Recurrent Neural Networks and Advanced Architectures for NLP

Jelke Bloem & Giovanni Colavizza

Text Mining
Amsterdam University College

April 8, 2024

Announcements

- Assignment 3 deadline: Tuesday 9th of April
- BERT reading assignment: Fri 12th of April
- Start talking about projects after the break
- Project update 0: Fri 12th of April

Overview

Questions/discussion Transformers reading assignment

- Does the Transformer yield better results translating certain languages with another just like the neural network models; can a certain model be more accurate in a certain language combination and what would the reason for that be linguistically ?
- The transformer was designed and tested for the English-French and English-German translation tasks, languages which, to some extent share many commonalities, such as characters. Can the same algorithm also be applied to perform a translation tasks for different characters such as English-Chinese?
- Thinking that some languages are more closely related, eg. languages that originate from Latin and share common word origins, can we say that the Transformer would perform better for some translation models than others, if those share a greater similarity in word origins?

Questions/discussion Transformers reading assignment

- what are the downsides of self-attention transformer models?
- What other uses have already been found and implemented for transformer models, other than text processing?
- Use in other contexts? Surely OpenAI's new video generator utilises self-attention extensively? Would it be possible to learn any more about how this may work?
- But then why is this not yet a big topic, or is is something not yet really implemented in bachelor level data science courses?
- With the transformers using self-attention, does that mean that the translations are being done/could be done in an interpreting fashion rather than a strict translating one?

Questions/discussion Transformers reading assignment

- How would a Transformer deal with translating a whole book? Are there any available models that can deal with this type of task as good as (or even better than) humans do?
- The paper mentions that the hyperparameters of the Transformer model were extensively fine-tuned. But in class we have learned that, once we apply a model on our test data, we cannot test it on the same data again. How did the authors manage to find enough data to train it so extensively, and did they reuse some of it?
- Why weren't transformers explored earlier?
- How did they made a whole model out of a single mechanism which is otherwise only used for upgrading an existing model?

Neural Language Models

Recap

With language models, we want to compute:

- The probability of a sequence of words: $P(w) = P(w_1, w_2, .., w_n)$.
- The probability of a new word given what came before it: $P(w_n|w_1, w_2, .., w_{n-1})$.
- We have also seen n-gram language models which make use of the Markov assumption. For example, a trigram language model would predict the probability of a sequence of words by conditioning on the two previous words at each step:

$$P(w) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2) \dots P(w_n|w_{n-2}, w_{n-1})$$

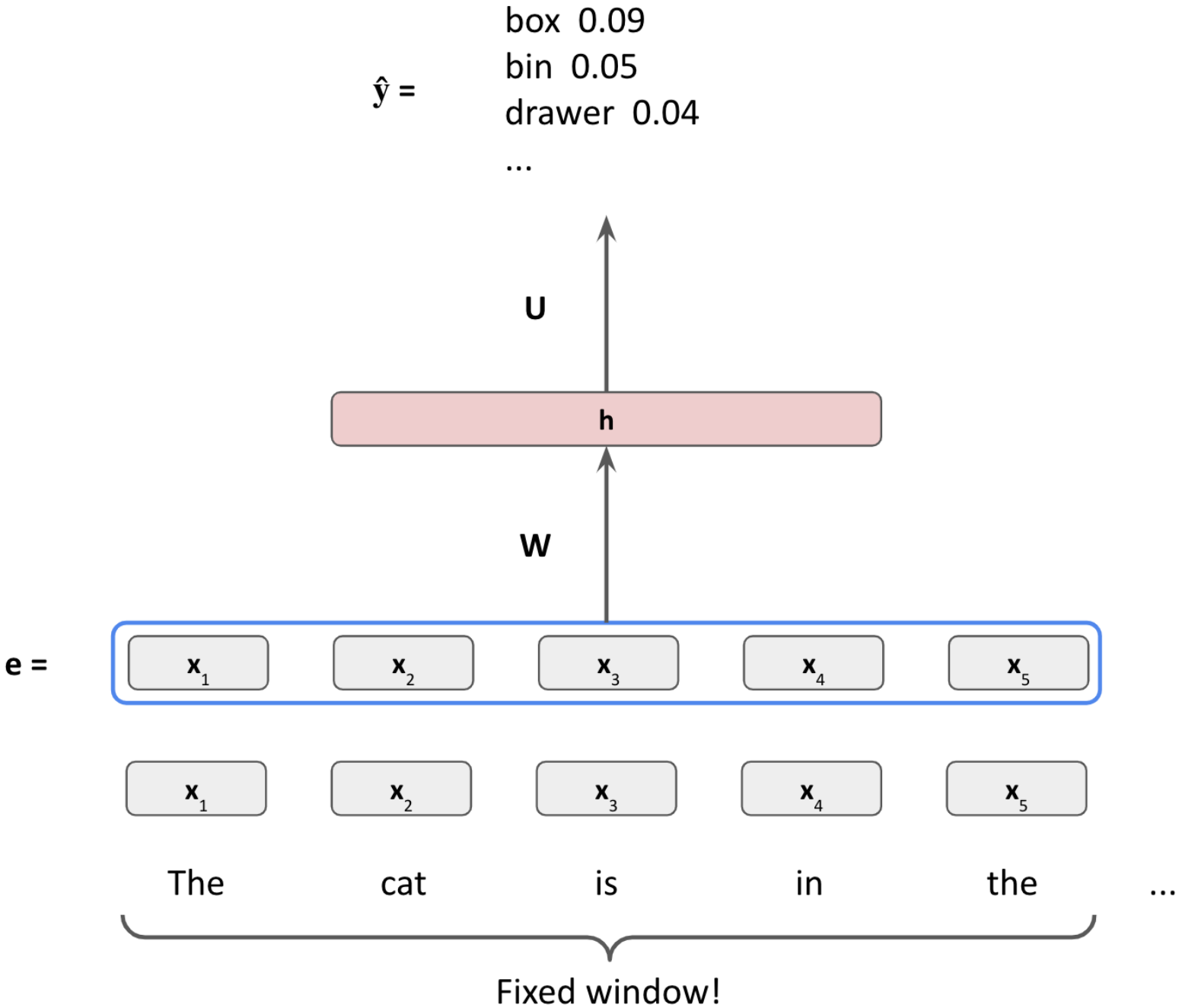
- n-gram language models have some issues, including sparsity and limited use of context.

A first neural language model

We can start by using word embeddings as features for a **fixed-window neural language model**:

- Given a fixed-window sequence of words represented using their embeddings $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t$;
- we are interested in estimating the probability of the next word $\hat{p}(\boldsymbol{x}_{t+1}|\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t)$.
- We can use the previous words' embeddings as features, concatenate them and feed them to a hidden layer with a non-linearity, and concluding by estimating probabilities with a softmax.

A first neural language model



A first neural language model

Where:

- Each $\boldsymbol{x}$ is a $1 \times d$ embedding vector of dimensionality d.
- $\boldsymbol{e} = [\boldsymbol{x}_1; \boldsymbol{x}_2; \ldots; \boldsymbol{x}_t]$ is the concatenation of the embeddings of the words in the fixed-window preceding w_{t+1}. Therefore $\boldsymbol{e}$ has dimensionality $1 \times dk$, where k is the number of words in the fixed-window.
- $\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e}^T)$ is the hidden layer, with f an appropriate activation function and $\boldsymbol{W}$ a weight matrix. $\boldsymbol{W}$ has dimensionality $dk \times h$ and $\boldsymbol{h}$ has dimensionality $1 \times h$. The dimensionality of the embeddings and the hidden layer are hyperparameters of the model.
- Finally, $\hat{y} = softmax(\boldsymbol{U}\boldsymbol{h}^T)$ is the predicted next word probability as estimated using a softmax function. Here $\boldsymbol{U}$ is another weight matrix of dimensionality $|V| \times h$, so that with the softmax we predict a probability distribution over the vocabulary V.

