NPFL099 Statistical Dialogue Systems
3. Neural Nets Basics

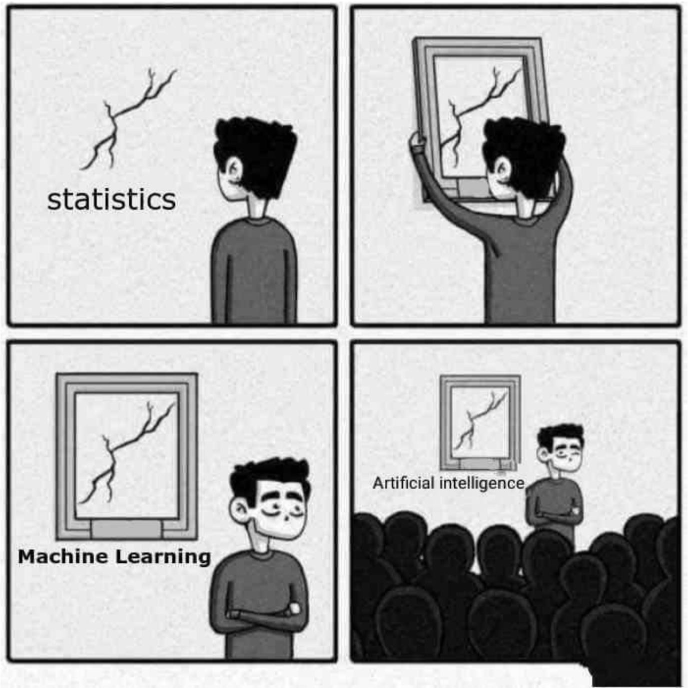
http://ufal.cz/npfl099

Ondřej Dušek, Simone Balloccu, Zdeněk Kasner, Mateusz Lango,
Ondřej Plátek, Patrícia Schmidtová
17. 10. 2023

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

Machine Learning

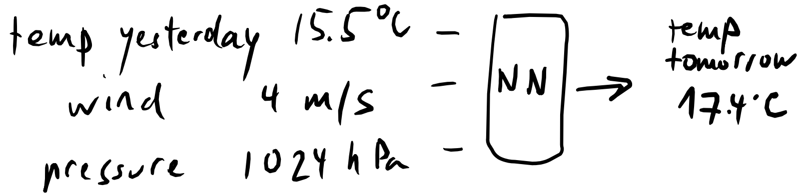
- ML is basically function approximation
- function: data (**features**)→ **labels**
- function shape:
  - this is where different ML algorithms differ
  - **neural nets**: compound non-linear functions
- training/learning = adjusting function parameters to minimize error (see next week)
  - **supervised learning** = based on data + labels given in advance
  - **reinforcement learning** = based on exploration & rewards given online



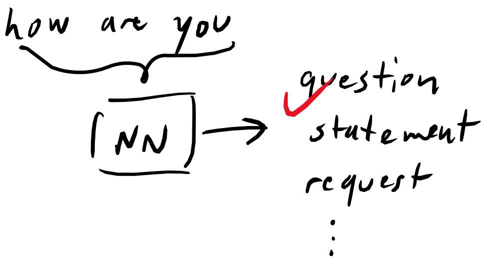
https://towardsdatascience.com/no-machine-learning-is-not-just-glorified-statistics-26d3952234e3

Typical machine learning problems in NLP

- **regression**
  - many inputs, 1 float output

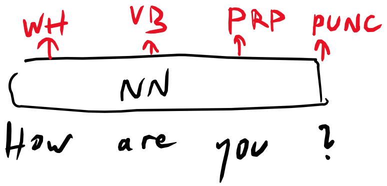


- **classification**
  - many inputs, 1 categorial output (k classes)

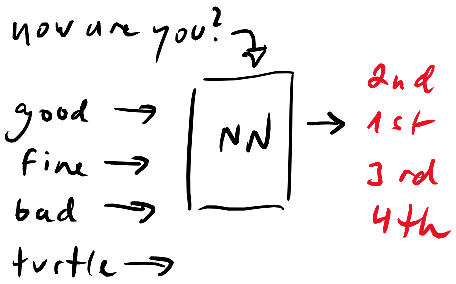


structured prediction

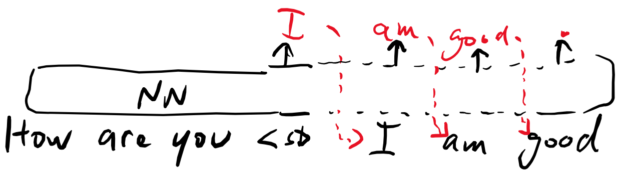
- **sequence labelling**
  - sequence of inputs, label each (~ repeated classification)
  - 1-to-1 input to output



- **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



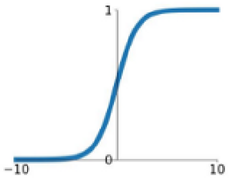
Neural networks

- **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:

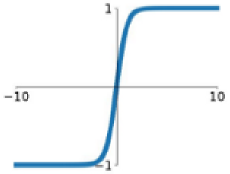
$$\mathrm{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{|\mathbf{x}|} \exp(x_j)}$$

