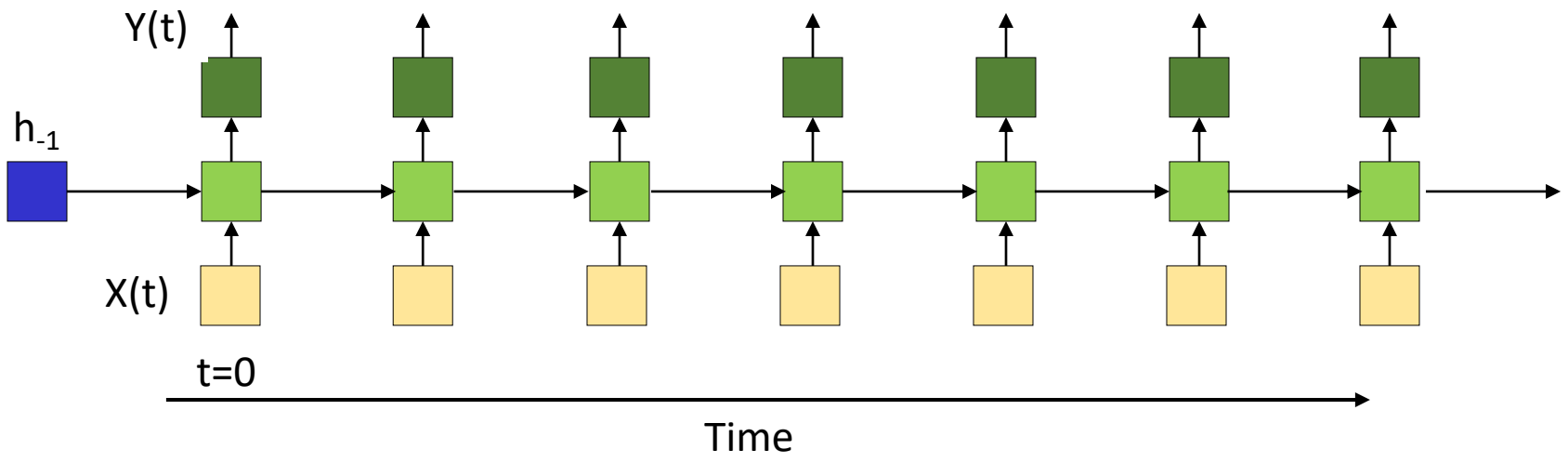
Recurrent Neural Networks: Stability analysis and LSTMs

M. Soleymani
Sharif University of Technology
Spring 2023

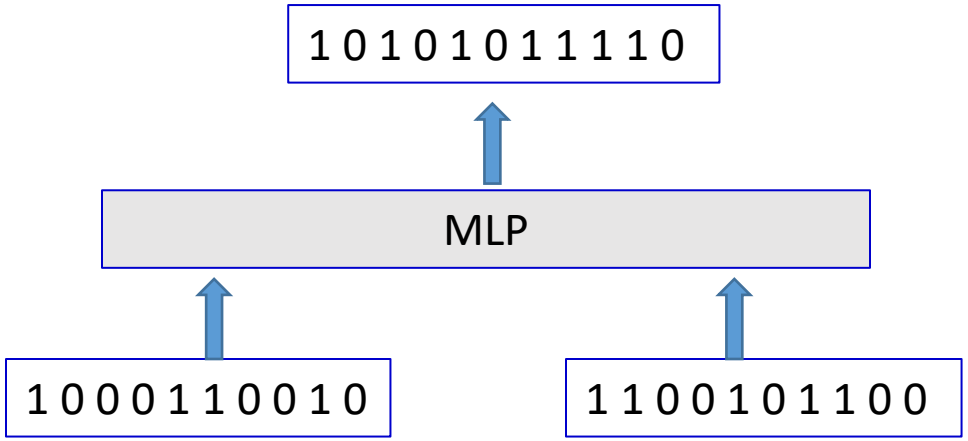
Most slides have been adopted from Bhiksha Raj, 11-785, CMU
and some from Fei Fei Li and colleagues lectures, cs231n, Stanford

Story so far



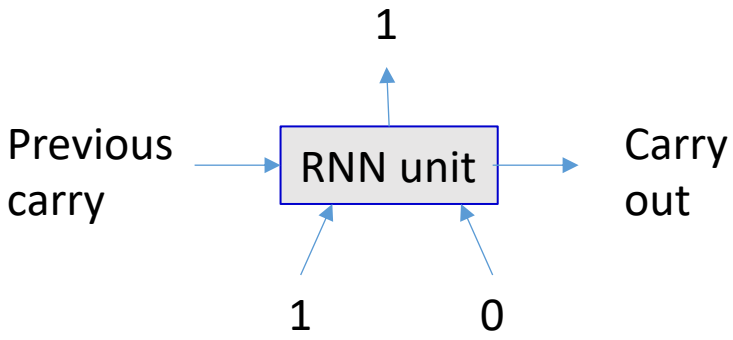
- ***Recurrent structures*** are good for analyzing time series data with ***long-term*** dependence on the past
  - These are ***recurrent*** neural networks

Recurrent structures can do what static structures cannot



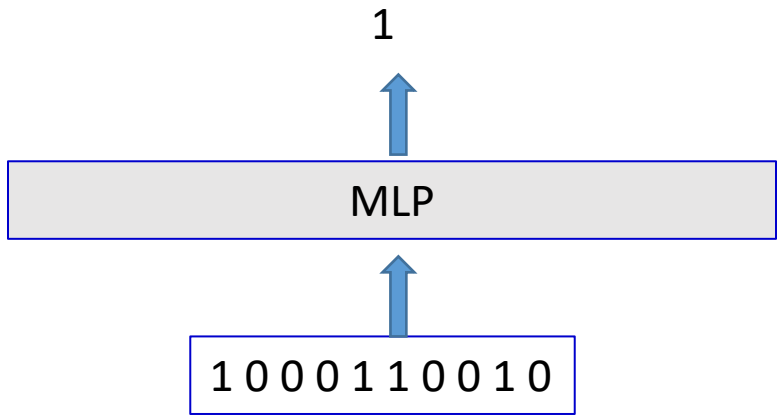
- The addition problem: Add two N-bit numbers to produce a N+1-bit number
  - Input is binary
  - Will require large number of training instances
    - Output must be specified for every pair of inputs
    - Weights that generalize will make errors
  - Network trained for N-bit numbers will not work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

MLPs vs RNNs



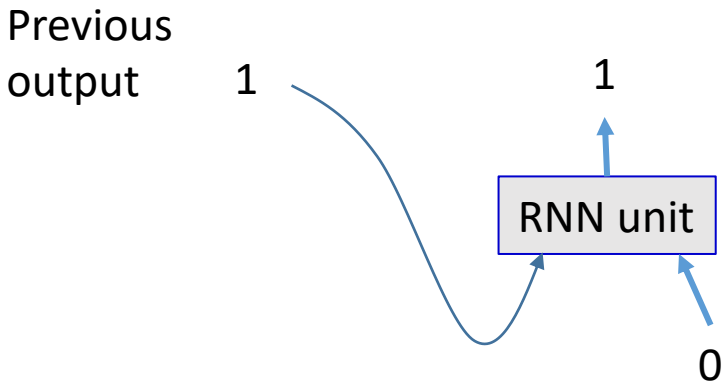
- The addition problem: Add two N-bit numbers to produce a N+1-bit number
- **RNN solution:** Very simple, can add two numbers of any size

MLP: The parity problem



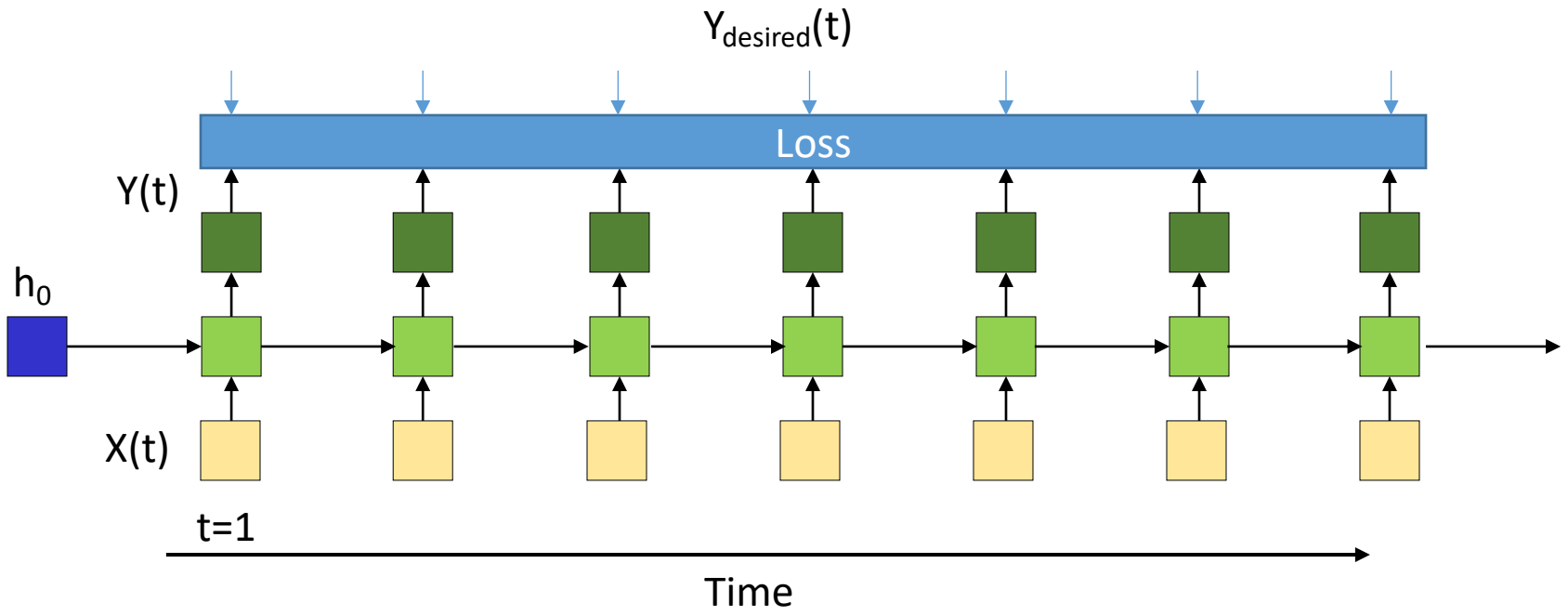
- Is the number of “ones” even or odd
- Network must be complex to capture all patterns
  - XOR network, quite complex
  - Fixed input size

RNN: The parity problem



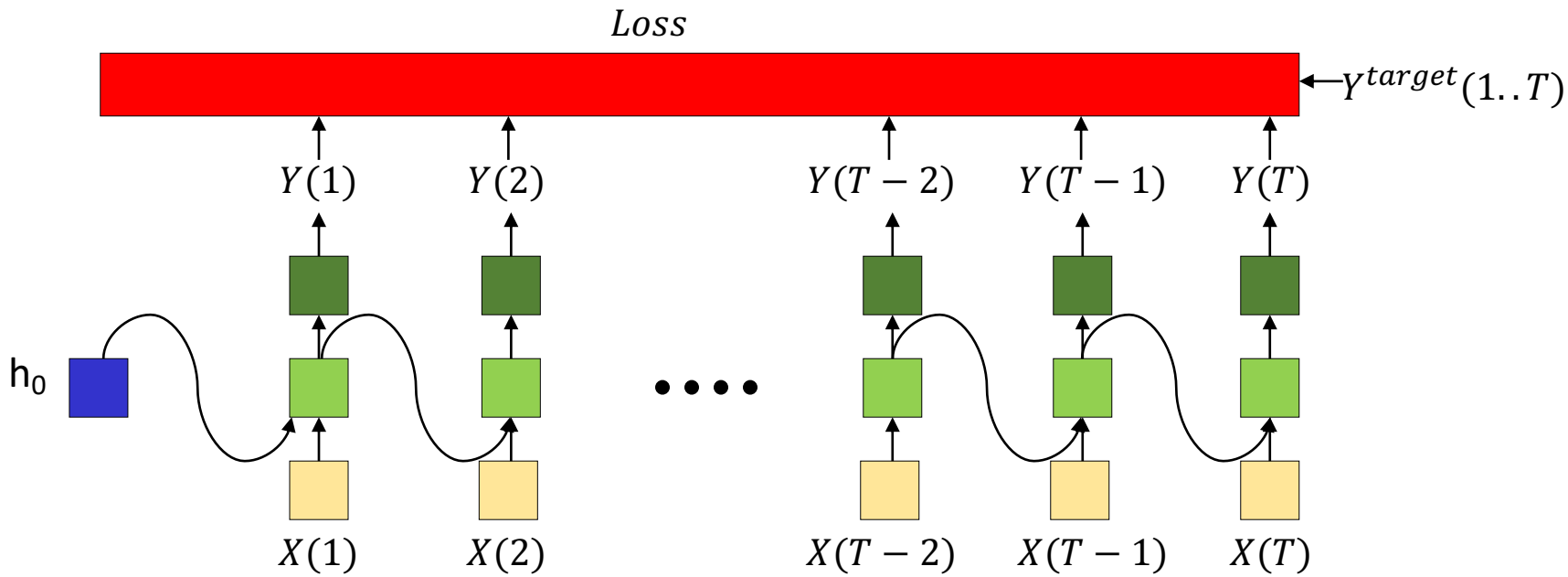
- Trivial solution
- Generalizes to input of any size

Story so far



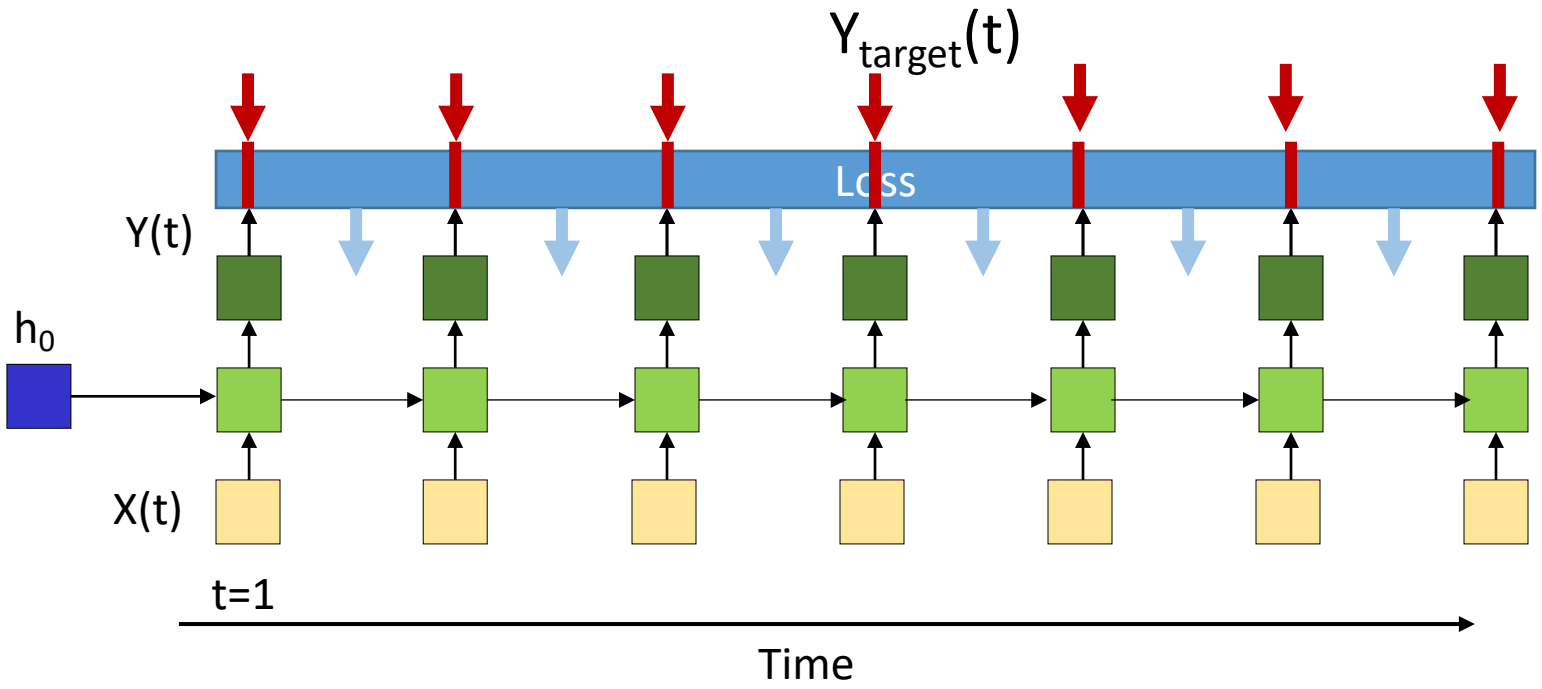
- Recurrent structures can be trained by minimizing the loss between the ***sequence*** of outputs and the ***sequence*** of desired outputs
  - Through gradient descent and backpropagation

Back Propagation Through Time



- The loss computed is between the *sequence of outputs* by the network and the *desired sequence of outputs*
- This is *not* always the sum of the divergences at individual times

Time-synchronous recurrence



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

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

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

Long-term behavior of RNNs

- In linear systems, long-term behavior depends entirely on the eigenvalues of the hidden-layer 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

