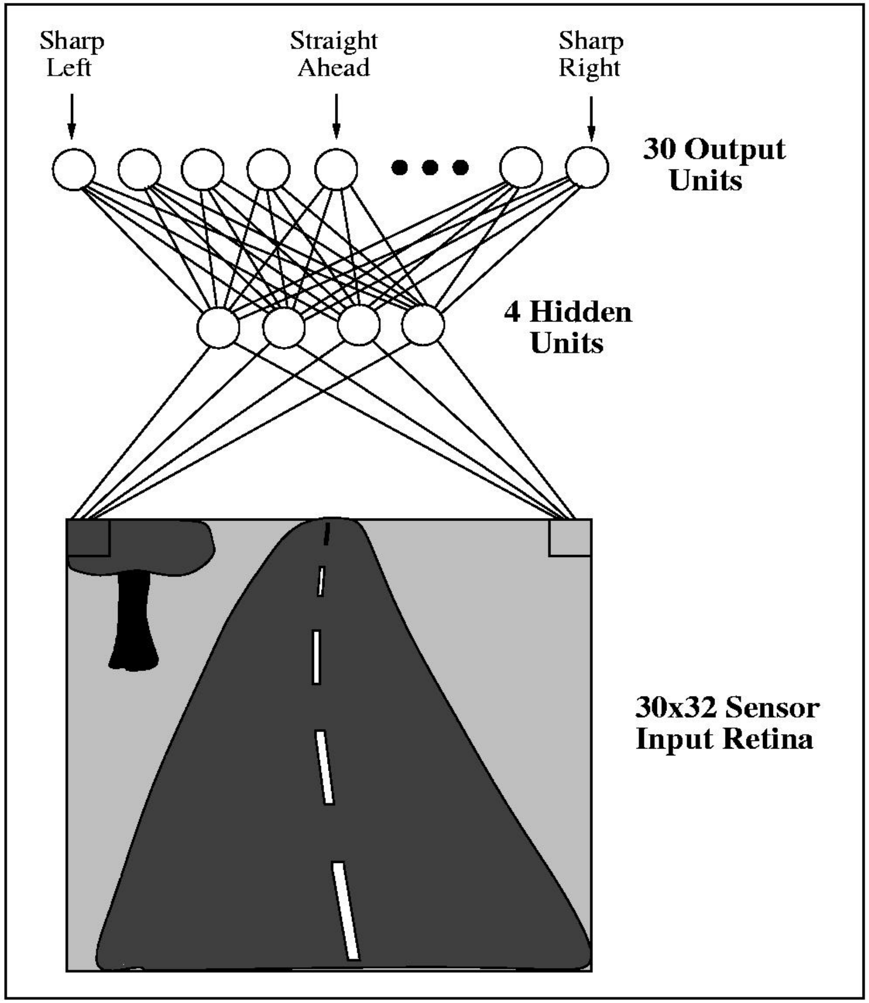
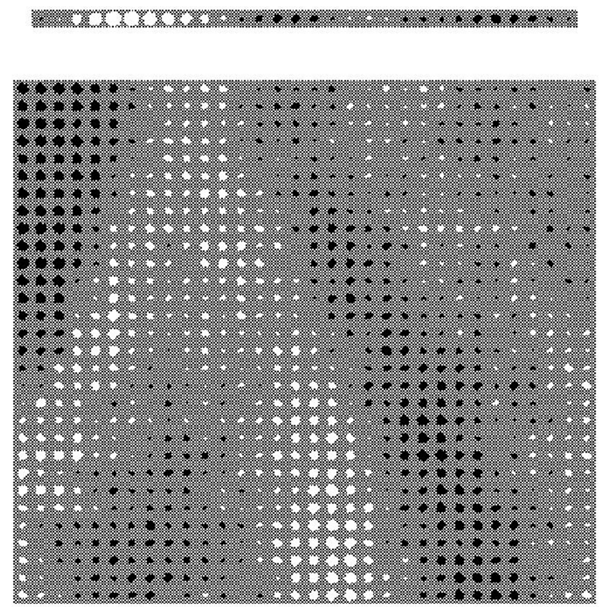
CSE P 517
Natural Language Processing
Winter 2021

Deep Learning

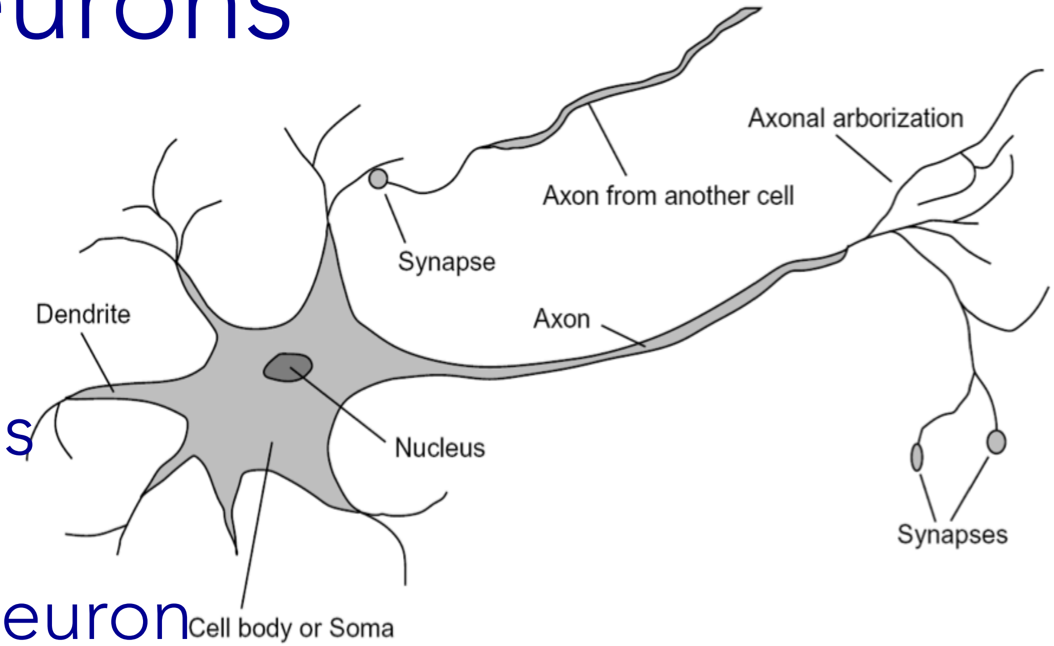
Yejin Choi
University of Washington

Next several slides are from Carlos Guestrin, Luke Zettlemoyer


Sharp Left
Straight Ahead
Sharp Right
30 Output Units
4 Hidden Units
30x32 Sensor Input Retina

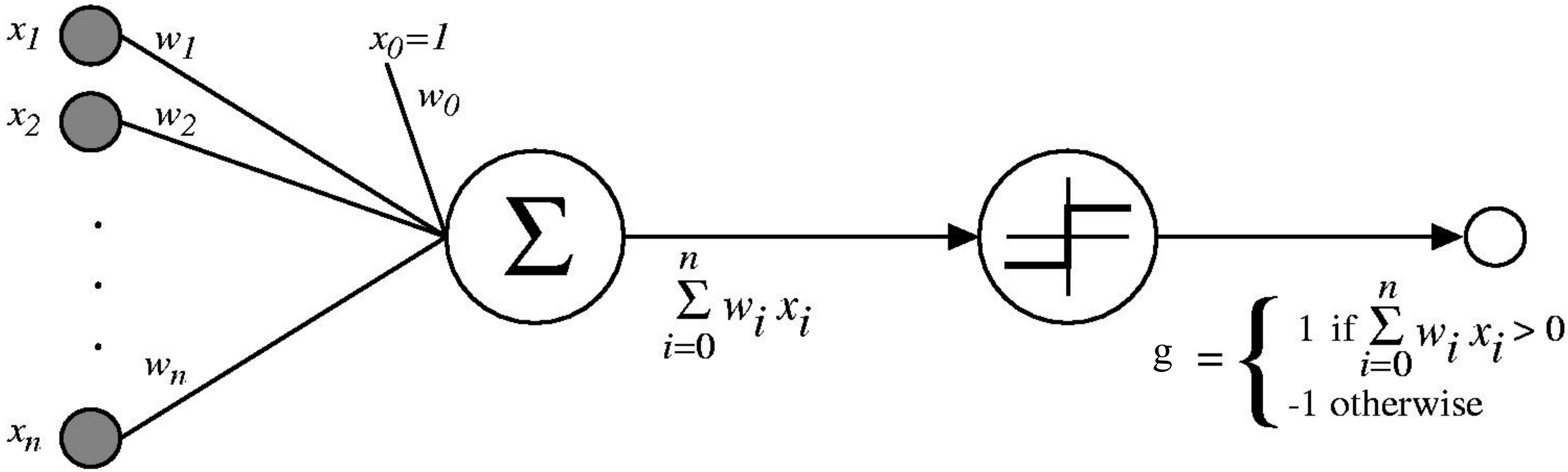


Human Neurons



- Switching time
  - ~ 0.001 second
- Number of neurons
  - 10^{10}
- Connections per neuron
  - $10^{4\text{-}5}$
- Scene recognition time
  - 0.1 seconds
- Number of cycles per scene recognition?
  - 100 → much parallel computation!

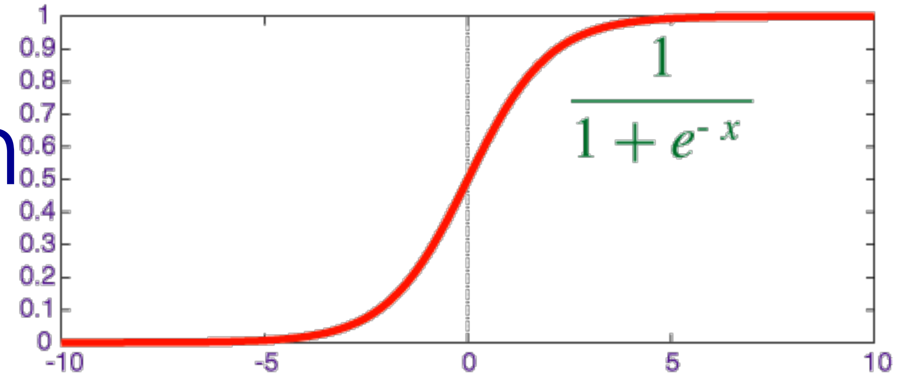
Perceptron as a Neural Network



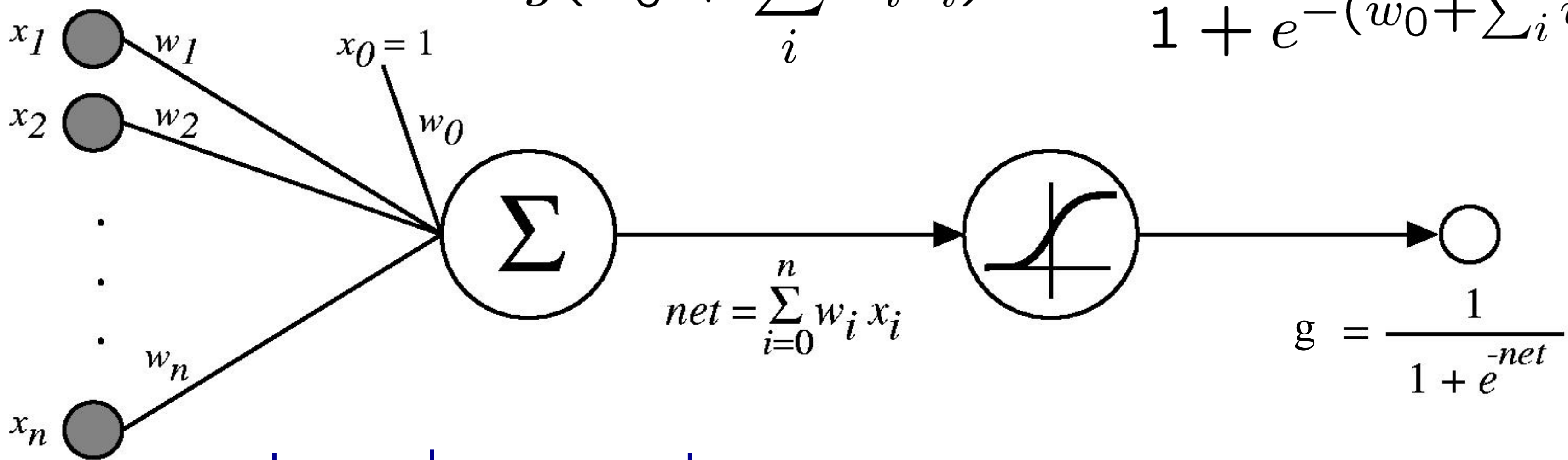
This is one neuron:

- Input edges x_1 ... x_n, along with basis
- The sum is represented graphically
- Sum passed through an activation function g

Sigmoid Neuron



$$g(w_0 + \sum_i w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$



Just change g!

- Why would we want to do this?
- Notice new output range [0,1]. What was it before?
- Look familiar?

Optimizing a neuron

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x)$$

We train to minimize sum-squared error

$$\ell(W) = \frac{1}{2}\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]^2$$

$$\frac{\partial l}{\partial w_i} = -\sum_j [y_j - g(w_0 + \sum_i w_i x_i^j)] \frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j)$$

$$\frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x_i^j) = x_i^j g'(w_0 + \sum_i w_i x_i^j)$$

$$\frac{\partial \ell(W)}{\partial w_i} = -\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]\ x_i^j\ g'(w_0 + \sum_i w_i x_i^j)$$

Solution just depends on g': derivative of activation function!

Sigmoid units: have to differentiate g

$$\frac{\partial \ell(W)}{\partial w_i} = -\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]\ x_i^j\ g'(w_0 + \sum_i w_i x_i^j)$$

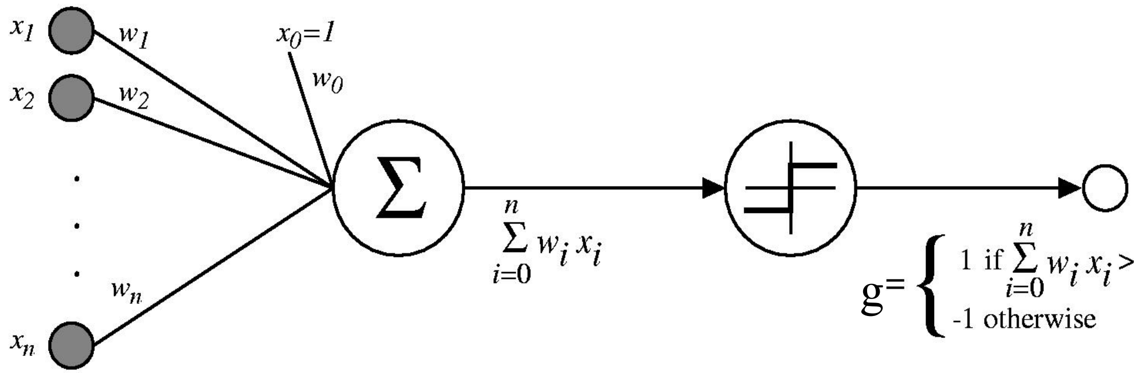
$$g(x) = \frac{1}{1+e^{-x}} \qquad g'(x) = g(x)(1-g(x))$$

$$\boxed{w_i \leftarrow w_i + \eta \sum_j x_i^j \delta^j}$$

$$\delta^j = [y^j - g(w_0 + \sum_i w_i x_i^j)] g^j (1 - g^j)$$

$$g^j = g(w_0 + \sum_i w_i x_i^j)$$

Perceptron, linear classification, Boolean functions: $x_i \in \{0,1\}$



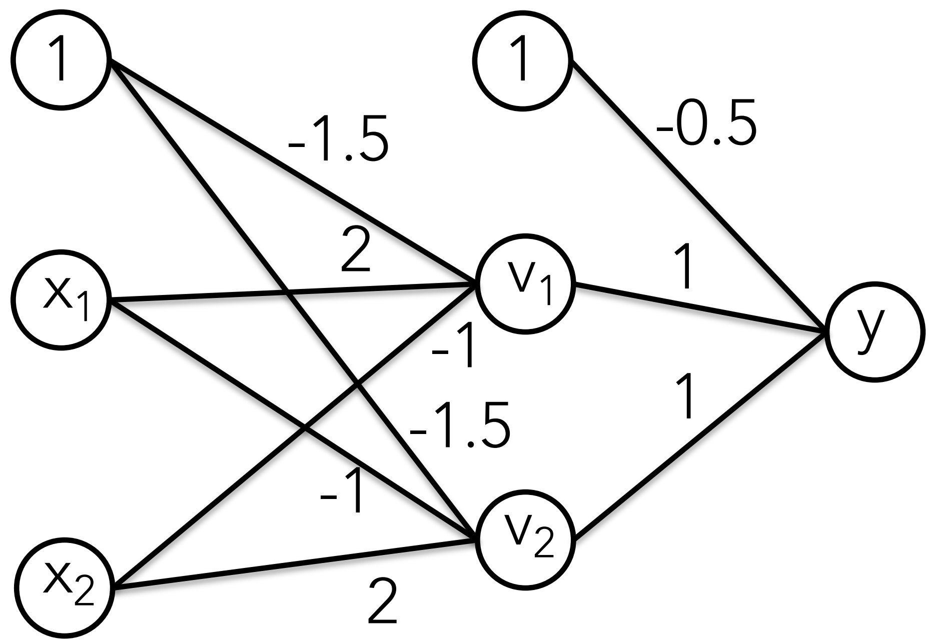
- Can learn $x_1 \vee x_2$?
  - $-0.5 + x_1 + x_2$
- Can learn $x_1 \wedge x_2$?
  - $-1.5 + x_1 + x_2$
- Can learn any conjunction or disjunction?
  - $-0.5 + x_1 + \ldots + x_n$
  - $(-n+0.5) + x_1 + \ldots + x_n$
- Can learn majority?
  - $(-0.5*n) + x_1 + \ldots + x_n$
- What are we missing? The dreaded XOR!, etc.

Going beyond linear classification

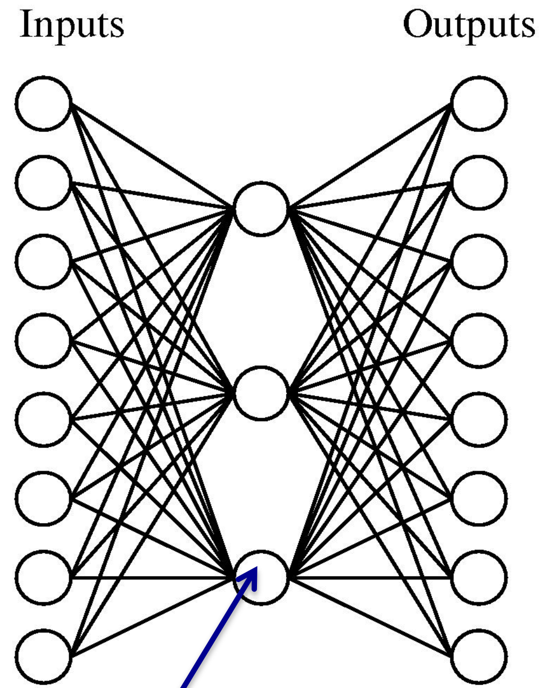
Solving the XOR problem

$$y = x_1 \text{ XOR } x_2 = (x_1 \wedge \neg x_2) \vee (x_2 \wedge \neg x_1)$$

$$v_1 = (x_1 \wedge \neg x_2)$$
$$= -1.5 + 2x_1 - x_2$$
$$v_2 = (x_2 \wedge \neg x_1)$$
$$= -1.5 + 2x_2 - x_1$$
$$y = v_1 \vee v_2$$
$$= -0.5 + v_1 + v_2$$



Hidden layer



- Single unit:

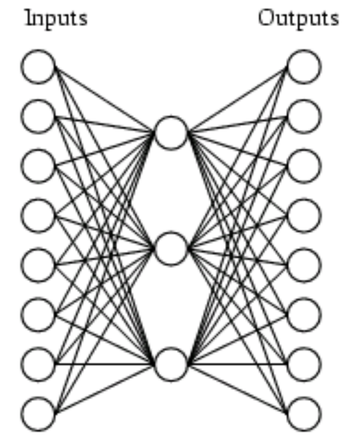
$$out(\mathbf{x}) \;=\; g(w_0 + \sum_i w_i x_i)$$

- 1-hidden layer:

$$out(\mathbf{x}) \;=\; g\left(w_0 + \sum_k w_k \underbrace{g(w_0^k + \sum_i w_i^k x_i)}\right)$$

- No longer convex function!

Example data for NN with hidden layer



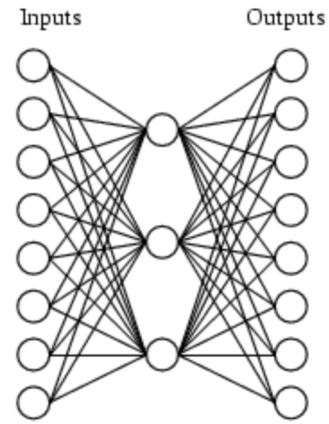
A target function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned??

Learned weights for hidden layer

A network:



Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Why "representation learning"?

- MaxEnt (multinomial logistic regression):

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

You design the feature vector

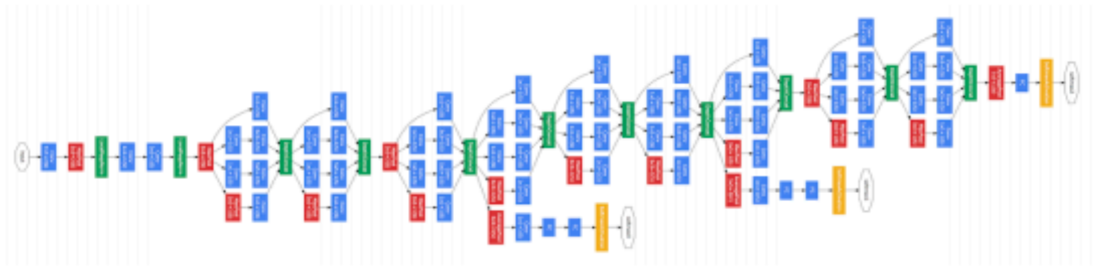
- NNs:

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

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

Feature representations are "learned" through hidden layers

Very deep models in computer vision



[1]Inception 5 (GoogLeNet)

Inception 7a

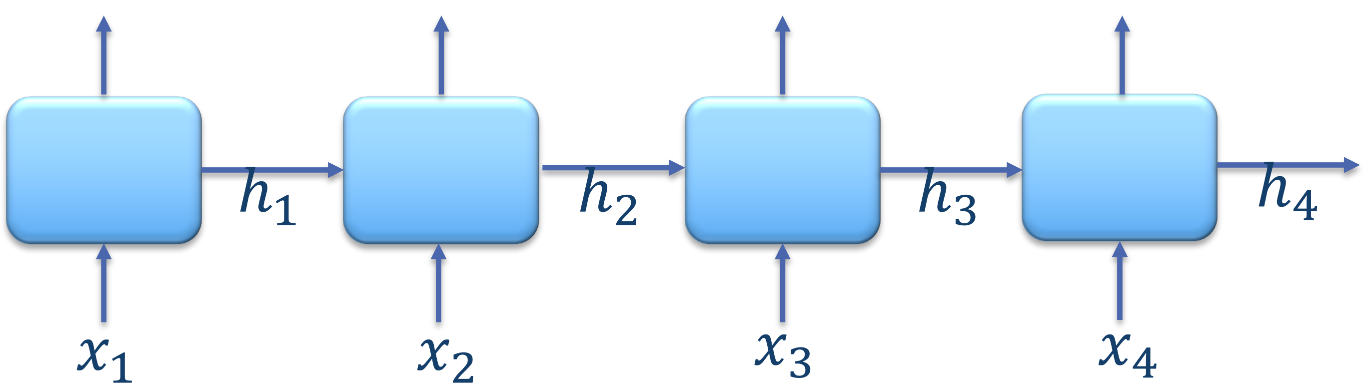
[1]Going Deeper with Convolutions, [C. Szegedy et al, CVPR 2015]

RECURRENT NEURAL NETWORKS

Recurrent Neural Networks (RNNs)

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

- Hidden states $h_t \in R^D$ are continuous vectors
  - Can represent very rich information
  - Possibly the entire history from the beginning
- Parameters are shared (tied) across all RNN units (unlike feedforward NNs)



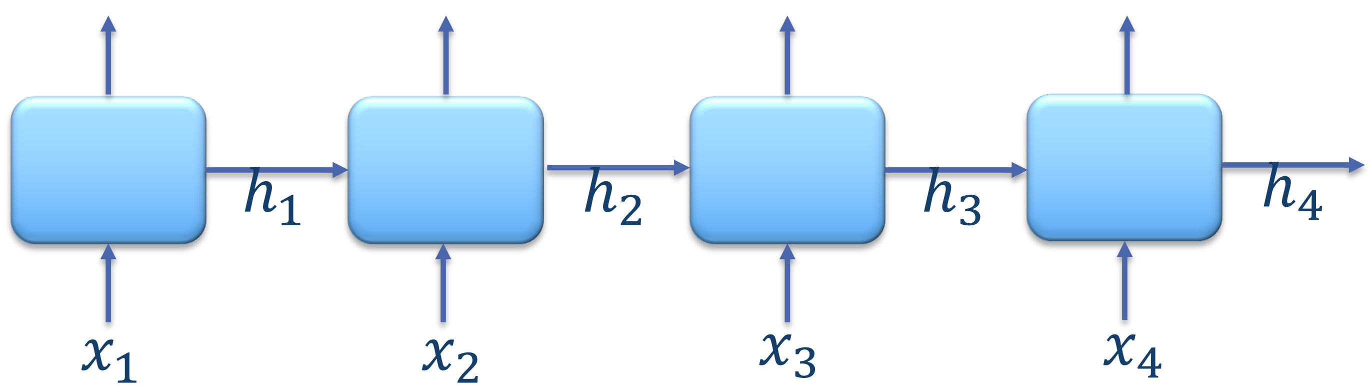
Recurrent Neural Networks (RNNs)

- Generic RNNs:

$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(V h_t)$$

- Vanilla RNN:

$$h_t = \tanh(U x_t + W h_{t-1} + b)$$
$$y_t = \text{softmax}(V h_t)$$



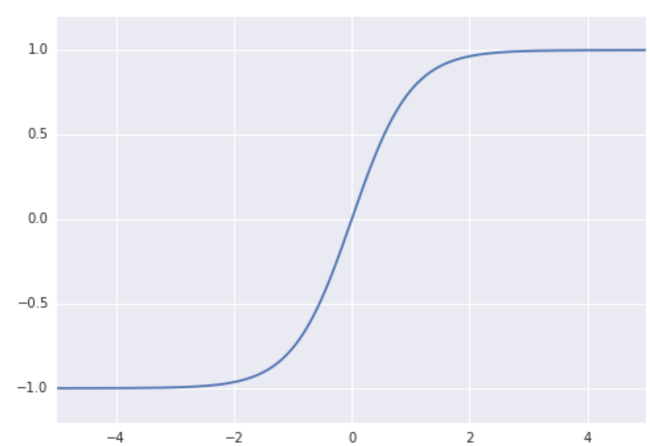
Tanh

- Often used for hidden states & cells in RNNs, LSTMs
- Pro: differentiable, often converges faster than sigmoid
- Con: gradients easily saturate to zero => vanishing gradients

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\text{tanh'(x)} = 1 - \tanh^2(x)$$

$$\tanh(x) = 2\sigma(2x) - 1$$

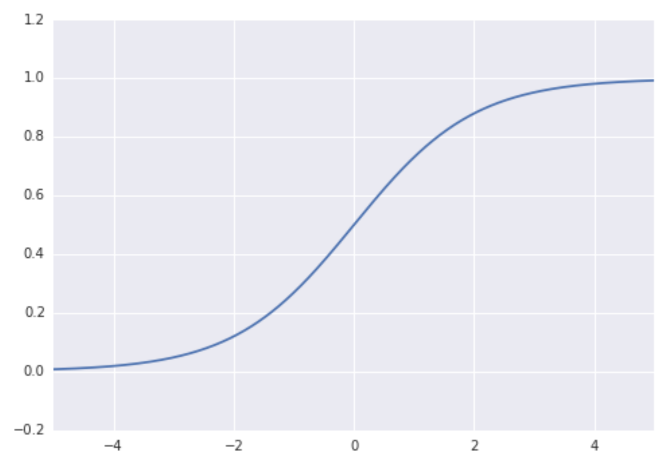


Sigmoid

- Often used for gates
- Pro: neuron-like, differentiable
- Con: gradients saturate to zero almost everywhere except x near zero => vanishing gradients
- Batch normalization helps

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

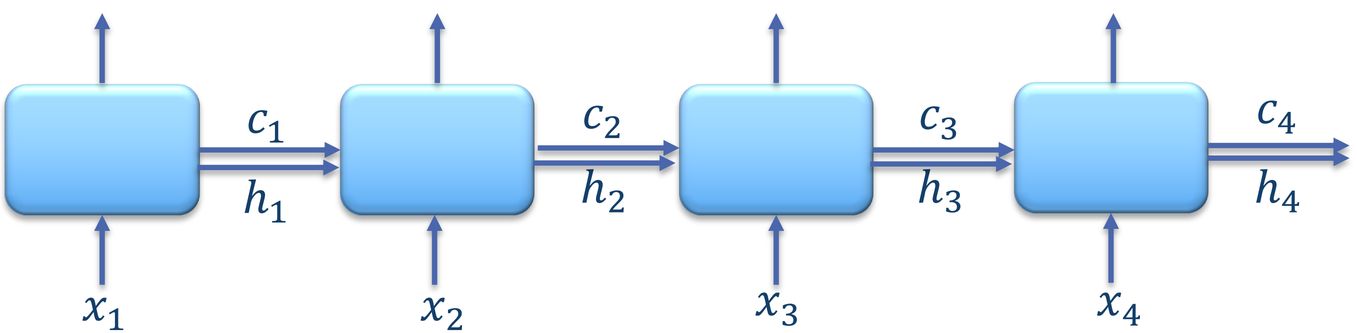


Recurrent Neural Networks (RNNs)

- Generic RNNs: $h_t = f(x_t, h_{t-1})$
- Vanilla RNNs: $h_t = \tanh(Ux_t + Wh_{t-1} + b)$
- LSTMs (Long Short-term Memory Networks):

$$i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)})$$
$$f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)})$$
$$o_t = \sigma(U^{(o)}x_t + W^{(o)}h_{t-1} + b^{(o)})$$
$$\tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)})$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$
$$h_t = o_t \circ \tanh(c_t)$$

There are many known variations to this set of equations!



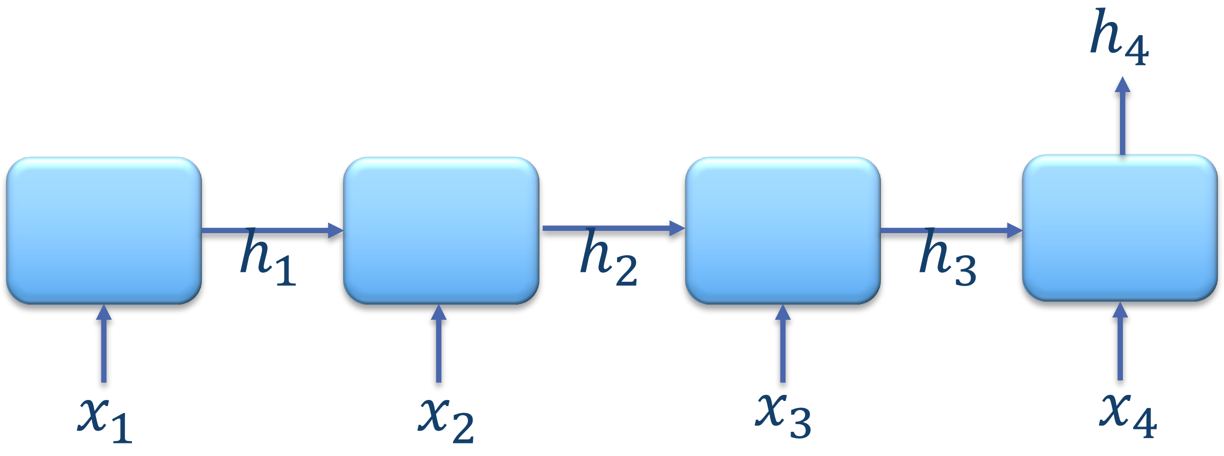
c_t : cell state

h_t : hidden state

Many uses of RNNs
1. Classification (seq to one)

- Input: a sequence
- Output: one label (classification)
- Example: sentiment classification

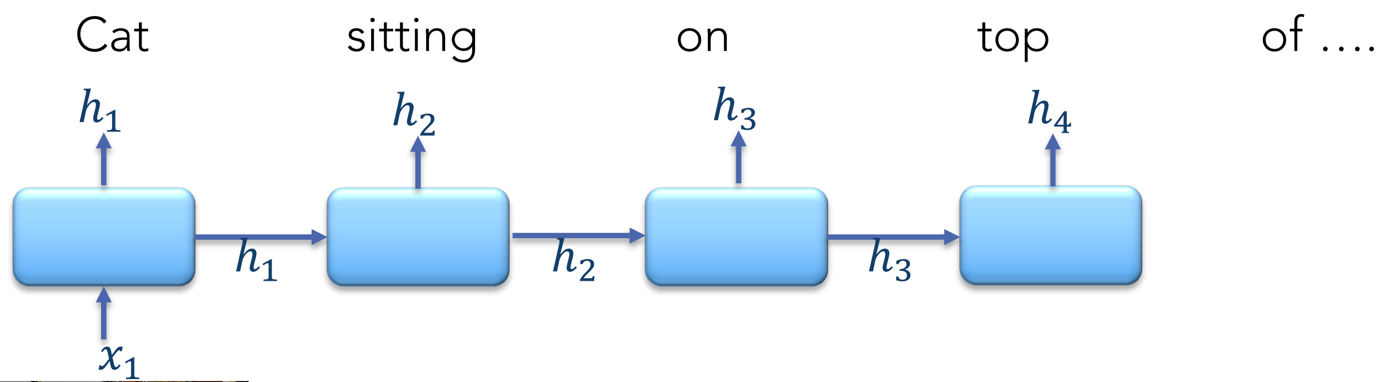
$$h_t = f(x_t, h_{t-1})$$
$$y = \text{softmax}(V h_n)$$



Many uses of RNNs
2. one to seq

- Input: one item
- Output: a sequence
- Example: Image captioning

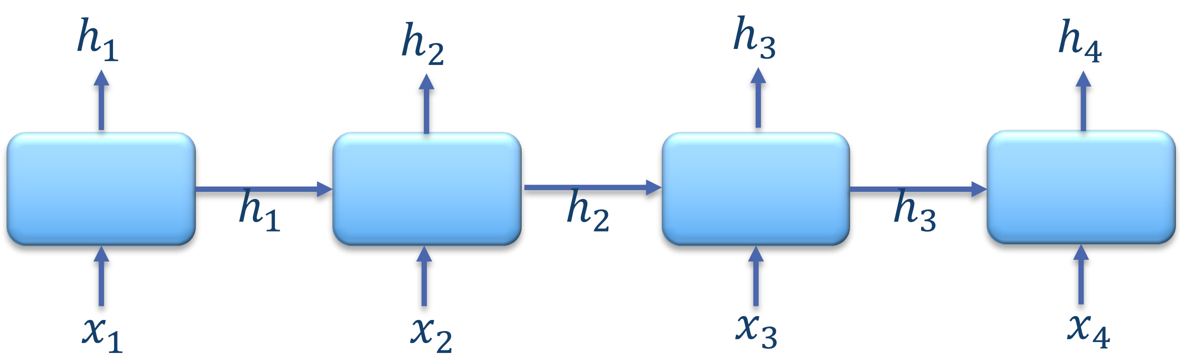
$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(V h_t)$$



Many uses of RNNs
3. sequence tagging

- Input: a sequence
- Output: a sequence (of the same length)
- Example: POS tagging, Named Entity Recognition
- How about Language Models?
  - Yes! RNNs can be used as LMs!
  - RNNs make markov assumption: T/F?

