Carnegie Mellon University
Language
Technologies
Institute

11-411/11-611 Natural Language Processing

Deep Learning for Sequence Processing

Lori Levin, David Mortensen and Suyash Chavan

March 16, 2023

Language Technologies Institute

Learning Objectives

At the end of this lecture, students should be able to:

- Describe Simple RNNs, Stacked RNNs, Bidirectional RNNs, and LSTMs
  - Understand what properties of language and what NLP tasks require these models
- Apply Encoder-Decoder models to tasks like machine translation.
- Describe attention mechanisms as they apply to LSTMs and Transformers
- Apply Transformers to sequence-modeling tasks

Overview

We will discuss three classes of models in this lecture: RNNs, LSTMs, and Transformers.

	Variable-Length Input	Recurrence	Self-Attention
FFNN	✗	✗	✗
Simple RNN	✓	✓	✗
LSTM	✓	✓	✗
Transformer	✓	✗	✓

Review: why do we need all of these different neural models?

- Long distance dependencies in natural language
- When left context isn’t enough

Long distance dependencies

- Something in a sentence depends on something that is more than a few words away
  - problem for n-gram language models
  - problem for vanishing influence of prior context

Long distance: adding a modifier to the subject makes it far away from the verb it agrees with

- The **teachers** **is/are** happy.

- The **teachers** who spoke to the student about the project **is/are** happy.
- The **teachers** in the large classroom in Gates-Hillman **is/are** happy.

Verb-final languages: the subject is always far from the verb

```
Ladki ladke ko    dekh       rah-i      hai.
girl  boy   CASE  see.PART   PROG-FEM   COP
''The girl sees a boy.''

Ladka ladki ko    dekh       rah-a      hai.
boy   girl  CASE  see.PART   PROG-MASC  COP
''The boy sees a girl''
```

Hindi, written in Roman characters.

Long distance: when there needs to be a “gap”

Consider the following sentences

- You wrote long stories about _____ last year.
  - You wrote long stories about **ø** last year.
  - You wrote long stories about **dragons** last year.

Long distance: when there needs to be a “gap”

Consider the following sentences

- You wrote long stories about _____ last year.
  - You wrote long stories about **ø** last year.
  - You wrote long stories about **dragons** last year.

- **What did** you write long stories about ____ last year?
  - **What did** you write long stories about **ø** last year?
  - **What did** you write long stories about **dragons** last year?

Left context is not enough: resolving ambiguity

- [**The old man**] is rowing.
  - "man" is a noun
- **The old [man** the boats.]
  - "man" is a verb

- **I seeded** the field.
  - put seeds on
- **I seeded** the watermelon.
  - took seeds out

- **I know that**.
  - "that" is a pronoun
- **I know that** student.
  - "that" is a determiner
- **I know that** you yawned.
  - "that" is a complementizer (a word that introduces an embedded sentence).

LSTMs and Bi-LSTMs

Comparing Simple RNNs to LSTMs

Simple RNN

- **Recurrent**
- Hidden state at time *t* is a function of the **input** and the **hidden state at time *t-1***
- Input is combined with previous hidden state using **element-wise addition**.
- **Severe vanishing gradient problem**

Long Short-Term Memory (LSTM)

- **Recurrent**
- Hidden state at *t* is a function of **input**, **hidden state at *t-1*** and **context vector**
- Input is combined with preceding hidden state and context vector using **articulated gating mechanism**
- **Reduced vanishing gradient problem**

LSTMs Add Cells

Three gates in each cell:

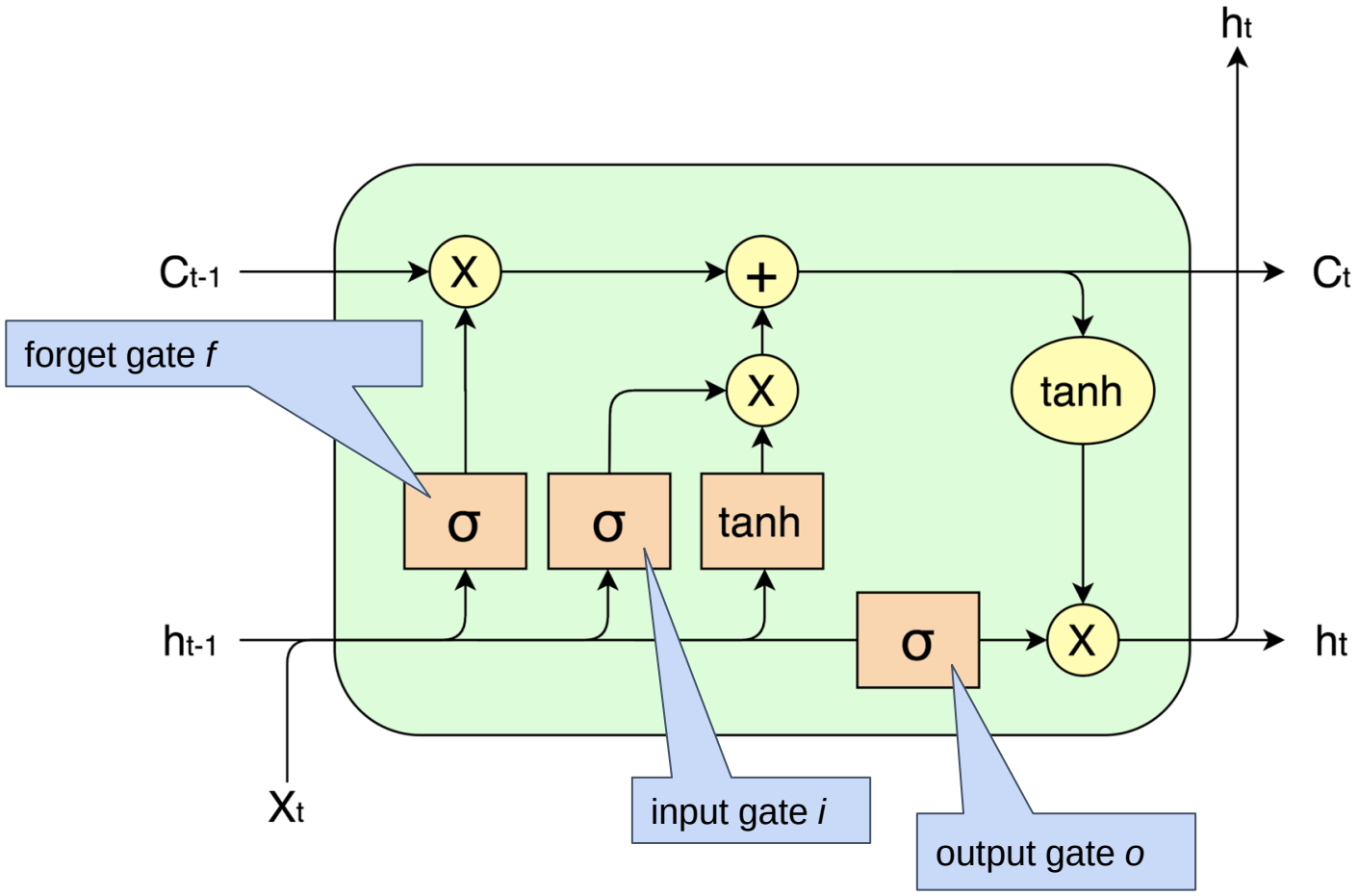
1. Input gate
2. Output gate
3. Forget gate

These control the flow of information used for prediction and make LSTMs better than RNNs at handling the kinds of long-distance patterns we discussed previously.