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

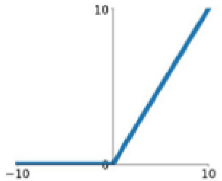
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



tanh
$\tanh(x)$



ReLU
$\max(0, x)$



https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

Layers visualization

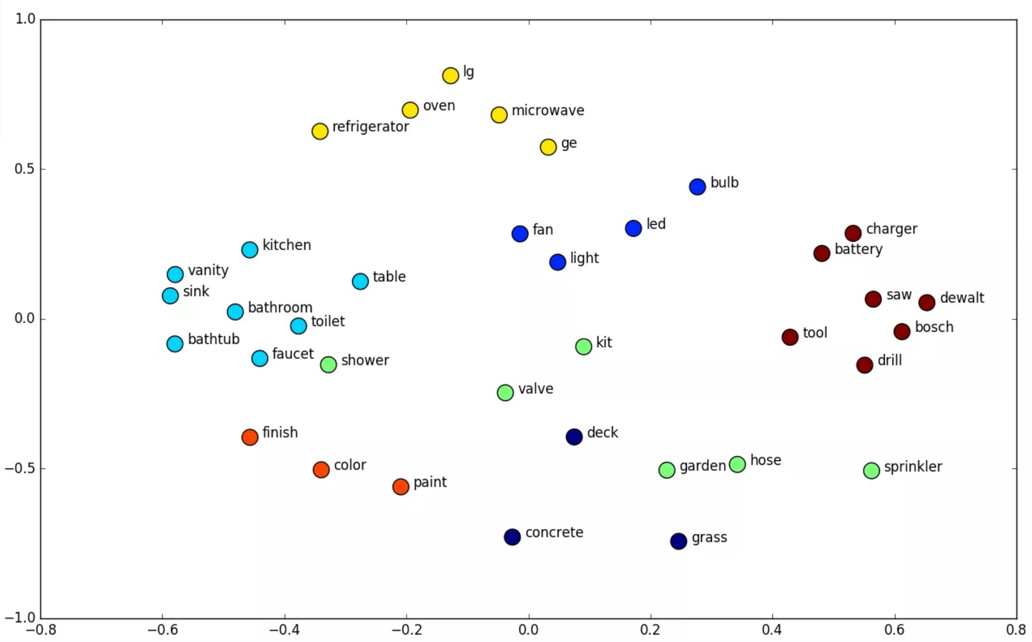
- https://playground.tensorflow.org/
  - 2 numeric features (=2 input variables) → binary classification (=1 output, 2 classes)
    - easiest case, but you can see the internals
    - more complex input features (→)
  - **feed-forward = fully connected = multi-layer perceptron** here
    - easiest case: 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 flow graph 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 → numeric
  - a dictionary of trainable features
- part of network parameters – trained
  a) random initialization
  b) pretraining
- the network learns which words are used similarly
  - they end up having close embedding values
  - embeddings end up different with different tasks & data & settings
- embedding size: ~100s-1000
- vocab size: ~50-100k

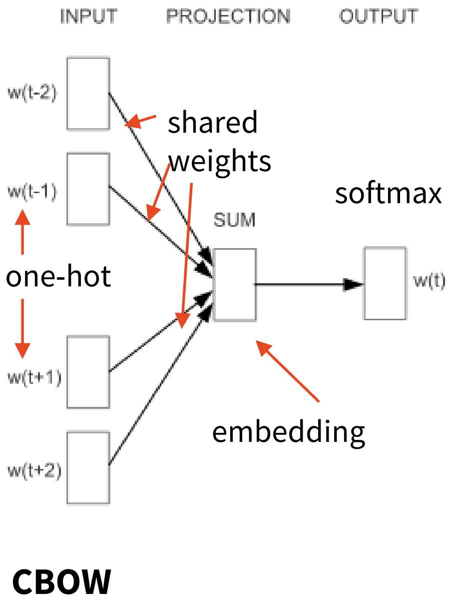


http://blog.kaggle.com/2016/05/18/home-depot-product-search-relevance-winners-interview-1st-place-alex-andreas-nurlan/

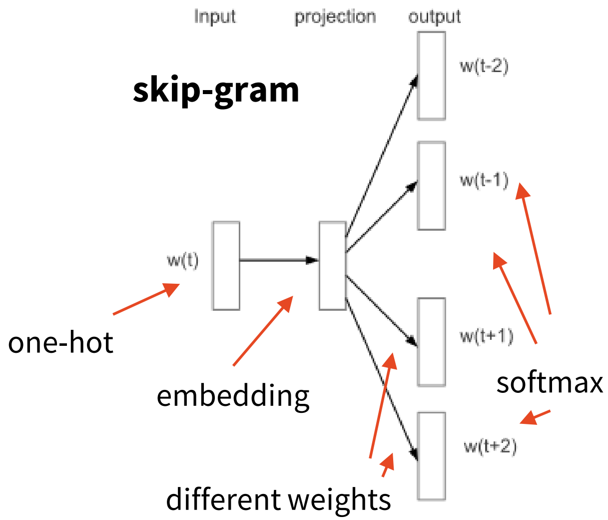
http://ruder.io/word-embeddings-2017/

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

https://geekyisawesome.blogspot.com/2017/03/word-embeddings-how-word2vec-and-glove.html
https://machinelearninginterview.com/topics/natural-language-processing/what-is-the-difference-between-word2vec-and-glove/

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”

(Sennrich et al., 2016)
https://www.aclweb.org/anthology/P16-1162/

f a s t _
f a s t e r _
t a l l _
t a l l e r _

→

fast er _
tall er _
s l o w er _
tall e s t _

- **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

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 (multiple) filter(s) 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

