Introduction to Neural Network

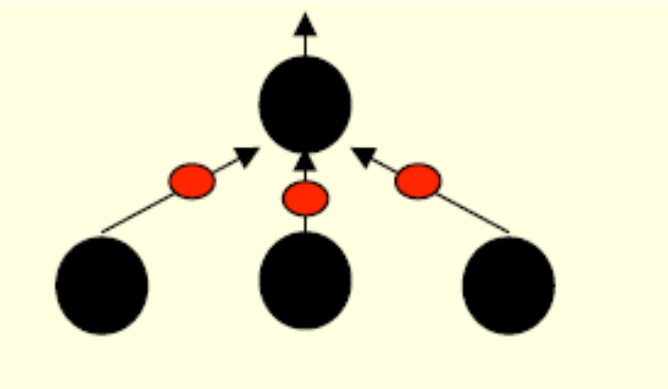
LING 572
March 5, 2019
(Some slides/graphs come from NAACL 2013 tutorial,
COLING 2014 tutorials, Yejin Choi's CSE 517 slides, etc.)

Outline

- Motivation
- Neuron and activation functions
- RNN and LSTM
- Training deep networks
- Summary

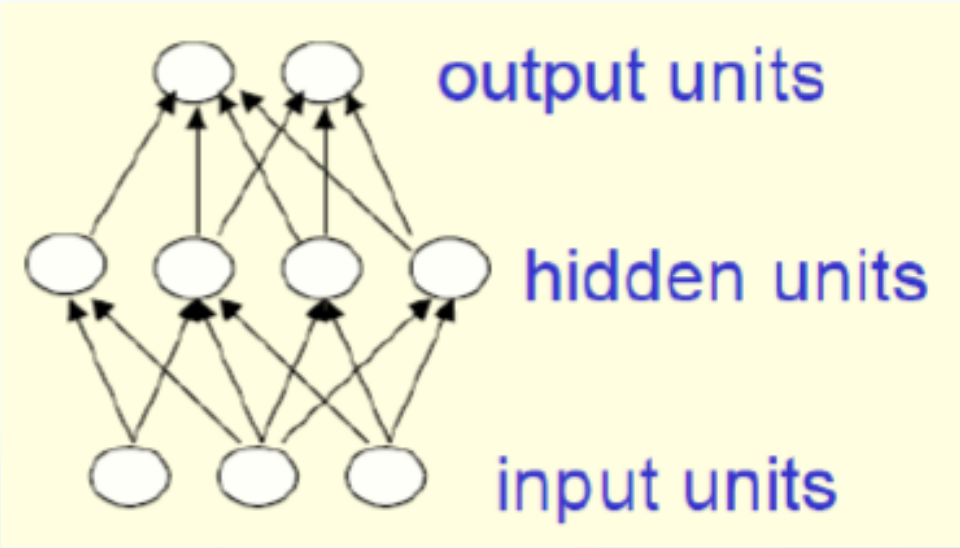
How does the brain work?

- Each neuron receives inputs from other neurons.
- Number of neurons: about 100 billion
- Number of connections per neuron: about 10 thousand
- The effect of each input line on the neuron is controlled by a weight
- The weights adapt so that the whole network learns to perform useful computations.



What is a neural network (NN)?

- A graph with multiple layers:
  - input layer
  - output layer
  - zero or more hidden layers



- Each layer has multiple nodes

  (aka neurons or units)

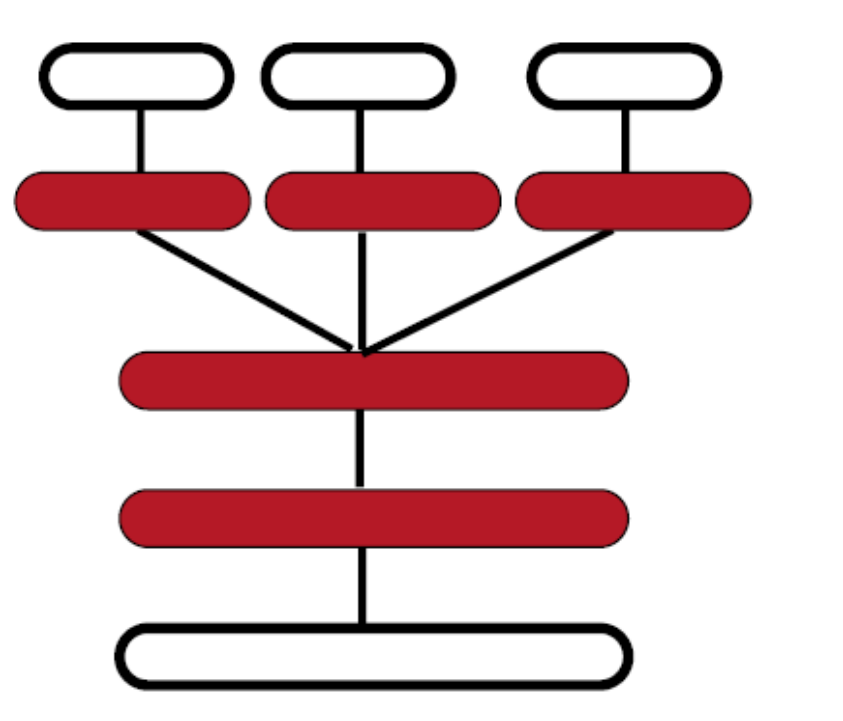
- Each arc has a weight.

Why NN?

- The need for distributed representations:
  - Previous NLP systems were fragile because of their atomic symbol representations (e.g., king vs. queen)
  - Learned word representations help enormously in NLP:
    - Distributed representations deals with the curse of dimensionality
    - We can now use bigger context (e.g., going beyond n-gram)
  - Go beyond word embedding: embeddings of phrases, sentences, documents, etc.
  - Ex: word analogy task, neural LM
- Learn representations:
  - Handcrafting features is time-consuming
  - Representation learning attempts to automatically learn good features or representations
  - Deep learning algorithms attempt to learn multiple levels of representation of increasing complexity/abstraction
- Good intermediate representations can be shared across tasks, languages, and domains:
  - Ex: multi-task learning

Multi-task learning

- Deep architectures learn good intermediate representations that can be shared across tasks
- Good representations make sense for many tasks
- Examples:
  - Multilingual parsing
  - Translating multiple language pairs
  - Word segmentations using different segmentation standards
  - Different NLP tasks



Why now?

- NN is an old algorithm, and was not successful before 2006.

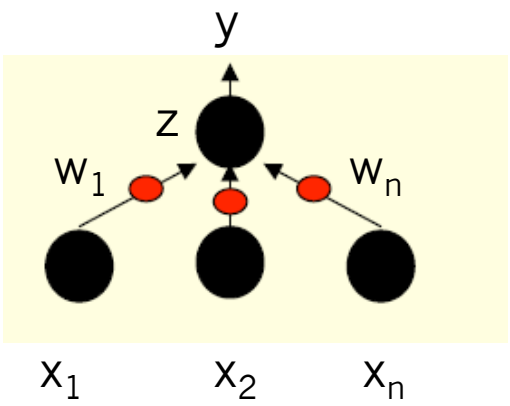
- What has changed?
  - New methods for unsupervised pre-training have been developed
  - More efficient parameter estimation methods
  - Better understanding of model regularization

➔ NN has produced state-of-the-art results on many NLP tasks

Outline

- Motivation
- Neuron and activation functions
- RNN and LSTM
- Training deep networks
- Summary

A neuron



$$z = b + \sum_i x_i w_i$$

b is called the bias

$$y = f(x)$$

f is called an activation function

Activation functions: $y=f(z)$

$z = b + \Sigma_i x_i w_i$

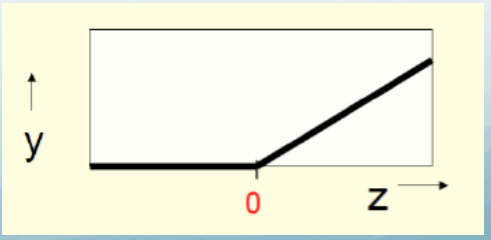
- Linear neuron: $y=z$
- Binary threshold neuron:

$$y = \begin{cases} 1 \text{ if } z \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

- Rectified activation function:

$y=y=\max(0,z)$

$$y = \begin{cases} z \text{ if } z > 0 \\ 0 \text{ otherwise} \end{cases}$$

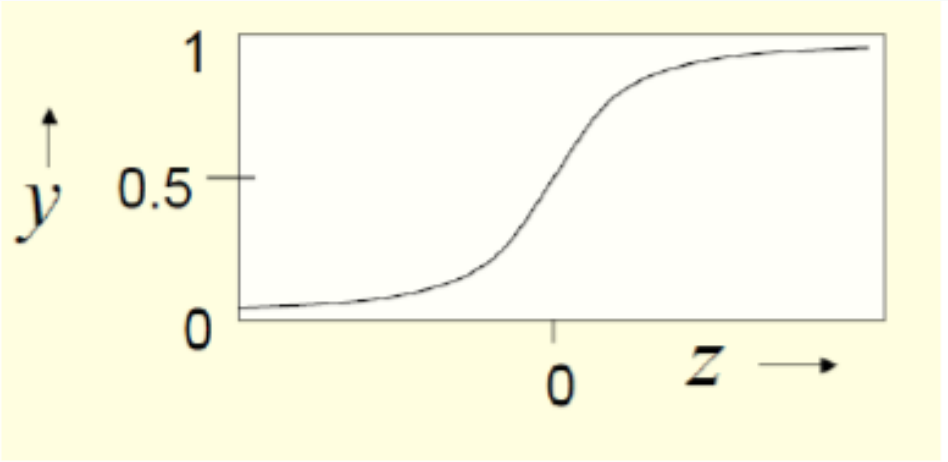


Sigmoid function

$y = f(z) = 1/1+e^{-z}$

Good derivative property:

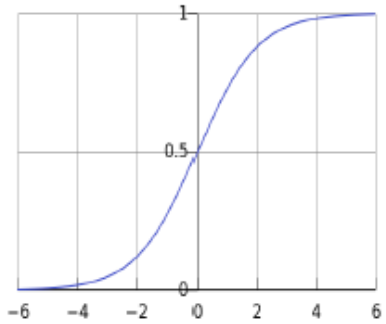
$f'(z)=f(z)(1-f(z))$



sigmoid vs. tanh

logistic ("sigmoid")

