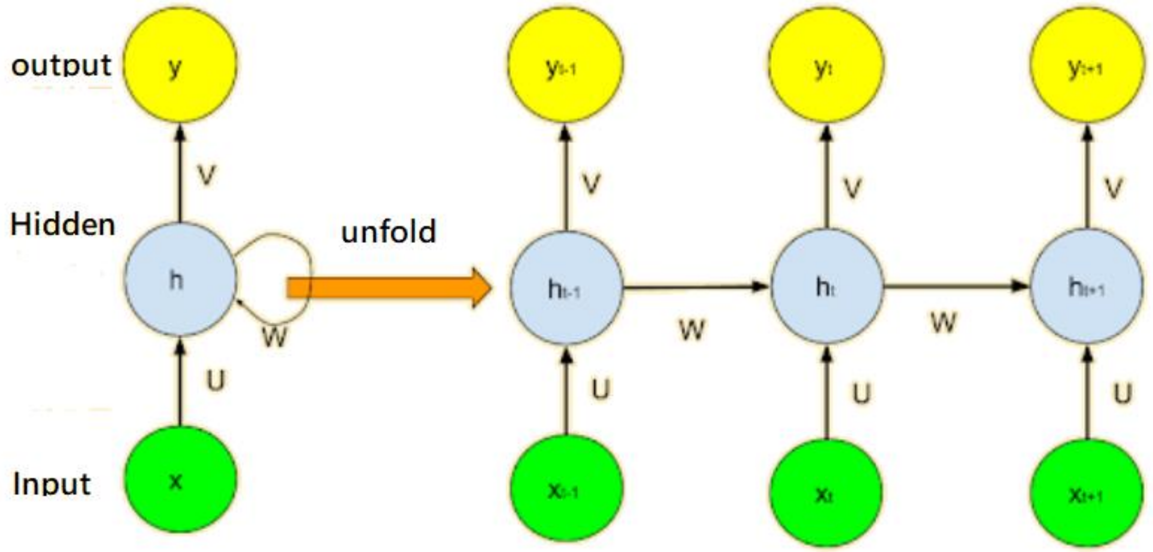
Recurrent Neural Network(RNN)

- **Architecture**
- **An example**
- **Gradient vanishing and exploding**
- **Long Short Term Memory (LSTM) and GRU**
- **Applications of RNN (Sentiment & NLP)**

Architecture of RNN Ashing's Blog

- Left: Visual illustration of the RNN recurrence relation. $S_t = S_{t-1} * W + X_t * U$ (here: S refers to h in the right-hand side Figure)
- Right: RNN states recurrently unfolded over steps t-1, t, t+1.
- Note that the parameters U,V, and W are shared between all the states (S_{t-1} , S_t , S_{t+1})
- $S_t = \tanh(S_{t-1} * W + X_t * U)$
- W defines a transformation from state to state, and U is a transformation from input to state
- The final output will be $y_t = V * S_t$



An illustrative example

Some relationship

At time t=0, U, W , V and h0 are randomly initialized, h0 usually is initialized to 0, or a 0 vector

$$h_1 = f(U \cdot x_1 + W \cdot h_0)$$
$$y_1 = g(V \cdot h_1)$$

f(.) is the activation function of the hidden layer; g(.) is the activation function of the output layer.
f could be chosen from tanh, sigmoid or relu; g could be Softmax

h1 is the state of the hidden layer at time t=1 and y1 is the output at t=1.

At time step t,

$$h_t = f(U \cdot x_t + W \cdot h_{t-1})$$
$$y_t = g(V \cdot h_t)$$

RNN has memory, through W it keeps the past history as an auxiliary input at time step t.

Meaning of the hidden state

- A hidden state can be regarded as

$$h = f(\text{current input} + \text{past history})$$

- Based on h, we make the prediction.

Total error

- Total error is:

$$E = \sum_{i} e_i = \sum_{i=1}^{t} f_e\,(y_i - d_i)$$

- Y_i is the predicted value, d_i is the real value. f_e could be cross entropy or square error.

- RNN has memory over time because the state h contains information based on the previous steps.

- RNN can look back only a few time steps because of the vanishing of gradients problem.

- Therefore, GRU or LSTM were proposed!

A forward propagation example

h0=[0.0, 0.0]

x1=[1.0]

U=[0.5,0.6]

V=[1.0,2.0] W=[0.1,0.2]

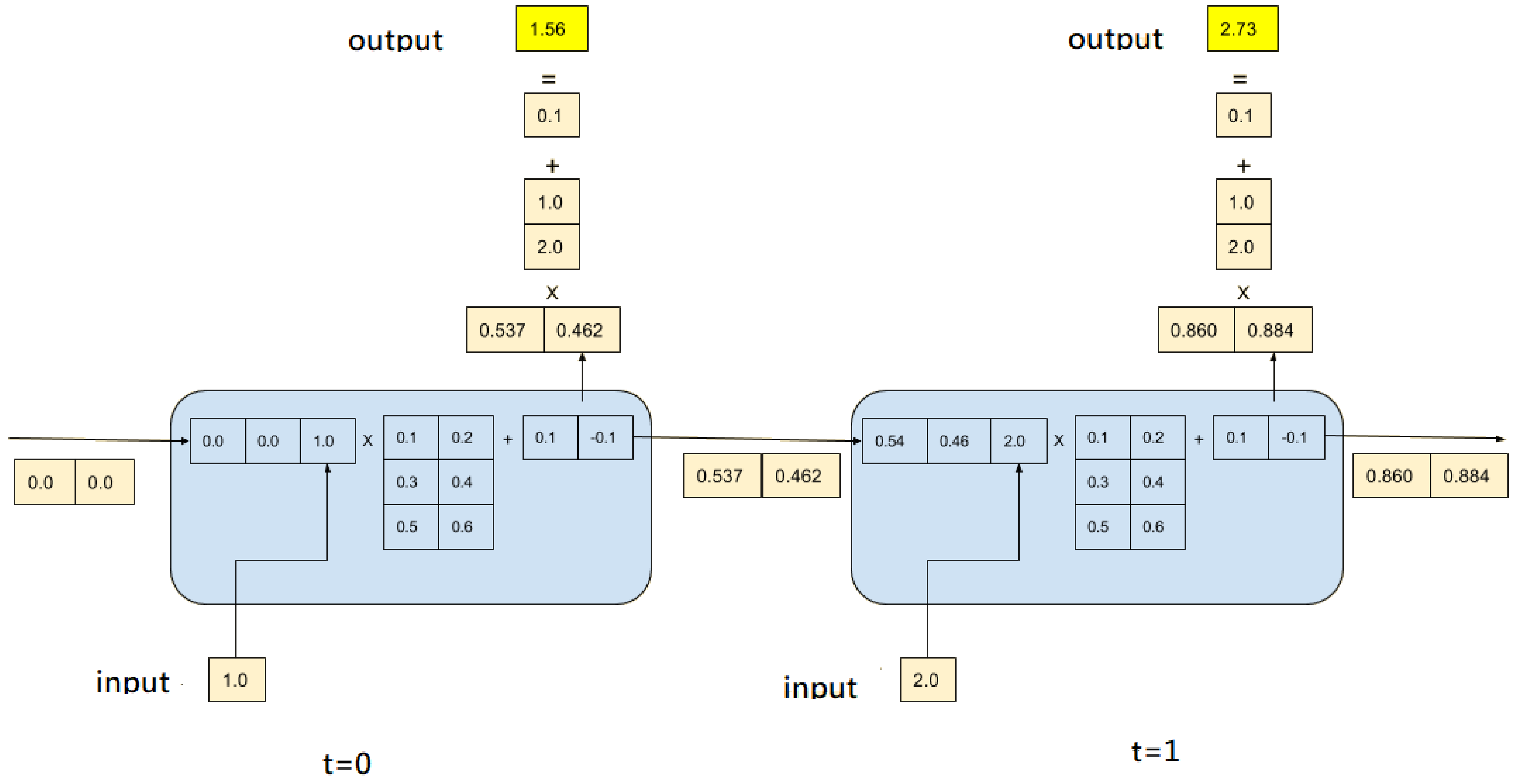
[0.3,0.4]

Hidden layer bias=[1.0,-1.0]

Output bias=[0.1]

State dim. is 2 and input dim is 1.