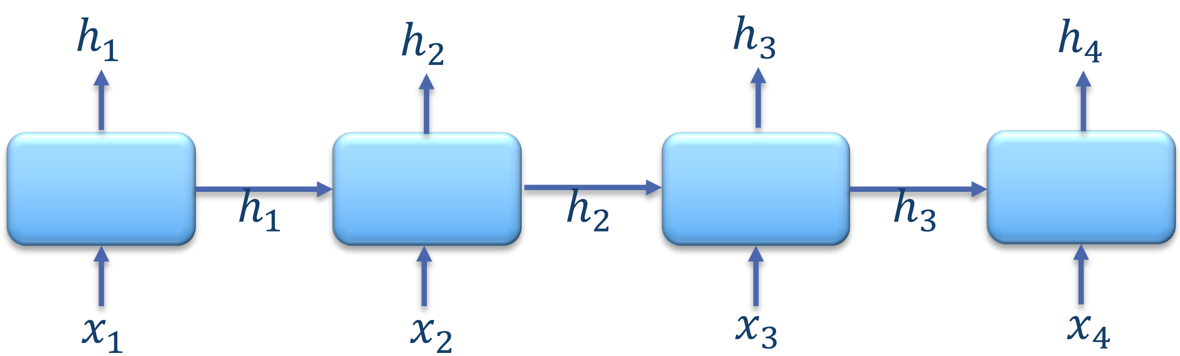
$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(V h_t)$$



Many uses of RNNs
4. Language models

- Input: a sequence of words
- Output: one next word
- Output: or a sequence of next words
- During training or if used for measuring LM score, x_t is the actual word in the training sentence.
- If used for sampling, x_t is the word predicted from the previous time step.
- Does RNN LMs make Markov assumption?
  - i.e., the next word depends only on the previous N words?

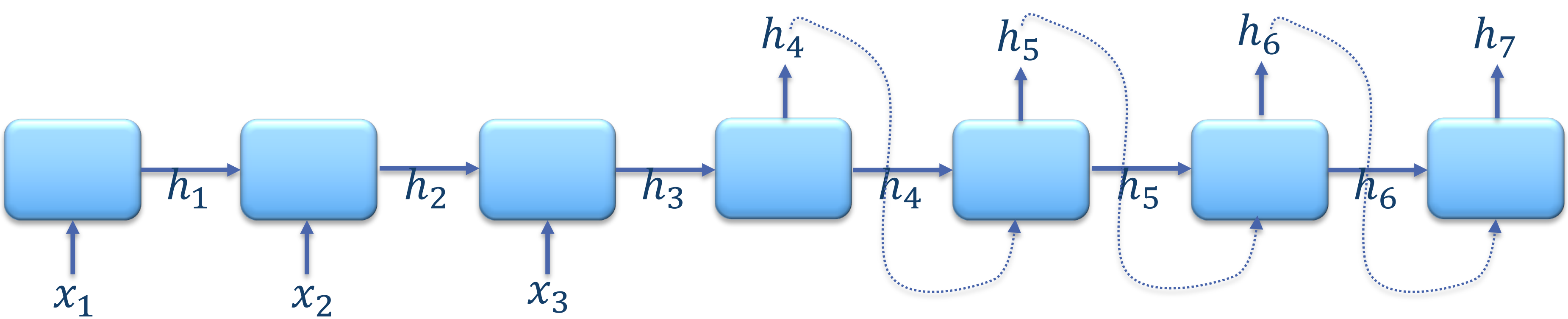
$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(V h_t)$$



Many uses of RNNs
5. seq2seq (aka "encoder-decoder")

- Input: a sequence
- Output: a sequence (of *different* length)
- Examples?

$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(Vh_t)$$



Many uses of RNNs

4. seq2seq (aka "encoder-decoder")

- Conversation and Dialogue
- Machine Translation

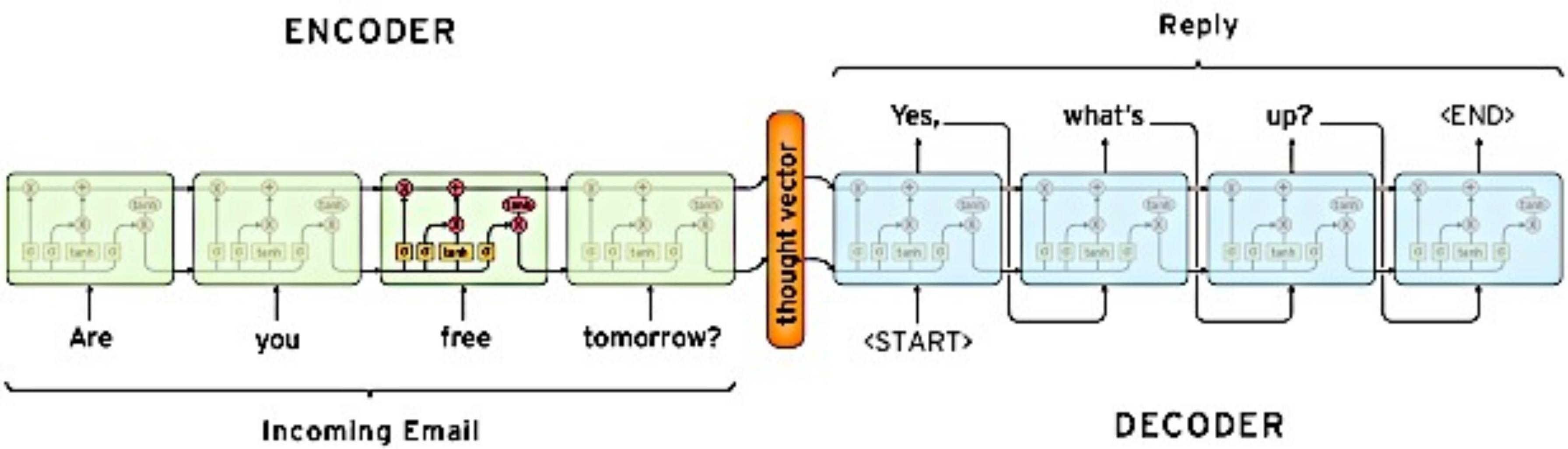
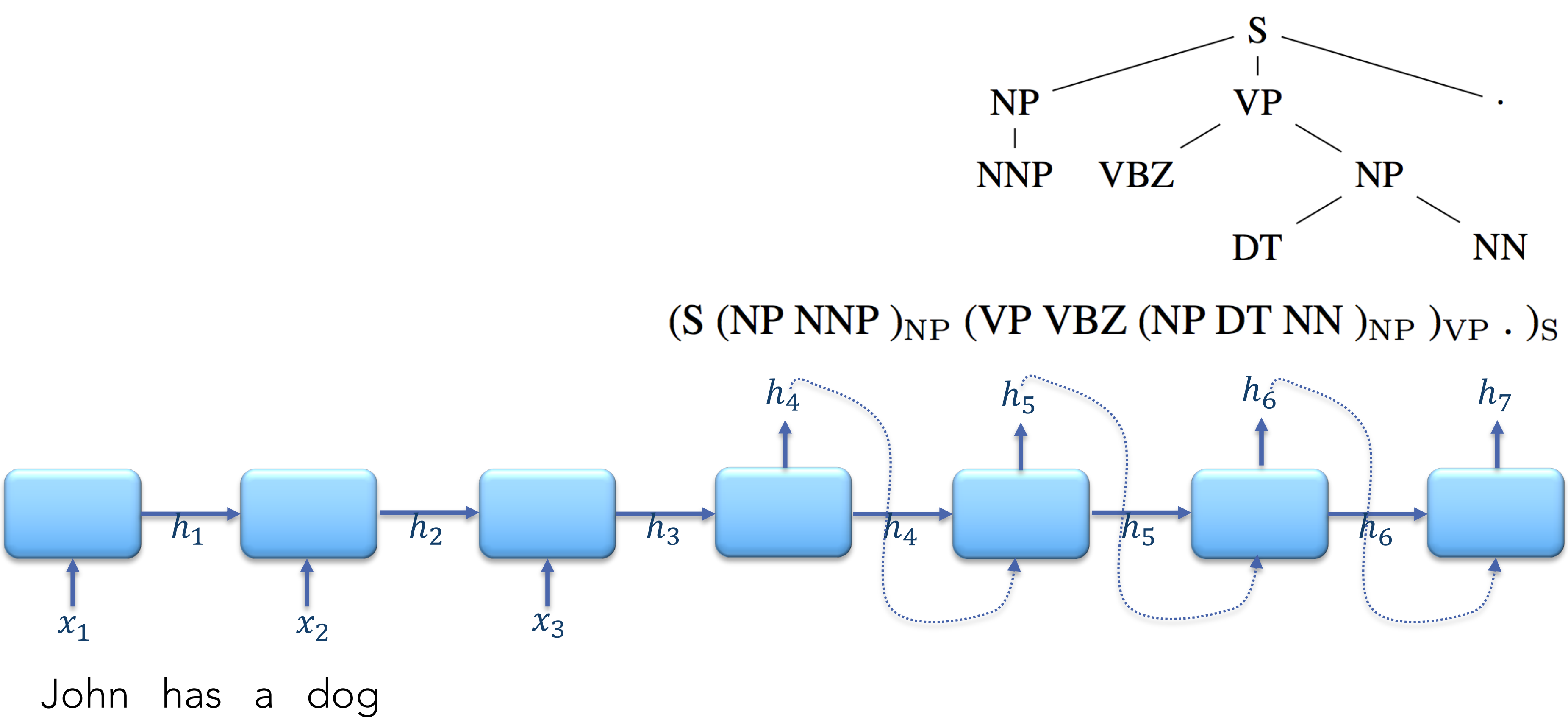


Figure from http://www.wildml.com/category/conversational-agents/

Many uses of RNNs
4. seq2seq (aka "encoder-decoder")

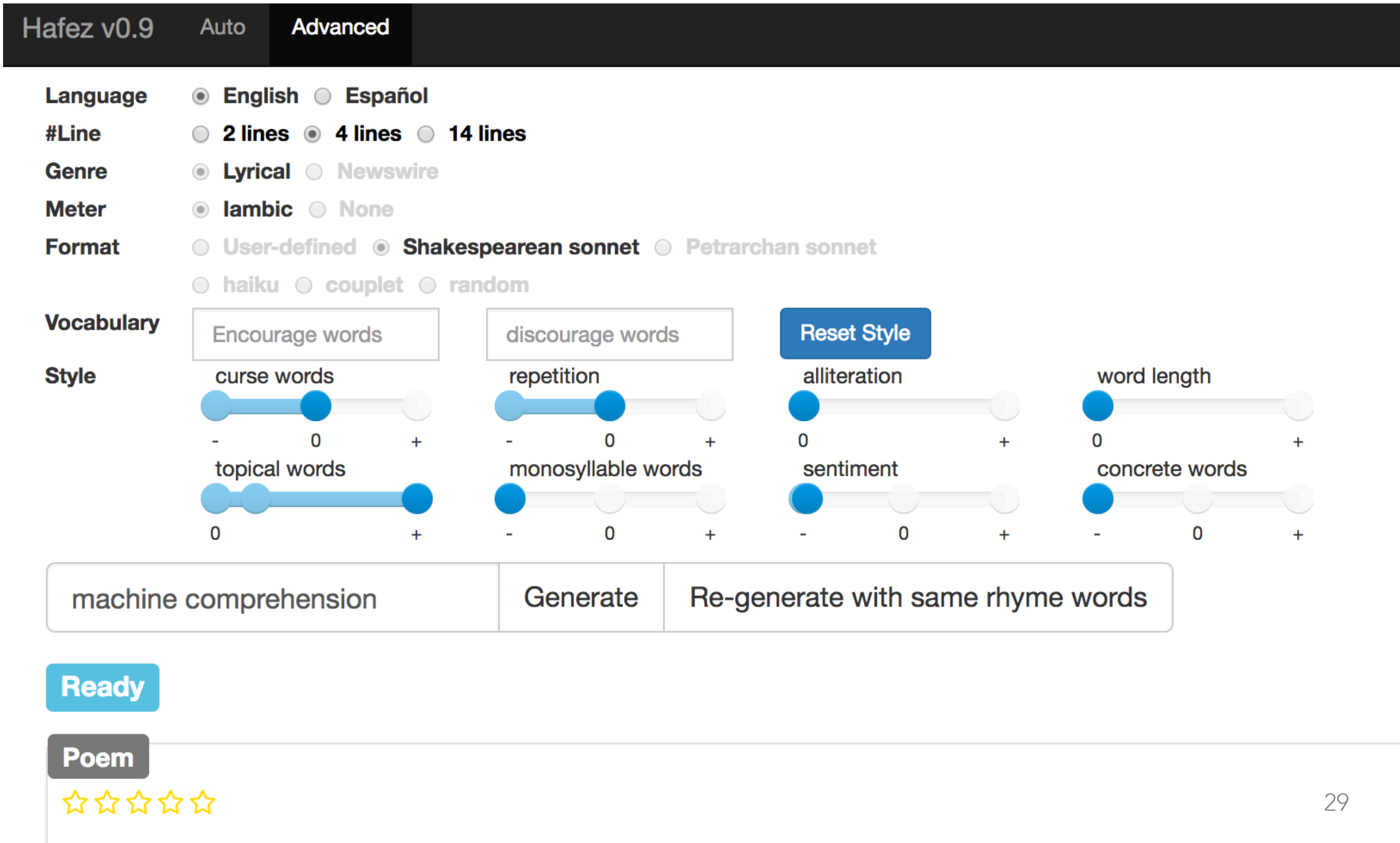
Parsing!

- *"Grammar as Foreign Language"* (Vinyals et al., 2015)



Hafez: Neural Sonnet Writer

(Ghazvininejad et al. 2016)



Neural Sonnets

Deep Convolution Network

Outrageous channels on the wrong *connections,*
An empty space without an open *layer,*
A closet full of black and blue *extensions,*
Connections by the *closure operator.*

Theory

Another way to reach the wrong *conclusion!*
A vision from a total *transformation,*
Created by the great *magnetic fusion,*
Lots of people need an *explanation.*

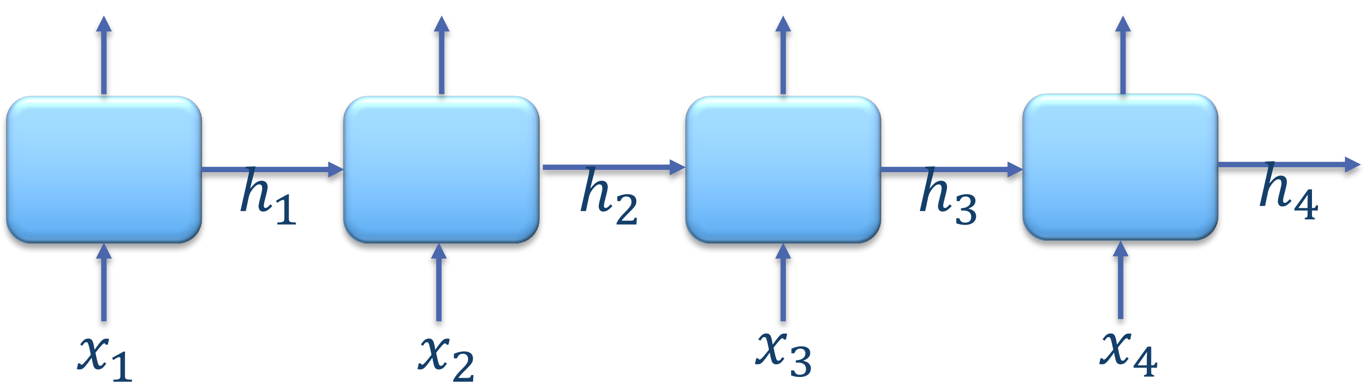
Recurrent Neural Networks (RNNs)

- Generic RNNs:

$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(V h_t)$$

- Vanilla RNN:

$$h_t = \tanh(U x_t + W h_{t-1} + b)$$
$$y_t = \text{softmax}(V h_t)$$

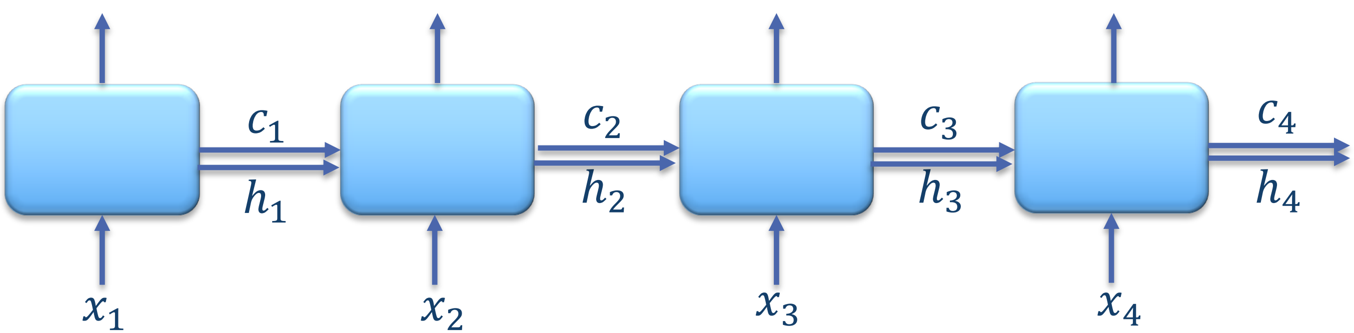


Recurrent Neural Networks (RNNs)

- Generic RNNs: $h_t = f(x_t, h_{t-1})$
- Vanilla RNNs: $h_t = \tanh(Ux_t + Wh_{t-1} + b)$
- LSTMs (Long Short-term Memory Networks): (Hochreiter et al, 1997)

$$
\begin{aligned}
i_t &= \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)}) \\
f_t &= \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)}) \\
o_t &= \sigma(U^{(o)}x_t + W^{(o)}h_{t-1} + b^{(o)}) \\
\tilde{c}_t &= \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)}) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

There are many known variations to this set of equations!



c_t : cell state

h_t : hidden state

LSTMS (LONG SHORT-TERM MEMORY NETWORKS

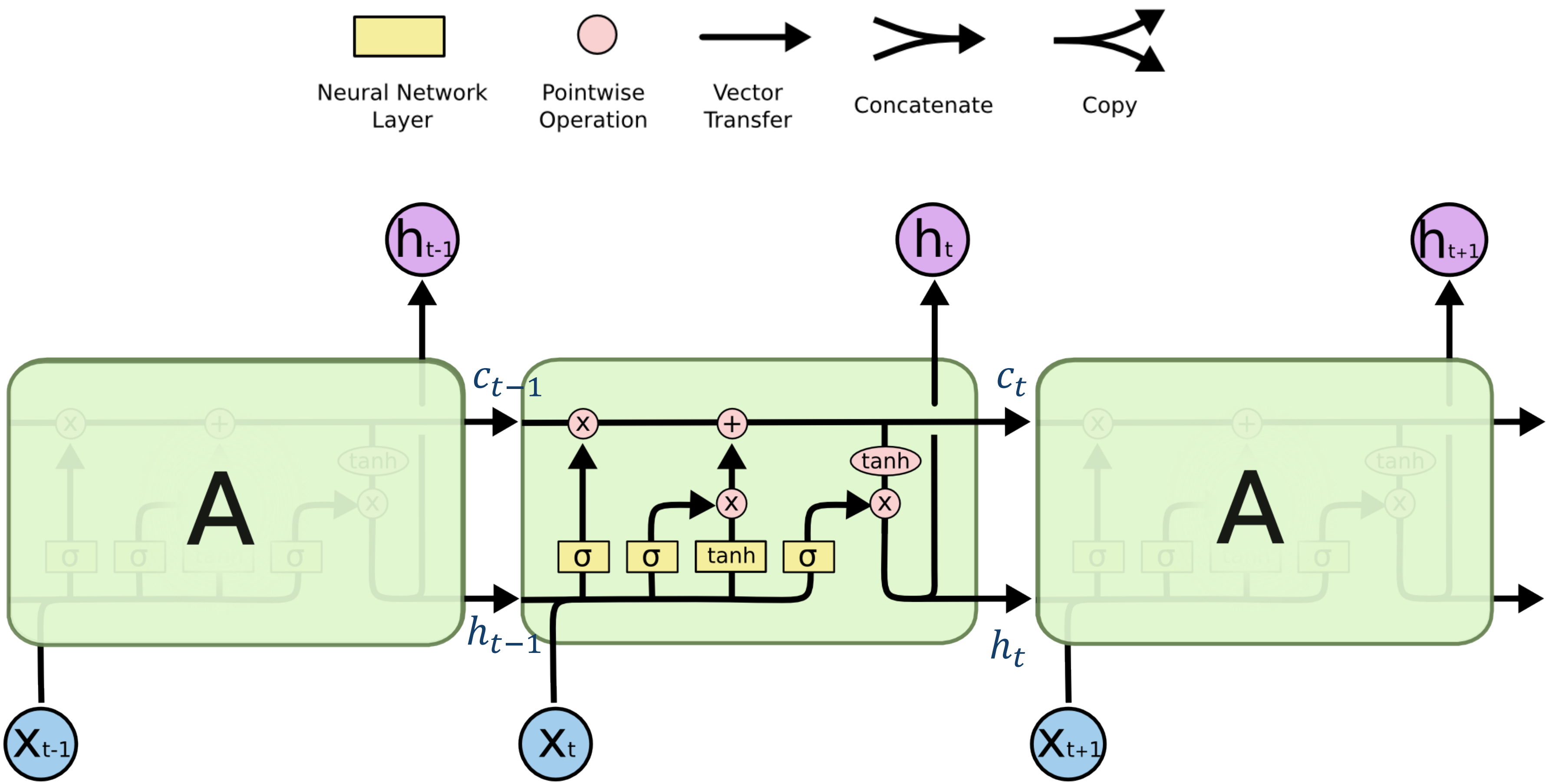
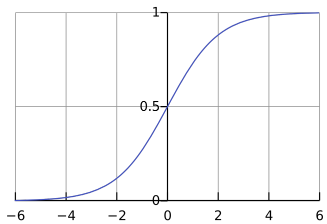


Figure by Christopher Olah (colah.github.io)

LSTMS (LONG SHORT-TERM MEMORY NETWORKS

sigmoid:
[0,1]



Forget gate: forget the past or not

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

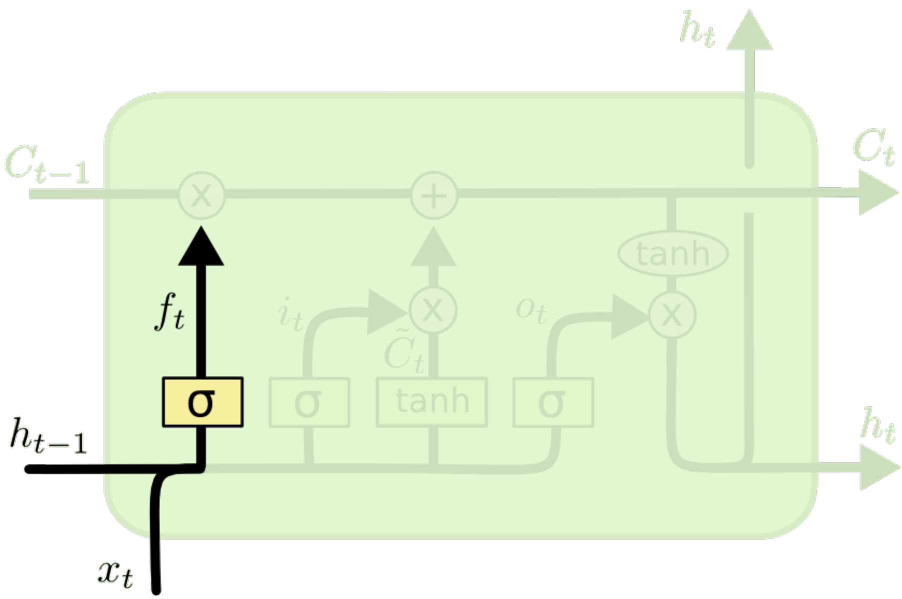
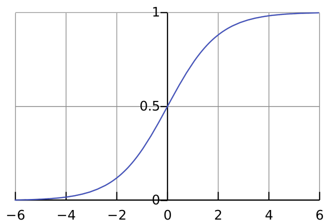


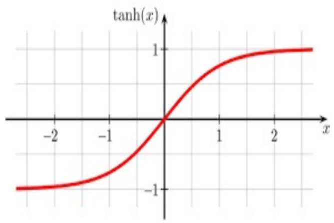
Figure by Christopher Olah (colah.github.io)

LSTMS (LONG SHORT-TERM MEMORY NETWORKS

sigmoid:
[0,1]



tanh:
[-1,1]



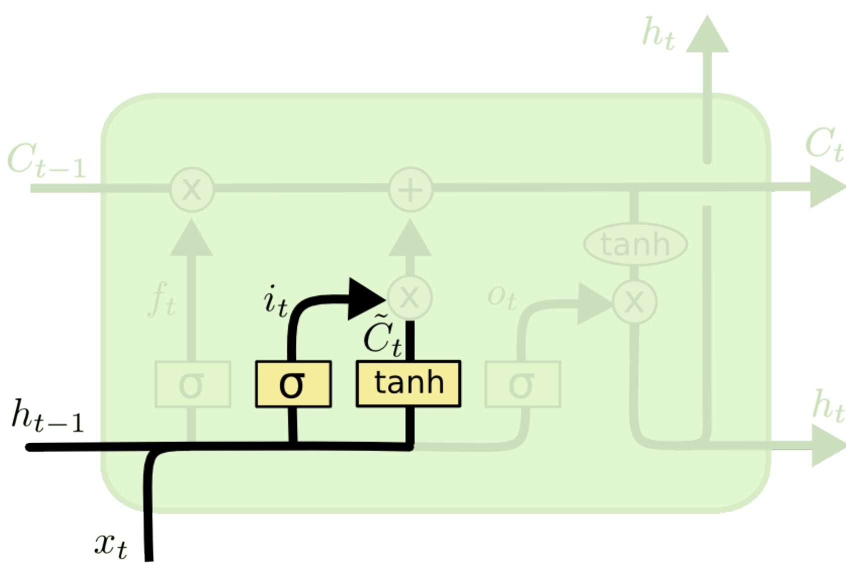


Figure by Christopher Olah (colah.github.io)

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

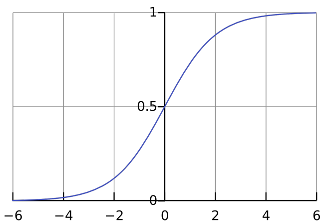
$$i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)})$$

New cell content (temp):

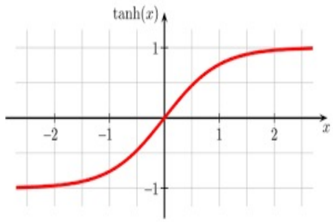
$$\tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)})$$

LSTMS (LONG SHORT-TERM MEMORY NETWORKS

sigmoid:
[0,1]



tanh:
[-1,1]



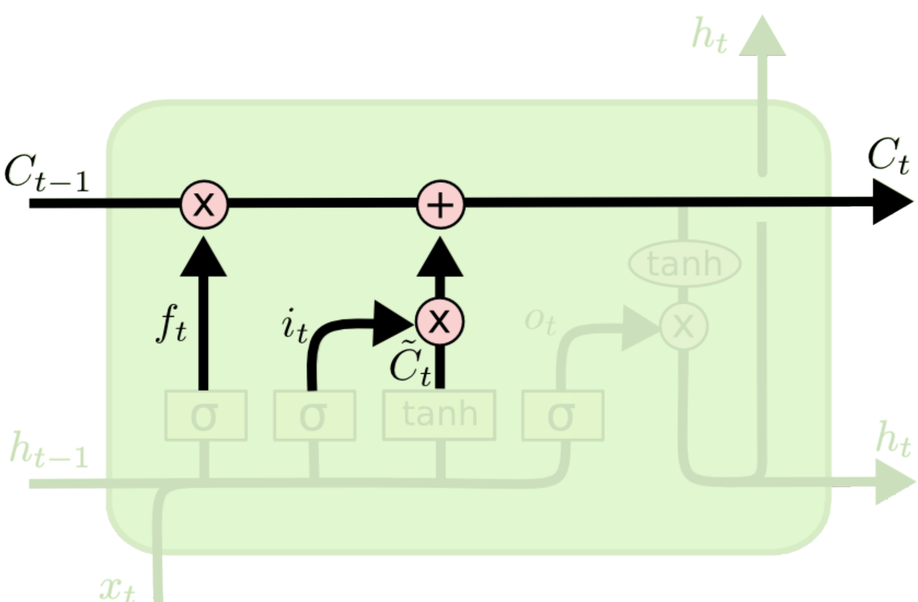


Figure by Christopher Olah (colah.github.io)

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

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

LSTMS (LONG SHORT-TERM MEMORY NETWORKS

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:

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

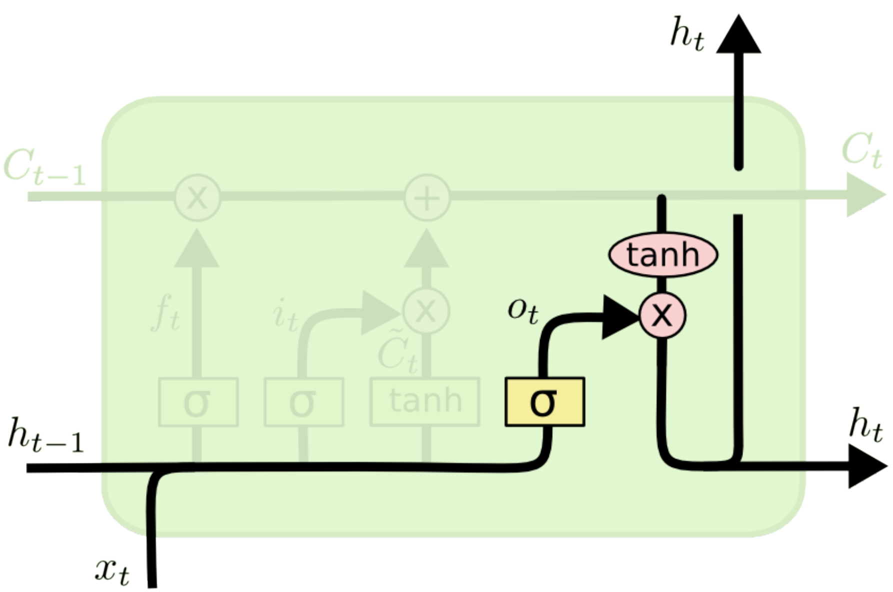


Figure by Christopher Olah (colah.github.io)

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

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

LSTMS (LONG SHORT-TERM MEMORY NETWORKS)

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):

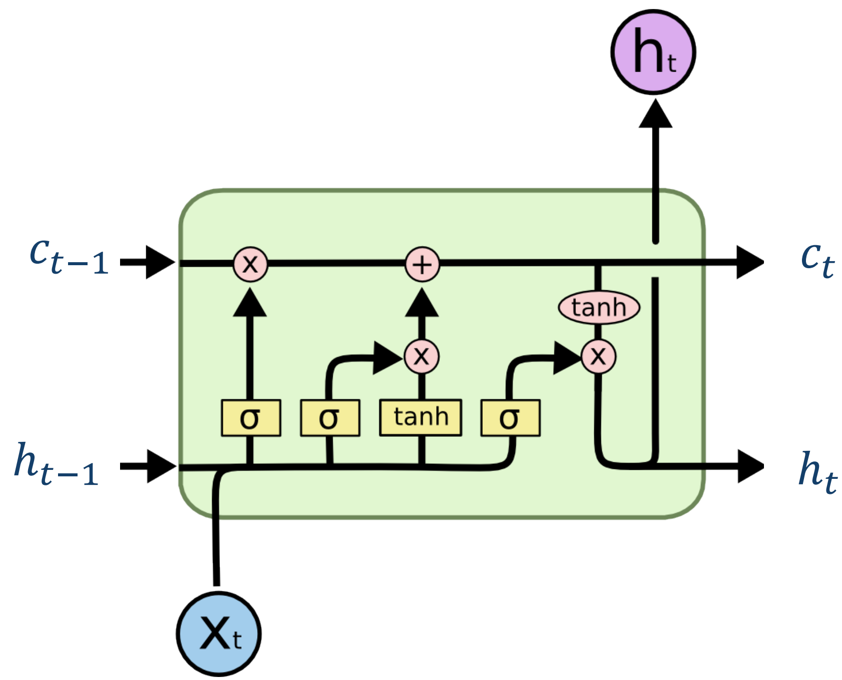
$$\tilde{c}_t = \tanh(U^{(c)} x_t + W^{(c)} h_{t-1} + b^{(c)})$$

New cell content:
- mix old cell with the new temp cell

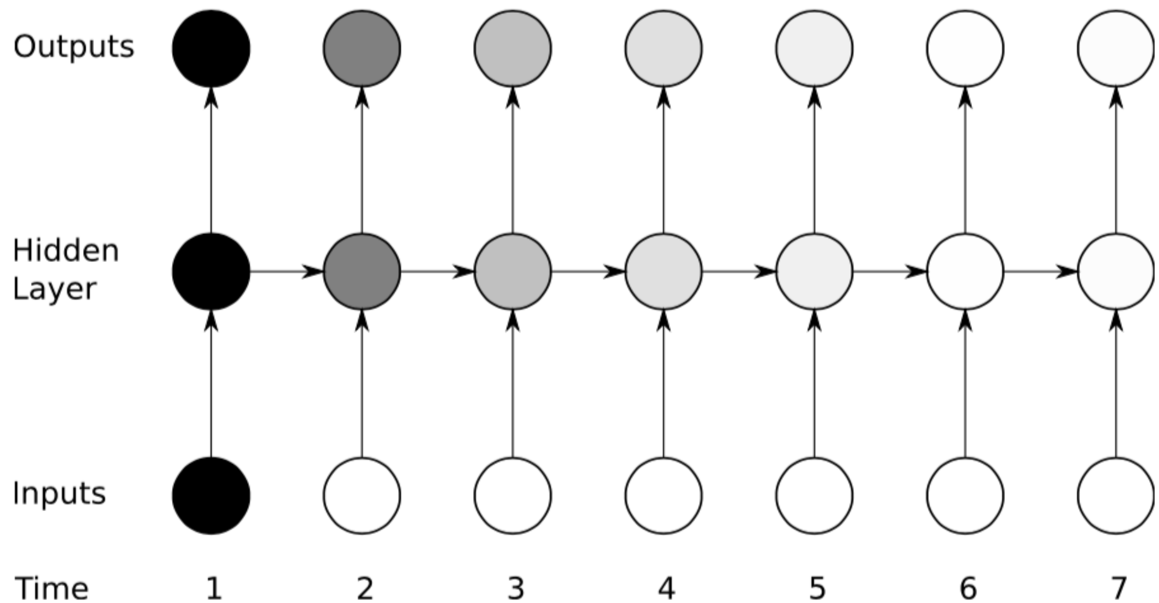
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

Hidden state:

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



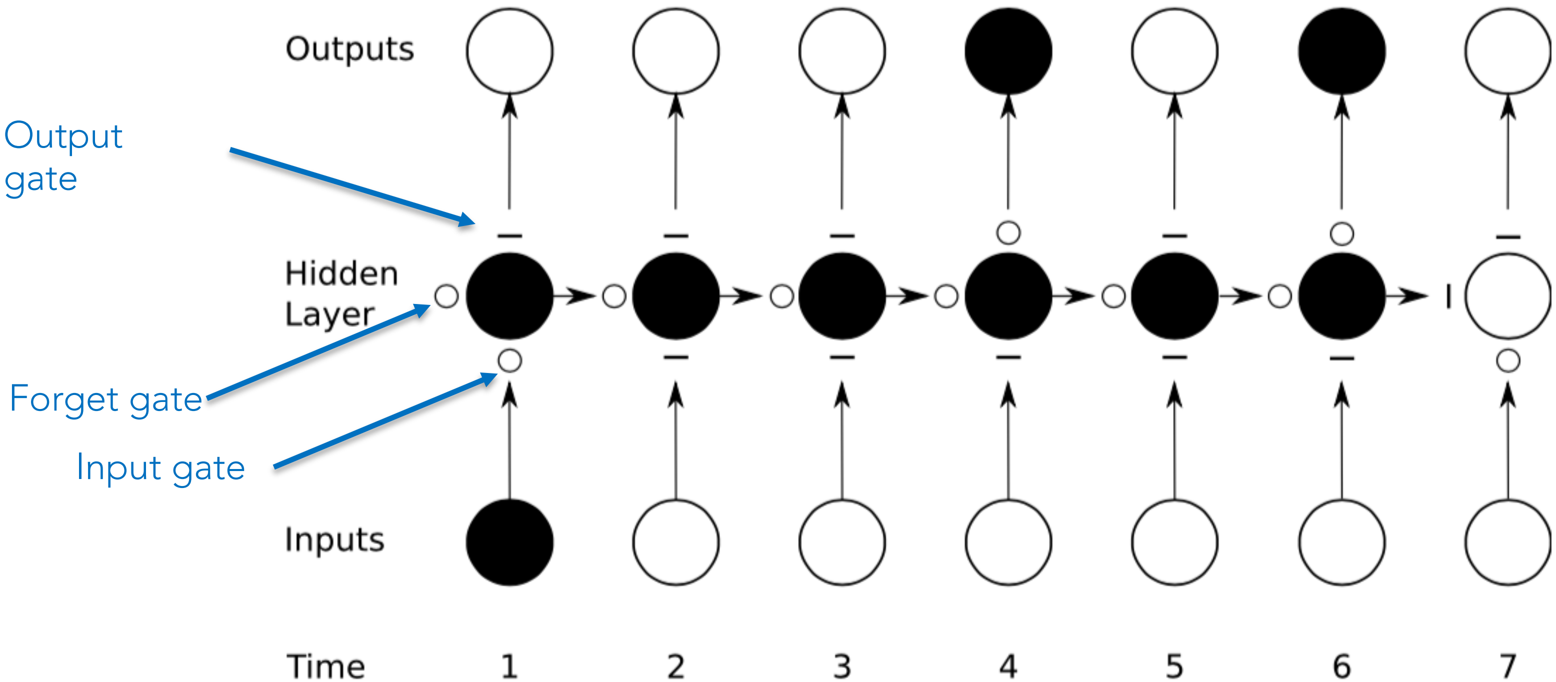
vanishing gradient problem for RNNs.



