# Neural Architectures for NLP

Jindřich Helcl, Jindřich Libovický

📅 February 26, 2020

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

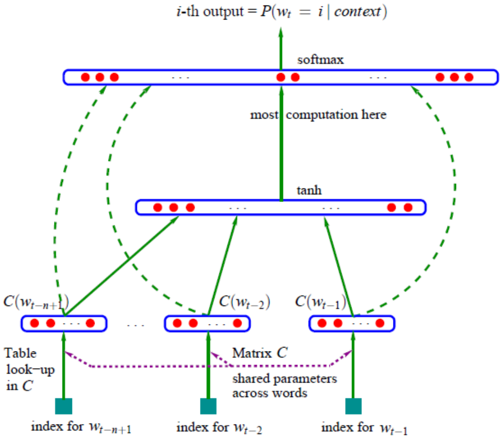
BY NC SA
unless otherwise stated

# Outline

# Symbol Embeddings

# Discrete symbol vs. continuous representation

Simple task: predict next word given three previous:



Source: Bengio, Yoshua, et al. "A neural probabilistic language model." Journal of machine learning research 3.Feb (2003): 1137-1155.
`http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf`

# Embeddings

- Natural solution: one-hot vector (vector of vocabulary length with exactly one 1)
- It would mean a huge matrix every time a symbol is on the input
- Rather factorize this matrix and share the first part ⇒ embeddings
- “Embeddings” because they embed discrete symbols into a continuous space

# Embeddings

- Natural solution: one-hot vector (vector of vocabulary length with exactly one 1)
- It would mean a huge matrix every time a symbol is on the input
- Rather factorize this matrix and share the first part ⇒ embeddings
- "Embeddings" because they embed discrete symbols into a continuous space

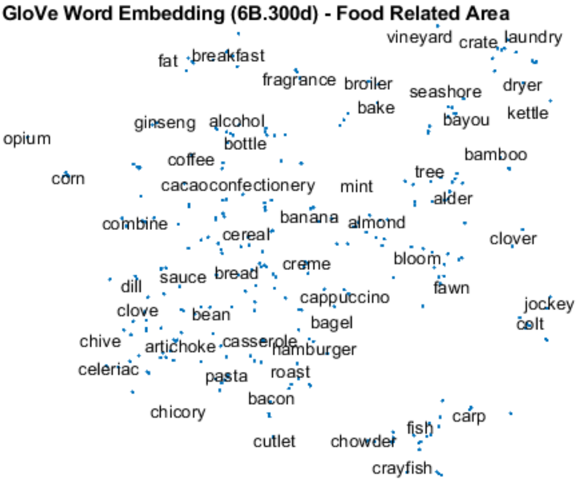
*What is the biggest problem during training?*

# Embeddings

- Natural solution: one-hot vector (vector of vocabulary length with exactly one 1)
- It would mean a huge matrix every time a symbol is on the input
- Rather factorize this matrix and share the first part ⇒ embeddings
- "Embeddings" because they embed discrete symbols into a continuous space

*What is the biggest problem during training?*
Embeddings get updated only rarely – only when a symbol appears.

# Properties of embeddings



GloVe Word Embedding (6B.300d) - Food Related Area

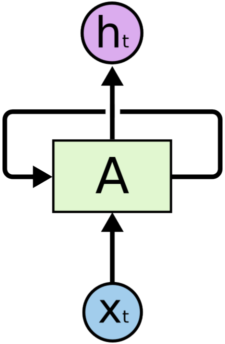
athworks.com/loren/2017/09/21/math-with-words-word-embeddings-with-matlab-and-text-analytics-toolbox/

# Recurrent Networks

# Why RNNs

- `for` loops over sequential data
- the most frequently used type of network in NLP

# General Formulation

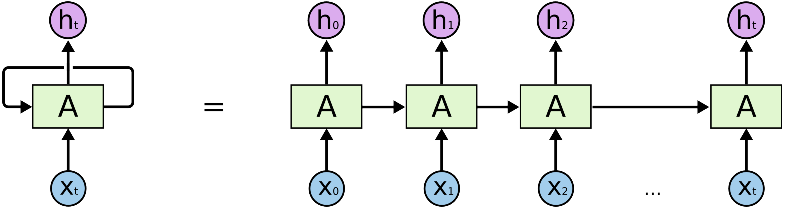


- inputs: $x_{,} \ldots , x_T$
- initial state $h_0 = \mathbf{0}$, a result of previous computation, trainable parameter
- recurrent computation: $h_t = A(h_{t-1}, x_t)$

