Introduction to NLP

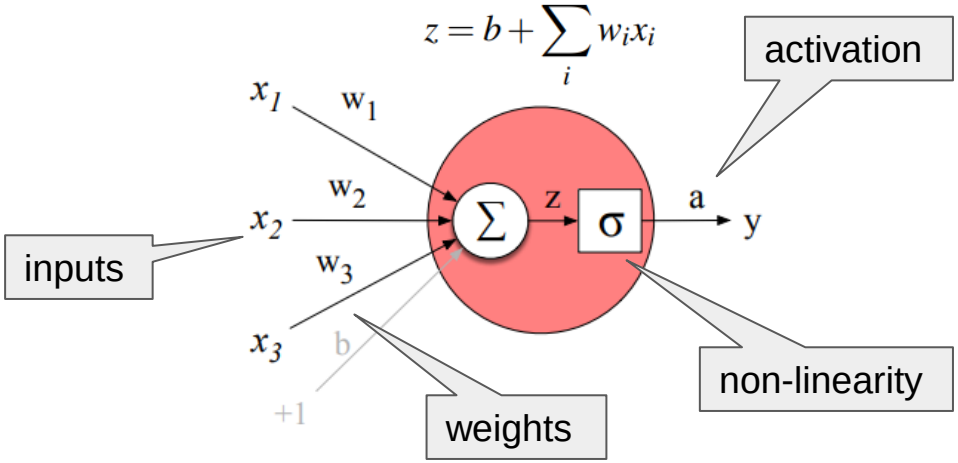
Neural models

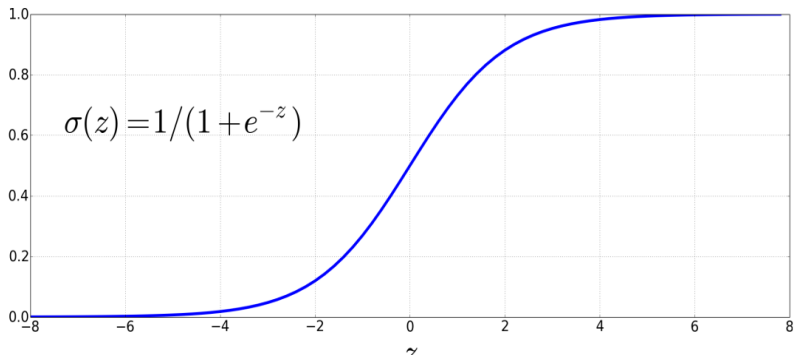
Compressed out of NLP courses from **Dan Jurafsky** (Stanford), & David Bamman (Berkeley), Michael Collins (MIT & Columbia), and some online (Udemy) courses

Book: **Speech and Language Processing** by Jurafsky & Martin (3rd edition)

A Neuron

- Basically a **Logistic Regression**
  - also used in NLP a lot





$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

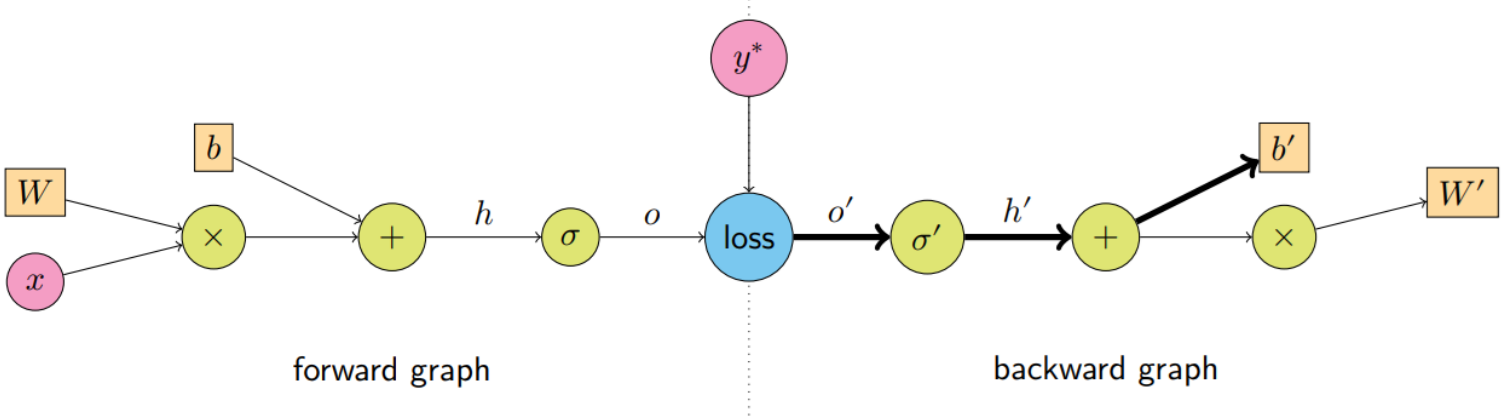
Computational graph: logistic regression

source: https://ufal.mff.cuni.cz/jindrich-libovicky

Logistic regression:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

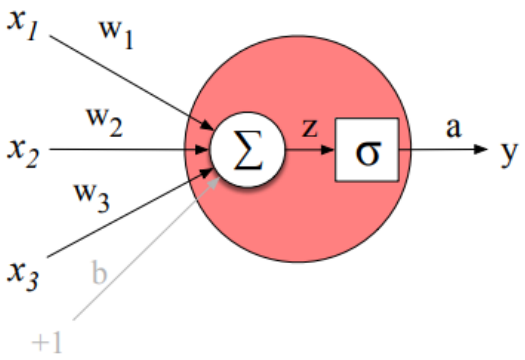
Computation graph:

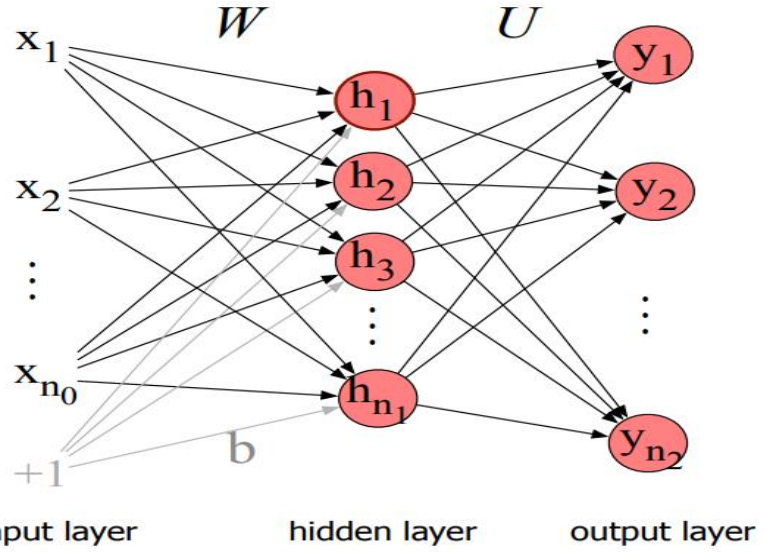


Feed-Forward Neural Networks

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

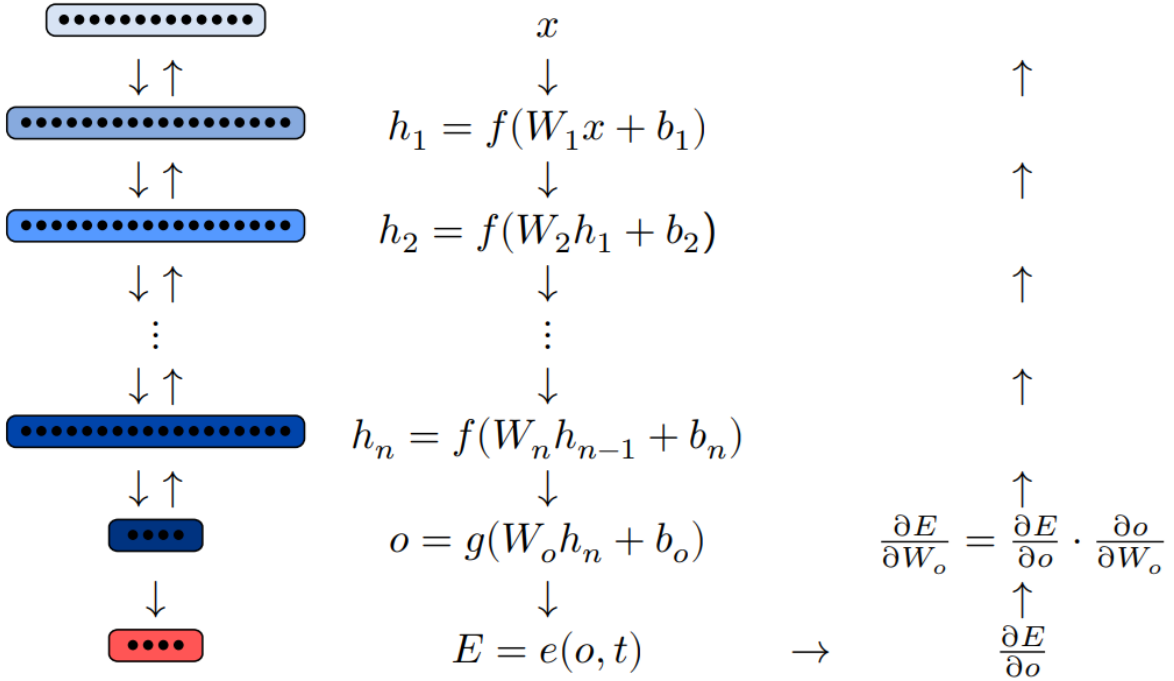
- Stacking the neurons into “hidden” layers
  - with which we move to **matrix notation**
- You are already familiar with these models from RPZ, SSU, UI, …





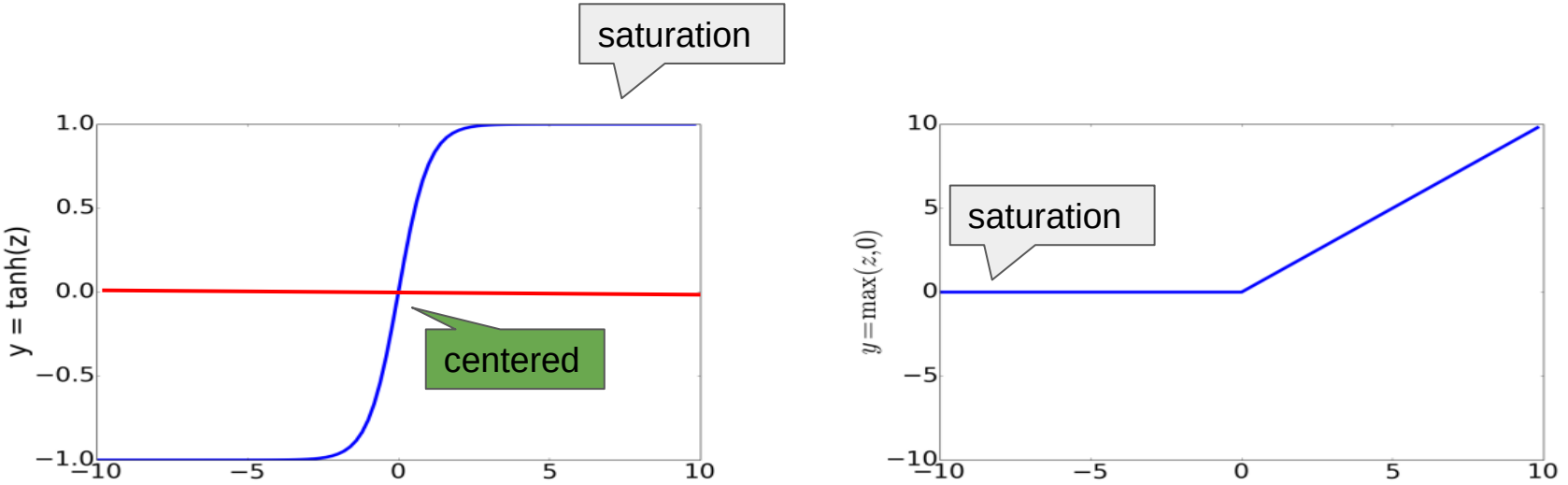
Feed-Forward Neural Networks

source: https://ufal.mff.cuni.cz/jindrich-libovicky



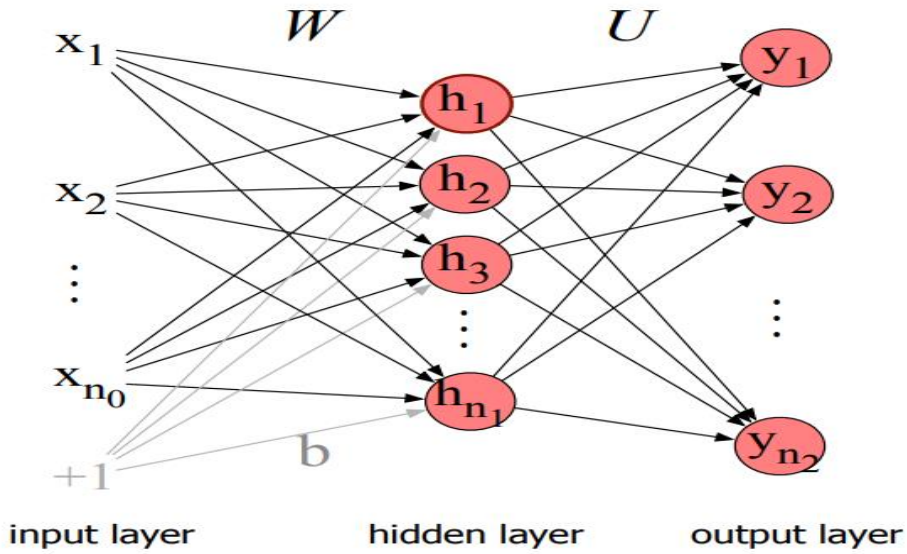
Activations

- Sigmoid non-linearity has some problems in practice
  - deviation + saturation
- Some better choices:



Feed-Forward Neural Networks

- To obtain multi-class probabilities, we should **normalize** the values

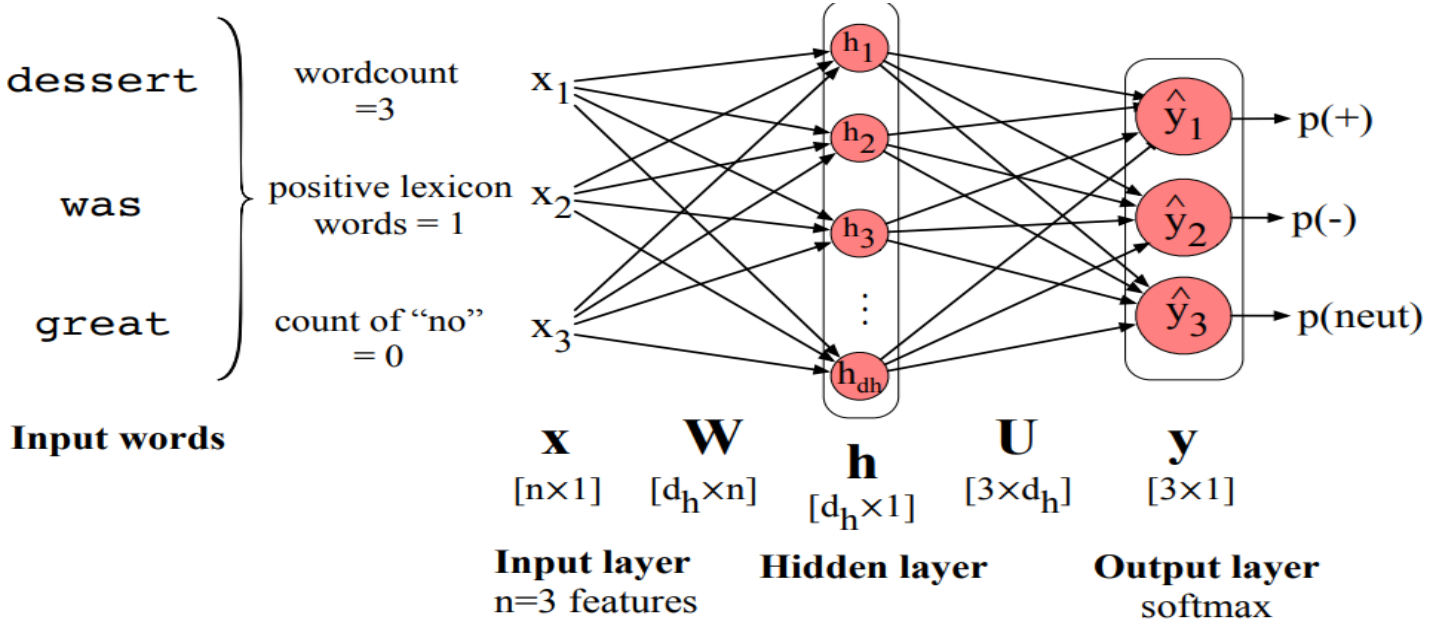


$$\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_N]$$
$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$
$$\mathbf{z} = \mathbf{U}\mathbf{h}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d} \exp(\mathbf{z}_j)} \quad 1 \leq i \leq d$$

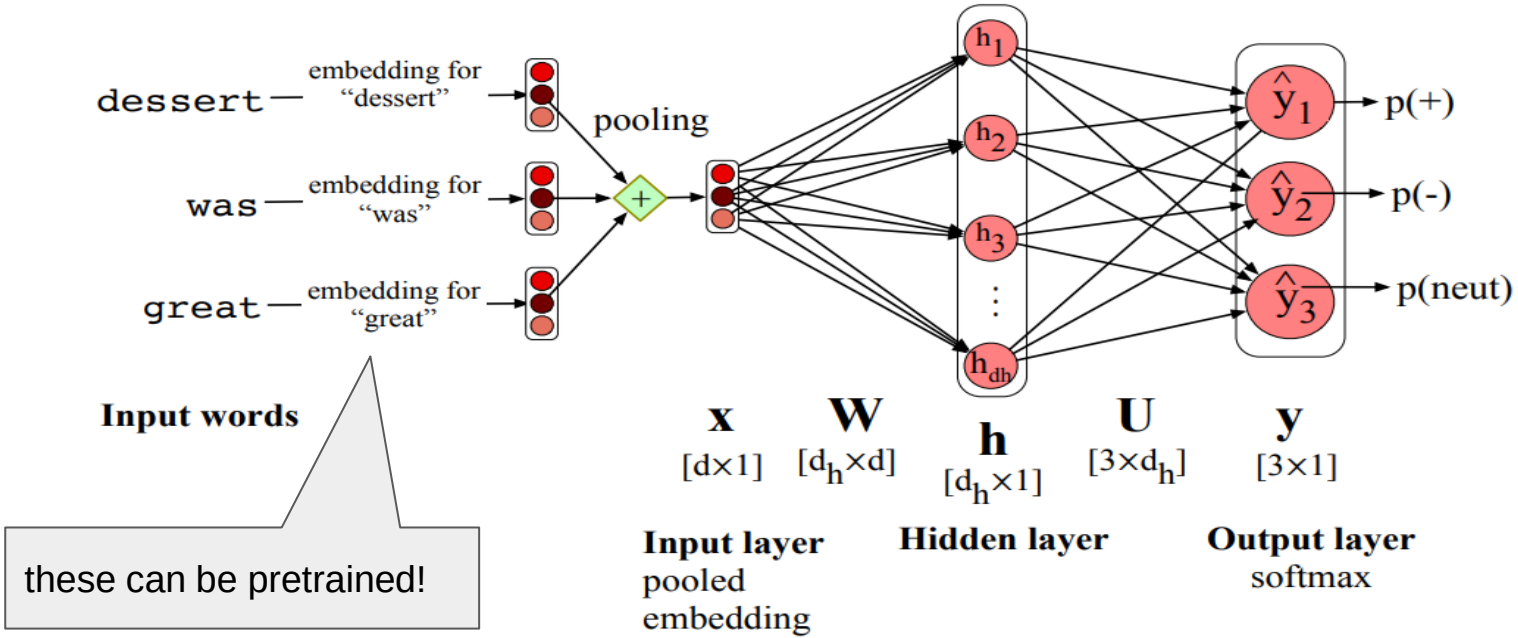
Feed-Forward Neural Networks

- Such neural networks can be readily applied to a range of NLP tasks



Deep Learning

- **Deep Learning** idea - instead of crafting features, we learn from **raw** text - aka **representation learning** - with our word **embeddings**!



