

NPFL099 Statistical Dialogue Systems

3. Neural Nets Basics

<http://ufal.cz/npfl099>

Ondřej Dušek, Vojtěch Hudeček & Tomáš Nekvinda

18. 10. 2021



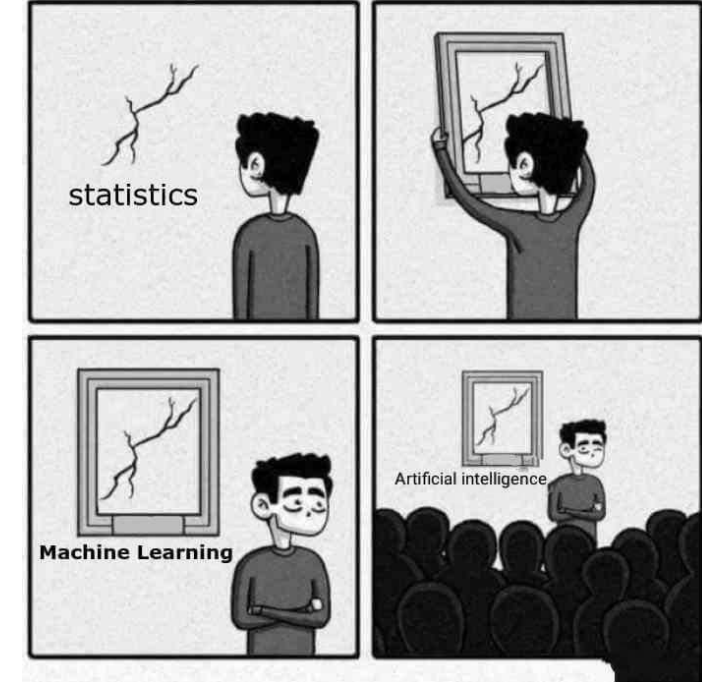
Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics



unless otherwise stated

Machine Learning

- ML is basically function approximation
- function: data (**features**) → **labels**
 - discrete labels = **classification**
 - continuous labels = **regression**
- function shape
 - this is where different algorithms differ
 - neural nets: complex functions, composed of simple building blocks (linear, sigmoid, tanh...)
- training/learning = adjusting function parameters to minimize error
 - **supervised learning** = based on data + labels given in advance
 - **reinforcement learning** = based on exploration & rewards given online

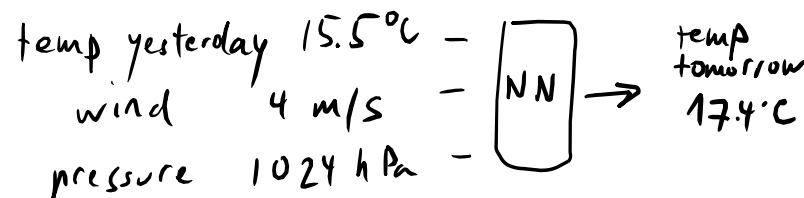


<https://towardsdatascience.com/no-machine-learning-is-not-just-glorified-statistics-26d3952234e3>

Typical machine learning problems in NLP

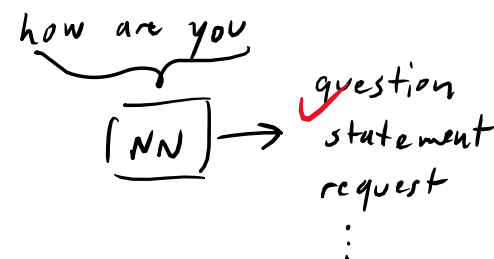
- **regression**

- many inputs, 1 float output



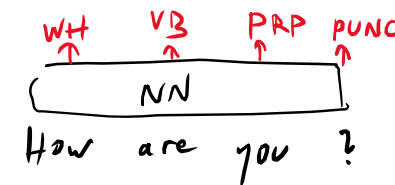
- **classification**

- many inputs, 1 categorical output (k classes)



- **sequence labelling**

- sequence of inputs, label each (~ repeated classification)
- 1-to-1 input to output

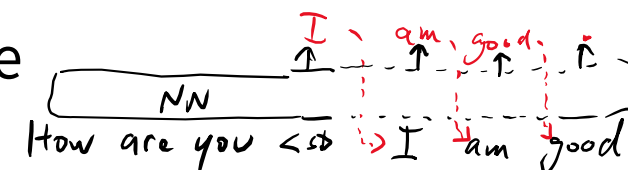
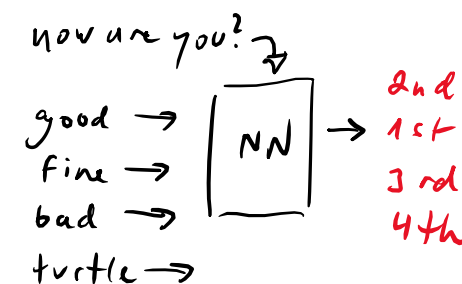


- **ranking**

- multiple inputs, choose best one (~ diff regression)

- **sequence prediction (autoregressive generation)**

- some inputs (sequence/something else)
- generate outputs, use previous output in predicting next one



Neural networks

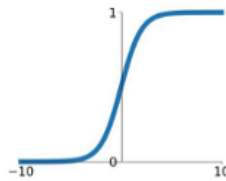
- Can be used for both classification & sequence models
- **Non-linear functions**, composed of basic building blocks
 - stacked into **layers**
- Layers are made of **activation functions**:
 - linear functions (~basic, default)
 - nonlinearities – sigmoid, tanh, ReLU
 - softmax – probability estimates:

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{|\mathbf{x}|} \exp(x_j)}$$

- Fully differentiable – training by **gradient descent**
 - network output incurs loss/cost
 - gradients **backpropagated** from loss to all parameters (composite function differentiation)

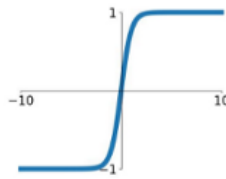
Sigmoid

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



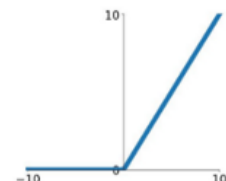
tanh

$$\tanh(x)$$



ReLU

$$\max(0, x)$$



https://medium.com/@shrutija_don10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