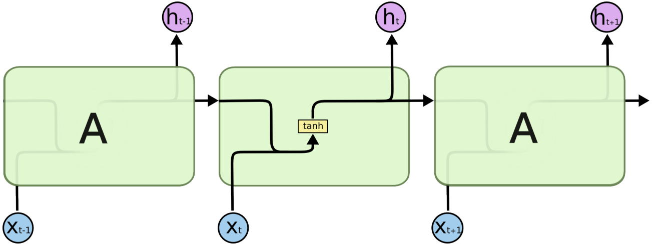
# RNN as Imperative Code

```
def rnn(initial_state, inputs):
  prev_state = initial_state
  for x in inputs:
    new_state, output = rnn_cell(x, prev_state)
    prev_state = new_state
    yield output
```

# RNN as a Fancy Image



# Vanilla RNN



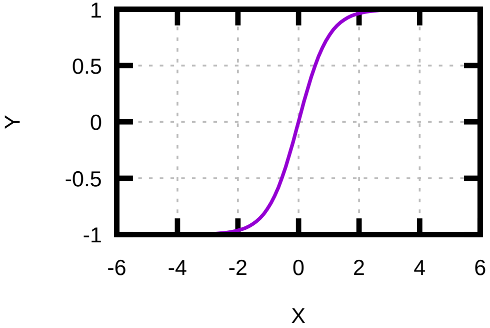
$$h_t = \tanh\left(W[h_{t-1}; x_t] + b\right)$$

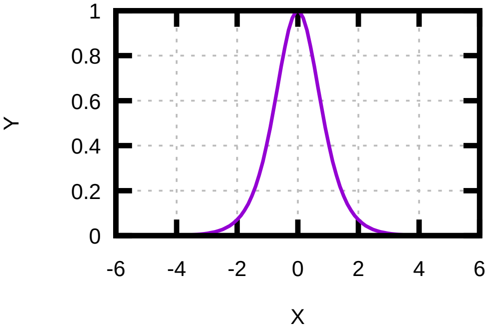
- cannot propagate long-distance relations
- vanishing gradient problem

# Vanishing Gradient Problem (1)

$$\tanh x = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

$$\frac{\tanh x}{x} = 1 - \tanh^2 x \in (0, 1]$$

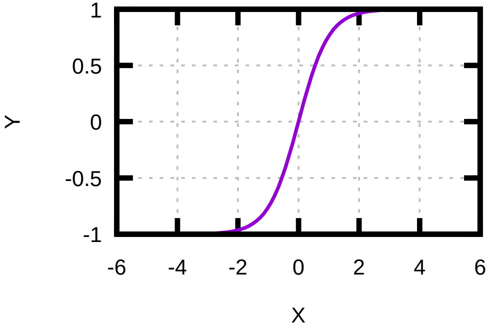


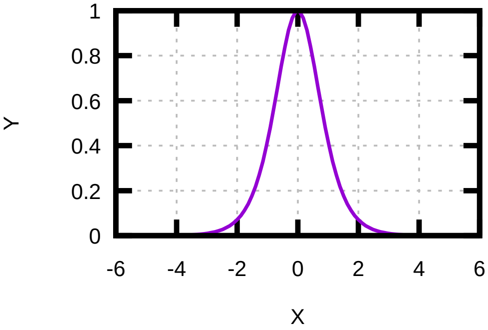


# Vanishing Gradient Problem (1)

$$\tanh x = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

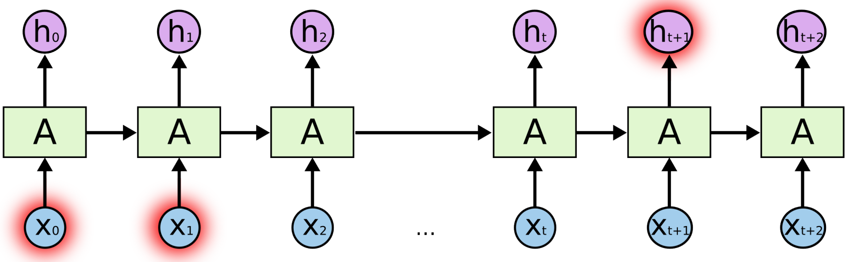
$$\frac{\tanh x}{x} = 1 - \tanh^2 x \in (0, 1]$$