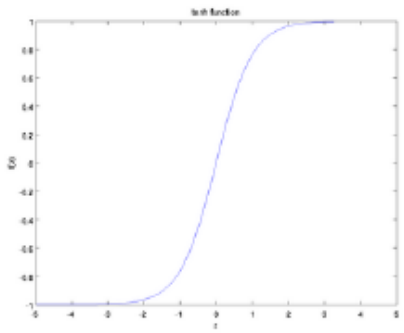
$$f(z) = \frac{1}{1 + \exp(-z)}.$$



$$f'(z) = f(z)(1 - f(z))$$

tanh

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



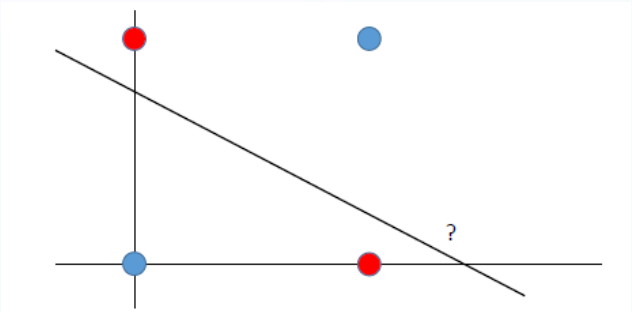
$$f'(z) = 1 - f(z)^2$$

tanh is just a rescaled and shifted sigmoid $\tanh(z) = 2\text{logistic}(2z) - 1$

tanh is what is most used and often performs best for deep nets

Activation function

- There are many activation functions.
- Which one to choose?
  - non-linearity is crucial
  - prefer ones with nice derivatives which make learning easy
- Commonly used: tanh



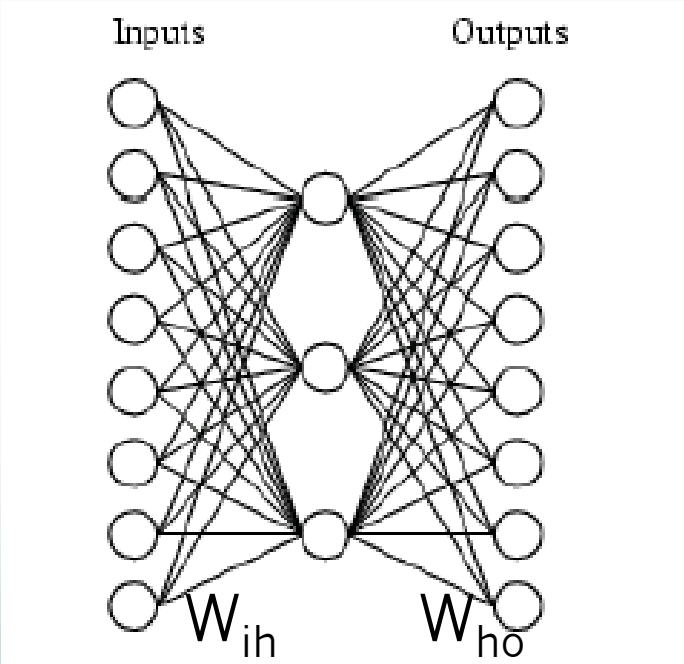
There is no linear classifier that can solve the XOR p

Hidden layer

- Single neuron: $y=f(b+\Sigma_i w_i x_i)$
- 1-hidden layer:

$y_h=f(W_{ih}x)$

$Y_o=f(W_{ho}y_h)$
$=f(W_{ho}(W_{ih}x))$



If $z=[z_1,z_2,.....z_n]$
Then $f(z)=[f(z_1),f(z_2),...,f(z_n)]$

Why "representation learning"?

- MaxEnt (multinomial logistic regression):

$$y = \text{softmax}(w \cdot f(x, y))$$

You design the feature vector

- NNs:

$$y = \text{softmax}(w \cdot \sigma(Ux))$$

$$y = \text{softmax}(w \cdot \sigma(U^{(n)}(...\sigma(U^{(2)}\sigma(U^{(1)}x)))))$$

Feature representations are "learned" through hidden layers

softmax function

- Softmax function turns a vector of arbitrary real values into a probability distribution (i.e., the values are in [0, 1] and sum to one)

- Let $z=[z\downarrow 1 , z\downarrow 2 , ..., z\downarrow n]$

softmax(z) = $[y\downarrow 1 , y\downarrow 2 , ..., y\downarrow n]$,

where $y\downarrow j = e\uparrow z\downarrow j \ / \sum k\uparrow \square e\uparrow z\downarrow k$

- In MaxEnt, $z=w\, f(x,y)$; *that is*, $z\downarrow j = w\, f(x, c\downarrow j) = \sum i\uparrow \square w\downarrow i \, f\downarrow i (x, c\downarrow j)$

- In NN, the softmax function is often used as the final layer of an NN for classification.

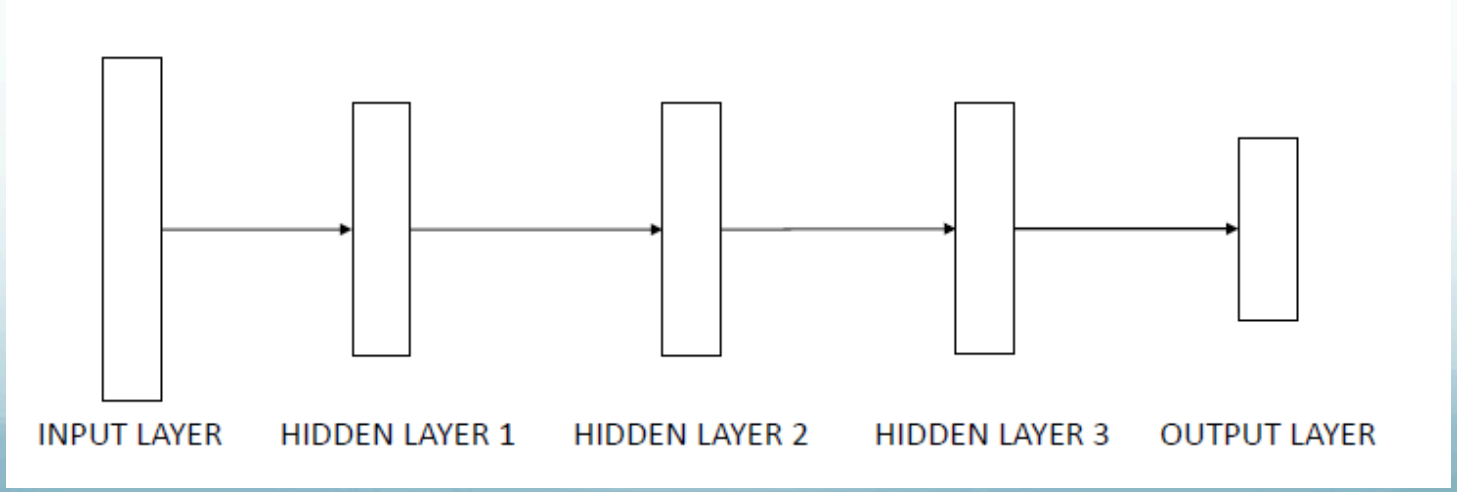
- For more detail, see https://en.wikipedia.org/wiki/Softmax_function

Softmax Function

- Softmax function turns a vector of arbitrary real values into a probability distribution (i.e., the values are in [0, 1] and sum to one)
- Let $z=[z_1,z_2,\ldots,z_n]$
- $\text{softmax}(z) = [y_1,y_2,\ldots,y_n]$,
- Where $y_j=e^{Zj}/\Sigma_k e^{Zk}$
- In MaxEnt, $z=wf(x,y)$; this $z_j=wf(x,c_j)=\Sigma_i w_i f_i(x,c_j)$
- In NN, the softmax function is often used as the final layer of an NN for classification.
- •For more detail, see Wikipedia Softmax_function

Deep learning

- Deep nets have many hidden layers in the model.
- Deep learning aims to learn patterns that cannot be learned efficiently with shallow models.
- When we try to learn a complex function that is a composition of simpler functions, it may be beneficial to use deep nets.
- Many proposed deep models do not learn anything more than what a shallow model can learn. Beware the hype.



Outline

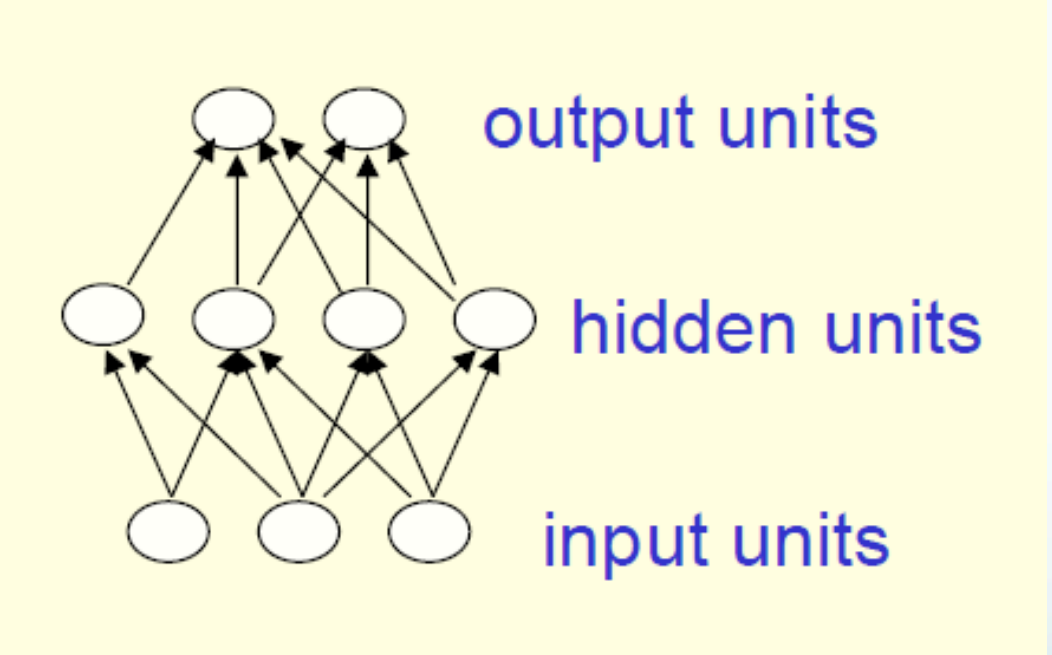
- Motivation
- Neuron and activation functions
- RNN and LSTM
- Training deep networks
- Summary

Feed-forward neural network

Simplest type of NN:

- The first layer is the input and the last layer is the output
- If there is more than one hidden layer, we call them deep NN

Training: learn the weights on the arcs, using back propagation.