“BIBO” Stability

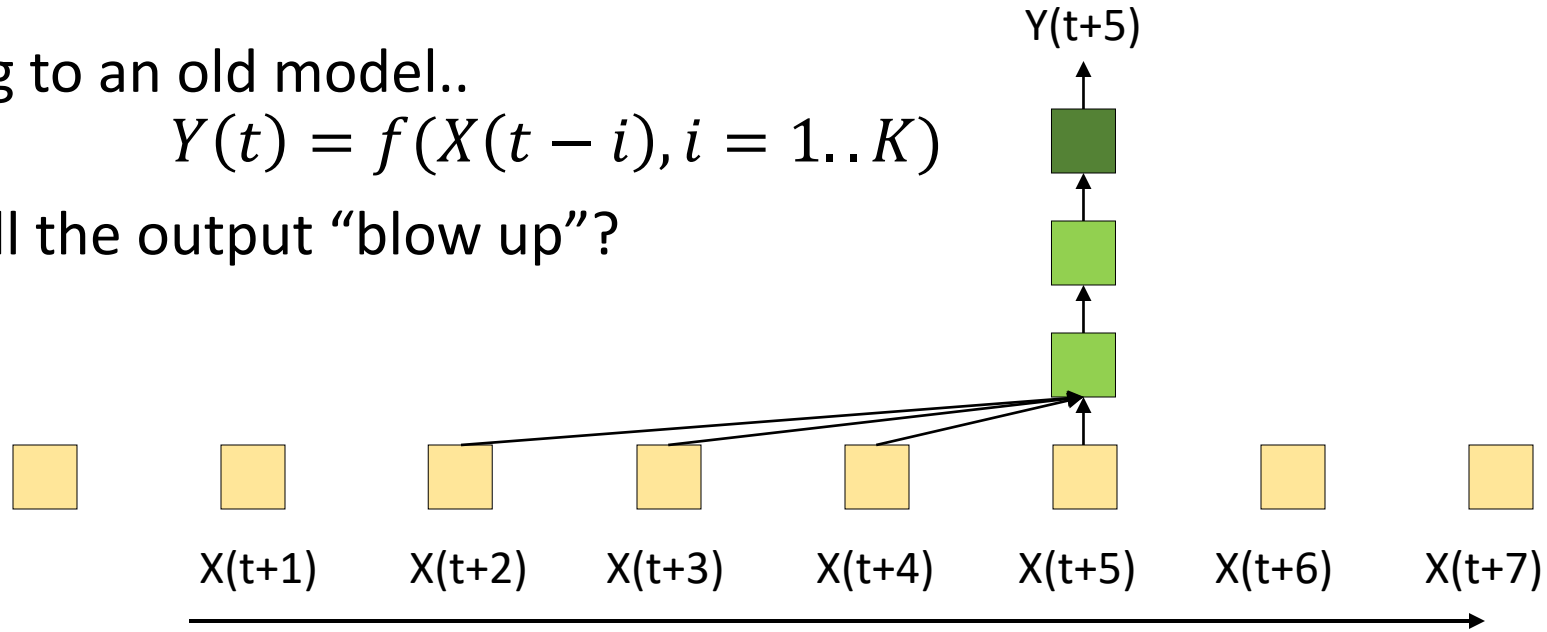
- “Bounded Input Bounded Output” stability
  - This is a highly desirable characteristic

"BIBO" Stability

- Returning to an old model..

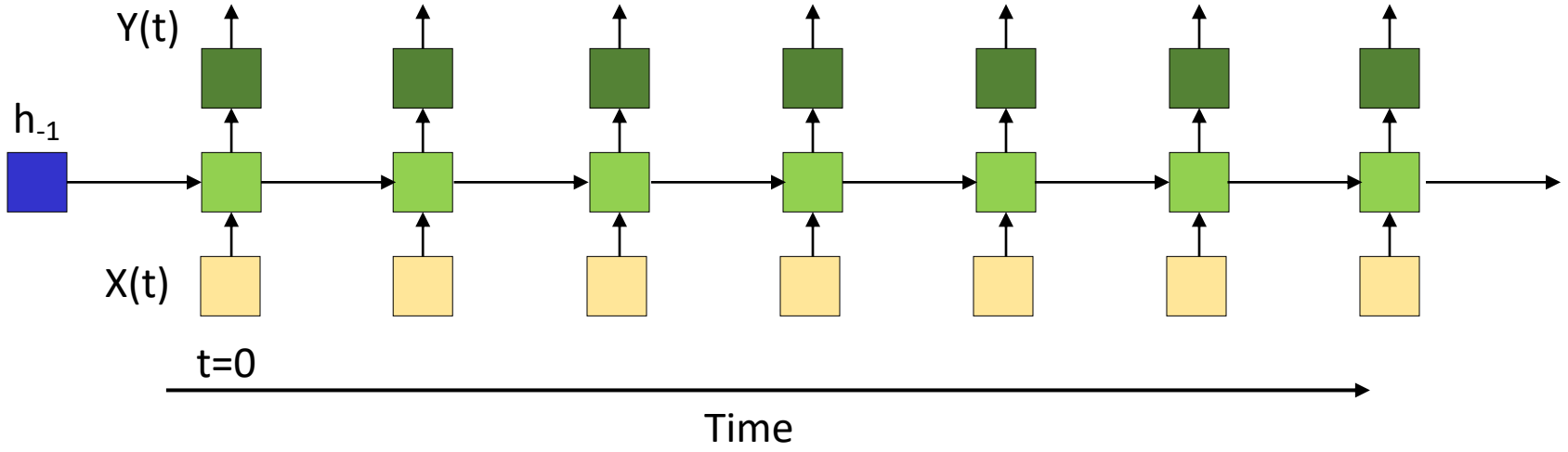
$$Y(t) = f(X(t-i), i = 1..K)$$

- When will the output "blow up"?



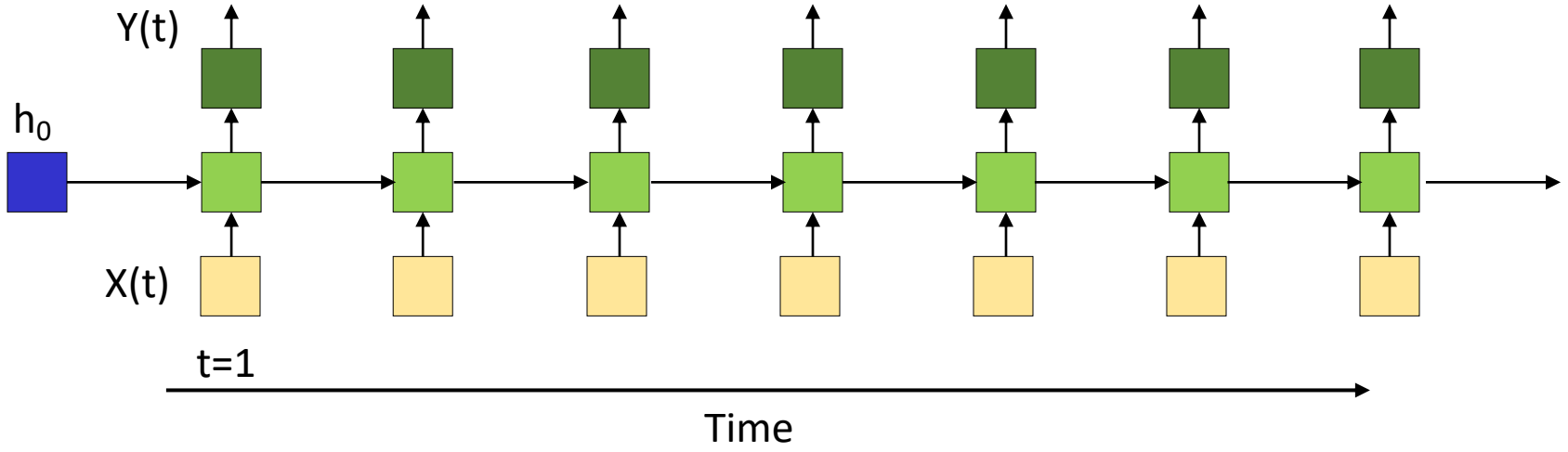
- Time-delay structures have bounded output if
  - The function $f(.)$ has bounded output for bounded input
    - Which is true of almost every activation function
  - $X(t)$ is bounded

Is this BIBO?



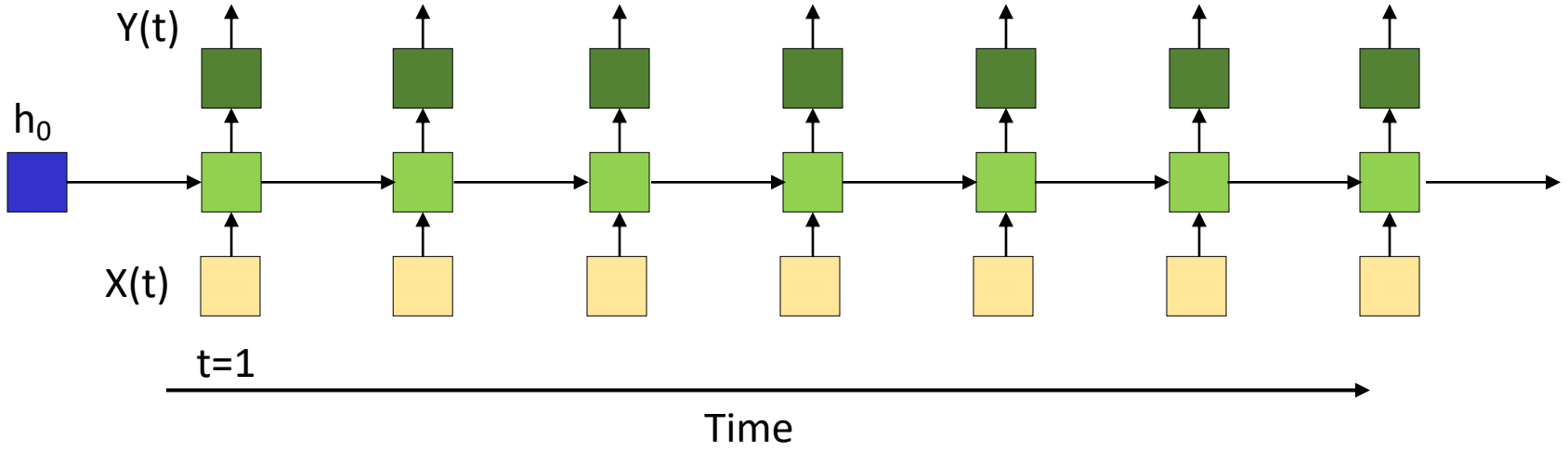
- Will RNN necessarily be BIBO?

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?

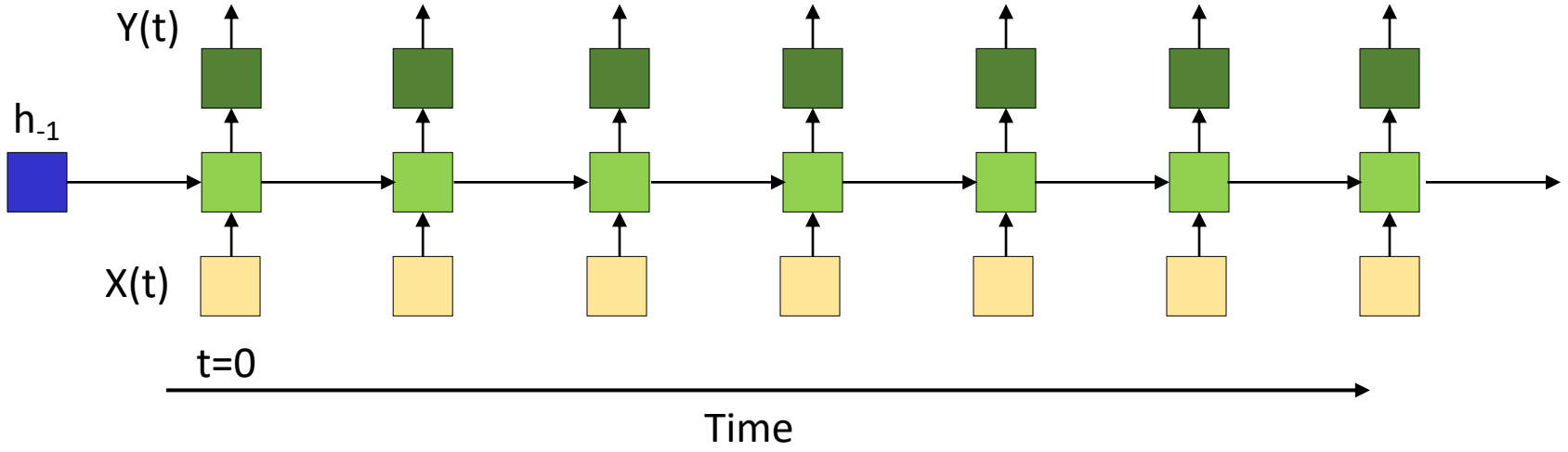
Analyzing recurrence



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

Analyzing Recursion

Streetlight effect



- Easier to analyze *linear* systems
  - Will attempt to extrapolate to non-linear systems subsequently
- All activations are identity functions
  - $z_k = W_h h_{k-1} + W_x x_k, \qquad h_k = z_k$

Linear systems

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

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

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

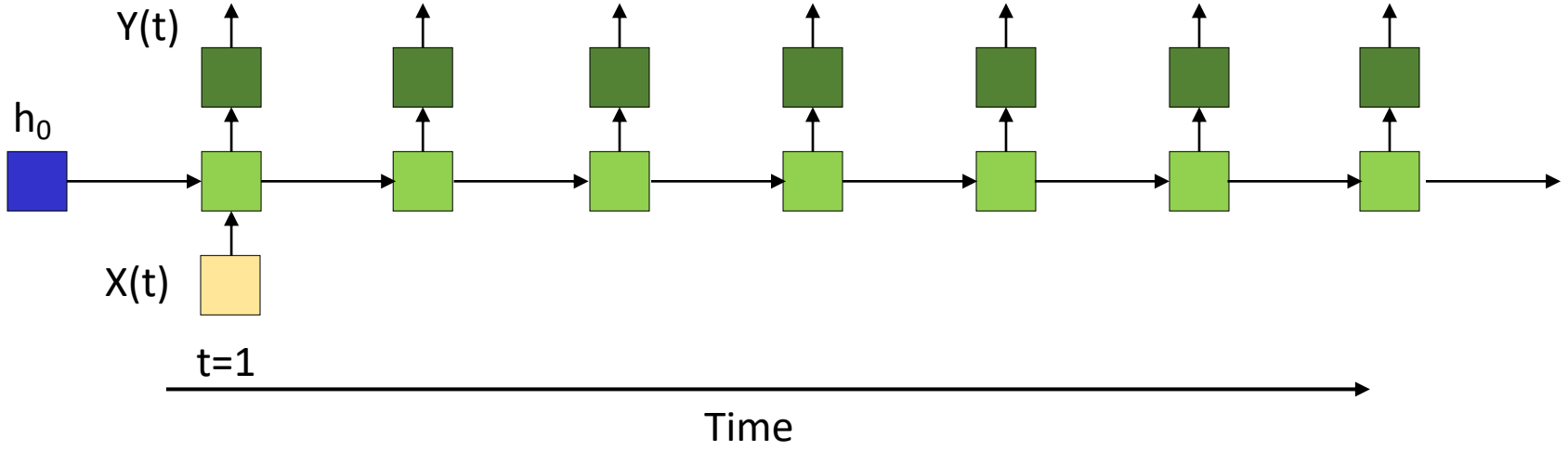
Linear systems

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

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

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

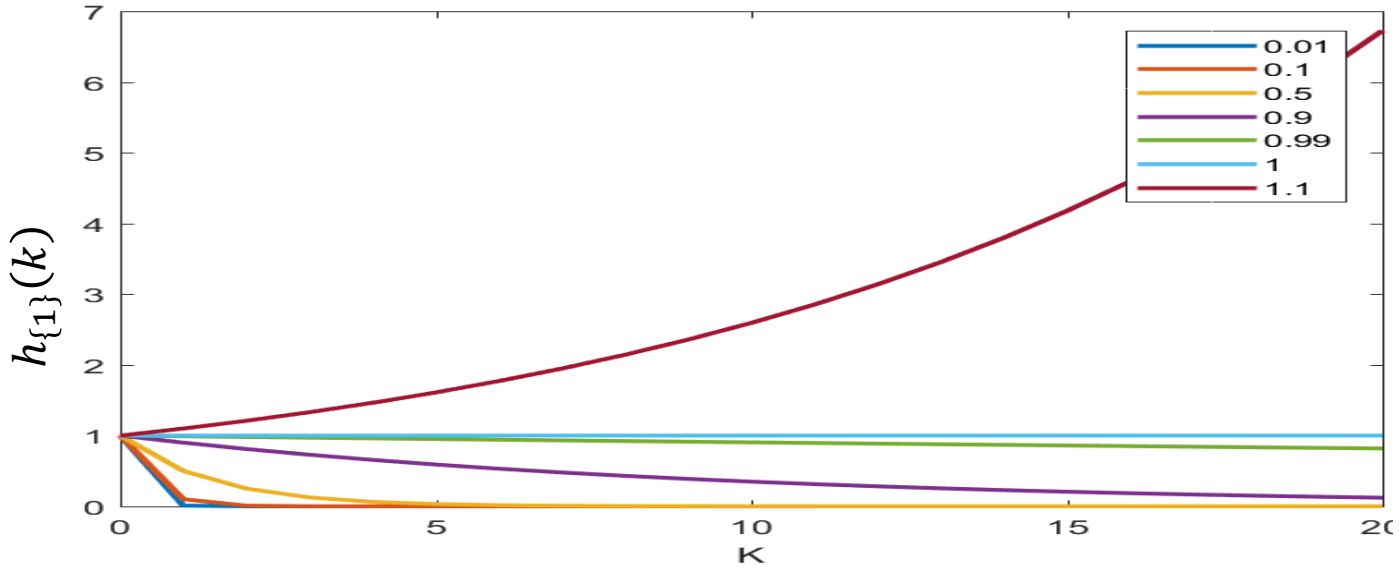
Streetlight effect



- Sufficient to analyze the response to a single input at $t = 1$
  - Principle of superposition in linear systems