LSTMs Use a Context Layer

- In parallel with the hidden layer *h*, LSTMs have a context layer *C* (of context vectors $C_1,...,C_n$) that carries a certain kind of information across timesteps.
- The context vector indicates what gets passed on about the current input and how much the current input will continue to contribute to updating the hidden state.

LSTMs Have an Articulated Internal Structure



Operations:

- + is element-wise addition
- × is element-wise multiplication

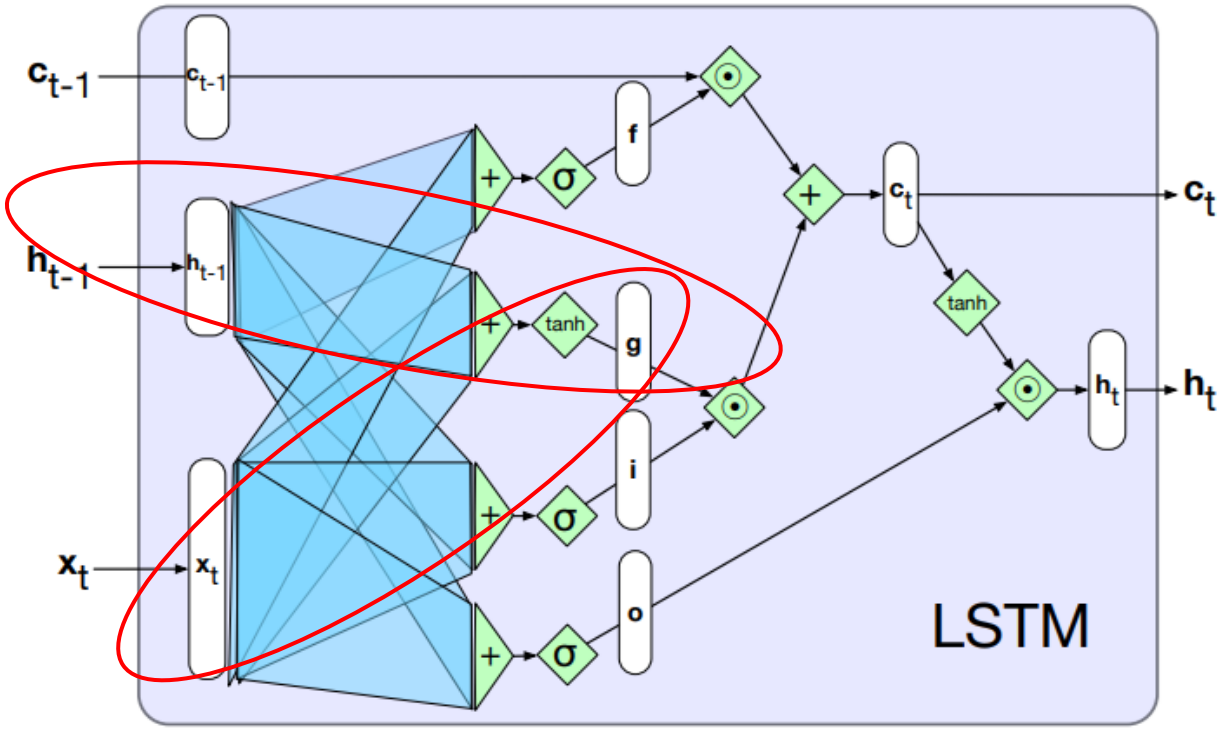
Non-linear functions

- σ is sigmoid
- tanh is hyperbolic tangent

Vectors

- C_{t-1} is the context vector
- h_{t-1} and h_t are the hidden layer at the previous and current timestep
- X_t is the input (at the current timestep)

Do the usual RNN thing with the current input and the hidden state for the previous time step, and their weight matrices



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

U is a weight matrix that is multiplied with the hidden layer of the previous time step.

W is a weight matrix that is multiplied by the current input.

Use tanh as the activation function.

The result is called g in the diagram.

Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

How to forget

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

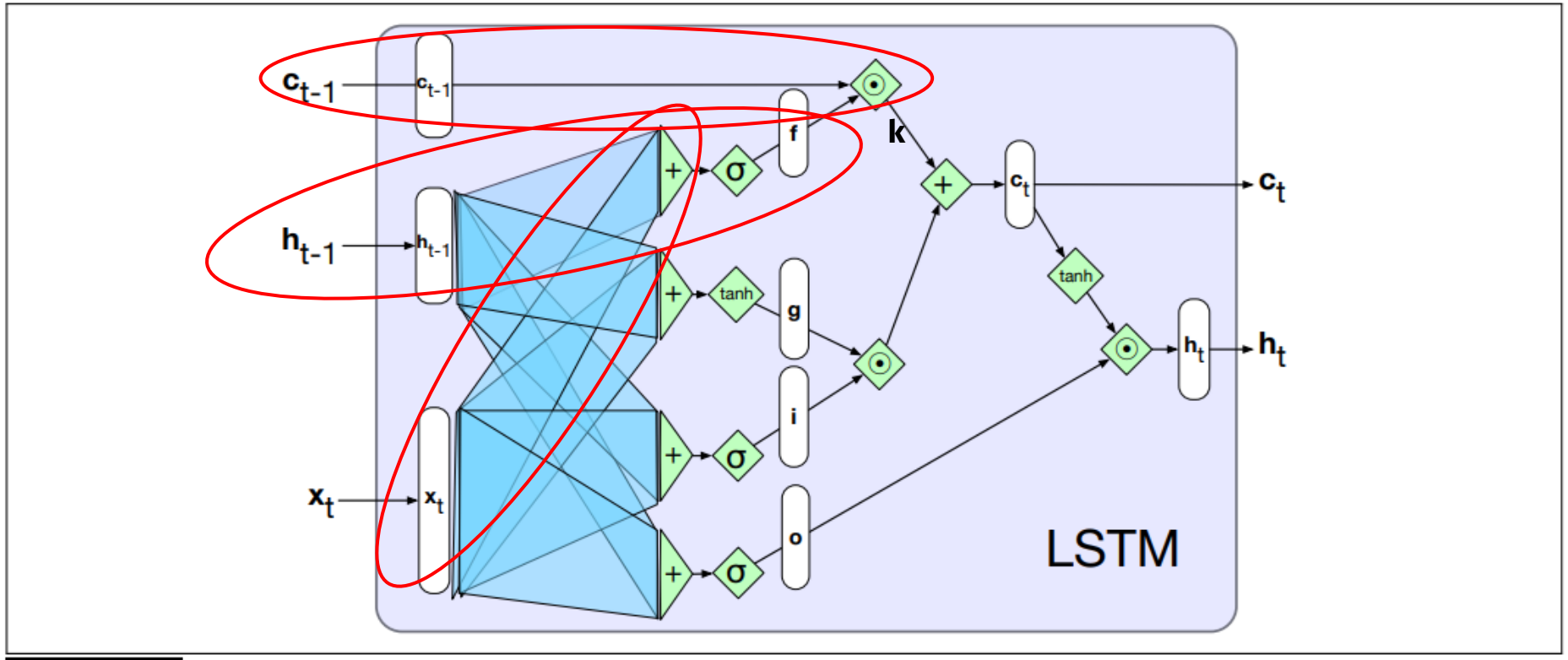


Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

U is a weight matrix that is multiplied with the hidden layer of the previous time step.