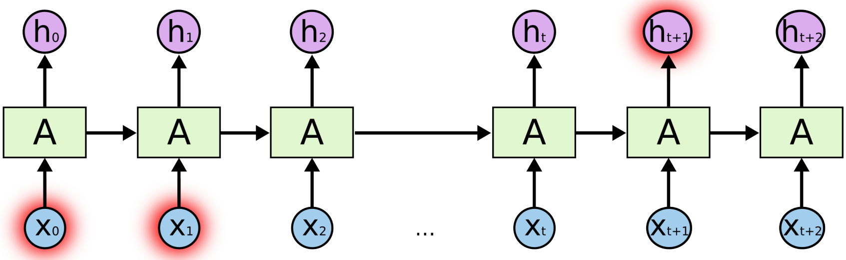


Weight initialized $\sim \mathcal{N}(0, 1)$ to have gradients further from zero.

# Vanishing Gradient Problem (2)

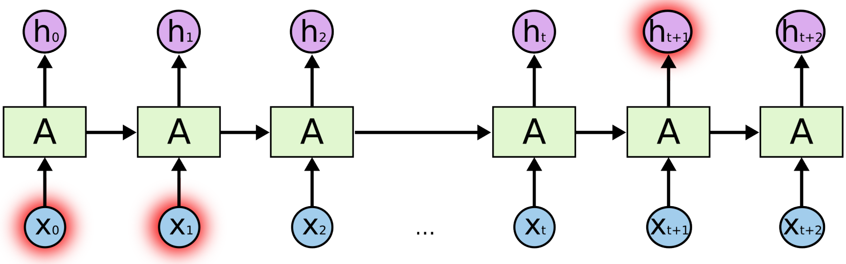


# Vanishing Gradient Problem (2)



$$\frac{\partial E_{t+1}}{\partial b} =$$

# Vanishing Gradient Problem (2)



$$\frac{\partial E_{t+1}}{\partial b} = \frac{\partial E_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial b}$$ (chain rule)

# Vanishing Gradient Problem (3)

$$\frac{\partial h_t}{\partial b} =$$

# Vanishing Gradient Problem (3)

$$\frac{\partial h_t}{\partial b} = \frac{\partial \tanh \overbrace{(W_h h_{t-1} + W_x x_t + b)}^{=z_t \text{ (activation)}}}{\partial b}$$

($\tanh'$ is derivative of $\tanh$)

# Vanishing Gradient Problem (3)

$$\begin{aligned}\frac{\partial h_t}{\partial b} &= \frac{\partial \tanh \overbrace{(W_h h_{t-1} + W_x x_t + b)}^{=z_t \text{ (activation)}}}{\partial b} \qquad \text{(}\tanh' \text{ is derivative of } \tanh\text{)} \\ &= \tanh'(z_t) \cdot \left( \frac{\partial W_h h_{t-1}}{\partial b} + \underbrace{\frac{\partial W_x x_t}{\partial b}}_{=0} + \underbrace{\frac{\partial b}{\partial b}}_{=1} \right)\end{aligned}$$