Linear recursions

- Consider simple, scalar, linear recursion (note change of notation)
  - $h(t) = W_h h(t-1) + W_x x(t)$
  - $h_{\{1\}}(t) = W_h^t W_x x(1)$
    - Response to a single input appearing at 1



Linear recursions: Vector version

- Vector linear recursion (note change of notation)
  - $h(t) = W_h h(t-1) + W_x x(t)$
  - $h_{\{1\}}(t) = W_h^t W_x x(1)$
    - Length of response vector to a single input appearing at 1 is $|h_{\{1\}}(t)|$

- We can write $W_h = U\Lambda U^{-1}$
  - $W_h u_i = \lambda_i u_i$
  - For any vector v we can write
    - $v = a_1 u_1 + a_2 u_2 + \cdots + a_n u_n$
    - $W_h v = a_1 \lambda_1 u_1 + a_2 \lambda_2 u_2 + \cdots + a_n \lambda_n u_n$
    - $W_h^t v = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots + a_n \lambda_n^t u_n$
  - $\lim_{t\to\infty} W_h^t v = a_m \lambda_m^t u_m$ where $m = \underset{j}{\operatorname{argmax}}\, \lambda_j$

Linear recursions: Vector version

- Vector linear recursion (note change of notation)
  - $h(t) = W_h h(t-1) + W_x x(t)$
  - $h_{\{1\}}(t) = W_h^t W_x x(1)$
    - Length of response vector to a single input appearing at 1 is $|h_{\{1\}}(t)|$

- We can write $W_h = U\Lambda U^{-1}$
  - $W_h u_i = \lambda_i u_i$

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 hidden-layer weight matrix

    - $W_h v = a_1\lambda_1 u_1 + a_2\lambda_2 u_2 + \cdots + a_n\lambda_n u_n$
    - $W_h^t v = a_1\lambda_1^t u_1 + a_2\lambda_2^t u_2 + \cdots + a_n\lambda_n^t u_n$
  - $\lim_{t\to\infty} W_h^t v = a_m\lambda_m^t u_m$ where $m = \underset{j}{\operatorname{argmax}}\, \lambda_j$

Linear recursions: Vector version

- Vector linear recursion (note change of notation)
  - $h(t) = W_h h(t-1) + W_x x(t)$
  - $h_{\{1\}}(t) = W_h^t W_x x(1)$
    - Length of response vector to a single input appearing at 1 is $|h_{\{1\}}(t)|$

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 of the hidden-layer weight matrix

Unless it has no component along the eigen vector corresponding to the largest eigen value. In that case it will grow according to the *second* largest Eigen value.. And so on..

  - For any vector v we can write
    - $v = a_1 u_1 + a_2 u_2 + \cdots + a_n u_n$
    - $W_h v = a_1 \lambda_1 u_1 + a_2 \lambda_2 u_2 + \cdots + a_n \lambda_n u_n$
    - $W_h^t v = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots + a_n \lambda_n^t u_n$
  - $\lim_{t \to \infty} W_h^t v = a_m \lambda_m^t u_m$ where $m = \operatorname{argmax}_j \lambda_j$

Linear recursions: Vector version

- Vector linear recursion (note change of notation)

If $|\lambda_{max}| > 1$ it will blow up, otherwise it will contract and shrink to 0 rapidly

  - Length of response vector to a single input appearing at 1 is $|h_{\{1\}}(t)|$

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 of the hidden-layer weight matrix

Unless it has no component along the eigen vector corresponding to the largest eigen value. In that case it will grow according to the *second* largest Eigen value..And so on..

  - For any vector v we can write
    - $v = a_1 u_1 + a_2 u_2 + \cdots + a_n u_n$
    - $W_h v = a_1 \lambda_1 u_1 + a_2 \lambda_2 u_2 + \cdots + a_n \lambda_n u_n$
    - $W_h^t v = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots + a_n \lambda_n^t u_n$
  - $\lim_{t \to \infty} W_h^t v = a_m \lambda_m^t u_m$ where $m = \underset{j}{\operatorname{argmax}} \lambda_j$

Linear recursions: Vector version

What about at middling values of t? It will depend on the other eigen values

If $|\lambda_{max}| > 1$ it will blow up, otherwise it will contract and shrink to 0 rapidly

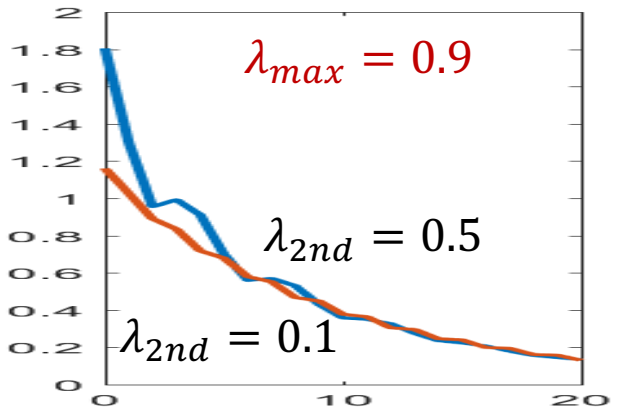
For any input, for large t the length of the hidden vector will expand or contract according to the t th power of the largest eigen value of the hidden-layer weight matrix

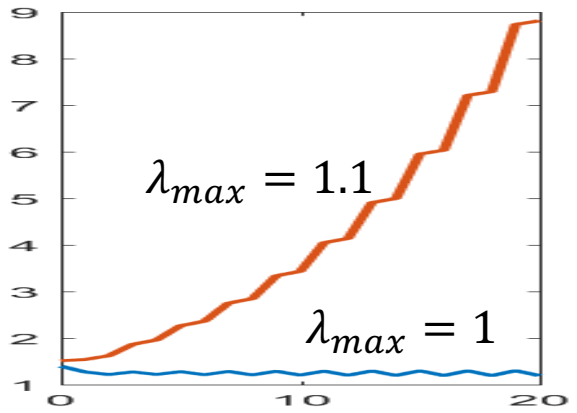
Unless it has no component along the eigen vector corresponding to the largest eigen value. In that case it will grow according to the *second* largest Eigen value.. And so on..

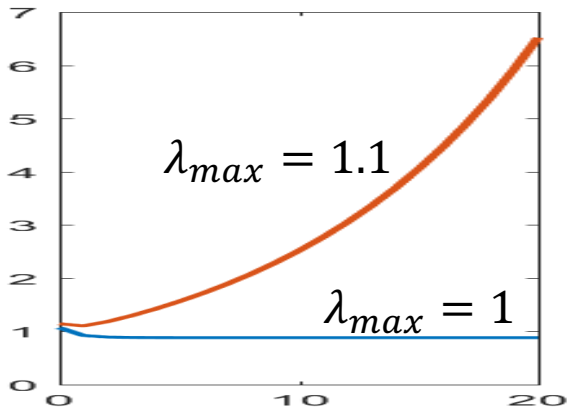
- For any vector v we can write
  - $v = a_1 u_1 + a_2 u_2 + \cdots + a_n u_n$
  - $W_h v = a_1 \lambda_1 u_1 + a_2 \lambda_2 u_2 + \cdots + a_n \lambda_n u_n$
  - $W_h^t v = a_1 \lambda_1^t u_1 + a_2 \lambda_2^t u_2 + \cdots + a_n \lambda_n^t u_n$
- $\lim_{t \to \infty} W_h^t v = a_m \lambda_m^t u_m$ where $m = \operatorname{argmax}_j \lambda_j$

Linear recursions

- Vector linear recursion
  - $h(t) = W_h h(t-1) + W_x x(t)$
  - $h_{\{1\}}(t) = W_h^t W_x x(1)$
    - Response to a single input [1 1 1 1] appearing at 1







Complex Eigenvalues

Lesson..

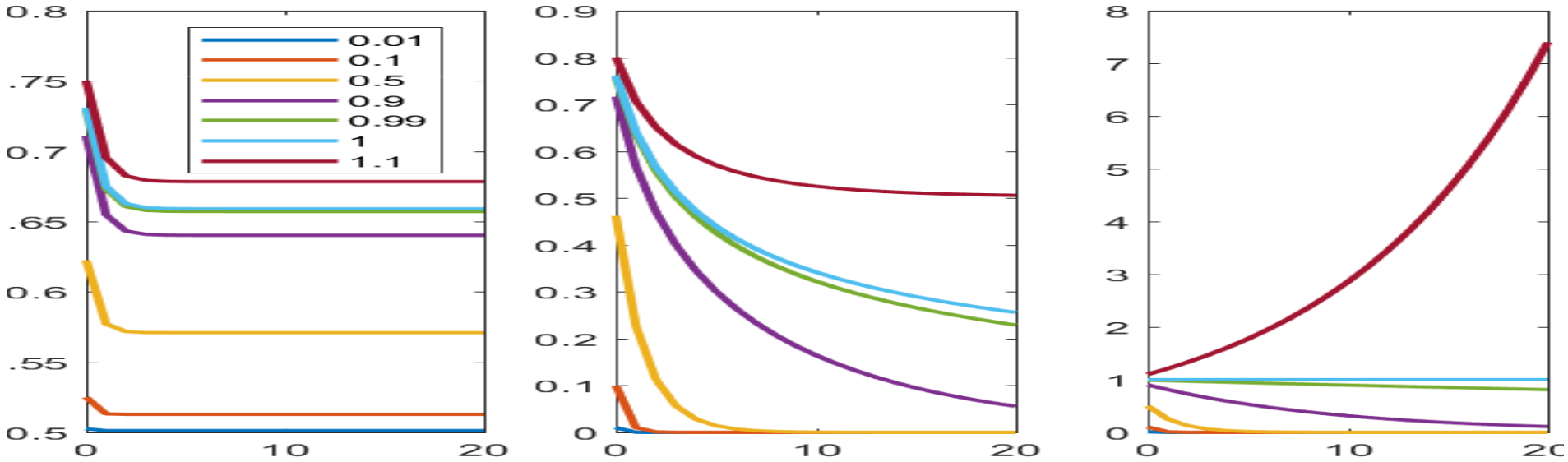
- In linear systems, long-term behavior depends entirely on the eigenvalues of the hidden-layer weights matrix
  - If the largest Eigen value is greater than 1, the system will “blow up”
  - If it is lesser than 1, the response will “vanish” very quickly
  - Complex eigen values cause oscillatory response
    - Which we may or may not want
    - For smooth behavior, must force the weights matrix to have real Eigen values
      - Symmetric weight matrix

How about non-linearities (scalar)

- The behavior of scalar non-linearities

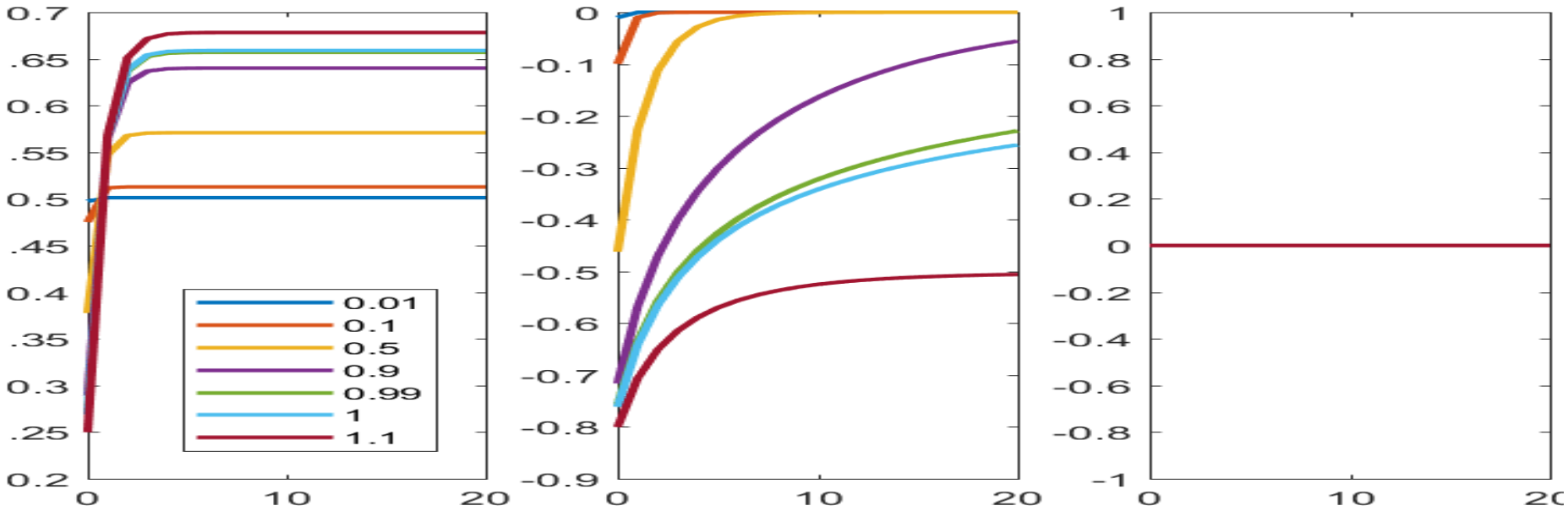
$$h(t) = f(W_h h(t-1) + W_x x(t))$$

- Left: Sigmoid, Middle: Tanh, Right: Relu
  - Sigmoid: Saturates in a limited number of steps, regardless of w_h
  - Tanh: Sensitive to W_h, but eventually saturates
    - "Prefers" weights close to 1.0
  - Relu: Sensitive to W_h, can blow up



How about non-linearities (scalar)

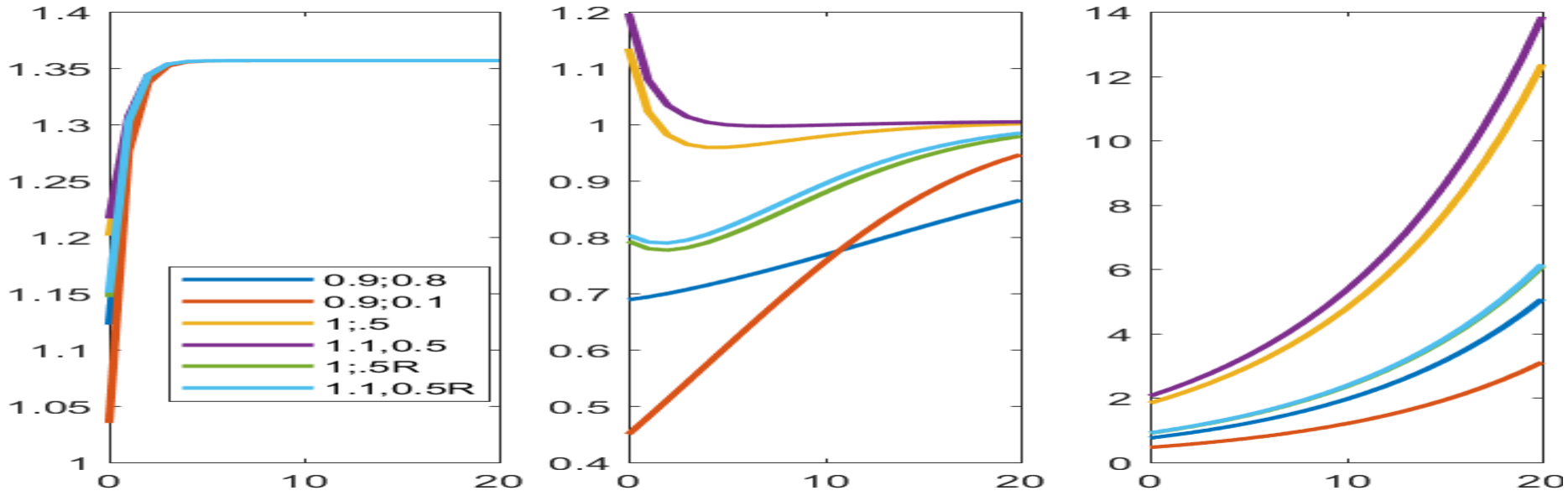
- With a negative start $h(t) = f(w_h h(t-1) + w_x x(t))$
- Left: Sigmoid, Middle: Tanh, Right: Relu
  - Sigmoid: Saturates in a limited number of steps, regardless of w_h
  - Tanh: Sensitive to w_h, but eventually saturates
  - Relu: For negative starts, has no response