Note that in equations, we use column vectors; here, in the figure, we use row vectors.



output
1.56
=
0.1
+
1.0
2.0
x
0.537
0.462
0.0
0.0
0.0
0.0
1.0
x
0.1
0.2
0.3
0.4
0.5
0.6
+
0.1
-0.1
input
1.0
t=0
output
2.73
=
0.1
+
1.0
2.0
x
0.860
0.884
0.537
0.462
0.54
0.46
2.0
x
0.1
0.2
0.3
0.4
0.5
0.6
+
0.1
-0.1
0.860
0.884
input
2.0
t=1

Note that we combine U and W; also combine x and h.
Also, note that $(A*B)^t = B^t * A^t$

$$h_1 = f(U \cdot x_1 + W \cdot h_0)$$
$$y_1 = g(V \cdot h_1)$$

Note that one inconsistency exists, the W matrix in this example is actually
the transpose of W in the above equation. However, the example and its code are correct!

Merge h0 and x1 to vector (0.0, 0.0, 1.0).
Merge W and U as :

$$W_{rnn} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{bmatrix}$$

The output:

Output weight:

$$V = \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix}$$

$$\begin{bmatrix} 0.537 & 0.462 \end{bmatrix} \times \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix} + 0.1 = 1.56$$

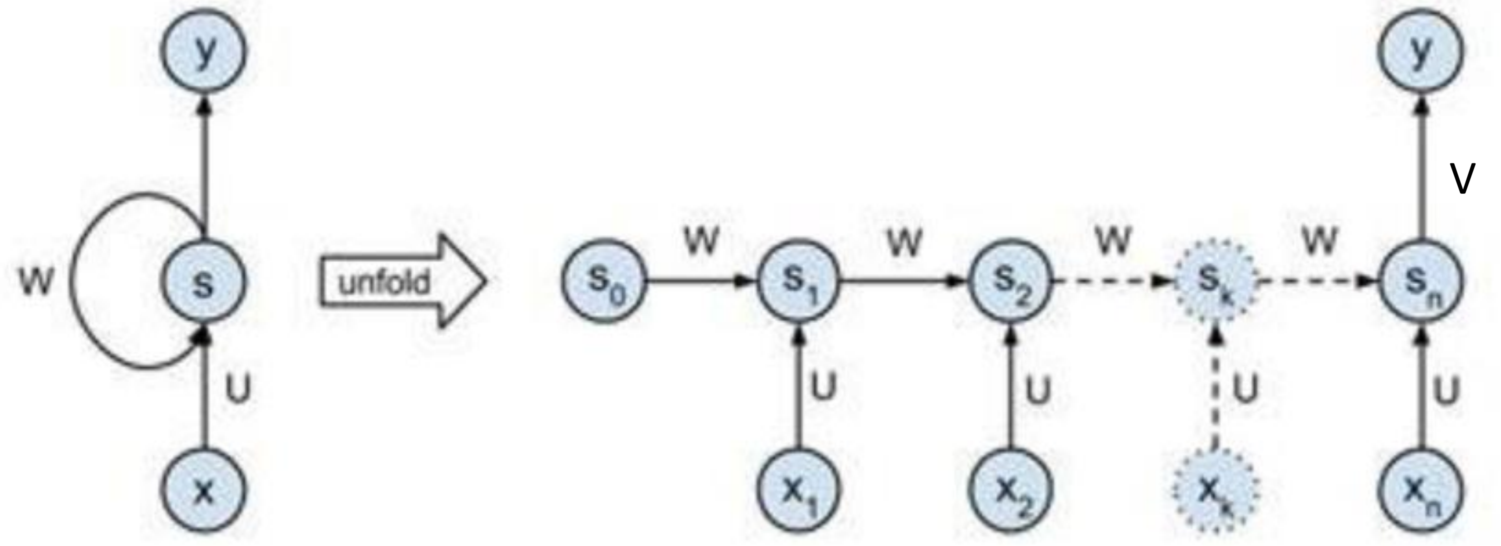
The calculation of h1:

$$\tanh\left(\begin{bmatrix} 0.0 & 0.0 & 1.0 \end{bmatrix} \times \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{bmatrix} + \begin{bmatrix} 0.1 & -0.1 \end{bmatrix} \right) = \tanh(\begin{bmatrix} 0.6 & 0.5 \end{bmatrix}) = \begin{bmatrix} 0.537 & 0.462 \end{bmatrix}$$

Backward propagation (BPTT: backward propagation through time)

$$V(t+1) = V(t) + \alpha \cdot \nabla V$$
$$U(t+1) = U(t) + \alpha \cdot \nabla U$$
$$W(t+1) = W(t) + \alpha \cdot \nabla W$$

Explain Gradient vanishing or exploding using a Simplified case: $S_t = s_{t-1} * w + x_t * U$

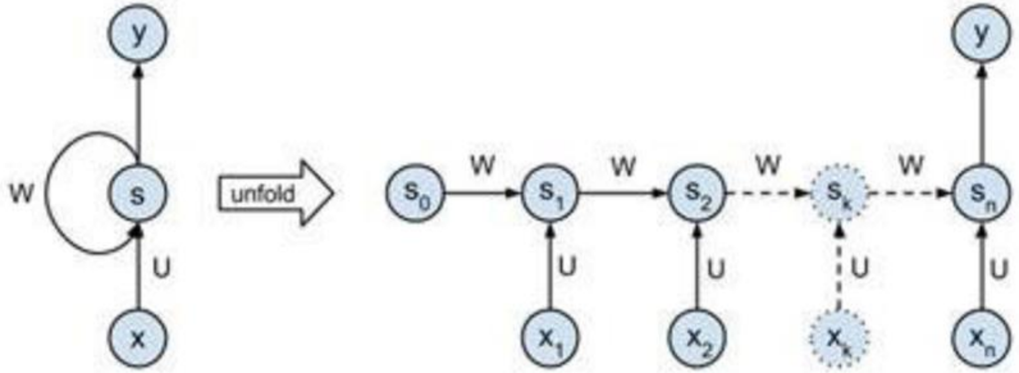


- Want to count the number of ones in a sequence of ones and zeros

In: (0, 0, 0, 0, 1, 0, 1,0, 1, 0)   Output: 3

Set V to 1, the best solution (theoretical) for U, W are: U = 1, W = 1 (never known)； (Initialized state) S_0 = 0

the recurrence relation: $S_t = s_{t-1} * w + x_t * U,$ all are scalar (Remember! simplified case!)



Simplified case: $S_t = s_{t-1} * w + x_t * U$

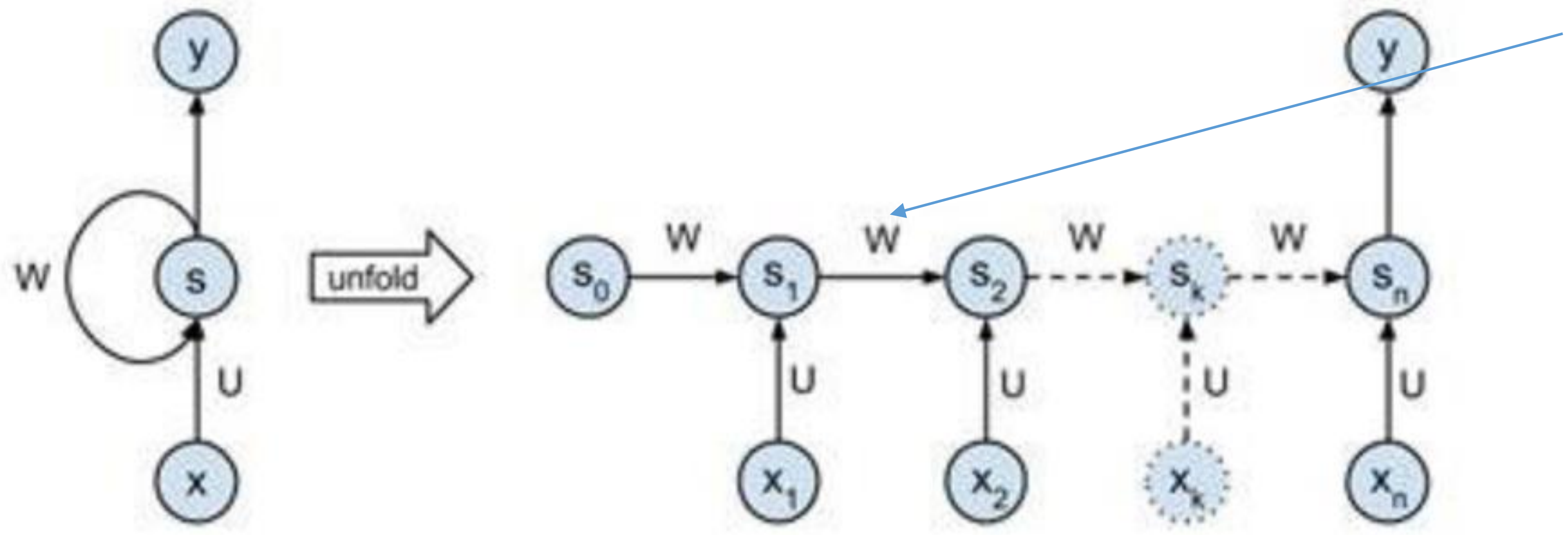
- Cost function $\xi = \sum_i (targets_i - y_i)^2$
- Note that, the output y is the value of the last state S_n
- Therefore, gradient= $2\Sigma\,(y_i - targets_i)$, this gradient then propagates backward to the previous stage
- Let ξ denote the cost function, then

$$\frac{\partial \xi}{\partial S_{t-1}} = \frac{\partial \xi}{\partial S_t}\frac{\partial S_t}{\partial S_{t-1}} = \frac{\partial \xi}{\partial S_t} W \qquad \text{(chain rule)}$$

An important relationship! There is a factor of W between the two partial derivatives.