Layers visualization

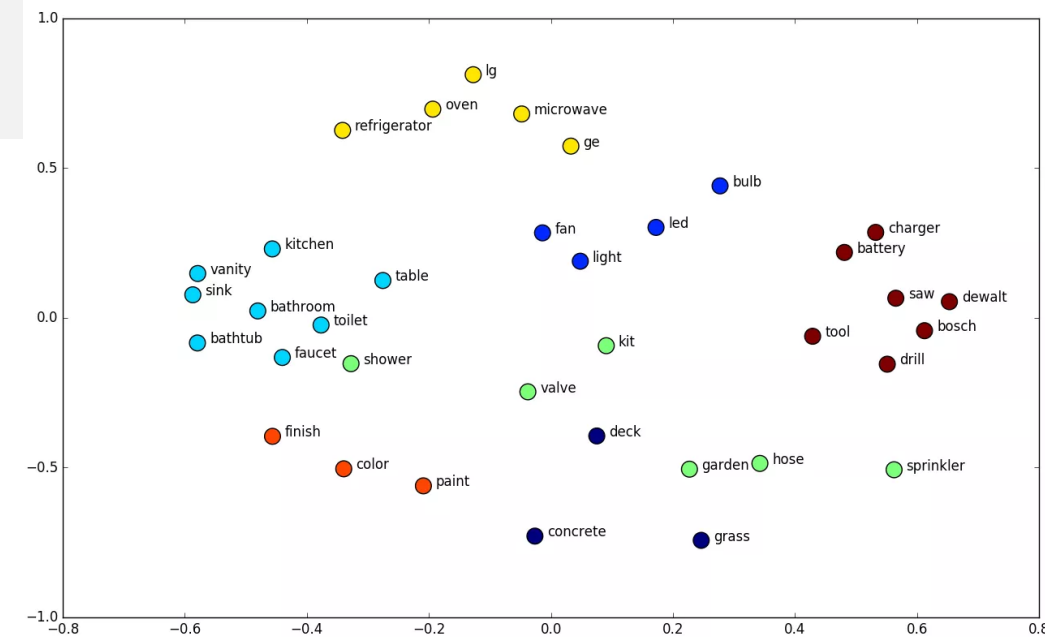
- <https://playground.tensorflow.org/>
 - 2 numeric features (=2 input variables) → binary classification (=1 output, 2 classes)
 - easiest case, but you can see the internals
 - more complex input features (→)
 - **feed-forward = fully connected = multi-layer perceptron** here
 - easiest case: connect everything & let the network figure it out
 - nice but gets too large very quickly, not good for variable-sized inputs
 - added layers & power to distinguish different classes
 - fits the training data Y/N ?
 - different activation functions
 - without them, it's just linear – no matter how many layers!
- best NN conceptualization – pipeline / flow (computational graph)
 - data flows through individual layers, gets changed
 - corresponds to a math formula, but can be easier to read

Feature representation

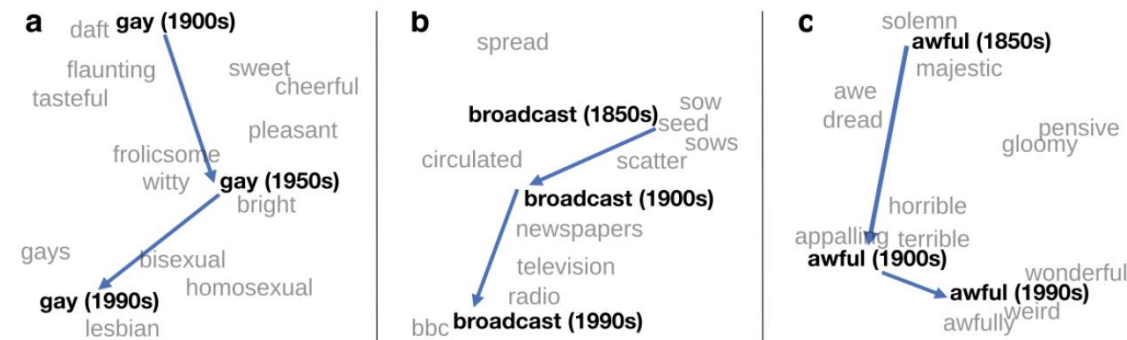
- technically can be anything, as long as it's meaningful
 - the network will learn to assign meaning/values itself
- **1-hot/binary**
 - words – numbered vocabulary
 - bigrams, n-grams, positional...
 - other features – especially handcrafted
 - word classes
 - various word combinations
 - outputs of other classifiers (sentiment, part-of-speech...)
 - is capitalized/is loud?
- **numeric (floats)**
 - best for continuous inputs: vision, audio
 - raw pixels, MFCCs...
- **vectors (embeddings) →**

Embeddings

- distributed (word) representation
 - each word = a vector of floats**
 - basically an easy conversion of 1-hot \rightarrow numeric
 - a dictionary of trainable features
- part of network parameters – trained
 - random initialization
 - pretraining
- the network learns which words are used similarly – for the given task!
 - they end up having close embedding values
 - different embeddings for different tasks
- embedding size: ~100s-1000
- vocab size: ~50-100k



<http://blog.kaggle.com/2016/05/18/home-depot-product-search-relevance-winners-interview-1st-place-alex-andreas-nurlan/>



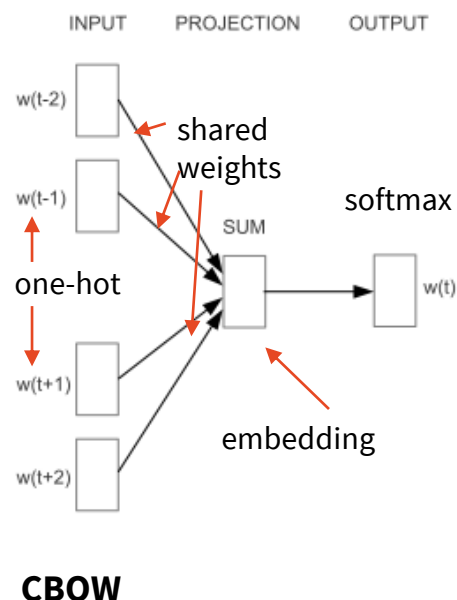
<http://ruder.io/word-embeddings-2017/>

Pretrained Word Embeddings

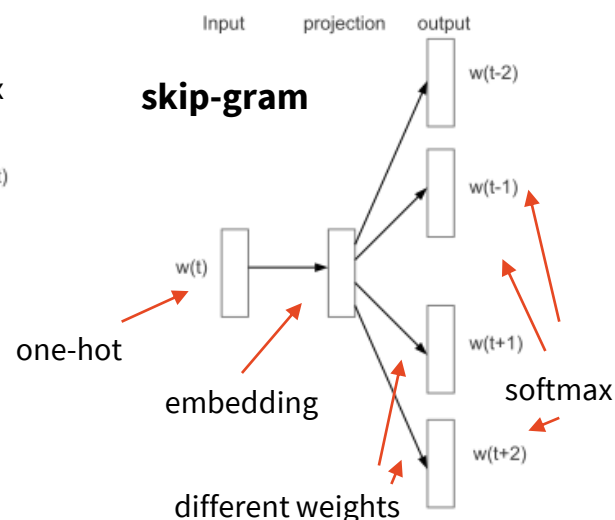
- **Word2Vec**

<https://projector.tensorflow.org/>

- Continuous Bag-of-Words (CBOW) (~ “masked LM”)
 - predict a word, given $\pm k$ words window
 - disregarding word order within the window
- Skip-gram: reverse
 - given a word, predict its $\pm k$ word window
 - closer words = higher weight in training



