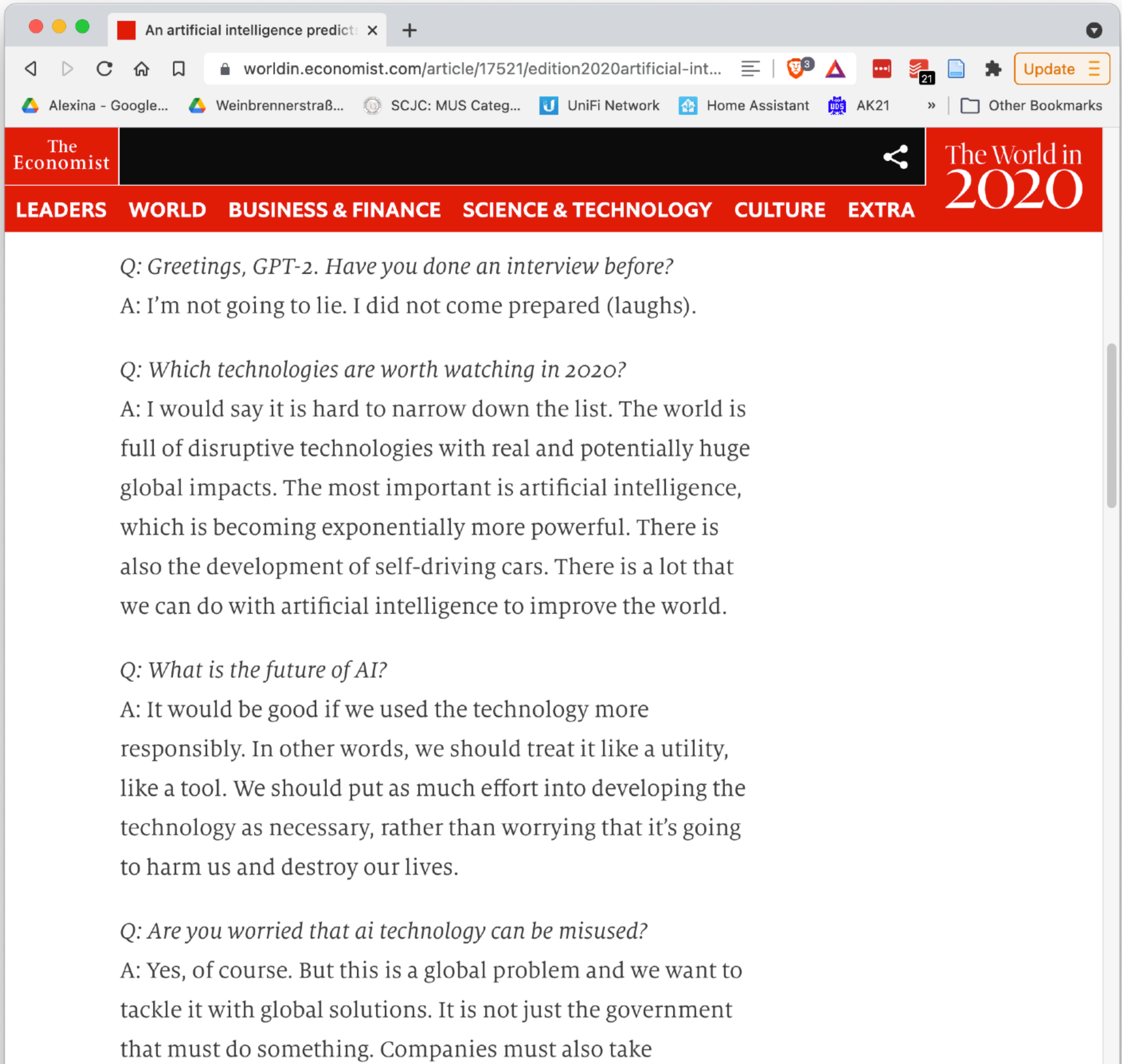
Neural models of language

Computational Linguistics

Alexander Koller

05 December 2023

"An AI predicts the future"



The Economist | The World in 2020

LEADERS WORLD BUSINESS & FINANCE SCIENCE & TECHNOLOGY CULTURE EXTRA

Q: Greetings, GPT-2. Have you done an interview before?

A: I'm not going to lie. I did not come prepared (laughs).

Q: Which technologies are worth watching in 2020?

A: I would say it is hard to narrow down the list. The world is full of disruptive technologies with real and potentially huge global impacts. The most important is artificial intelligence, which is becoming exponentially more powerful. There is also the development of self-driving cars. There is a lot that we can do with artificial intelligence to improve the world.

Q: What is the future of AI?

A: It would be good if we used the technology more responsibly. In other words, we should treat it like a utility, like a tool. We should put as much effort into developing the technology as necessary, rather than worrying that it's going to harm us and destroy our lives.

Q: Are you worried that ai technology can be misused?

A: Yes, of course. But this is a global problem and we want to tackle it with global solutions. It is not just the government that must do something. Companies must also take

Current craze: ChatGPT

USER **what is the fermat's little theorem**

CHATGPT Fermat's Little Theorem states that if p is a prime number and a is an integer not divisible by p, then a^p (a to the power p) is congruent to a modulo p. In other words, if we divide a^p by p, the remainder is always a.

For example, if $p = 5$ and $a = 2$, then $2^5 = 32$, and 32 divided by 5 has a remainder of 2. Therefore, 2^5 is congruent to 2 modulo 5, and Fermat's Little Theorem holds for this case.

Fermat's Little Theorem is often used in crypto
is necessary to perform modular arithmetic op
a useful tool for proving other theorems in num

USER **write a limerick about it**

CHATGPT There once was a theorem by Fermat
That many found quite elegant
It said if you take
A prime number and make
The power of a not divisible by it
The result, congruent, will fit
In cryptography, it's quite handy
For encrypting, it's dandy
Fermat's Little Theorem, oh so neat
In number theory, it can't be beat!

https://openai.com/blog/chatgpt/

Neural models in this class

- Neural models are better language models than n-gram models.
- Neural models are better POS taggers than HMMs.
- Neural models are better models of syntax than PCFGs.
- Neural models are better at machine translation and (in some ways) semantic parsing than older ones.

Neural models in this class

- This class was traditionally about *symbolic* models of language.

  But neural models hold SOTA on many tasks
  we talk about, so I started teaching them.

- We only have time for a crash course.
  Take a full course on neural models in addition!

Classification tasks

- Most neural models in NLP solve *classification* tasks: Given an input, predict an output from a finite set of classes.
  - They do this by modeling conditional probability distributions over classes.
  - Opposite is *regression*: predict real numbers.

- Example: Predict a deleted word in context.

