MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

CENG 403
Introduction to Deep Learning

Week 13b

Sinan Kalkan

Feedforward through Vanilla RNN

Previously on CENG403

The Vanilla RNN Model

First time-step ($t = 1$):

$$\mathbf{h}_1 = tanh\left(W^{xh} \cdot \mathbf{x}_1 + W^{hh} \cdot \mathbf{h}_0\right)$$

$$\hat{\mathbf{y}}_1 = softmax\left(W^{hy} \cdot \mathbf{h}_1\right)$$

$$\mathcal{L}_1 = CE(\hat{\mathbf{y}}_1, \mathbf{y}_1)$$

In general:

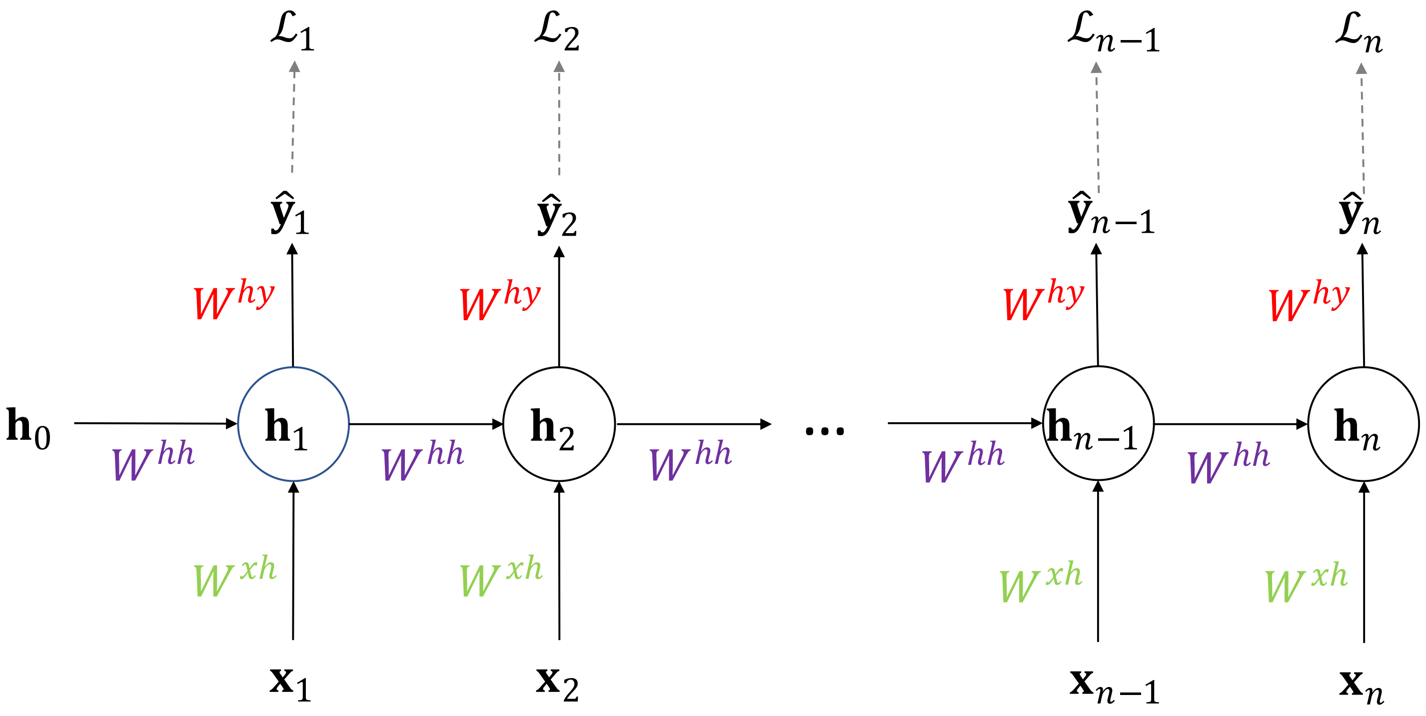
$$\mathbf{h}_t = tanh\left(W^{xh} \cdot \mathbf{x}_t + W^{hh} \cdot \mathbf{h}_{t-1}\right)$$

$$\hat{\mathbf{y}}_t = softmax\left(W^{hy} \cdot \mathbf{h}_t\right)$$

$$\mathcal{L}_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

In total:

$$\mathcal{L} = \sum_t \mathcal{L}_t$$



Character-level Text Modeling

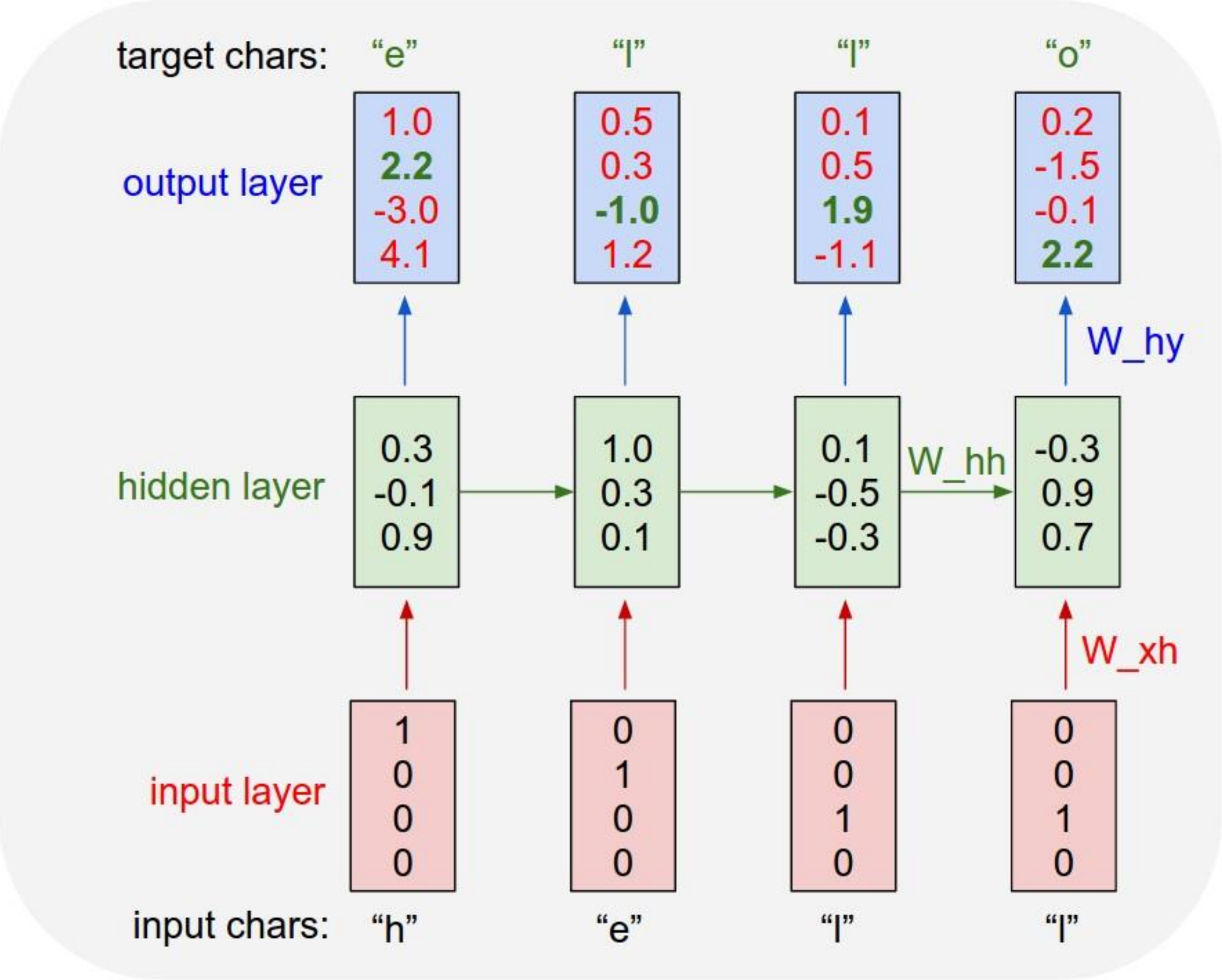
Previously on CENG403

- Problem definition: Find c_{n+1} given $c_1, c_2, \ldots, c_n$.
- Modelling (autoregressive language modeling):
$$p(c_{n+1} \mid c_n, \ldots, c_1)$$
- In general, we just take the last N characters:
$$p(c_{n+1} \mid c_n, \ldots, c_{n-(N-1)})$$
- Learn $p(c_{n+1} = 'a' \mid 'Ankar')$ from data such that
$$p(c_{n+1} = 'a' \mid 'Ankar') > p(c_{n+1} = 'o' \mid 'Ankar')$$

A simple scenario

Previously on CENG403

- Alphabet: h, e, l, o
- Text to train to predict: "hello"



http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Previously on CENG403

Sampling: Greedy

- Greedy sampling: Take the most likely word at each step

```
from numpy import array
from numpy import argmax

# greedy decoder
def greedy_decoder(data):
    # index for largest probability each row
    return [argmax(s) for s in data]

# define a sequence of 10 words over a vocab of 5 words
data = [[0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1]]
data = array(data)
# decode sequence
result = greedy_decoder(data)
print(result)
```

Running the example outputs a sequence of integers that could then be mapped back to words in the vocabulary.

```
[4, 0, 4, 0, 4, 0, 4, 0, 4, 0]
```

Code: https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/

Sampling: Beam Search

- What happens if we want *k* most likely sequences instead of one?
- Beam search: Consider *k* most likely words at each step, and expand search.

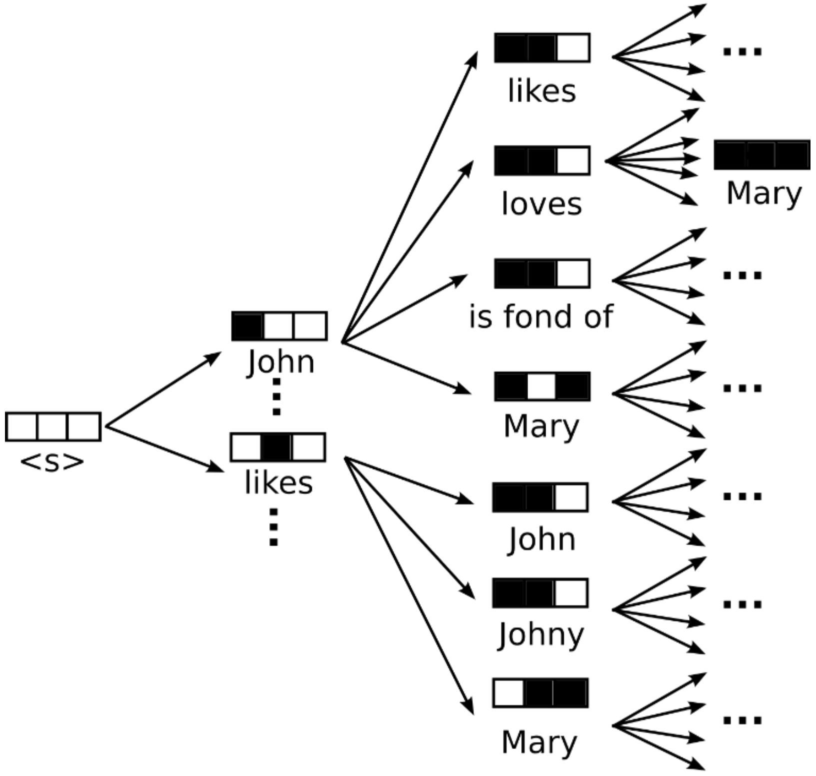


Figure: http://mttalks.ufal.ms.mff.cuni.cz/index.php

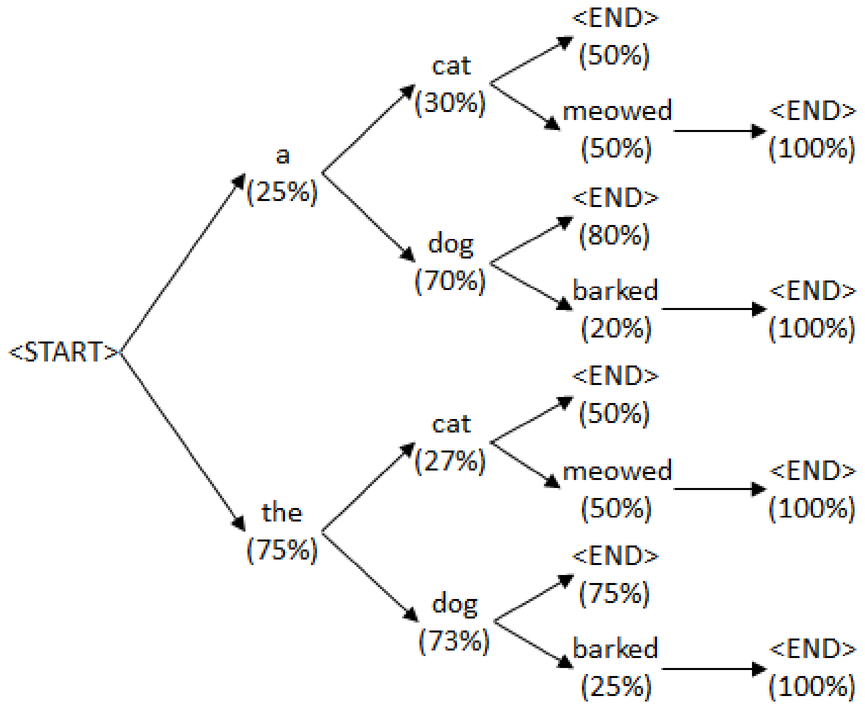


Figure: https://geekyisawesome.blogspot.com.tr/2016/10/using-beam-search-to-generate-most.html

Previously on CENG403

Sampling: Beam Search

- Beam search: Consider *k* most likely words at each step, and expand search.

