

**Deep Learning**  
Summer semester '24

---



## 6. Recurrent Neural Networks

# Data Science Research at Chair of Computer Science

Digital Humanities

Social-Media-Analysis

Structured Knowledge (KG)

**TEXT ANALYSIS &  
KNOWLEDGE GRAPHS**

**ENVIRONMENTAL  
DATA SCIENCE**

Estimation of air quality

Analysis of bee behavior

Climate models



**Knowledge Enriched  
Natural Language  
Processing**

**DEEP LEARNING**



**Deep Learning for  
Dynamical Systems**

**RECOMMENDER  
SYSTEMS**

**AI-SECURITY &  
FRAUD DETECTION**

Product recommendations

Supporting medical diagnosis

User analysis and modelling



**Recommendation  
and Security**

Explainable AI

Detection of hacker attacks

Fraud detection in ERP systems

# Content of this Chapter

---

1. Recurrent Neural Networks in Theory
  1. Vanilla RNNs
  2. Backpropagation Through Time (BPTT)
  3. The Long Term: LSTMs and Friends
2. Sequence to Sequence
3. Attention

# 5.1 Recurrent Neural Networks in Theory

---

- Vanilla RNNs
- Backpropagation Through Time (BPTT)
- LSTM and friends

# Character-based Text Generation

- Idea:
  - Given a sequence of characters  $d = (d_1, d_2, \dots, d_n)$  of length  $n$ . Learn a model  $M$  that predicts the next character in the sequence,  $d_{n+1}$
- Example:
  - „The Dursleys had everything they wanted, ...“
  - $M(\text{„T“}) = \text{„h“}$
  - $M(\text{„Th“}) = \text{„e“}$
  - $M(\text{„The Dursleys ha“}) = \text{„d“}$



Train as a classification problem with samples  $((d_1, d_2, \dots, d_k), d_{k+1})$

How can we process sequences?

# Character-based Text Generation (FCN)

- Remember a fully connected (FCN) layer is defined as:

$$y = f(Wx + b)$$

- The layer takes only one input  $x$ , but we have a sequence of characters!

- Possible Solutions:

- Concatenate the individual character vectors:  $x = \text{concat}(d_1, d_2, \dots, d_k)$

- We already saw that this leads to large weight matrices!
- Furthermore, we can not process sequences of arbitrary length ☹

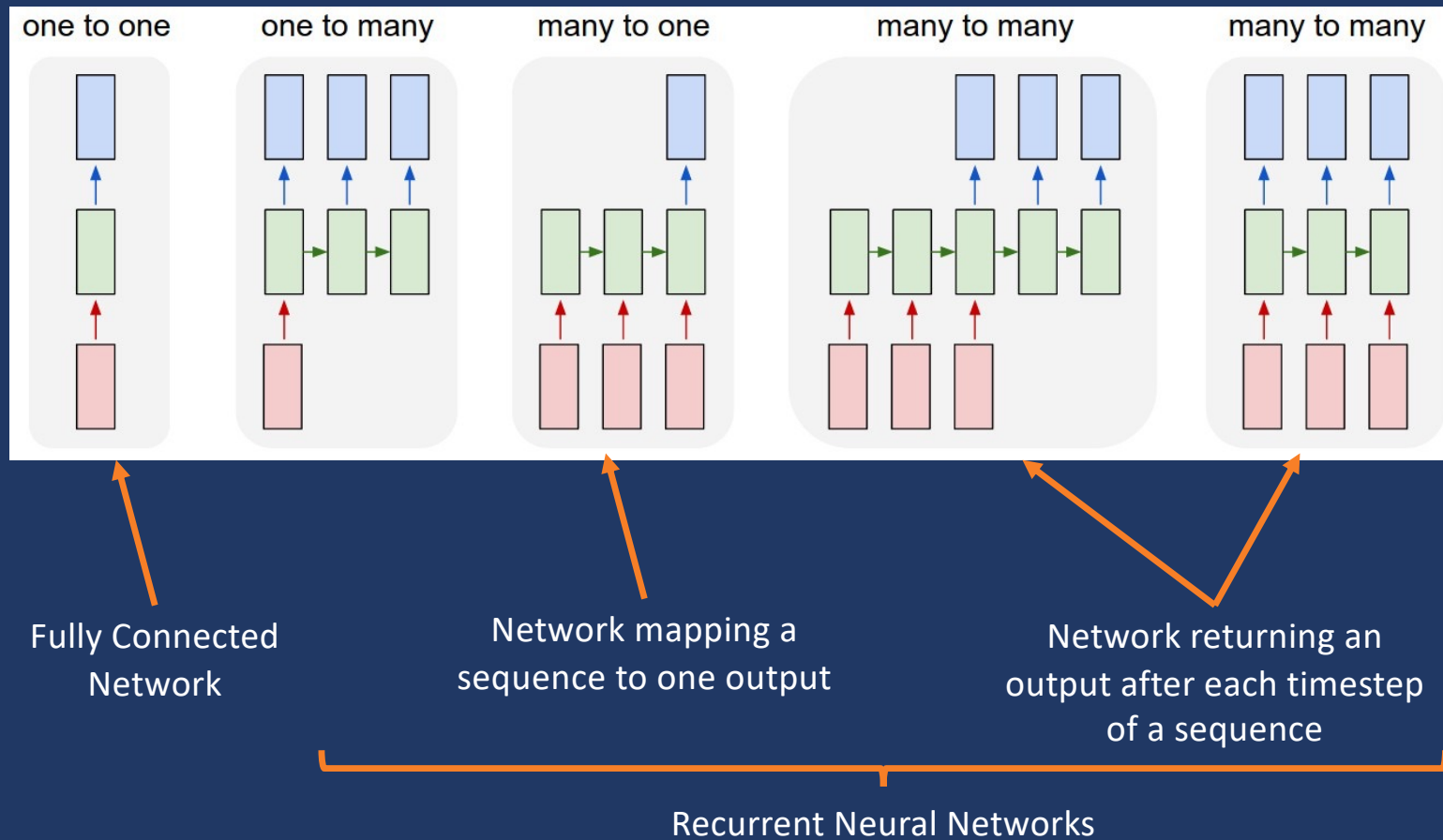
- Average over the individual character vectors:  $x = \frac{1}{k} \sum_{1 \leq i \leq k} d_i$

- Input vector can be kept small
- Sequences of arbitrary length are possible
- All structure is lost ☹

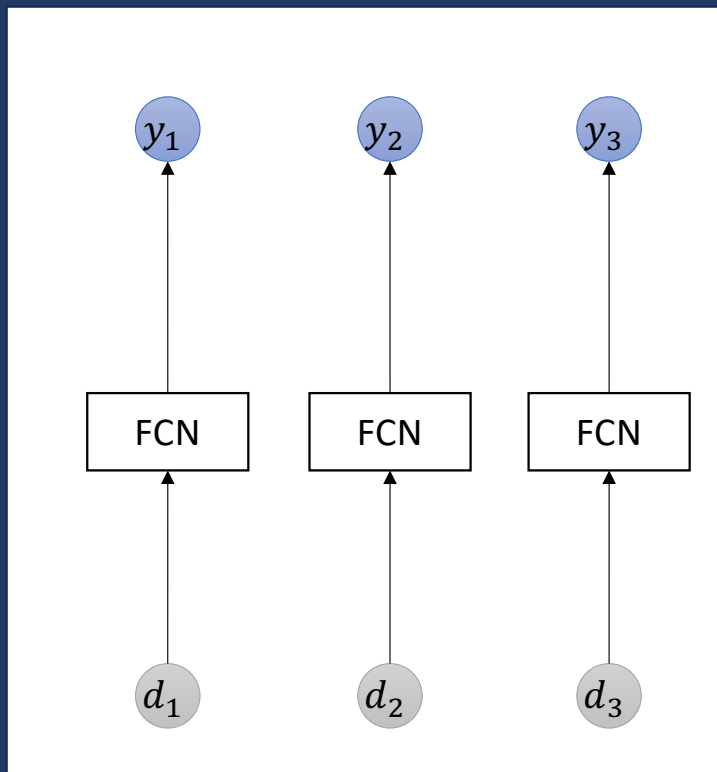
- ...?

→ How to preserve structure, yet process sequences of arbitrary length?

# Overview: Types of Neural Networks for all Purposes



# Character-based Text Generation: From FCN to RNN

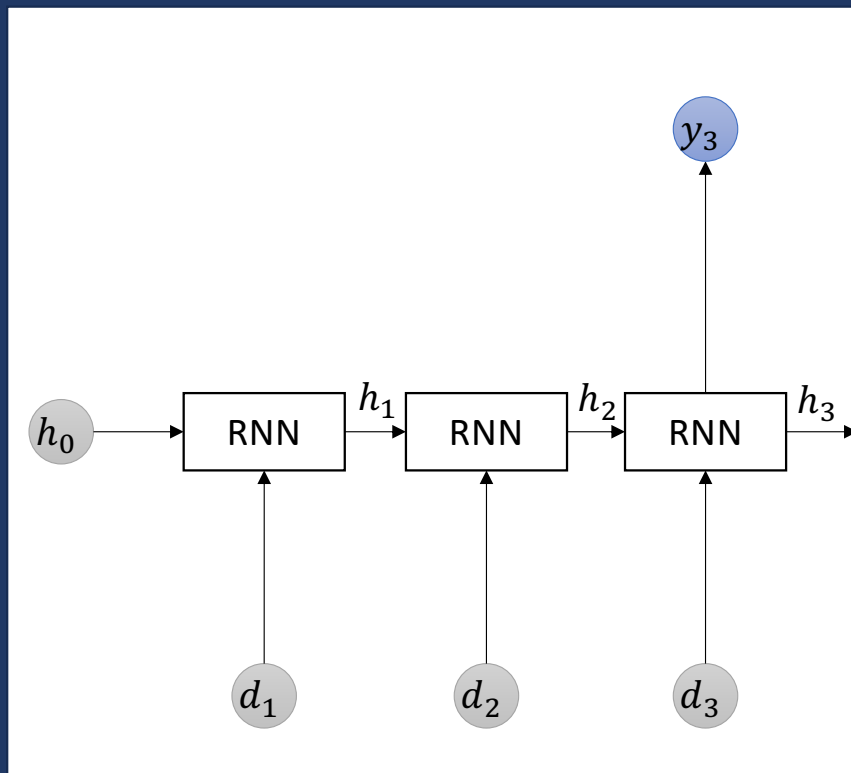


$$y_1 = f(Wd_1 + b)$$
$$y_2 = f(Wd_2 + b)$$
$$y_3 = f(Wd_3 + b)$$

- Every character is considered individually.
  - How do we preserve information about prior characters?
- **Add connections between the networks!**



# Character-based Text Generation: From FCN to RNN



➔ This view is called an **unrolled RNN**

## Idea

- add a state  $h$  that is carried between inputs.
- Update state with current input
- extract output  $y$  from current state  $h$
- **Share the weights between the timesteps!**

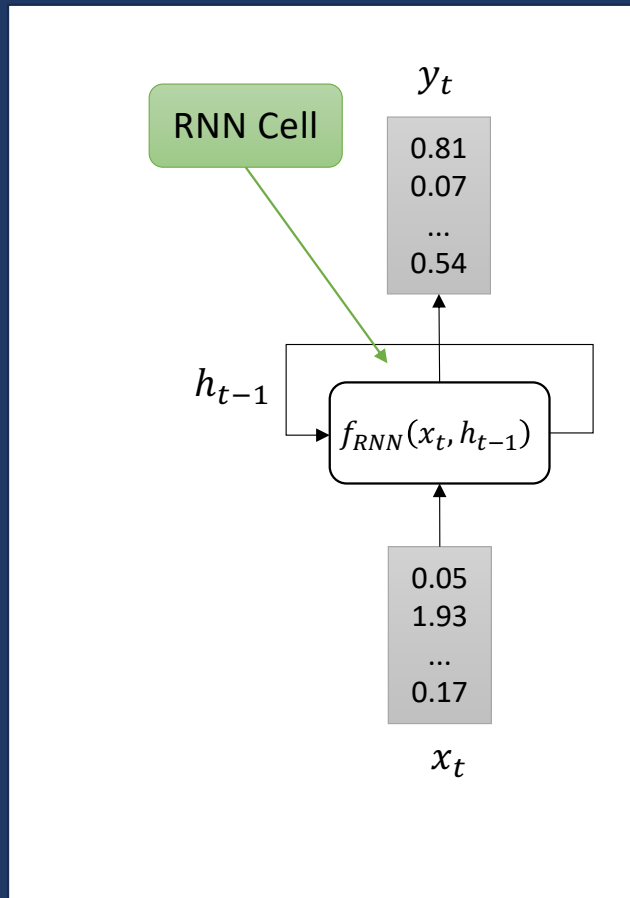
$$h_1 = \sigma_h(W_h d_1 + U_h h_0)$$

$$h_2 = \sigma_h(W_h d_2 + U_h h_1)$$

$$h_3 = \sigma_h(W_h d_3 + U_h h_2)$$

$$y_3 = \sigma_y(W_y h_3 + b_y)$$

# Vanilla RNN



$$h_t = f_{RNN}(x_t, h_{t-1}) = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$

- $W_h$ : maps input  $x_t$  into internal state space
- $U_h$ : extracts relevant information from prior state  $h_{t-1}$
- $b_h$ : bias. As usual, can be omitted using the bias trick
- $\sigma_h$ : internal activation function

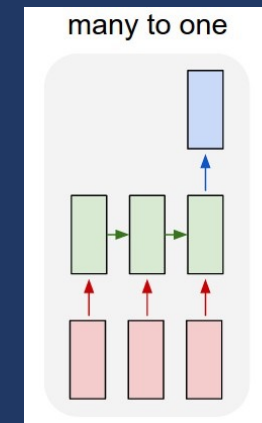
$$y_t = f_{out}(h_t) = \sigma_y(W_y h_t + b_y)$$

- $W_y$ : maps state  $h_t$  into output space
  - $b_y$ : bias
  - $\sigma_y$ : output activation function
- ➡ Basically a fully connected layer

- Learnable parameters:  $W_h, U_h, W_y, b_h, b_y$
- Initial state  $h_0$  is commonly initialized as 0-vector

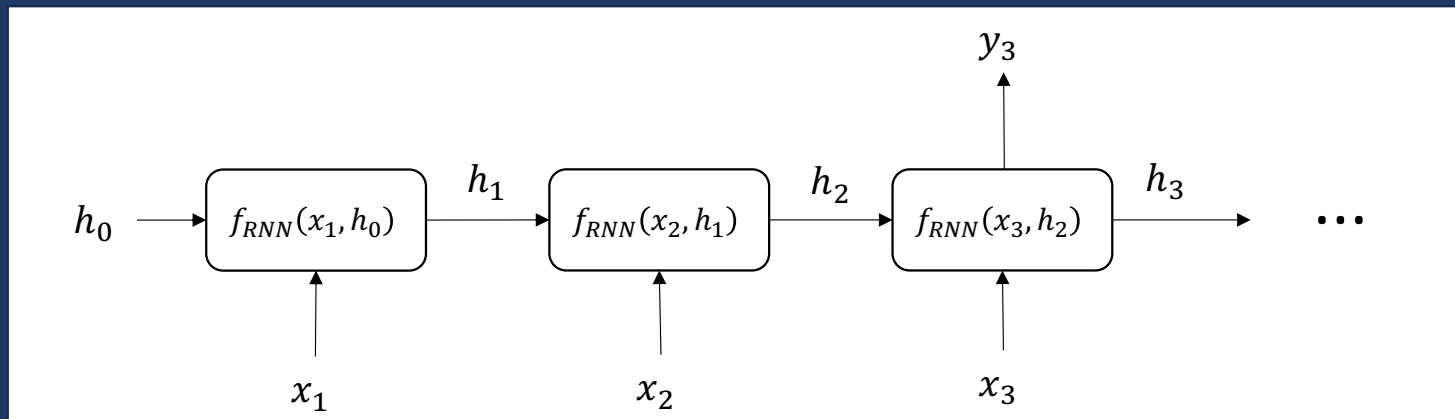
# Training Recurrent Neural Networks

- How do we learn the parameters of the RNN?
- Let's apply Backpropagation on the unrolled network!
- Similar for all types mentioned above
  - Focus on many-to-one for „handy“ gradients



# Backpropagation Through Time

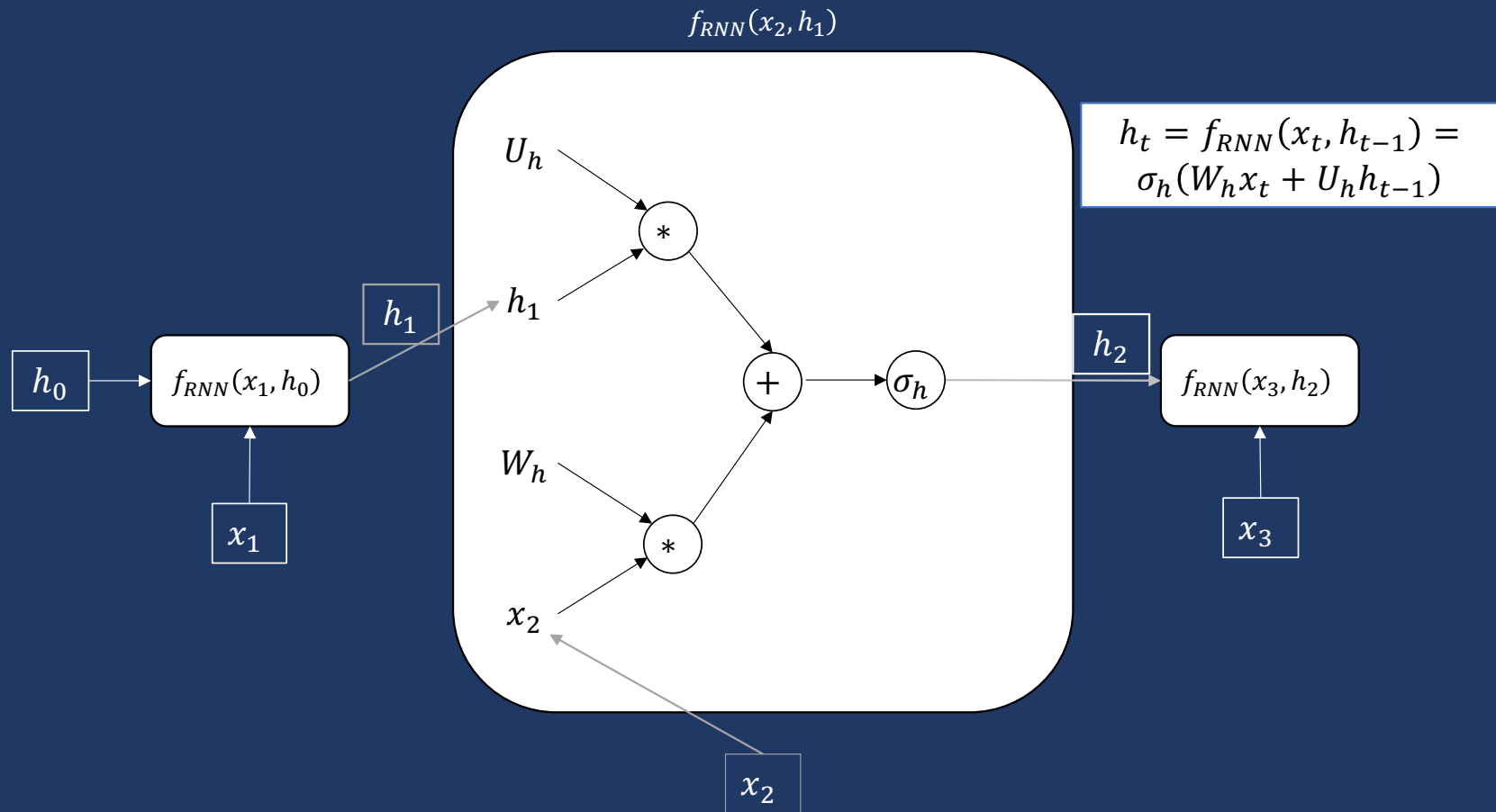
# Unrolling the RNN over Time



- Remember: An unrolled RNN is basically a feedforward network with some additional properties:
  - State  $h_t$  accumulates information about the sequence
  - All weights are shared between timesteps/layers
- Errors are backpropagated through  $h_t$  over all  $t$  steps of the sequence.