W is a weight matrix that is multiplied by the current input.

Use sigmoid as the activation function in order to push outputs toward 0 or 1.

This results in a “mask”, shown as f in the diagram.

Multiply the mask by the context vector from the previous time step, resulting in a vector k.

How to remember

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

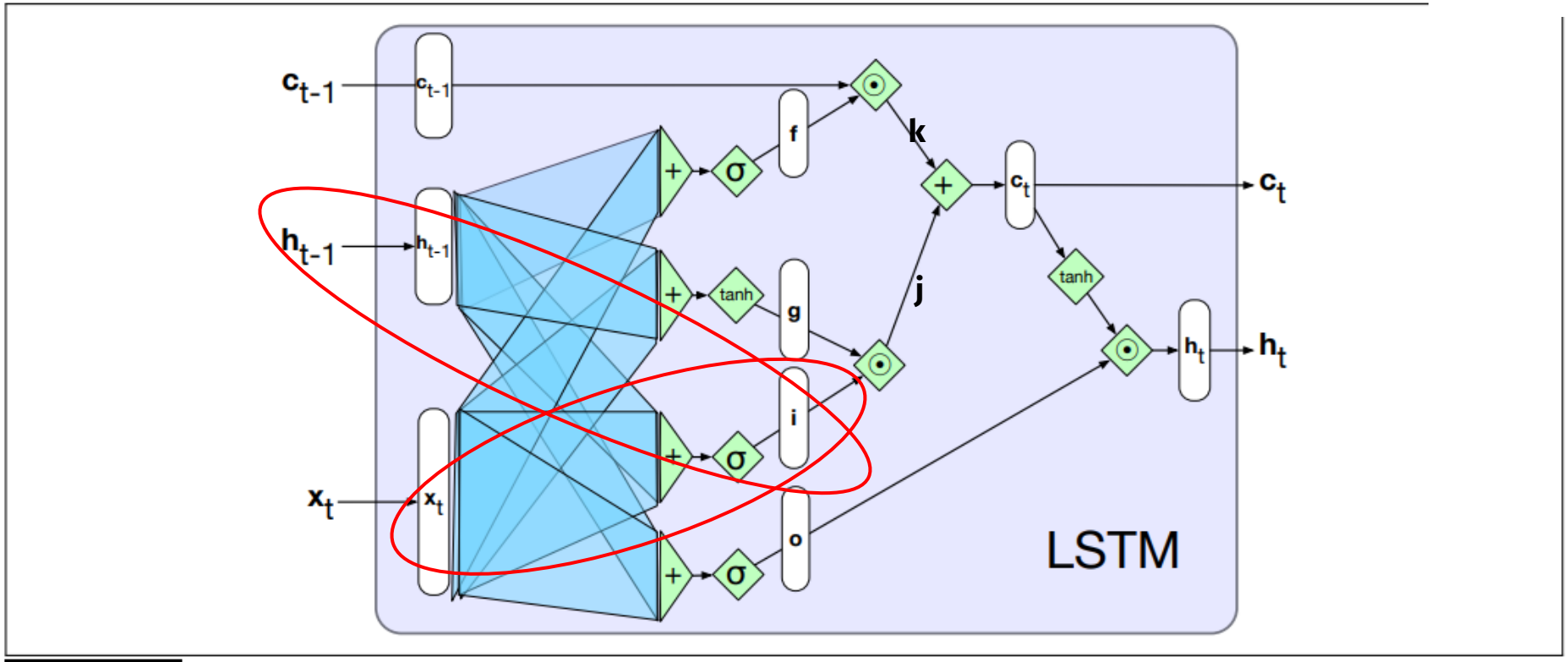


Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

U is a weight matrix that is multiplied with the hidden layer of the previous time step.

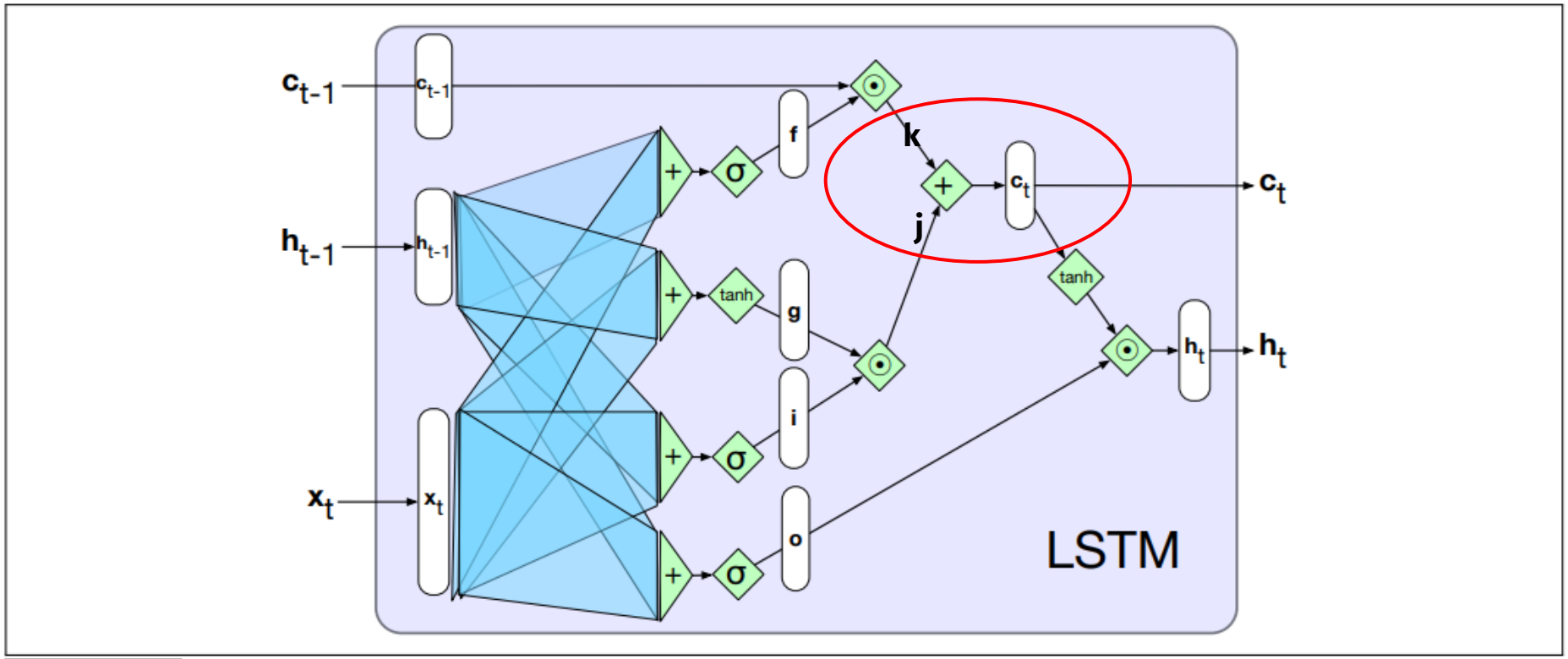
W is a weight matrix that is multiplied by the current input.