Recurrent neural network (RNN)

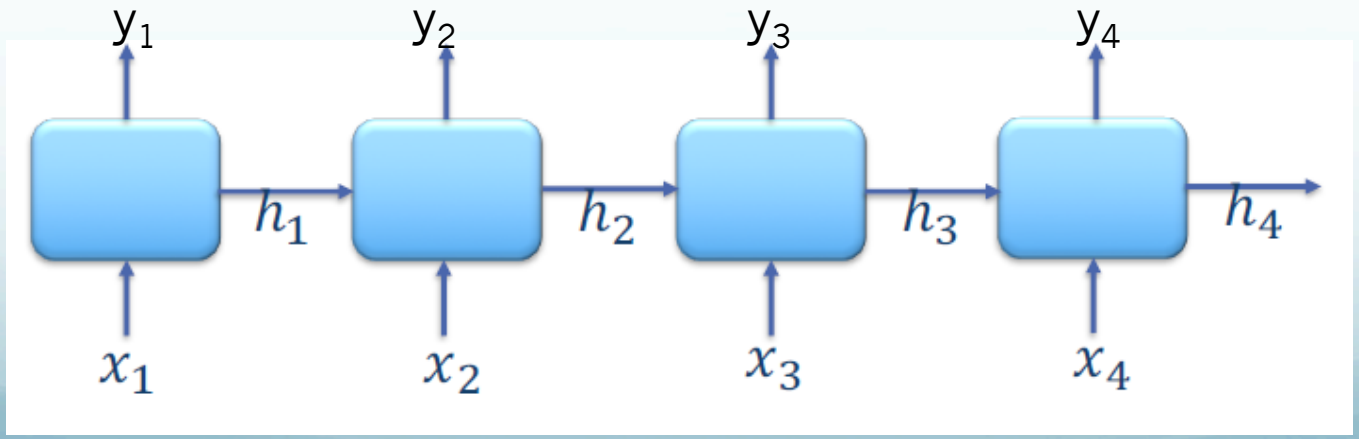
- Each RNN unit computes a new hidden state using the previous state and a new input:

$$h_t = f(x_t, h_{t-1})$$

- Each RNN unit (optionally) makes an output using the current state:

$$y_t = \text{softmax}(V h_t)$$

- Parameters are shared (tied) across all RNN units



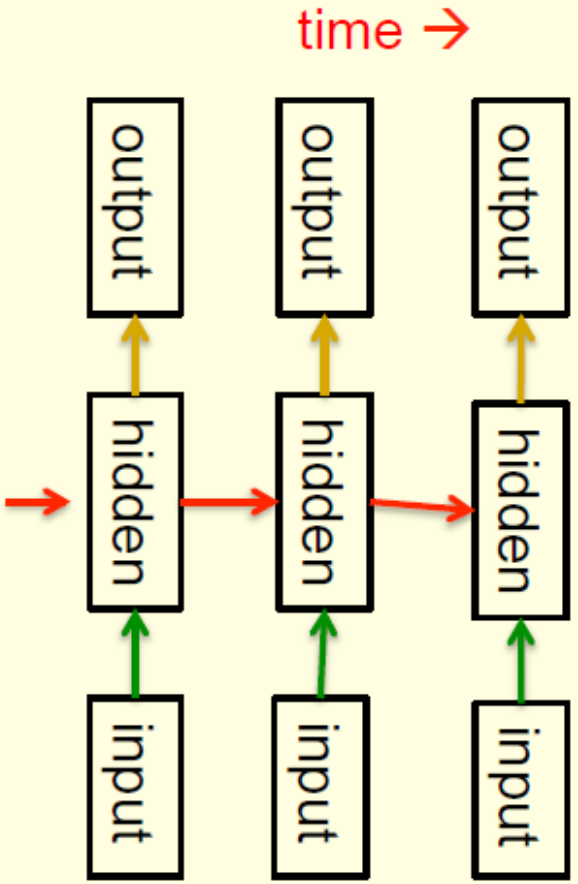
RNN

- Hidden states are continuous vectors:
  - Can represent rich information (e.g., context)
  - Possibly the entire history from the beginning

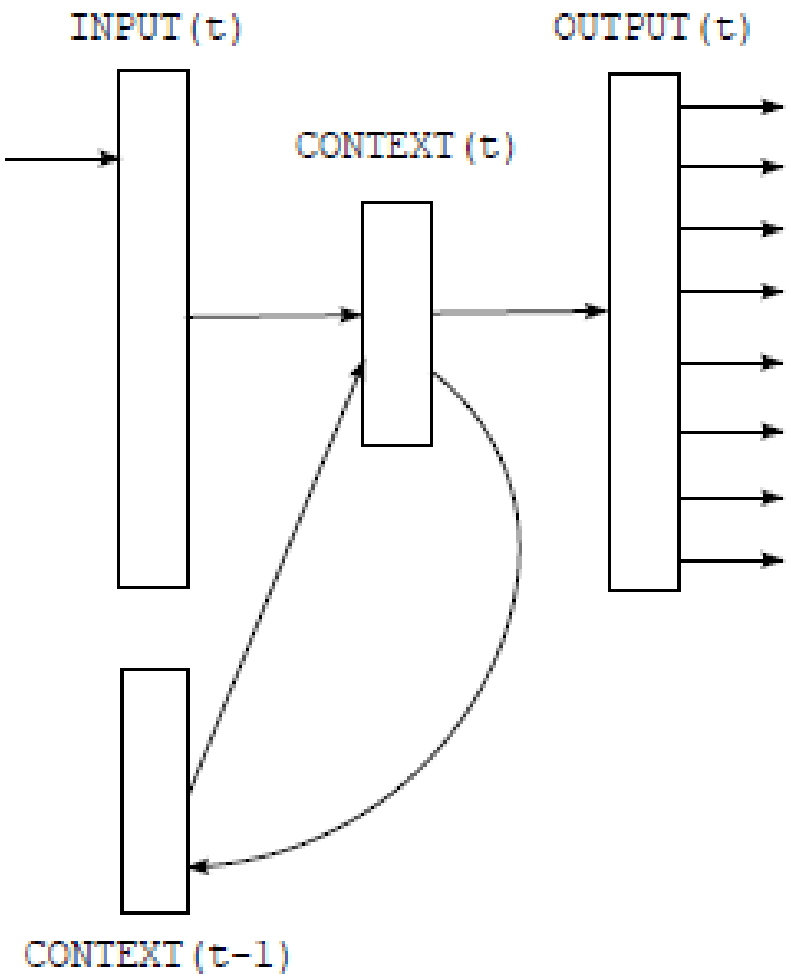
- One of the most commonly used NN types in the NLP field right now:
  - Ex: sequence labeling (e.g., POS tagging)
  - Ex: seq-to-seq (e.g., MT)

RNNs for modeling sequences

- Recurrent neural networks are a very natural way to model sequential data:
  - They are equivalent to very deep nets with one hidden layer per time slice.
  - Except that they use the same weights at every time slice and they get input at every time slice.
- They have the ability to remember information in their hidden state for a long time.
  - But its very hard to train them to use this potential.



Recurrent Neural Network LM (Mikolov et al., 2010)

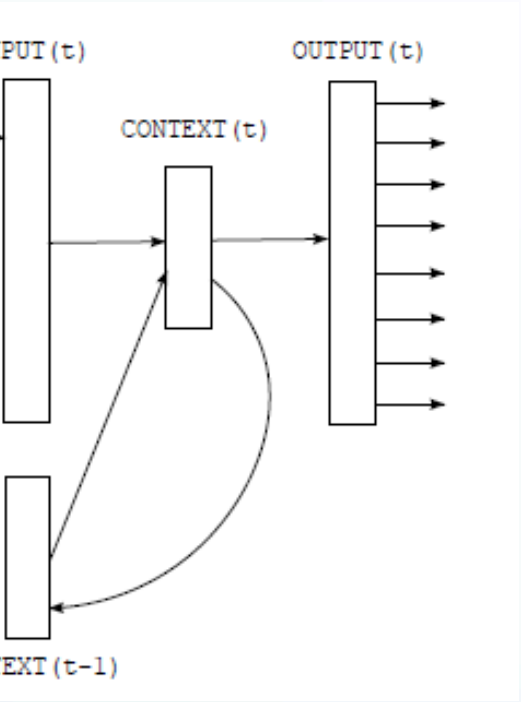


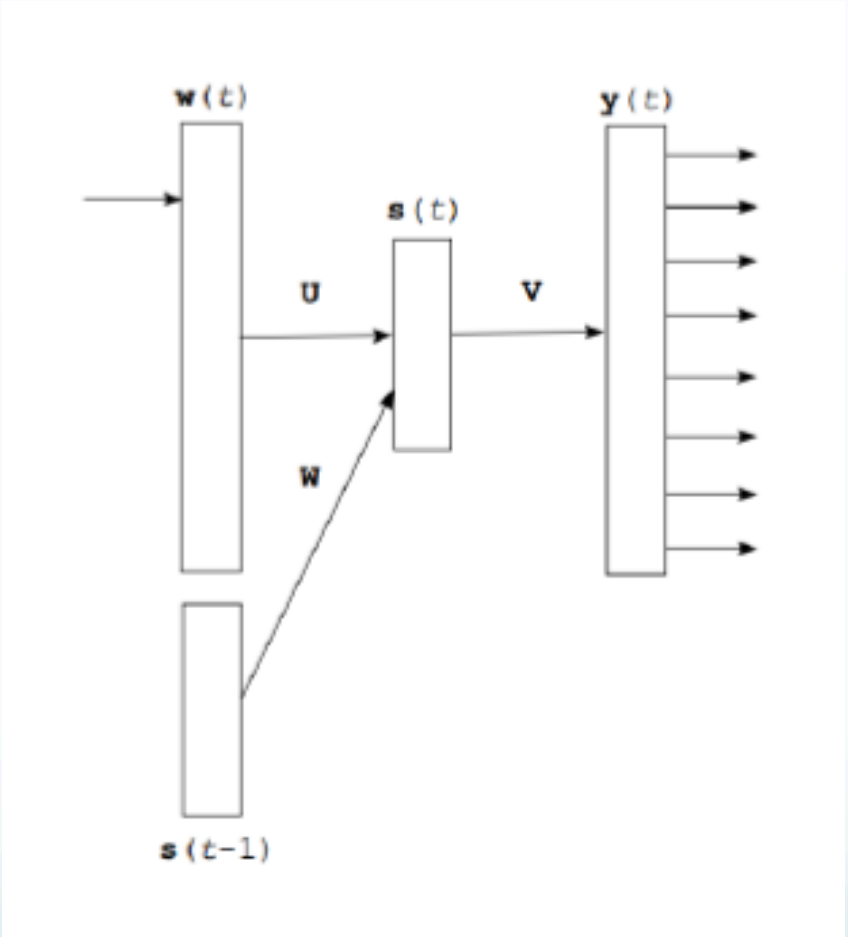
Input layer is w(t) and s(t-1)
Hidden layer represents context s(t)
Output layer is the prob distribution w(t+1)