  (take log for numerical stability; take –log() for minimizing the score)

```python
from math import log
from numpy import array
from numpy import argmax

# beam search
def beam_search_decoder(data, k):
    sequences = [[list(), 0.0]]
    # walk over each step in sequence
    for row in data:
        all_candidates = list()
        # expand each current candidate
        for i in range(len(sequences)):
            seq, score = sequences[i]
            for j in range(len(row)):
                candidate = [seq + [j], score - log(row[j])]
                all_candidates.append(candidate)
        # order all candidates by score
        ordered = sorted(all_candidates, key=lambda tup:tup[1])
        # select k best
        sequences = ordered[:k]
    return sequences
```

```python
# define a sequence of 10 words over a vocab of 5 words
data = [[0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1]]
data = array(data)
# decode sequence
result = beam_search_decoder(data, 3)
# print result
for seq in result:
    print(seq)
```

```
[[4, 0, 4, 0, 4, 0, 4, 0, 4, 0], 6.931471805599453]
[[4, 0, 4, 0, 4, 0, 4, 0, 4, 1], 7.154615356913663]
[[4, 0, 4, 0, 4, 0, 4, 0, 3, 0], 7.154615356913663]
```

Code: https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/

Word-level Text Modeling

Previously on CENG403

- Problem definition: Find ω_{n+1} given $\omega_1, \omega_2, \ldots, \omega_n$.
- Modelling:

$$p(\omega_{n+1} \mid \omega_n, \ldots, \omega_1)$$

- In general, we just take the last N words:

$$p(\omega_{n+1} \mid \omega_n, \ldots, \omega_{n-(N-1)})$$

- Learn $p(\omega_{n+1} = 'Turkey' \mid 'Ankara\ is\ the\ capital\ of\ ')$ from data such that:

$$p(\omega_{n+1} = 'Turkey' \mid 'Ankara\ is\ the\ capital\ of\ ') \quad > \quad p(\omega_{n+1} = 'UK' \mid 'Ankara\ is\ the\ capital\ of\ ')$$

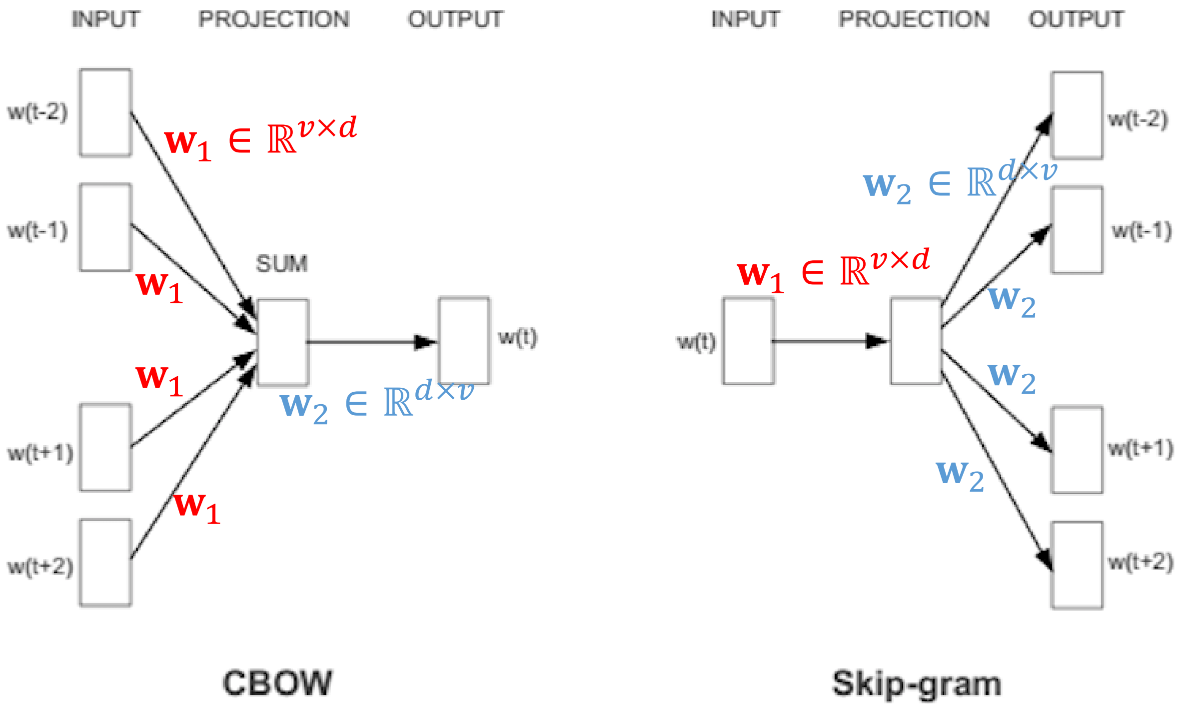
Previously on CENG403

Two different ways to train

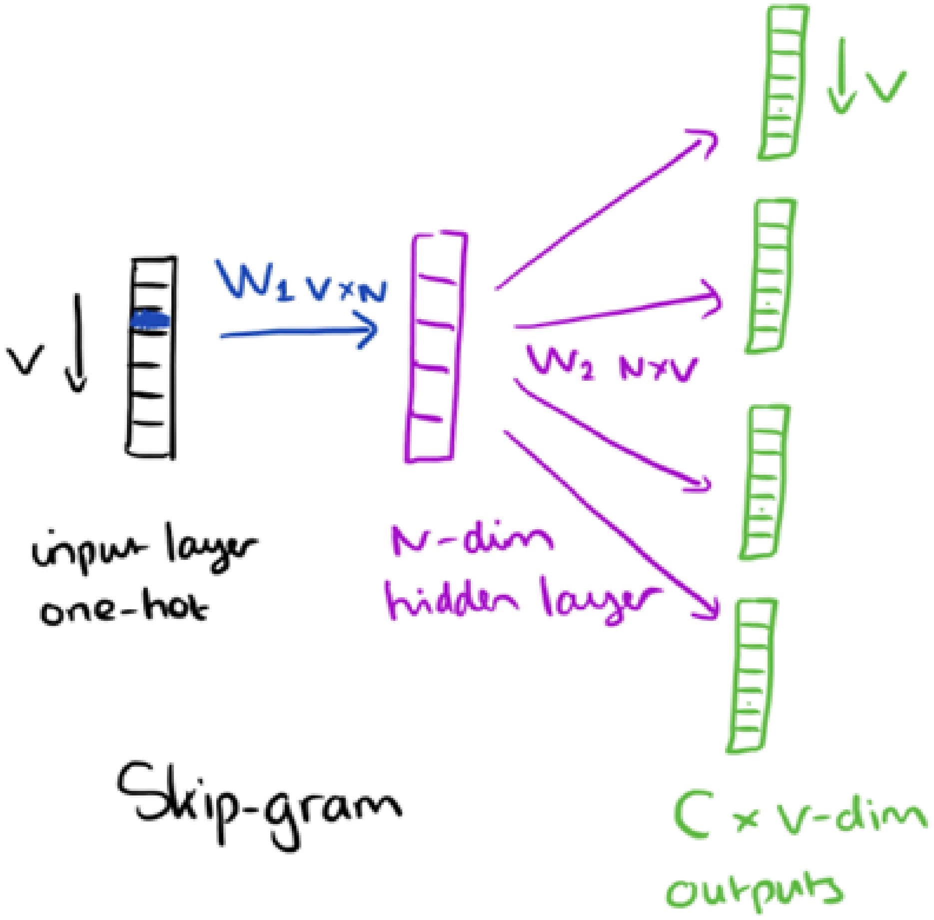
1. Using context to predict a target word (~ continuous bag-of-words)
2. Using word to predict a target context (skip-gram)

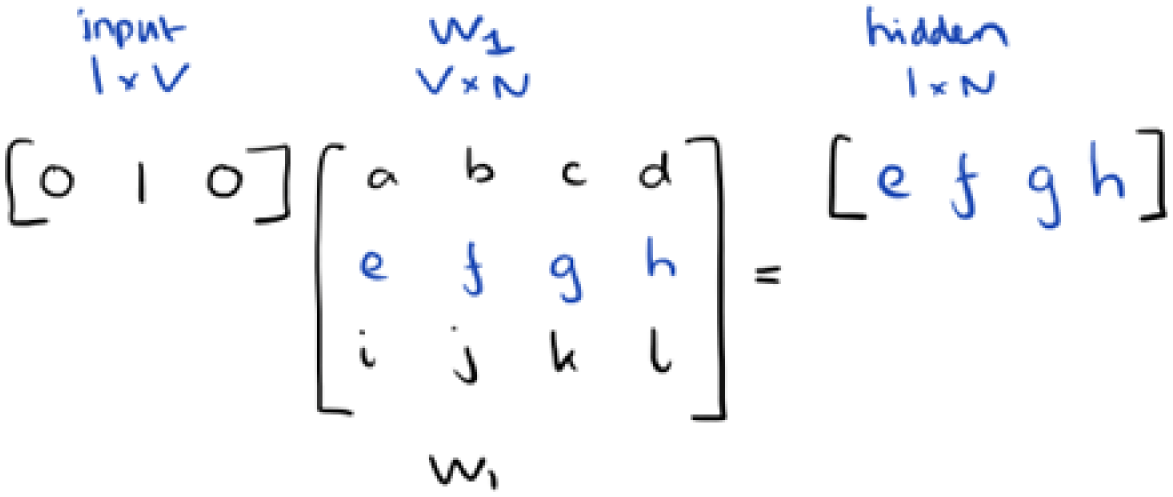
- If the vector for a word cannot predict its context, learning adjusts the mapping to the vector space
- Since similar words should predict the same or similar contexts, their vector representations should end up being similar

v: vocabulary size
d: hidden representation size



Note that the weight matrix is a look-up table





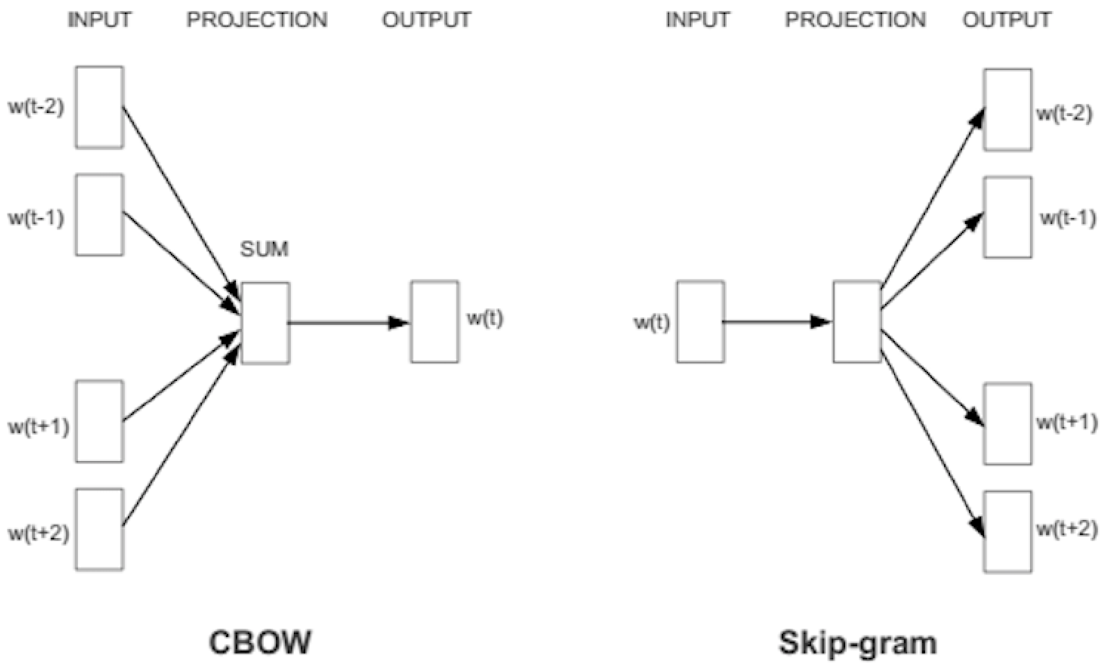
https://medium.com/@zafaralibagh6/a-simple-word2vec-tutorial-61e64e38a6a1

Today

- Recurrent Neural Networks (RNNs)
  - Image captioning
  - Machine translation
  - Echo State Networks
  - Attention in RNNs