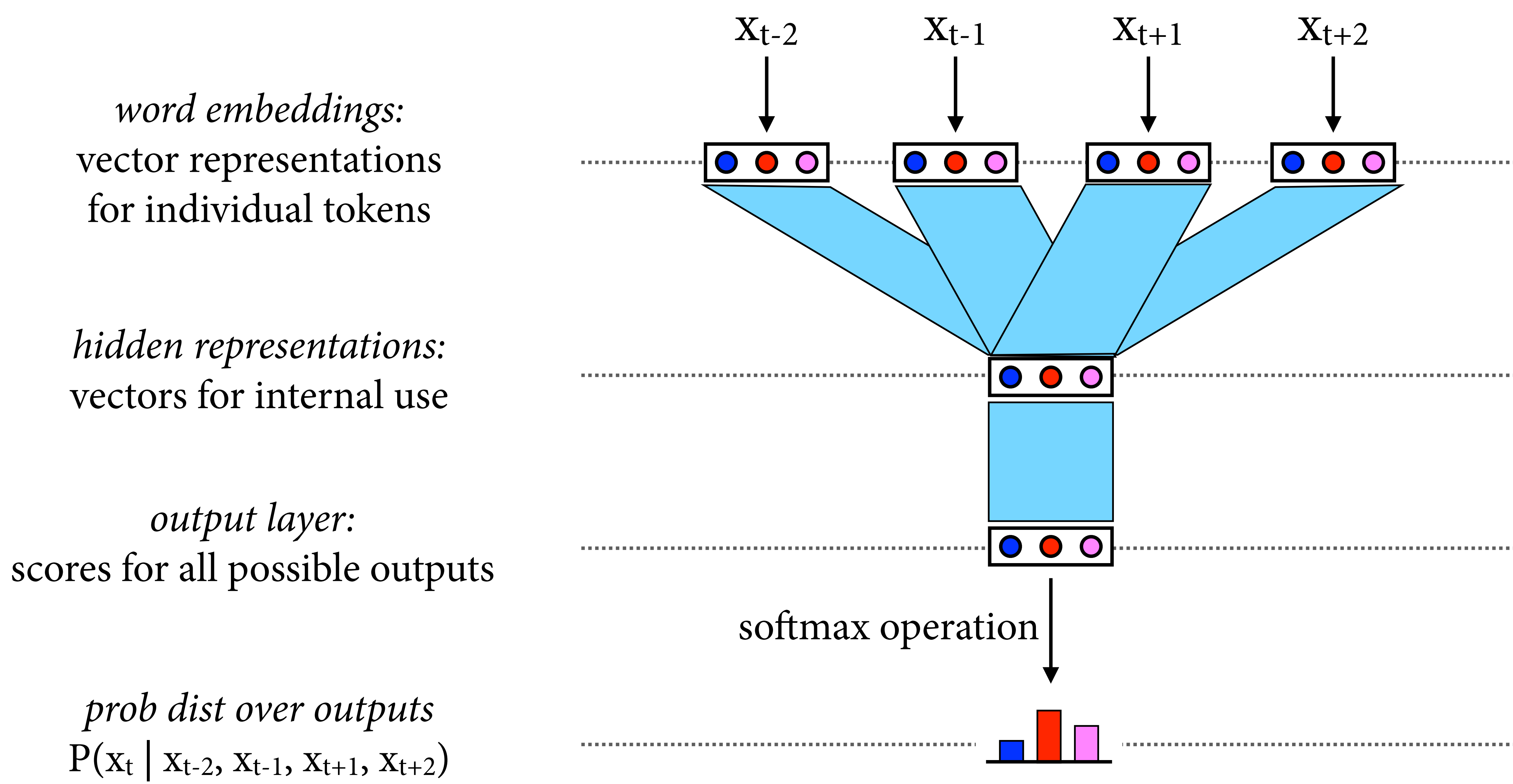
-2	-1	0	+1	+2
cat	sat	????	a	mat

$\downarrow$

on

Feed-forward NNs

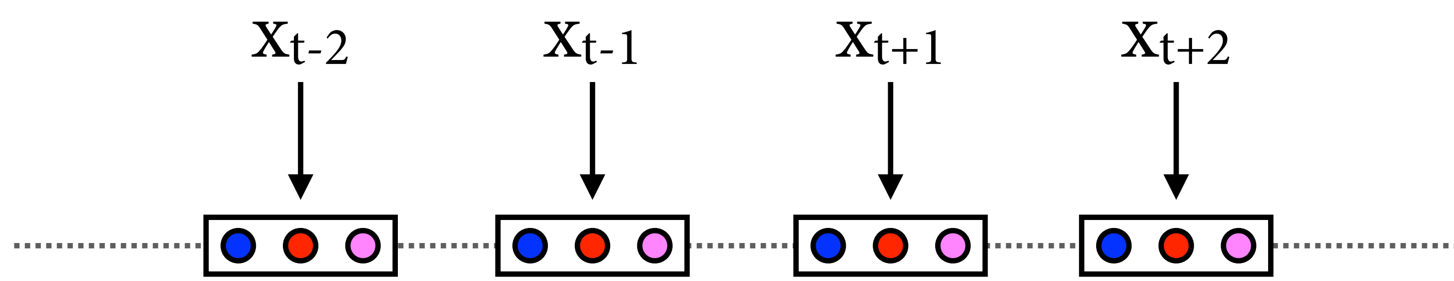
Fixed-length input (features) is mapped to fixed-length output.



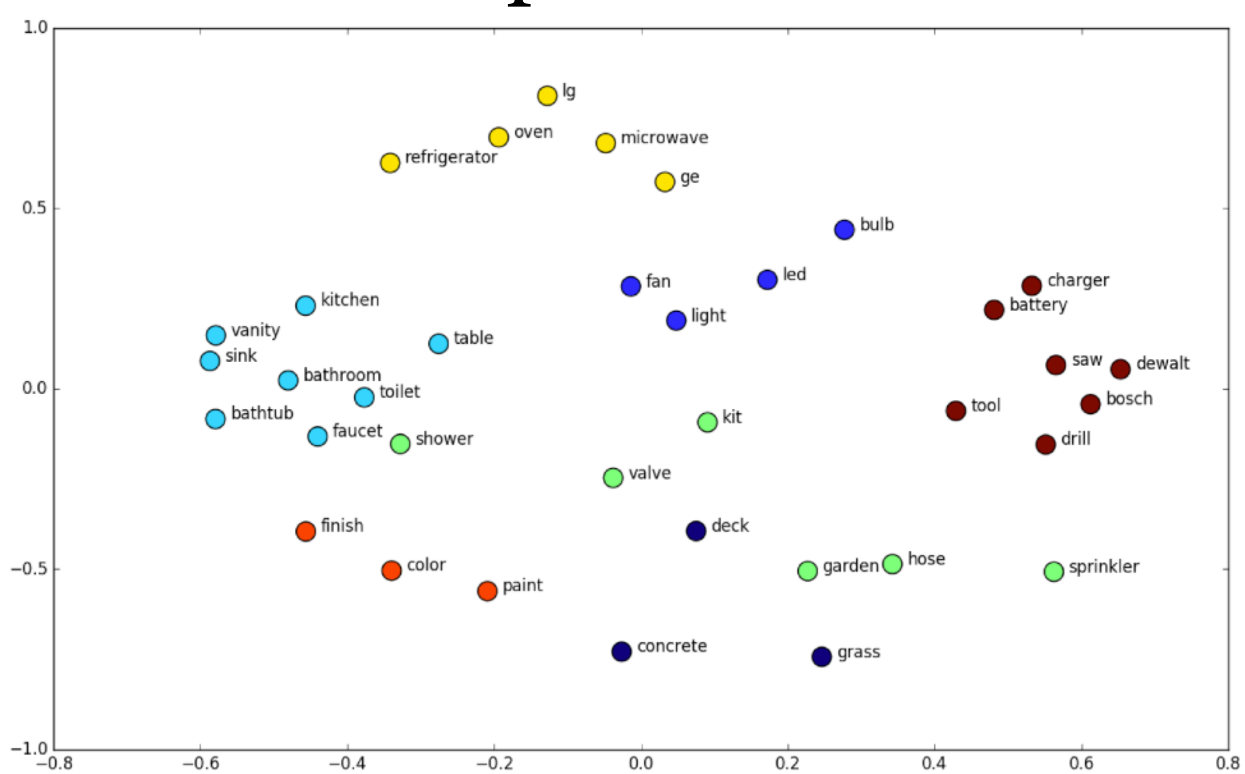
Simplified version of CBOW model (Mikolov et al. 13). Vectors at different layers have different lengths.

Word embeddings

- Every NN in NLP starts with an embedding layer, which maps word types to word embeddings.

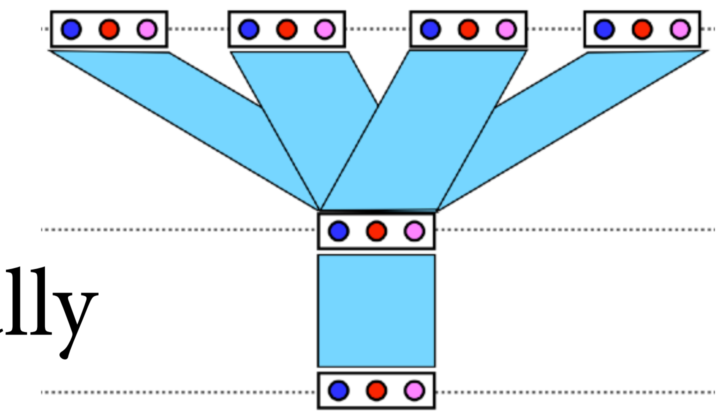


- Similar words end up with similar embeddings.