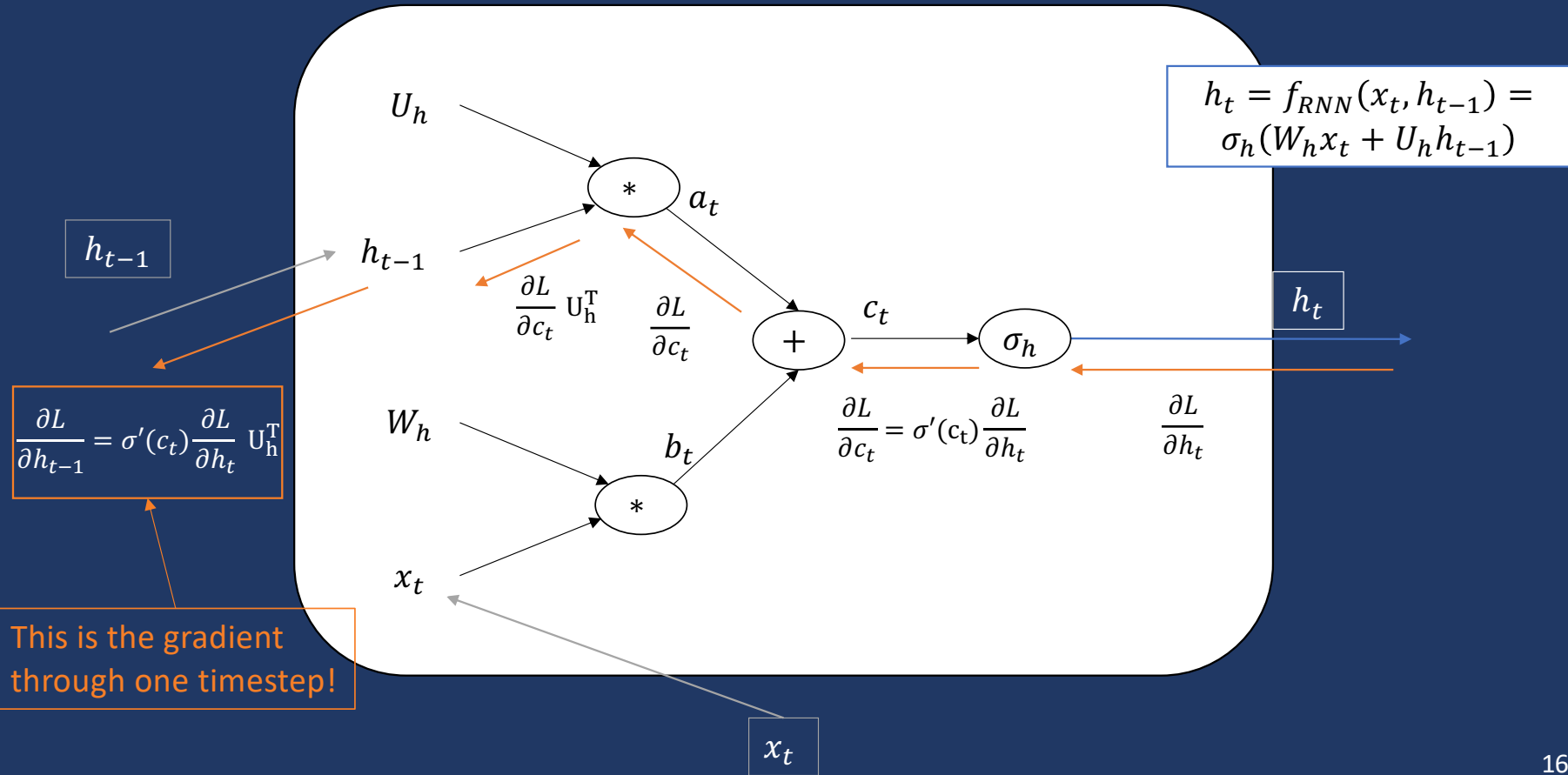
→ How does the gradient actually flow through an RNN cell?

# A look at the computational graph...



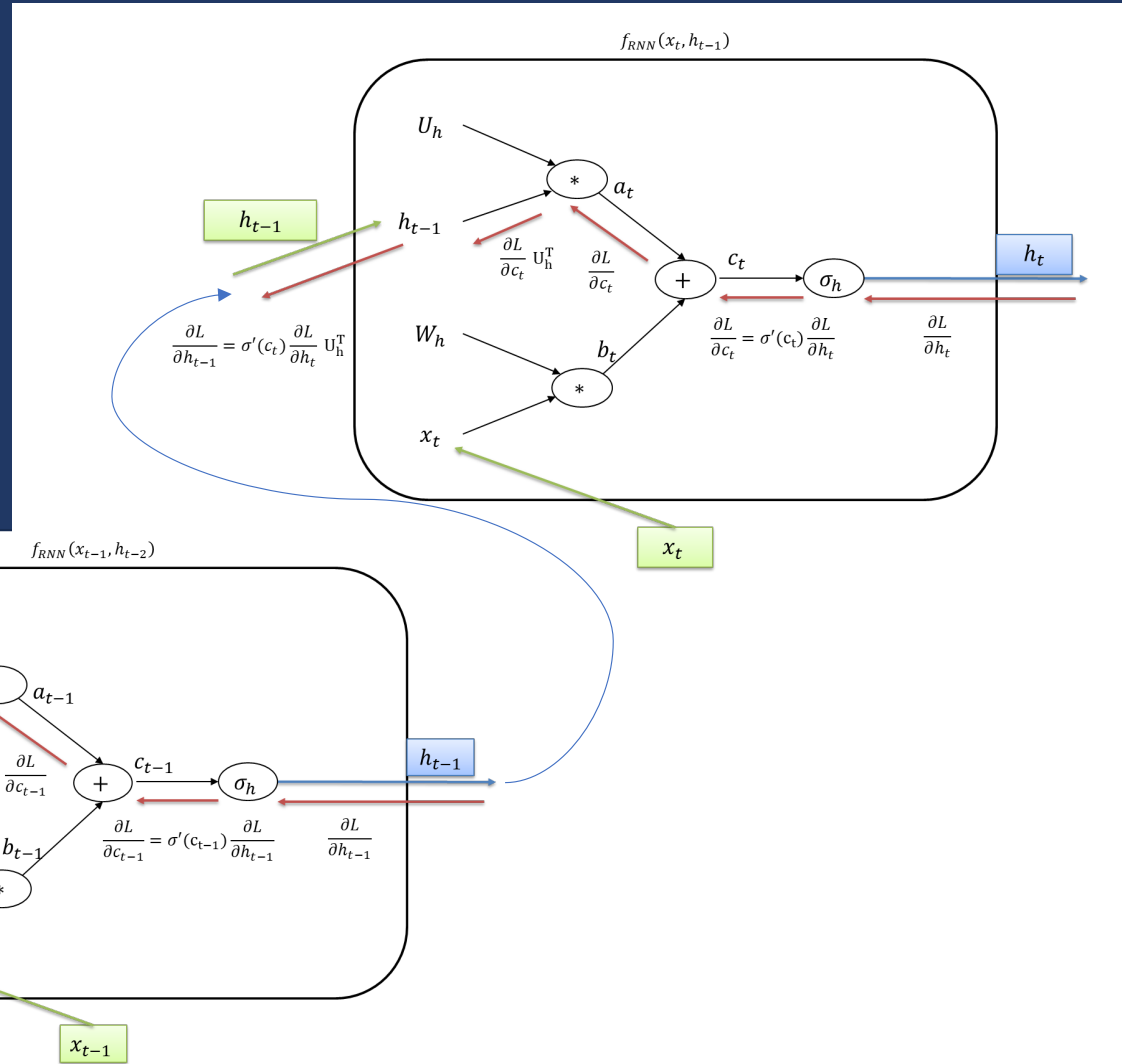
# ... and the gradient flow

$$f_{RNN}(x_t, h_{t-1})$$



# Going Deeper

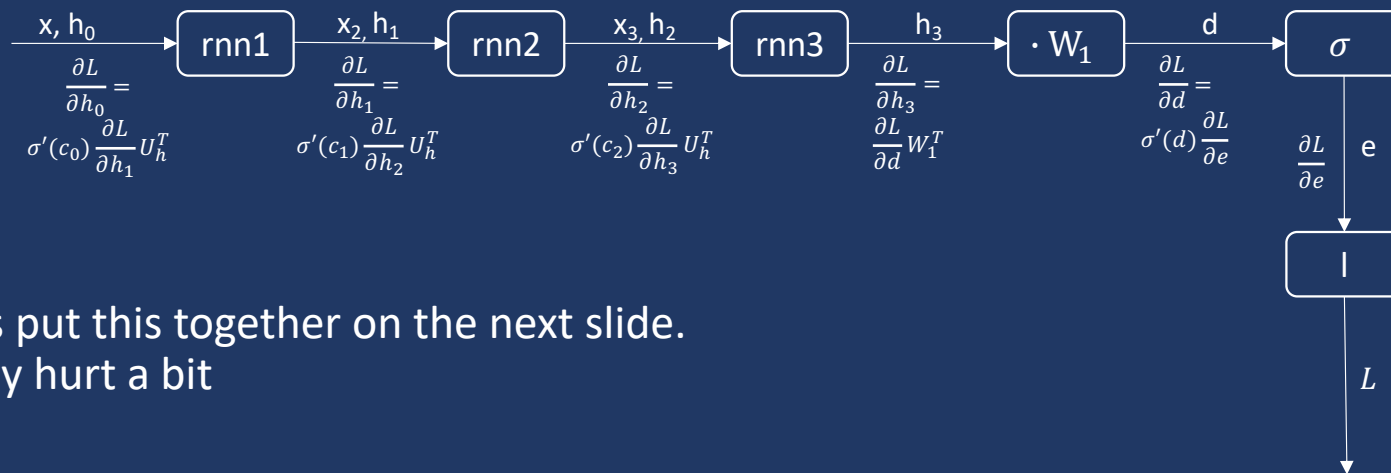
Adding a second timestep works just the same:





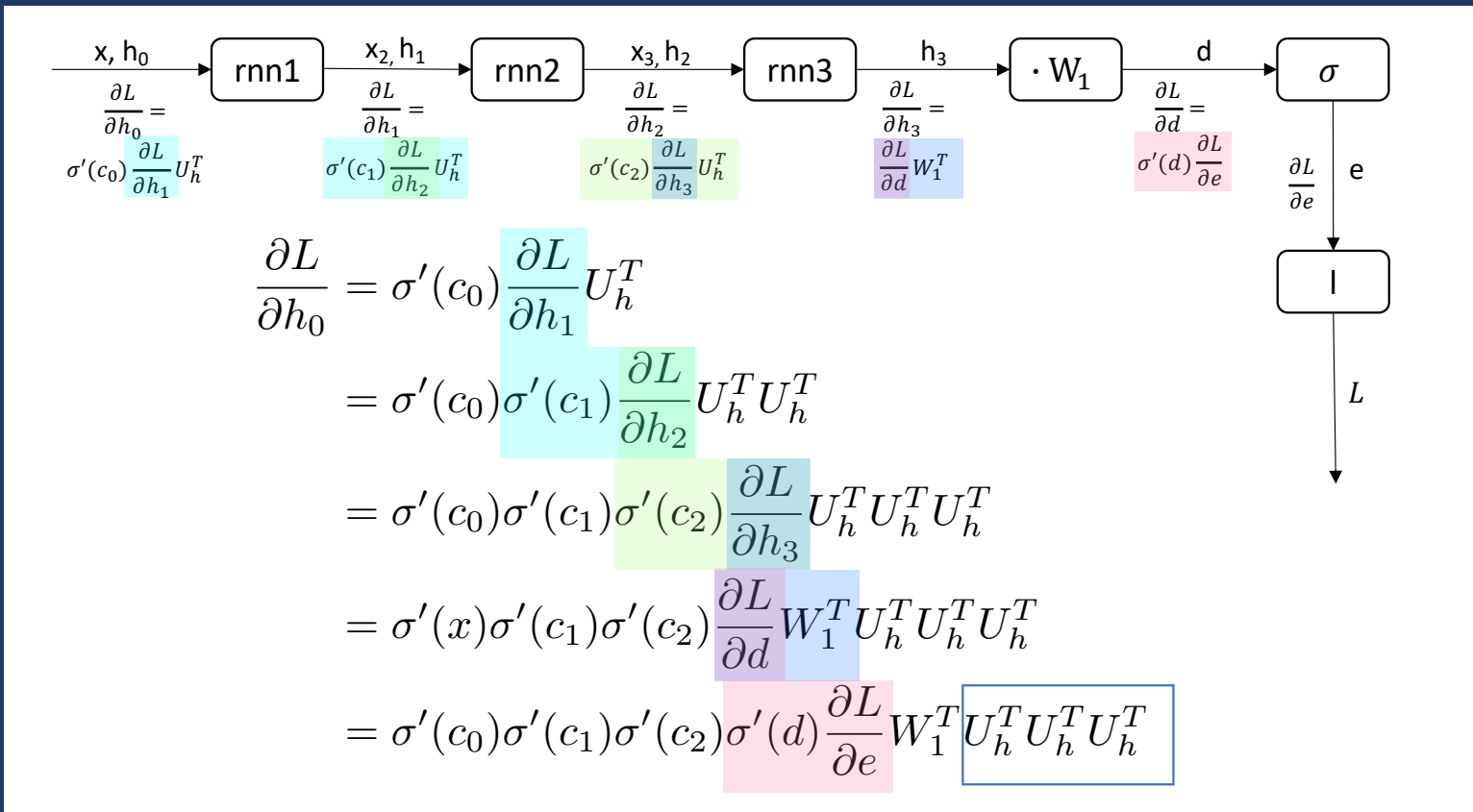
# And for a Full Network

- BPTT over a network with
  - one RNN cell, unrolled over three timesteps
  - one fully connected layer
  - and a loss function at the end



- Now let's put this together on the next slide. It will only hurt a bit

# Backpropagation Through Time



→ Matrix  $U_h^T$  appears once per timestep!

# Exploding and Vanishing Gradients

$$\frac{\partial L}{\partial x} = \sigma'(x)\sigma'(h_1)\sigma'(h_2)\sigma'(d) \frac{\partial L}{\partial e} W_1^T \boxed{U_h^T U_h^T U_h^T}$$

- Common problem in RNNs: **vanishing** or **exploding** gradients

**Vanishing Gradient:** Gradients from early timesteps get very small, having almost no influence on the update

**Exploding Gradient:** Gradients from early timesteps get very large, causing the optimiser to overshoot its goal

- Why?

→ After backpropagation over  $k$  timesteps, the gradient contains  $(U_h^T)^k$ . This either gets very large or very small!

(Similar to the scalar case, where  $x^k \rightarrow \infty$  for  $x > 1$  and  $x^k \rightarrow 0$  for  $x < 1$ )

# Combatting Exploding Gradients: Gradient Clipping

- Simple and effective remedy for exploding gradients:  
**Gradient Clipping**
  - During training, if a gradient gets larger than a predefined threshold, clip it to the threshold.

- Slightly more sophisticated alternative:  
**Gradient Rescaling**
  - Instead of clipping, rescale the gradients using their norm.