- 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 information 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

Recurrent Neural Networks (RNNs)

- 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): (Cho et al, 2014)

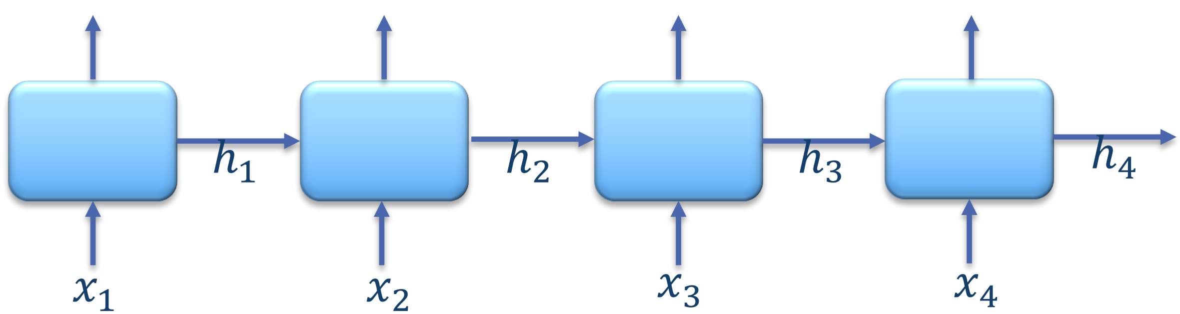
$$z_t = \sigma(U^{(z)}x_t + W^{(z)}h_{t-1} + b^{(z)})$$

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

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

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

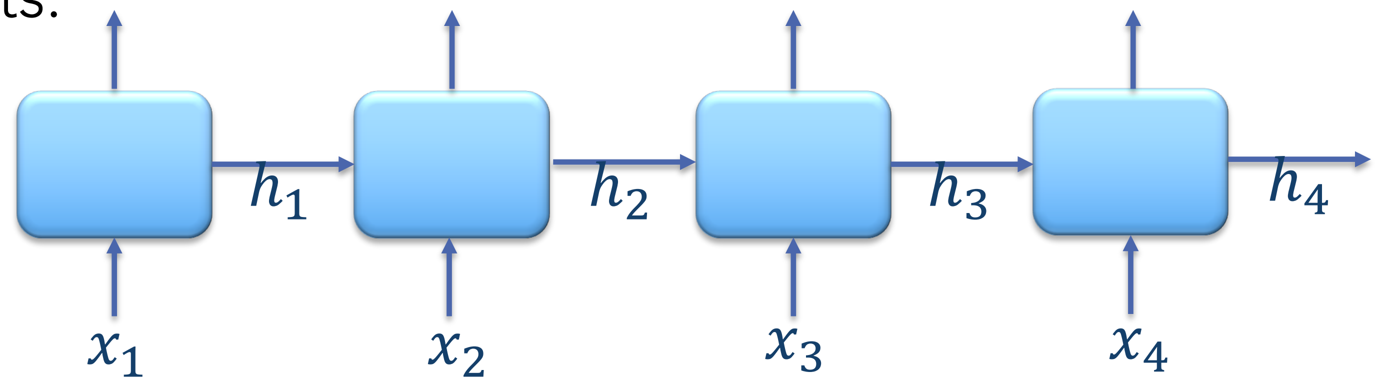
Z: Update gate
R: Reset gate



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

RNN Learning: Backprop Through Time (BPTT)

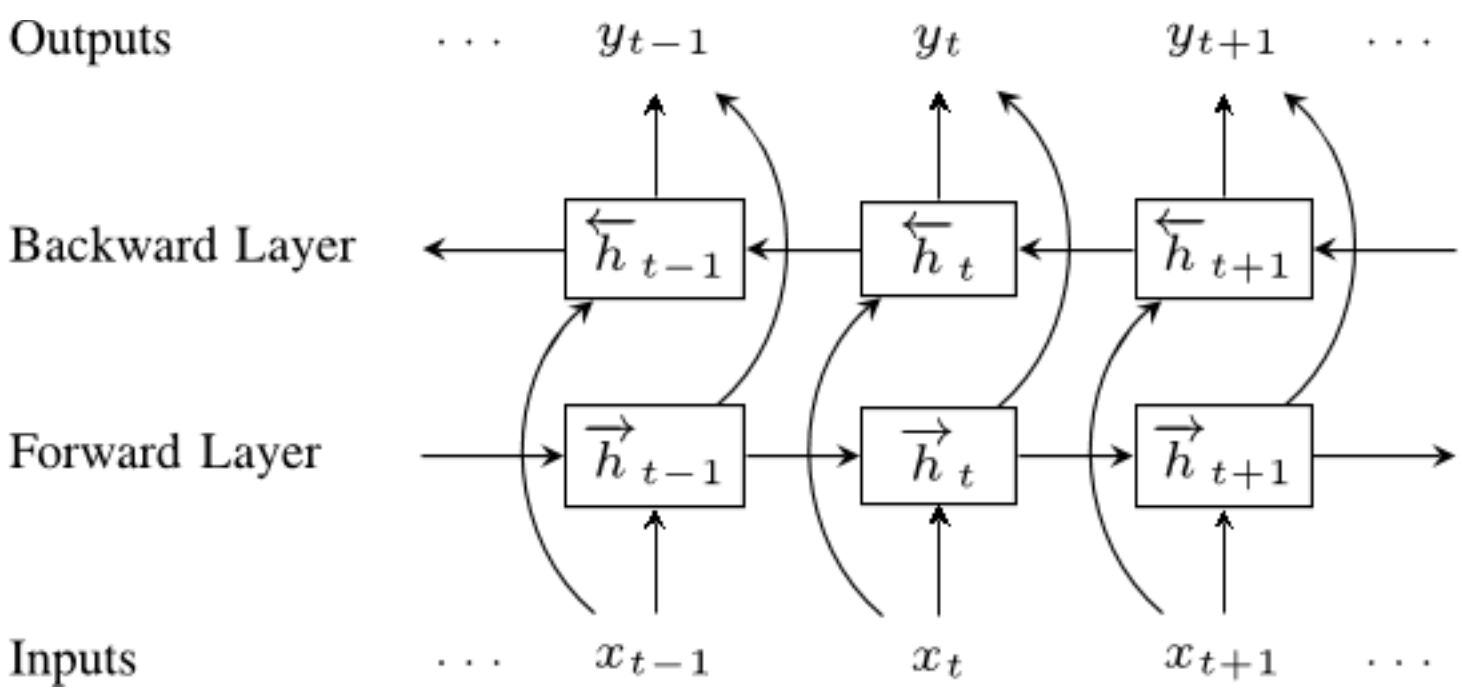
- Similar to backprop with non-recurrent NNs
- But unlike feedforward (non-recurrent) NNs, each unit in the computation graph repeats the exact same parameters…
- Backprop gradients of the parameters of each unit as if they are different parameters
- When updating the parameters using the gradients, use the average gradients throughout the entire chain of units.



Gates

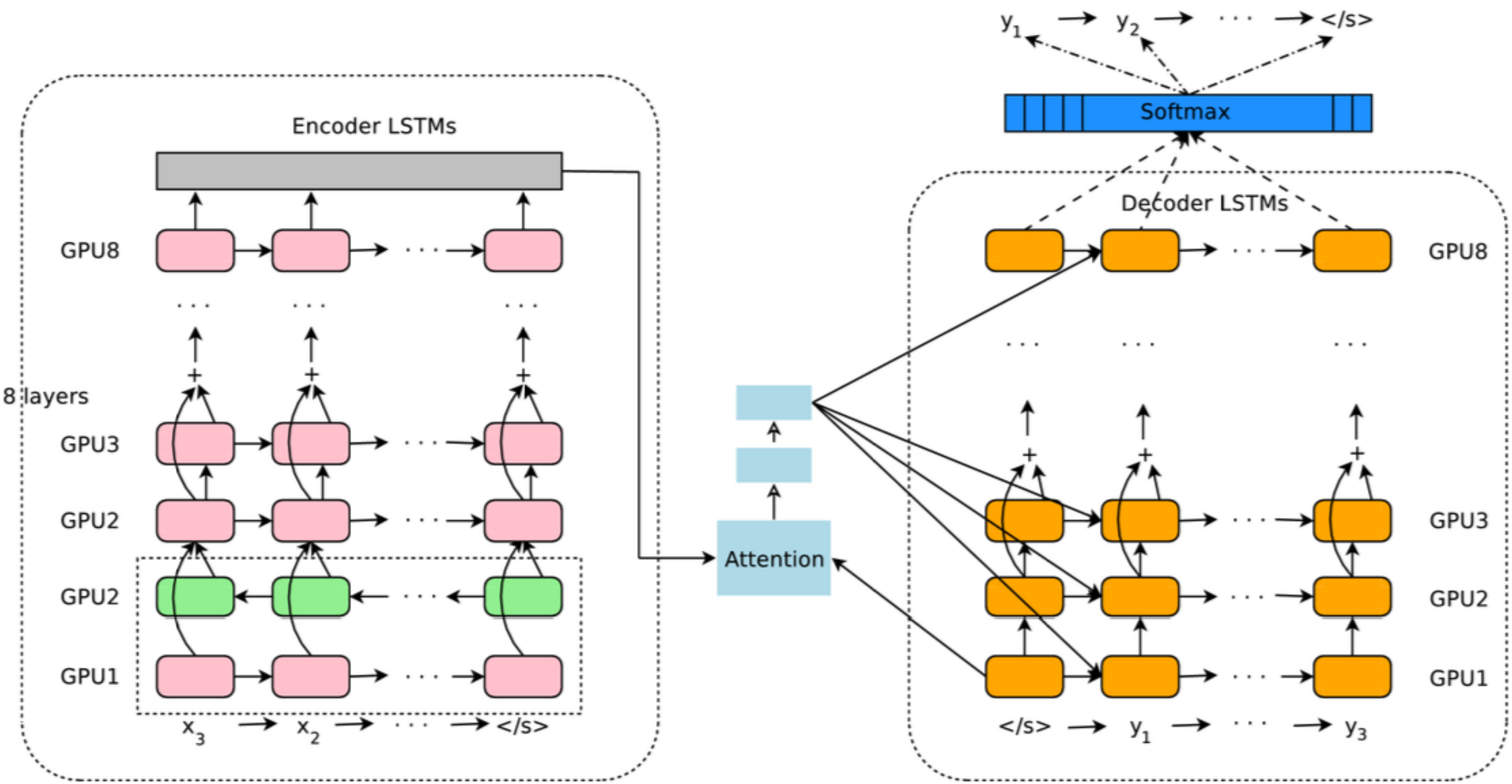
- Gates contextually control information flow
- Open/close with sigmoid
- In LSTMs and GRUs, they are used to (contextually) maintain longer term history

Bi-directional RNNs



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

Google NMT (Oct 2016)



Tree LSTMs

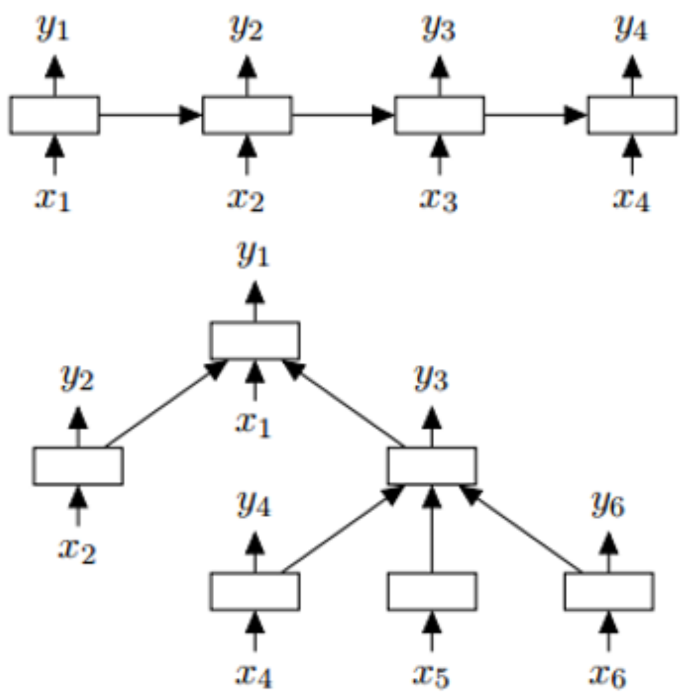
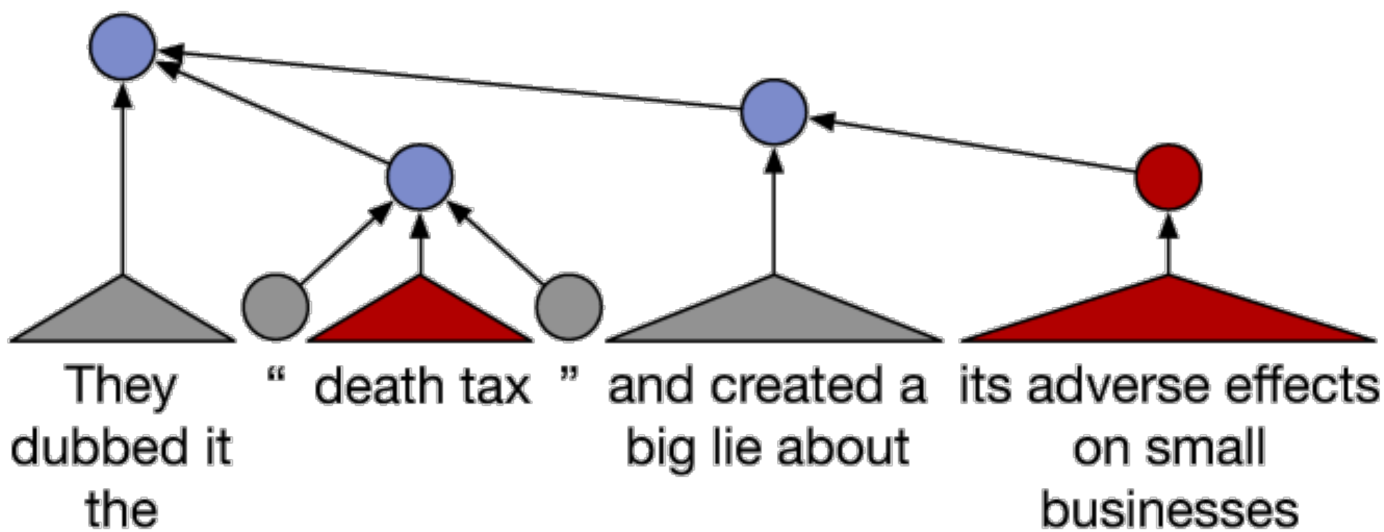


Figure 1: **Top:** A chain-structured LSTM network. **Bottom:** A tree-structured LSTM network with arbitrary branching factor.

- Are tree LSTMs more expressive than sequence LSTMs?
- I.e., *recursive* vs *recurrent*
- When Are Tree Structures Necessary for Deep Learning of Representations? Jiwei Li, Minh-Thang Luong, Dan Jurafsky and Eduard Hovy. EMNLP, 2015.

Recursive Neural Networks

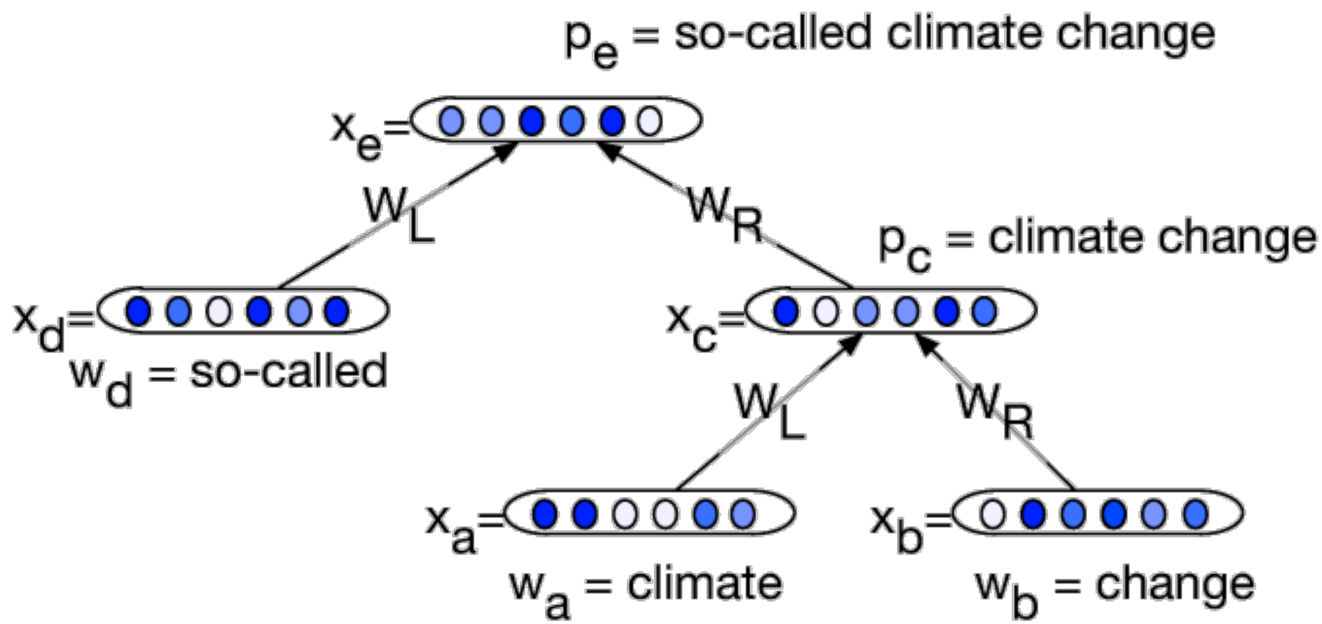
- Sometimes, inference over a tree structure makes more sense than sequential structure
- An example of compositionality in ideological bias detection (red → conservative, blue → liberal, gray → neutral) in which modifier phrases and punctuation cause polarity switches at higher levels of the parse tree



Example from Iyyer et al., 2014

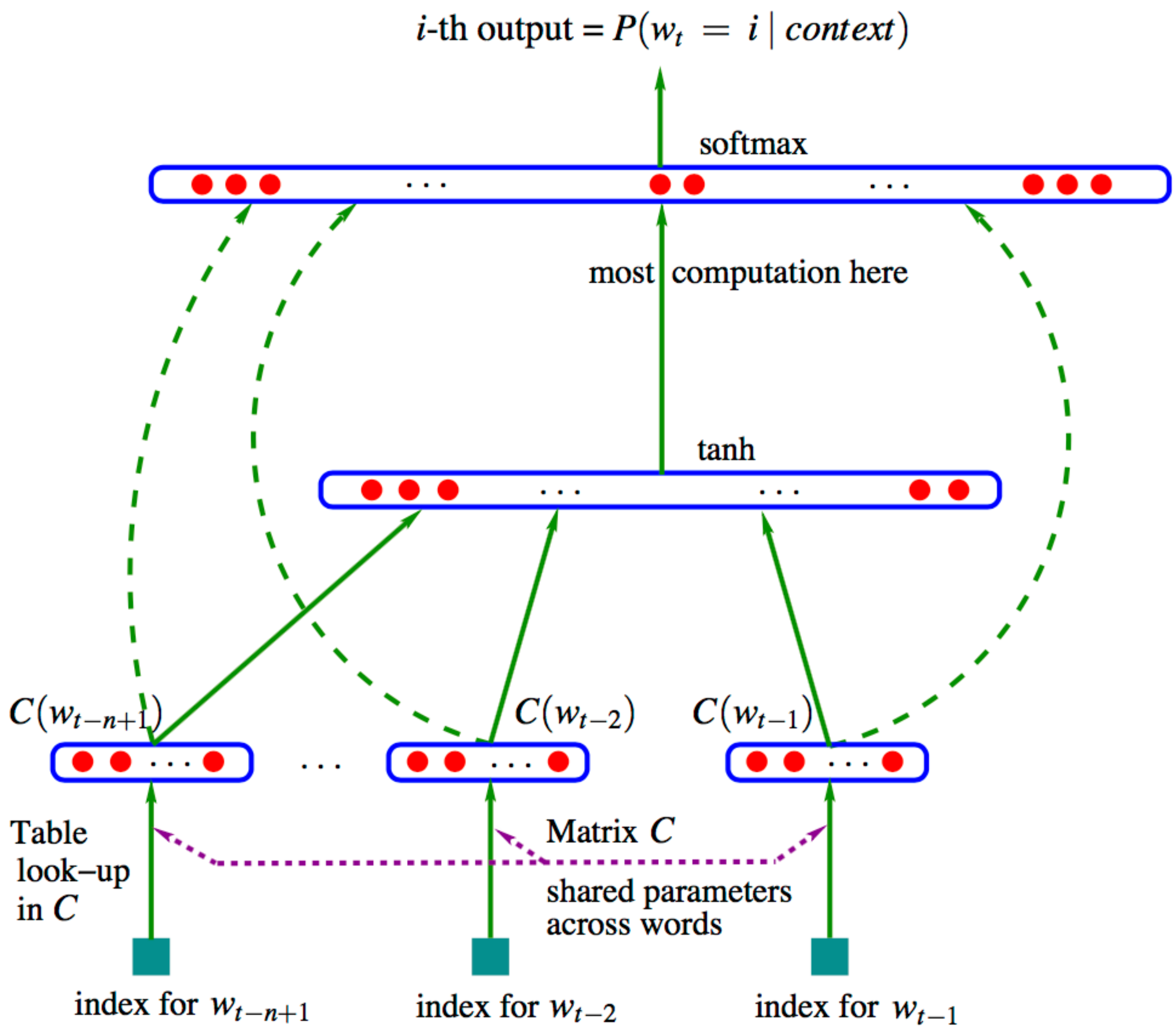
Recursive Neural Networks

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

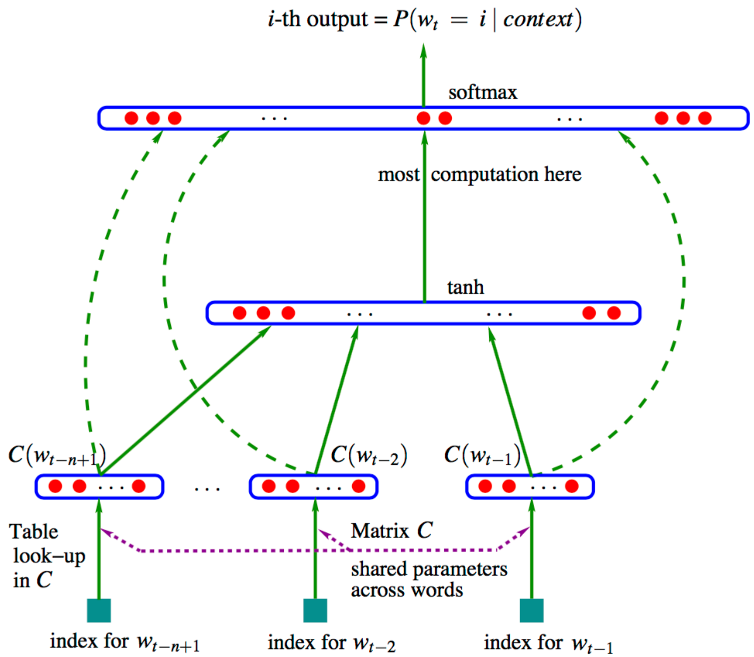


Example from Iyyer et al., 2014

Neural Probabilistic Language Model (Bengio 2003)



Neural Probabilistic Language Model (Bengio 2003)



- Each word prediction is a separate feed forward neural network
- Feedforward NNLM is a Markovian language model
- Dashed lines show optional direct connections

$$NN_{DMLP1}(\mathbf{x}) = [\tanh(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1), \mathbf{x}]W^2 + \mathbf{b}^2$$

- $\mathbf{W}^1 \in \mathbb{R}^{d_{\text{in}} \times d_{\text{hid}}}$, $\mathbf{b}^1 \in \mathbb{R}^{1 \times d_{\text{hid}}}$; first affine transformation
- $\mathbf{W}^2 \in \mathbb{R}^{(d_{\text{hid}} + d_{\text{in}}) \times d_{\text{out}}}$, $\mathbf{b}^2 \in \mathbb{R}^{1 \times d_{\text{out}}}$; second affine transformation

ATTENTION!

Encoder – Decoder Architecture

Sequence-to-Sequence

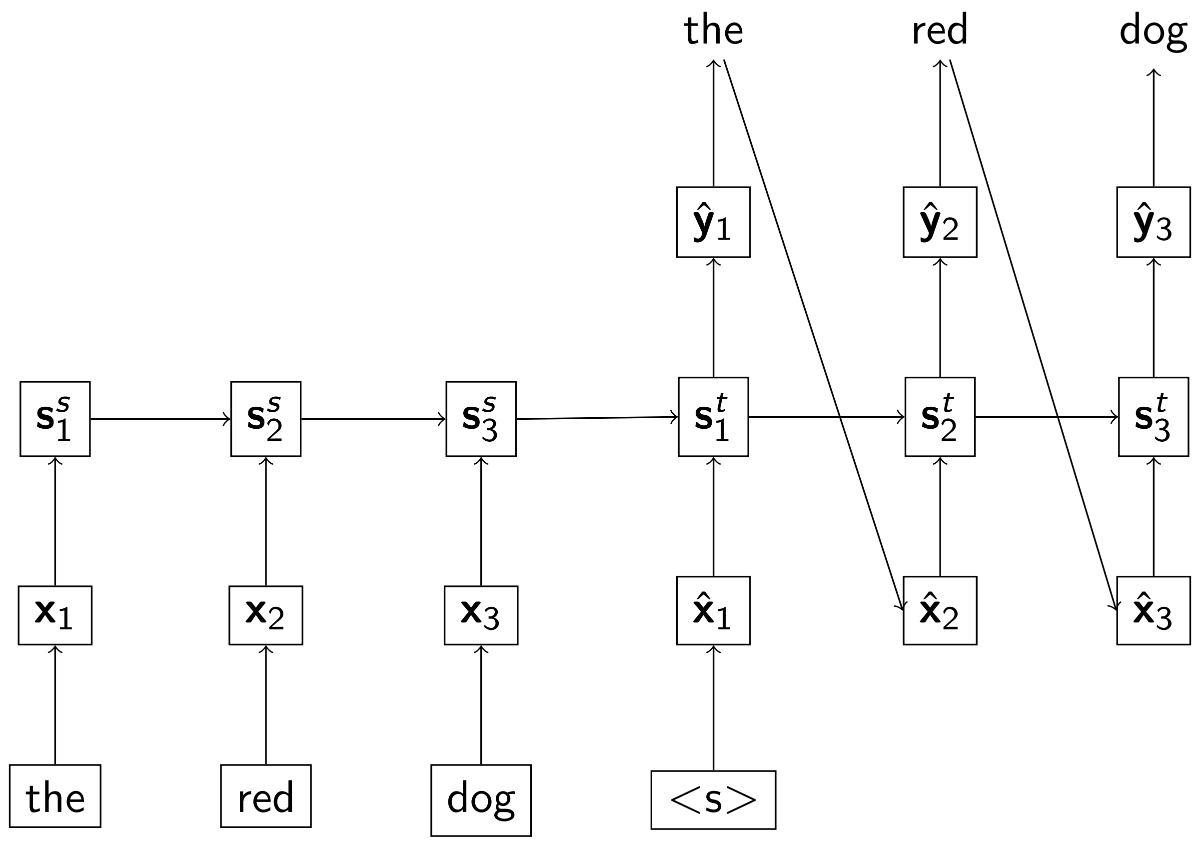


Diagram borrowed from Alex Rush

Trial: Hard Attention

- At each step generating the target word $\mathbf{s}_i^t$
- Compute the best alignment to the source word $\mathbf{s}_j^s$
- And incorporate the source word to generate the target word

$$y_i^t = \mathrm{argmax}_y O(y, s_i^t, s_j^s)$$

- Contextual *hard* alignment. How?

$$z_j = \tanh([s_i^t, s_j^s]W + b)$$

$$j = \mathrm{argmax}_j z_j$$

- Problem?

Attention: Soft Alignments

- At each step generating the target word $\mathbf{s}_i^t$
- Compute the attention $\mathbf{c}$ to the source sequence $\mathbf{s}^s$
- And incorporate the attention to generate the target word

$$y_i^t = \text{argmax}_y O(y, s_i^t, s_j^s)$$

- Contextual *attention* as *soft* alignment. How?

$$z_j = \tanh([s_i^t, s_j^s]W + b)$$

$$\alpha = \text{softmax}(z)$$

$$c = \sum_j \alpha_j s_j^s$$

  - Step-1: compute the attention weights
  - Step-2: compute the attention vector as interpolation

Attention

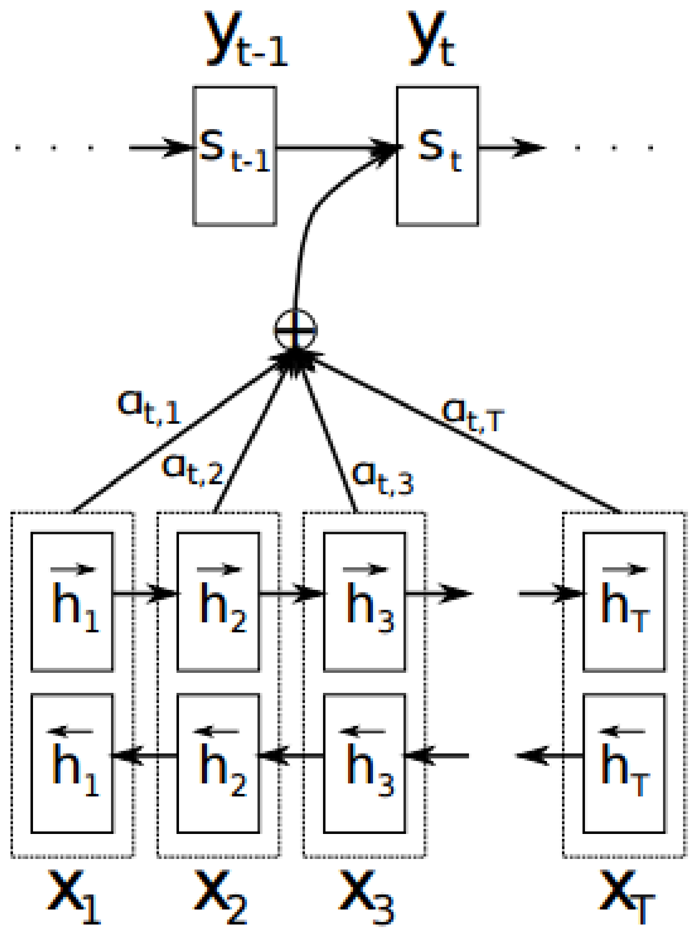


Diagram borrowed from Alex Rush

Attention parameterization

- Feedforward NNs

$$z_j = \tanh([s_i^t; s_j^s]W + b)$$

$$z_j = \tanh([s_i^t; s_j^s; s_i^t \circ s_j^s]W + b)$$

- Dot product

$$z_j = s_i^t \cdot s_j^s$$

- Cosine similarity

$$z_j = \frac{s_i^t \cdot s_j^s}{||s_i^t|| \, ||s_j^s||}$$

- Bi-linear models

$$z_j = {s_i^t}^T W s_j^s$$

Learned Attention!

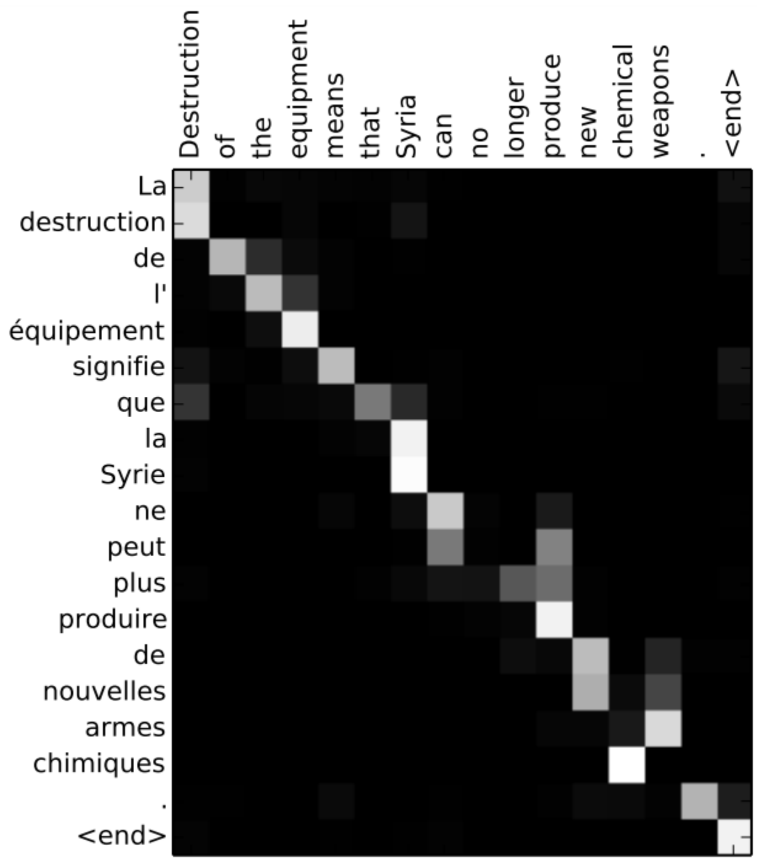


Diagram borrowed from Alex Rush

Qualitative results

Figure 2. Attention over time. As the model generates each word, its attention changes to reflect the relevant parts of the image. "soft" (top row) vs "hard" (bottom row) attention. (Note that both models generated the same captions in this example.)

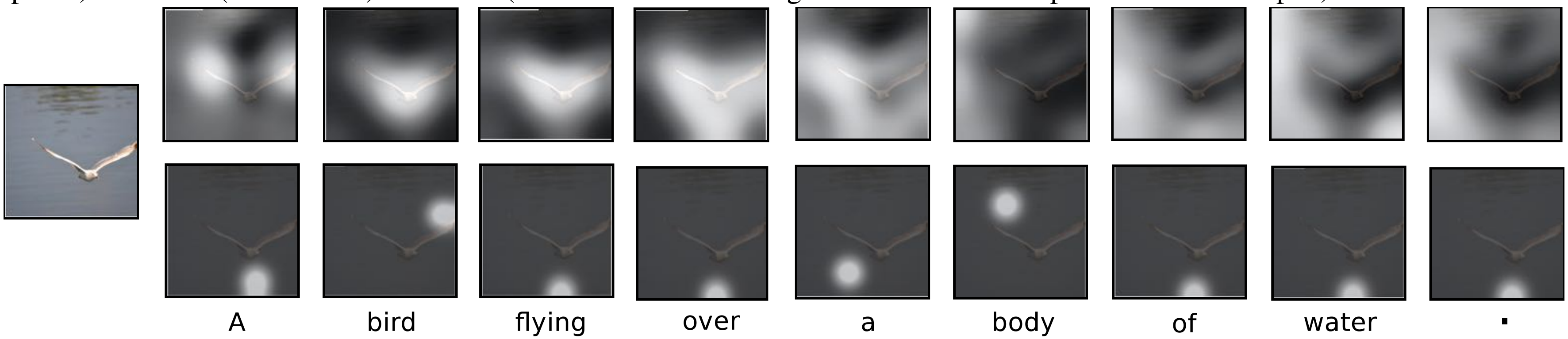
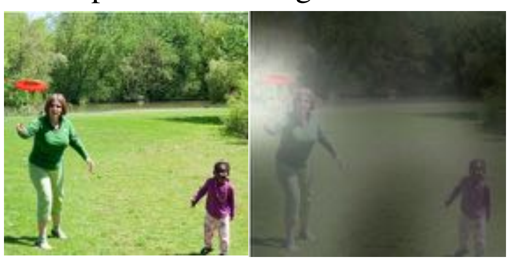
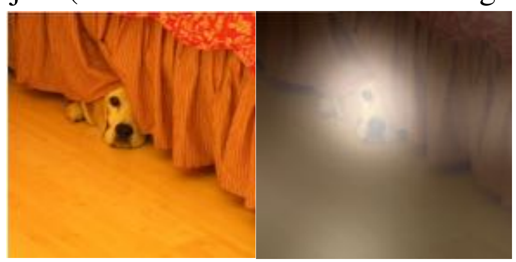


Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)

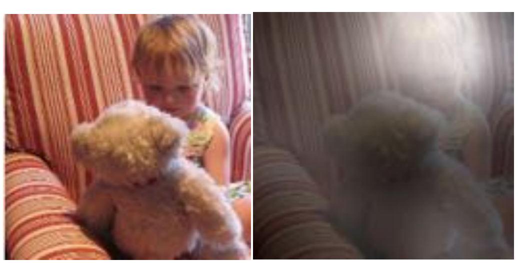


A woman is throwing a frisbee in a park.



