89688: Statistical Machine Translation

Neural Machine Translation

May 2020

Roee Aharoni
Computer Science Department
Bar Ilan University

Based in part on slides by Kevin Duh and Hermann Ney from the DL4MT winter school

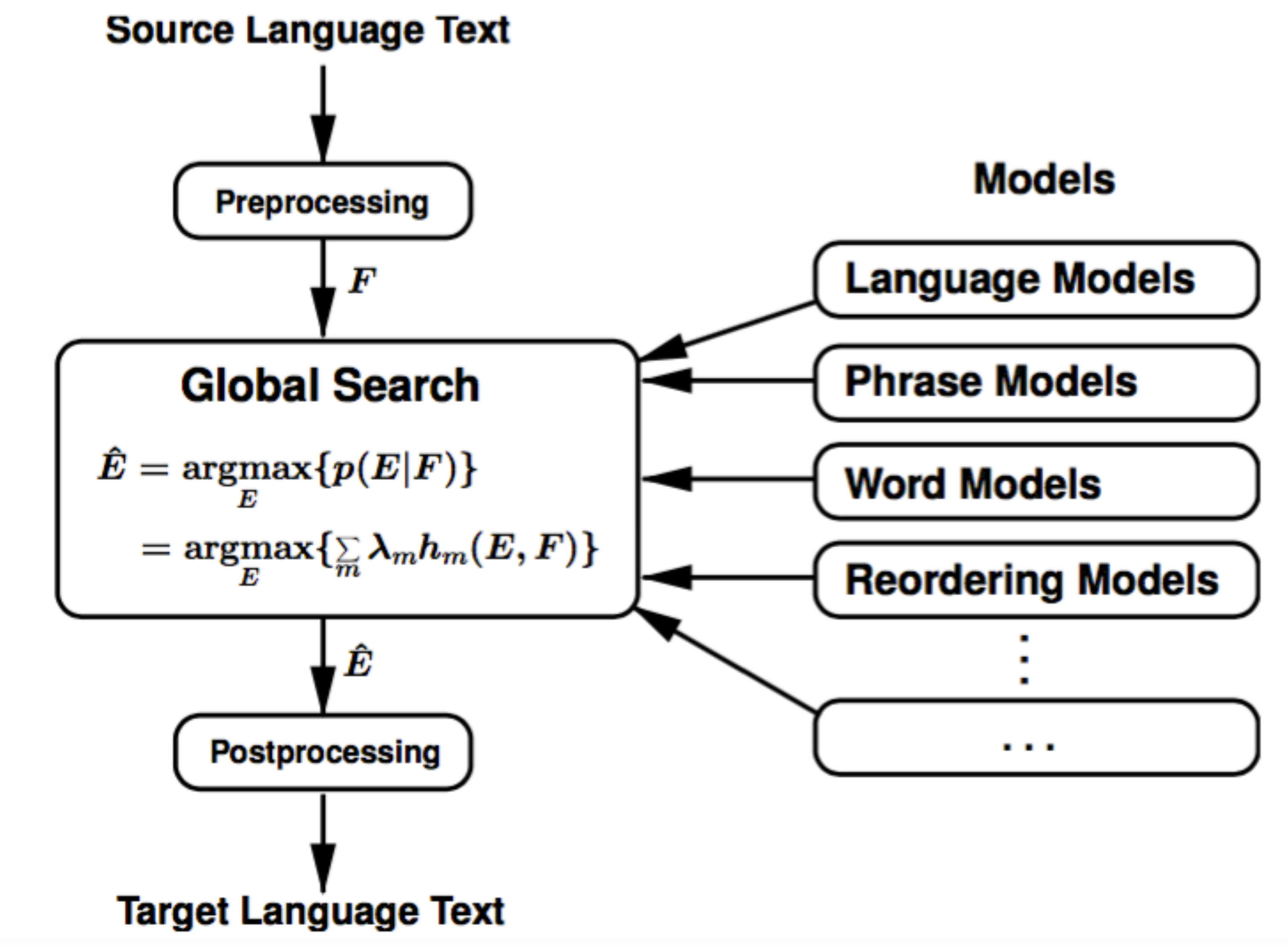
The problems of statistical MT

The problems of statistical MT

- Lots of moving parts (language model, alignment model, phrase table construction, distortion model, reordering models, tuning…)

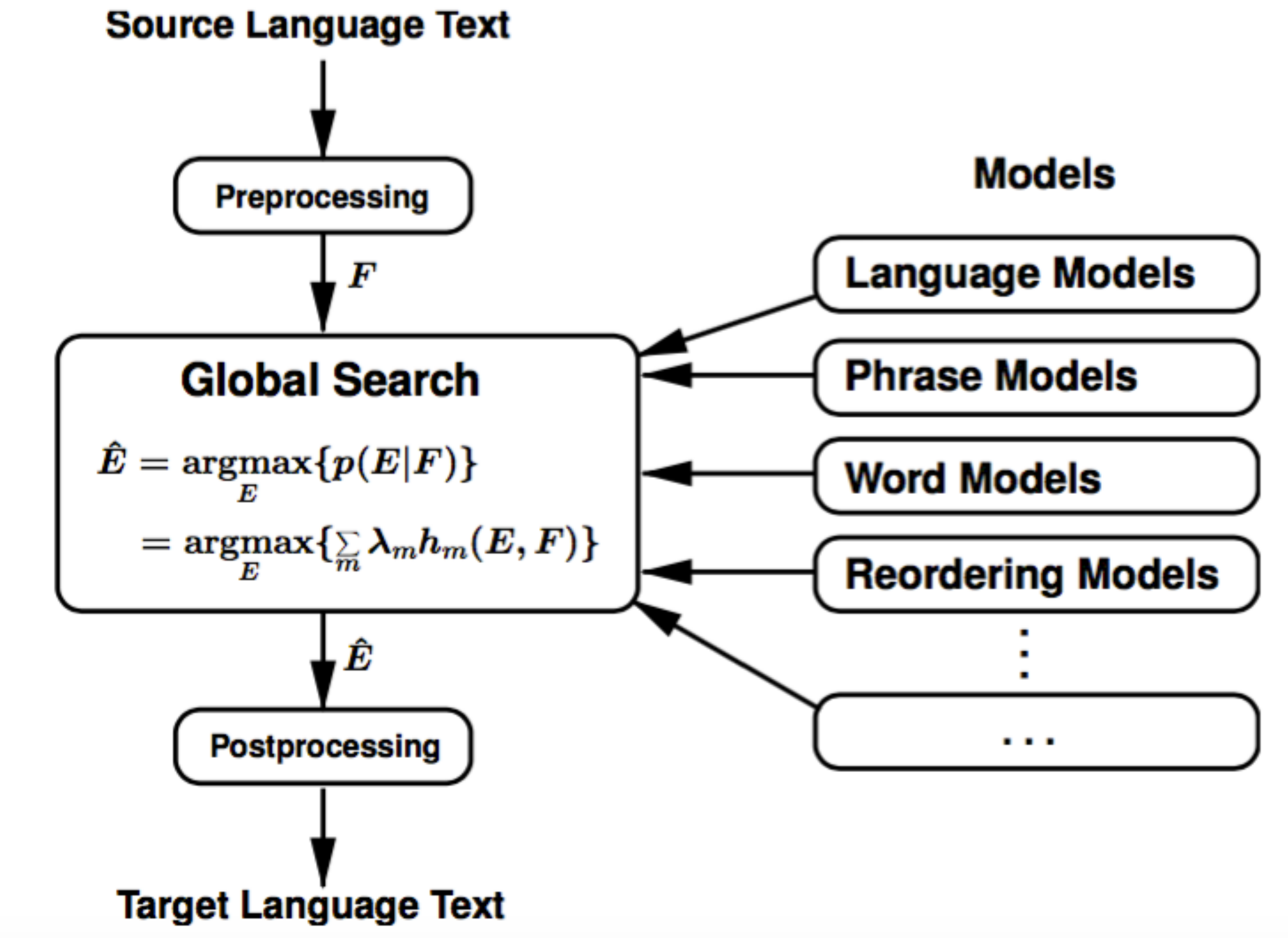
The problems of statistical MT

- Lots of moving parts (language model, alignment model, phrase table construction, distortion model, reordering models, tuning…)



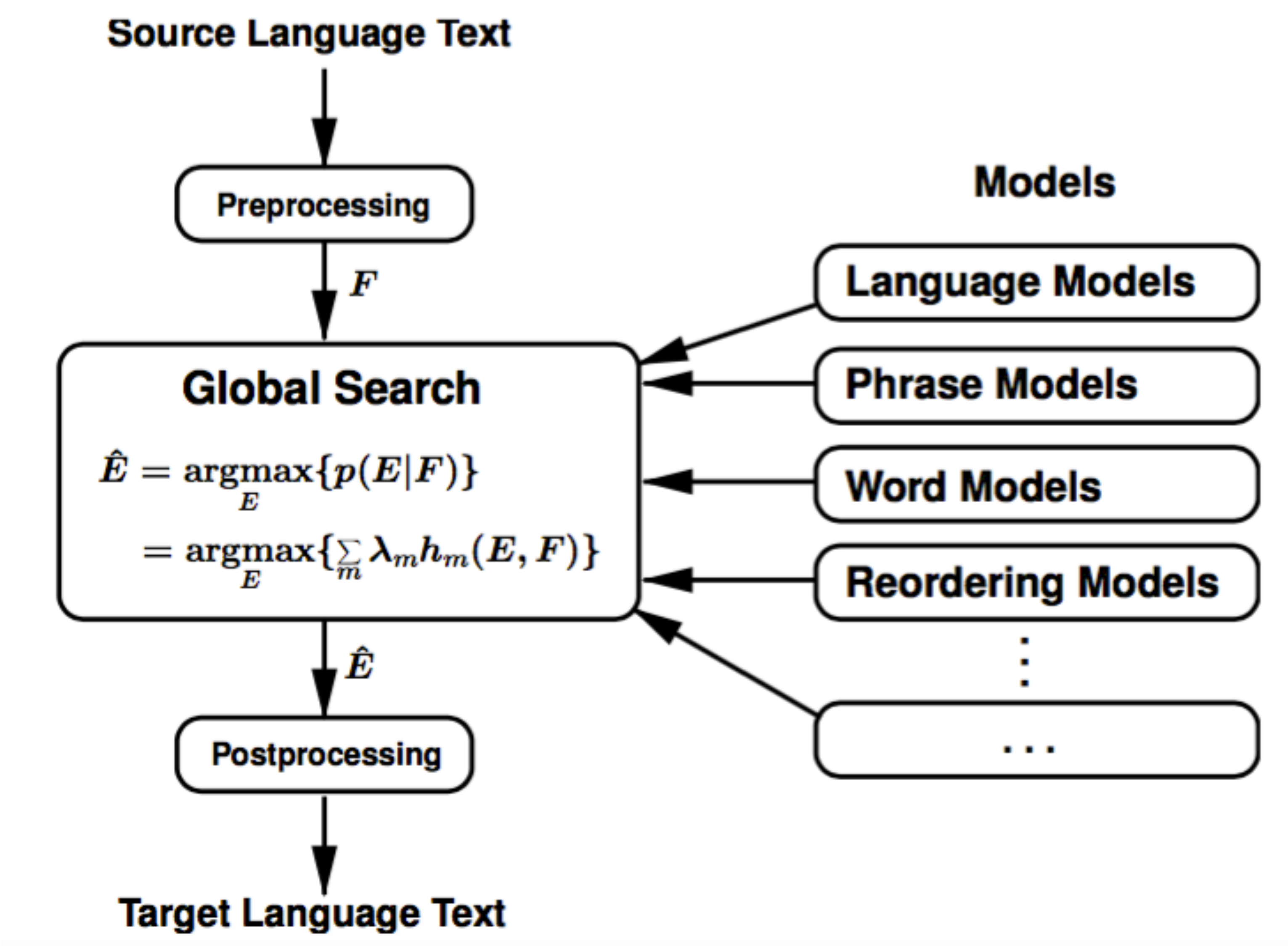
The problems of statistical MT

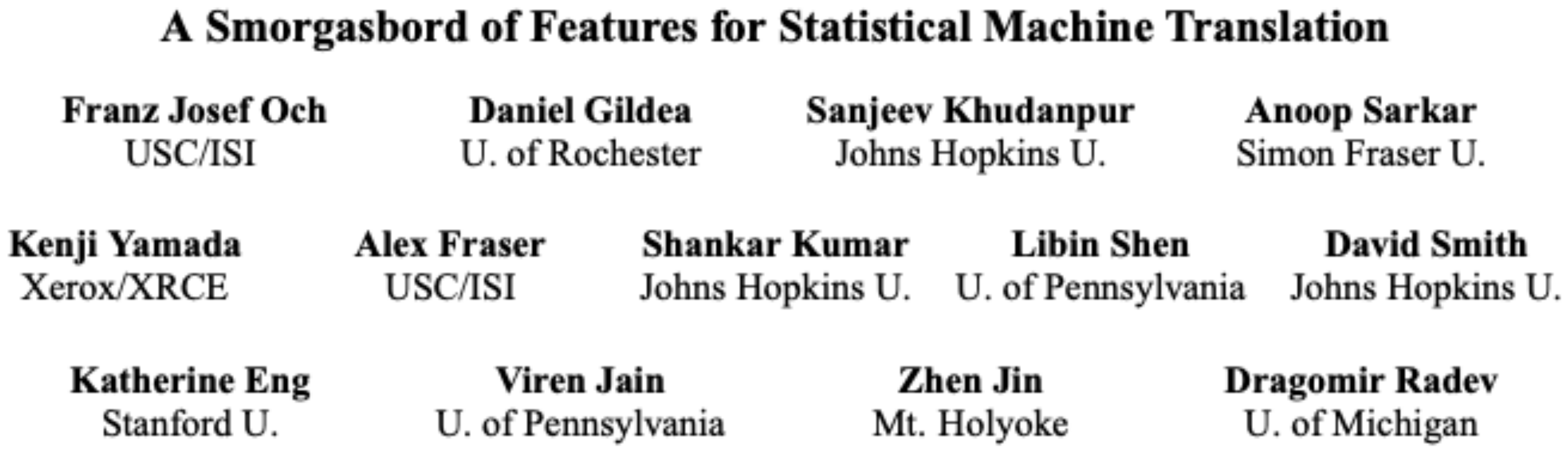
- Lots of moving parts (language model, alignment model, phrase table construction, distortion model, reordering models, tuning…)
- Requires extensive feature engineering



The problems of statistical MT

- Lots of moving parts (language model, alignment model, phrase table construction, distortion model, reordering models, tuning…)
- Requires extensive feature engineering



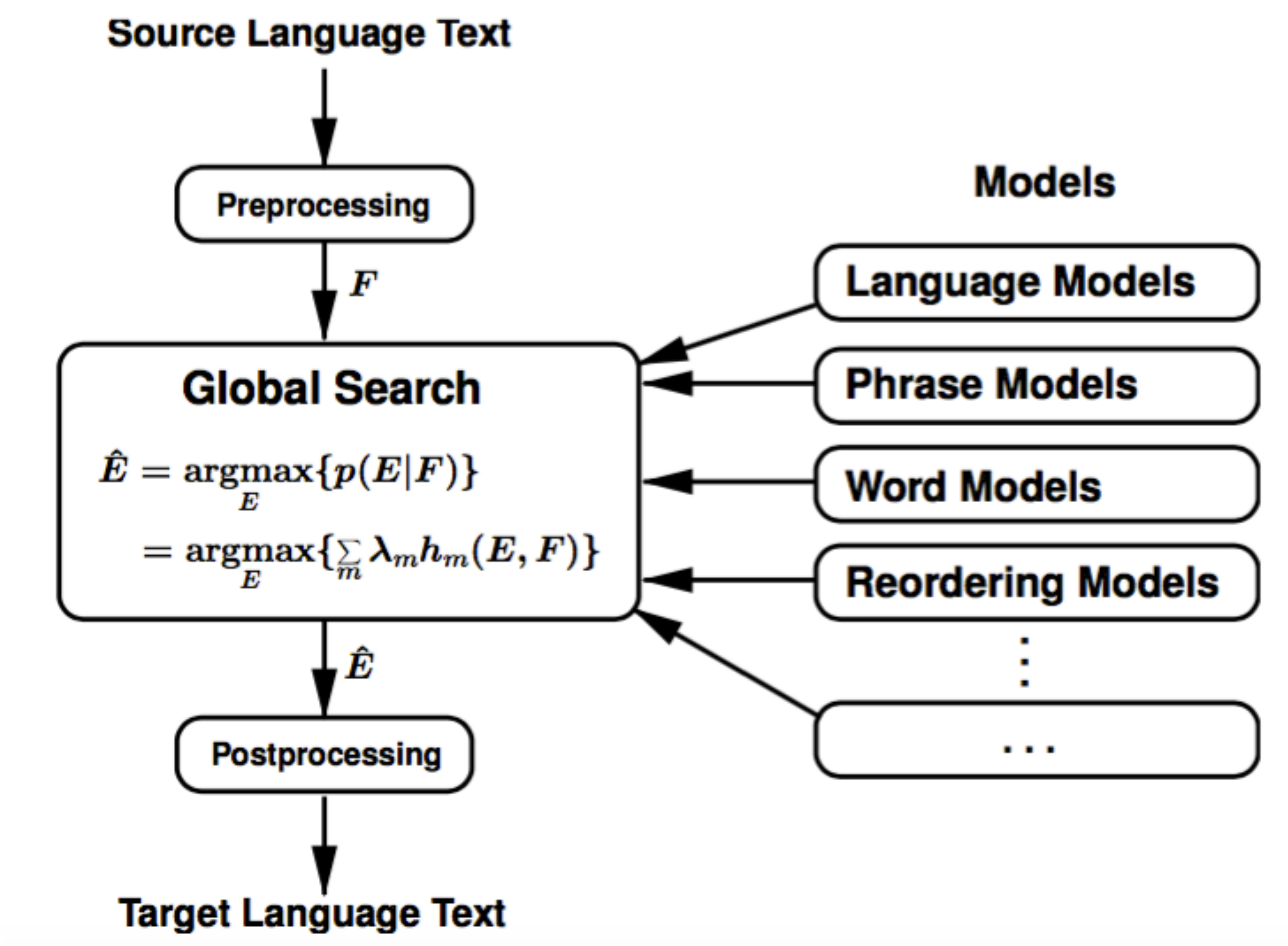


A Smorgasbord of Features for Statistical Machine Translation

Franz Josef Och USC/ISI
Daniel Gildea U. of Rochester
Sanjeev Khudanpur Johns Hopkins U.
Anoop Sarkar Simon Fraser U.
Kenji Yamada Xerox/XRCE
Alex Fraser USC/ISI
Shankar Kumar Johns Hopkins U.
Libin Shen U. of Pennsylvania
David Smith Johns Hopkins U.
Katherine Eng Stanford U.
Viren Jain U. of Pennsylvania
Zhen Jin Mt. Holyoke
Dragomir Radev U. of Michigan

The problems of statistical MT

- Lots of moving parts (language model, alignment model, phrase table construction, distortion model, reordering models, tuning…)
- Requires extensive feature engineering



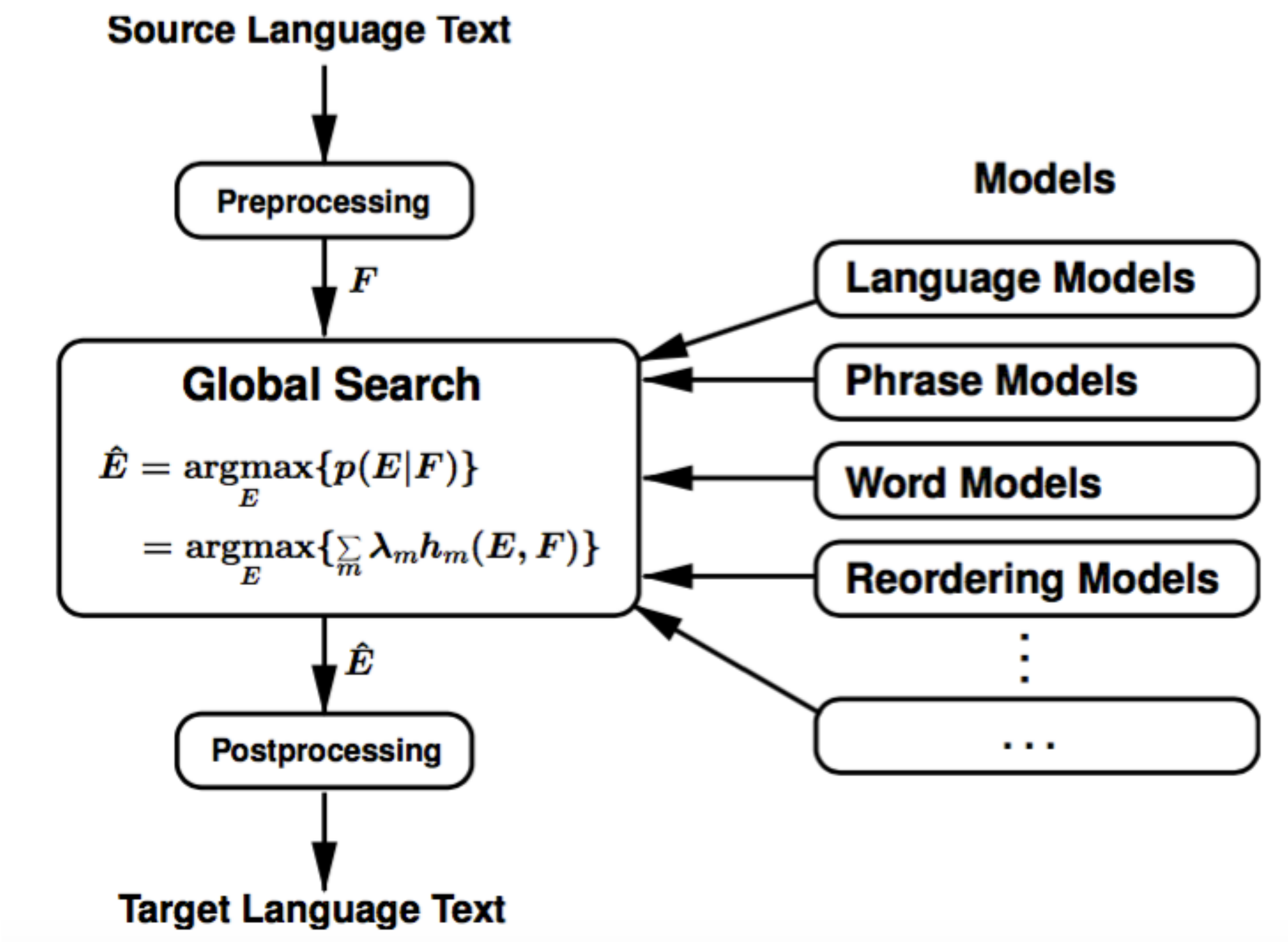
11,001 New Features for Statistical Machine Translation*