(Mikolov et al., 2013)
<http://arxiv.org/abs/1301.3781>



- **GloVe**

- optimized directly from corpus co-occurrences (= w_1 close to w_2)
- target: $e_1 \cdot e_2 = \log(\# \text{co-occurrences})$
 - number weighted by distance, weighted down for low totals
- trained by minimizing reconstruction loss on a co-occurrence matrix

(Pennington et al., 2014)
<http://aclweb.org/anthology/D14-1162>

Word Embeddings

- Vocabulary is unlimited, embedding matrix isn't
 - + the bigger the embedding matrix, the slower your models
- Special **out-of-vocabulary token** *<unk>*
 - “default” / older option
 - all words not found in vocabulary are assigned this entry
 - can be trained using some rare words in the data
 - problem for generation – you don't want these on the output
- Using limited sets
 - **characters** – very small set
 - works, but makes for very long sequences (20 words ~ 80-100 characters)
 - slower, might be less accurate
 - **subwords** – compromise →

Subwords

- group of characters that:
 - make shorter sequences than using individual characters
 - cover everything

- **byte-pair encoding**

- start from individual characters
- iteratively merge most frequent bigram, until you get desired # of subwords
- *sub@@ word* – the @@ marks “no space after”


- **SentencePiece** – don’t pre-tokenize

- criterium: likelihood of joined vs. separate
- *sub word_* – the _ marks a space

- 20-50k subwords for 1 language
 - ~250k subwords to cover them all

(Sennrich et al., 2016)

<https://www.aclweb.org/anthology/P16-1162/>

*fast_
faster_
tall_
taller_*  *fast er_
tall er_
slow er_
tall est_*

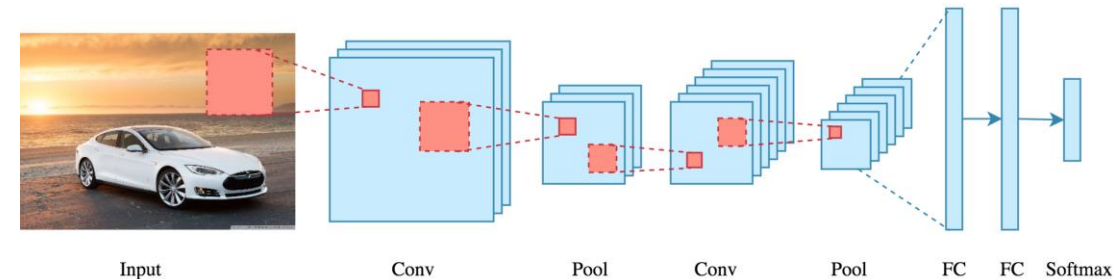
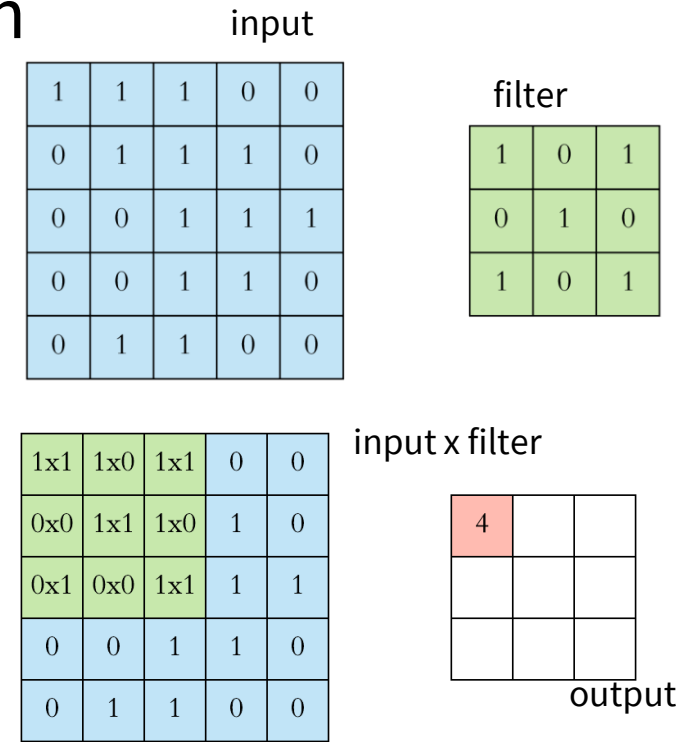
<https://github.com/google/sentencepiece>

<https://blog.floydhub.com/tokenization-nlp/>

https://d2l.ai/chapter_natural-language-processing-pretraining/subword-embedding.html

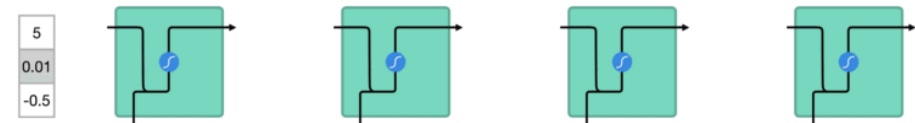
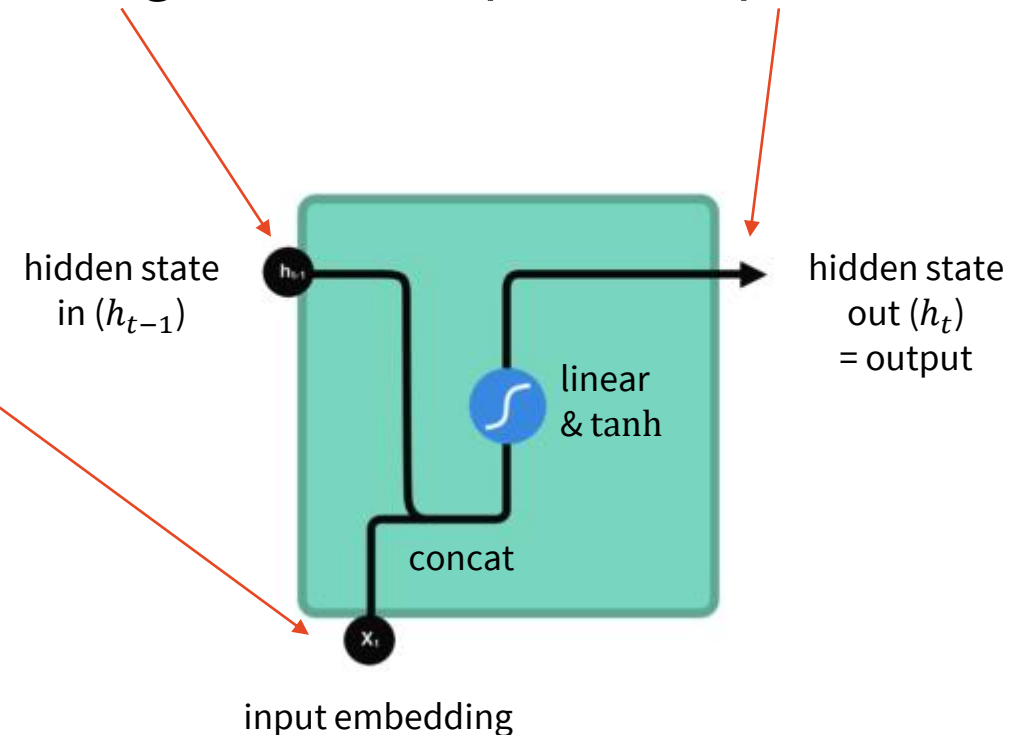
Convolutional Networks