A dog is standing on a hardwood floor.

A stop sign is on a road with a mountain in the background.

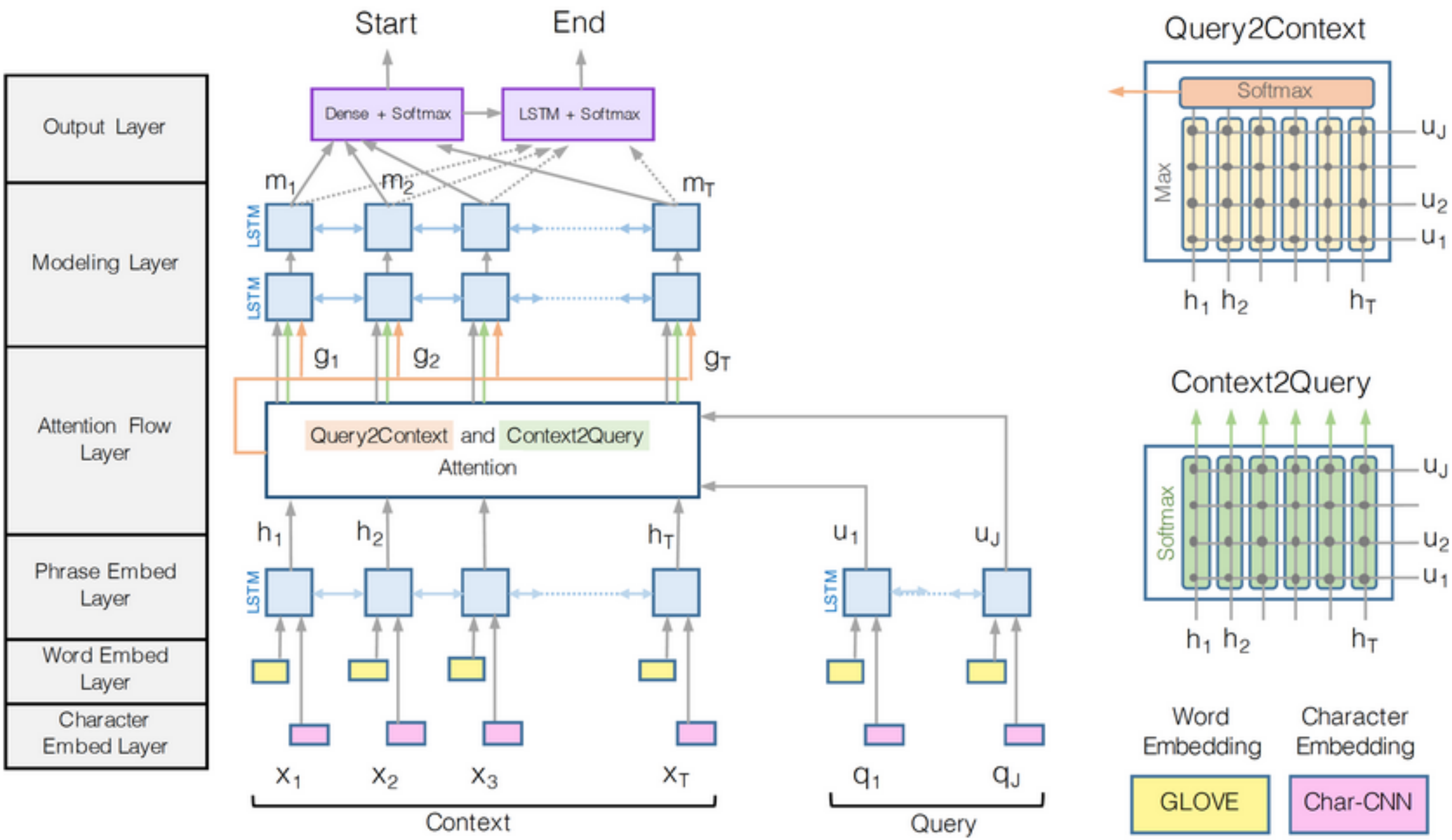


A little girl sitting on a bed with a teddy bear.

A group of people sitting on a boat in the water.

A giraffe standing in a forest with trees in the background.

BiDAF



Start
End
Dense + Softmax
LSTM + Softmax
Output Layer
Modeling Layer
Attention Flow Layer
Phrase Embed Layer
Word Embed Layer
Character Embed Layer
Query2Context and Context2Query Attention
Context
Query
Query2Context
Softmax
Max
Context2Query
Word Embedding
Character Embedding
GLOVE
Char-CNN

LEARNING: TRAINING DEEP NETWORKS

Vanishing / exploding Gradients

- Deep networks are hard to train
- Gradients go through multiple layers
- The multiplicative effect tends to lead to *exploding* or *vanishing* gradients
- Practical solutions w.r.t.
  - network architecture
  - numerical operations

Vanishing / exploding Gradients

- Practical solutions w.r.t. network architecture
  - Add skip connections to reduce distance
    - Residual networks, highway networks, …
  - Add gates (and memory cells) to allow longer term memory
    - LSTMs, GRUs, memory networks, …

Highway Network (Srivastava et al., 2015)

- A plain feedforward neural network:

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}).$$

  - H is a typical affine transformation followed by a non-linear activation

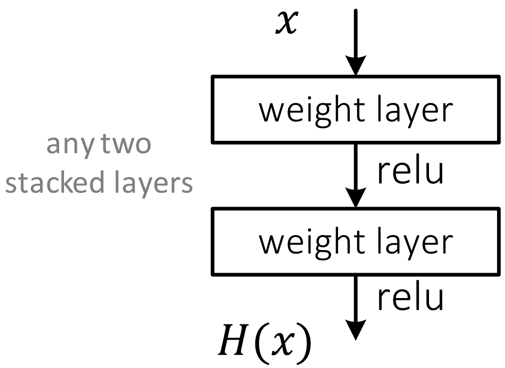
- Highway network:

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot C(\mathbf{x}, \mathbf{W_C}).$$

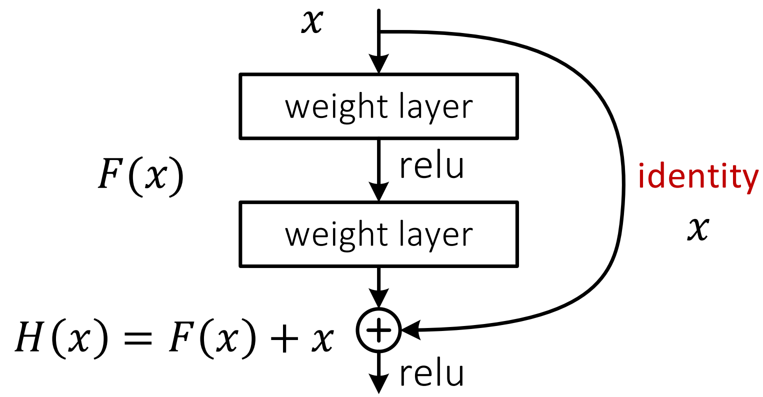
  - T is a “transform gate”
  - C is a “carry gate”
  - Often C = 1 – T for simplicity

Residual Networks

- Plaint net



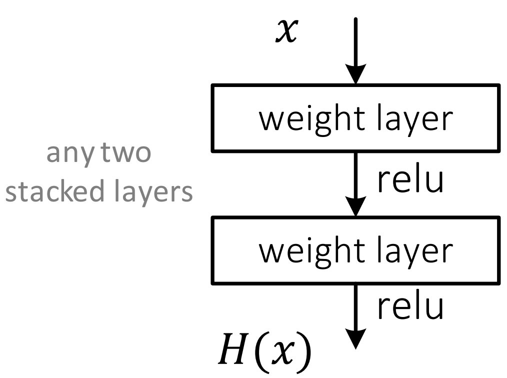
- Residual net



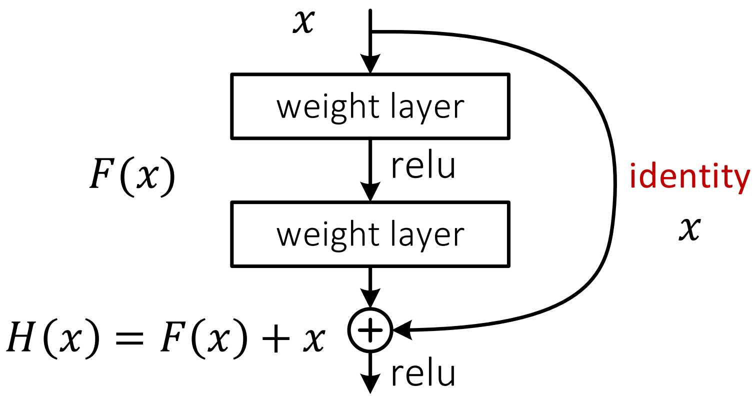
- ResNet (He et al. 2015): first very deep (152 layers) network successfully trained for object recognition

Residual Networks

- Plaint net



- Residual net

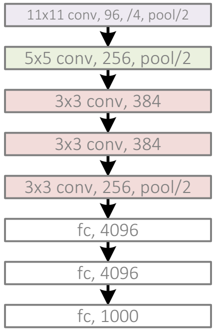


- F(x) is a residual mapping with respect to identity
- Direct input connection +x leads to a nice property w.r.t. back propagation --- more direct influence from the final loss to any deep layer
- In contrast, LSTMs & Highway networks allow for long distance input connection only through “gates”.

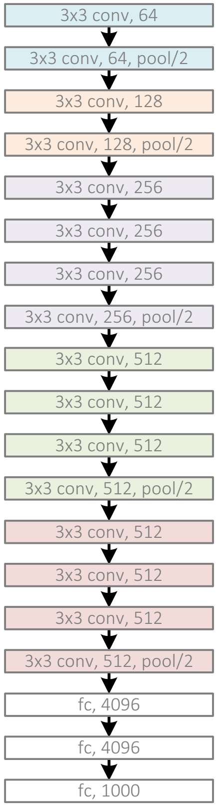
Residual Networks

Revolution of Depth

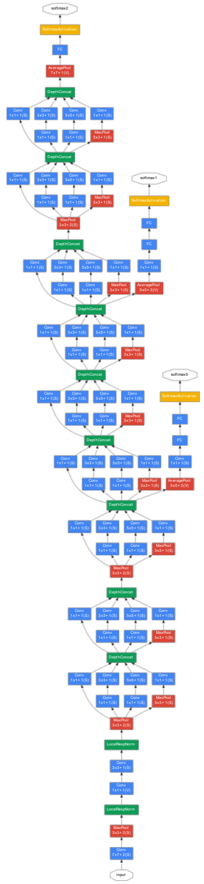
AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)



GoogleNet, 22 layers
(ILSVRC 2014)

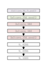


Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016

Residual Networks

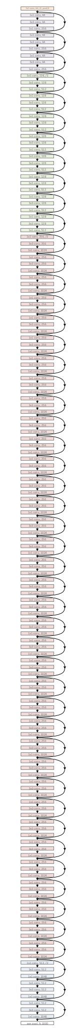
Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)



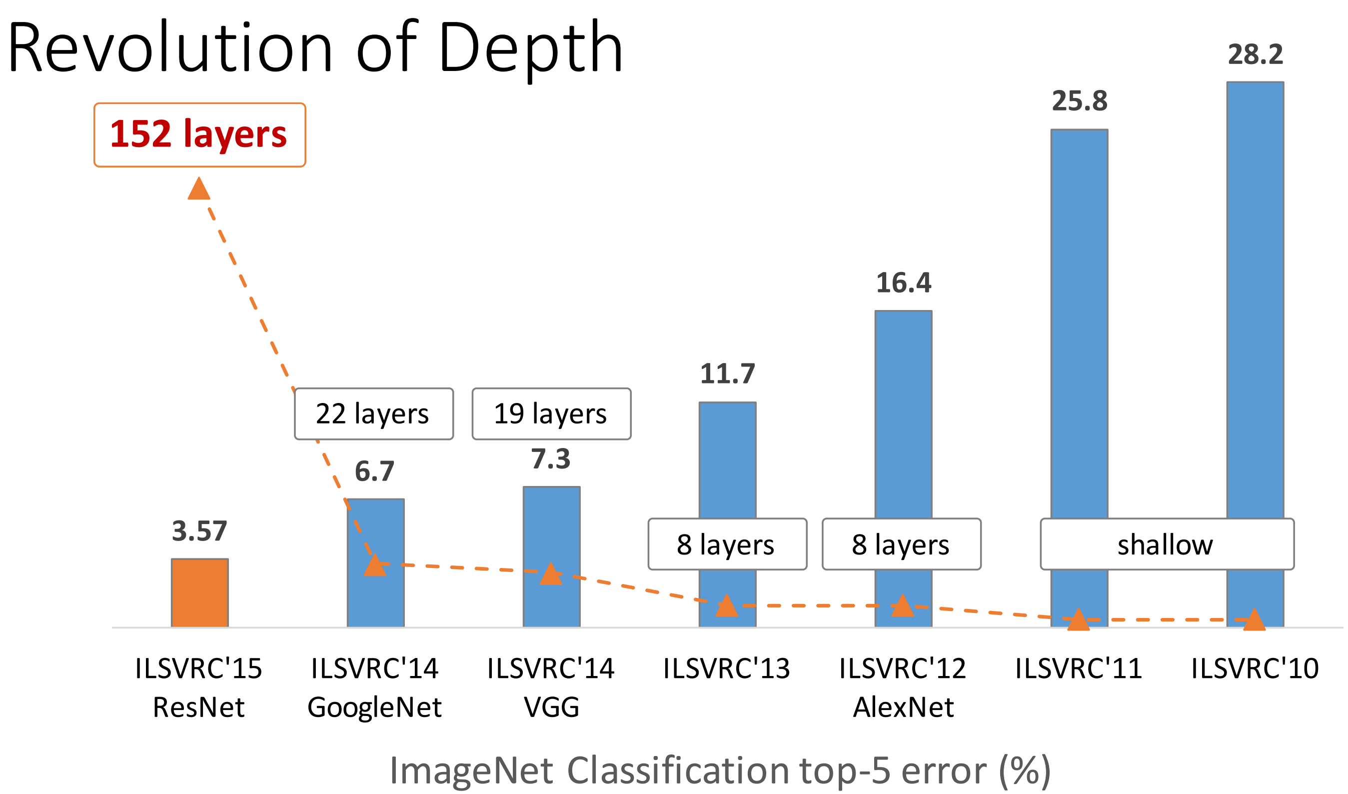
VGG, 19 layers
(ILSVRC 2014)

ResNet, 152 layers
(ILSVRC 2015)



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

Residual Networks



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

Vanishing / exploding Gradients

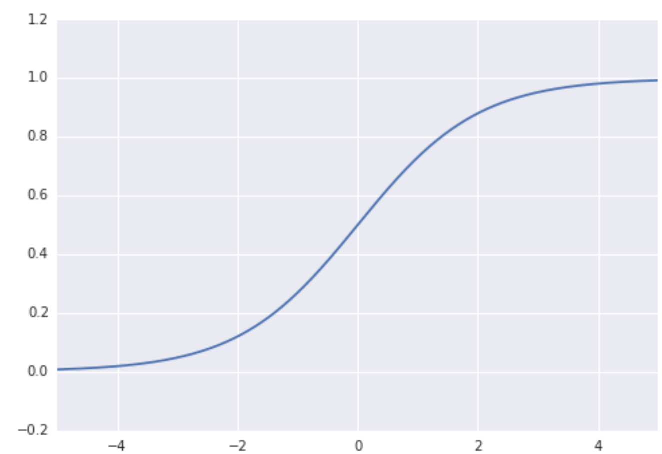
- Practical solutions w.r.t. numerical operations
  - Gradient Clipping: bound gradients by a max value
  - Gradient Normalization: renormalize gradients when they are above a fixed norm
  - Careful initialization, smaller learning rates
  - Avoid saturating nonlinearities (like tanh, sigmoid)
    - ReLU or hard-tanh instead

Sigmoid

- Often used for gates
- Pro: neuron-like, differentiable
- Con: gradients saturate to zero almost everywhere except x near zero => vanishing gradients
- Batch normalization helps

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$



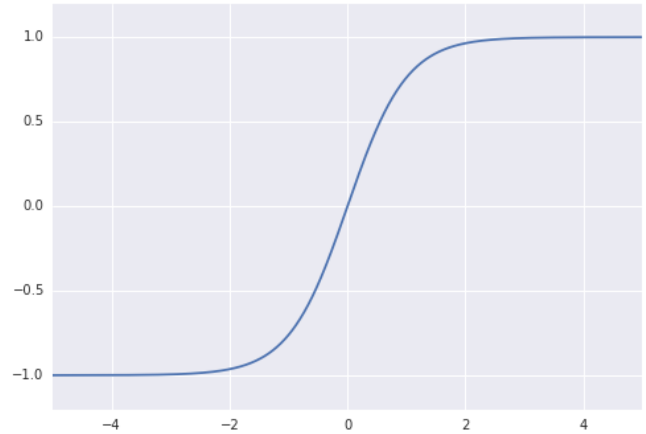
Tanh

- Often used for hidden states & cells in RNNs, LSTMs
- Pro: differentiable, often converges faster than sigmoid
- Con: gradients easily saturate to zero => vanishing gradients

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\text{tanh'(x)} = 1 - \tanh^2(x)$$

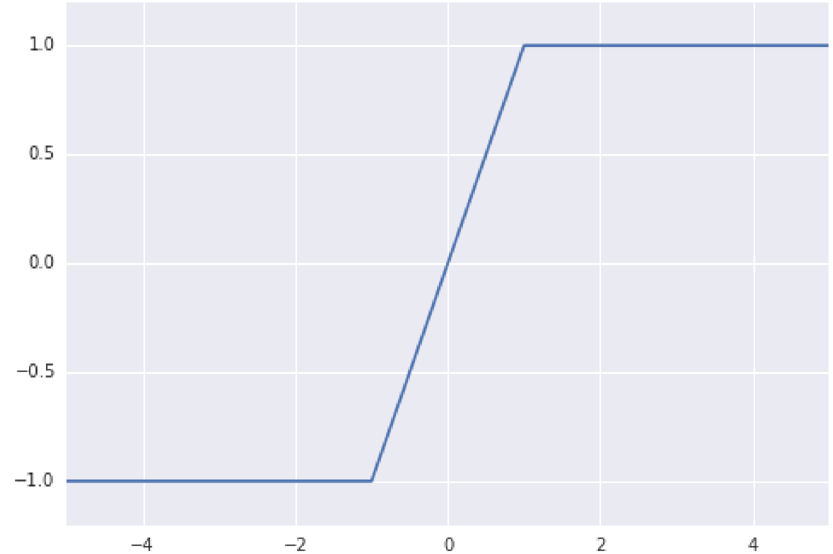
$$\tanh(x) = 2\sigma(2x) - 1$$



Hard Tanh

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

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

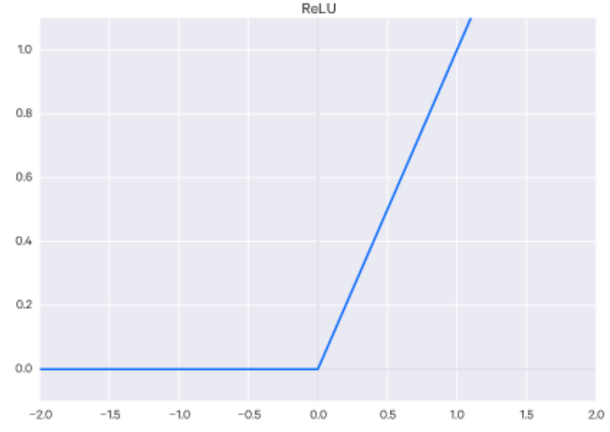


ReLU

- 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:

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

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



Vanishing / exploding Gradients

- Practical solutions w.r.t. 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 rates
  - Avoid saturating nonlinearities (like tanh, sigmoid)
    - ReLU or hard-tanh instead
  - Batch Normalization: add intermediate input normalization layers

Batch Normalization

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

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

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

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

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

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

Regularization

- Regularization by objective term

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \max\{0, 1 - (\hat{y}_c - \hat{y}_{c'})\} + \lambda||\theta||^2$$

  - Modify loss with L1 or L2 norms
- Less depth, smaller hidden states, early stopping
- Dropout
  - Randomly delete parts of network during training
  - 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
  - Vs. Bagging

Convergence of backprop

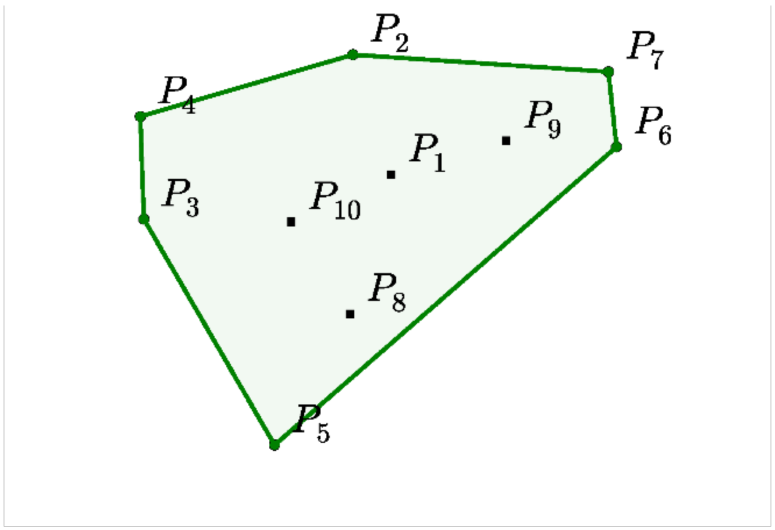
- Without non-linearity or hidden layers, learning is convex optimization
  - Gradient descent reaches **global minima**
- Multilayer neural nets (with nonlinearity) are **not convex**
  - Gradient descent gets stuck in local minima
  - Selecting number of hidden units and layers = fuzzy process
  - NNs have made a HUGE comeback in the last few years
    - Neural nets are back with a new name
      - Deep belief networks
      - Huge error reduction when trained with lots of data on GPUs

SUPPLEMENTARY TOPICS

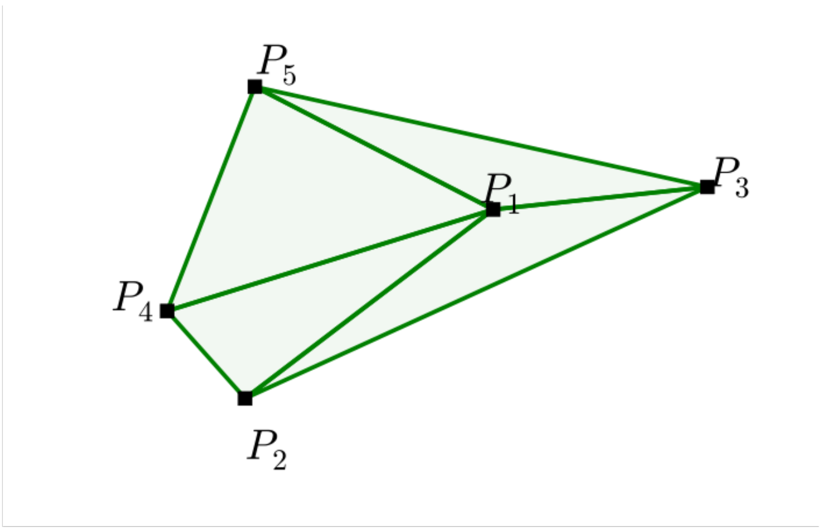
POINTER NETWORKS

Pointer Networks! (Vinyals et al. 2015)

- NNs with attention: content-based attention to input
- Pointer networks: location-based attention to input
- Applications: Convex haul, Delaunay Triangulation, Traveling Salesman

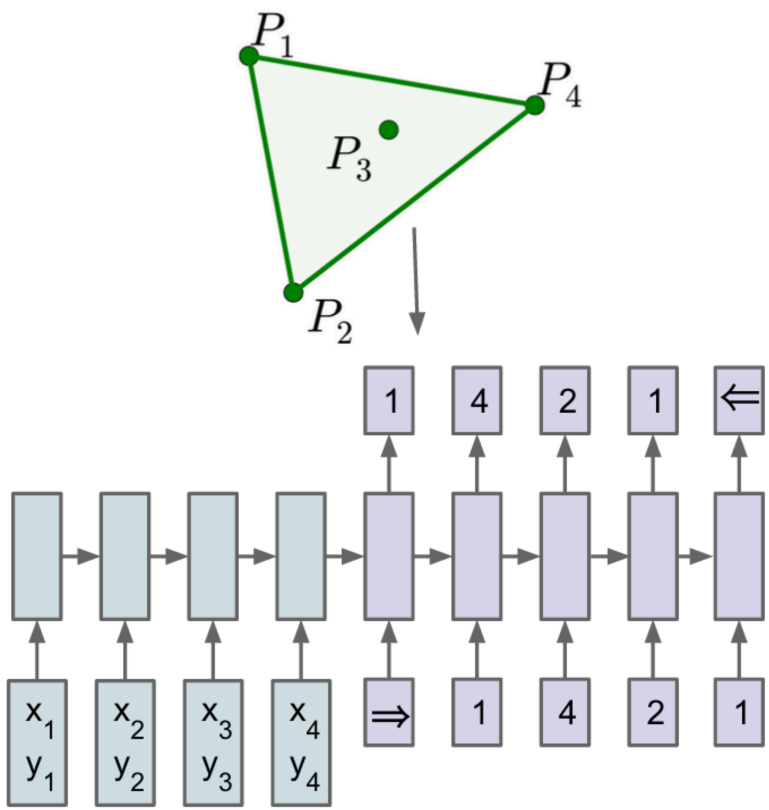


(a) Input $\mathcal{P} = \{P_1, \ldots, P_{10}\}$, and the output sequence $\mathcal{C}^{\mathcal{P}} = \{\Rightarrow, 2, 4, 3, 5, 6, 7, 2, \Leftarrow\}$ representing its convex hull.

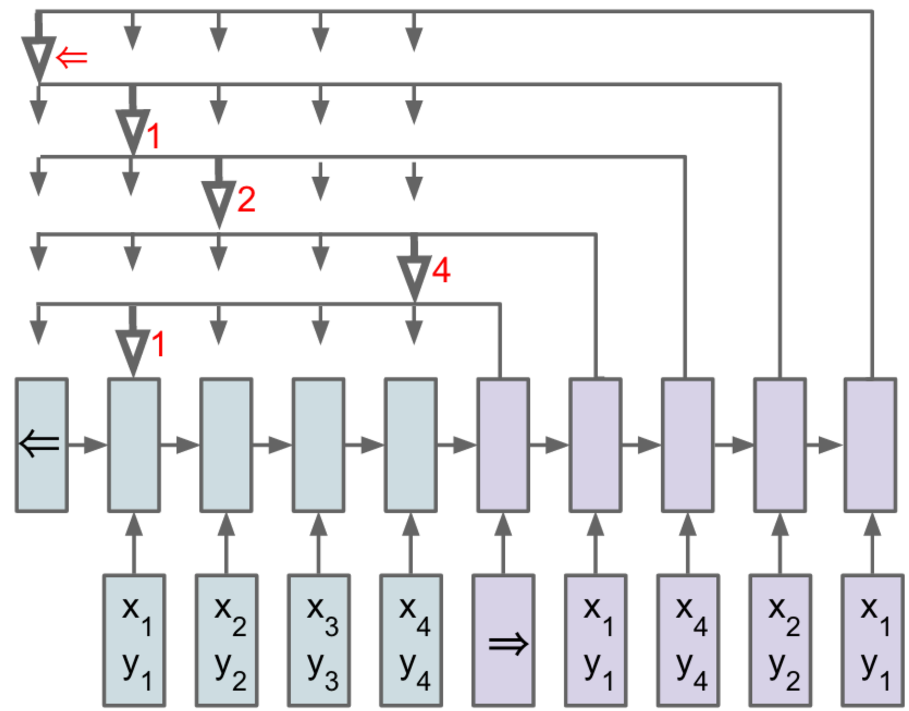


(b) Input $\mathcal{P} = \{P_1, \ldots, P_5\}$, and the output $\mathcal{C}^{\mathcal{P}} = \{\Rightarrow, (1, 2, 4), (1, 4, 5), (1, 3, 5), (1, 2, 3), \Leftarrow\}$ representing its Delaunay Triangulation.

Pointer Networks



(a) Sequence-to-Sequence



(b) Ptr-Net

Pointer Networks

Attention Mechanism vs Pointer Networks

$$e_{ij} = v_a^\top \tanh\left(W_a s_{i-1} + U_a h_j\right)$$

$$\alpha_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)}$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

Attention mechanism

$$e_{ij} = v_a^\top \tanh\left(W_a s_{i-1} + U_a h_j\right)$$

$$p(C_i | C_1, \ldots, C_{i-1}, \mathcal{P}) = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)}$$

Ptr-Net

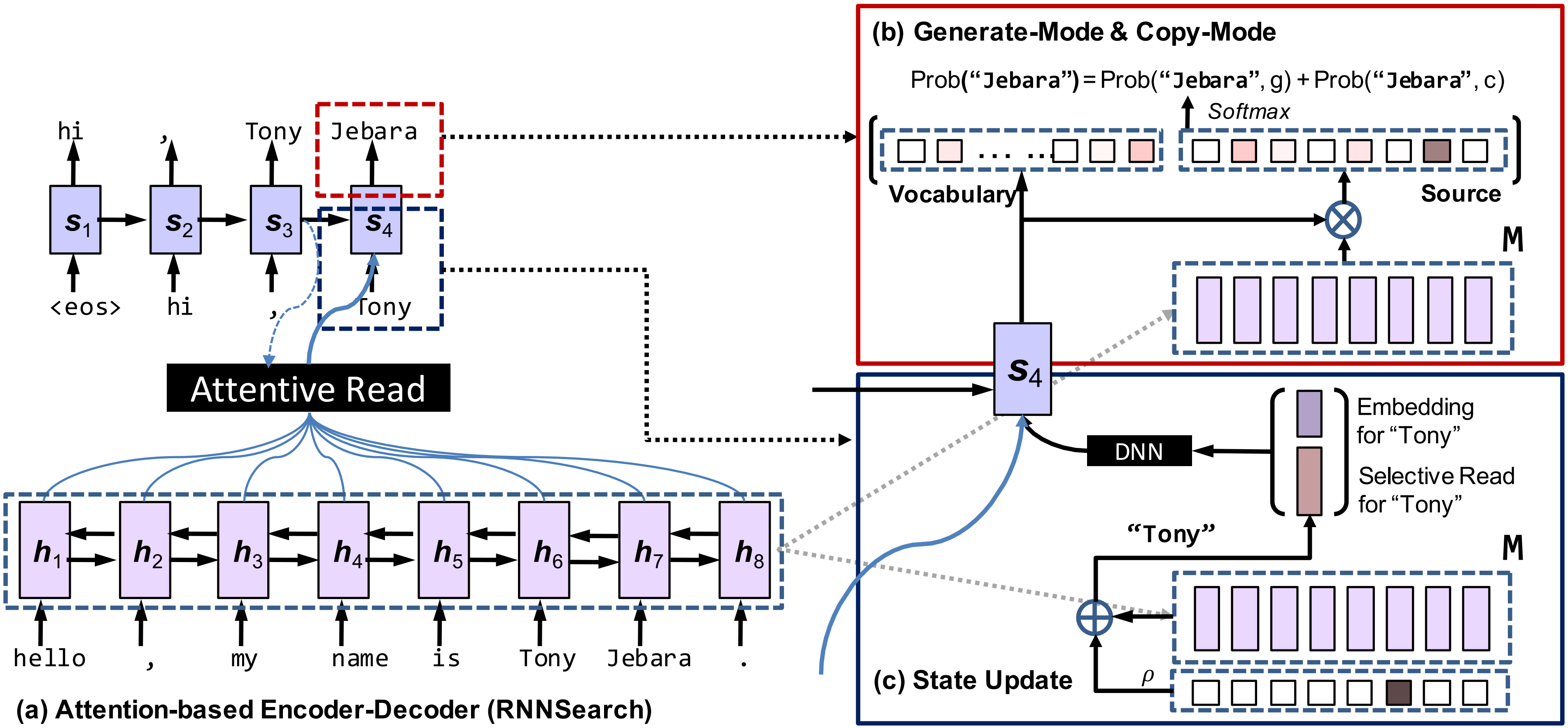
Softmax normalizes the vector e_{ij} to be an output distribution over the dictionary of inputs

Diagram borrowed from Keon Kim

CopyNet (Gu et al. 2016)

- Conversation
  - I: Hello Jack, my name is Chandralekha
  - R: Nice to meet you, Chandralekha
  - I: This new guy doesn’t perform exactly as expected.
  - R: what do you mean by “doesn’t perform exactly as expected?”
- Translation

CopyNet (Gu et al. 2016)



CopyNet (Gu et al. 2016)

- Key idea: interpolation between generation model & copy model

$$p(y_t|\mathbf{s}_t, y_{t-1}, \mathbf{c}_t, \mathbf{M}) = p(y_t, \mathsf{g}|\mathbf{s}_t, y_{t-1}, \mathbf{c}_t, \mathbf{M}) + p(y_t, \mathsf{c}|\mathbf{s}_t, y_{t-1}, \mathbf{c}_t, \mathbf{M}) \quad (4)$$

$$p(y_t, \mathsf{g}|\cdot) = \begin{cases} \frac{1}{Z} e^{\psi_g(y_t)}, & y_t \in \mathcal{V} \\ 0, & y_t \in \mathcal{X} \cap \bar{V} \\ \frac{1}{Z} e^{\psi_g(\mathrm{UNK})} & y_t \notin \mathcal{V} \cup \mathcal{X} \end{cases} \quad (5)$$

$$p(y_t, \mathsf{c}|\cdot) = \begin{cases} \frac{1}{Z} \sum_{j:x_j=y_t} e^{\psi_c(x_j)}, & y_t \in \mathcal{X} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

Generate-Mode: The same scoring function as in the generic RNN encoder-decoder (Bahdanau et al., 2014) is used, i.e.

$$\psi_g(y_t = v_i) = \mathbf{v}_i^\top \mathbf{W}_o \mathbf{s}_t, \quad v_i \in \mathcal{V} \cup \mathrm{UNK} \quad (7)$$

where $\mathbf{W}_o \in \mathbb{R}^{(N+1)\times d_s}$ and $\mathbf{v}_i$ is the one-hot indicator vector for v_i.