Some notes on word2vec methods

- CBOW is called continuous BOW since the context is regarded as a BOW and it is continuous.
- In both approaches, the networks are composed of linear units
- The output units are usually normalized with the softmax
- According to Mikolov:
  - *“Skip-gram: works well with small amount of the training data, represents well even rare words or phrases.*
  - *CBOW: several times faster to train than the skip-gram, slightly better accuracy for the frequent words”*



http://deeplearning4j.org/word2vec

Byte-pair Encoding (BPE)

Example from: https://huggingface.co/learn/llm-course/en/chapter6/5

- Represent frequent byte-pairs as tokens
- E.g., given the corpus:

  Corpus: ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

  our vocabulary would be:

  ("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
- “ug” and “un” can be recognized to be very frequent. So, combine them:

  Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]

  Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)

A Comparison among Embeddings

- Out-of-vocabulary (OOV) words
  - Word embeddings struggle with out-of-vocabulary (OOV) words
  - Char embeddings are better with OOV words as they use chars. However, char embeddings fail at capturing semantically meaningful entities larger than chars
- Vocabulary size
  - Word embeddings have large vocabulary size
  - Char embeddings are better in this regard
  - BPE provides a good balance
- Sub-word (prefix, suffix, root/stem) semantics
  - BPE handles sub-words better
- Language specificity
  - Word embeddings are language specific

a man is playing tennis on a tennis court

a train is traveling down the tracks at a train station

a cake with a slice cut out of it

a bench sitting on a patch of grass next to a sidewalk

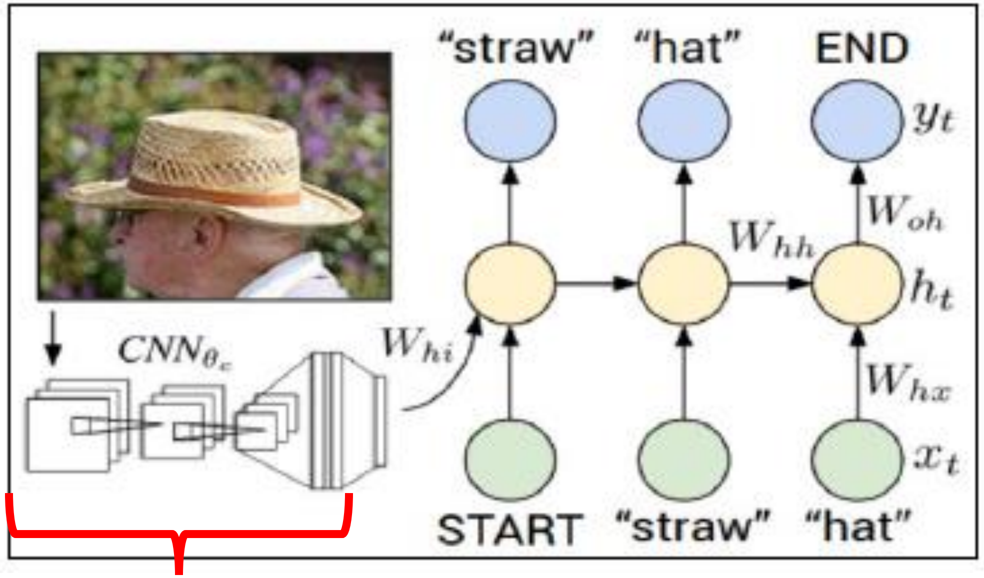
Fig: https://github.com/karpathy/neuraltalk2

Example: Image Captioning

Demo video

https://vimeo.com/146492001

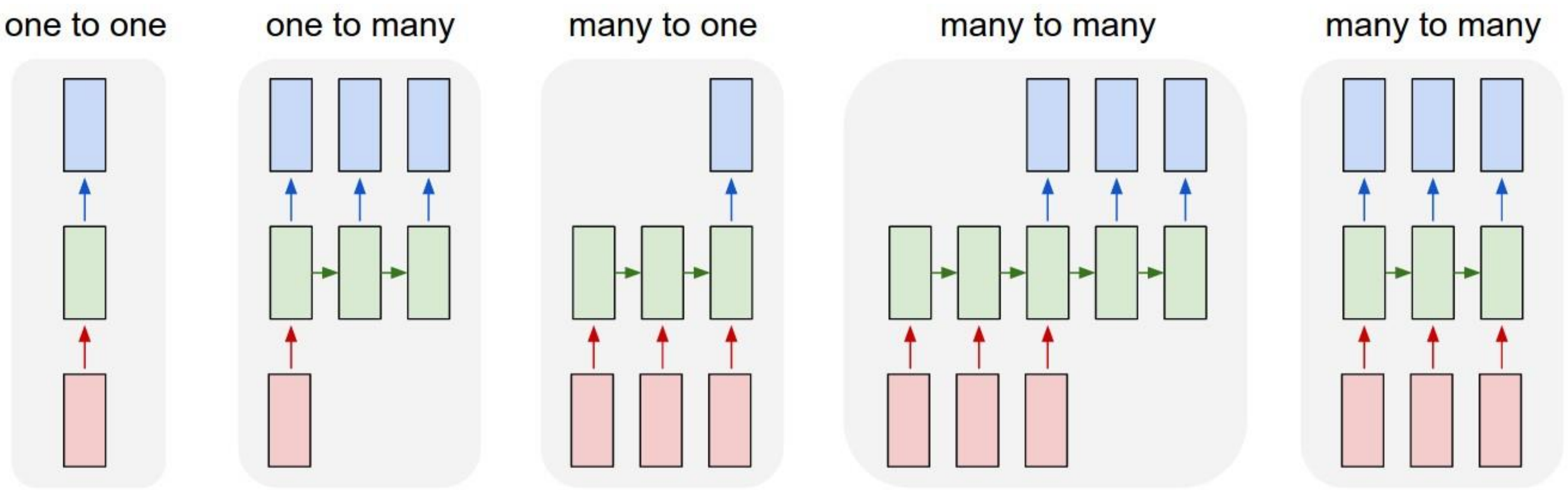
Overview



Pre-trained word embedding is also used

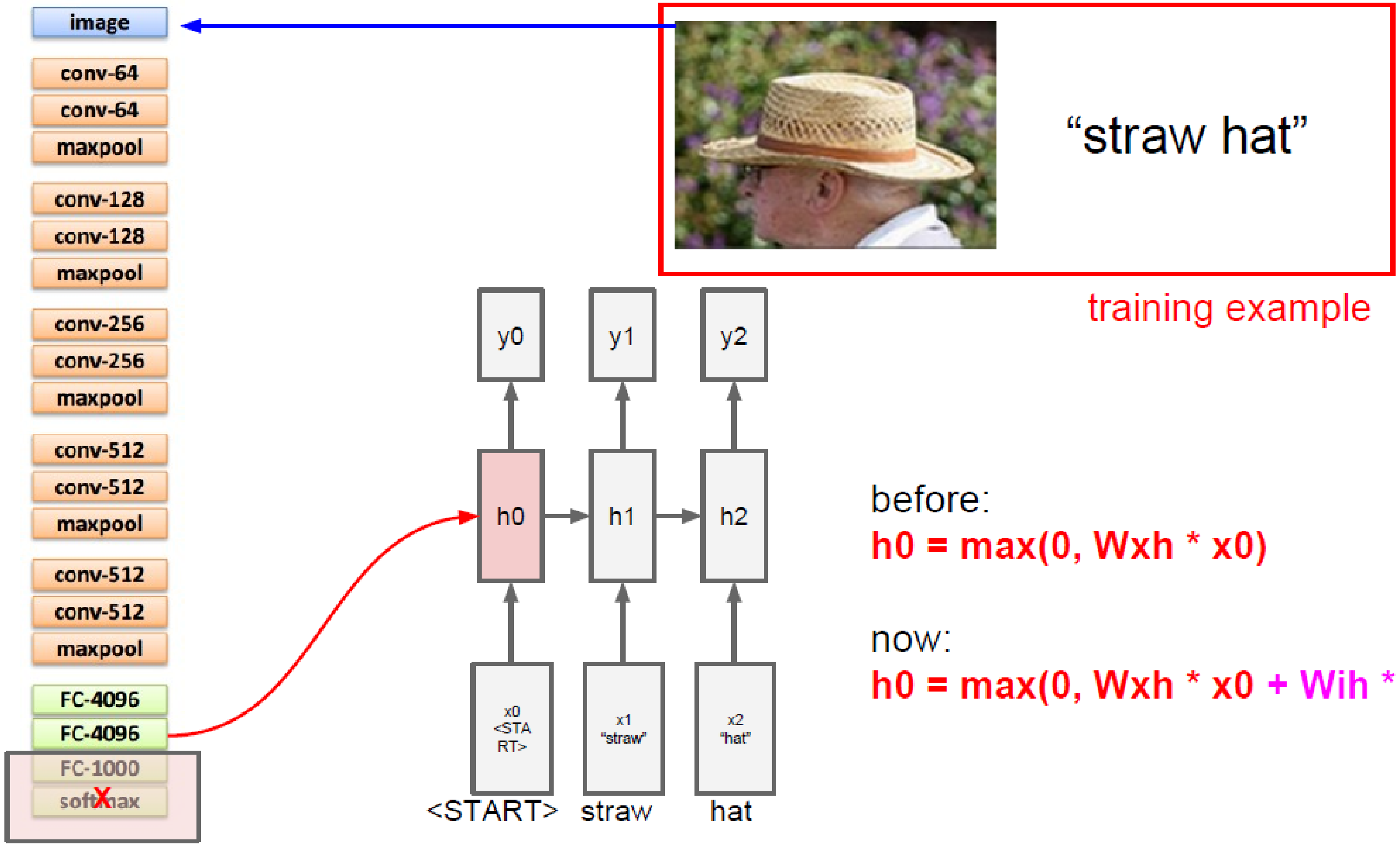
Pre-trained CNN
(e.g., on imagenet)

Image: Karpathy



http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Training



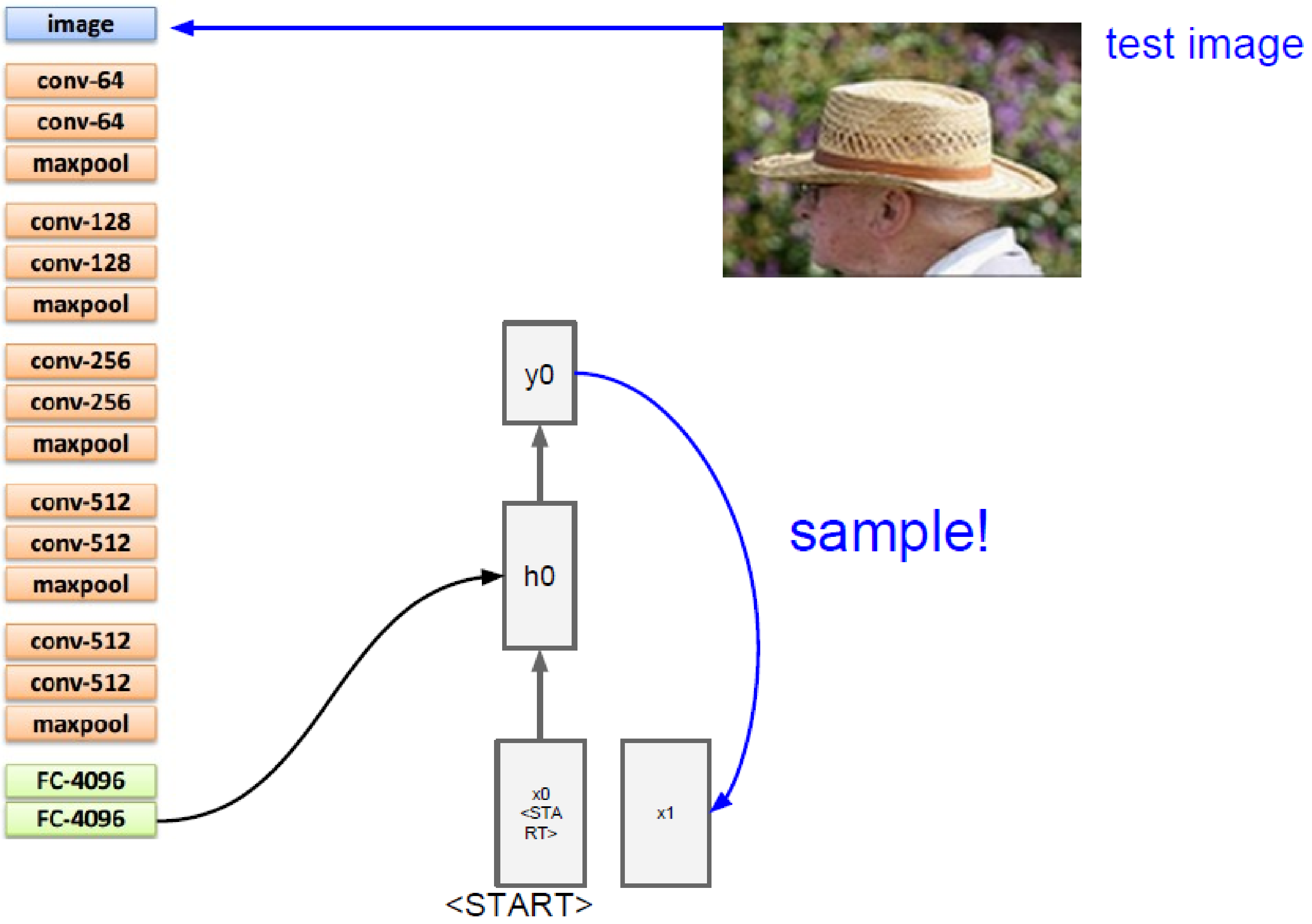
before:

h0 = max(0, Wxh * x0)