- Designed for computer vision – inspired by human vision
 - works for language in 1D, too!
- less parameters than fully connected – **filter/kernel**
- Apply filter repeatedly over the input
 - element-wise multiply window of input x filter
 - sum + apply non-linearity (ReLU) to result
 - => produce 1 element of output
 - can have more dimensions (~“set of filters”)
- **Stride** – how many steps to skip
 - less overlap, reducing output dimension
- **Pooling** – no filter, pre-set operation
 - **maximum**/average on each window
 - typical CNN architecture alternates convolution & pooling



Recurrent Neural Networks

- Identical layers with shared parameters (**cells**)
 - ~ the same layer is applied multiple times, taking its own outputs as input
 - ~ same number of layers as there are tokens
 - output = **hidden state** – fed to the next step
 - additional input – next token features
- **basic RNN: linear + tanh**
 - tanh: squashes everything to $[-1,1]$
 - good for repeated application
 - very simple structure
 - numeric problem: vanishing gradients
 - training updates get too small
 - can't hold long sequences well

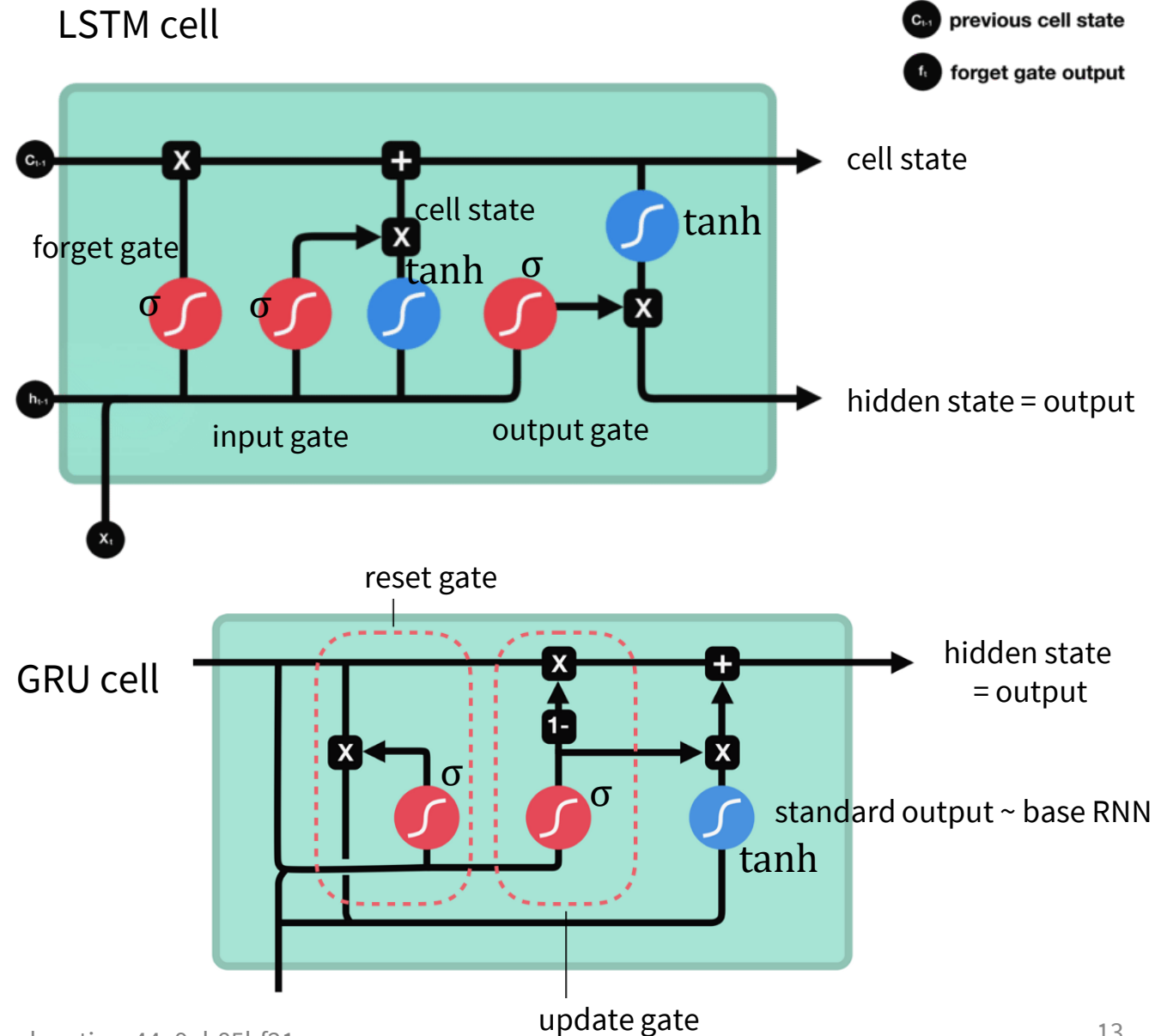


<https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>

<https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57>

LSTMs & GRUs

- **GRU, LSTM:** more complex, to make training more stable
 - “gates” to keep old values
 - $\sigma \sim [0,1]$ decisions:
 - forget stuff from previous?
 - take input into account?
 - put stuff onto output?
 - over individual dimensions (e.g. input has 100 dims, forget gate forgets dims 1-3 & 4-25)
 - all based on current input & state
 - LSTM is older & more complex
 - GRU almost as good but faster
 - both slower than base RNN
 - both handle long recurrences