Copy-Mode: The score for "copying" the word x_j is calculated as

$$\psi_c(y_t = x_j) = \sigma\left(\mathbf{h}_j^\top \mathbf{W}_c\right) \mathbf{s}_t, \quad x_j \in \mathcal{X} \quad (8)$$

CONVOLUTION NEURAL NETWORK

Next several slides borrowed from Alex Rush

Models with Sliding Windows

- Classification/prediction with sliding windows
  - E.g., neural language model
- Feature representations with sliding window
  - E.g., sequence tagging with CRFs or structured perceptron

$$[w_1 \; w_2 \; w_3 \; w_4 \; w_5] \; w_6 \; w_7 \; w_8$$

$$w_1 \; [w_2 \; w_3 \; w_4 \; w_5 \; w_6] \; w_7 \; w_8$$

$$w_1 \; w_2 \; [w_3 \; w_4 \; w_5 \; w_6 \; w_7] \; w_8$$

$$\vdots$$

Sliding Windows w/ Convolution

Let our input be the embeddings of the full sentence, $\mathbf{X} \in \mathbb{R}^{n \times d^0}$

$$\mathbf{X} = [v(w_1), v(w_2), v(w_3), \ldots, v(w_n)]$$

Define a window model as $NN_{window} : \mathbb{R}^{1 \times (d_{\mathrm{win}} d^0)} \mapsto \mathbb{R}^{1 \times d_{\mathrm{hid}}}$,

$$NN_{window}(\mathbf{x}_{win}) = \mathbf{x}_{win}\mathbf{W}^1 + \mathbf{b}^1$$

The convolution is defined as $NN_{conv} : \mathbb{R}^{n \times d^0} \mapsto \mathbb{R}^{(n - d_{\mathrm{win}} + 1) \times d_{\mathrm{hid}}}$,

$$NN_{conv}(\mathbf{X}) = \tanh \begin{bmatrix} NN_{window}(\mathbf{X}_{1:d_{\mathrm{win}}}) \\ NN_{window}(\mathbf{X}_{2:d_{\mathrm{win}}+1}) \\ \vdots \\ NN_{window}(\mathbf{X}_{n-d_{\mathrm{win}}:n}) \end{bmatrix}$$

Pooling Operations

- Pooling "over-time" operations $f : \mathbb{R}^{n \times m} \mapsto \mathbb{R}^{1 \times m}$
  1. $f_{max}(\mathbf{X})_{1,j} = \max_i X_{i,j}$
  2. $f_{min}(\mathbf{X})_{1,j} = \min_i X_{i,j}$
  3. $f_{mean}(\mathbf{X})_{1,j} = \sum_i X_{i,j} / n$

$$f(\mathbf{X}) = \begin{bmatrix} \Downarrow & \Downarrow & \ldots \\ \Downarrow & \Downarrow & \ldots \\ & \vdots & \\ \Downarrow & \Downarrow & \ldots \end{bmatrix} = \begin{bmatrix} & \ldots & \end{bmatrix}$$

Convolution + Pooling

$$\hat{y} = \text{softmax}(f_{max}(NN_{conv}(\mathbf{X}))\mathbf{W}^2 + \mathbf{b}^2)$$

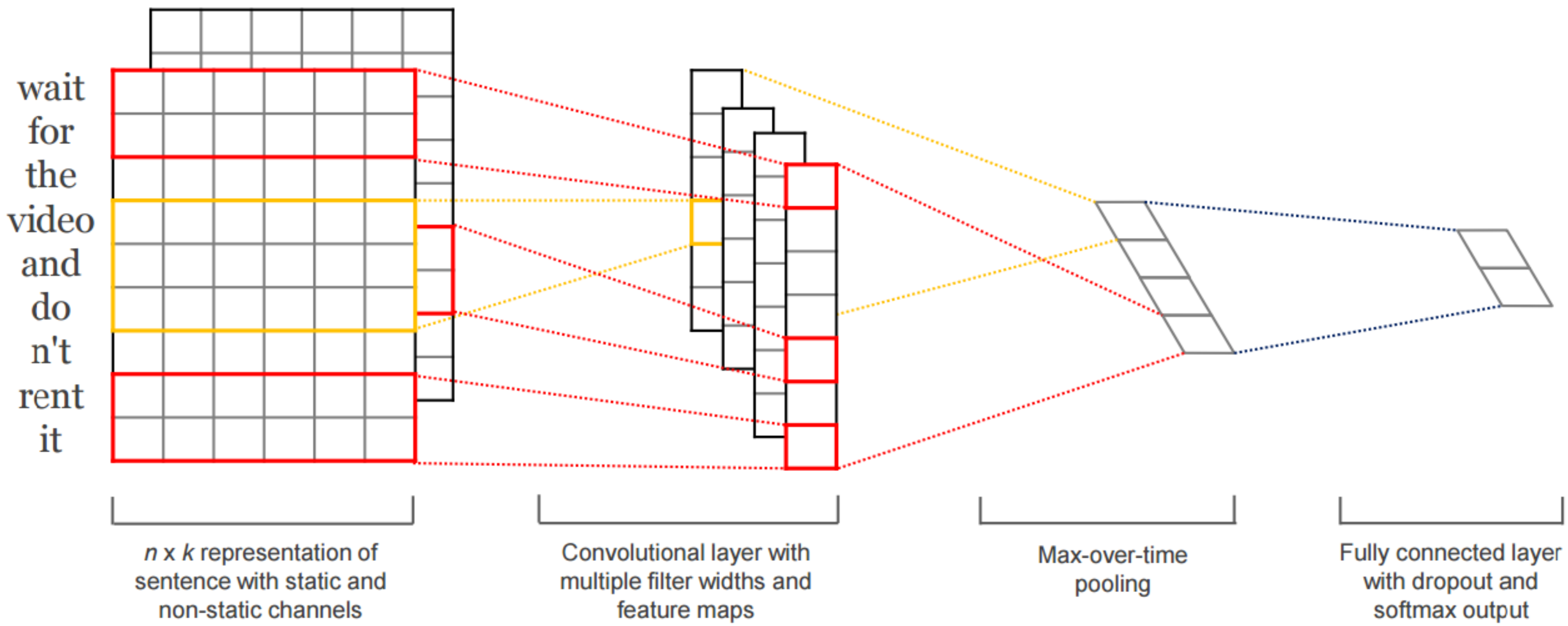
- $\mathbf{W}^2 \in \mathbb{R}^{d_{\text{hid}} \times d_{\text{out}}}$, $\mathbf{b}^2 \in \mathbb{R}^{1 \times d_{\text{out}}}$
- Final linear layer $\mathbf{W}^2$ uses learned window features

Multiple Convolutions

$$\hat{y} = \text{softmax}([f(NN^1_{conv}(\mathbf{X})), f(NN^2_{conv}(\mathbf{X})), \ldots, f(NN^f_{conv}(\mathbf{X}))]\mathbf{W}^2 + \mathbf{b}^2)$$

- Concat several convolutions together.
- Each NN^1, NN^2, etc uses a different d_{win}
- Allows for different window-sizes (similar to multiple n-grams)

Convolution Diagram (kim 2014)



- $n = 9$, $d_{\text{hid}} = 4$, $d_{\text{out}} = 2$
- red- $d_{\text{win}} = 2$, blue- $d_{\text{win}} = 3$, (ignore back channel)

Text Classification (Kim 2014)

Model	MR	SST-1	SST-2	Subj	TREC	CR	MPQA
CNN-rand	76.1	45.0	82.7	89.6	91.2	79.8	83.4
CNN-static	81.0	45.5	86.8	93.0	92.8	84.7	**89.6**
CNN-non-static	**81.5**	48.0	87.2	93.4	93.6	84.3	89.5
CNN-multichannel	81.1	47.4	**88.1**	93.2	92.2	**85.0**	89.4
RAE (Socher et al., 2011)	77.7	43.2	82.4	–	–	–	86.4
MV-RNN (Socher et al., 2012)	79.0	44.4	82.9	–	–	–	–
RNTN (Socher et al., 2013)	–	45.7	85.4	–	–	–	–
DCNN (Kalchbrenner et al., 2014)	–	48.5	86.8	–	93.0	–	–
Paragraph-Vec (Le and Mikolov, 2014)	–	**48.7**	87.8	–	–	–	–

AlexNet (krizhevsky et al., 2012)

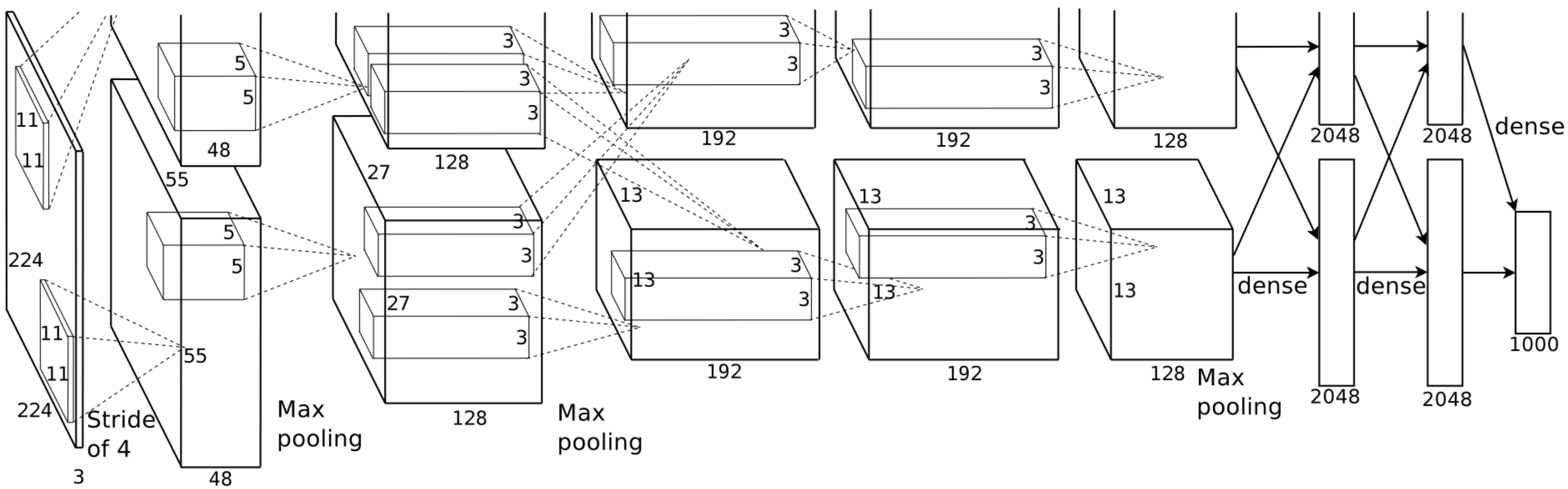


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Discussion Points

- Strength and challenges of deep learning?

- Representation learning
  - Less efforts on feature engineering (at the cost of more hyperparameter tuning!)
  - NN learned representation today is significantly better than human engineered features
  - But the architecture that works well for image representation learning is entirely different from the architecture that works well for language
- Data
  - Most success from massive amount of clean (expensive) data
  - Recent surge of data creation type papers (especially AI challenge type tasks)
  - Which significantly limits the domains & applications
  - Need stronger models for unsupervised & distantly supervised approaches

Discussion Points

- Strength and challenges of deep learning?
- Architecture
  - allows for flexible, expressive, and creative modeling
- Easier entry to the field
  - Recent breakthrough from engineering advancements than theoretic advancements
  - Several NN platforms, code sharing culture

Transformers!
a.k.a. *"Attention Is All You Need"*

Following slides borrowed from Chris Manning

GPT-2 language model (cherry-picked) output

SYSTEM PROMPT (HUMAN-WRITTEN)	*In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.*
MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)	The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. Pérez and the others then ventured further into the valley. …

METRO NEWS... BUT NOT AS YOU KNOW IT

NEWS SPORT ENTERTAINMENT SOAPS MORE TRENDING

UK WORLD WEIRD TECH

Elon Musk's OpenAI builds artificial intelligence so powerful it must be kept locked up for the good of humanity

Jasper Hamill Friday 15 Feb 2019 10:06 am

272 SHARES

Elon Musk's scientists have announced the creation of a terrifying artificial intelligence that's so smart they refused to release it to the public.

OpenAI's GPT-2 is designed to write just like a human and is an impressive leap forward capable of penning chillingly convincing text.

It was 'trained' by analysing eight million web pages and is capable of writing large tracts based upon a 'prompt' written by a real person.

But the machine mind will not be released in its fully-fledged form because of the risk of it being used for 'malicious purposes' such as generating fake news, impersonating people online, automating the production of spam or churning out 'abusive or faked content to post on social media'.

OpenAI wrote: 'Due to our concerns about malicious applications of the technology, we are not releasing the trained model.

Elon Musk @elonmusk Follow

Replying to @georgezachary

To clarify, I've not been involved closely with OpenAI for over a year & don't have mgmt or board oversight

8:19 PM - 16 Feb 2019

500 Retweets 14,573 Likes

229 500 15K

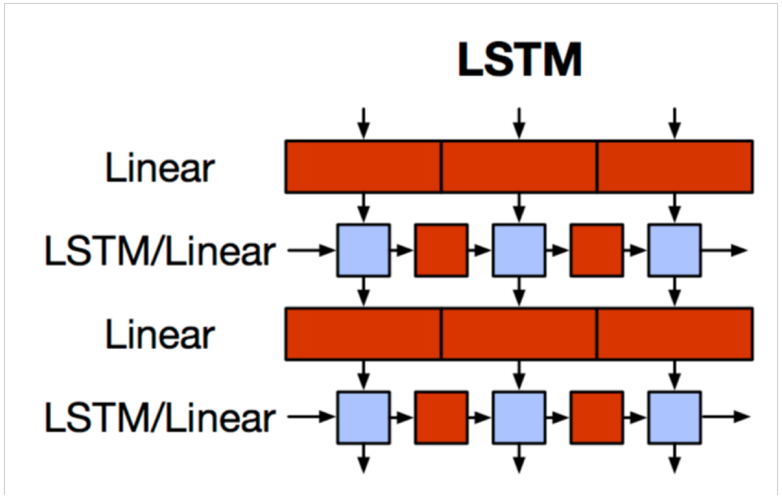
Transformer models

All of these models are Transformer architecture models … so maybe we had better learn about Transformers?

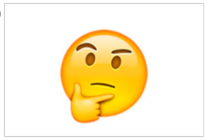
ULMfit	GPT	BERT	GPT-2
Jan 2018	June 2018	Oct 2018	Feb 2019
Training:	Training	Training	Training
1 GPU day	240 GPU days	256 TPU days ~320–560 GPU days	~2048 TPU v3 days according to a reddit thread
fast.ai	OpenAI	Google AI	OpenAI

4. The Motivation for Transformers

- We want **parallelization** but RNNs are inherently sequential



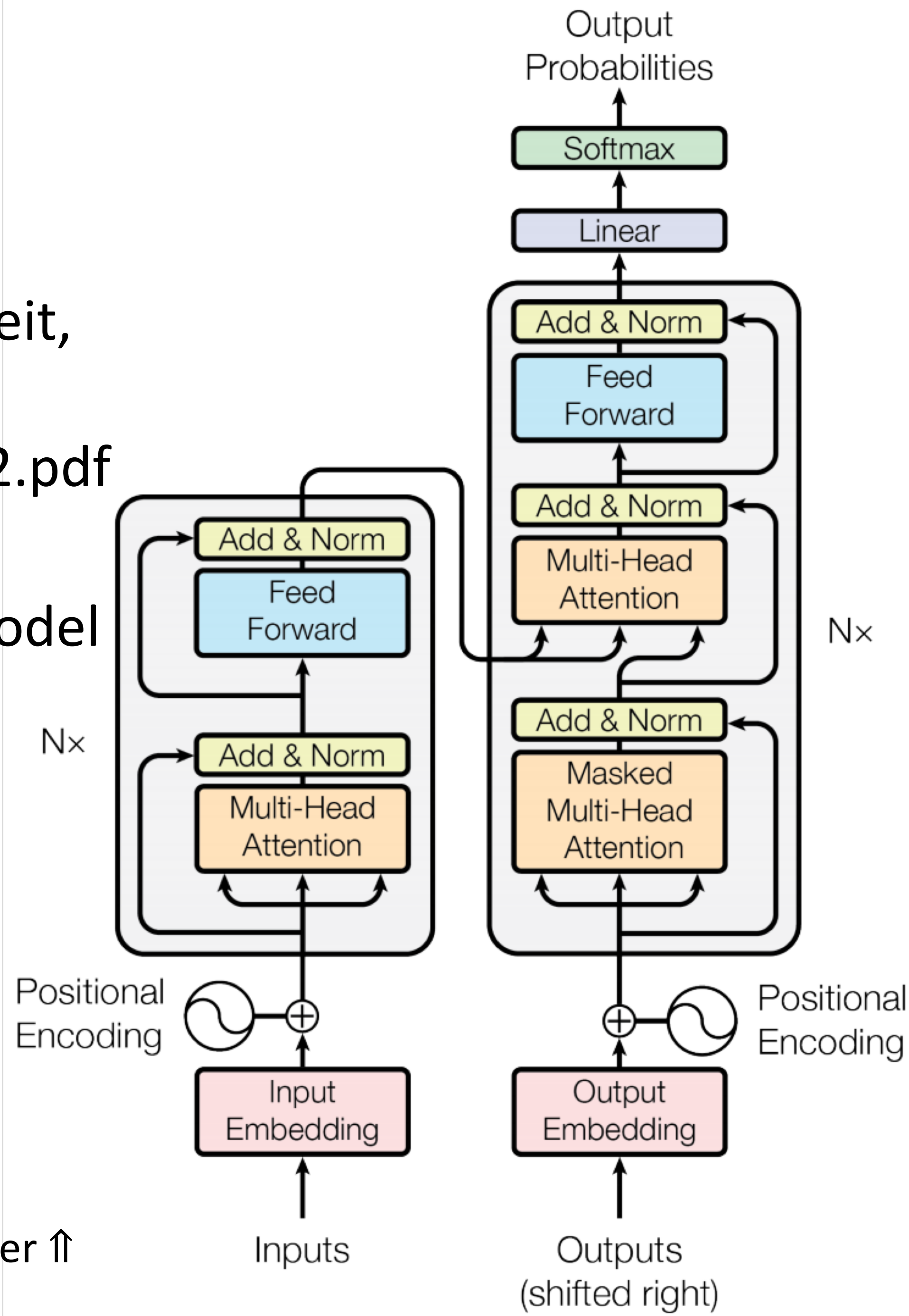
- Despite GRUs and LSTMs, RNNs still need attention mechanism to deal with long range dependencies – **path length** between states grows with sequence otherwise
- But if **attention** gives us access to any state… maybe we can just use attention and don't need the RNN?



Transformer Overview

Attention is all you need. 2017.
Aswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin
https://arxiv.org/pdf/1706.03762.pdf

- Non-recurrent sequence-to-sequence encoder-decoder model
- Task: machine translation with parallel corpus
- Predict each translated word
- Final cost/error function is standard cross-entropy error on top of a softmax classifier



This and related figures from paper ⇑

Transformer Basics

- Learning about transformers on your own?
  - Key recommended resource:
    - http://nlp.seas.harvard.edu/2018/04/03/attention.html
    - The Annotated Transformer by Sasha Rush
  - An Jupyter Notebook using PyTorch that explains everything!

- For now: Let's define the basic building blocks
  of transformer networks: first, new attention layers!

Dot-Product Attention (Extending our previous def.)

- Inputs: a query q and a set of key-value (k-v) pairs to an output
- Query, keys, values, and output are all vectors

- Output is weighted sum of values, where
- Weight of each value is computed by an inner product of query and corresponding key
- Queries and keys have same dimensionality d_k value have d_v

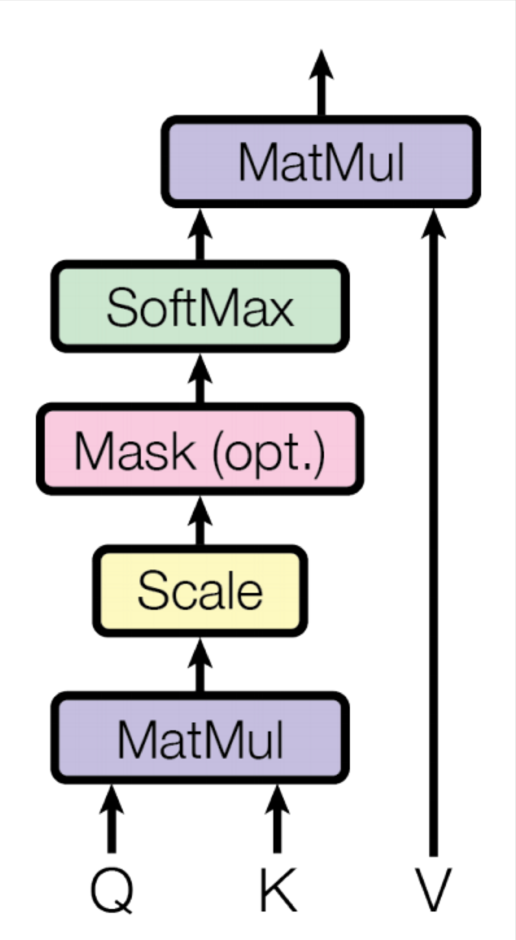
$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

Scaled Dot-Product Attention

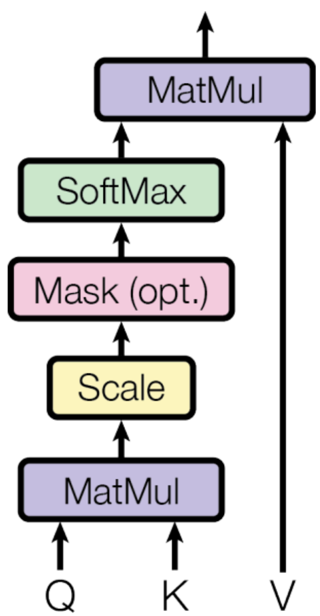
- Problem: As d_k gets large, the variance of q^Tk increases → some values inside the softmax get large → the softmax gets very peaked → hence its gradient gets smaller.

- Solution: Scale by length of query/key vectors:

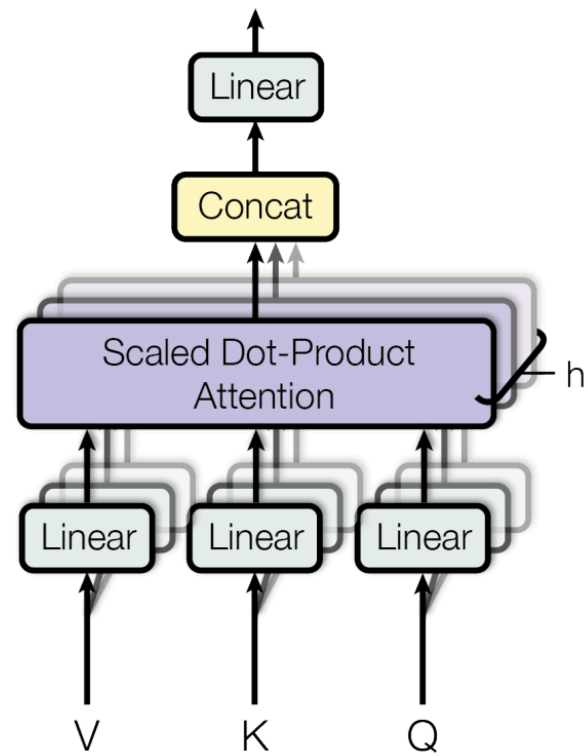
$$A(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



Scaled Dot-Product Attention



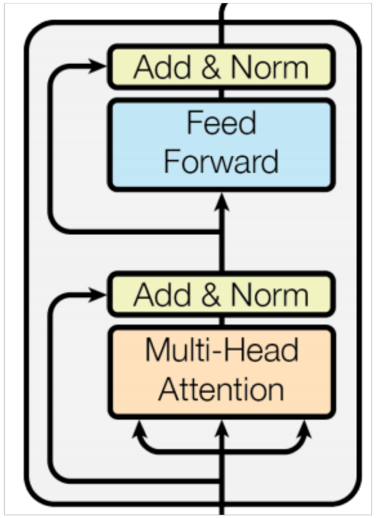
Multi-Head Attention



Self-attention in the encoder

- The input word vectors are the queries, keys and values
- In other words: the word vectors themselves select each other
- Word vector stack = Q = K = V
- We'll see in the decoder why we separate them in the definition

Complete transformer block



- Each block has two "sublayers"

1. Multihead attention
2. 2-layer feed-forward NNet (with ReLU)

Each of these two steps also has:

Residual (short-circuit) connection and LayerNorm

LayerNorm(x + Sublayer(x))

Layernorm changes input to have mean 0 and variance 1,
per layer and per training point (and adds two more parameters)

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

$$h_i = f\left(\frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i\right)$$

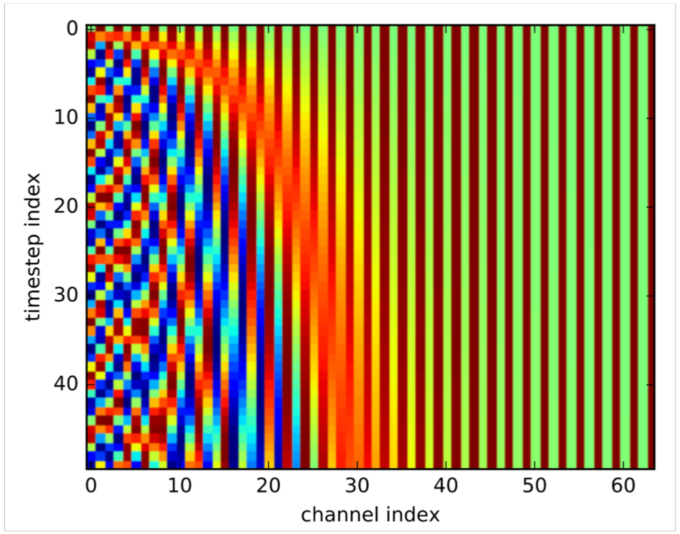
Layer Normalization by Ba, Kiros and Hinton, https://arxiv.org/pdf/1607.06450.pdf

Encoder Input

- Actual word representations are byte-pair encodings
  - As in last lecture

- Also added is a **positional encoding** so same words at different locations have different overall representations:

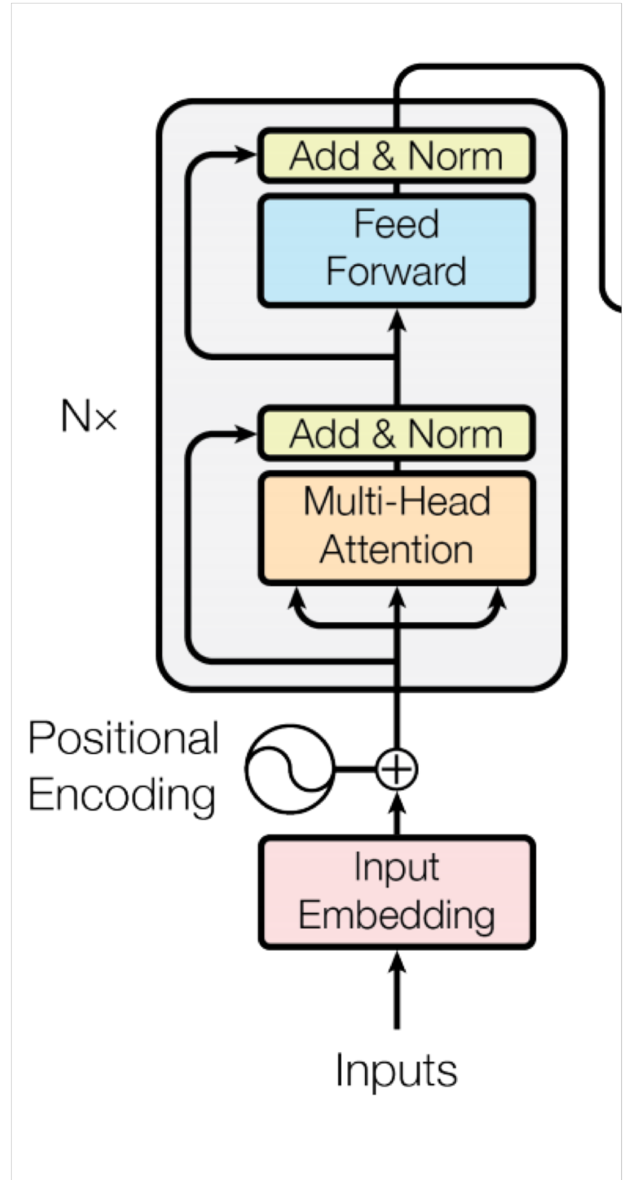
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$



Complete Encoder

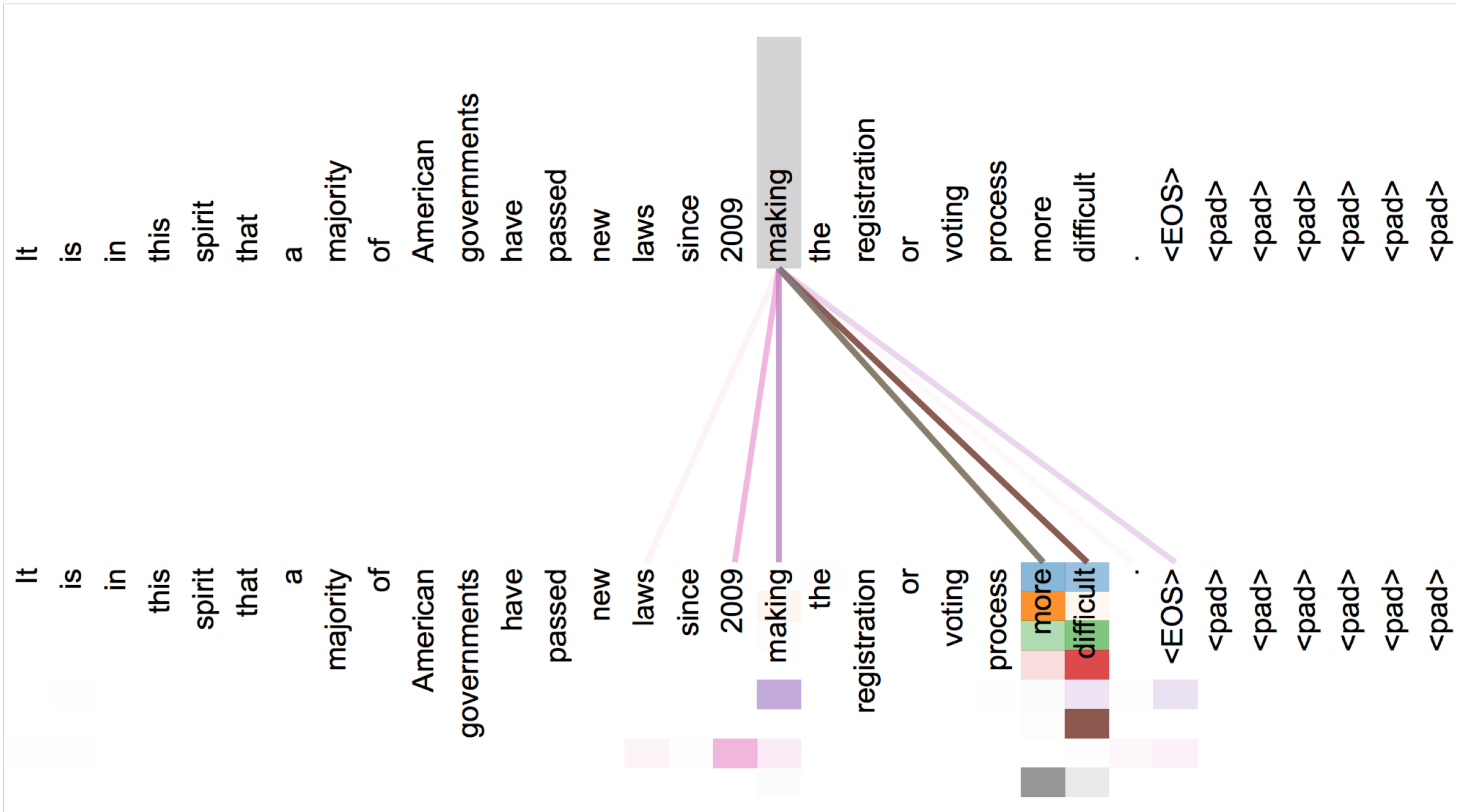
- For encoder, at each block, we use the same Q, K and V from the previous layer

- Blocks are repeated 6 times
  - (in vertical stack)

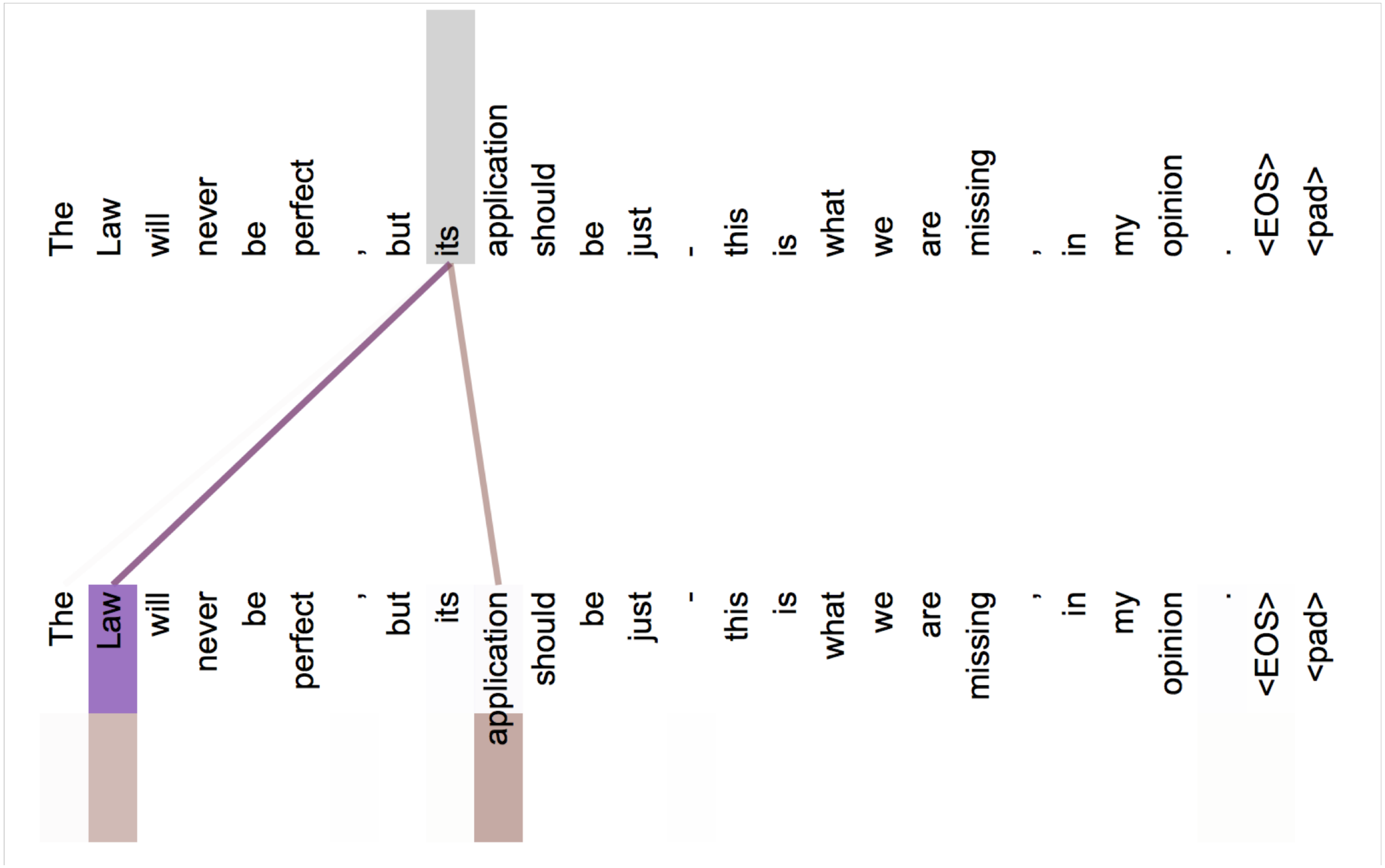


Attention visualization in layer 5

- Words start to pay attention to other words in sensible ways



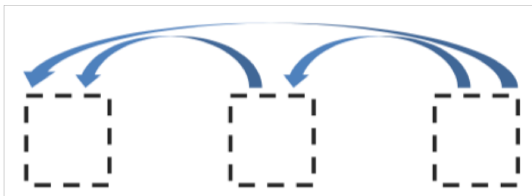
Attention visualization: Implicit anaphora resolution



In 5th layer. Isolated attentions from just the word 'its' for attention heads 5 and 6. Note that the attentions are very sharp for this word.

