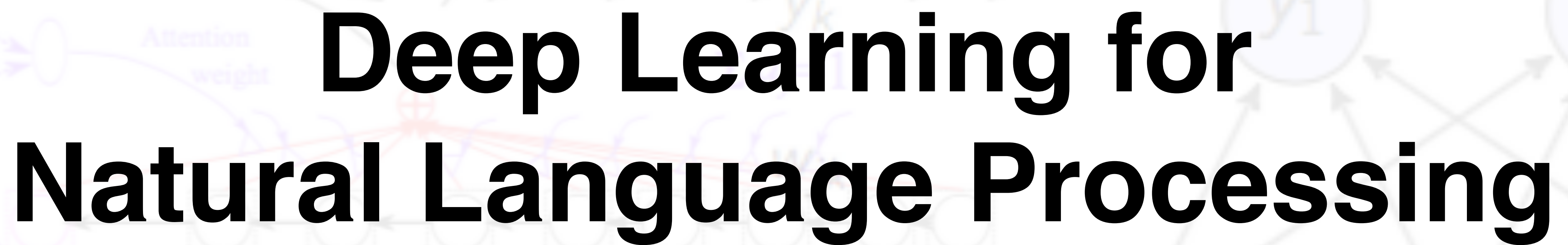
Deep Learning for Natural Language Processing

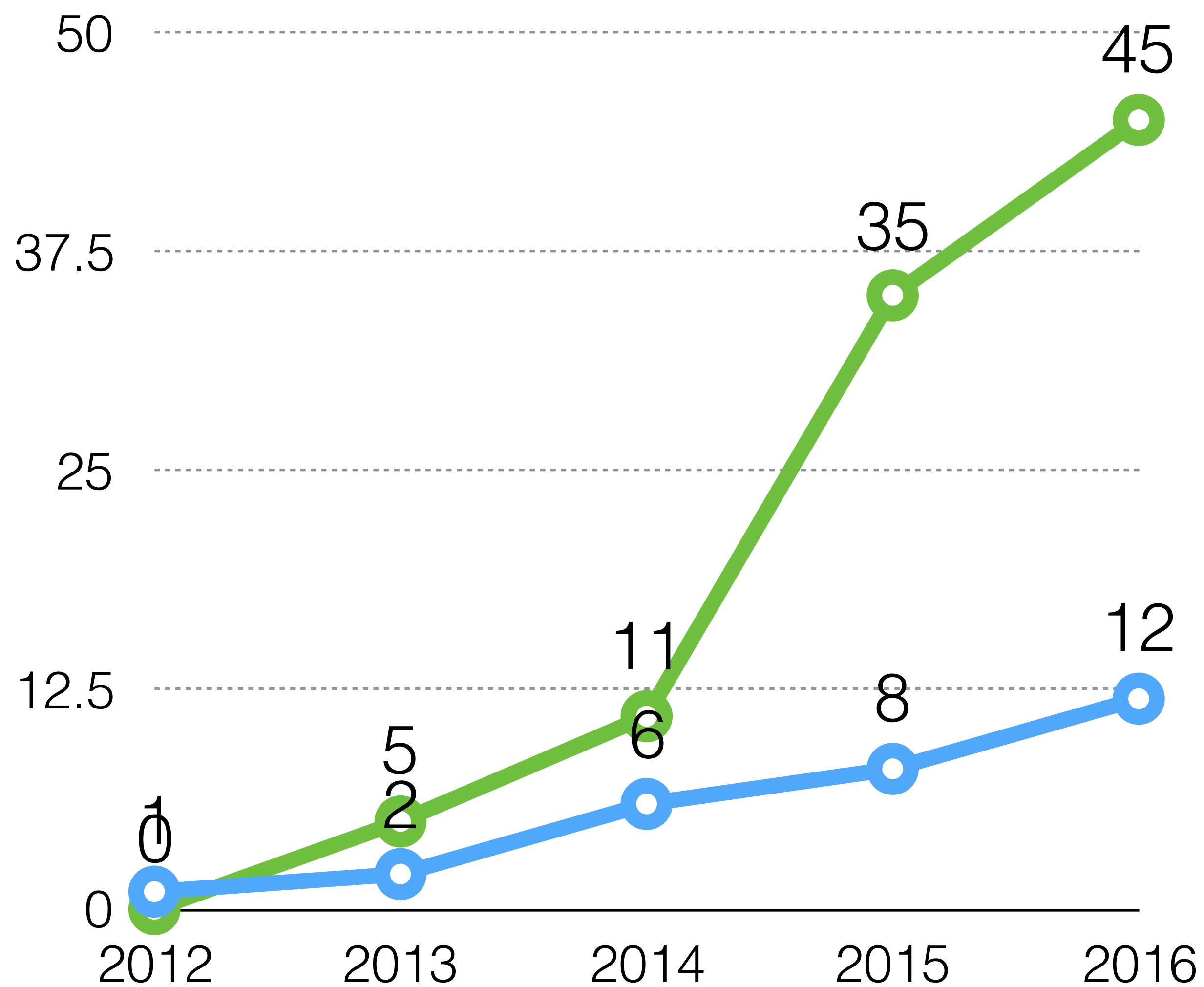


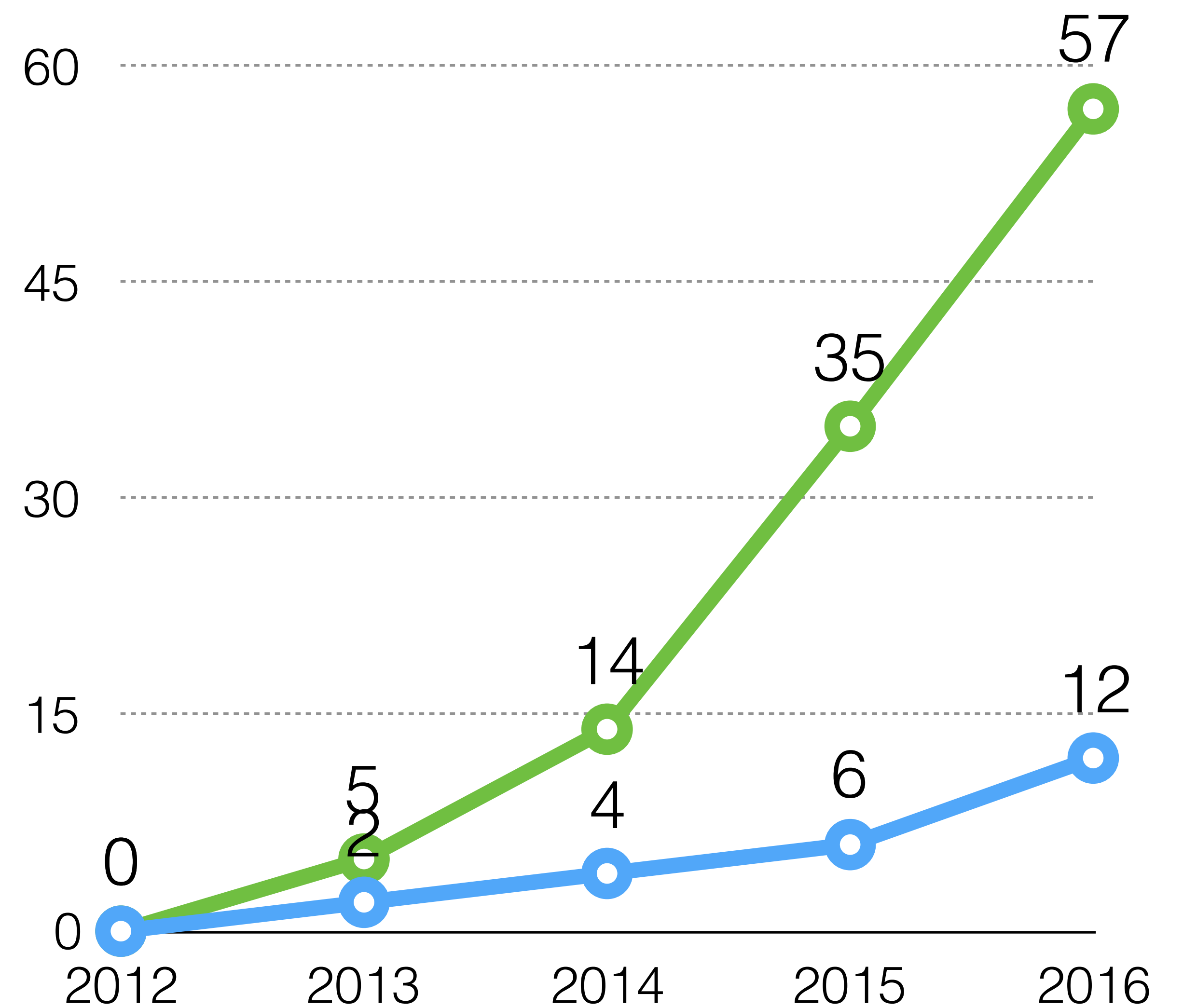
An Introduction

Roee Aharoni
Bar-Ilan University NLP Lab
Berlin PyData Meetup, 10.8.16

Motivation

of mentions in paper titles at top-tier annual NLP conferences (ACL, EMNLP) from 2012 to 2016:





What is deep learning?

“A family of learning methods that use deep architectures to learn high-level feature representations”

What is deep learning?

“A family of **learning methods** that use **deep architectures** to learn **high-level** feature representations”

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…)
  - 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 “1-layer” network

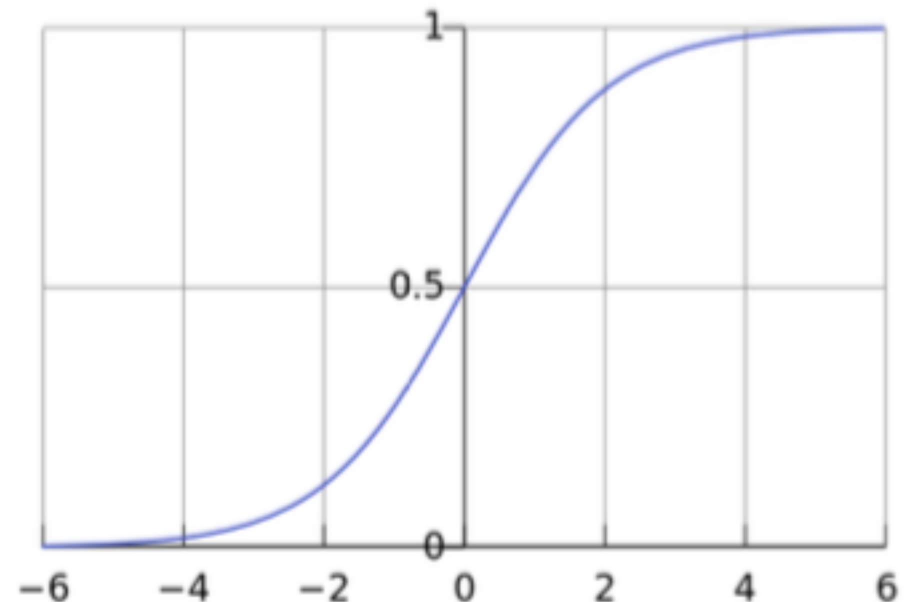
- Model the classier as:

$$f(x) = \sigma(w^T \cdot x) = \sigma(\sum_i w_i x_i)$$

- Learn the weight vector: $w \in R^d$ using gradient-descent (next slide)

- σ is a non-linearity, e.g. the sigmoid function (creates dependency between the features, maps $f(x)$ to [0,1]):

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



Training (log. regression) with gradient-descent

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

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

- Derive the loss-function w.r.t. the weight vector, w:

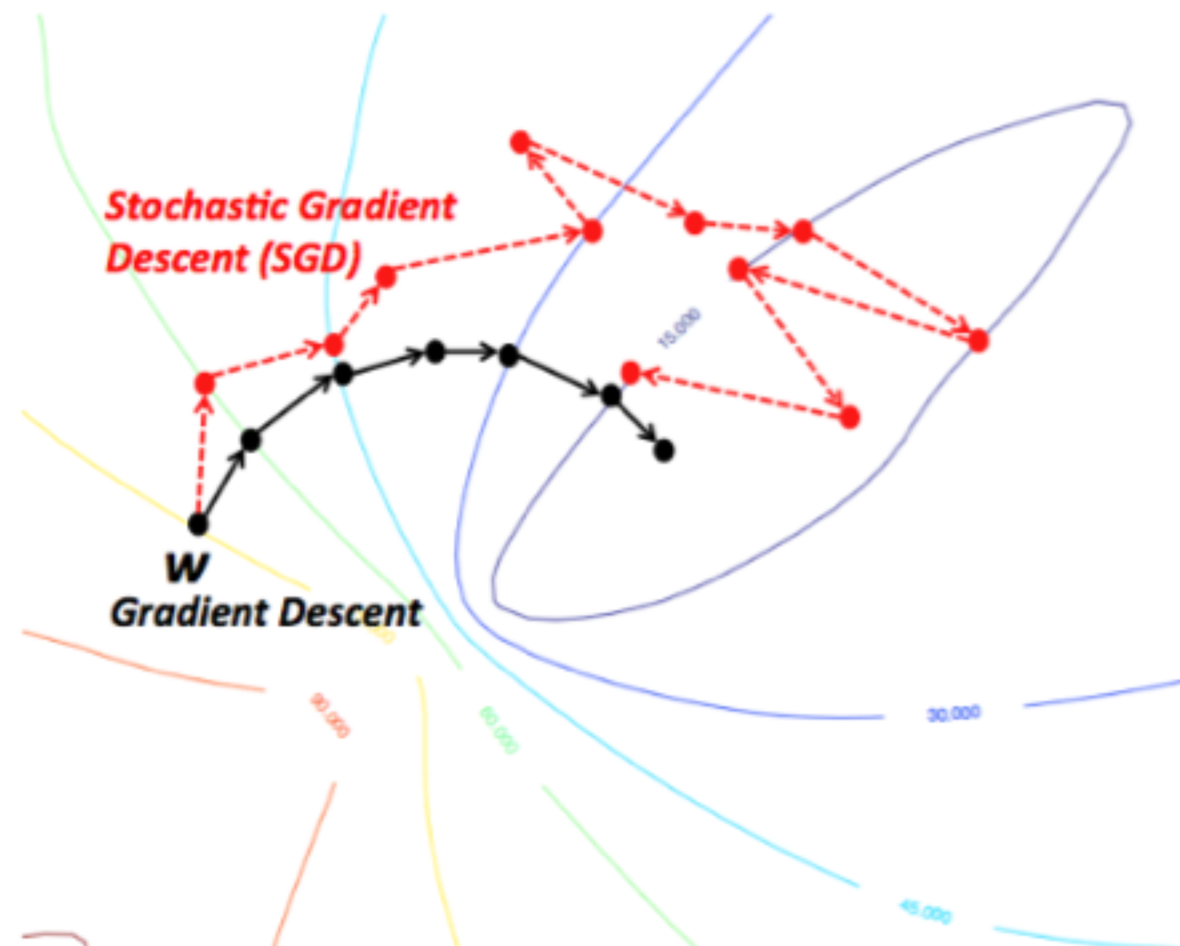
$$\nabla_w Loss = \sum_m \left[\sigma(w^T x^{(m)}) - y^{(m)}\right] \sigma'(w^T x^{(m)}) x^{(m)}$$