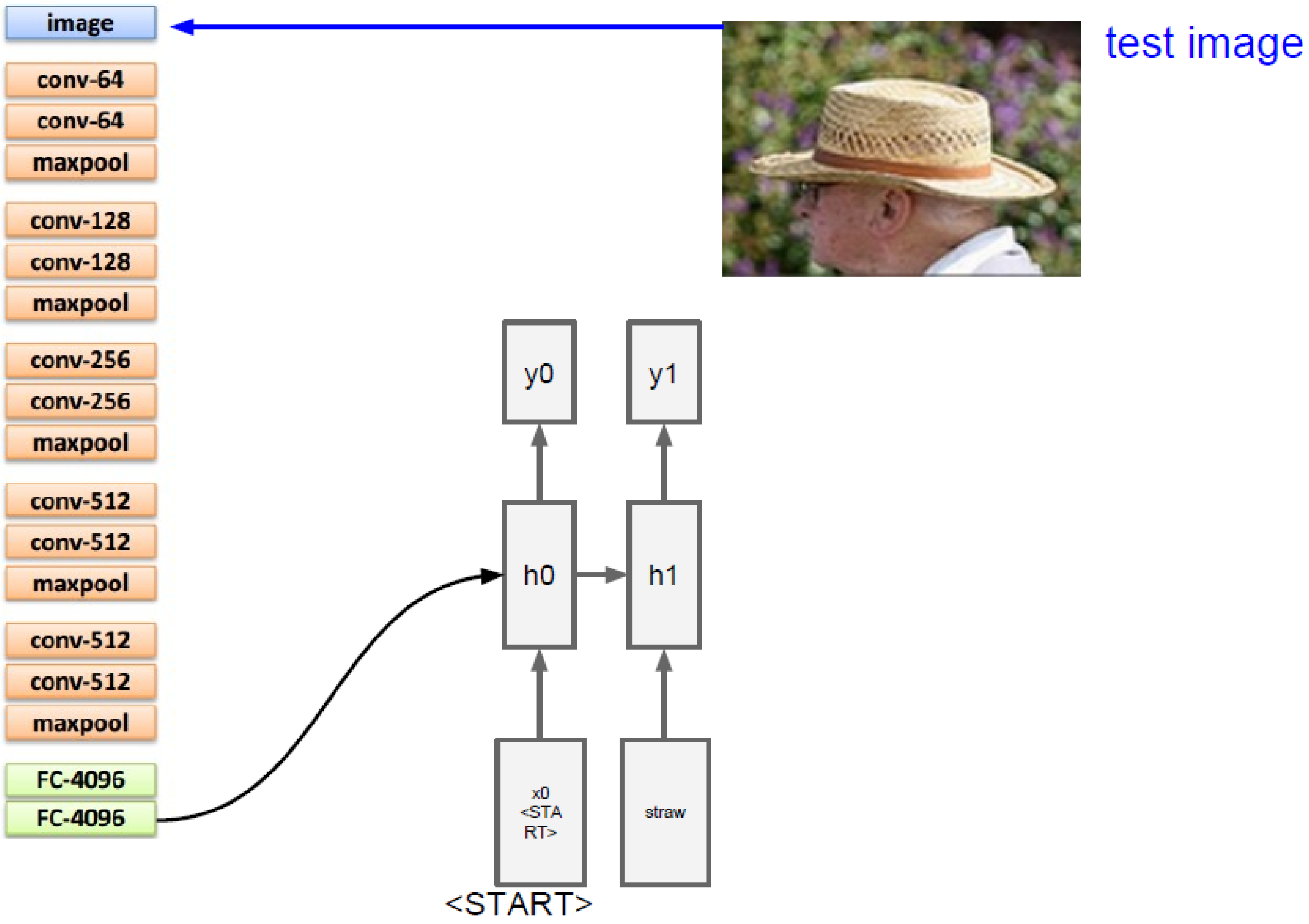
now:

h0 = max(0, Wxh * x0 + Wih * v)

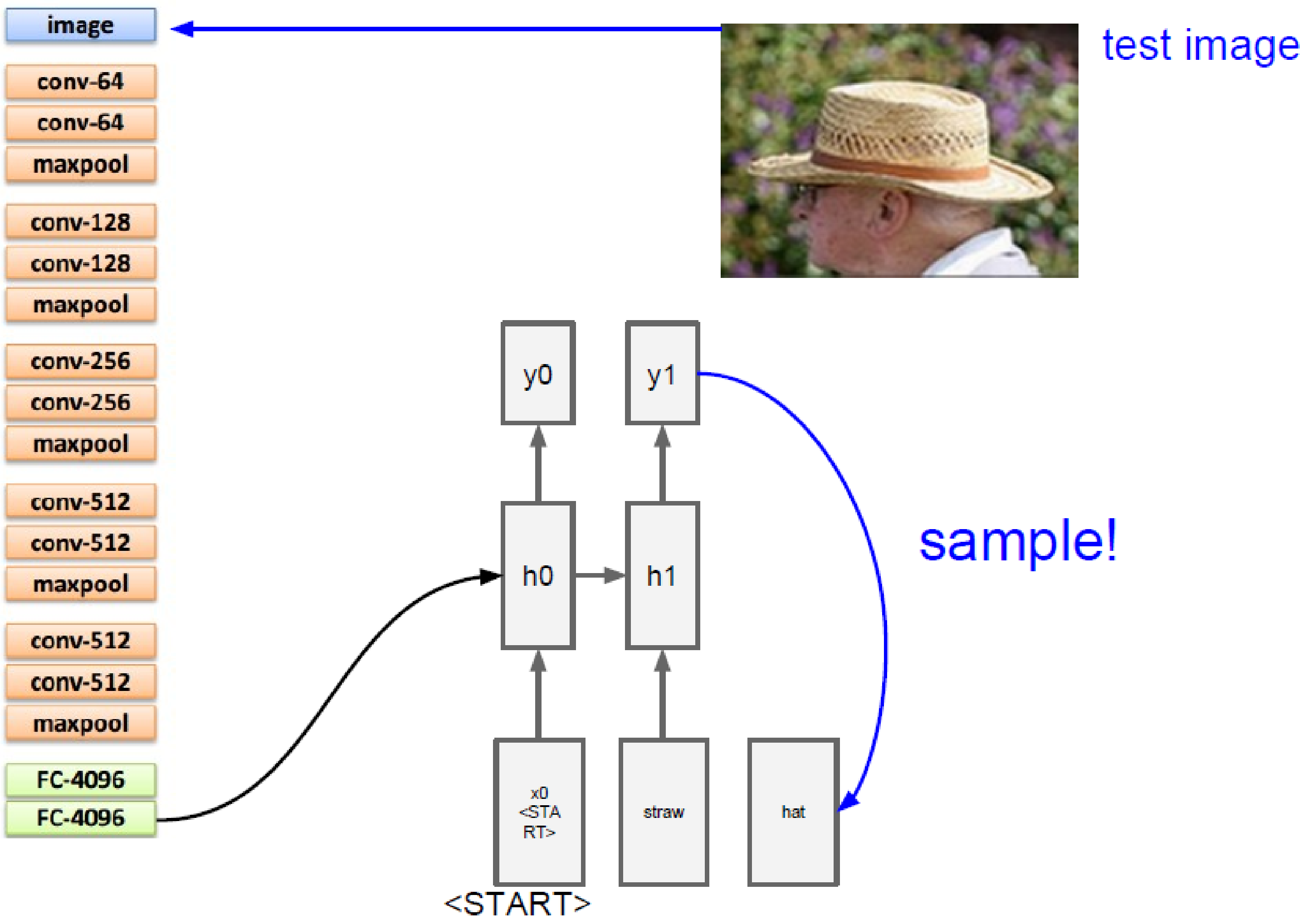
Slide: Karpathy



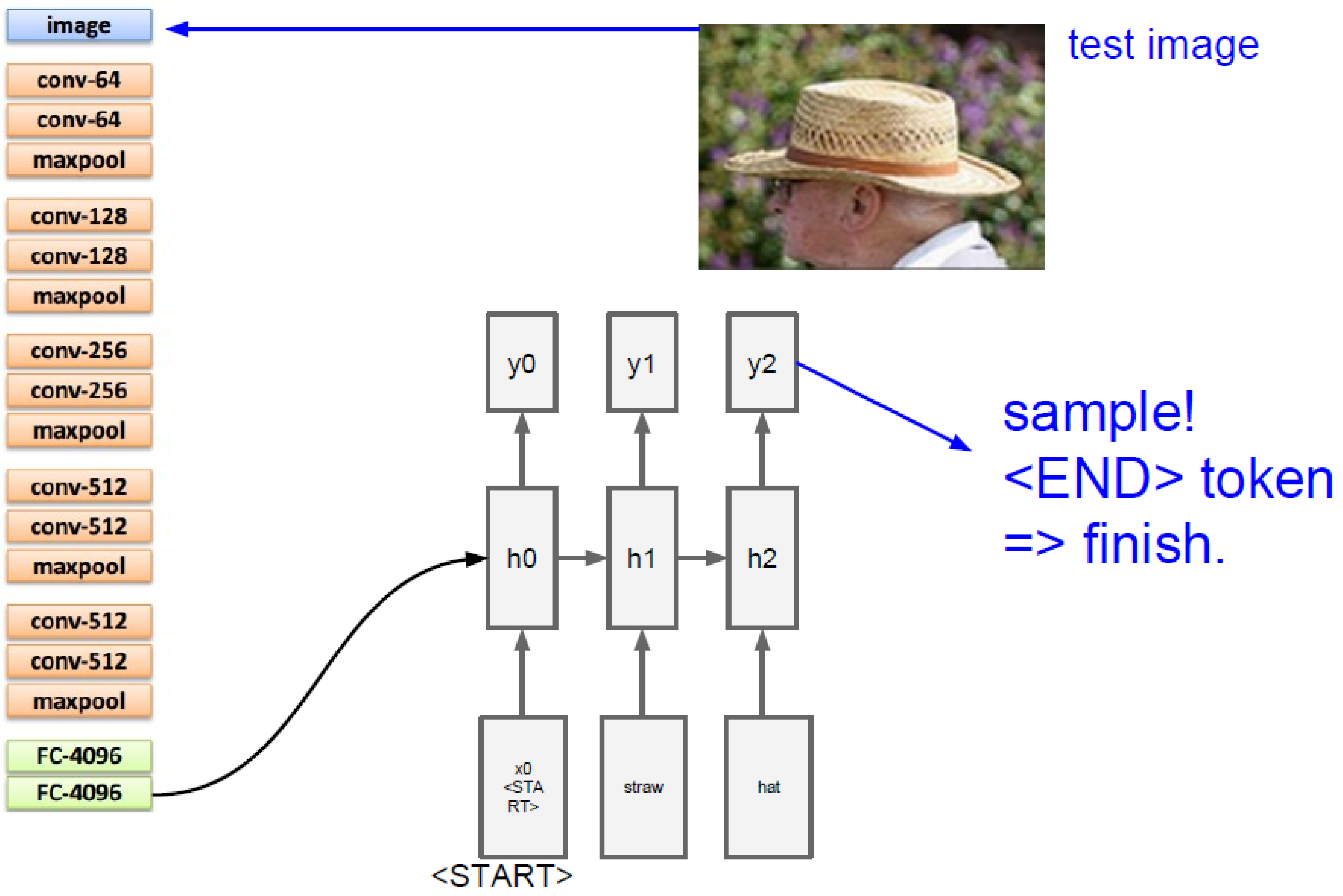
Slide: Karpathy



Slide: Karpathy



Slide: Karpathy



Slide: Karpathy

Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

Kyunghyun Cho
Bart van Merriënboer **Caglar Gulcehre**
Université de Montréal
firstname.lastname@umontreal.ca

Dzmitry Bahdanau
Jacobs University, Germany
d.bahdanau@jacobs-university.de

Fethi Bougares **Holger Schwenk**
Université du Maine, France
firstname.lastname@lium.univ-lemans.fr