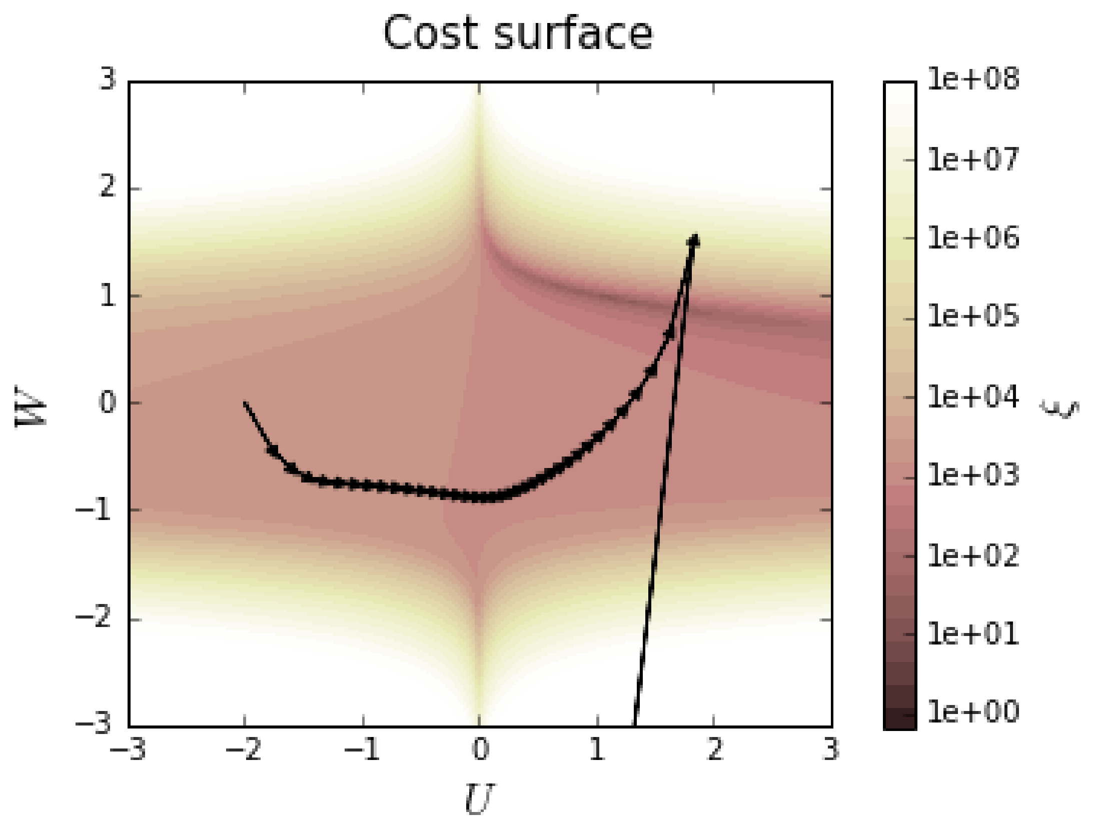
- The gradient of the parameters are accumulated with this:

$$\frac{\partial \xi}{\partial U} = \sum_{t=0}^{n} \frac{\partial \xi}{\partial S_t} x_t \qquad\qquad \frac{\partial \xi}{\partial W} = \sum_{t=1}^{n} \frac{\partial \xi}{\partial s_t} \frac{\partial s_t}{\partial w} = \sum\nolimits_{t=1}^{n} \frac{\partial \xi}{\partial S_t} s_{t-1}$$



- $\frac{\partial \xi}{\partial W} = \frac{\partial \xi}{\partial S_2} \frac{\partial S_2}{\partial W}$

- With the initial value of U, W = -2, 0, the execution of the program ends up with U, W = NaN (Not a Number), too small, big negative values
- U, W move toward the optimum (U=W=1) until it overshoots and hits approximately (U=W=1.5). Then the gradient values blow up and make the parameter values jump out of the plot. Why? gradients exploding


Cost surface
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
U
W
1e+08
1e+07
1e+06
1e+05
1e+04
1e+03
1e+02
1e+01
1e+00
ξ

Vanishing and exploding of gradients

- The gradient exploding problem brings RNN training to an unstable state due to the blowing up of long-term components of RNN. (may not converge)
- The vanishing gradient problem happens when gradients of the long-term components go to zero exponentially fast, and the model is unable to learn from temporally distant events. (cannot learn from distant events or steps)

Reasons:

Dependency of the previous m steps

- $\frac{\partial \xi}{\partial S_{t-m}} = \frac{\partial \xi}{\partial S_t} * \frac{\partial S_t}{\partial S_{t-m}}$
- $\frac{\partial S_t}{\partial S_{t-m}} = \frac{\partial S_t}{\partial S_{t-1}} * \cdots * \frac{\partial S_{t-m+1}}{\partial S_{t-m}} = W^m$

- $\frac{\partial \xi}{\partial W} = \sum_{t=1}^{n} \frac{\partial \xi}{\partial S_t} s_{t-1}$,; while for gradient component of (n-m) step before, we have :

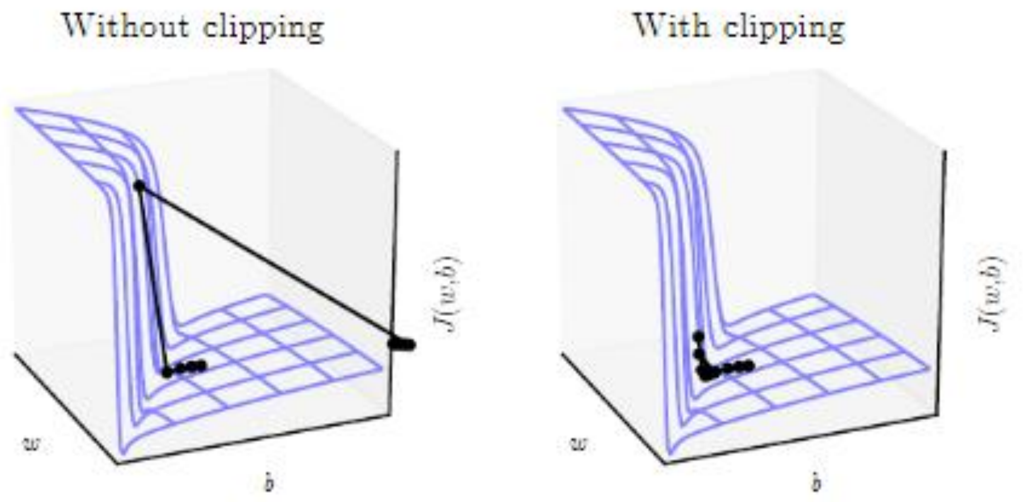
$$\frac{\partial \xi}{\partial S_{n-m}} = \frac{\partial \xi}{\partial S_n} * \frac{\partial S_n}{\partial S_{n-1}} * \frac{\partial S_{n-1}}{\partial S_{n-2}} * \cdots * \frac{\partial S_{n-m+1}}{\partial S_{n-m}} = \frac{\partial \xi}{\partial S_n} * W^m$$

- If W = 1.5, W^{50}= 1.5^{50}=$6*10^{8}$, gradient exploding
- If W= 0.6, W^{20}=0.6^{20}=$3*10^{-5}$, gradient vanishing
- Gradient vanishing is serious, if we use a nonlinear activation function, we may aggravate (加重) the gradient vanishing problem, why? Consider:
- S_t=$\tanh(S_{t-1} * W + X_t * U)$
- Derivative of tanh is (1+f)(1-f)=1-f^2, where |f| <=1, we must multiply by $(1-f^2)^m$

