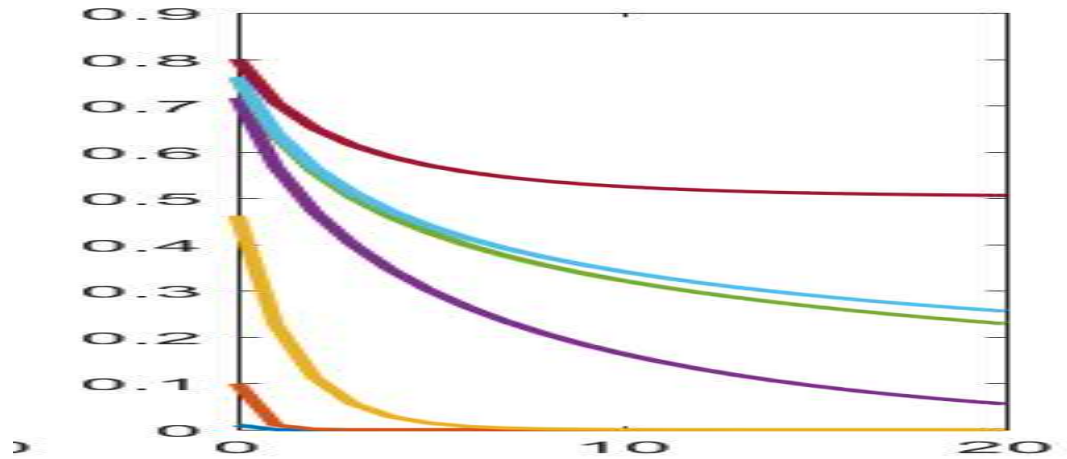


$$\nabla_{f_k} Div = \nabla D . \nabla f_N . W_N . \nabla f_{N-1} . W_{N-1} \ldots \nabla f_{k+1} W_{k+1}$$

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

Recall: Vanishing stuff..





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

Poll 3

Select all that are true

- The derivatives for most parameters will become vanishingly small as we backpropagate the loss gradient through deep networks
- The derivatives for a small number of parameters will blow up and become large and unstable as we propagate the los gradient through deep networks
- The derivatives would be more stable if the recurrent weight matrices had singular values equal to 1
- The derivatives would be more stable if the recurrent activations were identity transforms (with identity Jacobian matrices)

Select all that are true of recurrent networks

- The memory of the recurrent layer is limited because the recurrent weight matrices are not unitary (with all eigen values equal to 1)
- The memory is also limited by nonlinear activation functions
- The memory would be more stable if the recurrent weight matrix were an identity matrix (i.e. a diagonal matrix with diagonal values equal to “1”)
- The memory would be more stable if the recurrent activations were identity transforms (which are linear and do not scale up or shrink the output)

Poll 3

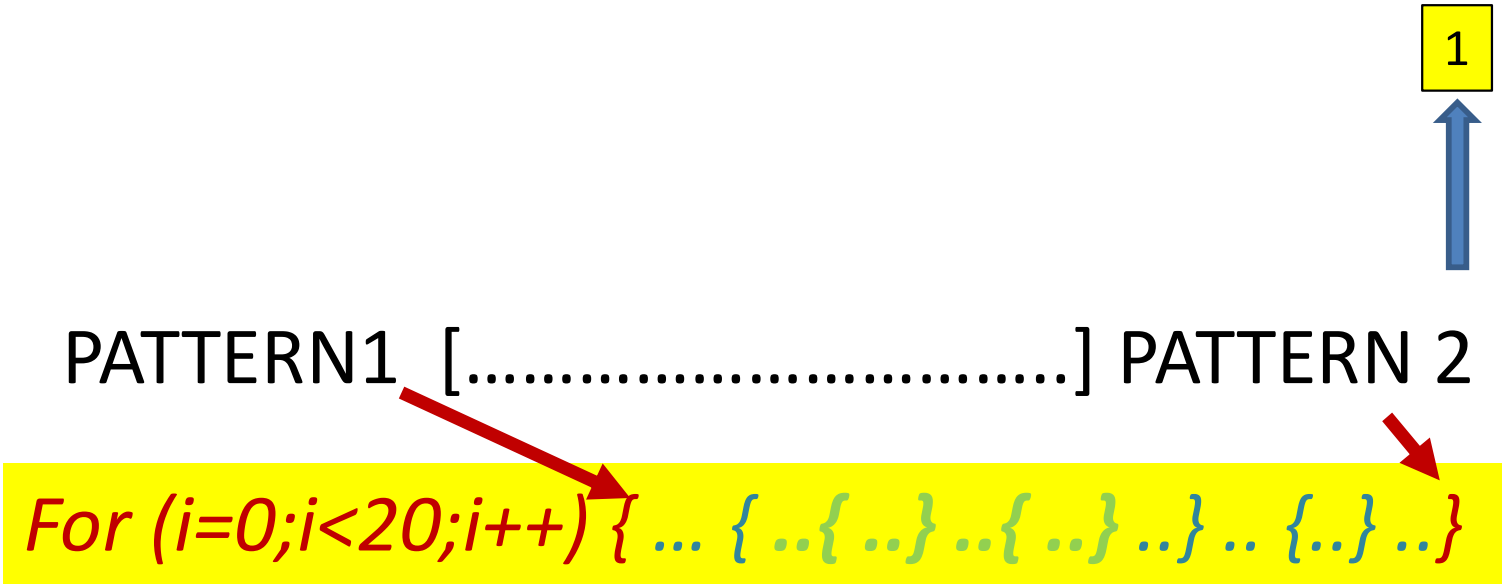
Select all that are true

- **The derivatives for most parameters will become vanishingly small as we backpropagate the loss gradient through deep networks**
- **The derivatives for a small number of parameters will blow up and become large and unstable as we propagate the los gradient through deep networks**
- **The derivatives would be more stable if the recurrent weight matrices had singular values equal to 1**
- **The derivatives would be more stable if the recurrent activations were identity transforms (with identity Jacobian matrices)**

Select all that are true of recurrent networks

- **The memory of the recurrent layer is limited because the recurrent weight matrices are not unitary (with all eigen values equal to 1)**
- **The memory is also limited by nonlinear activation functions**
- **The memory would be more stable if the recurrent weight matrix were an identity matrix (i.e. a diagonal matrix with diagonal values equal to "1")**
- **The memory would be more stable if the recurrent activations were identity transforms (which are linear and do not scale up or shrink the output)**

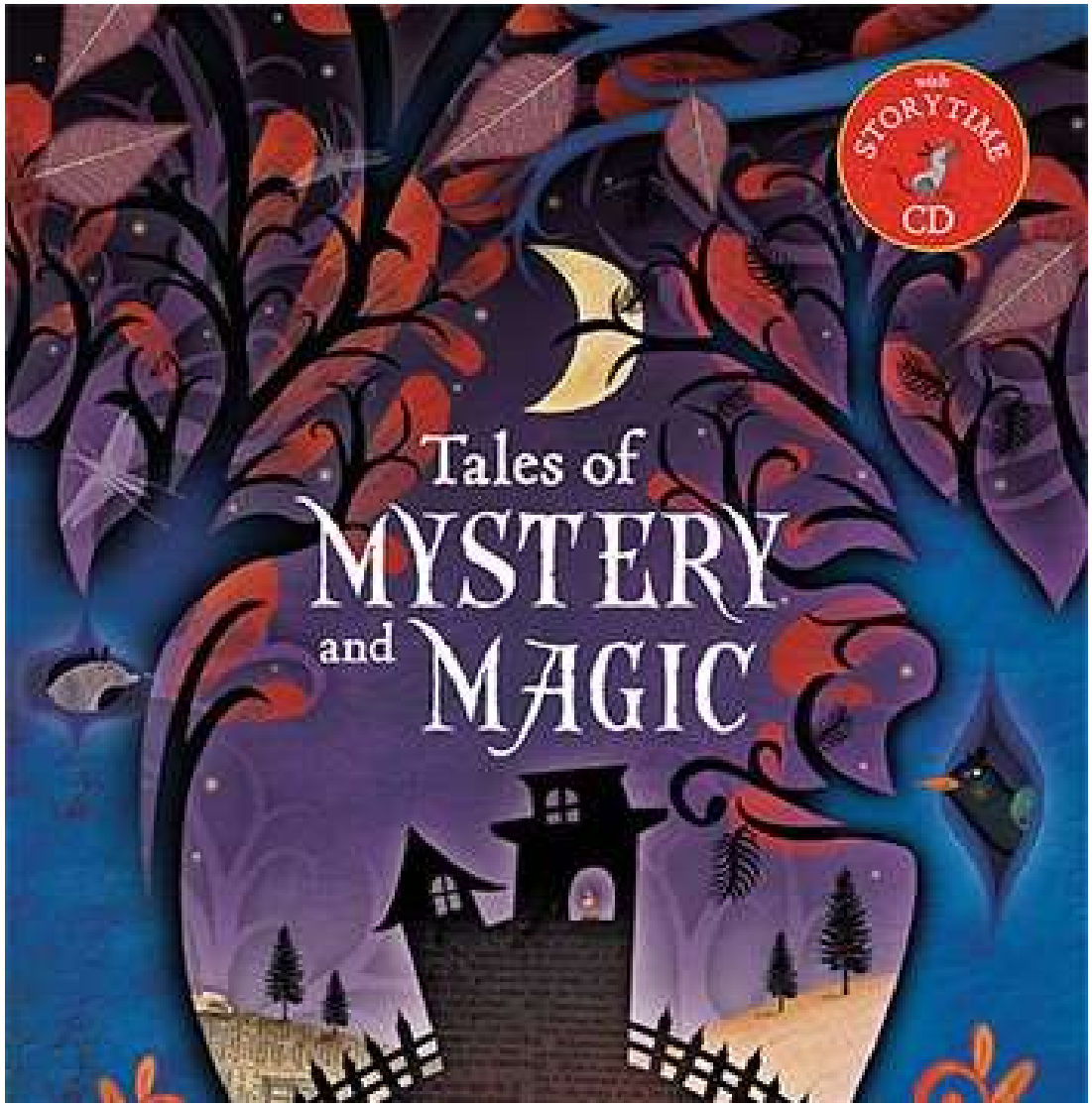
The long-term dependency problem



- 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
  - "{" → must be remembered until the closing "}" regardless of the length of the program or how many other braces are opened or closed in-between

- Must know to "remember" for extended periods of time and "recall" when necessary
  - Can be performed with a multi-tap recursion, but how many taps?
  - Need an alternate way to "remember" stuff

And now we enter the domain of..