input

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

filter

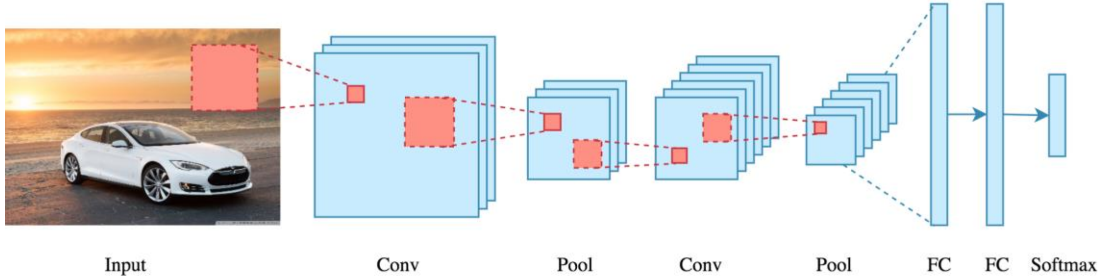
1	0	1
0	1	0
1	0	1

input x filter

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

4		

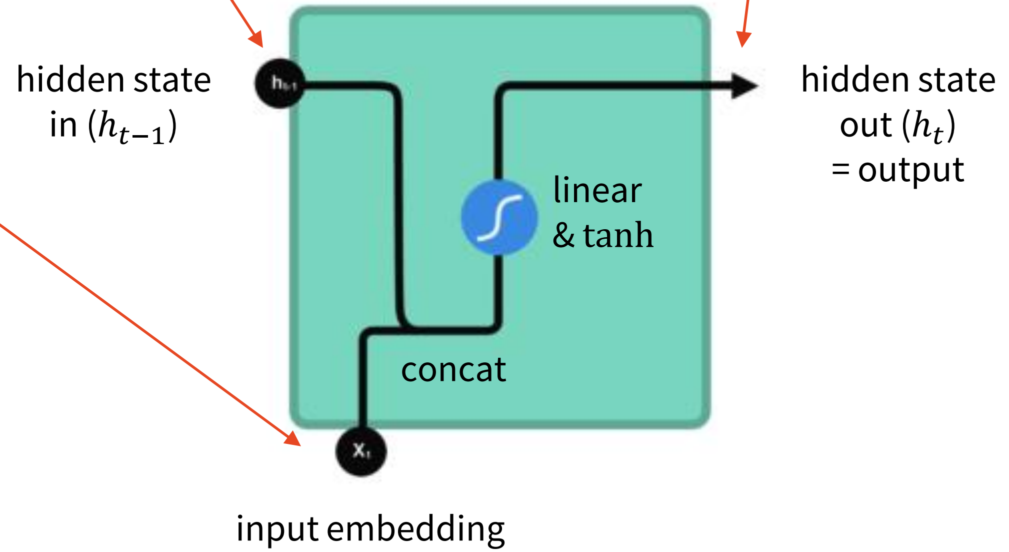
output

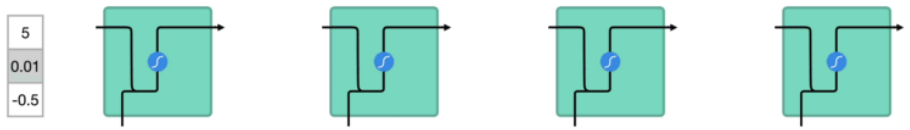


https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

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

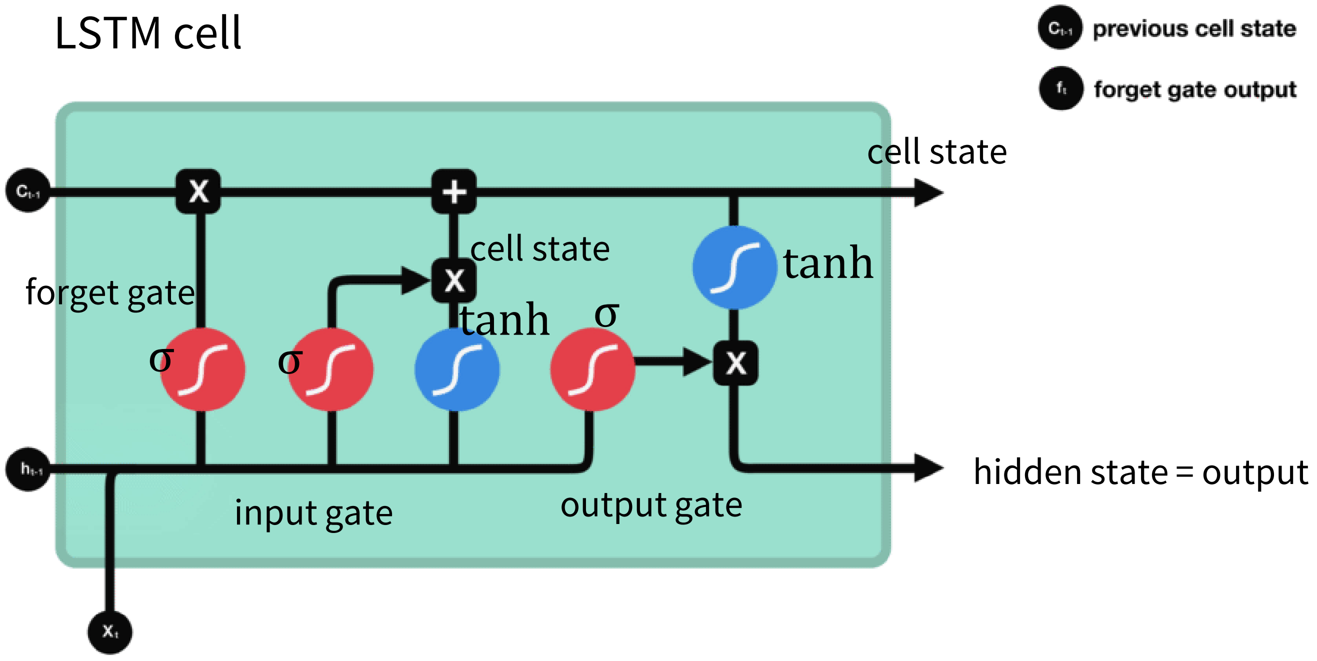


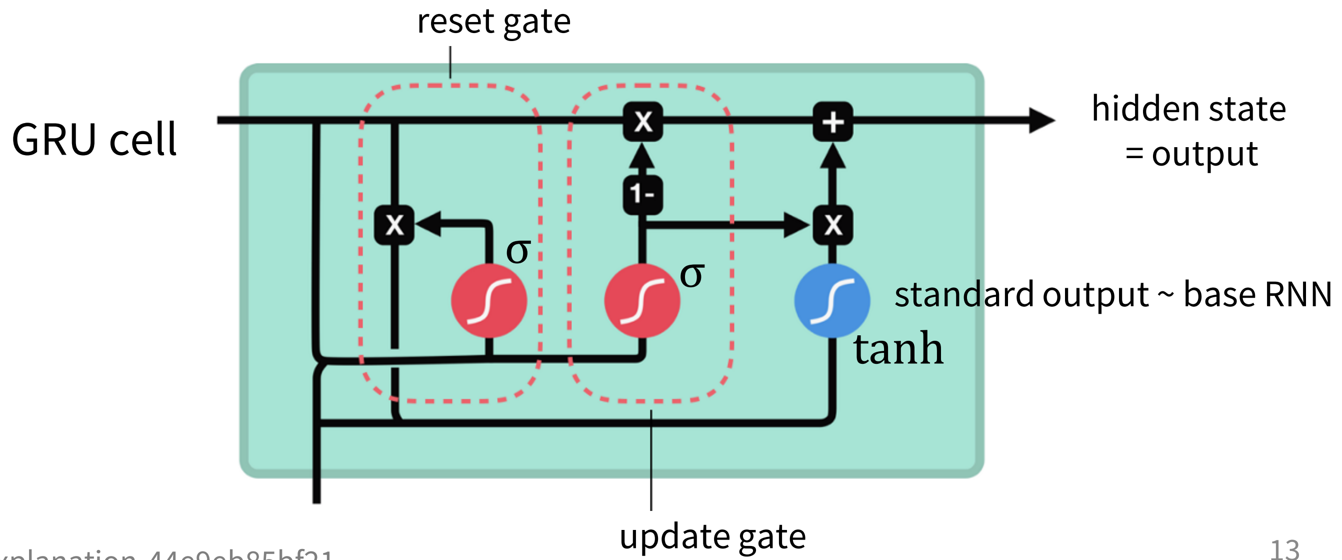


https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21
https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57

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