Neural models

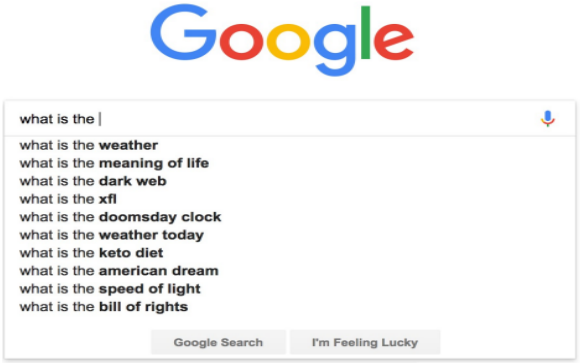
- revisiting Language Modelling

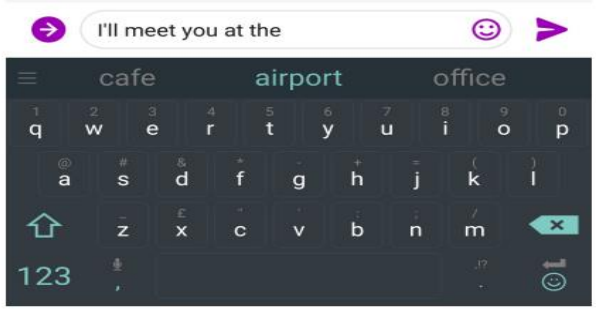
Neural Language Models

- Predict (upcoming) word given some (previous) context

$$P(w_t | w_1, \ldots, w_{t-1}) \approx P(w_t | w_{t-N+1}, \ldots, w_{t-1})$$

Lanuage modelling with n-grams





Sparsity Problem 1

Problem: What if *"students opened their w"* never occurred in data? Then *w* has probability 0!

(Partial) Solution: Add small δ to the count for every $w \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count}(\text{students opened their } \boldsymbol{w})}{\text{count}(\text{students opened their})}$$

Sparsity Problem 2

Problem: What if *"students opened their"* never occurred in data? Then we can't calculate probability for *any w*!

(Partial) Solution: Just condition on *"opened their"* instead. This is called *backoff*.

Note: Increasing *n* makes sparsity problems *worse*. Typically, we can't have *n* bigger than 5.

Neural Language Models

- Predict (upcoming) word given some (previous) context

$$P(w_t | w_1, \ldots, w_{t-1}) \approx P(w_t | w_{t-N+1}, \ldots, w_{t-1})$$

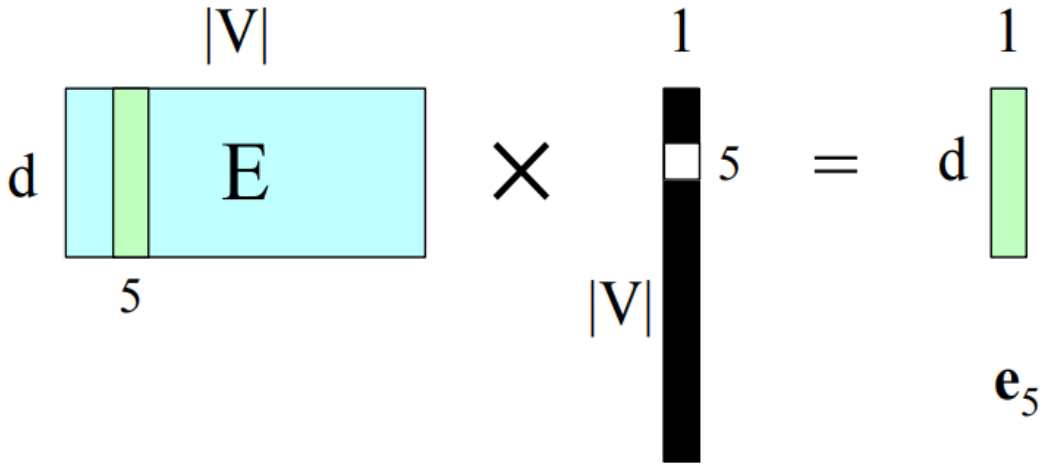
- We already did this with the **n-gram** counting Markov models
- However, neural LMs:
  - \+ handle longer histories
  - \+ **generalize over similar words**
  - \+ are generally more accurate
  - \- less interpretable
  - \- slower to train

*I have to make sure that the **cat** gets fed*
vs.
*I have to make sure that the **dog** gets fed*

One-hot vectors & embeddings

- A vector of length |V|
- 1 for the target word and 0 for other words
  - So if “apple” is vocabulary word **5** the **one-hot vector** is
  - [0 0 0 0 1 0 0 ... 0 0 0 0]
    1 2 3 4 5 6 7 ... … ...|V|

- Embedding matrix **E**
  - stores embeddings of all words in vocab



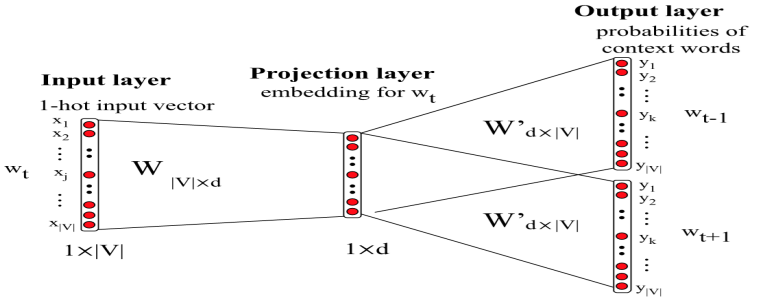
Revisiting **word2vec**

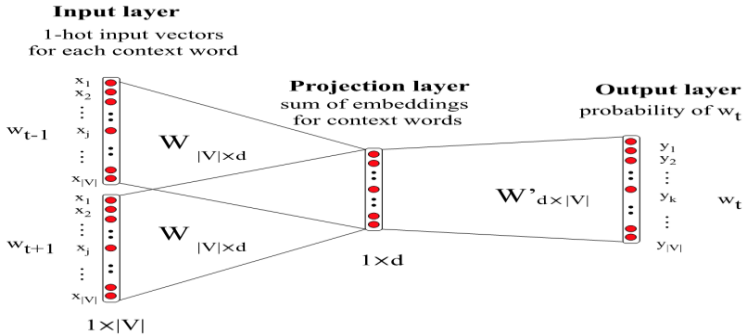
Word2vec = possibly the simplest “neural” LM

- 2 variants of the model training:

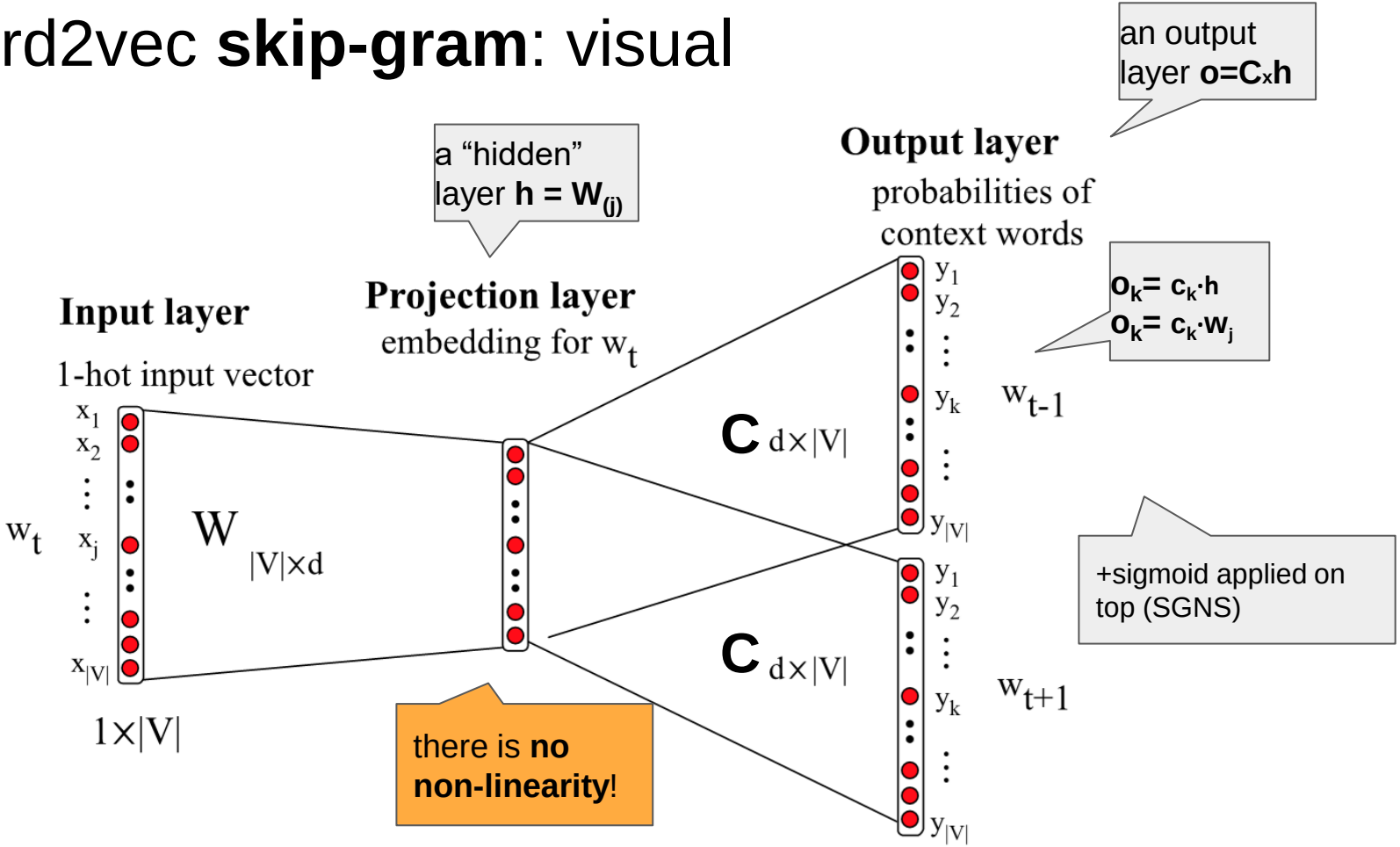
- Skip-gram
  - Predict each neighboring word from a given “middle” word

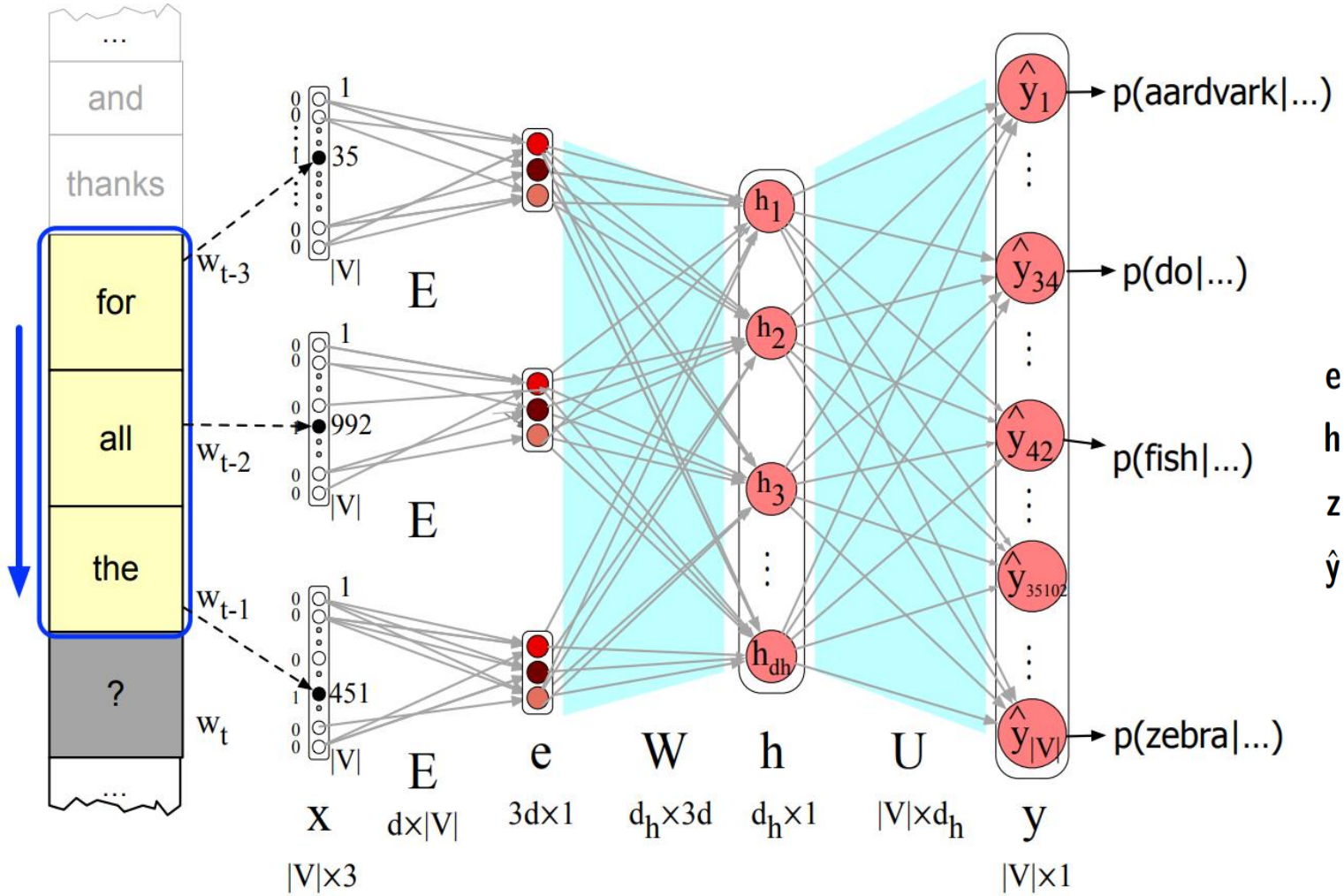
- CBOW (Continuous Bag Of Words)
  - Predict the “missing” middle word from its neighbors

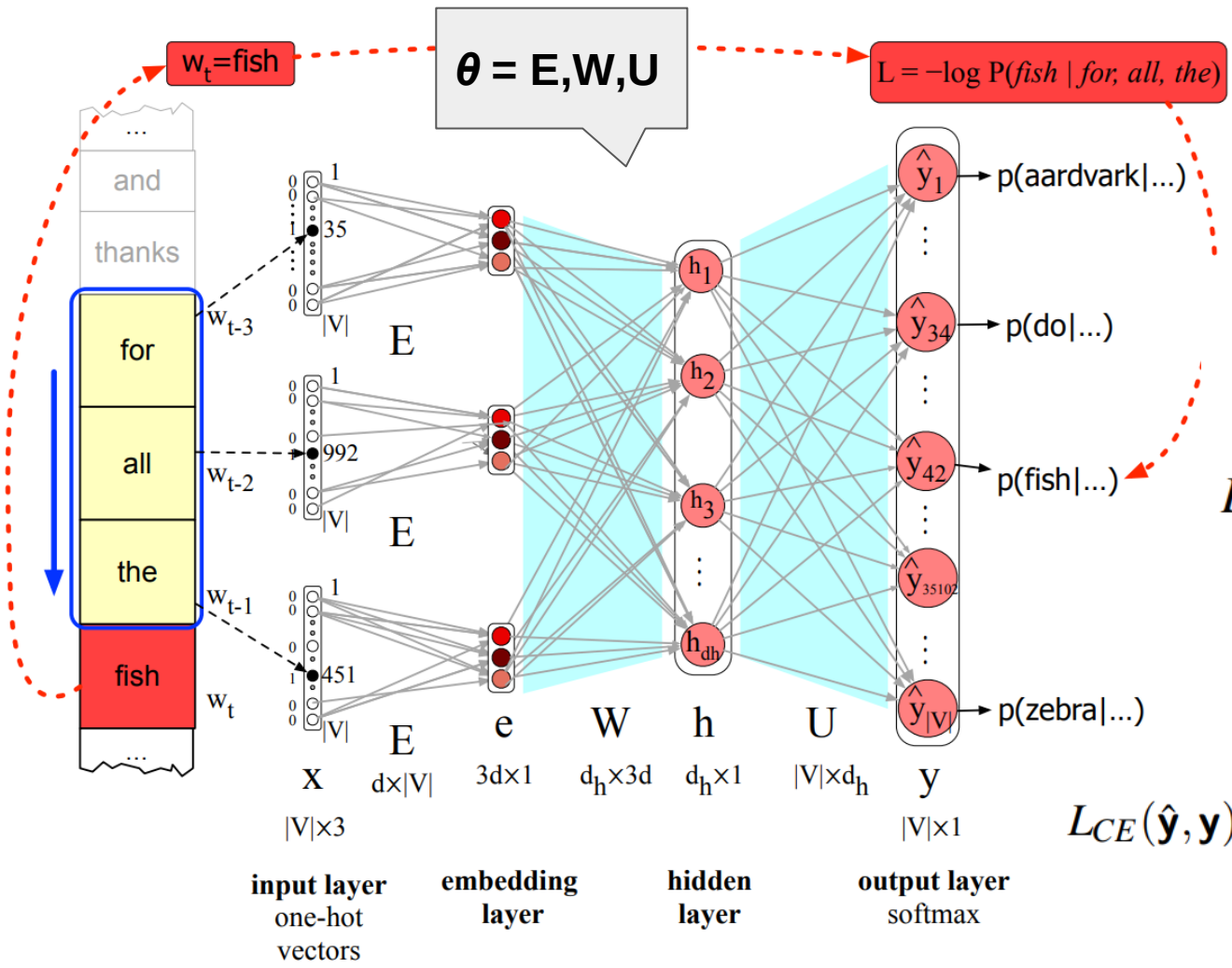




Word2vec **skip-gram**: visual




...
and
thanks
for
all
the
?
...
w_{t-3}
w_{t-2}
w_{t-1}
w_t
1
35
992
451
|V|
E
x
|V|×3
d×|V|
e
3d×1
W
d_h×3d
h
d_h×1
U
|V|×d_h
y
|V|×1
$\hat{y}_1$
$\hat{y}_{34}$
$\hat{y}_{42}$
$\hat{y}_{35102}$
$\hat{y}_{|V|}$
p(aardvark|...)
p(do|...)
p(fish|...)
p(zebra|...)
LM window of size = 3
$\mathbf{e} = [\mathbf{E}x_{t-3}; \mathbf{E}x_{t-2}; \mathbf{E}x_{t-1}]$
$\mathbf{h} = \sigma(\mathbf{W}\mathbf{e} + \mathbf{b})$
$\mathbf{z} = \mathbf{U}\mathbf{h}$
$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$



we train with **cross-entropy**

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k$$

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \hat{\mathbf{y}}_c$$

aka **negative log-likelihood**

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)}$$

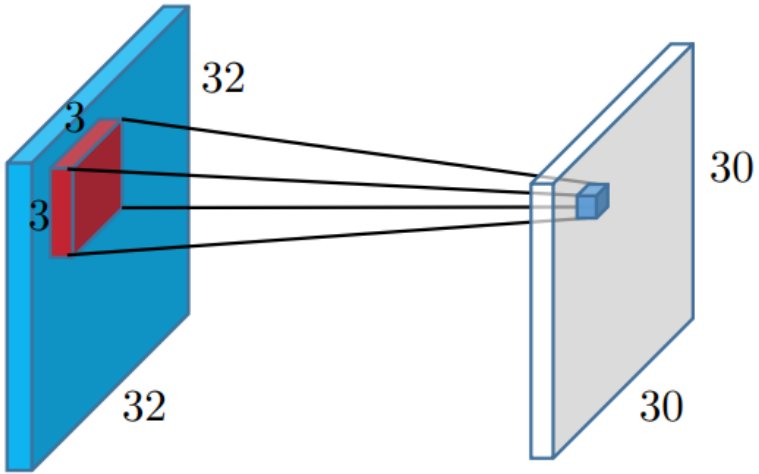
Neural Language models: **training**

- We initialize all the parameters θ = **E,W,U** randomly
  - good to start with zero mean and unit variance
- We iteratively move through the text, predicting each word w_t at a time
  - given the context of N previous words $W_{t-1} \ldots W_{t-n}$
  - Note that the embeddings are shared for all the N positions
- This is essentially a multi-class |V| classification problem
- We train against **cross-entropy** $L_{CE} = -\log p(w_t|w_{t-1}, \ldots, w_{t-n+1})$
- And optimize with **gradient-descent**
  - + tricks such as dropout

$$\theta^{s+1} = \theta^s - \eta \frac{\partial \left[-\log p(w_t|w_{t-1}, \ldots, w_{t-n+1})\right]}{\partial \theta}$$