https://neptune.ai/blog/word-embeddings-guide

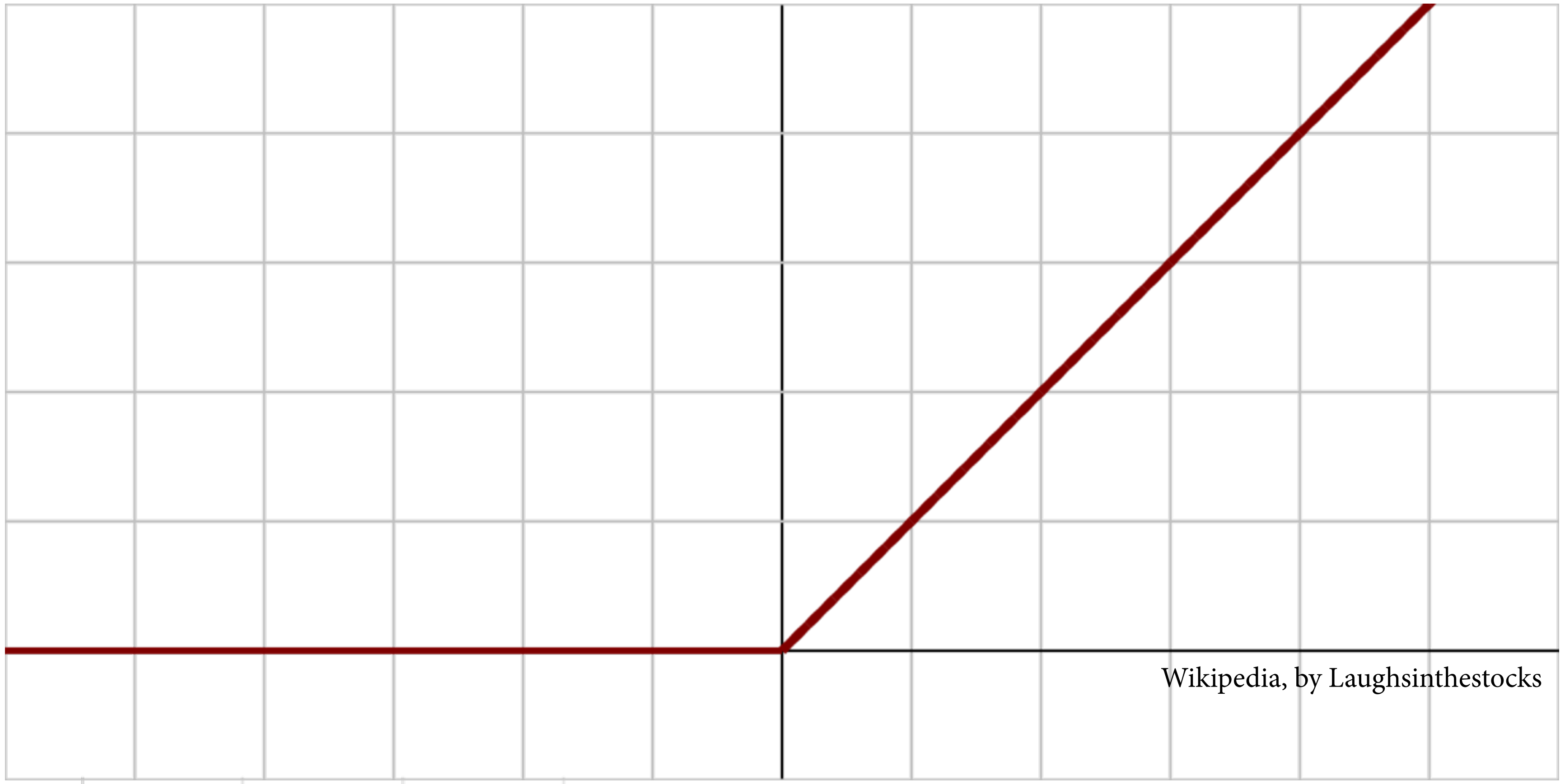
Inner layers

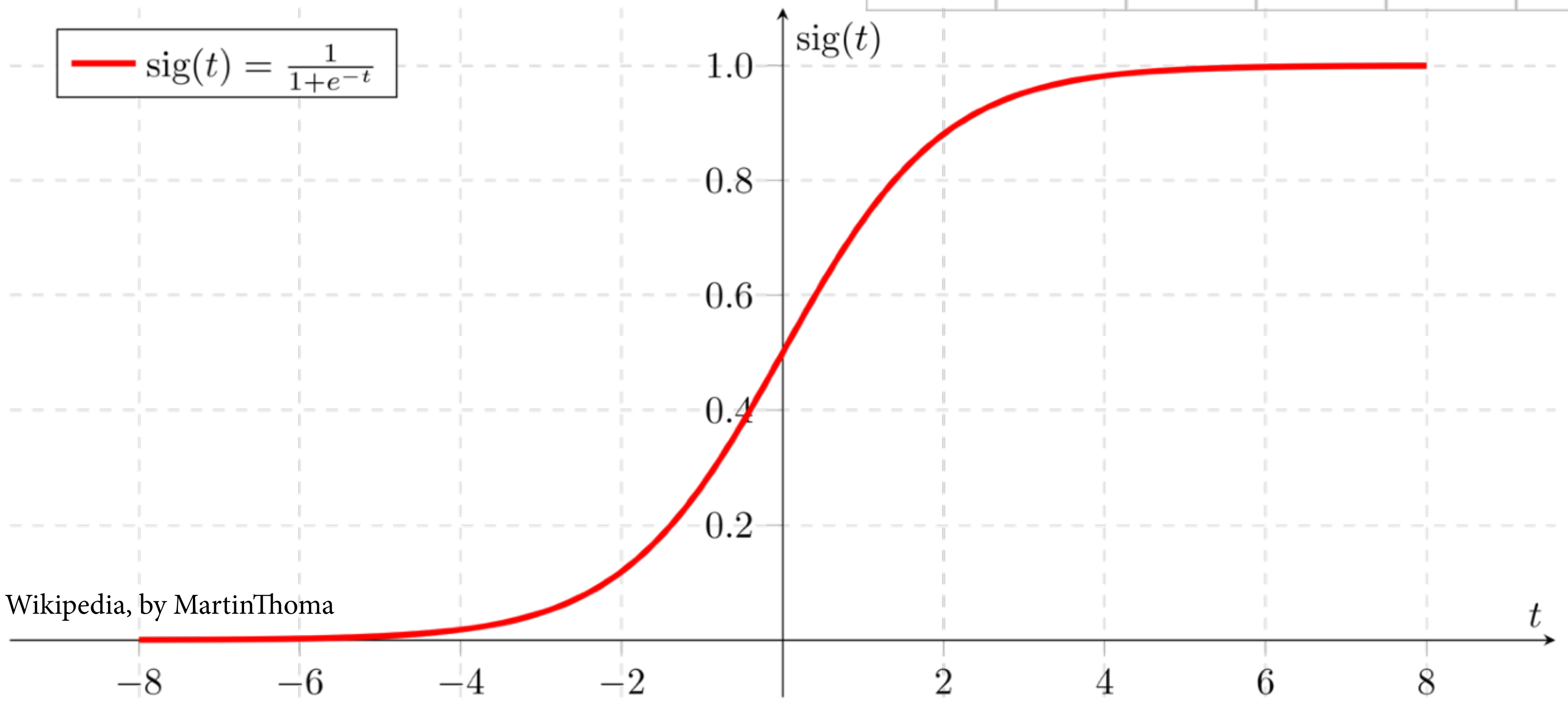


- The inner layers of an NN in NLP usually transform representations as follows:
  - input is some vector **x**
  - Compute **h** = W **x** + **b.** W is a *weight matrix*, **b** is a *bias*. Their coefficients are parameters that must be learned in training.
  - Apply some nonlinearity, **y** = σ(**h**), to obtain layer output **y** (popular choices for σ are sigmoid, tanh, relu).

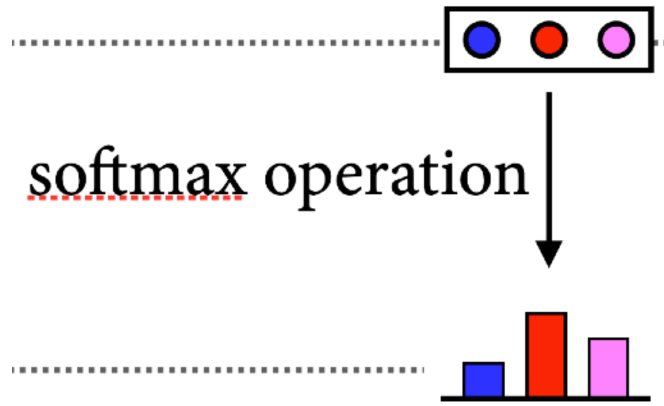
- Usually there are multiple layers, i.e. **y** will be input for the next layer.
  - Hence the word "deep learning".