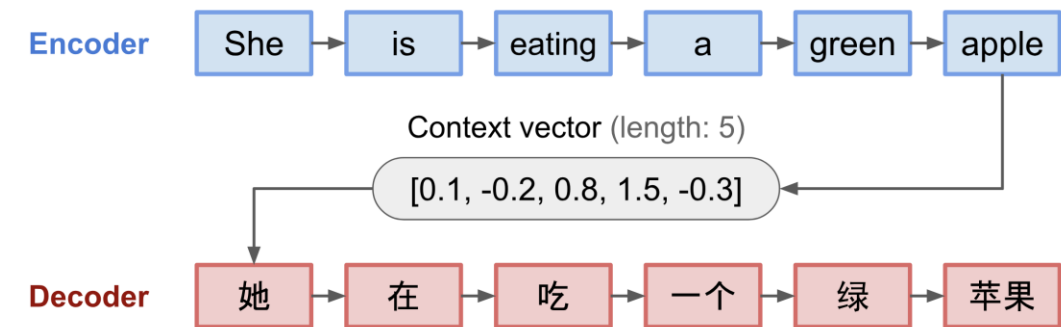
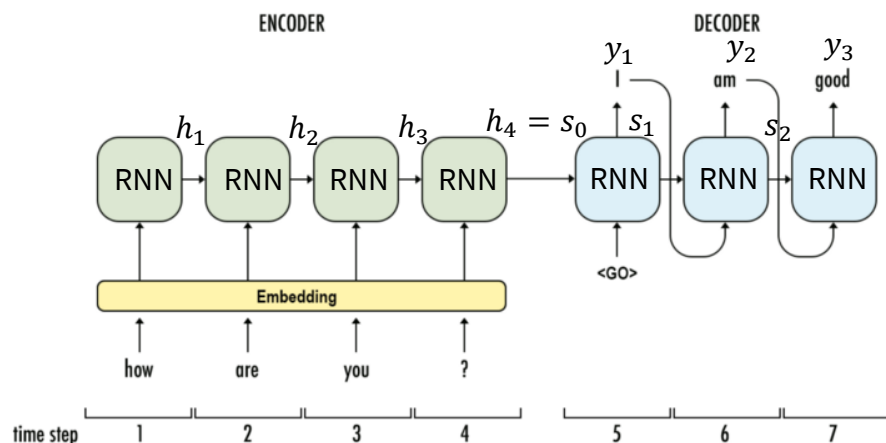


Encoder-Decoder Networks (Sequence-to-sequence)

- Default RNN paradigm for sequences/structure prediction
 - **encoder** RNN: encodes the input token-by-token into **hidden states** h_t
 - next step: last hidden state + next token as input
 - **decoder RNN**: constructs the output token-by-token
 - initialized by last encoder hidden state
 - output: hidden state & softmax over output vocabulary + argmax
 - next step: last hidden state + last generated token as input
 - LSTM/GRU cells over vectors of ~ embedding size
 - used in MT, dialogue, parsing...
 - more complex structures linearized to sequences

$$h_0 = 0$$
$$h_t = \text{cell}(x_t, h_{t-1})$$

$$s_0 = h_T$$
$$p(y_t | y_1, \dots, y_{t-1}, \mathbf{x}) = \text{softmax}(s_t)$$
$$s_t = \text{cell}(y_{t-1}, s_{t-1})$$

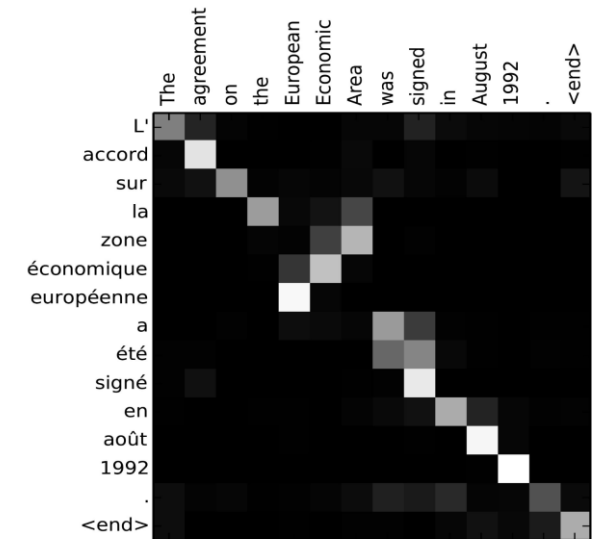
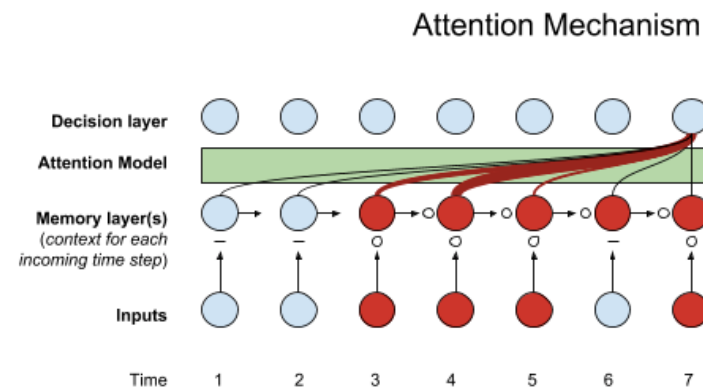


<https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html>

<https://medium.com/syncedreview/a-brief-overview-of-attention-mechanism-13c578ba9129>