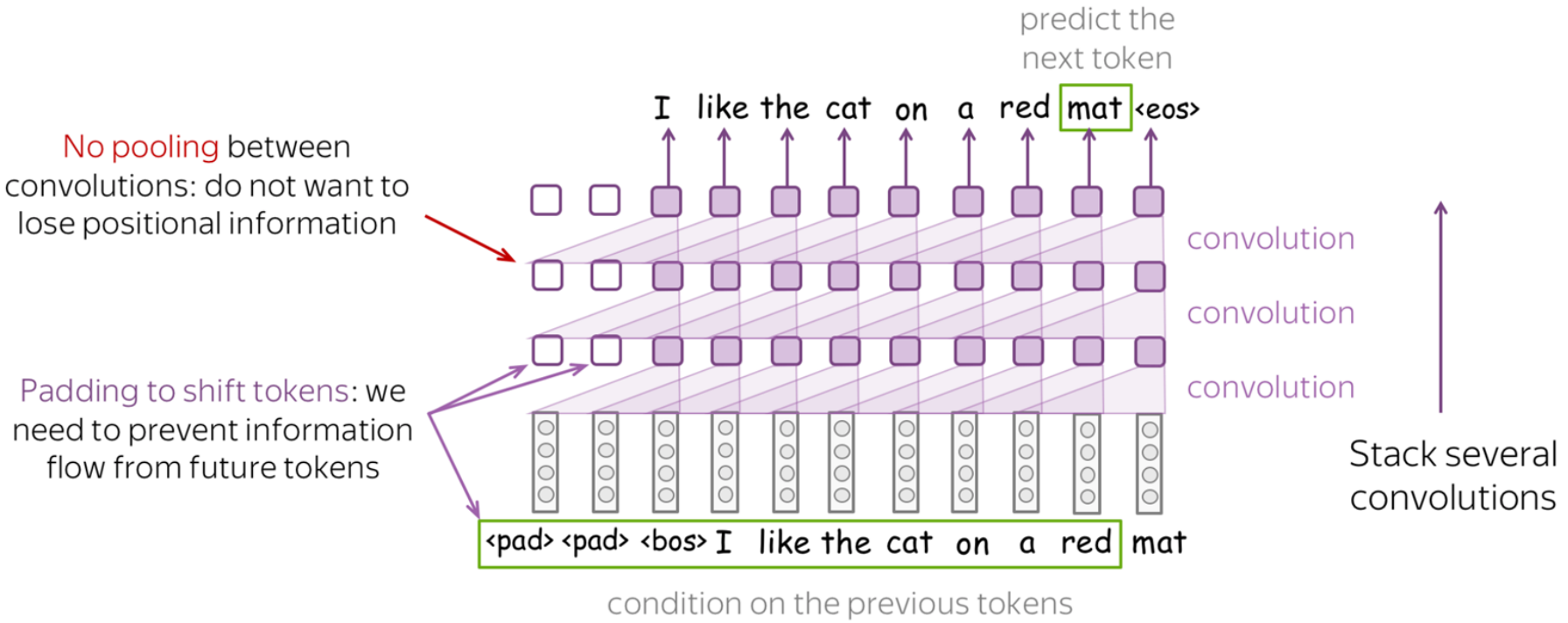
Neural models

- CNNs for text



Language modelling with CNNs

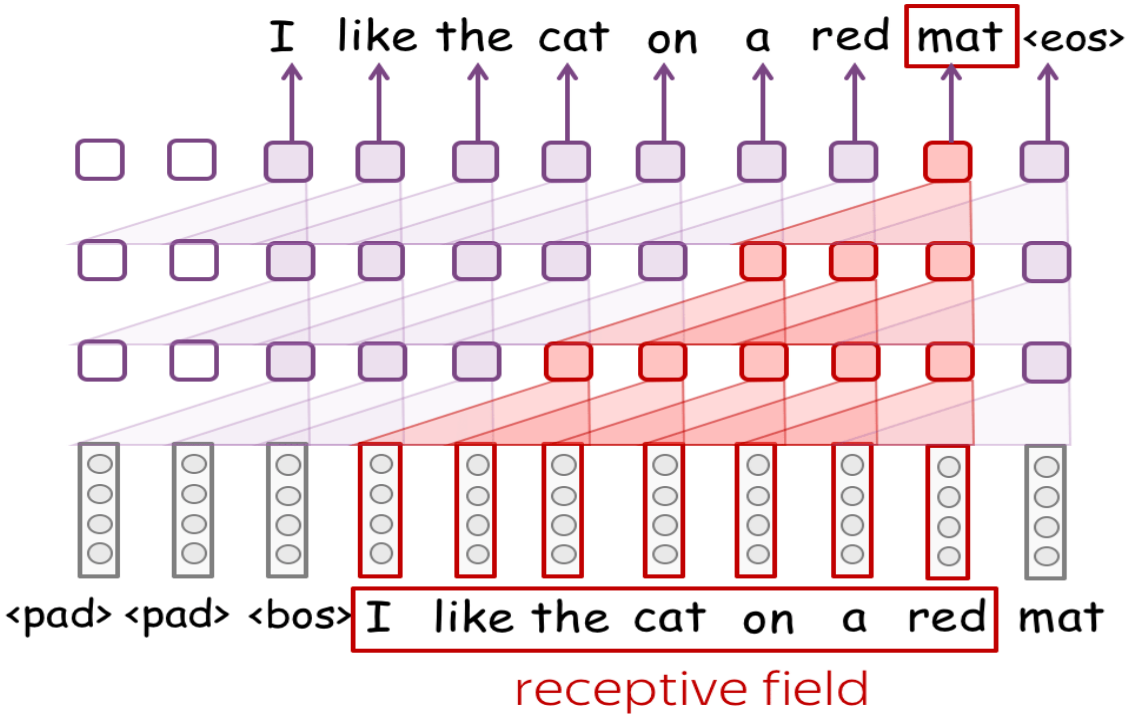
source: lena-voita.github.io/nlp_course.html



CNNs : receptive field

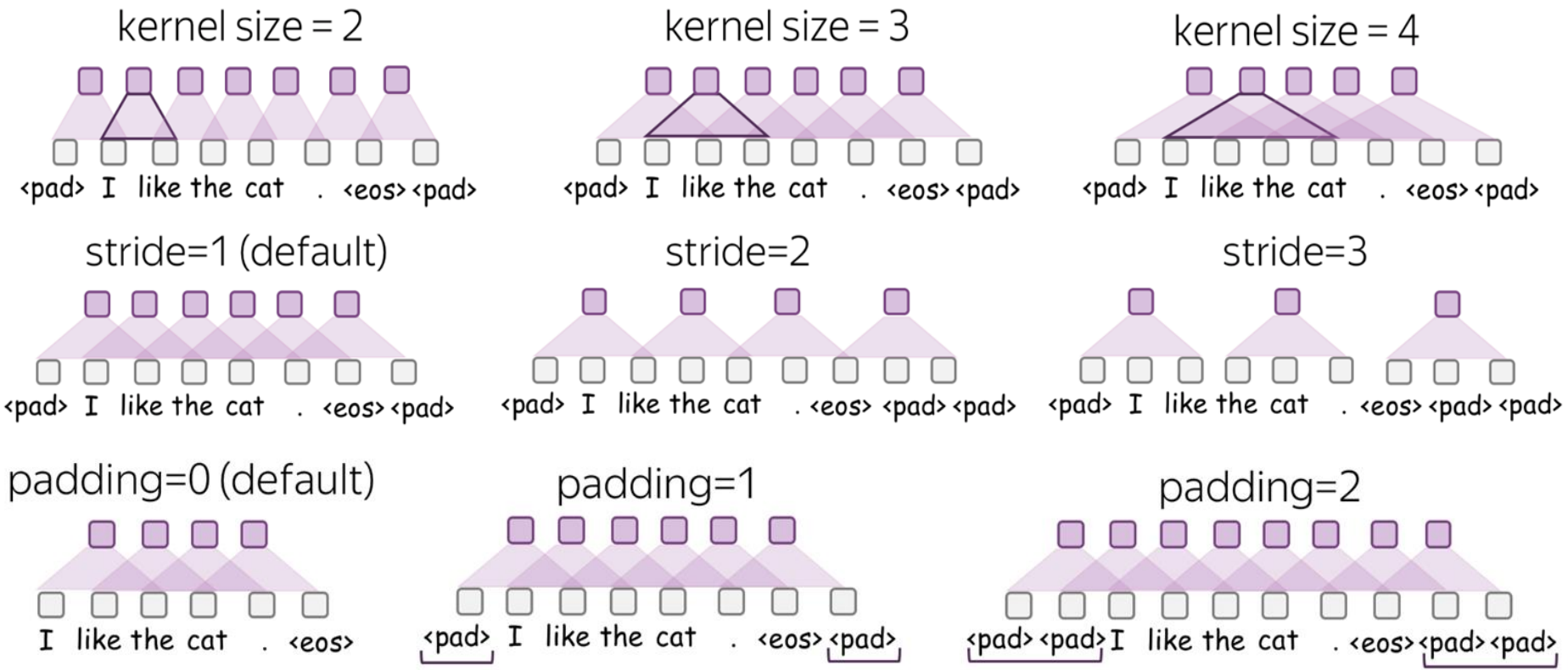
Close connection to
our Markov models

source: lena-voita.github.io/nlp_course.html



1D Convolutions on text

source: lena-voita.github.io/nlp_course.html

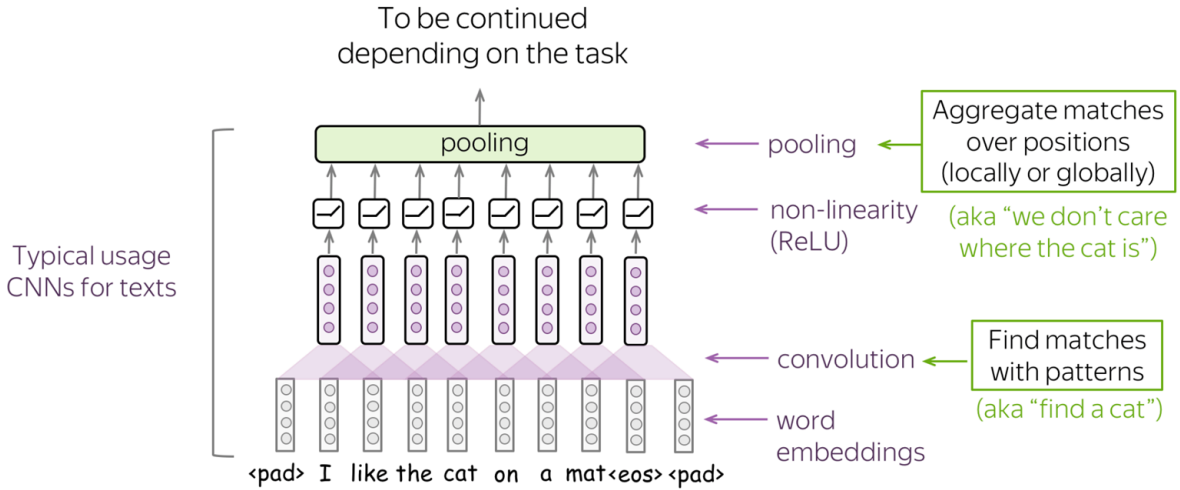


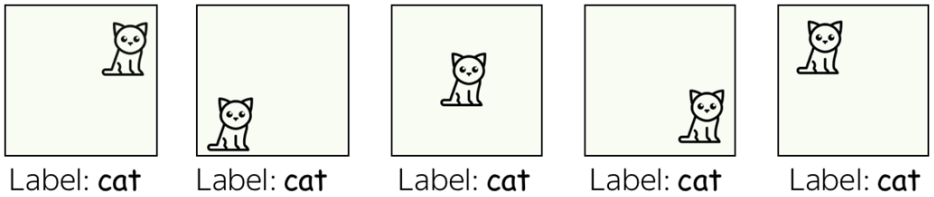
Classification with CNNs

source: lena-voita.github.io/nlp_course.html

Equivariance → invariance

- via pooling





We don't care where the cat is, we care that it is somewhere.

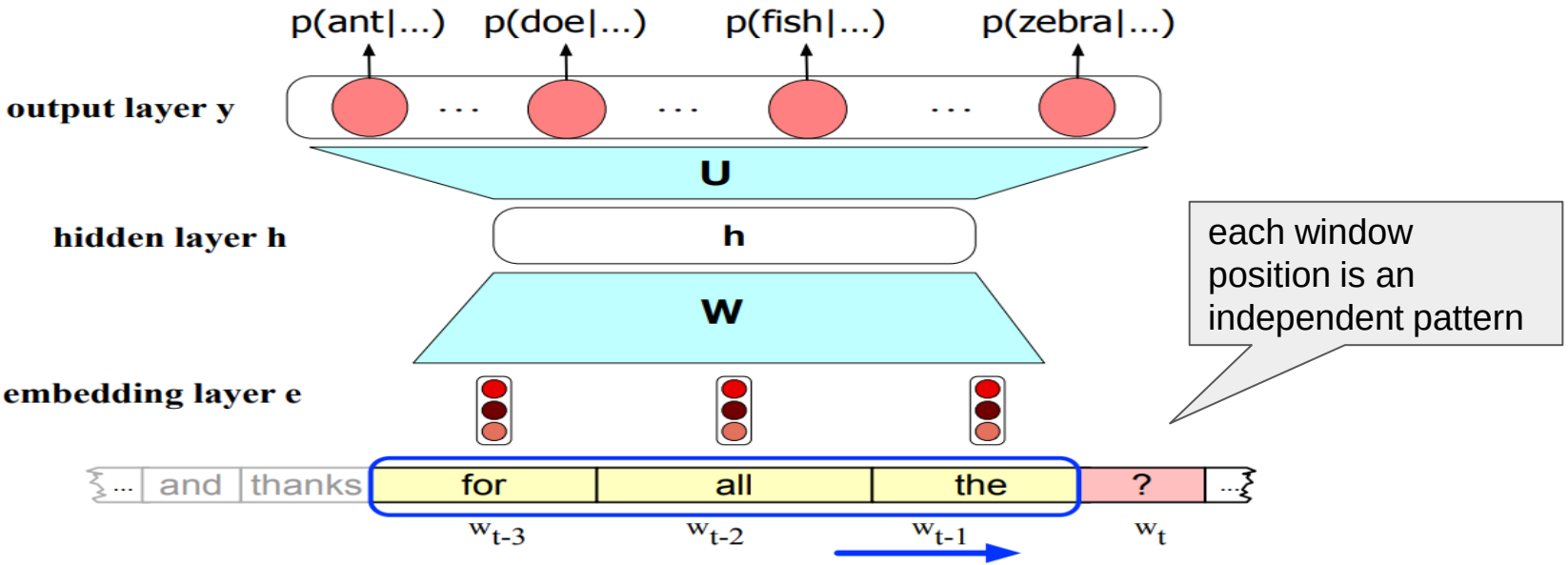
Then why don't we process all these cats similarly?

Neural models

- NLP with Sequential Models

Sequential models

- Language is inherently **sequential**
  - so far we either ignored this completely (BoW)
  - or restricted to a small-size history (Markov, sliding window)

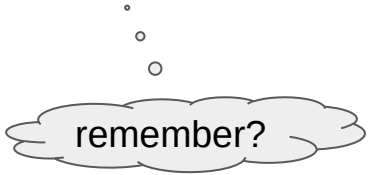


Sequential models

- Language is inherently sequential
  - so far we either ignored this completely (BoW)
  - or restricted to a small-size history (Markov, sliding window)

- Many linguistic phenomena require longer distance interactions
  - *“The **computer** which I had just put into the machine room on the fifth floor **crashed**.”*

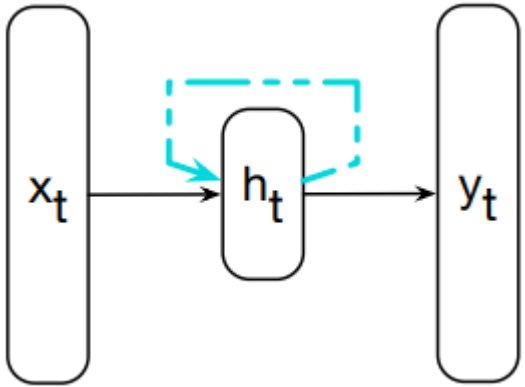
- Modern neural models address this:
  1. Recurrent Neural Networks
  2. Transformer Networks



Recurrent Neural Networks

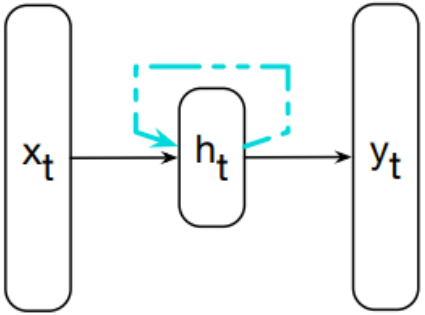
- Recurrent Neural Networks (RNN) = contain **cycles**
  - introduces dependency on earlier inputs/outputs
- The most simple RNN model: **Elman Networks**
- Compute with hidden value

  from the **preceding** item
- We process one item (word) at a time
  - no fixed-length limit on the prior context!
  - In contrast to the window-based approaches

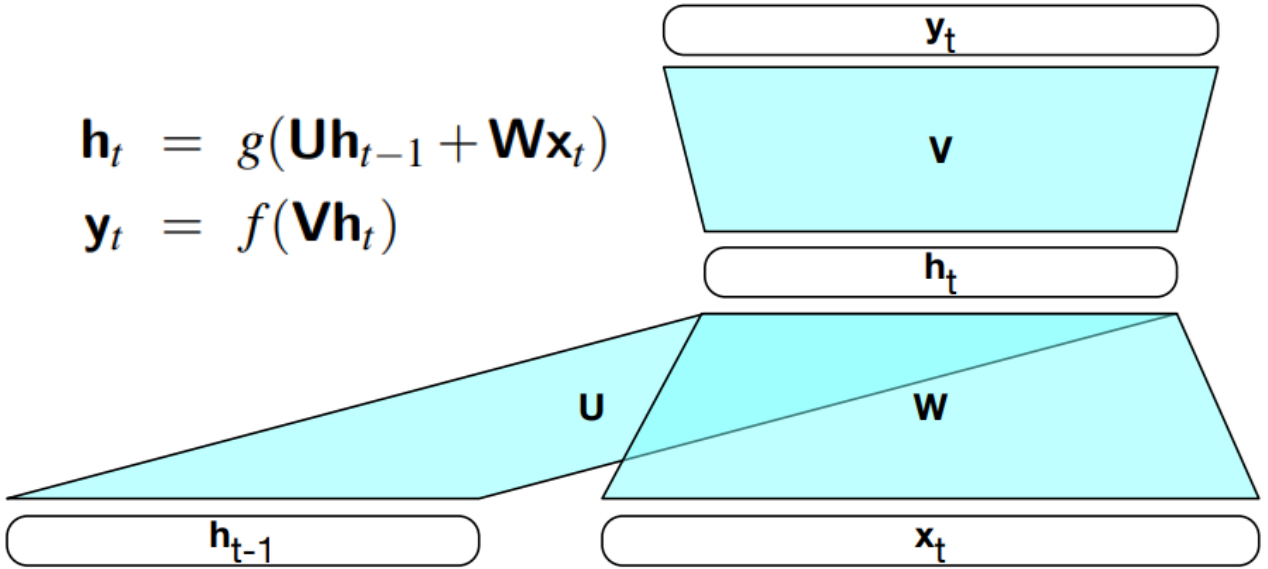


Recurrent Neural Networks

- Recurrent Neural Networks (RNN) = contain **cycles**
- These can be **unrolled** to make them feed-forward



$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t)$$
$$\mathbf{y}_t = f(\mathbf{V}\mathbf{h}_t)$$