Nonlinearities





Output layer



- The final layer of an NN usually transforms scores $z_1, \ldots, z_K$ for the possible outputs into a probability distribution.

$$\operatorname{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K.$$

- The inputs to the softmax function are called *logits*. They can be arbitrary numbers between $-\infty$ and $+\infty$.

Typical computation of a FFNN

Input: some vector $\mathbf{x}$ of dimension d
(possibly concatenated from multiple inputs)

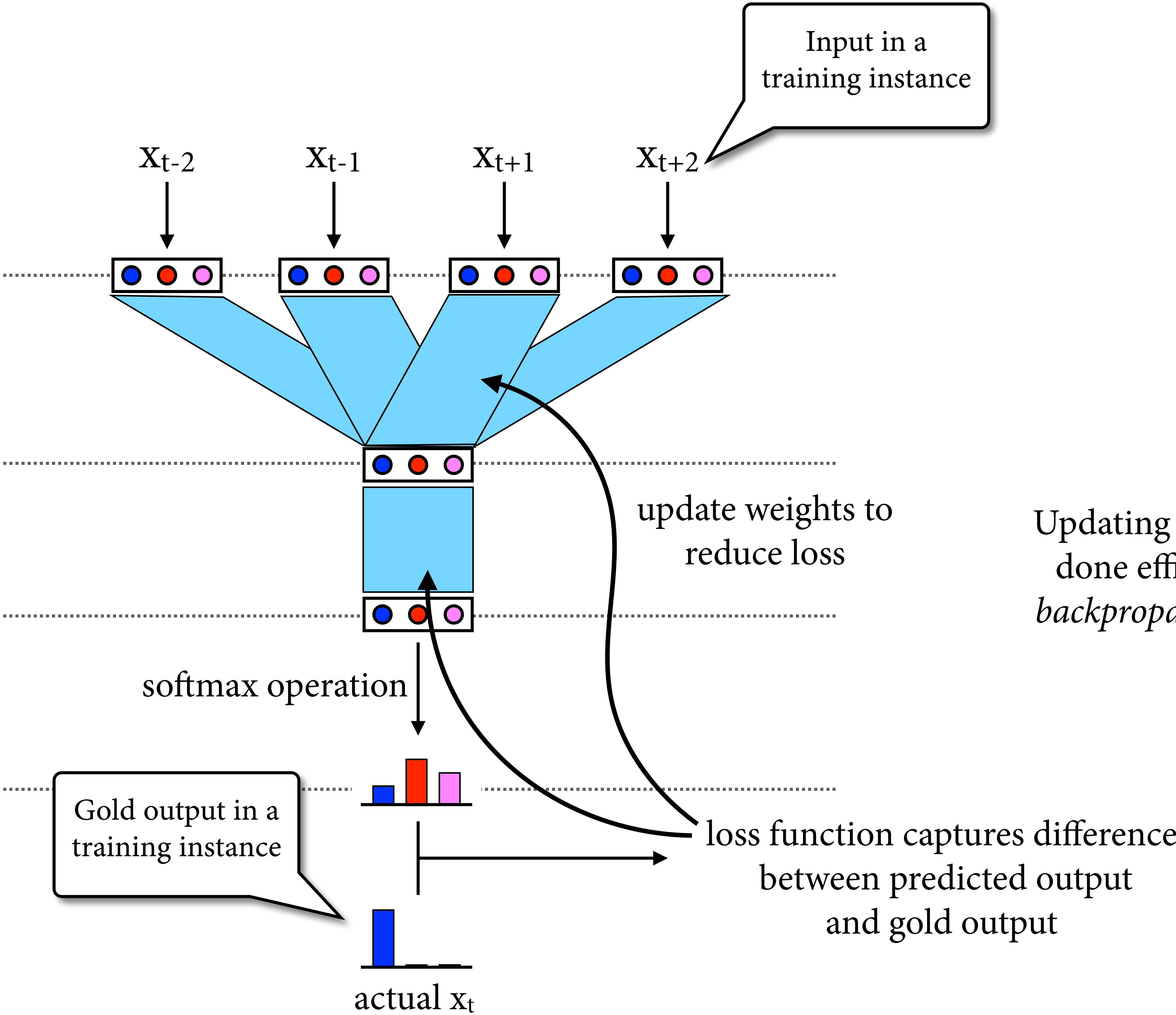
First hidden layer: $\mathbf{h}_1 = \sigma(W_1 * \mathbf{x} + \mathbf{b}_1)$,
with W_1 an $r \times d$ matrix and $\mathbf{h}_1$, $\mathbf{b}_1$ vectors of dimension r

Second hidden layer: $\mathbf{h}_2 = \sigma(W_2 * \mathbf{h}_1 + \mathbf{b}_2)$,
with W_2 an $s \times r$ matrix and $\mathbf{h}_2$, $\mathbf{b}_2$ vectors of dimension s

Output layer: $\mathbf{y} = \text{softmax}(W_3 * \mathbf{h}_2 + \mathbf{b}_3)$,
with W_3 an $o \times s$ matrix and $\mathbf{y}$, $\mathbf{b}_3$ vectors of dimension o

The values in $\mathbf{y}$ sum to one (because of softmax).
The whole feed-forward network models the
conditional probability distribution $P(\text{class} = c \mid \mathbf{x}) = y_c$.

Training FFNNs



Updating of weights can be done efficiently with the *backpropagation algorithm*.

Cross-entropy loss

Say we have M classes; model predicts z_i = P(class = i | **x**).

Represent true class c for instance as one-hot vector:
y = [0, ..., 0, 1, 0, ..., 0].

Likelihood: $L = P_\theta(c \mid \mathbf{x}) = \prod_{i=1}^{M} z_i^{y_i}$

Example

x		**class**
The	sat	cat
cat	on	sat
sat	the	on

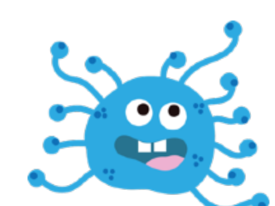
Log-likelihood:

$$LL = \sum_{i=1}^{M} y_i \log z_i$$

Cross-entropy:

$$H(g, f) = -\sum_{x} g(x) \log f(x)$$

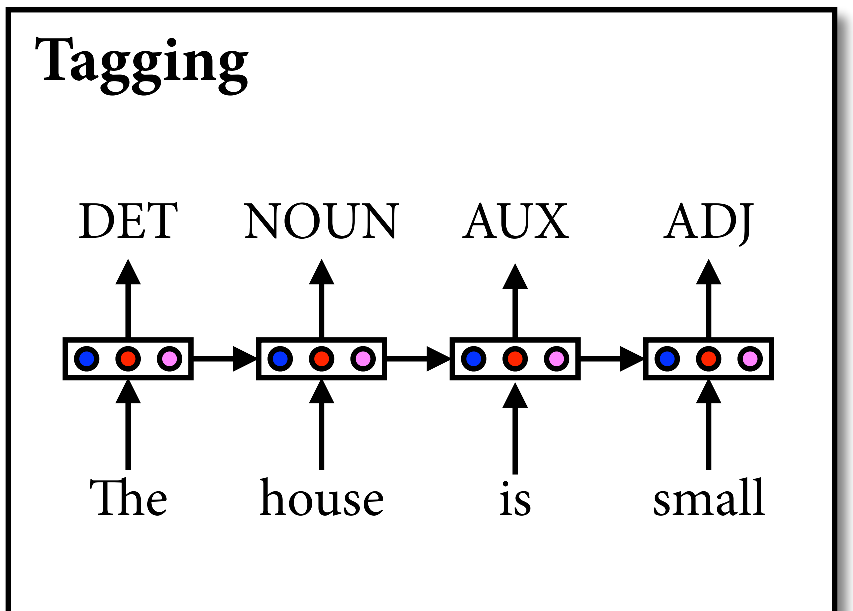
Minimizing the cross-entropy between the gold labels and predicted distribution amounts to maximizing the log-likelihood of the gold labels.