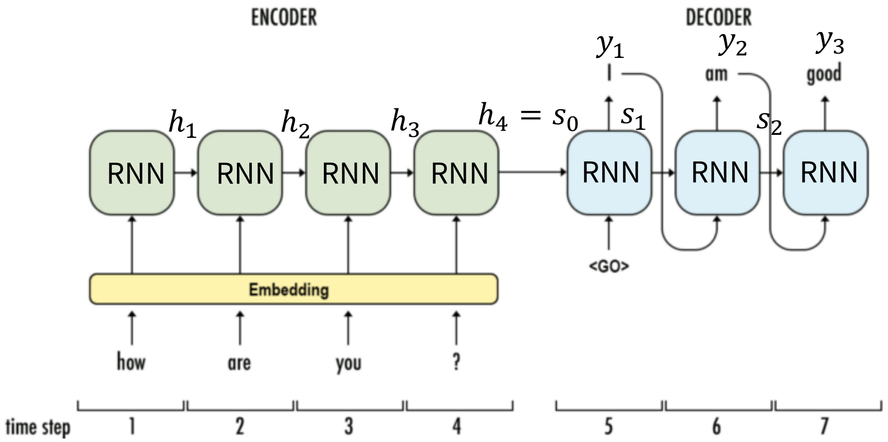


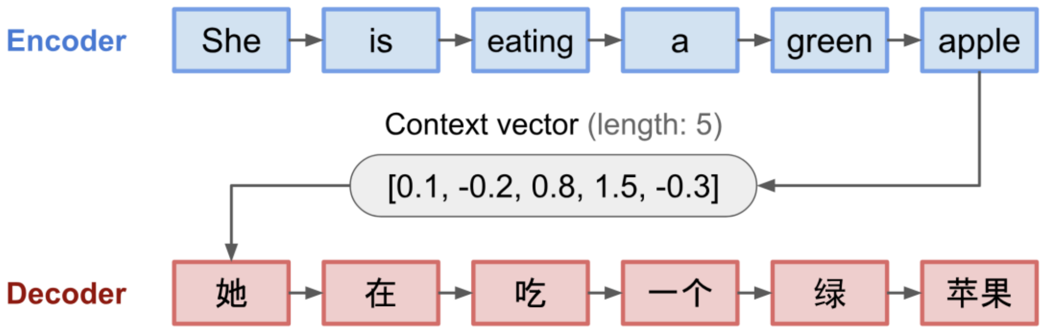
Encoder-Decoder Networks (Sequence-to-sequence)

- 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

$$\boldsymbol{h}_0 = \mathbf{0}$$
$$\boldsymbol{h}_t = \mathrm{cell}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$$

$$\boldsymbol{s}_0 = \boldsymbol{h}_T$$
$$p(y_t | y_1, \dots y_{t-1}, \mathbf{x}) = \mathrm{softmax}(\boldsymbol{s}_t)$$
$$\boldsymbol{s}_t = \mathrm{cell}(\boldsymbol{y}_{t-1}, \boldsymbol{s}_{t-1})$$



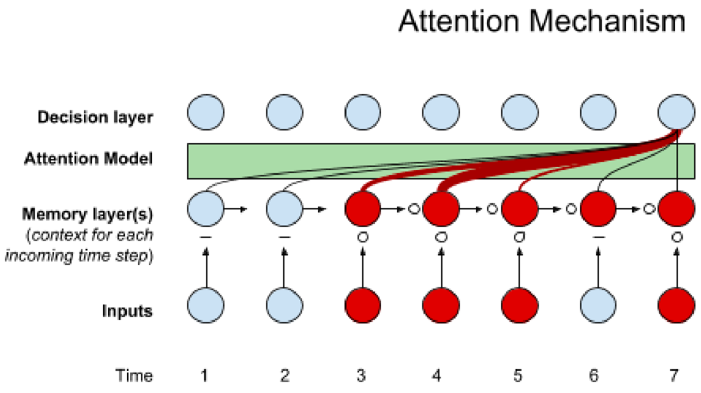


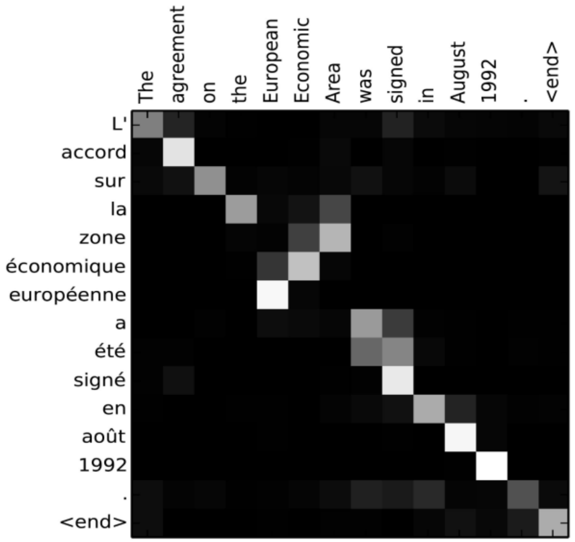
https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