- 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)

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

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



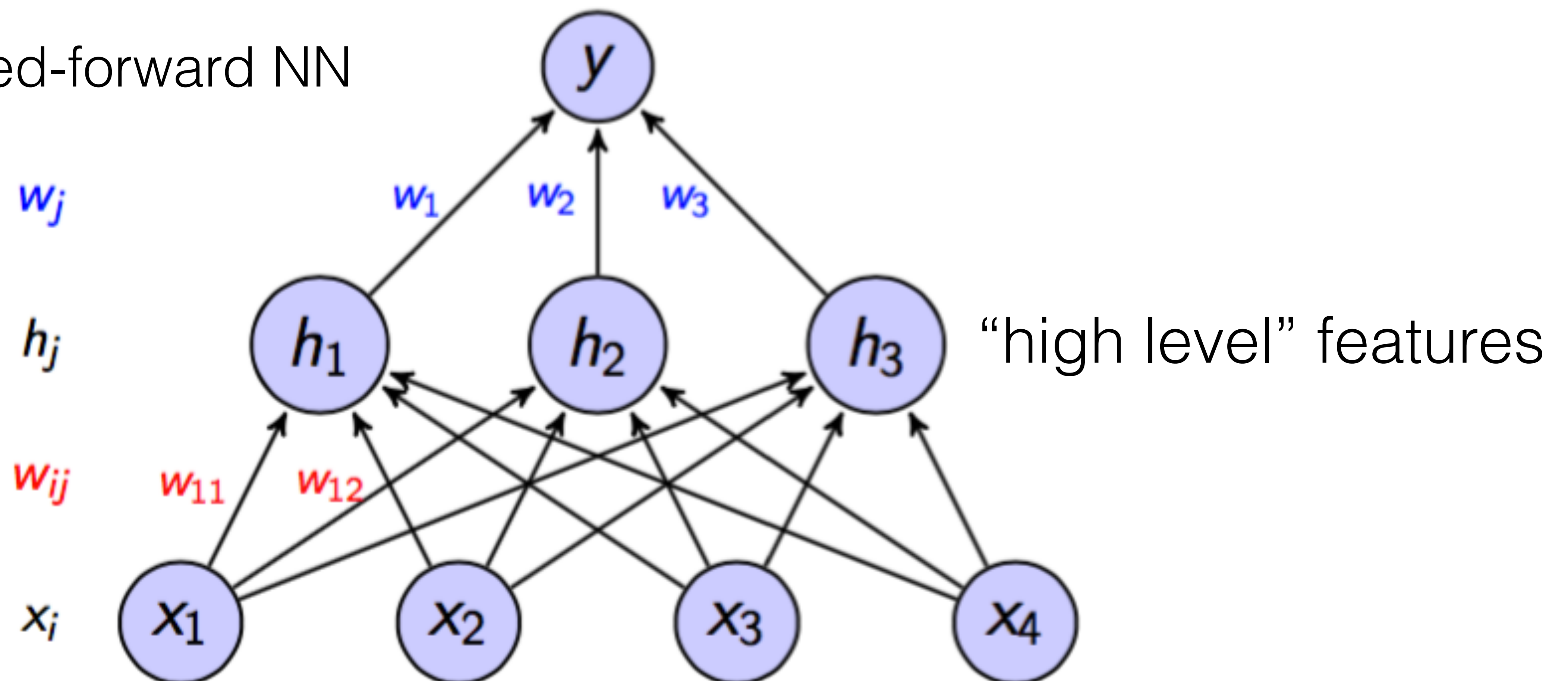
Multi layer perceptron (MLP) - a multi-layer NN

- Model the classifier as:

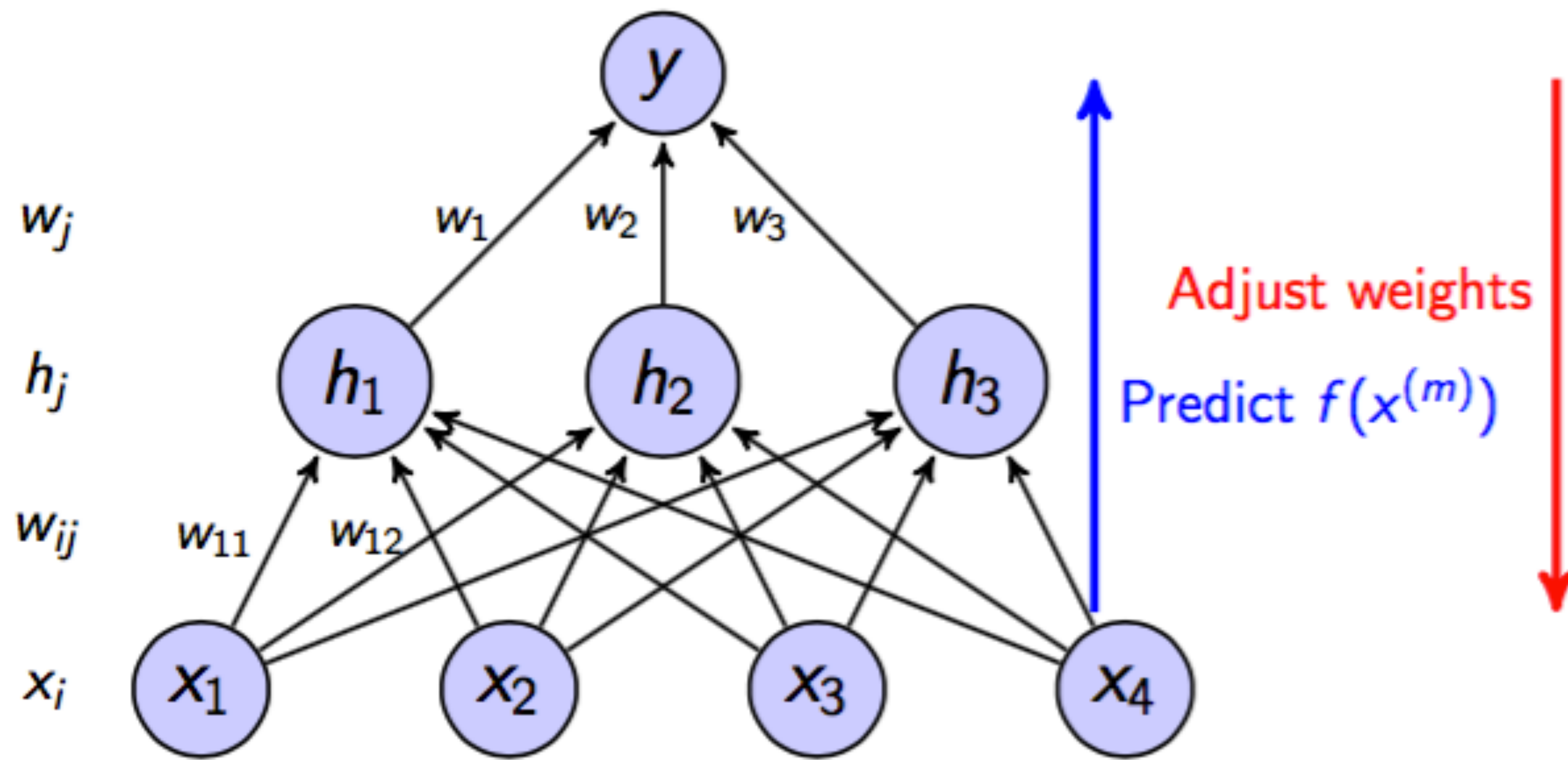
$$f(x) = \sigma(\sum_j w_j \cdot h_j) = \sigma(\sum_j w_j \cdot \sigma(\sum_i w_{ij} x_i))$$

- Can be seen as multilayer logistic regression

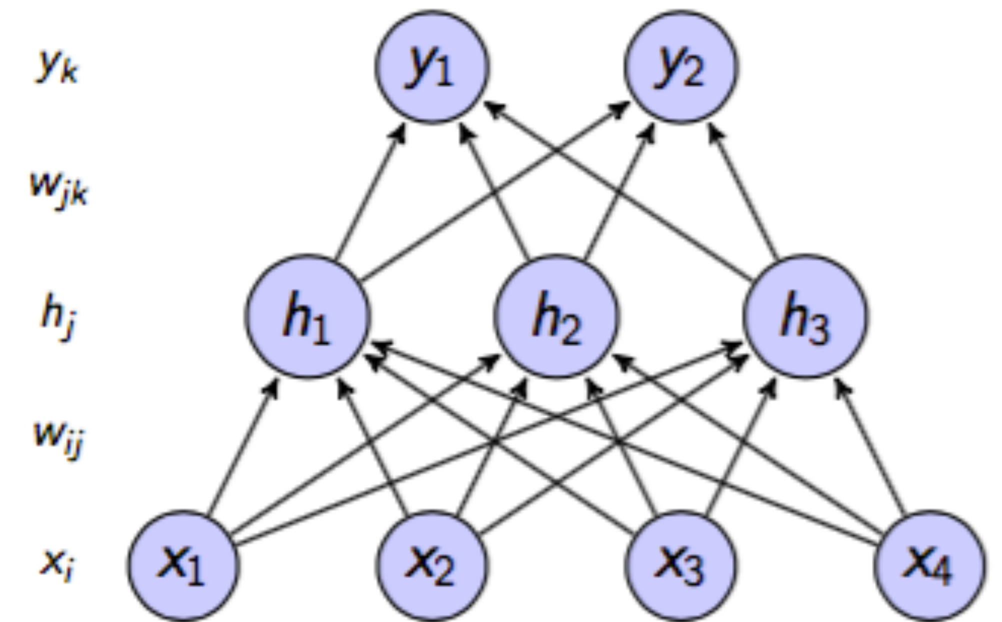
- a.k.a feed-forward NN



Training (an MLP) with Backpropagation:



Training (an MLP) with Backpropagation:



- Assume two outputs per input:
- Define the loss-function per example:

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

- Derive the loss-function w.r.t. the last layer:

$$\frac{\partial Loss}{\partial w_{jk}} = \frac{\partial Loss}{\partial in_k}\frac{\partial in_k}{\partial w_{jk}} = \delta_k \frac{\partial(\sum_j w_{jk} h_j)}{\partial w_{jk}} = \delta_k h_j$$

- Derive the loss function w.r.t. the first layer:

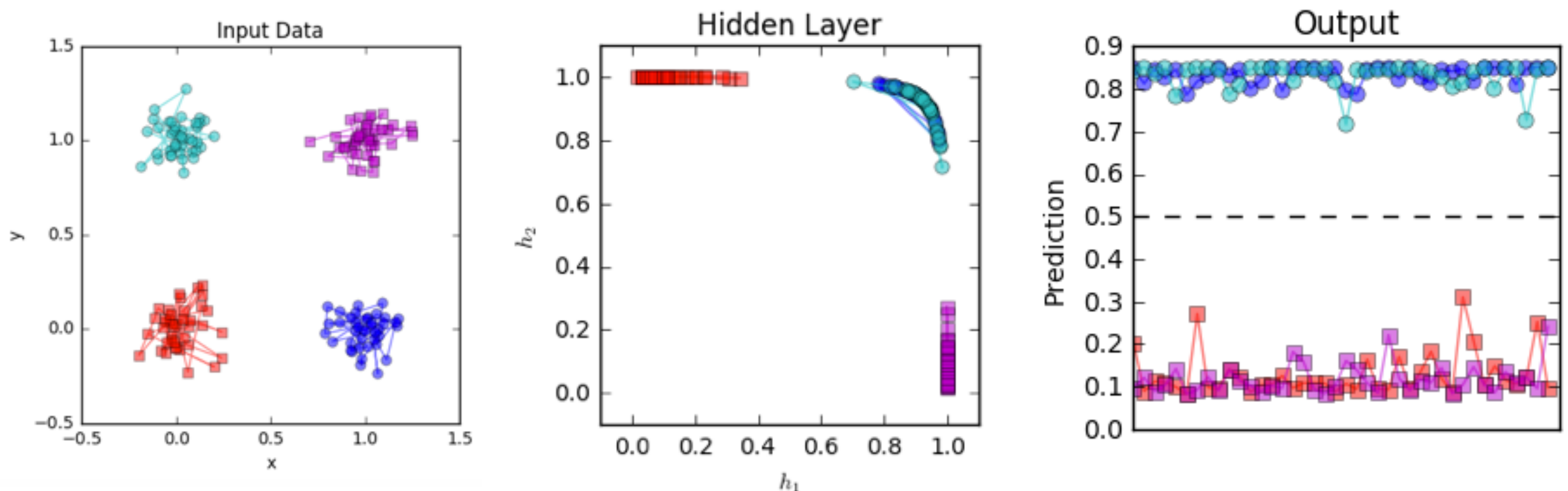
$$\frac{\partial Loss}{\partial w_{ij}} = \frac{\partial Loss}{\partial in_j}\frac{\partial in_j}{\partial w_{ij}} = \delta_j \frac{\partial(\sum_j w_{ij} x_i)}{\partial w_{ij}} = \delta_j x_i$$

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

Why deeper is better?