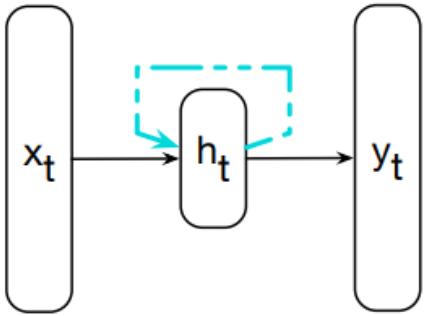
Recurrent Neural Networks

- Recurrent Neural Networks (RNN) = contain **cycles**
- These can be **unrolled** to make them feed-forward
  - at each (time) step i

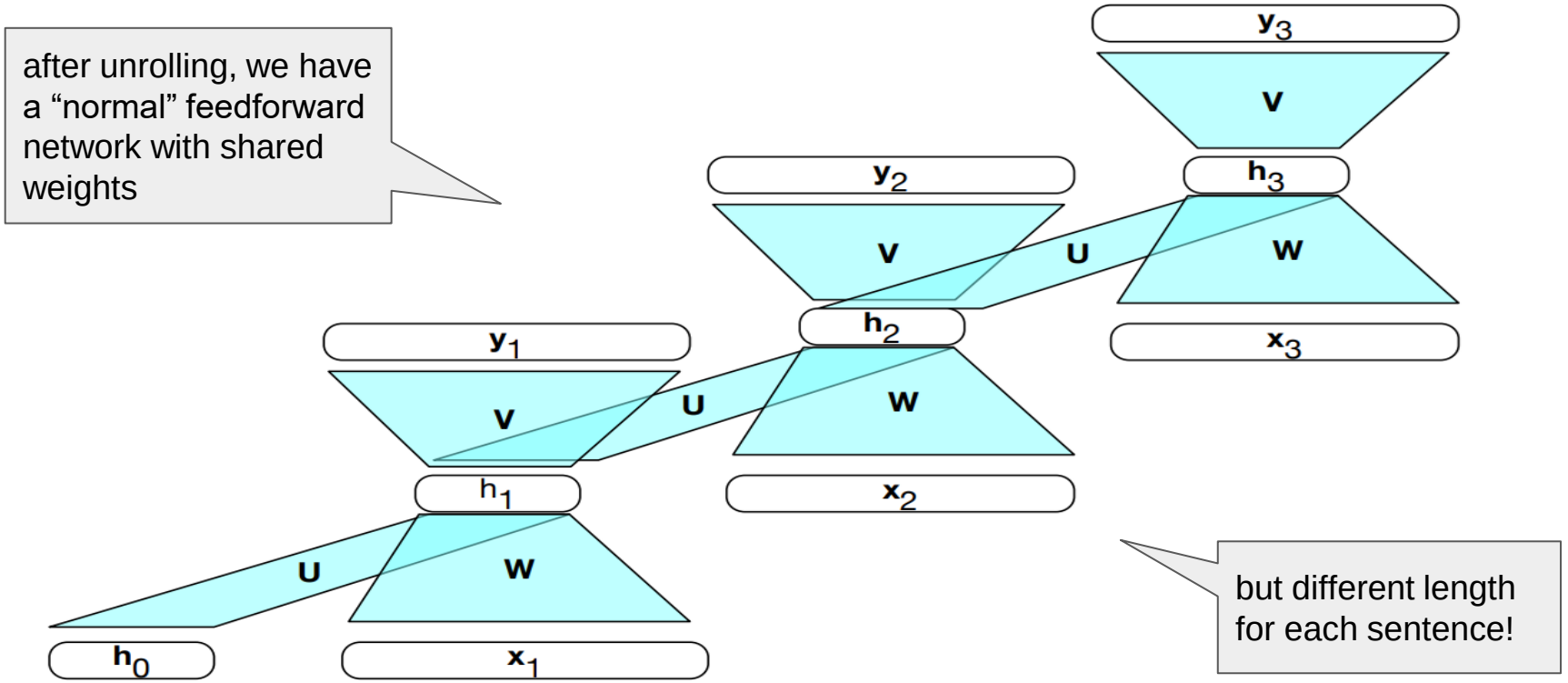
function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}^0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH($\mathbf{x}$) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

the weights are shared across all the timesteps!



RNN Training: forward unrolling



RNN Training

Training RNNs is also similar to what we have already seen:

1. Choose a loss function
2. Calculate gradients for all the parameters:
    - **W:** input-to-hidden layer
    - **U:** previous-to-current hidden layer
    - **V:** hidden-to-output layer
3. Train with gradient descent

Some differences in gradient computation:

- computing loss at time t requires h_{t-1}

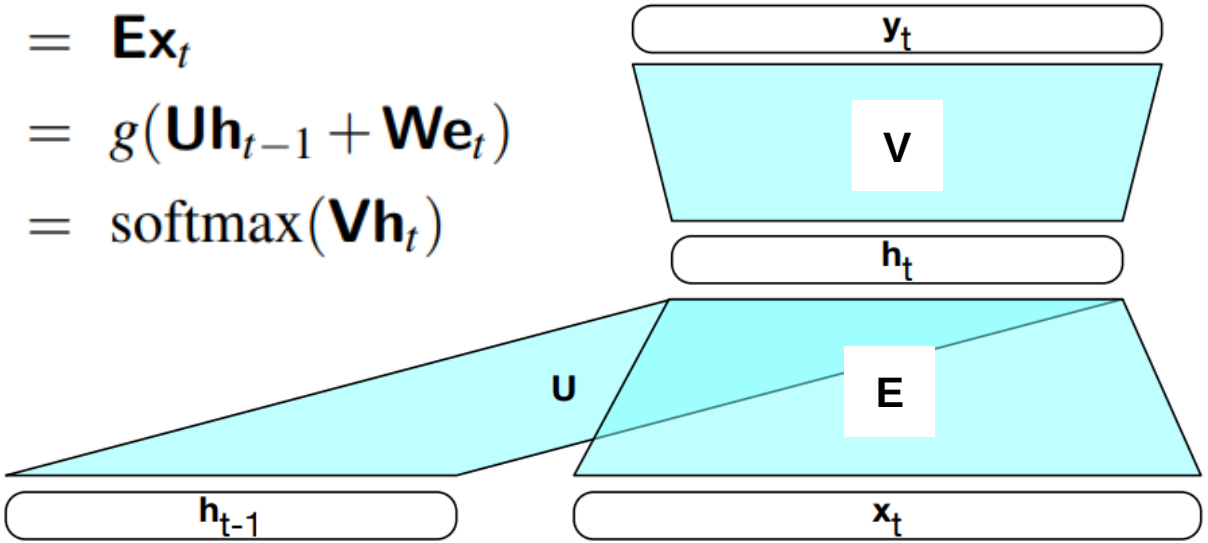
“Backpropagation Through Time”

RNNs as Language Models: weight tying

The input (E) and output (V) embedding matrices serve similar purpose

- just as in the word embedding models

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t$$
$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t)$$
$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$$



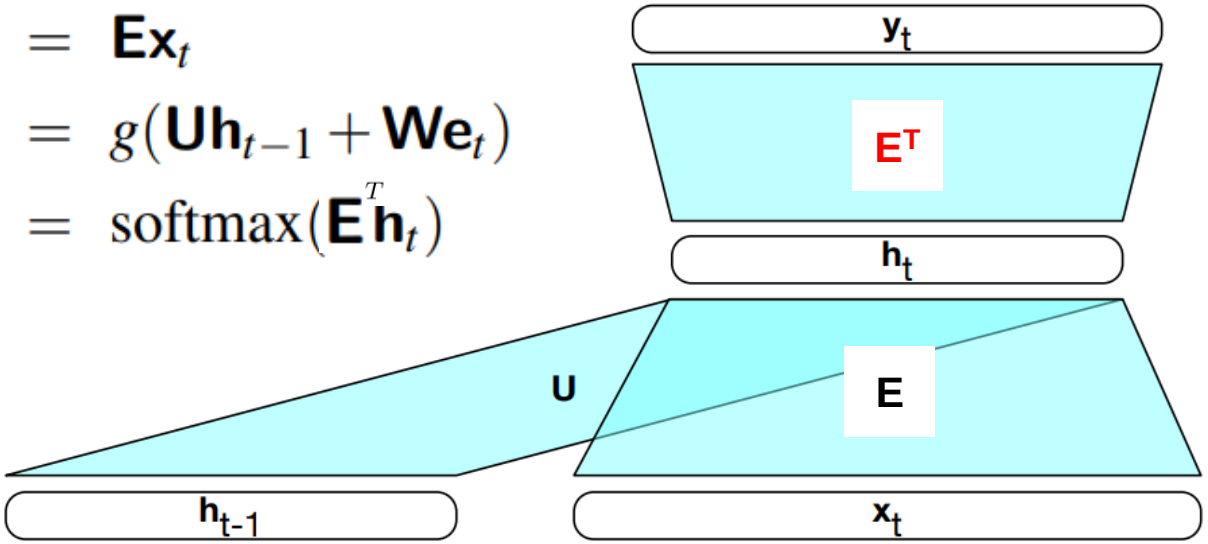
RNNs as Language Models: weight tying

The input (E) and output (V) embedding matrices serve similar purpose

- just as in the word embedding models

Idea: **weight tying**

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t$$
$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t)$$
$$\mathbf{y}_t = \text{softmax}(\mathbf{E}^T\mathbf{h}_t)$$



RNNs as Language Models

RNN Language model

RNN language models (T. Mikolov, again)

- work analogically to the window-based LMs
- but don't have the limited context problem

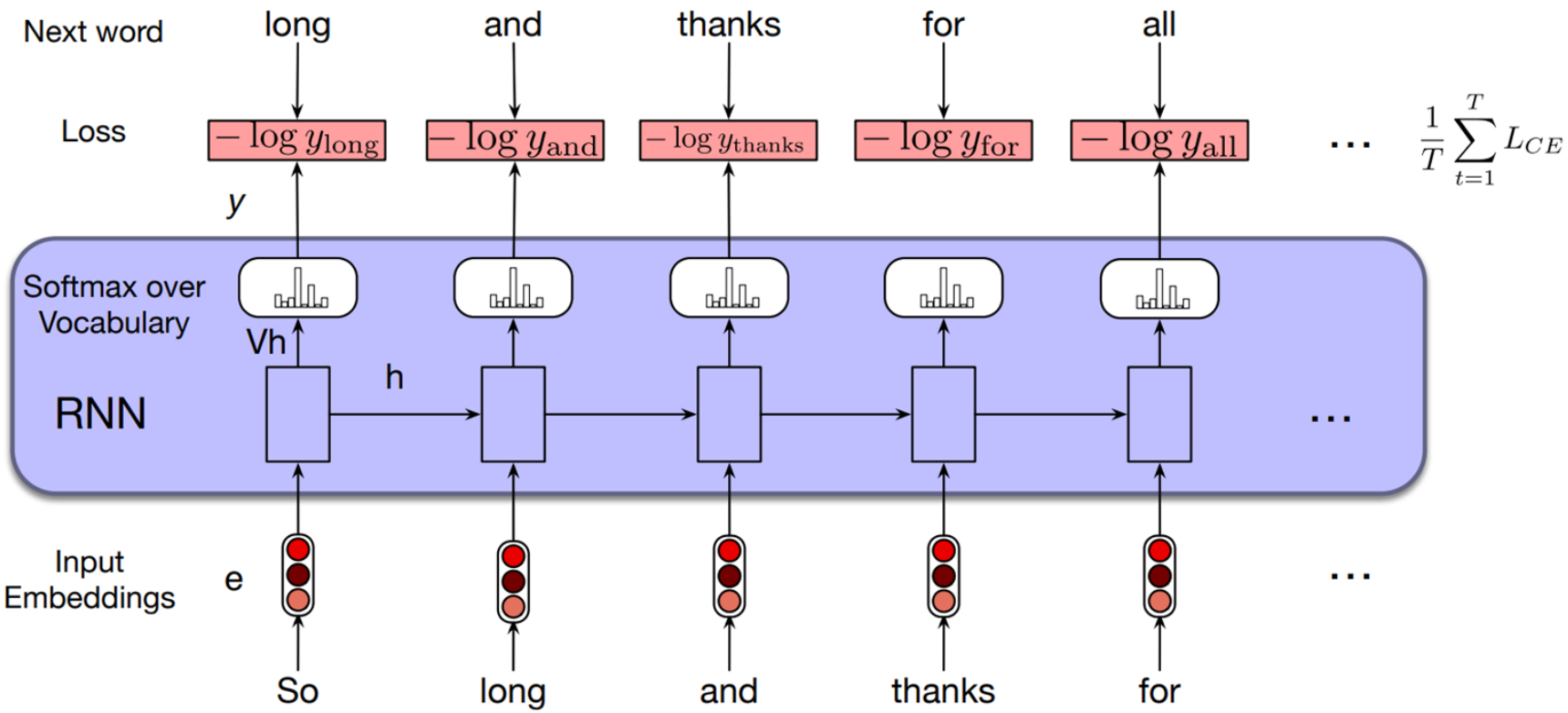
$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t$$
$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t)$$
$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$$

Goal: predict $P(w_{t+1} = i | w_1, \ldots, w_t) = \mathbf{y}_t[i]$

Train with cross-entropy $L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w]$

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}]$$

RNNs as Language Models

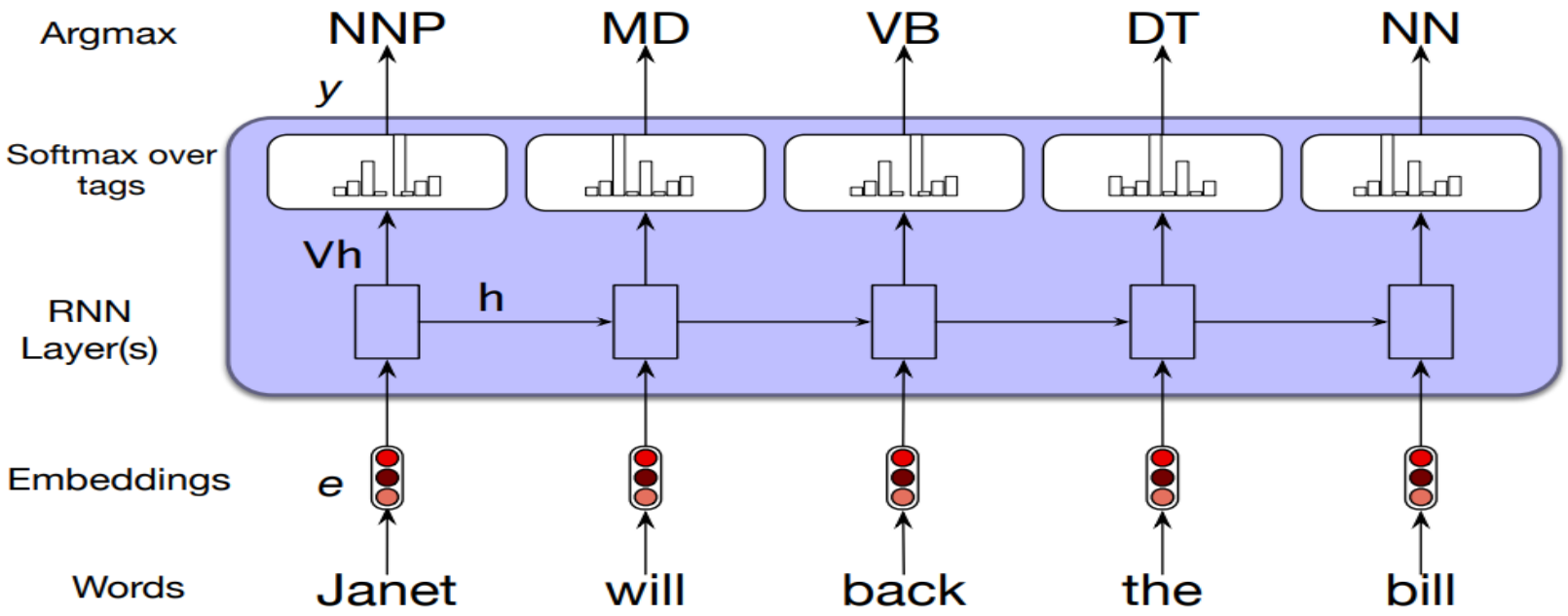


Neural models

- RNN applications

Sequence Labeling

Assign a label to **each element** of a sequence (e.g., part-of-speech tagging)



Sequence Classification

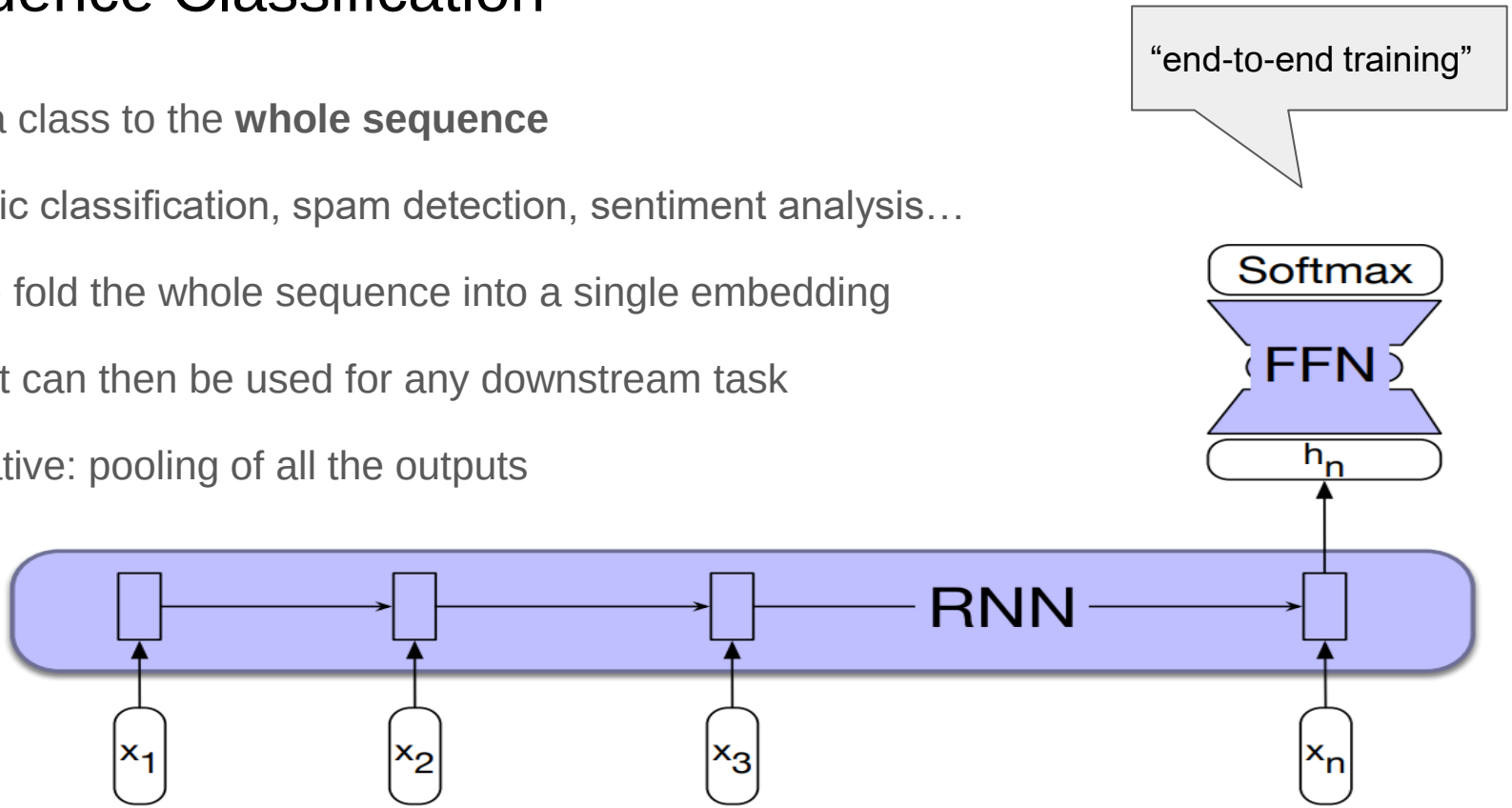
Assign a class to the **whole sequence**