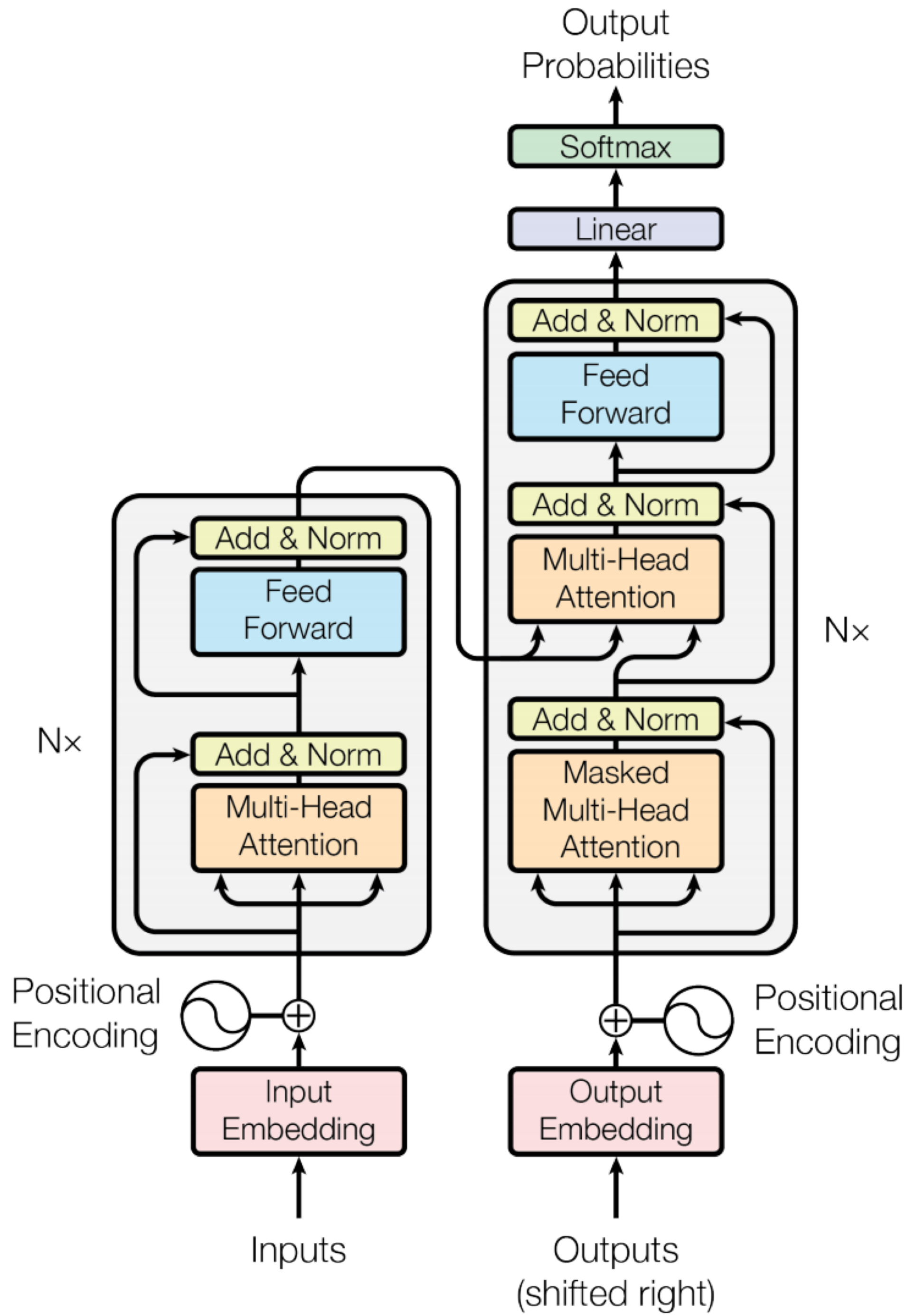
Transformer Decoder

- 2 sublayer changes in decoder
- Masked decoder self-attention on previously generated outputs:



- Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder

- Blocks repeated 6 times also



Tips and tricks of the Transformer

Details (in paper and/or later lectures):

- Byte-pair encodings
- Checkpoint averaging
- ADAM optimizer with learning rate changes
- Dropout during training at every layer just before adding residual
- Label smoothing
- Auto-regressive decoding with beam search and length penalties

- → Use of transformers is spreading but they are hard to optimize and unlike LSTMs don't usually just work out of the box and they don't play well yet with other building blocks on tasks.

LEARNING: BACKPROPAGATION

[Inside-outside and forward-backward algorithms are just backprop]().
Jason Eisner (2016).
In *EMNLP Workshop on Structured Prediction for NLP*.

Inside-Outside & Forward-Backward Algorithms are just Backprop

(tutorial paper)

Jason Eisner

Johns Hopkins University

Center for Language and Speech Processing

"The inside-outside algorithm is the hardest algorithm I know."

– a senior NLP researcher, in the 1990's

↑ ↑

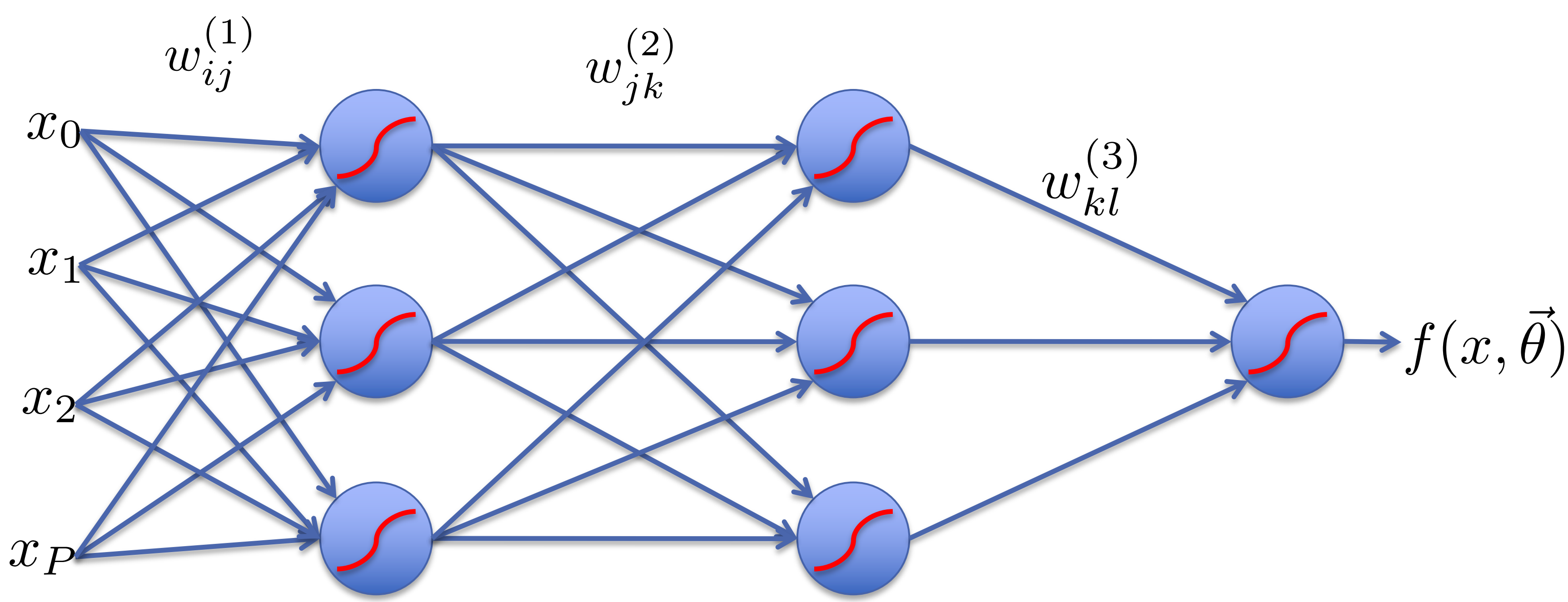
β α

Next 10 slides on back propagation are adapted from Andrew Rosenberg

Error Backpropagation

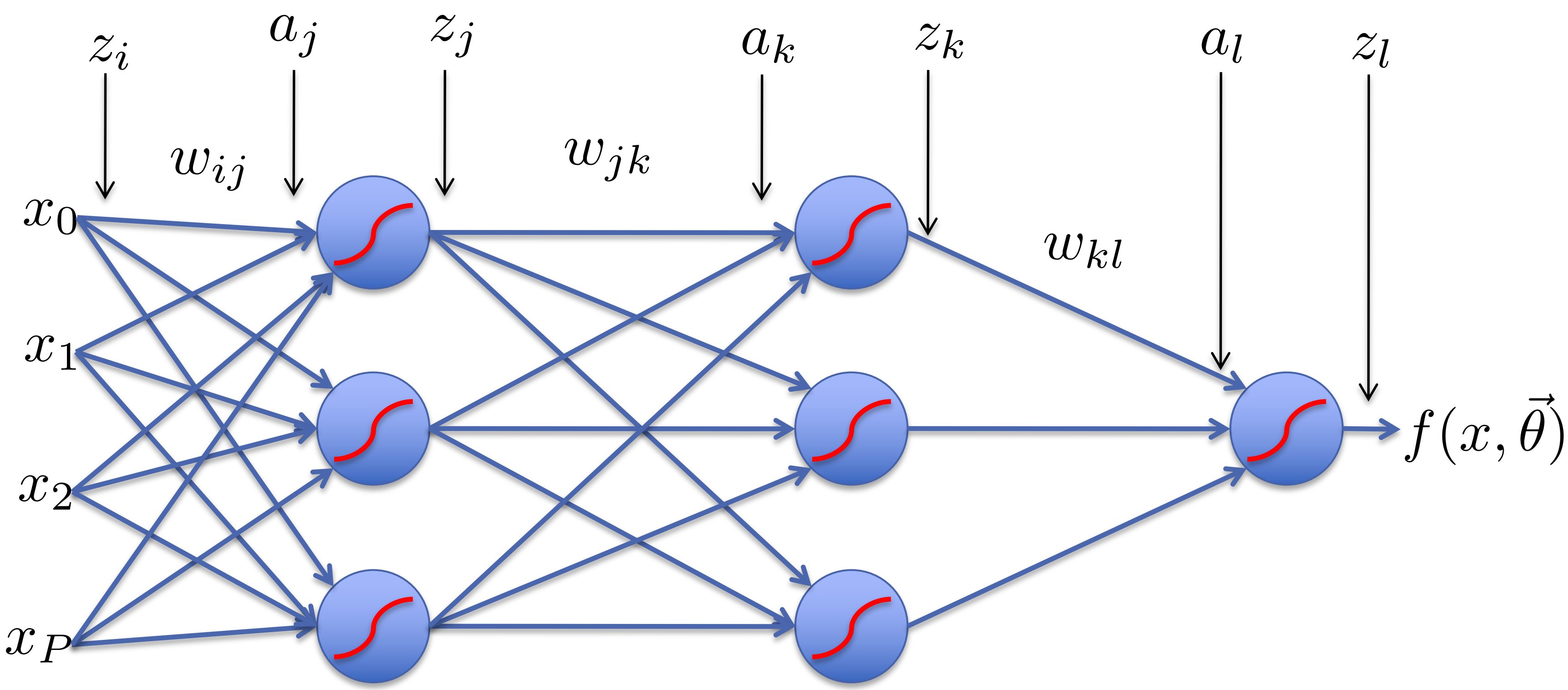
- Model parameters: $\vec{\theta} = \{w_{ij}^{(1)}, w_{jk}^{(2)}, w_{kl}^{(3)}\}$

  for brevity: $\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$



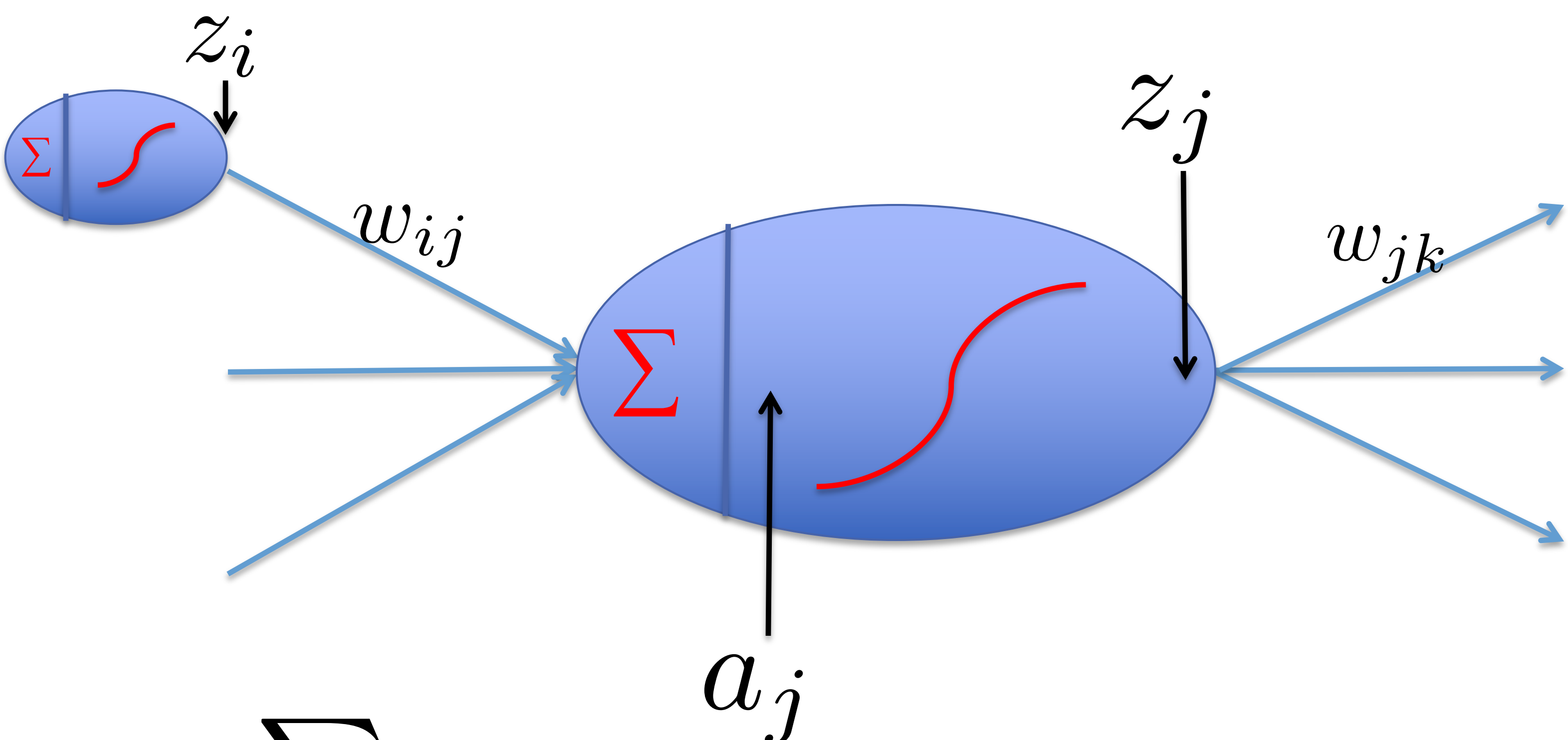
Error Backpropagation

- Model parameters: $\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$
- *Let a and z be the input and output of each node*



Error Backpropagation

$$z_j = g(a_j)$$

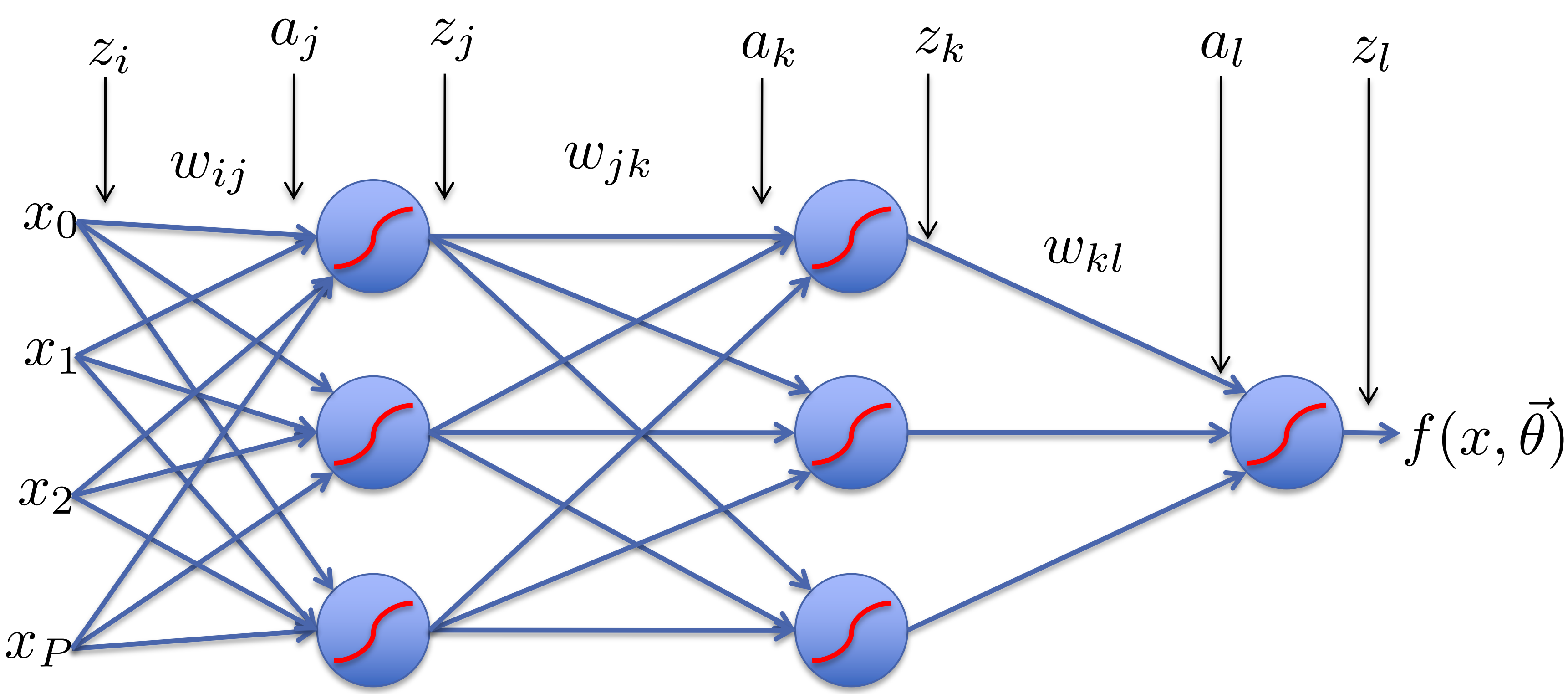


$$a_j = \sum_i w_{ij} z_i$$

- *Let a* and *z be the input and output of each node*

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \boxed{} \qquad a_l = \boxed{}$$

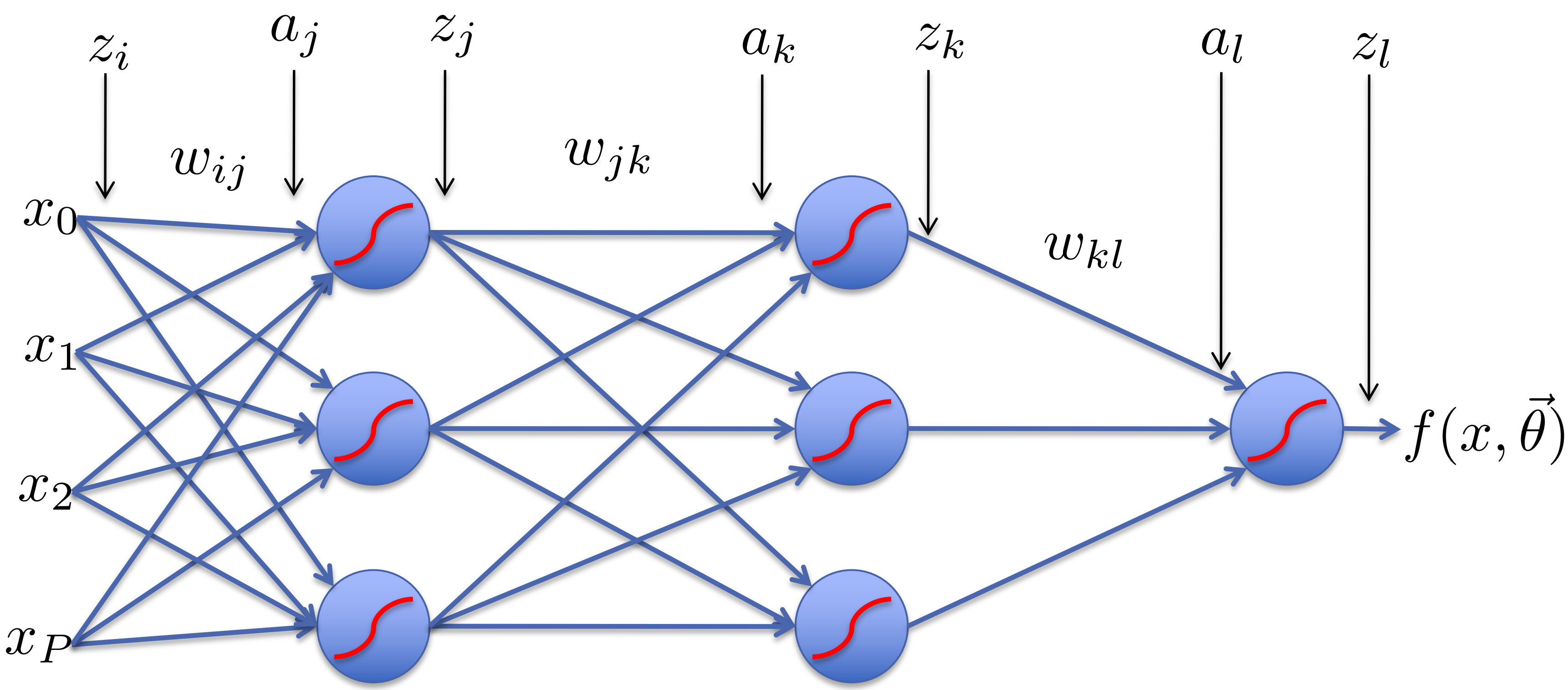
$$z_j = g(a_j) \qquad z_k = \boxed{} \qquad z_l = \boxed{}$$



- *Let a and z be the input and output of each node*

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



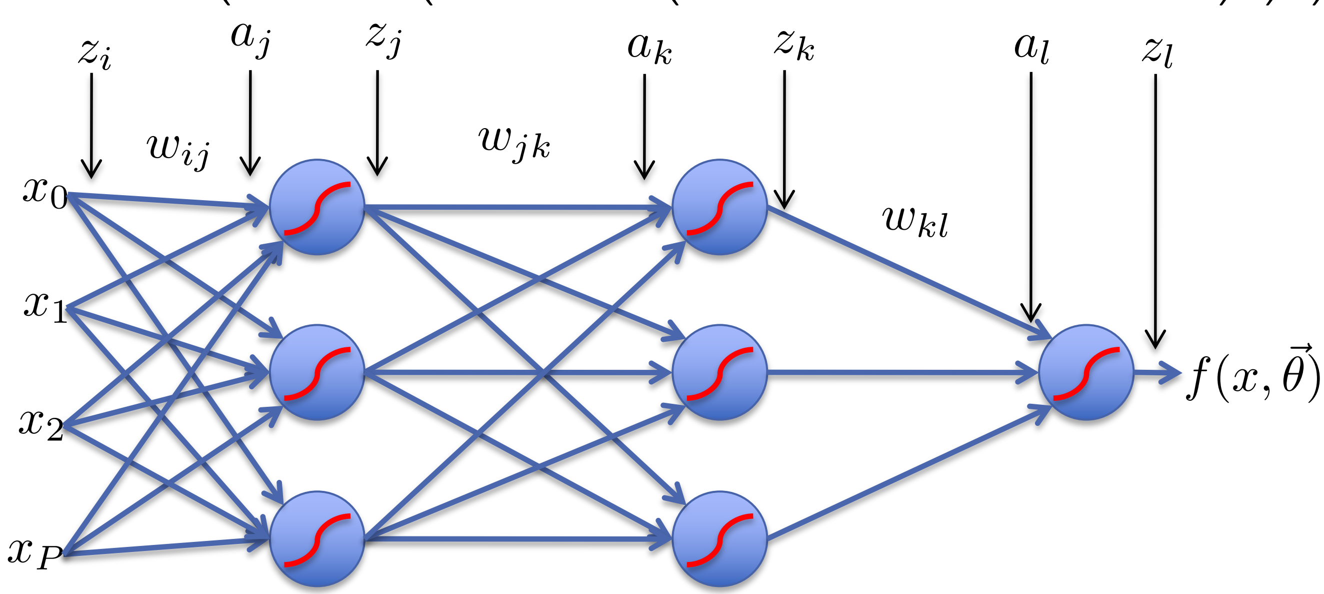
Training: minimize loss

$$R(\theta) = \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n))$$

Empirical Risk Function

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\qquad g\left(\qquad g\left(\qquad x_{n,i}\right)\right)\right)\right)^2$$



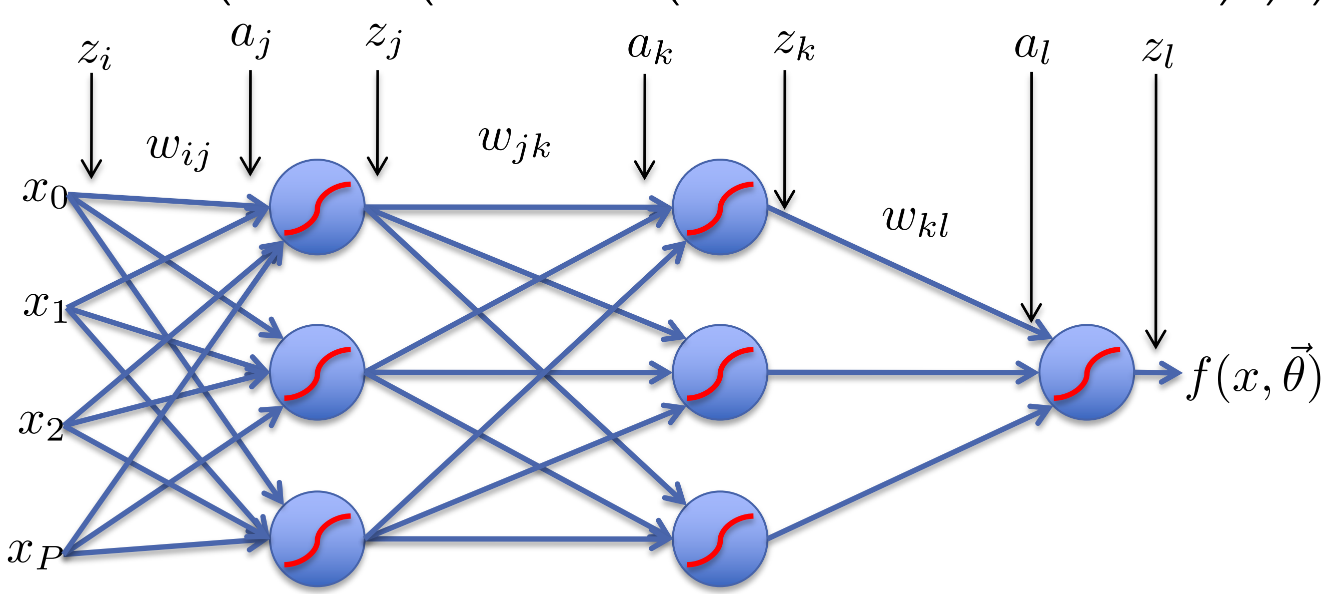
Training: minimize loss

$$R(\theta) = \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n))$$

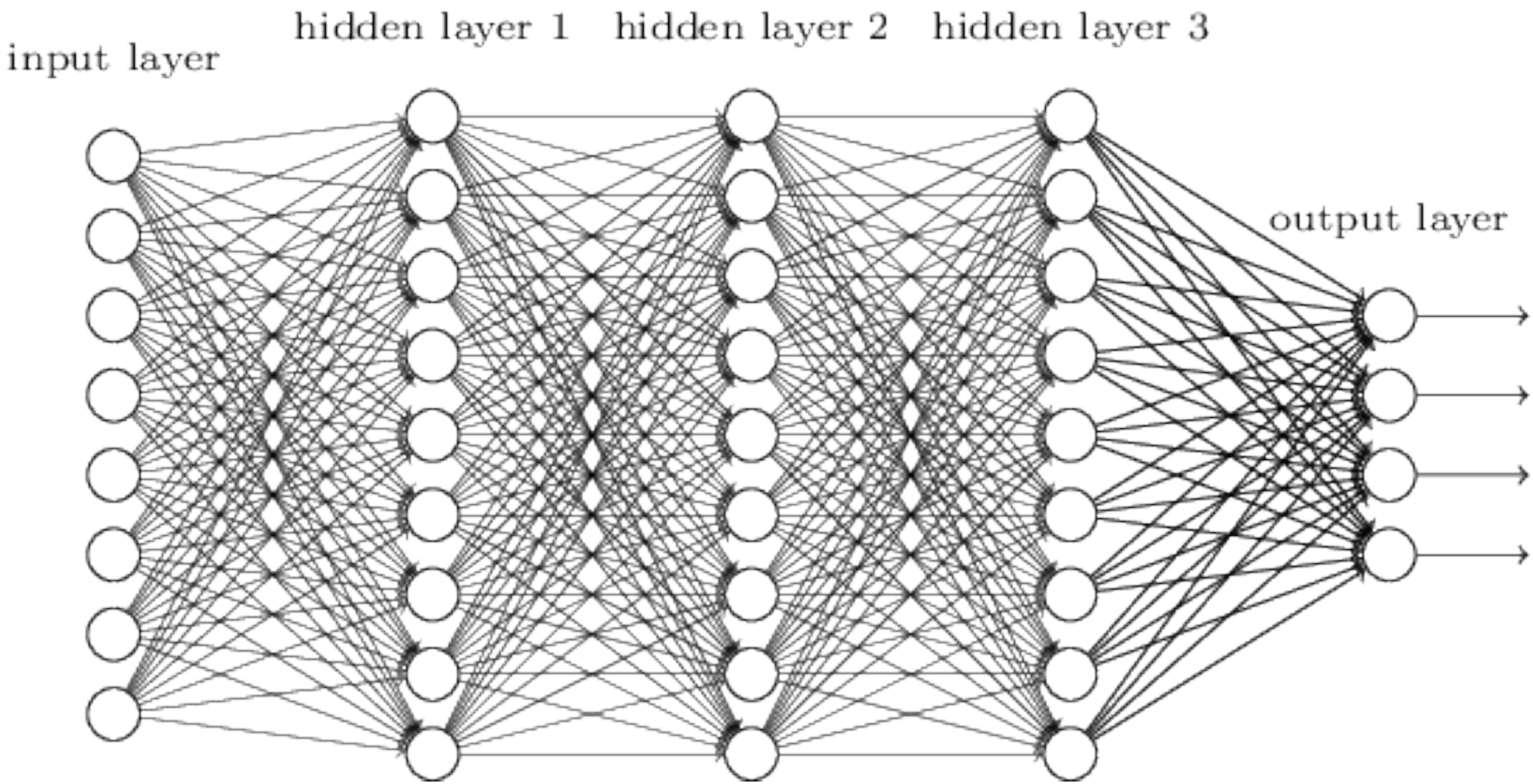
Empirical Risk Function

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\sum_k w_{kl}\, g\left(\sum_j w_{jk}\, g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2$$



Taking Partial Derivatives…



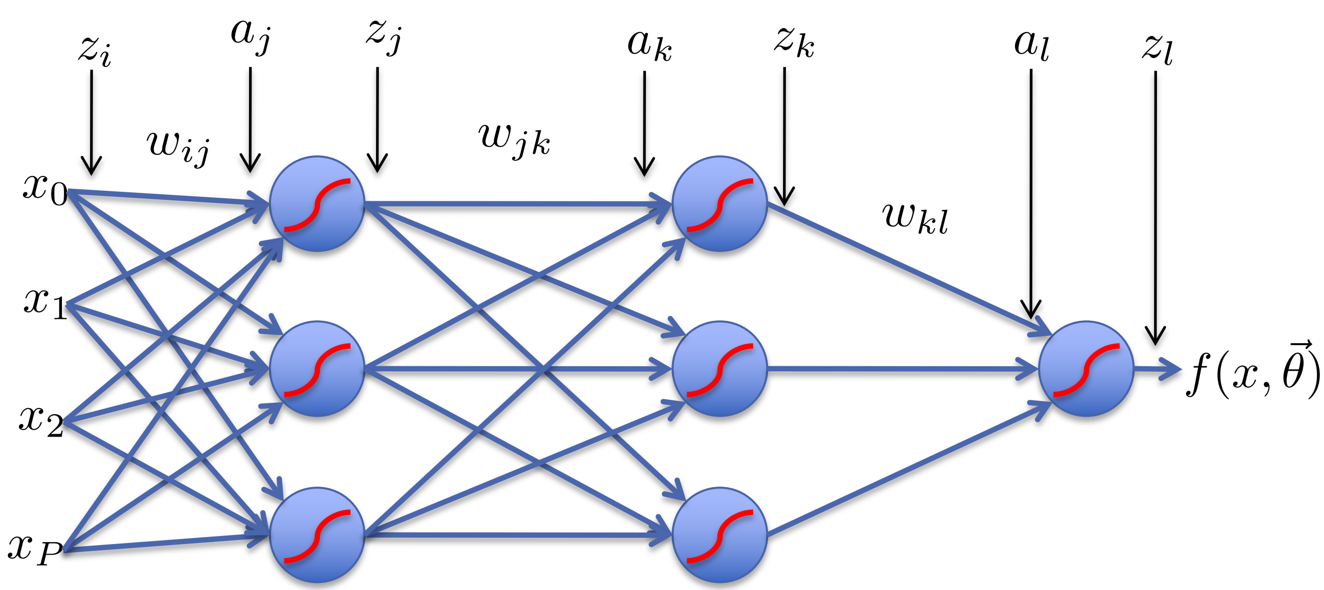
Error Backpropagation

Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule



Error Backpropagation

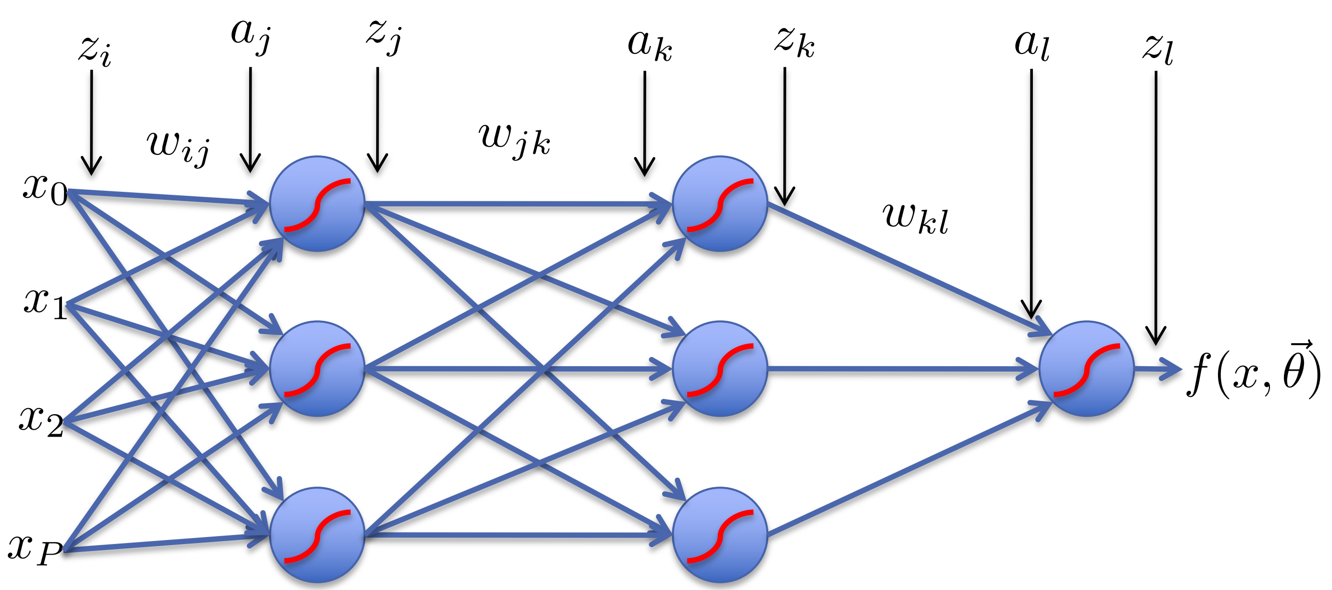
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$



Error Backpropagation

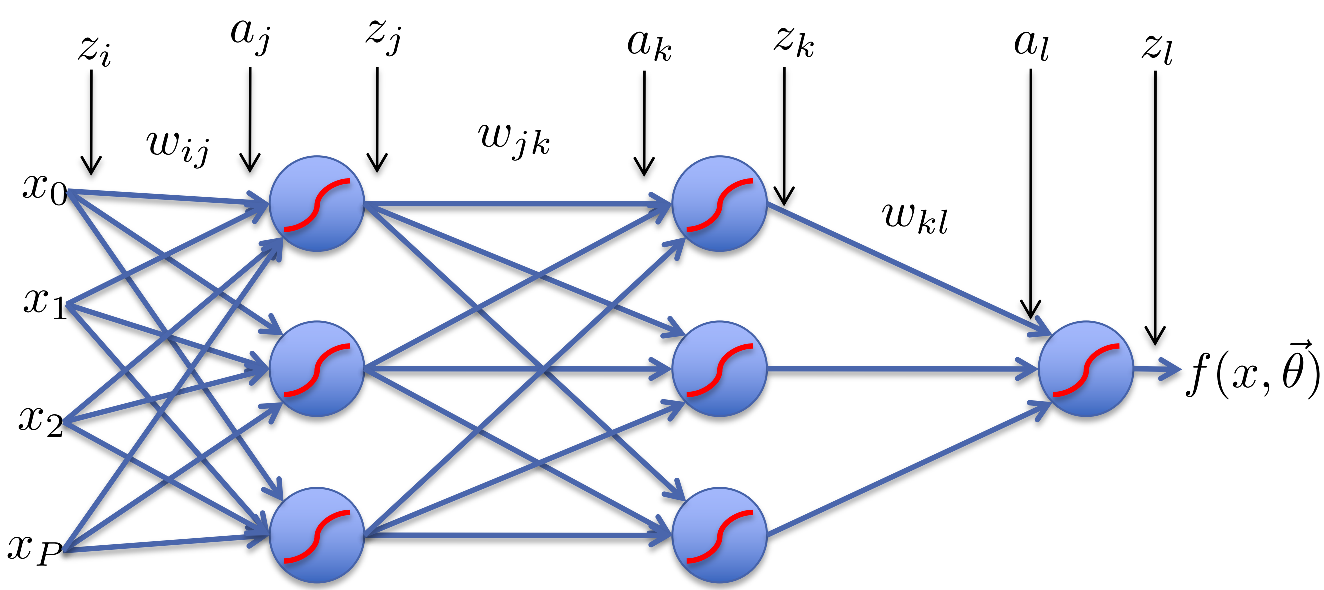
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right]$$