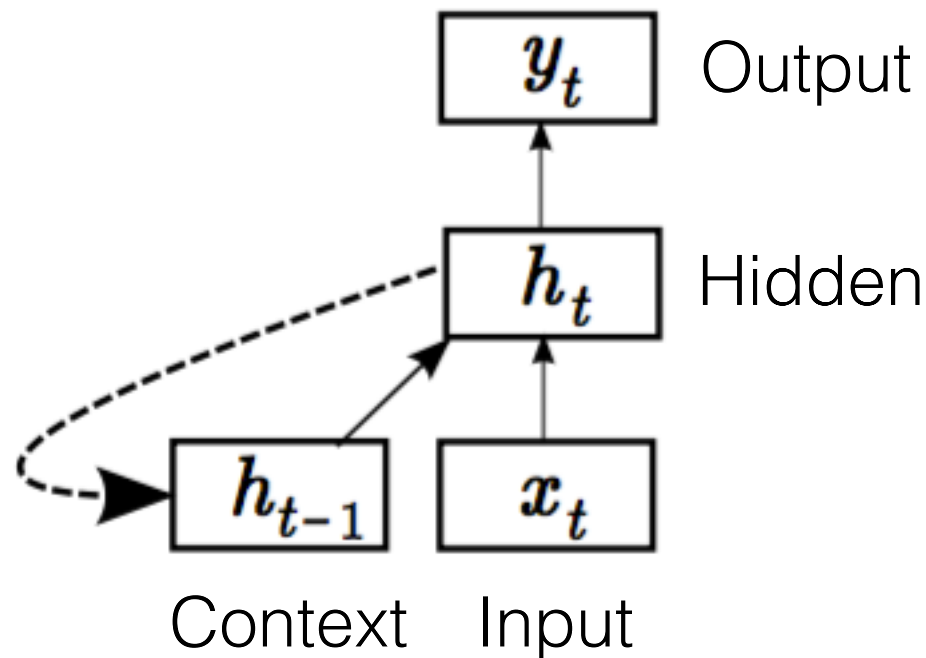
- A deeper architecture is more expressive than a shallow one given same number of nodes [Bishop, 1995]
  - 1-layer nets (log. regression) can only model linear hyperplanes
  - 2-layer nets can model any continuous function (given sufficient nodes)
  - >3-layer nets can do so with fewer nodes

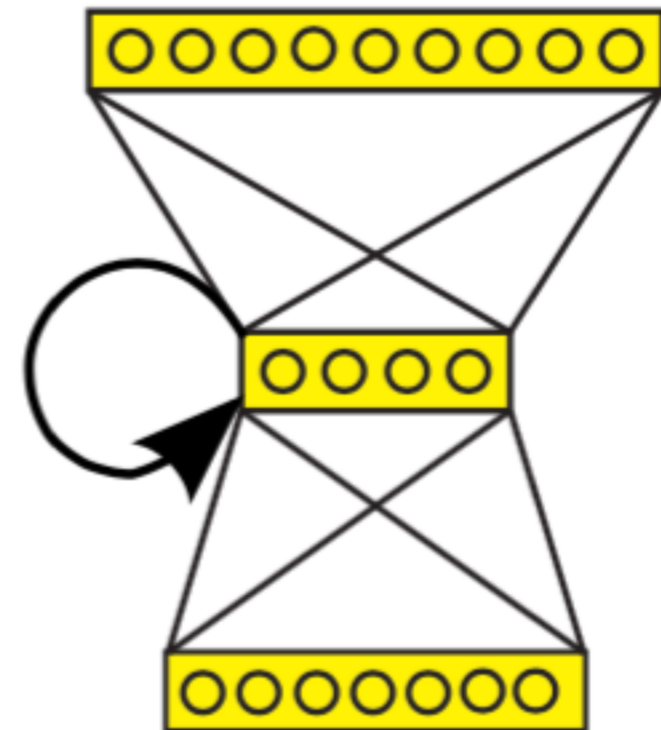
Example - the XOR problem: $x \oplus y = (x \vee y) \wedge \neg(x \wedge y)$



Recurrent Neural Networks (RNN's)

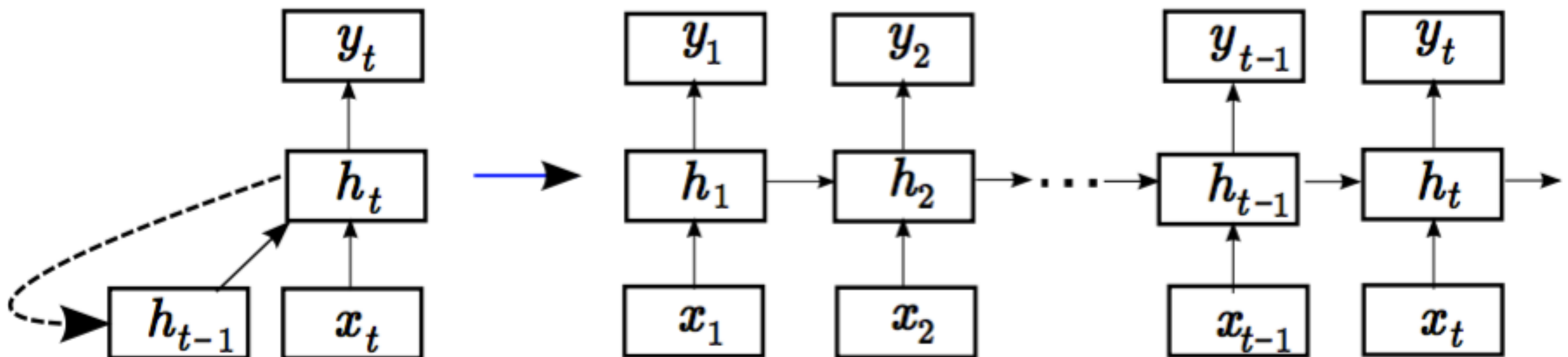
- Enable variable length inputs (sequences)
- Modeling internal structure in the input or output
- Introduce a "memory/context" component to utilize history





Recurrent Neural Networks (RNN's)

- "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 implemented as softmax



The Softmax Function

- Enables to output a probability distribution over k possible classes
- can be seen as trying to minimize the cross-entropy between the predictions and the truth
- y_i usually holds log-likelihood values

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

Training (RNN's) with Backpropagation Through Time

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

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

- Derive the loss function w.r.t. parameters Θ, starting at $t = T$:

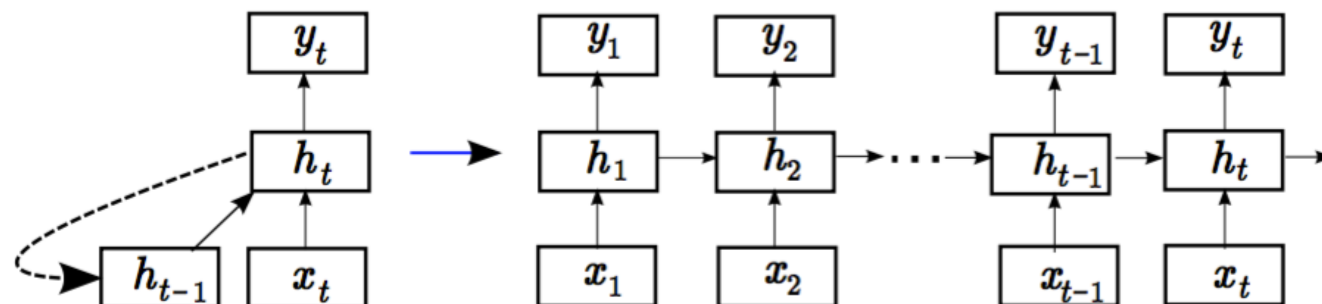
$$\nabla\Theta = \frac{\partial J_t}{\partial \Theta}$$

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

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

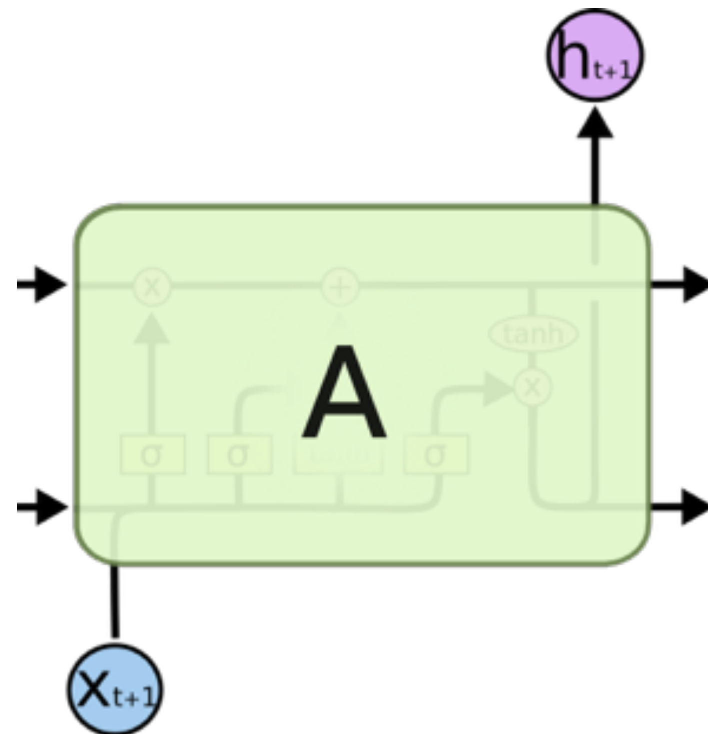
- Eventually, update the weights:

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



Vanishing gradients, 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]
- Most of the recent RNN works utilize such architectures



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 memorized (input gate), old content should be erased (forget gate), and 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 memorized (input gate), old content should be erased (forget gate), and 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 memorized (input gate), old content should be erased (forget gate), and current content should be exposed (output gate). More formally:

II compute current memory cell using input and forget gates

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

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 memorized (input gate), old content should be erased (forget gate), and 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 $$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

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

compute current hidden state using output gate and memory cell

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 memorized (input gate), old content should be erased (forget gate), and current content should be exposed (output gate). More formally:

I compute current input, forget, output gates and memory cell update

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

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

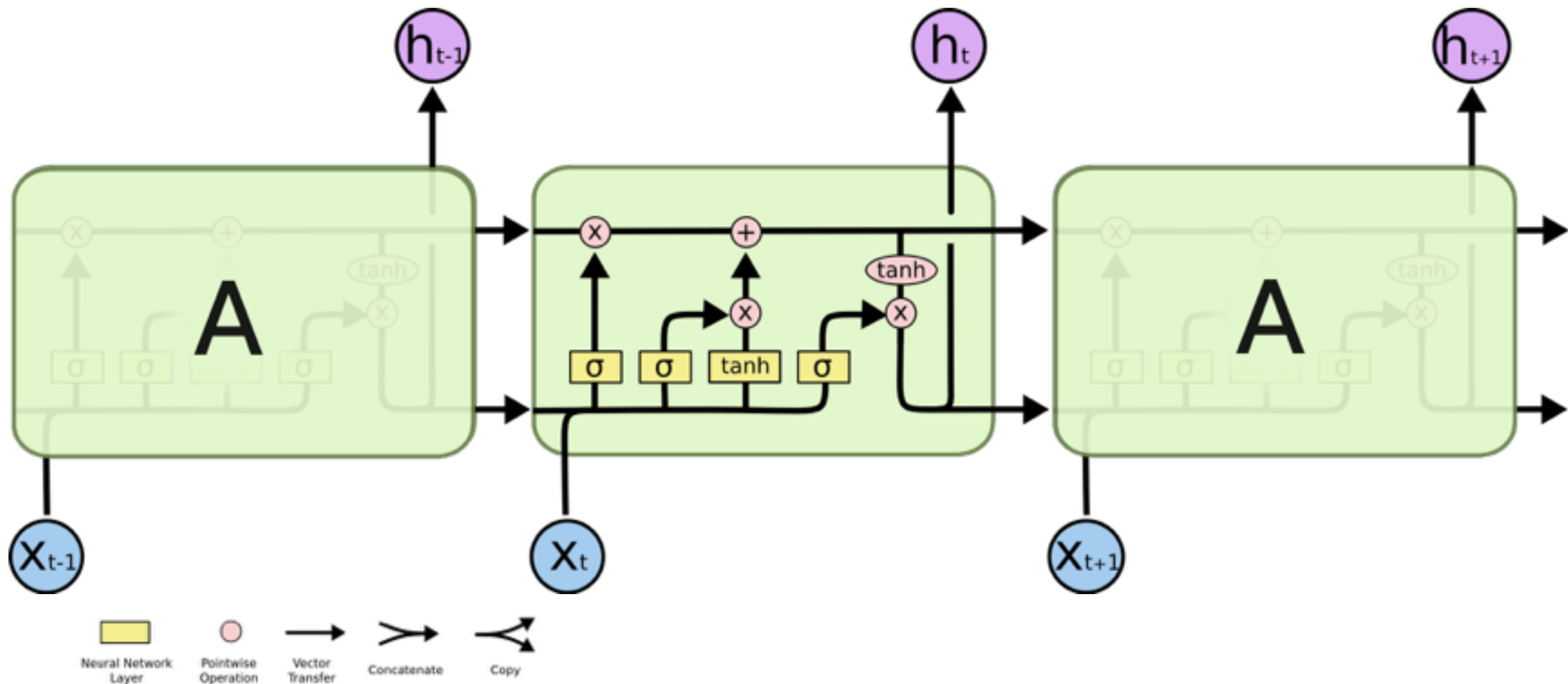
$$h_t = o_t \odot \tanh(c_t)$$

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

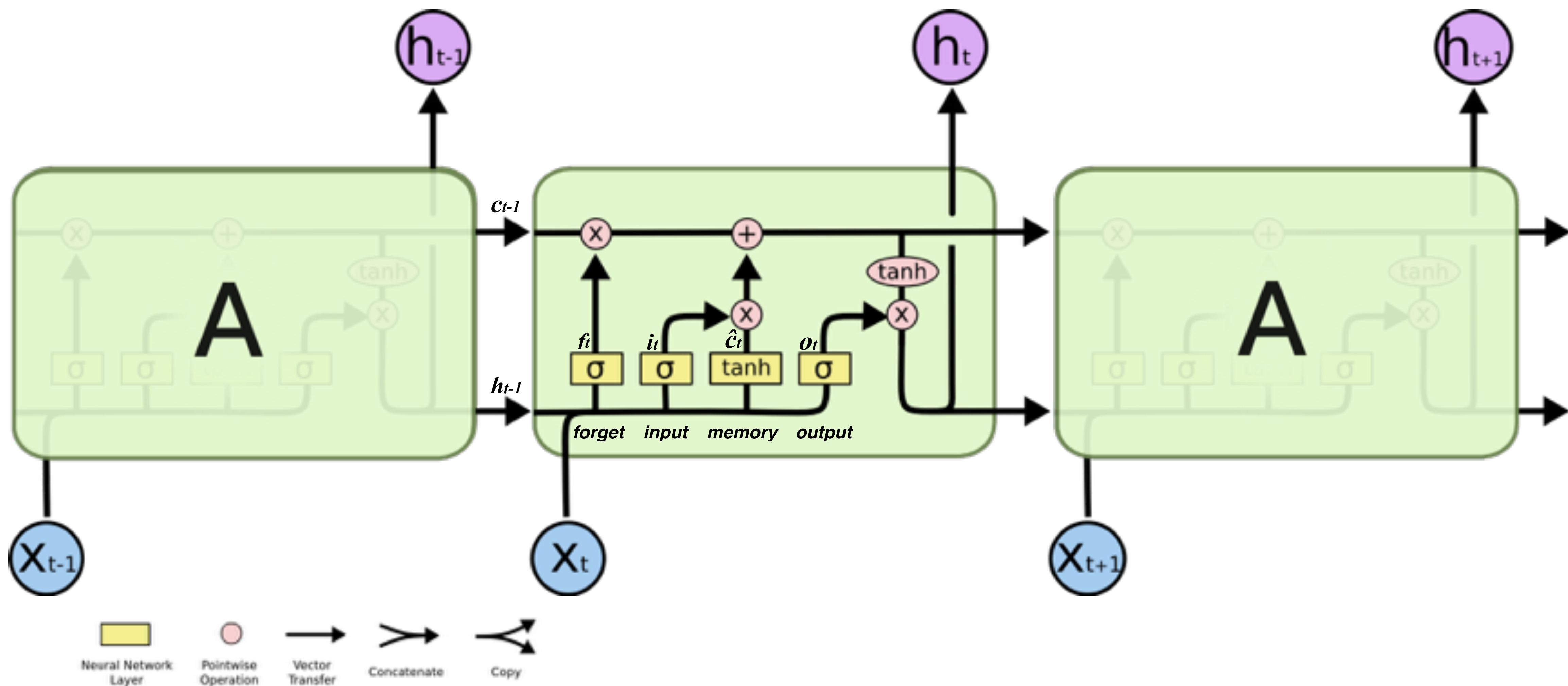
$$p(x_{t+1} = w | x_1, \cdots, x_t) = \exp(u(w, h_t))/Z$$

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

LSTM walkthrough in 4 steps



LSTM walkthrough in 4 steps

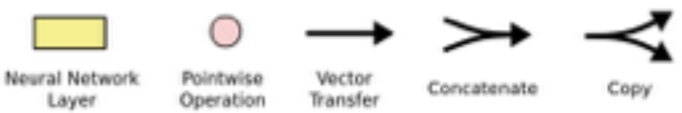


I

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



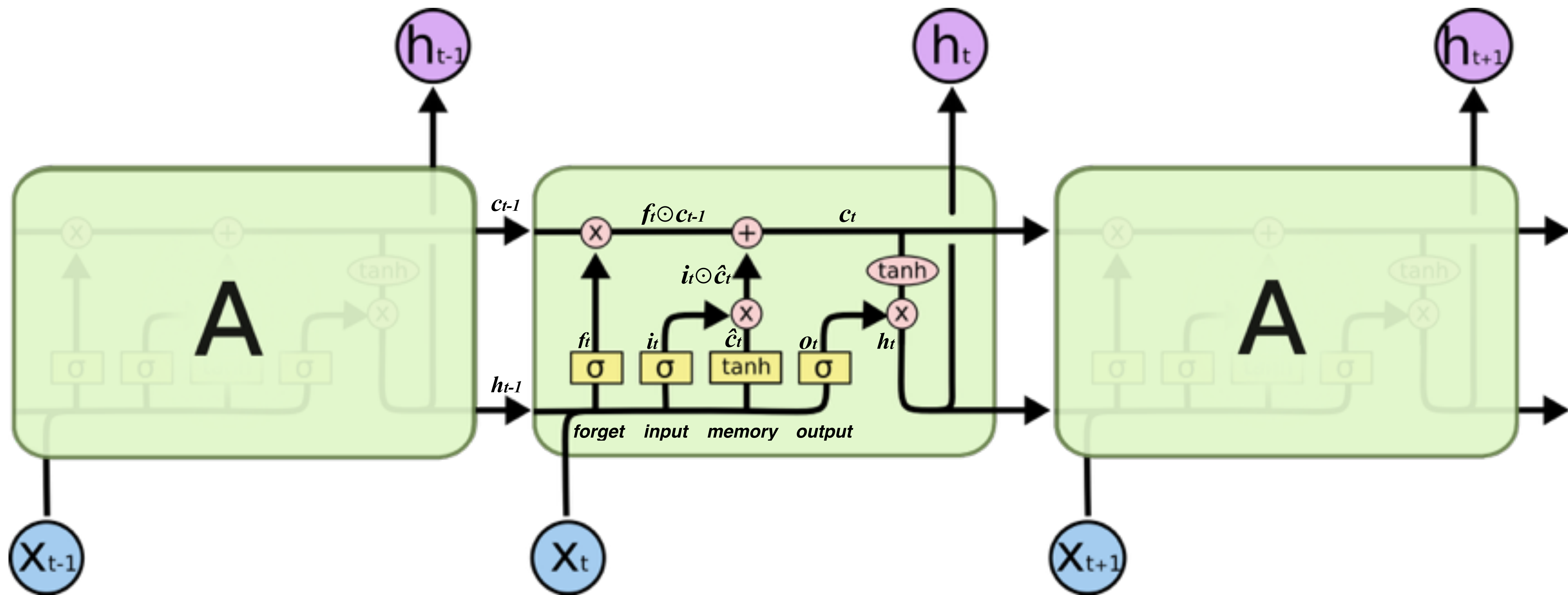
II compute current memory cell using input and forget gates

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

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



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

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

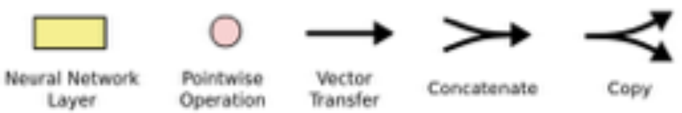
III

compute current hidden state using output gate and memory cell

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

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

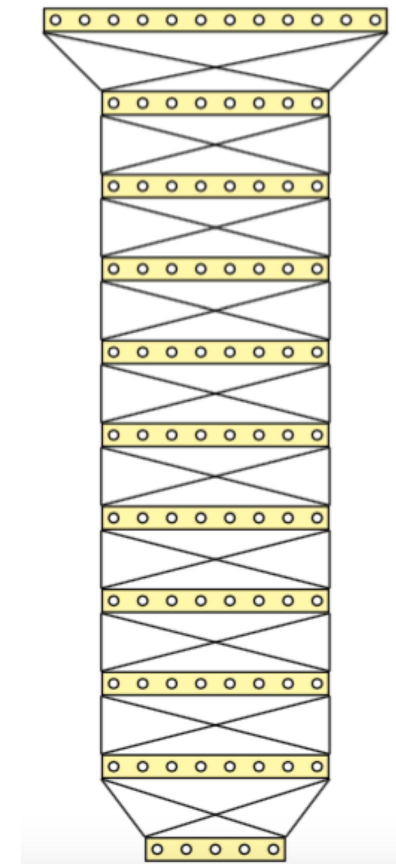
compute current output probabilities for prediction by using softmax over the hidden state

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

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

Why now? Today vs. 80's-90's

- Number of hidden layers: 10 (or more) rather than 2-3
- Number of output nodes: 5000 (or more) rather then 50
- Better optimization strategies, heuristics (layer-by-layer pre-training, dropout…)
- Much more **computation power**



Neural Network Models for Natural Language Processing

What is a Language Model?

- A language model $p(w_1^N)$ measures how likely is the sentence: $w_1^N = x_1, x_2, ..., x_N$

- Usually modeled as a product of conditionals:

$$p(x_1, x_2, \ldots, x_N) = \prod_{t=1}^{T} p(x_t \mid x_1, ..., x_{t-1})$$

- The “conventional” approach: assume a Markov chain of order n and count:

$$p(x_1, x_2, \ldots, x_N) = \prod_{t=1}^{T} p(x_t \mid x_{t-n}, ..., x_{t-1})$$

$$p(x_t \mid x_{t-n}, \ldots, x_{t-1}) = \frac{count(x_{t-n}, \ldots, x_{t-1}, x_t)}{count(x_{t-n}, \ldots, x_{t-1})}$$

What is a Language Model?

Lets compute:

$p(i, would, like, to, ..., < /s >)$

- uni-gram LM:

$p(i)p(would)p(like)...p(< /s >)$

- bi-gram LM:

$p(i)p(would \mid i)p(like \mid would)...p(< /s >\mid .)$

- tri-gram LM:

$p(i)p(would \mid i)p(like \mid i, would)...p(< /s >\mid work, .)$

word	unigram	bigram	trigram	4-gram
i	6.684	3.197	3.197	3.197
would	8.342	2.884	2.791	2.791
like	9.129	2.026	1.031	1.290
to	5.081	0.402	0.144	0.113
commend	15.487	12.335	8.794	8.633
the	3.885	1.402	1.084	0.880
rapporteur	10.840	7.319	2.763	2.350
on	6.765	4.140	4.150	1.862
his	10.678	7.316	2.367	1.978
work	9.993	4.816	3.498	2.394
.	4.896	3.020	1.785	1.510
</s>	4.828	0.005	0.000	0.000
average	8.051	4.072	2.634	2.251
perplexity	265.136	16.817	6.206	4.758

Perplexity - The lower, the better

The conventional approach - Issues

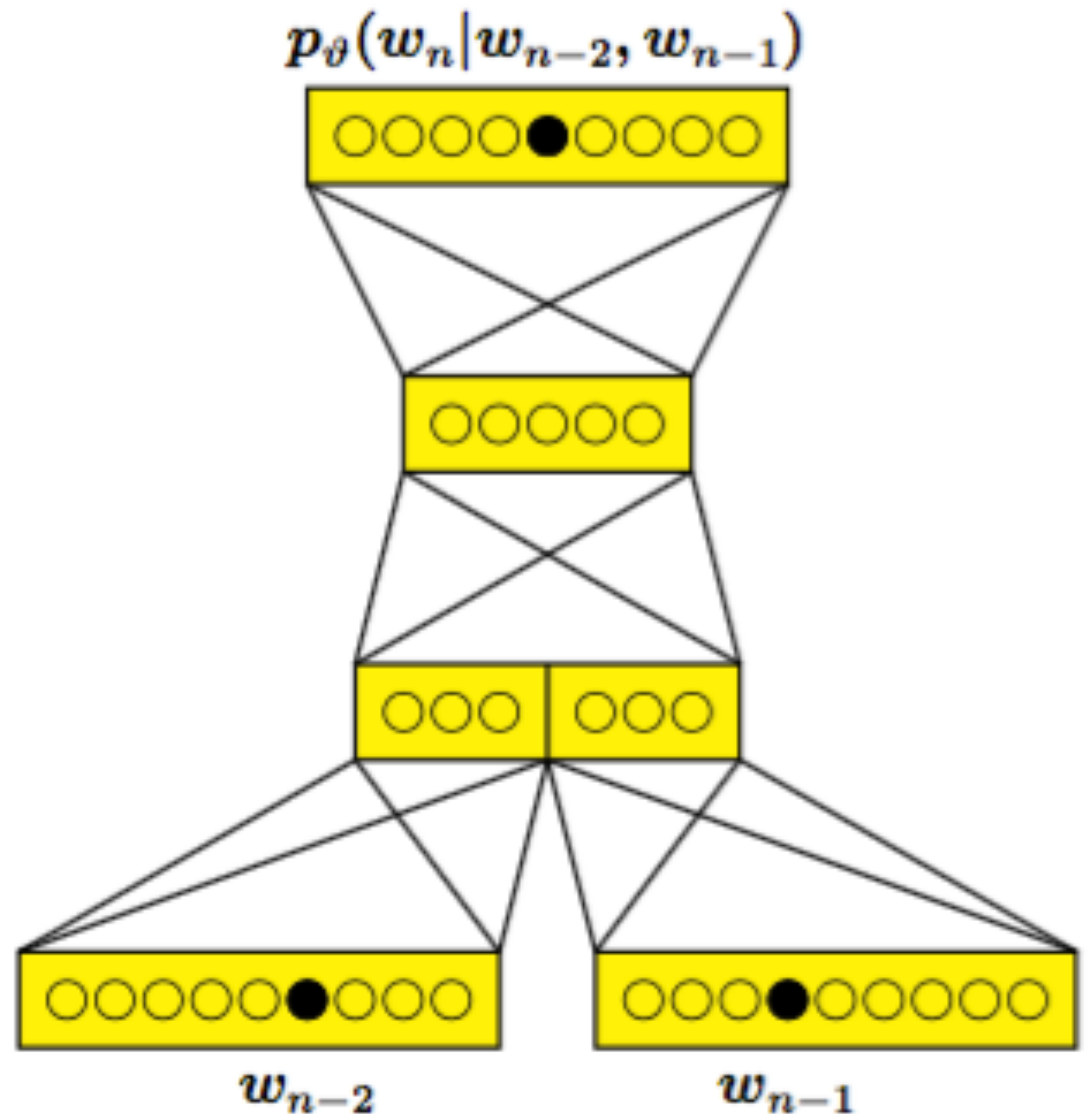
- Data sparsity - many n-grams do not appear in the training data
  - Can be handled by smoothing, back-off
- Lack of generalization
  - “chases a cat”, “chases a dog”, “chases a rabbit”
  - “chases an ostrich”?