- topic classification, spam detection, sentiment analysis…

Here we fold the whole sequence into a single embedding

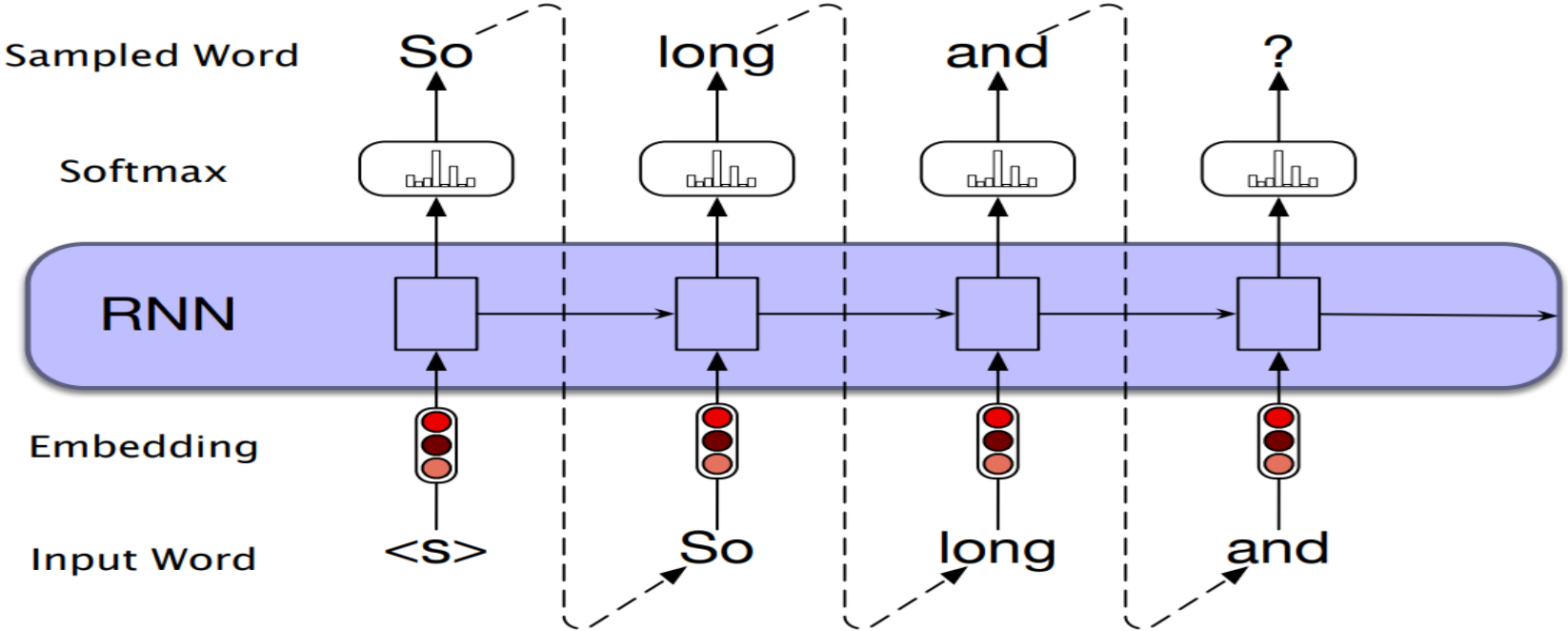
- that can then be used for any downstream task

Alternative: pooling of all the outputs



Text Generation

Just like with n-grams. Here called **autoregressive generation** (non-linear)



Text Generation

Just like with n-grams. Here called **autoregressive generation** (non-linear)

This simple idea actually forms basis for sophisticated tasks!

- machine translation, text summarization, question answering

There we prime the generation with an appropriate context

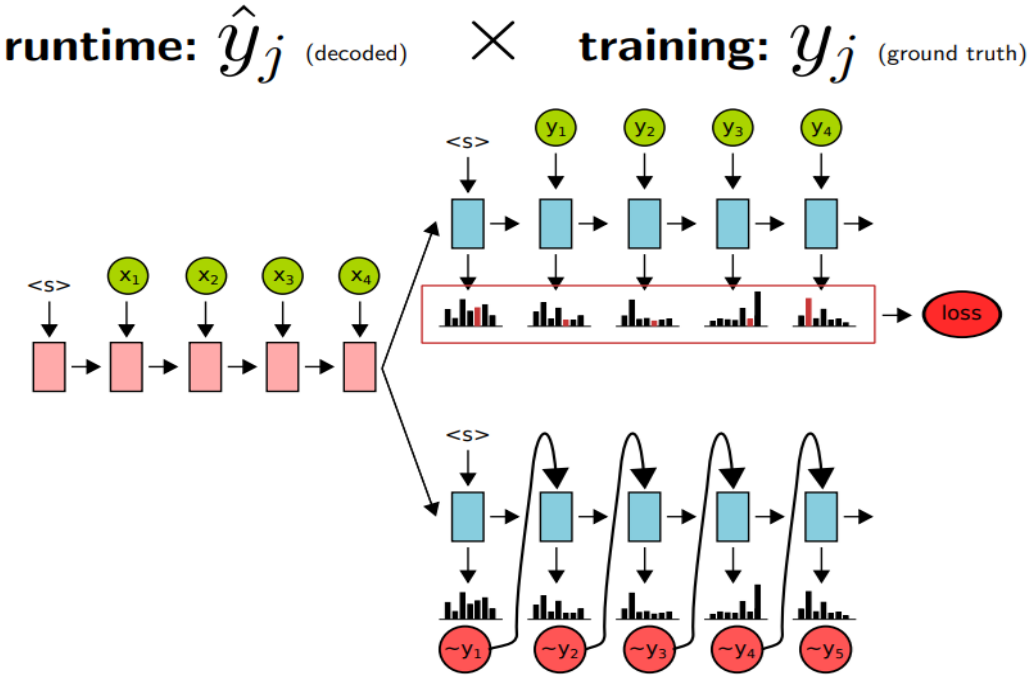
- instead of using just dummy <s>
- e.g., the original text (for translation or summarization)

RNNs as Language Models

source: https://ufal.mff.cuni.cz/jindrich-libovicky

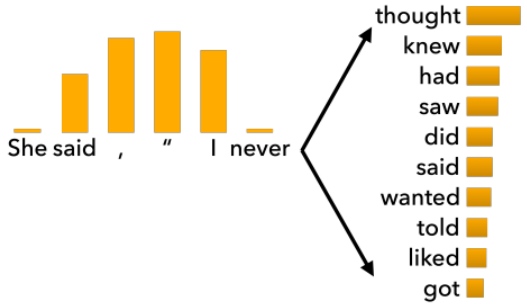
Train with **teacher forcing**:

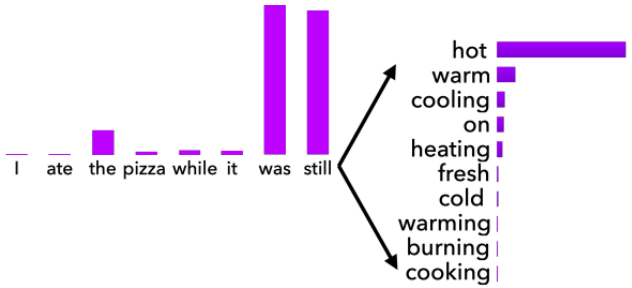
In training, at each step t
we feed the model
the **correct** sequence $w_{1:t-1}$



Text Generation: Practice

- Greedy Decoding
  - At each time step, select **the most probable** word
- Random Sampling
  - Just **sample** randomly
- Top-k Sampling
  - Sample, but only from the **top-k** most likely tokens
- Nucleus sampling
  - Sample from **top-p** probability mass tokens



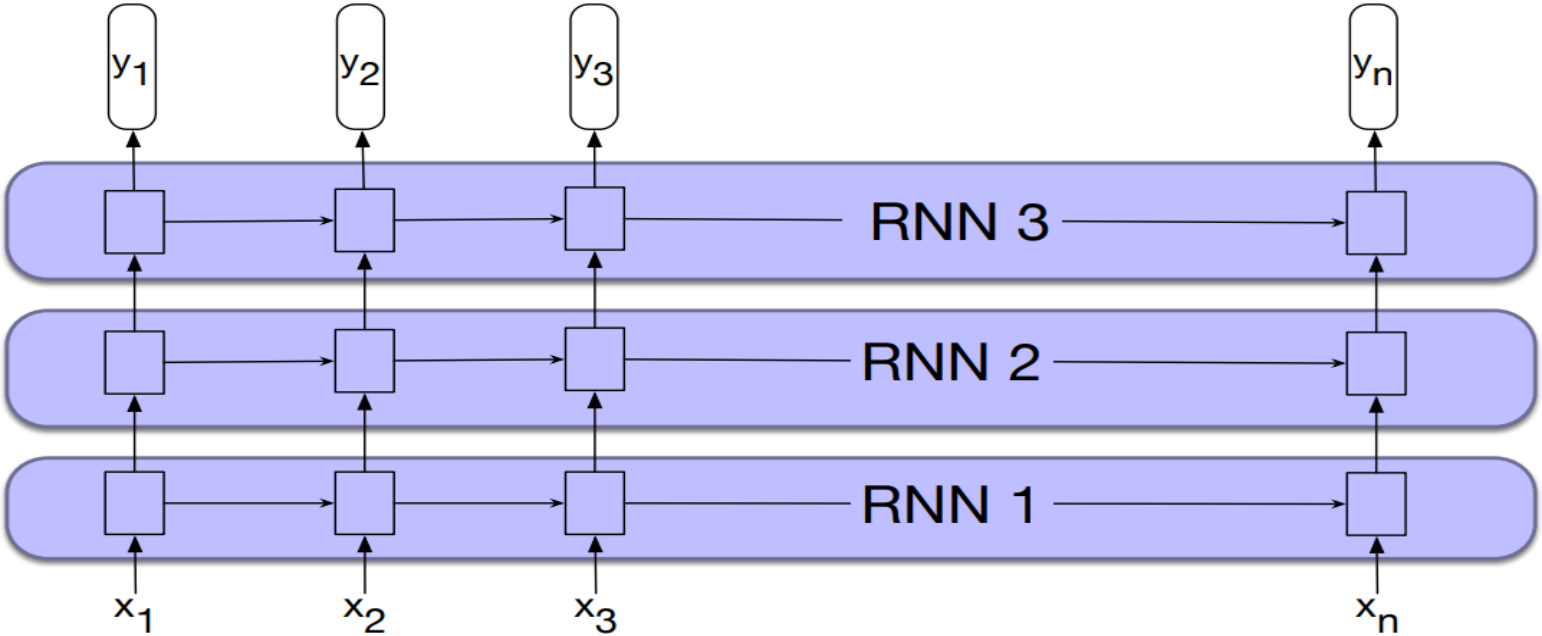


Neural models

- RNN extensions

Stacked RNNs

= hidden layer output can be fed as input to another hidden layer (deep RNNs)



Bi-directional RNNs

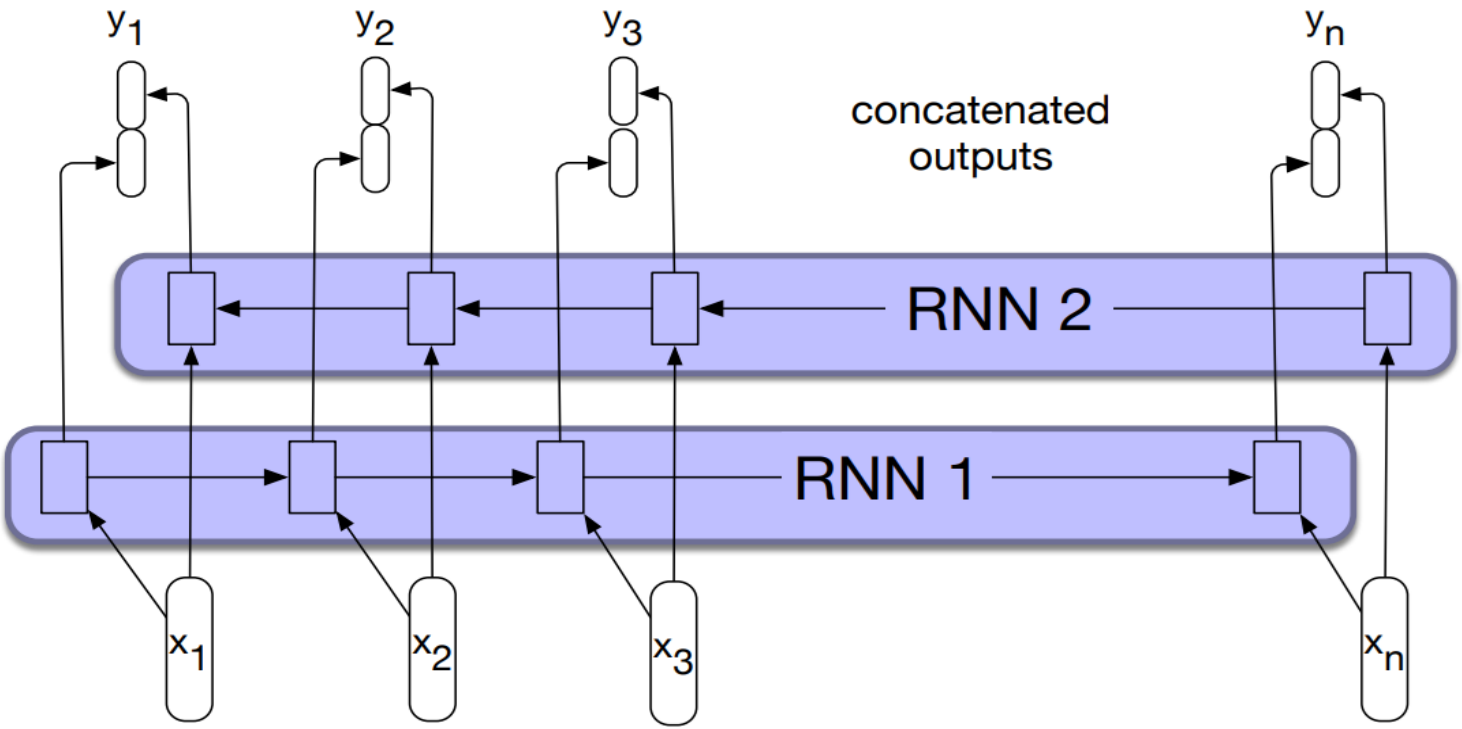
We commonly have the entire sequence for training

- no need to use just the “left” context $\mathbf{h}_t^f = RNN_{forward}(\mathbf{x}_1, \ldots, \mathbf{x}_t)$
- we can also utilize the “right” context $\mathbf{h}_t^b = RNN_{backward}(\mathbf{x}_t, \ldots \mathbf{x}_n)$

And combine these 2 representations

- into a bi-directional one: $\mathbf{h}_t = [\mathbf{h}_t^f ; \mathbf{h}_t^b]$
    - via **concatenation**, element-wise addition, multiplication…

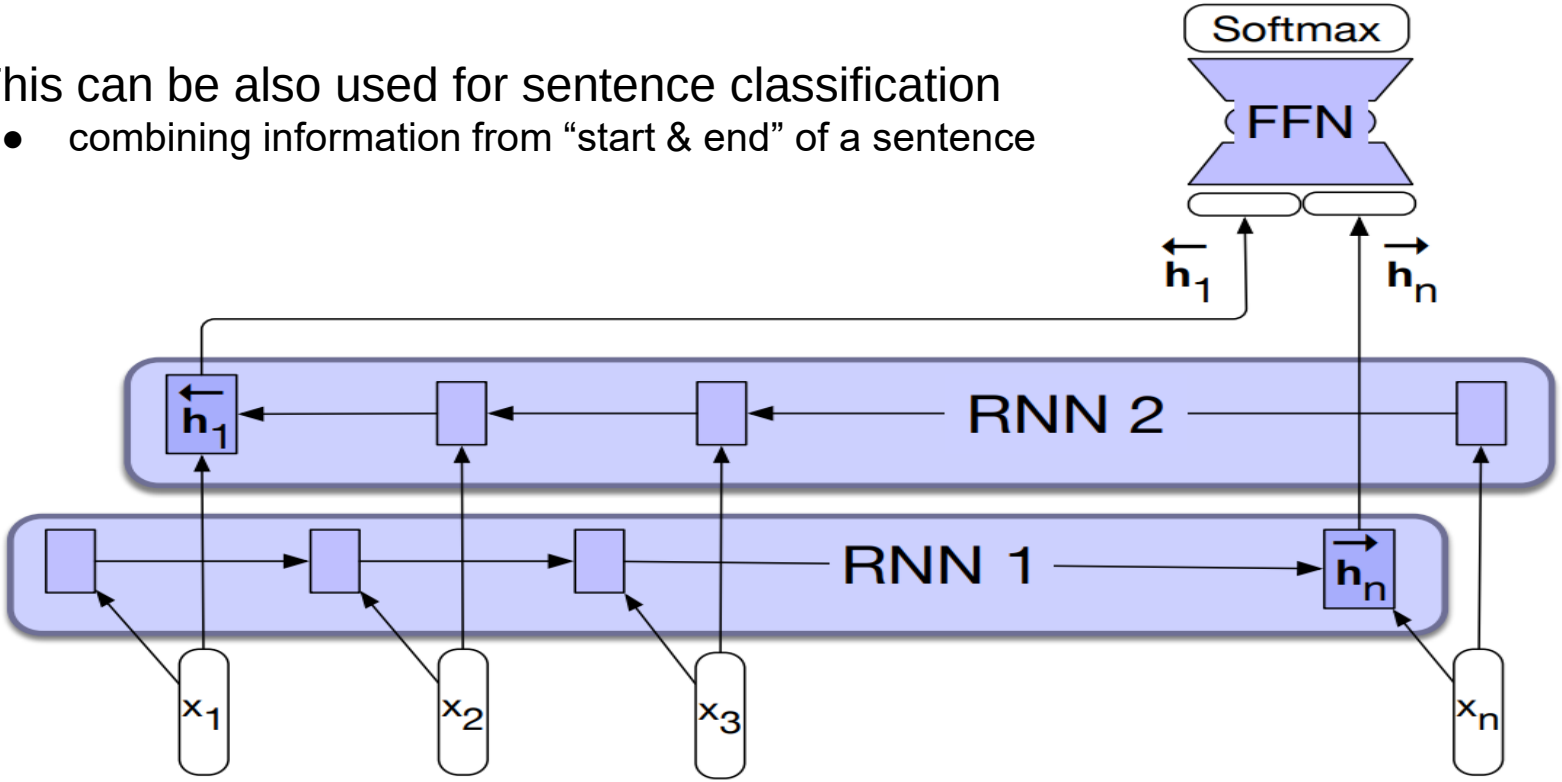
Bi-directional RNNs for sequence labeling



Bi-directional RNNs for sentence classification

This can be also used for sentence classification

- combining information from “start & end” of a sentence



Neural models

- Long-Short Term Memory (LSTM)

Long dependencies in text

Distant information is critical to many language applications:

- *“The **computer** which I had just put into the machine room on the fifth floor **crashed**.”*

RNN can theoretically process unlimited context, but:

- need to reflect current & future information at the same time
- long sentences (deep unrolled NNs) lead to **vanishing gradients**

Consequently, the information contained in the hidden states tend to be fairly local

Long short-term memory (LSTM) network

Divide the latent context management into 2 sub-problems

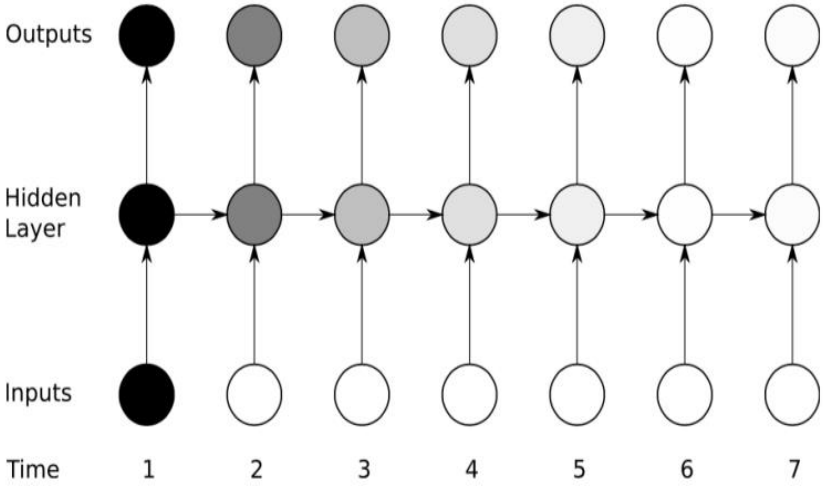
- removing information when no longer needed
- adding information likely to be needed later