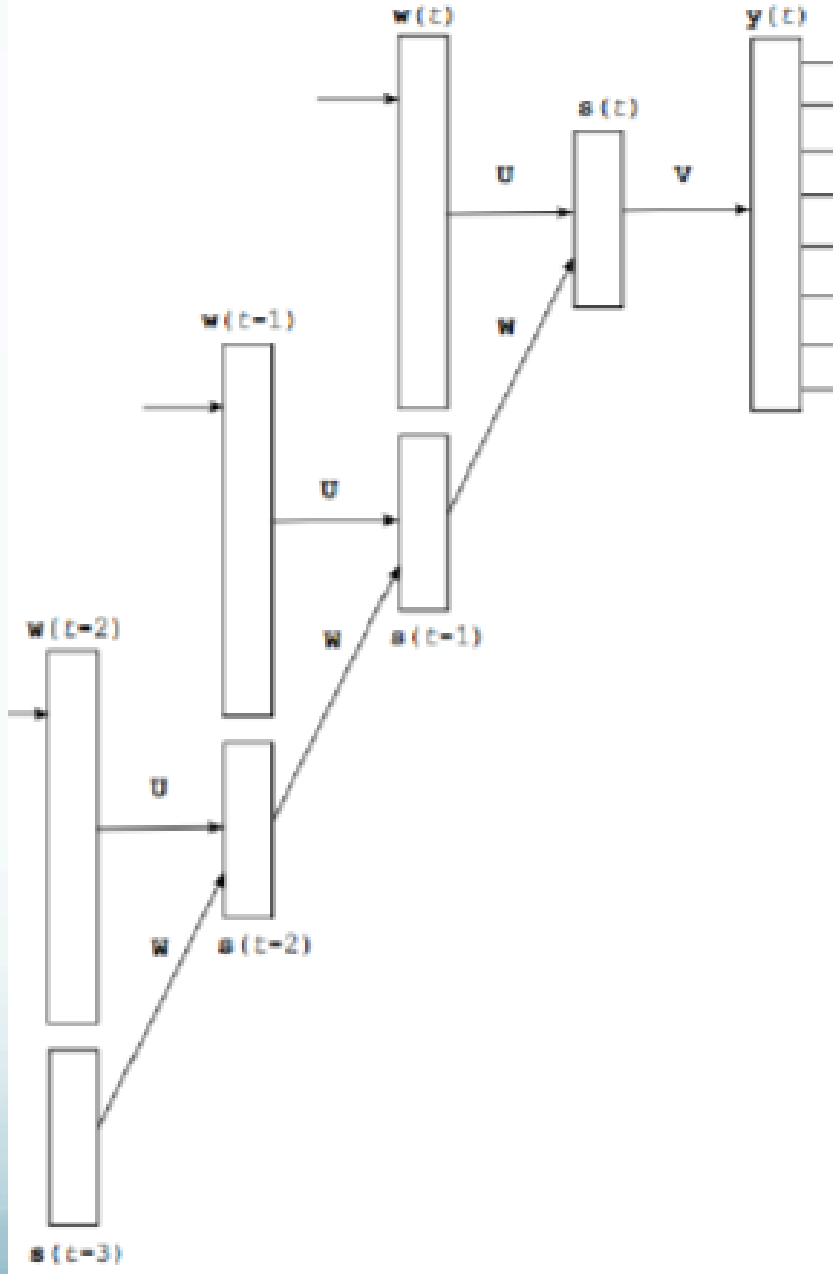
W(t) uses 1-to-N coding.

Training is slow
➔ Cannot use large amount of data

Unfold RNN in time







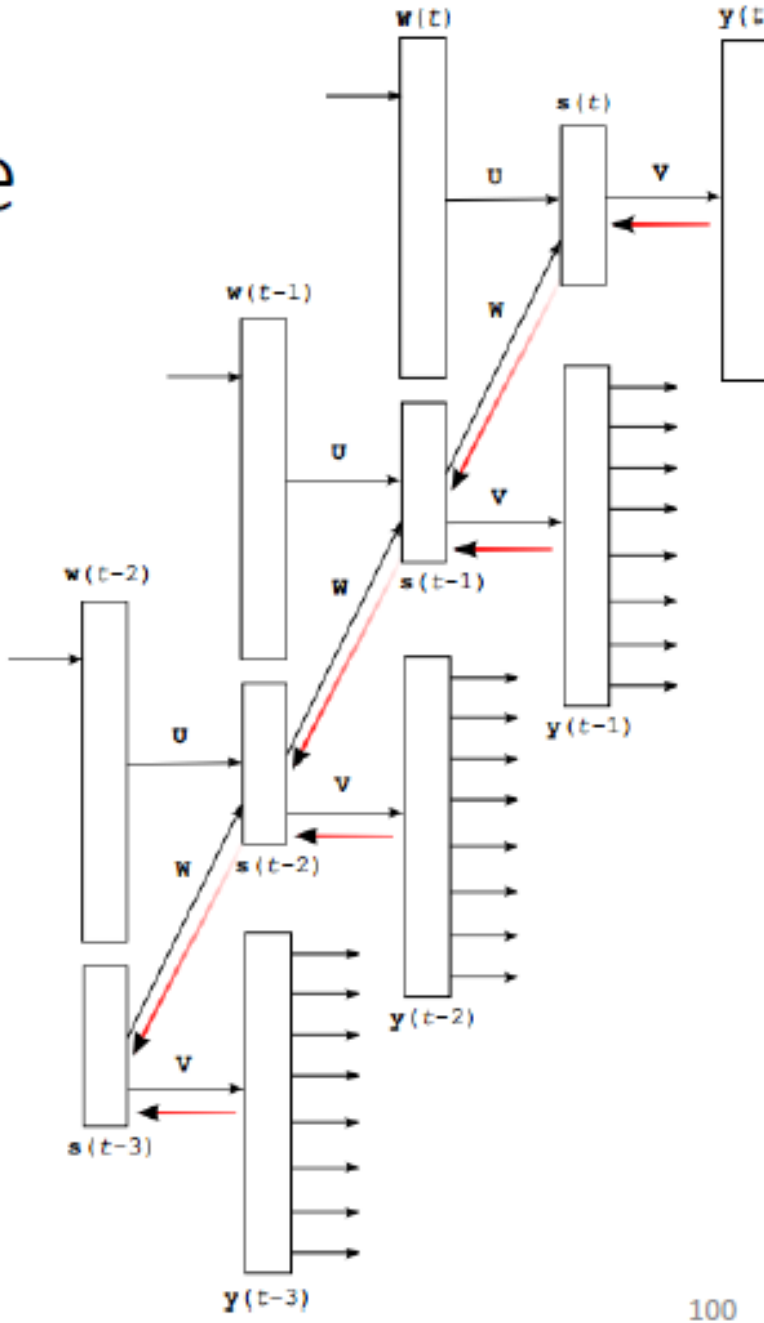
RNN training: Backprop through time (BPTT)

- "through time": Use unfolded RNN

- Similar to backprop with non-recurrent NNs:
  - Backprop gradients of the parameters of each unit as if they are different parameters
  - But when updating the parameters using the gradient, use the average gradients throughout the entire chain of units

Backpropagation through time

- We train the unfolded RNN using normal backpropagation + SGD
- In practice, we limit the number of unfolding steps to 5 – 10
- It is computationally more efficient to propagate gradients after few training examples (batch mode)

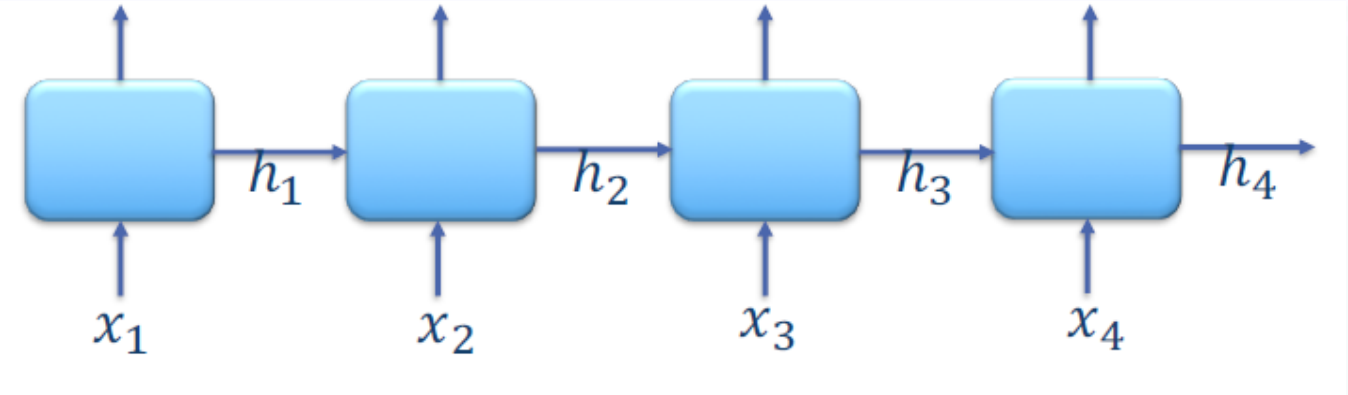


Vanishing / exploding gradients

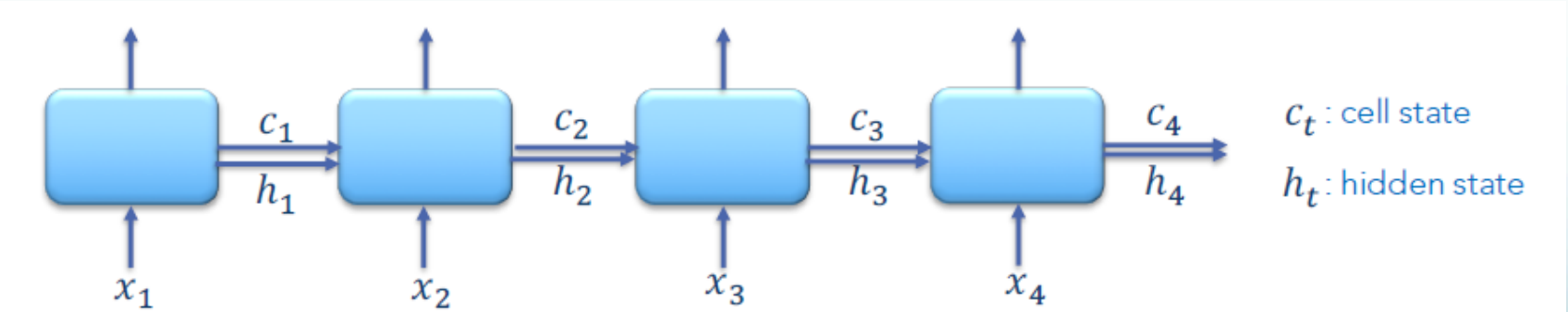
- Gradients go through multiple layers and the multiplicative effect tends to lead to:
  - Vanishing gradients: gradients decrease and quickly approach tiny values
  - Exploding gradients: gradients become huge and quickly forget what has been learned so far
- Practical solutions:
  - Network architecture: e.g., LSTM
  - Numerical operations: e.g., clip the gradient values

LSTM (Long Short-term Memory Network)

RNN:



LSTM:



LSTM

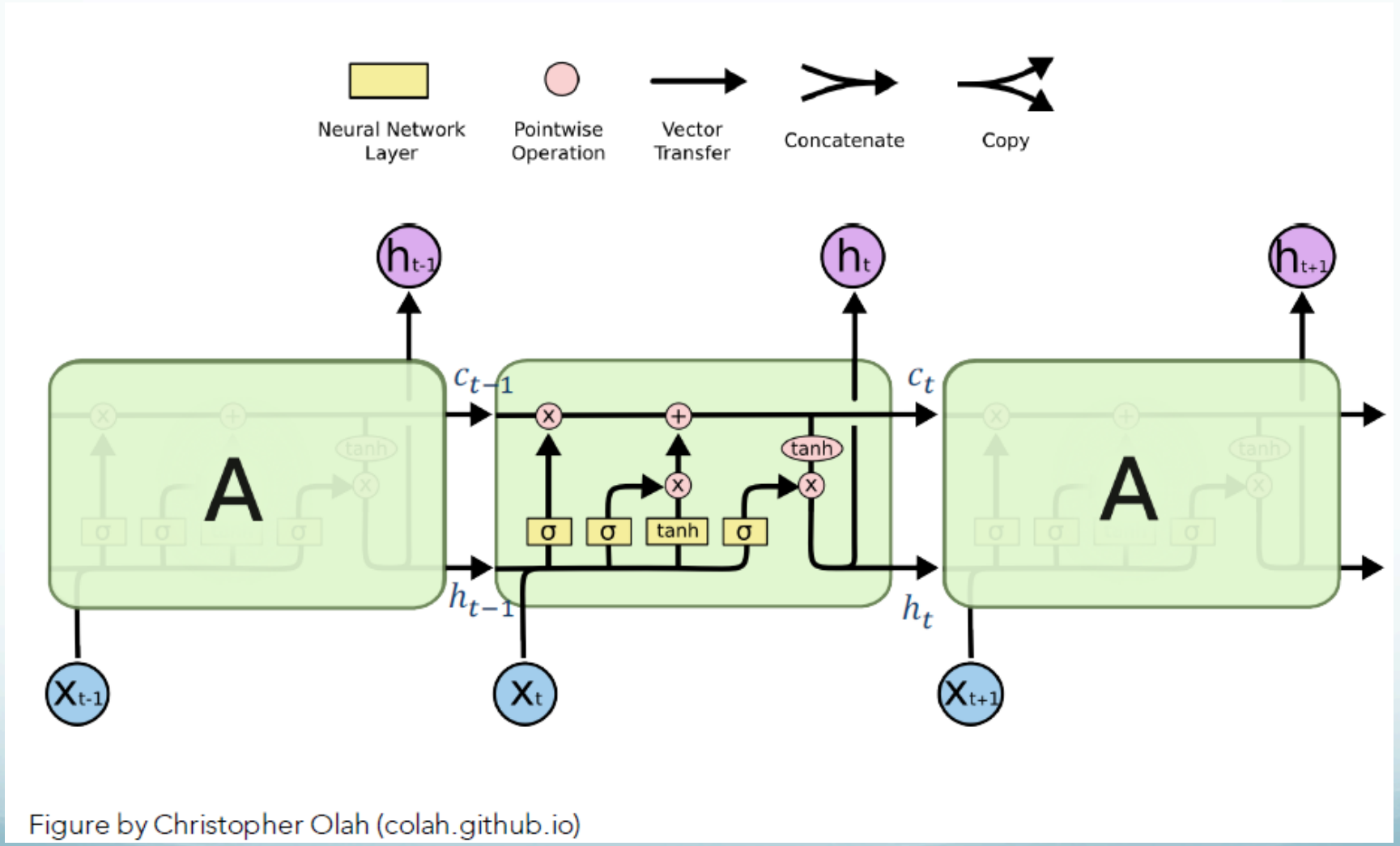
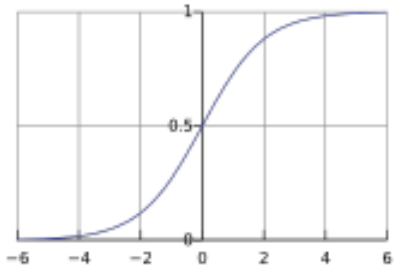


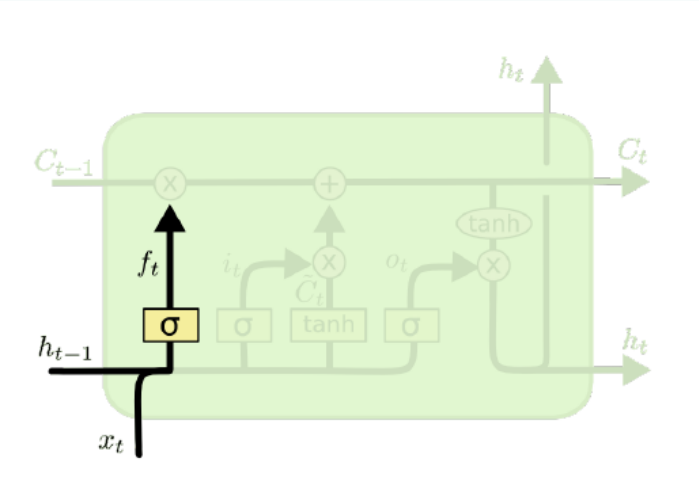
Figure by Christopher Olah (colah.github.io)

sigmoid:
[0,1]

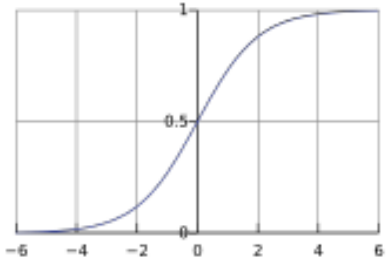


Forget gate: forget the past or not

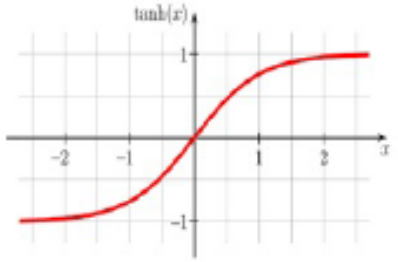
$$f_t = \sigma(U^{(f)} x_t + W^{(f)} h_{t-1} + b^{(f)})$$



sigmoid:
[0,1]



tanh:
[-1,1]



Forget gate: forget the past or not

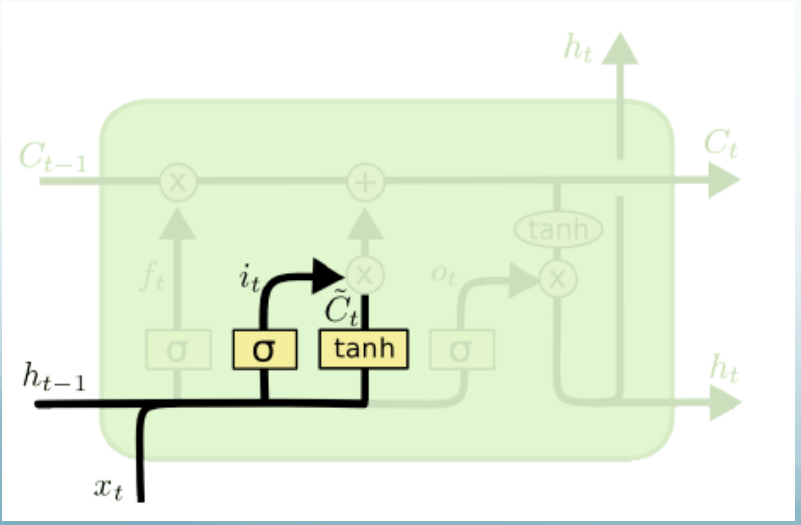
$$f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)})$$

Input gate: use the input or not

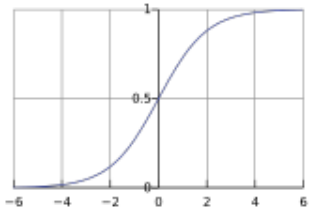
$$i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)})$$

New cell content (temp):

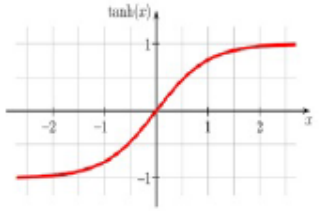
$$\tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)})$$

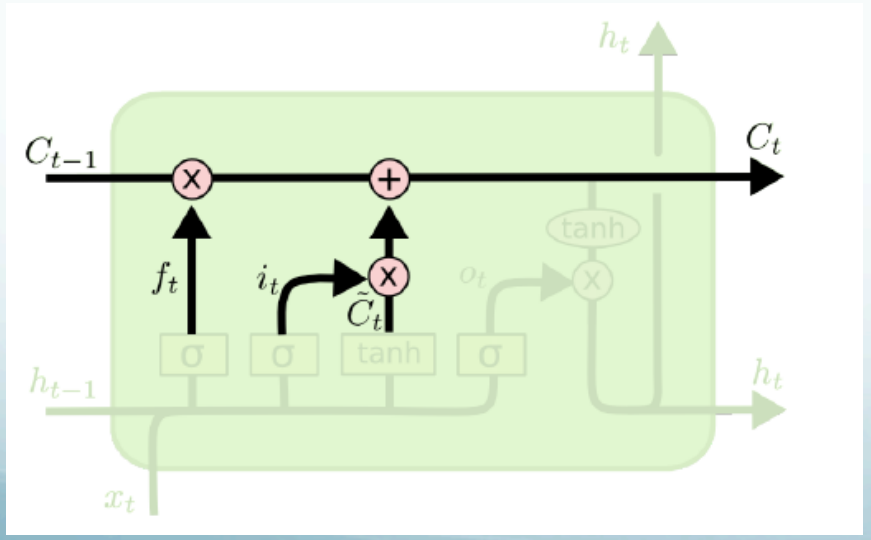


sigmoid:
[0,1]



tanh:
[-1,1]





Forget gate: forget the past or not

$$f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)})$$

Input gate: use the input or not

$$i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)})$$

New cell content (temp):

$$\tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)})$$

New cell content:

- mix old cell with the new temp cell

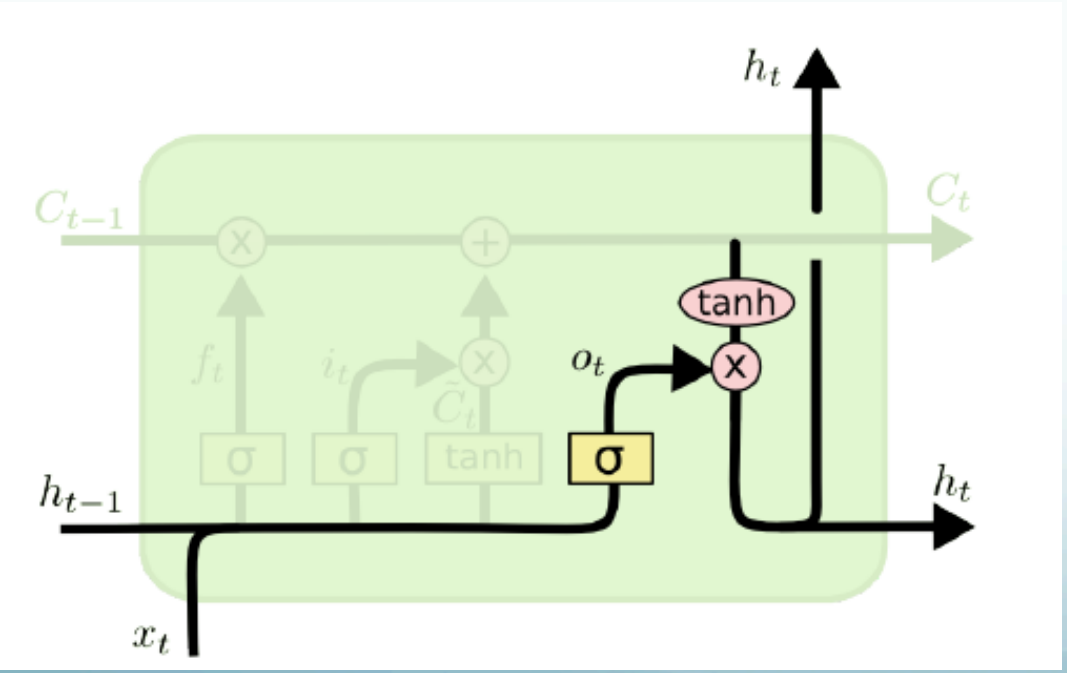
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