To deal with the gradient exploding problem

1. Gradient clipping, where we threshold the maximum value of a gradient to get

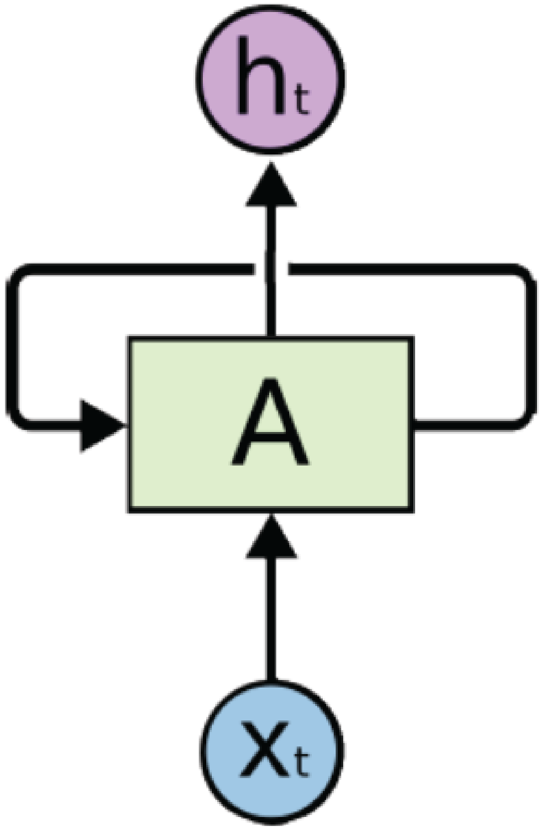
$$if\ \|\boldsymbol{g}\| > \beta,\ \ g \leftarrow \frac{\beta\ \boldsymbol{g}}{\|\boldsymbol{g}\|},$$ directions of gradients are the same, value is restricted to β



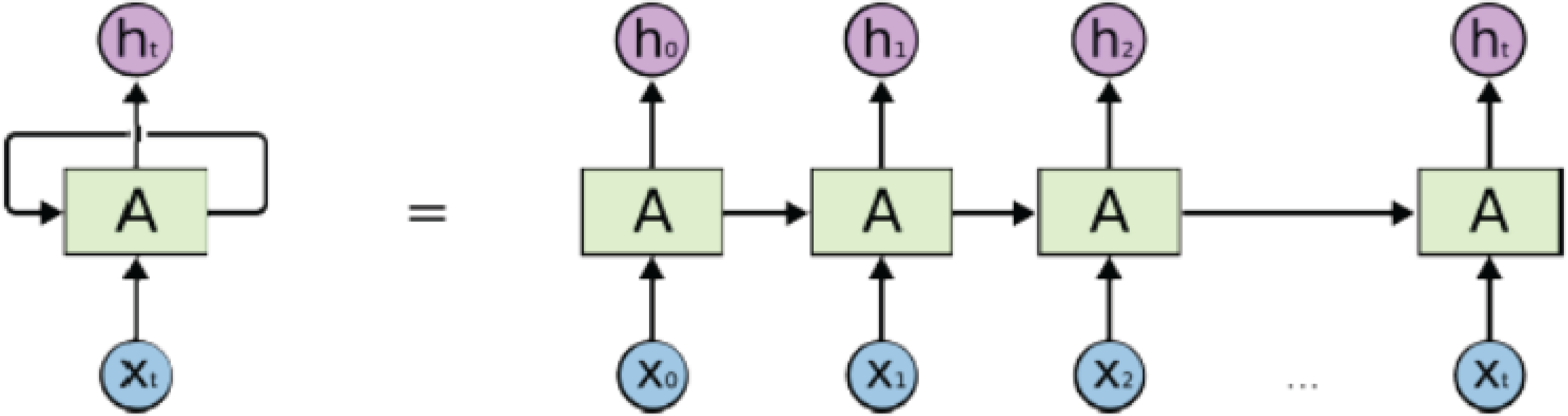
Long short-term Memory (LSTM) https://blog.xpgreat.com/file/lstm.pdf --Sepp Hochreiter, Germany and Jurgen Schmidhuber, Switzerland

- Note that we have two different states: **cell state (C)**and **hidden state (h)**
- Long short-term memory utilizes three gates to control the information passing through.
- The three gates are **forget gate, input gate,** and **output gate**.
- These gates are composed of a logistic sigmoid function that can only output values between 0 and 1; if its value is 1, it lets information to pass through; if its value is 0, the information is blocked. So, we can control the amount of information to pass through using a sigmoid function
- **Forget gate** control information from the previous cell state; **input gate** control information of the current input; **output gate** controls the amount of output from the cell state