# Vanishing Gradient Problem (3)

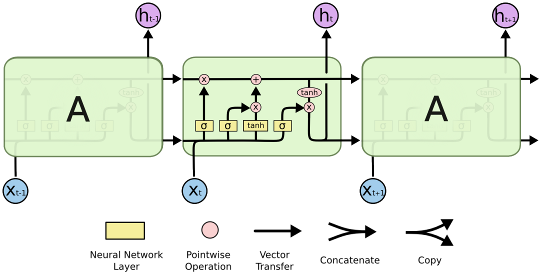
$$\begin{aligned}
\frac{\partial h_t}{\partial b} &= \frac{\partial \tanh \overbrace{(W_h h_{t-1} + W_x x_t + b)}^{= z_t \text{ (activation)}}}{\partial b} \qquad (\tanh' \text{ is derivative of } \tanh) \\
&= \tanh'(z_t) \cdot \left( \frac{\partial W_h h_{t-1}}{\partial b} + \underbrace{\frac{\partial W_x x_t}{\partial b}}_{=0} + \underbrace{\frac{\partial b}{\partial b}}_{=1} \right) \\
&= \underbrace{W}_{\sim \mathcal{N}(0,1)} \underbrace{\tanh'(z_t)}_{\in (0;1]} \frac{\partial h_{t-1}}{\partial b} + \tanh'(z_t)
\end{aligned}$$

# LSTMs

LSTM = Long short-term memory

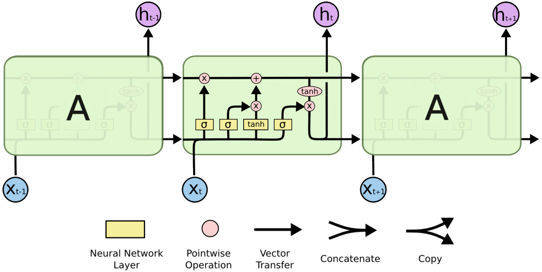
# LSTMs

LSTM = Long short-term memory



# LSTMs

LSTM = Long short-term memory

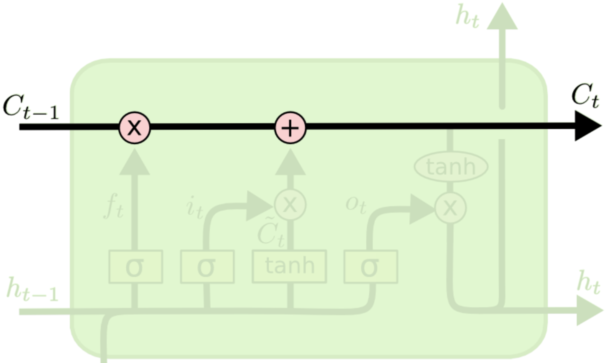


Control the gradient flow by explicitly gating:

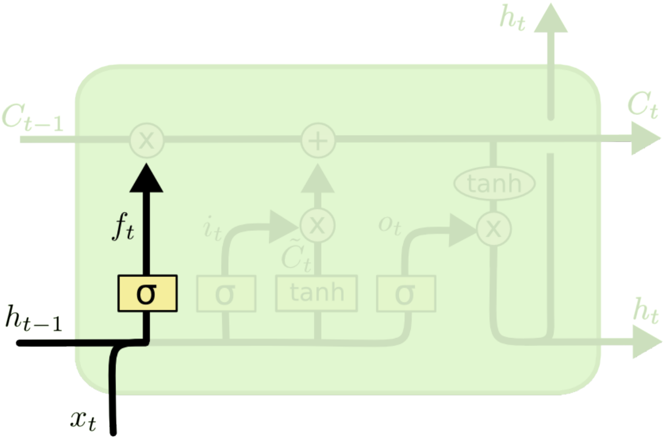
- what to use from input,
- what to use from hidden state,
- what to put on output

# Hidden State

- two types of hidden states
- $h_t$ — “public” hidden state, used an output
- $c_t$ — “private” memory, no non-linearities on the way
  - direct flow of gradients (without multiplying by $\leq$ derivatives)
  - only vectors guaranteed to live in the same space are manipulated
- information highway metaphor