David Chiang and **Kevin Knight**
USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292 USA

Wei Wang
Language Weaver, Inc.
4640 Admiralty Way, Suite 1210
Marina del Rey, CA 90292 USA

The problems of statistical MT

- Lots of moving parts (language model, alignment model, phrase table construction, distortion model, reordering models, tuning…)
- Requires extensive feature engineering
- Hard and expensive to capture long-range dependencies



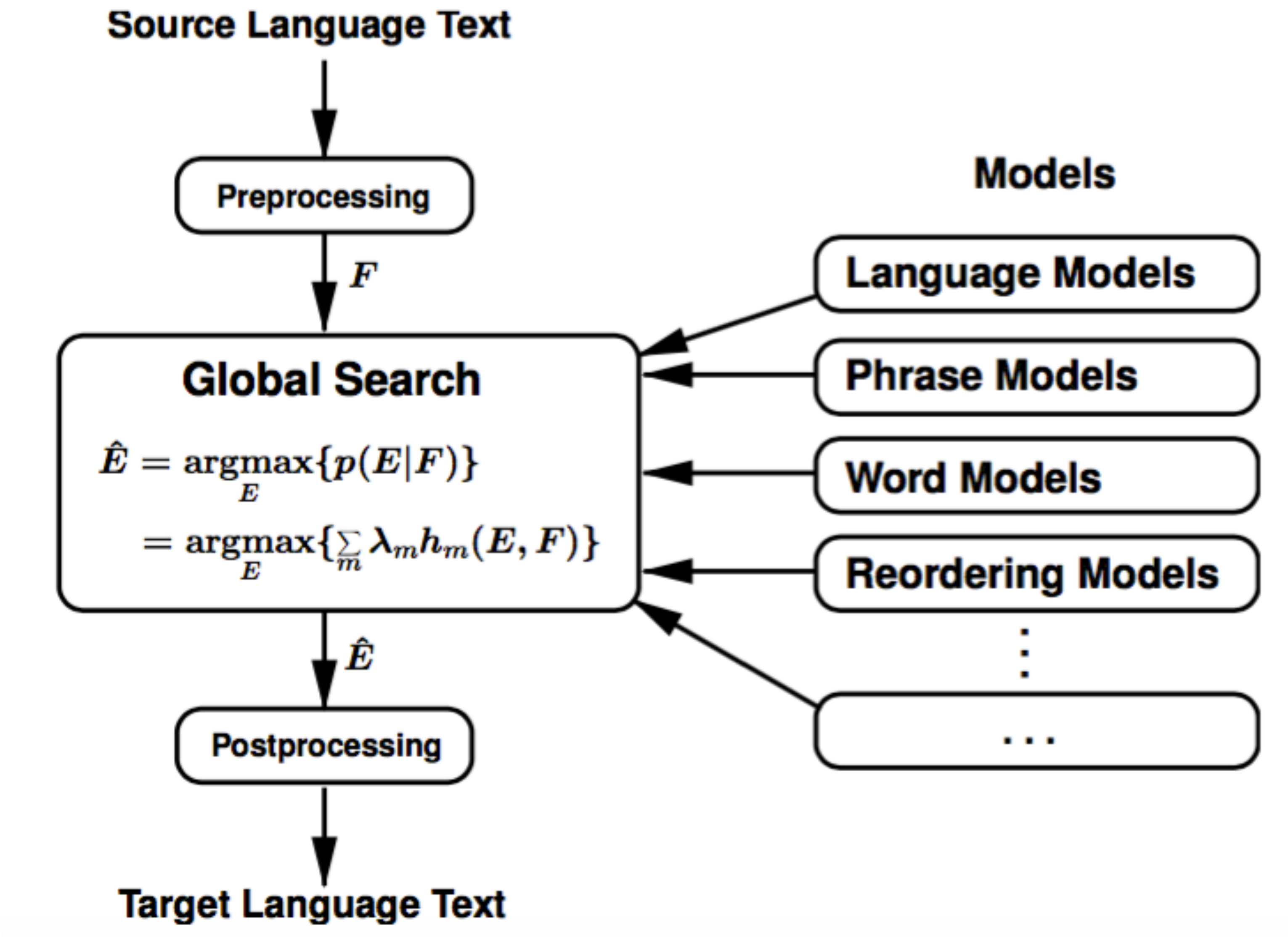
11,001 New Features for Statistical Machine Translation*

David Chiang and **Kevin Knight**
USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292 USA

Wei Wang
Language Weaver, Inc.
4640 Admiralty Way, Suite 1210
Marina del Rey, CA 90292 USA

The problems of statistical MT

- Lots of moving parts (language model, alignment model, phrase table construction, distortion model, reordering models, tuning…)
- Requires extensive feature engineering
- Hard and expensive to capture long-range dependencies
- Does not generalize for similar words

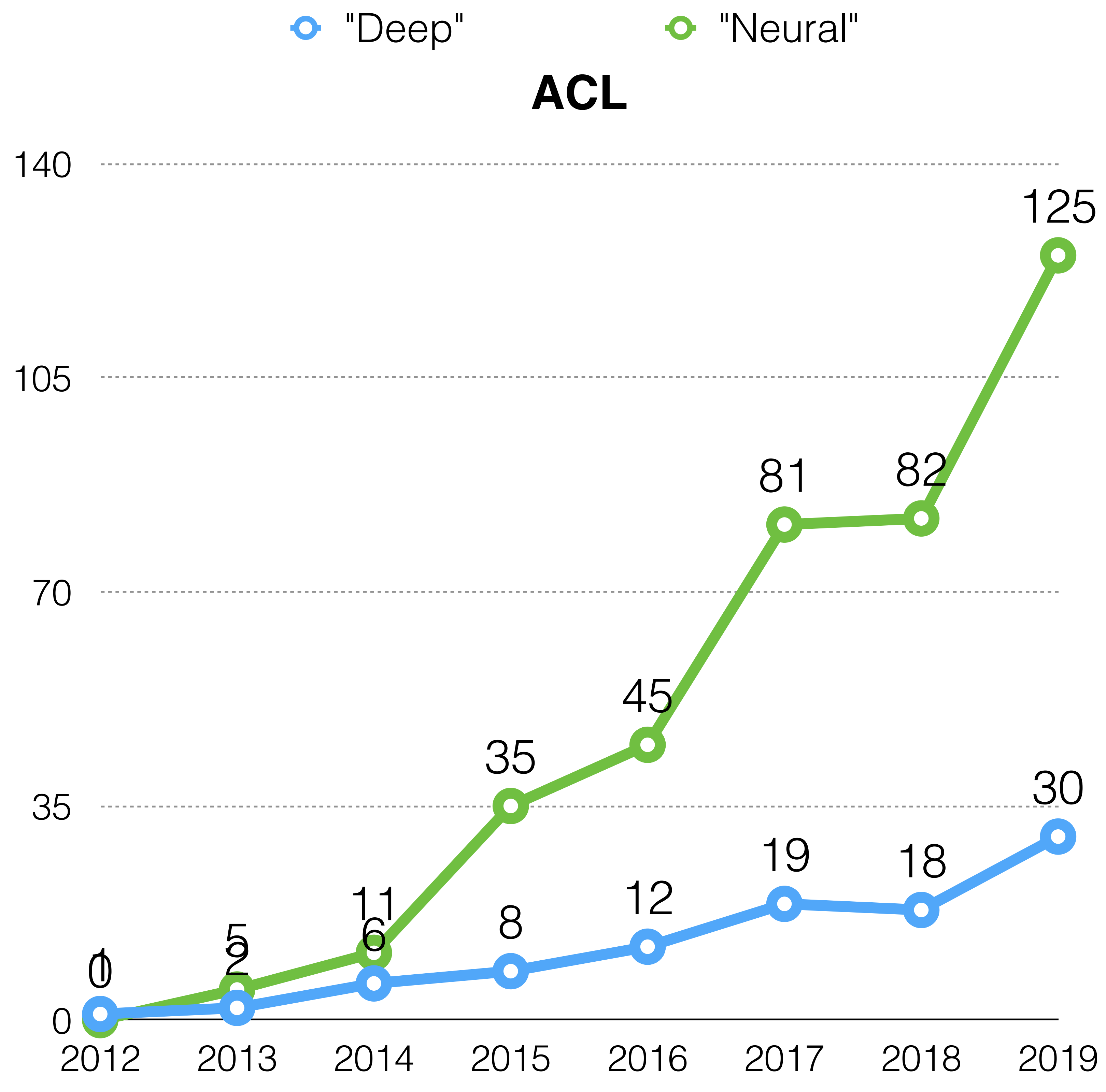


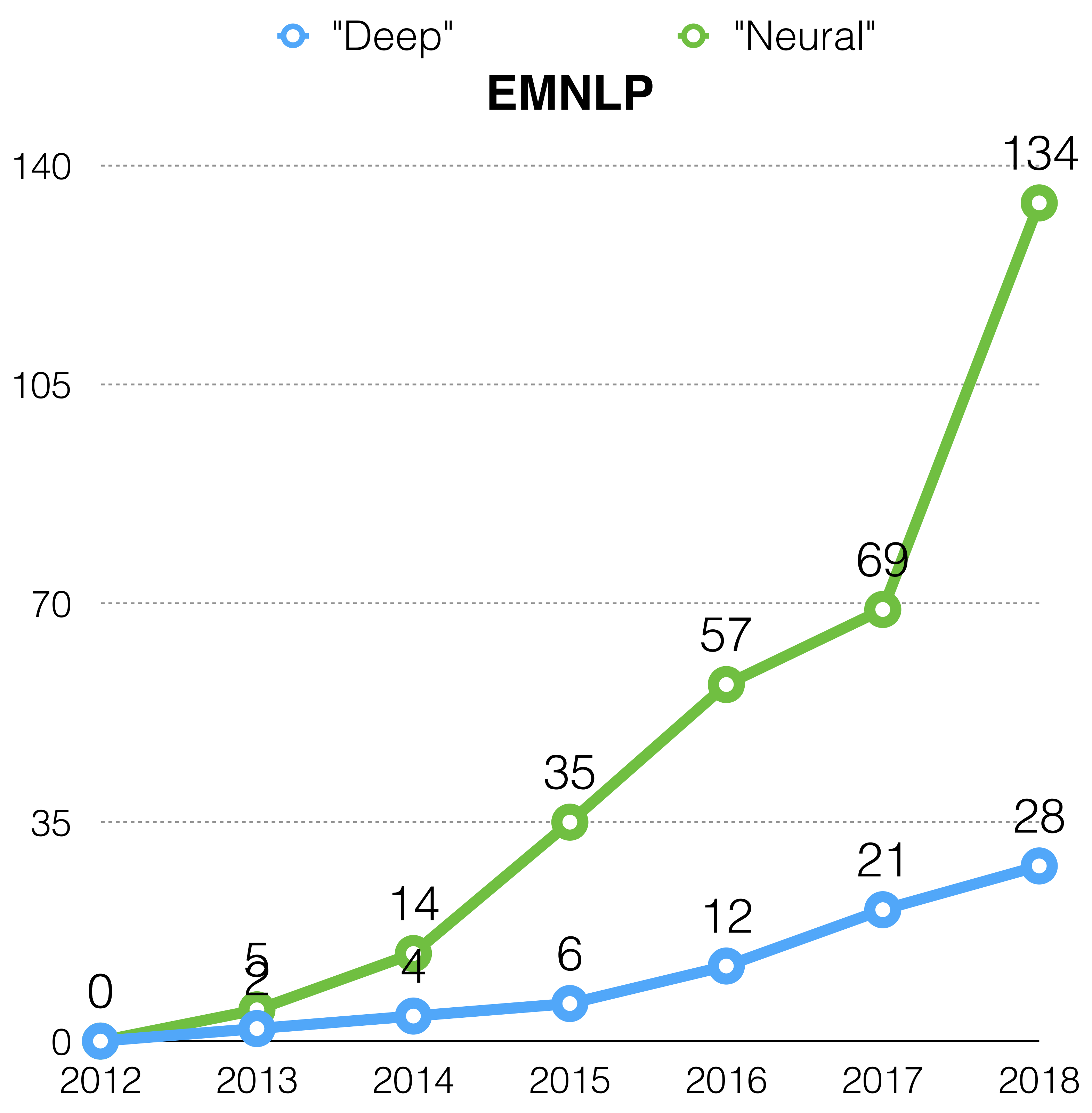
11,001 New Features for Statistical Machine Translation*

David Chiang and **Kevin Knight**
USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292 USA

Wei Wang
Language Weaver, Inc.
4640 Admiralty Way, Suite 1210
Marina del Rey, CA 90292 USA

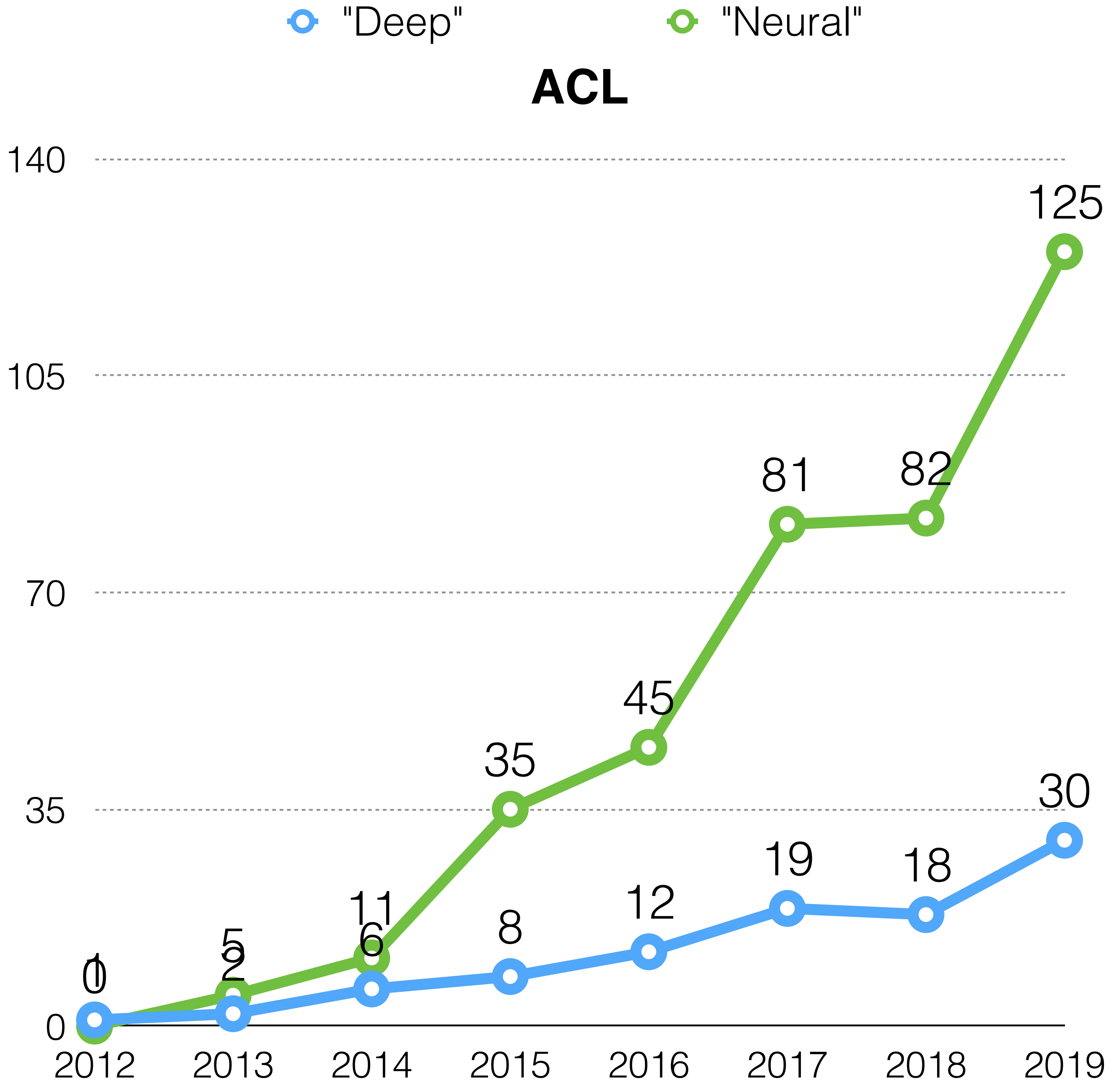
A New Paradigm?

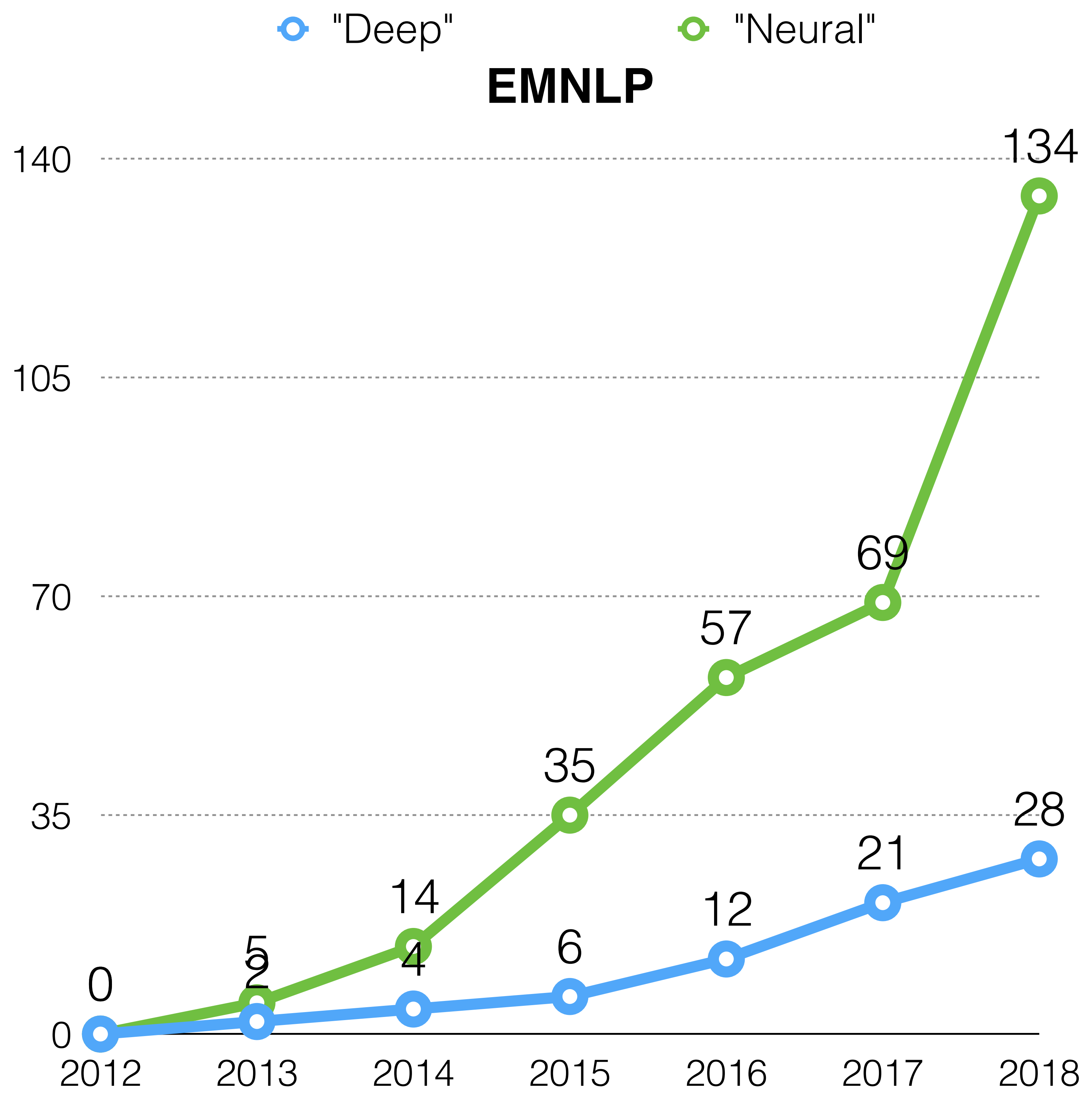




A New Paradigm?

of mentions in paper titles at top-tier NLP conferences (ACL, EMNLP) from 2012 to 2018:





What is Deep Learning?