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

Attention

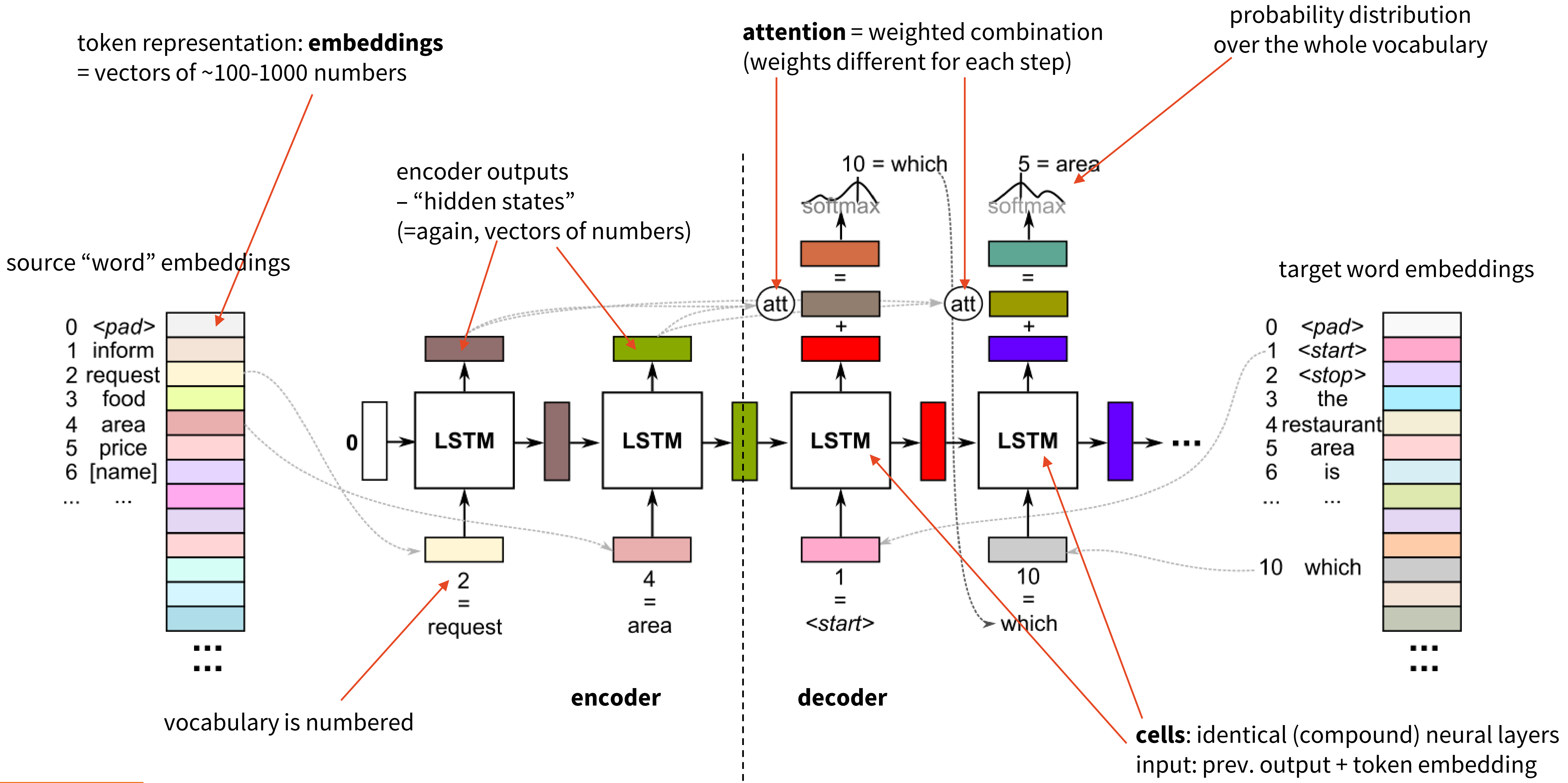
- Encoder-decoder is too crude for complex sequences
  - the whole input is crammed into a fixed-size vector (last hidden state)
- **Attention** = “memory” of **all encoder** hidden states
  - weighted combination, re-weighted for every decoder step
    → can focus on currently important part of input
  - 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

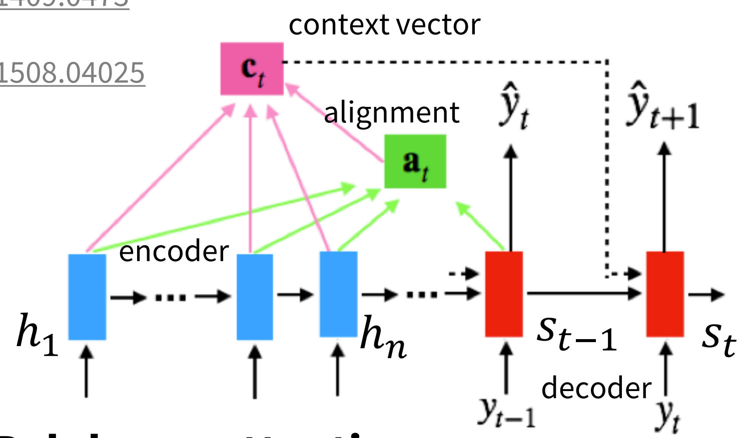


(Bahdanau et al., 2015) http://arxiv.org/abs/1409.0473

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



Bahdanau attention

attention value = context vector

sum of encoder hidden states
weighted by attention weights α_{ti}

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

attention weights = alignment model

decoder state
trained parameters

Bahdanau:

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

encoder hidden state

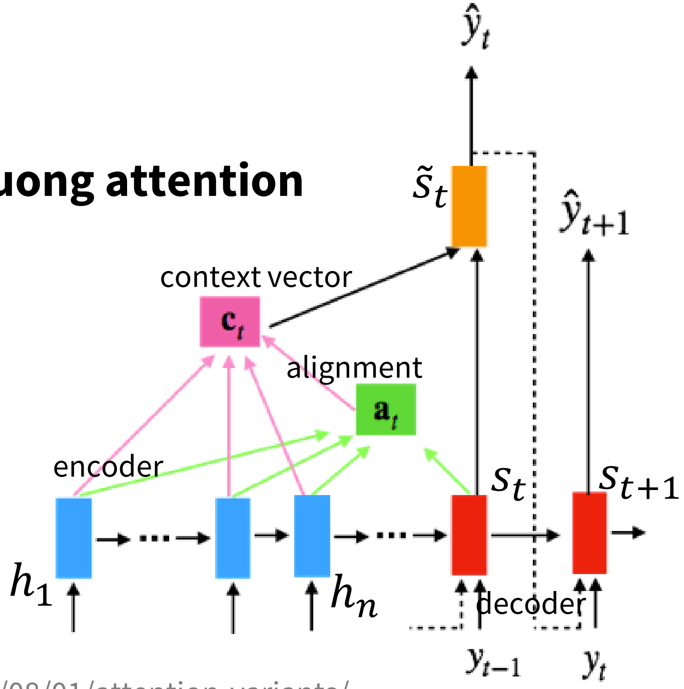
Luong:

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

decoder state

encoder hidden state

Luong attention

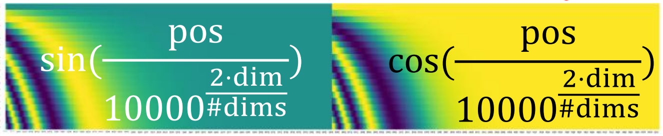


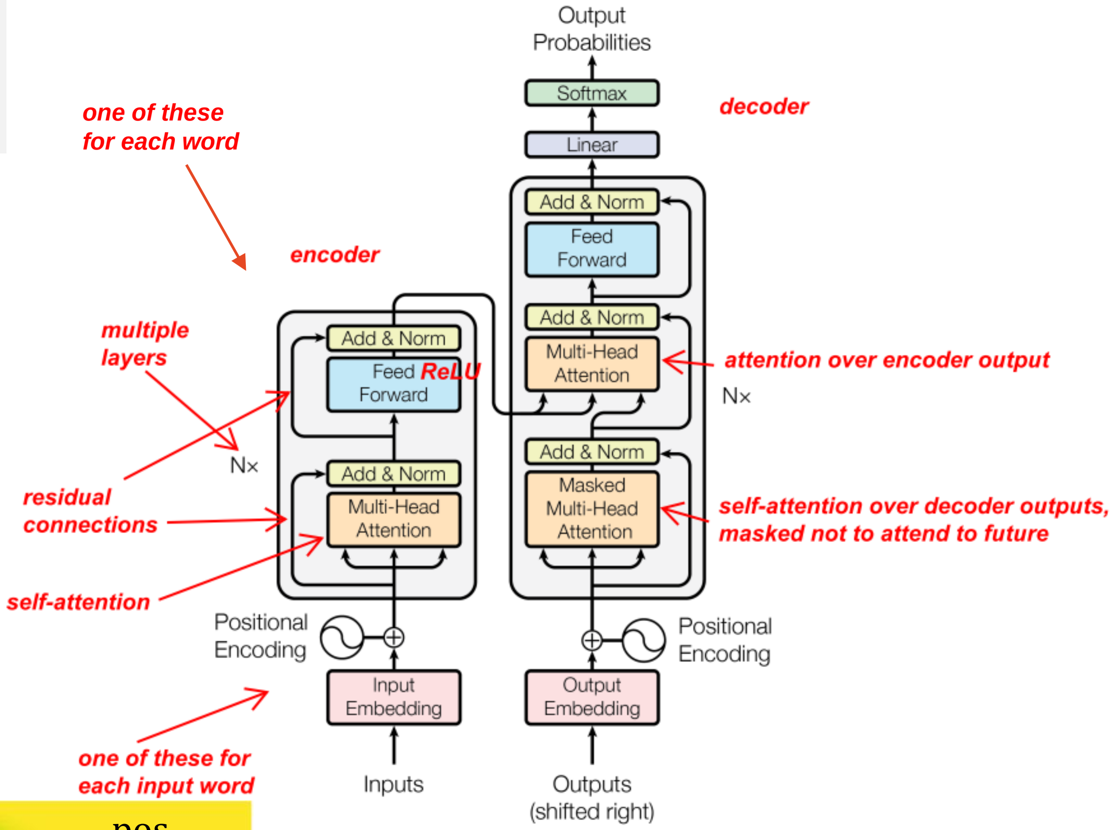
http://cnyah.com/2017/08/01/attention-variants/