Use sigmoid as the activation function.

The result is called i in the diagram.

Multiply i times g to get a vector j of things to remember.

How to make the context vector for timestep t



$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

How to make the output gate

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

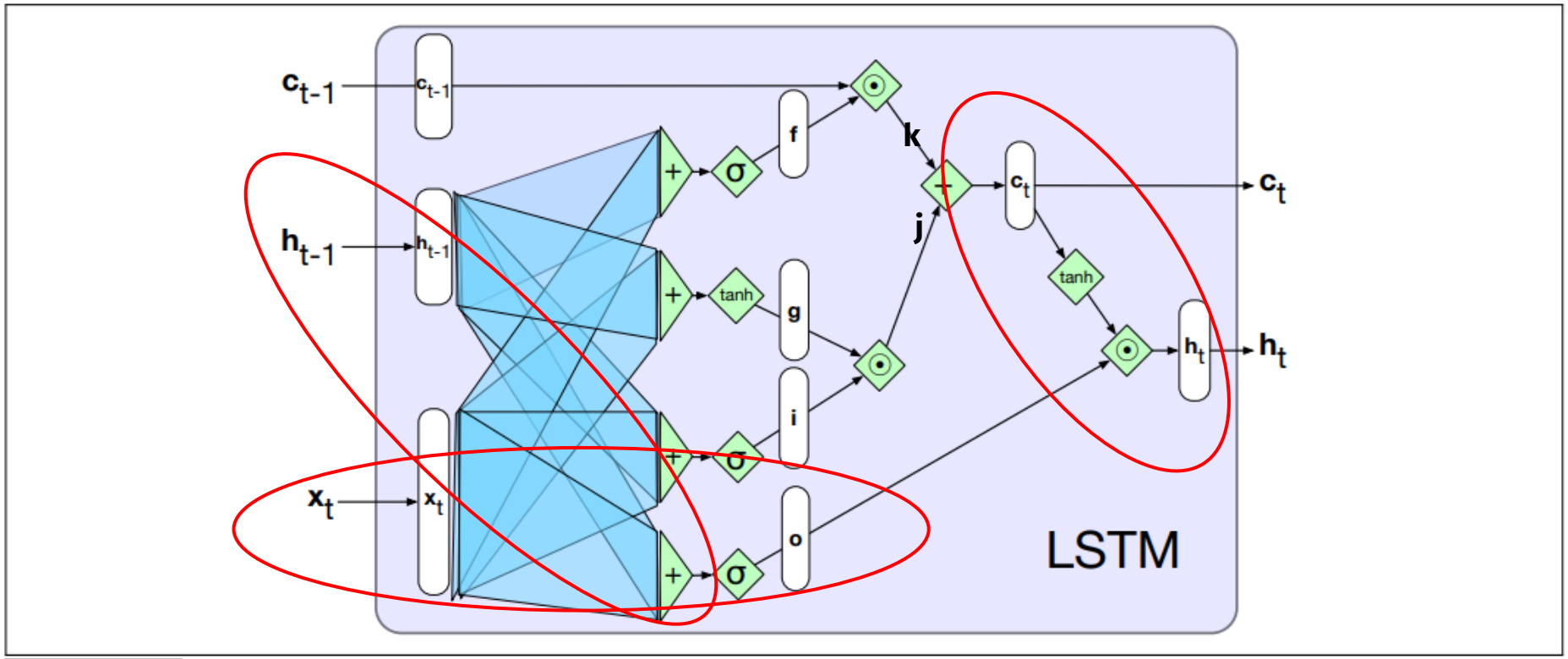


Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

The output gate is used to decide what information is required for the hidden state.

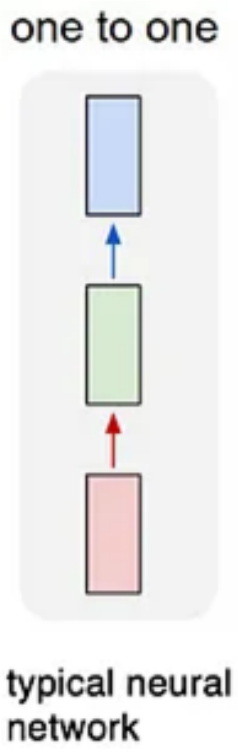
The context vector preserves information for future decisions.

Apply tanh to the context vector, and then multiply that by the output gate to get the hidden state.

Output the hidden state and the context for use in the next timestep.

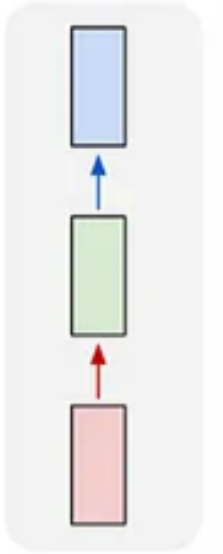
Deep Learning Architectures for Sequence Modelling

We have seen regular Neural Networks ....