Yoshua Bengio
Université de Montréal, CIFAR Senior Fellow
find.me@on.the.web

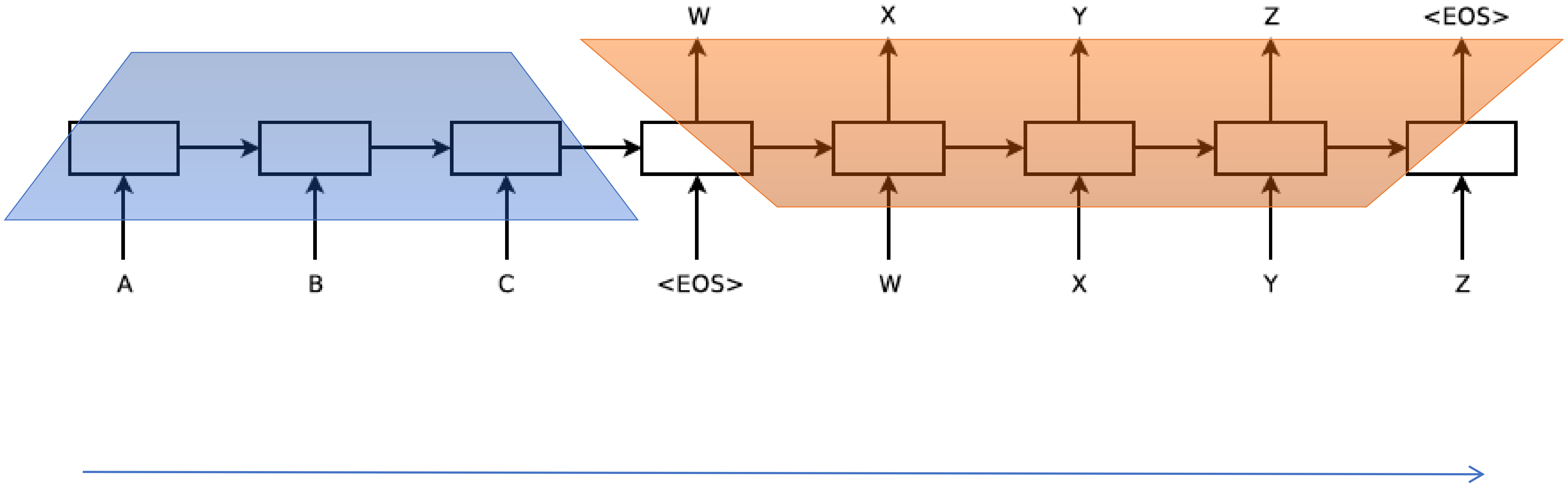
2014

Example: Neural Machine Translation

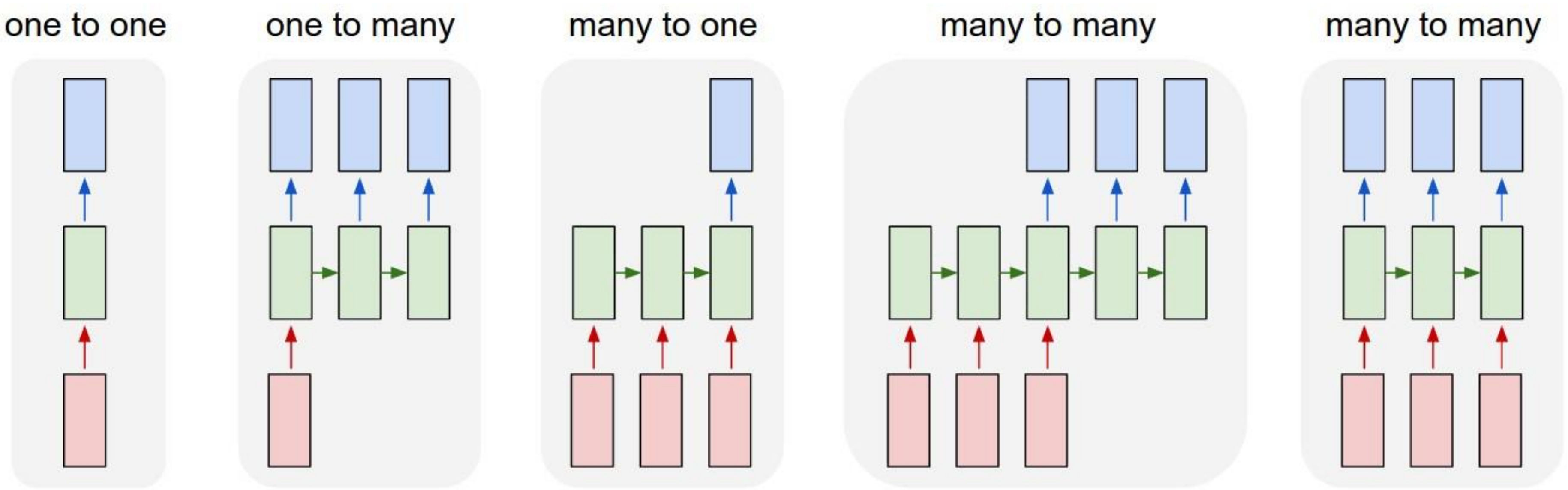
Neural Machine Translation

- Model

Each box is an LSTM or GRU cell.