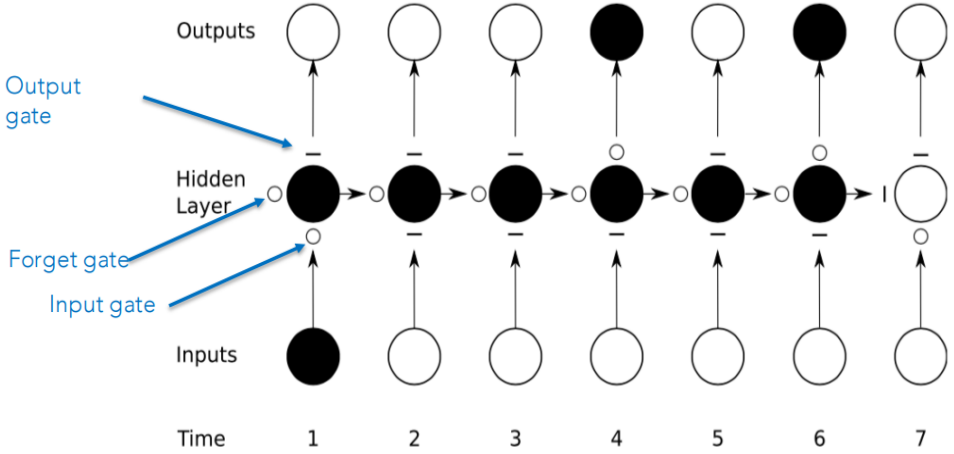
Approached by:

1. adding an explicit **context** representation layer
    a. in addition to the common hidden layer representation
2. introduce new units: neural **gates**
    a. to control the flow of information

source: courses.cs.washington.edu/courses/cse517/

Gating Intuition





Long short-term memory (LSTM) network

Gate units share common design:

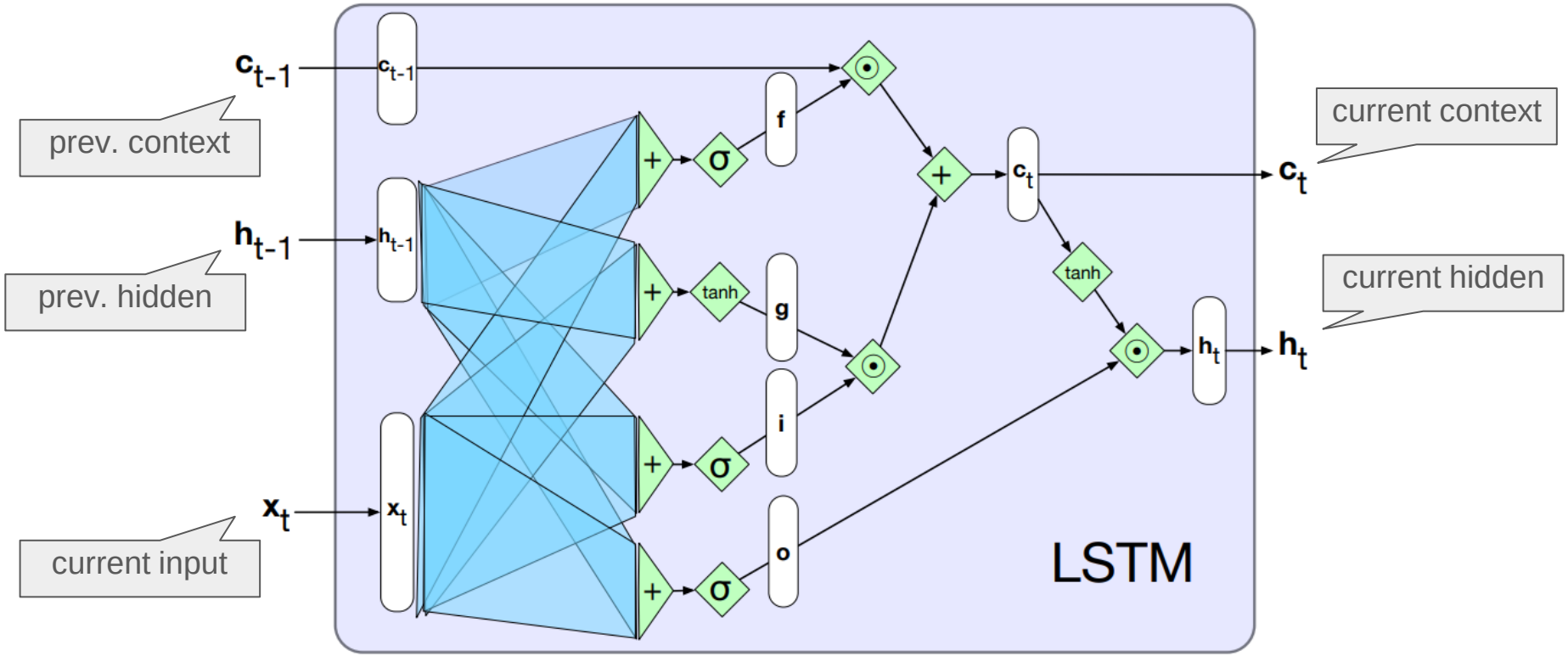
1. feed-forward layer
2. followed by sigmoid fcn
3. follower by pointwise multiplication with the gated layer

The sigmoid serves as a soft “binary” **mask**

- values aligned with ~1 pass through
- values aligned with ~0 are deleted

Each LSTM cell (“neuron”) contains 3 of these gates

Long short-term memory (LSTM) network



Long short-term memory (LSTM) network

forget gate - purpose: **delete** information from the **context**

- mask calculation: $\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t)$
- mask gating: $\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$

element-wise (Hadamard) product

+ standard RNN hidden unit: $\mathbf{g}_t = \tanh(\mathbf{U}_g \mathbf{h}_{t-1} + \mathbf{W}_g \mathbf{x}_t)$

input gate - purpose: **add** information to the current **context**

- mask calculation: $\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t)$
- mask gating: $\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t$

Long short-term memory (LSTM) network cont'd

Next, we add this to the ("masked") context vector $\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$

- to obtain new context at time t

add-gate masking of input+hidden

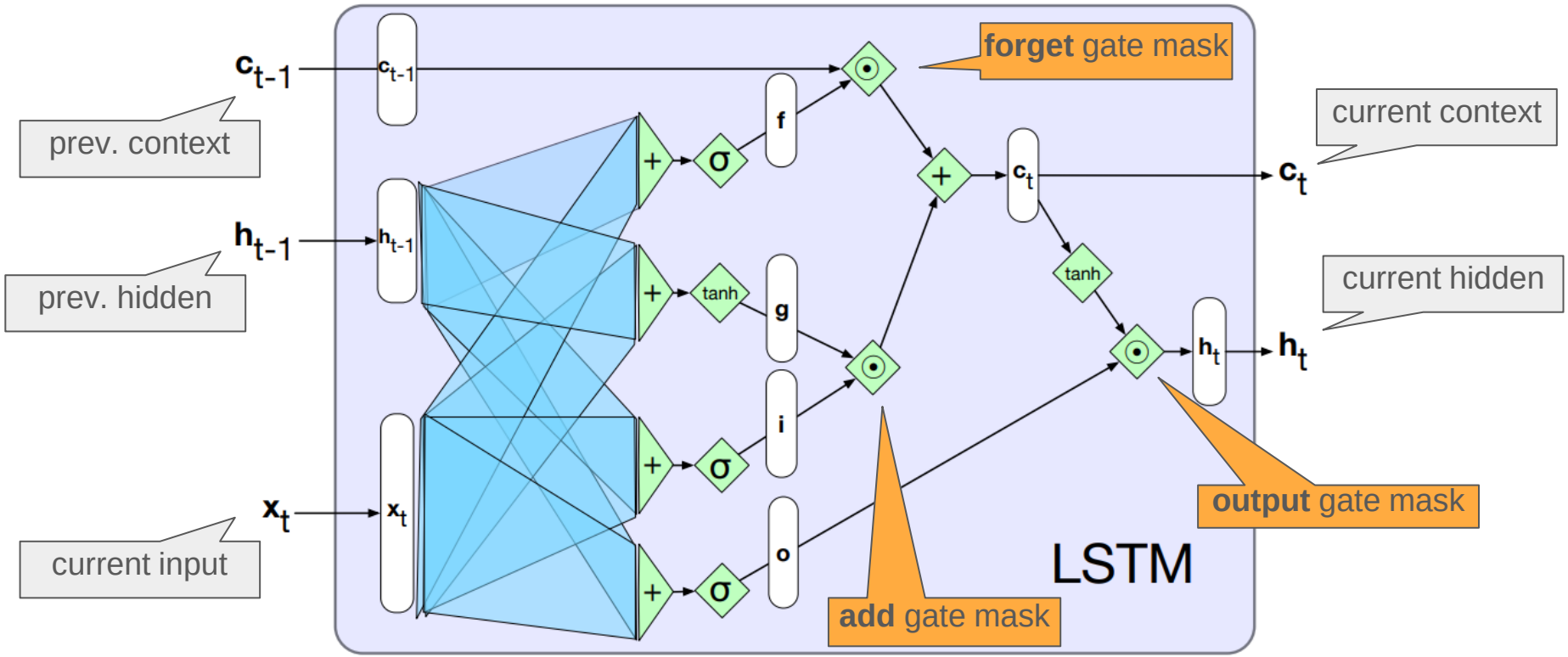
forget-gate masking of prev. context

Finally, there's an **output gate**

- decides solely what to output for the **current** hidden state

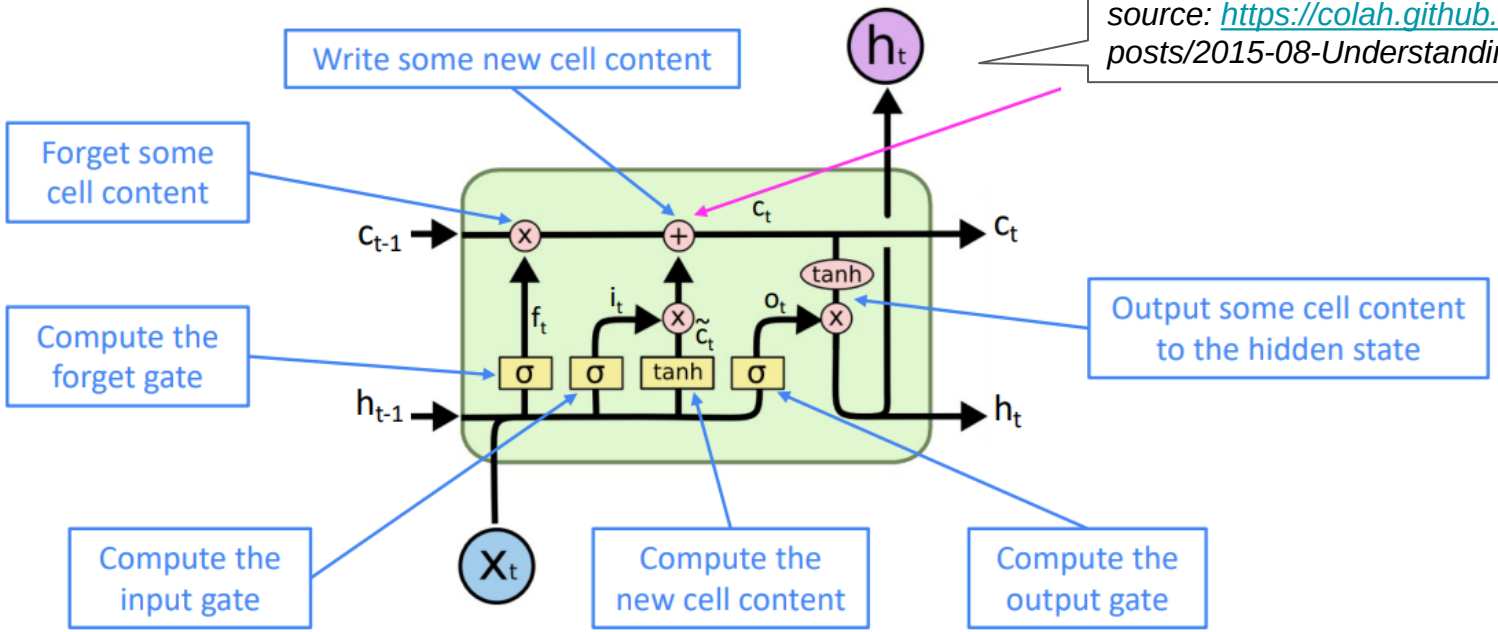
$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

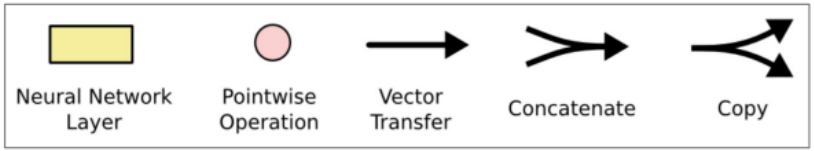
Long short-term memory (LSTM) network

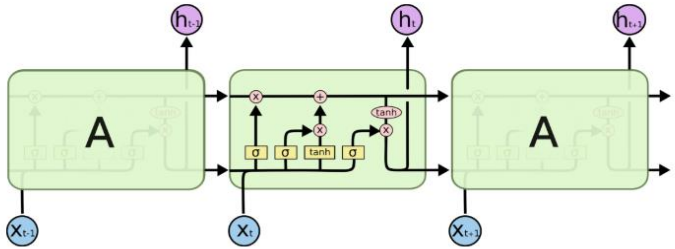


You can think of the LSTM equations visually like this:

source: https://colah.github.io/posts/2015-08-Understanding-LSTMs



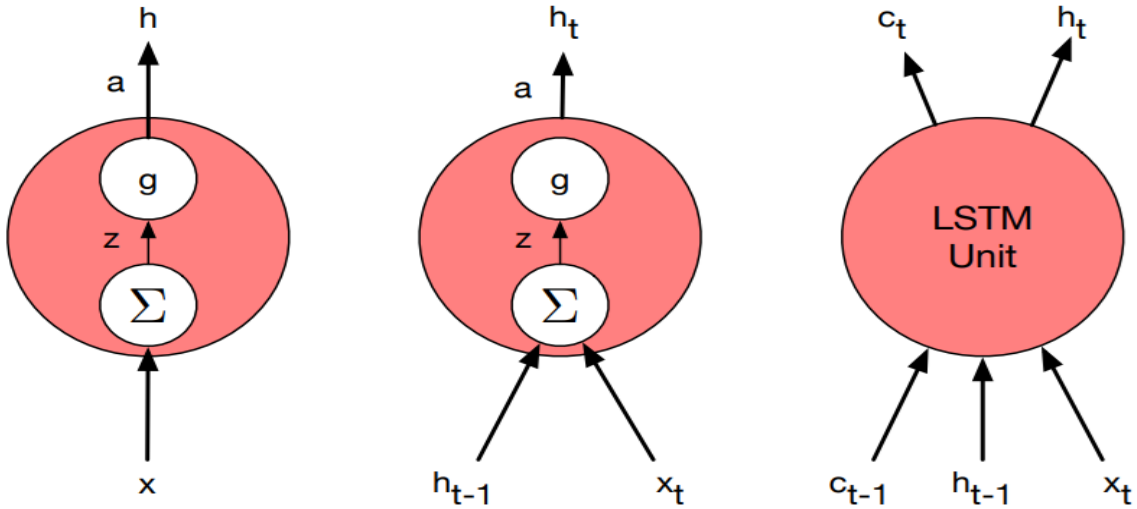




RNNs interface

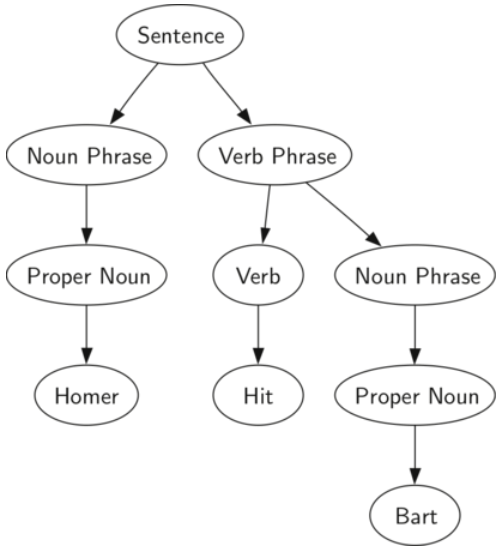
In practice, you are nicely interfaced from all this complexity

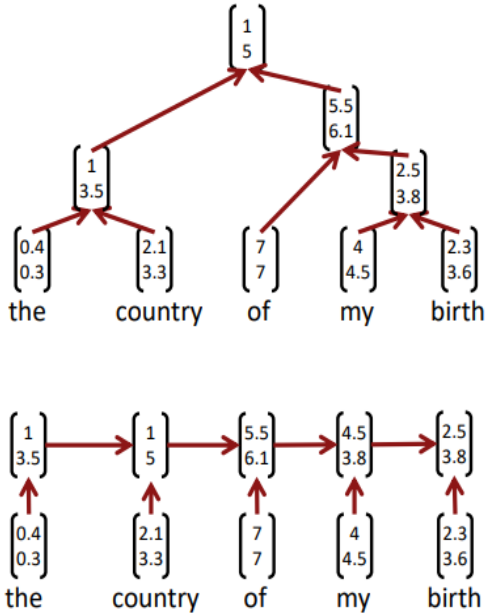
- and can just plug LSTM everywhere we have seen RNN so far…

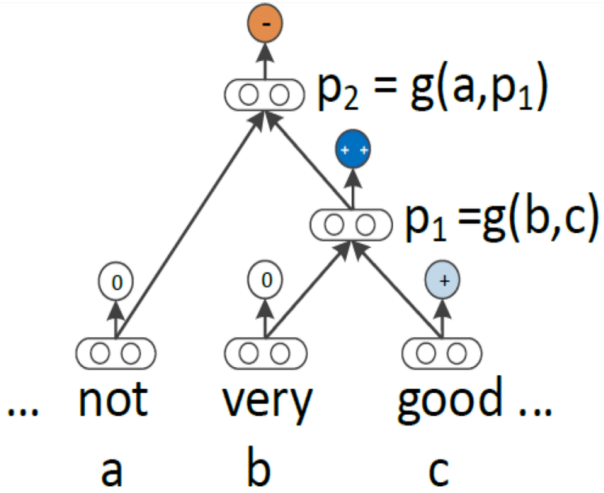


Recursive NNs (TreeNNs)

Operate over regular trees instead of sequences







Neural models

- Attention and Transformers

Transformers

…LSTM are great, but the sequential processing can be problematic in practice

Next idea: **Transformers**

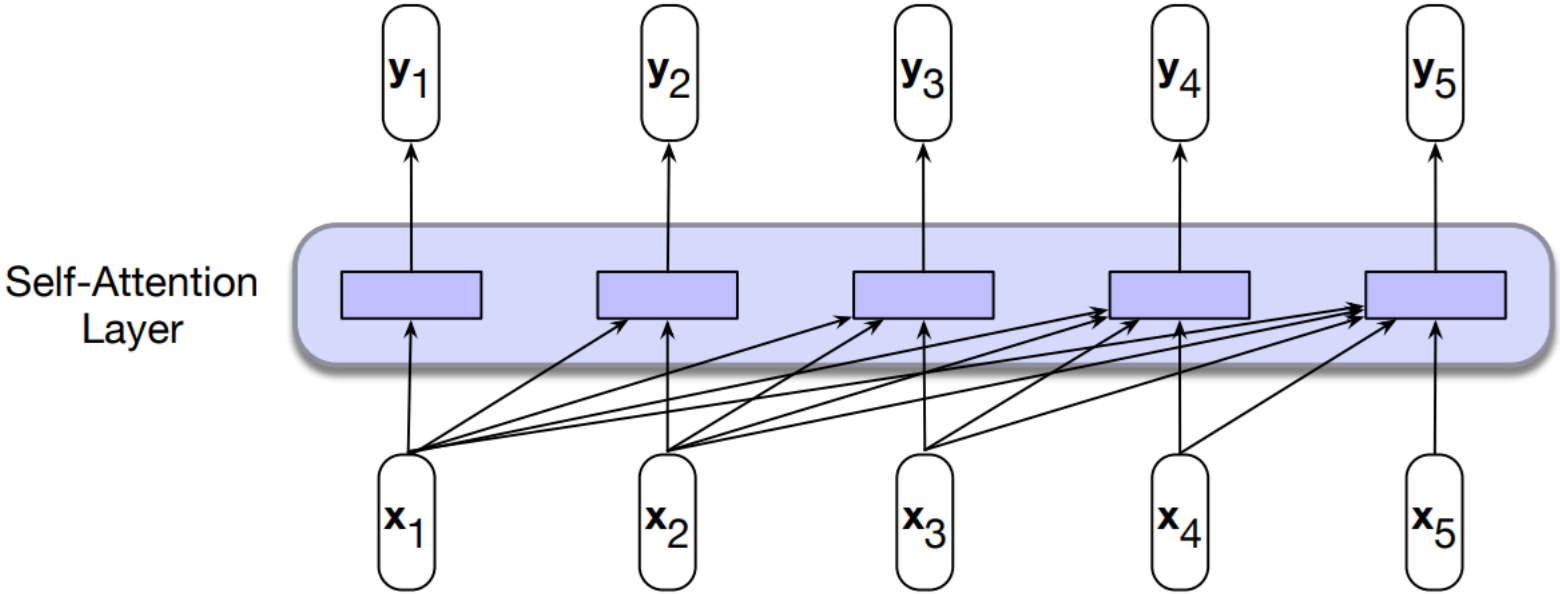
- maps sequences $(x_1,...,x_n)$ directly to sequences $(y_1,...,y_n)$
- no recurrent connections (similar to feed-forward NNs)
- the sequence is processed at once in parallel