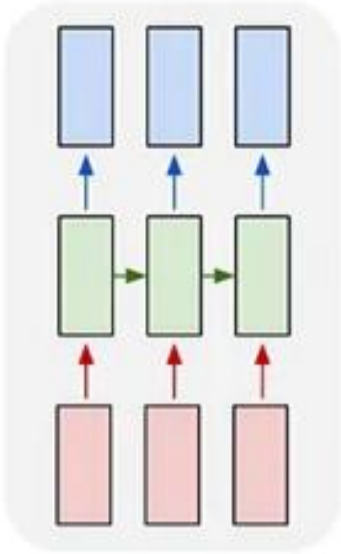
… and Regular RNNs

one to one

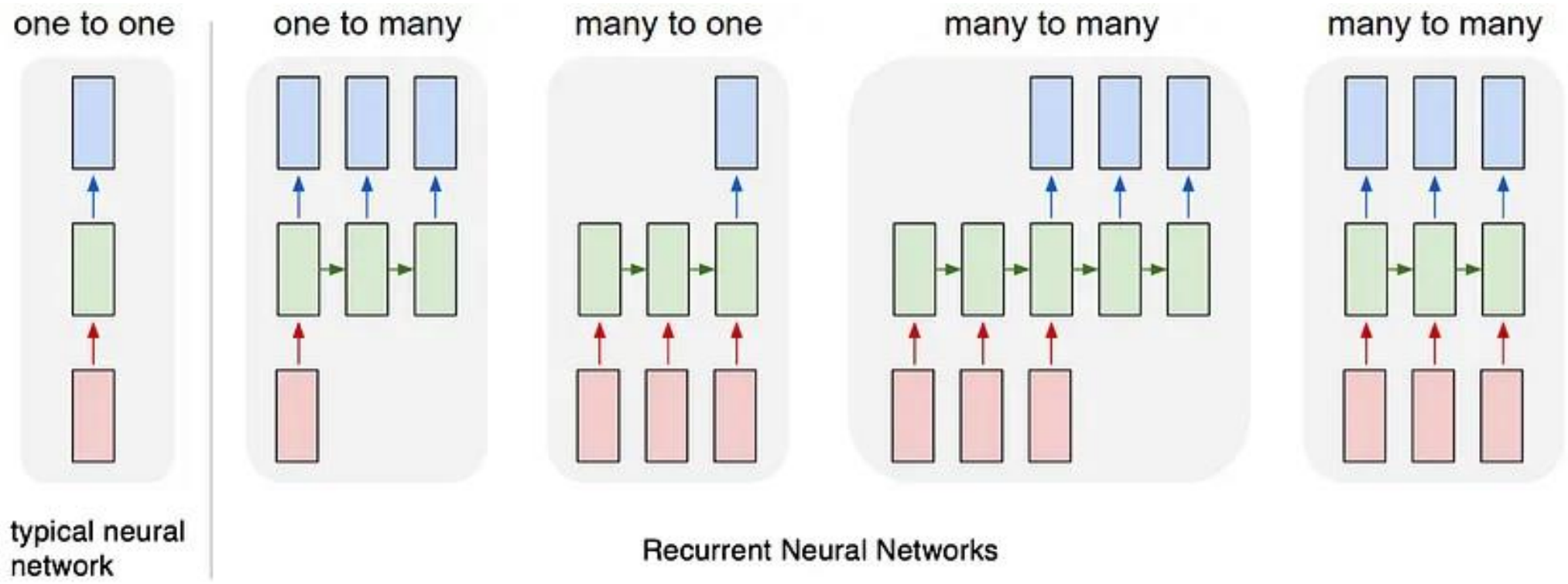


typical neural network

many to many



But these can be modified for many use-cases



Encoder-Decoder (Sequence to Sequence) Models

Encoder-Decoder (Sequence to Sequence) Models in a Nutshell

Encoder-Decoder models take a sequence as input, encode it as a vector or other real-valued representation, and then decode it into another sequence.

Example—

Input: Ich bin auch ein Berliner [German]

[vector]

Output: I am also a Berliner [English] *or*

I am also a jelly doughnut* [English]

Originally, this was done with RNNs (then LSTMs)

*A Berliner is not *exactly* a jelly doughnut but a sweet pastry filled with jam and covered with powdered sugar.

Tasks Other than Machine Translation

Encoder-Decoder models can potentially be used for any task that involves converting one sequence/string into another:

- Morphological analysis (coverts strings of characters into string of characters [lemmas] and sequences of morphological tags)
- Summarization (coverts a sequence of words [the original document] into a shorter sequence of words [the summary])
- Dialog (converts a sequence of words [the preceding turn] into another sequence of words [the following turn])
- And so on…

An Overview of the Encoder-Decoder Model for MT

The following slides will show a high-level overview of Machine Translation using an RNN

- At each time step, the RNN will consume an encoded word
- The state, at that step, will be a function of the embedding of the word and the state at the preceding step
- At the end of the input (the German sentence) the RNN is fed an <s> token—it's showtime
- The RNN then starts generating English outputs based on the preceding hidden state and the preceding output, until the end token </s> is reached

An Overview of the Encoder-Decoder Model for MT

ENCODER

DECODER

An Overview of the Encoder-Decoder Model for MT

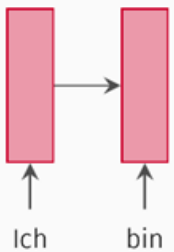
ENCODER

DECODER

An Overview of the Encoder-Decoder Model for MT

ENCODER

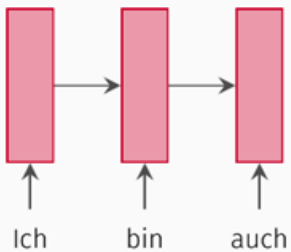
DECODER



An Overview of the Encoder-Decoder Model for MT

ENCODER

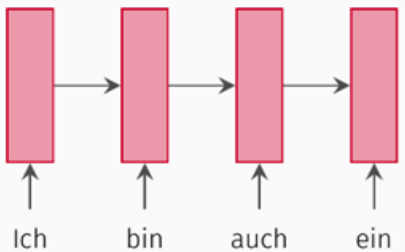
DECODER



An Overview of the Encoder-Decoder Model for MT

ENCODER

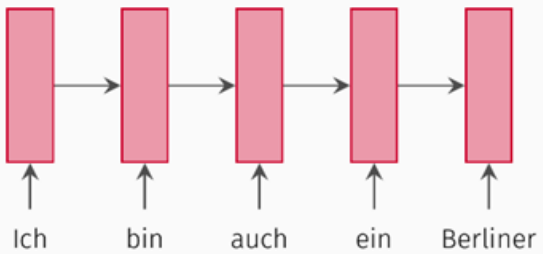
DECODER



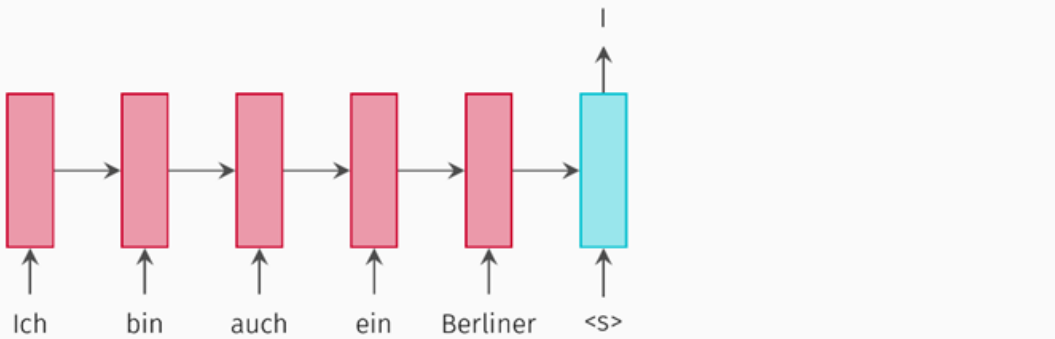
An Overview of the Encoder-Decoder Model for MT