Exploding/Vanishing gradients

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

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

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

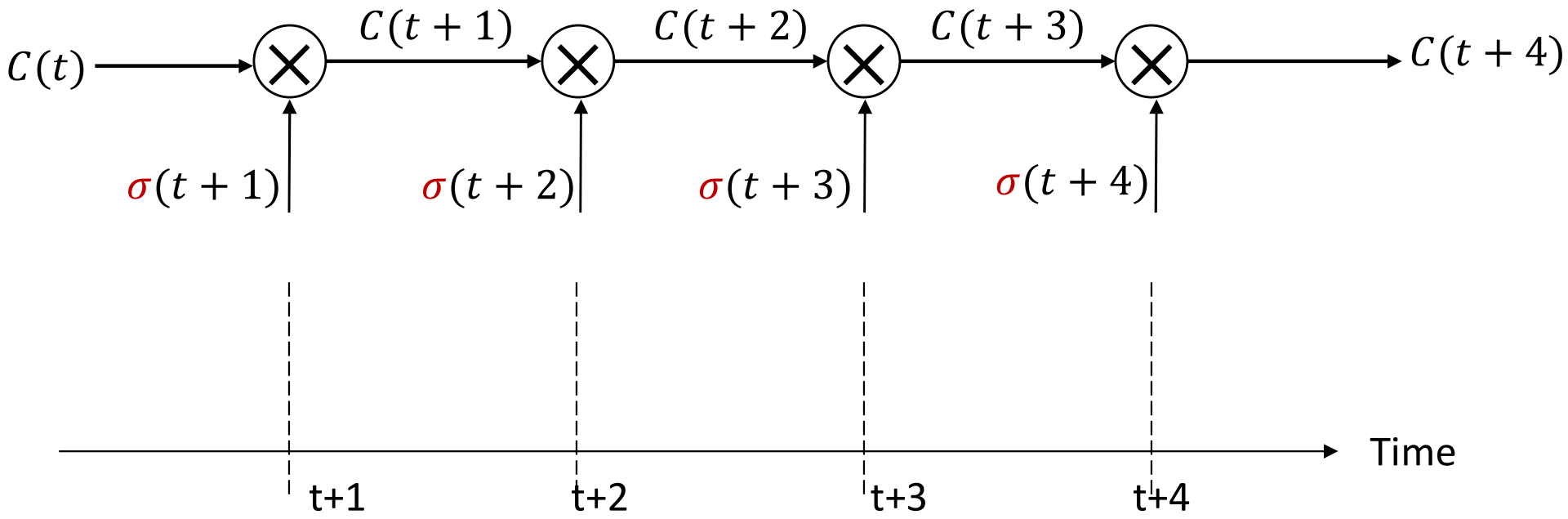
Exploding/Vanishing gradients

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

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

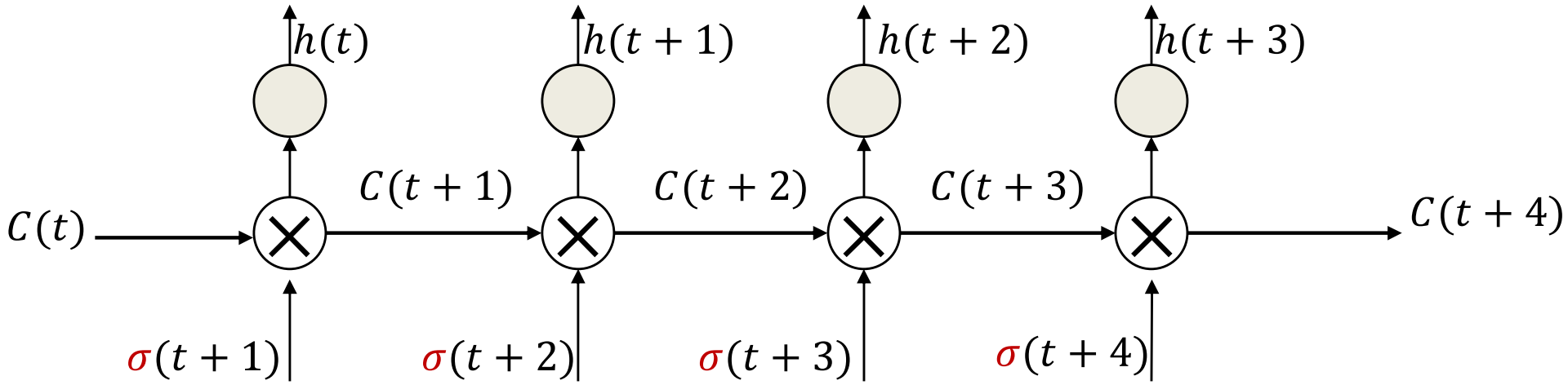
- Replace this with something that doesn't fade or blow up?
- Network that "retains" *useful* memory arbitrarily long, to be recalled on demand?
  - Input-based determination of *whether it must be remembered*
  - **Retain memories until a switch *based on the input* flags them as ok to forget**
    - 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



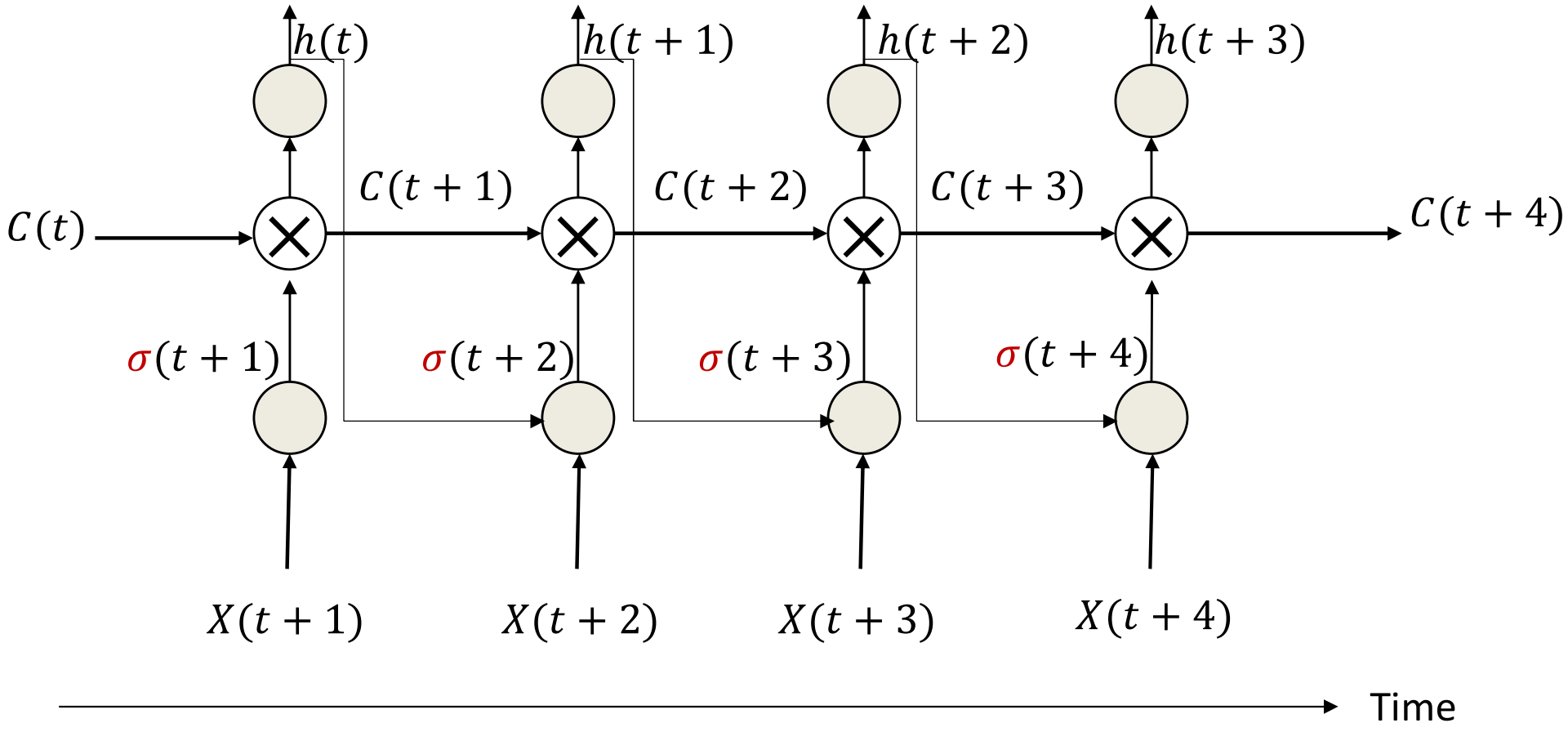
- History is carried through uncompressed
  - No weights, no nonlinearities
  - Only scaling is through the σ "gating" term that captures other triggers
  - E.g. "Have I seen Pattern2"?

Enter – the constant error carousel



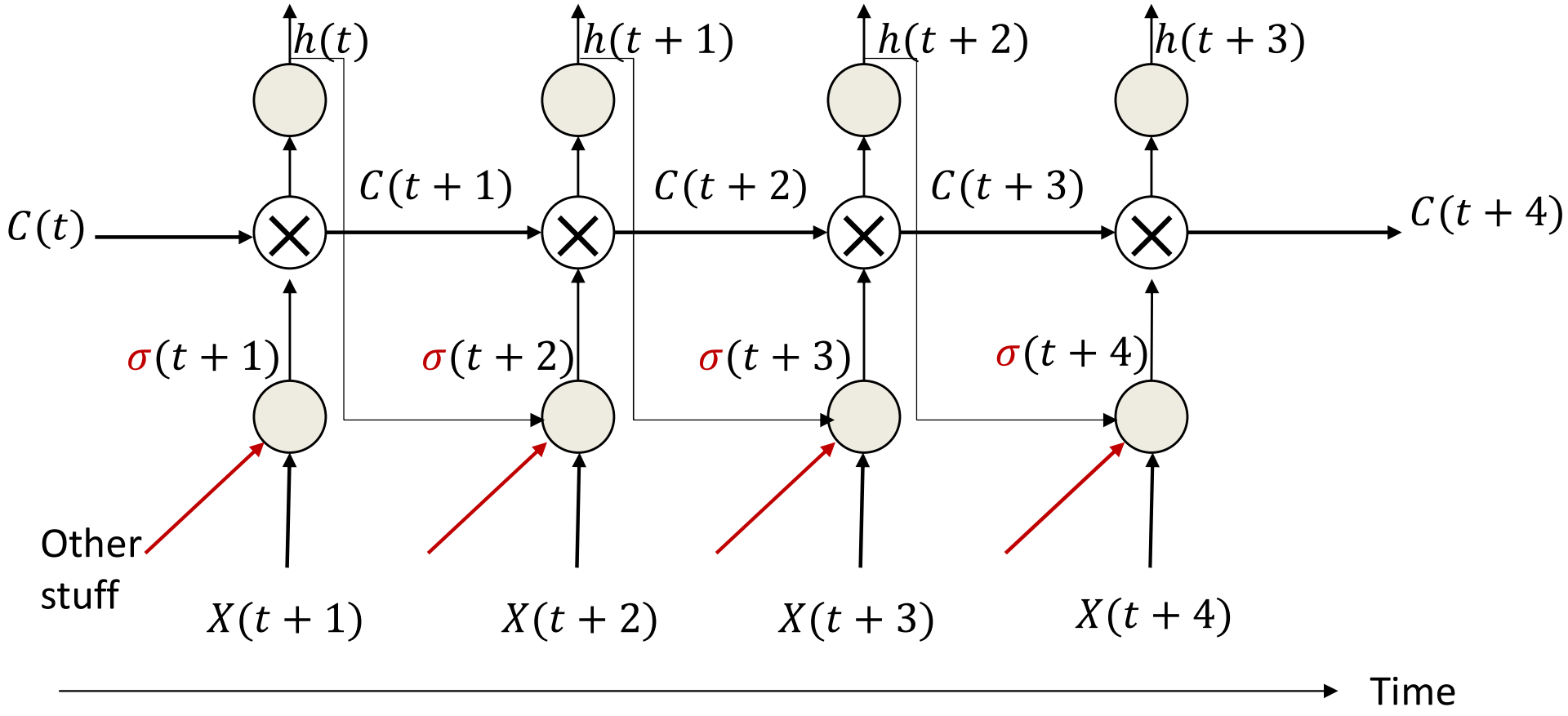
- Actual non-linear work is done by other portions of the network
  - Neurons that compute the workable state from the memory

Enter – the constant error carousel



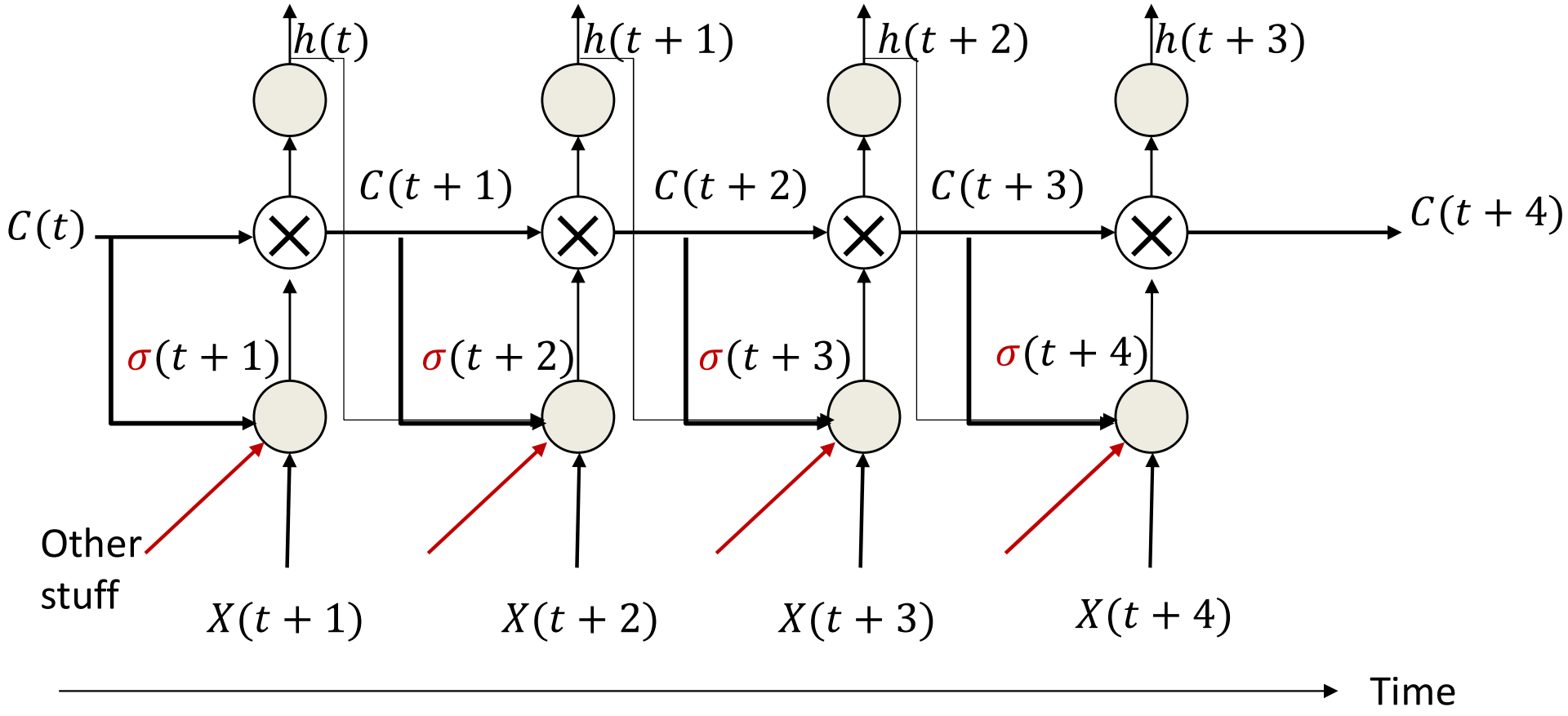
- The gate σ depends on current input, current hidden state…