A first neural language model

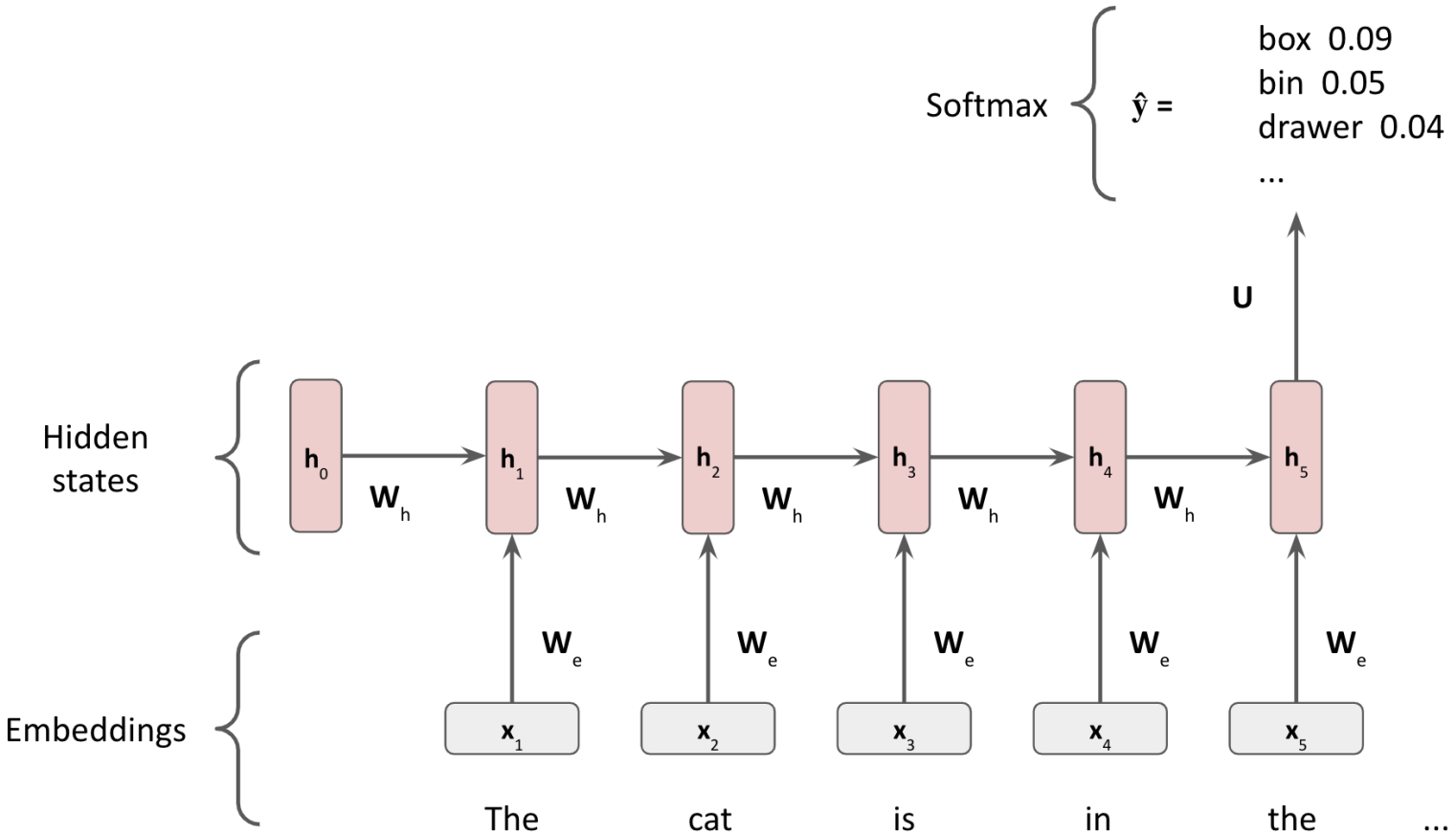
This model is interesting, yet unfortunately:

- it does not solve the use of a broader context with flexible word windows;
- larger windows would mean more parameters so scale is also an issue;
- and it does not make an efficient use of parameters and shared weights.

Can we do better? Yes, with **Recurrent Neural Networks**.

Recurrent Neural Networks

Recurrent architecture



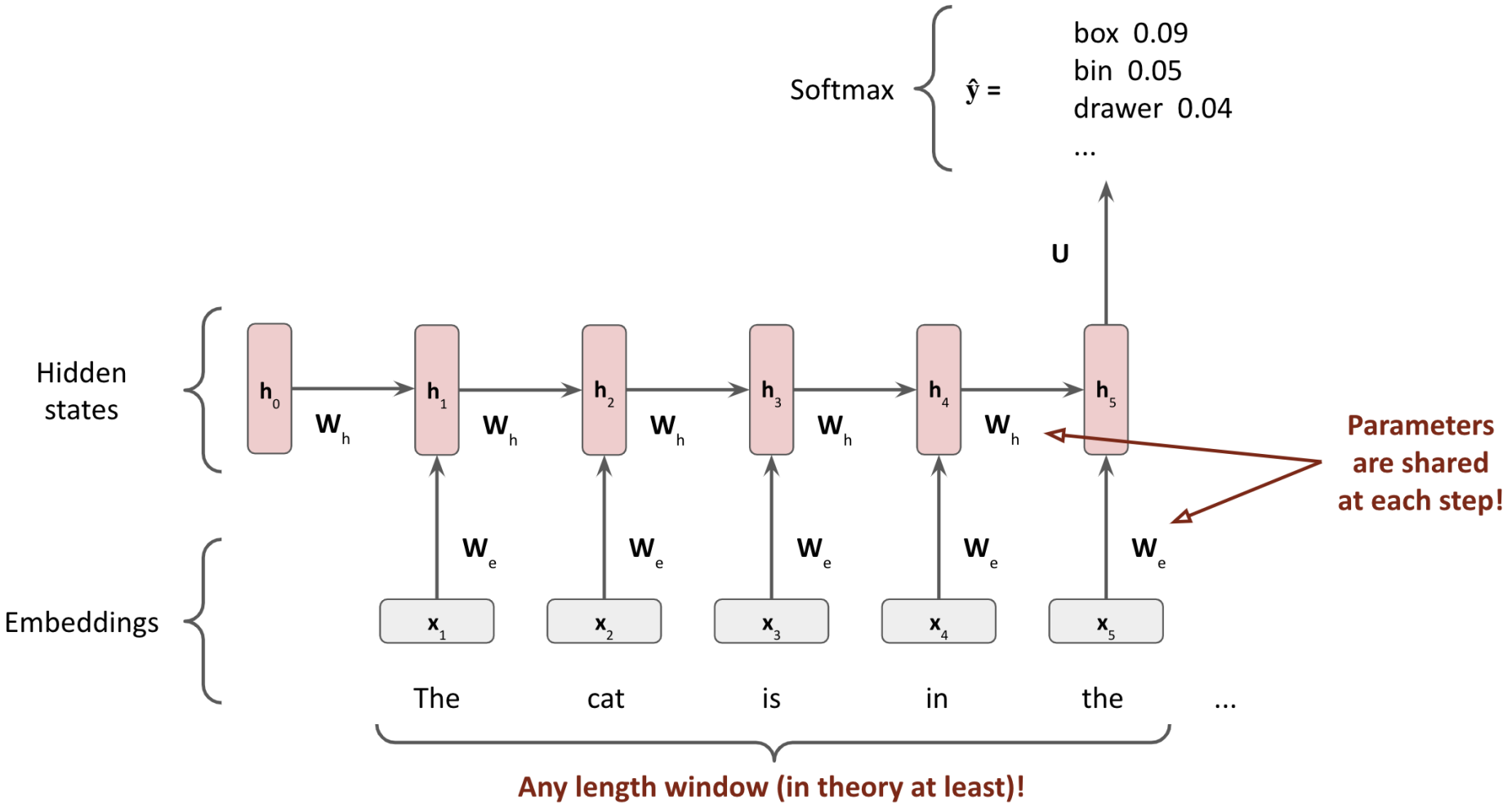
Recurrent architecture

Where:

- Each $\boldsymbol{x}$ is a $1 \times d$ embedding vector of dimensionality d.
- $\boldsymbol{h}_t = f(\boldsymbol{W}_h \boldsymbol{h}_{t-1}^T + \boldsymbol{W}_e \boldsymbol{x}_t^T)$ is the hidden layer, with f an appropriate activation function, $\boldsymbol{W}_e$ and $\boldsymbol{W}_h$ weight matrices. $\boldsymbol{W}_h$ has dimensionality $h \times h$, $\boldsymbol{W}_e$ has dimensionality $h \times d$, and $\boldsymbol{h}$ has dimensionality $1 \times h$. $\boldsymbol{h}_0$ is the initial hidden layer. The dimensionality of the embeddings and the hidden layer are hyperparameters of the model.
- Finally, $\hat{y} = softmax(\boldsymbol{U}\boldsymbol{h}^T)$ is the predicted next word probability as estimated using a softmax function. Here $\boldsymbol{U}$ is another weight matrix of dimensionality $|V| \times h$, so that with the softmax we predict a probability distribution over the vocabulary V.

(Notebook 8.1 Part II: Model)

Recurrent architecture



Why RNNs?

RNNs are a big step forward re. our previous concerns:

- Can process inputs on any length and use previous context of any length (in theory);
- model size does not depend on window size ($\boldsymbol{W}$ matrices remain of the same dimension);
- weights are shared across time steps.

Nevertheless, RNNs can be slow and won't really remember information from many steps back.

Training RNNs

How can we train RNNs?