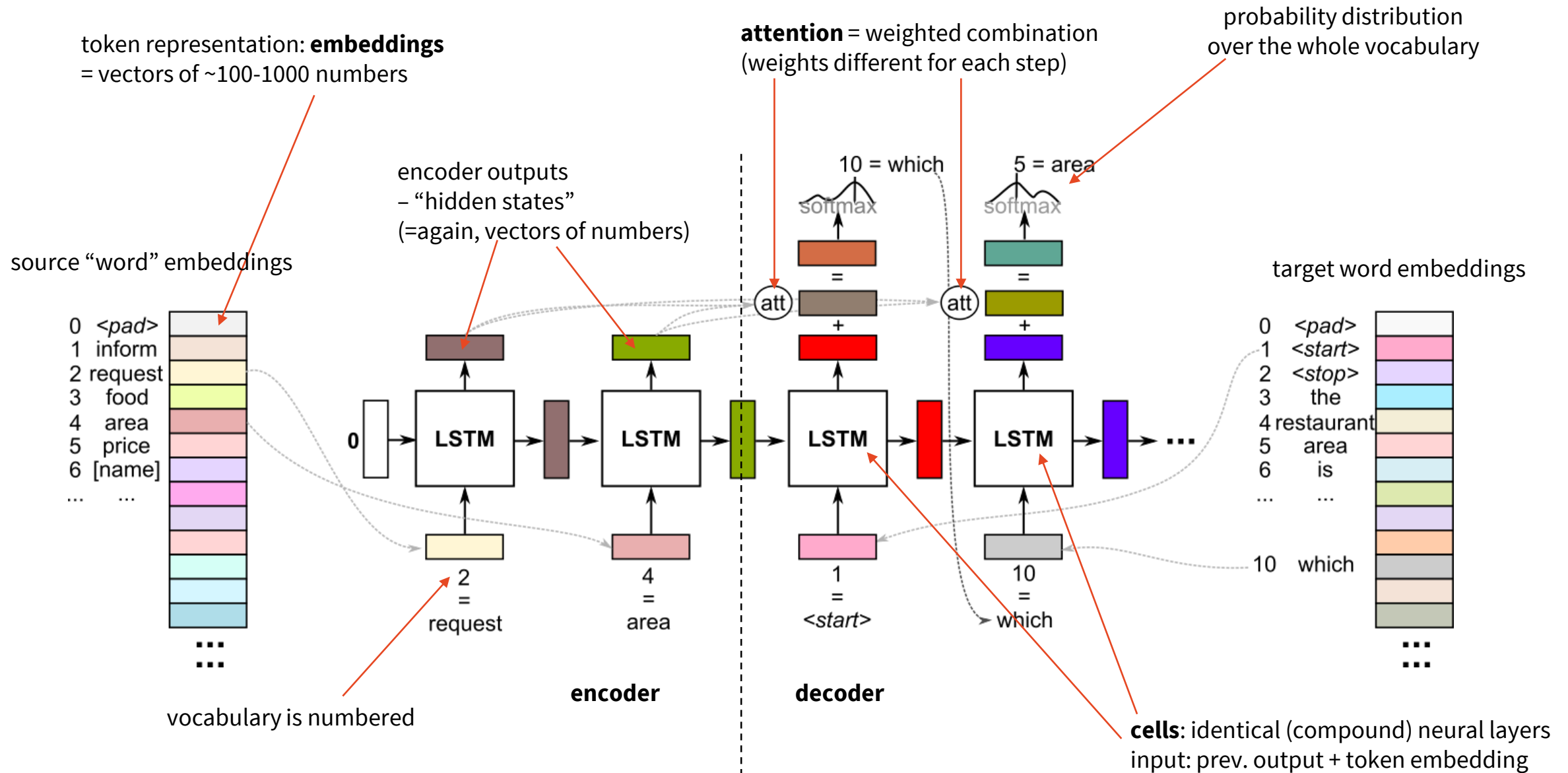
Attention

- Encoder-decoder is too crude for complex sequences
 - the whole input is crammed into a fixed-size vector (last hidden state)
- **Attention** = “memory” of **all encoder** hidden states
 - weighted combination, re-weighted for every decoder step
→ can focus on currently important part of input
 - fed into decoder inputs + decoder softmax layer
- **Self-attention** – over **previous decoder steps**
 - increases consistency when generating long sequences



<https://skymind.ai/wiki/attention-mechanism-memory-network>

Seq2seq RNNs with Attention



Bahdanau & Luong Attention

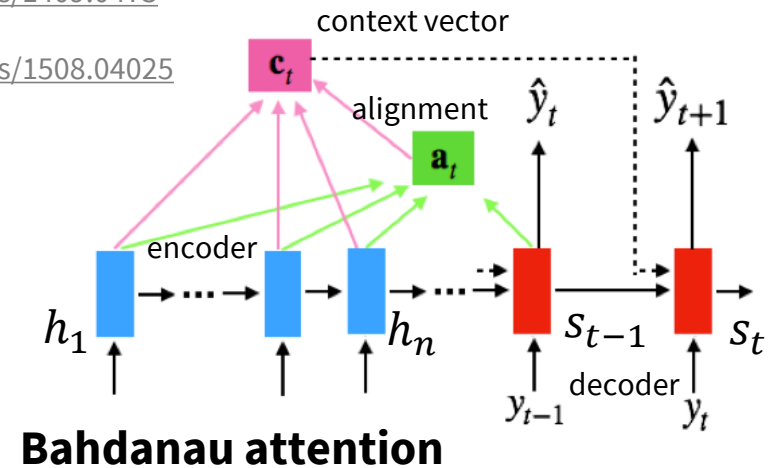
- different combination with decoder state
 - Bahdanau: use on input to decoder cell
 - Luong: modify final decoder state
- different weights computation
- both work well – exact formula not important

(Bahdanau et al., 2015)

<http://arxiv.org/abs/1409.0473>

(Luong et al., 2015)

<http://arxiv.org/abs/1508.04025>



attention weights = alignment model

Bahdanau:

$$\alpha_{ti} = \text{softmax}(\mathbf{v}_\alpha \cdot \tanh(\mathbf{W}_\alpha \cdot \mathbf{s}_{t-1} + \mathbf{U}_\alpha \cdot \mathbf{h}_i))$$

decoder state
trained parameters
encoder hidden state

Luong:

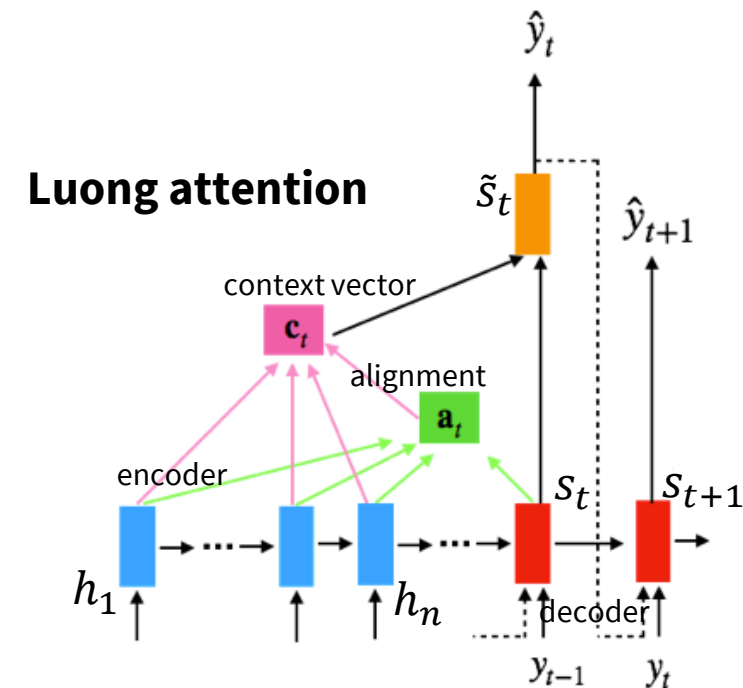
$$\alpha_{ti} = \text{softmax}(\mathbf{h}_i^\top \cdot \mathbf{s}_t)$$