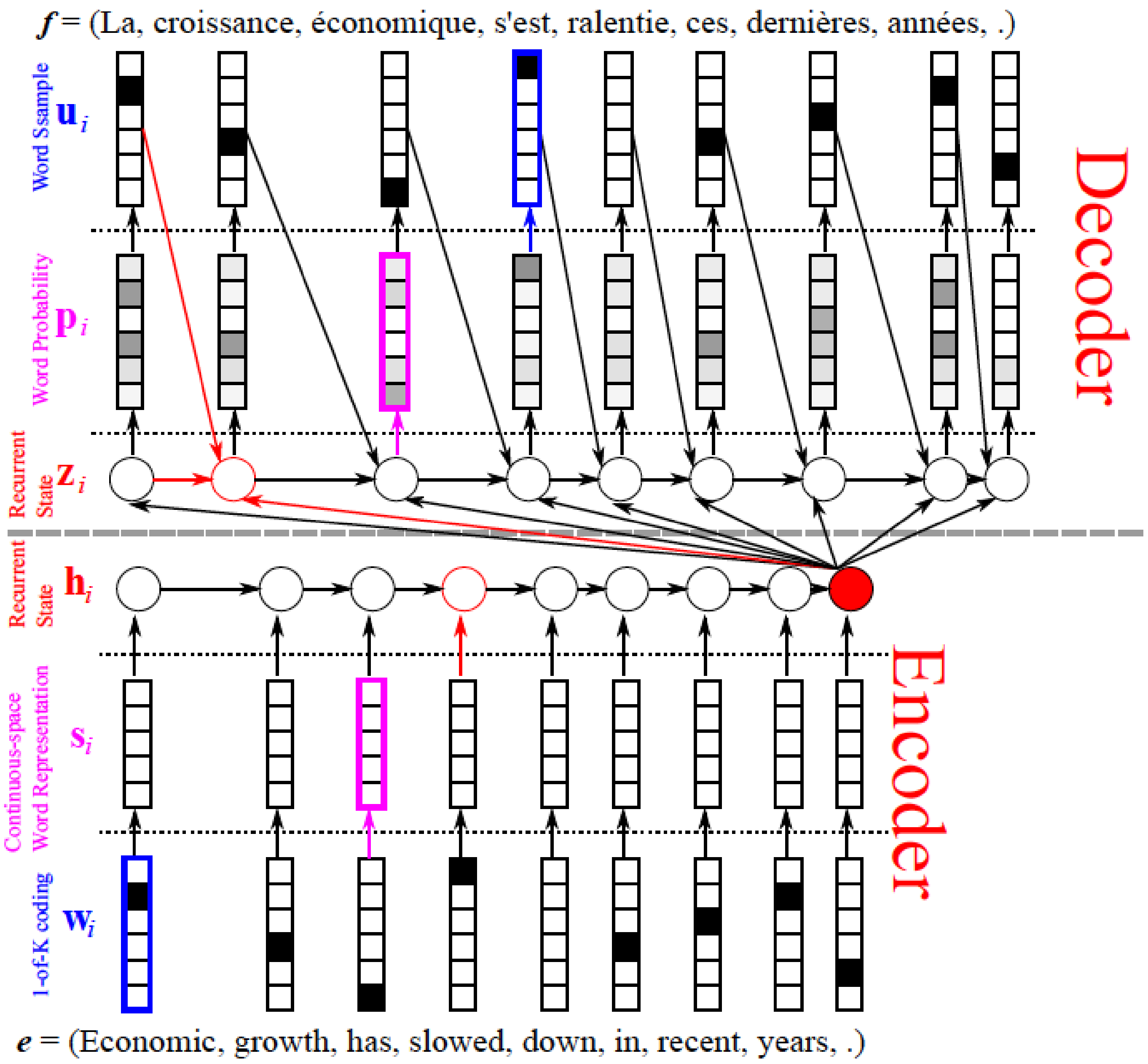


Sutskever et al. 2014



http://karpathy.github.io/2015/05/21/rnn-effectiveness/

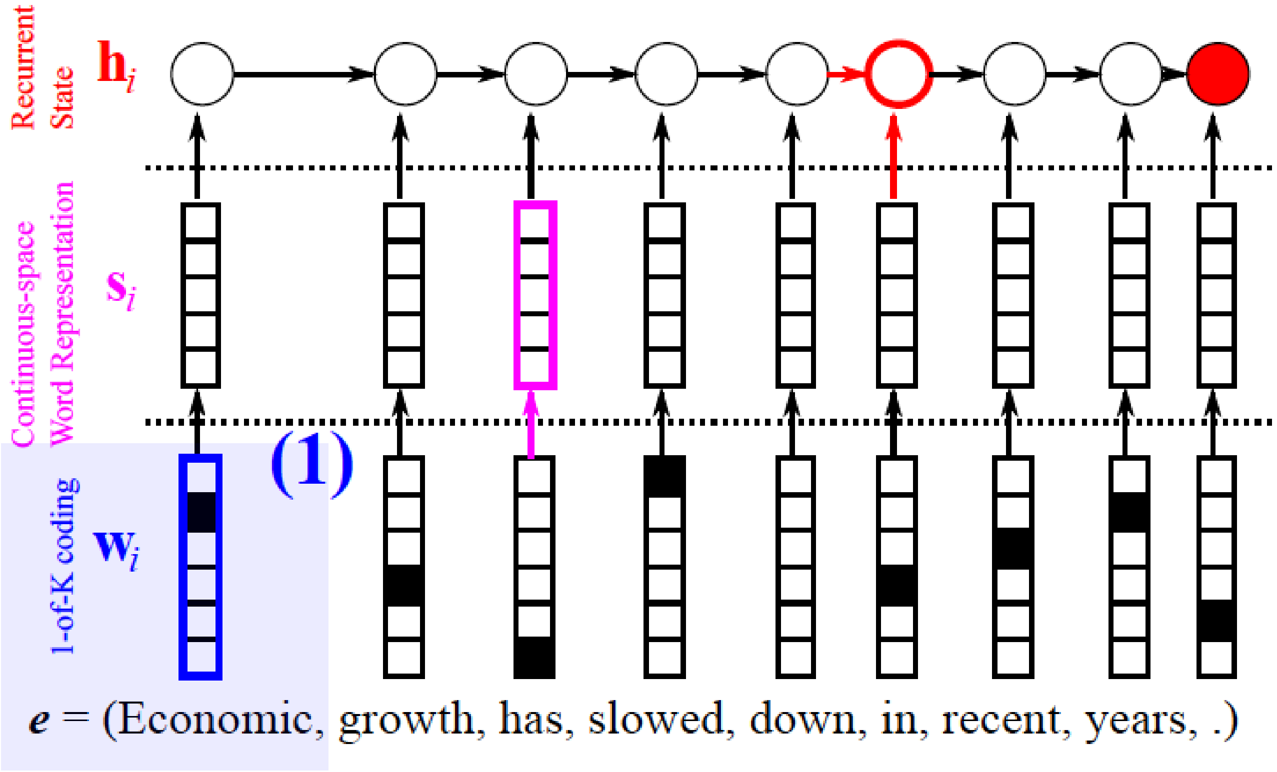
Neural Machine Translation



Cho: From Sequence Modeling to Translation

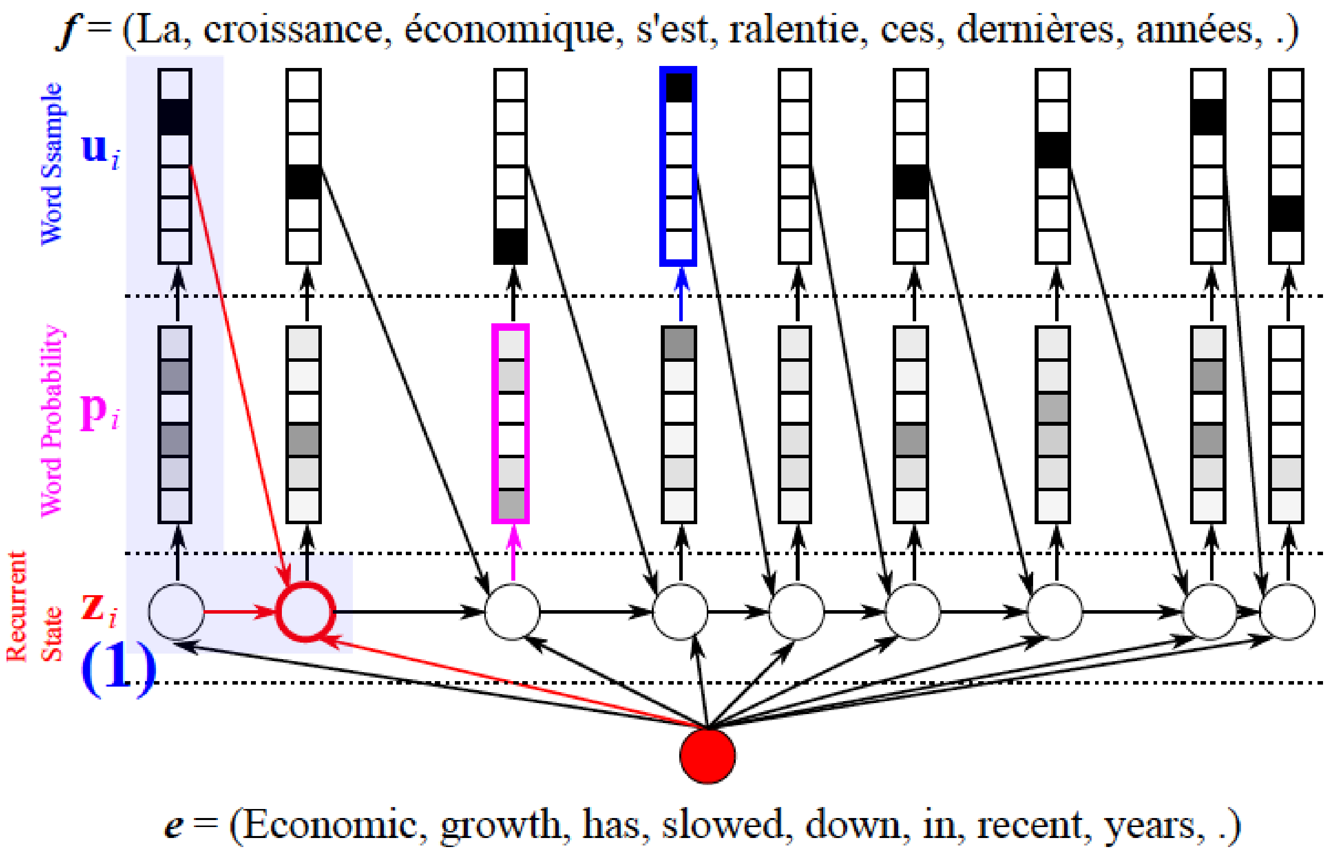
Neural Machine Translation

- Model- *encoder*



Neural Machine Translation

- Model- *decoder*



Cho: From Sequence Modeling to Translation

Decoder in more detail

Given

(i) the "summary" ($\mathbf{h}$) of the input sequence,

(ii) the previous output / word (f_{t-1})

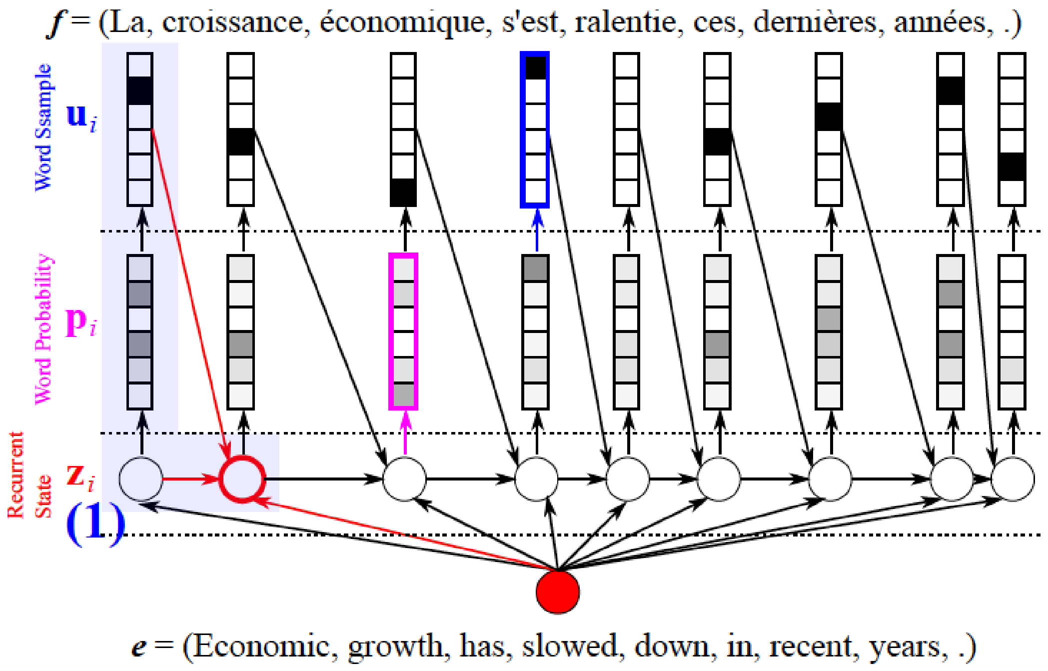
(iii) the previous state ($\mathbf{z}_{t-1}$)

the hidden state of the decoder is:

$$\mathbf{z}_t = RNN(\mathbf{z}_{t-1}, f_{t-1}, \mathbf{h})$$

Then, we can find the most likely next word:

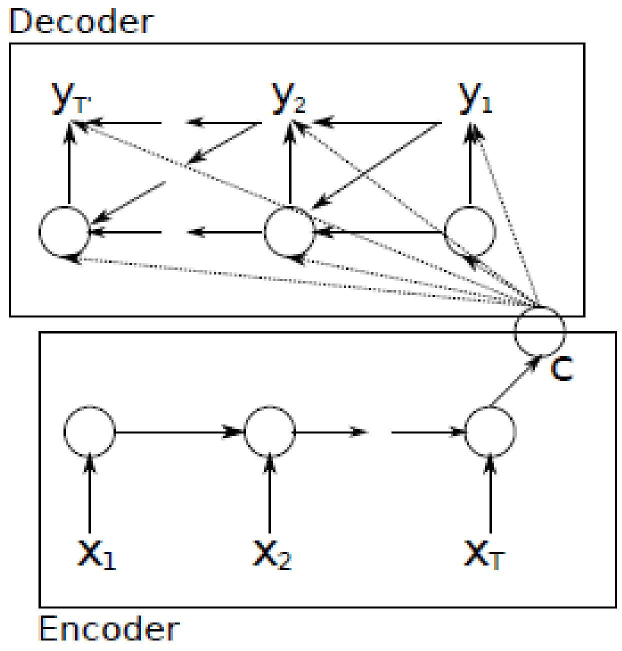
$$P(f_t \mid f_{t-1}, f_{t-2}, \ldots, \mathbf{h}) = p(f_t \mid \mathbf{z}_t, f_{t-1}, \mathbf{h})$$



Encoder-decoder

- Jointly trained to maximize

$$\max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \log p_{\theta}(\mathbf{y}_n \mid \mathbf{x}_n),$$



NMT can be done at char-level too

- http://arxiv.org/abs/1603.06147

This can be done with CNNs

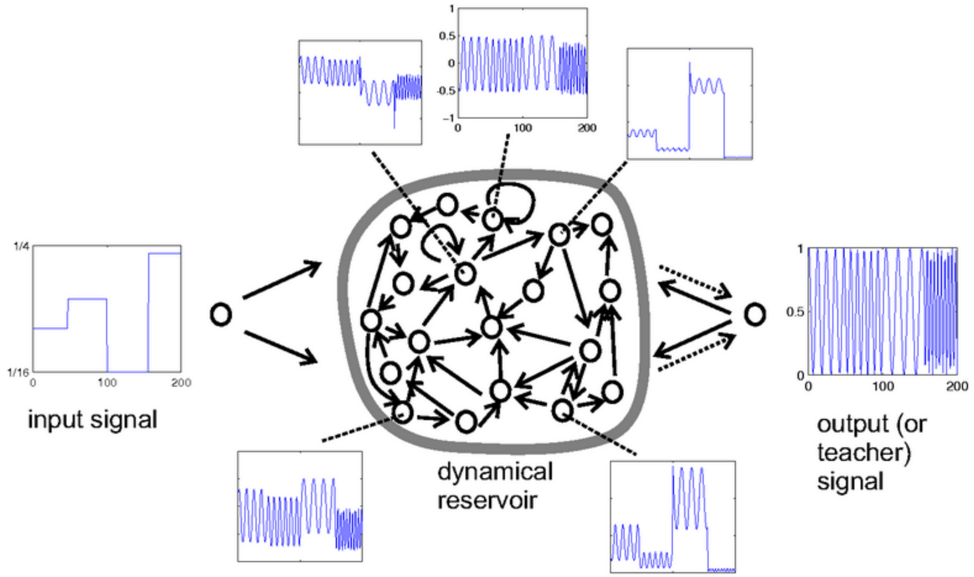
Convolutional Sequence to Sequence Learning

Jonas Gehring
Michael Auli
David Grangier
Denis Yarats
Yann N. Dauphin
Facebook AI Research

2017

Check the following tutorial

- http://smerity.com/articles/2016/google_nmt_arch.html



Echo State Networks

Reservoir Computing

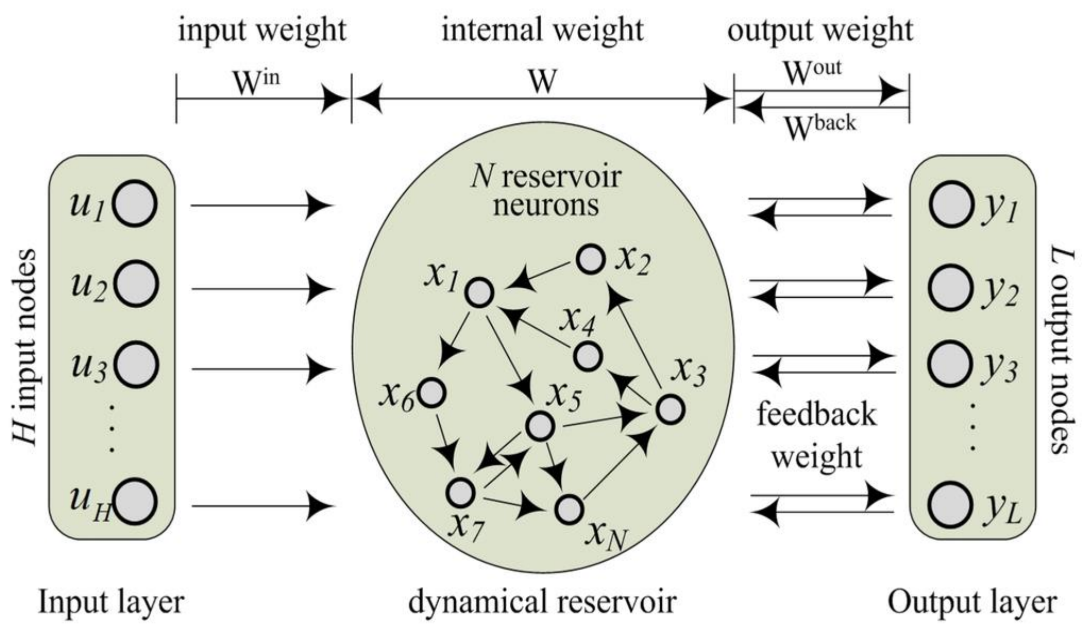
Motivation

- "Schiller and Steil (2005) also showed that in traditional training methods for RNNs, where all weights (not only the output weights) are adapted, the dominant changes are in the output weights. In cognitive neuroscience, a related mechanism has been investigated by Peter F. Dominey in the context of modelling sequence processing in mammalian brains, especially speech recognition in humans (e.g., Dominey 1995, Dominey, Hoen and Inui 2006). Dominey was the first to explicitly state the principle of reading out target information from a randomly connected RNN. The basic idea also informed a model of temporal input discrimination in biological neural networks (Buonomano and Merzenich 1995)."

http://www.scholarpedia.org/article/Echo_state_network

Echo State Networks (ESN)

- Reservoir of a set of neurons
  - Randomly initialized and fixed
  - Run input sequence through the network and keep the activations of the reservoir neurons
  - Calculate the “readout” weights using linear regression.
- Has the benefits of recurrent connections/networks
- No problem of vanishing gradient



Li et al., 2015.

The reservoir

- Provides non-linear expansion
  - This provides a “kernel” trick.
- Acts as a memory
- Parameters:
  - W_{in}, W and α (leaking rate).
- Global parameters:
  - Number of neurons: The more the better.
  - Sparsity: Connect a neuron to a fixed but small number of neurons.
  - Distribution of the non-zero elements: Uniform or Gaussian distribution. W_{in} is denser than W.
  - Spectral radius of W: Maximum absolute eigenvalue of W, or the width of the distribution of its non-zero elements.
    - Should be less than 1. Otherwise, chaotic, periodic or multiple fixed-point behavior may be observed.
    - For problems with large memory requirements, it should be bigger than 1.
  - Scale of the input weights.