- Predict the next word at each step, and calculate the loss accordingly.
- The loss at step t is the usual **cross-entropy** (now on multiple classes), calculated for the word to be predicted at $t+1$:

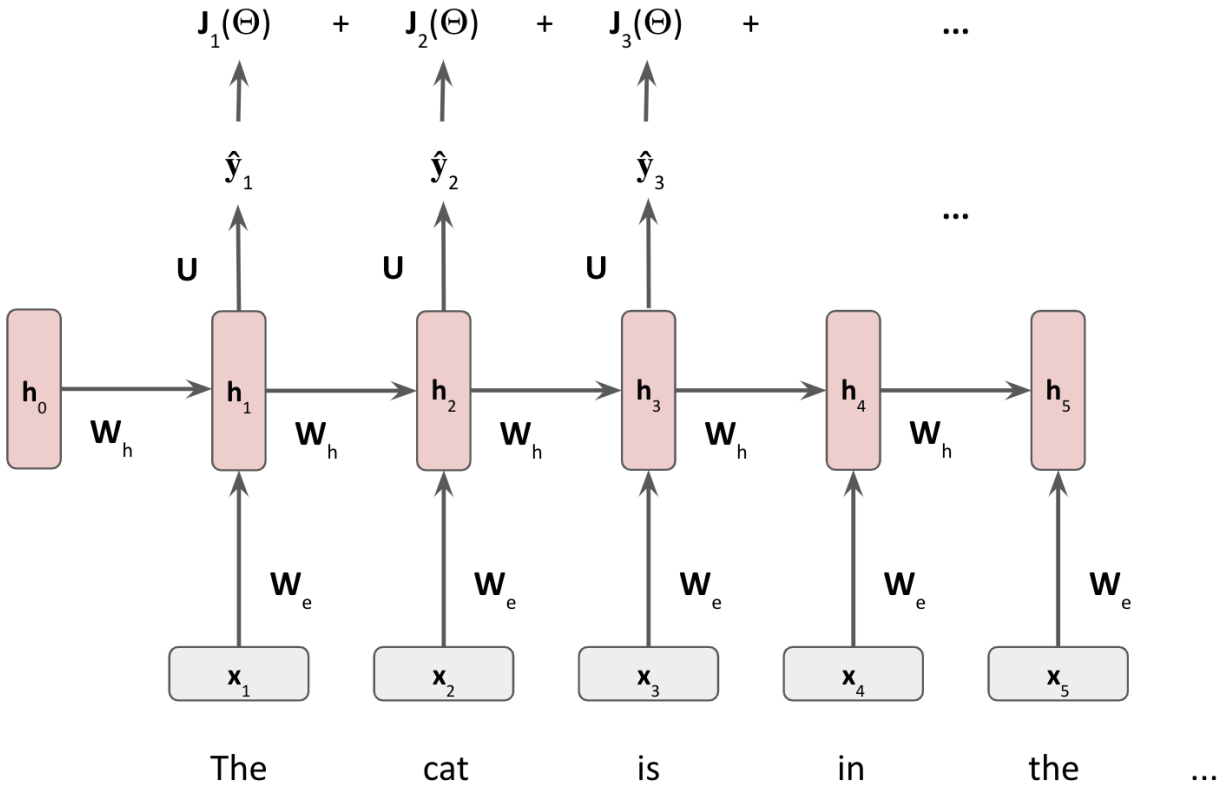
$$\mathcal{L}_t(\boldsymbol{W}) = -\sum_{w \in V} y_{w_{t+1}} log(\hat{y}_{w_{t+1}}) = -log(\hat{y}_{w_{t+1}})$$

- The overall loss is the average of the sum of the losses, calculated at each step:

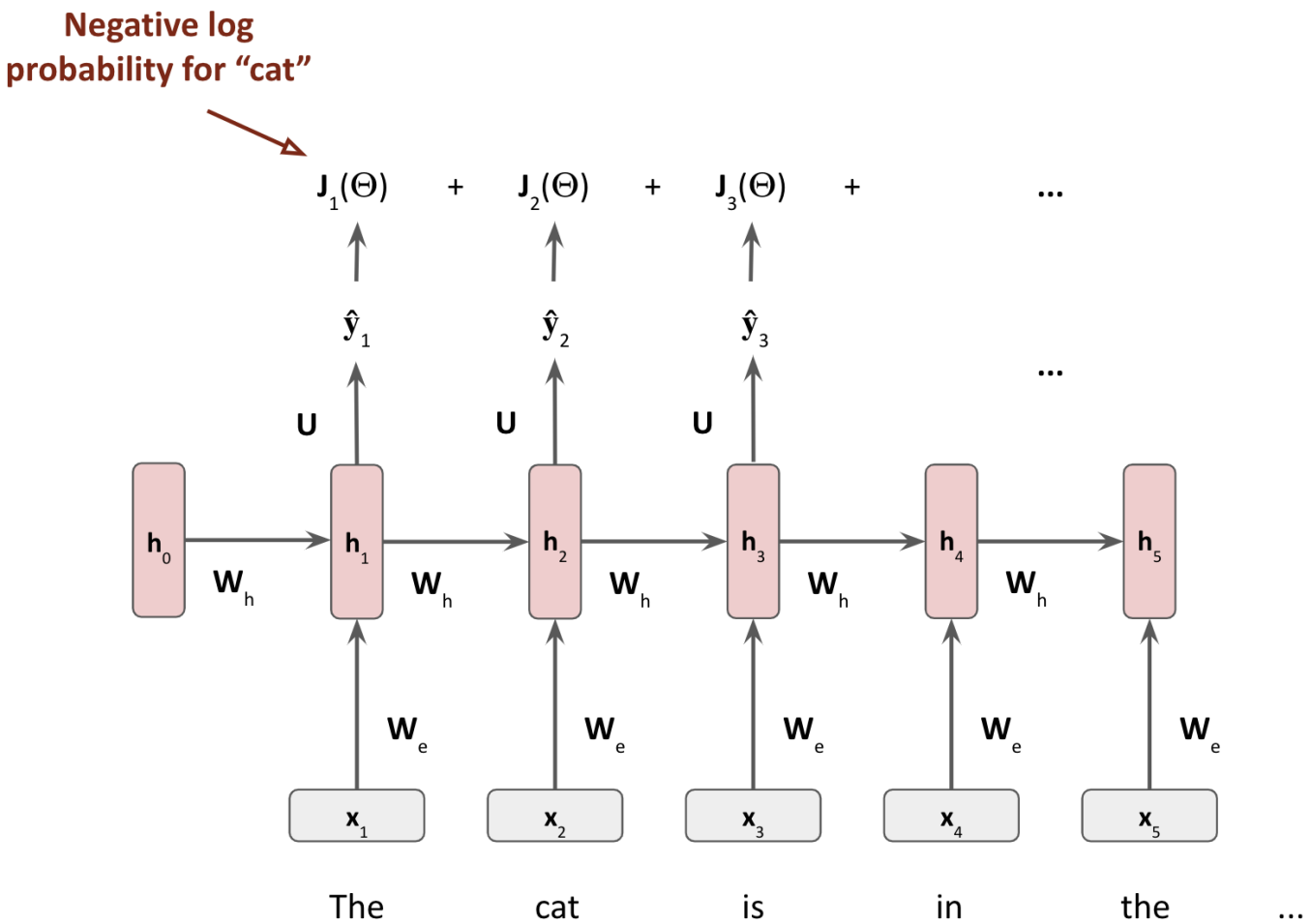
$$\mathcal{L}(\boldsymbol{W}) = \frac{1}{T}\sum_{t=1}^{T} \mathcal{L}_t(\boldsymbol{W}) = \frac{1}{T}\sum_{t=1}^{T} -log(\hat{y}_{w_{t+1}})$$

- Optimization can be done via SGD (backpropagation). Since the parameters $\boldsymbol{W}$ are used repeatedly, this is sometimes called **backpropagation through time**.