ENCODER

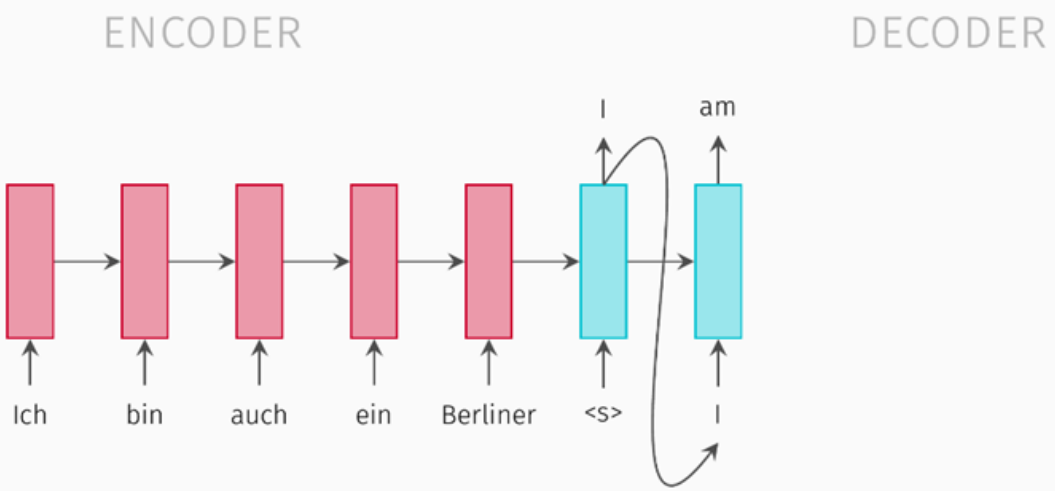
DECODER



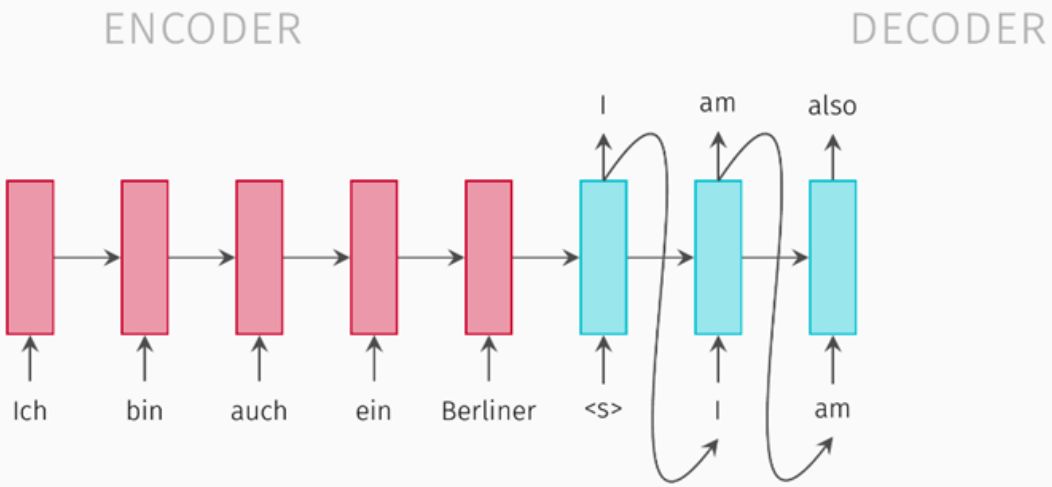
An Overview of the Encoder-Decoder Model for MT



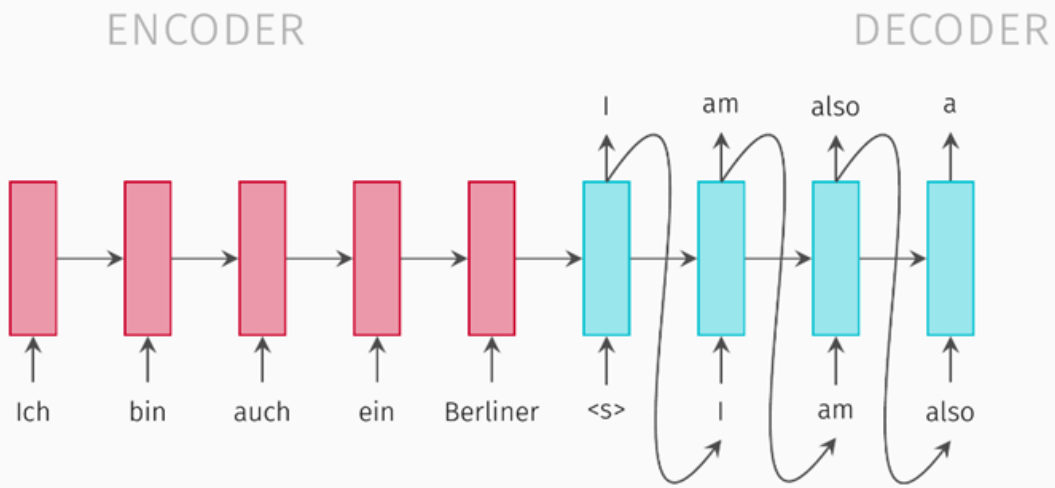
An Overview of the Encoder-Decoder Model for MT



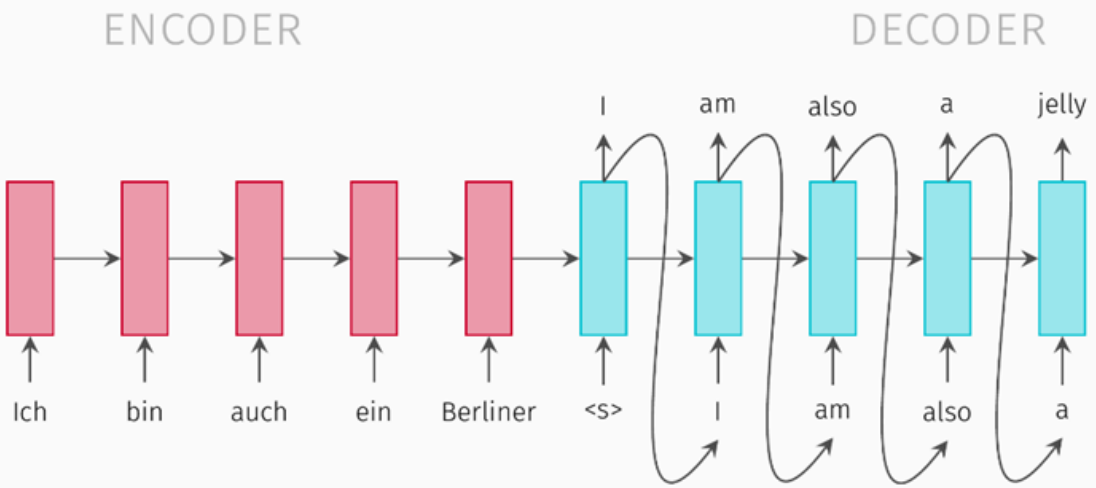
An Overview of the Encoder-Decoder Model for MT



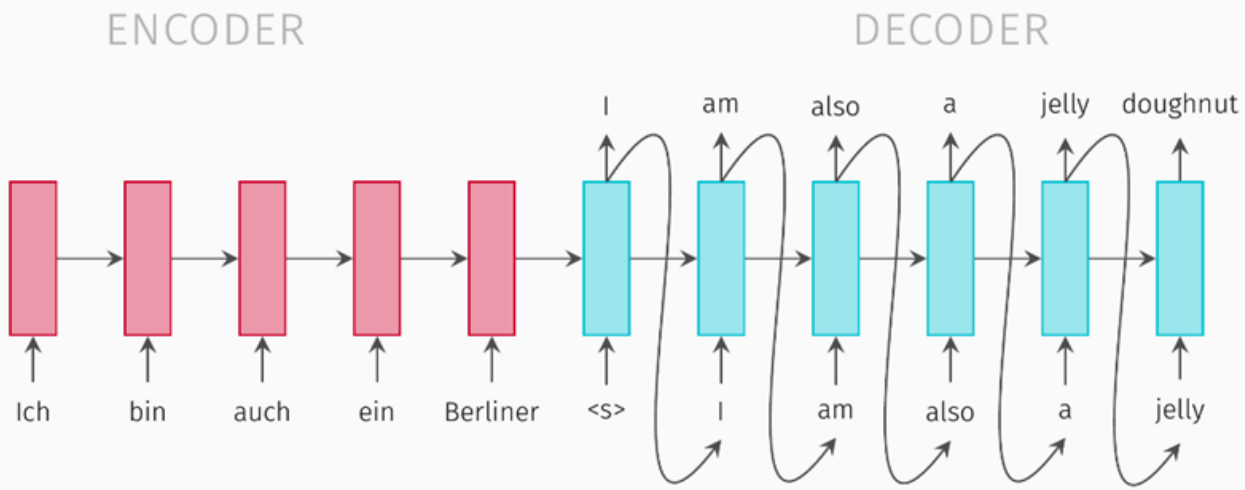
An Overview of the Encoder-Decoder Model for MT