Vector Process

- Assuming a uniform unit vector initialization
  - $[1,1,1,\ldots]/\sqrt{N}$
  - Behavior similar to scalar recursion
  - Interestingly, RELU is more prone to blowing up (why?)
- Eigenvalues less than 1.0 retain the most "memory"

$$h(t) = f(W_h h(t-1) + W_x x(t))$$

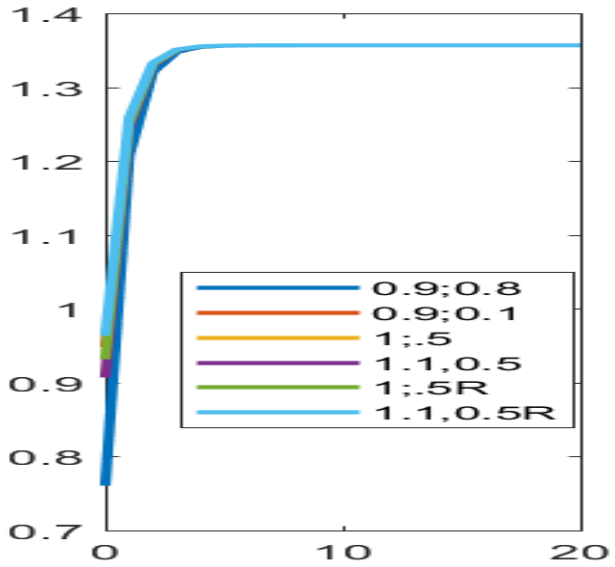


sigmoid tanh relu

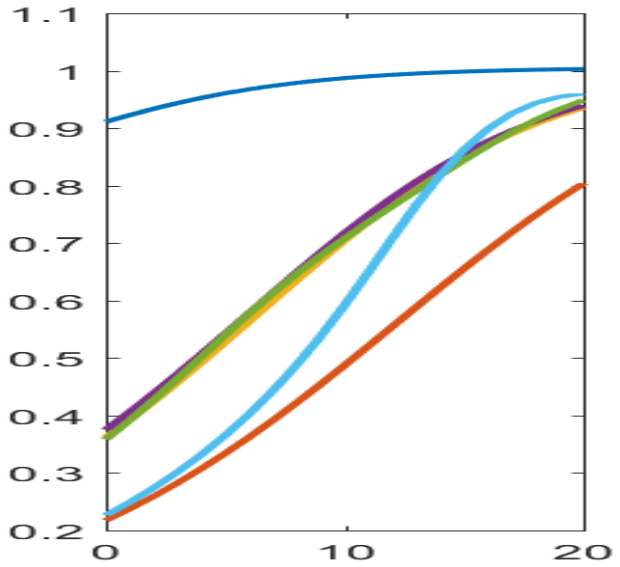
Vector Process

- Assuming a uniform unit vector initialization
  - $[-1, -1, -1, \ldots]/\sqrt{N}$
  - Behavior similar to scalar recursion
  - Interestingly, RELU is more prone to blowing up (why?)

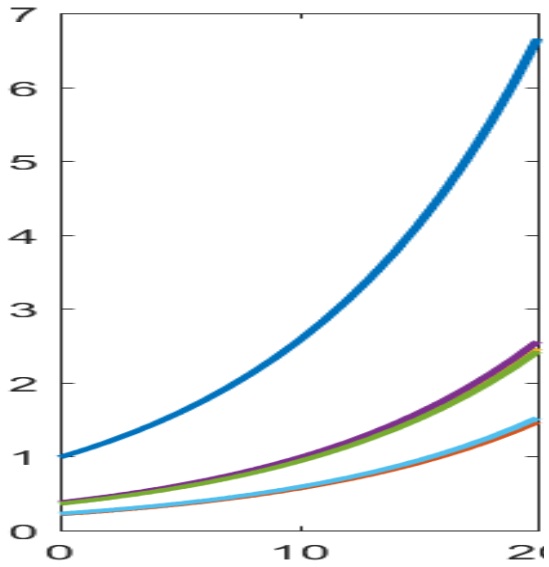
$$h(t) = f(W_h h(t-1) + W_x x(t))$$



sigmoid



tanh



relu

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 its less than one, the response dies down very quickly
- The “memory” of the network also depends on the activation of the hidden units
  - Sigmoid activations saturate and the network becomes unable to retain new information
  - RELUs blow up
  - Tanh activations are the most 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 prediction..
  - They can even simplify problems that are difficult for MLPs

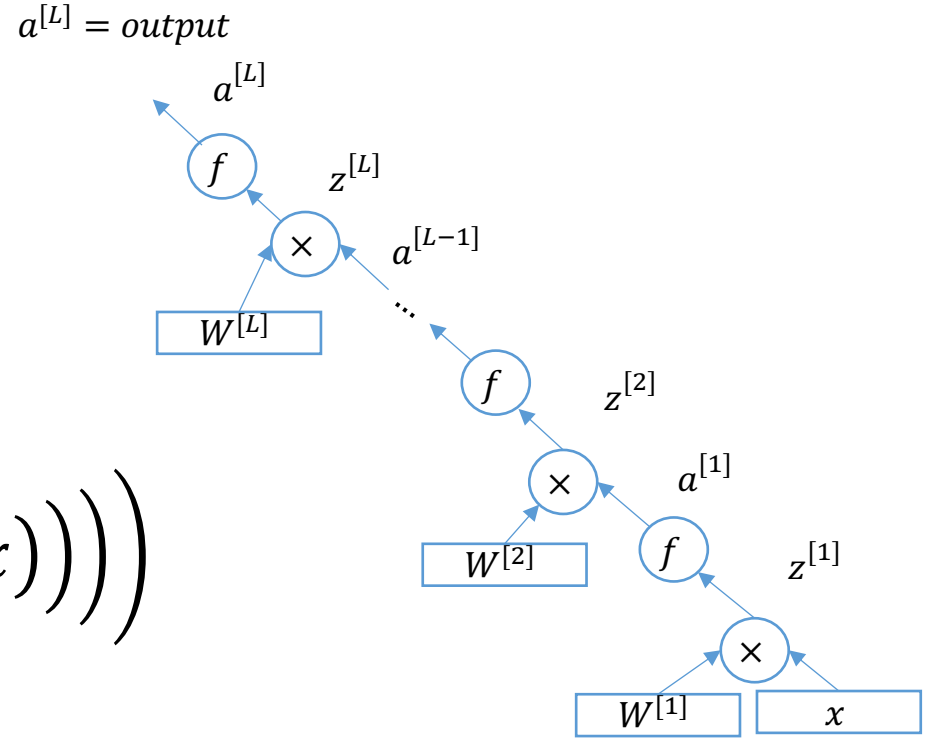
- But the memory isn’t all that great..
  - Also..

The vanishing gradient problem

- 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..

Recap: Training deep networks

$$Output = a^{[L]}$$
$$= f\left(z^{[L]}\right)$$
$$= f\left(W^{[L]}a^{[L-1]}\right)$$
$$= f\left(W^{[L]}f(W^{[L-1]}a^{[L-2]}\right)$$
$$= f\left(W^{[L]}f\left(W^{[L-1]}\ldots f\left(W^{[2]}f\left(W^{[1]}x\right)\right)\right)\right)$$



For convenience, we use the same activation functions for all layers.

However, output layer neurons most commonly do not need activation function (they show class scores or real-valued targets.)

Recap: Training deep networks

- For

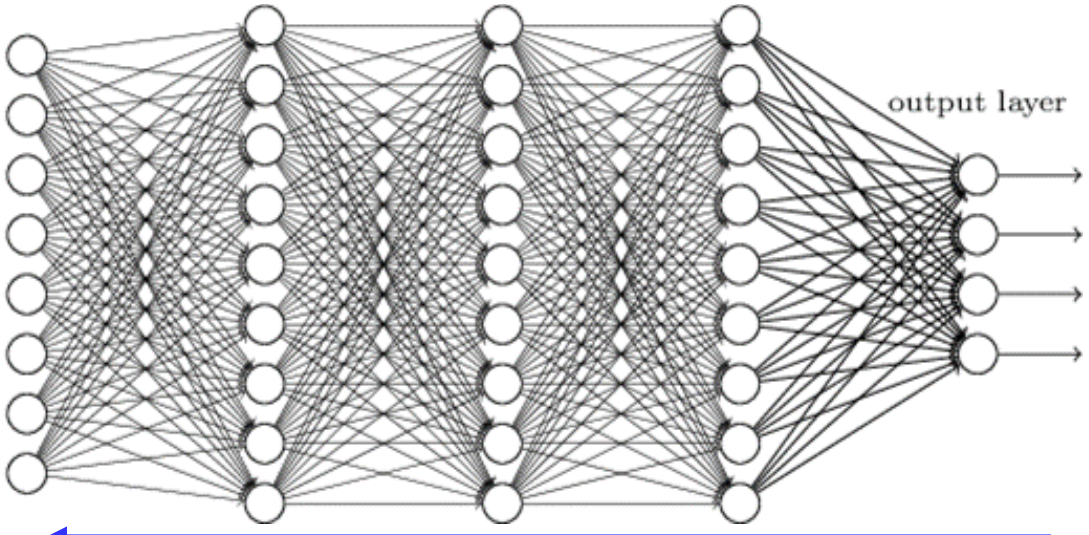
$$Loss(x) = E\left(f^{[L]}\left(W^{[L]}f^{[L-1]}\left(W^{[L-1]}f^{[L-2]}\left(\ldots W^{[1]}x\right)\right)\right)\right)$$

- We get:

$$\nabla_{f^{[l]}}Loss = \nabla_{f^{[L]}}Loss.\nabla f^{[L]}.W^{[L]}.\nabla f^{[L-1]}.W^{[L-1]}\ldots\nabla f^{[l+1]}.W^{[l+1]}$$

- Where
  - $\nabla_{f^{[l]}}Loss$ is the gradient of the error w.r.t the output of the l-th layer of the network
    - Needed to compute the gradient of the error w.r.t $W^{[l]}$
  - $\nabla f^{[l]}$ is *jacobian* of $f^{[l]}$ w.r.t. to its current input
  - All blue terms are matrices

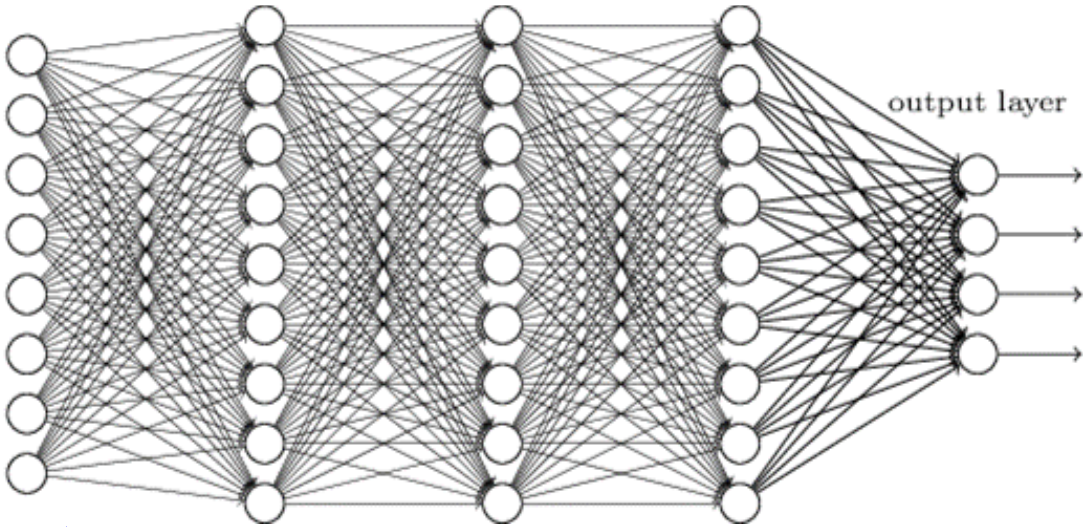
Recap: Gradient problems in deep networks



$$\nabla_{f^{[l]}} Loss = \nabla_{f^{[L]}} Loss . \nabla f^{[L]} . W^{[L]} . \nabla f^{[L-1]} . W^{[L-1]} \ldots \nabla f^{[l+1]} . W^{[l+1]}$$

- 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

Recap: 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 loss at any time is totally "forgotten"

Exploding/Vanishing gradients

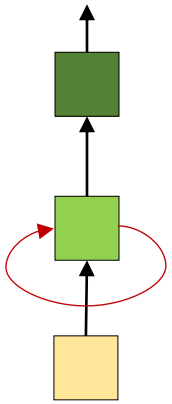
$$\nabla_{f_k} Loss = \nabla Loss . \underline{\nabla f_N . W_{N-1} .} \underline{\nabla f_{N-1} . W_{N-2}} \ \underline{\dots \nabla f_{k+1} W_k}$$

- Every blue term is a matrix
- The chain product for $\nabla_{f_k} Loss$ will
  - E*xpand* $\nabla Loss$ in directions where each stage has singular values greater than 1
  - S*hrink* $\nabla Loss$ in directions where each stage has singular values less than 1

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.

The Jacobian of the hidden layers for an RNN

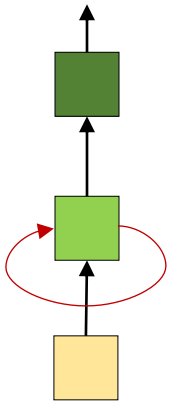


$$h_i(t) = f\big(z_i(t)\big)$$

$$\nabla f(z(t)) = \begin{bmatrix} f'(z_{t,1}) & 0 & \cdots & 0 \\ 0 & f'(z_{t,2}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(z_{t,N}) \end{bmatrix}$$

- $\nabla f(.)$ is the derivative of the output of the (layer of) hidden recurrent neurons with respect to their input
  - For vector activations: A full matrix
  - For scalar activations: A matrix where the diagonal entries are the derivatives of the *activation* of the recurrent hidden layer

The Jacobian



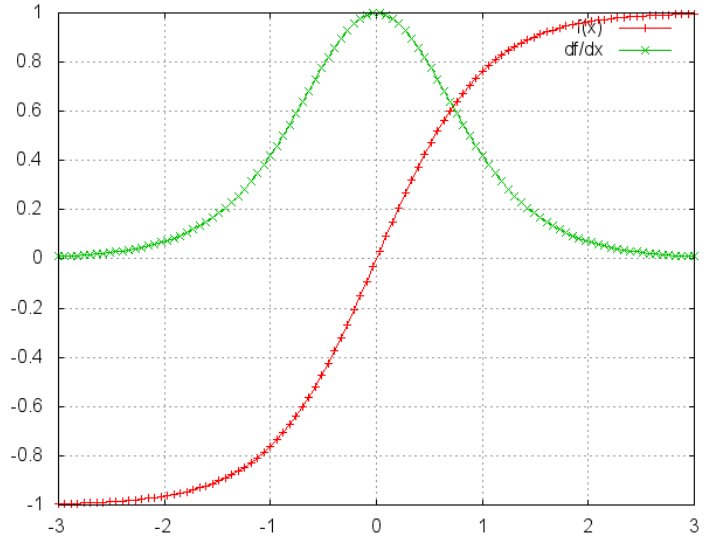
$$h_i(t) = f\big(z_i(t)\big)$$

$$\nabla f(z(t)) = \begin{bmatrix} f'(z_{t,1}) & 0 & \cdots & 0 \\ 0 & f'(z_{t,2}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(z_{t,N}) \end{bmatrix}$$

- The derivative (or subgradient) of the activation function is always bounded
  - The diagonals (or singular values) of the Jacobian are bounded
- There is a limit on how much multiplying a vector by the Jacobian will scale it

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 no more 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)

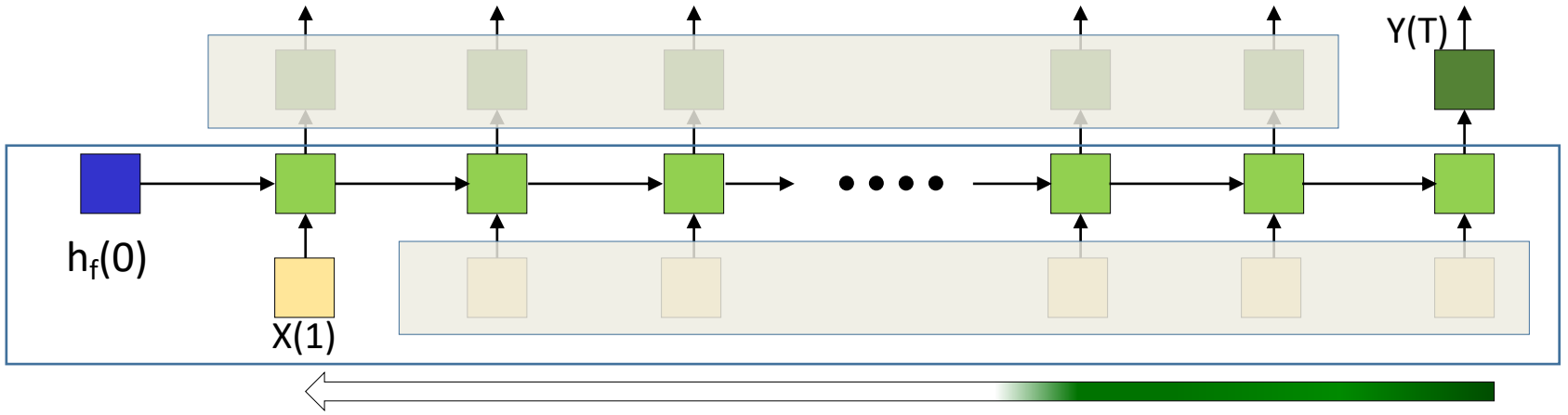
- **Multiplication by the Jacobian is always a *shrinking* operation**