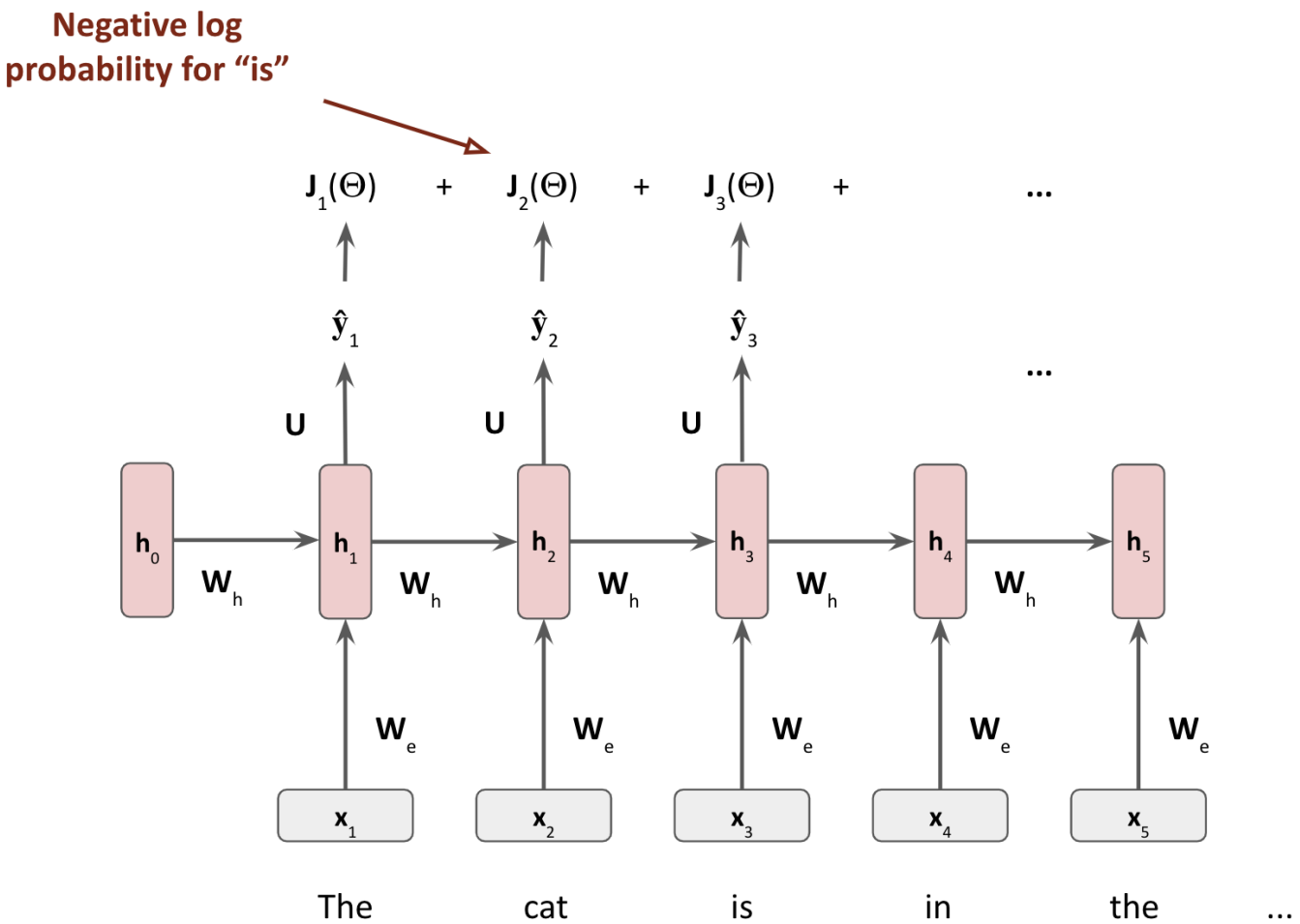
Training RNNs



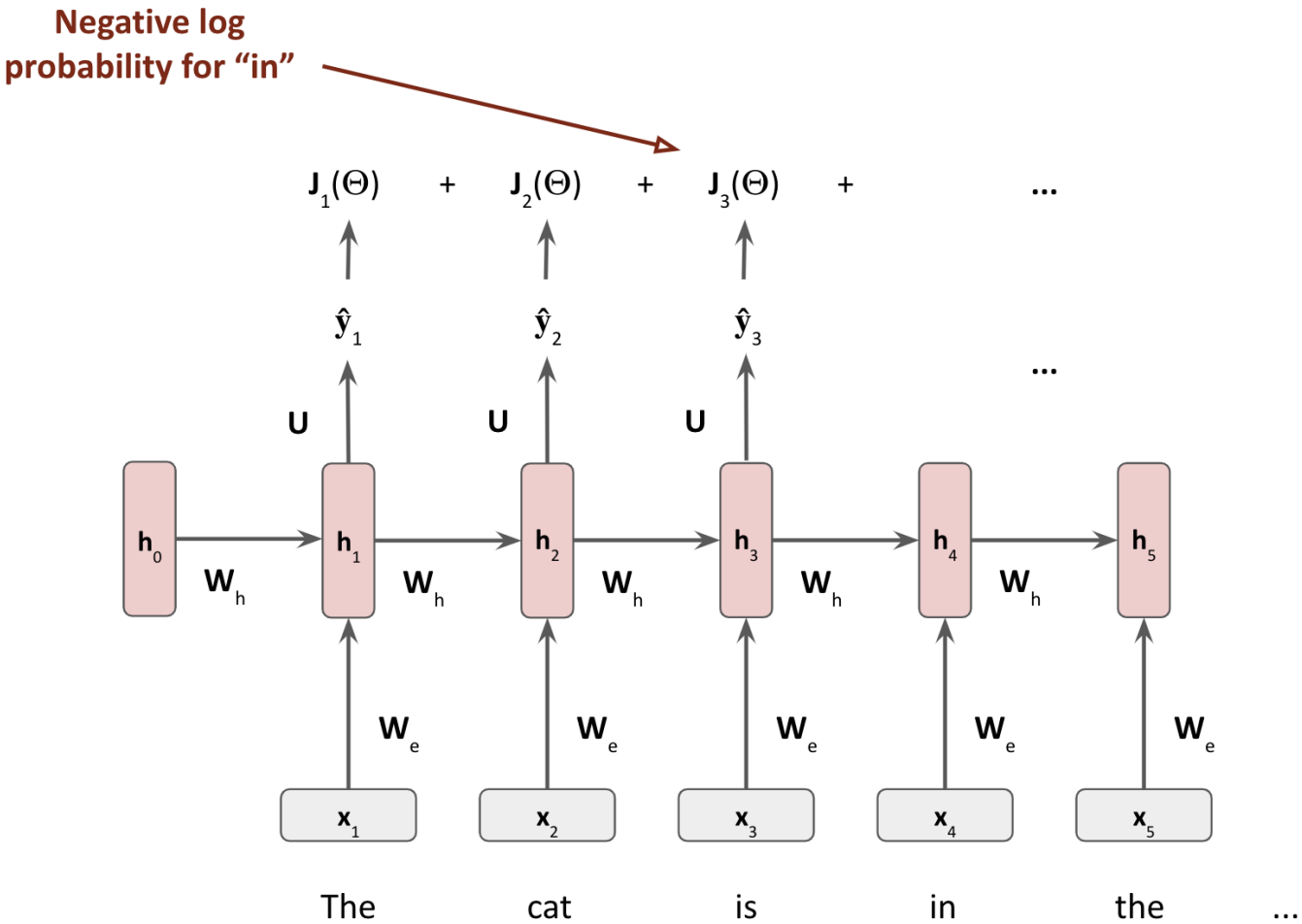
Training RNNs



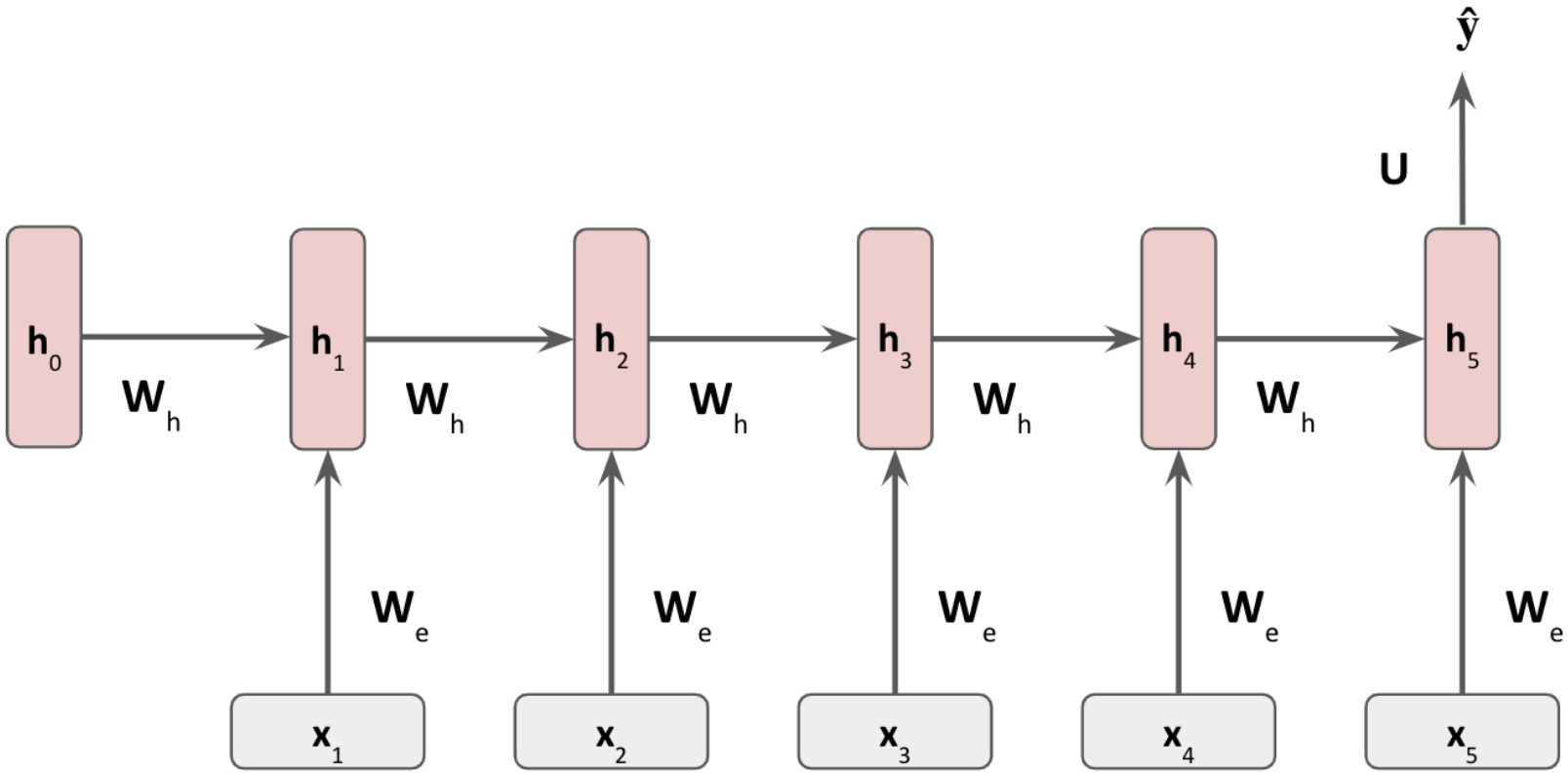
Training RNNs



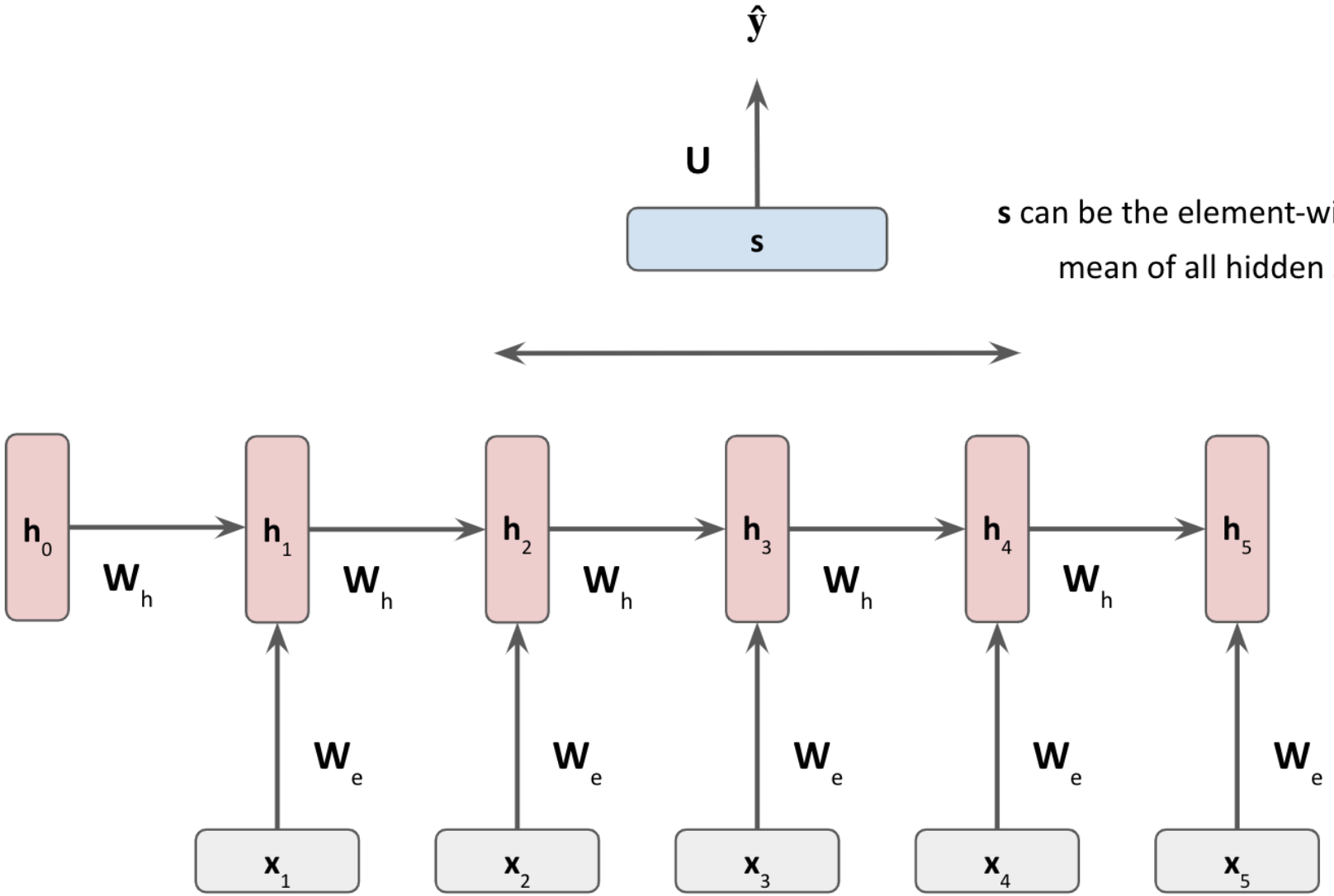