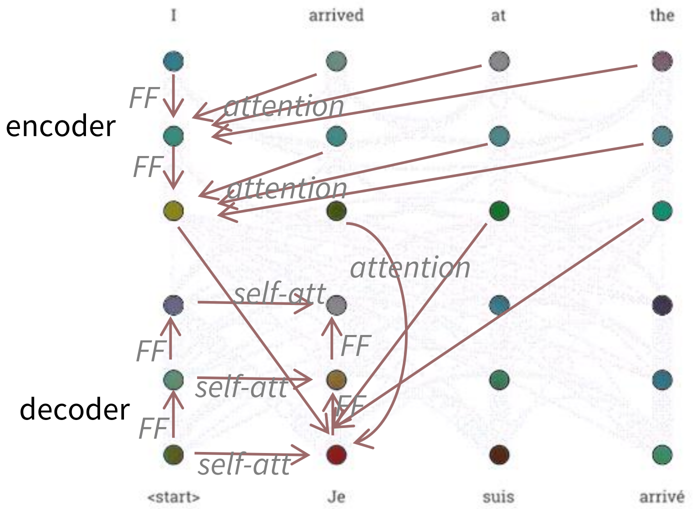
Transformer

(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
  - ⇒ needs more layers
  - ⇒ needs to encode positions
- positional encoding
  - adding position-dependent patterns to the input
- attention – dot-product (Luong style)
  - scaled by $\frac{1}{\sqrt{\#\mathrm{dims}}}$ (so values don't get too big)
  - **more heads** (attentions in parallel)
    – focus on multiple inputs





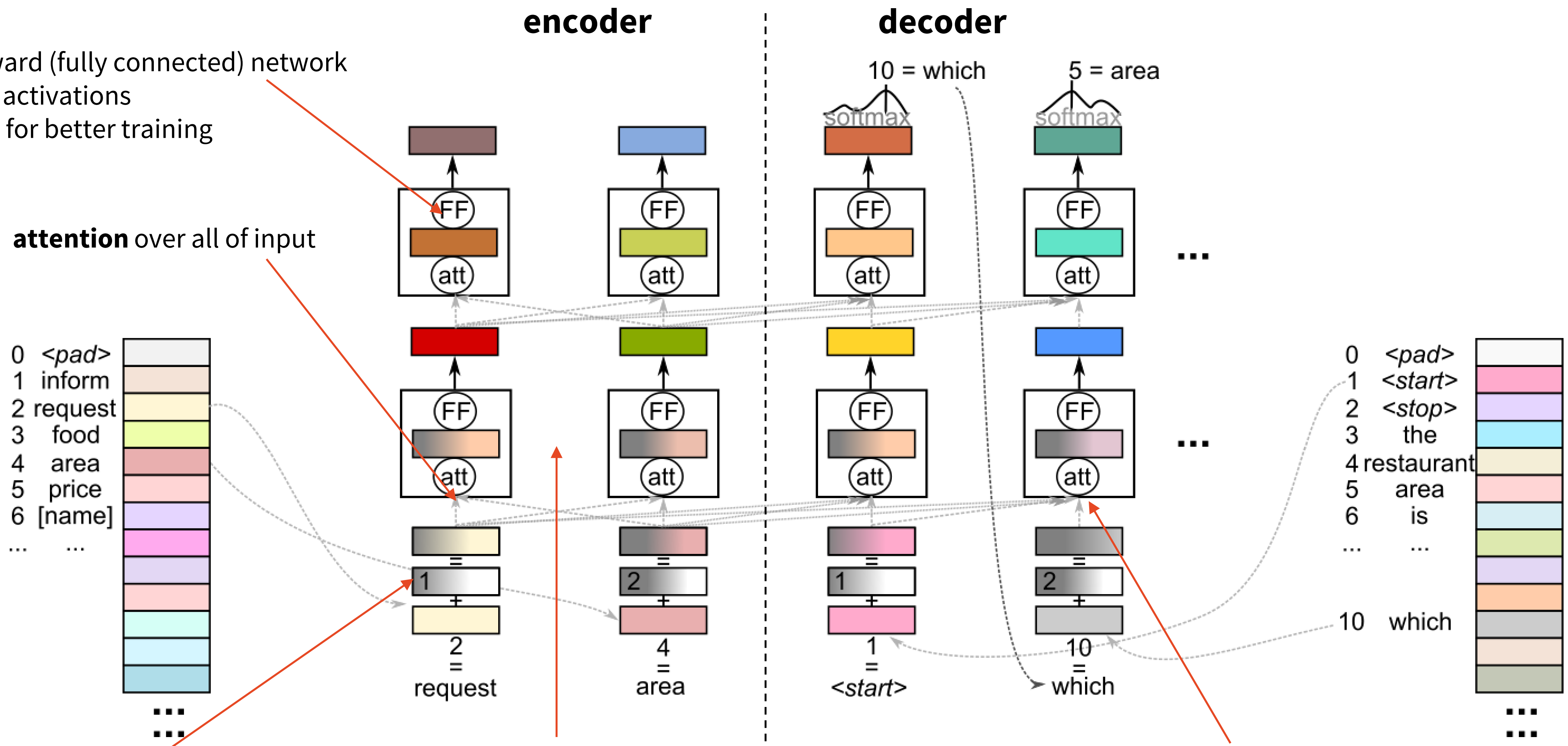


http://jalammar.github.io/illustrated-transformer/ https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

Transformer

feed-forward (fully connected) network
- ReLU activations
- tricks for better training

attention over all of input



no recurrent connections

attention over all of input
& output generated so far (**self-attention**)

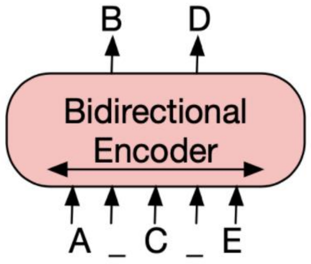
positional encoding
(indicate position in sentence)

(Vaswani et al., 2017) http://arxiv.org/abs/1706.03762

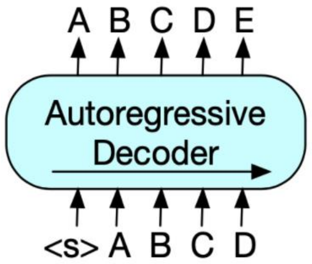
Pretrained Language Models

https://lilianweng.github.io/posts/2019-01-31-lm/
https://github.com/jessevig/bertviz