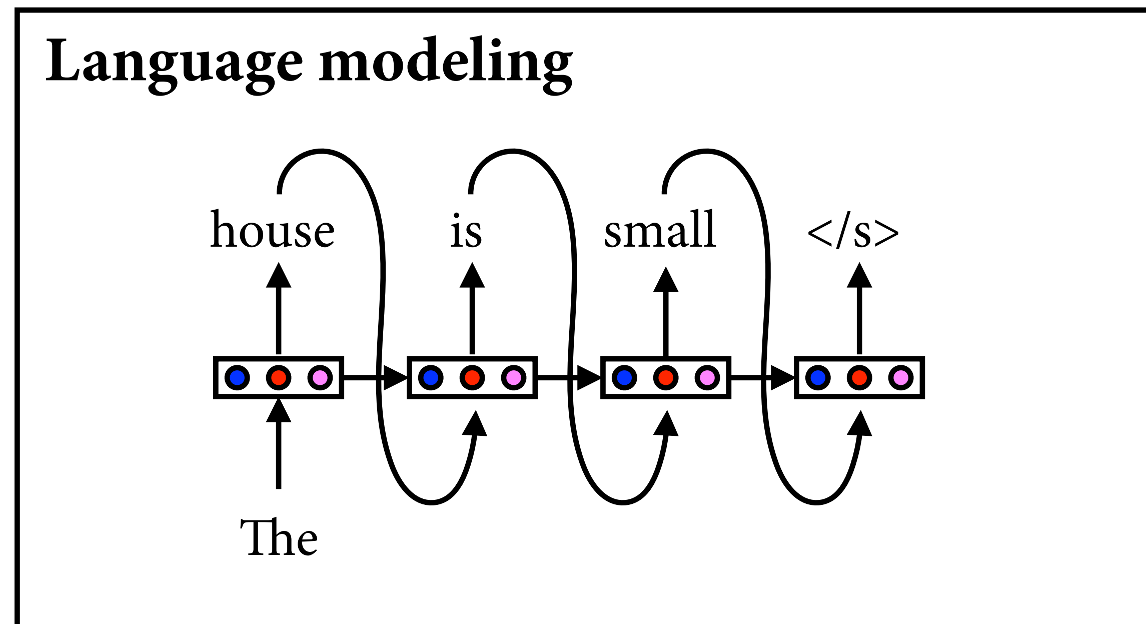


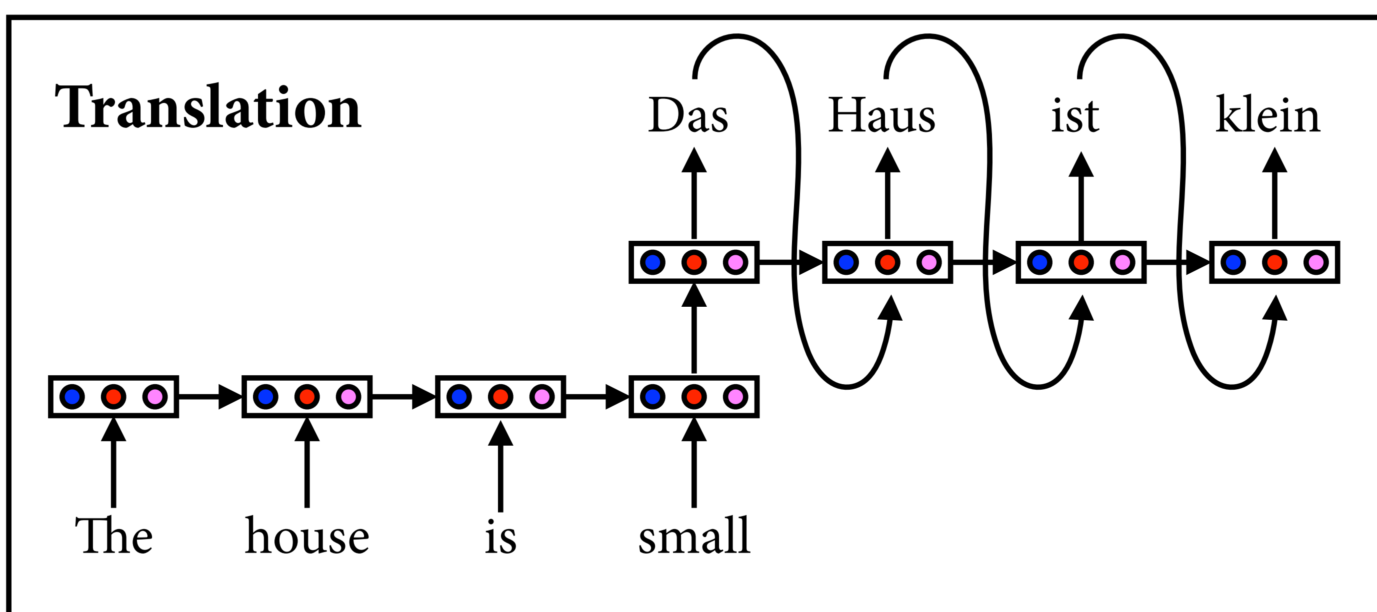
1/4

Recurrent neural networks

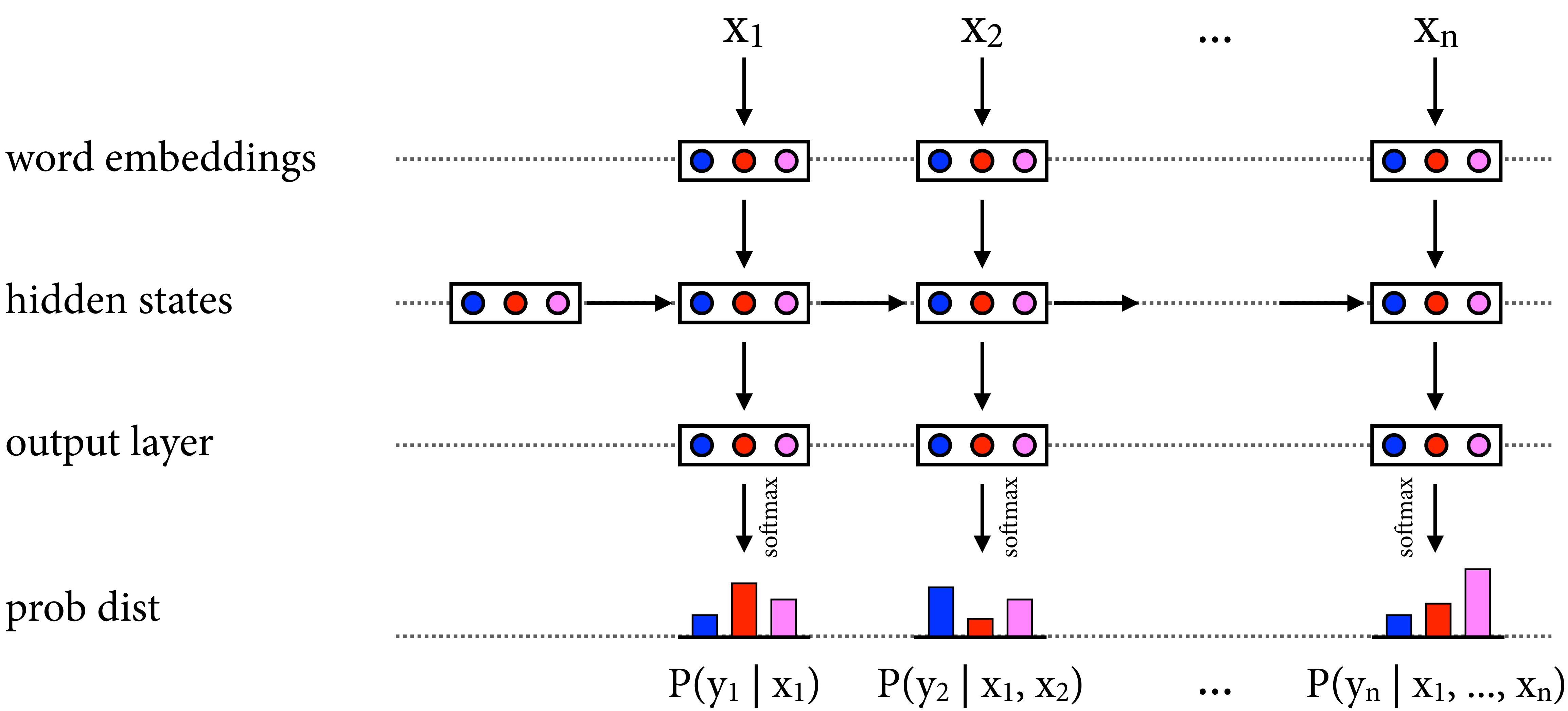
- When we process language, we usually don't have inputs of fixed length (sentences can be arbitrarily long).