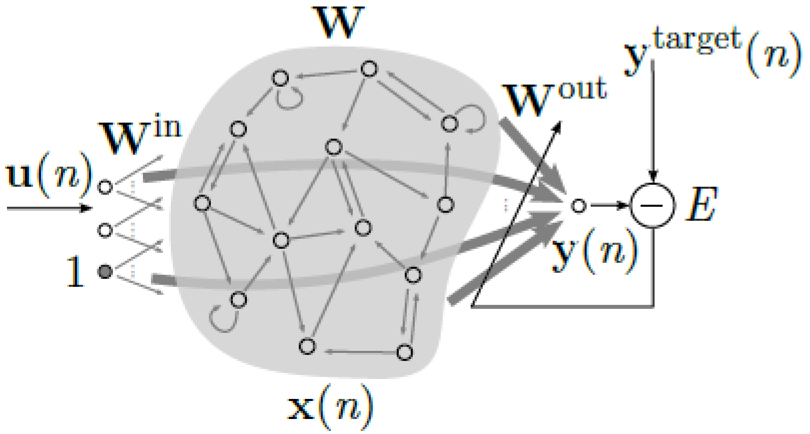


Fig. 1: An echo state network.

A Practical Guide to Applying Echo State Networks

Mantas Lukoševičius

A Practical Guide to Applying Echo State Networks

Mantas Lukoševičius

$$\tilde{\mathbf{x}}(n) = \tanh\left(\mathbf{W}^{\text{in}}[1; \mathbf{u}(n)] + \mathbf{W}\mathbf{x}(n-1)\right), \tag{2}$$

$$\mathbf{x}(n) = (1-\alpha)\mathbf{x}(n-1) + \alpha\tilde{\mathbf{x}}(n), \tag{3}$$

where $\mathbf{x}(n) \in \mathbb{R}^{N_x}$ is a vector of reservoir neuron activations and $\tilde{\mathbf{x}}(n) \in \mathbb{R}^{N_x}$ is its update, all at time step n, $\tanh(\cdot)$ is applied element-wise, $[\cdot;\cdot]$ stands for a vertical vector (or matrix) concatenation, $\mathbf{W}^{\text{in}} \in \mathbb{R}^{N_x \times (1+N_u)}$ and $\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$ are the input and recurrent weight matrices respectively, and $\alpha \in (0, 1]$ is the leaking rate. Other sigmoid wrappers can be used besides the tanh, which however is the most common choice. The model is also sometimes used without the leaky integration, which is a special case of $\alpha = 1$ and thus $\tilde{\mathbf{x}}(n) \equiv \mathbf{x}(n)$.

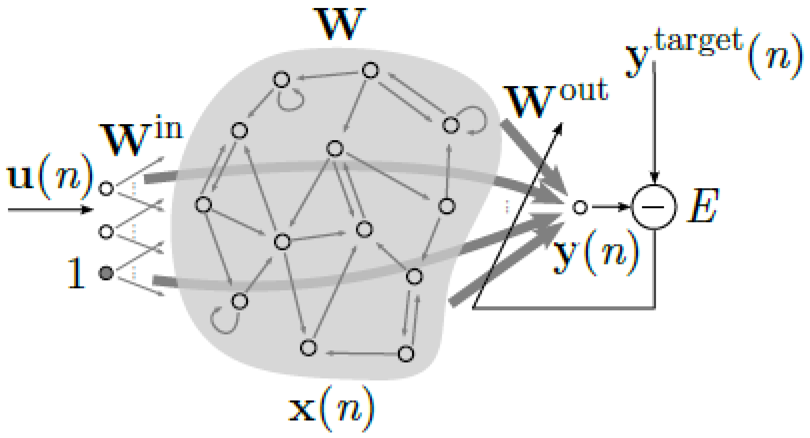


Fig. 1: An echo state network.

$$\mathbf{y}(n) = \mathbf{W}^{\text{out}}[1; \mathbf{u}(n); \mathbf{x}(n)],$$

again stands for a vertical vector (or matrix) concatenation. An additional nonlinearity can be applied to $\mathbf{y}(n)$ in (4), as well as feedback connections $\mathbf{W}^{\text{fb}}$ from $\mathbf{y}(n-1)$ to $\tilde{\mathbf{x}}(n)$ in (2). A graphical

Training ESN

$$\mathbf{Y}^{\text{target}} = \mathbf{W}^{\text{out}}\mathbf{X}$$

Probably the most universal and stable solution to (8) in this context is ridge regression, also known as regression with Tikhonov regularization:

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}}\mathbf{X}^{\mathrm{T}}\left(\mathbf{X}\mathbf{X}^{\mathrm{T}} + \beta\mathbf{I}\right)^{-1}, \tag{9}$$

where β is a regularization coefficient explained in Section 4.2, and $\mathbf{I}$ is the identity matrix.

Regularization:

$$\mathbf{W}^{\text{out}} = \underset{\mathbf{W}^{\text{out}}}{\arg\min}\, \frac{1}{N_{\mathrm{y}}} \sum_{i=1}^{N_{\mathrm{y}}} \left(\sum_{n=1}^{T} \left(y_i(n) - y_i^{\text{target}}(n)\right)^2 + \beta \left\| \mathbf{w}_i^{\text{out}} \right\|^2 \right),$$

Beyond echo state networks

- Good aspects of ESNs
  Echo state networks can be trained very fast because they just fit a linear model.
- They demonstrate that it's very important to initialize weights sensibly.
- They can do impressive modeling of one-dimensional time-series.
  - but they cannot compete seriously for high-dimensional data.

- Bad aspects of ESNs

  They need many more hidden units for a given task than an RNN that learns the hidden→hidden weights.

Slide: Hinton

Similar models

- Liquid State Machines (Maas et al., 2002)
  - A spiking version of Echo-state networks
- Extreme Learning Machines
  - Feed-forward network with a hidden layer.
  - Input-to-hidden weights are randomly initialized and never updated

Attention

BLEU: Bilingual Evaluation Understudy

https://cloud.google.com/translate/automl/docs/evaluate#bleu

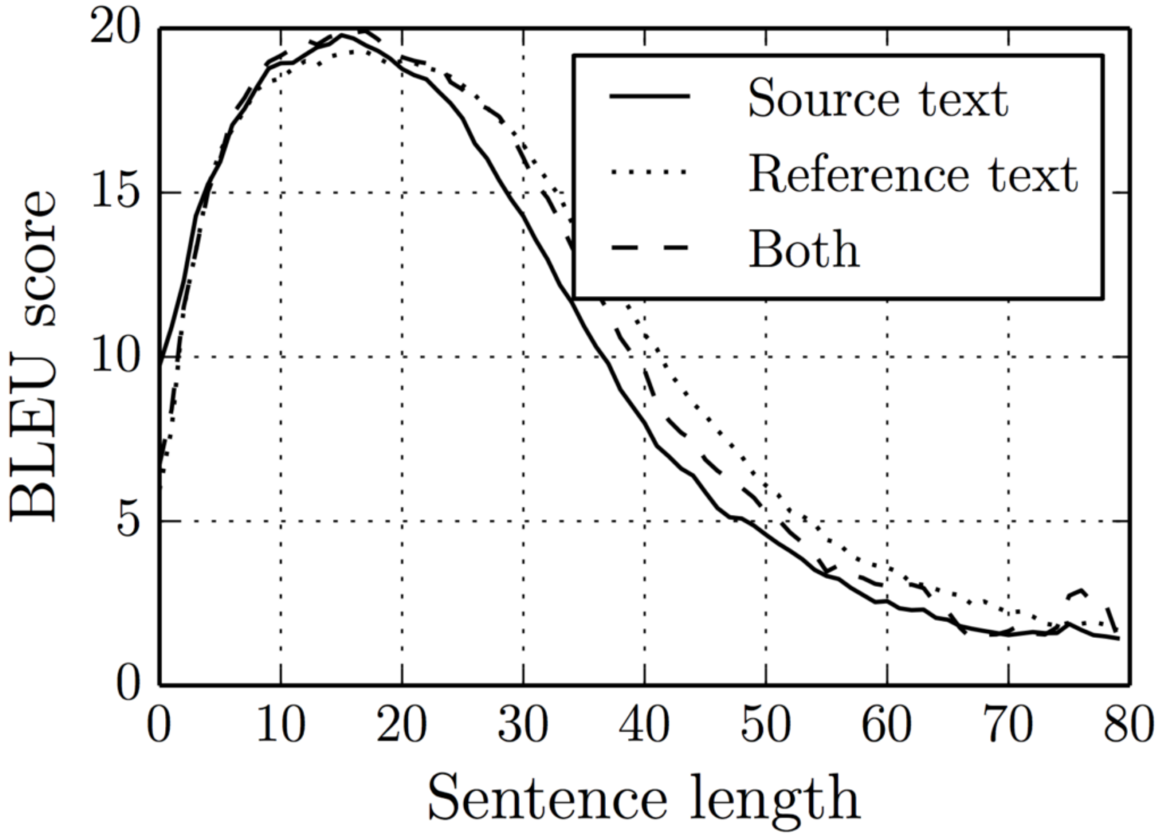
Attention

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal



Attention

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

Attention

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

In a new model architecture, we define each conditional probability in Eq. (2) as:

$$p(y_i|y_1,\ldots,y_{i-1},\mathbf{x}) = g(y_{i-1}, s_i, c_i), \tag{4}$$

where s_i is an RNN hidden state for time i, computed by

$$s_i = f(s_{i-1}, y_{i-1}, c_i).$$

It should be noted that unlike the existing encoder–decoder approach (see Eq. (2)), here the probability is conditioned on a distinct context vector c_i for each target word y_i.

The context vector c_i depends on a sequence of *annotations* $(h_1, \cdots, h_{T_x})$ to which an encoder maps the input sentence. Each annotation h_i contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence. We explain in detail how the annotations are computed in the next section.

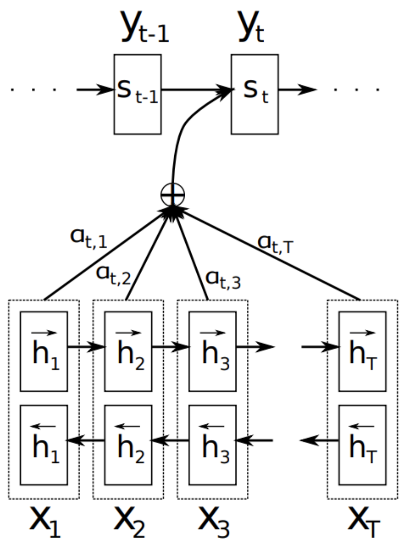


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

The context vector c_i is, then, computed as a weighted sum of these annotations h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \tag{5}$$

The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}, \tag{6}$$

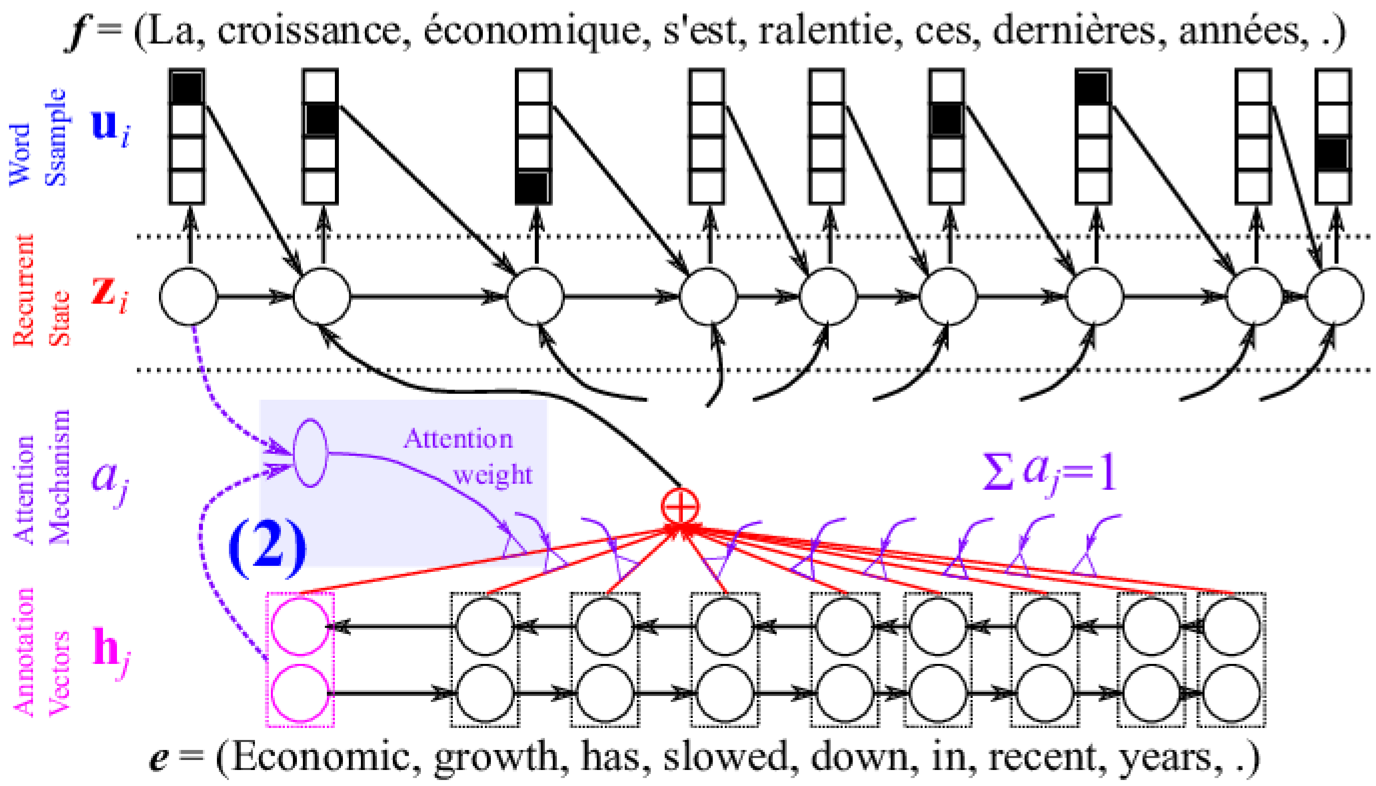
where

$$e_{ij} = a(s_{i-1}, h_j)$$

is an *alignment model* which scores how well the inputs around position j and the output at position i match. The score is based on the RNN hidden state s_{i-1} (just before emitting y_i, Eq. (4)) and the j-th annotation h_j of the input sentence.