A family of machine learning methods that use deep architectures to learn high-level feature representations from data

What is Deep Learning?

A family of **machine learning methods** that use **deep architectures** to learn **high-level feature representations** from data

A basic machine learning setup

A basic machine learning setup

- Given a dataset of: $(x^{(m)}, y^{(m)})_{m=\{1,2,..M\}}$ training examples,

A basic machine learning setup

- Given a dataset of: $(x^{(m)}, y^{(m)})_{m=\{1,2,..M\}}$ training examples,
  - input: $x^{(m)} \in R^d$

A basic machine learning setup

- Given a dataset of: $(x^{(m)}, y^{(m)})_{m=\{1,2,..M\}}$ training examples,
  - input: $x^{(m)} \in R^d$
  - output: $y^{(m)} = \{0, 1\}$

A basic machine learning setup

- Given a dataset of: $(x^{(m)}, y^{(m)})_{m=\{1,2,..M\}}$ training examples,
  - input: $x^{(m)} \in R^d$
  - output: $y^{(m)} = \{0,1\}$
- Learn a function $f : x \to y$ to predict correctly on new inputs.

A basic machine learning setup

- Given a dataset of: $(x^{(m)}, y^{(m)})_{m=\{1,2,..M\}}$ training examples,
  - input: $x^{(m)} \in R^d$
  - output: $y^{(m)} = \{0, 1\}$
- Learn a function $f : x \to y$ to predict correctly on new inputs.
  - step I: pick a learning algorithm (SVM, log. reg., NN…)

A basic machine learning setup

- Given a dataset of: $(x^{(m)}, y^{(m)})_{m=\{1,2,..M\}}$ training examples,
  - input: $x^{(m)} \in R^d$
  - output: $y^{(m)} = \{0, 1\}$
- Learn a function $f : x \rightarrow y$ to predict correctly on new inputs.
  - step I: pick a learning algorithm (SVM, log. reg., NN…)
  - step II: optimize it w.r.t a loss, i.e: $\min_w \sum_{m=1}^{M} (f_w(x^{(m)}) - y^{(m)})^2$

Logistic Regression - The “Single Layer” Neural Network

Logistic Regression - The “Single Layer” Neural Network

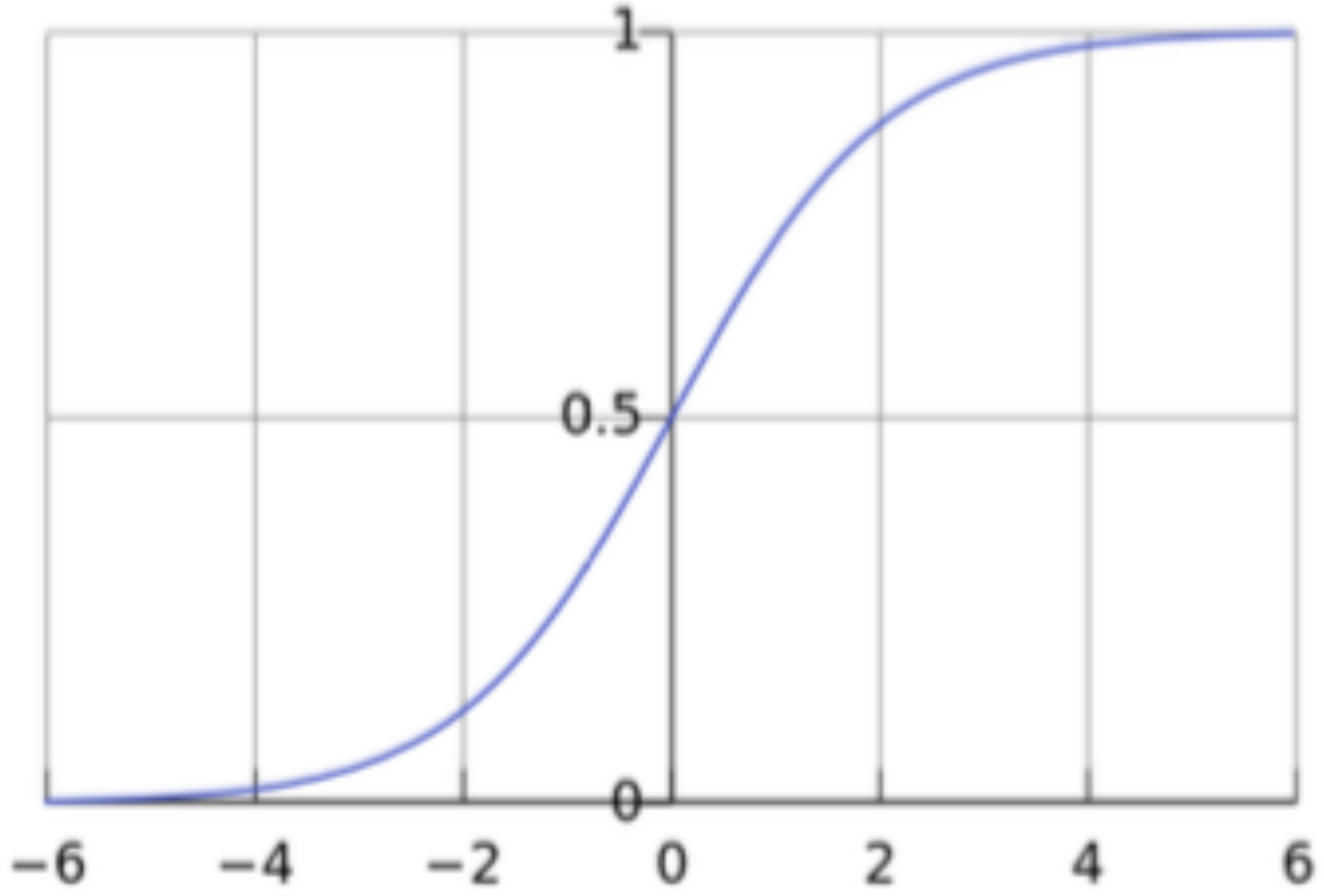
- Model the classifier as: $f(x) = \sigma(w^T \cdot x) = \sigma(\sum_i w_i x_i)$

Logistic Regression - The “Single Layer” Neural Network

- Model the classifier as: $f(x) = \sigma(w^T \cdot x) = \sigma(\sum_i w_i x_i)$
- Learn the weight vector $w \in R^d$ using gradient-descent (next slide)

Logistic Regression - The “Single Layer” Neural Network

- Model the classifier as: $f(x) = \sigma(w^T \cdot x) = \sigma(\sum_i w_i x_i)$
- Learn the weight vector $w \in R^d$ using gradient-descent (next slide)
- $\sigma(z) = \frac{1}{1+e^{-z}}$ is a non-linearity, e.g. the sigmoid function (creates dependency between the features, maps $f(x)$ to [0,1]):



Training (Logistic Regression) with Gradient Descent

Training (Logistic Regression) with Gradient Descent

- Define the loss-function (squared error, cross entropy...):

$$Loss(w) = \frac{1}{2} \sum_m (\sigma(w^T x^{(m)}) - y^{(m)})^2$$

Training (Logistic Regression) with Gradient Descent

- Define the loss-function (squared error, cross entropy…):

$$Loss(w) = \frac{1}{2}\sum_m(\sigma(w^T x^{(m)}) - y^{(m)})^2$$

- Compute the gradient of the loss-function w.r.t. the weight vector, w:

$$\nabla_w Loss = \sum_m \left[\sigma(w^T x^{(m)}) - y^{(m)}\right] \sigma'(w^T x^{(m)}) x^{(m)}$$

Training (Logistic Regression) with Gradient Descent

- Define the loss-function (squared error, cross entropy…):

$$Loss(w) = \frac{1}{2}\sum_m(\sigma(w^T x^{(m)}) - y^{(m)})^2$$

- Compute the gradient of the loss-function w.r.t. the weight vector, w:

$$\nabla_w Loss = \sum_m \left[\sigma(w^T x^{(m)}) - y^{(m)}\right]\sigma'(w^T x^{(m)})x^{(m)}$$

- Perform gradient-descent:

Training (Logistic Regression) with Gradient Descent

- Define the loss-function (squared error, cross entropy...):

$$Loss(w) = \frac{1}{2}\sum_m(\sigma(w^T x^{(m)}) - y^{(m)})^2$$

- Compute the gradient of the loss-function w.r.t. the weight vector, w:

$$\nabla_w Loss = \sum_m \left[\sigma(w^T x^{(m)}) - y^{(m)}\right]\sigma'(w^T x^{(m)})x^{(m)}$$

- Perform gradient-descent:
  - Start with a random weight vector

Training (Logistic Regression) with Gradient Descent

- Define the loss-function (squared error, cross entropy...):

$$Loss(w) = \frac{1}{2}\sum_m(\sigma(w^T x^{(m)}) - y^{(m)})^2$$

- Compute the gradient of the loss-function w.r.t. the weight vector, w:

$$\nabla_w Loss = \sum_m \left[\sigma(w^T x^{(m)}) - y^{(m)}\right]\sigma'(w^T x^{(m)})x^{(m)}$$

- Perform gradient-descent:
  - Start with a random weight vector
  - Repeat until convergence: $w \leftarrow w - \gamma(\nabla_w Loss)$

Training (Logistic Regression) with Gradient Descent

- Define the loss-function (squared error, cross entropy…):

$$Loss(w) = \frac{1}{2}\sum_m(\sigma(w^T x^{(m)}) - y^{(m)})^2$$

- Compute the gradient of the loss-function w.r.t. the weight vector, w:

$$\nabla_w Loss = \sum_m \left[\sigma(w^T x^{(m)}) - y^{(m)}\right]\sigma'(w^T x^{(m)})x^{(m)}$$

- Perform gradient-descent:
  - Start with a random weight vector
  - Repeat until convergence: $w \leftarrow w - \gamma(\nabla_w Loss)$
  - γ is the learning rate, which is a hyper-parameter

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD)

Instead of deriving the loss on all training examples per iteration, use only a sub-set of (random) examples per iteration ("mini-batch"):

Stochastic Gradient Descent (SGD)

Instead of deriving the loss on all training examples per iteration, use only a sub-set of (random) examples per iteration ("mini-batch"):

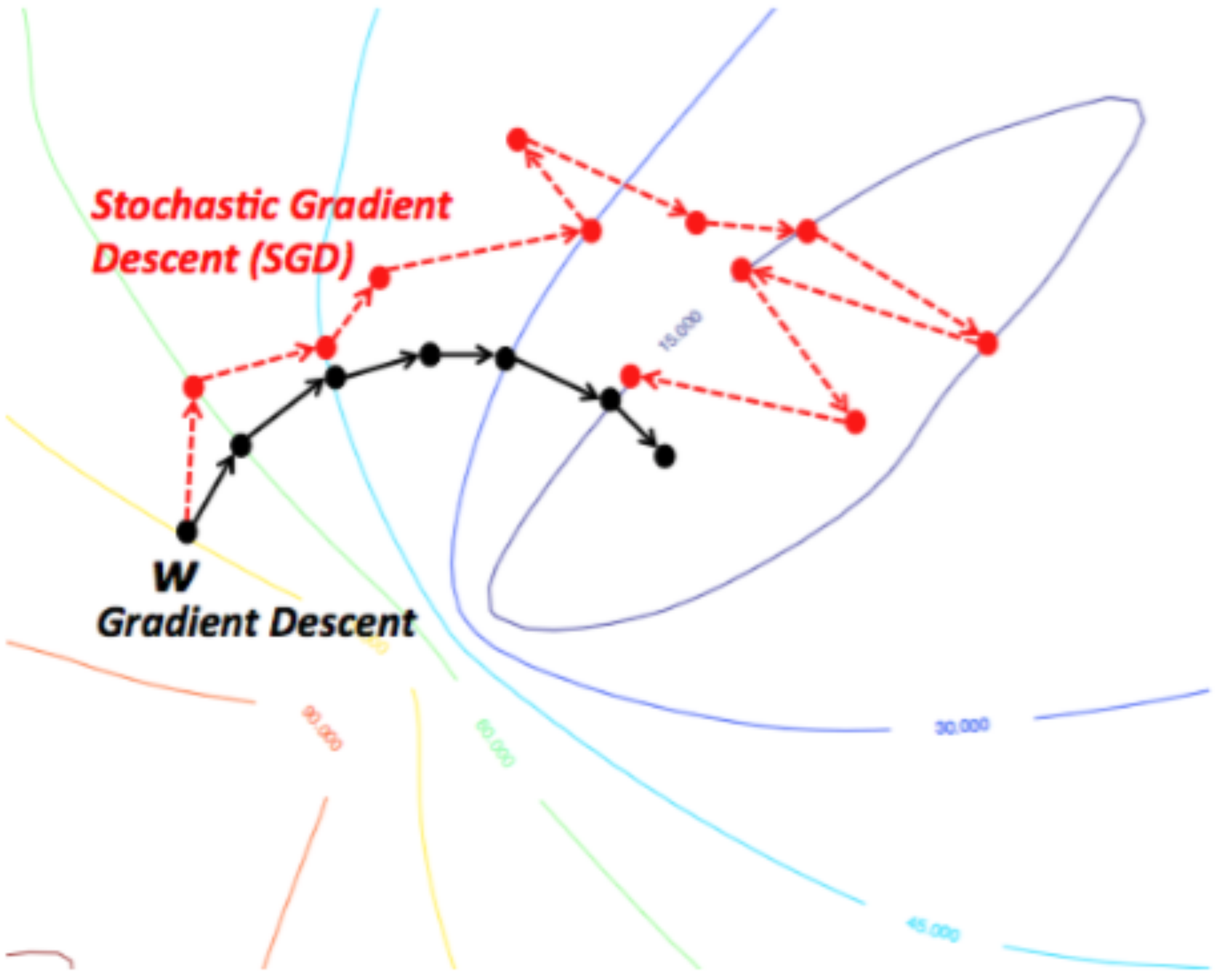
$$w \leftarrow w - \gamma\left(\frac{1}{|B|} \sum_{m \in B} Error^{(m)} * \sigma'(in^{(m)}) * x^{(m)}\right)$$

Stochastic Gradient Descent (SGD)

Instead of deriving the loss on all training examples per iteration, use only a sub-set of (random) examples per iteration (“mini-batch”):

$$w \leftarrow w - \gamma\left(\frac{1}{|B|}\sum_{m \in B} Error^{(m)} * \sigma'(in^{(m)}) * x^{(m)}\right)$$

Faster to converge (more updates per epoch), but more noisy



Multi Layer Perceptron (MLP) - a “Deep" NN

Multi Layer Perceptron (MLP) - a “Deep" NN

- Model the classifier as:

$$f(x) = \sigma(\sum_j w_j \cdot h_j) = \sigma(\sum_j w_j \cdot \sigma(\sum_i w_{ij} x_i))$$

Multi Layer Perceptron (MLP) - a “Deep" NN

- Model the classifier as:

$$f(x) = \sigma(\sum_j w_j \cdot h_j) = \sigma(\sum_j w_j \cdot \sigma(\sum_i w_{ij} x_i))$$

- Can be seen as multilayer logistic regression

Multi Layer Perceptron (MLP) - a “Deep" NN

- Model the classifier as:

$$f(x) = \sigma(\sum_j w_j \cdot h_j) = \sigma(\sum_j w_j \cdot \sigma(\sum_i w_{ij} x_i))$$

- Can be seen as multilayer logistic regression

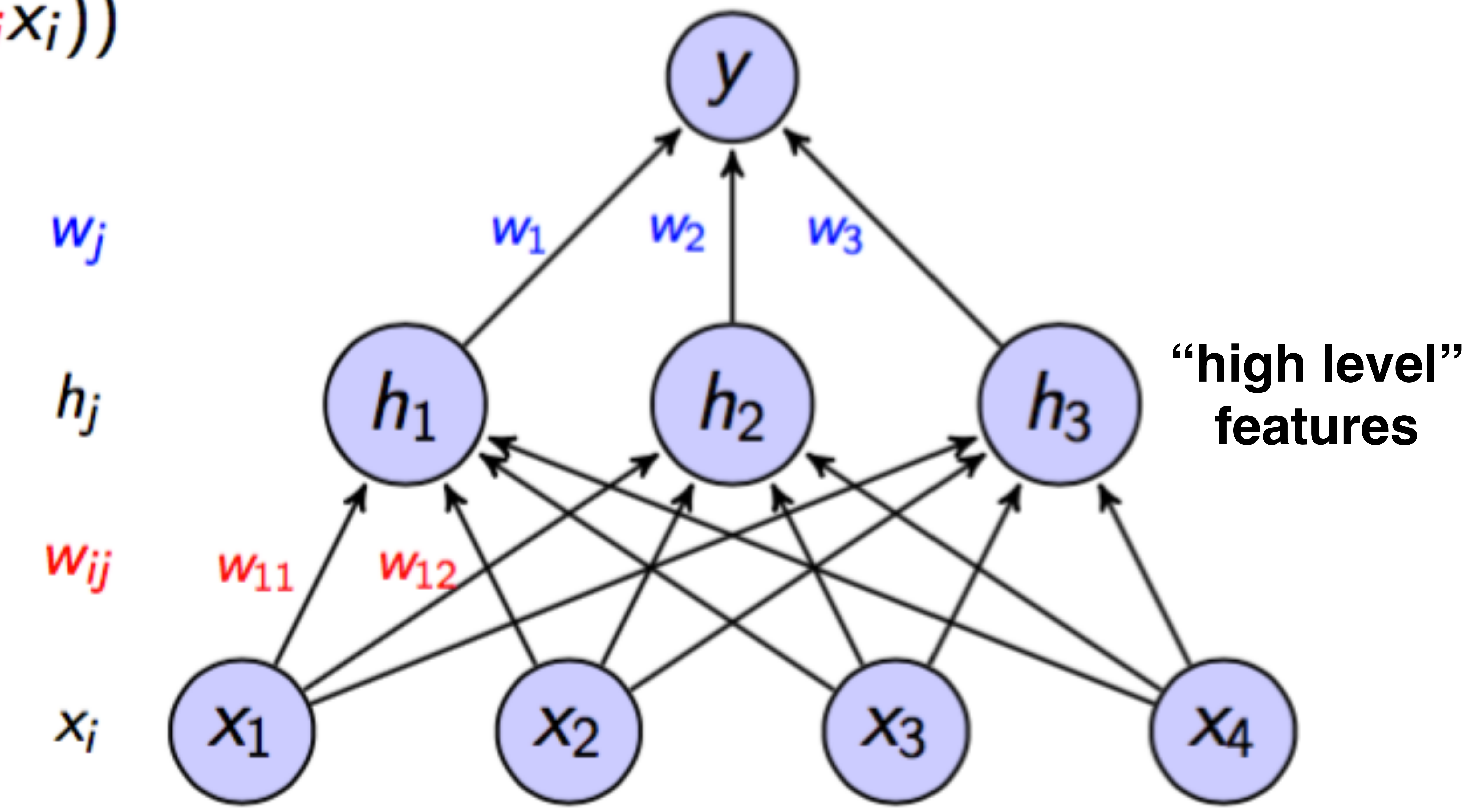
- a.k.a “Feed-Forward NN”

Multi Layer Perceptron (MLP) - a “Deep" NN

- Model the classifier as:

$$f(x) = \sigma(\textstyle\sum_j w_j \cdot h_j) = \sigma(\textstyle\sum_j w_j \cdot \sigma(\textstyle\sum_i w_{ij} x_i))$$