Enter – the constant error carousel



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

Enter – the constant error carousel



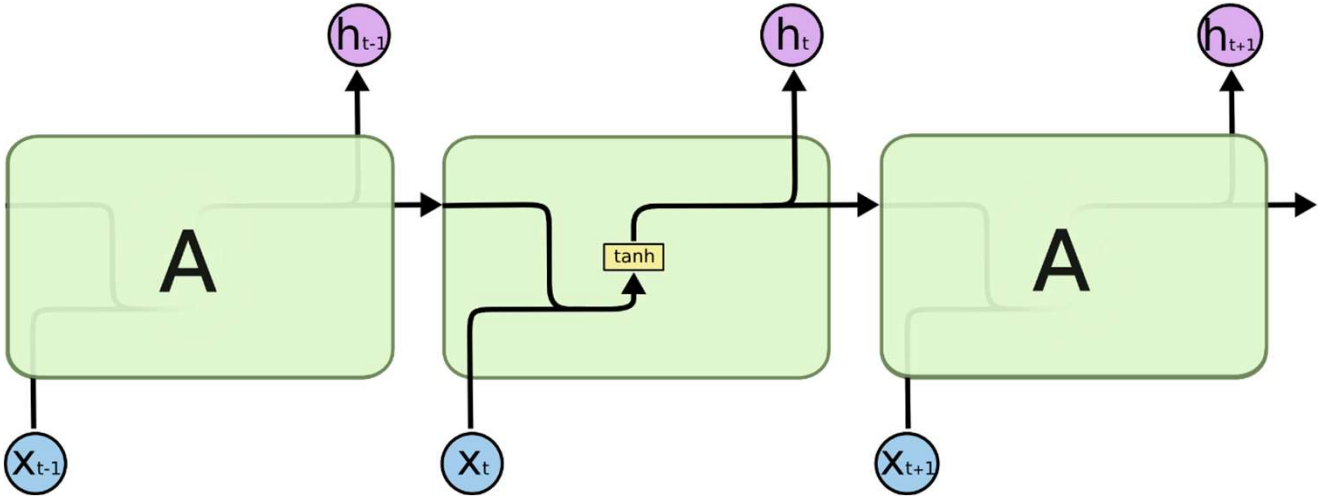
- The gate σ depends on current input, current hidden state… and other stuff…
- Including, obviously, what is currently in raw memory

Enter the *LSTM*

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

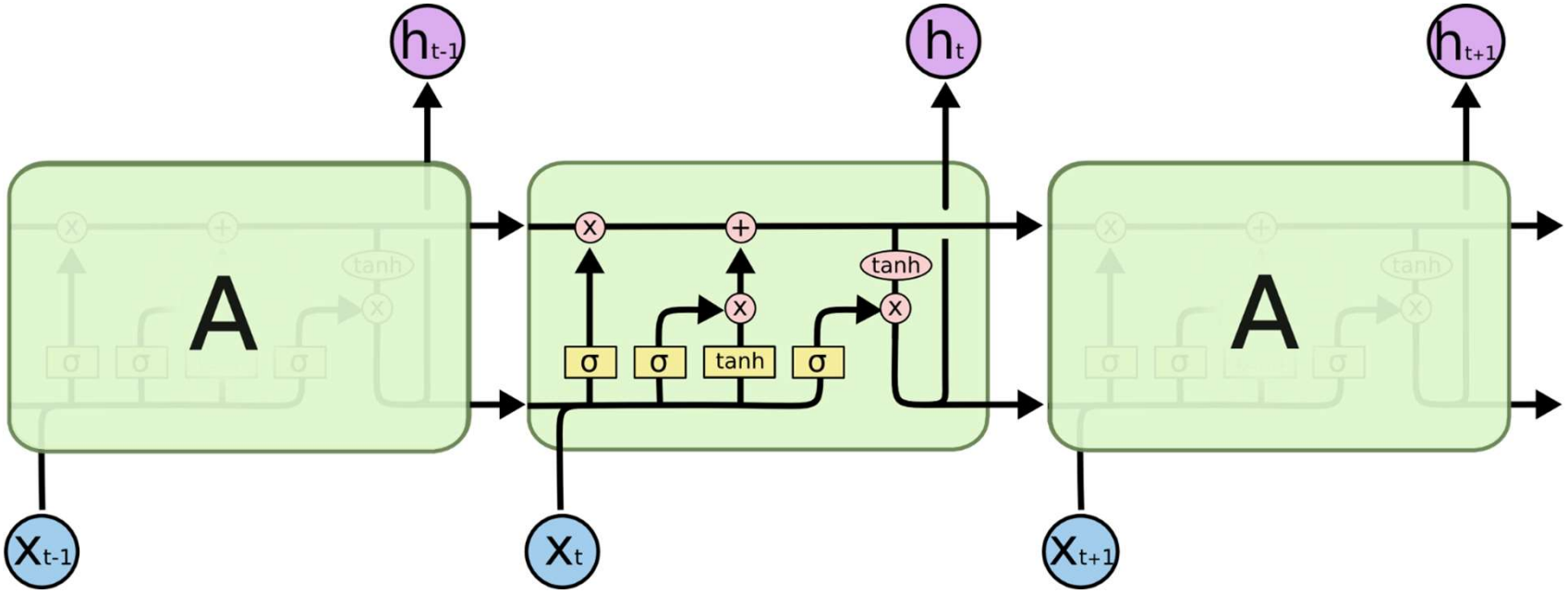
- Following notes borrow liberally from
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Standard RNN



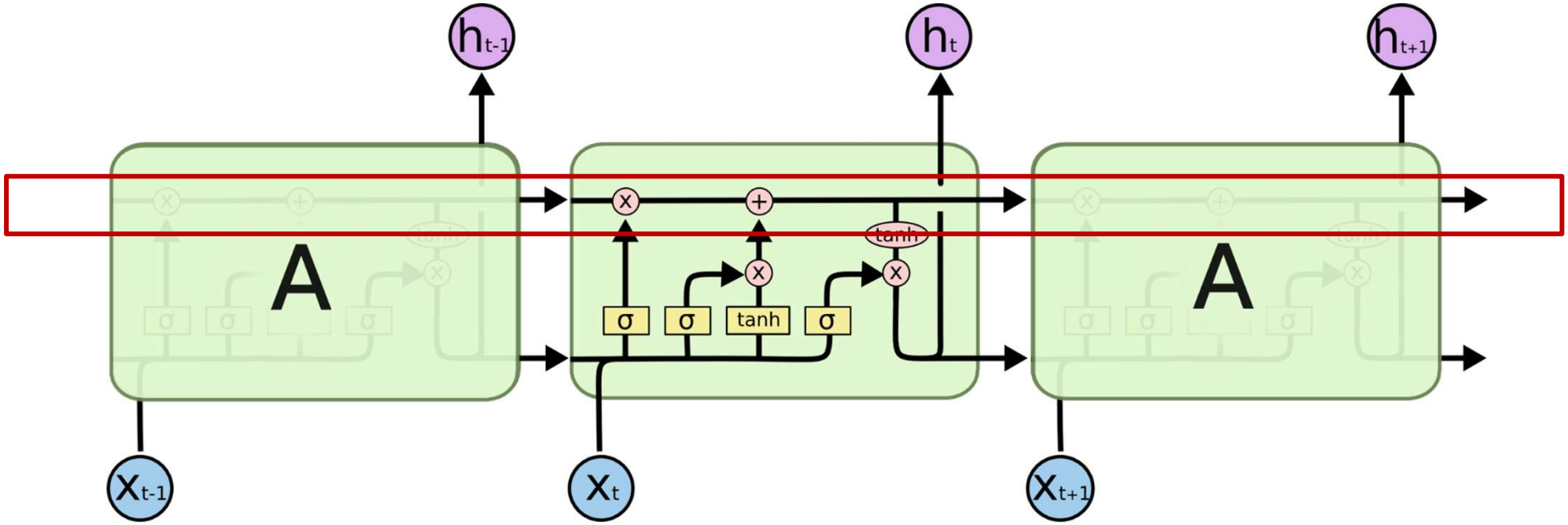
- Recurrent neurons receive past recurrent outputs and current input as inputs
- Processed through a tanh() activation function
  - As mentioned earlier, tanh() is the generally used activation for the hidden layer
- Current recurrent output passed to next higher layer and next time instant

Long Short-Term Memory



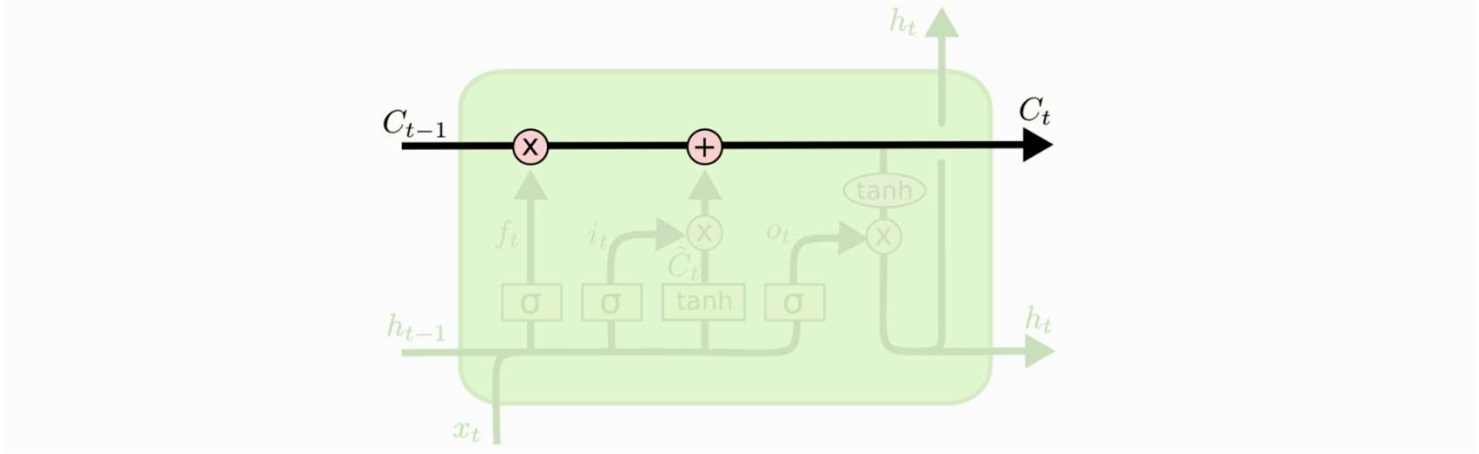
- The $\sigma()$ are *multiplicative gates* that decide if something is important or not
- Remember, every line actually represents a *vector*

LSTM: Constant Error Carousel



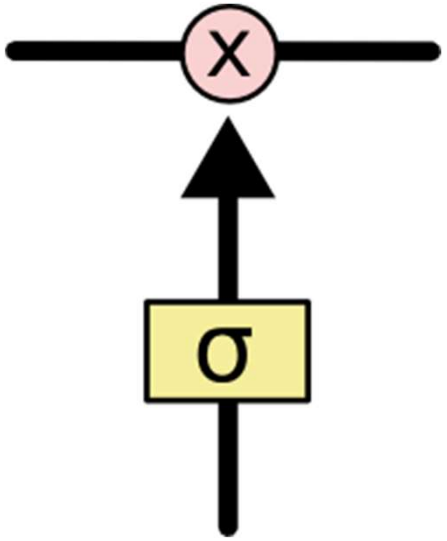
- Key component: a *remembered cell state*

LSTM: CEC



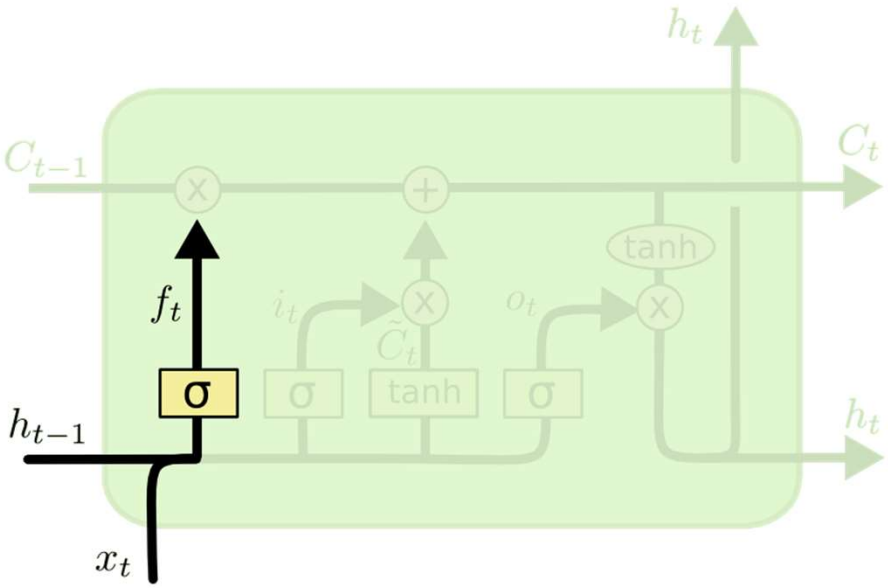
- C_t is the linear history carried by the *constant-error carousel*
- Carries information through, only affected by a gate
  - And *addition of history,* which too is gated..