What about the weights

$$\nabla_{f[t]} Loss = \nabla_{f[T]} Loss . \nabla f^{[T]} . W . \nabla f^{[T-1]} . W \ldots \nabla f^{[t+1]} . W$$

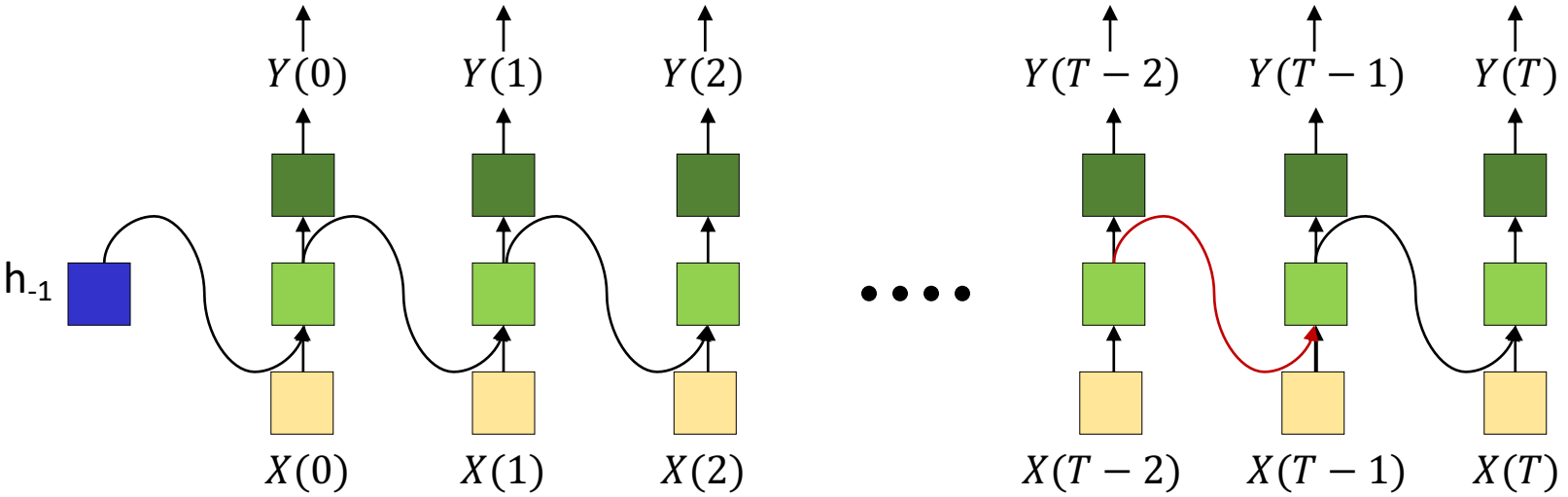
- In a single-layer RNN, the weight matrices are identical
  - The conclusion below holds for any deep network, though
- The chain product for $\nabla_{f[t]} Loss$ will
  - E*xpand* $\nabla_{f[T]} Loss$ along directions in which the singular values of the weight matrices are greater than 1
  - S*hrink*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

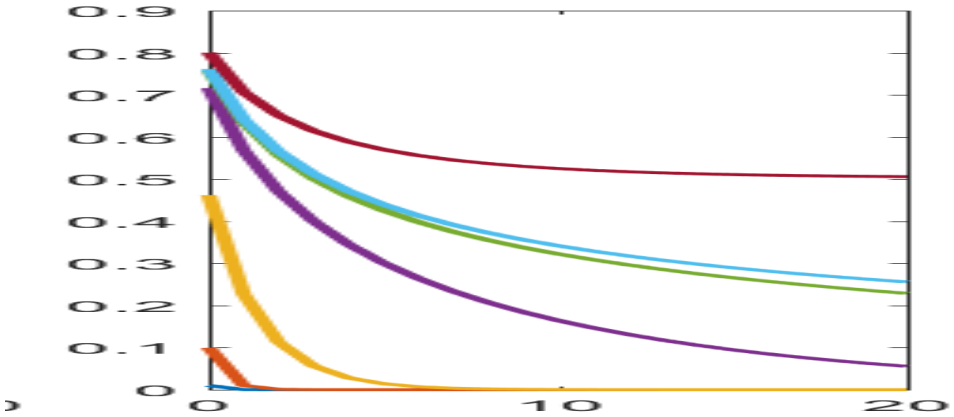
Recurrent nets are very deep nets



- The relation between $X(1)$ 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 $t = 1$

Recall: Vanishing stuff..





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

Training RNNs is hard

- The unrolled network can be very deep and inputs from many time steps ago can modify output
  - Unrolled network is very deep

- Multiply the same matrix at each time step during forward prop

The vanishing gradient problem: Example

- In the case of language modeling words from time steps far away are not taken into consideration when training to predict the next word

- Example: Jane walked into the room. John walked in too. It was late in the day. Jane said hi to ____

This slide has been adopted from Socher lectures, cs224d, Stanford

The long-term dependency problem

- 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

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

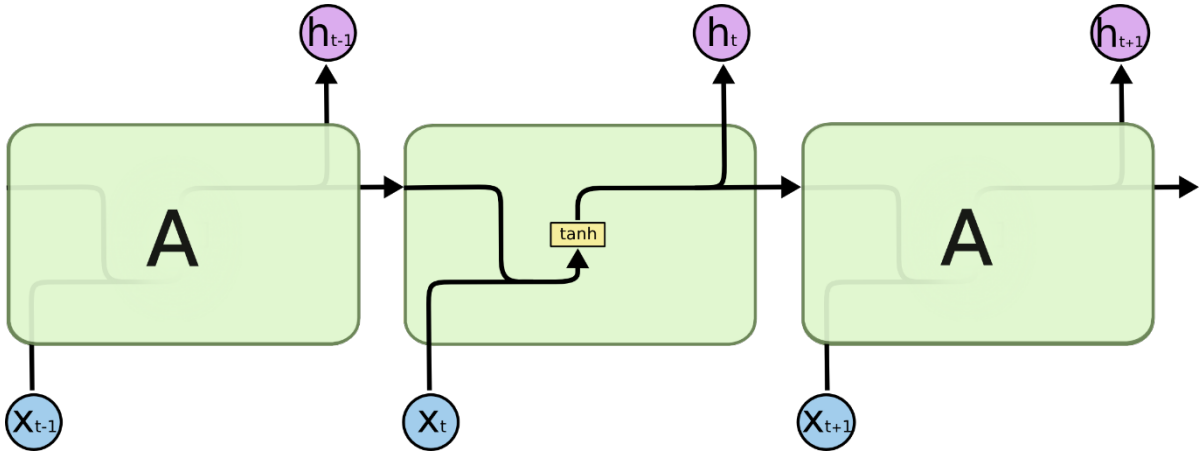
The long-term dependency problem

PATTERN1 [.....] PATTERN 2

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

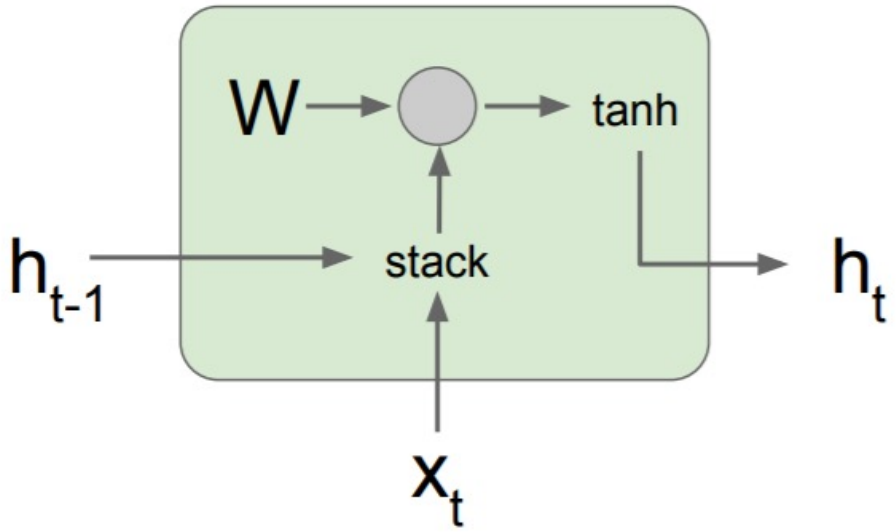
- Any other pattern of any length can happen between pattern 1 and pattern 2
  - RNN will "forget" pattern 1 if intermediate stuff is too long
  - "Jane" → the next pronoun referring to her will be "she"
- Must know to "remember" for extended periods of time and "recall" when necessary
  - Need an alternate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

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

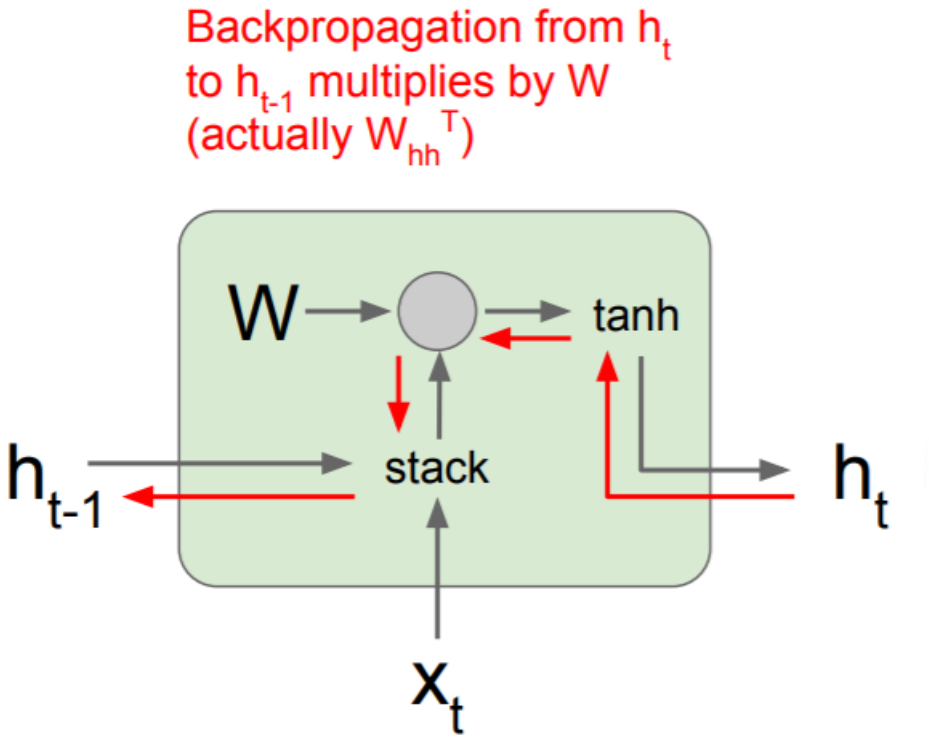
Vanilla RNN Gradient Flow



$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)
\end{aligned}
$$

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

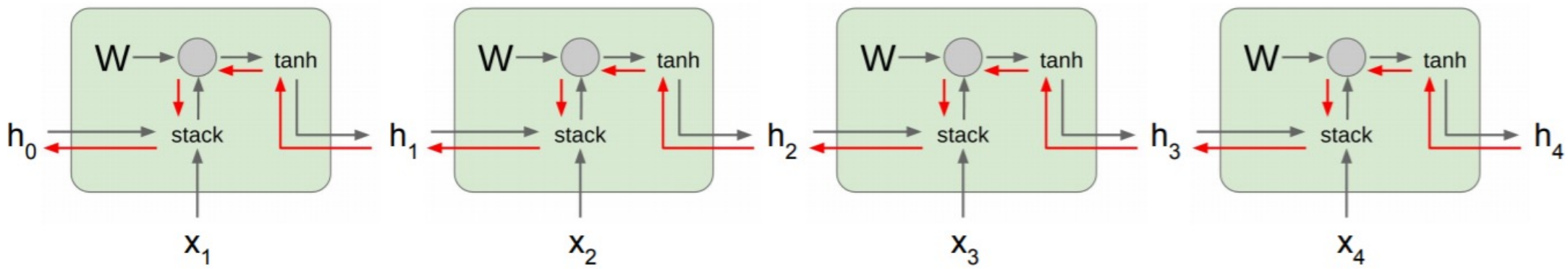
Vanilla RNN Gradient Flow



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left((W_{hh} \quad W_{hx})\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$
$$= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

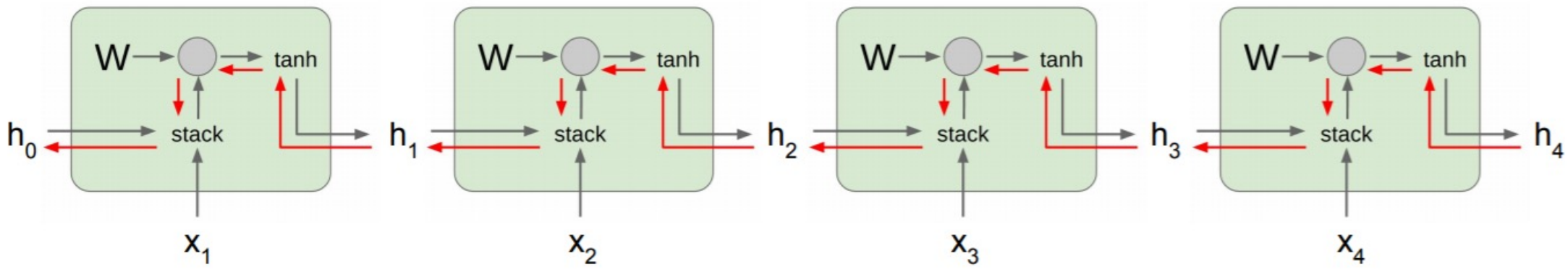
Vanilla RNN Gradient Flow



Largest singular value > 1: Exploding gradients
Largest singular value < 1: Vanishing gradients