We parametrize the alignment model a as a feedforward neural network which is jointly trained with all the other components of the proposed system. Note that unlike in traditional machine translation,

Attention



Attention mechanism: A two-layer neural network.

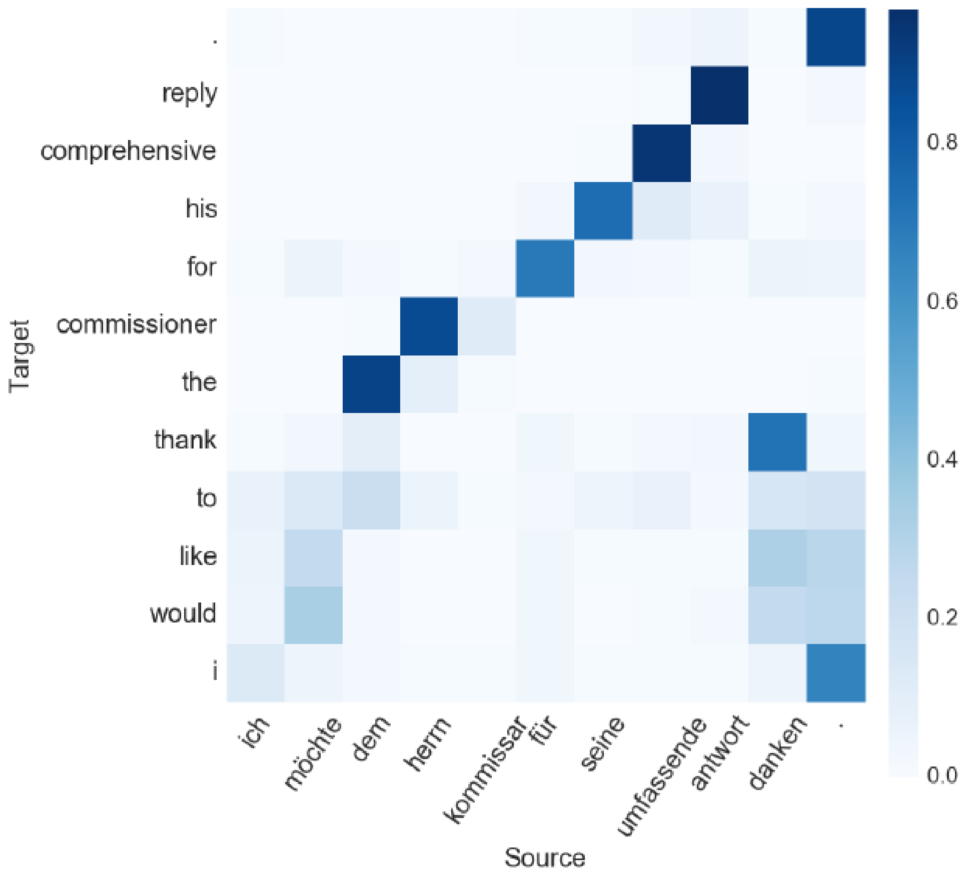
Input: z_i and h_j

Output: e_j, a scalar for the importance of word j.

The scores of words are normalized: $a_j = \mathrm{softmax}(\mathrm{e_j})$

https://devblogs.nvidia.com/introduction-neural-machine-translation-gpus-part-3/

Attention



What does Attention in Neural Machine Translation Pay Attention to?

Hamidreza Ghader and **Christof Monz**
Informatics Institute, University of Amsterdam, The Netherlands
`h.ghader, c.monz@uva.nl`

2017

Attention Types

- Let's rewrite Bahdanau et al.'s attention model:

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{t,i} \boldsymbol{h}_i \qquad \text{; Context vector for output } y_t$$

$$\alpha_{t,i} = \text{align}(y_t, x_i) \qquad \text{; How well two words } y_t \text{ and } x_i \text{ are aligned.}$$

$$= \frac{\exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i))}{\sum_{i'=1}^{n} \exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_{i'}))} \qquad \text{; Softmax of some predefined alignment score..}$$

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$$

where both $\mathbf{v}_a$ and $\mathbf{W}_a$ are weight matrices to be learned in the alignment model.

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Attention Types

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

(*) Referred to as "concat" in Luong, et al., 2015 and as "additive attention" in Vaswani, et al., 2017.
(^) It adds a scaling factor $1/\sqrt{n}$, motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Attention Types

Name	Definition	Citation
Self-Attention(&)	Relating different positions of the same input sequence. Theoretically the self-attention can adopt any score functions above, but just replace the target sequence with the same input sequence.	Cheng2016
Global/Soft	Attending to the entire input state space.	Xu2015
Local/Hard	Attending to the part of input state space; i.e. a patch of the input image.	Xu2015; Luong2015

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Self-attention

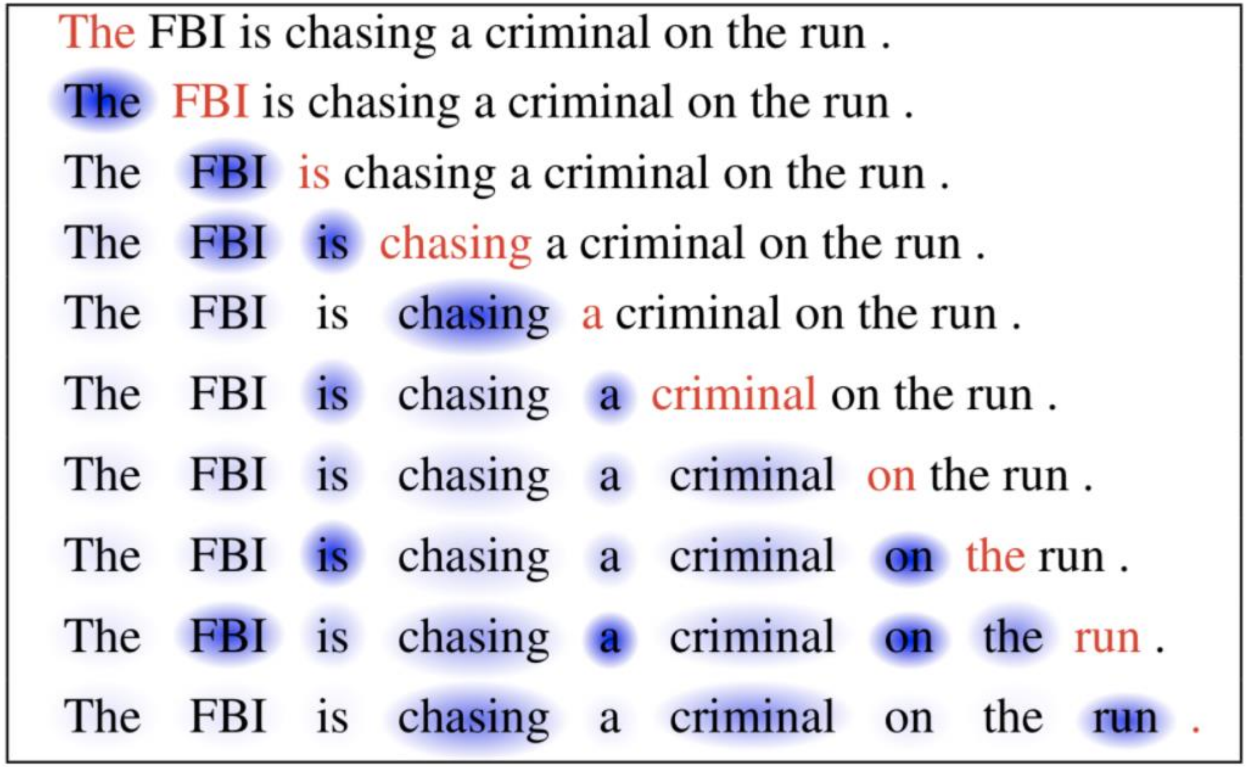


Fig. 6. The current word is in red and the size of the blue shade indicates the activation level. (Image source: Cheng et al., 2016)

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Soft/hard attention

Fig. 7. "A woman is throwing a frisbee in a park." (Image source: Fig. 6(b) in Xu et al. 2015)

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Global/local attention

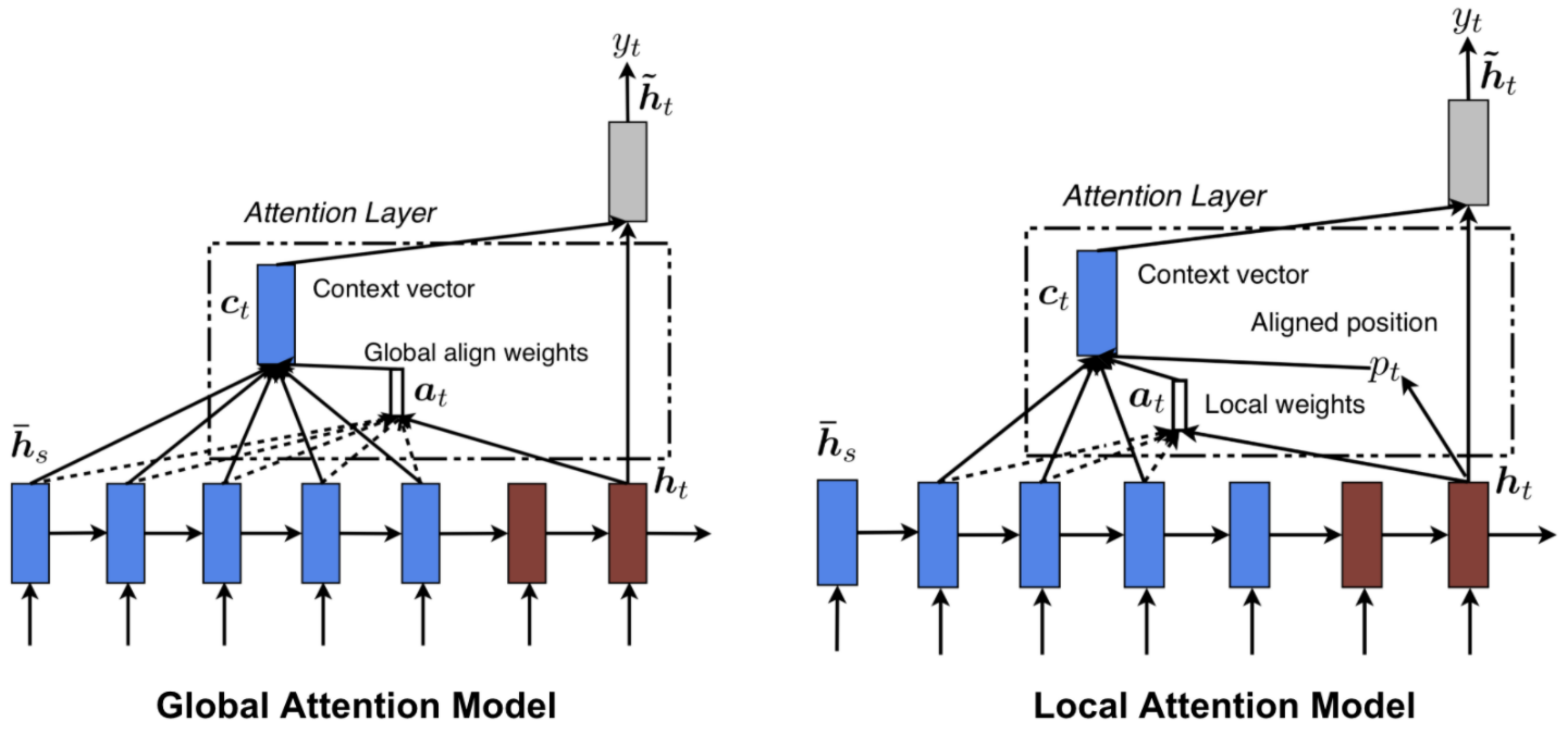


Fig. 8. Global vs local attention (Image source: Fig 2 & 3 in Luong, et al., 2015)

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html