By Warren Long [CC BY 2.0  
(<https://creativecommons.org/licenses/by/2.0>)],  
via Wikimedia Commons

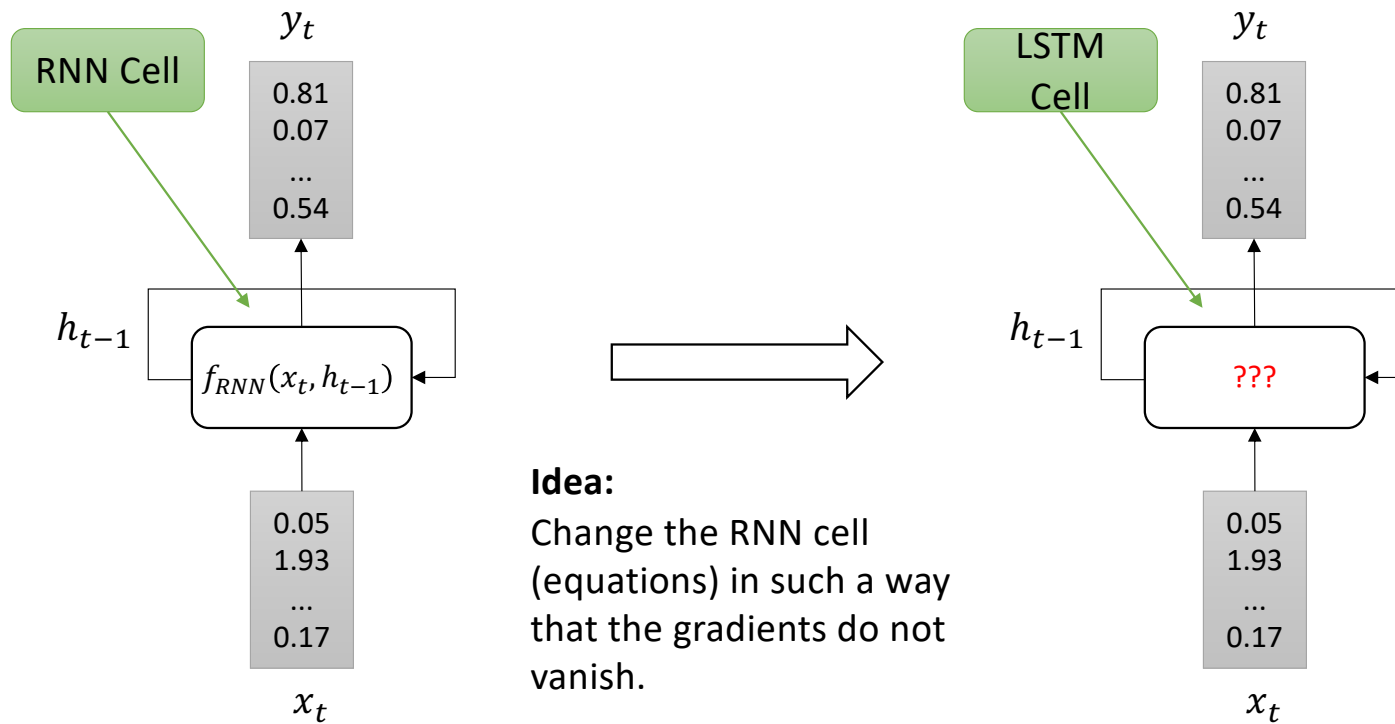
# Modifications of the Vanilla RNN

---

- Instead of Gradient Clipping/Rescaling, why not modify the RNN itself?
  - Hochreiter and Schmidhuber (1997) did just that!
  - **Long Short Term Memory (LSTM)** is a variant of RNNs that can better model long-term dependencies

# LSTM and Friends

# Long- Short Term Memory (LSTM)



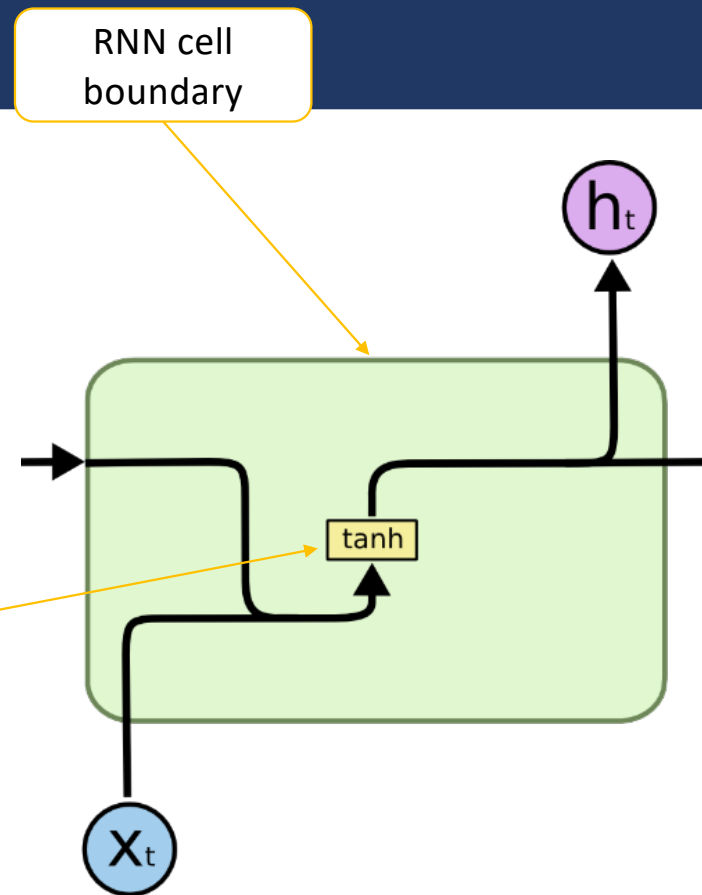
The following slides contain illustrations taken from Christopher Olah's Blog:  
<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

# Recall Vanilla RNN

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1})$$

neural network layer

Note that  $h_{t-1}$  is transformed into  $h_t$ !





# LSTM — Principles

---

- State changes are incremental:  $c_{t+1} = c_t + \Delta c_{t+1}$ 
  - Contrast to Vanilla RNNs: Vanilla state change is a transformation/matrix multiplication!
- State updates are selective
  - We only want to write things to the state that help us
- State access is selective
  - We need a way to select the most relevant knowledge from the state
- Information can be deleted from the state
  - Some information may become out-dated and must make way for more important stuff



Each type of selectivity is modeled as a „gate“

# LSTM — Gates

- Gates are modeled as layers similar to the Vanilla RNN
- They depend on the current input  $x_t$  and the last **shadow state**  $h_{t-1}$ 
  - We will see that the LSTM cell carries two states
    - 1) the state (or memory)  $c_t$
    - 2) the shadow state  $h_t$
- All gates have the following form:

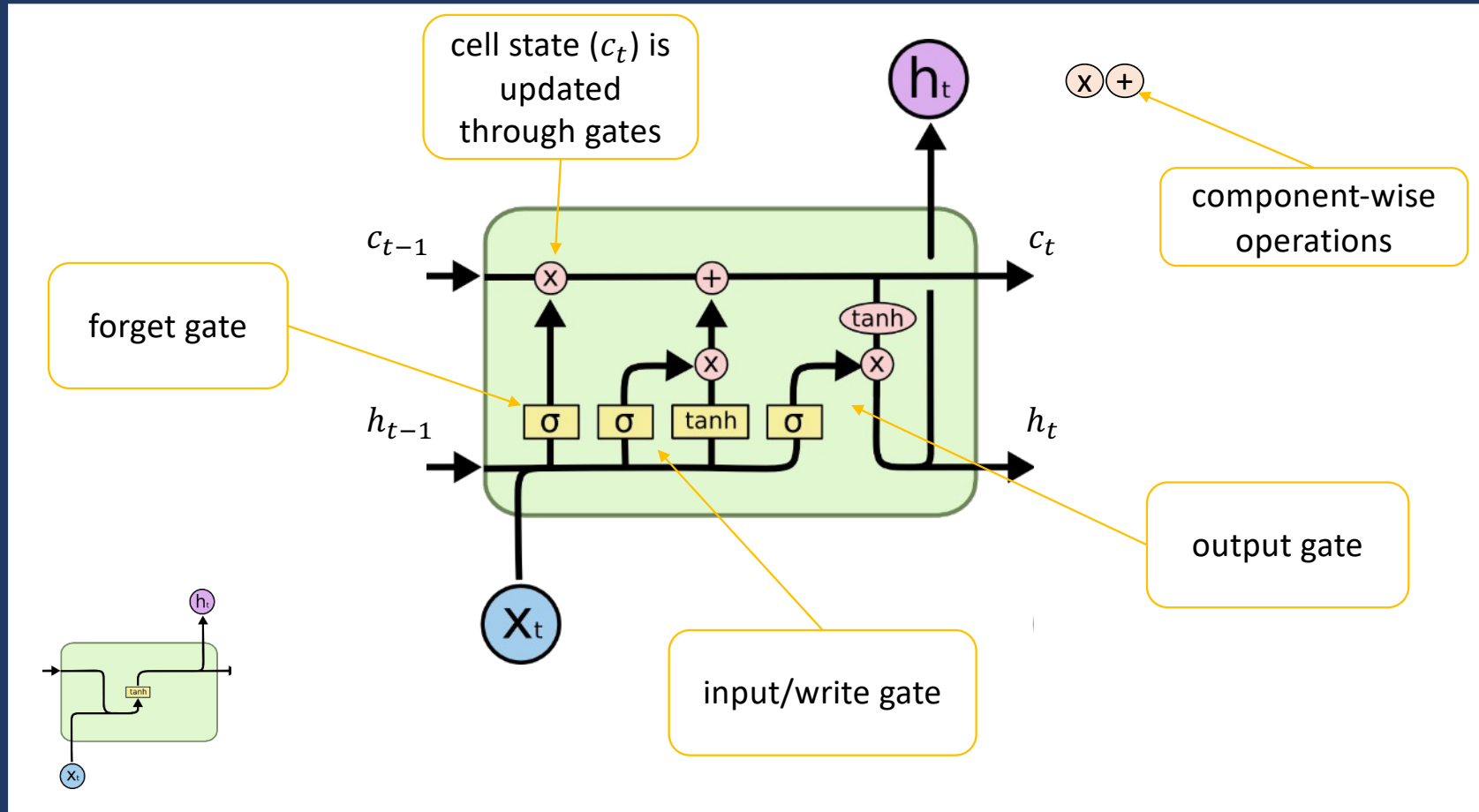
$$g(h_{t-1}, x_t) = \sigma(Wx_t + Uh_{t-1} + b)$$

Sigmoid activation function forces values in range [0, 1]

Interpretation (depending on the specific gate):

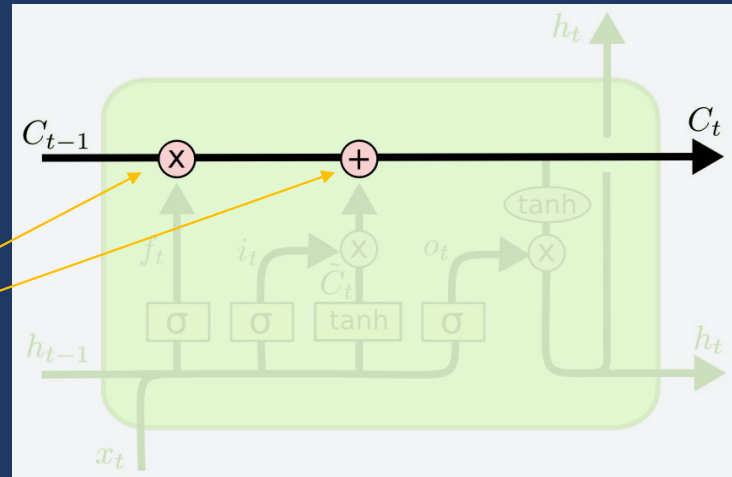
- 1 – keep all information, or read/write all information
- 0 – forget all information, or read/write none of that information

# LSTM — Overview

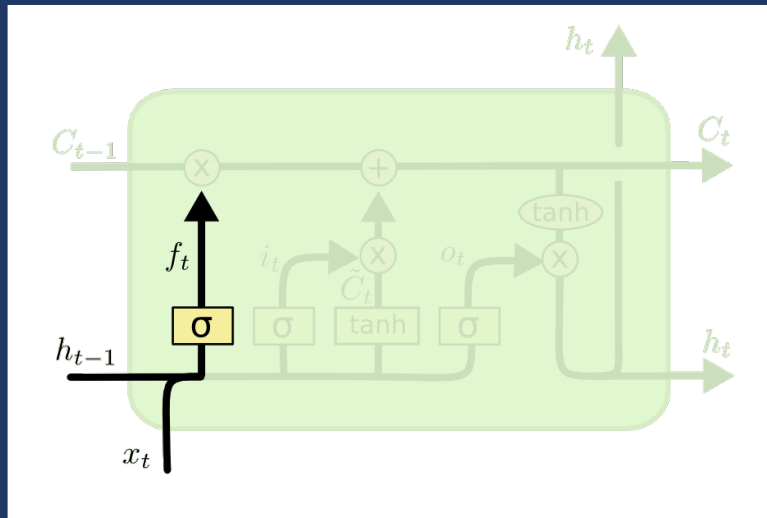


# LSTM — Memory state

- The state or memory  $c_{t-1}$  is never transformed by matrix multiplication
- Both interactions are carried out component-wise



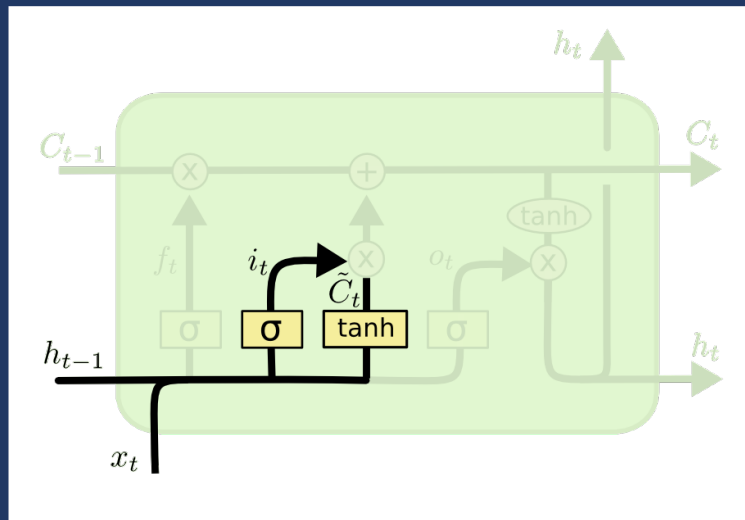
# LSTM — Forget gate



$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

- The forget gate determines which information is outdated and can be discarded
- The update of the cell state  $c_t$  is done by component-wise multiplication, resulting in a scaling of the previous state

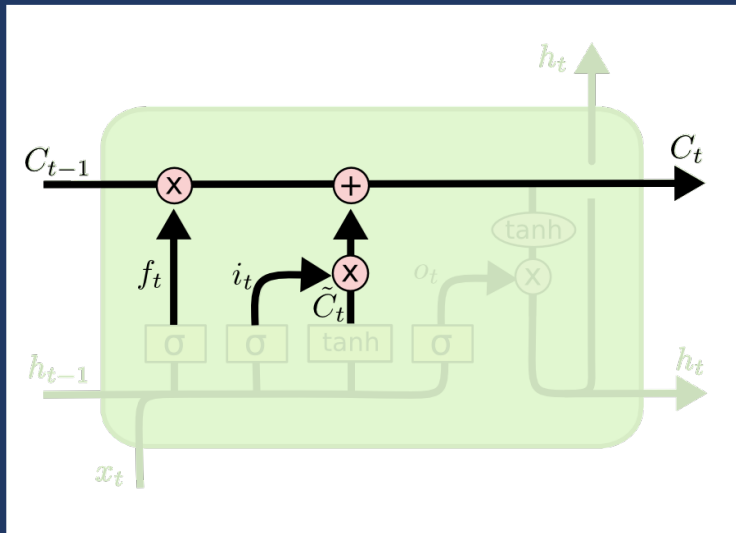
# LSTM — Input gate



$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

- $\tilde{C}_t$  consists of candidate values for updating the memory state
  - tanh is used instead of sigmoid, since it produces values in  $[-1, 1]$
  - remember that we want to model incremental state changes, the value range enables us to add ( $> 0$ ) or remove ( $< 0$ ) information
- $i_t$  is called the input gate and produces scaling factors (or weights) for the candidate values

# LSTM — Memory state update

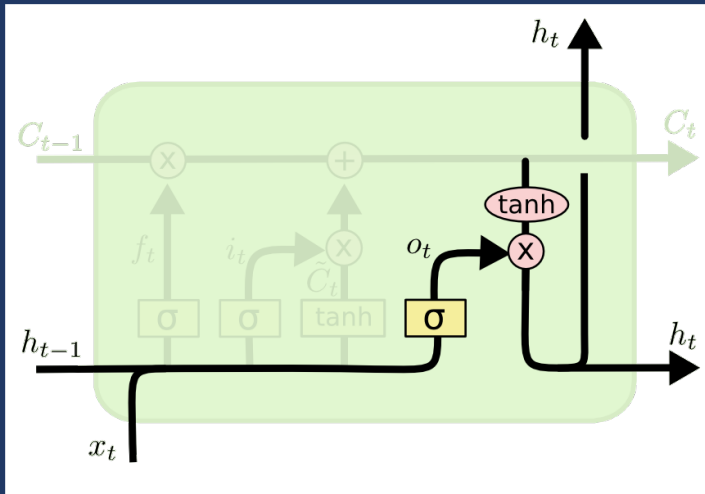


$$c_t = f_t \circ C_{t-1} + i_t \circ \tilde{c}_t$$

Component-wise  
multiplication

- The new memory state is computed by:
  - scaling the former memory state and thus „forgetting“ unnecessary information (forget gate)
  - adding information from the scaled candidate values (input/write gate)

# LSTM — Output gate



$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$h_t = o_t \circ \tanh(c_t)$$

- the final step is to decide what the output should be, this is controlled by the output gate  $o_t$