Training RNNs



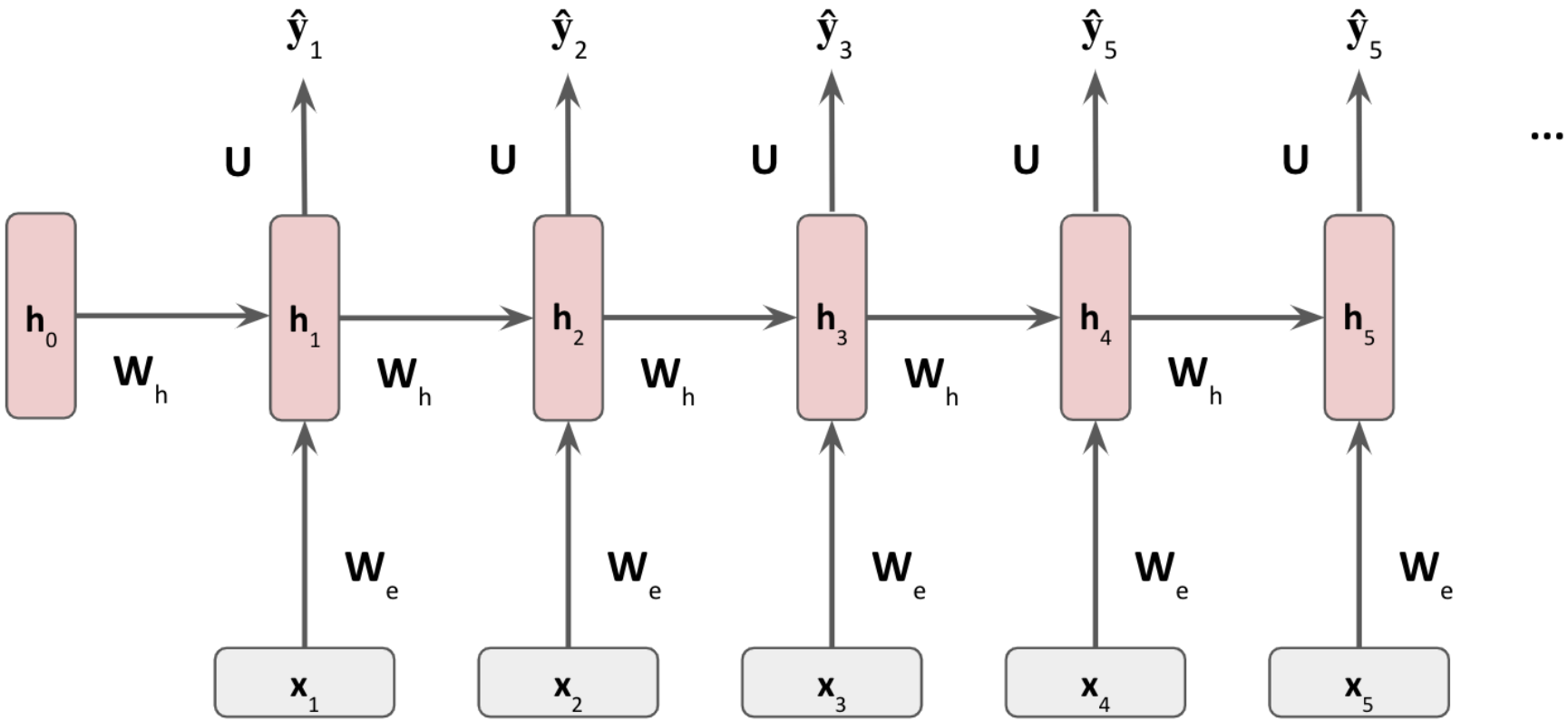
RNN flavors: Many to one



RNN flavors: Many to one



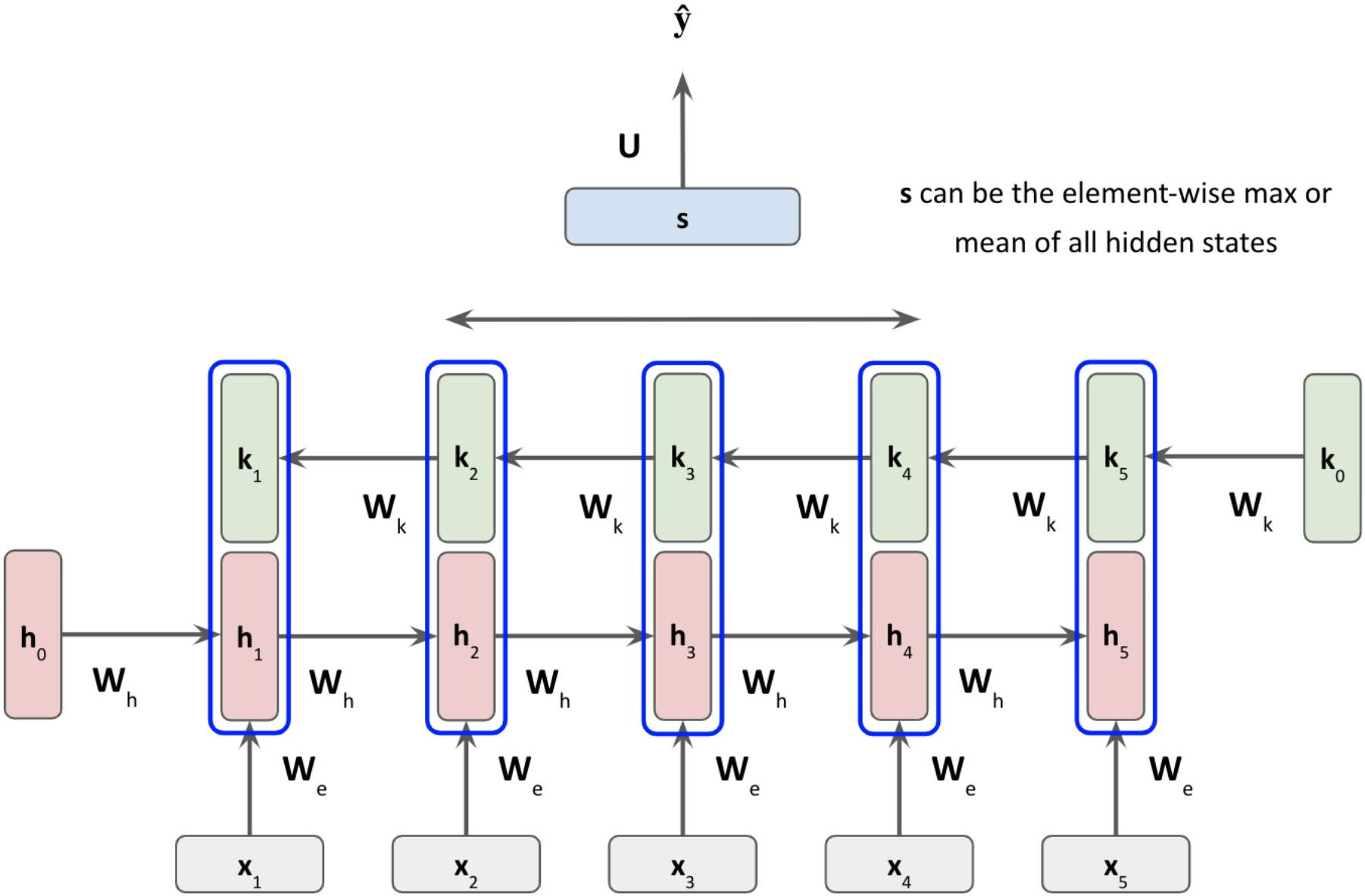
RNN flavors: Many to many



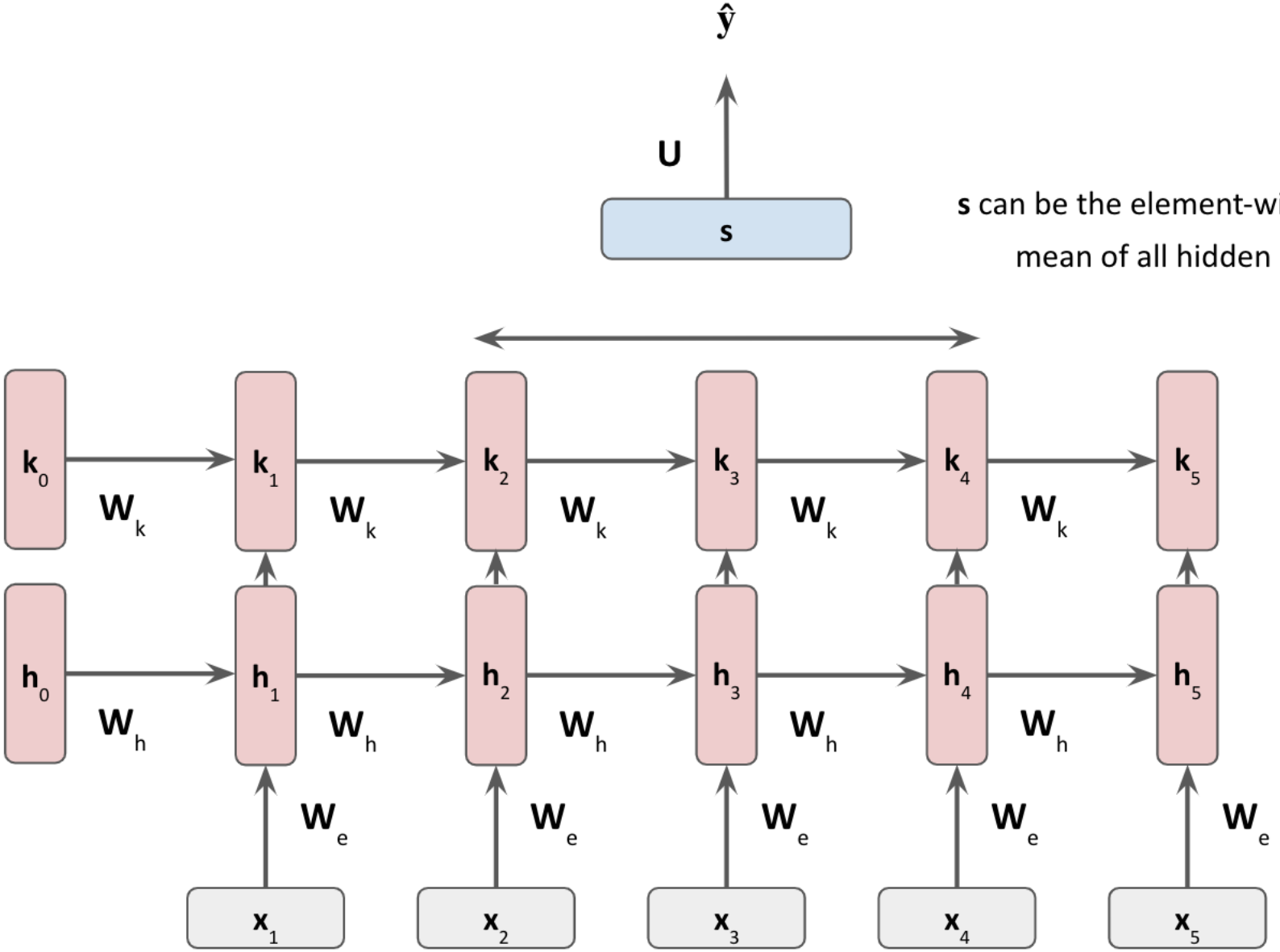