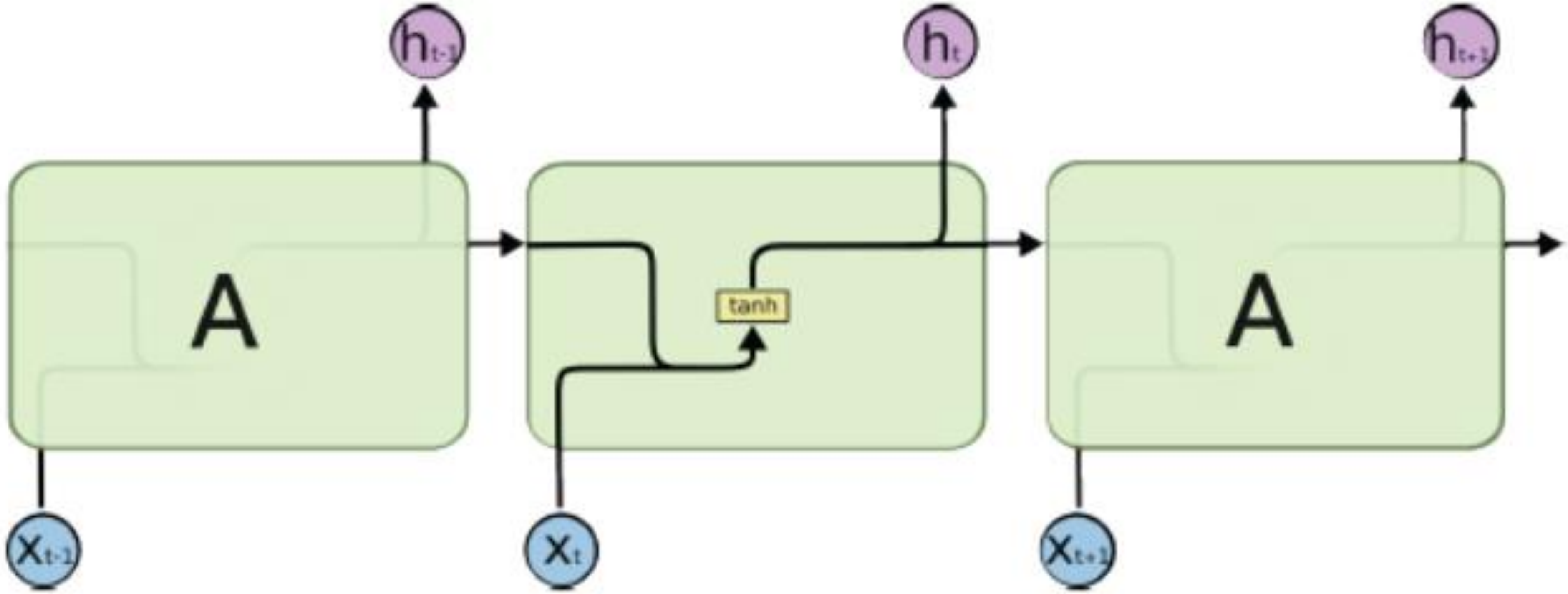
Simple (vanilla) RNN



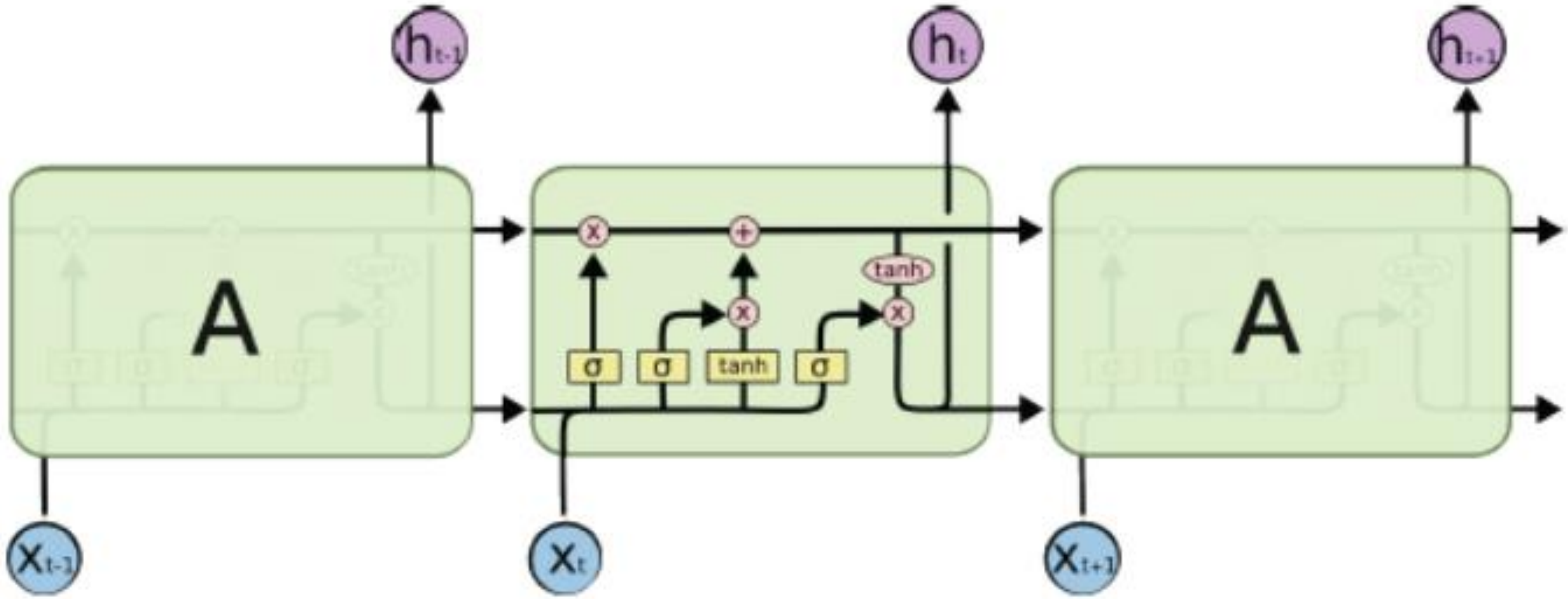
RNN unfolding: has only one state h_i



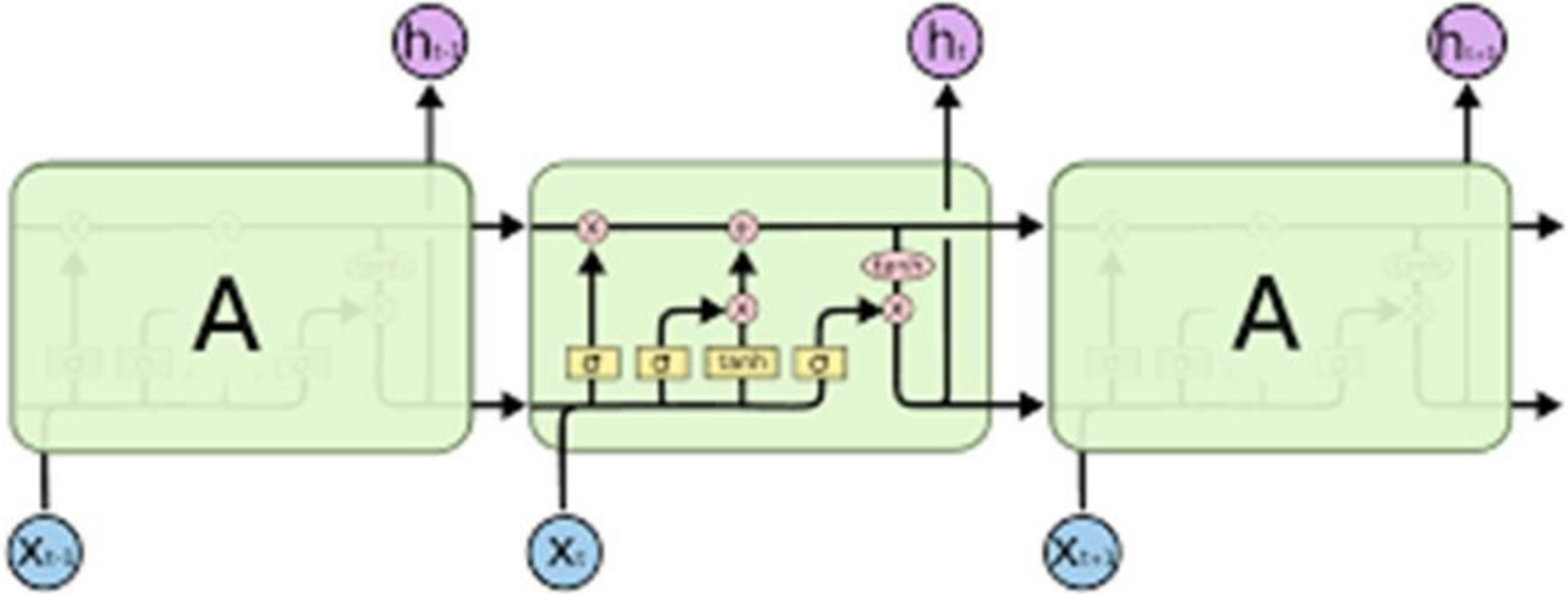
Input, output and memory



3 control gates, **forget**, **input** and **output**



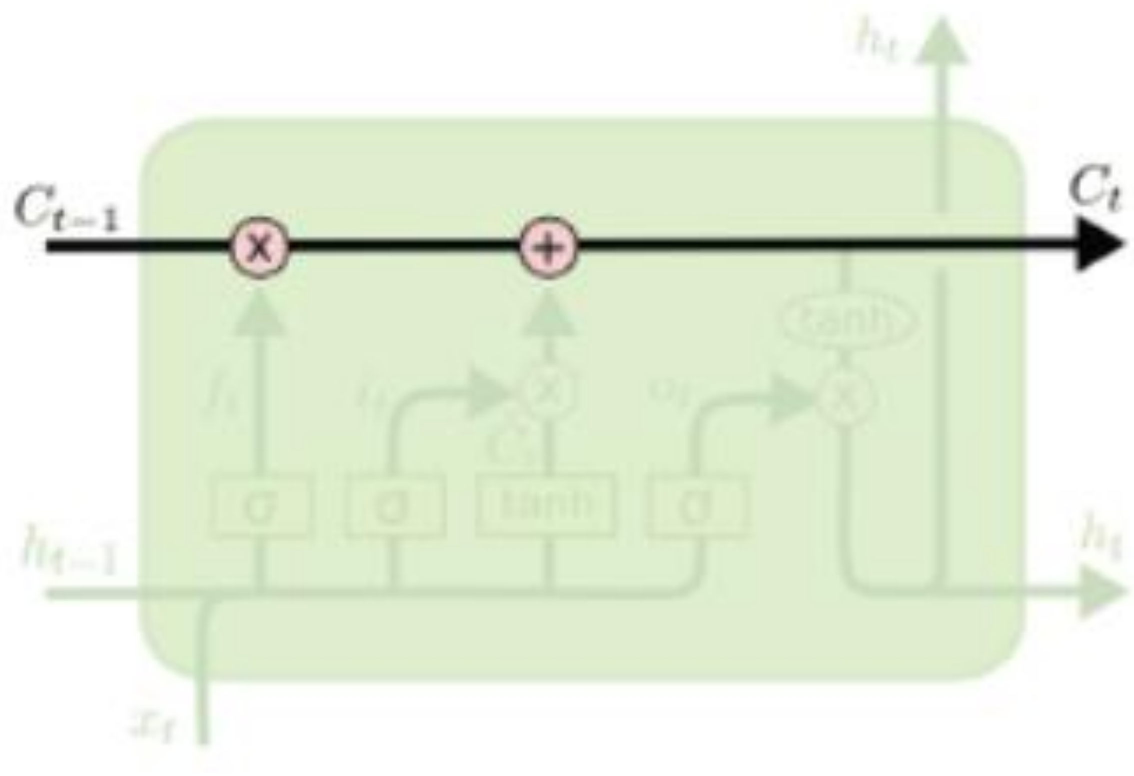
http://colah.github.io/posts/2015-08-Understanding-LSTMs/



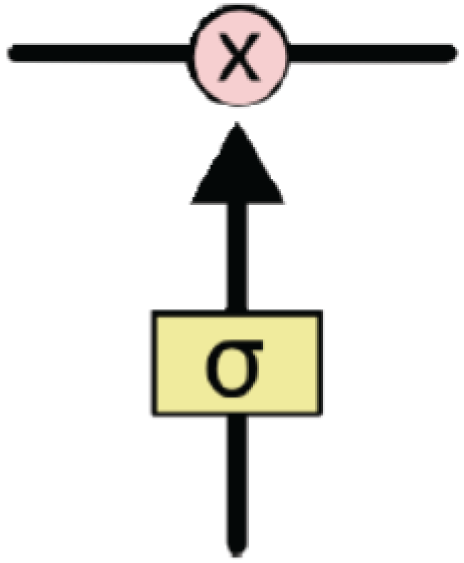
The repeating module in an LSTM contains four interacting layers.