# Final Comparison

## Vanilla RNN

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$

## LSTM

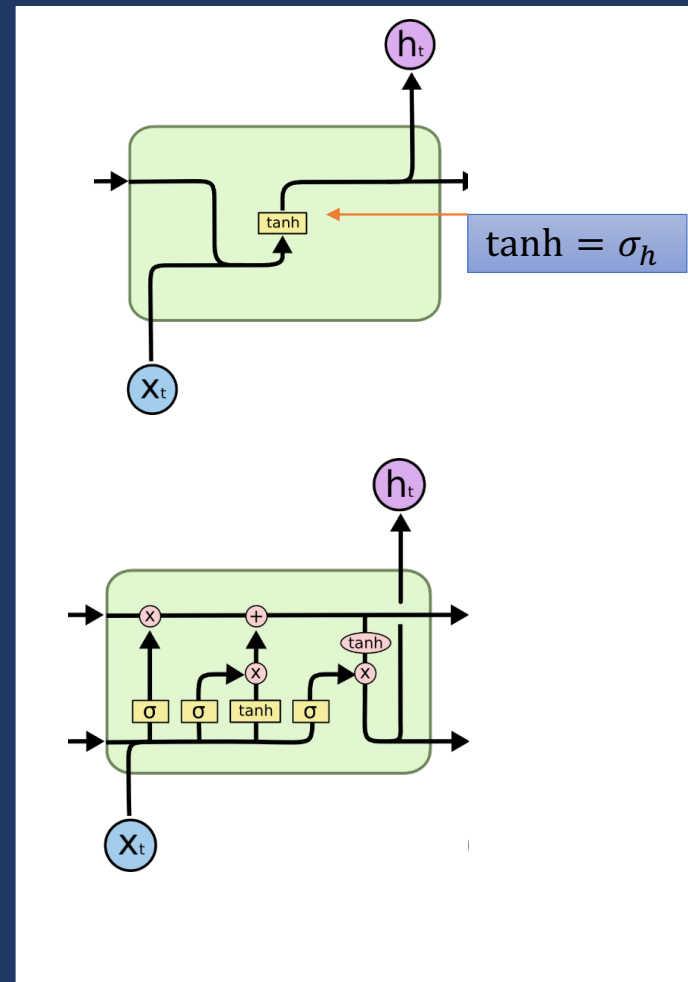
$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

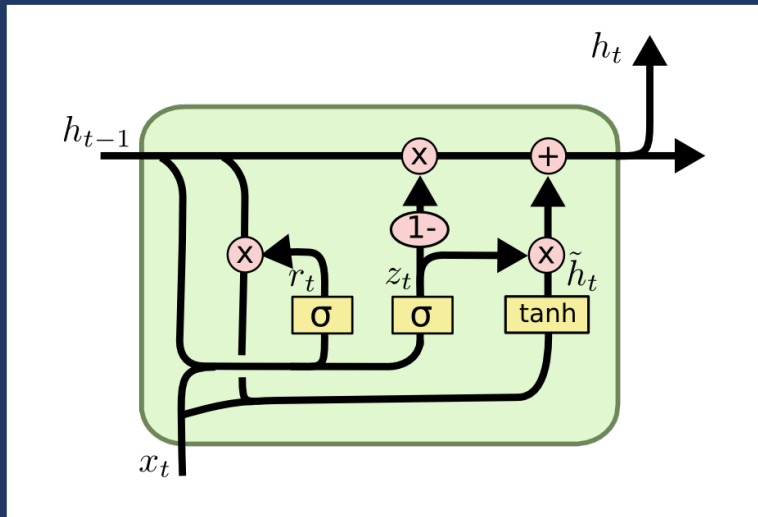
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

$$h_t = o_t \circ \tanh(c_t)$$



# Gated Recurrent Unit (GRU)



$$\begin{aligned}z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\\tilde{h}_t &= \tanh(W_h x_t + U_h (r_t \circ h_{t-1}) + b_h) \\h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t\end{aligned}$$

- GRU is a popular (and simpler) alternative cell
- merges input and forget gates
- uses only one state instead of two

# Practical Considerations

# Backpropagation Through a Long Time

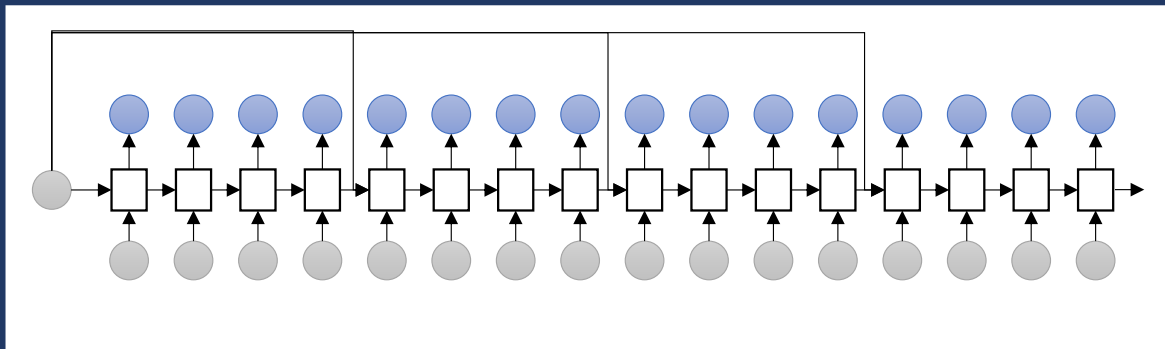
---

- Note that BPTT can get quite computationally expensive
- Especially, when the sequences get very long

→ In practice, BPTT is usually not done over the full sequence, but only over a part →  
**Truncated BPTT**

# Truncated BPTT I

- Backpropagating over the whole sequence is often not computationally feasible
- **Naive solution:**  
Split the sequence into chunks of  $k$  steps and treat every chunk as an individual training instance.
- **Drawback:**  
This prevents the RNN from learning long-range dependencies that span more than  $k$  steps.



chunk size:  $k = 4$

# Truncated BPTT II

There is a different variant that preserves the state:

for  $t$  from 1 to  $T$  do

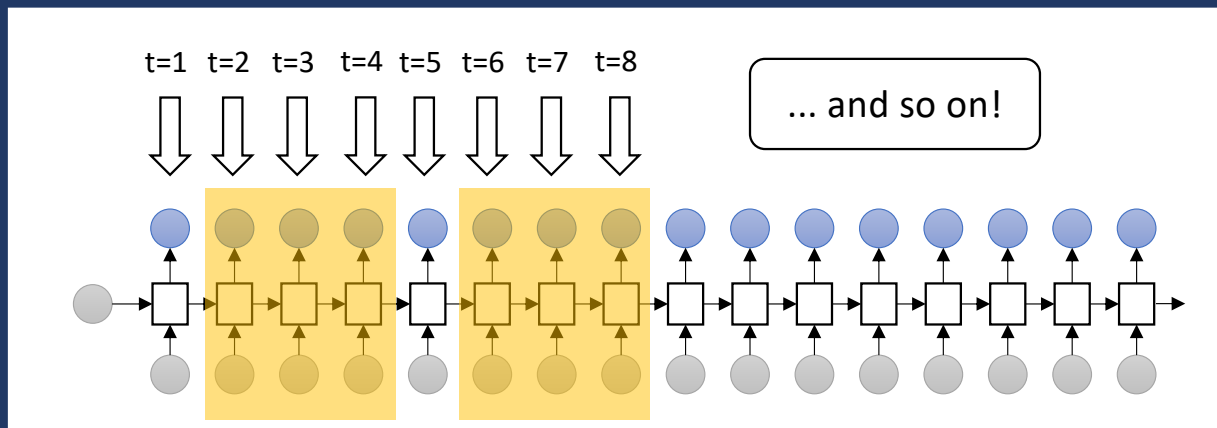
$$h_t = f_{RNN}(x_t, h_{t-1})$$

$$y_t = f_{out}(h_t)$$

if  $t$  divides  $k_1$  then

Run BPTT on the chunk from  $t$  down to  $t - k_2$

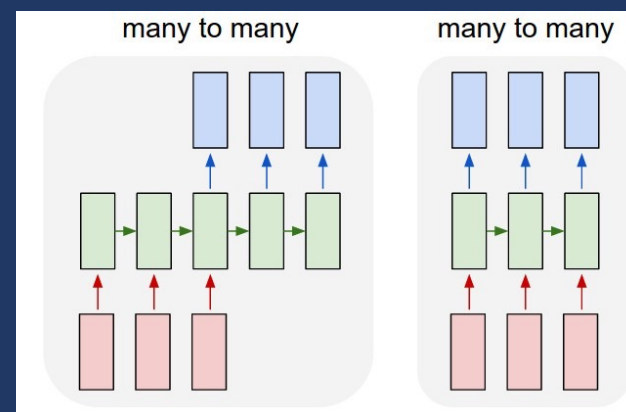
$T$  – length of the sequence  
 $t$  – current step  
 $k_1$  – controls how often BPTT occurs  
 $k_2$  – controls how many steps are included in BPTT



$T=12$   
 $k_1 = 4$   
 $k_2 = 3$

# Truncated BPTT

- Note: The variants from the two previous slides only work with many-to-many networks!
- Why?
  - Need to calculate a loss at least every  $k$  or  $k_1$  steps!
- What can we do for many-to-one tasks?
  - Calculate the forward pass over the full sequence, then do backpropagation for  $k_2$  steps



# RNNs and BPTT in PyTorch

---

- PyTorch provides `nn.LSTM`, `nn.RNN`, and `nn.GRU` modules
- They return the output and hidden state for each timestep
- Backpropagate through all timesteps as you know it
- Truncated BPTT can be implemented by hand → not easy
- Truncated BPTT is also implemented in PyTorch Ignite → easy





# Recurrent Neural Networks in Practice - RNNs for Text Generation

# RNNs for Text Generation

---

- Text Generation = Given some training corpus, create new text that is „similar“ to the corpus
- Well-suited task for RNNs:
  - Internal State encodes the text so far
  - Output layer predicts the next token
- Possible at different levels:
  - Word level
  - Character level
  - Phoneme level
  - ...

# RNNs for Text Generation

- Popular blog post: „The Unreasonable Effectiveness of Recurrent Neural Networks” (Andrej Karpathy)
- Train a character-level RNN on different corpora
- Generate new text
- Investigate what is going on!
  - Look at different RNN cells
  - Play around with hyperparameters

Clown: Come, sir, I will make did  
behold your worship.  
VIOLA: I'll drink it.

# Shakespeare!

PANDARUS:

Alas, I think he shall be come approached and the day  
When little strain would be attain'd into being never fed,  
And who is but a chain and subjects of his death,  
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,  
Breaking and strongly should be buried, when I perish  
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and  
my fair nues begun out of the fact, to be conveyed,  
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

# Algebraic Geometry!

```
\begin{proof}
We may assume that  $\mathcal{I}$  is an abelian sheaf on
 $\mathcal{C}$ .
\item Given a morphism  $\Delta : \mathcal{F} \rightarrow \mathcal{I}$ 
is an injective and let  $\mathcal{q}$  be an abelian sheaf on  $X$ .
Let  $\mathcal{F}$  be a fibered complex. Let  $\mathcal{F}$  be a
category.
\begin{enumerate}
\item \hyperref[setain-construction-phantom]{Lemma}
\label{lemma-characterize-quasi-finite}
Let  $\mathcal{F}$  be an abelian quasi-coherent sheaf on
 $\mathcal{C}$ .
Let  $\mathcal{F}$  be a coherent  $\mathcal{O}_X$ -module. Then
 $\mathcal{F}$  is an abelian catenary over  $\mathcal{C}$ .
\item The following are equivalent
\begin{enumerate}
\item  $\mathcal{F}$  is an  $\mathcal{O}_X$ -module.
\end{enumerate}
\end{lemma}
```

## Some errors:

- Opens proof, closes lemma
  - Opens enumerate, doesn't close it
- Fix these manually!

# Algebraic Geometry!

*Proof.* Omitted. 😊

**Lemma 0.1.** *Let  $\mathcal{C}$  be a set of the construction.*

*Let  $\mathcal{C}$  be a gerber covering. Let  $\mathcal{F}$  be a quasi-coherent sheaves of  $\mathcal{O}$ -modules. We have to show that*

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

*Proof.* This is an algebraic space with the composition of sheaves  $\mathcal{F}$  on  $X_{\text{étale}}$  we have

$$\mathcal{O}_X(\mathcal{F}) = \{\text{morph}_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where  $\mathcal{G}$  defines an isomorphism  $\mathcal{F} \rightarrow \mathcal{F}$  of  $\mathcal{O}$ -modules.

**Lemma 0.2.** *This is an integer  $Z$  is injective.*

*Proof.* See Spaces, Lemma ??.

**Lemma 0.3.** *Let  $S$  be a scheme. Let  $X$  be a scheme and  $X$  is an affine open covering. Let  $U \subset X$  be a canonical and locally of finite type. Let  $X$  be a scheme. Let  $X$  be a scheme which is equal to the formal complex.*

*The following to the construction of the lemma follows.*

*Let  $X$  be a scheme. Let  $X$  be a scheme covering. Let*

$$b : X \rightarrow Y' \rightarrow Y \rightarrow Y \rightarrow Y' \times_X Y \rightarrow X.$$

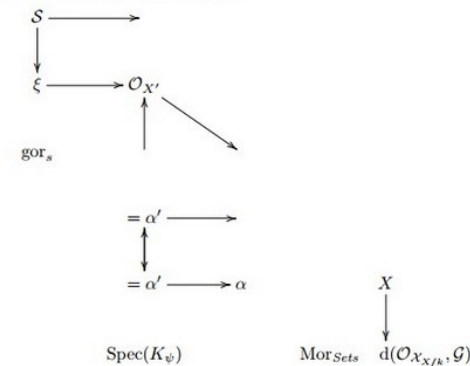
*be a morphism of algebraic spaces over  $S$  and  $Y$ .*

*Proof.* Let  $X$  be a nonzero scheme of  $X$ . Let  $X$  be an algebraic space. Let  $\mathcal{F}$  be a quasi-coherent sheaf of  $\mathcal{O}_X$ -modules. The following are equivalent

- (1)  $\mathcal{F}$  is an algebraic space over  $S$ .
- (2) If  $X$  is an affine open covering.

Consider a common structure on  $X$  and  $X$  the functor  $\mathcal{O}_X(U)$  which is locally of finite type.

This since  $\mathcal{F} \in \mathcal{F}$  and  $x \in \mathcal{G}$  the diagram



is a limit. Then  $\mathcal{G}$  is a finite type and assume  $S$  is a flat and  $\mathcal{F}$  and  $\mathcal{G}$  is a finite type  $f_*$ . This is of finite type diagrams, and

- the composition of  $\mathcal{G}$  is a regular sequence,
- $\mathcal{O}_{X'}$  is a sheaf of rings.

*Proof.* We have see that  $X = \text{Spec}(R)$  and  $\mathcal{F}$  is a finite type representable by algebraic space. The property  $\mathcal{F}$  is a finite morphism of algebraic stacks. Then the cohomology of  $X$  is an open neighbourhood of  $U$ .

*Proof.* This is clear that  $\mathcal{G}$  is a finite presentation, see Lemmas ??. A reduced above we conclude that  $U$  is an open covering of  $\mathcal{C}$ . The functor  $\mathcal{F}$  is a "field"

$$\mathcal{O}_{X,x} \rightarrow \mathcal{F}_{\bar{x}}^{-1}(\mathcal{O}_{X_{\text{étale}}}) \rightarrow \mathcal{O}_{X_x}^{-1}(\mathcal{O}_{X_x}(\mathcal{O}_{X_x}^{\bar{v}_n}))$$

is an isomorphism of covering of  $\mathcal{O}_{X_x}$ . If  $\mathcal{F}$  is the unique element of  $\mathcal{F}$  such that  $X$  is an isomorphism.

The property  $\mathcal{F}$  is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme  $\mathcal{O}_X$ -algebra with  $\mathcal{F}$  are opens of finite type over  $S$ .

If  $\mathcal{F}$  is a scheme theoretic image points.

If  $\mathcal{F}$  is a finite direct sum  $\mathcal{O}_{X_x}$  is a closed immersion, see Lemma ??. This is a sequence of  $\mathcal{F}$  is a similar morphism.

## 5.2 Sequence-to-Sequence

---

- Sequence-to-Sequence Models
- From RNN to Seq2Seq
- Seq2Seq model (Encoder/Decoder)
- Enhancements



# Motivation for Sequence to Sequence models - Machine Translation



# Machine Translation

- Task in Machine Translation (MT):
  - Given a piece of text in a **source language**...
  - ... translate the text to a different **target language**

Sindarin:



English:

Pedo mellon a minno!



Say friend and enter!

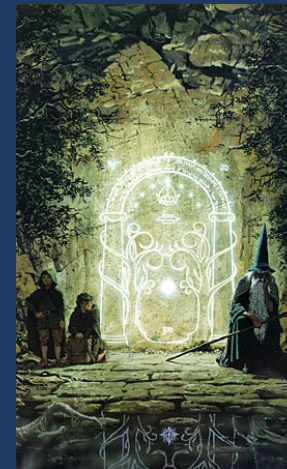


Image: Ted Nasmith

# Neural Machine Translation

- How to build a neural network for Machine Translation?

→ RNNs seem like a natural choice

- Sequences as input and sequences as output
- But: some problems that we need to address to get better translations!
  - Implicit model for alignment

Sindarin:



English:

Mae govannen, mellon nin!



Hello, my friend!

- Performance deteriorates for long sentences


Aragorn Arathornion Edhelharn, aran Gondor ar Hir i Mbair Annui, anglennatha I  
Varanduiniant erin dolothen Ethuil, egor ben genediad Drannail erin Gwirth edwen

• ...



Aragorn, Arathorn's son, Elfstone, King of Gondor and Lord of the Westlands  
.....?!?

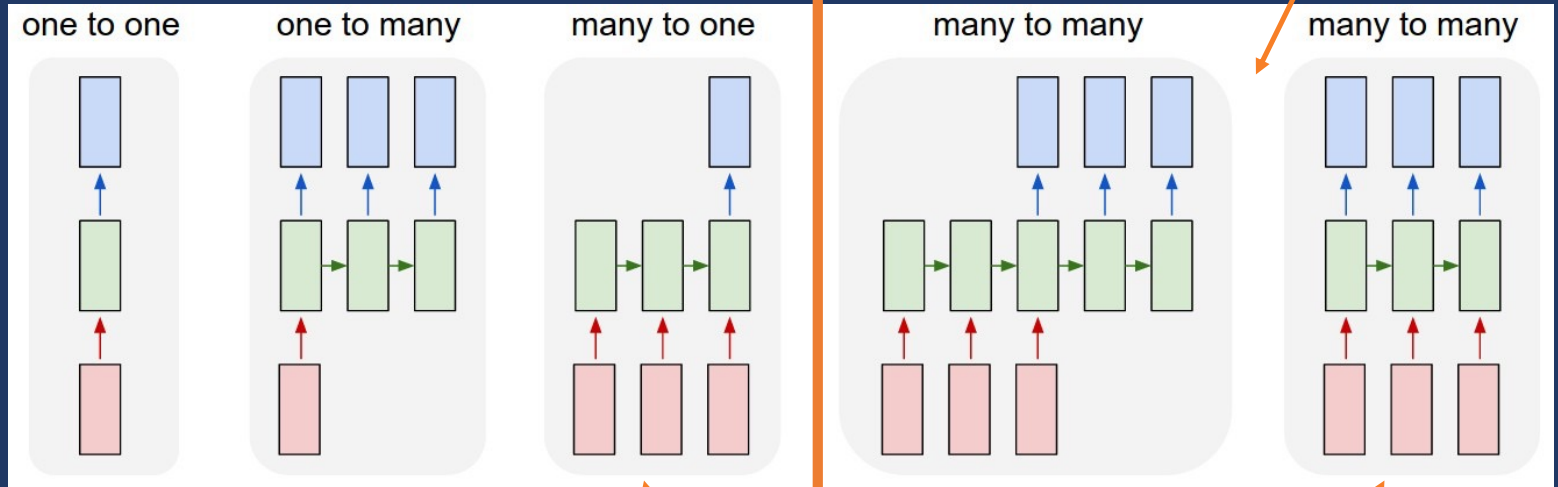
→ Use **Sequence to Sequence** models with some further tweaks!