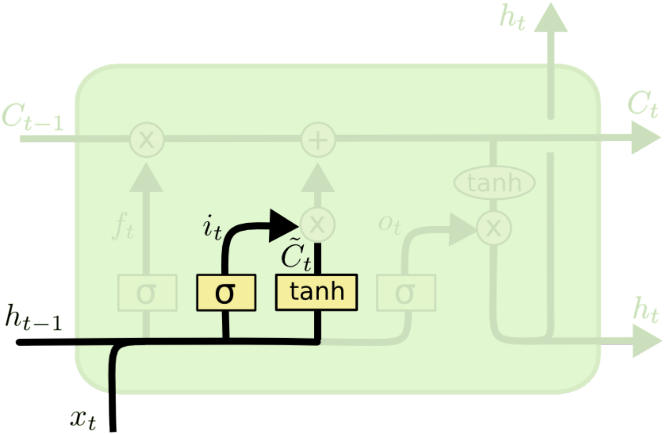
# Forget Gate



$$f_t = \sigma\left(W_f[h_{t-1}; x_t] + b_f\right)$$

- based on input and previous state, decide what to forget from the memory

# Input Gate

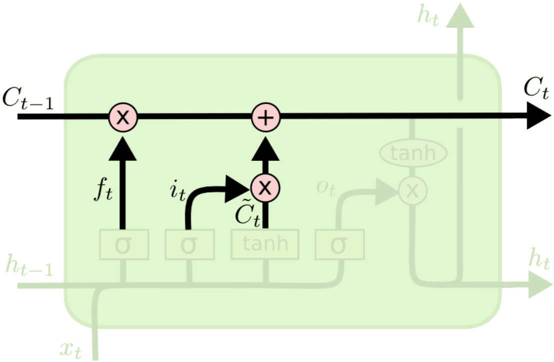


$$i_t = \sigma\left(W_i \cdot [h_{t-1}; x_t] + b_i\right)$$

$$\tilde{C}_t = \tanh\left(W_c \cdot [h_{t-1}; x_t] + b_C\right)$$

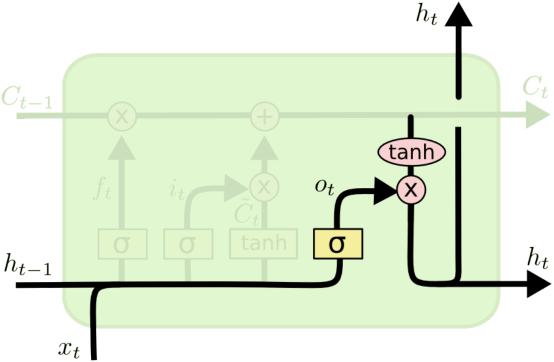
- $\tilde{C}$ — candidate what may want to add to the memory
- $i_t$ — decide how much of the information we want to store

# Cell State Update



$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

# Output Gate



$$o_t = \sigma\left(W_o \cdot [h_{t-1}; x_t] + b_o\right)$$

$$h_t = o_t \odot \tanh C_t$$

# Here we are!

$$\begin{aligned}
f_t &= \sigma\left(W_f[h_{t-1};x_t] + b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1};x_t] + b_i\right) \\
o_t &= \sigma\left(W_o \cdot [h_{t-1};x_t] + b_o\right) \\
\tilde{C}_t &= \tanh\left(W_c \cdot [h_{t-1};x_t] + b_C\right) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= o_t \odot \tanh C_t
\end{aligned}$$

# Here we are!

$$
\begin{aligned}
f_t &= \sigma\left(W_f[h_{t-1}; x_t] + b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}; x_t] + b_i\right) \\
o_t &= \sigma\left(W_o \cdot [h_{t-1}; x_t] + b_o\right) \\
\tilde{C}_t &= \tanh\left(W_c \cdot [h_{t-1}; x_t] + b_C\right) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= o_t \odot \tanh C_t
\end{aligned}
$$

*How would you implement it efficiently?*

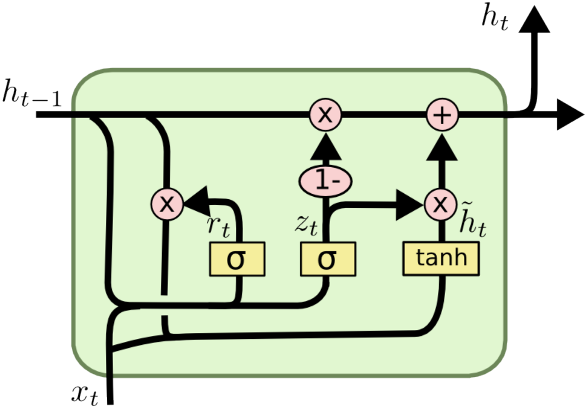
# Here we are!

$$\begin{aligned}
f_t &= \sigma\left(W_f[h_{t-1}; x_t] + b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}; x_t] + b_i\right) \\
o_t &= \sigma\left(W_o \cdot [h_{t-1}; x_t] + b_o\right) \\
\tilde{C}_t &= \tanh\left(W_c \cdot [h_{t-1}; x_t] + b_C\right) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= o_t \odot \tanh C_t
\end{aligned}$$

*How would you implement it efficiently?*

Compute all gates in a single matrix multiplication.