The conventional approach - Issues

- Data sparsity - many n-grams do not appear in the training data
  - Can be handled by smoothing, back-off
- Lack of generalization
  - “chases a cat”, “chases a dog”, “chases a rabbit”
  - “chases an ostrich”?

Language Modeling with MLP

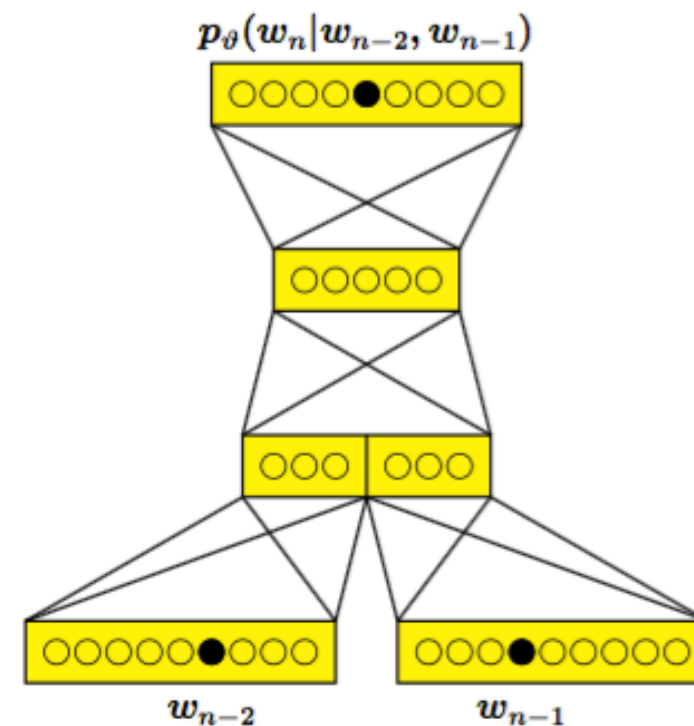
- Start with one-hot encoding of each word
- Learn **continuous space** word representations
- Non-Linear hidden Layer
- Output probabilities using the Softmax function



Language Modeling with MLP

- Experiment details:
  - vocabulary size: 128k words
  - training text: 50M words
  - development corpus: 39k words
  - evaluation corpus: 35k words
- Network structure:
  - projection layer: 300 nodes (per word)
  - hidden layer: 600 nodes
  - total amount of params:

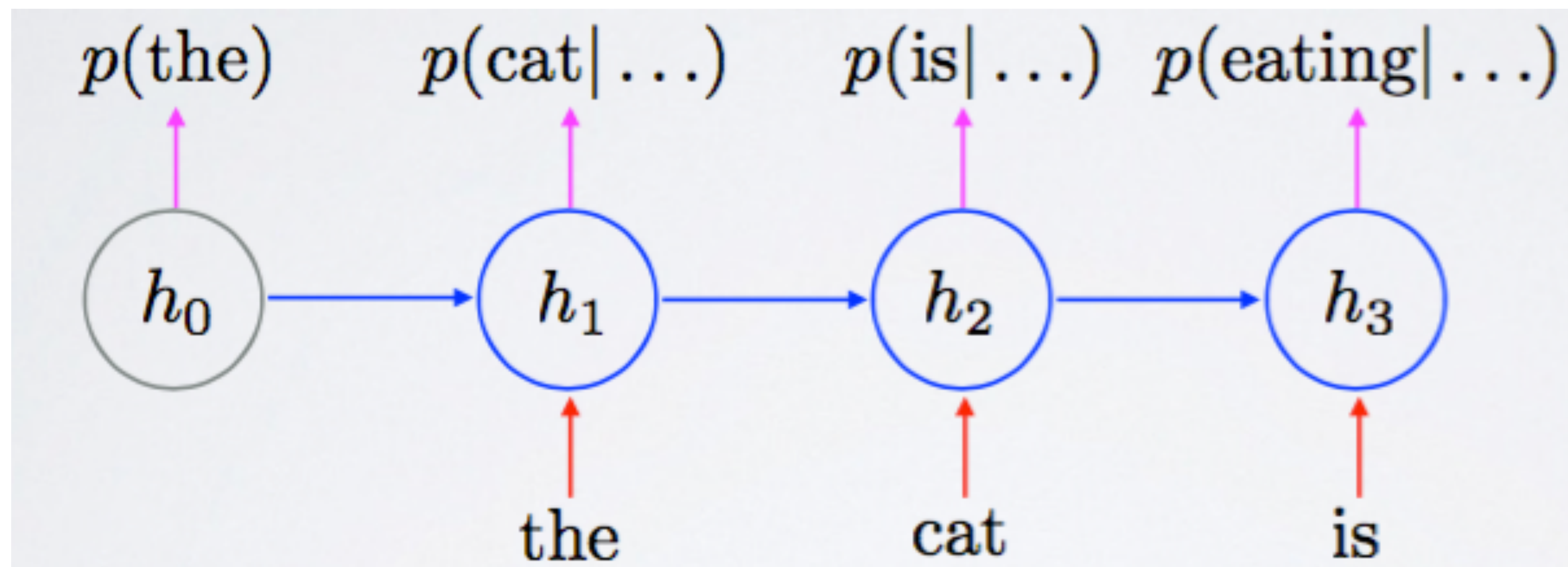
$$128k \cdot 300 + 600 \cdot 128k = 115M$$



Approach	PPL
4-gram count model	163.7
10-gram MLP	136.5
10-gram MLP with 2 layers	130.9

Language Modeling with RNN's

- MLP LM's are still limited in history (use n-gram assumption)
- We would like to use RNN's to model the entire sentence "at once"
- Every input is a 1-hot vector, every output is the LM probabilities as softmax



Language Modeling with RNN's

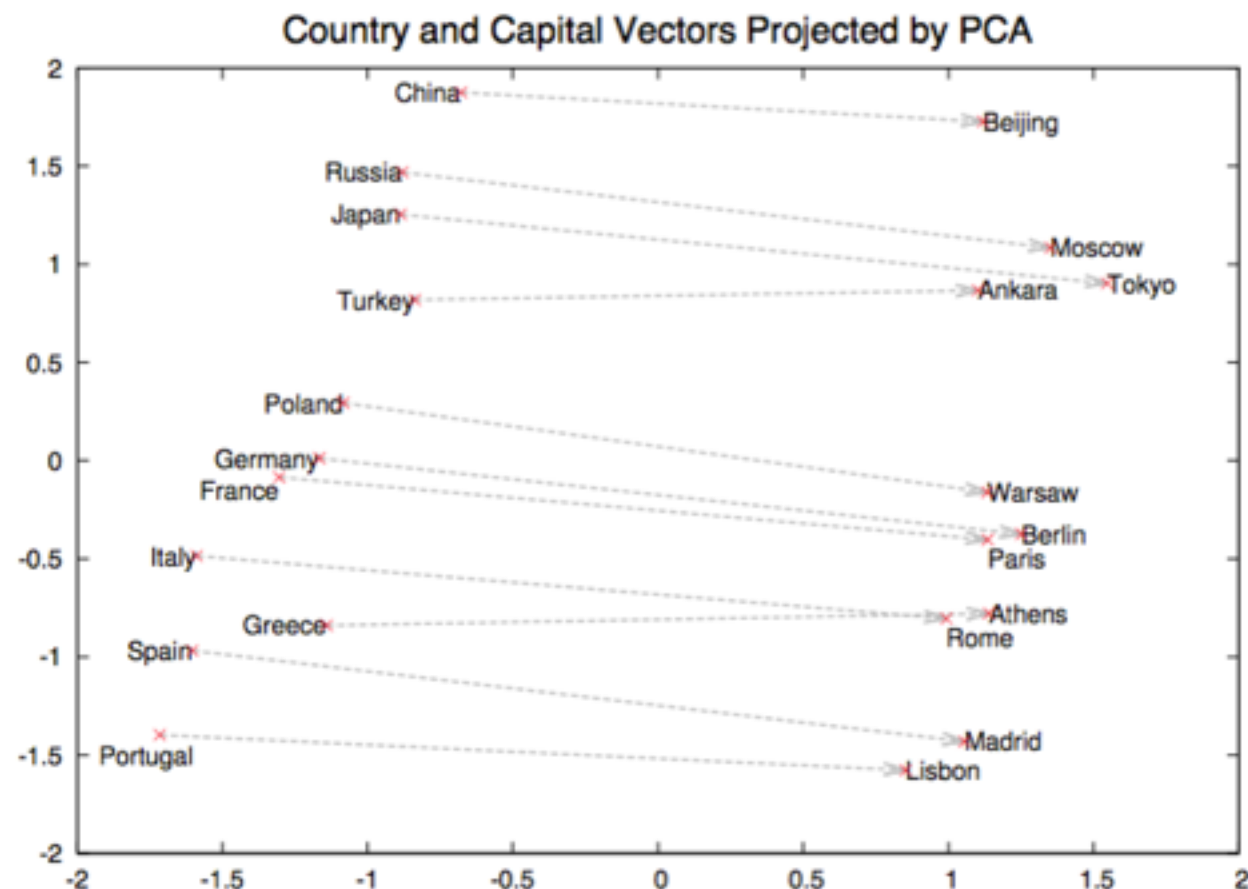
- RNN's provide a significant improvement over previous models
- A price to pay: very long training time
- A solution: use both, but train on different size data sets

Approach	PPL
count model	163.7
10-gram MLP	136.5
RNN	125.2
LSTM-RNN	107.8
10-gram MLP with 2 layers	130.9
LSTM-RNN with 2 layers	100.5

Models	PPL	CPU Time (Order)
Count model	163.7	30 min
MLP	136.5	1 week
LSTM-RNN	107.8	3 weeks

Distributed word representations using word2vec

- As we saw previously, a continuous word representation is learned for each word as part of the network training (many times referred as word embedding)
- These representations were shown as a successful tool for various tasks such as word similarity and word analogies:



word2vec - how it works?

- In word2vec, two similar models are introduced:
- CBOW (left) and skip-gram (right), both can be seen as MLP's
- Have been shown to approximate the PMI matrix [Levy & Goldberg, 2015]

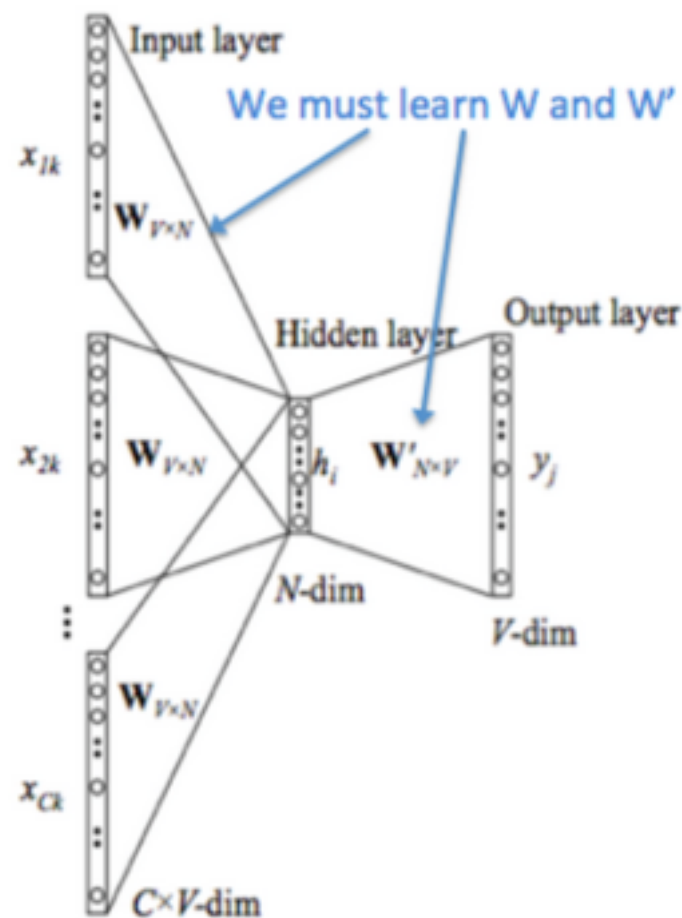


Figure 1: This image demonstrates how CBOW works and how we must learn the transfer matrices