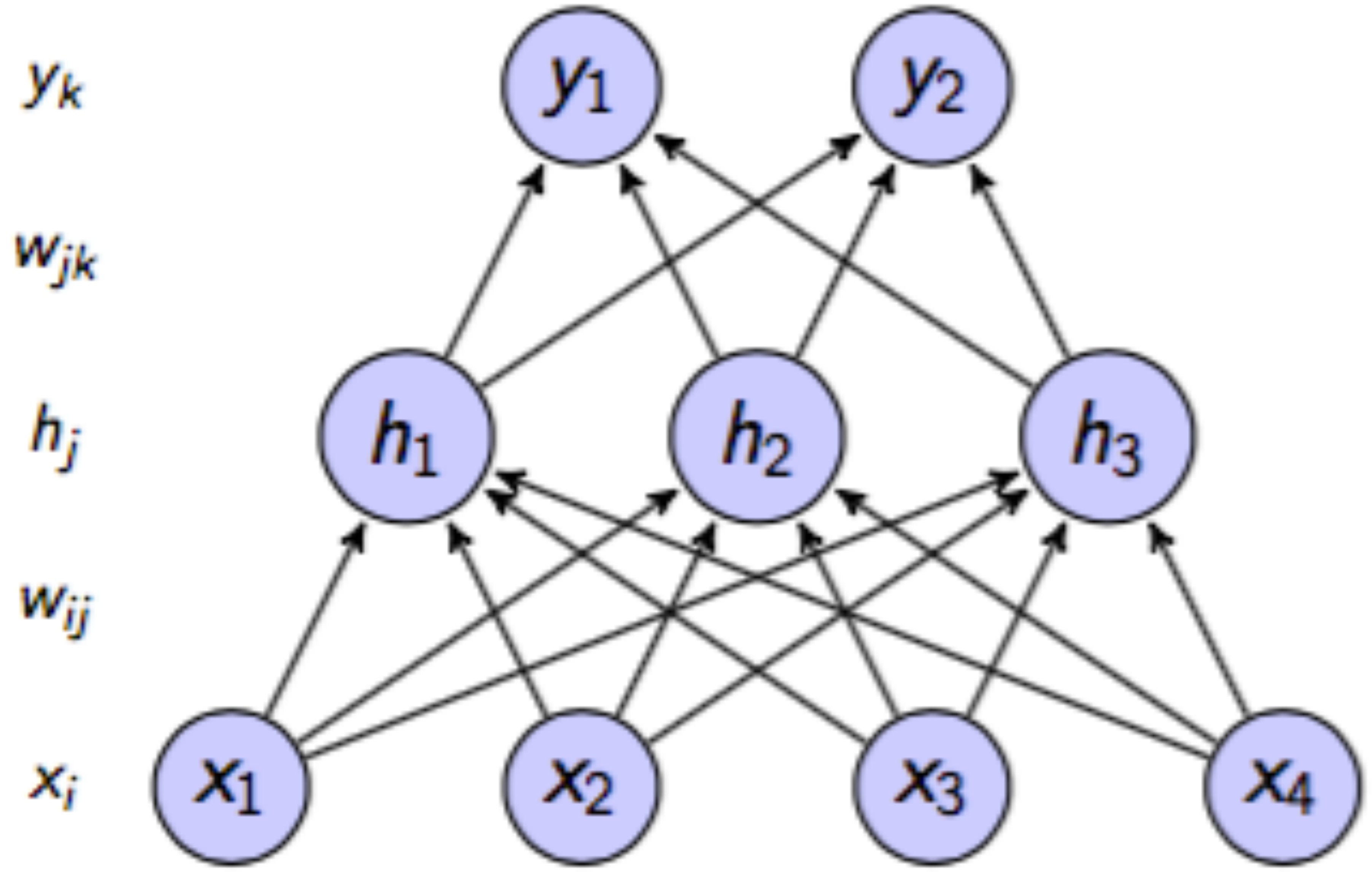
- Can be seen as multilayer logistic regression
- a.k.a “Feed-Forward NN”
- The inputs to the final classifier are learned (“representation learning”)



Training an MLP with Back-Propagation

Training an MLP with Back-Propagation

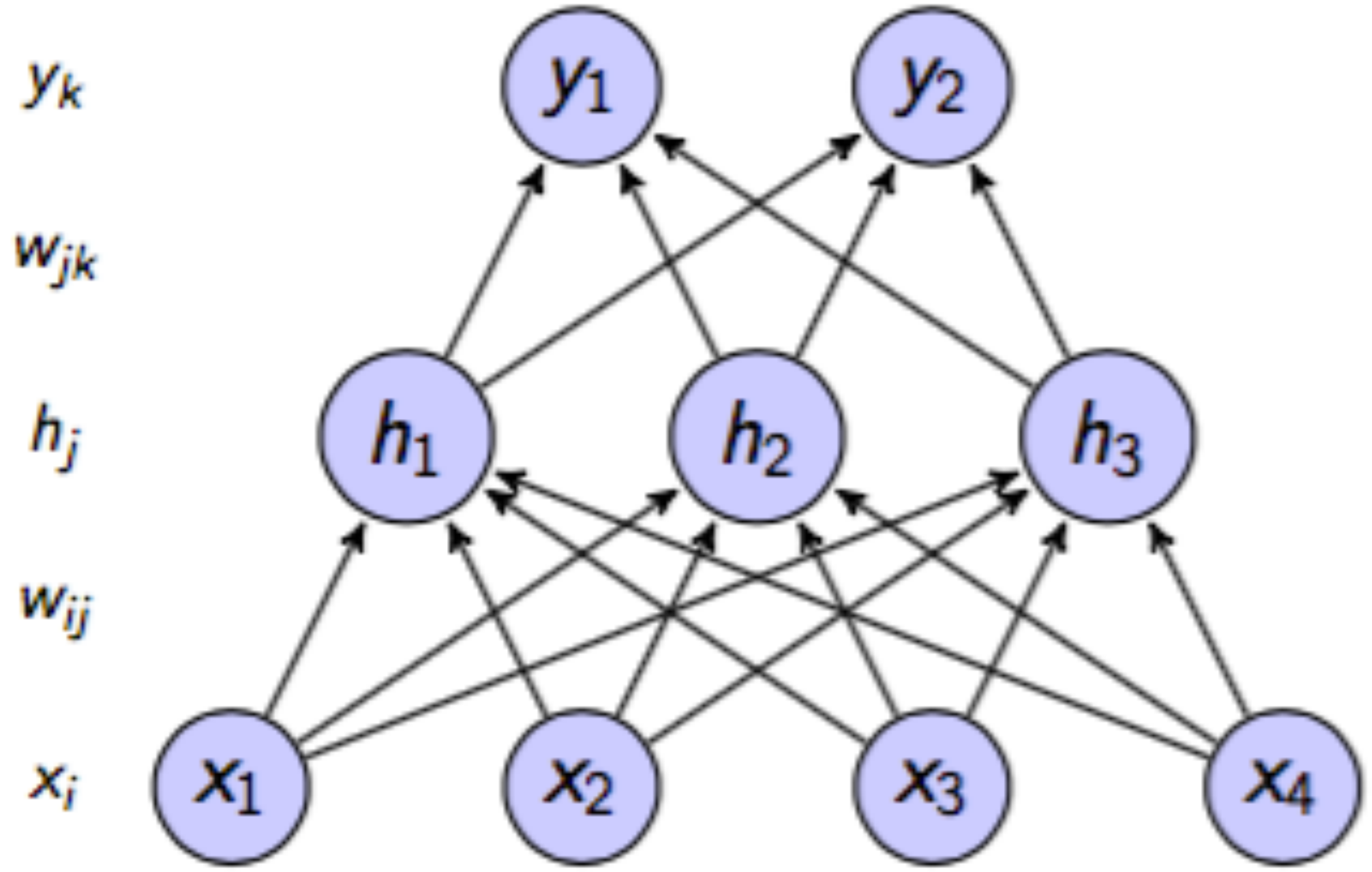
- Assume k=2 outputs per input:



Training an MLP with Back-Propagation

- Assume k=2 outputs per input:
- Define the loss-function per example:

$$Loss = \sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2$$



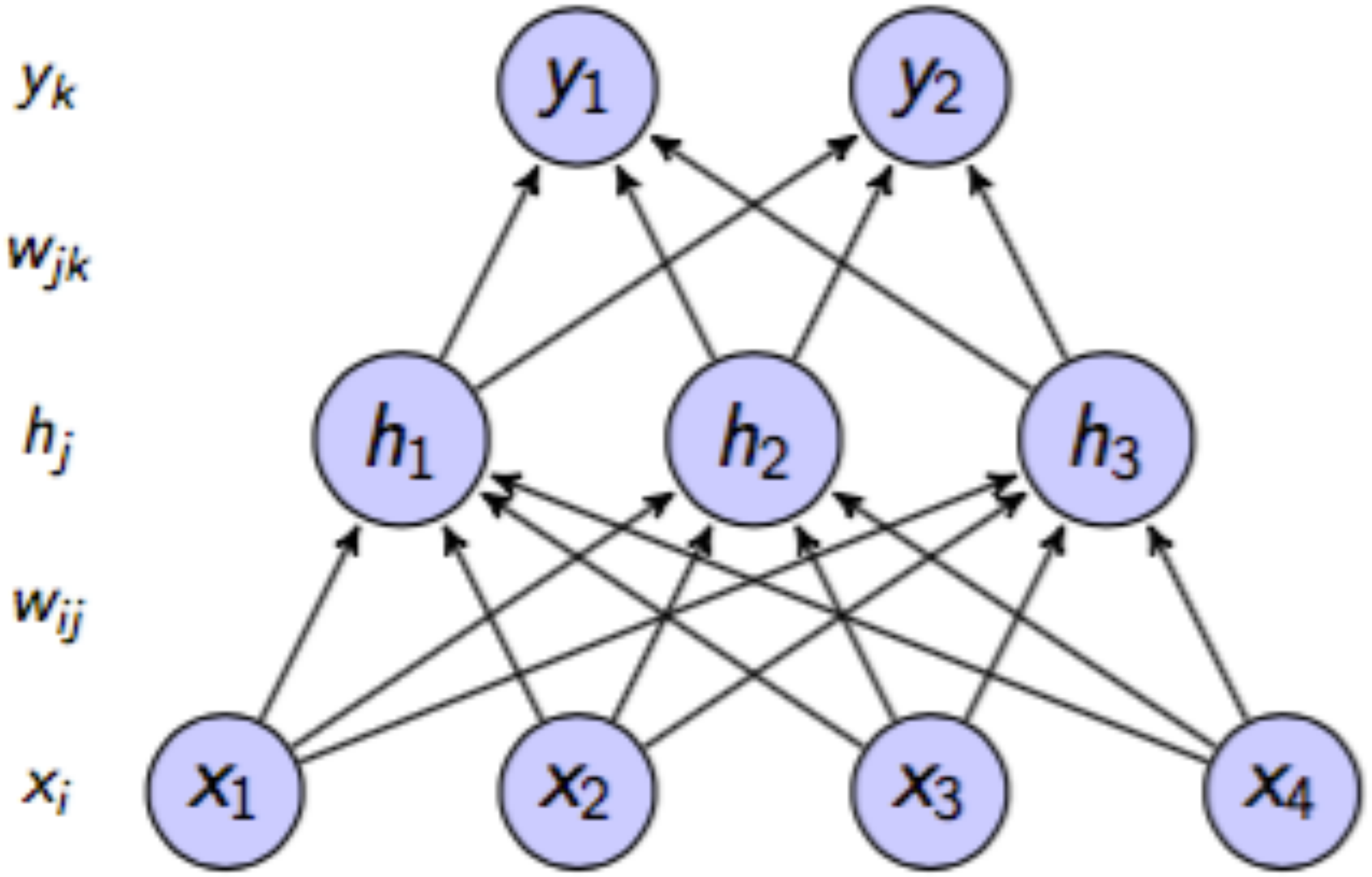
Training an MLP with Back-Propagation

- Assume k=2 outputs per input:
- Define the loss-function per example:

$$Loss = \sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2$$

- Compute the gradient w.r.t. the last layer:

$$\frac{\partial Loss}{\partial w_{jk}} = \delta_k h_j \text{ where } \delta_k = [\sigma(in_k) - y_k]\,\sigma'(in_k)$$



Training an MLP with Back-Propagation

- Assume k=2 outputs per input:
- Define the loss-function per example:

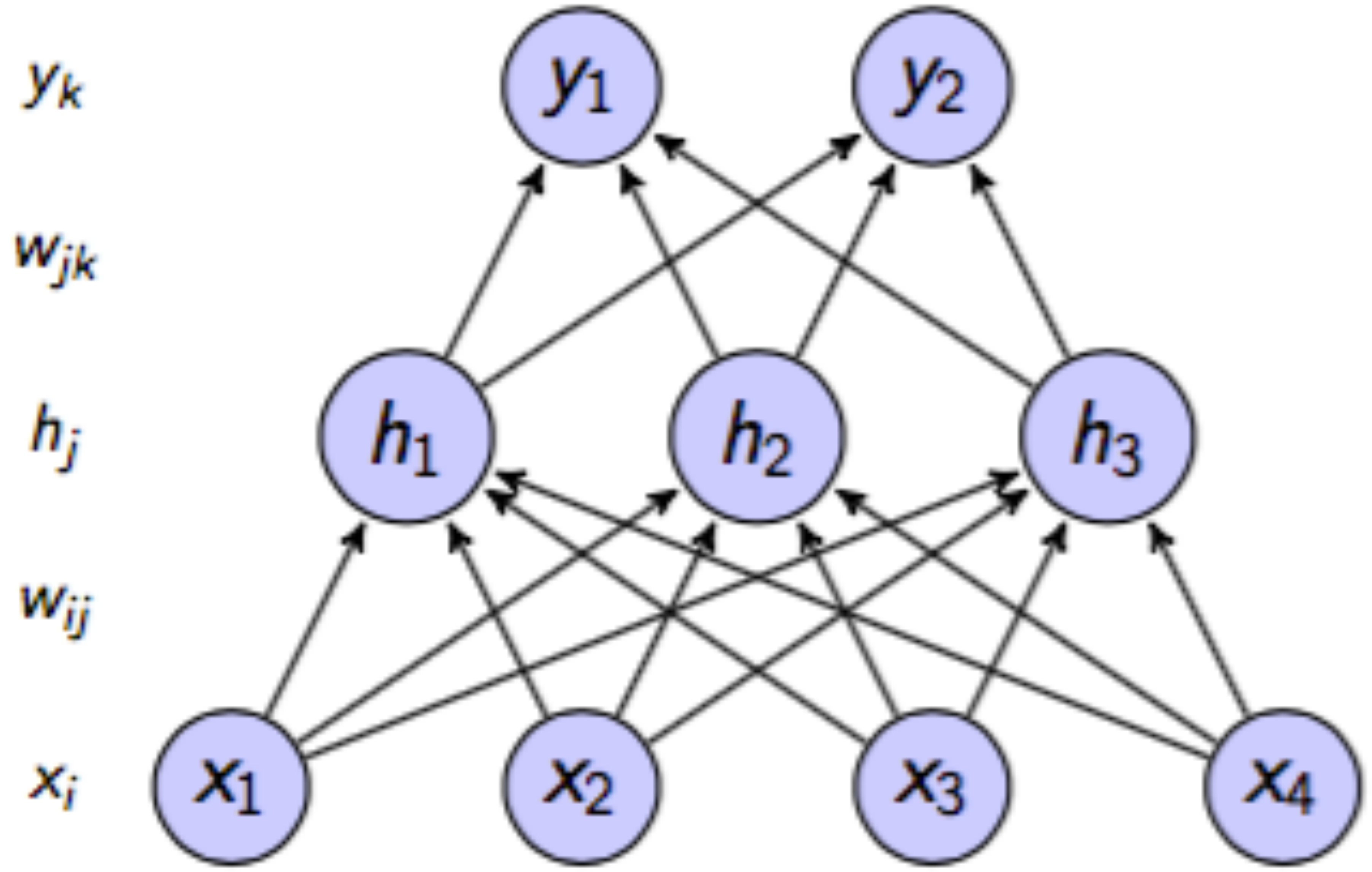
$$Loss = \sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2$$

- Compute the gradient w.r.t. the last layer:

$$\frac{\partial Loss}{\partial w_{jk}} = \delta_k h_j \text{ where } \delta_k = \left[\sigma(in_k) - y_k\right]\sigma'(in_k)$$

- Compute the gradient w.r.t. the first layer:

$$\frac{\partial Loss}{\partial w_{ij}} = \delta_j x_i \text{ where } \delta_j = \left[\sum_k \delta_k w_{jk}\right]\sigma'(in_j)$$



Training an MLP with Back-Propagation

- Assume k=2 outputs per input:
- Define the loss-function per example:

$$Loss = \sum_k \frac{1}{2}\left[\sigma(in_k) - y_k\right]^2$$

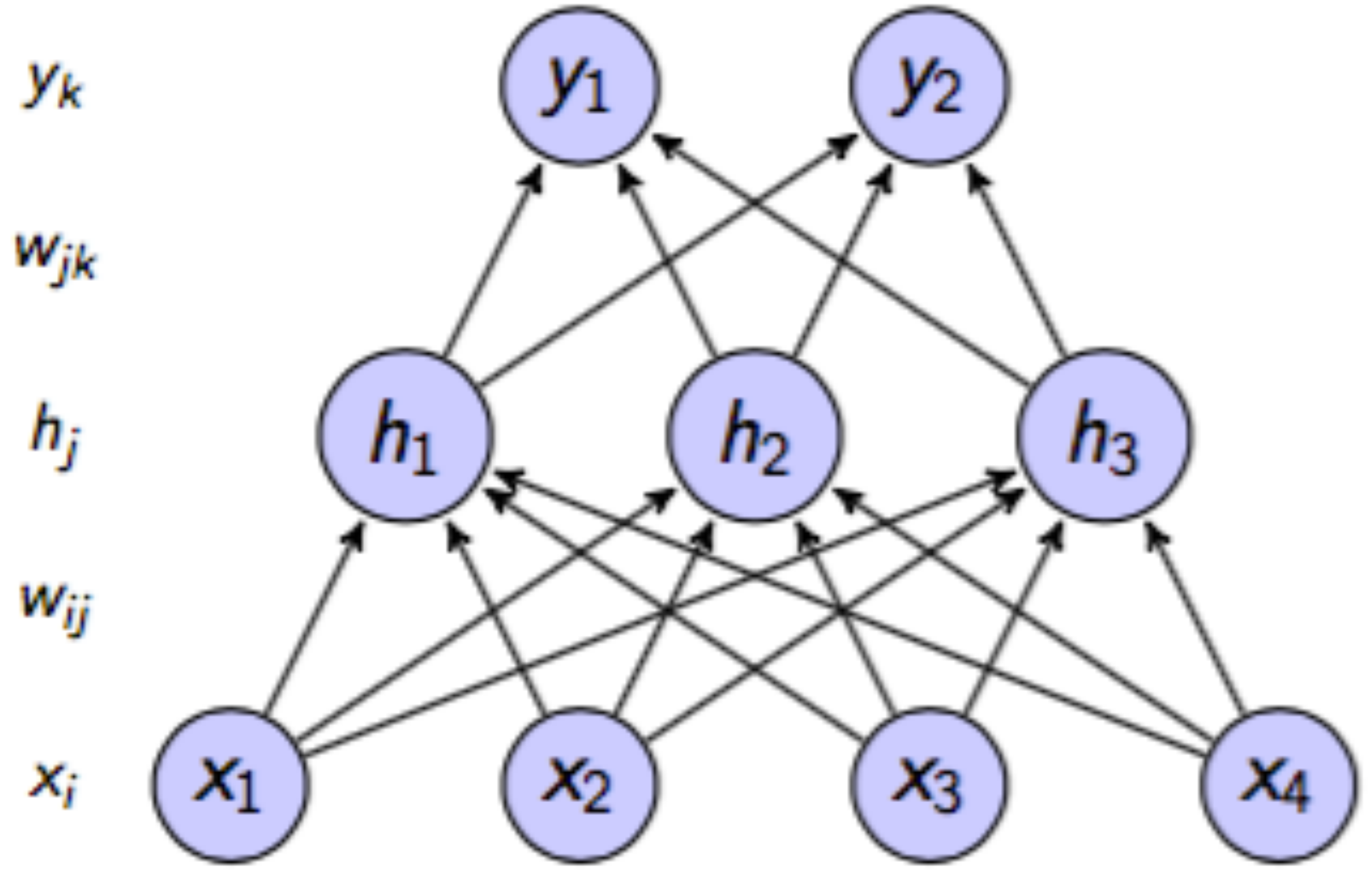
- Compute the gradient w.r.t. the last layer:

$$\frac{\partial Loss}{\partial w_{jk}} = \delta_k h_j \text{ where } \delta_k = [\sigma(in_k) - y_k]\,\sigma'(in_k)$$

- Compute the gradient w.r.t. the first layer:

$$\frac{\partial Loss}{\partial w_{ij}} = \delta_j x_i \text{ where } \delta_j = [\sum_k \delta_k w_{jk}]\,\sigma'(in_j)$$

- Update the weights: $w \leftarrow w - \gamma(\nabla_w Loss)$



Why Deeper is Better?

Why Deeper is Better?

- A deeper architecture is more expressive than a shallow one given same number of nodes (Bishop, 1995)

Why Deeper is Better?

- A deeper architecture is more expressive than a shallow one given same number of nodes (Bishop, 1995)
  - 1-layer nets (log. regression) can only model linear hyperplanes

Why Deeper is Better?