Notes

Fancy RNNs

Bi-RNNs

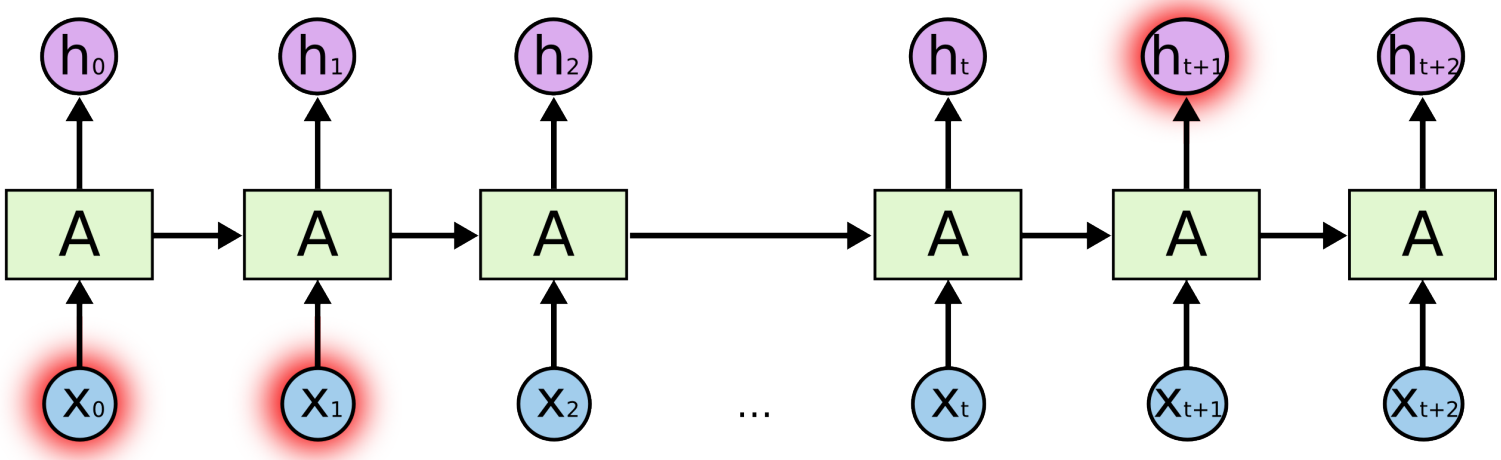

ŷ
U
s
s can be the element-wise max or mean of all hidden states
k1
k2
k3
k4
k5
k0
W_k
h0
h1
h2
h3
h4
h5
W_h
W_e
x1
x2
x3
x4
x5