decoder state
encoder hidden state

attention value = context vector

same for both – sum encoder hidden states weighted by α_{ti}

$$\mathbf{c}_t = \sum_{i=1}^n \alpha_{ti} \mathbf{h}_i$$

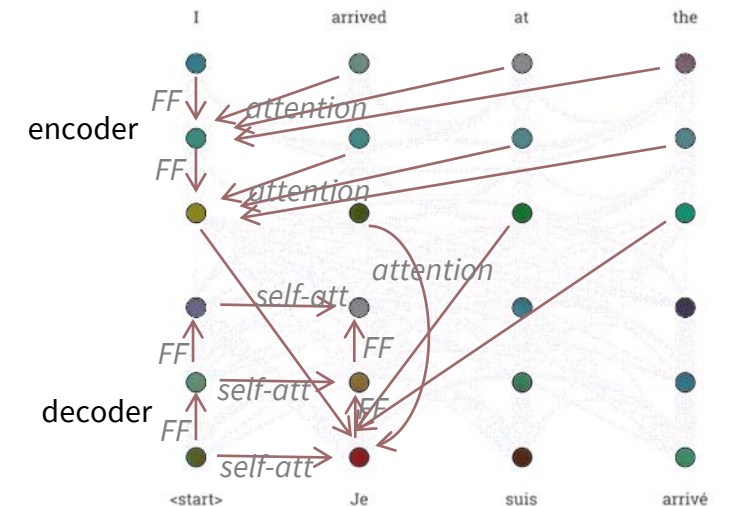
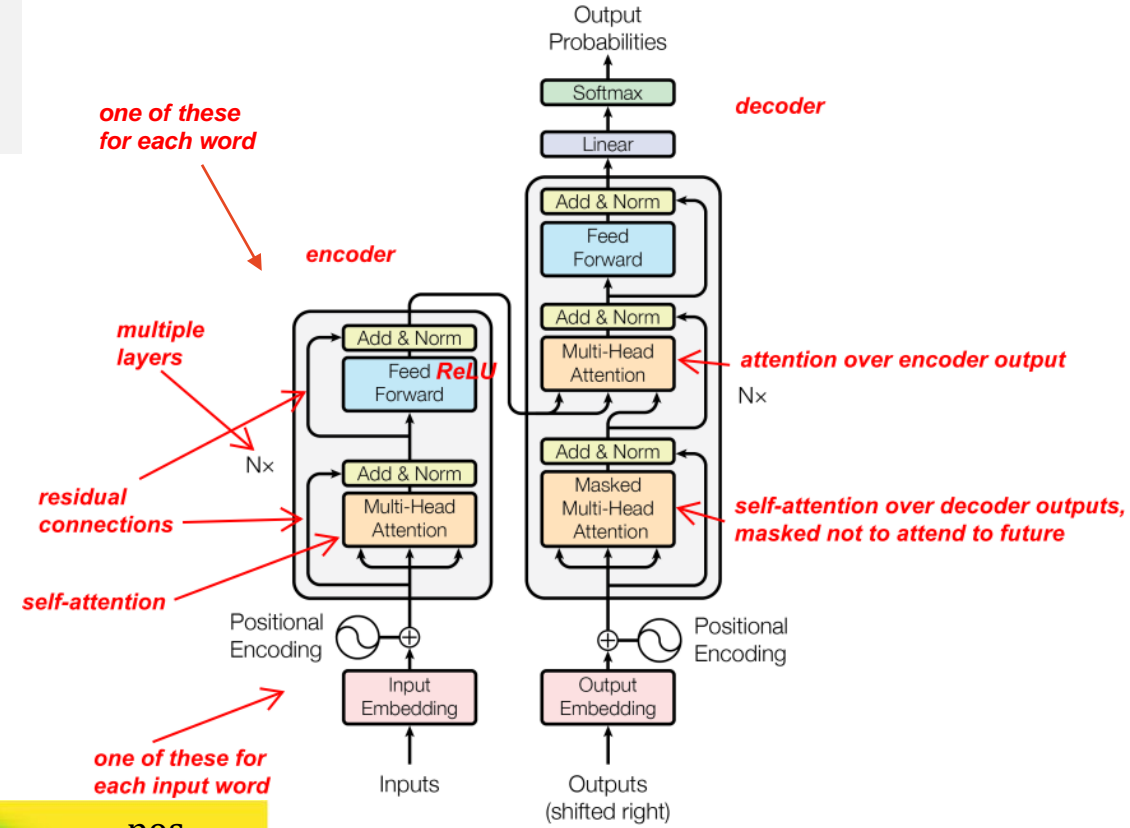
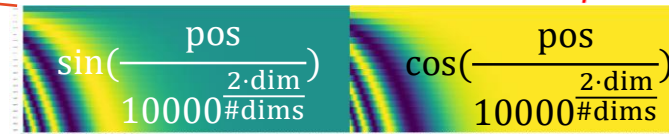


<http://cnyah.com/2017/08/01/attention-variants/>

Transformer

(Waswani et al., 2017)
<https://arxiv.org/abs/1706.03762>

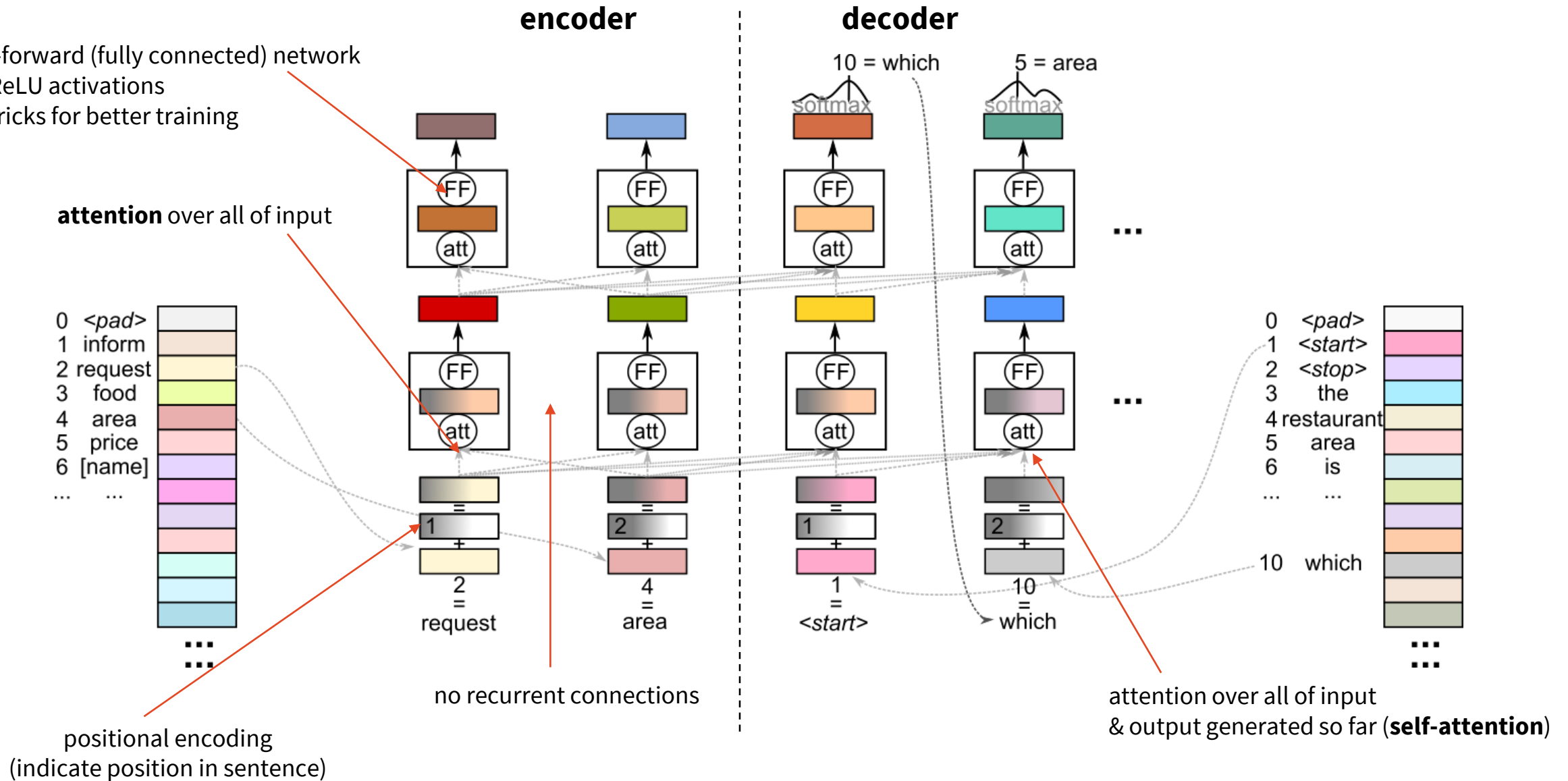
- getting rid of (encoder) recurrences
 - making it faster to train, allowing bigger nets
 - replace everything with attention + feed-forward networks
 - \Rightarrow needs more layers
 - \Rightarrow needs to encode positions
- positional encoding
 - adding position-dependent patterns to the input
- attention – dot-product (Luong style)
 - scaled by $\frac{1}{\sqrt{\text{\#dims}}}$ (so values don't get too big)
 - **more heads** (attentions in parallel)
 - focus on multiple inputs