Output gate: output from the new cell or not

$$o_t = \sigma(U^{(o)}x_t + W^{(o)}h_{t-1} + b^{(o)})$$

Hidden state:

$$h_t = o_t \circ \tanh(c_t)$$



Forget gate: forget the past or not

$$f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)})$$

Input gate: use the input or not

$$i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)})$$

New cell content (temp):

$$\tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)})$$

New cell content:

- mix old cell with the new temp cell

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

Forget gate: forget the past or not

$$f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)})$$

Input gate: use the input or not

$$i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)})$$

Output gate: output from the new cell or not

$$o_t = \sigma(U^{(o)}x_t + W^{(o)}h_{t-1} + b^{(o)})$$

New cell content (temp):

$$\tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)})$$

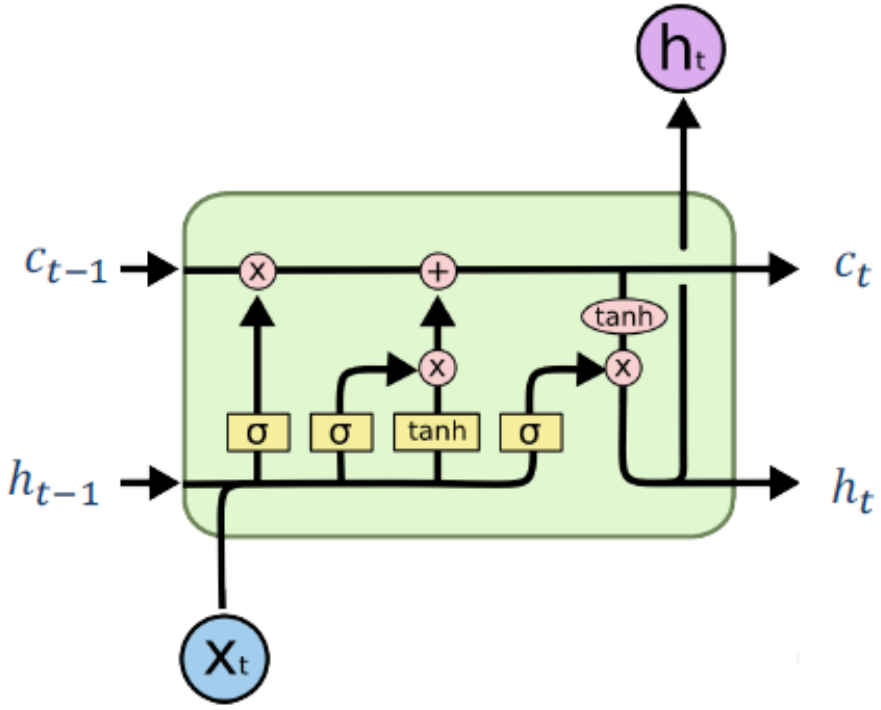
New cell content:

- mix old cell with the new temp cell

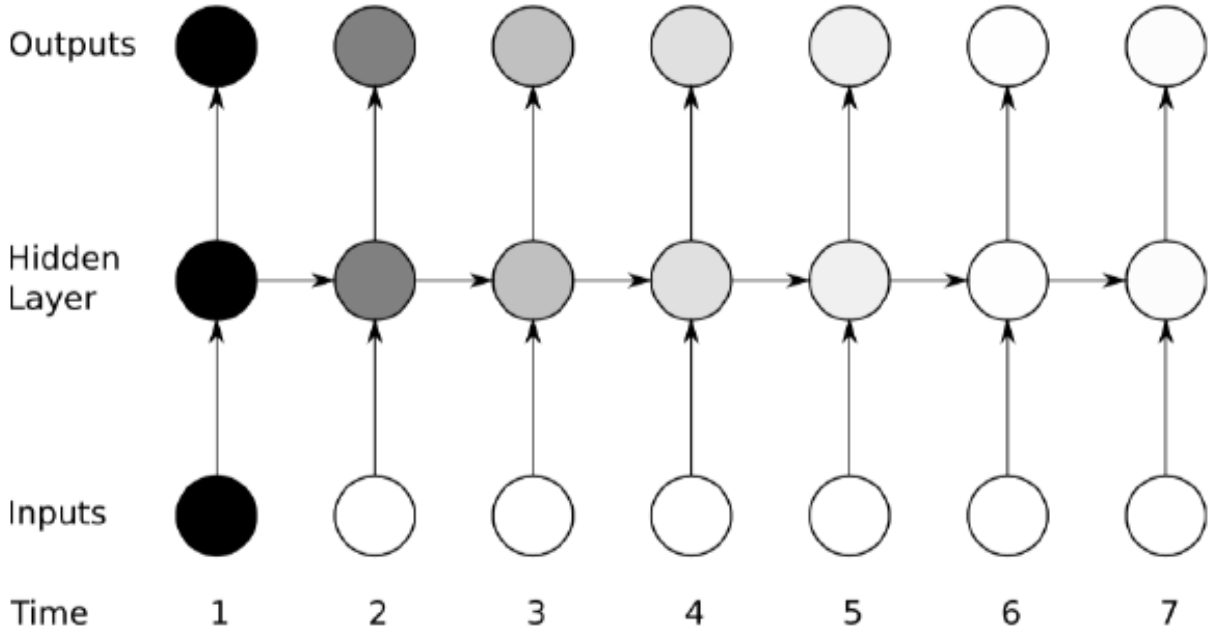
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

Hidden state:

$$h_t = o_t \circ \tanh(c_t)$$



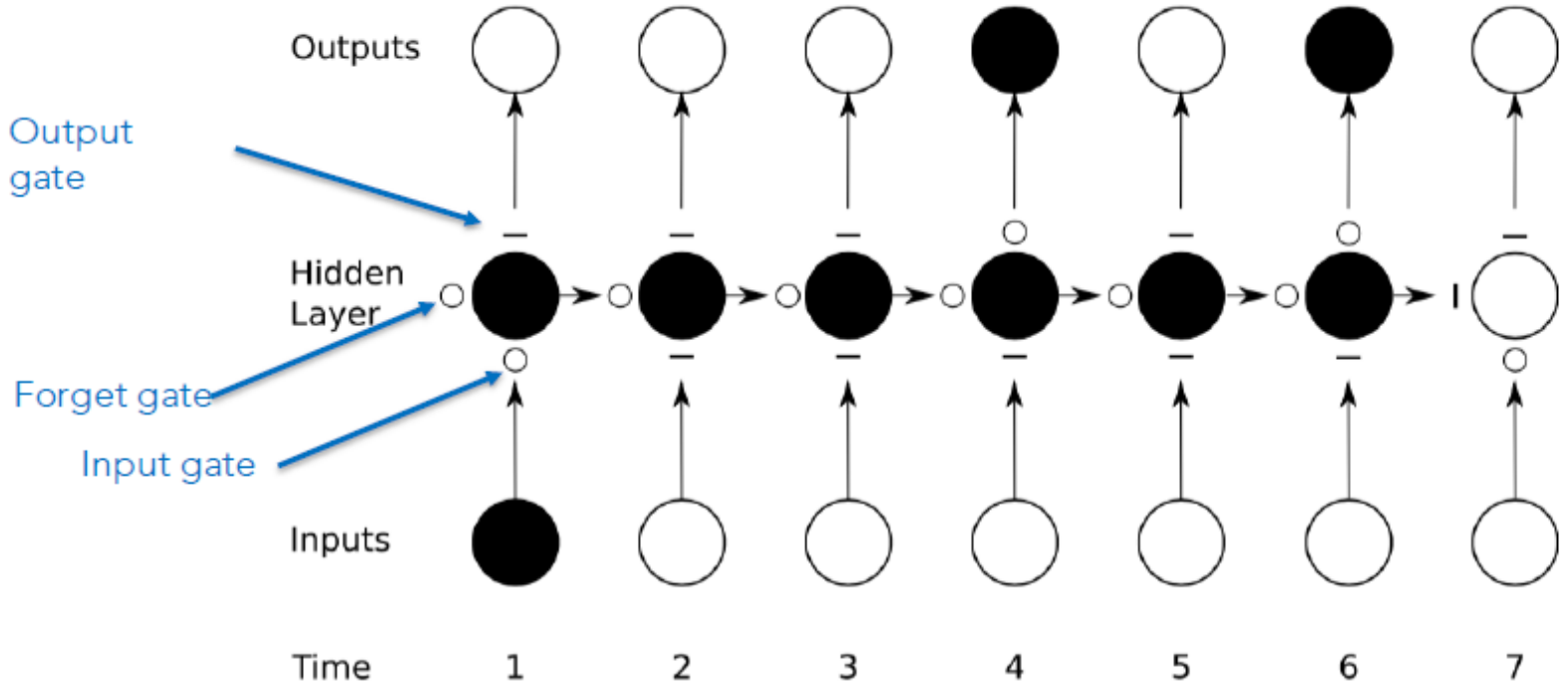
Vanishing gradient problem for RNNs



- The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity).
- The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network 'forgets' the first inputs.

Example from Graves 2012

Preservation of gradient info by LSTM



- For simplicity, all gates are either entirely open ('O') or closed ('—').
- The memory cell 'remembers' the first input as long as the forget gate is open and the input gate is closed.
- The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.