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

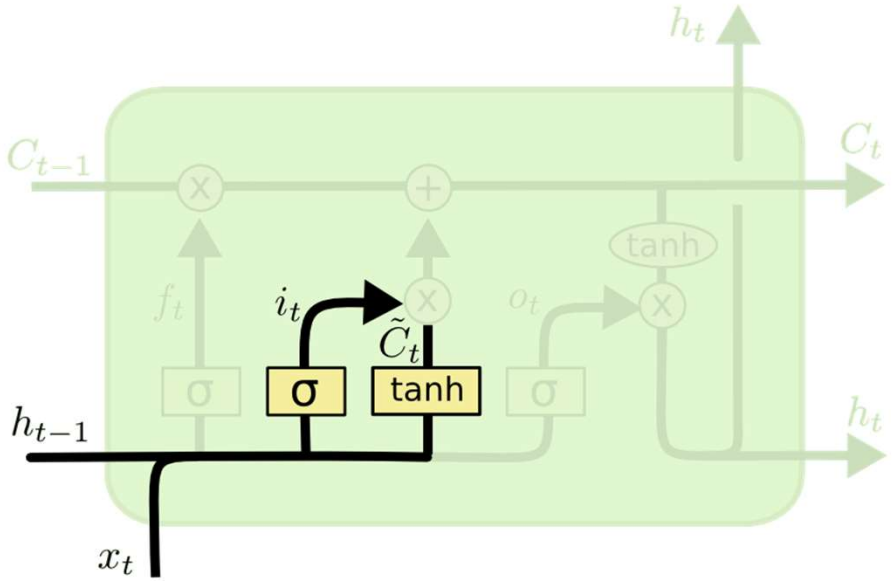
LSTM: Forget gate



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

- 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

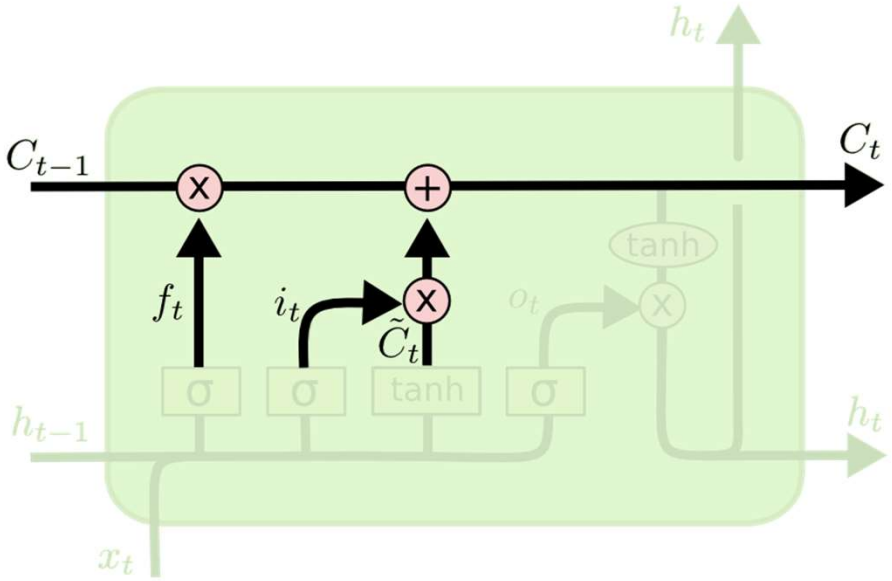


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

- 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

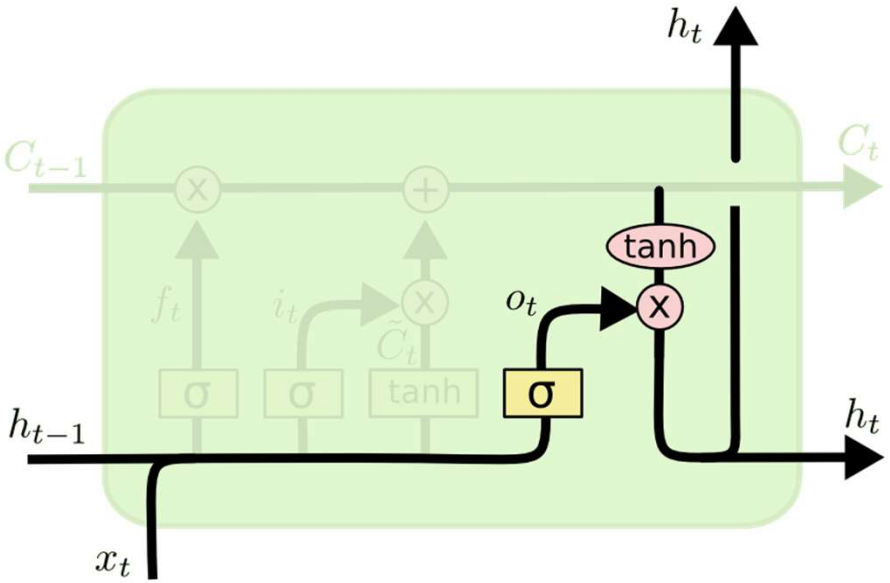
LSTM: Memory cell update



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

- 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

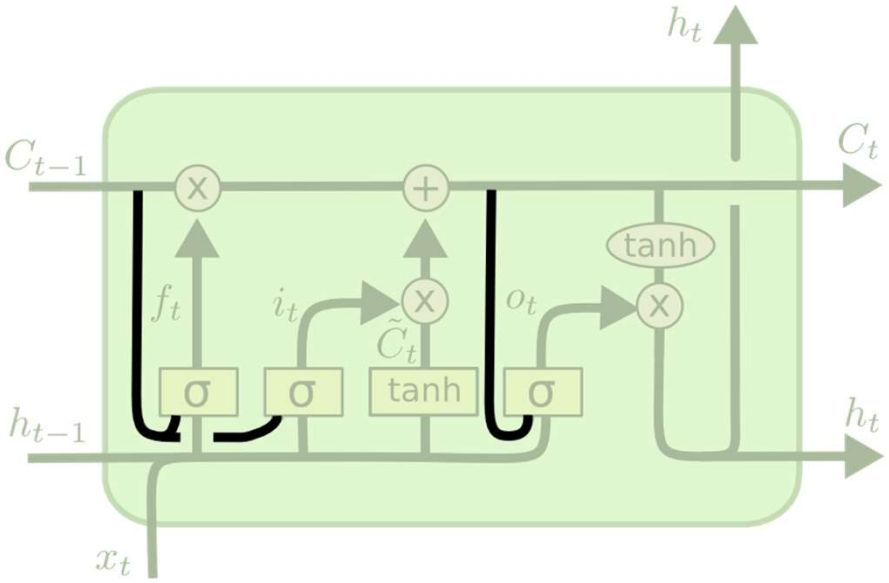


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

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

- 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

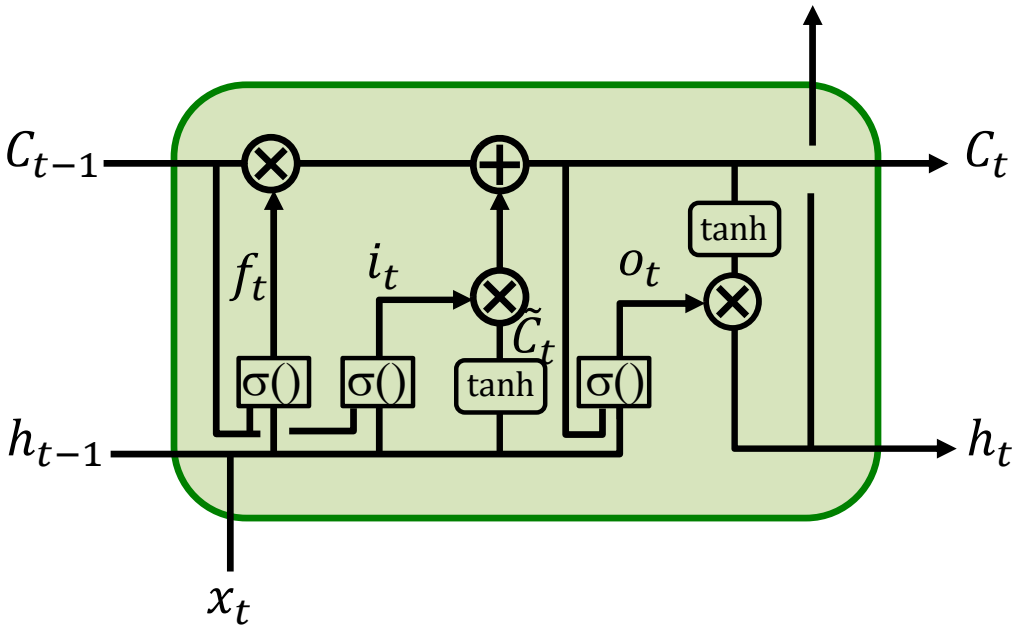
LSTM: The "Peephole" Connection



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

- 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..

Poll 4

Select all that are true about LSTMs

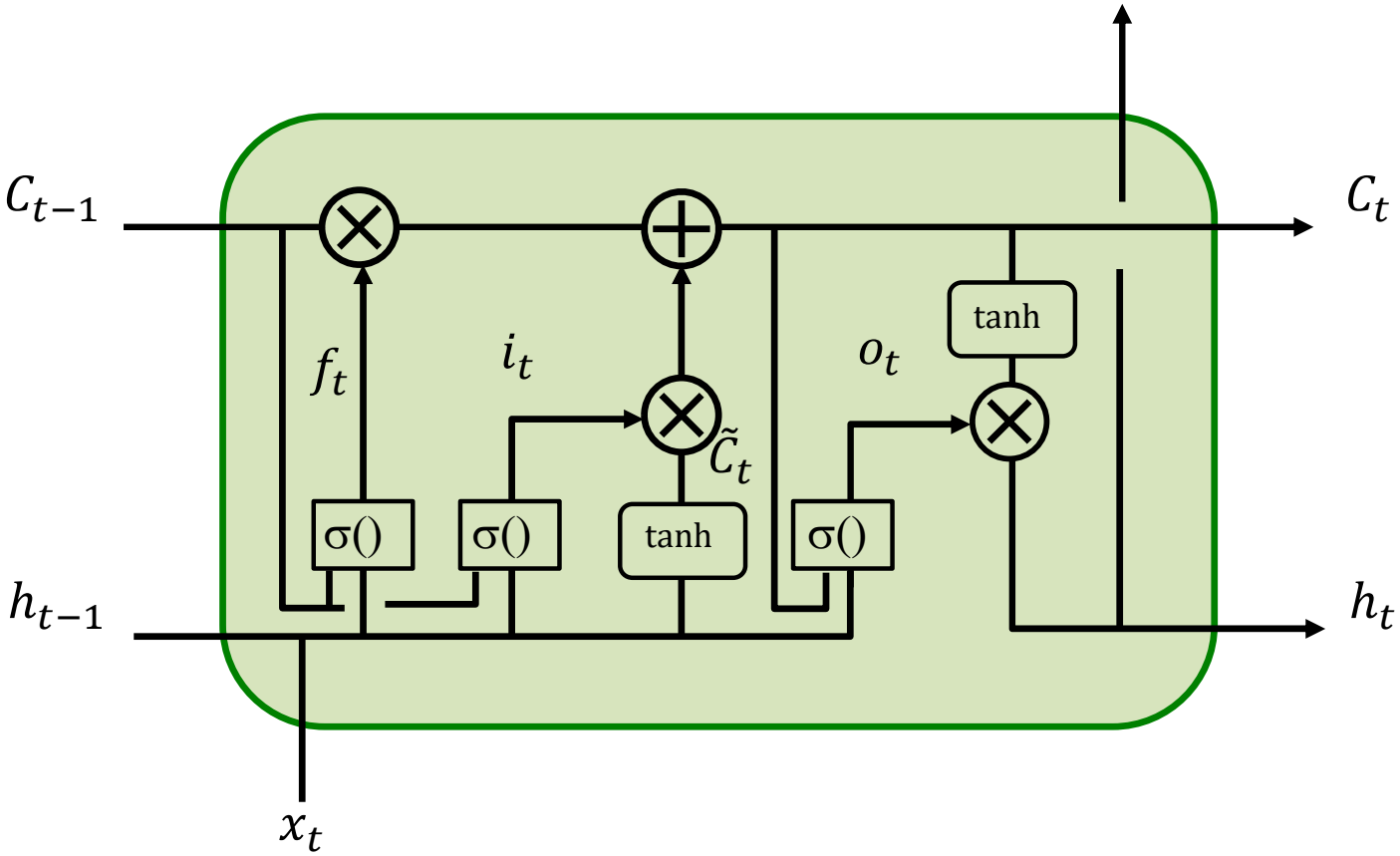
- LSTMs “stabilize” the memory by eliminating the problematic recurrent weights and activations
- They update the memory based on patterns detected in the input and the current context of what they already remember
- In the absence of external cues, they can “remember” a pattern forever
- LSTM are suited to building pattern analyzers requiring long-term memory, e.g. code parsers that can verify if an opened brace has been properly closed