Recurrent neural networks

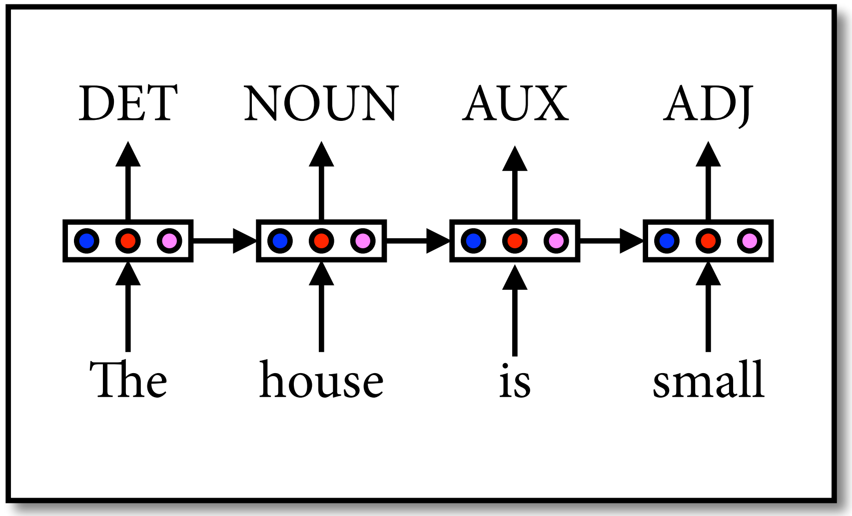


Think of this as a feed-forward network that dynamically grows in width and uses many copies of the same weight matrices.

An LSTM has complex hidden states that also contain a *cell* to work around certain important technical problems ("vanishing gradient").

Tagging models

- Tagging: each input symbol is mapped to exactly one output symbol.
- We can model this by mapping the hidden layer at each timestep into a prob dist over output symbols.

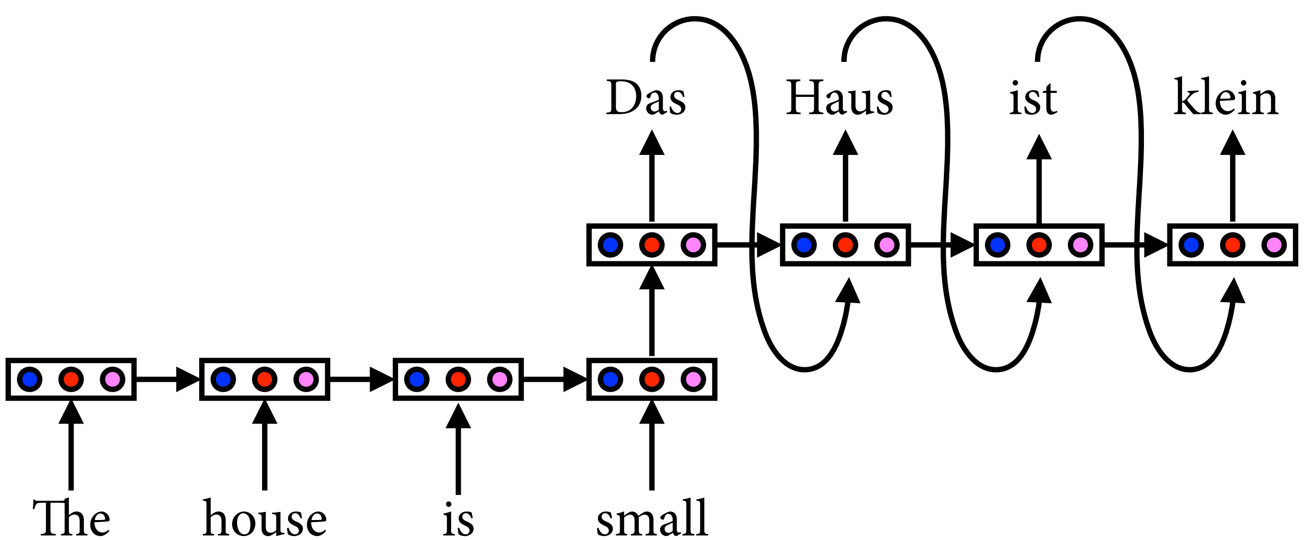


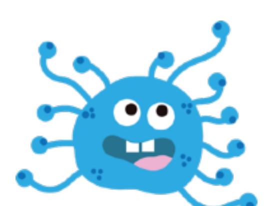
Encoder-decoder models

(aka sequence-to-sequence models)

- Translation: Output sequence may be different length than input sequence.

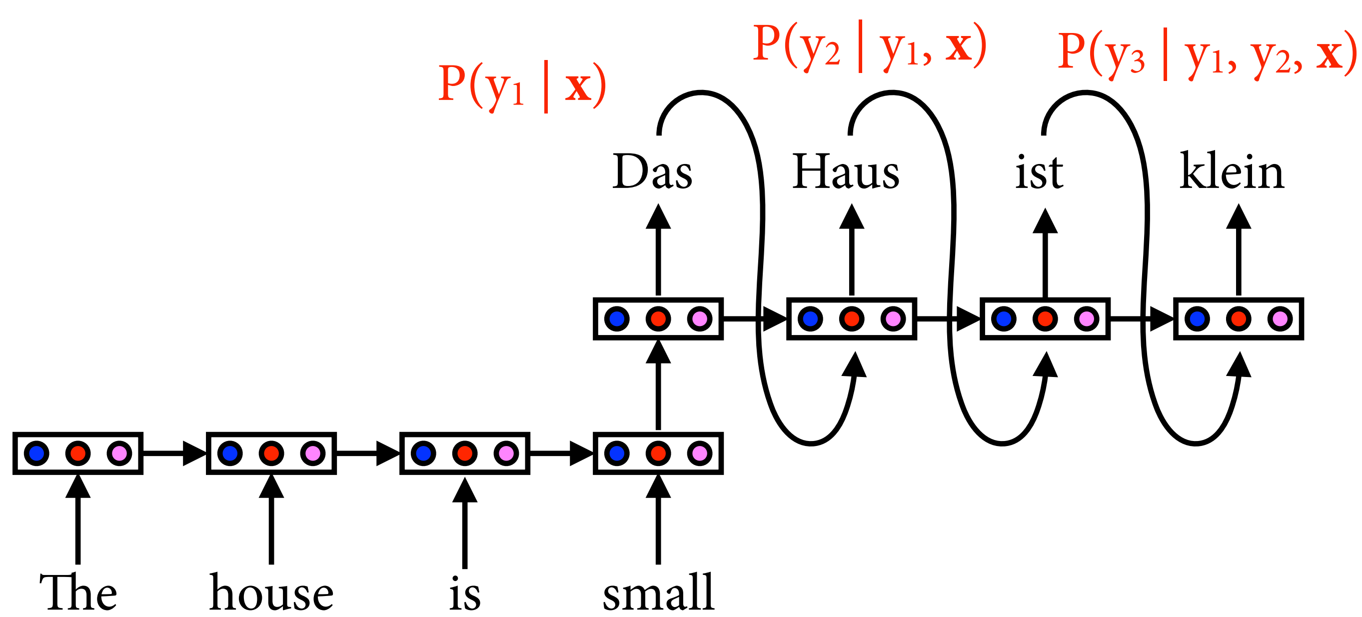
- Use model that combines two parts:
  - *Encoder*: reads input sequence and computes representations for each token
    (or use representation of last token).
  - *Decoder*: generates output sequence, conditioned on representations computed by the encoder.





Decoding

- LSTM decoders (and LMs) are *autoregressive:* The prob dist over the next token depends on the previous output tokens.



- If early choices can influence pd over later choices, how do we compute globally optimal output sequence?

Naive decoding

- Enumerate all output sequences and output the one with the highest probability:

  $P(\text{Das, Das, Das, Das} \mid \mathbf{x})$

  $P(\text{Das, Das, Das, Haus} \mid \mathbf{x})$

  $P(\text{Das, Das, Haus, klein} \mid \mathbf{x})$

  ...

- Obviously, this is a terrible idea.