- A deeper architecture is more expressive than a shallow one given same number of nodes (Bishop, 1995)
  - 1-layer nets (log. regression) can only model linear hyperplanes
  - 2-layer nets can model any continuous function (given sufficient parameters)

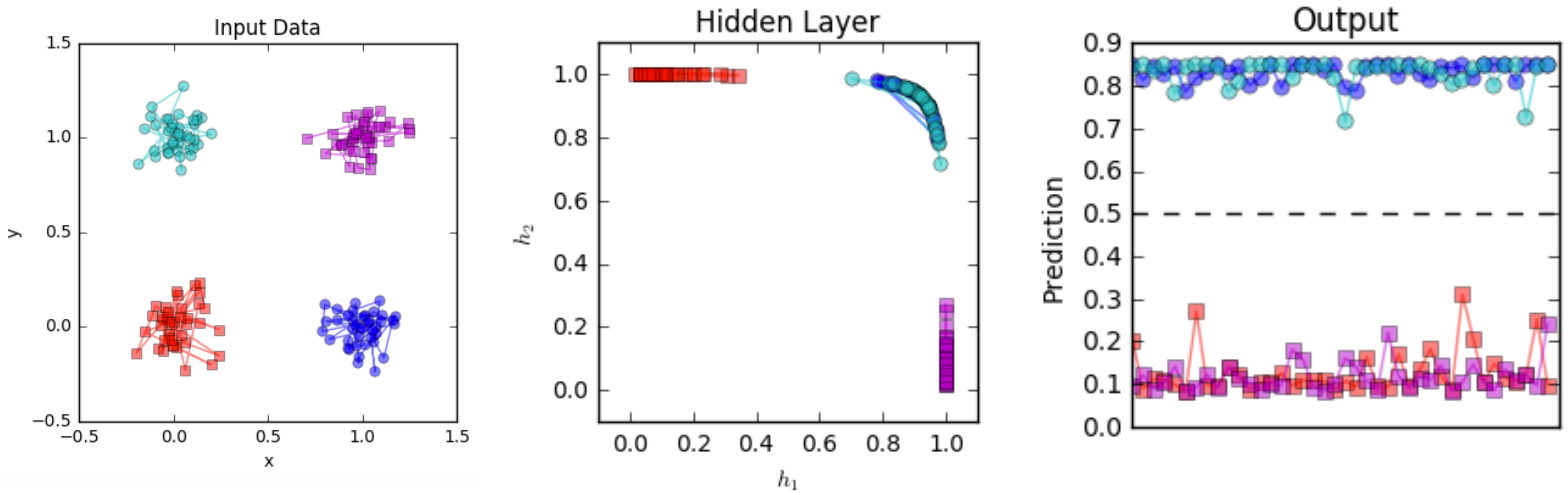
Why Deeper is Better?

- A deeper architecture is more expressive than a shallow one given same number of nodes (Bishop, 1995)
  - 1-layer nets (log. regression) can only model linear hyperplanes
  - 2-layer nets can model any continuous function (given sufficient parameters)
  - >3-layer nets can do so with fewer parameters

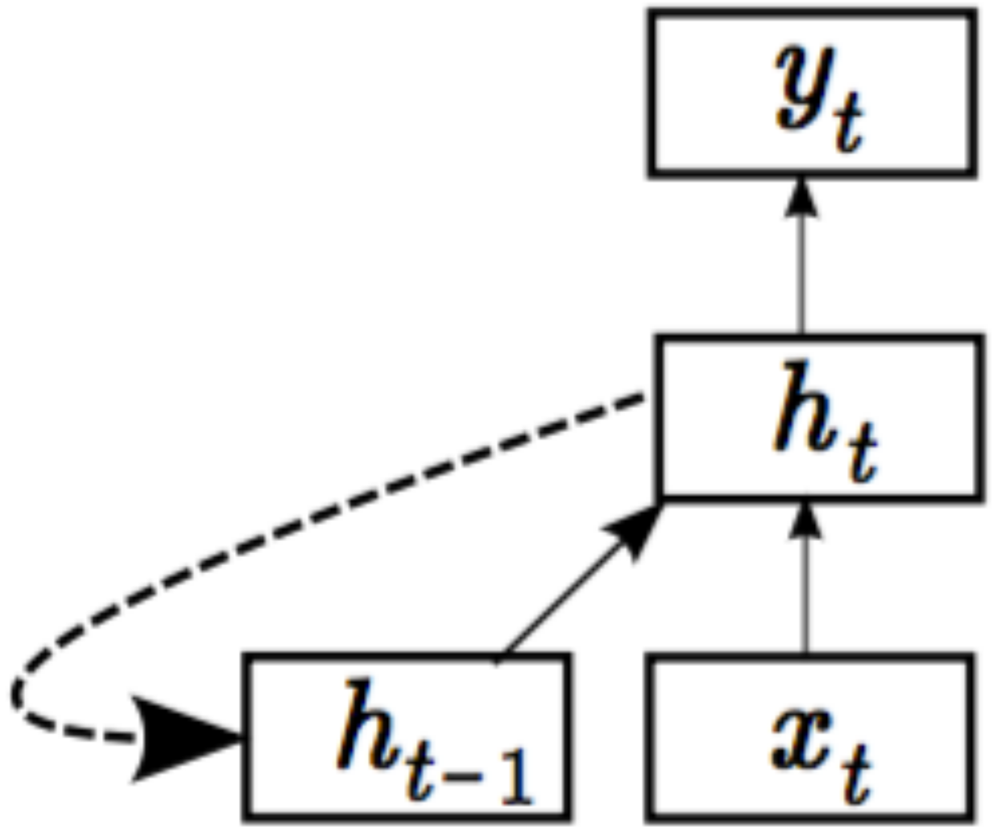
Why Deeper is Better?

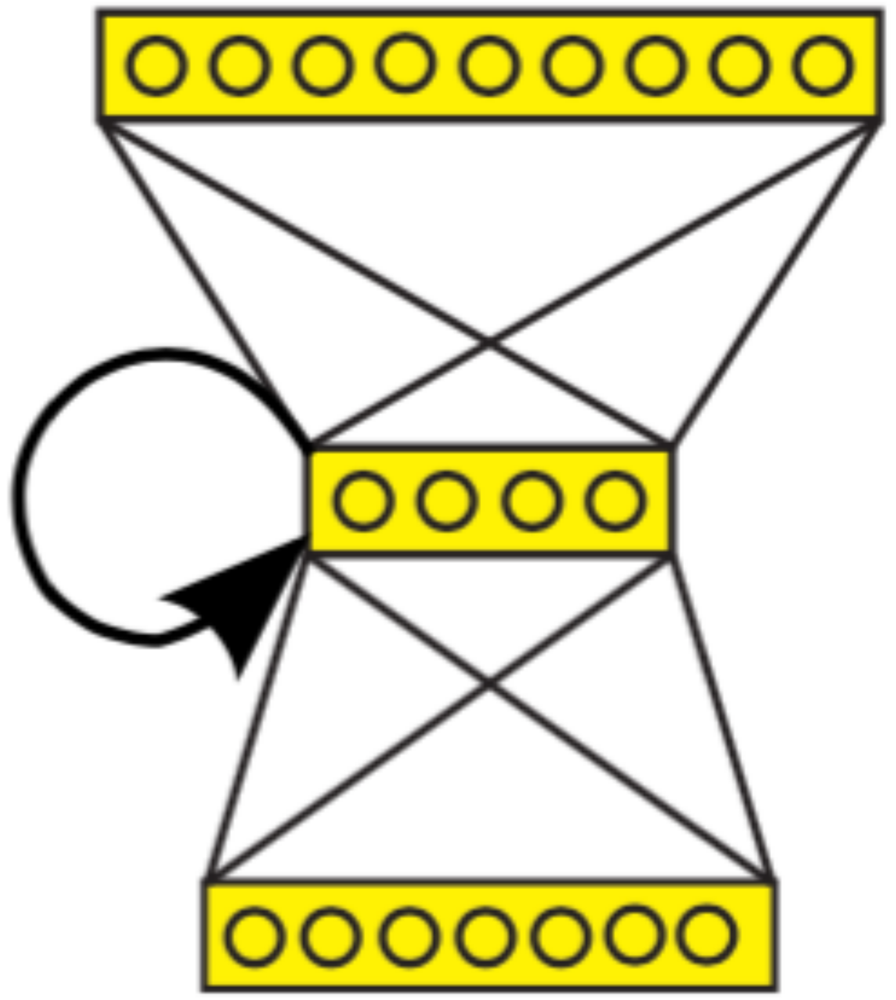
- A deeper architecture is more expressive than a shallow one given same number of nodes (Bishop, 1995)
  - 1-layer nets (log. regression) can only model linear hyperplanes
  - 2-layer nets can model any continuous function (given sufficient parameters)
  - >3-layer nets can do so with fewer parameters

Example - "learning to XOR": $x \oplus y = (x \vee y) \wedge \neg(x \wedge y)$



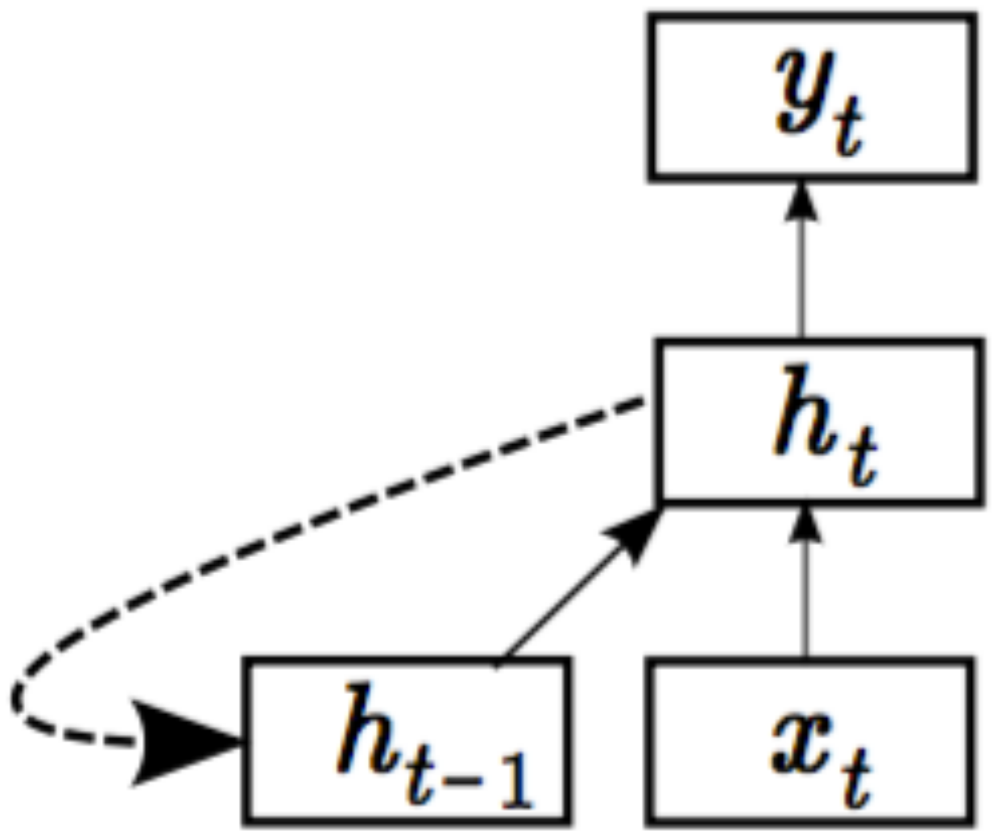
Recurrent Neural Networks (RNNs)

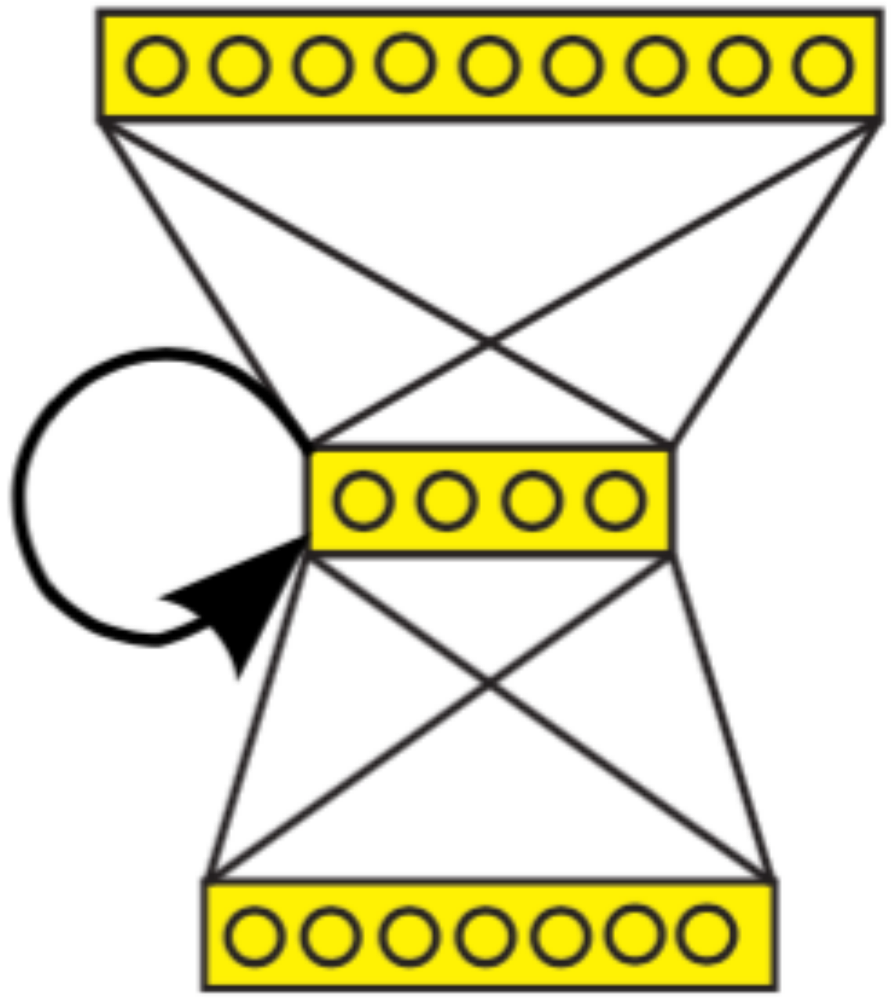




Recurrent Neural Networks (RNNs)

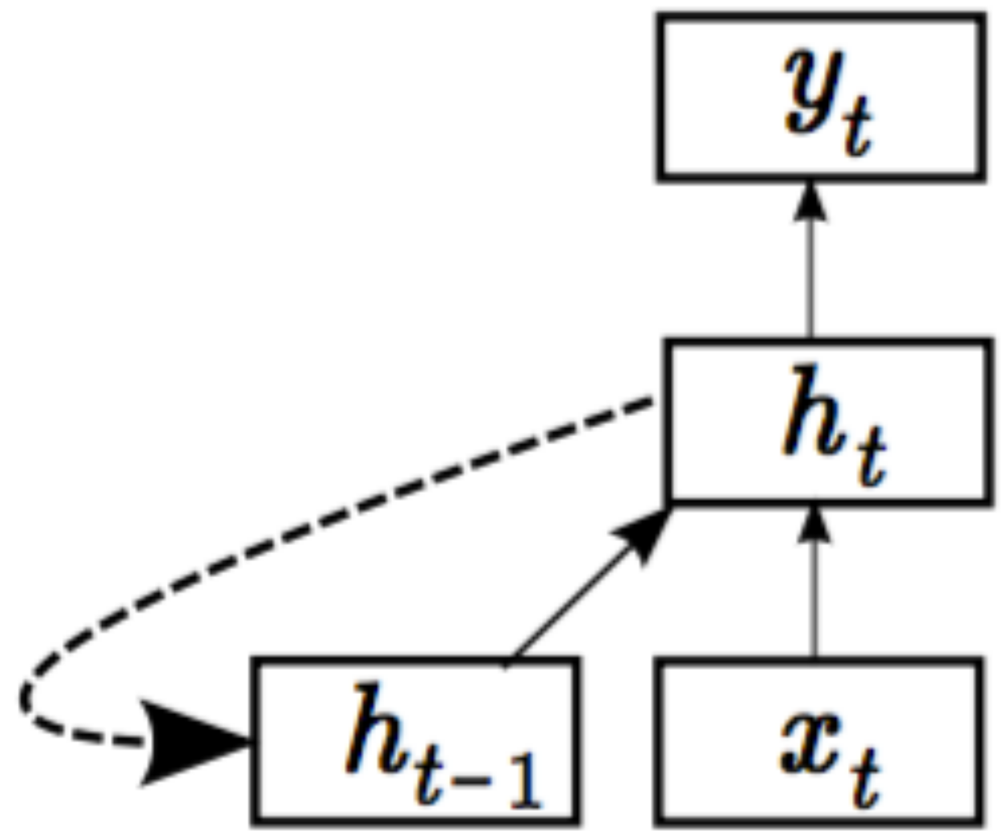
- Enable variable length inputs (sequences)

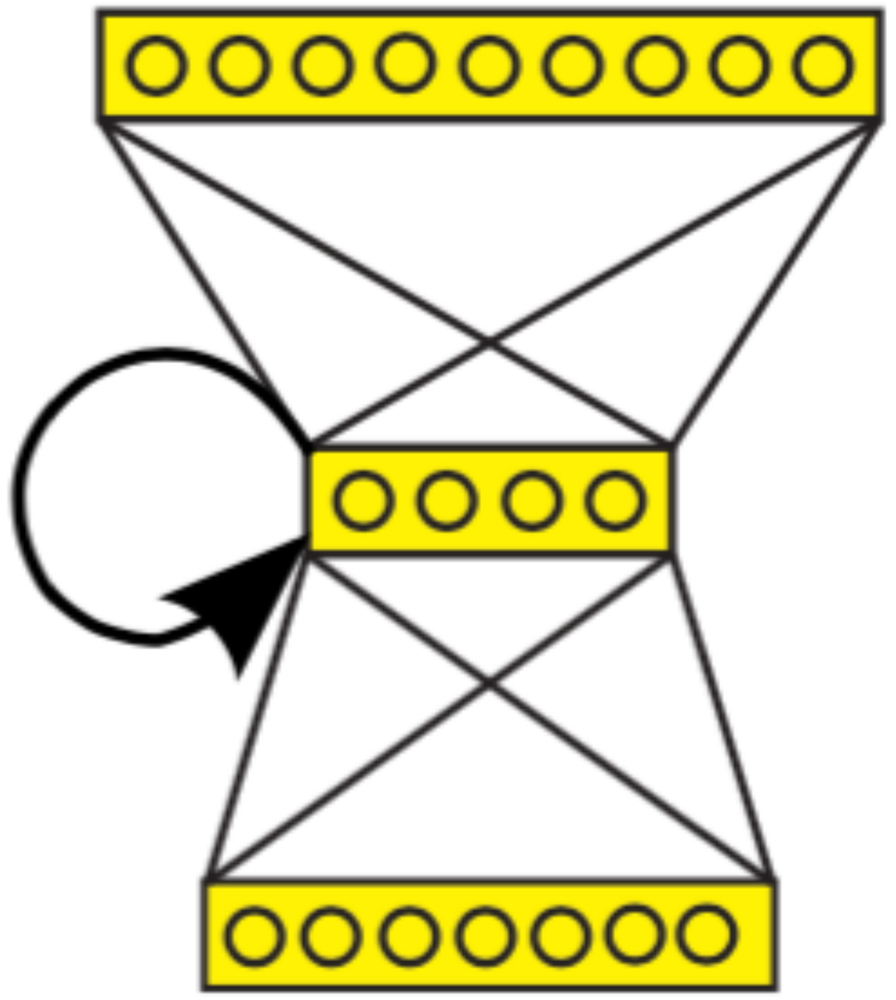




Recurrent Neural Networks (RNNs)

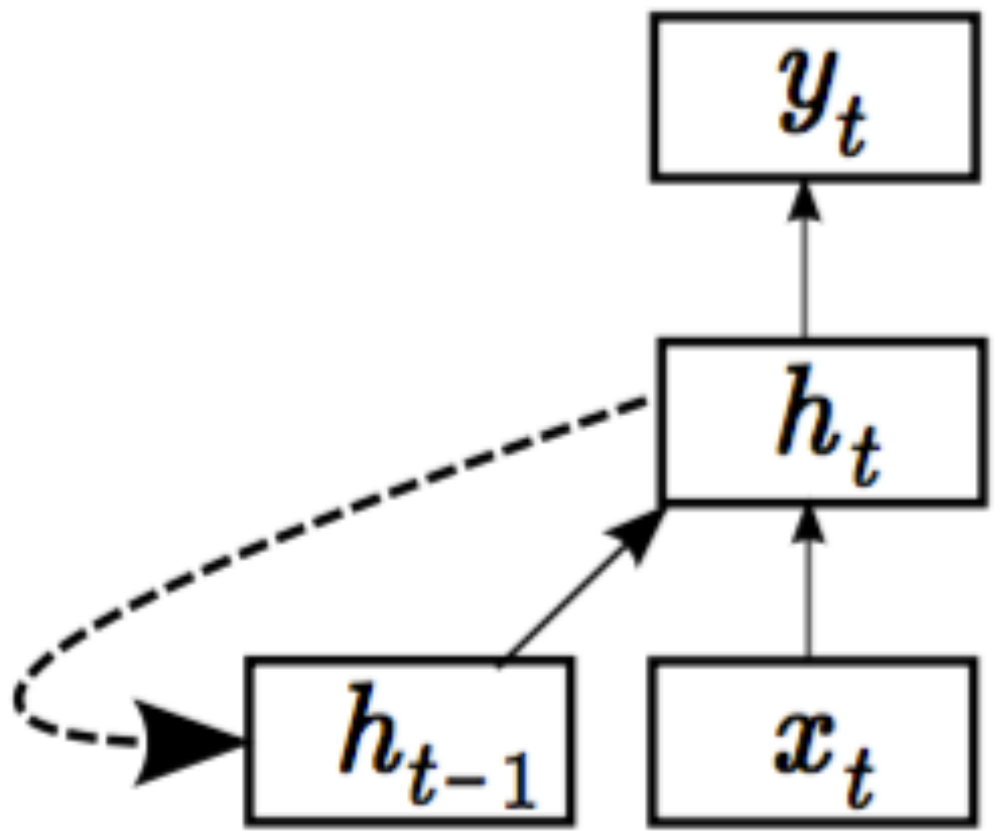
- Enable variable length inputs (sequences)
- Modelling internal structure in the input or output