Poll 4

Select all that are true about LSTMs

- **LSTMs "stabilize" the memory by eliminating the problematic recurrent weights and activations**
- **They update the memory based on patterns detected in the input and the current context of what they already remember**
- **In the absence of external cues, they can "remember" a pattern forever**
- **LSTM are suited to building pattern analyzers requiring long-term memory, e.g. code parsers that can verify if an opened brace has been properly closed**

LSTM computation: Forward



- Forward rules:

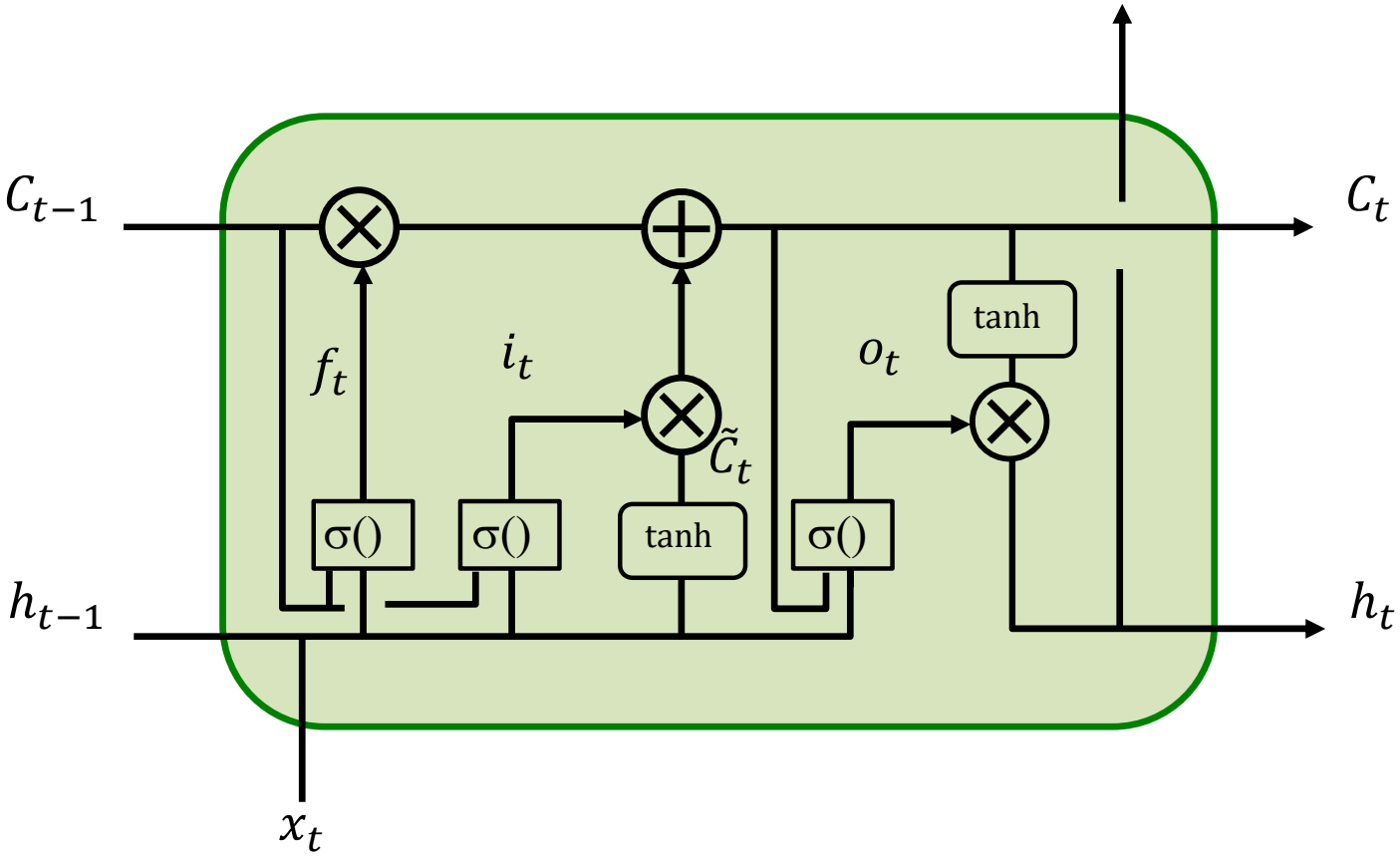
Gates

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

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

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

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_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, which is not standard