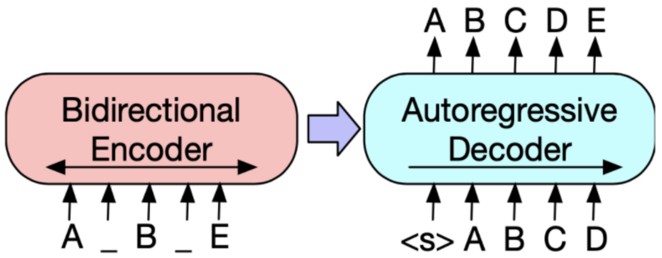
- Beyond pretrained word embeddings
  - reflects different word meanings in sentence context (~contextual embeddings)
  - used as input to added layers on top / base for model finetuning (next week)
- LSTM-based: **ELMo** (trained on language modelling)
  - weighted sum of static word embeddings & LSTM outputs



- **Transformer encoders**: **BERT**, **RoBERTa**…
  - for classification, sequence tagging
  - any Transformer layer used (typically the last one)

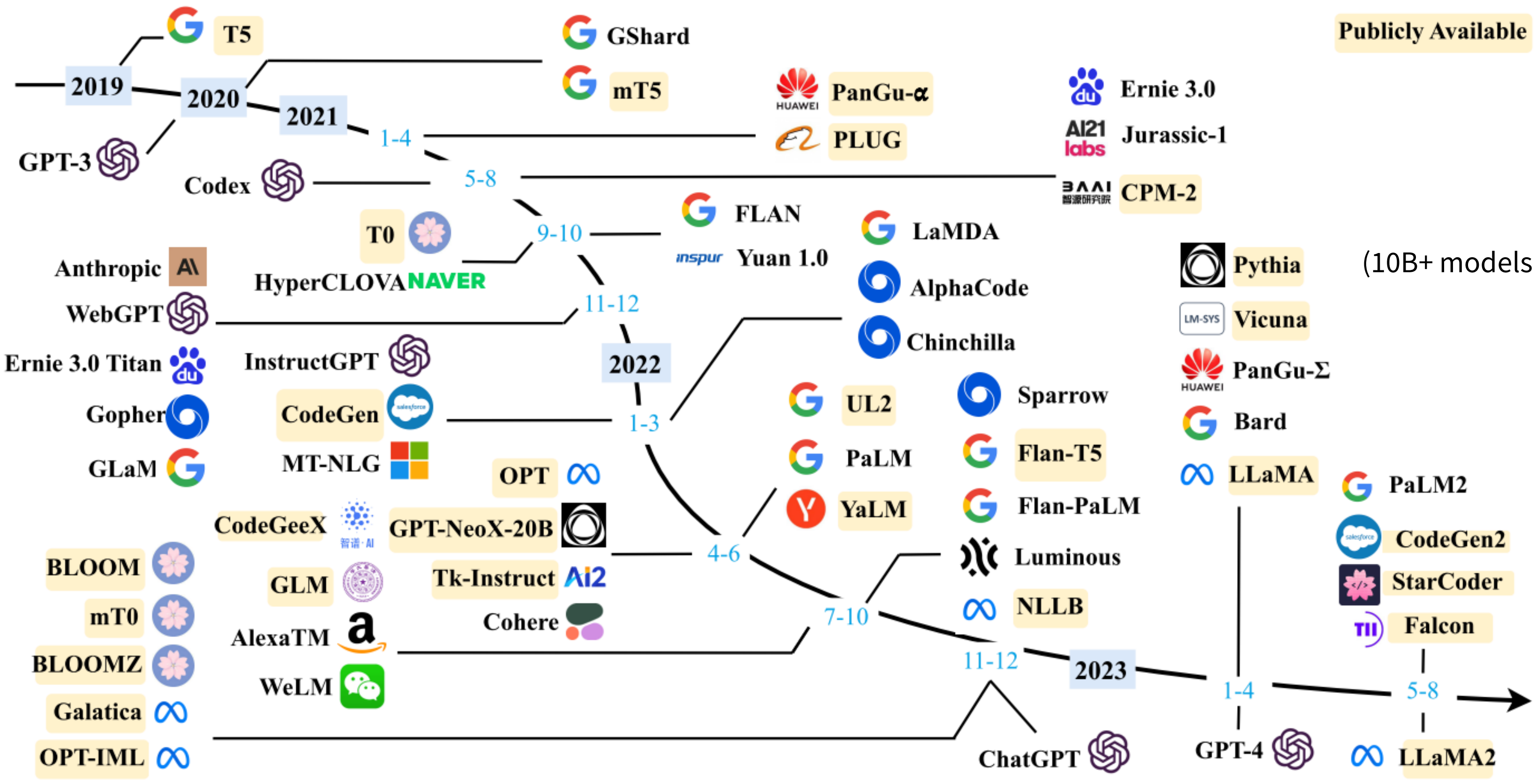


- **Transformer decoders**: **GPT-2**, **GPT-3**…
  - for generation, language modelling
  - input: **force-decoding**
- **Transformer encoder-decoders**: **BART**, **T5**…
  - same as ↑, explicit input



Large Language Models

- Most are just Transformer decoders
- Only difference w.r.t. previous: **size** (& training methods – see next time)
  - BERT/GPT2/T5 etc.: typically 100M-1B params
  - LLMs: 7B-60B (LlaMa), 100B+ (GPT3, PaLM…), unknown (ChatGPT, GPT4…)



(10B+ models shown here)

(Zhao et al., 2023)
http://arxiv.org/abs/2303.18223

Summary

- ML as a function mapping in → out
  - input features – 1-hot, numeric, **embeddings**
    - pretrained 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)
    - **pretrained models**
- Next week: how to train this stuff

Thanks

Contact us:

https://ufaldsg.slack.com/
odusek@ufal.mff.cuni.cz
Zoom/Skype/Troja

No lab today
Next week: lecture & lab
Tue 9:50

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/

LLM intro: https://www.understandingai.org/p/large-language-models-explained-with