- http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://stackoverflow.com/questions/44273249/in-keras-what-exactly-am-i-configuring-when-i-create-a-stateful-lstm-layer-wi
- http://www.deeplearningbook.org/

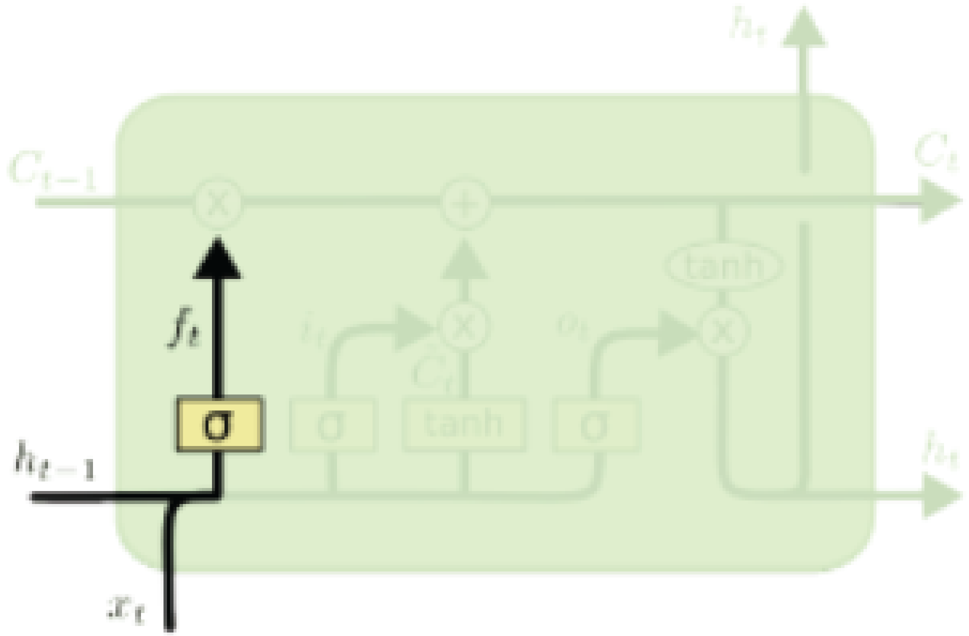
Cell state, from C_{t-1} to C_t



Control gate, Sigmoid func, control amount of information flow, from 0 to 1, component wise

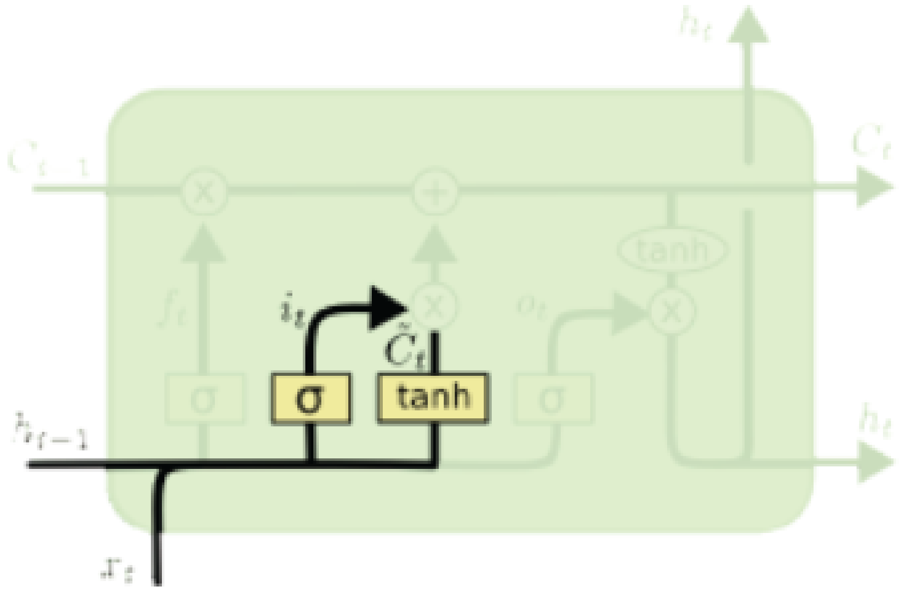


f_t, forget gate, 0- forget, 1-not forget



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

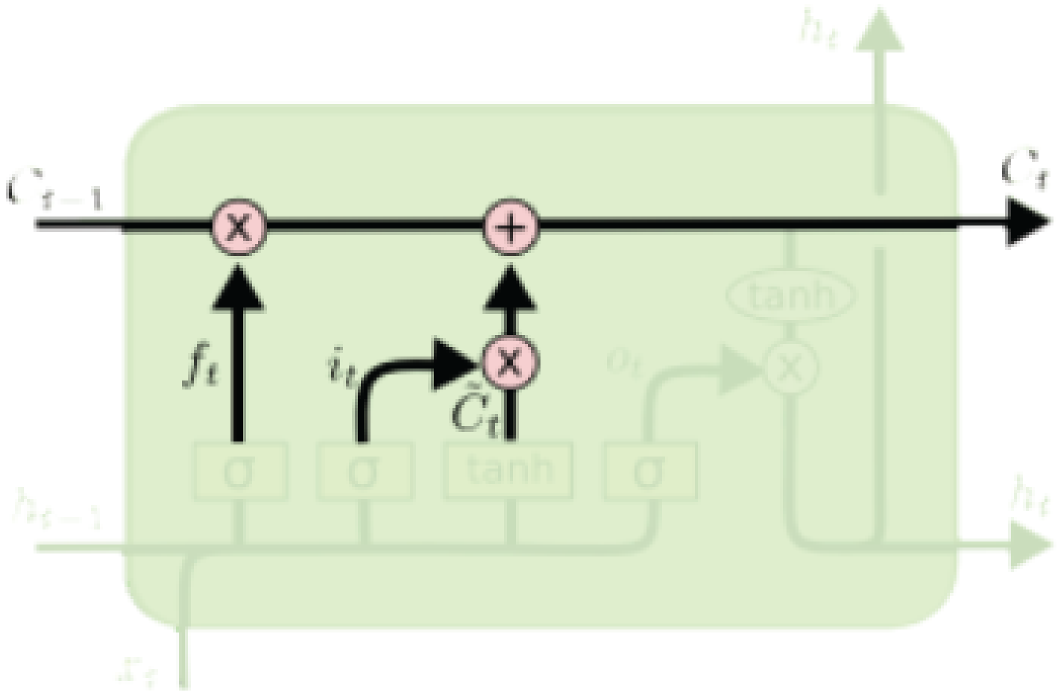
i_t Input gate, control amount of state change due to input



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

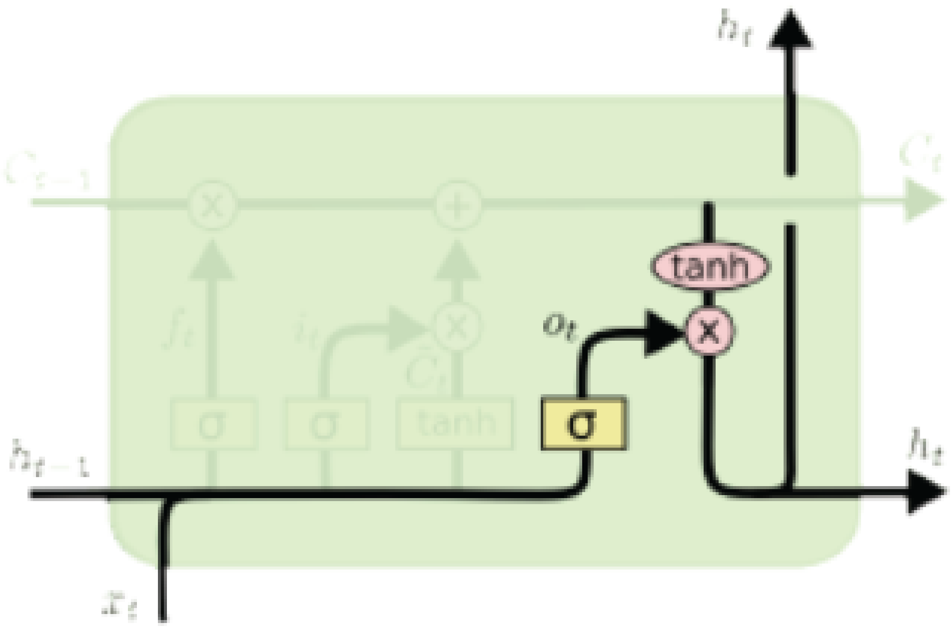
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Update the cell state