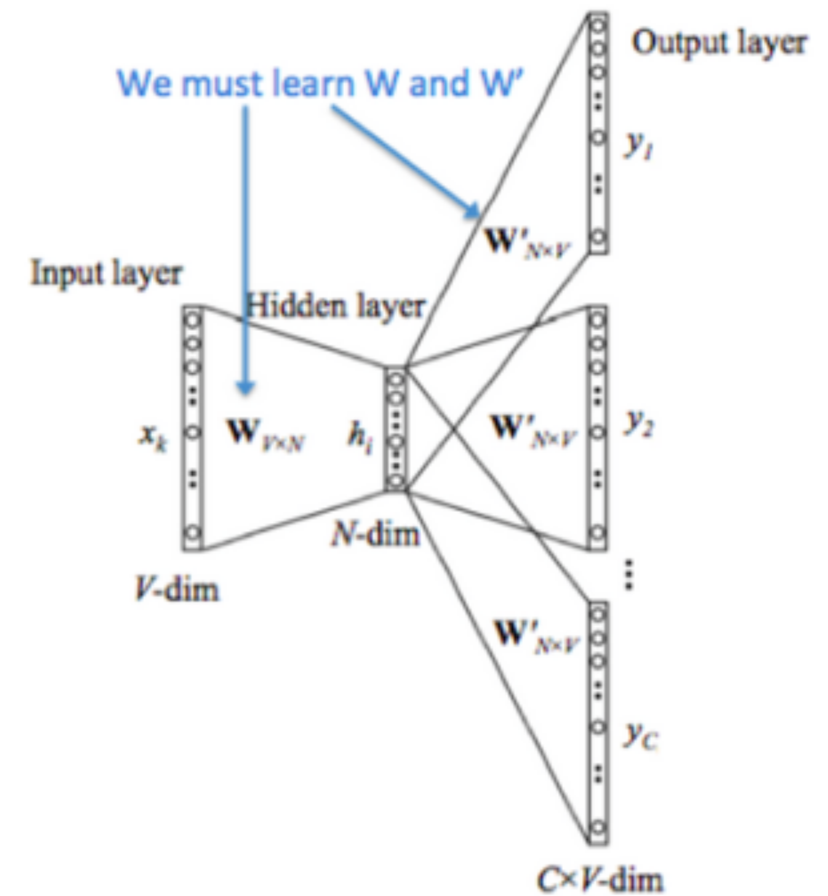
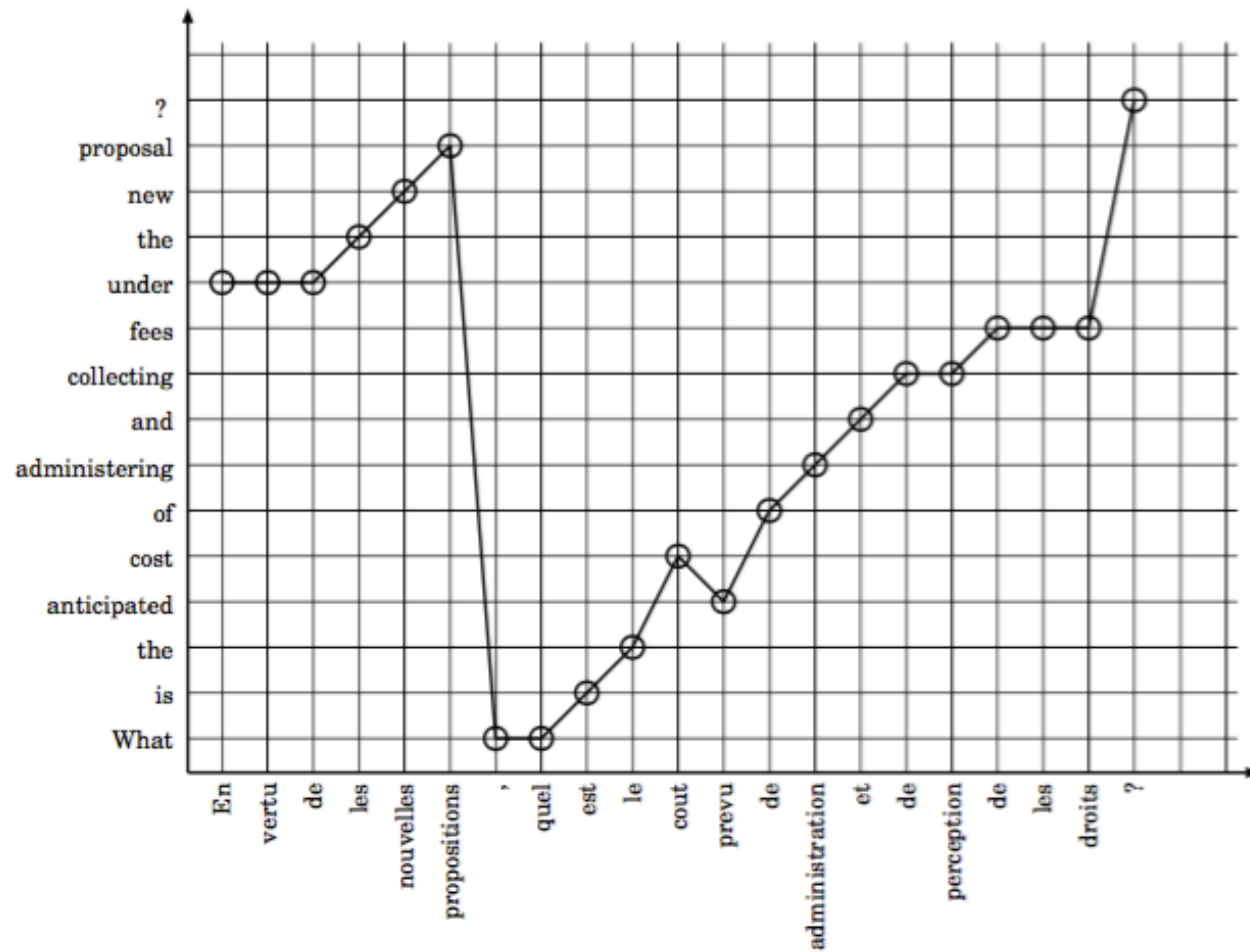


Figure 2: This image demonstrates how Skip-Gram works and how we must learn the transfer matrices

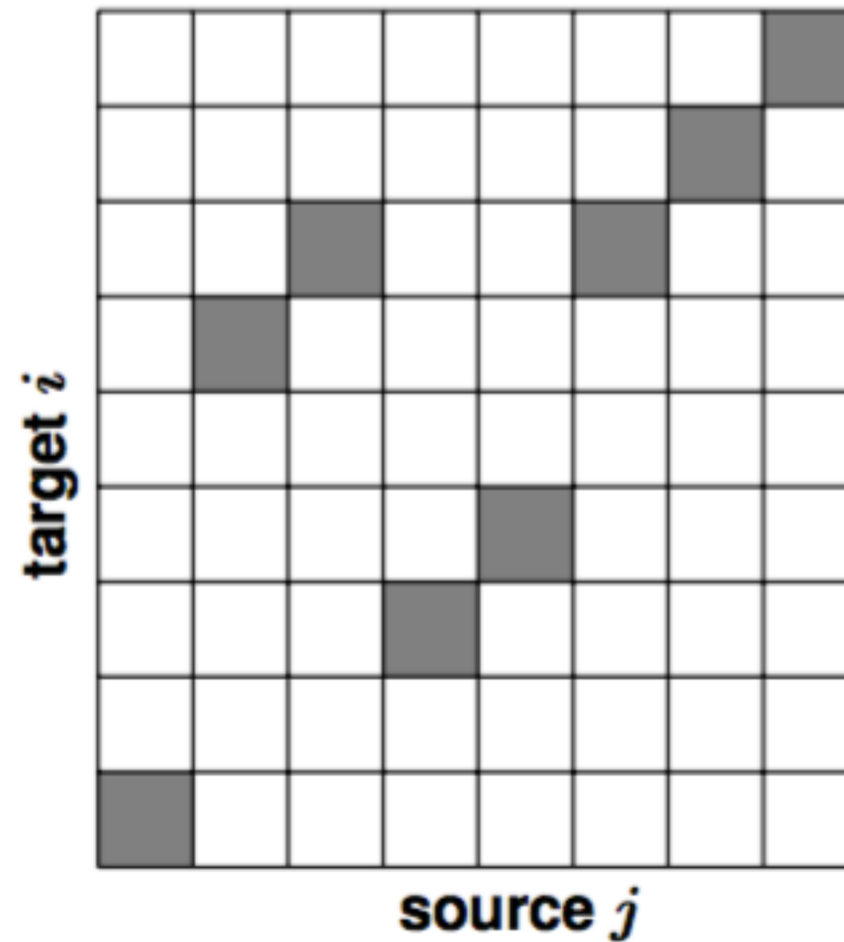
“Conventional” Statistical Machine Translation

Start with parallel text:



“Conventional” Statistical Machine Translation

Learn the alignments:



“Conventional” Statistical Machine Translation

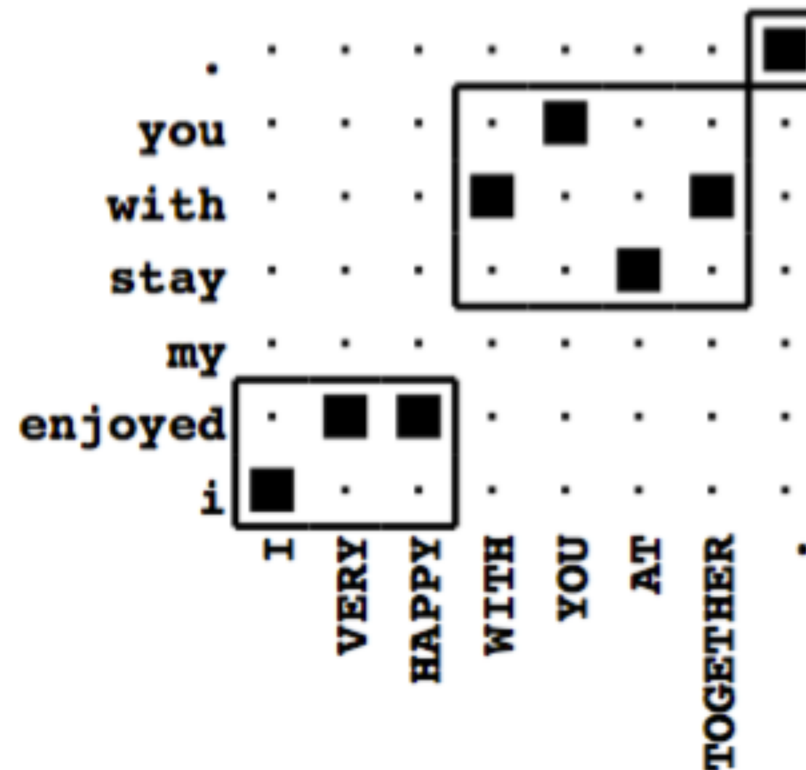
Extract phrase pairs:

source sentence 我 很 高兴 和 你 在 一起 .

gloss notation I VERY HAPPY WITH YOU AT TOGETHER .

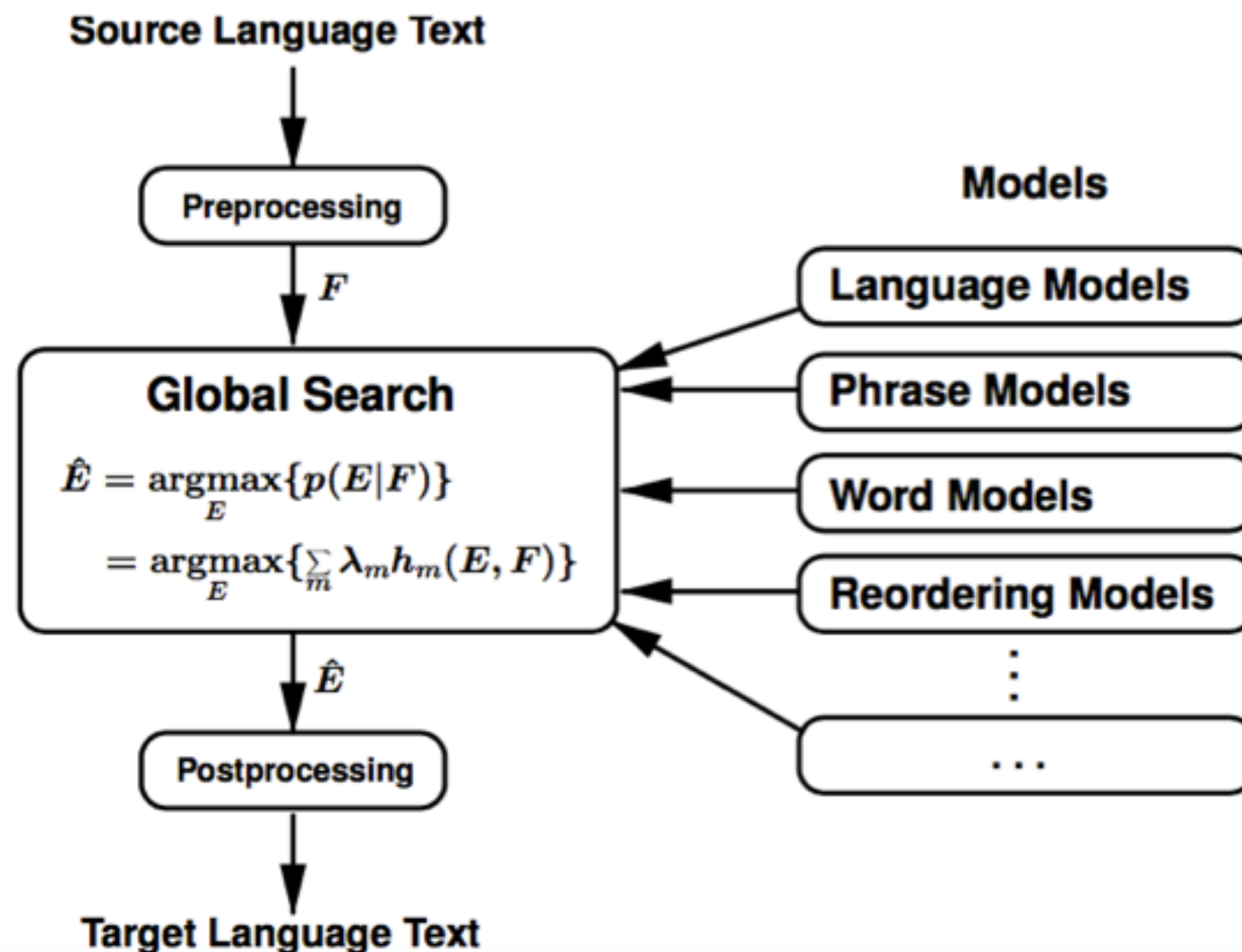
target sentence I enjoyed my stay with you .