Multilayer RNNs



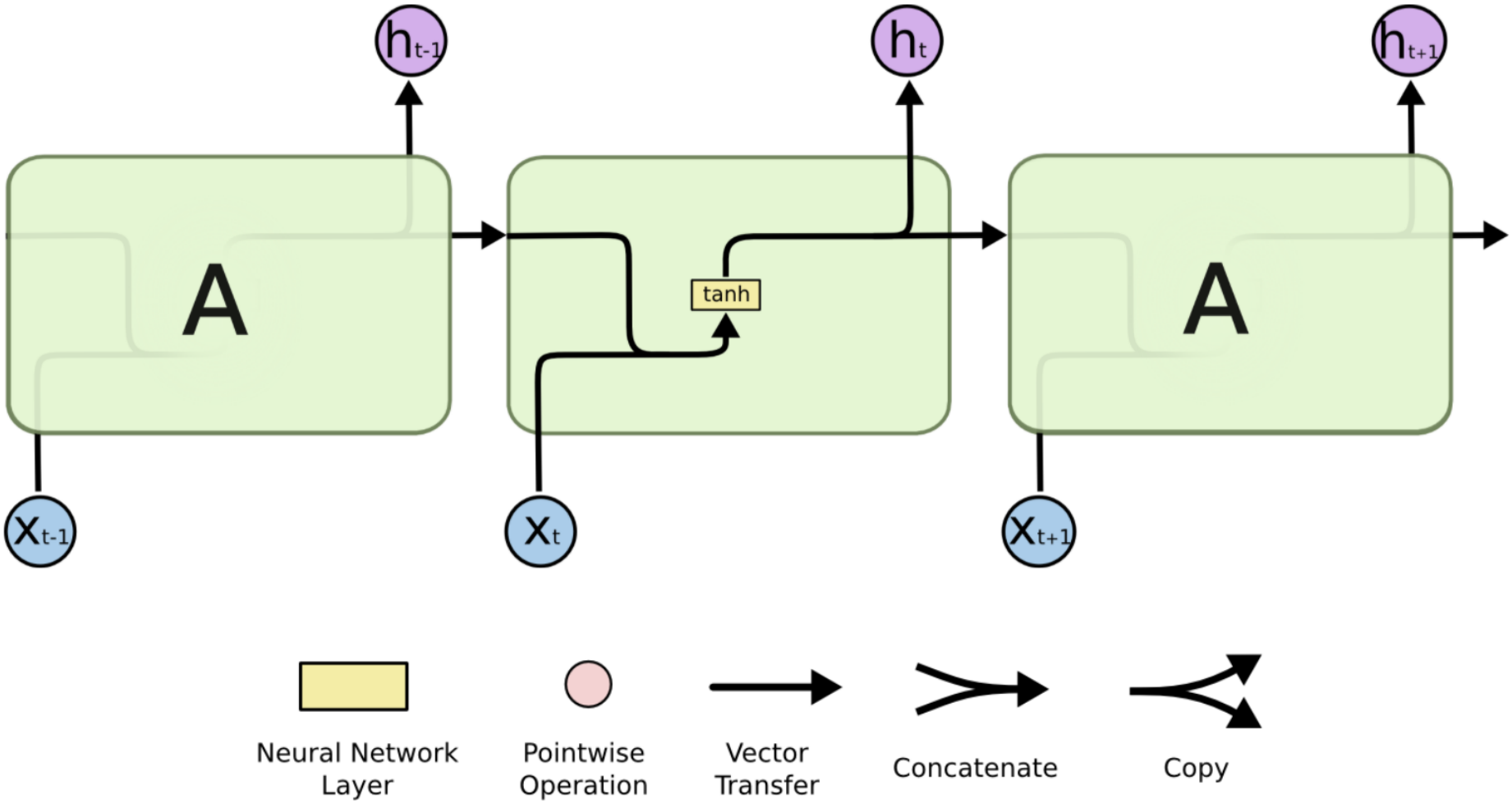
Vanishing Gradients

- RNNs have a crucial issue: **vanishing and exploding gradients**.
- Both occur when we backpropagate through time with multiplying several times by $\boldsymbol{W}$. If $\boldsymbol{W}$'s parameters are small, gradients can vanish to zero. If they are large, they can explode.
- This is an issue as it does not allow to model far away context (vanishing) or to properly converge (exploding).
- Solutions:
  1. Exploding: **gradient clipping**.
  2. Vanishing: **Long Short-Term Memory** networks.



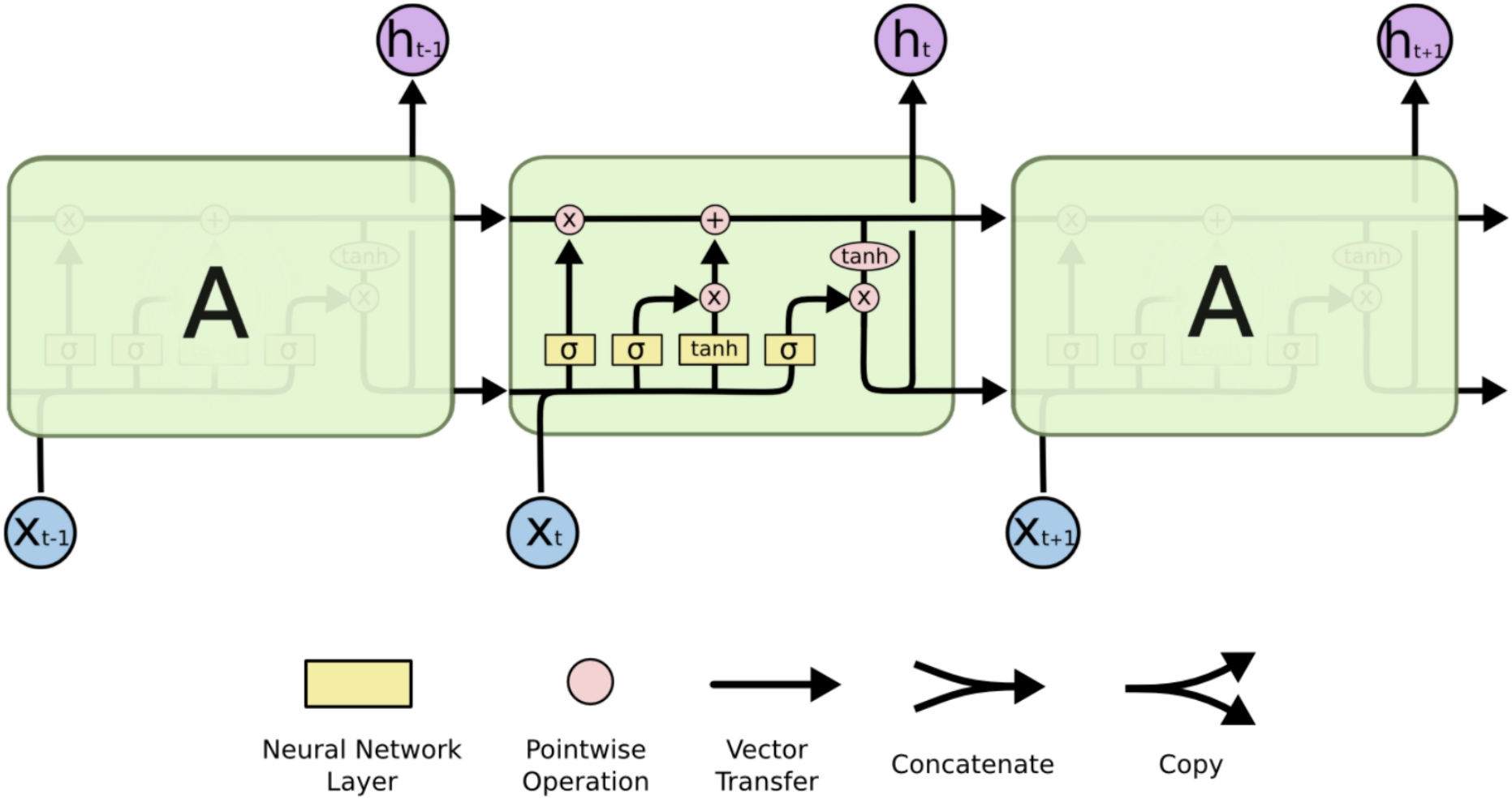
Credit: https://colah.github.io/posts/2015-08-Understanding-LSTMs

A different view on RNNs



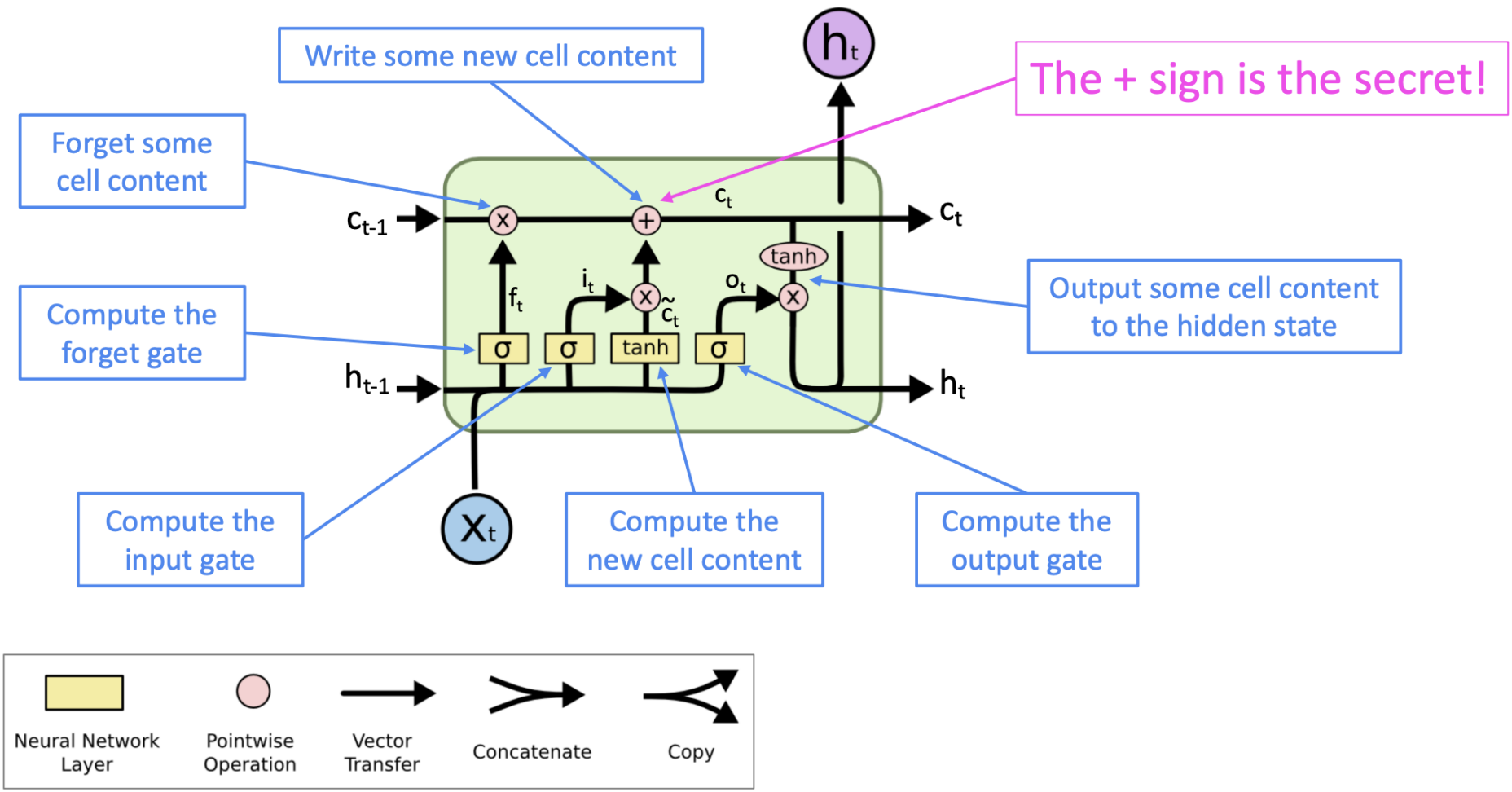
Credit: https://colah.github.io/posts/2015-08-Understanding-LSTMs