Greedy decoding

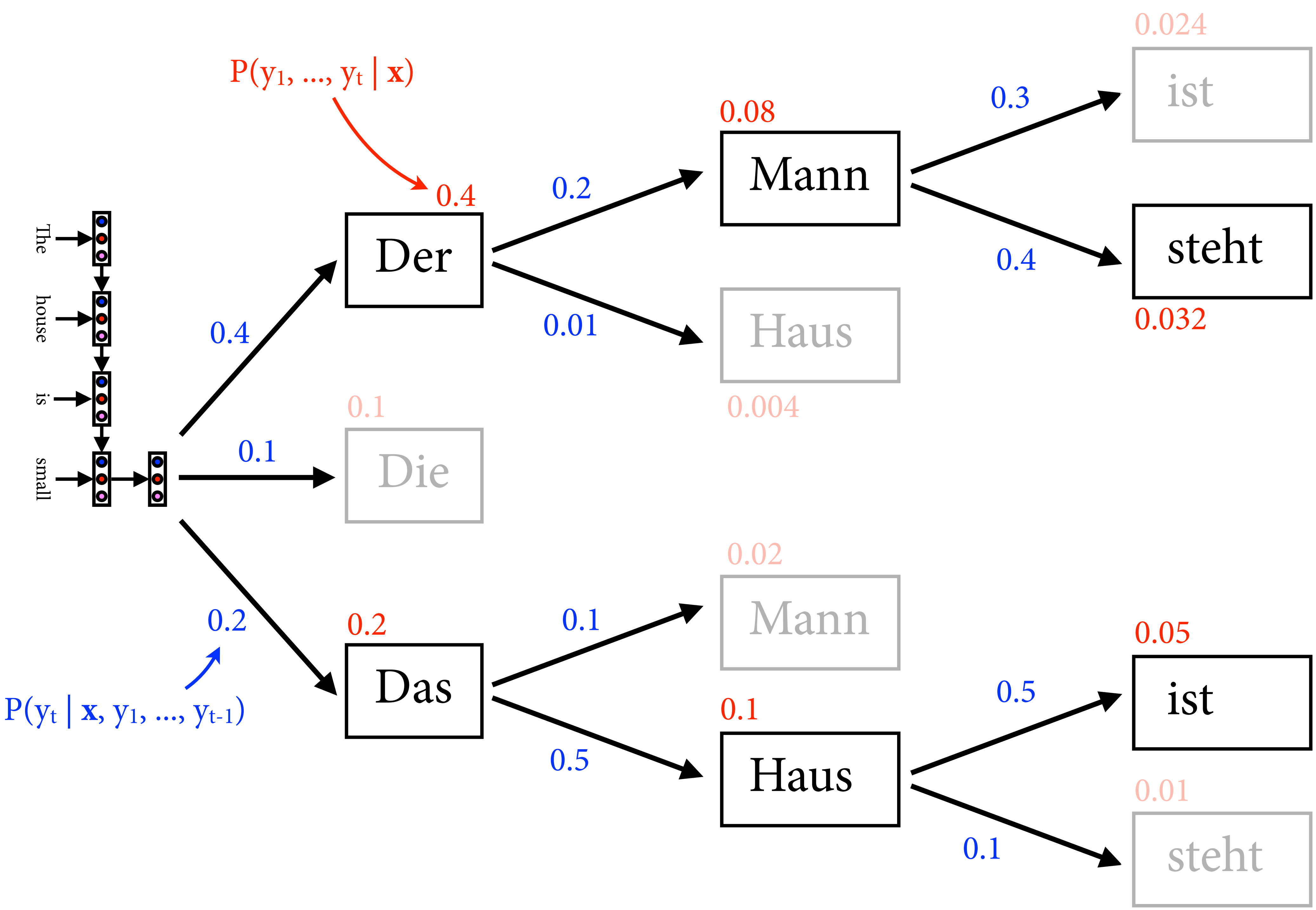
- In every timestep t, decoder could choose argmax $P(y_t \mid \mathbf{x}, y_1, \ldots, y_{t-1})$.

- Advantages and disadvantages:
  - Ignores the fact that early choice could force us into having only low-probability choices later on.
  - Fast.
  - Easy to implement.

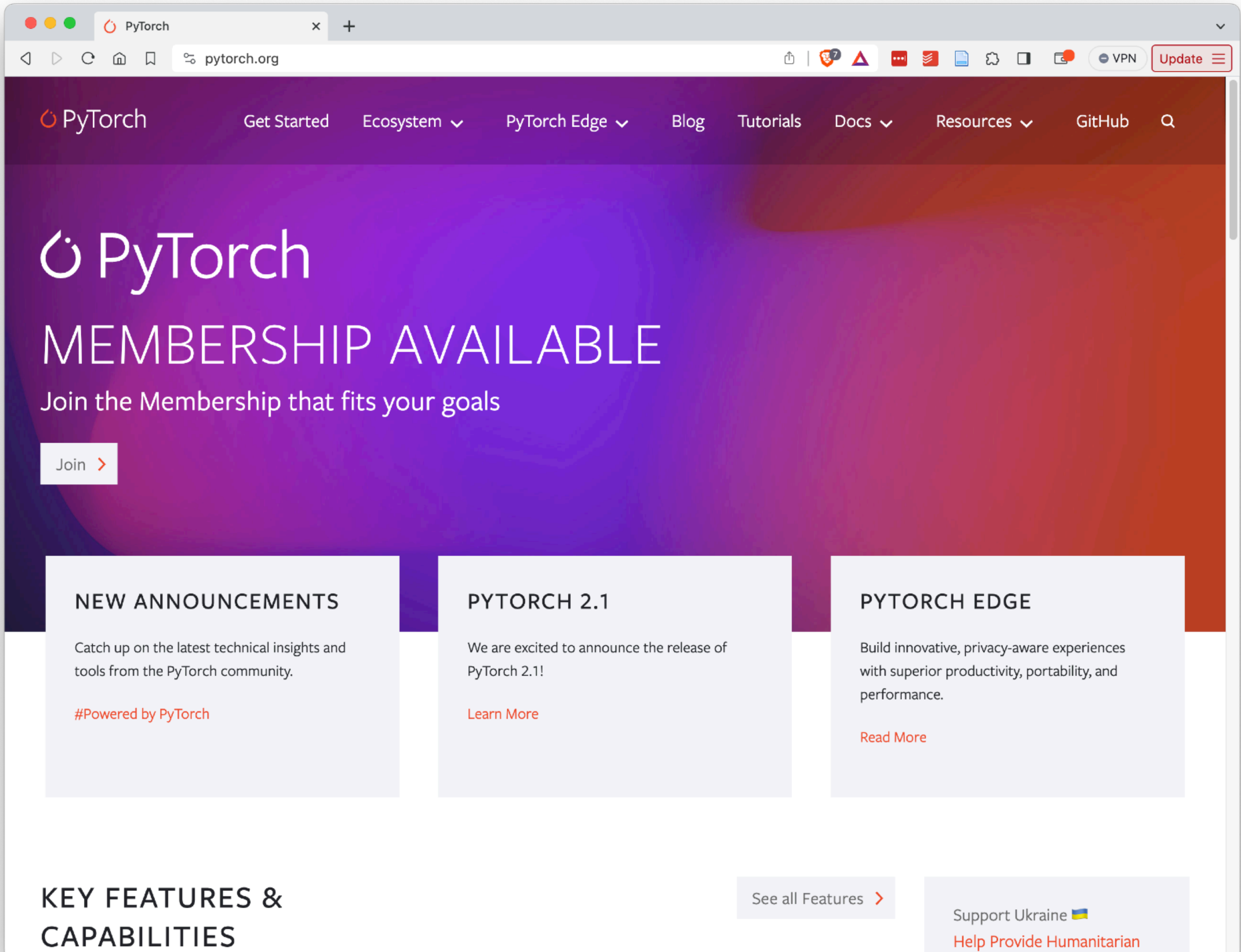
- Greedy decoding works surprisingly well for large neural models.

Beam search

- Retain only the best k sequences at each timestep.



Implementing neural models



Know your Pytorch classes

- Use Modules to define your NN.
  - Linear layer, torch.nn.functional.softmax, etc.
  - Embedding layer
- Use the CrossEntropyLoss class:
  - "This criterion computes the cross entropy loss between input logits and target."
  - Applies softmax to its inputs and then computes cross-entropy loss against the gold label.