# Sequence to Sequence models (Seq2Seq)

# Recap types of RNN

Seq2Seq



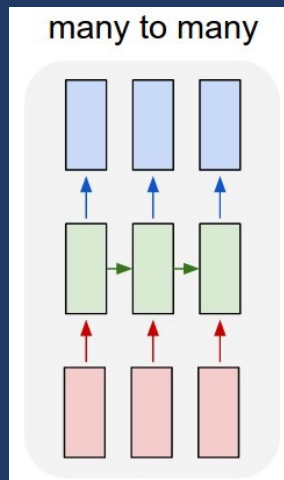
Fully Connected Network

Network mapping a sequence to one output

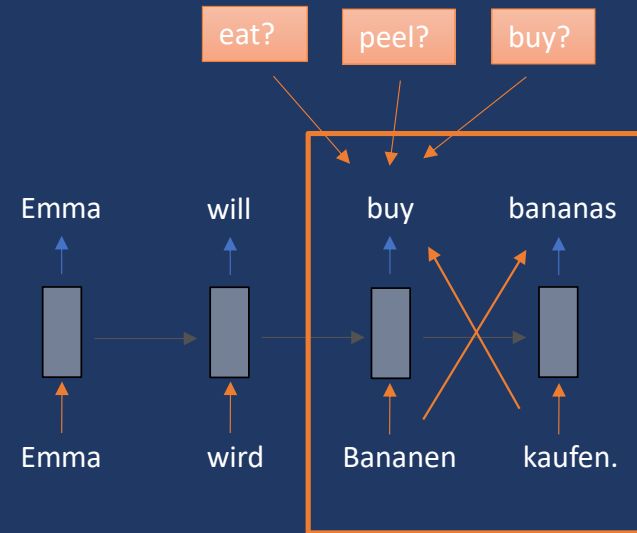
Network returning an output after each timestep of a sequence

Recurrent Neural Networks

# Single RNN



**Idea:**  
Use the RNN output at every timestep to predict the translated word.

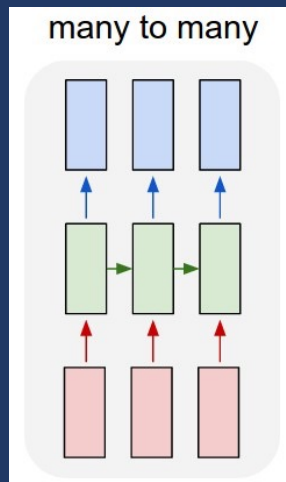


**Problem:**

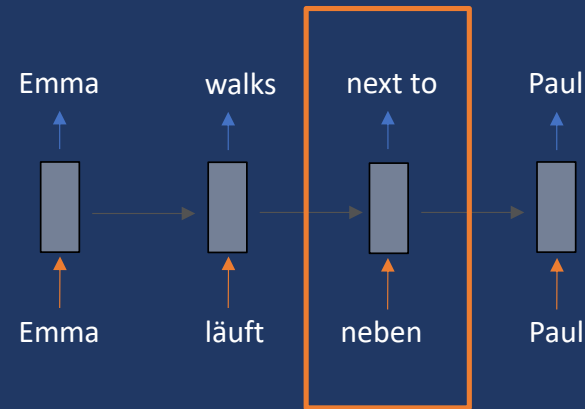
The word order differs between translated sentences.

In the above case, the model can't know whether Emma is going to eat, peel or buy the banana!

# Single RNN



**Idea:**  
Use the RNN output at every timestep to predict the translated word.



**Problem:**

Usually input and target sentence do not have the same number of words.

We can not accommodate source and target sentences of different lengths with a single RNN!

# From RNN to Seq2Seq

---

- Clearly, a simple RNN is not sufficient!
- How can we tweak the model to overcome these problems?

## Challenges:

### 1. Handle input and target sequences of different lengths

#### ➤ Use two distinct RNNs:

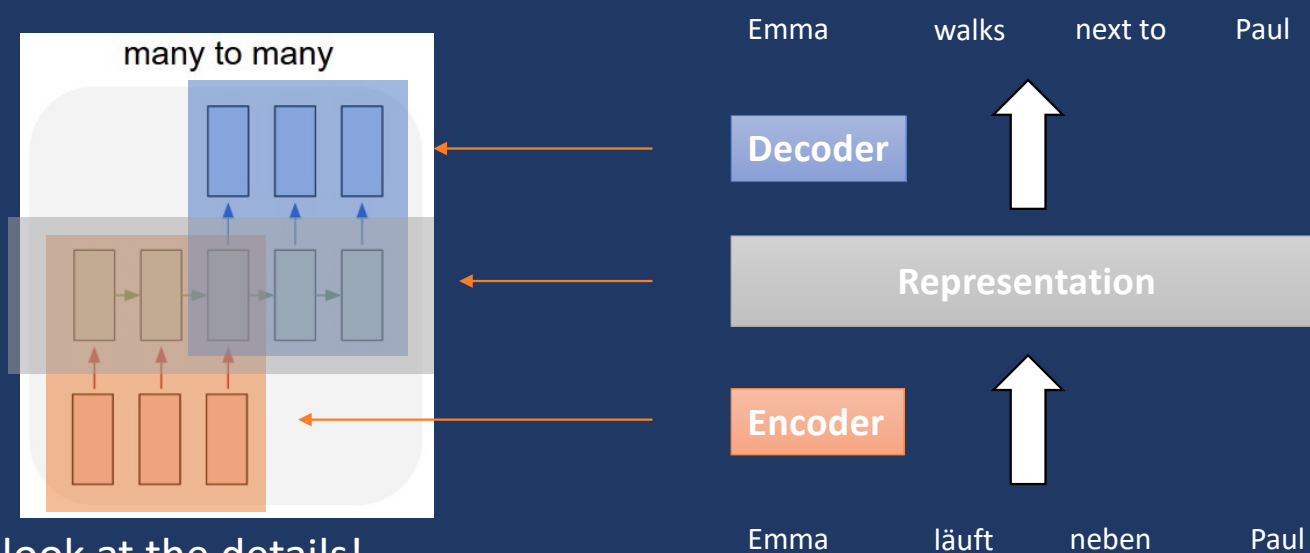
- An **encoder** network maps the input to an internal representation  $h$
- A **decoder** network generates a target sequence from  $h$

### 2. Generate target sequence based on the full input sequence

- Encoder reads the **entire sentence** before passing it to the decoder

# Encoder — Decoder Model

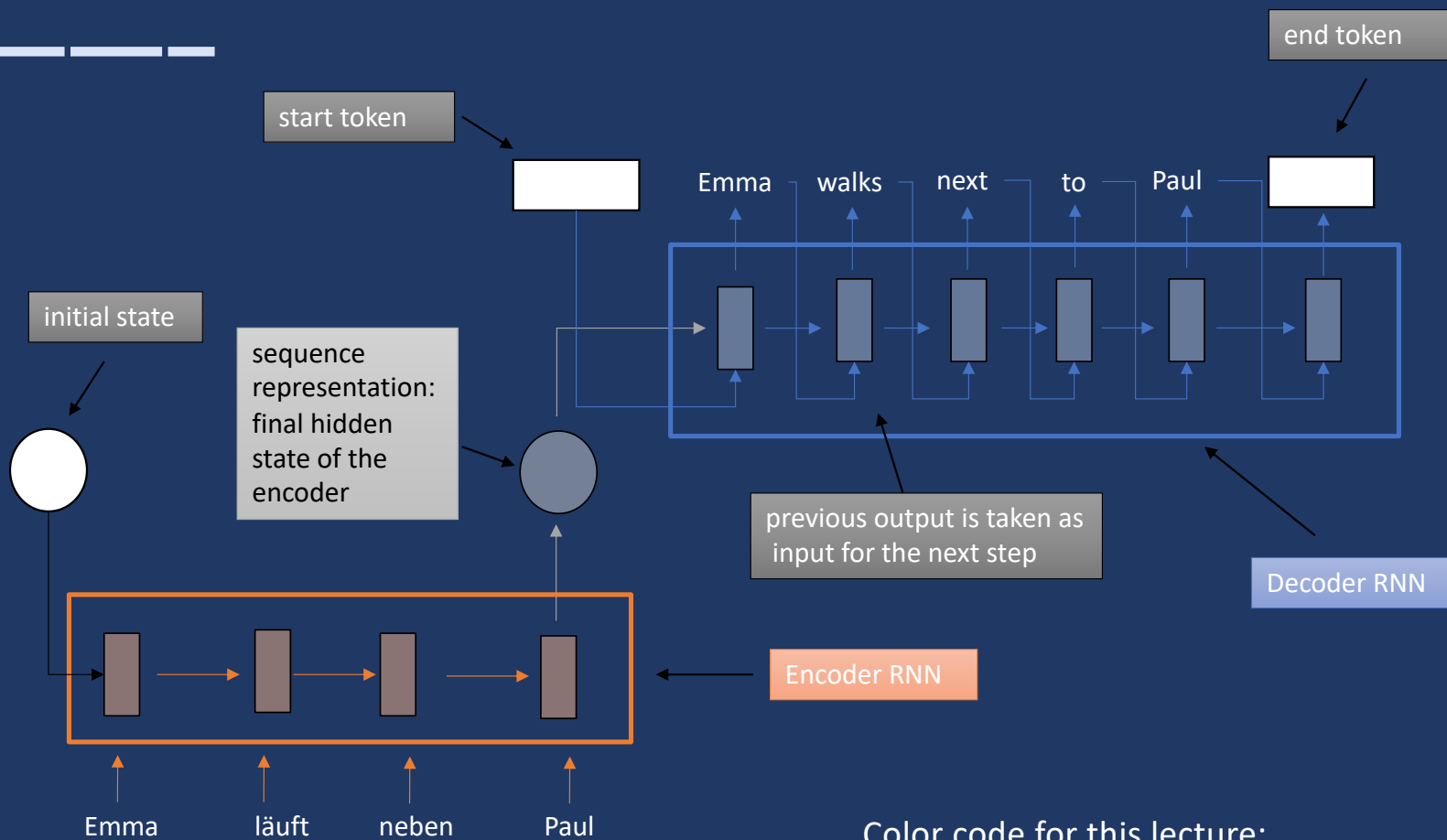
- High level visualisation of an Encoder-Decoder architecture:



- Now let's look at the details!



# Encoder — Decoder: RNN



Color code for this lecture:  
Encoder – Representation – Decoder

# Encoder — Decoder Loss

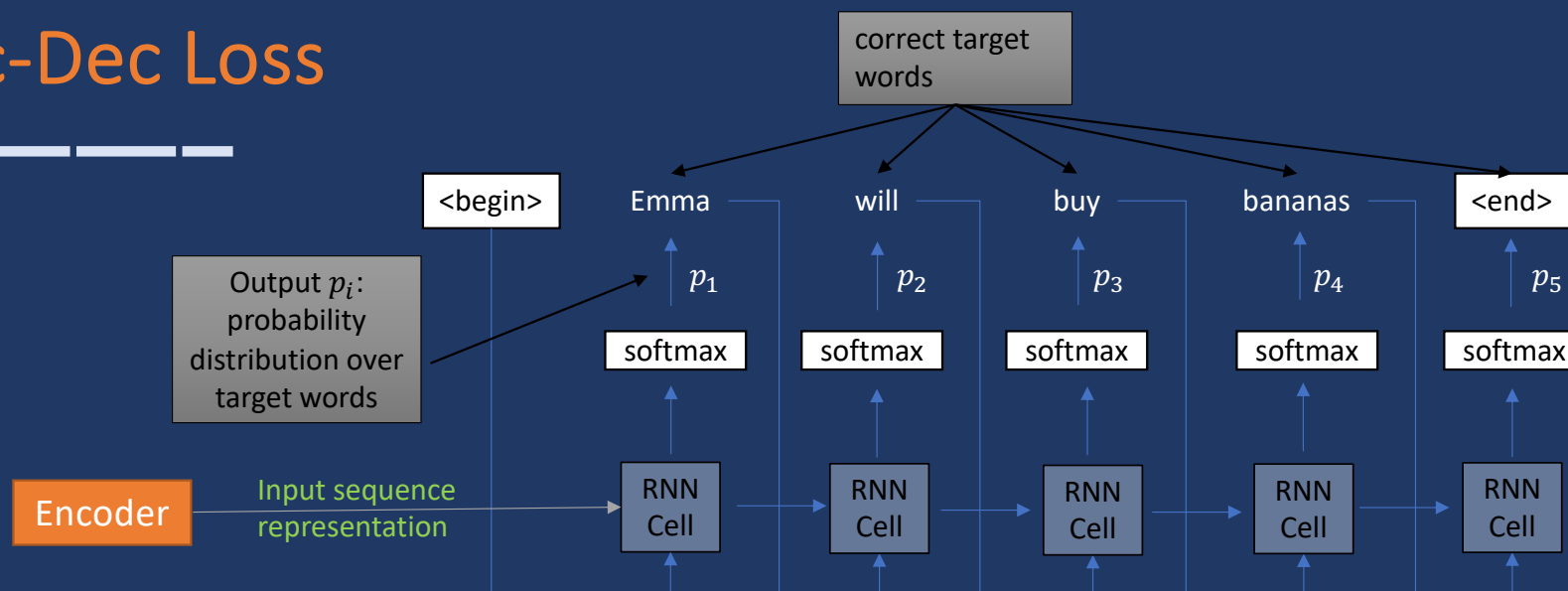
- To train the model, we need to define a loss function
- Since we have multiple outputs, we need to aggregate the loss somehow
- **Idea:** Use the probability the model assigns to the entire target sentence!

→ Probability of the target sentence  $(y_1, \dots, y_n)$ :

$$P(y_1, \dots, y_n) = \prod_{i=1}^n p_i[y_i]$$

↑  
Probability of target word  $y_i$  at  
timestep  $i$  according to the decoder

# Enc-Dec Loss



Probability of the target sentence according to the decoder:

$$P(y_1, \dots, y_n) = \prod_{i=1}^n p_i[y_i]$$

A good model should assign a high probability to the target translation!

This gives us the loss function:

$$L = -\log P(y_1, \dots, y_n) = -\log \prod_{i=1}^n p_i[y_i] = -\sum_{i=1}^n \log p_i[y_i]$$

# Improving Seq2Seq Models

---

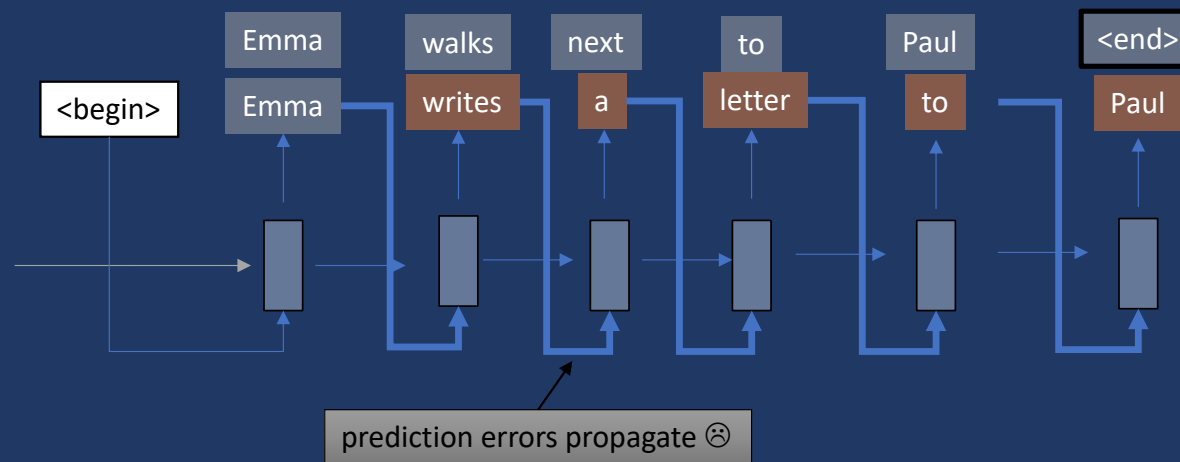
- That's basically all we need!
- We can now train Sequence to Sequence models with an Encoder-Decoder architecture
- Let's look at some optimisations to improve our results

No backprop today 😞

But you're welcome to do it yourself!

# Seq2Seq: Teacher Forcing

Remember: We use the output from the previous step as input to the next step.



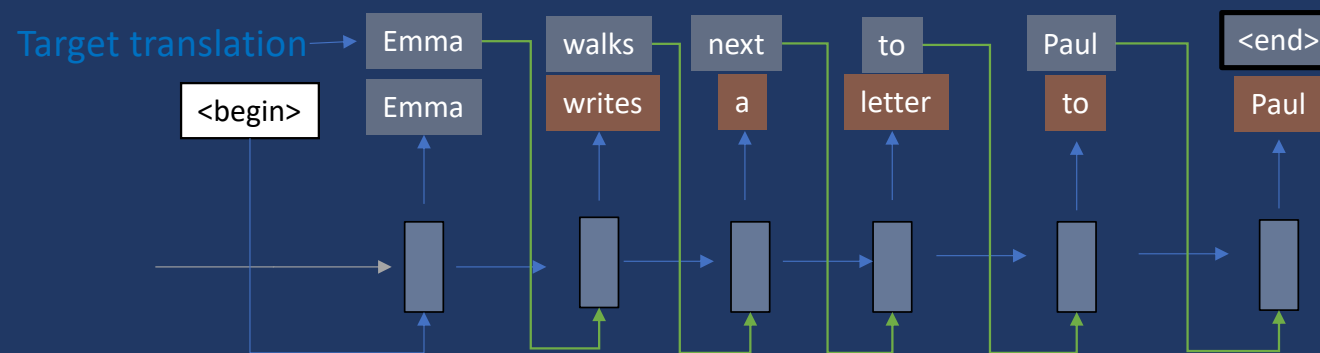
Prediction errors propagate as input for the next decoding step!  
This can slow down training or could even lead to a diverging model.