Comprised of transformer blocks made of

- simple linear layers
- feedforward networks
- **self-attention** layers

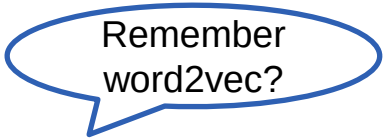
Causal (masked) Self-Attention

- The units are connected to reflect the sequentiality (causality)
- But can be processed in parallel, as there is no intermediate state



Self-Attention

Core: we **match** an item to a collection of related items



- revealing (pair-wise) relevance in the current context

How to score relevance? **Dot-product** (again)

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$$

- we then take softmax over all the comparisons

$$\begin{aligned} \alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \leq i \\ &= \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))} \;\; \forall j \leq i \end{aligned}$$

And output a weighted sum

- weighted by the relevance scores

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j$$

Query, Key, Value

The input embeddings $\mathbf{x}_i$ take on different roles here:

as a **Query**: $\mathbf{q}_i$ the current focus of attention

- will be projected as : $\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i$

as a **Key**: $\mathbf{k}_i$ the preceding input used for matching

- will be projected as : $\mathbf{k}_i = \mathbf{W}^K \mathbf{x}_i$

as a **Value**: $\mathbf{v}_i$ used for computing the output

- will be projected as : $\mathbf{v}_i = \mathbf{W}^V \mathbf{x}_i$

a notation adopted from information retrieval (or memory networks)

$$\mathbf{W} \in \mathbf{R}^{d \times d}$$

Query, Key, Value

Given these roles/projections:

- for the relevance score we use the query and key vectors

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j$$

- for the output calculation we use the value vectors

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j$$

- additionally, we normalize the dot-products
  - by the dimensionality of the vectors

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

Example: computing the 3rd position

The input embeddings' $\mathbf{x}_i$ roles:

Query: $\mathbf{q}_i$ - the current focus

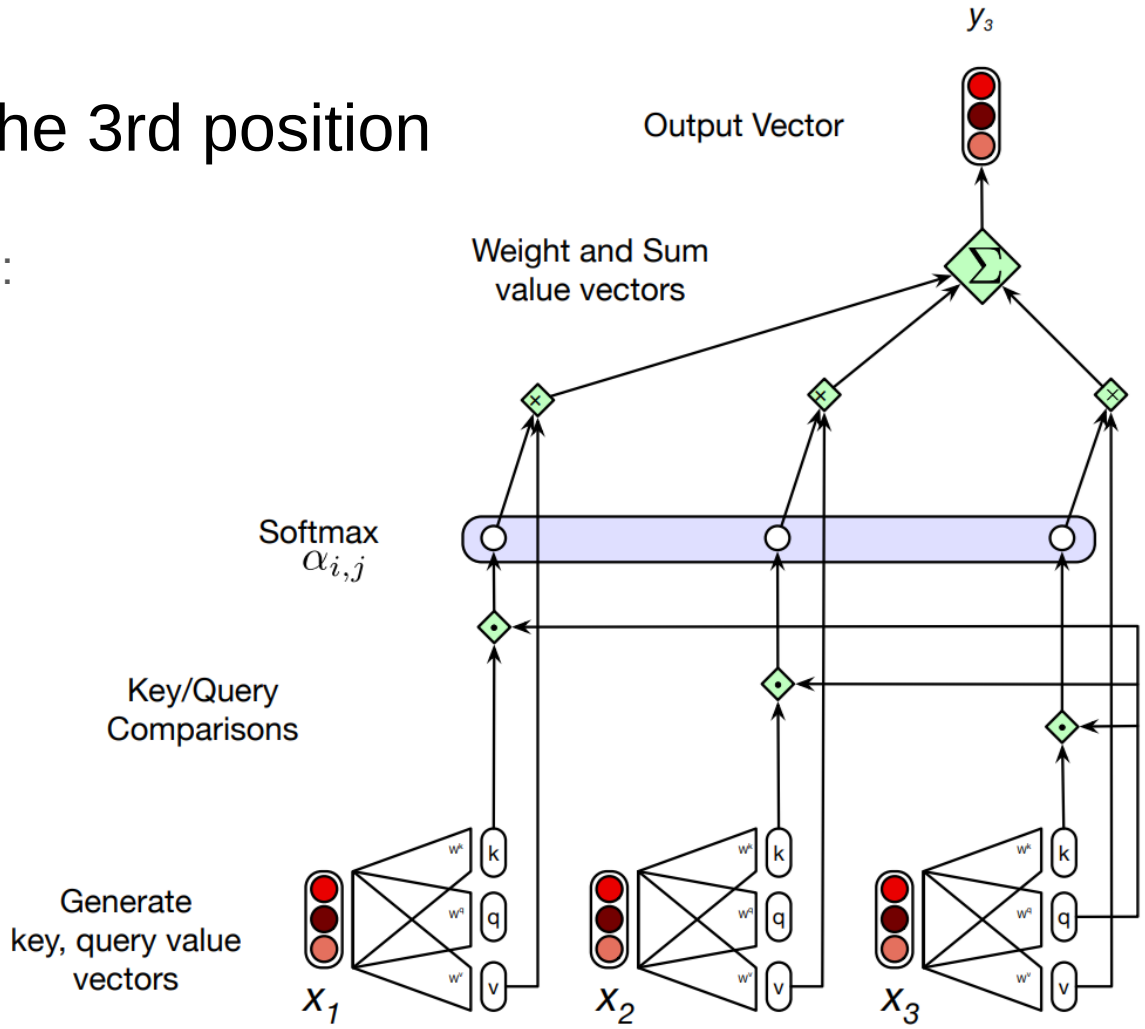
- $\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i$

Key: $\mathbf{k}_i$ - the preceding input

- $\mathbf{k}_i = \mathbf{W}^K \mathbf{x}_i$

Value: $\mathbf{v}_i$ - the output role

- $\mathbf{v}_i = \mathbf{W}^V \mathbf{x}_i$



Self-Attention: Query, Key, Value

Since the calculations are independent for each position, we can vectorize this

- all input embeddings $x_1..x_N$ form an input matrix **X**
- and the Query, Key, Value projections: $\mathbf{Q = XW^Q; K = XW^K; V = XW^V}$
- all the query-key comparisons come from $\mathbf{QK^T}$ (+softmax)
- and finally multiply the result by **V**

$$SelfAttention(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

- we have to limit these to the **preceding** pairs only

Note that attention is **quadratic** in the input sequence size N

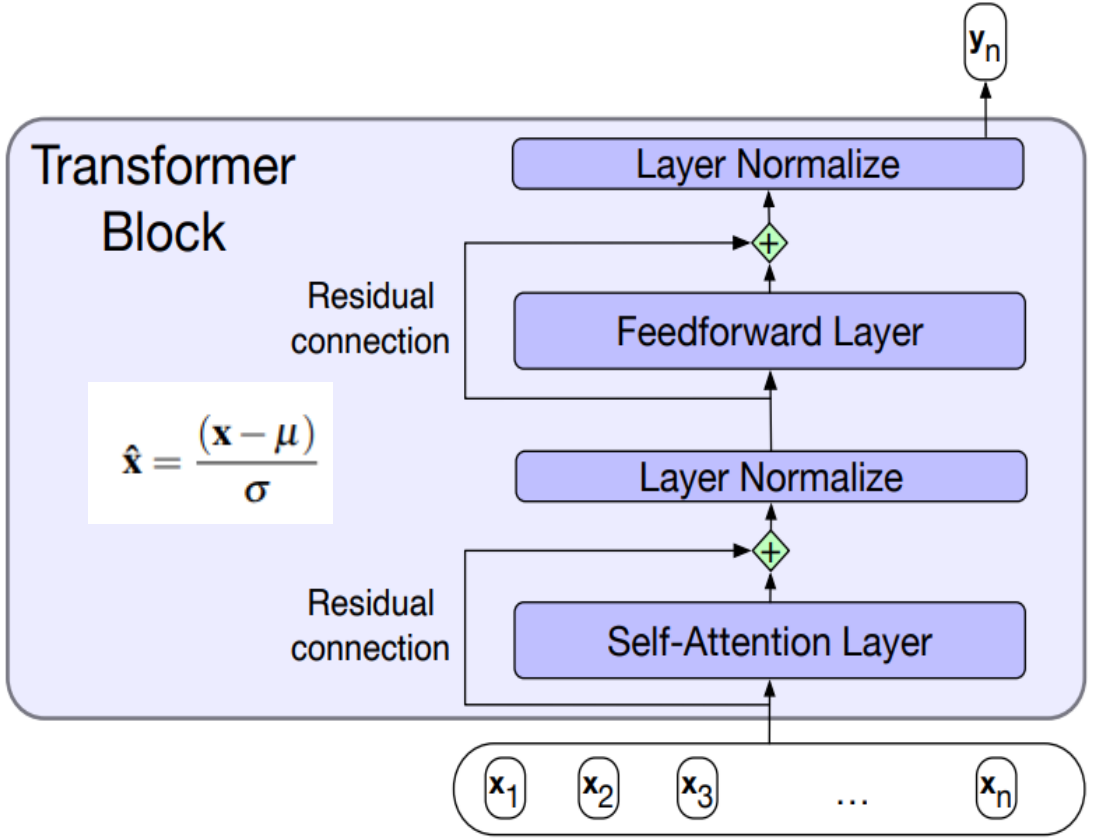
Transformer blocks

Additionally, we include

- feed-forward layer
- **residual connections**
  - just skip a layer
- **layer normalization**
  - layer-wise **z**-score
  - + linear projection
    - $L_{norm} = \gamma \mathbf{x} + \beta$

These can be stacked

- just like the RNNs



Multihead Attention

Problem: there can be multiple relationships between a pair of words

- syntactic, semantic, discourse…

Idea: **Multi-head self-attention**

- sets of parallel attention layers
- each with its own $\mathbf{W}^{Q}$; $\mathbf{W}^{K}$; $\mathbf{W}^{V}$

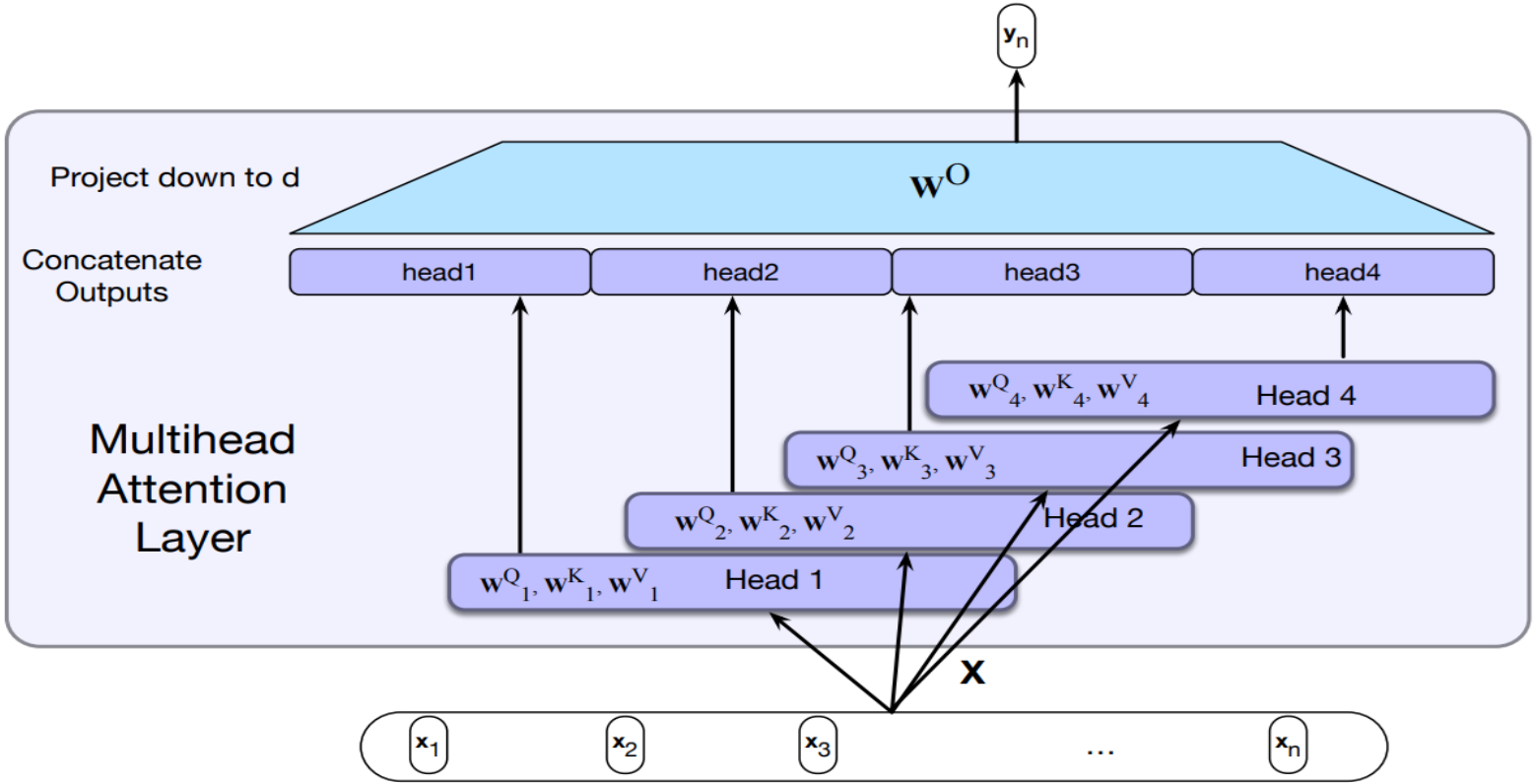
$$MultiHeadAttn(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W}^{O}$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^{Q} \; ; \; \mathbf{K} = \mathbf{X}\mathbf{W}_i^{K} \; ; \; \mathbf{V} = \mathbf{X}\mathbf{W}_i^{V}$$

$$\mathbf{head}_i = SelfAttention(\mathbf{Q}, \mathbf{K}, \mathbf{V})$$

The rest of the transformer block remains the same

Multihead Attention



Modeling word order

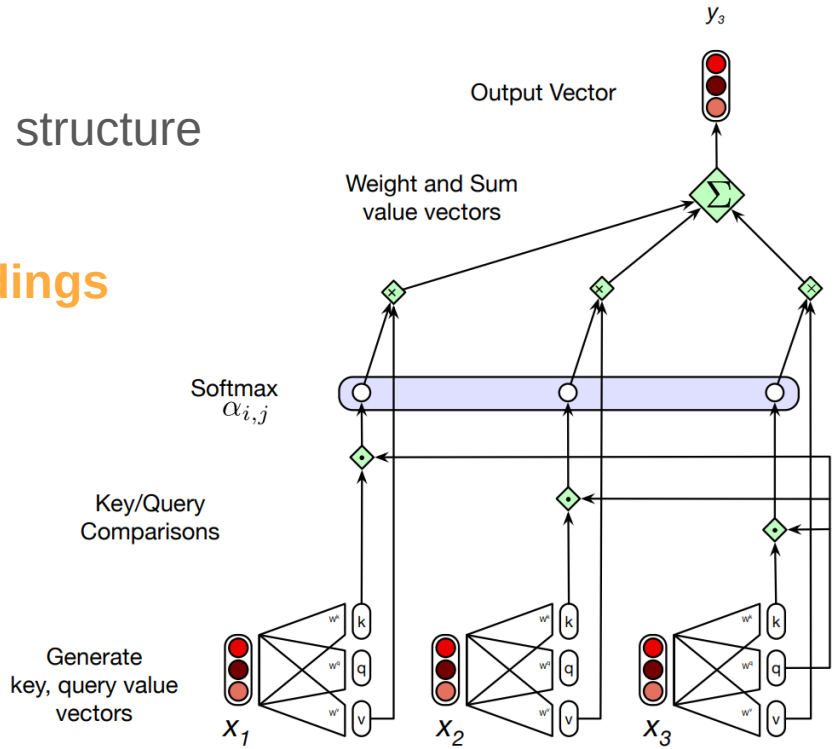
How about word positions?

- in RNNs, these were implicit in the model structure
- here we are invariant to their positions!