# Gated Recurrent Units



$$z_t = \sigma\left(W_z[h_{t-1}; x_t] + b_z\right)$$

$$r_t = \sigma\left(W_r[h_{t-1}; x_t] + b_r\right)$$

$$\tilde{h}_t = \tanh\left(W[r_t \odot h_{t-1}; x_t]\right)$$

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

# GRU and LSTM

*Are GRUs special case of LSTMs?*

**LSTM**

$$\begin{aligned}
f_t &= \sigma\left(W_f[h_{t-1};x_t]+b_f\right)\\
i_t &= \sigma\left(W_i\cdot[h_{t-1};x_t]+b_i\right)\\
o_t &= \sigma\left(W_o\cdot[h_{t-1};x_t]+b_o\right)\\
\tilde{C}_t &= \tanh\left(W_c\cdot[h_{t-1};x_t]+b_C\right)\\
C_t &= f_t\odot C_{t-1}+i_t\odot\tilde{C}_t\\
h_t &= o_t\odot\tanh C_t
\end{aligned}$$

**GRU**

$$\begin{aligned}
z_t &= \sigma\left(W_z[h_{t-1};x_t]+b_z\right)\\
r_t &= \sigma\left(W_r[h_{t-1};x_t]+b_r\right)\\
\tilde{h}_t &= \tanh\left(W[r_t\odot h_{t-1};x_t]\right)\\
h_t &= (1-z_t)\odot h_{t-1}+z_t\odot\tilde{h}_t
\end{aligned}$$

# GRU and LSTM

*Are GRUs special case of LSTMs?*

**LSTM**

$$\begin{aligned} f_t &= \sigma\left(W_f[h_{t-1};x_t]+b_f\right) \\ i_t &= \sigma\left(W_i \cdot [h_{t-1};x_t]+b_i\right) \\ o_t &= \sigma\left(W_o \cdot [h_{t-1};x_t]+b_o\right) \\ \tilde{C}_t &= \tanh\left(W_c \cdot [h_{t-1};x_t]+b_C\right) \\ C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\ h_t &= o_t \odot \tanh C_t \end{aligned}$$

**GRU**

$$\begin{aligned} z_t &= \sigma\left(W_z[h_{t-1};x_t]+b_z\right) \\ r_t &= \sigma\left(W_r[h_{t-1};x_t]+b_r\right) \\ \tilde{h}_t &= \tanh\left(W[r_t \odot h_{t-1};x_t]\right) \\ h_t &= (1-z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \end{aligned}$$

No, you cannot lay $C_t \equiv h_t$ because of the additional non-linearity in LSTMs.

# GRU or LSTM?

- GRU preserved the information highway property
- less parameters, should learn faster
- LSTM more general (although both Turing complete)
- empirical results: it's task-specific

Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." arXiv preprint arXiv:412.3555 (204).
Irie, Kazuki, et al. "LSTM, GRU, highway and a bit of attention: an empirical overview for language modeling in speech recognition." Interspeech, San Francisco, CA, USA (206).

# Recurrent Networks‘

+

- correspond to intuition of sequential processing
- theoretically strong

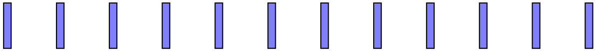
-

- cannot be parallelized, always need to wait for previous state

# Convolutional Networks

# 1-D Convolution

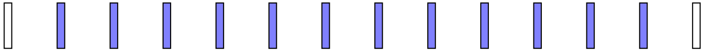
$\approx$ sliding window over the sequence