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

Vanilla RNN Gradient Flow



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1: Exploding gradients

Largest singular value < 1: Vanishing gradients

Gradient clipping: Scale Computing gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
  grad *= (threshold / grad_norm)
```

Change RNN architecture

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

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/

Better units for recurrent models

- More complex hidden unit computation in recurrence!
  - $h_t = LSTM(x_t, h_{t-1})$
  - $h_t = GRU(x_t, h_{t-1})$
- Main ideas:
  - keep around memories to capture long distance dependencies
  - allow error messages to flow at different strengths depending on the inputs

Exploding/Vanishing gradients

- Can we replace this with something that doesn't fade or blow up?

- 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*
  - Replace them, e.g., by a function of the input that decides if things must be forgotten or not

Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

Four gates

Cell state

Hidden state

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

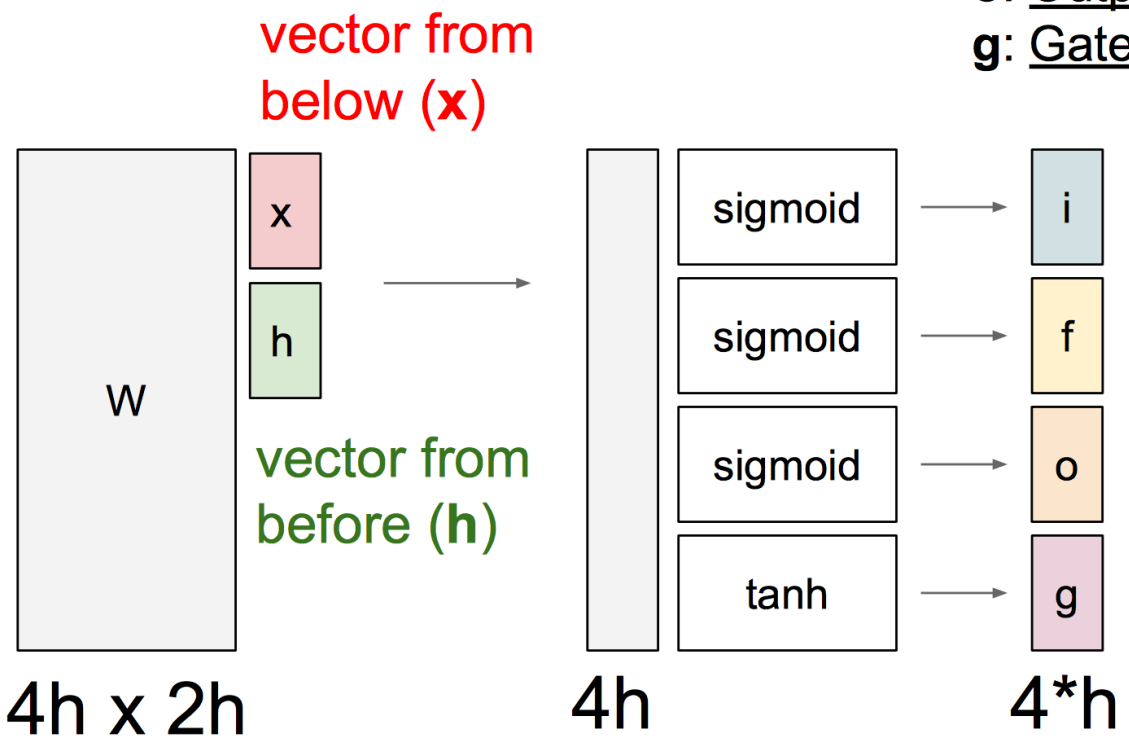
$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

i: Input gate, whether to write to cell
f: Forget gate, Whether to erase cell
o: Output gate, How much to reveal cell
g: Gate gate (?), How much to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

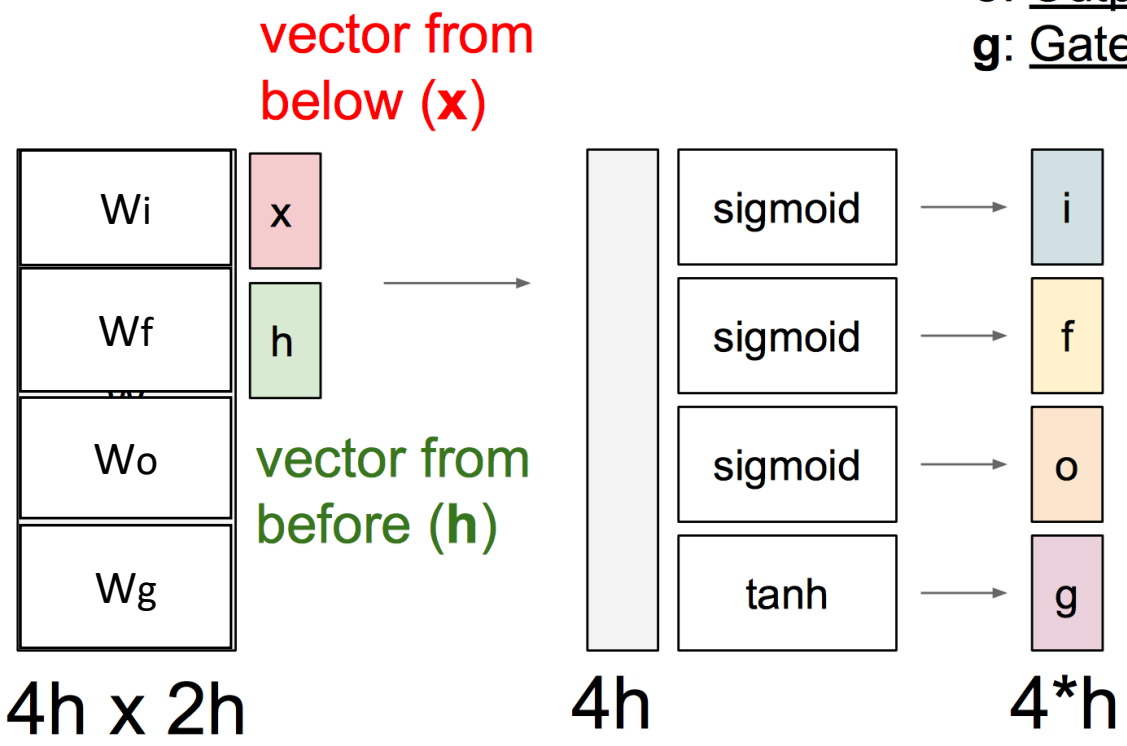
$$h_t = o \odot \tanh(c_t)$$

i, f, o, g, c, and h are vectors of the same length

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

i: Input gate, whether to write to cell
f: Forget gate, Whether to erase cell
o: Output gate, How much to reveal cell
g: Gate gate (?), How much to write to cell



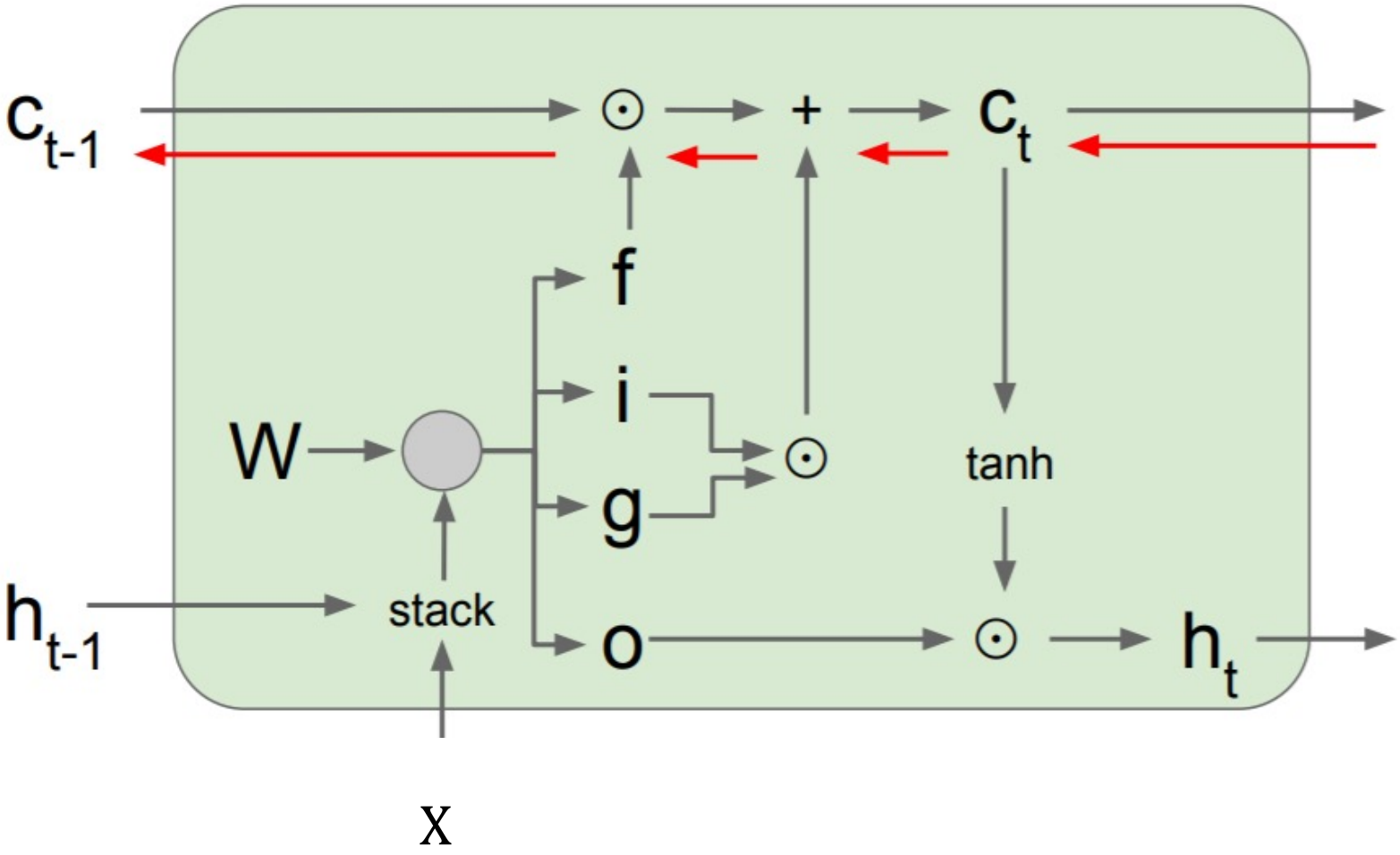
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

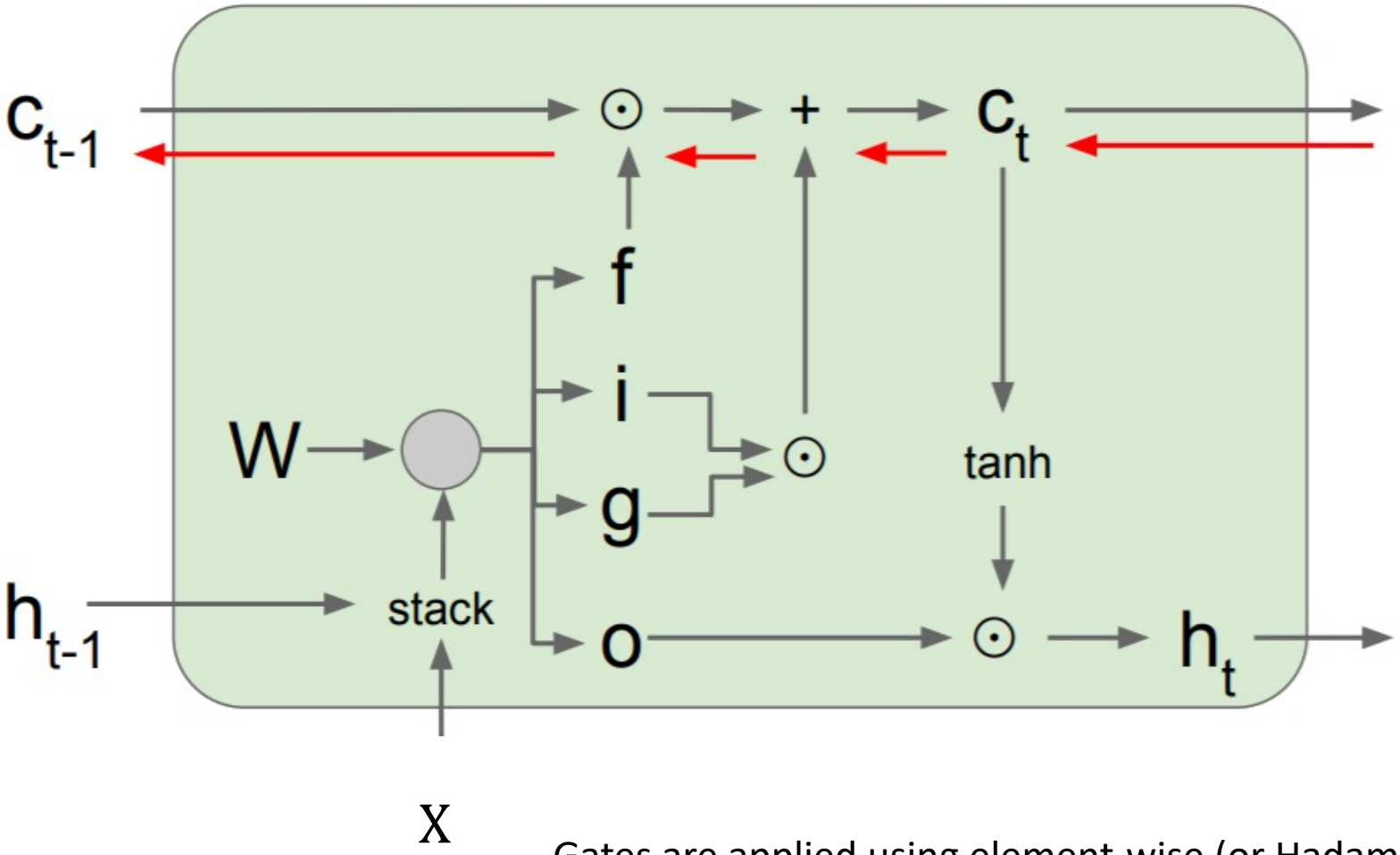
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

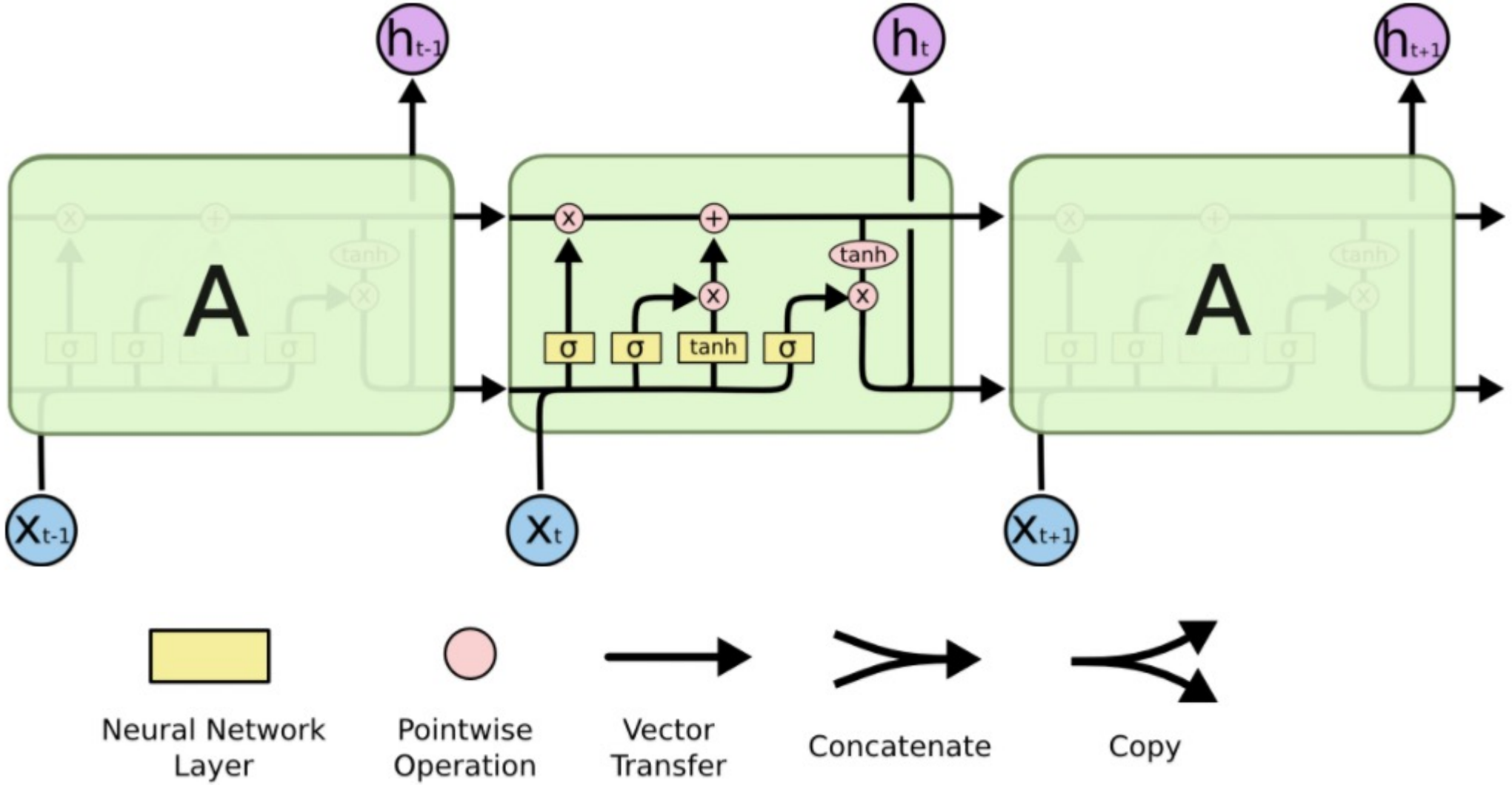
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

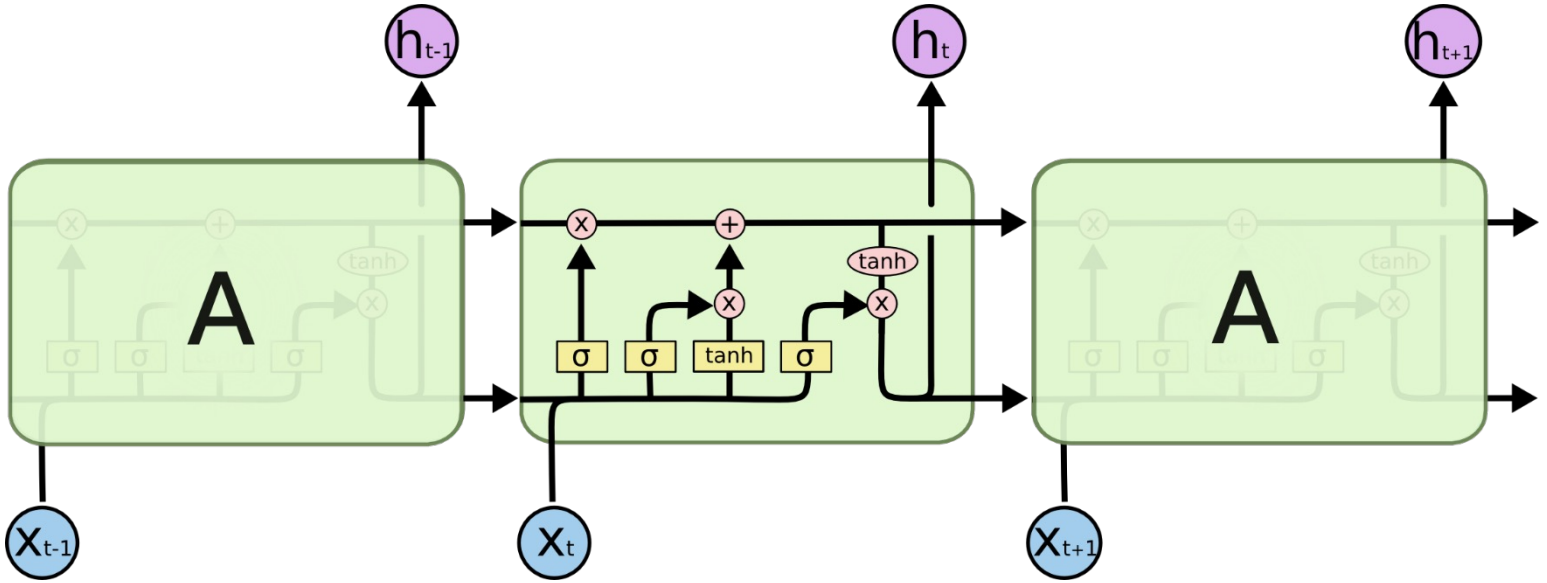
Gates are applied using element-wise (or Hadamard) product

Some visualization



Source: Colah's blog

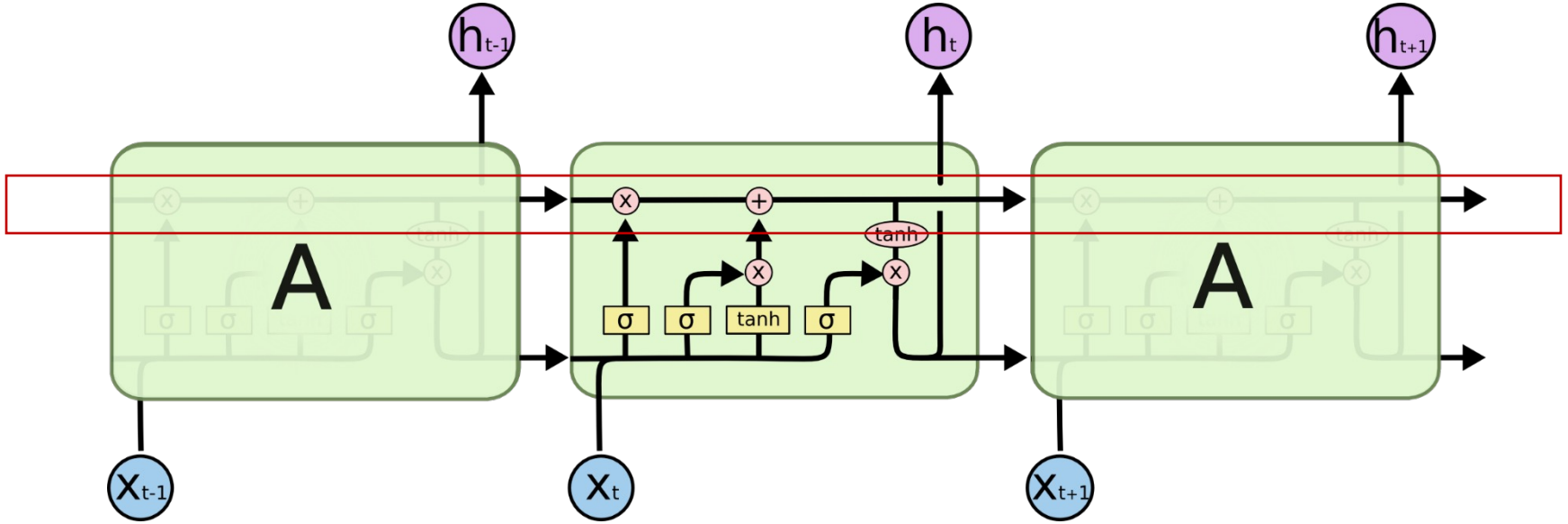
Long Short-Term Memory



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

Source: Colah's blog

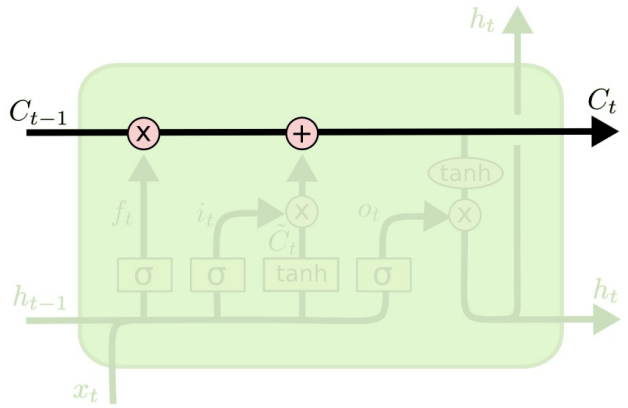
LSTM: Constant Error Carousel



- Key component: a *remembered cell state*

Source: Colah's blog

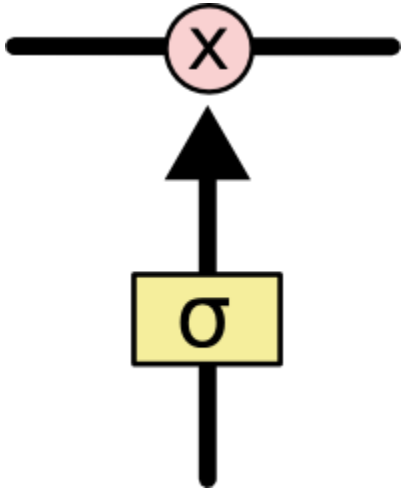
LSTM: CEC



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

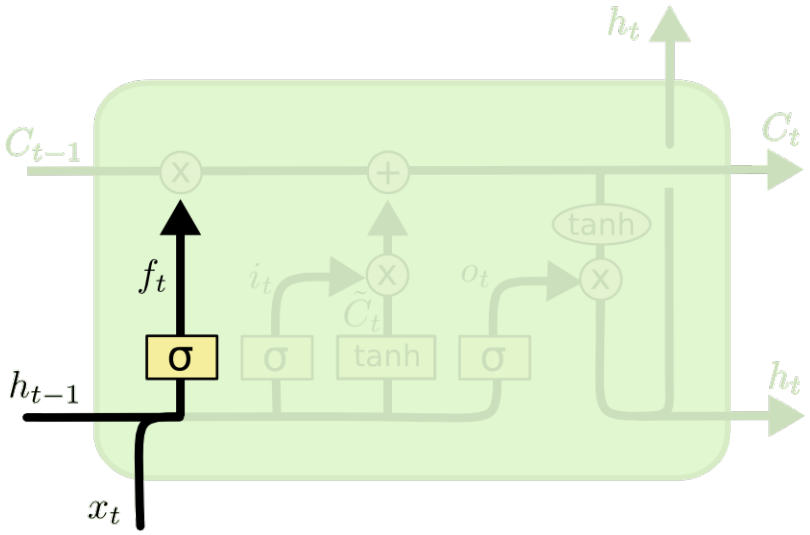
Source: Colah's blog

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

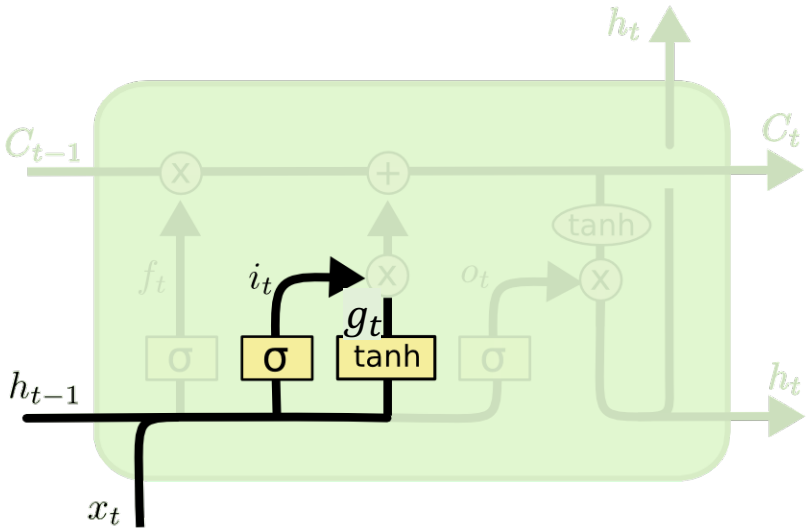


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- 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