Example from Graves 2012

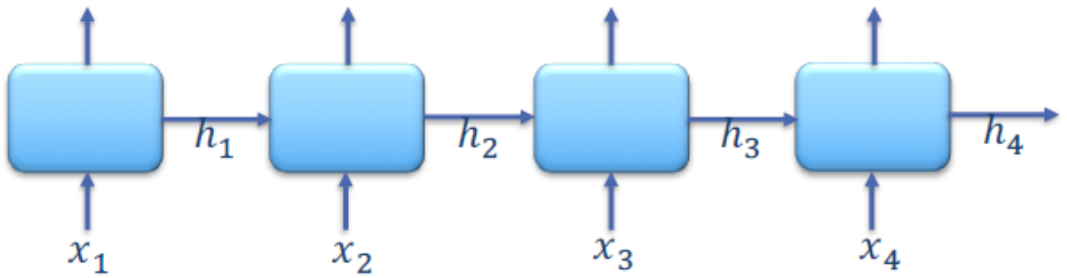
GRUs

- Generic RNNs: $h_t = f(x_t, h_{t-1})$
- Vanilla RNNs: $h_t = \tanh(Ux_t + Wh_{t-1} + b)$
- GRUs (Gated Recurrent Units):

$$z_t = \sigma(U^{(z)}x_t + W^{(z)}h_{t-1} + b^{(z)})$$
$$r_t = \sigma(U^{(r)}x_t + W^{(r)}h_{t-1} + b^{(r)})$$
$$\tilde{h_t} = \tanh(U^{(h)}x_t + W^{(h)}(r_t \circ h_{t-1}) + b^{(h)})$$
$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h_t}$$

Z: Update gate
R: Reset gate

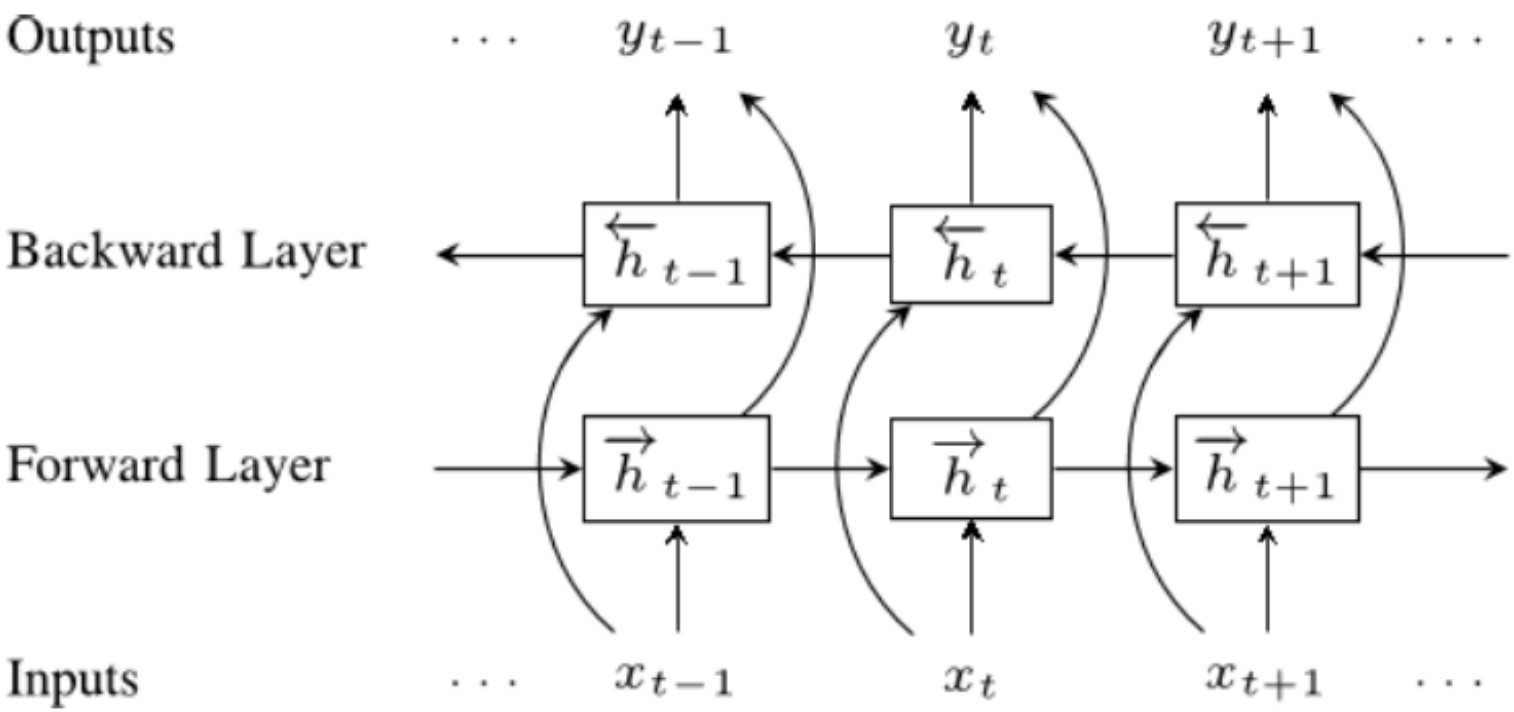
Less parameters than LSTMs. Easier to train for comparable performance!



Gates

- Gates contextually control information flow
- Open/close with sigmoid
- In LSTMs and GRUs, they are used to (contextually) maintain longer term history.

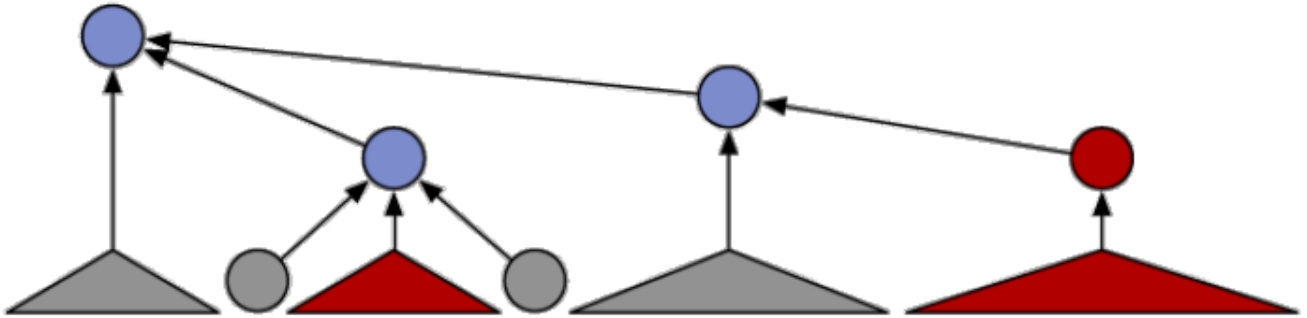
Bi-directional RNNs



- Can incorporate context from both directions
- Generally improves over uni-directional RNNs

Recursive NN

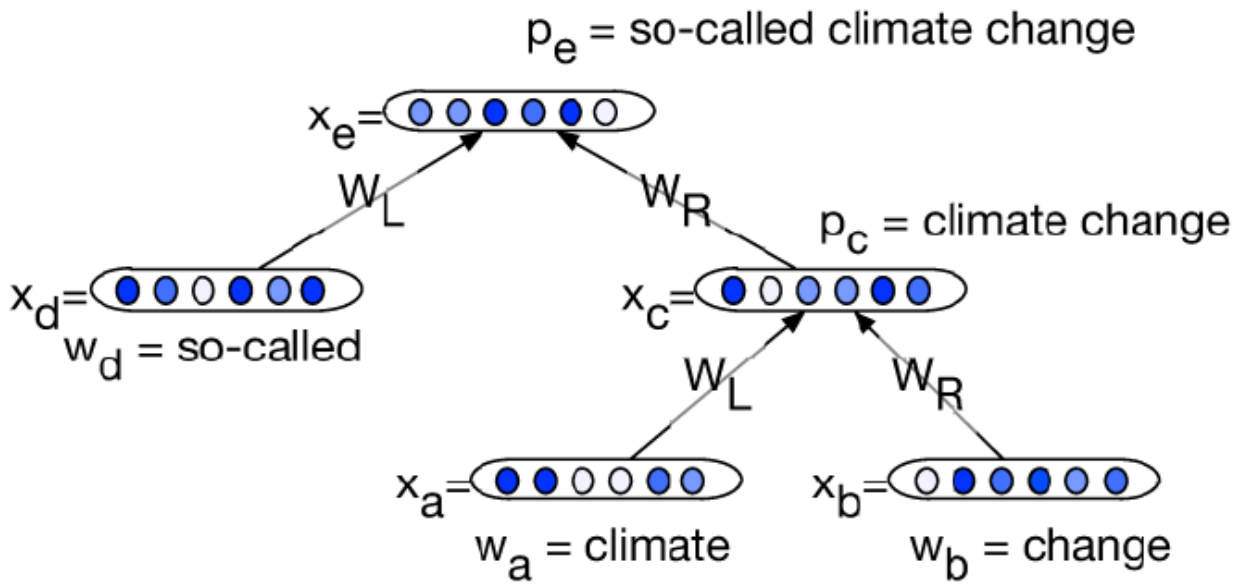
- Sometimes, inference over a tree structure makes more sense than sequential structure
- An example of compositionality in ideological bias detection (red → conservative, blue → liberal, gray → neutral) in which modifier phrases and punctuation cause polarity switches at higher levels of the parse tree



Example from Iyyer et al., 2014

Recursive NN

- NNs connected as a tree
- Tree structure is fixed a priori
- Parameters are shared, similarly as RNNs



Example from Iyyer et al., 2014

Outline

- Motivation
- Neuron and activation functions
- RNN and LSTM
- Training deep networks
- Summary

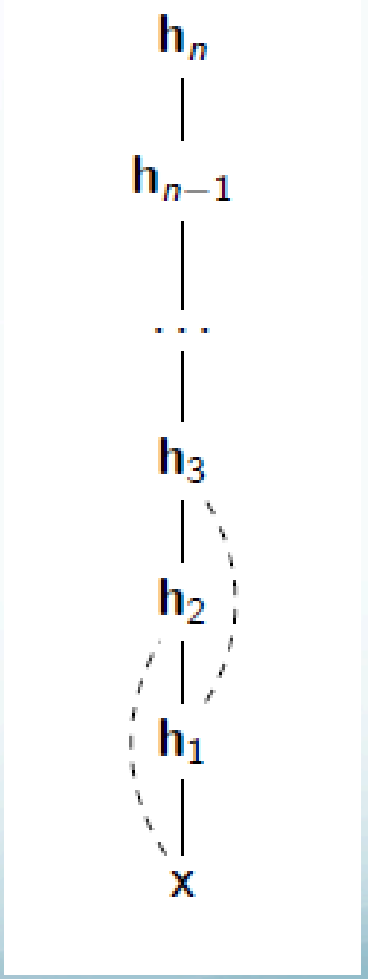
Training deep networks (including RNN)

- Deep networks are hard to train
- Main problems:
  - Vanishing / Exploding gradients:
  - Overfitting
- Practical solutions for Vanishing/exploding gradients:
  - Network architecture
  - Numerical operations