embeddings $\mathbf{x} = (x_1, \dots, x_N)$

# 1-D Convolution

$\approx$ sliding window over the sequence

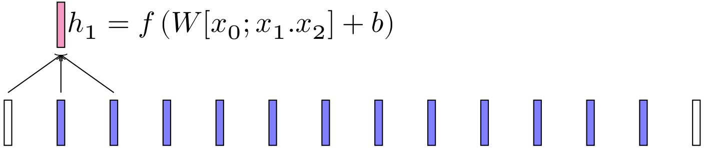


$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \ldots, x_N)$ $x_N = \vec{0}$

pad with 0s if we want to keep sequence length

# 1-D Convolution

$\approx$ sliding window over the sequence



$h_1 = f\left(W[x_0; x_1 . x_2] + b\right)$

$x_0 = \vec{0}$

embeddings $\mathbf{x} = (x_1, \ldots, x_N)$

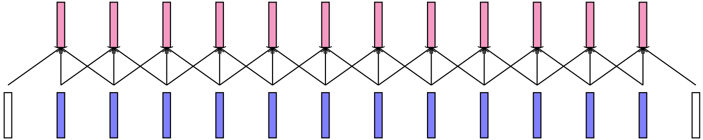
$x_N = \vec{0}$

pad with 0s if we want to keep sequence length

# 1-D Convolution

$\approx$ sliding window over the sequence

$$h_i = f\left(W\left[x_{i-1}; x_i; x_{i+1}\right] + b\right)$$



$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \dots, x_N)$ $x_N = \vec{0}$

pad with 0s if we want to keep sequence length

# 1-D Convolution: Code

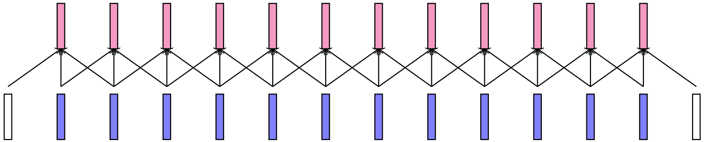
## Pseudocode

```
xs = ... # input sequnce

kernel_size = 3 # window size
filters = 300 # output dimensions
strides=1     # step size

W = trained_parameter(xs.shape[2] * kernel_size, filters)
b = trained_parameter(filters)
window = kernel_size // 2

outputs = []
for i in range(window, xs.shape[1] - window):
    h = np.mul(W, xs[i - window:i + window]) + b
    outputs.append(h)
return np.array(h)
```

## TensorFlow

# Residual Connections

$$h_i = f\left(W\left[x_{i-1}; x_i; x_{i+1}\right] + b\right)$$

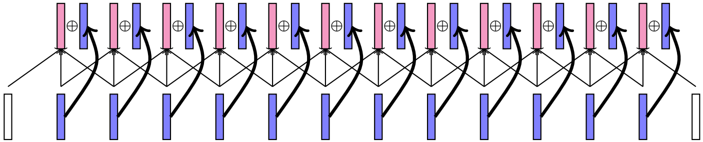


$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \dots, x_N)$ $x_N = \vec{0}$

Allows training deeper networks.

# Residual Connections

$$h_i = f\left(W\left[x_{i-1}; x_i; x_{i+1}\right] + b\right) + x_i$$

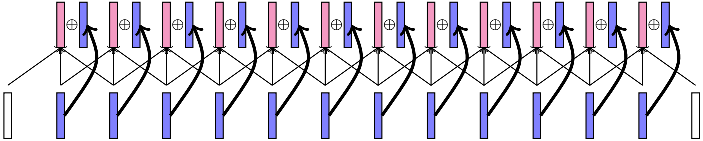


$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \ldots, x_N)$ $x_N = \vec{0}$

Allows training deeper networks.

# Residual Connections

$$h_i = f\left(W\left[x_{i-1}; x_i; x_{i+1}\right] + b\right) + x_i$$

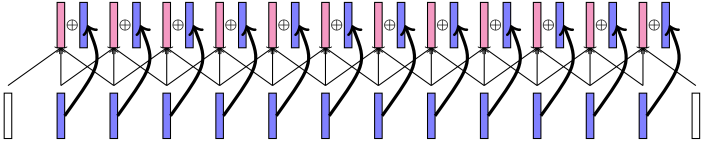


$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \ldots, x_N)$ $x_N = \vec{0}$

Allows training deeper networks.
*Why do you it helps?*

# Residual Connections

$$h_i = f\left(W\left[x_{i-1}; x_i; x_{i+1}\right] + b\right) + x_i$$



$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \ldots, x_N)$ $x_N = \vec{0}$

Allows training deeper networks.