LSTM network forward

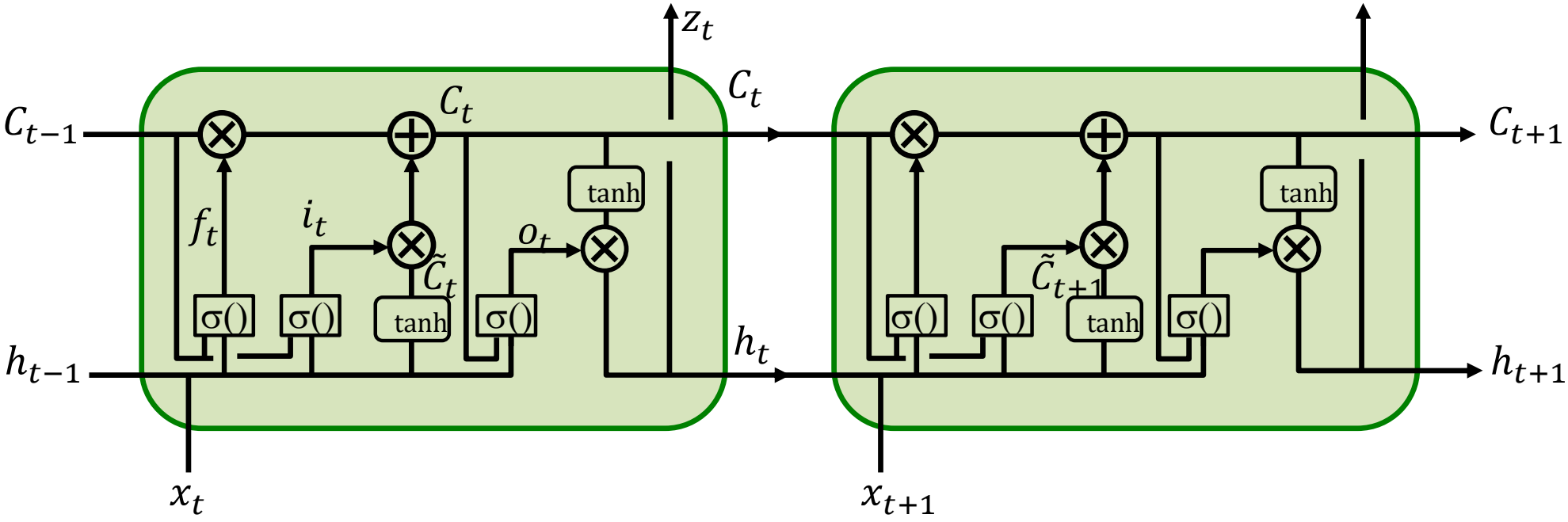
```
# Assuming h(-1,*) is known and C(-1,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
#                for the l^th hidden layer
# W_o and b_o are output layer weights and biases

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

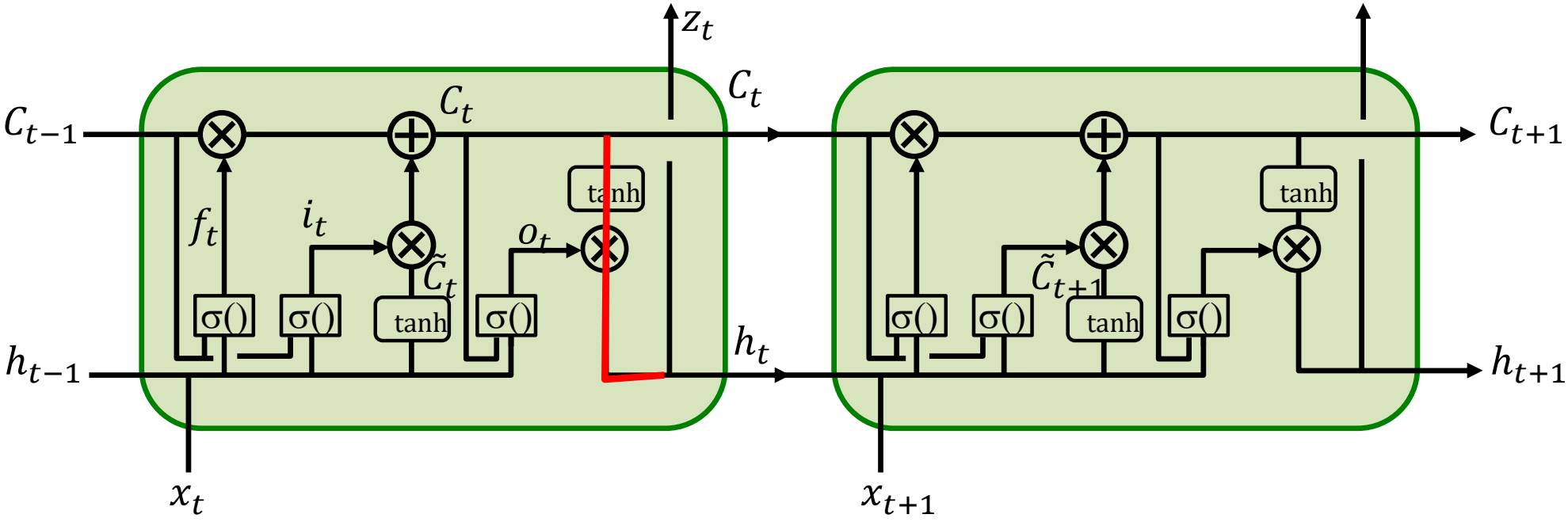
- Identical to training regular RNNs with one difference
  - Commonality: Define a sequence divergence and backpropagate its derivative through time
- Difference: Instead of backpropagating gradients through an RNN unit, we will backpropagate through an LSTM cell

Backpropagation rules: Backward



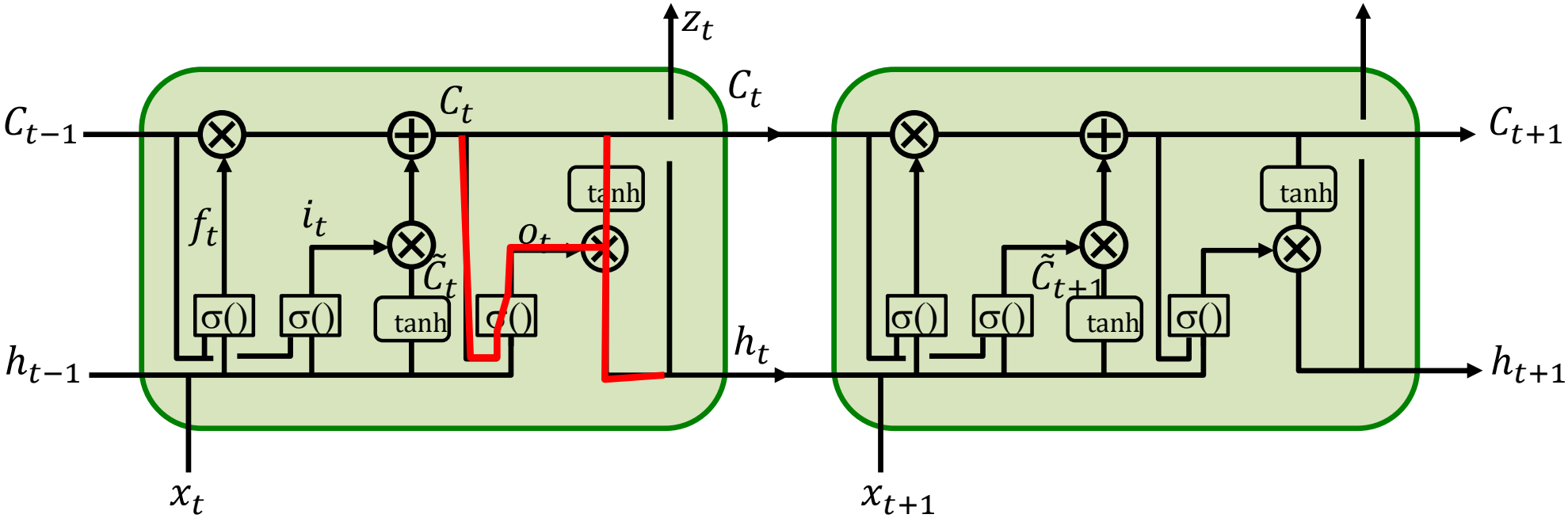
$$\nabla_{C_t} Div =$$

Backpropagation rules: Backward



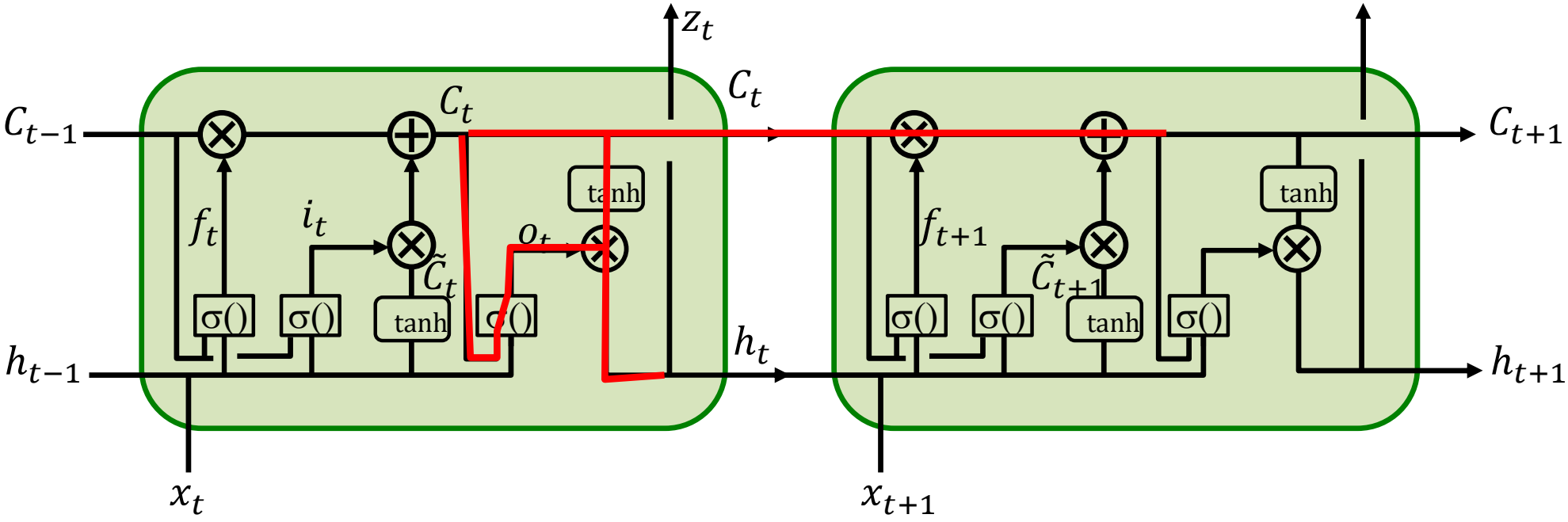
$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ o_t \circ tanh'(.)$$

Backpropagation rules: Backward



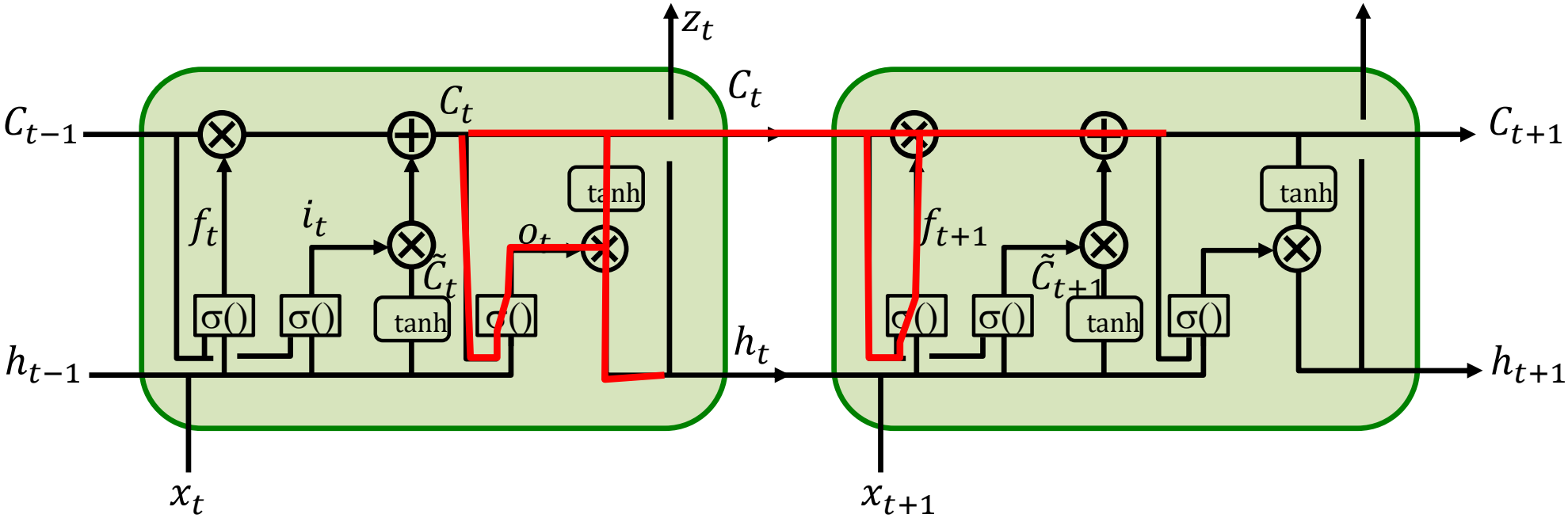
$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ tanh'(.) + tanh(.) \circ \sigma'(.) W_{Co})$$

Backpropagation rules: Backward



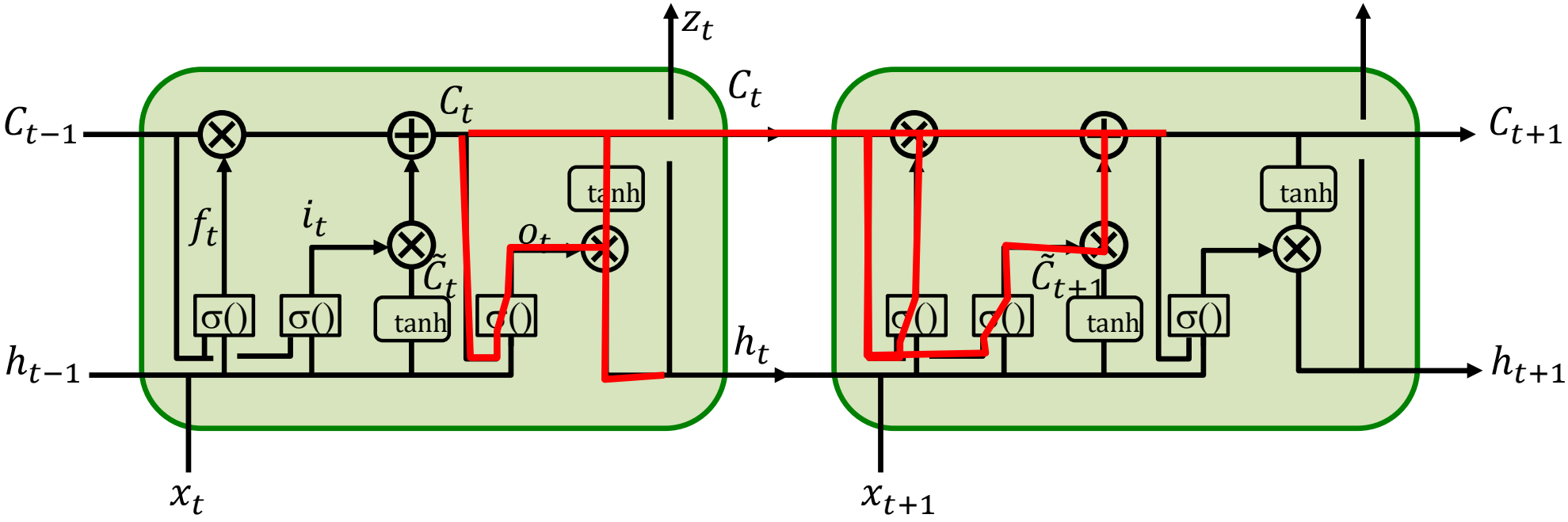
$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ tanh'(.) + tanh(.) \circ \sigma'(.) W_{Co}) +$$
$$\nabla_{C_{t+1}} Div \circ f_{t+1} +$$

Backpropagation rules: Backward



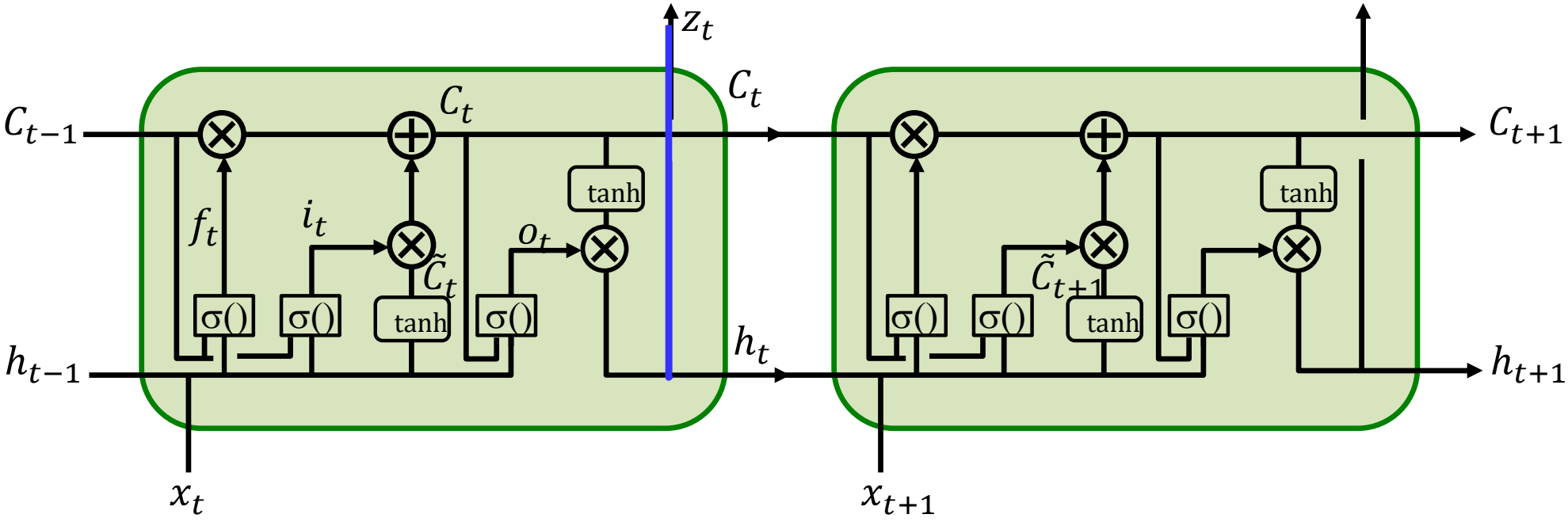
$$\nabla_{C_t} Div = \nabla_{h_t} Div \circ (o_t \circ tanh'(.) + tanh(.) \circ \sigma'(.) W_{Co}) +$$