Error Backpropagation

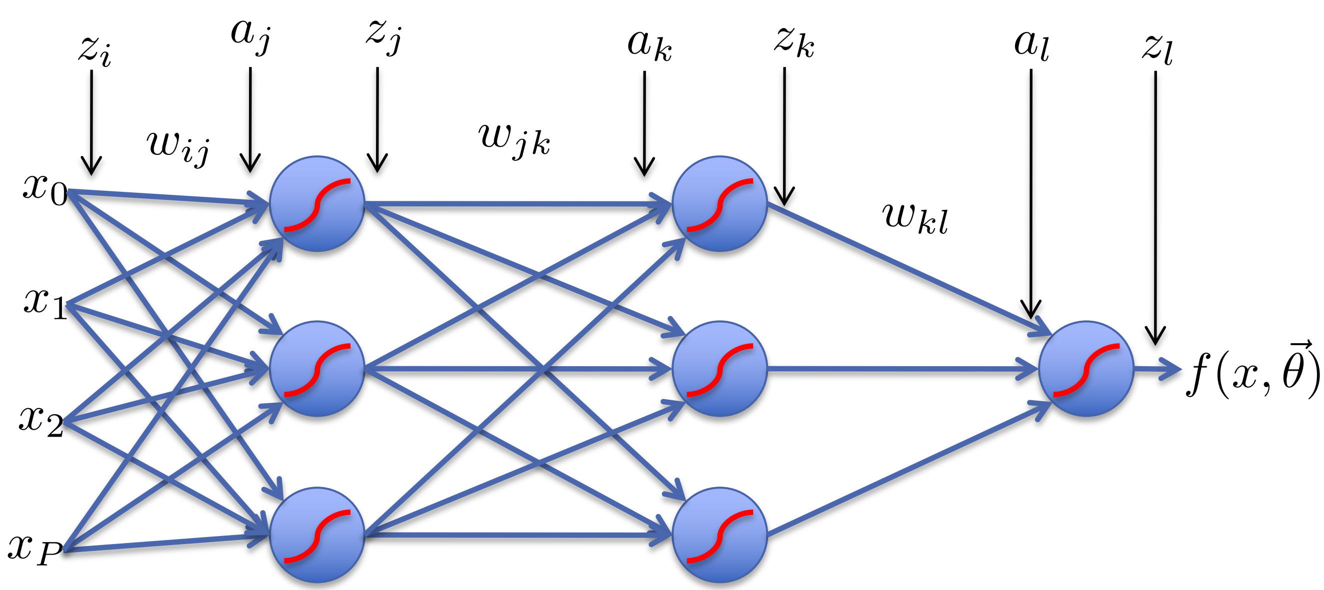
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$



Error Backpropagation

Optimize last layer weights w_{kl}

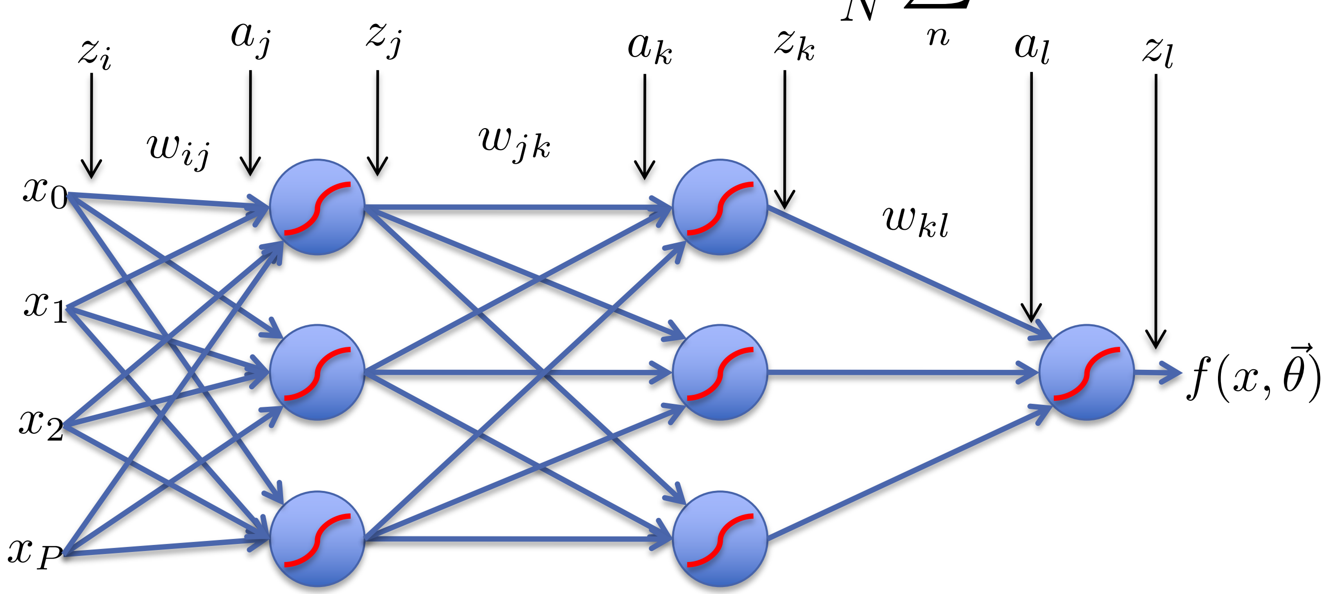
$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$

$$= \frac{1}{N}\sum_n \delta_{l,n} \quad z_{k,n}$$



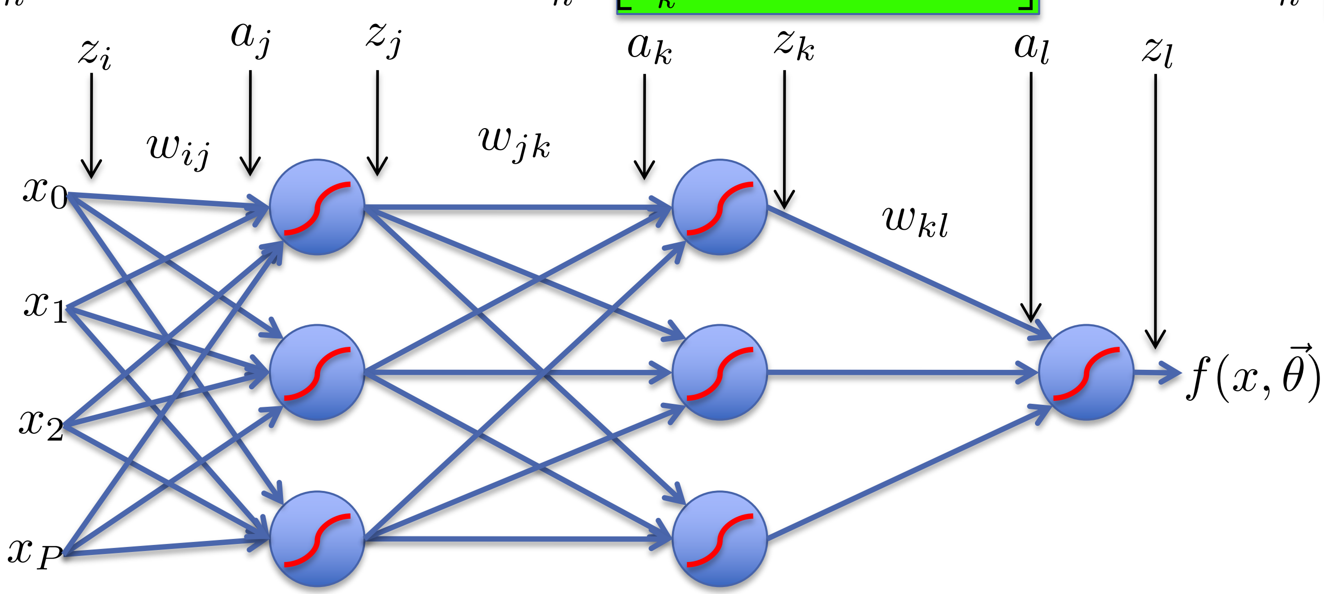
Error Backpropagation

Repeat for all previous layers

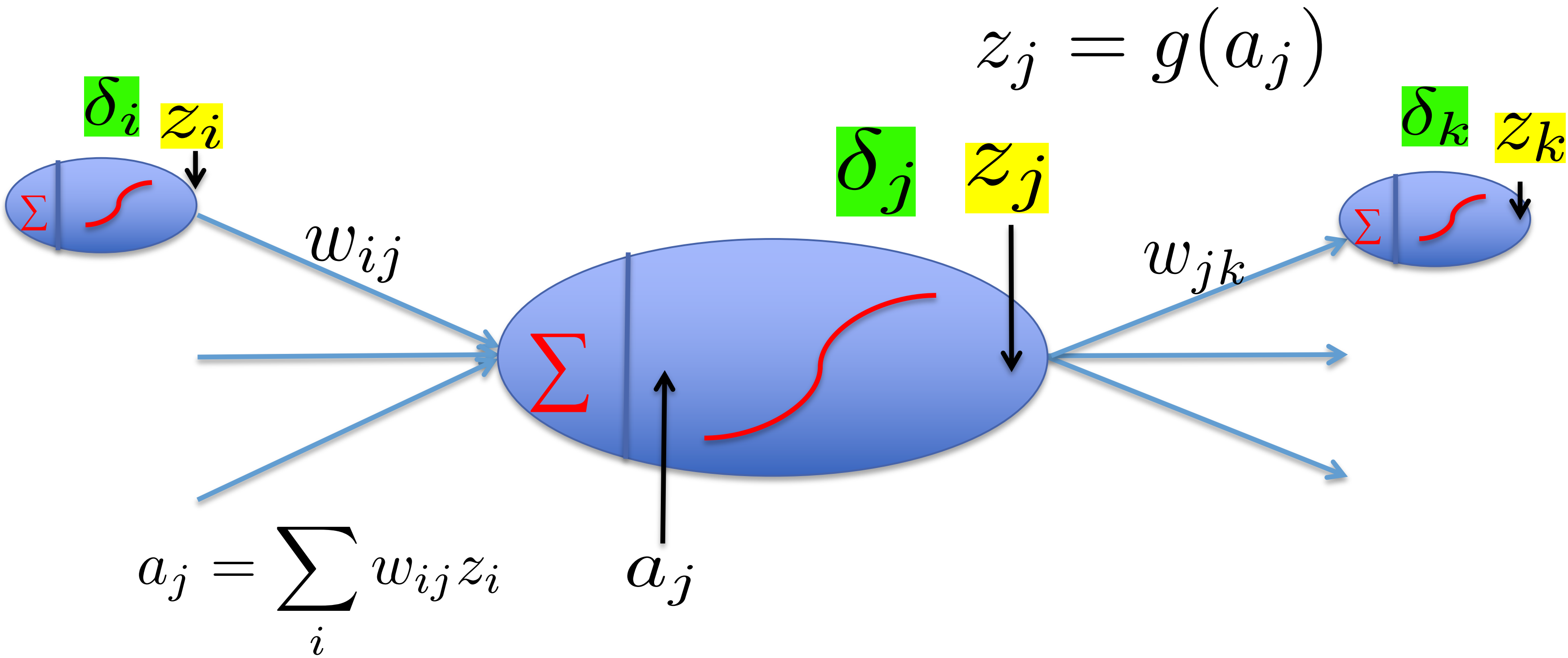
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right] = \frac{1}{N}\sum_n \left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right] z_{i,n} = \frac{1}{N}\sum_n \delta_{j,n} z_{i,n}$$



Backprop Recursion



$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

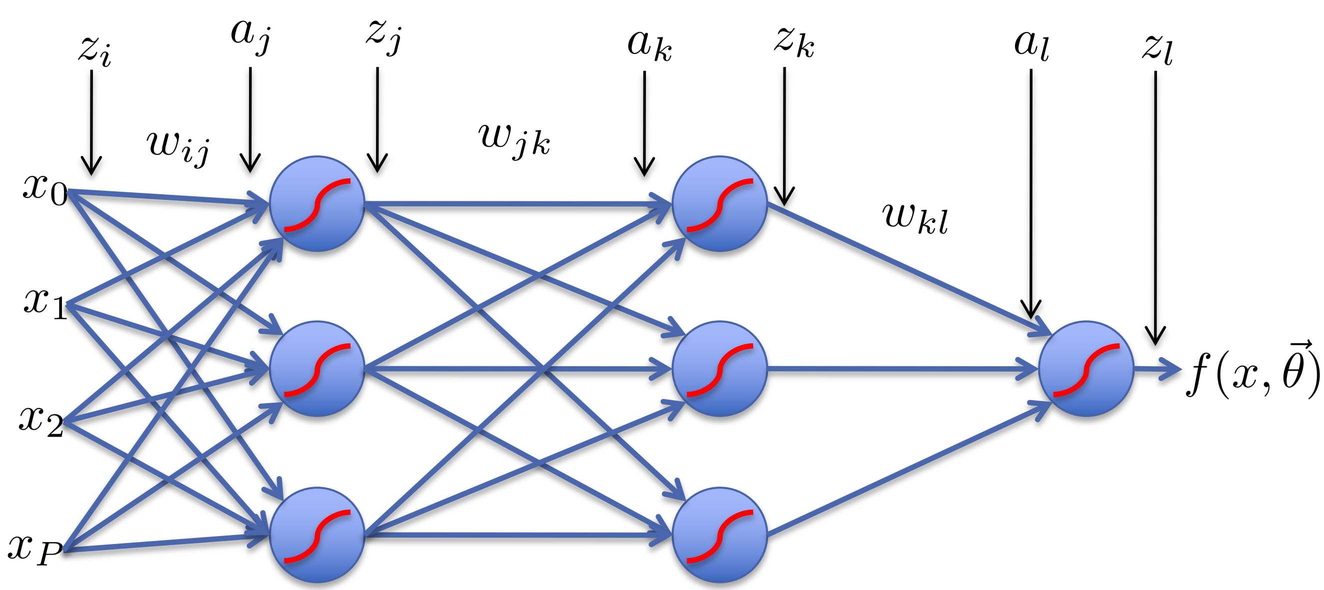
$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right] = \frac{1}{N}\sum_n \left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right] z_{i,n} = \frac{1}{N}\sum_n \delta_{j,n} z_{i,n}$$

Learning: Gradient Descent

$$w_{ij}^{t+1} = w_{ij}^{t} - \eta \frac{\partial R}{w_{ij}}$$

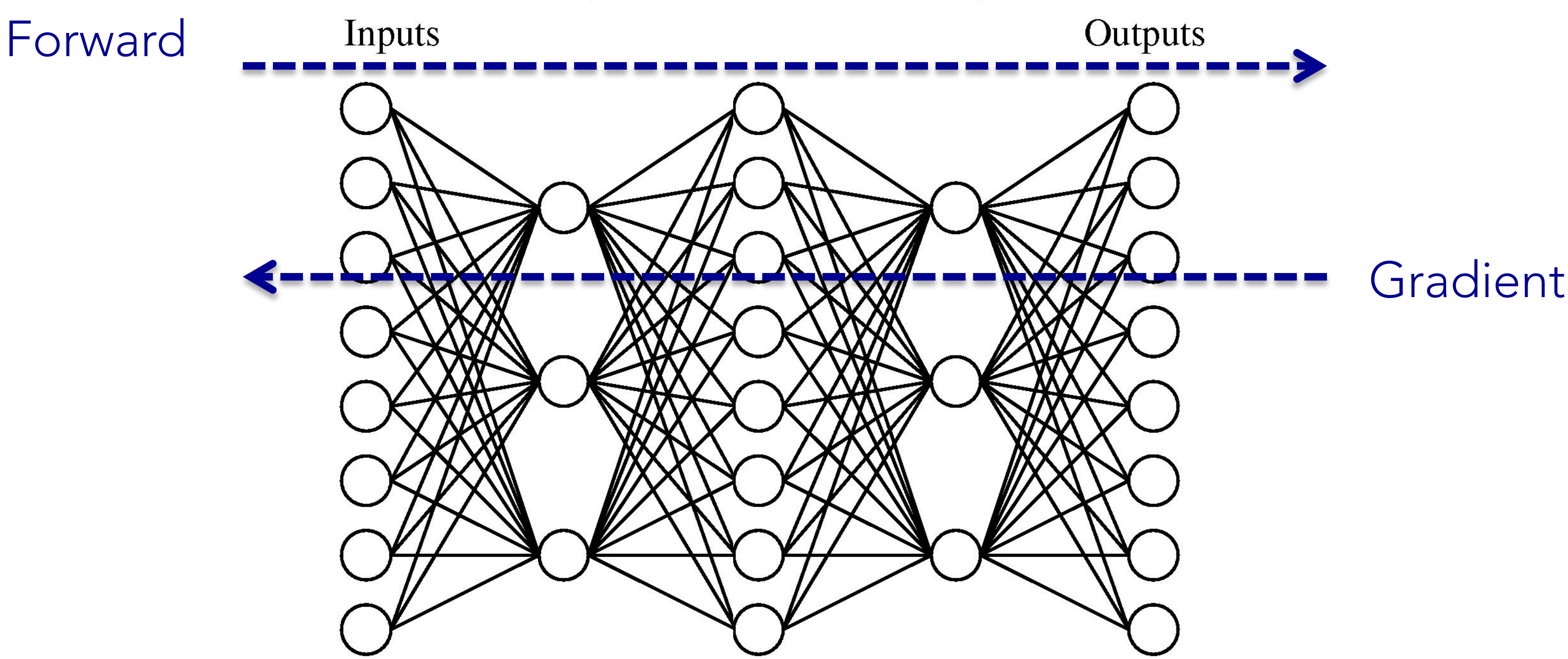
$$w_{jk}^{t+1} = w_{jk}^{t} - \eta \frac{\partial R}{w_{kl}}$$

$$w_{kl}^{t+1} = w_{kl}^{t} - \eta \frac{\partial R}{w_{kl}}$$



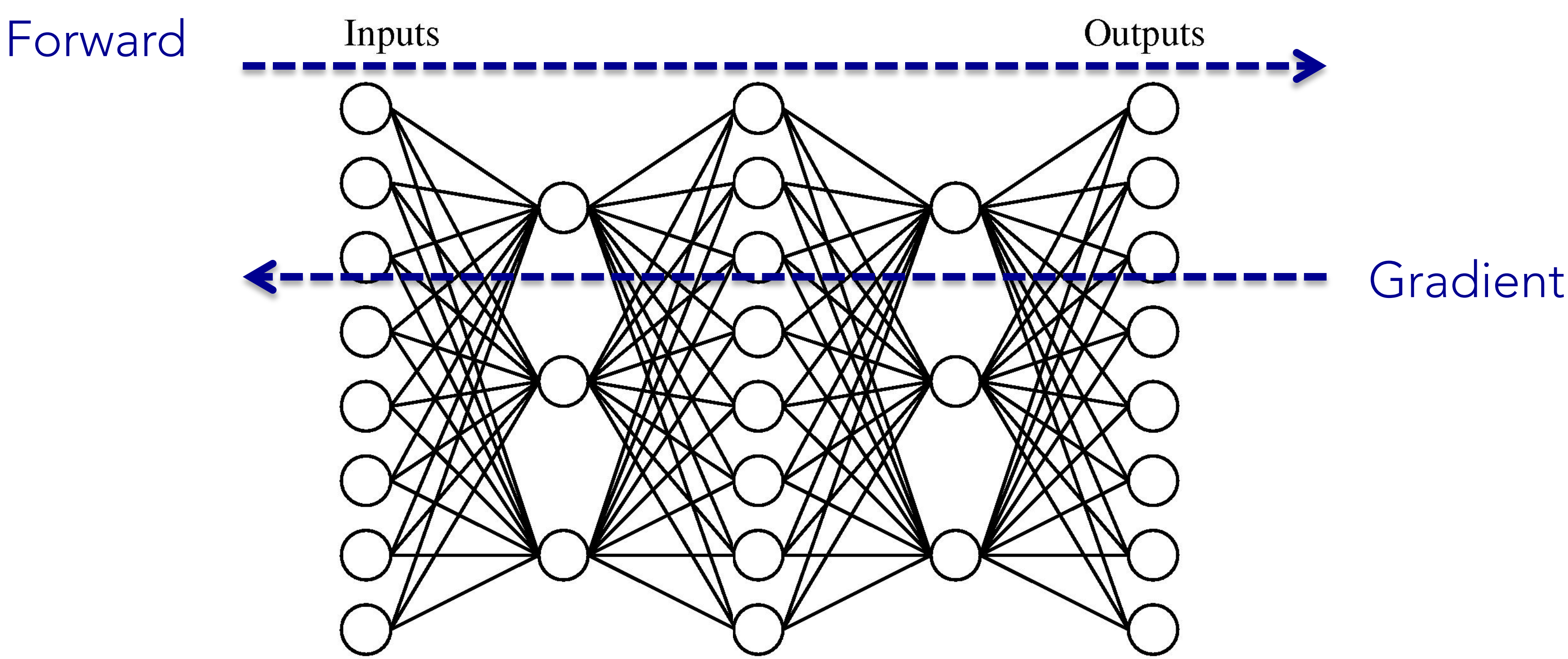
Backpropagation

- Starts with a forward sweep to compute all the intermediate function values z_i
- Through backprop, computes the partial derivatives recursively δ_j $\frac{\partial R}{\partial w_{ij}}$
- A form of dynamic programming
  - Instead of considering exponentially many paths between a weight w_ij and the final loss (risk), store and reuse intermediate results.
- A type of automatic differentiation. (there are other variants e.g., recursive differentiation only through forward propagation.



Backpropagation

	Primary Interface Language
TensorFlow (https://www.tensorflow.org/)	Python
Torch (http://torch.ch/)	Lua
Theano (http://deeplearning.net/software/theano/)	Python
CNTK (https://github.com/Microsoft/CNTK)	C++
cnn (https://github.com/clab/cnn)	C++
Caffe (http://caffe.berkeleyvision.org/)	C++



Cross Entropy Loss (aka log loss, logistic loss)

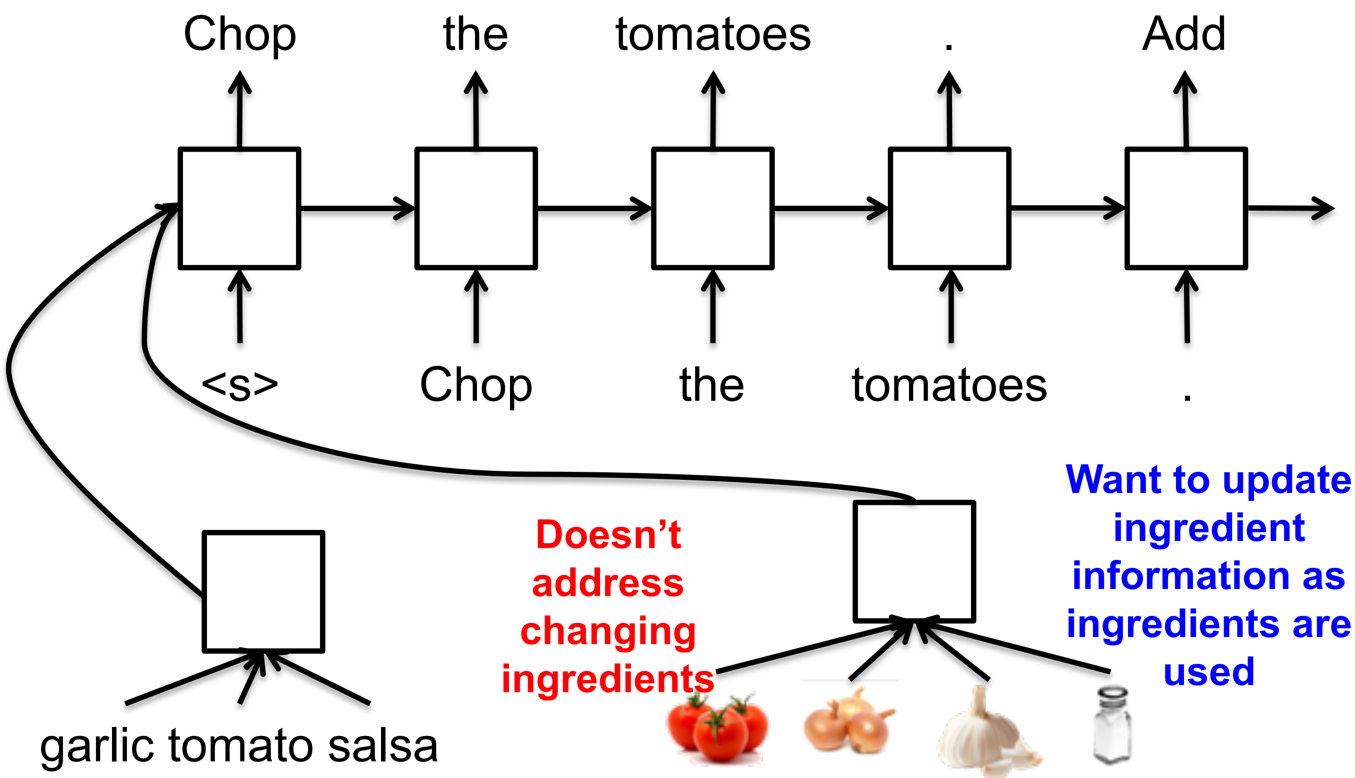
- Cross Entropy $$H(p,q) = -\sum_y p(y) \log q(y)$$
  - $q(y)$: Predicted prob
  - $p(y)$: True prob
- Related quantities
  $$H(p) = \sum_y p(y) \log p(y)$$
  - Entropy
  - KL divergence (the distance between two distributions p and q)
  $$D_{KL}(p||q) = \sum_y p(y) \log \frac{p(y)}{q(y)}$$
  $$H(p,q) = E_p[-\log q] = H(p) + D_{KL}(p||q)$$
- Use Cross Entropy for models that should have more probabilistic flavor (e.g., language models)
- Use Mean Squared Error loss for models that focus on correct/incorrect predictions
  $$\mathrm{MSE} = \frac{1}{2}(y - f(x))^2$$

NEURAL CHECK LIST

Neural Checklist Models
(Kiddon et al., 2016)

- What can we do with gating & attention?

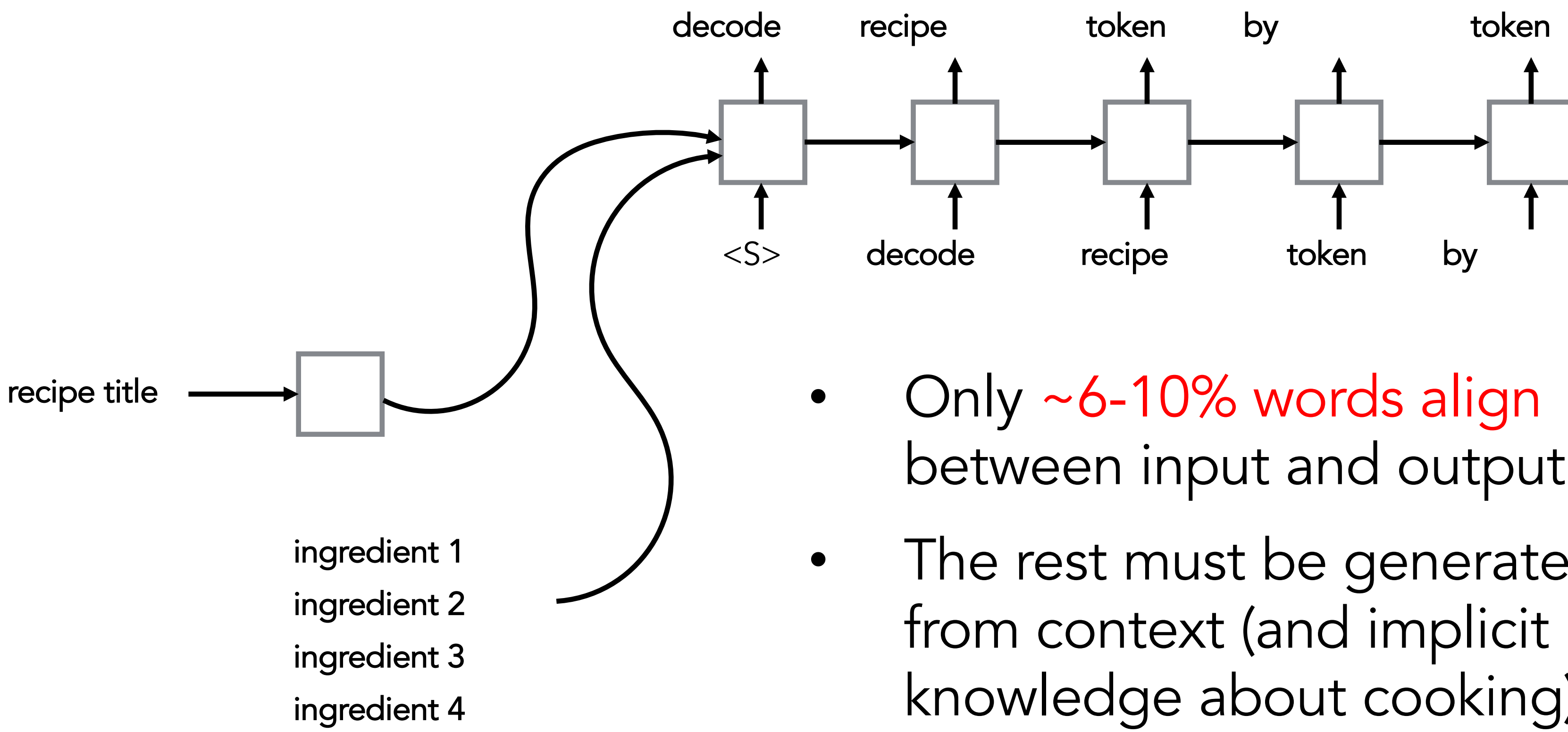
Encoder--Decoder Architecture



Encode title - decode recipe

sausage sandwiches → Cut each sandwich in halves.
Sandwiches with sandwiches.
Sandwiches, sandwiches, Sandwiches, sandwiches, sandwiches
sandwiches, sandwiches, sandwiches, sandwiches, sandwiches, sandwiches, or sandwiches or triangles, a griddle, each sandwich.
Top each with a slice of cheese, tomato, and cheese.
Top with remaining cheese mixture.
Top with remaining cheese.
Broil until tops are bubbly and cheese is melted, about 5 minutes.