Transformer

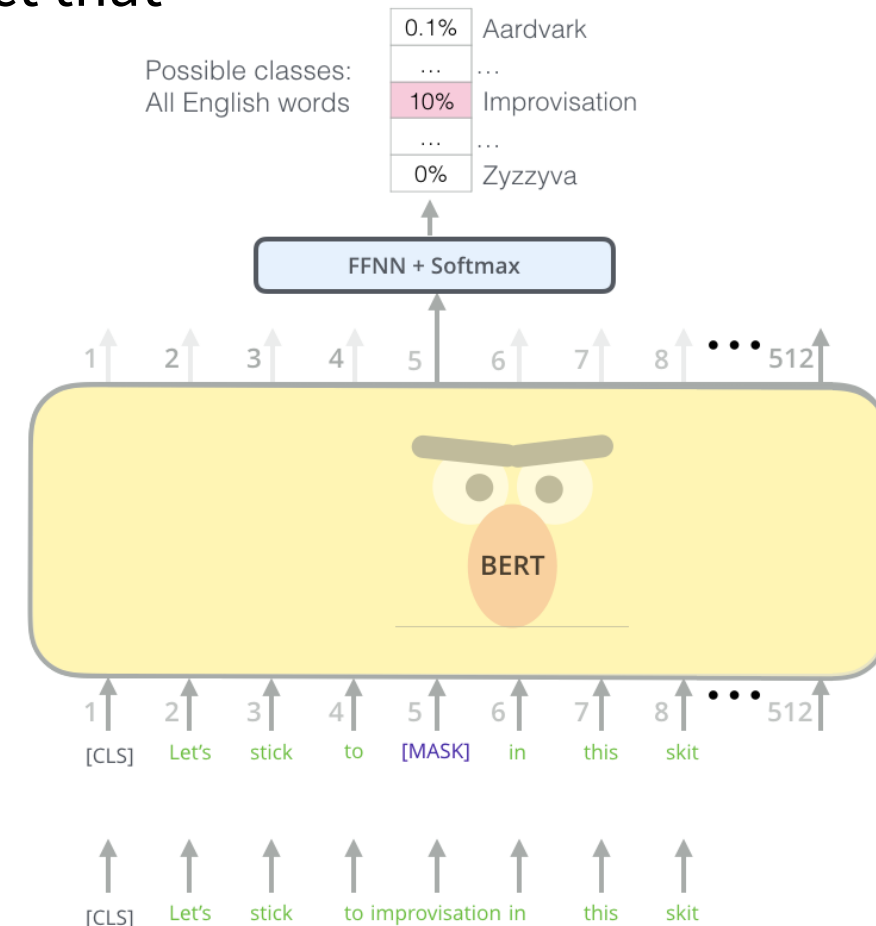
feed-forward (fully connected) network

- ReLU activations
- tricks for better training





- Beyond pretrained word embeddings
 - words have different meanings based on context
 - static word embeddings (word2vec/GloVe) don't reflect that
- **ELMo**
 - LSTMs trained for language modelling
 - ELMo embeddings = weighted sum of input static embeddings & LSTM outputs
 - the weights are trained for a specific downstream task
- **BERT**
 - huge Transformer encoder trained for:
 - masked word prediction
 - adjacent sentences detection (does B come right after A?)
 - BERT embeddings = any combination of the Transformer layers



Summary

- ML as a function mapping in → out
 - input features – 1-hot, numeric, **embeddings**
 - pretrained embeddings
 - contextual embeddings
 - function: layers ~ pipeline, data flows through (= complicated function)
 - outputs: classification (category), regression (float)
 - structured prediction – sequence tagging, ranking, generation
- Neural networks (~function shapes)
 - feed-forward/fully connected
 - CNNs (filters, pooling)
 - RNNs (LSTMs, GRUs)
 - encoder-decoder (seq2seq)
 - attention, **Transformer** (positional encoding & feed-forward & attention)
- Next week: how to train this stuff

Contact us:

[https://ufaldsg.slack.com/
{odusek,hudecek,nekvinda}@ufal.mff.cuni.cz](https://ufaldsg.slack.com/{odusek,hudecek,nekvinda}@ufal.mff.cuni.cz)
Troja N231/N233 (by agreement)

No lab today
Next week: lecture & lab
Monday 15:40

Get the slides here:

<http://ufal.cz/npfl099>

References/Further:

Goodfellow et al. (2016): Deep Learning, MIT Press (<http://www.deeplearningbook.org>)

Kim et al. (2018): Tutorial on Deep Latent Variable Models of Natural Language

(<http://arxiv.org/abs/1812.06834>)

Milan Straka's Deep Learning slides: <http://ufal.mff.cuni.cz/courses/npfl114/1819-summer>

Neural nets tutorials:

- <https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0>
- <https://minitorch.github.io/index.html>
- <https://objax.readthedocs.io/en/latest/>