Source: Colah's blog

LSTM: Input gate



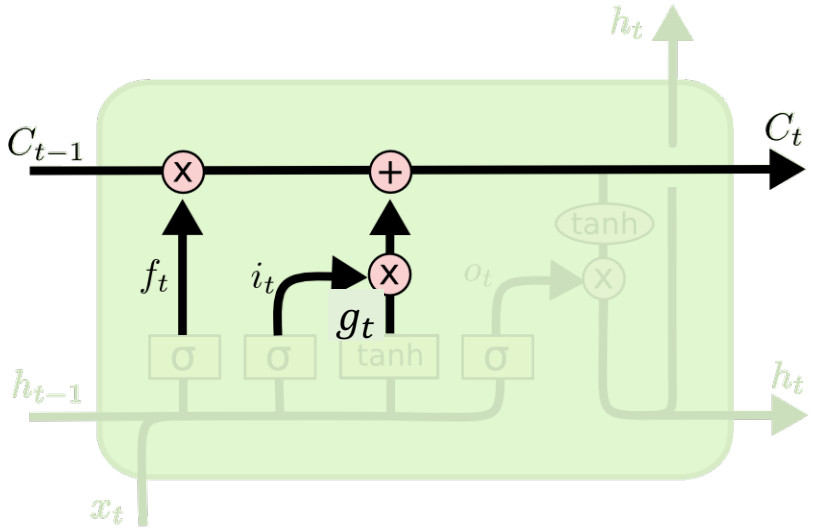
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

$$g_t = \tanh\left(W_g[h_{t-1}, x_t] + b_g\right)$$

- 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 it's worth remembering

Source: Colah's blog

LSTM: Memory cell update

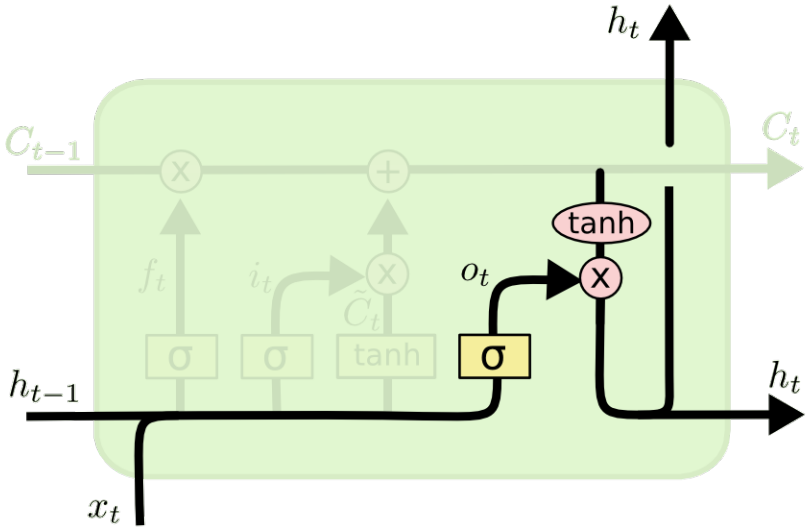


$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

- 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**

Source: Colah's blog

LSTM: Output and Output gate



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

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

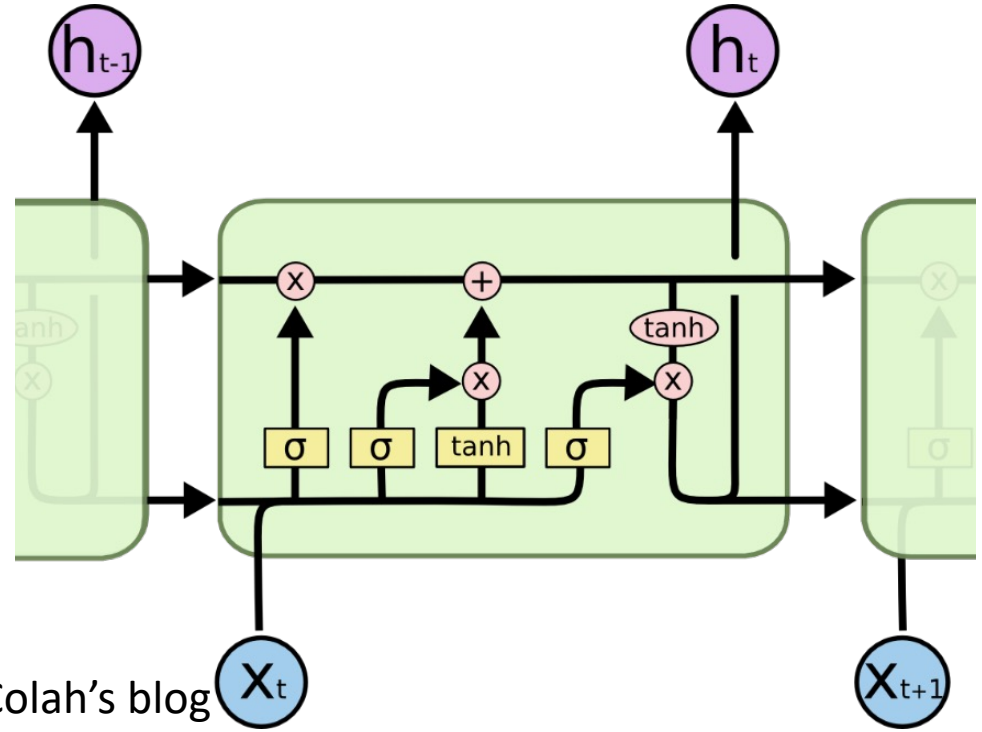
- The *output* of the cell
  - Simply compress it with tanh to make it lie between 1 and -1
    - Note that this compression no longer affects our ability to *carry* memory forward
  - Controlled by an *output* gate
    - To decide if the memory contents are worth reporting at *this* time

Source: Colah's blog

LSTM Equations

- i_t: input gate, how much of the new information will be let through the memory cell.
- f_t: forget gate, responsible for information should be thrown away from memory cell.
- o_t: output gate, how much of the information will be passed to expose to the next time step.
- g_t: self-recurrent which is equal to standard RNN

- c_t: internal memory of the memory cell
- h_t: hidden state

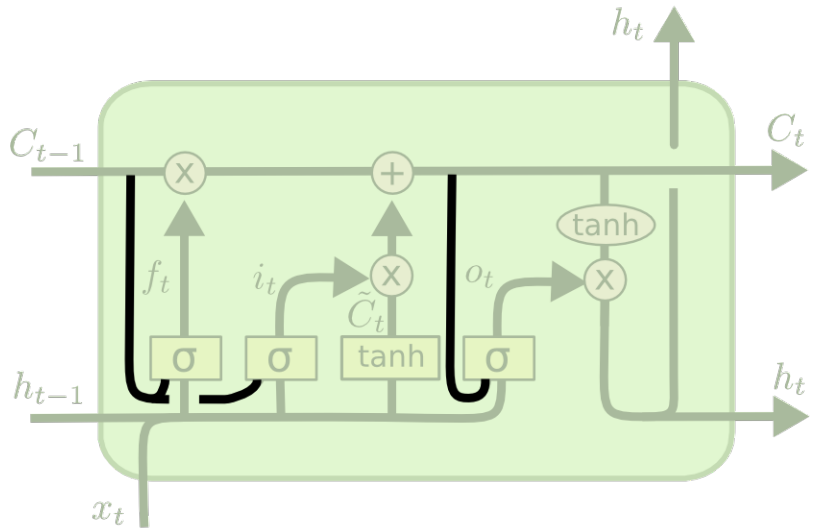


Source: Colah's blog

Long-short-term-memories (LSTMs)

- Gates
  - Input gate (current cell matter): $i_t = \sigma\left(W_i \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_i\right)$
  - Forget (gate 0, forget past): $f_t = \sigma\left(W_f \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_f\right)$
  - Output (how much cell is exposed): $o_t = \sigma\left(W_o \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_o\right)$
- Variables
  - New memory cell: $g_t = \tanh\left(W_c \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_c\right)$
  - Final memory cell: $C_t = i_t \circ g_t + f_t \circ C_{t-1}$
  - Final hidden state: $h_t = o_t \circ \tanh(C_t)$

LSTM: The "Peephole" connection

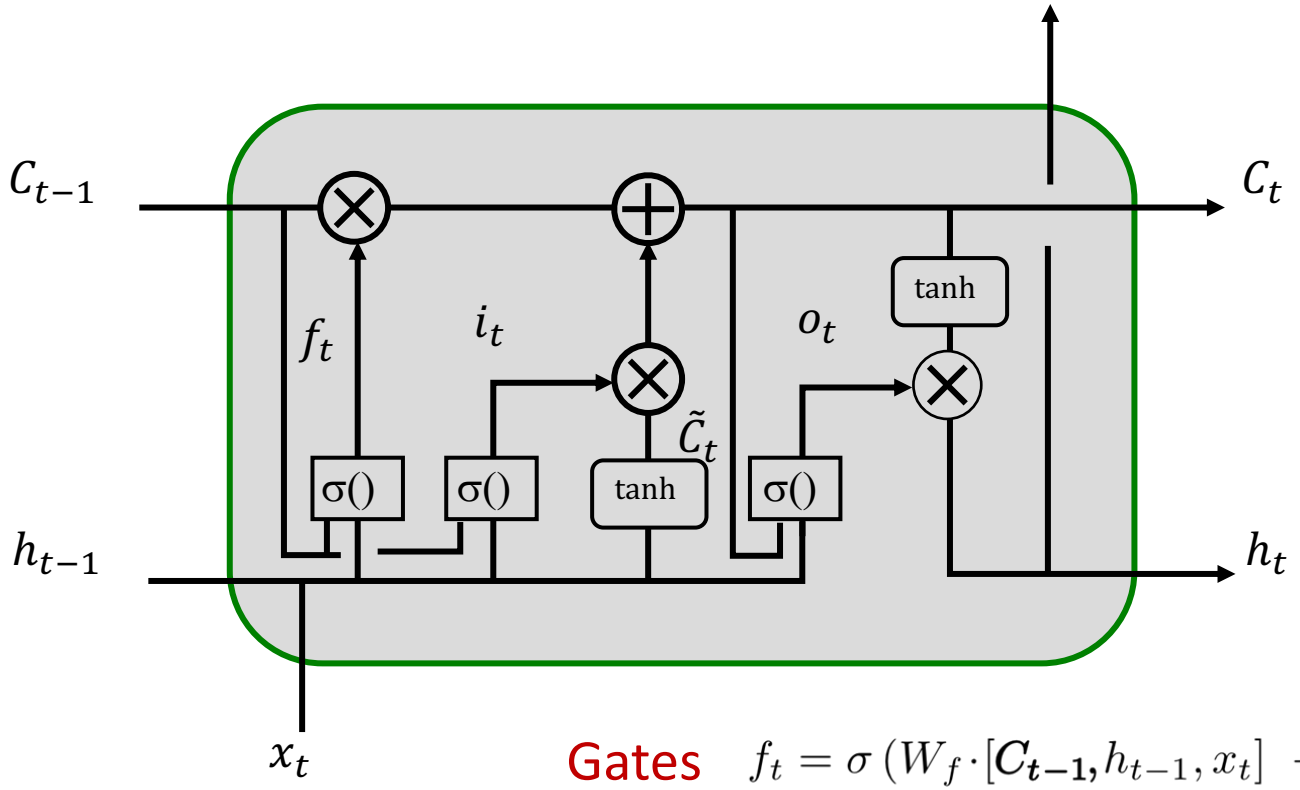


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

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

Source: Colah's blog

LSTM: The "Peephole" connection



- Forward rules:

Gates

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

Variables

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM network forward

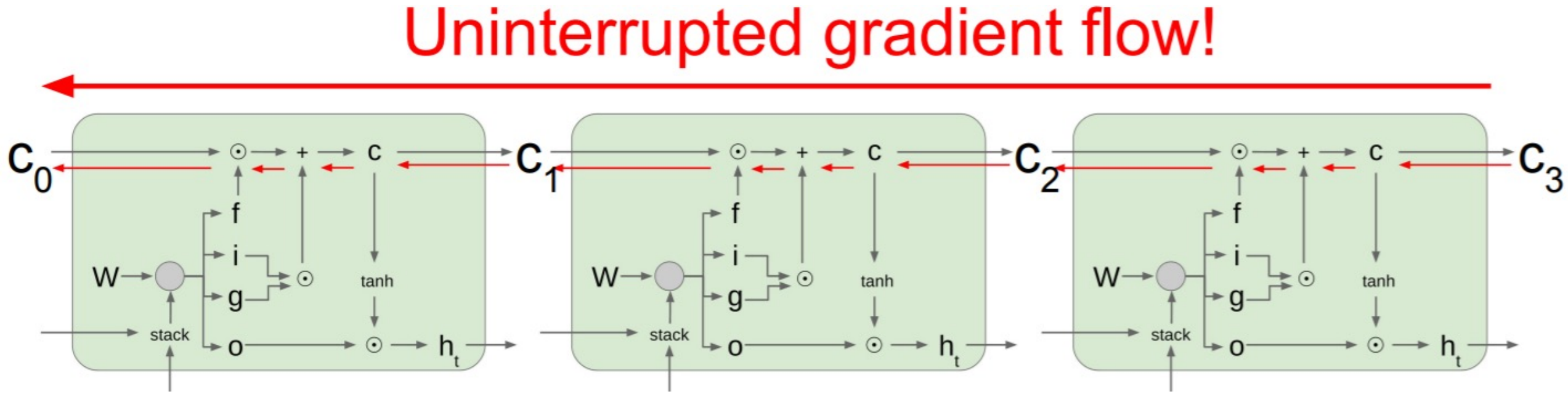
```
# Assuming h(0,*) is known and C(0,*)=0
# Assuming L hidden-state layers and an output layer
# Note: LSTM_cell is an indexed class with functions
# [W{l},b{l}] are the entire set of weights and biases
#                for the l^th hidden layer
# W_o and b_o are output layer weights and biases