$$\nabla_{C_{t+1}} Div \circ \left(f_{t+1} + C_t \circ \sigma'(.) W_{Cf}\right)$$

Backpropagation rules: Backward



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

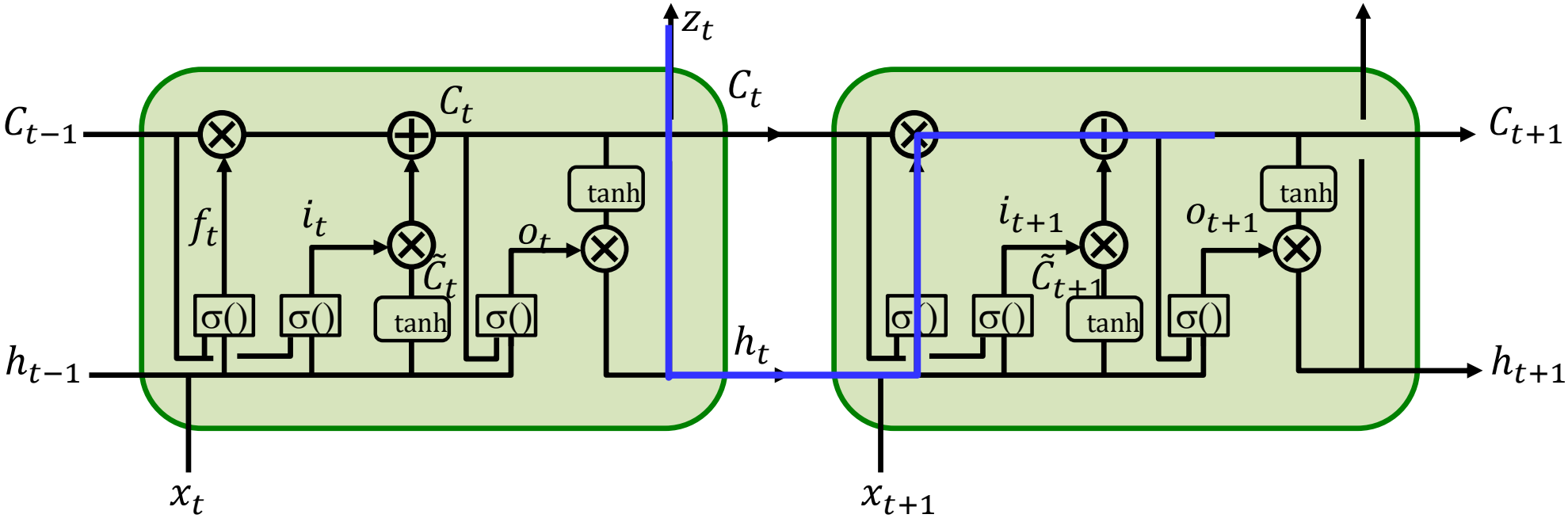
Backpropagation rules: Backward



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

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

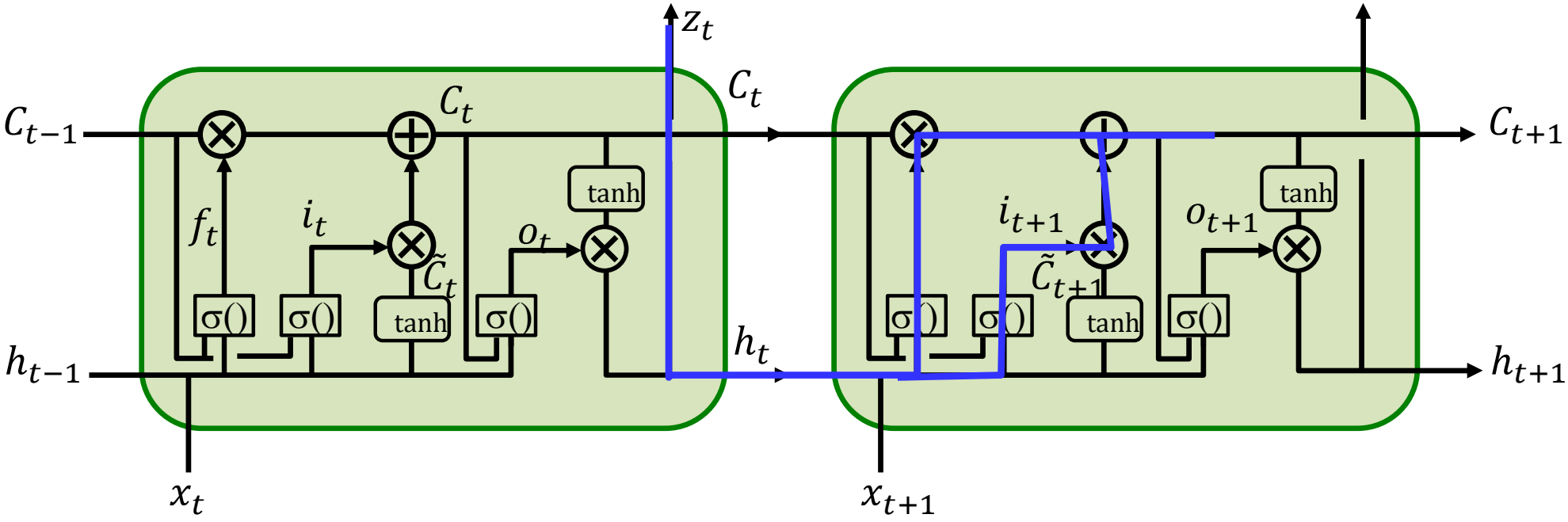
Backpropagation rules: Backward



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

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

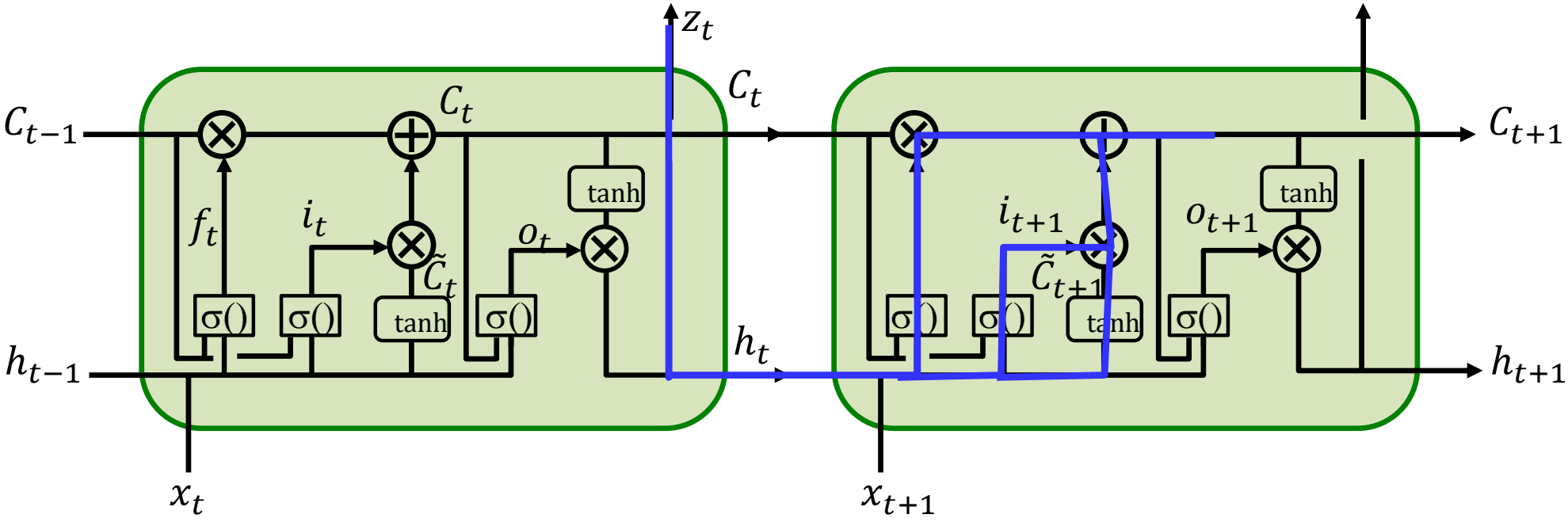
Backpropagation rules: Backward



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

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

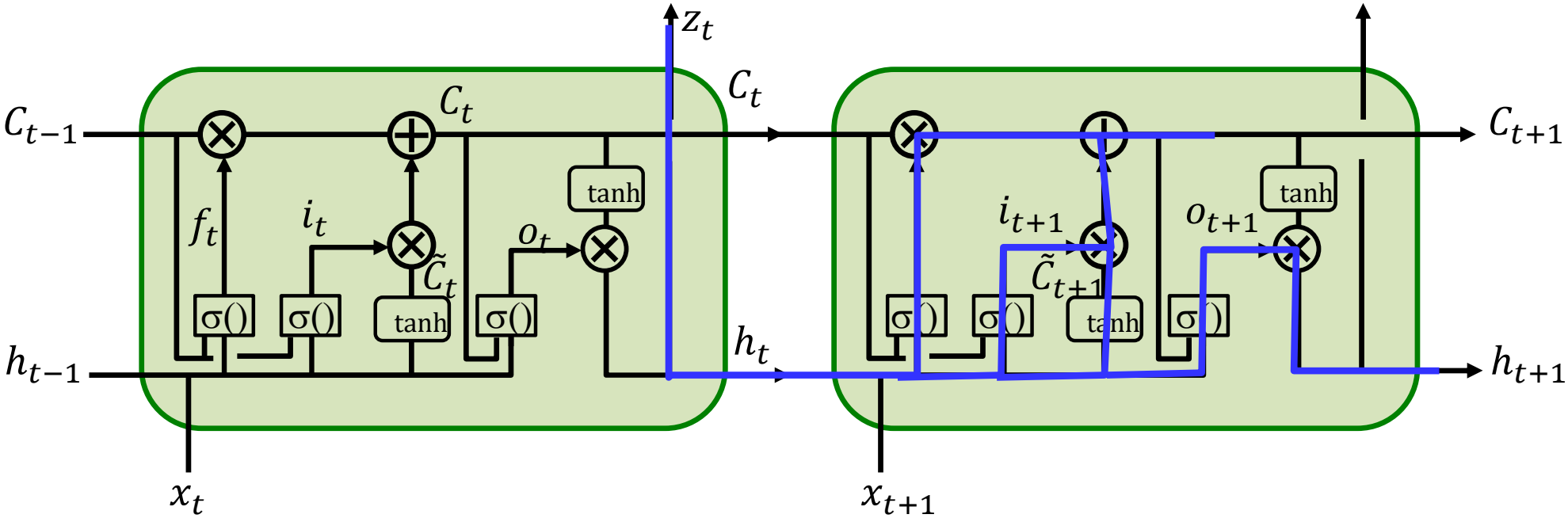
Backpropagation rules: Backward



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

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

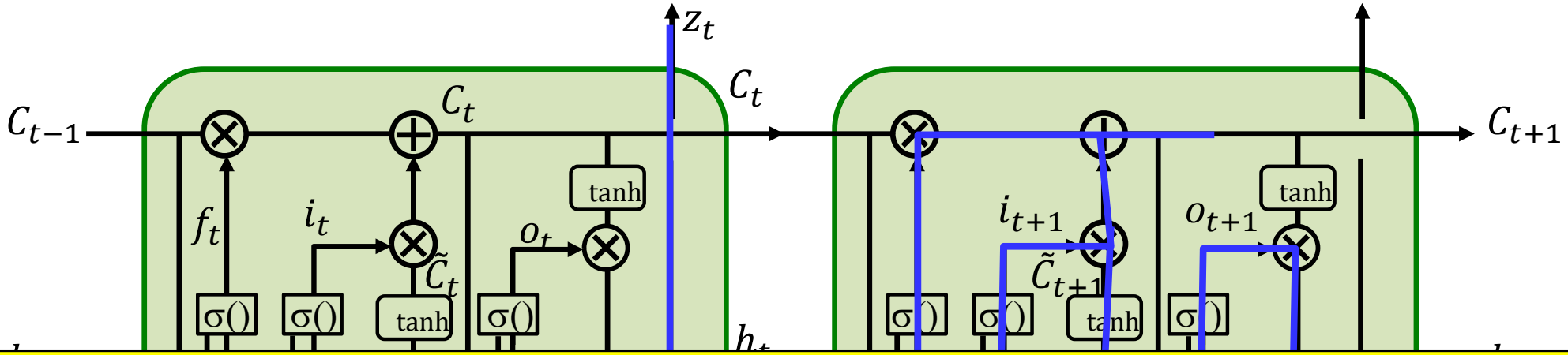
Backpropagation rules: Backward



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

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

Backpropagation rules: Backward



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

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

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

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
```