# Seq2Seq: Teacher Forcing

## Teacher Forcing:

During training, we already know the correct target words!

→ Feed those to the decoder instead of the previous prediction

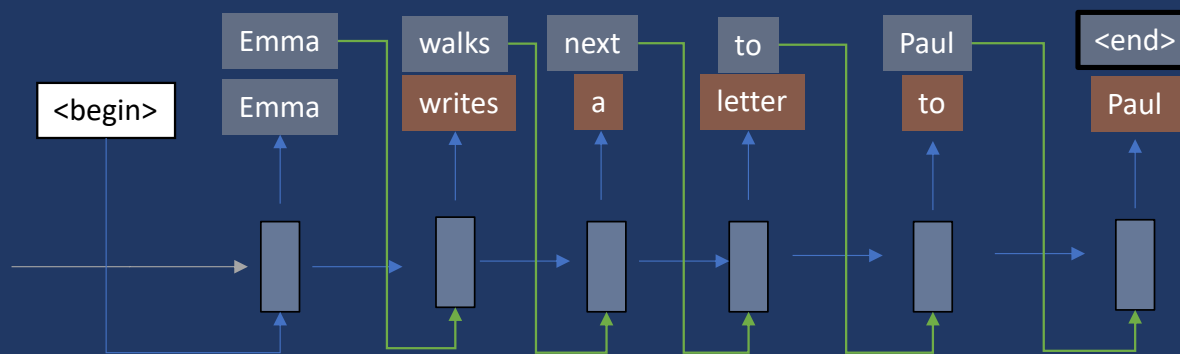


# Seq2Seq: Teacher Forcing

- Problem of Teacher Forcing:
  - During training, previous “output” is always correct  
→ Model relies too much on it!
  - During prediction, the network can not handle wrong inputs



Start training with teacher forcing, then switch to supplying the real outputs.



# Seq2Seq: Improved Sampling

---

- Remember:
  - Decoder returns a (local) probability distribution over target words at each timestep
  - We want to find the most likely sequence **globally**
- So far, we just selected the most likely word
  - **Greedy Search**
- Alternative: Consider multiple options and decide later
  - **Beam Search**



# Enhancing the Model

---

- Encoder – Decoder is a very general and flexible model
- We can change individual parts separately to tune the model for a task.

→ Let's take a look at some improvements!

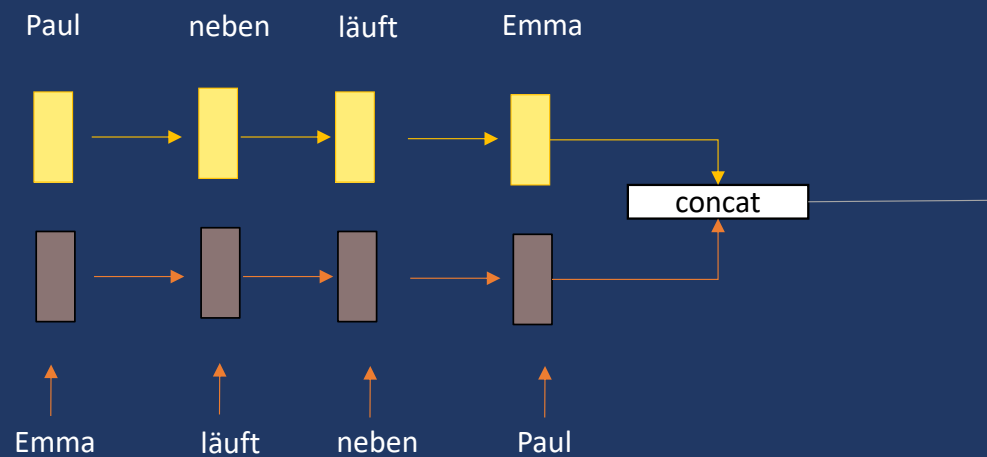
# Enhancement: BiRNN Encoder

## Problem:

Long sequences can cause the sequence representation to lose information about the first steps.

## Solution:

Additionally feed the reverse input sequence into a separate RNN and concatenate the output.



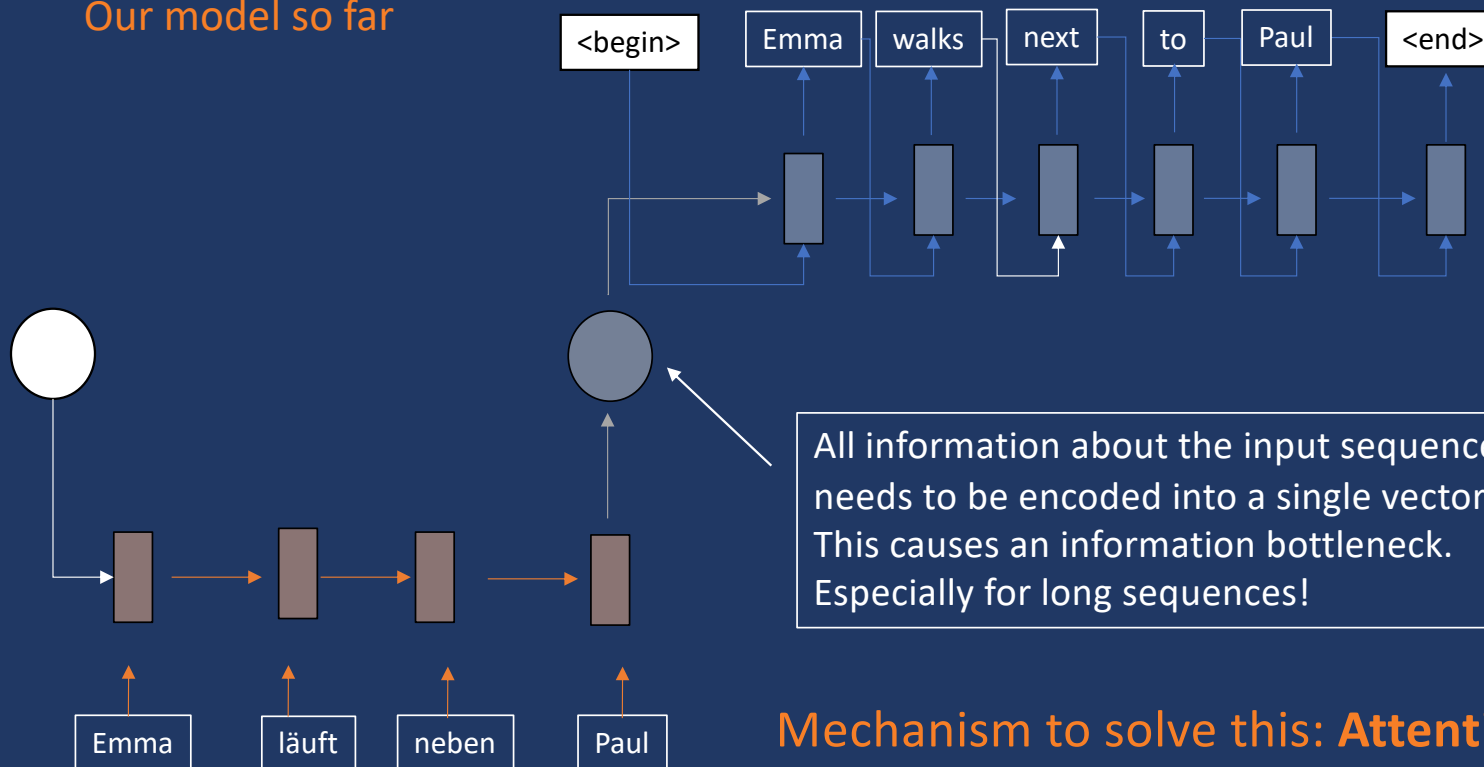
## 5.3 Model Attention

---

- Idea
- Modelling
- Evaluation

# Enhancement: Attention!

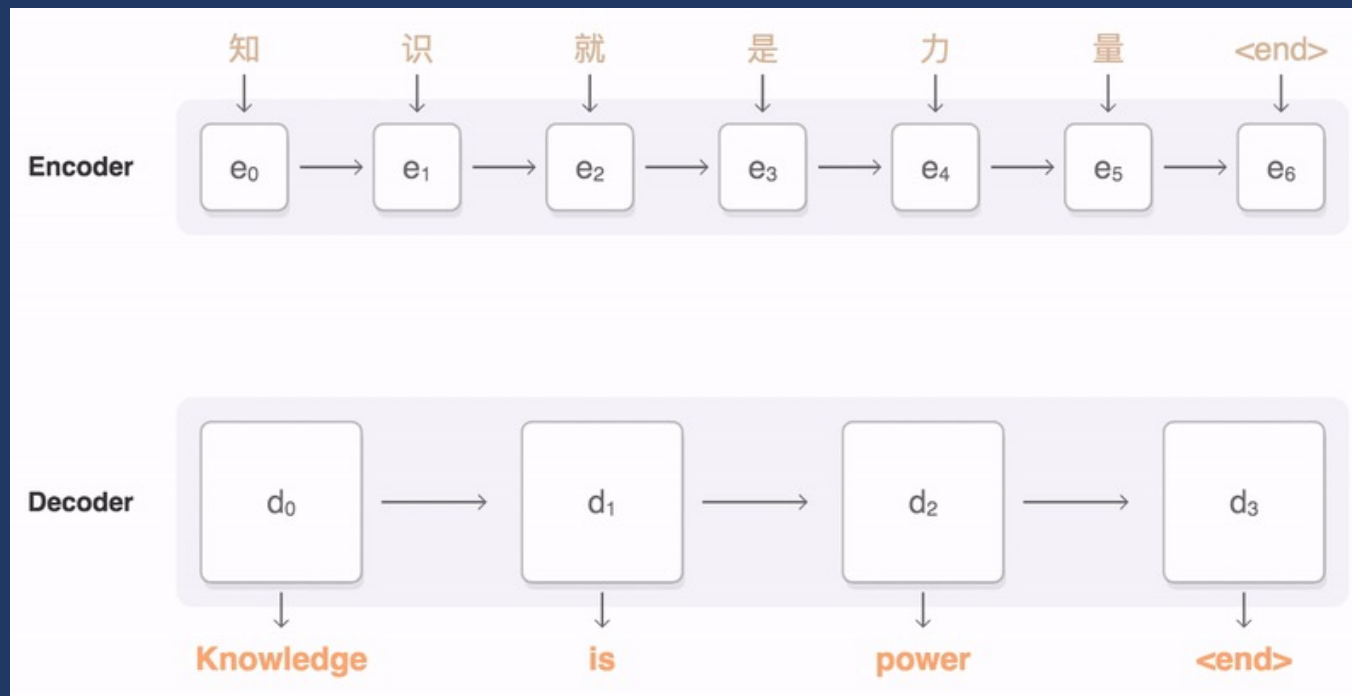
Our model so far



All information about the input sequence needs to be encoded into a single vector. This causes an information bottleneck. Especially for long sequences!

Mechanism to solve this: **Attention**

# Attention: Idea



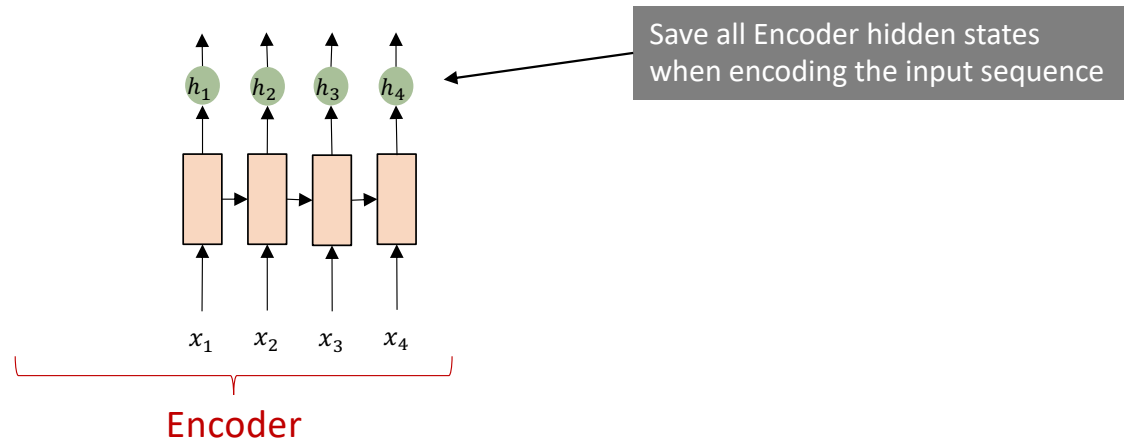
Animation from: <https://github.com/google/seq2seq>

## Attention: Idea

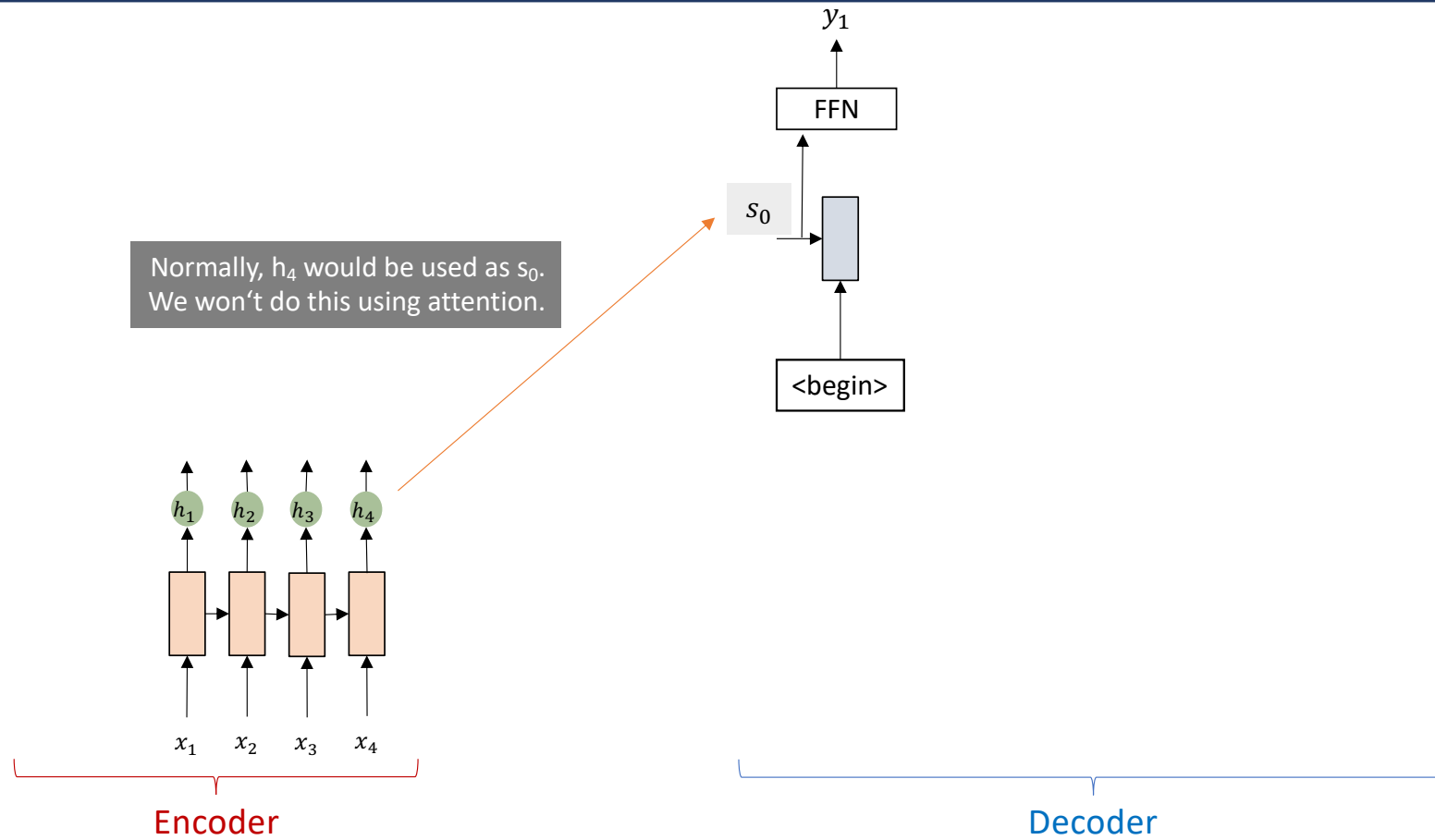
---

- Not just the last Encoder hidden state → save all Encoder hidden states
  - For every decoding step, find a weighting of the Encoder hidden states depending on the Decoder hidden state
  - Calculate a weighted sum of Encoder hidden states and use it in the Decoder
- Encoder can capture per-step information; no need to squeeze everything into one hidden state
- Decoder can pay attention to the important Encoder hidden states