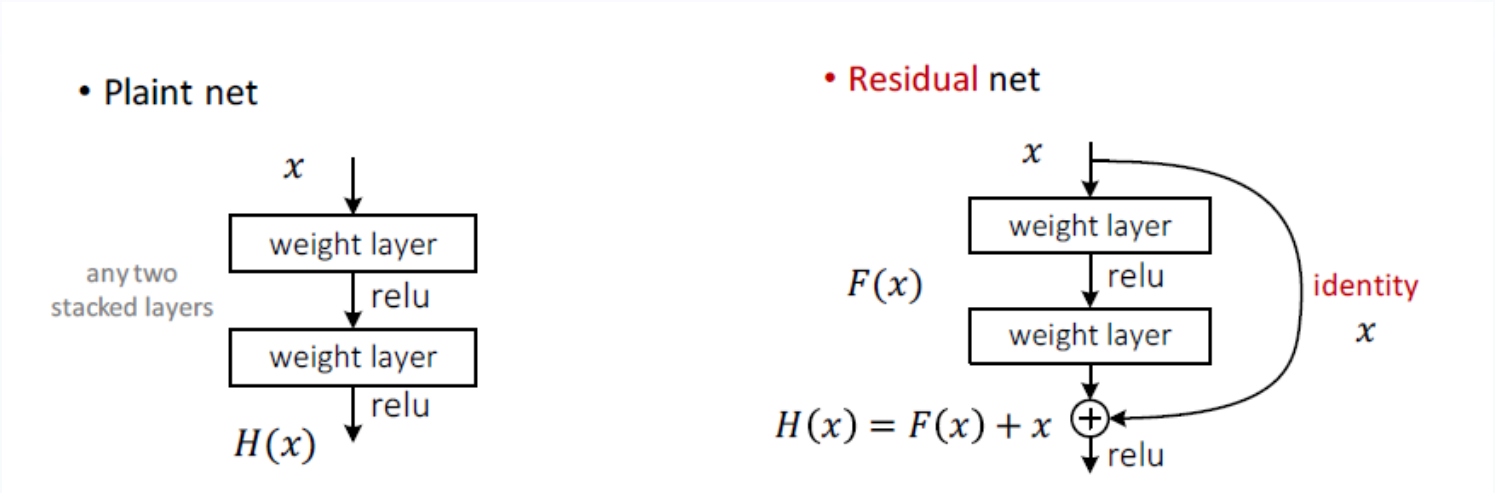
Solutions with network architecture

- Add skip connections to reduce distance:
  - Residual networks
  - Highway networks

- Add gates (and memory cells) to allow long-term memory:
  - LSTMs
  - GRUs
  - memory networks



Residual networks



ResNet (He et al, 2015): first very deep (152 layers) NN successfully trained for object recognition

Direct connection from the input x, compared to LSTMs where input connection has to go through “gates”

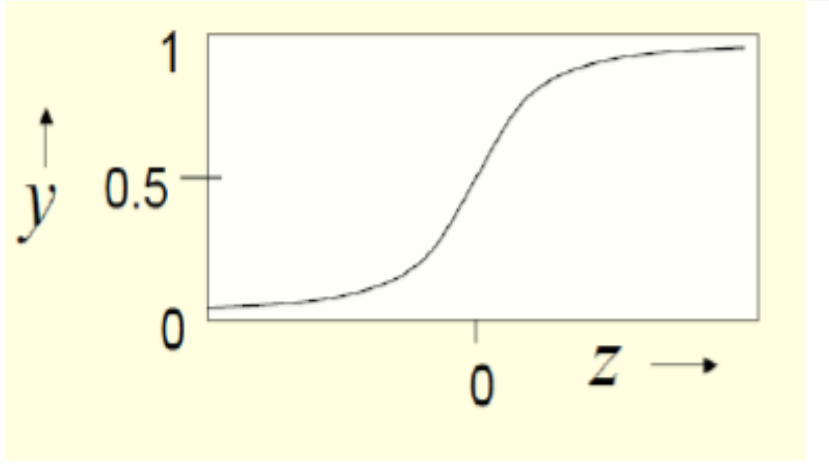
Solutions with numerical operations

- Gradient clipping: bound gradients by a max value
- Gradient normalization: renormalize gradients when they are above a fixed norm
- Careful initialization, smaller learning rate
- Avoid saturating nonlinearities (like tanh and sigmoid):
  - Use ReLU or hard-tanh instead
- Batch normalization: add intermediate input normalization layers

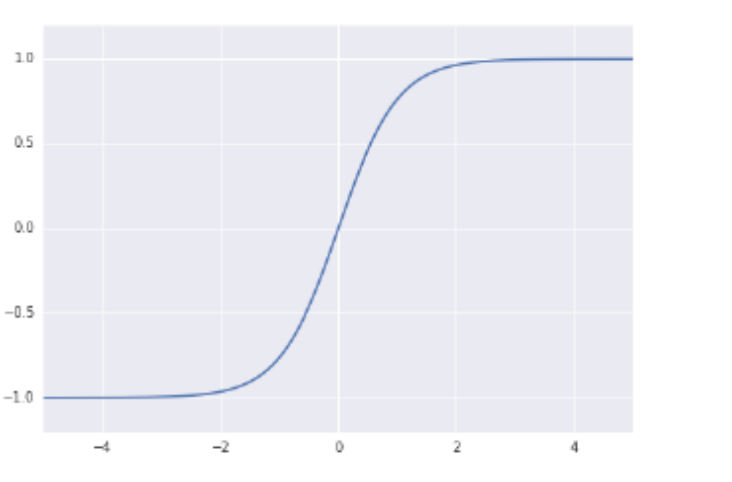
Sigmoid and tanh

- Pro: differentiable and nice derivative property
- Con: gradients saturate to zero almost everywhere except when x is near zero

➔ vanishing gradients



sigmoid: $f'(x)=f(x)(1-f(x))$

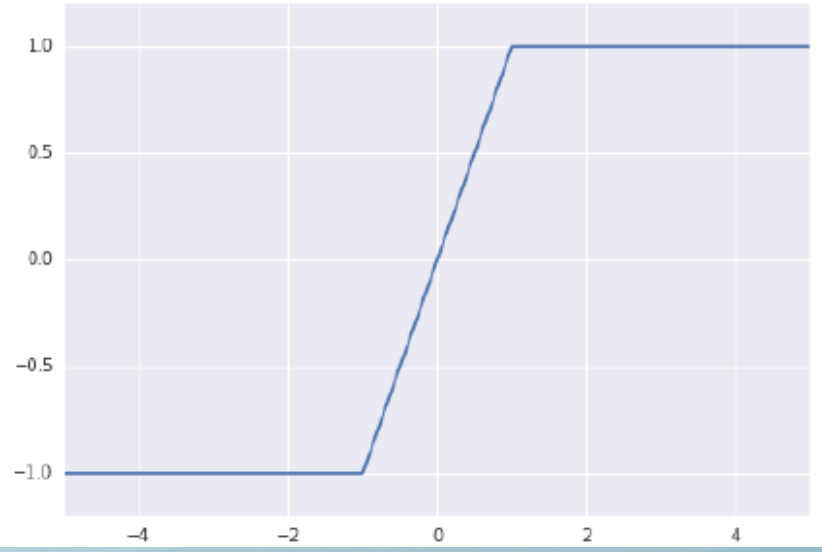


tanh:$f'(x)=1-f(x)^2$

Hard Tanh

- Pro: computationally cheaper
- Con: saturates to zero easily, doesn't differentiate at 1, -1

$$\text{hardtanh}(t) = \begin{cases} -1 & t < -1 \\ t & -1 \leq t \leq 1 \\ 1 & t > 1 \end{cases}$$

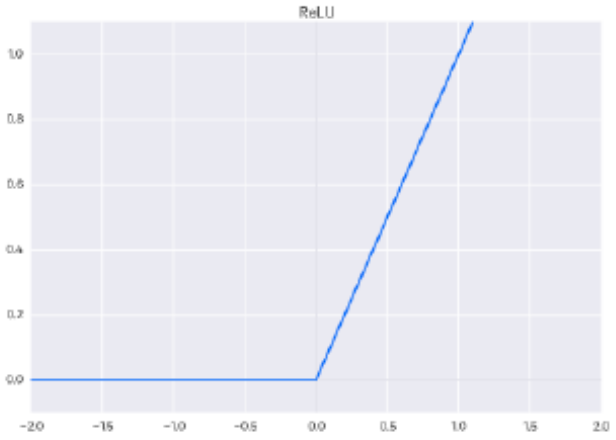


ReLU

- Pro: doesn't saturate for $x > 0$, computationally cheaper, induces sparse NNs
- Con: non-differentiable at 0
- Used widely in deep NN, but not as much in RNNs
- We informally use subgradients:

$$\mathrm{ReLU}(x) = \max(0, x)$$

$$\frac{d\,\mathrm{ReLU}(x)}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \\ 1 \text{ or } 0 & o.w \end{cases}$$



Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Dealing with overfitting

- Change objective function with L1 or L2 regularization
- Early stopping using validation set
- Change NN architecture: e.g., fewer hidden layers, fewer neurons at each layer
- Dropout (Hinton et al., 2012)

Dropout

- During training, randomly delete parts of network:
  - Each node (and its corresponding incoming and outgoing edges) dropped with a probability p
  - P is higher for internal nodes, lower for input nodes
- The full network is used for testing
- Faster training, better results
- Similar to Bagging

Convergence of backprop

- Without non-linearity or hidden layers, learning is convex optimization
  - Gradient descent reaches **global minima**

- Multilayer neural nets (with non-linearity) are **not convex**
  - Gradient descent gets stuck in local minima
  - Selecting number of hidden units and layers = fuzzy process
  - Pre-training can help

Outline

- Motivation
- Neuron and activation functions
- NN and its variants
- Summary

What was covered?

- Neurons and activity functions
- Many architectures:
  - feedforward, RNN, Recursive NN, LSTM
- Training a deep net is not easy:
  - Vanishing/exploding gradients
  - Solution with architectures: Residual network, LSTM, GRU
  - Solution with numeric operations: clipping gradients, batch normalization, etc.
- Not covered:
  - CNN, Attention, Encoder-decoder architecture, etc.
  - Applications: parsing, MT, QA, etc.

Using NNs

- Choose a NN architecture:
  - Given a task, how to represent x and y in the NN?
  - how many hidden layers? what does each layer represent?
  - what is used for each layer (e.g., CNN vs RNN, number of units, activation function)?
  - what is the objective function?
  - …
- Many decisions for training and tuning:
  - learning algorithms (e.g., L-BFGS, SGD)
  - how to prevent overfitting (e.g., regularization, dropout)
  - many hyper-parameters:
    - learning rate
    - number of training epochs
    - size of mini-batch
  - …

Advantages of NNs

- Able to learn feature representations and higher level of abstractions:
  - Less efforts on feature engineering (at the cost of more hyperparameter tuning)
  - NN induced representation can be concatenated with additional human engineered features
- Achieve state-of-the-art results on many NLP tasks (especially when there is a large amount of data)
- Can use unlabeled data and data from other languages/tasks/domains (e.g., multi-task learning)
- Often simple matrix derivatives (backprop) for training and matrix multiplication for testing
- Many existing packages: easy to build a model
- End-to-end systems, instead of pipeline approach

Challenges

- How to choose a good architecture:
  - How many hidden layers?
  - What type of NN for each layer?
  - How many neurons in each layer?
  - Any shortcut connections between layers?
  - ...
- Many hyper-parameters:
  - number of layers
  - number of units each layer
  - prevent overfitting (e.g., regularization, dropout)
  - learning rate
  - size of mini-batch
  - learning algorithms (e.g., L-BFGS, SGD)
  - ...
- Often need large amount of training data

Questions

- How to interpret the model?
- How to encode prior knowledge?
  - Choose architectures suitable for a problem domain
  - Include human-designed features in the first layer
- How to combine with existing NLP?
  - Ex: recursive NN for trees
  - Ex: recurrent NN for chains
- How to debug the system and do error analysis?