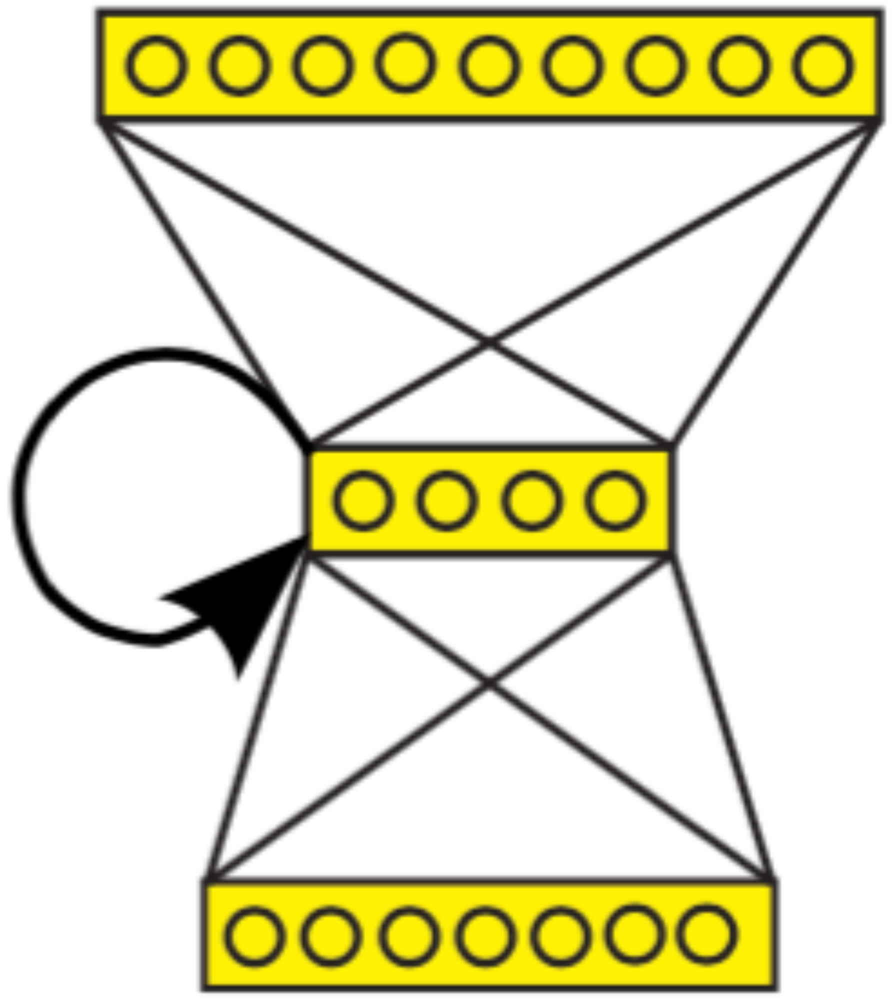




Recurrent Neural Networks (RNNs)

- Enable variable length inputs (sequences)
- Modelling internal structure in the input or output
- Introduce a “memory/context” component to utilize history





Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs)

- “Horizontally deep” architecture

Recurrent Neural Networks (RNNs)

- “Horizontally deep” architecture
- Recurrence equations:

Recurrent Neural Networks (RNNs)

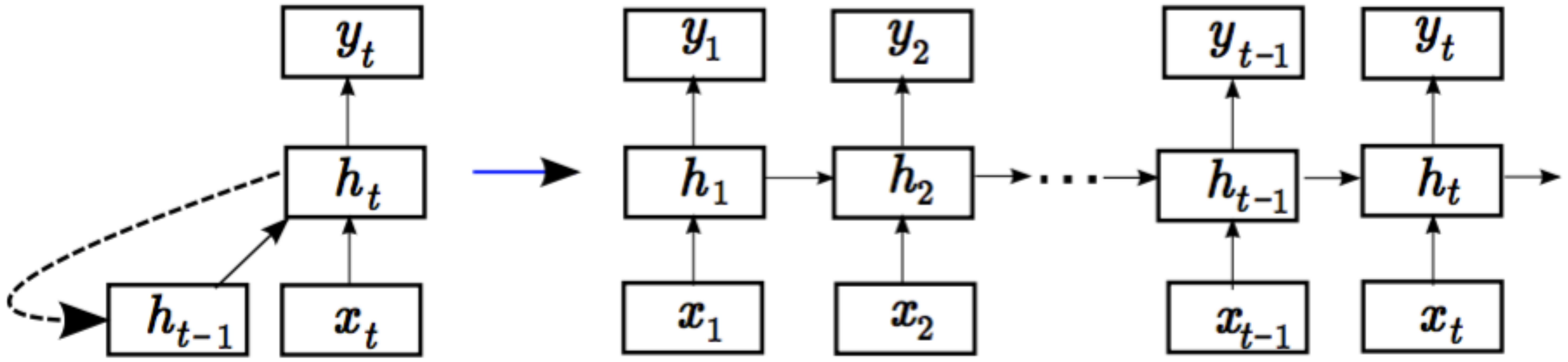
- “Horizontally deep” architecture
- Recurrence equations:
  - Transition function: $h_t = H(h_{t-1}, x_t) = tanh(Wx_{t-1} + Uh_{t-1} + b)$

Recurrent Neural Networks (RNNs)

- “Horizontally deep” architecture
- Recurrence equations:
  - Transition function: $h_t = H(h_{t-1}, x_t) = tanh(Wx_{t-1} + Uh_{t-1} + b)$
  - Output function: $y_t = Y(h_t)$, usually a sigmoid or softmax

Recurrent Neural Networks (RNNs)

- “Horizontally deep” architecture
- Recurrence equations:
  - Transition function: $h_t = H(h_{t-1}, x_t) = tanh(Wx_{t-1} + Uh_{t-1} + b)$
  - Output function: $y_t = Y(h_t)$, usually a sigmoid or softmax



The Softmax Function and Negative Log Likelihood

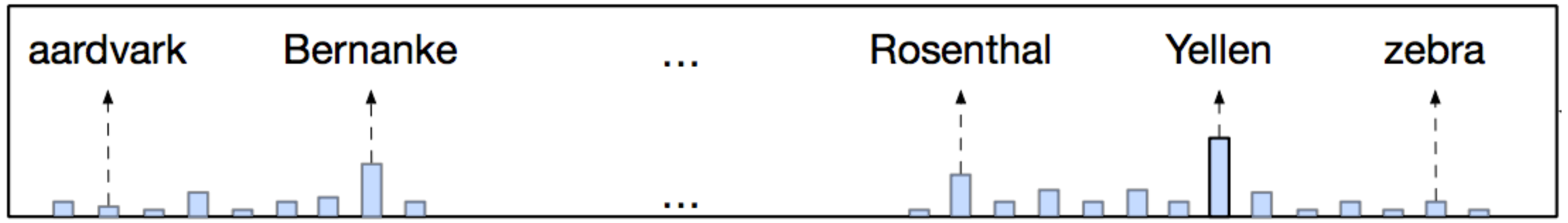
The Softmax Function and Negative Log Likelihood

- Enables to output a **probability distribution** over **k possible classes** (words, in our case)

The Softmax Function and Negative Log Likelihood

- Enables to output a **probability distribution** over **k possible classes** (words, in our case)
- y_i (the value of the network output vector in position i) is expected to hold the log-likelihood (probability) of a specific class (in our case, word):

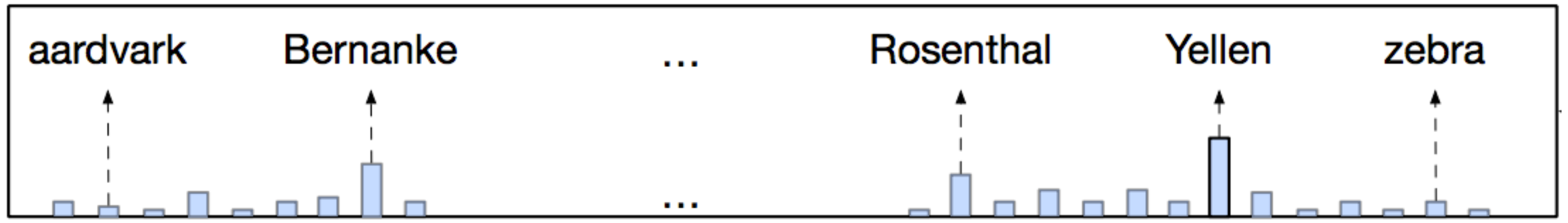
$$p(x=i) = \frac{e^{y_i}}{\sum_{j=1}^{k} e^{y_j}}$$



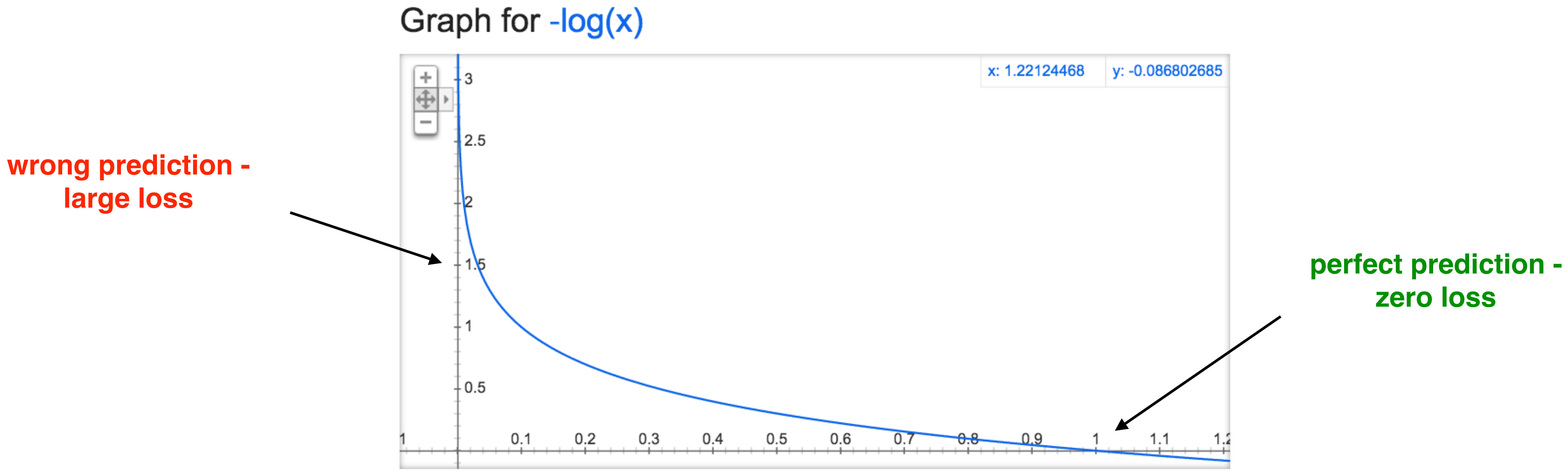
The Softmax Function and Negative Log Likelihood

- Enables to output a **probability distribution** over **k possible classes** (words, in our case)
- y_i (the value of the network output vector in position i) is expected to hold the log-likelihood (probability) of a specific class (in our case, word):

$$p(x=i) = \frac{e^{y_i}}{\sum_{j=1}^{k} e^{y_j}}$$



- The loss function is usually the **sum of negative log softmax** values for the **correct sequence**



Training (RNN's) with Backpropagation Through Time

- As usual, define a loss function (per sample, through time $t = 1, 2, ..., T$):

$$Loss = J(\Theta, x) = -\sum_{t=1}^{T} J_t(\Theta, x_t)$$

- Compute the gradient w.r.t. parameters Θ, starting at $t = T$:

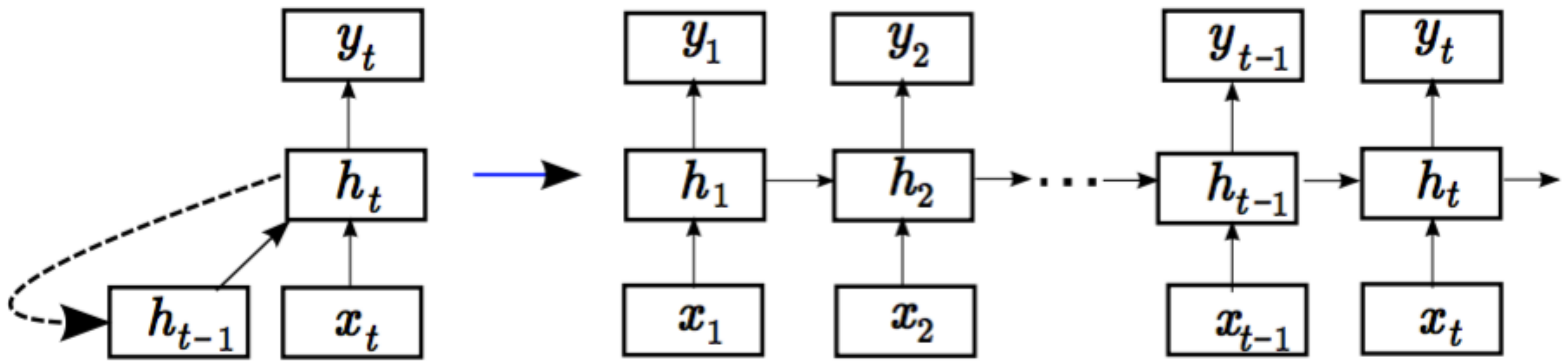
$$\nabla\Theta = \frac{\partial J_t}{\partial \Theta}$$

- Backpropagate through time - sum and repeat for $t - 1$, until $t = 1$:

$$\nabla\Theta = \nabla\Theta + \frac{\partial J_t}{\partial \Theta}$$

- Eventually, update the weights:

$$\Theta = \gamma \nabla\Theta$$



Vanishing Gradients in Vanilla RNNs

- If we dive deeper into the gradients of the RNN loss function, for example:

$$\frac{\partial J_{t+n}}{\partial h_t} = \frac{\partial J_{t+n}}{\partial g} \frac{\partial g}{\partial h_{t+N}} \frac{\partial h_{t+N}}{\partial h_{t+N-1}} \cdots \frac{\partial h_{t+1}}{\partial h_t}$$

- Given $h_t = tanh(a), a = W_{x_{t-1}} + Uh_{t-1}, + b$ we get that:

$$\frac{\partial h_{t+1}}{\partial h_t} = U^T \frac{\partial tanh(a)}{\partial a}$$

- And Eventually:

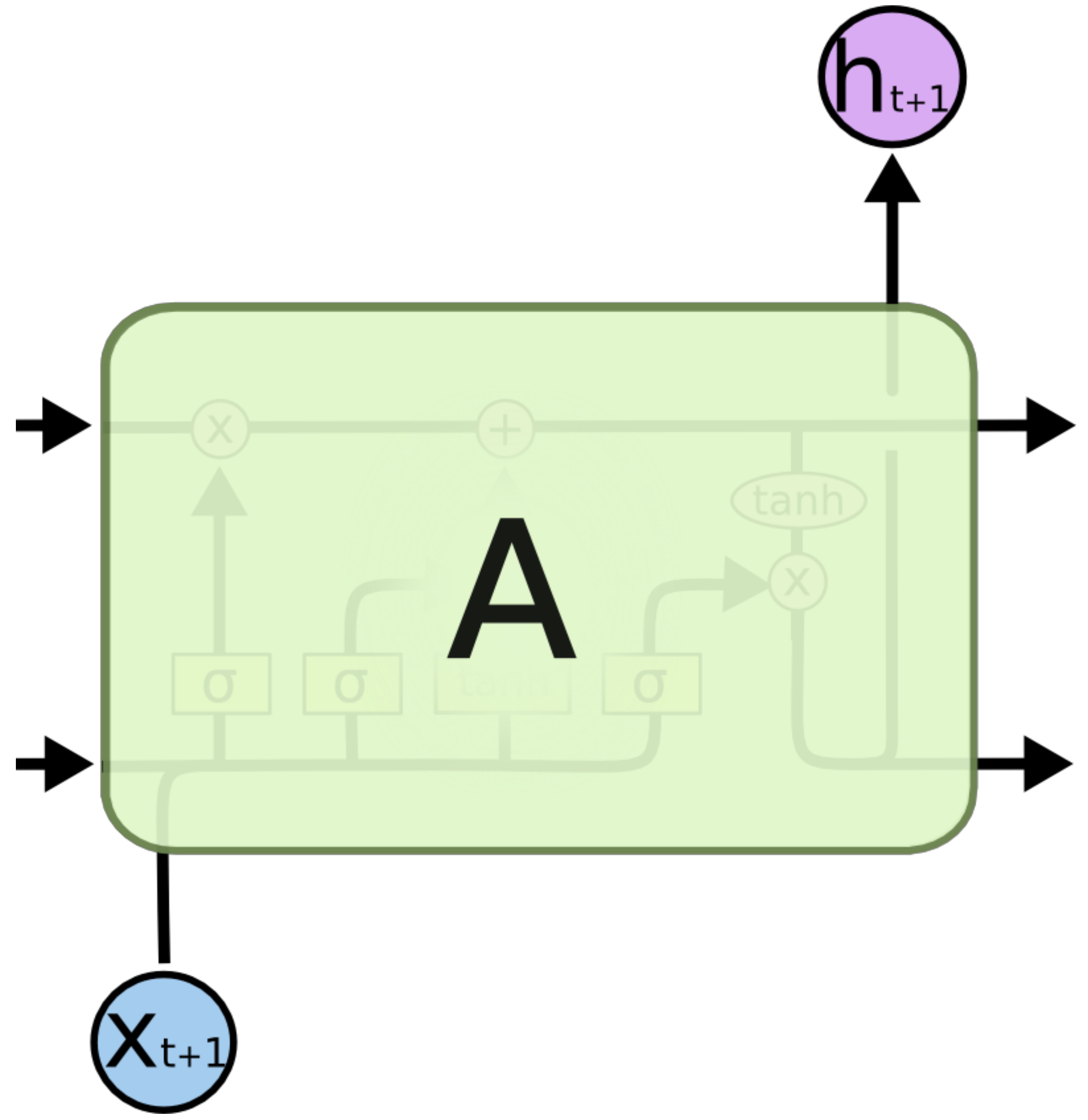
$$\frac{\partial J_{t+n}}{\partial h_t} = \frac{\partial J_{t+n}}{\partial g} \frac{\partial g}{\partial h_{t+N}} \prod_{n=1}^{N} U^T diag\left(\frac{\partial tanh(a_{t+n})}{\partial a_{t+n}}\right)$$

- This easily makes the gradients vanish (get close to 0) so that no learning takes place, as noted in Bengio et al (94’):

$$\prod_{n=1}^{N} U^T diag\left(\frac{\partial tanh(a_{t+n})}{\partial a_{t+n}}\right) \rightarrow 0$$

Vanishing gradients, LSTM’s and GRU’s

- In order to cope with the vanishing gradients problem in RNN’s, more complex recurrent architectures emerged:
  - Long Short-Term Memory (Hochreiter & Schmidhuber,1999)
  - Gated Recurrent Unit (Cho et al, 2014)
- These architectures introduce additive terms that relax the vanishing gradient problem
- Most of the recent RNN works utilize such architectures



Figures by Chris Olah

LSTM Walkthrough in 4 Steps

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output
- Has gates controlling the extent to which:

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output
- Has gates controlling the extent to which:
  - New content should be consumed (input gate)

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output
- Has gates controlling the extent to which:
  - New content should be consumed (input gate)
  - Old content should be erased (forget gate)

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output
- Has gates controlling the extent to which:
  - New content should be consumed (input gate)
  - Old content should be erased (forget gate)
  - Current content should be exposed (output gate). More formally:

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output
- Has gates controlling the extent to which:
  - New content should be consumed (input gate)
  - Old content should be erased (forget gate)
  - Current content should be exposed (output gate). More formally:

I

compute current input, forget, output gates and memory cell <u>update</u>

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \hat{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \cdot [h_{t-1}, x_t]$$

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output
- Has gates controlling the extent to which:
  - New content should be consumed (input gate)
  - Old content should be erased (forget gate)
  - Current content should be exposed (output gate). More formally:

I **compute current input, forget, output gates and memory cell <u>update</u>**

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \hat{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \cdot [h_{t-1}, x_t]$$

compute current memory cell using input and forget gates **II**

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output
- Has gates controlling the extent to which:
  - New content should be consumed (input gate)
  - Old content should be erased (forget gate)
  - Current content should be exposed (output gate). More formally:

I compute current input, forget, output gates and memory cell <u>update</u>

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \hat{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \cdot [h_{t-1}, x_t]$$

II compute current memory cell using input and forget gates

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

III compute current hidden state using output gate and memory cell

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

LSTM Walkthrough in 4 Steps

- Processes a variable length input sequence: $x = (x_1, x_2, \cdots, x_n)$
- At any time step, holds a memory cell c_t and a hidden state h_t used for predicting an output
- Has gates controlling the extent to which:
  - New content should be consumed (input gate)
  - Old content should be erased (forget gate)
  - Current content should be exposed (output gate). More formally:

I compute current input, forget, output gates and memory cell <u>update</u>

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \hat{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \cdot [h_{t-1}, x_t]$$

II compute current memory cell using input and forget gates

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

III compute current hidden state using output gate and memory cell

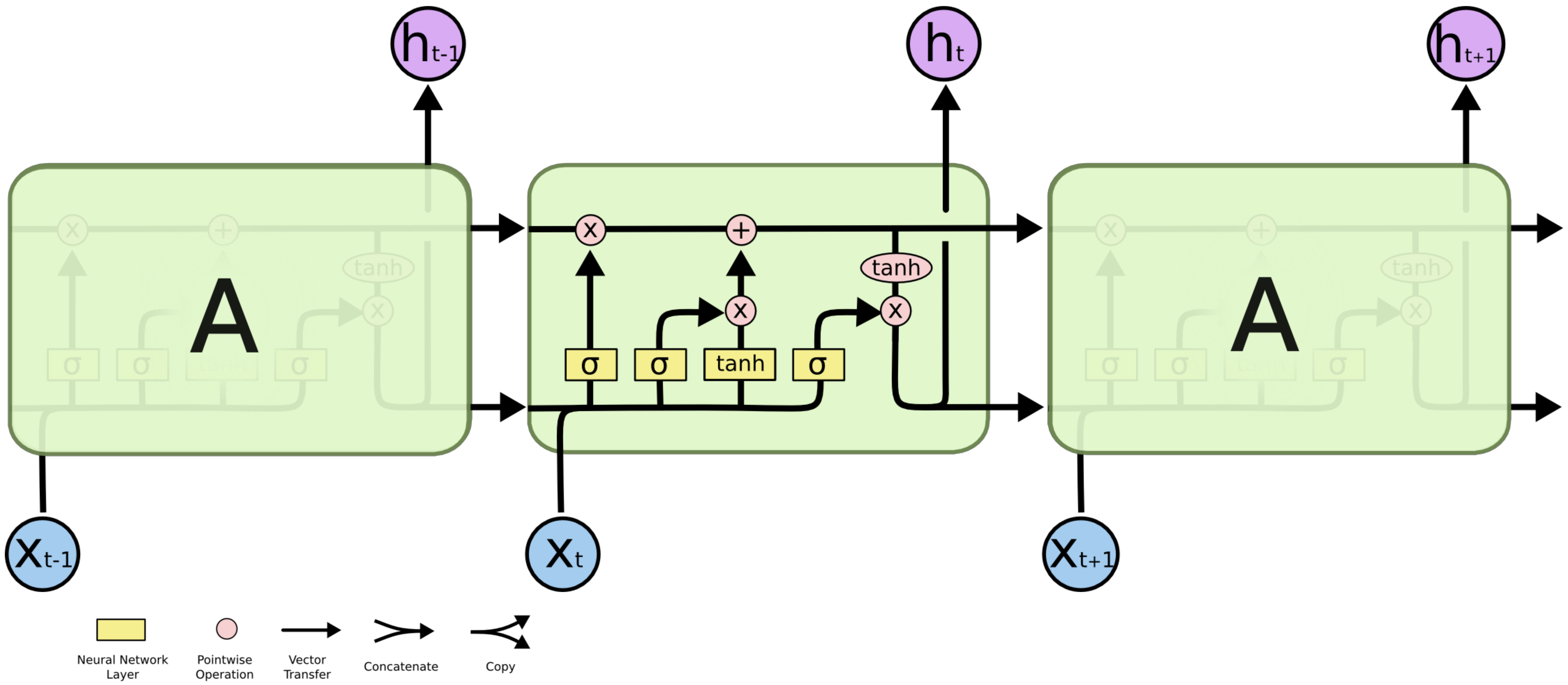
$$h_t = o_t \odot \tanh(c_t)$$

IV compute current output probabilities for prediction by using softmax over the hidden state

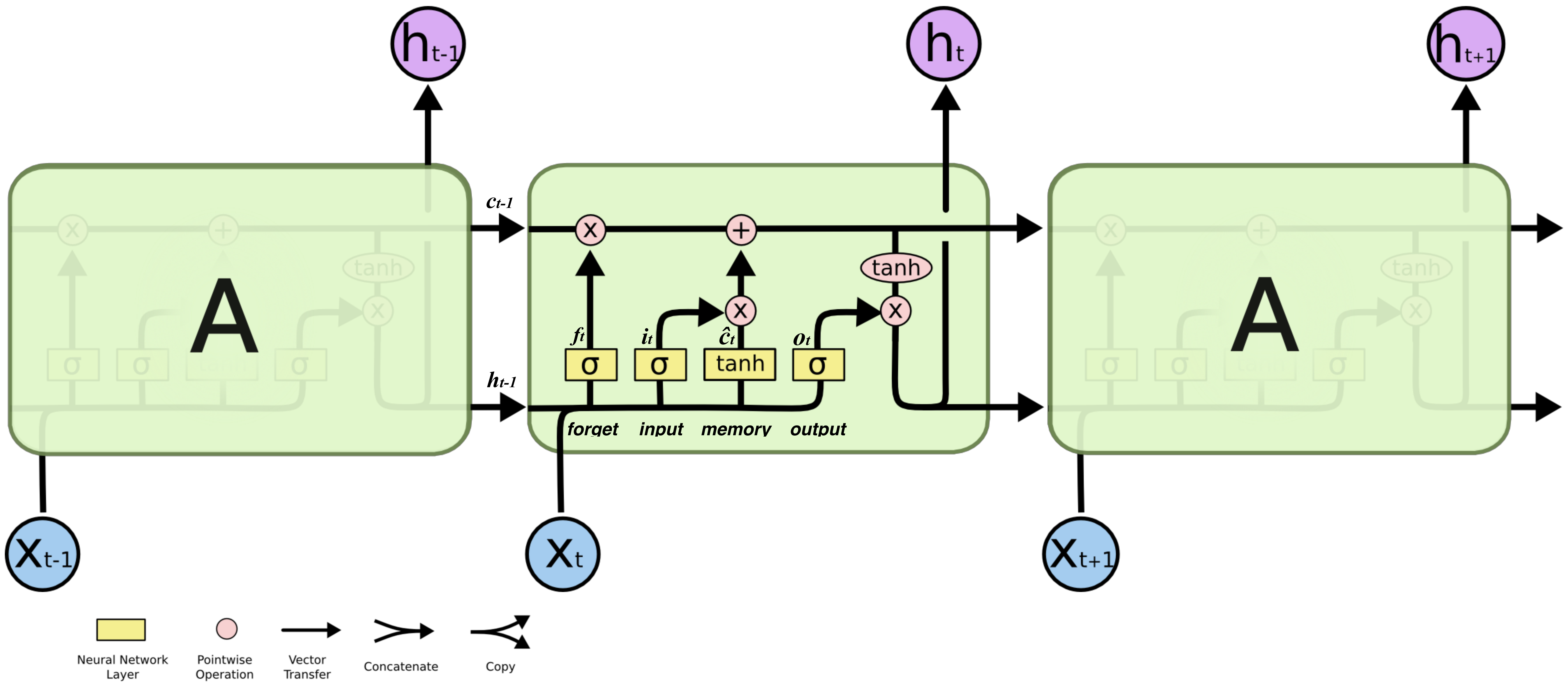
$$p(x_{t+1} = w | x_1, \cdots, x_t) = \exp(u(w, h_t))/Z$$

$$Z = \sum_{w' \in V} \exp(u(w', h_t))$$

LSTM walkthrough in 4 steps



LSTM walkthrough in 4 steps

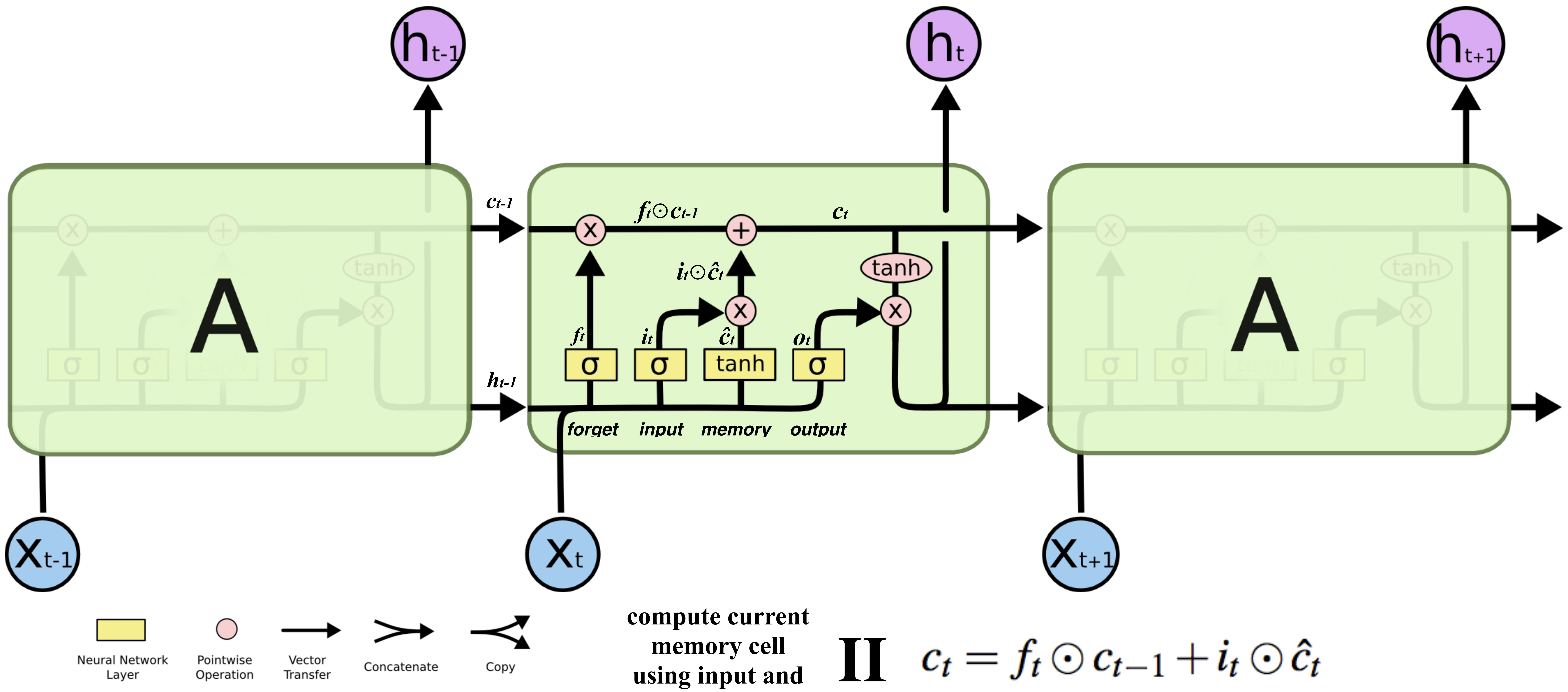


I

compute current input, forget, output, memory gate values

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \hat{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \cdot [h_{t-1}, x_t]$$

LSTM walkthrough in 4 steps



compute current memory cell using input and forget gates

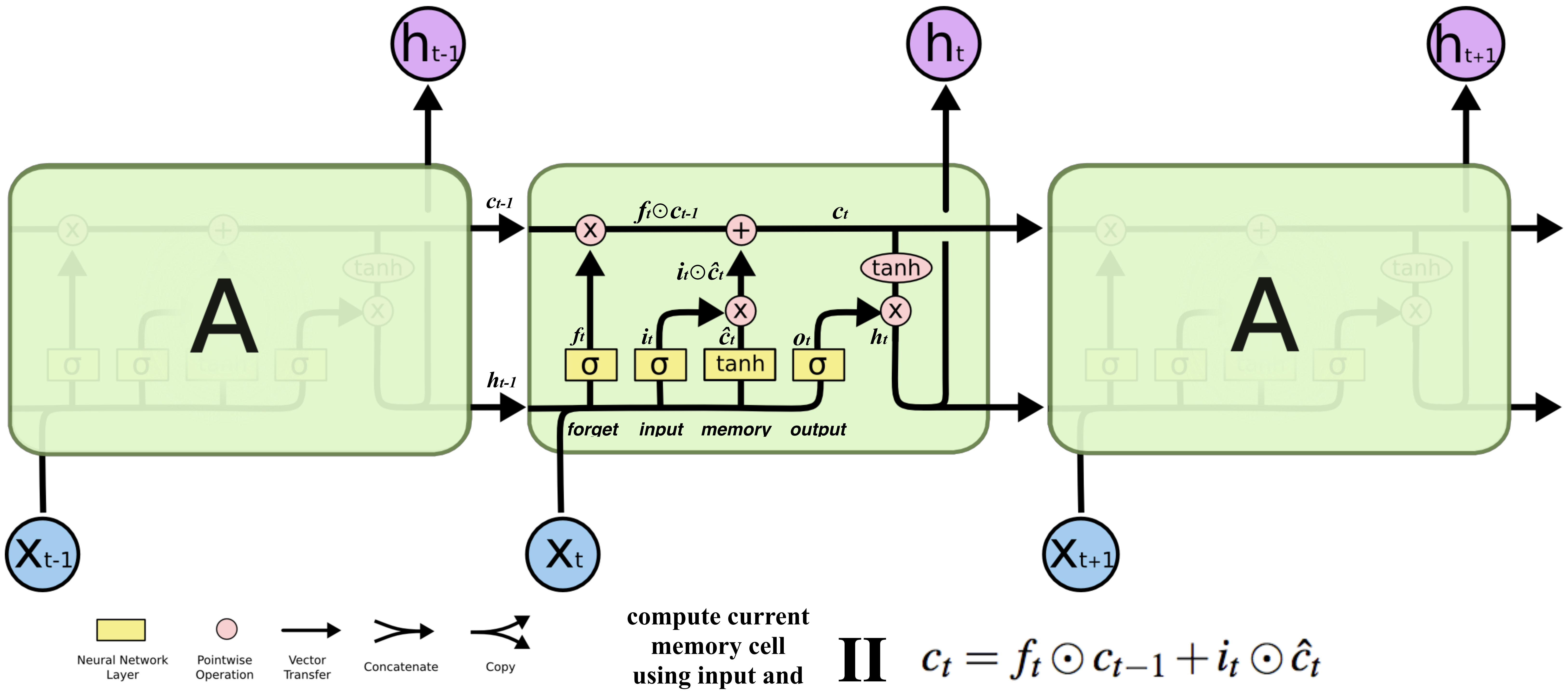
II $c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$

I

compute current input, forget, output, memory gate values

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \hat{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \cdot [h_{t-1}, x_t]$$

LSTM walkthrough in 4 steps



compute current memory cell using input and forget gates

II $$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

III $$h_t = o_t \odot \tanh(c_t)$$

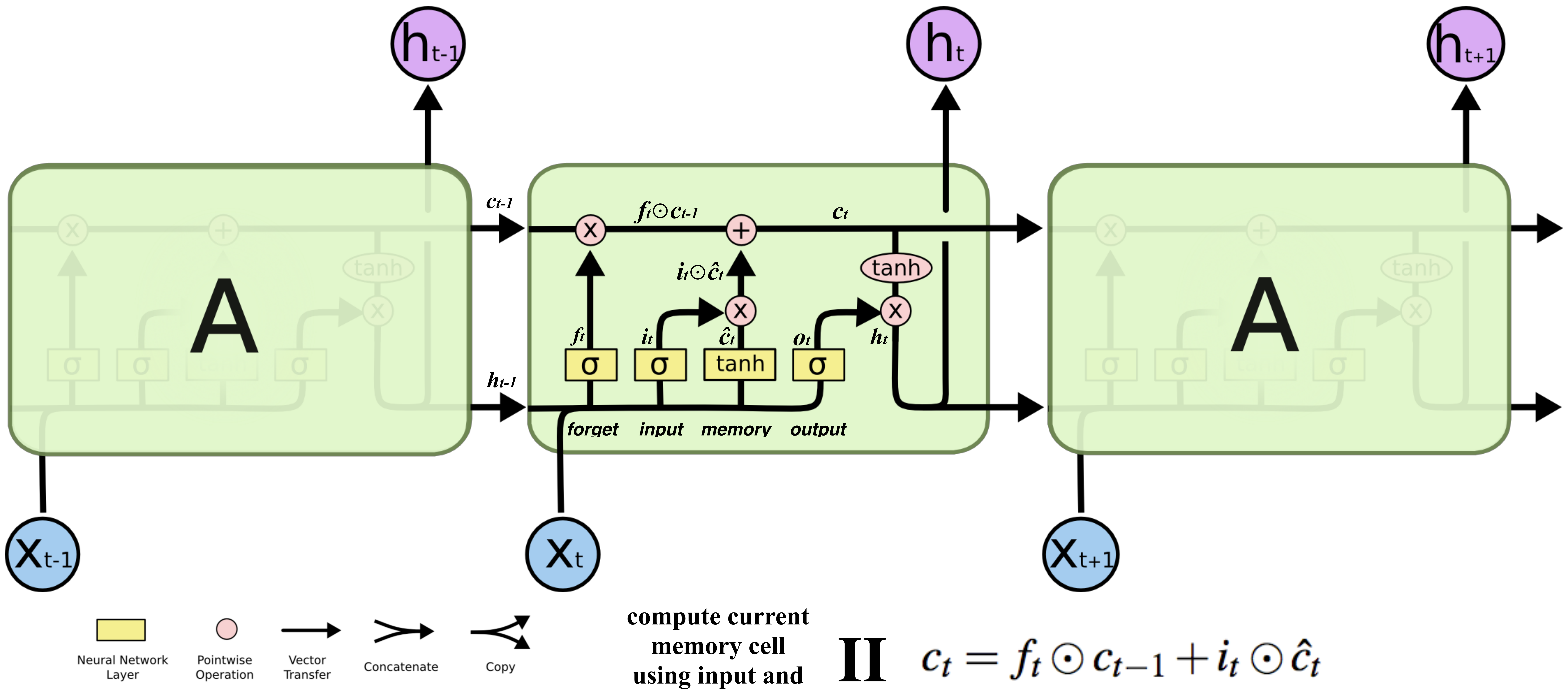
compute current hidden state using output gate and memory cell

I $$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \hat{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \cdot [h_{t-1}, x_t]$$

compute current input, forget, output, memory gate values

LSTM walkthrough in 4 steps

$p(x_{t+1} = w|x_1, \cdots, x_t) = \exp(u(w, h_t))/Z$



I compute current input, forget, output, memory gate values

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \hat{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \cdot [h_{t-1}, x_t]$$

II compute current memory cell using input and forget gates

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

III compute current hidden state using output gate and memory cell

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

IV compute current output probabilities for prediction by using softmax over the hidden state