# Attention

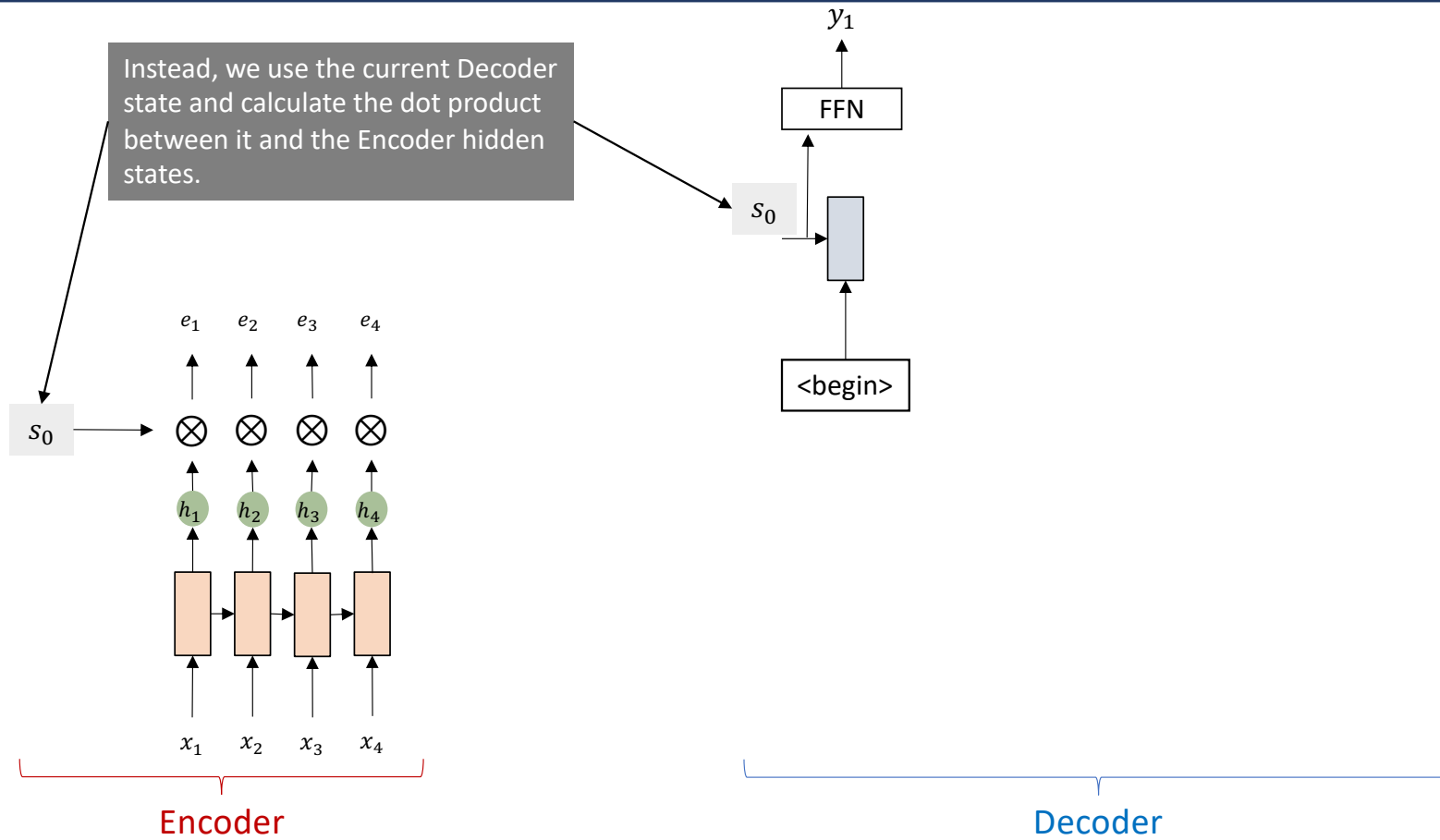


# Attention



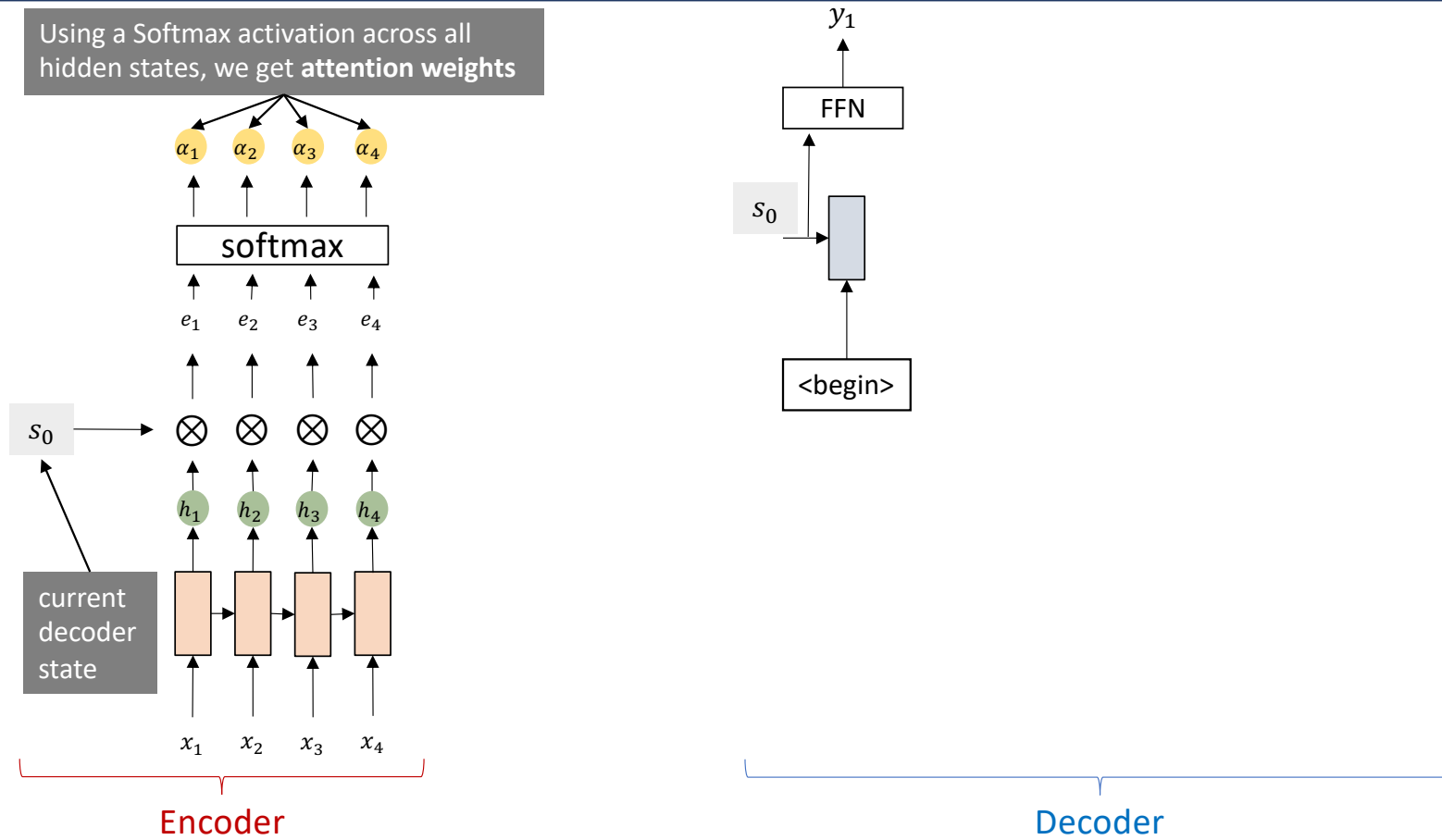


# Attention



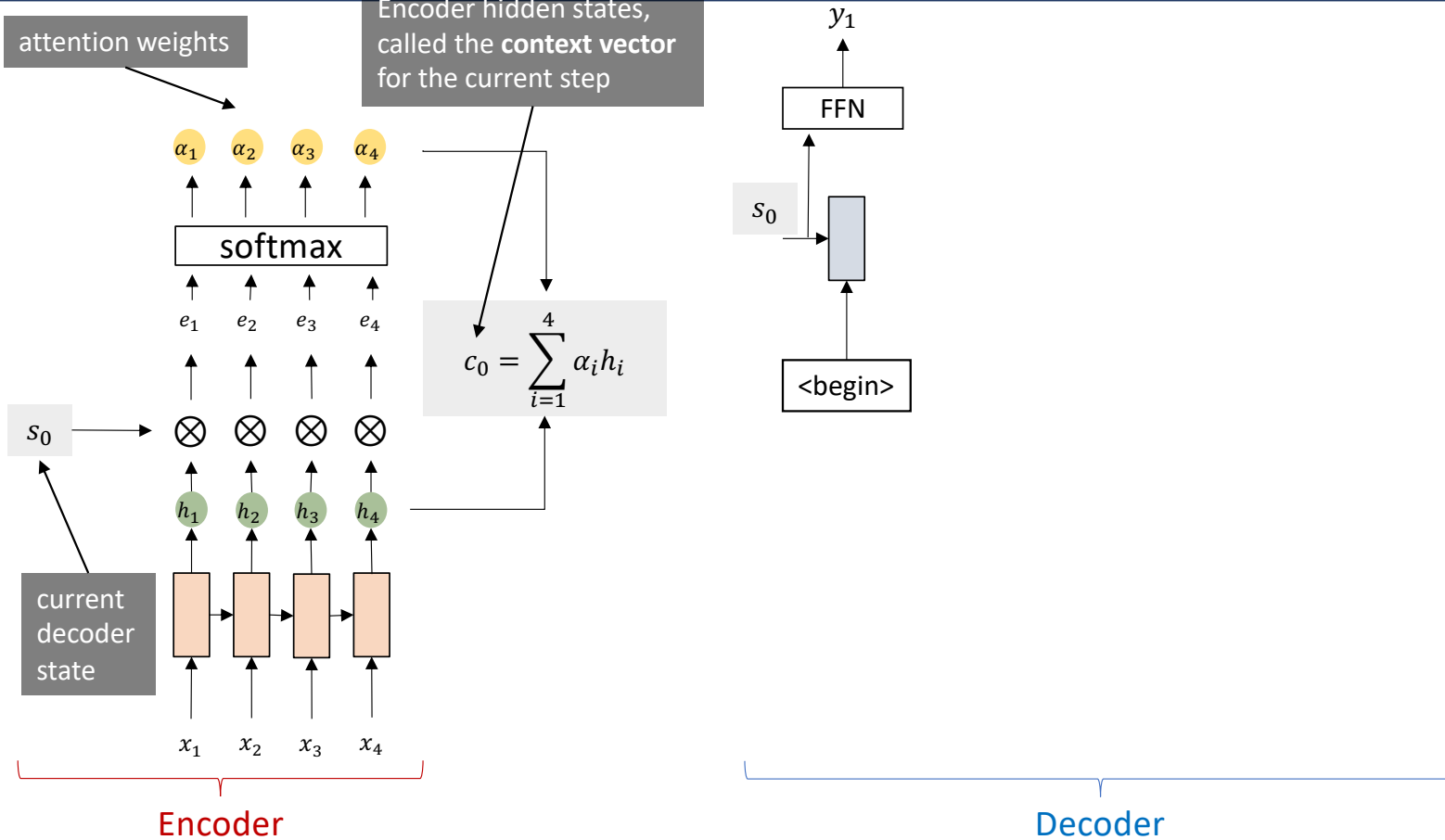
# Attention

Using a Softmax activation across all hidden states, we get **attention weights**

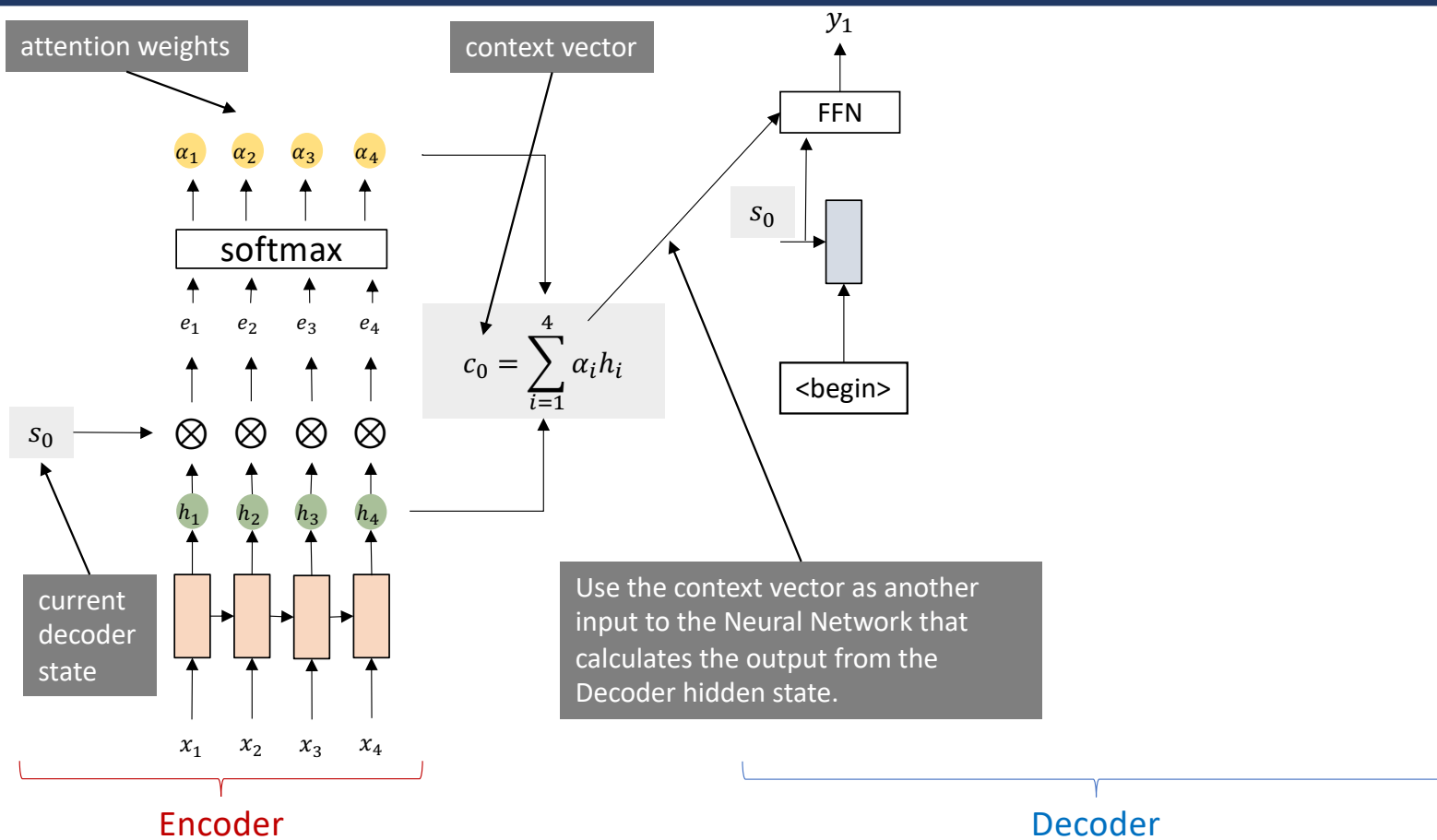


# Attention

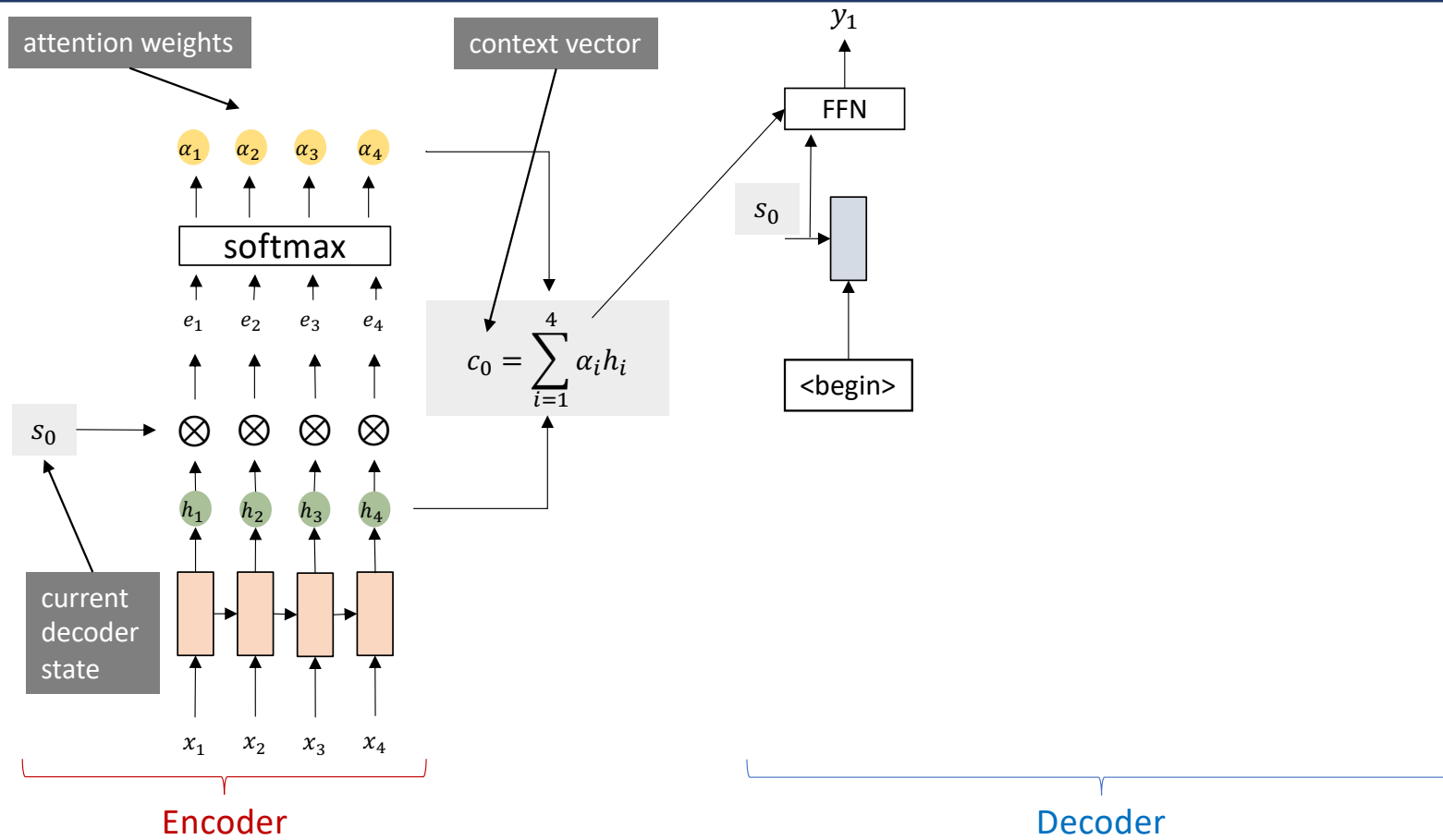
These weights are then used to calculate a weighted sum of the Encoder hidden states, called the **context vector** for the current step