Viterbi alignment for $F \rightarrow E$:



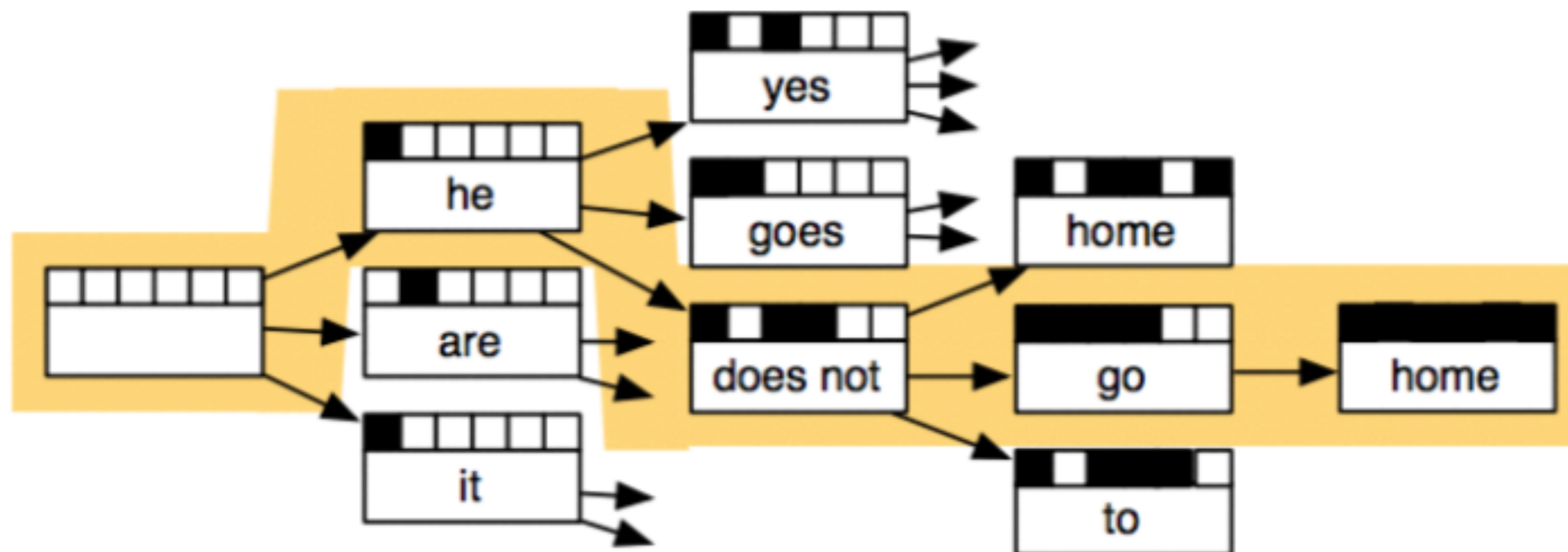
“Conventional” Statistical Machine Translation

Use a log linear model combination to score hypotheses:



“Conventional” Statistical Machine Translation

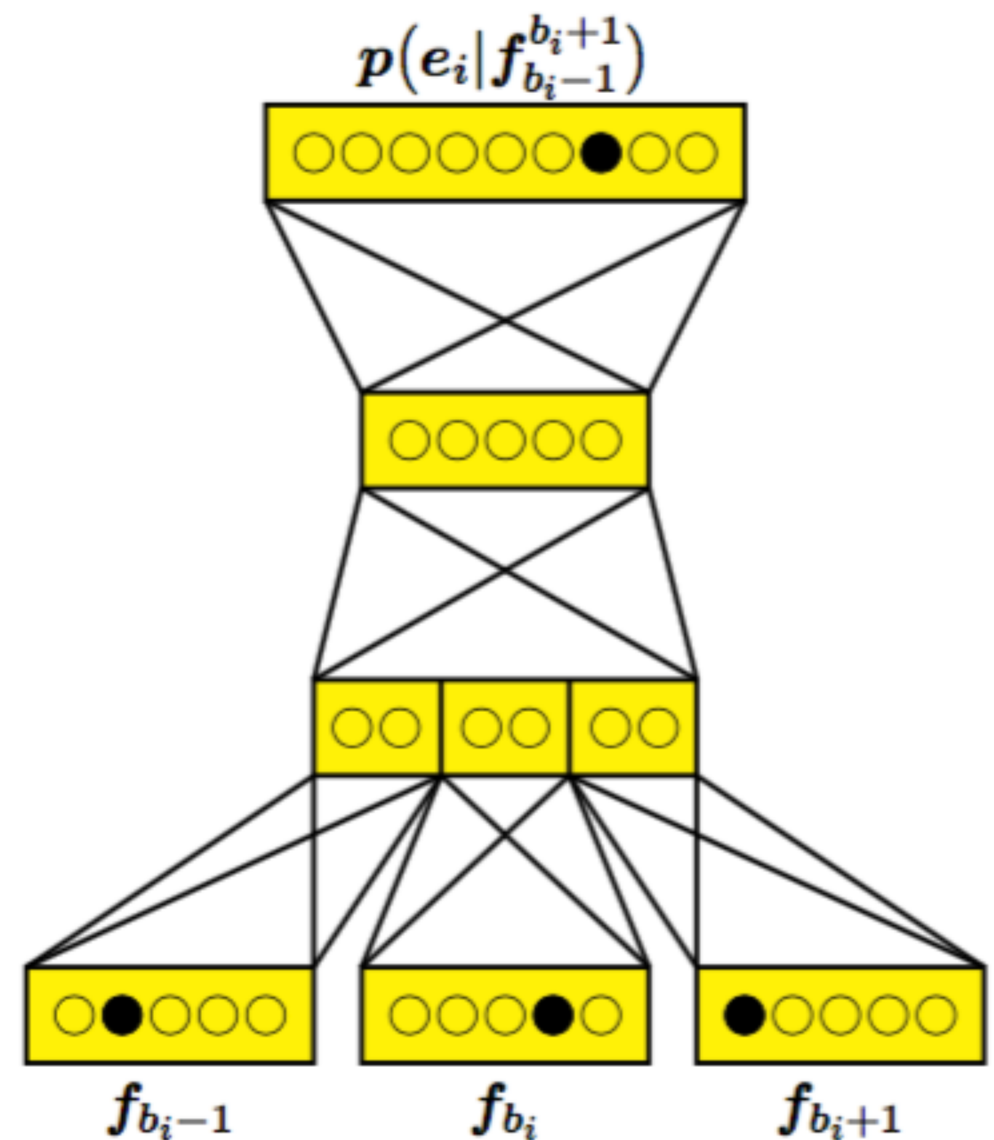
Beam search to output best hypothesis



backtrack from highest scoring complete hypothesis

“Hybrid” Statistical Machine Translation

- Use an MLP to train a translation model
- Inputs are 1-hot encodings of the words in the aligned source language window
- Combine this model with the rest while decoding or rescoring

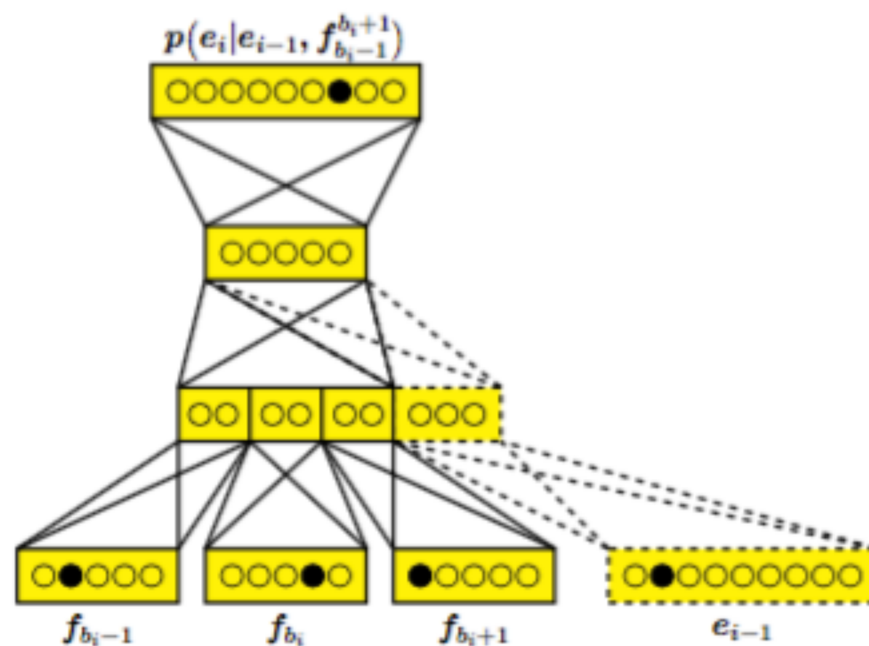


“Hybrid” Statistical Machine Translation

- Another option: Bilingual LM
- Inputs are 1-hot encodings of the words in the aligned source language window and previous words in the translation hypothesis
- ACL 2014 Best Paper [Devlin et al, 2014]

pollutant
this
of
full
offspring
first
their
pump
mothers
these

die Mutter Tiere pumpen ihre ersten Jungen mit diesem Schad Stoff voll



Neural Machine Translation

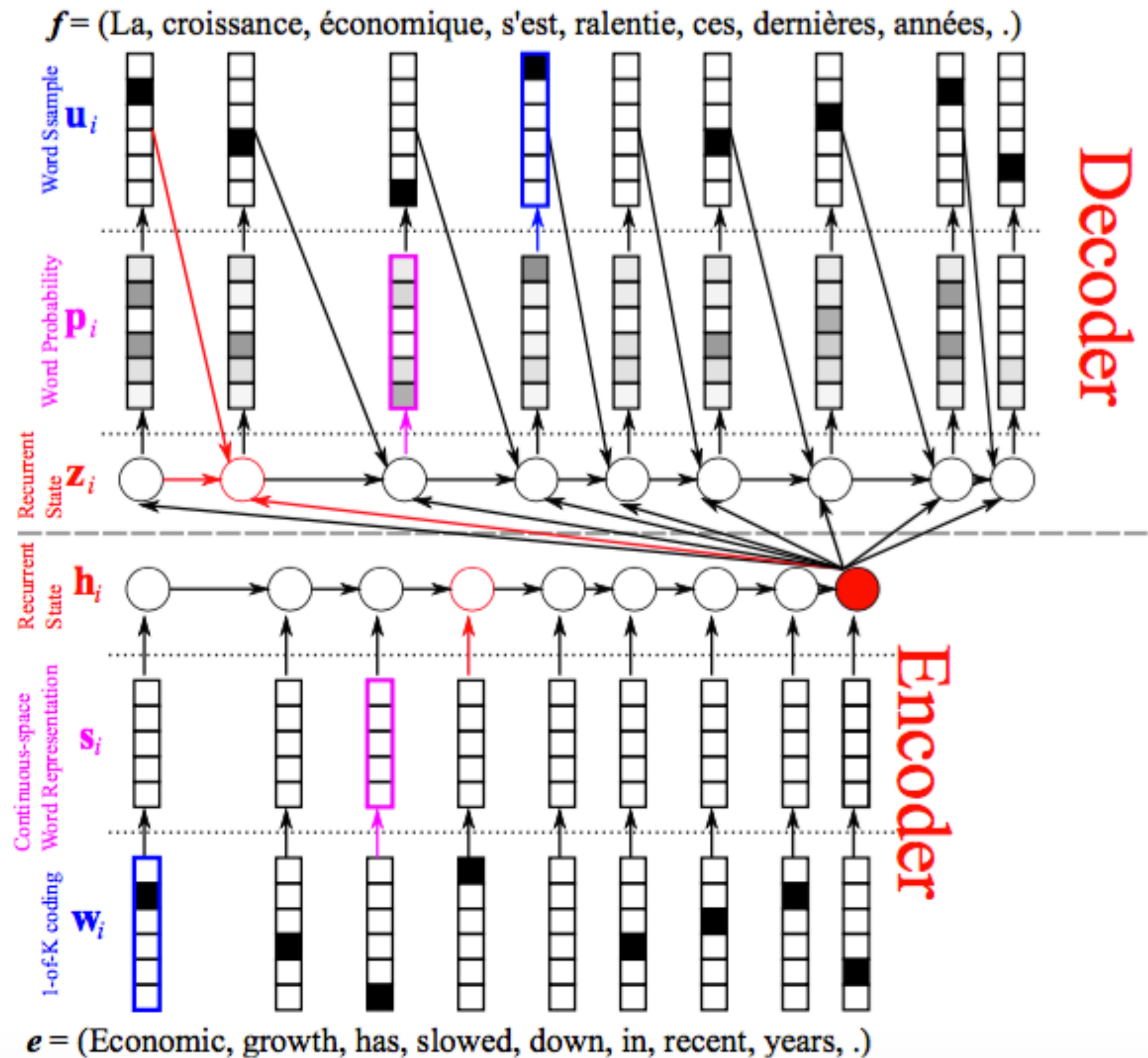
Forcada&Ñeco, 1997;