Idea: combine words with **positional embeddings**

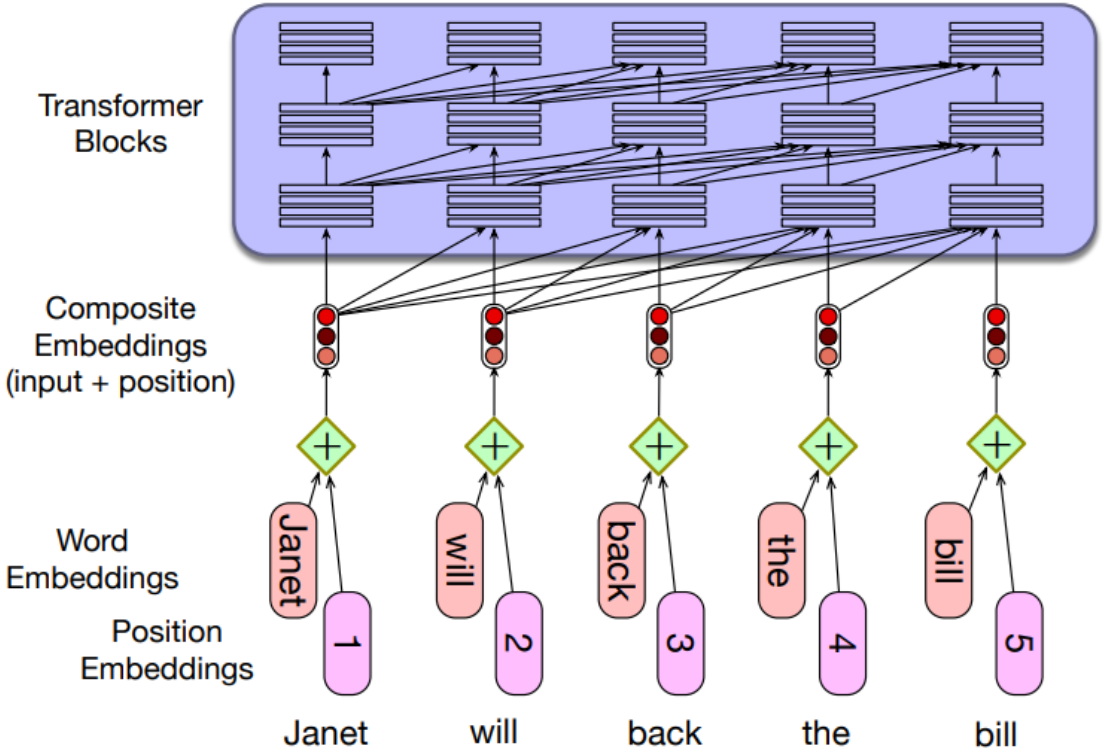
- just like words, we can embed positions
- e.g. position 3 = some learnable vector
- up to some max. N

In the model, we just sum with the words



Modeling word order

Positional embeddings:



problem:

- higher positions will receive fewer updates

Positional encoding

Better idea: use a static $\mathbb{N} \rightarrow \mathbb{R}^d$ function instead

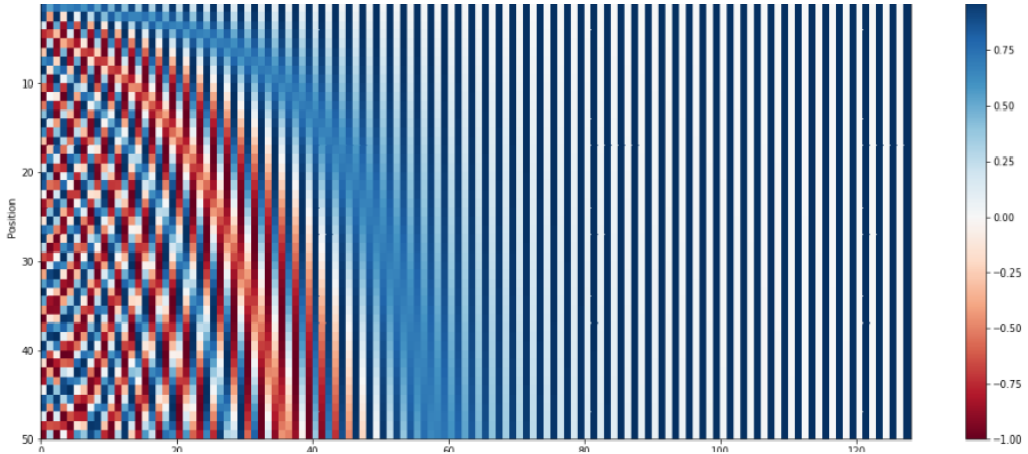
E.g. sin + cos with varying frequencies

- from the original “Attention is all you need” paper

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

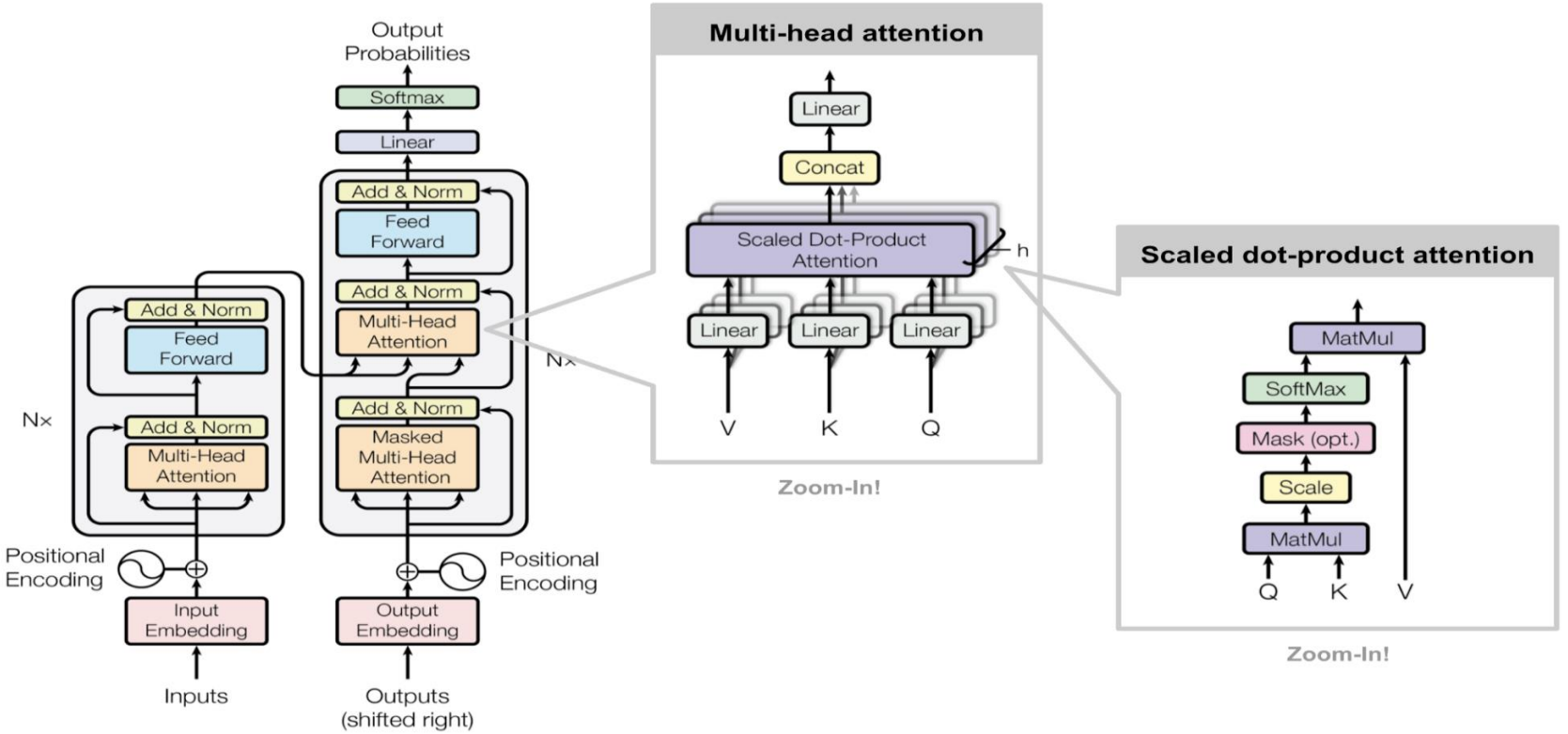
0:	0 0 0 0	8:	1 0 0 0
1:	0 0 0 1	9:	1 0 0 1
2:	0 0 1 0	10:	1 0 1 0
3:	0 0 1 1	11:	1 0 1 1
4:	0 1 0 0	12:	1 1 0 0
5:	0 1 0 1	13:	1 1 0 1
6:	0 1 1 0	14:	1 1 1 0
7:	0 1 1 1	15:	1 1 1 1



Source: https://kazemnejad.com/blog/transformer_architecture_positional_encoding

Complete Transformer block

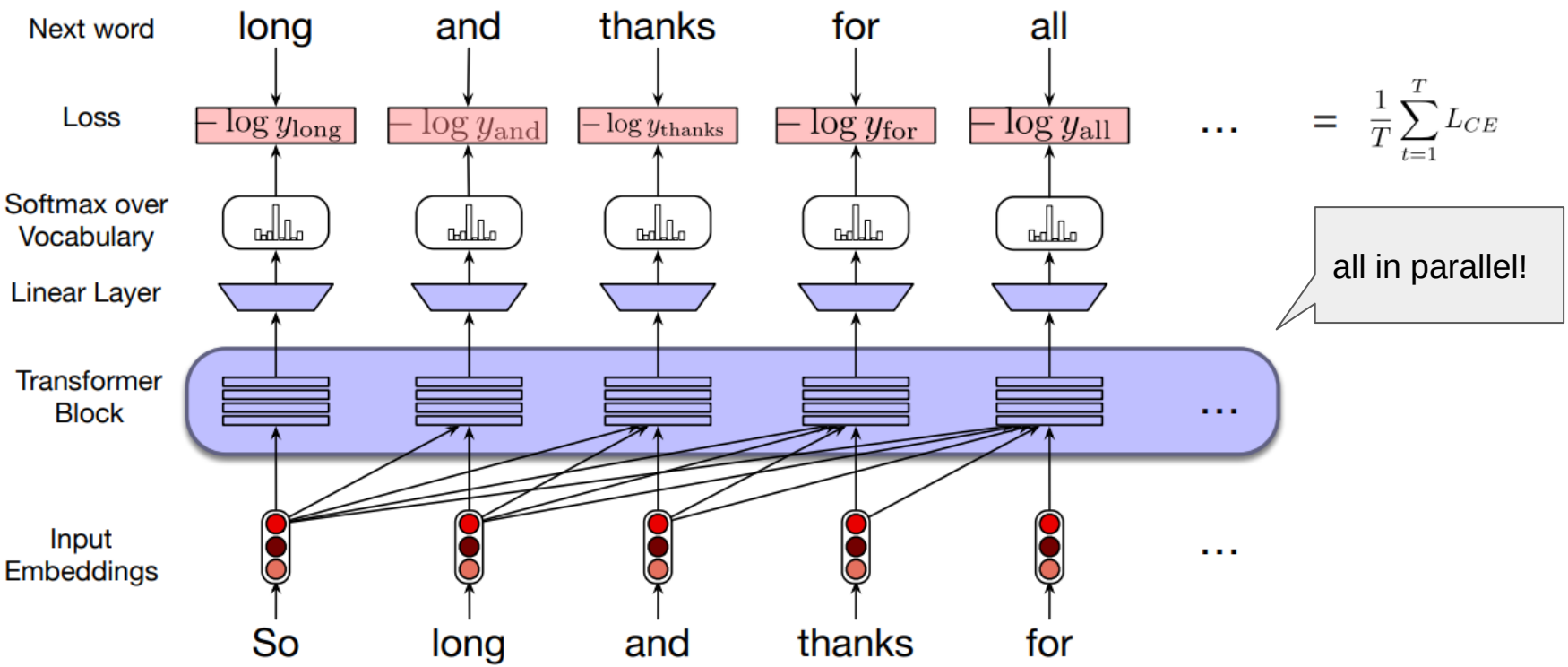
source: Attention is all you need
https://arxiv.org/abs/1706.03762



Neural models

- Some more applications with Transformers

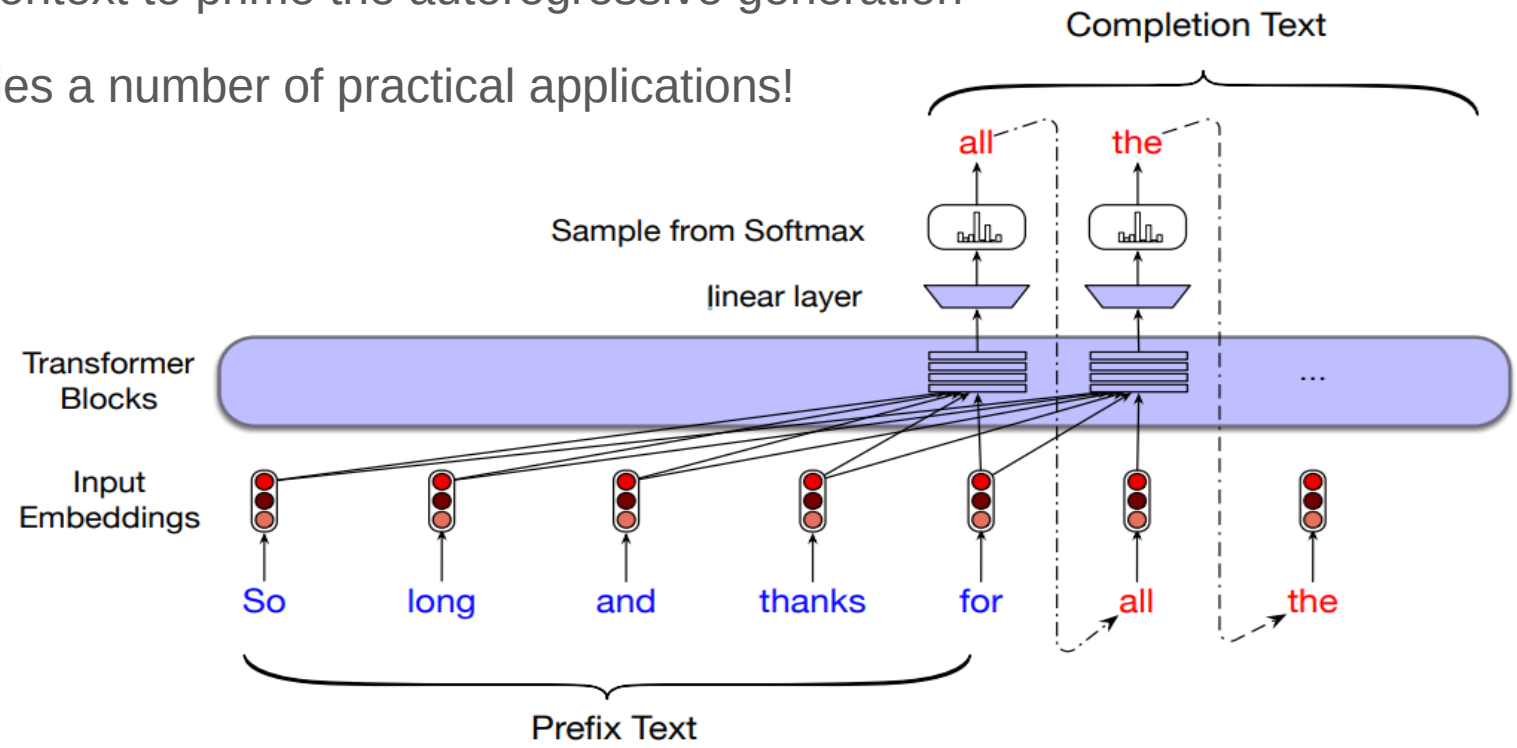
Transformers as Language Models



Contextual Generation

Idea: use context to prime the autoregressive generation

- underlies a number of practical applications!



Text summarization

Abstractive version

One practical application of the context-based autoregressive generation

Supervised training regime (data e.g. from news):

- full articles + their summaries
- $(x_1,..., x_m)$ + $(y_1,..., y_n)$

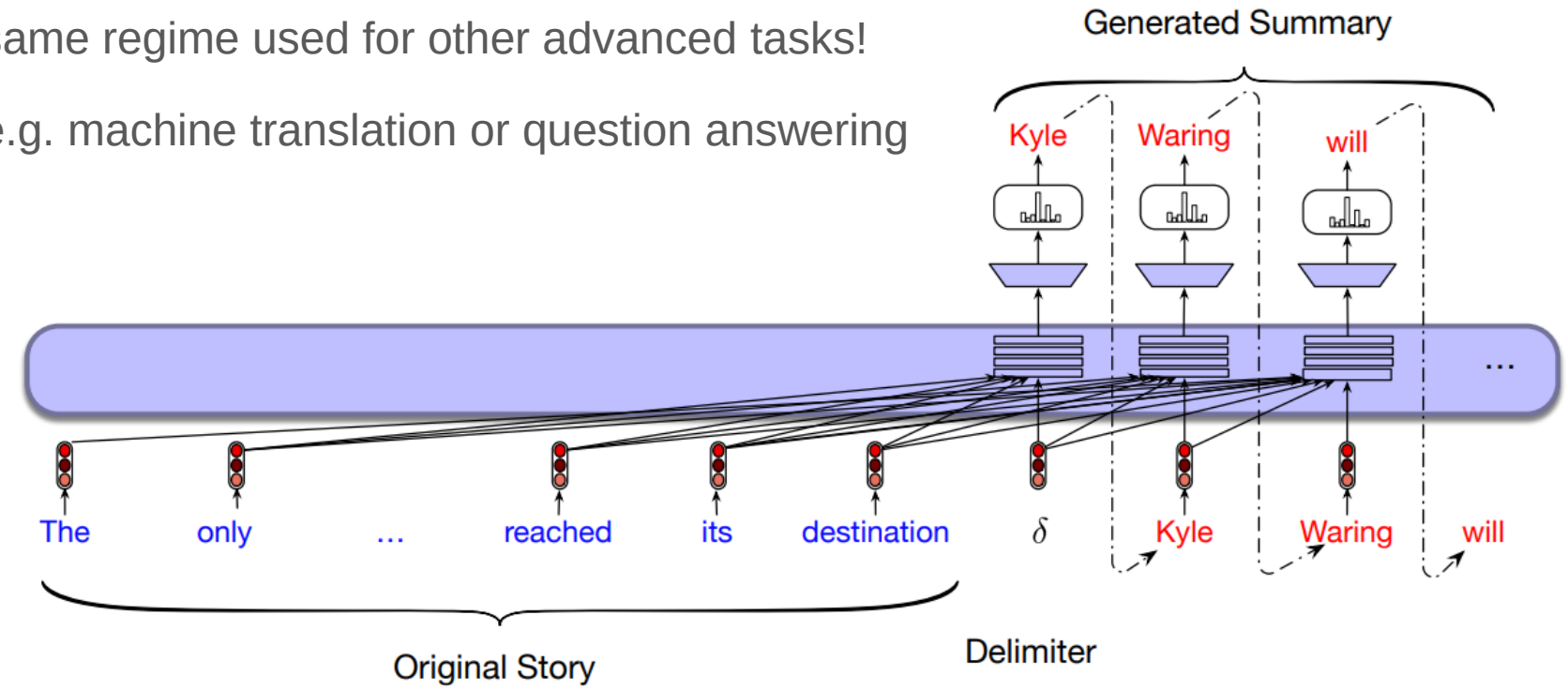
With transformers, we simply concatenate these: $(x_1,..., x_m, \boldsymbol{\delta}, y_1,...y_n)$

- and train a **standard (!)** autoregressive language model
  - with teacher forcing, exactly as we did before!

Text summarization

The same regime used for other advanced tasks!

- e.g. machine translation or question answering



Transformers summary

Transformers are used extensively across all NLP tasks now

- sequence labeling: part-of-speech, named entities, …
- sequence classification: sentiment, spam, …

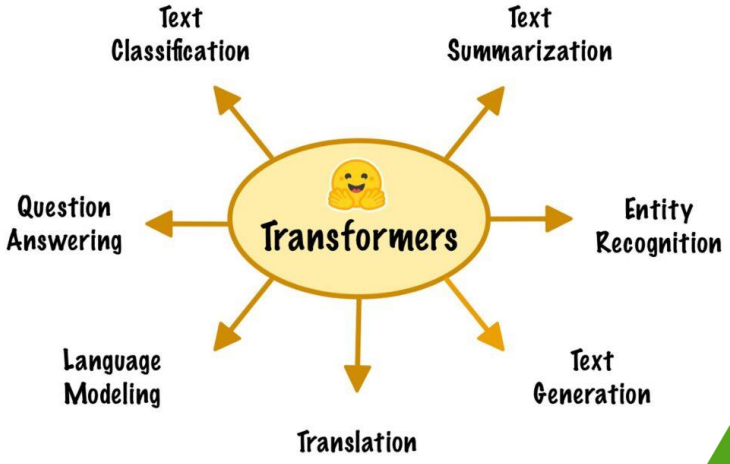
We commonly use **pre-training** on large textual corpora

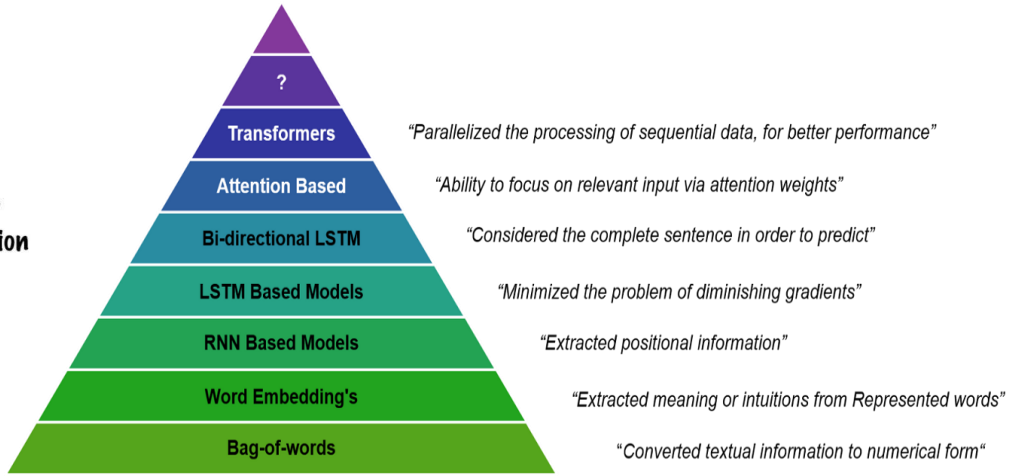
- in a self-supervised manner as a standard LM

And then **fine-tune** on top of the learned representations

- This was the key to the many recent breakthroughs in NLP…

NLP journey overview





Things we did not cover…

- All of linguistics
  - Parsing
    - Dependency
    - Constituency
  - Grammars
  - Lexicons, corpora
- Text preprocessing
  - Regular expressions
  - Edit distance
- Applications
  - Basic: Part of speech tagging, NER, RE, …
  - Advanced: machine translation, QA, dialog systems, …