*Why do you it helps?*

Better gradient flow – the same as in RNNs.

# Residual Connections: Numerical Stability

Numerically unstable, we need activation to be in similar scale ⇒ layer normalization.
Activation before non-linearity is normalized:

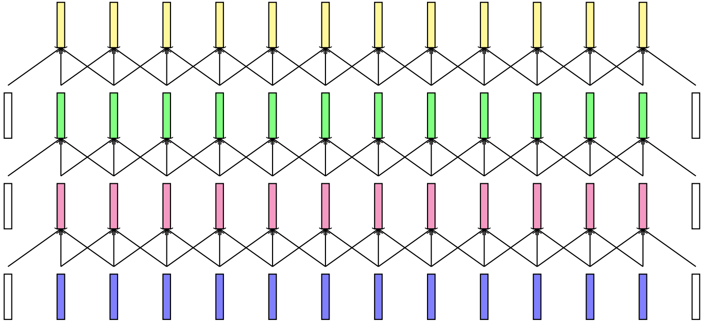
$$\overline{a}_i = \frac{g_i}{\sigma_i}\left(a_i - \mu_i\right)$$

...$g$ is a trainable parameter, $\mu$, $\sigma$ estimated from data.

$$\mu = \frac{1}{H}\sum_{i=1}^{H} a_i$$

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

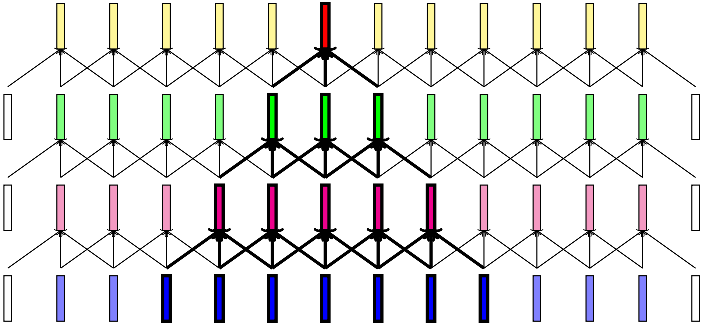
# Receptive Field



$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \dots, x_N)$ $x_N = \vec{0}$

Can be enlarged by dilated convolutions.

# Receptive Field



$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \dots, x_N)$ $x_N = \vec{0}$

Can be enlarged by dilated convolutions.

# Convolutional architectures

+

- extremely computationally efficient

-

- limited context
- by default no aware of $n$-gram order

# Self-attentive Networks

# Main idea of self-attention

- matrix multiplication can be used for to get dot-product similarity between all sequence vectors
- while using the same vector space, information might be gathered by summing up

**Both regardless the distance in the sequence!**

# Naive code

```
xs = ... # input sequence, time x dimension
dimension = xs.shape[1]
hidden_size = 400 # size of additional projection

for x_1 in xs:
    similarities = np.array(np.sum(x_1 * x_2) for x_2 in xs)
    distribution = softmax(similarities)
    context = np.sum(xs * distribution, axis=1)

    hidden_layer_input = layer_norm(context + xs)
    hidden_layer_middle = relu(
        dense_layer(hidden_input, hidden_size))
    hidden_layer_output = relu(
        dense_layer(hidden_input, hidden_size))

    yield layer_norm(
        hidden_layer_input + hidden_layer_output)
```

# Self-attentive architectures

+

- computationally efficient
- unlimited context
- empower state-of-the-art models

-

- memory requirements grow quadratically with sequence length
- not aware or positions in the sequence (requires positional embeddings)

# Reading Assignment

# Reading for the Next Week

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." arXiv preprint arXiv:1409.0473 (2014).
`https://arxiv.org/pdf/1409.0473.pdf`

Questions:
**The authors report 5 BLEU points worse score than the previous encoder-decoder architecture (Sutskever et al., 2014). Why is their model better then?**

**If someone asked you to create automatically a dictionary. Would you use the attention mechanism for it? Why yes? Why not?**