LSTMs



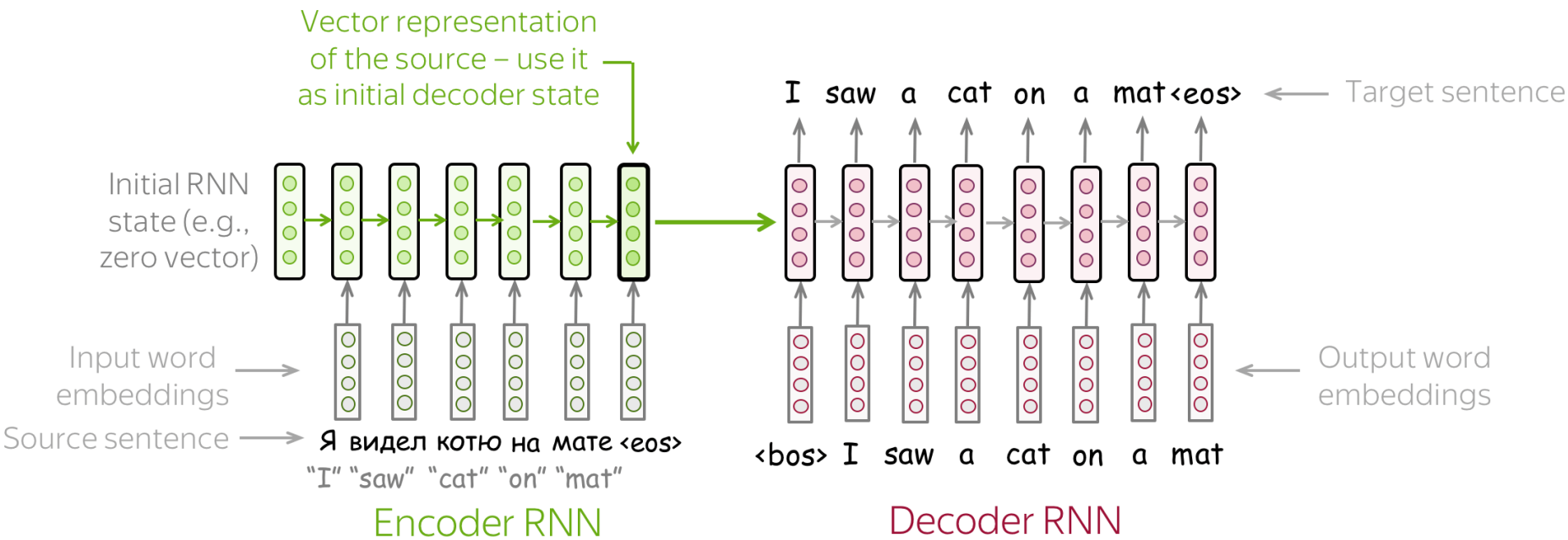
Credit: https://colah.github.io/posts/2015-08-Understanding-LSTMs

LSTMs



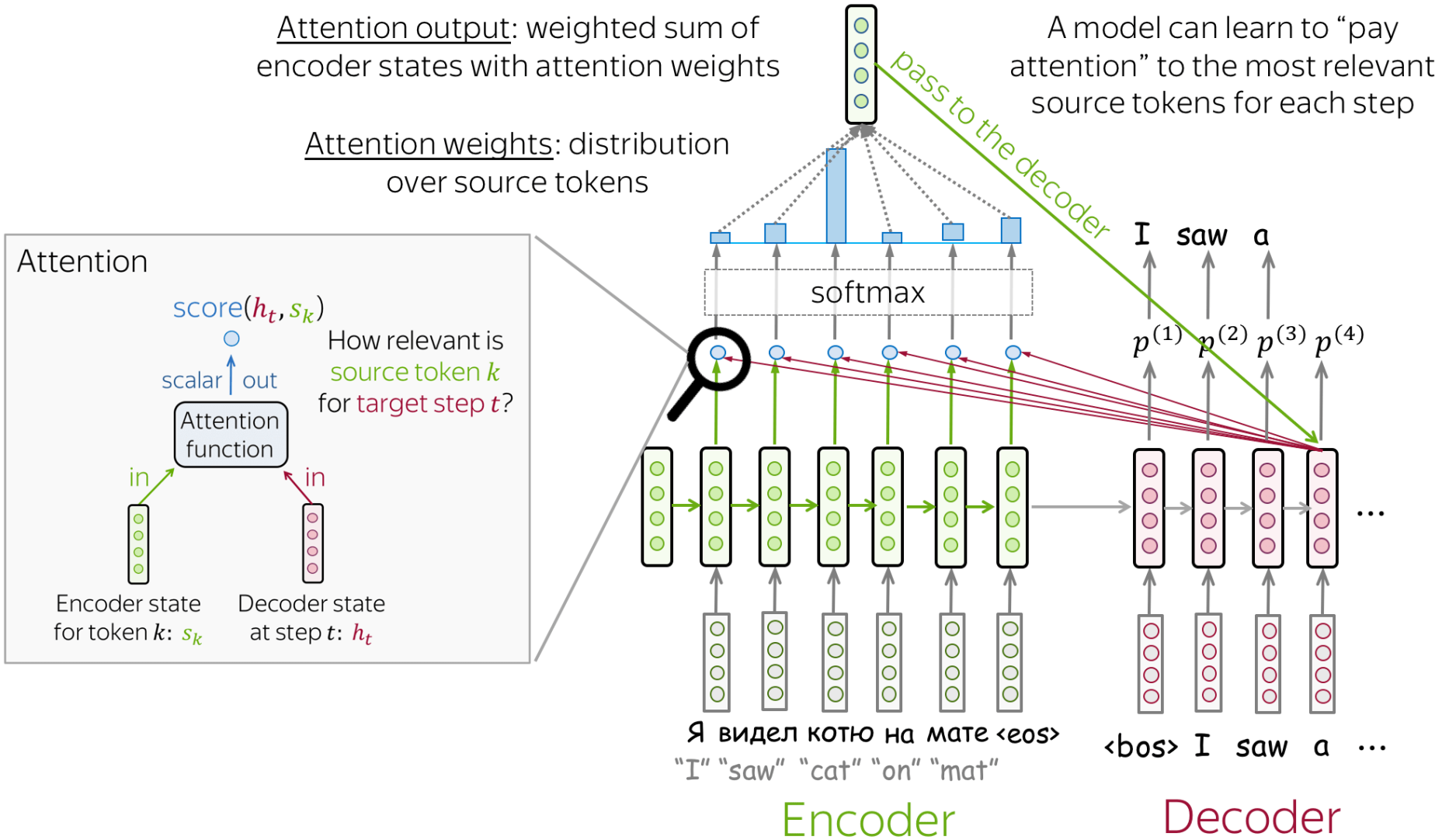
Credit: Stanford CS224N

Seq2Seq



Credit: Lena Voita `https://lena-voita.github.io/nlp_course.html`

Attention



Credit: Lena Voita `https://lena-voita.github.io/nlp_course.html`

What's next

- The most recent advances in neural networks for NLP come from the shift from recurrent architectures to using **attention-based architectures**.
- For your reading assignments, you have explored **transformers** (which combine attention with other ideas from neural networks literature) and will explore **BERT** (which is a neural language model making use of transformers).
- While there is much more to it, in this way you will have a window into contemporary models for NLP.

The next part of the course turns to other topics instead: Web scraping and APIs, recommender systems, corpus annotation, sentiment analysis and clustering with topic modelling, ethics.

Notes

Recall perplexity?

A second look at perplexity:

- We defined it as the inverse probability of the corpus, normalized by the number of words. For a corpus composed of n words:

$$PP = P(w_1, w_2, \ldots, w_n)^{-\frac{1}{n}} = \prod_{i=1}^{n} \left(\frac{1}{P(w_i | w_1, \ldots, w_{i-1})} \right)^{\frac{1}{n}}$$

- It is actually equal to the exponential of the cross-entropy loss:

$$PP = \prod_{i=1}^{n} \left(\frac{1}{\hat{y}_{w_{i+1}}} \right)^{\frac{1}{n}} = exp\left(\frac{1}{n} \sum_{i=1}^{n} -log(\hat{y}_{w_{i+1}}) \right) = exp(\mathcal{L})$$

- So *lower perplexity == lower loss == higher data likelihood*.