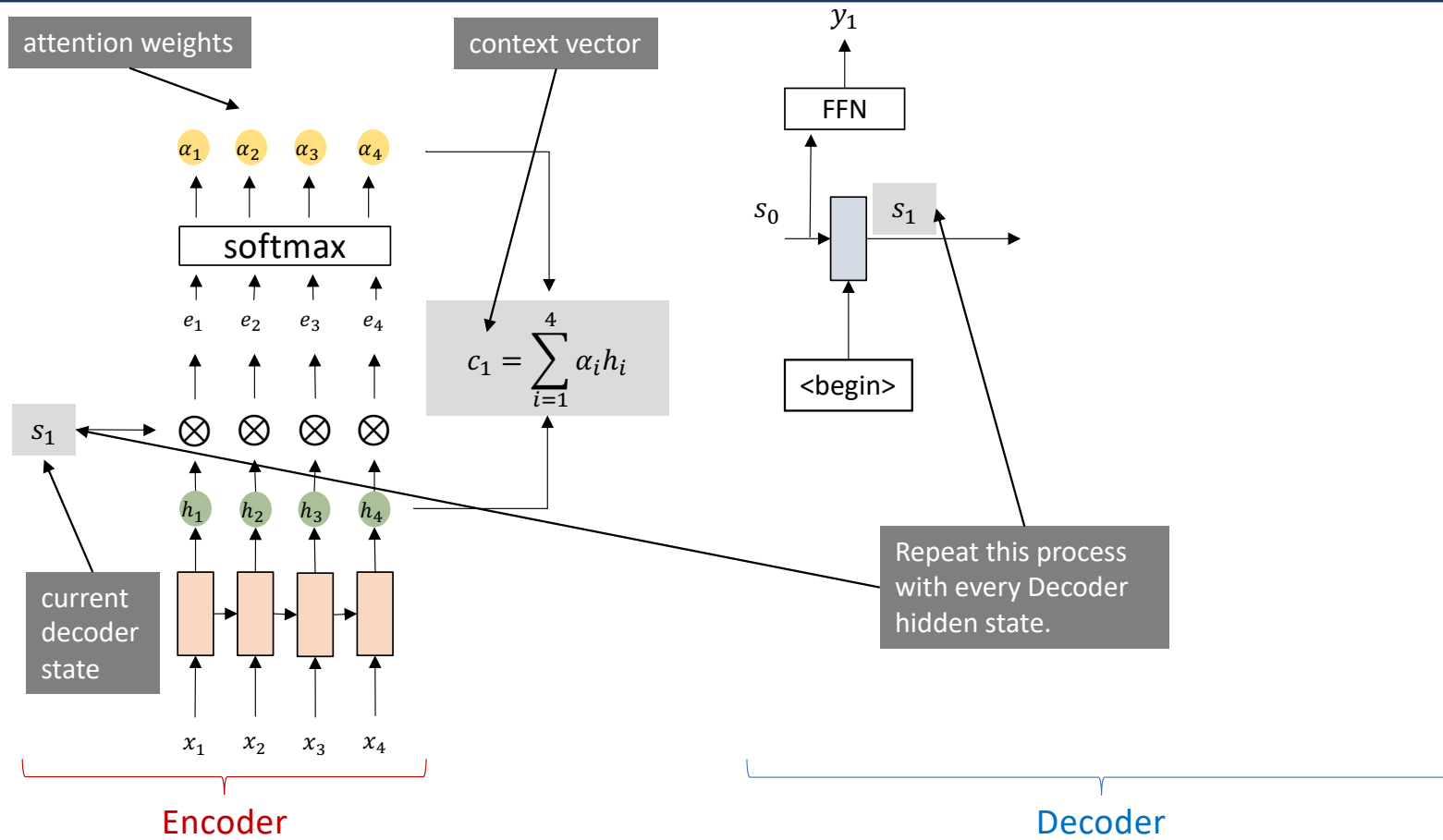
# Attention



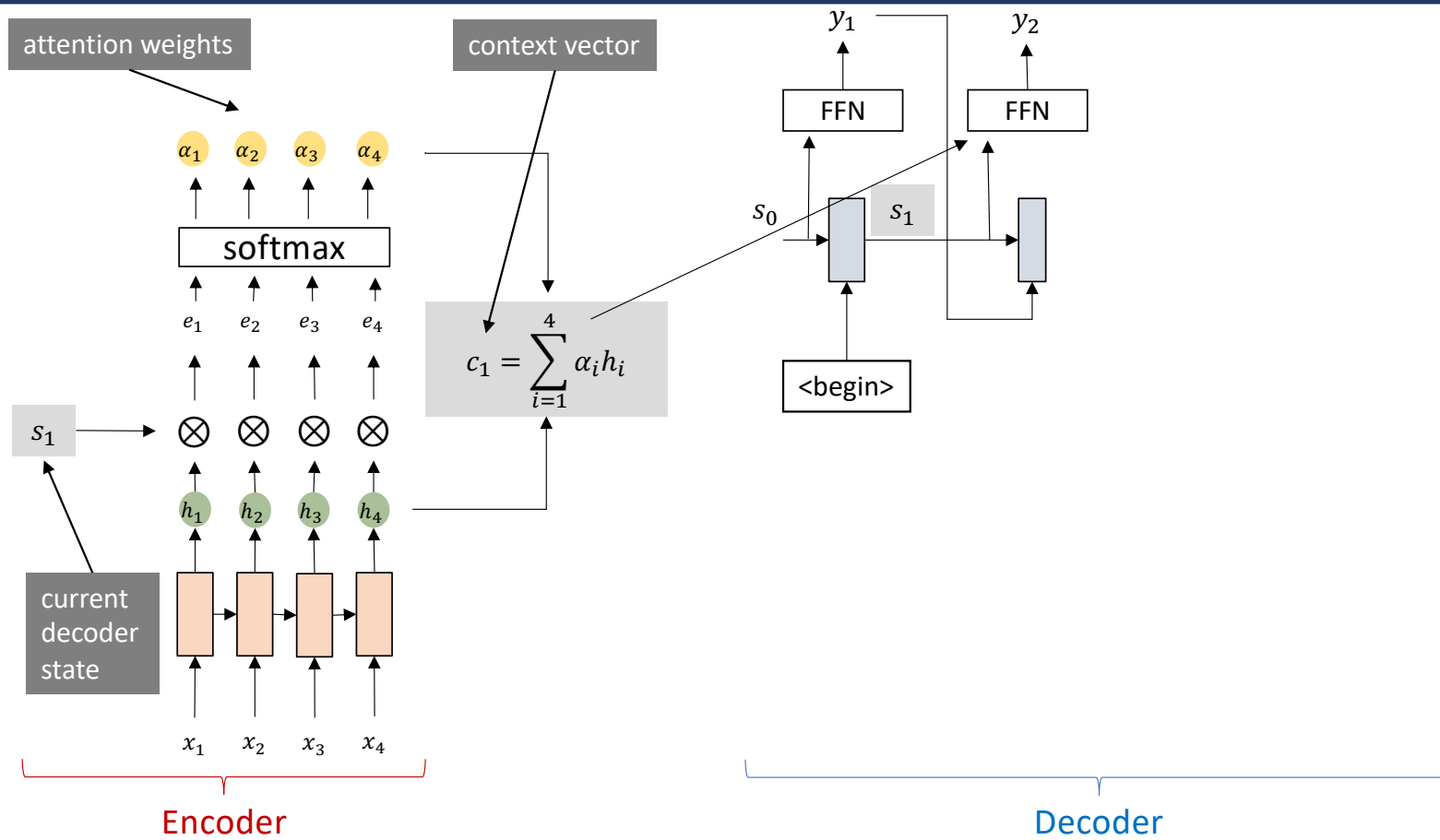
# Attention



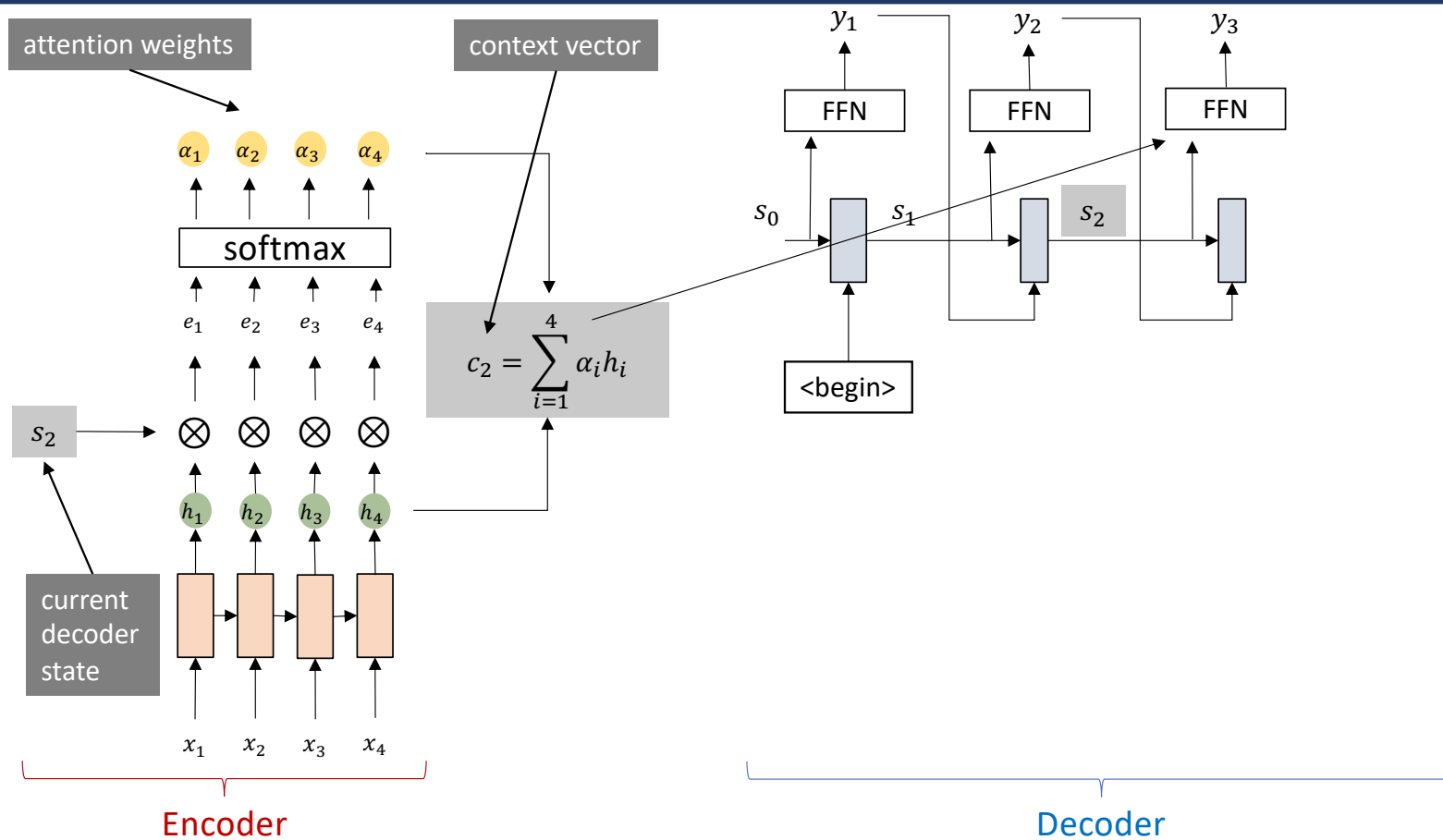
# Attention



# Attention



# Attention





# Attention

- A context vector  $c_i$  is computed as a weighted sum over  $k$  encoder states at every decoding step  $i$ :

$$c_i = \sum_{j=1}^k \alpha_{ij} h_j$$

- Where  $\alpha_{ij}$  is a scalar weighting factor computed as:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{l=1}^k \exp(e_{il})}$$

- And  $e_{ij}$  is an alignment model:

$$e_{ij} = a(s_{i-1}, h_j)$$

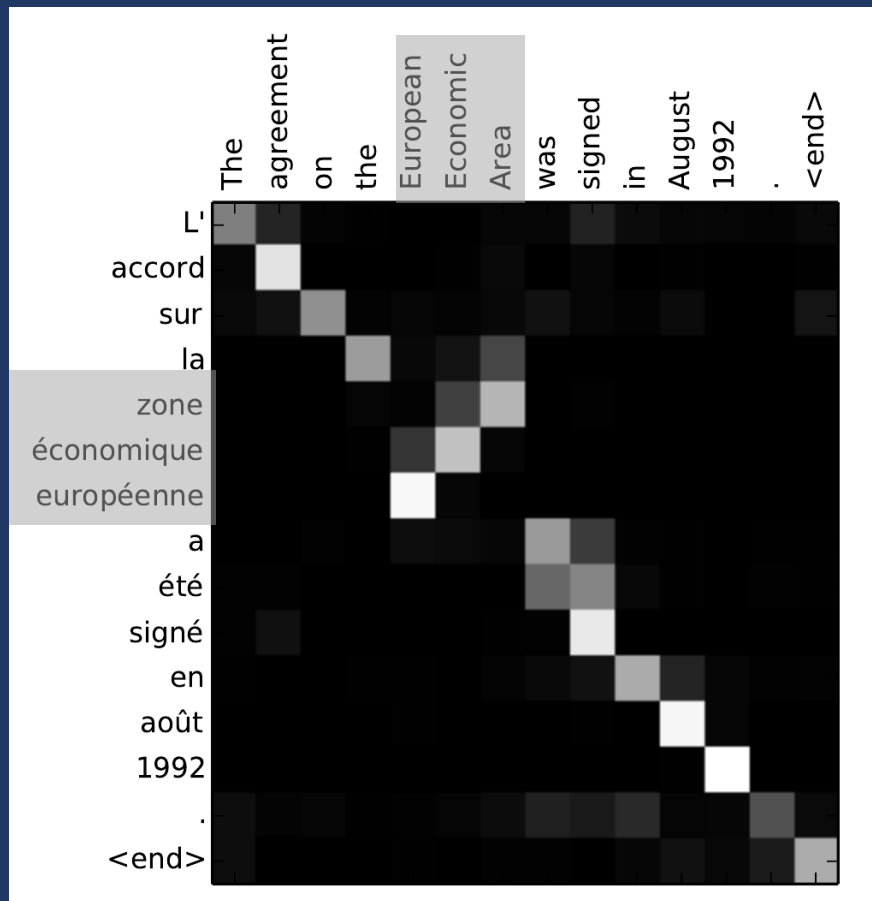
where  $a$  can be any transformation (e.g. a dot product)

# Attention

---

- Allows the model to distribute information across encoder states and extract only necessary bits at every decoder step.
- The attention weights can be visualized to allow insights into the models behaviour.

# Visualising Attention



Here some words are in a different order.

The plotted weights show how the attention shifts accordingly.

# Was that useful?

---

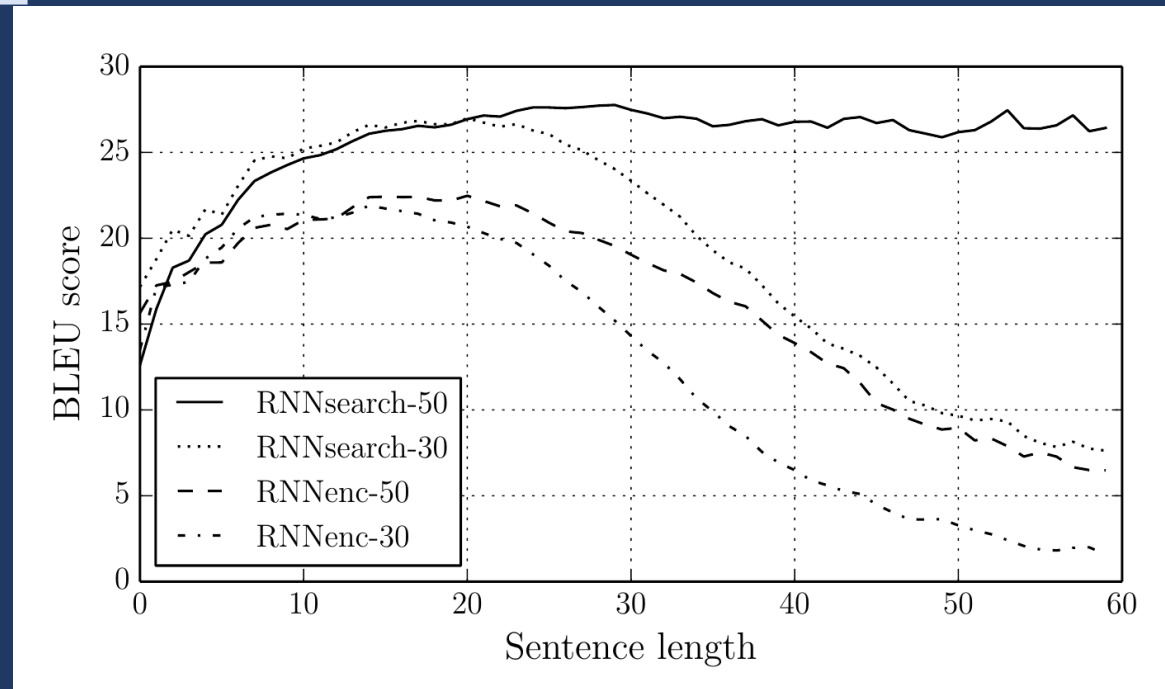
- We now know a lot about techniques for machine translation
  - We also know many „improvements“ over the vanilla model
  - But are they really useful?  
How well do our models perform?
- We need some way to measure translation quality!

# Evaluating Machine Translation — BLEU

---

- Papineni et al, 2002: **Bilingual evaluation understudy (BLEU)**
  - In practice: Use Fourgram Precision
  - Use lots of target translations
  - **Do not evaluate single sentences!**  
BLEU only works well for large corpora
- Now how well does the Attention model perform?

## Attention: Results (Translation / Bahdanau et. al.)



All models are trained to translate sentences of length 30/50

- **RNNsearch-k** is an attention model trained on sentences of length k
- **RNNenc-k** is a normal seq2seq model trained on sentences of length k