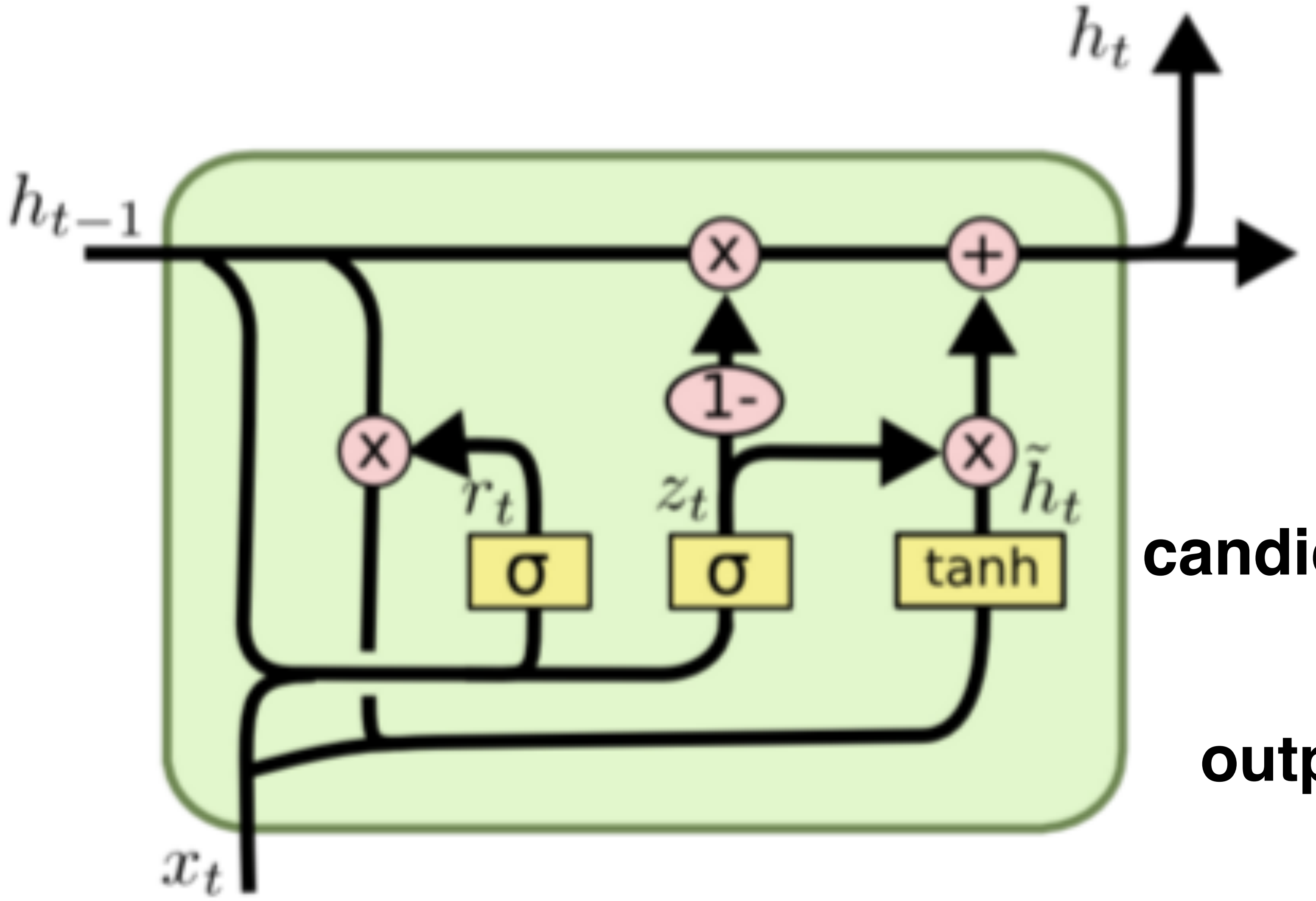
$$p(x_{t+1} = w|x_1, \cdots, x_t) = \exp(u(w, h_t))/Z$$

$$Z = \sum_{w' \in V} \exp(u(w', h_t))$$

The GRU Architecture

The GRU Architecture

- “Gated Recurrent Unit” Cho et al. (2014)



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

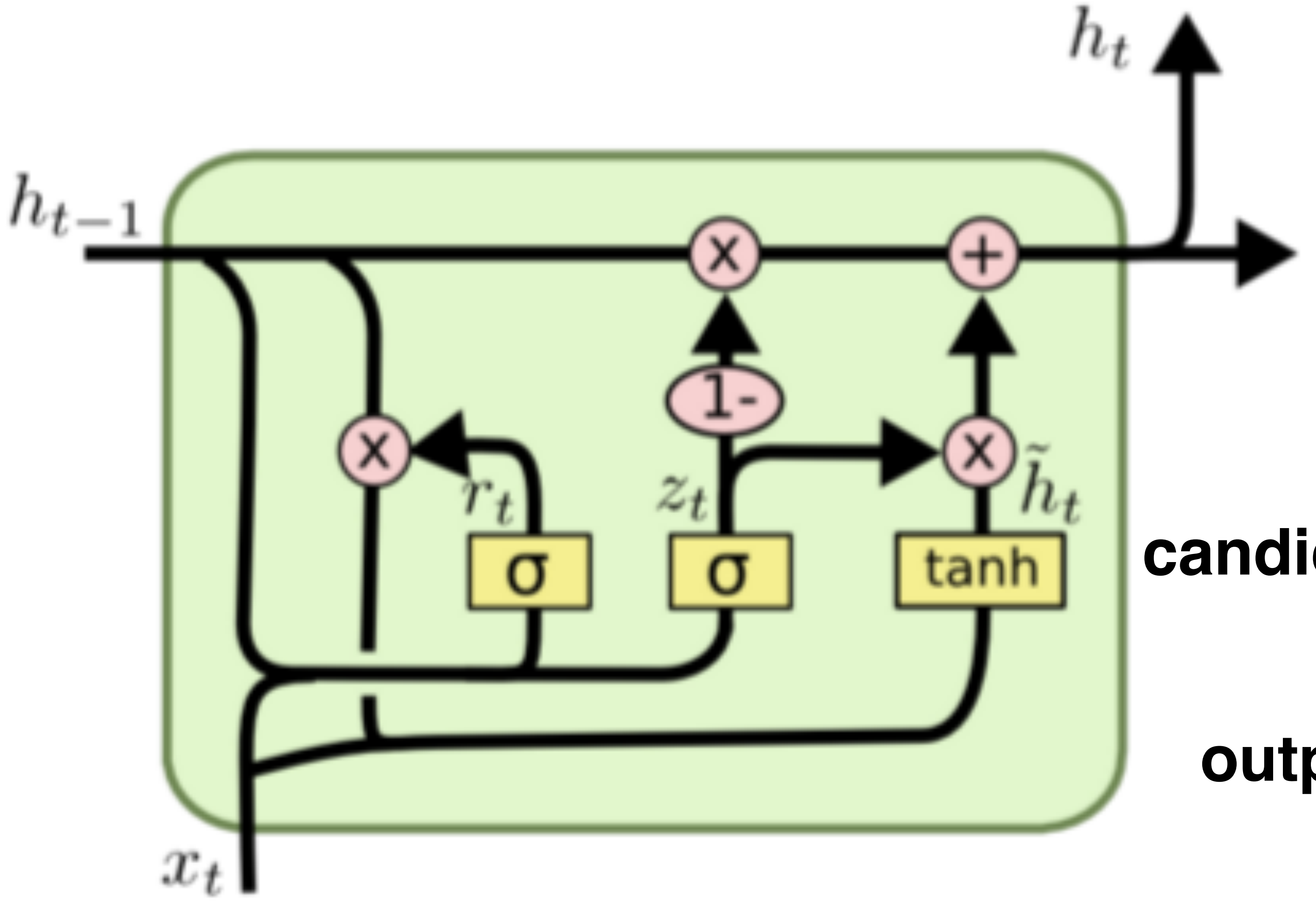
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$ **reset gate**

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

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

The GRU Architecture

- “Gated Recurrent Unit” Cho et al. (2014)
- Also widely used, simpler than the LSTM



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

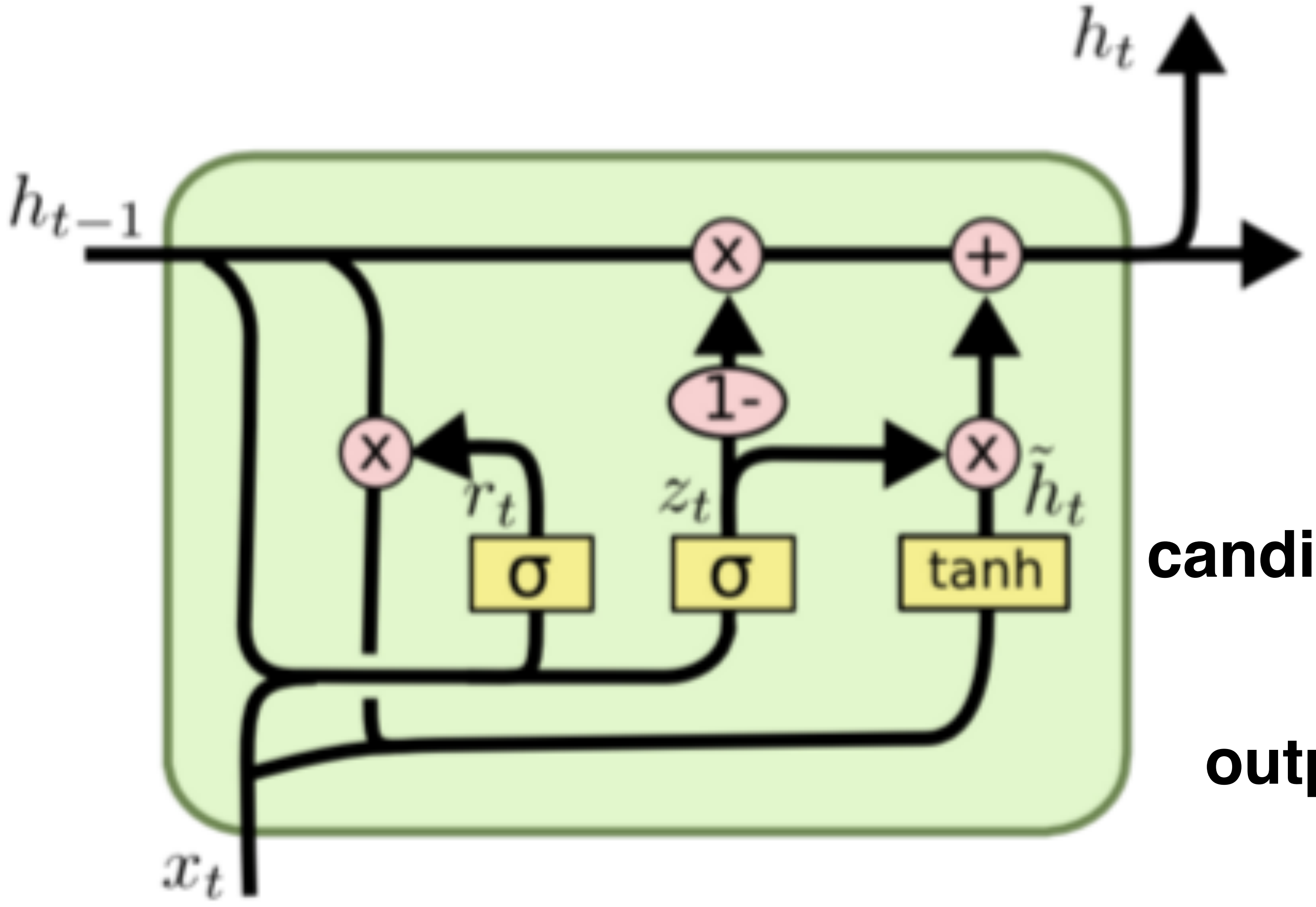
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$ **reset gate**

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

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

The GRU Architecture

- “Gated Recurrent Unit” Cho et al. (2014)
- Also widely used, simpler than the LSTM
- But weaker in “counting” tasks (Weiss and Goldberg, 2018)

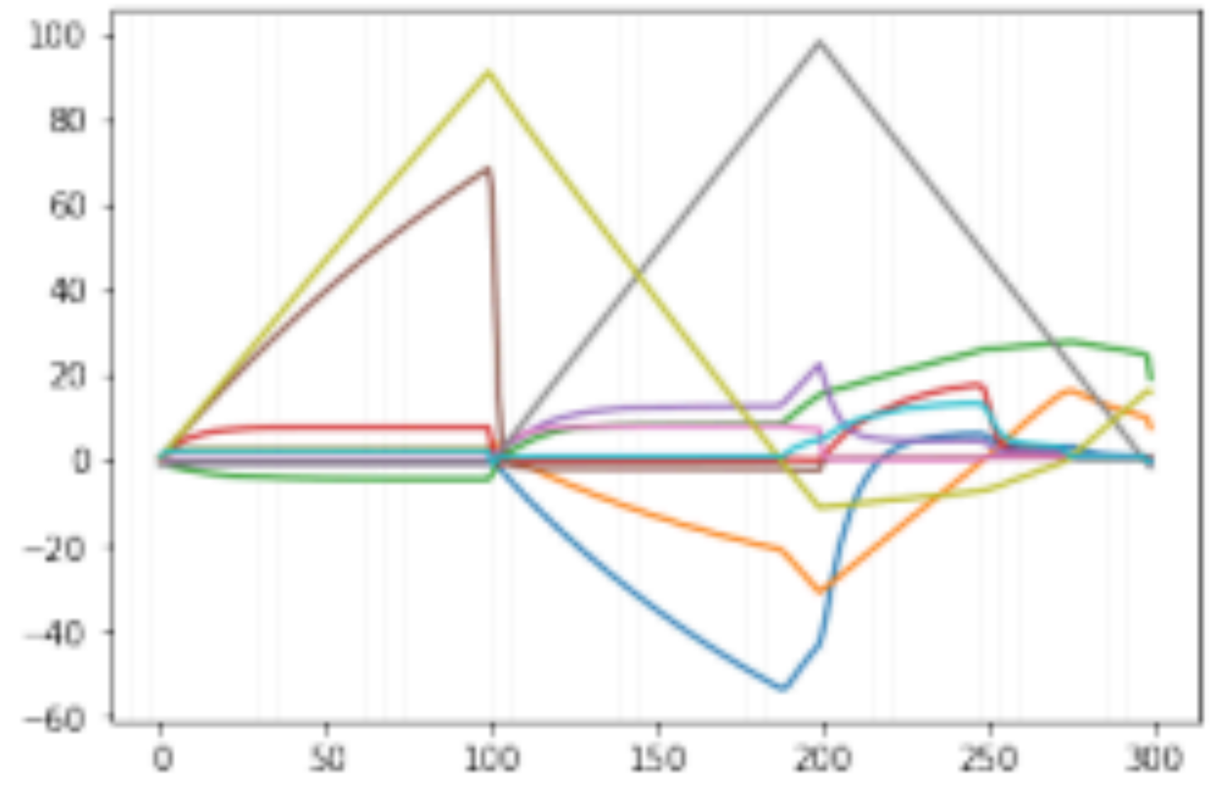


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$ **update gate**

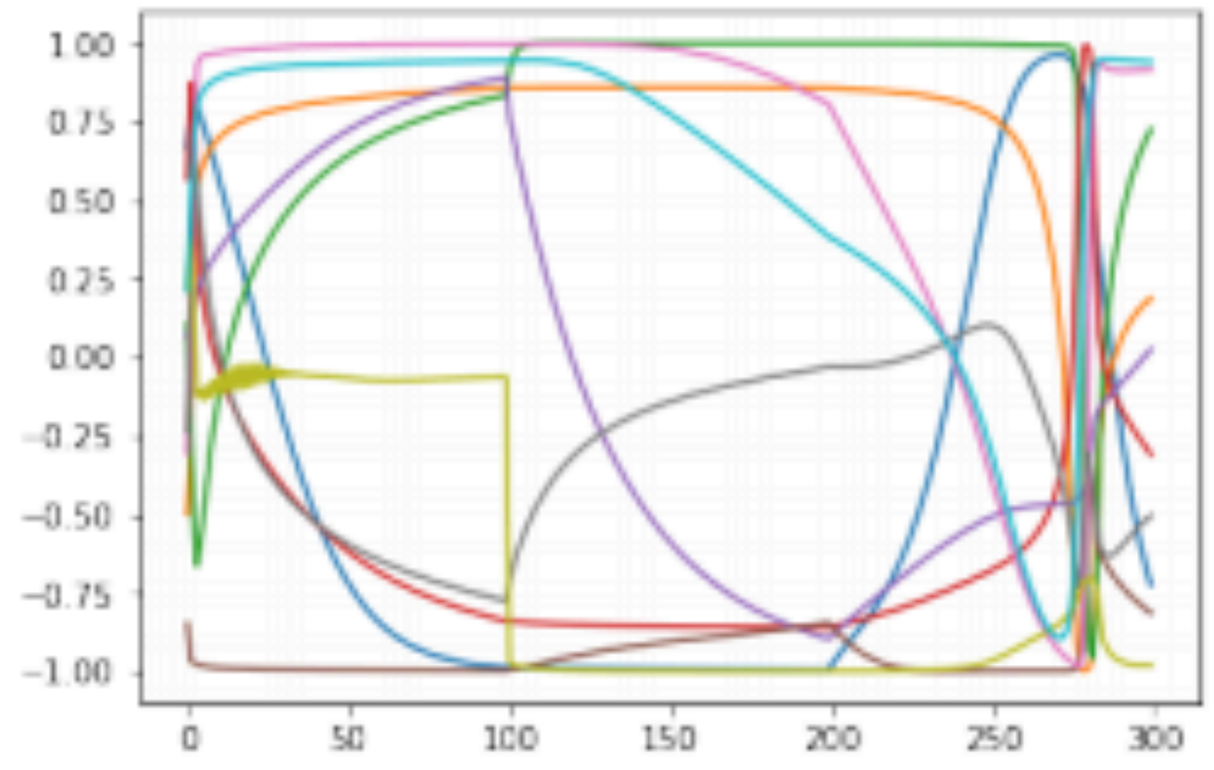
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$ **reset gate**

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

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



(b) $a^n b^n c^n$-LSTM on $a^{100}b^{100}c^{100}$



(d) $a^n b^n c^n$-GRU on $a^{100}b^{100}c^{100}$

Summary

Summary

- Neural networks can learn rich features representations

Summary

- Neural networks can learn rich features representations
- Trained using end-to-end optimization (e.g. SGD)

Summary

- Neural networks can learn rich features representations
- Trained using end-to-end optimization (e.g. SGD)
- Main components: weight matrices, activation functions

Summary

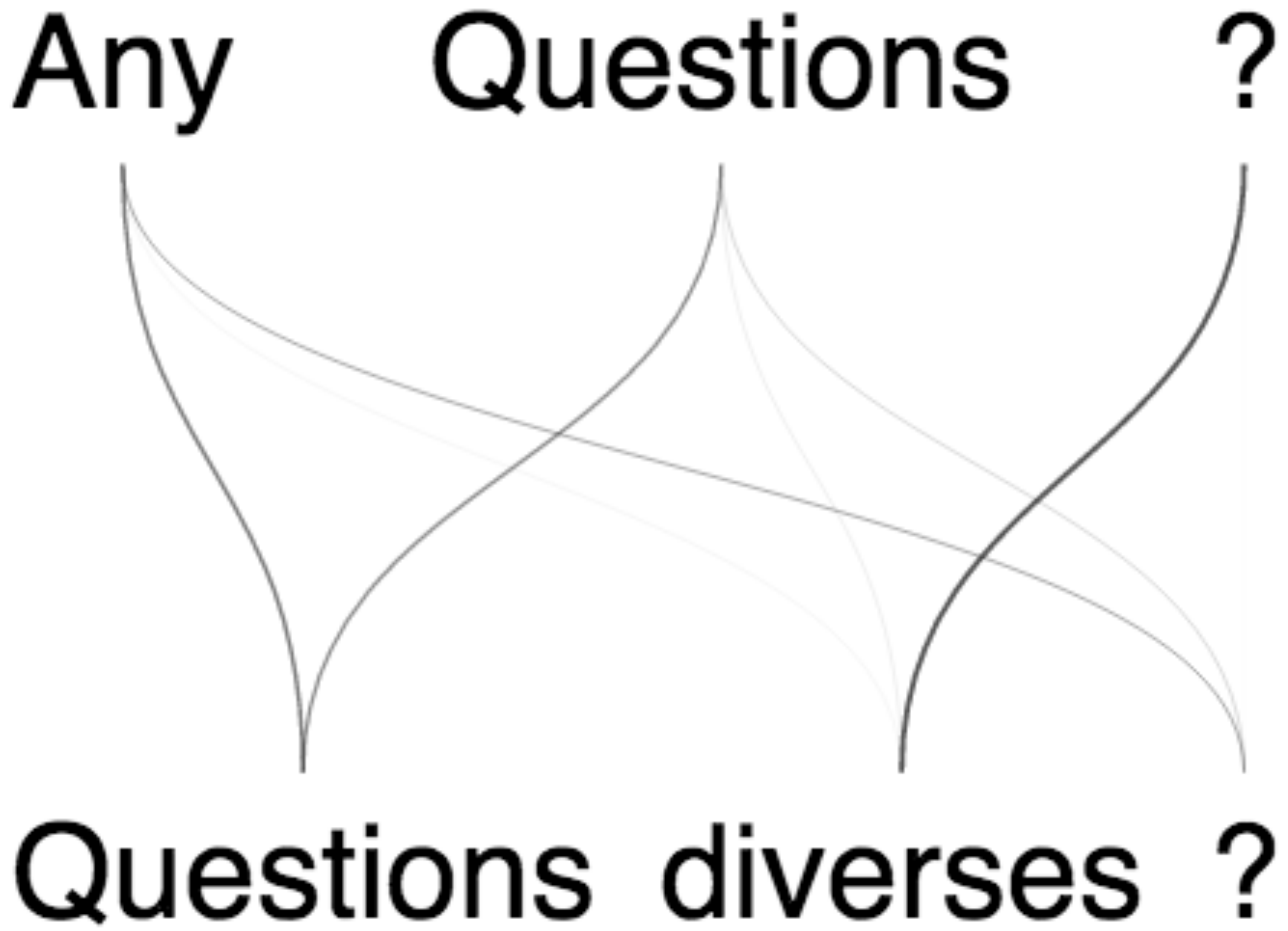
- Neural networks can learn rich features representations
- Trained using end-to-end optimization (e.g. SGD)
- Main components: weight matrices, activation functions
- Deeper networks are more expressive

Summary

- Neural networks can learn rich features representations
- Trained using end-to-end optimization (e.g. SGD)
- Main components: weight matrices, activation functions
- Deeper networks are more expressive
- Use RNNs to model variable length sequences

Summary

- Neural networks can learn rich features representations
- Trained using end-to-end optimization (e.g. SGD)
- Main components: weight matrices, activation functions
- Deeper networks are more expressive
- Use RNNs to model variable length sequences
- Vanishing gradients, LSTMs and GRUs


Any
Questions
?
Questions
diverses
?