Hyperparameters

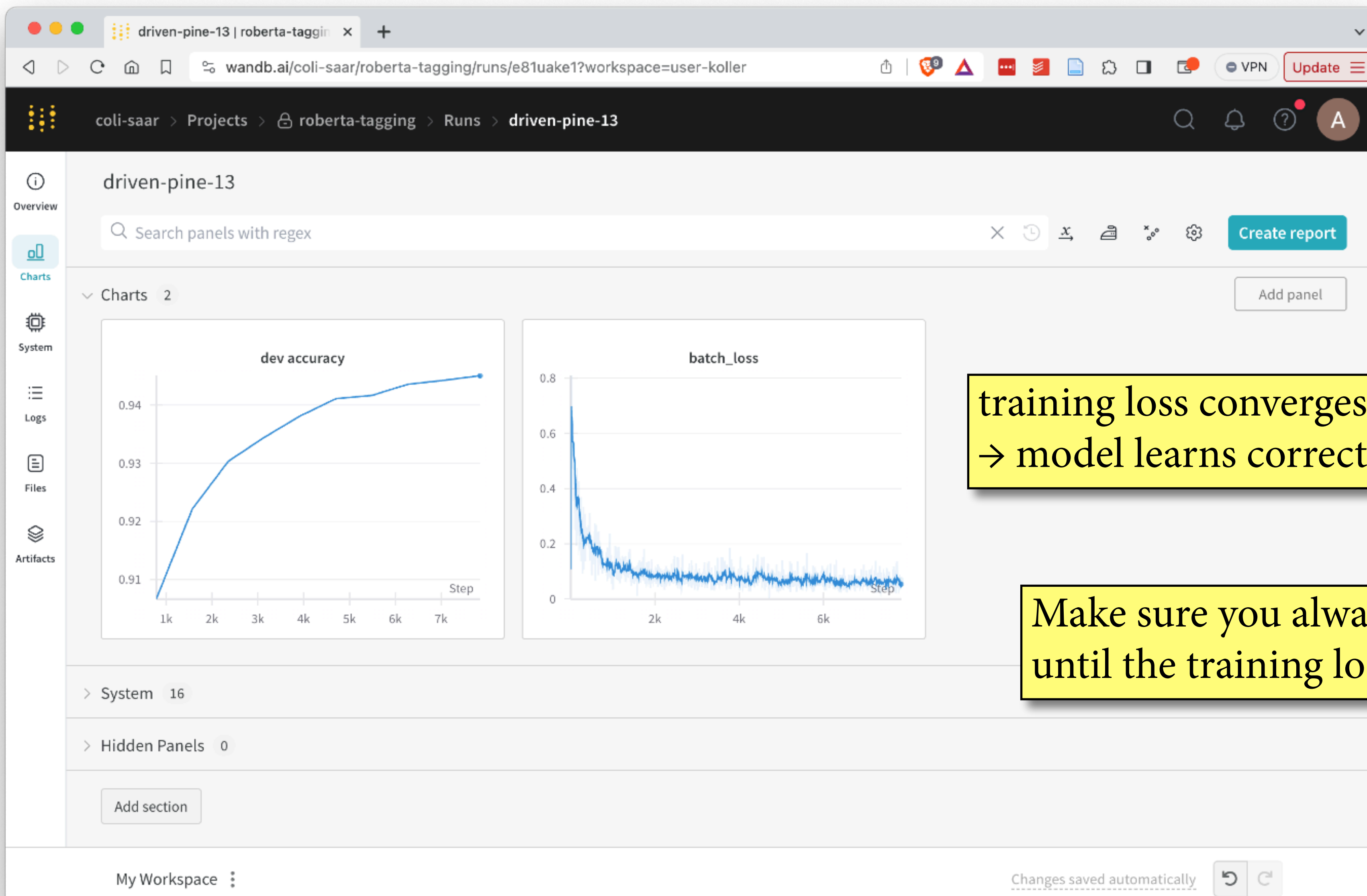
- The *parameters* of an NN are the weight matrices; these are learned from training data.
- There are also *hyperparameters*, which determine the model structure and the training dynamics, e.g.:
  - *dimensions* of the vector representations and weight matrices
  - choice of *nonlinearities*
  - *learning rate* and *optimization algorithm*
  - *overfitting* prevention methods, e.g. dropout, regularization
- Optimize hyperparameters on development dataset to find values that give you good results.

Developing neural networks

- Step 1: Implement data loading and confirm that it works.
- Step 2: Implement basic neural model + training loop and check on a reduced dataset that it works.
- Step 3: Train and evaluate on real dataset and improve.
  - bug fixing
  - model improvements
  - hyperparameter optimization

Learning curves

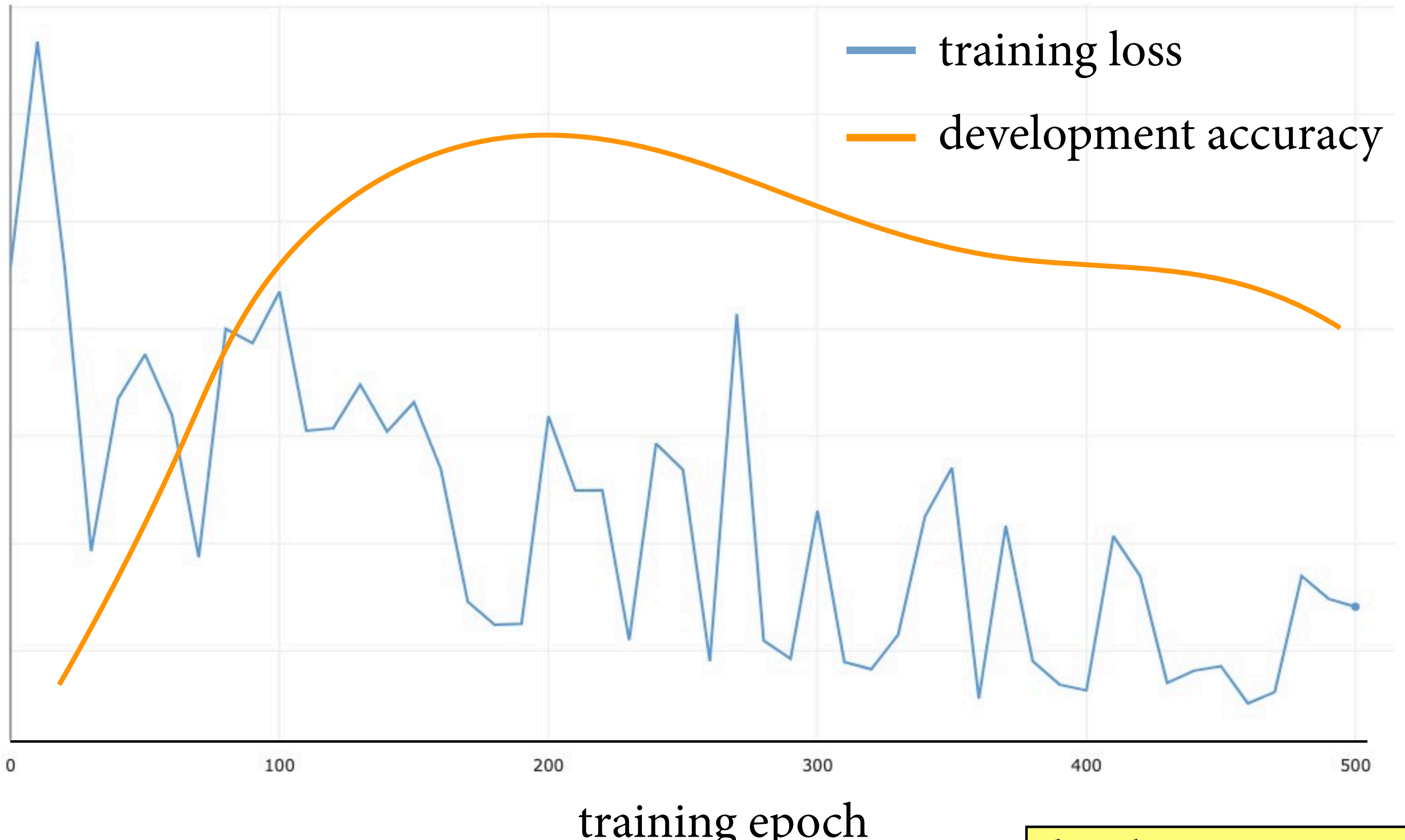
- You must, must, must visualize your learning progress to understand how your model behaves.
  - Online tools like wandb.ai make this easy and convenient.



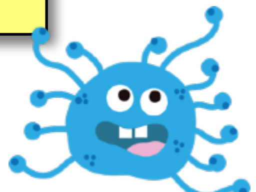
training loss converges to zero
→ model learns correctly from training data

Make sure you always train your models until the training loss stabilizes.

Diagnostics



development accuracy deterioriates while training loss improves further → overfitting



see also: https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

Where to go from here

- Go to tutorials to learn how to implement a neural network with Pytorch.
- Work through online tutorials for Pytorch and Huggingface transformers.
- Please ask lots of questions on Moodle.