for t = 1:T  # Including both ends of the index
    h(t,0) = x(t) # Vectors. Initialize hidden layer 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 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, f, i, o, C_i
function [C_o, h_o] = LSTM_cell.forward(C,h,x, [W,h])
    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) # "∘" is component-wise multiply

    return C_o,h_o
```

Long Short Term Memory (LSTM)

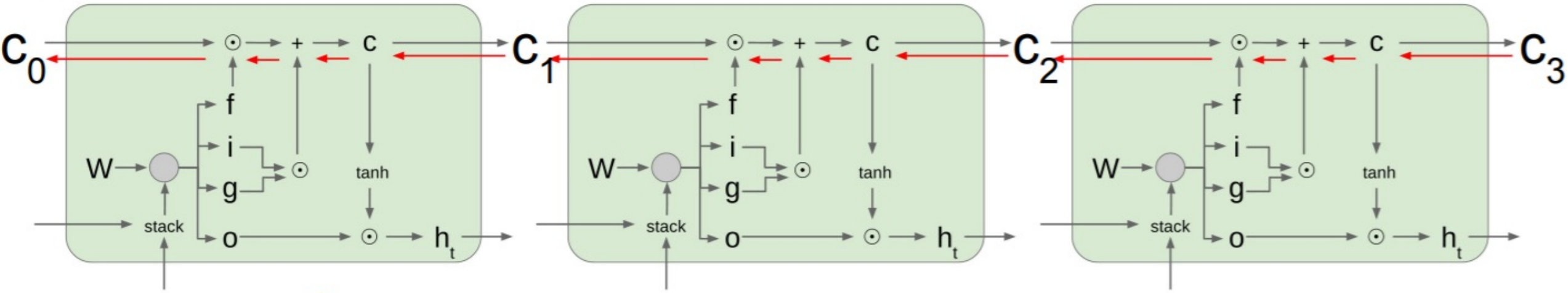
[Hochreiter et al., 1997]



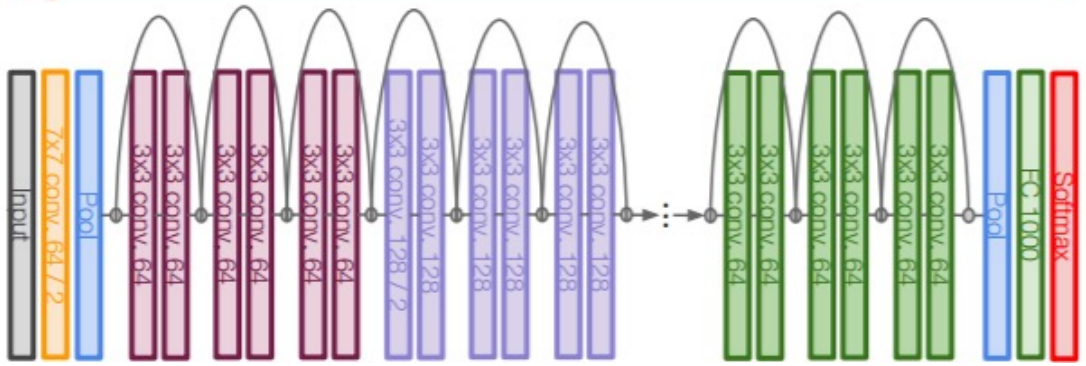
Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

Uninterrupted gradient flow!



Similar to ResNet!



In between:

Highway Networks

$$g = T(x, W_T)$$

$$y = g \odot H(x, W_H) + (1 - g) \odot x$$

Srivastava et al, "Highway Networks", ICML DL Workshop 2015

Gated Recurrent Units: Lets simplify the LSTM

- Don’t bother to separately maintain compressed and regular memories
  - Pointless computation!
- But compress it before using it to decide on the usefulness of the current input!

GRUs

- Gated Recurrent Units (GRU) introduced by Cho et al. 2014
- **Update gate**

$$z_t = \sigma\left(W_z \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_z\right)$$

- **Reset gate**

$$r_t = \sigma\left(W_r \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_r\right)$$

- **Memory**

$$\hat{h}_t = \tanh\left(W_m \begin{bmatrix} r_t \circ h_{t-1} \\ x_t \end{bmatrix} + b_m\right)$$

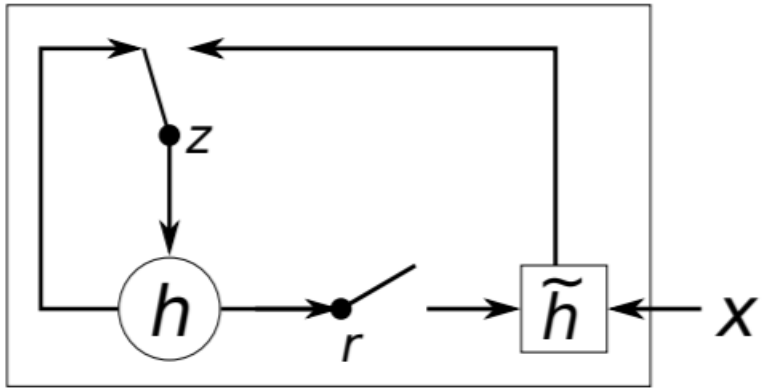
- **Final Memory**

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

If reset gate unit is ~0, then this ignores previous memory and only stores the new input

GRU intuition

- Units with long term dependencies have active update gates z
- Illustration:



This slide has been adopted from Socher lectures, cs224d, Stanford

GRU intuition

- If reset is close to 0, ignore previous hidden state
  →Allows model to drop information that is irrelevant in the future

- Update gate z controls how much of past state should matter now.
  – If z close to 0, then we can copy information in that unit through many time steps! Less vanishing gradient!

- Units with short-term dependencies often have reset gates very active

This slide has been adopted from Socher lectures, cs224d, Stanford

Other RNN Varients

GRU [*Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Cho et al. 2014]

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$
$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$
$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

[*LSTM: A Search Space Odyssey*, Greff et al., 2015]

[*An Empirical Exploration of Recurrent Network Architectures*, Jozefowicz et al., 2015]

MUT1:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + b_z) \\ r &= \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + \tanh(x_t) + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT2:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + W_{hz}h_t + b_z) \\ r &= \text{sigm}(x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + W_{xh}x_t + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT3:

$$\begin{aligned} z &= \text{sigm}(W_{xz}x_t + W_{hz}\tanh(h_t) + b_z) \\ r &= \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r) \\ h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + W_{xh}x_t + b_h) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

Which of these variants is best?

- Do the differences matter?
  - Greff et al. (2015), perform comparison of popular variants, finding that they're all about the same.
  - Jozefowicz et al. (2015) tested more than ten thousand RNN architectures, finding some that worked better than LSTMs on certain tasks.

LSTM Achievements

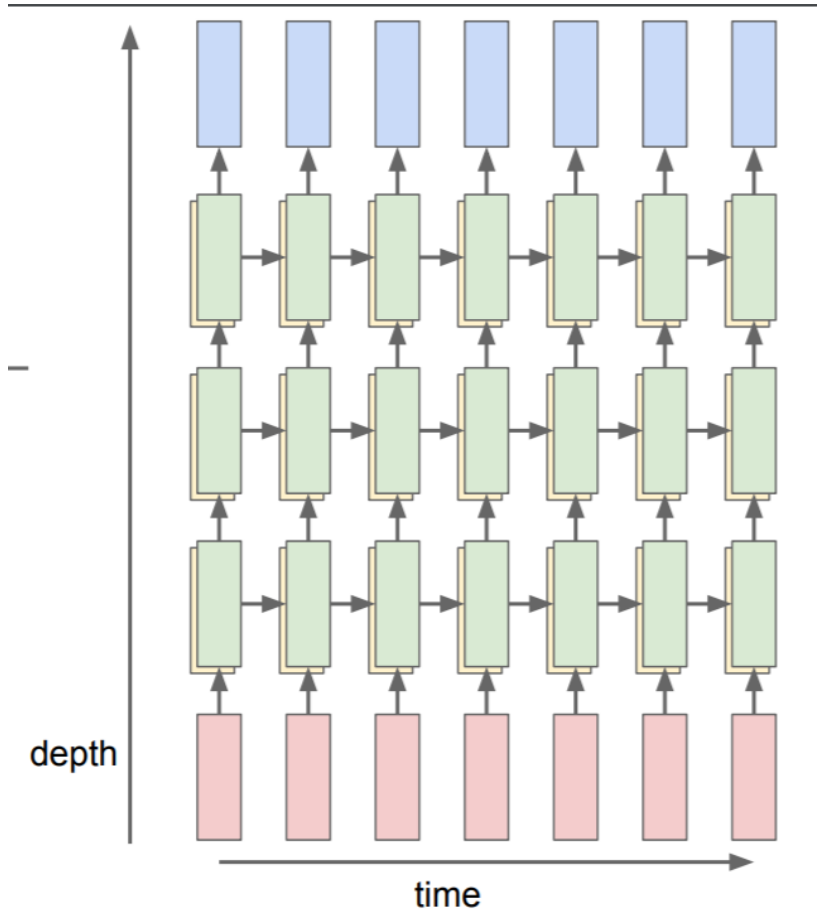
- In 2013–2015, LSTMs started achieving state-of-the-art results
  - LSTMs have essentially replaced n-grams as language models for **speech**.
  - **Image captioning** and other multi-modal tasks which were very difficult with previous methods became feasible with LSTMs.
  - **Neural MT**: broken away from plateau of SMT, especially for grammaticality (partly because of characters/subwords), but not yet industry strength.
  - Many **traditional NLP tasks** work very well with LSTMs, but not necessarily the top performers: e.g., POS tagging and NER: Choi 2016.

[Ann Copestake, Overview of LSTMs and word2vec, 2016.]
https://arxiv.org/ftp/arxiv/papers/1611/1611.00068.pdf

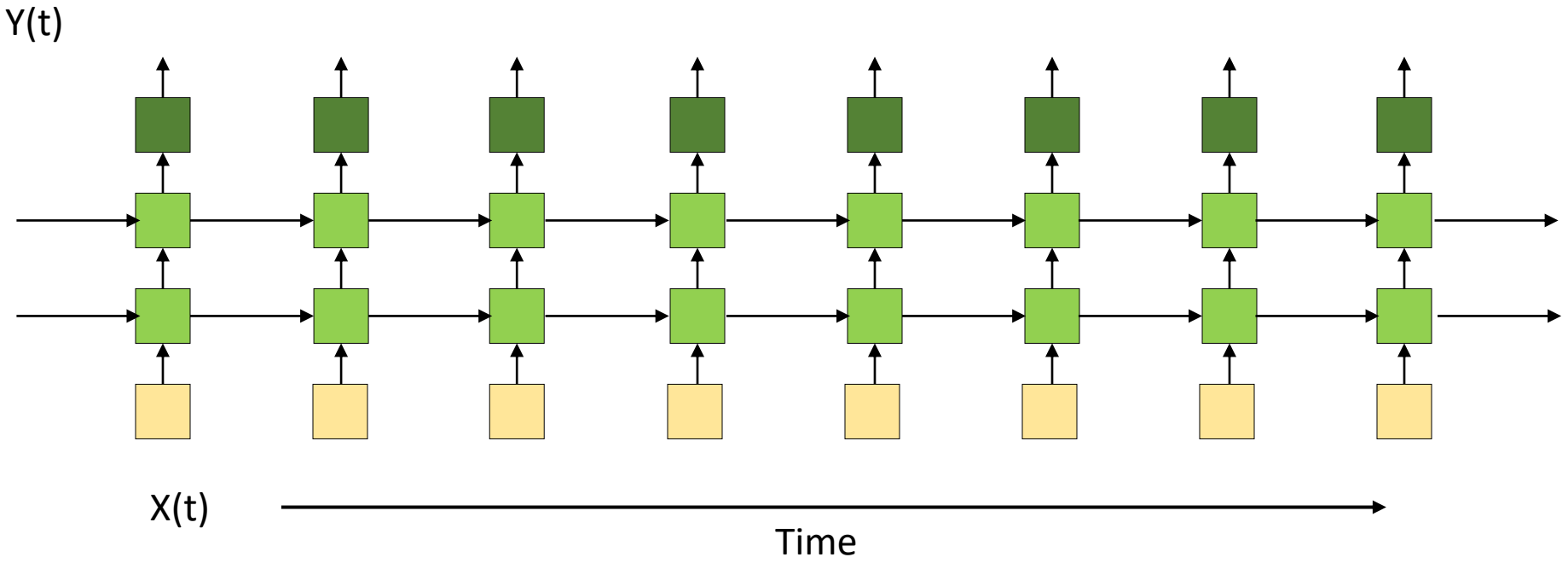
Multi-layer RNN

$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$h \in \mathbb{R}^n$ W^l $[n \times 2n]$



Multi-layer LSTM architecture



- Each green box is now an entire LSTM or GRU unit
- Also keep in mind each box is an *array* of 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 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**

RNN: Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don’t work very well
- Backward flow of gradients in RNN can explode or vanish.
  - Exploding is controlled with gradient clipping.
  - Vanishing is controlled with additive interactions (LSTM)
- Common to use LSTM or GRU: their additive interactions improve gradient flow