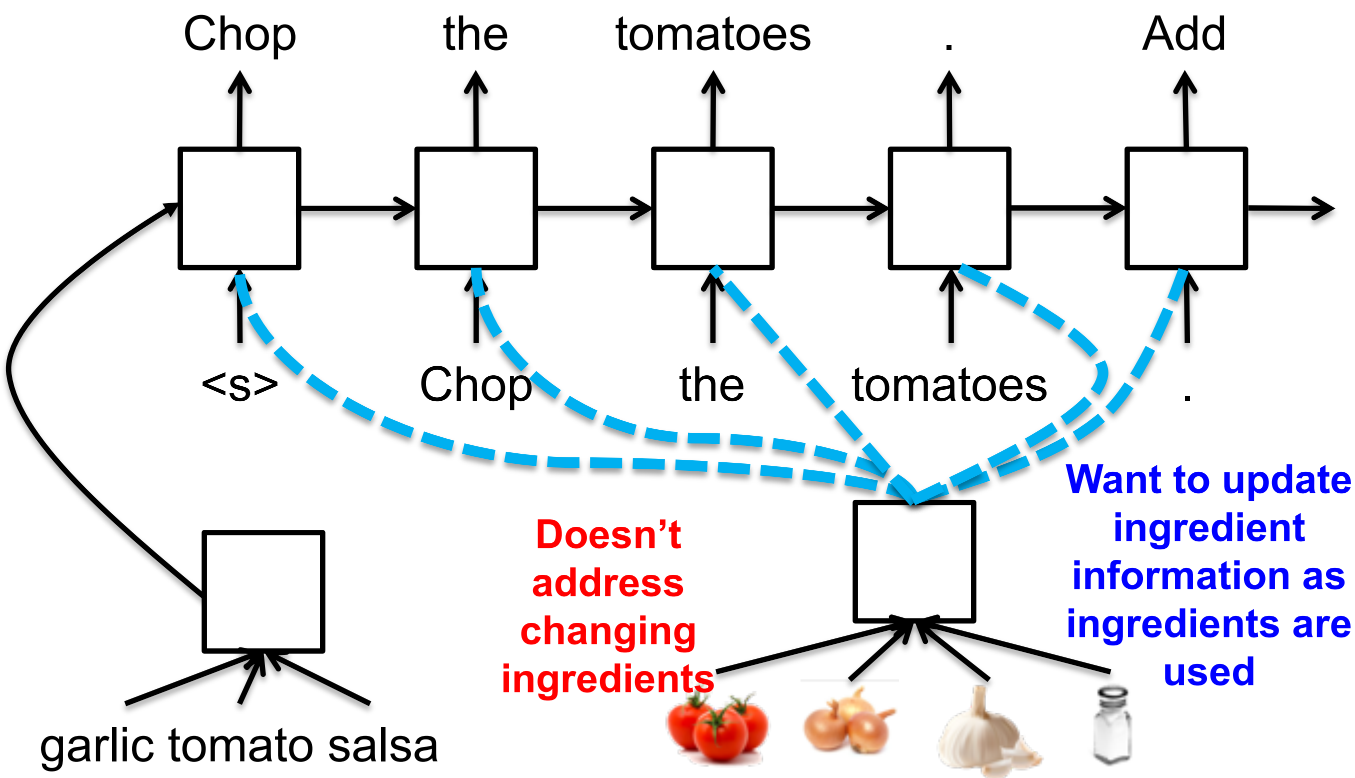
Recipe generation **vs** machine translation



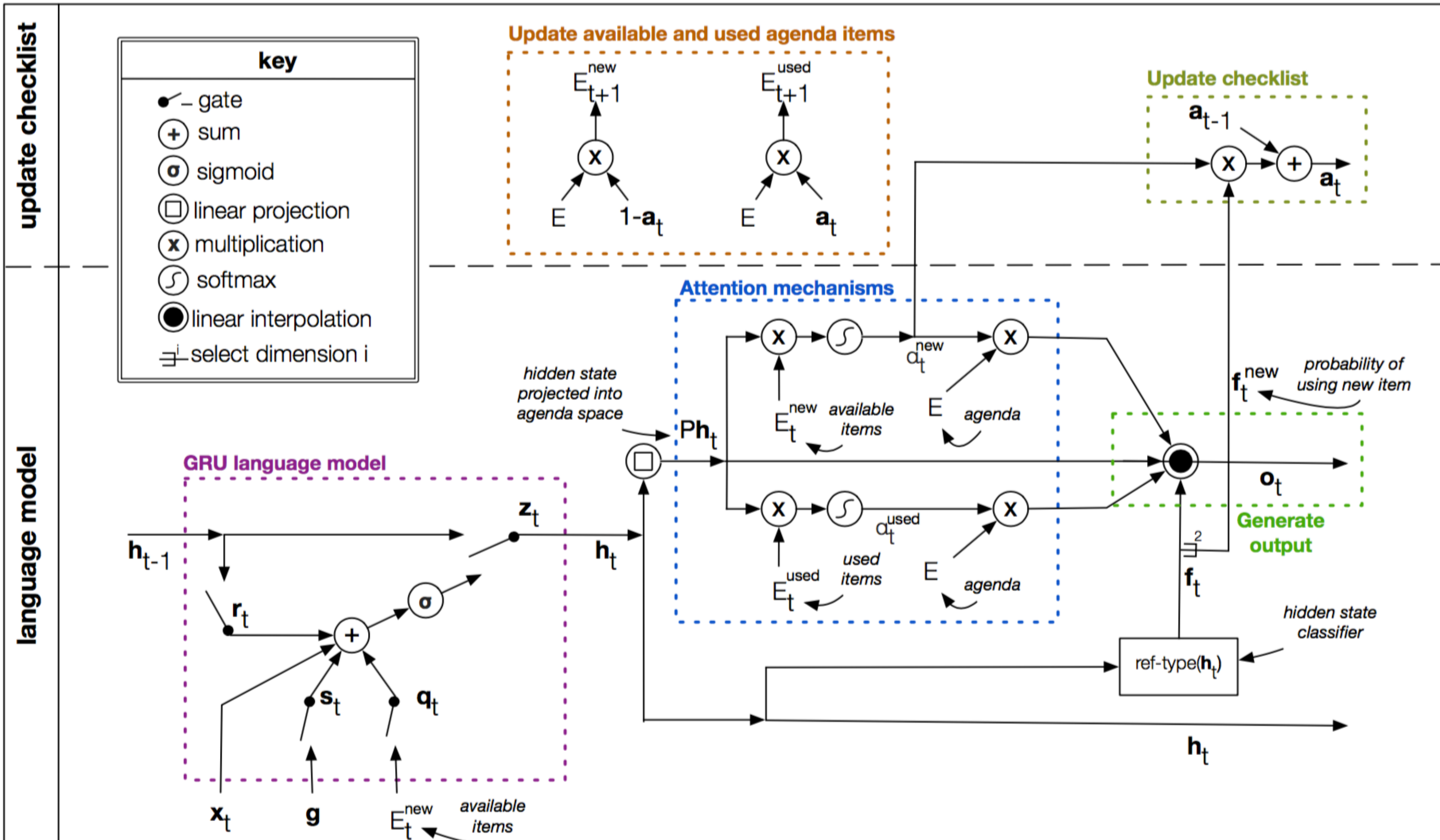
- Only ~6-10% words align between input and output.
- The rest must be generated from context (and implicit knowledge about cooking)
- Contextual switch between two different input sources

Two input sources

Encoder--Decoder with Attention



Neural checklist model

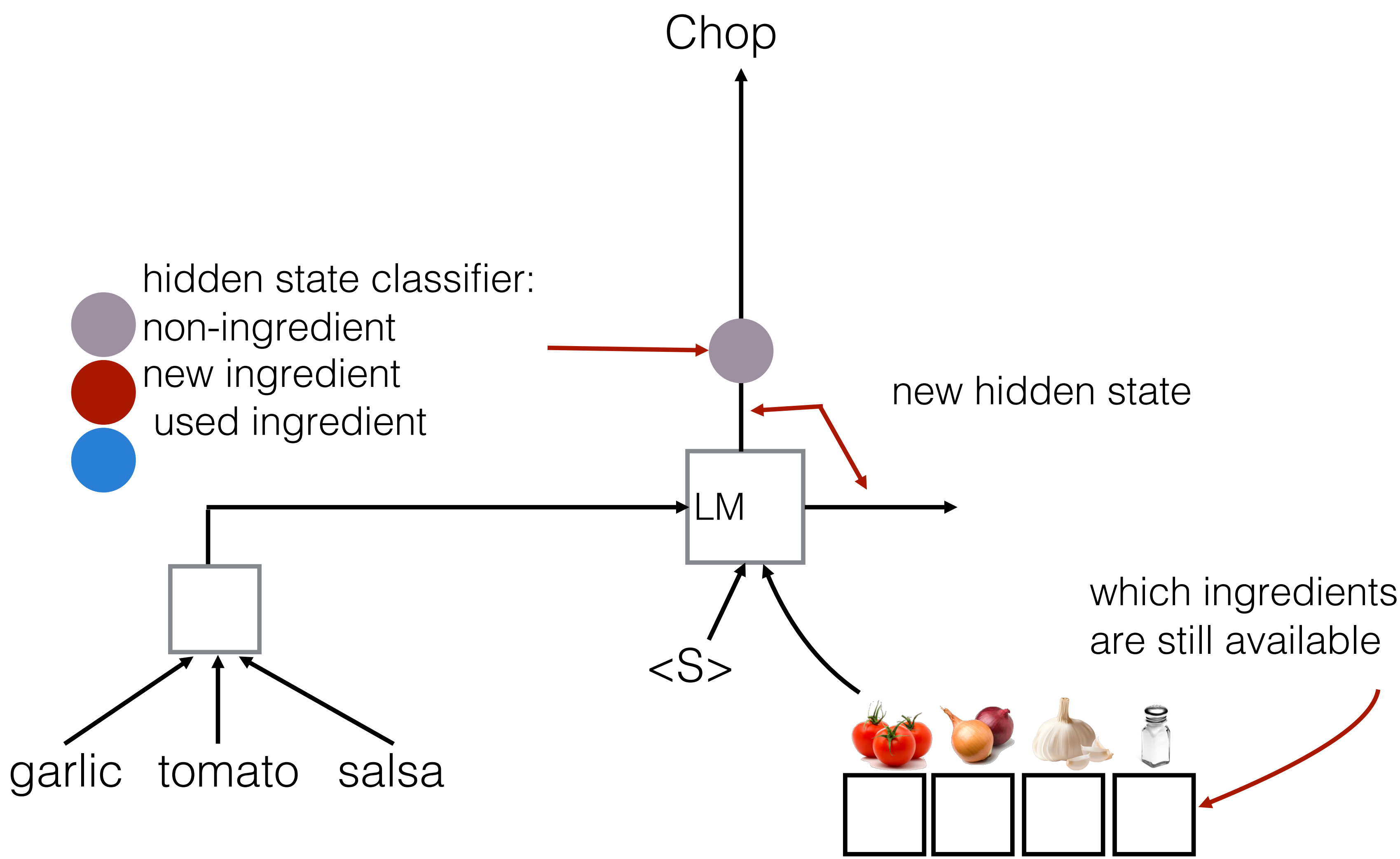


Let’s make salsa!

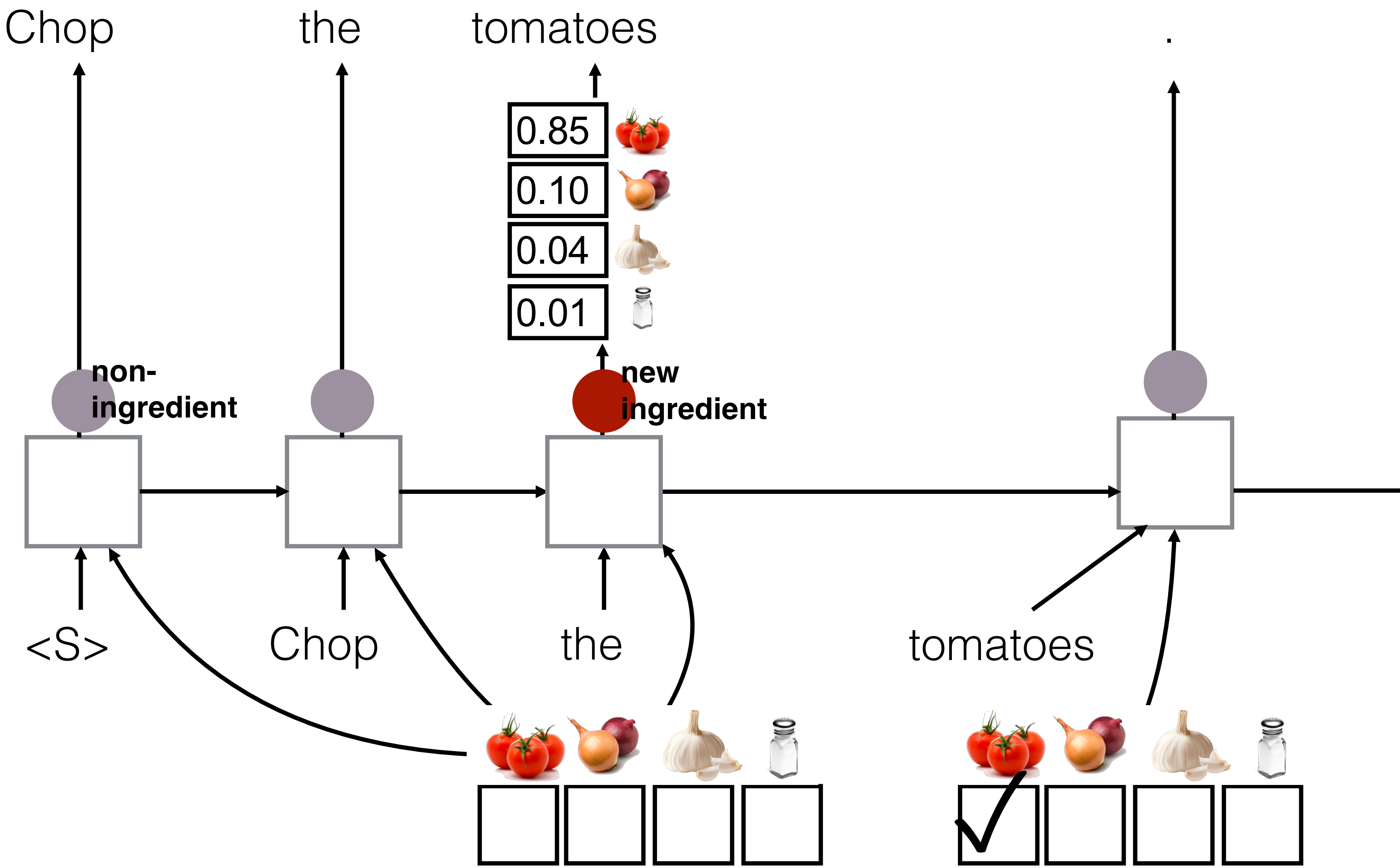
Garlic tomato salsa

tomatoes
onions
garlic
salt

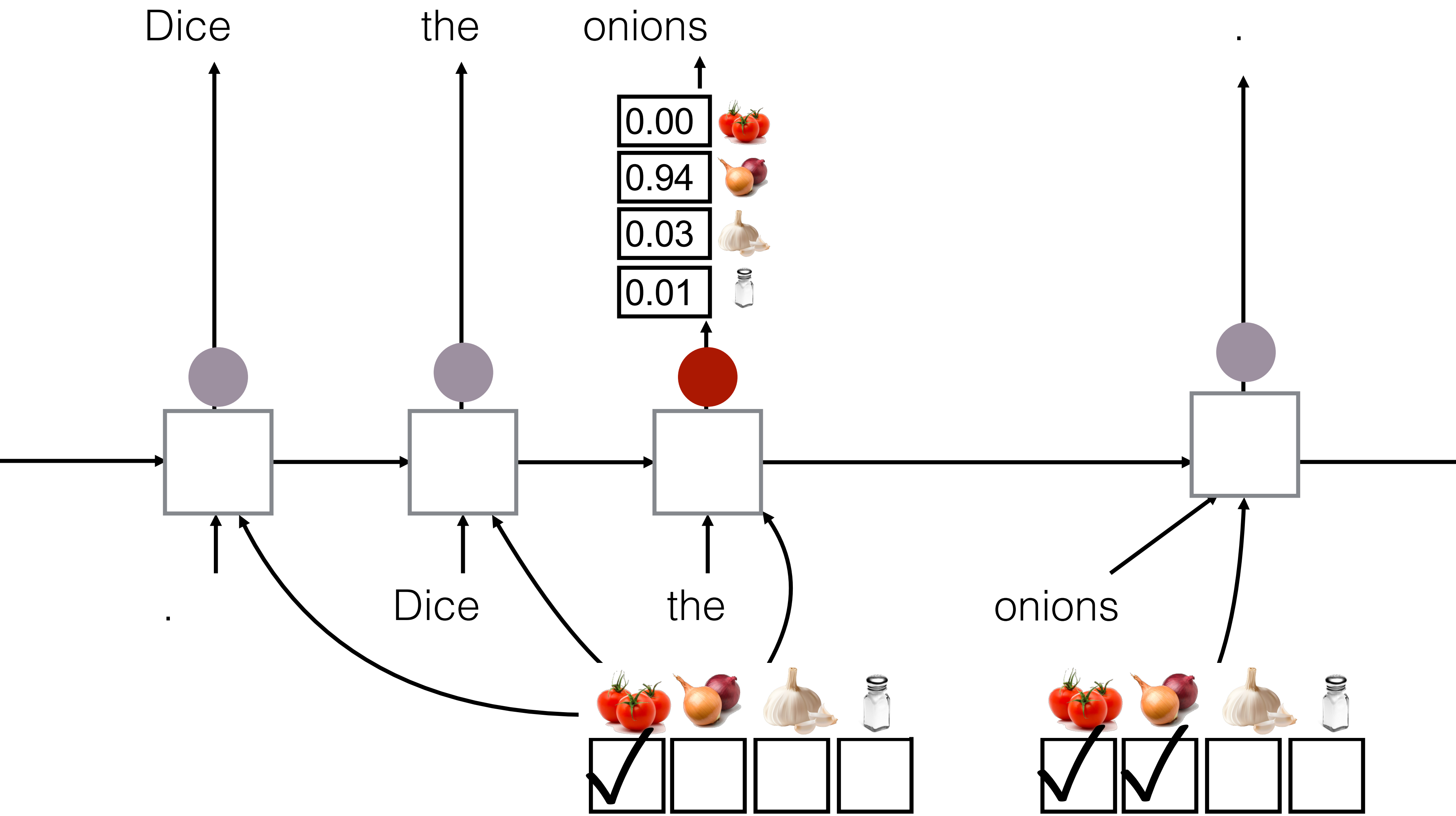
Neural checklist model



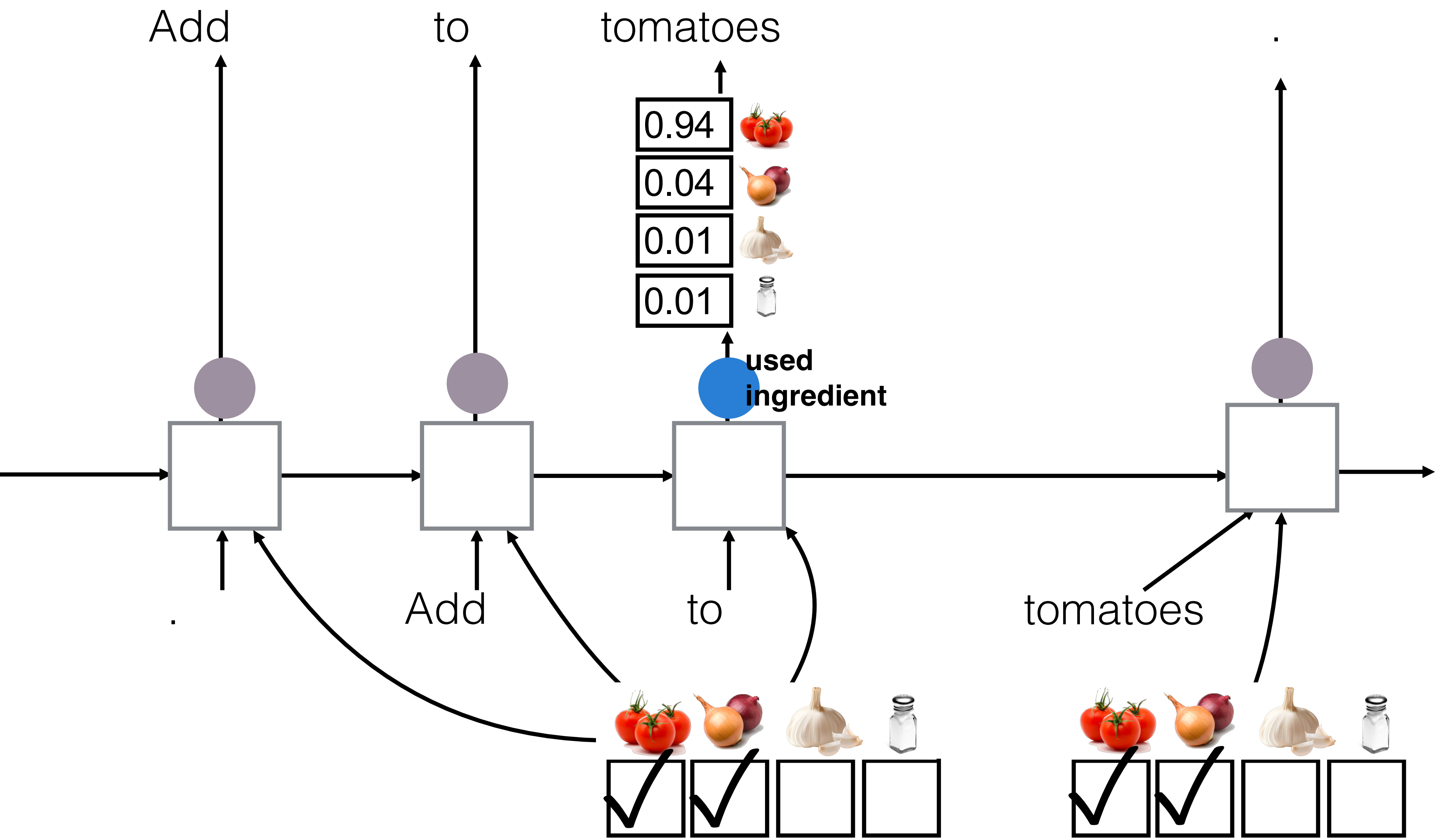
Neural checklist model



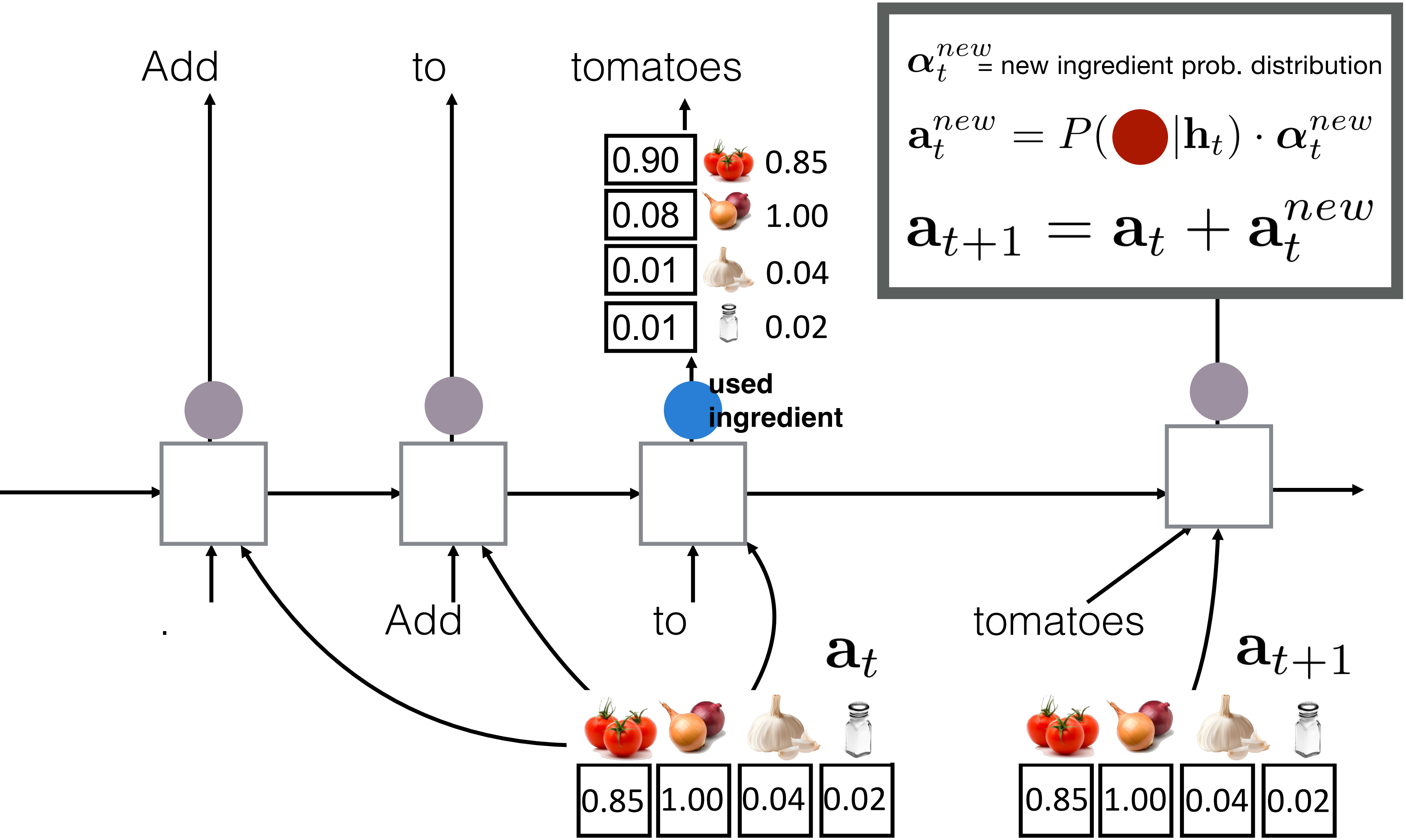
Neural checklist model



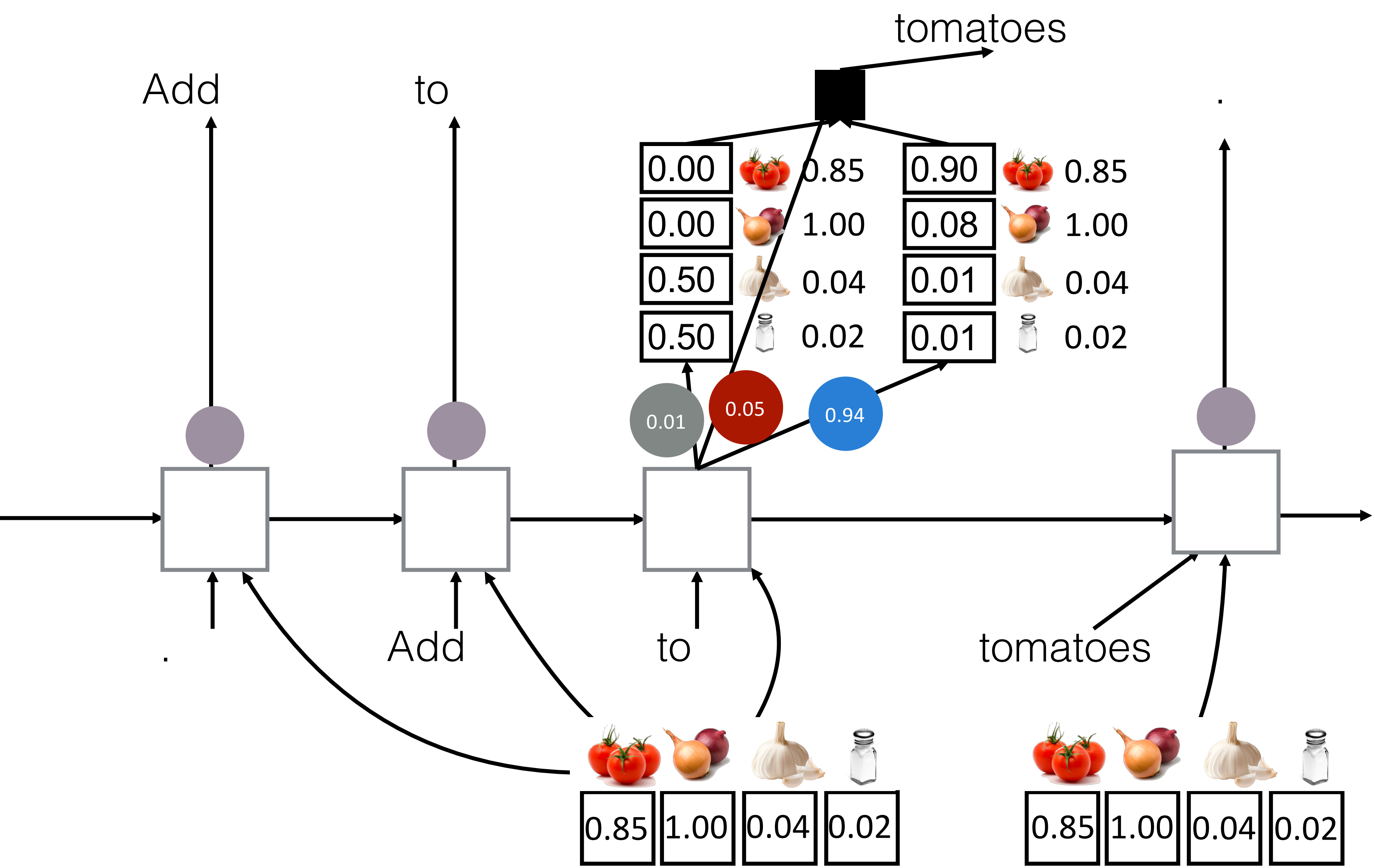
Neural checklist model



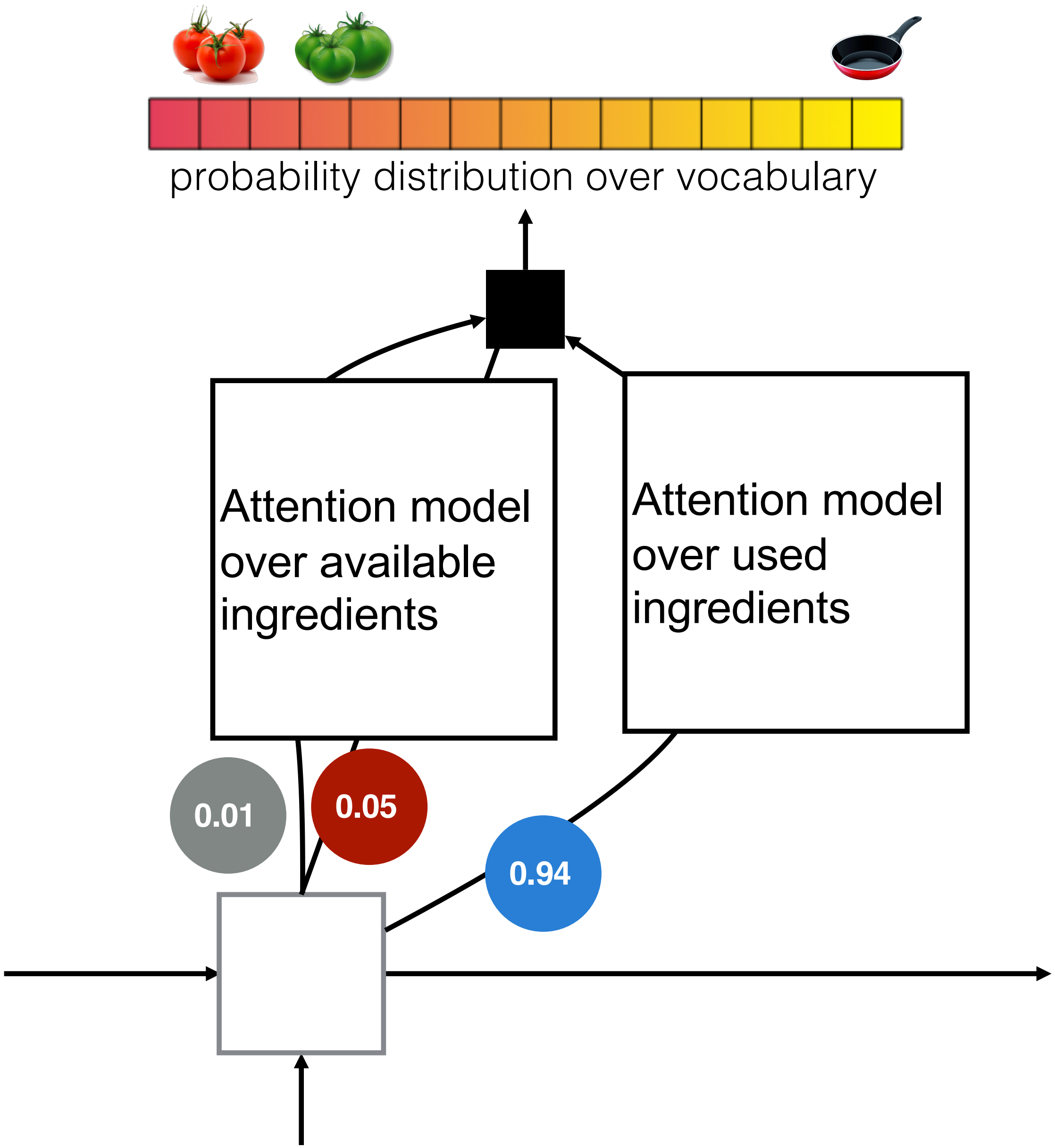
Checklist is probabilistic



Hidden state classifier is soft



Interpolation



$$W_o \in \mathbb{R}^{|V| \times k}$$

$$\mathbf{w}_t = \mathbf{softmax}(W_o \mathbf{o}_t)$$

$$\mathbf{o}_t = P(\textcolor{gray}{\bullet}|\mathbf{h}_t)\mathbf{c}_t^{LM} + P(\textcolor{red}{\bullet}|\mathbf{h}_t)\mathbf{c}_t^{new} + P(\textcolor{blue}{\bullet}|\mathbf{h}_t)\mathbf{c}_t^{used}$$

$$\mathbf{c}_t^{LM} = W_h \mathbf{h}_t$$

$$W_h \in \mathbb{R}^{k \times k}$$

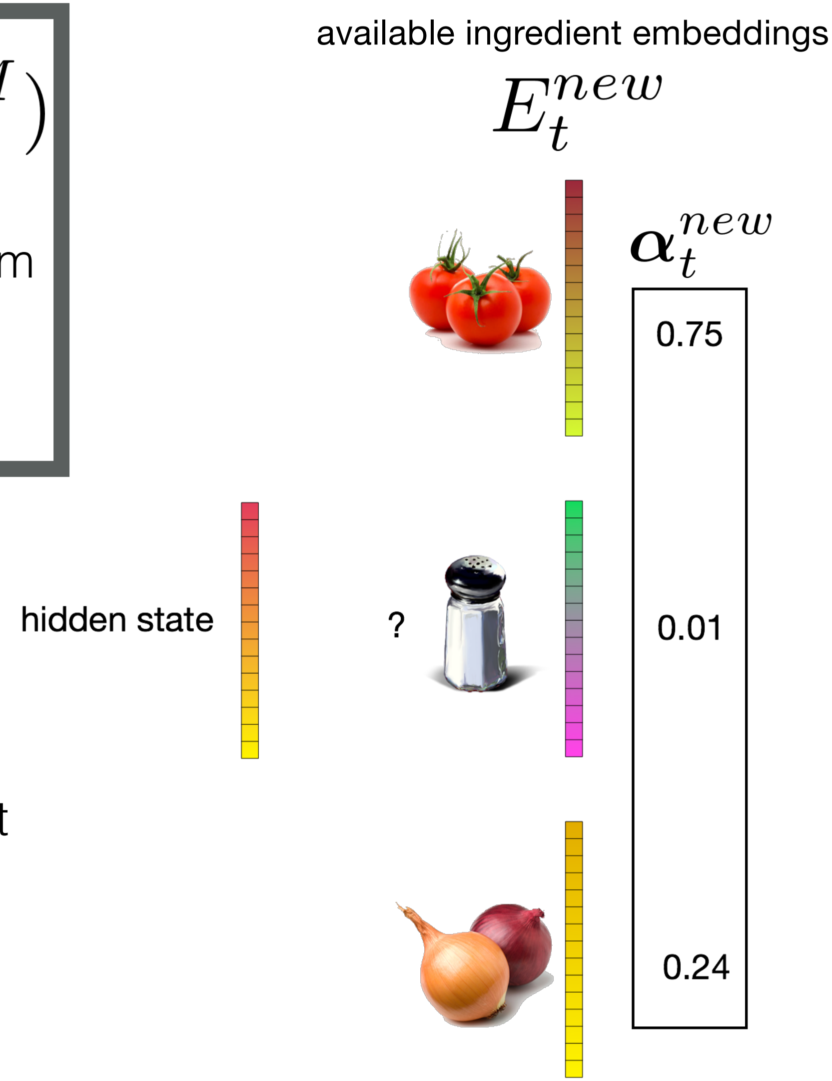
Choose ingredient via attention

$$\boldsymbol{\alpha}_t^{new} = \mathrm{softmax}(\gamma E_t^{new} \mathbf{c}_t^{LM})$$

temperature term → γ

available ingredient embeddings → E_t^{new}

content vector from language model → $\mathbf{c}_t^{LM}$



<u>Attention models for other NLP tasks</u>

MT (Balasubramanian et al. 13, Bahdanau et al. 14)

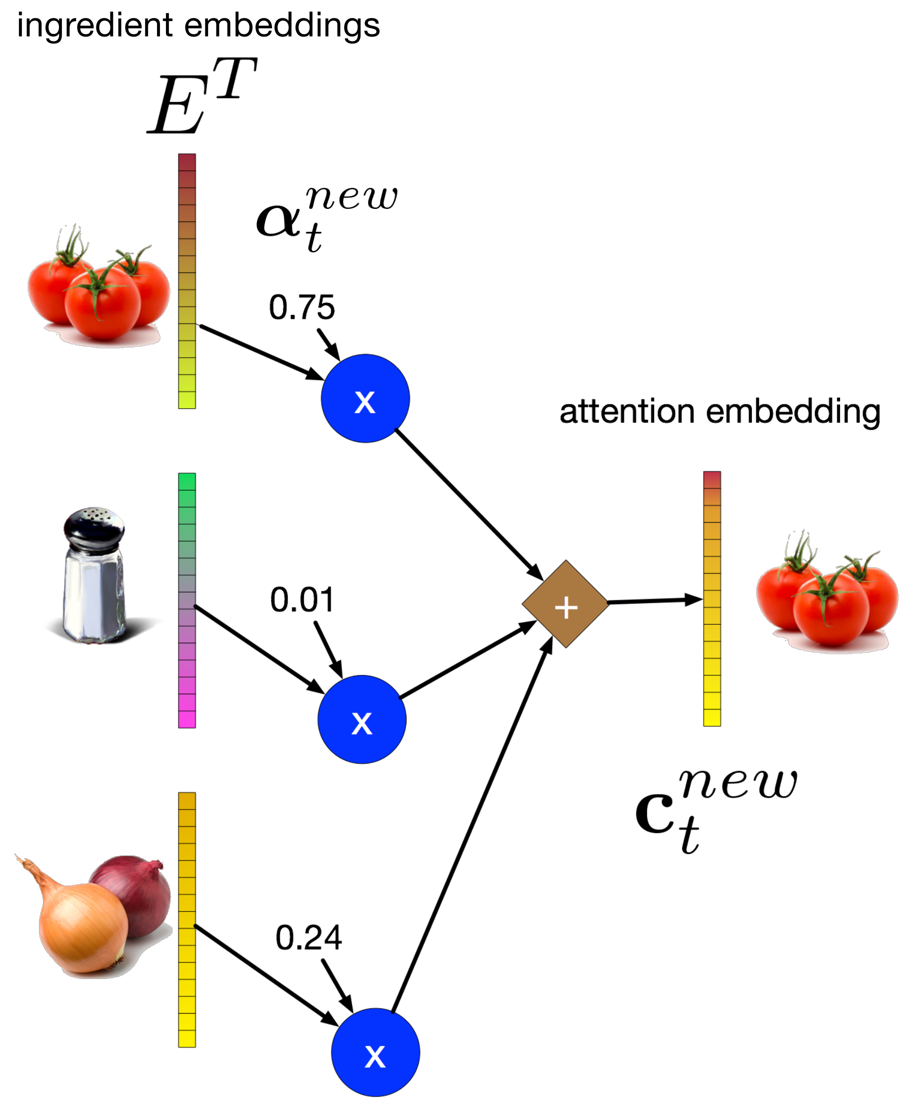
Sentence summarization (Rush et al. 15)

Machine reading (Cheng et al. 16)

Image captioning (Xu et al. 15)

Attention-generated embeddings

Can generate an embedding from the attention probabilities



$$\mathbf{c}_t^{new} = E^T \boldsymbol{\alpha}_t^{new}$$

Neural Recipe Example #1

title: oven eggplant

In a small bowl , combine the cheese , eggplant , basil , oregano , tomato sauce and onion . Mix well . Shape mixture into 6 patties , each about 3/4-inch thick.
Place on baking sheet .
Bake at 350 degrees for 30 minutes or until lightly browned .
Southern living magazine , sometime in 1980 .
Typed for you by nancy coleman .

eggplant
cheese cottage lowfat
chopped onion
bay ground leaf
basil
oregano
tomato sauce
provolone

Cook eggplant in boiling water , covered , for 10 min .
Drain and cut in half lengthwise . scoop out insides leaving 1/2 '' shell . Mash insides with cottage cheese , onion , bay leaf , basil , oregano and tomato sauce .
Preheat oven to 350 ^ stuff eggplant halves , place in casserole dish and bake covered for 15 min .
Add a little water to bottom of pan to keep eggplant moist . top with provolone cheese .
Bake 5 more min uncovered 1 serving =