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

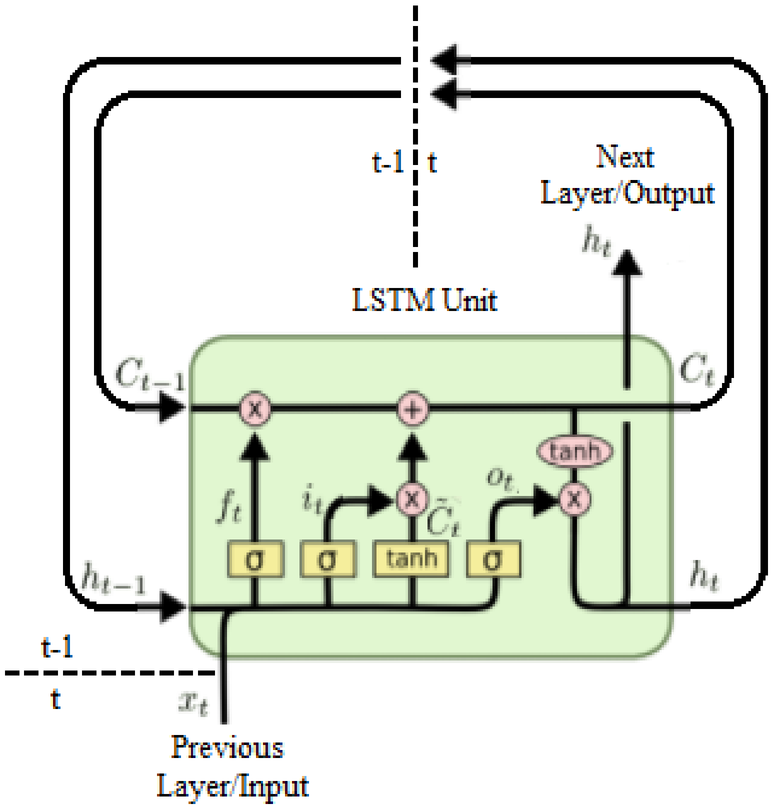
O_t, output gate, control the cell state output



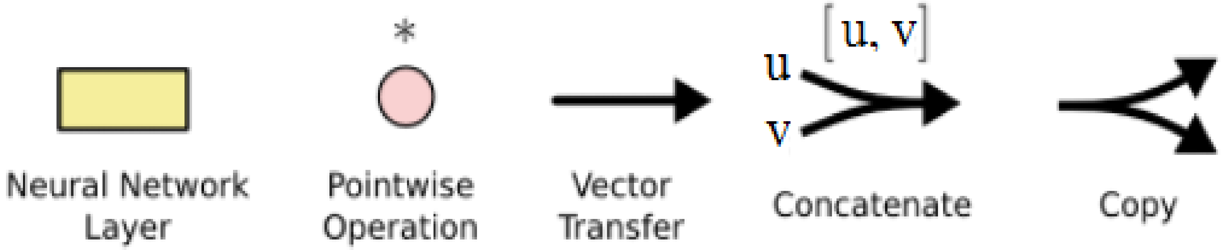
$$o_t = \sigma\left(W_o\,[h_{t-1}, x_t] + b_o\right)$$

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

Understanding LSTM Networks



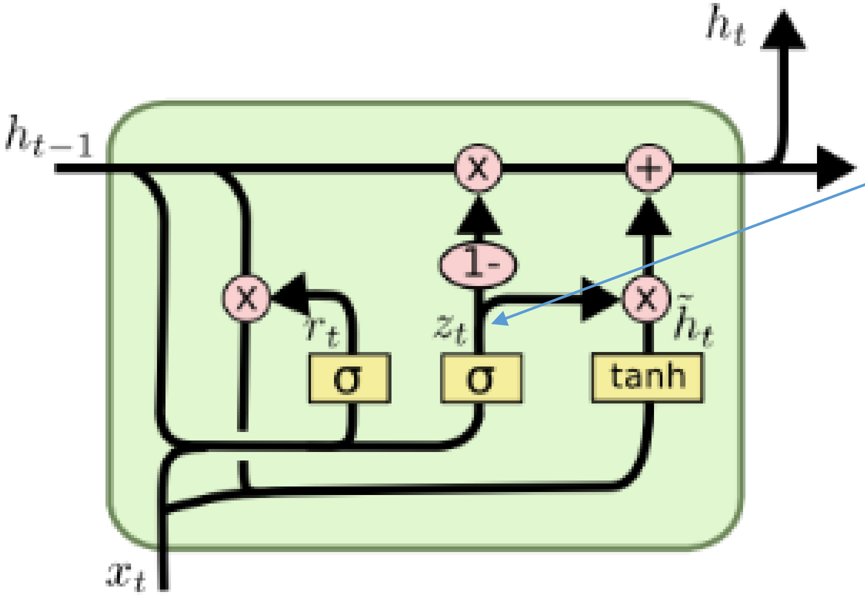
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$
$$h_t = o_t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



- The four W’s and b’s are the parameters that need to be learned from training examples.
- The training algorithm will be presented later (neglected)

- LSTM protects us from the vanishing gradient problem. Note that the cell state is copied identically from one step to the next step if the forget gate is 1 and the input gate is 0. Only the forget gate can completely keep the cell’s memory. As a result, memory can remain unchanged over a long period of time.
- Also, since the input is a tanh activation added to the current cell’s memory; this means that the cell memory doesn’t blow up and is quite stable.

Gated Recurrent Unit (GRU), a variant of LSTM by removing the cell state



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

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

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

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

State due to input & prev. state

Current state is a linear combination of the previous state and the State due to input & prev. state.

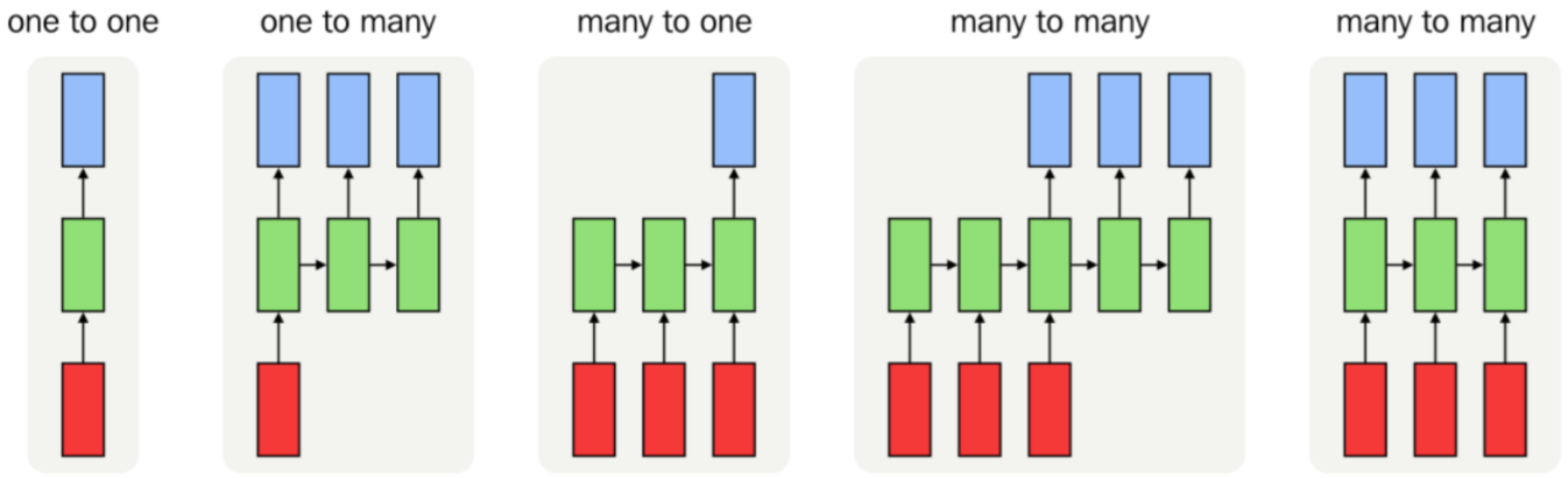
GRU, Cho, et al. (2014).