Assuming a peephole connection into the tanh, which is not standard

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
    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)

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

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

$\# \text{ Next invert } z_c = W_{cc}C + W_{ch}h + W_{cx}x + b_c$

$dC \mathrel{+}= dz_c W_{cc}$

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

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

$dW_{cc} = C\ dz_c$

$dW_{ch} = h\ dz_c$

$dW_{cx} = x\ dz_c$

$db_c = dz_c$

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

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

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

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

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

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

$dW_{ic} = C\ dz_i$

$dW_{ih} = h\ dz_i$

$dW_{ix} = x\ dz_i$

$db_i = dz_i$

LSTM cell backward (continued)

Next invert $f = \text{sigmoid}(z_f)$

$dz_f = df \circ \text{sigmoid}(z_f)^T \circ (1-\text{sigmoid}(z_f))^T$

Finally invert $z_f = W_{fc}C + W_{fh}h + W_{fx}x + b_f$

$dC \mathrel{+}= dz_f\, W_{fc}$

$dh \mathrel{+}= dz_f\, W_{fh}$

$dx \mathrel{+}= dz_f\, W_{fx}$

$dW_{fc} = C\, dz_f$

$dW_{fh} = h\, dz_f$

$dW_{fx} = x\, dz_f$

$db_f = dz_f$

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) )
```

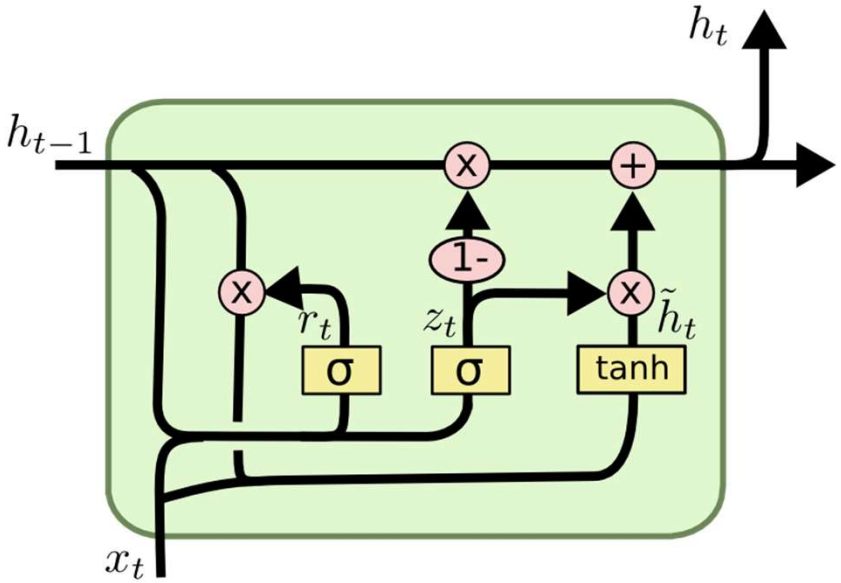
LSTM network backward

```
# 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]
```

Gated Recurrent Units: Lets simplify the LSTM



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

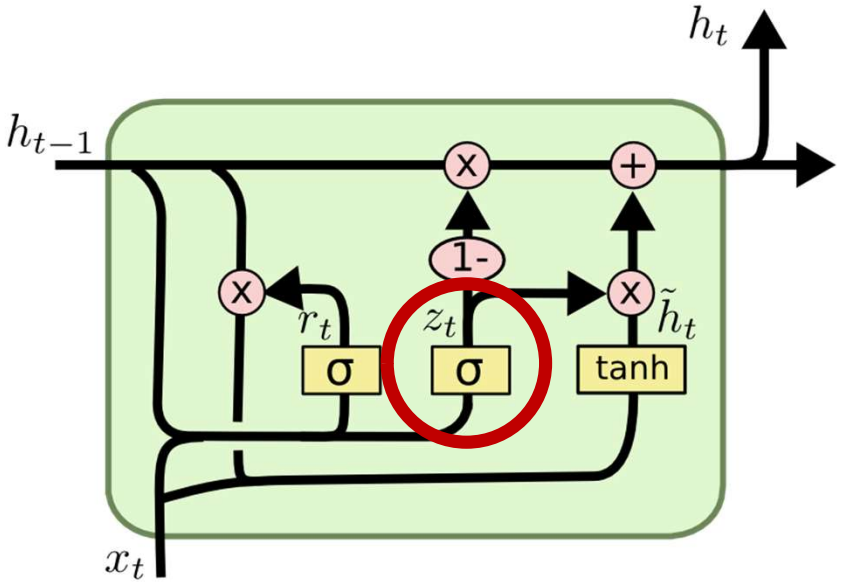
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

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

Gated Recurrent Units: Lets simplify the LSTM



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

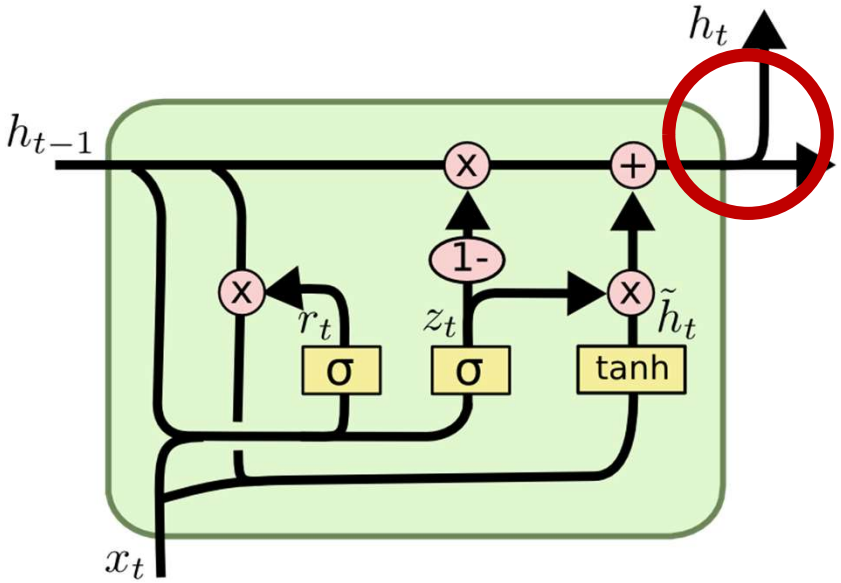
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

- Combine forget and input gates
  - If 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



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

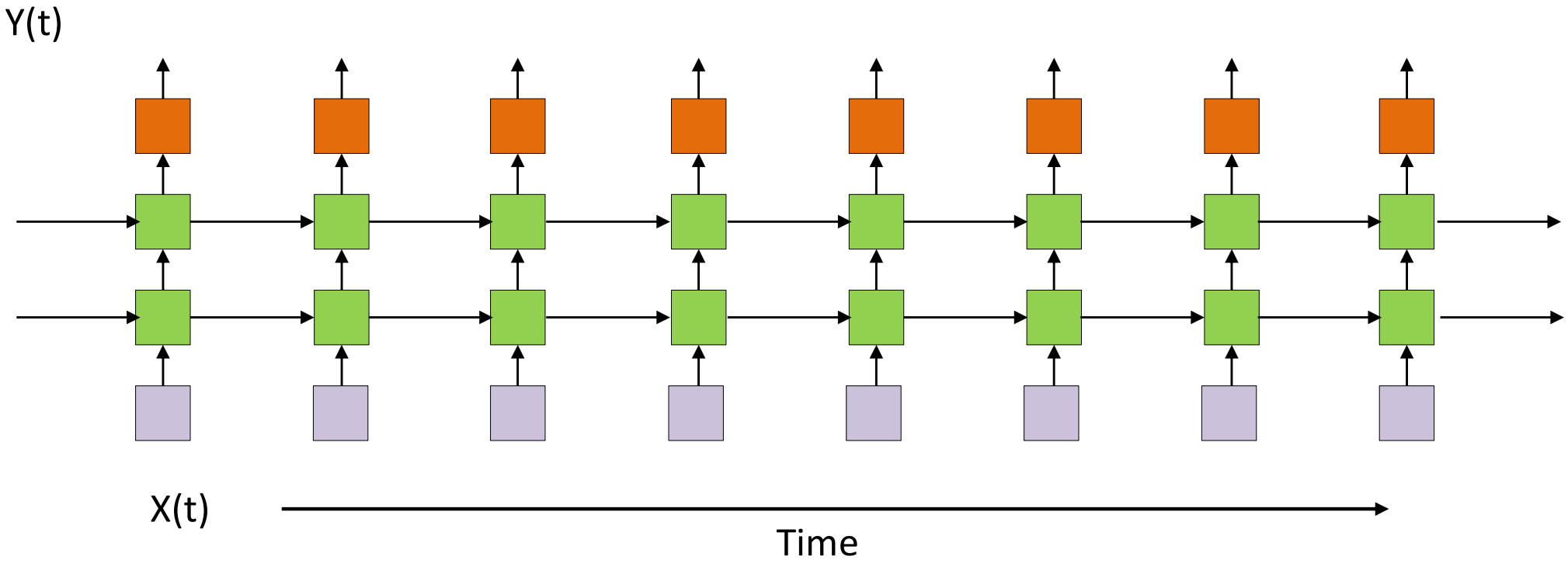
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

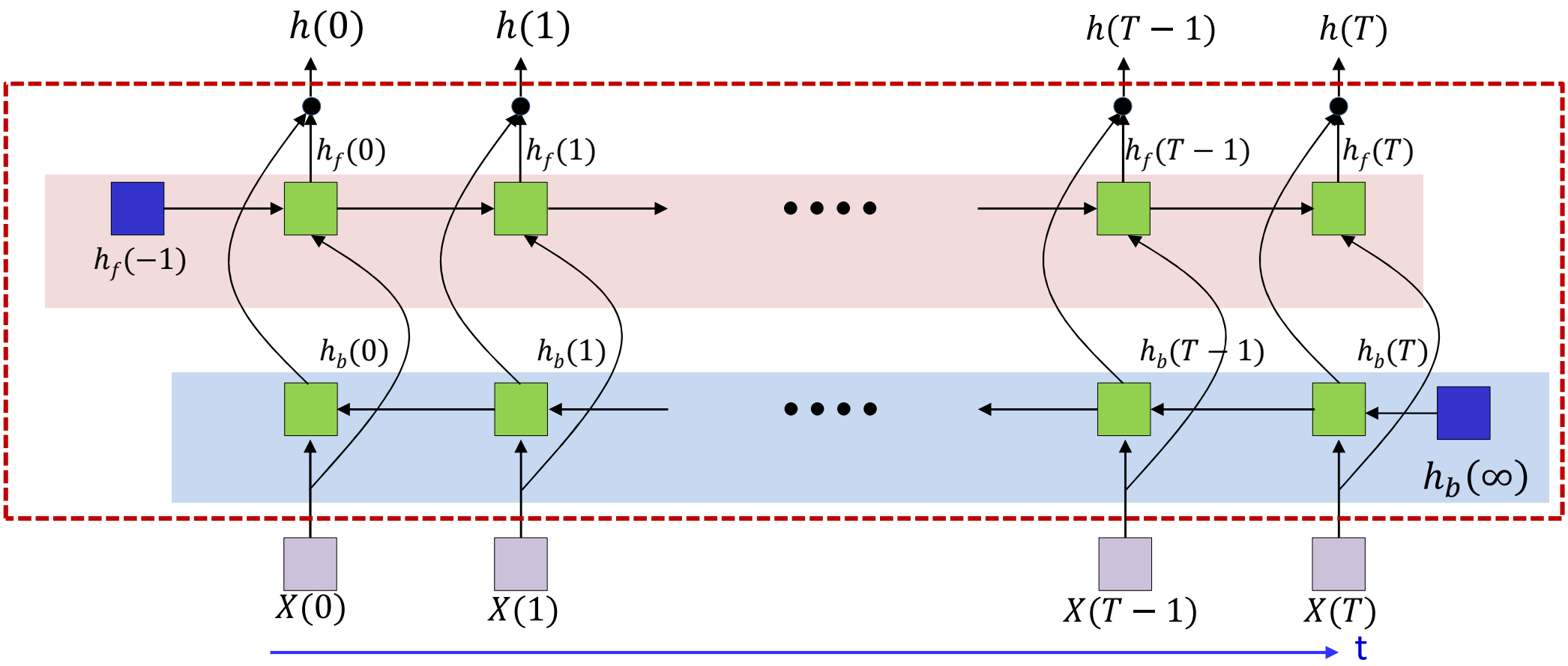
- Don't bother to separately maintain compressed and regular memories
  - Pointless computation!
  - Redundant representation

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 next couple of classes..