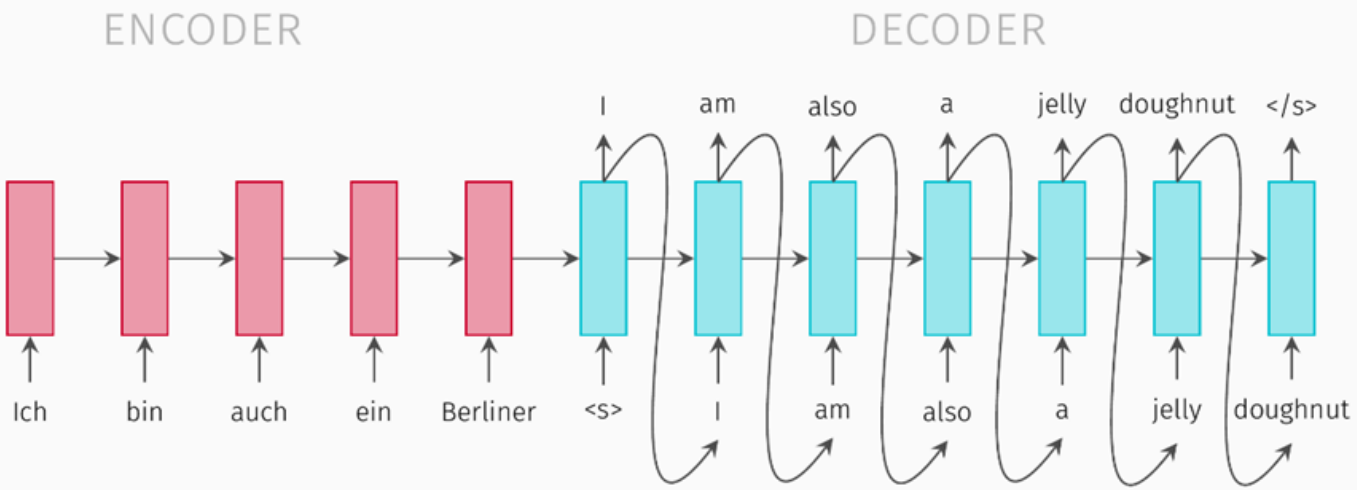
An Overview of the Encoder-Decoder Model for MT



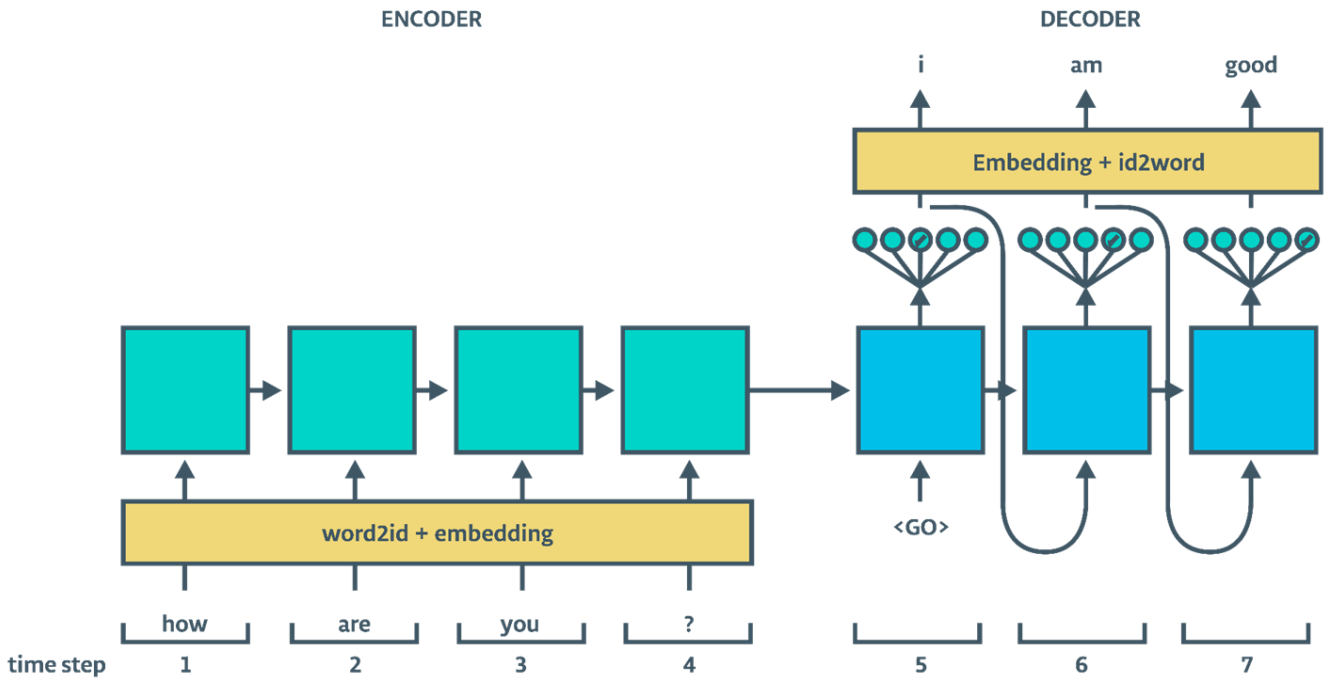
An Overview of the Encoder-Decoder Model for MT