Castaño&Casacuberta, 1997;

Kalchbrenner&Blunsom,

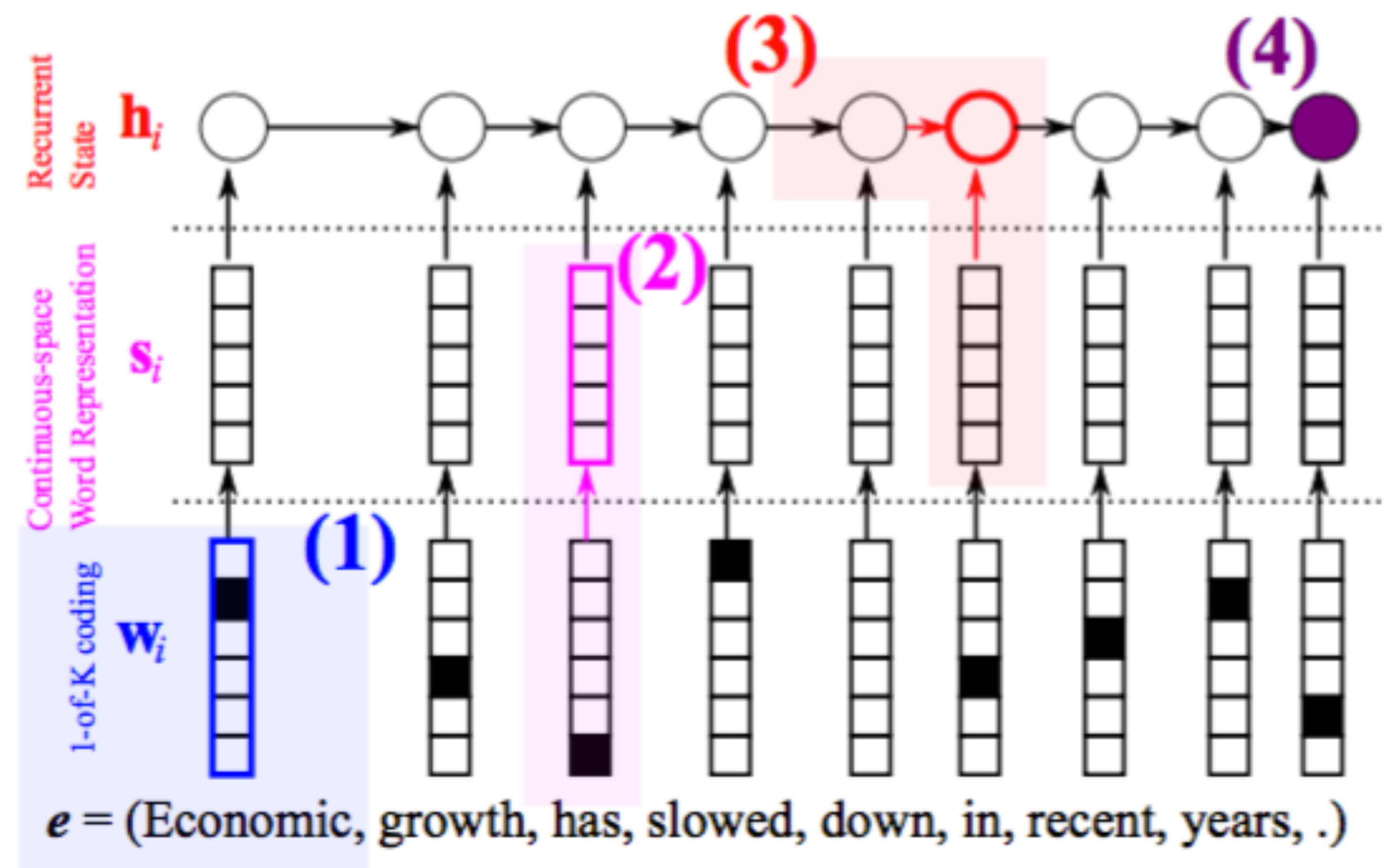
2013; Sutskever et al., 2014;

Cho et al., 2014



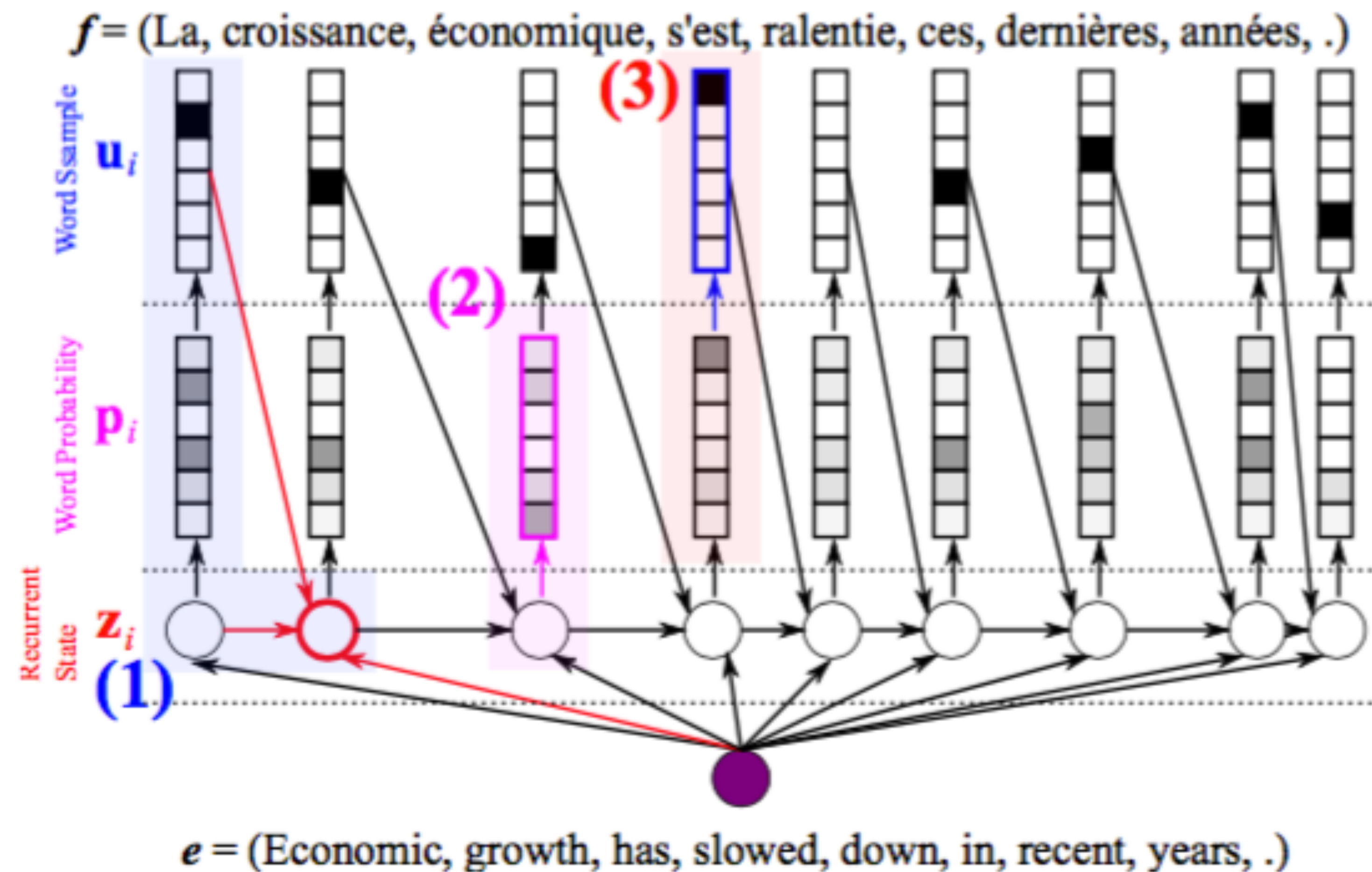
Neural Machine Translation - seq2seq Encoder

1. 1-hot vectors
2. continuous representation (word embeddings)
3. recursively read the words using an RNN (LSTM/GRU)
4. output a sentence representation for the decoder

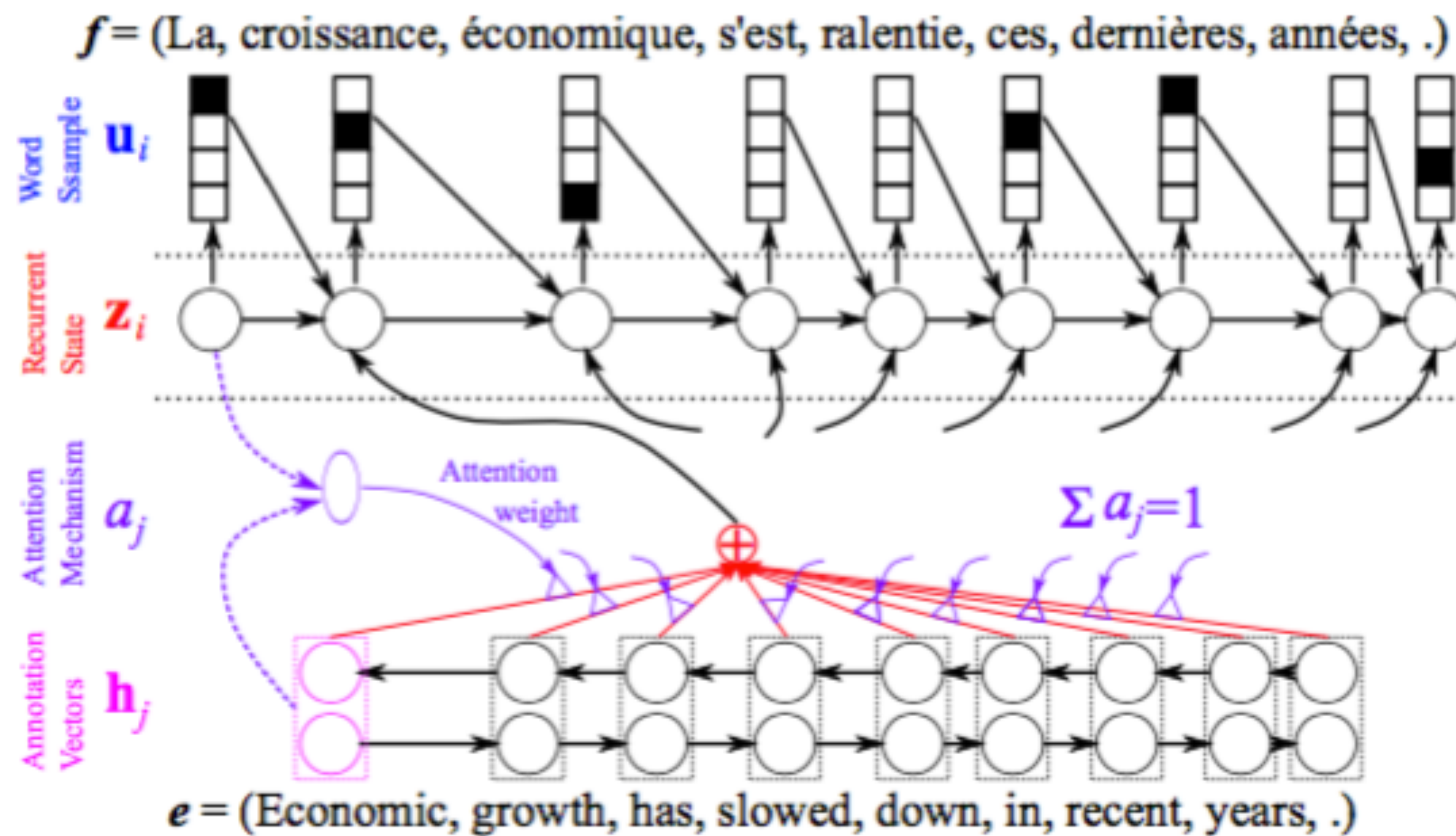


Neural Machine Translation - seq2seq Decoder

1. recursively update the memory
2. compute the next word probabilities
3. sample the next word (sometimes using beam-search)



Neural Machine Translation - Attention



(b) **English→German (WMT-15)**

Model	Note
24.8	Neural MT
24.0	U.Edinburgh, Syntactic SMT
23.6	LIMSI/KIT
22.8	U.Edinburgh, Phrase SMT
22.7	KIT, Phrase SMT

(c) **English→Czech (WMT-15)**

Model	Note
18.3	Neural MT
18.2	JHU, SMT+LM+OSM+Sparse
17.6	CU, Phrase SMT
17.4	U.Edinburgh, Phrase SMT
16.1	U.Edinburgh, Syntactic SMT

Summary

- Pros:
  - Neural network models provide state of the art results on many tasks
  - Continuous representations (rather then 1-hot vectors) - generalize better
  - Better modeling of sequences and context using recurrent architectures
- Cons:
  - Lots of hyper-parameter tuning
  - Harder to interpret model parameters
  - Usually a very long training time, computationally expensive

How Can I Start?

- Y. Goldberg, A Primer on Neural Network Models for Natural Language Processing

- PyCNN Tutorial (+IPython Notebooks!)
- Slides are available at: www.roeeaharoni.com

Questions?

The Israeli Natural Language Processing Meetup

You are invited! (when in Tel Aviv ;)

References

- Y. Goldberg, A Primer on Neural Network Models for Natural Language Processing
- K. Cho, Natural Language Understanding with Distributed Representation
- K. Duh, Deep Learning Tutorial at DL4MT winter school
- H. Ney, Language Modeling and Machine Translation using Neural Networks
- C. Olah, Understanding LSTM Networks
- C. Manning, Computational Linguistics and Deep Learning