- Combines the forget and input gates into a single “update gate.”
- Merges the cell state and hidden state, and makes some other changes.
- GRU is simpler than standard LSTM models, and has been growing increasingly popular.
- z_t=1, totally forget; z_t=0 ignore the input

Applications of RNN (good for NLP and many others)

RNN used for different combinations of input and output

- RNNs are not limited to processing input of fixed size. They can be used to process sequences of different lengths or images of varied sizes.
- Red:input X, Green: state S, Blue: output O;

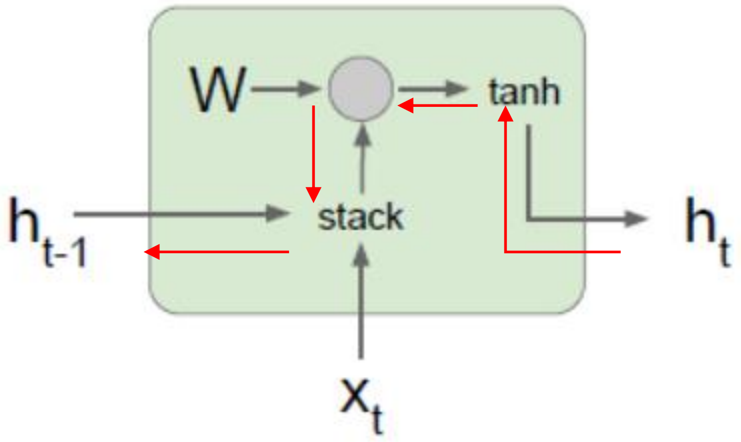


- One to many: this generates a sequence based on a single input, for example, caption generation from an image
- Many to one: output a single result based on a sequence, for example, sentiment classification from text, time-series analysis
- Many to many indirect: a sequence is encoded into a state vector, after which this state vector is decoded into a new sequence, for example, language translation
- Many to many direct: output a result for each input step, for example, frame phoneme labeling in speech recognition (in speech recognition.)

Vanilla RNN Gradient Flow

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

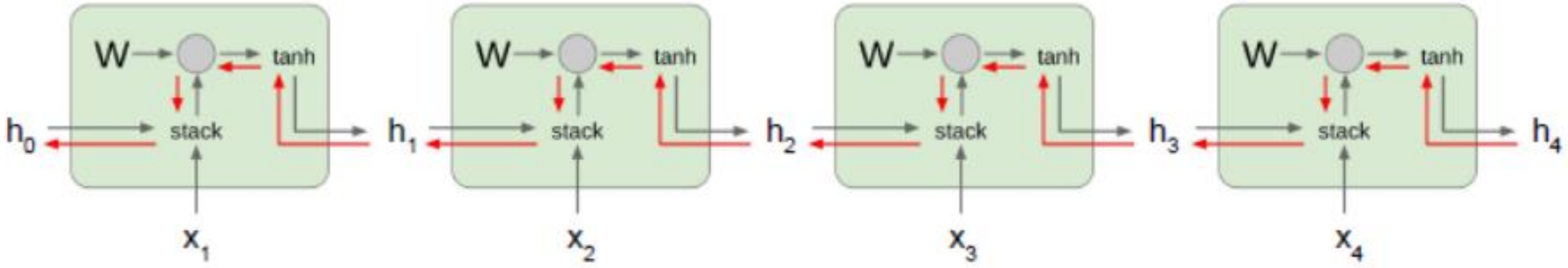
Backpropagation from h_t to h_{t-1} multiplies by W (actually W_{hh}^T)



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

Vanilla RNN Gradient Flow

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



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

grad_norm=np.sum(**grad***grad**)

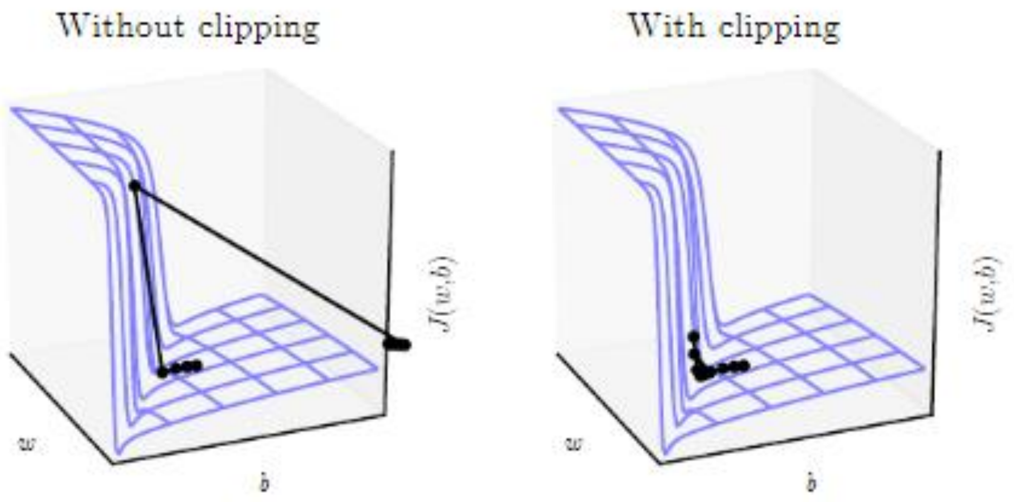
If grad_norm>threshold

grad *= (threshold /grad_norm)

To deal with the gradient exploding problem

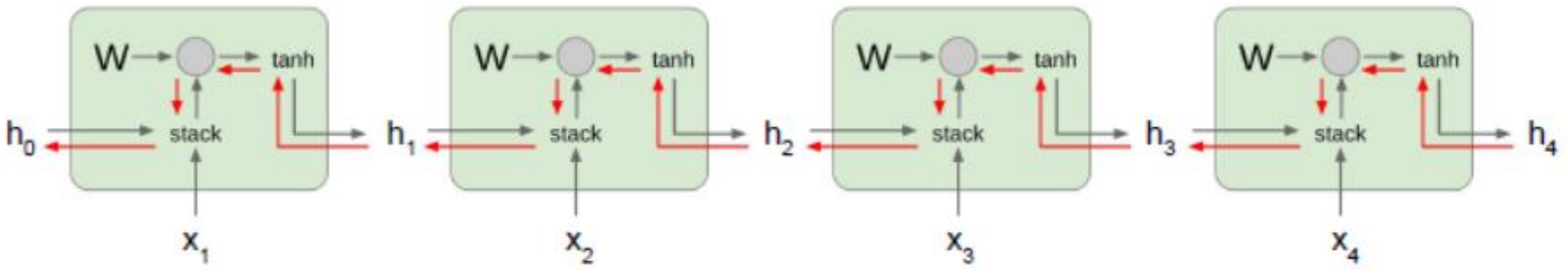
1. Gradient clipping, where we threshold the maximum value a gradient can get

$$if\ \|\boldsymbol{g}\| > \beta,\ \ g \leftarrow \frac{\beta\ \boldsymbol{g}}{\|\boldsymbol{g}\|}$$



Vanilla RNN Gradient Flow

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



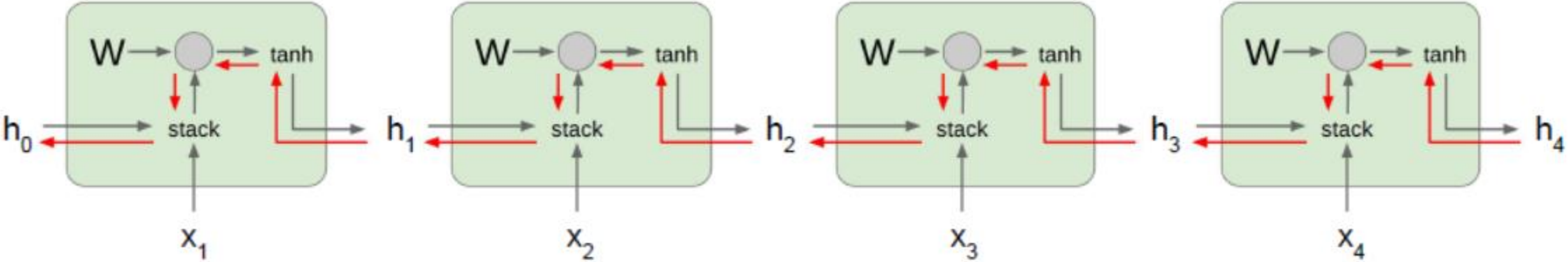
Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value < 1: Vanishing gradients

Gradient vanishing:
Change RNN structure to LSTM, GRU

Vanilla RNN Gradient Flow

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



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

→ Change RNN architecture

Long Short Term Memory (LSTM)

Vanilla RNN

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

LSTM

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

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

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

Summary

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