An Overview of the Encoder-Decoder Model for MT



The Same Thing, in More Detail



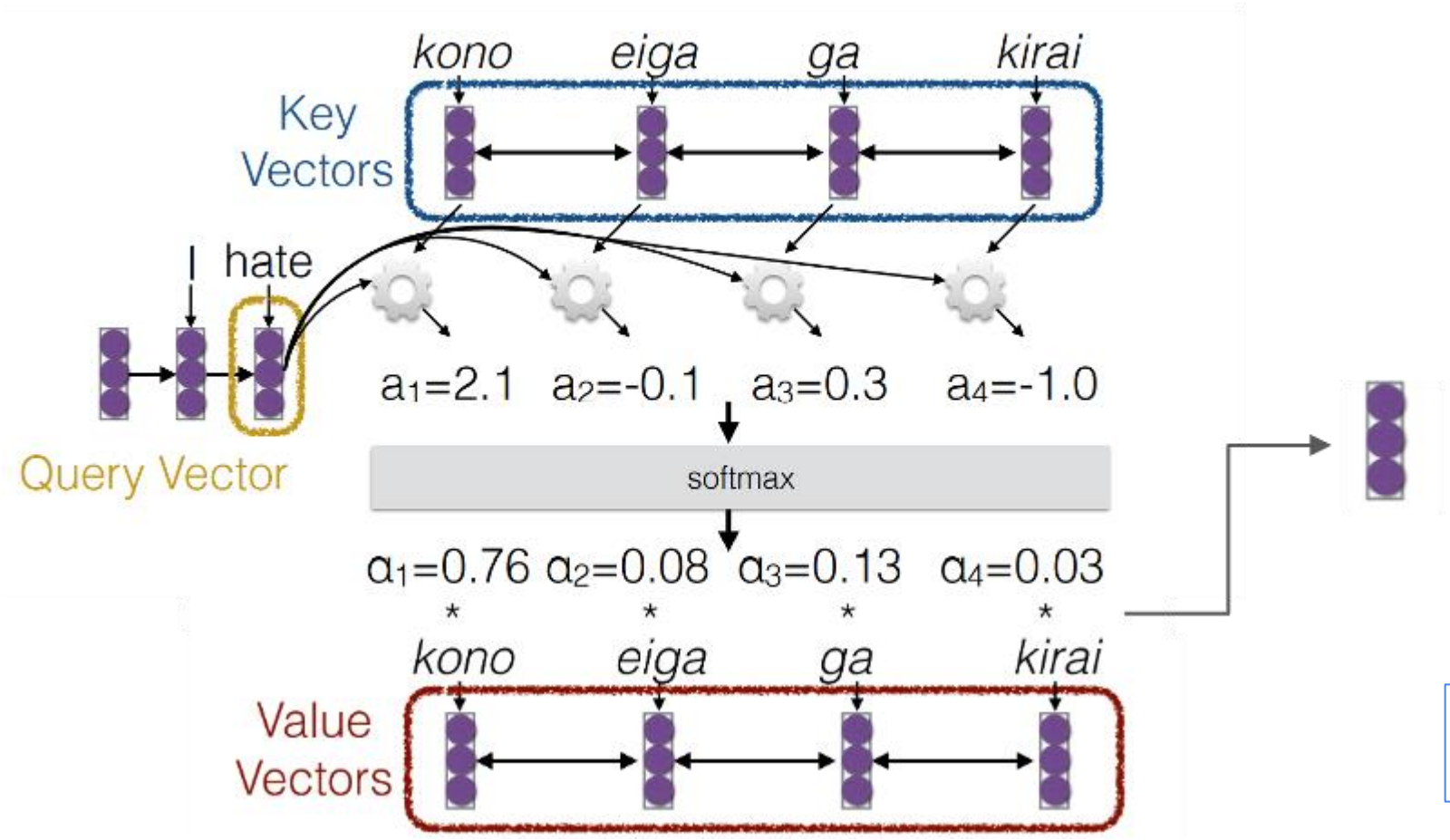
- Encode words as IDs
- Map IDs to embeddings
- Encode input, token-by-token, in hidden state
- Pass hidden state and start token to decoder
- At each step, pass output vector to softmax
- Decode, map embeddings to IDs to tokens—the output

Attention and Self-Attention

Building intuition

Imagine I ask you to translate your favourite book

Attention Illustrated: RNN



The illustration shows an encoder-decoder model in a state where it has encoded the Japanese sentence, “Kono eiga ga kirai” and has decoded the first two words of the English translation, “I hate”.

In the illustration, the word “hate” is the **query**. It is being compared to each of the four Japanese words (**keys**). We will show the math later.

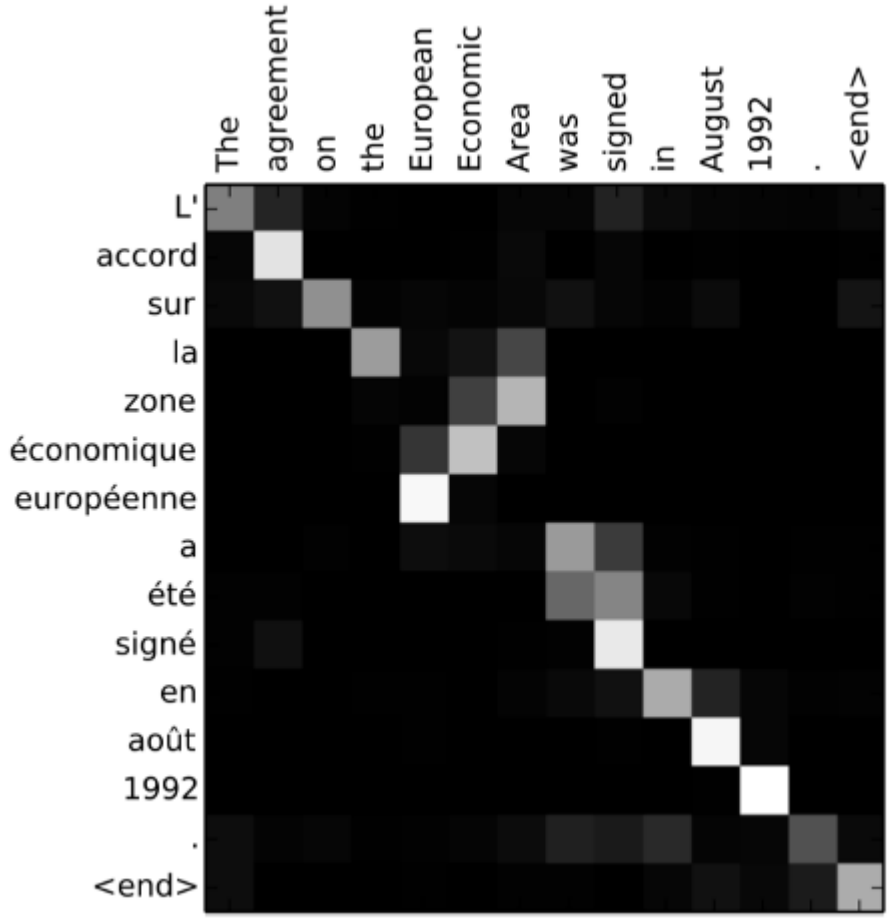
“Hate” has the highest attention to “kono” (this), indicating that the next English word might be “this”.

Explanation of the Japanese sentence:
Kono (this) eiga (movie) ga (NOM) kirai (hate)

The Stronger the Relationship, the Stronger the Attention

Attention is a way of establish many-to-many relationships among words in an encoder-decoder model.

The weight matrices by which key, query, and value matrices are obtained from words are trained so that keys will be most similar to queries when their associated words are most related to each other.



Self-Attention is a Special Kind of Attention in which Words Can Attend to Words in the Same Sequence

Self-attention, as we will define it, is the value matrix **V** weighted by the softmax of the dot product of the query matrix **Q** and the transpose of the key matrix **K** (divided by the square root of the dimensions of the key vector).

$$\textbf{Attention}(Q, K, V) = \textbf{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Intuition: the stronger the relationship of an item at position *j* is to that at position *i*, the larger the dot product of the query vector for *i* and the key vector for *j*. The larger this dot product, the more the value vector at position *j* will contribute to the output.

Computing Self-Attention, Step One: Compute Key, Query, and Value Vectors

d_x-dimensional embeddings

$d_k \times d_x$-dimensional Weight Matrices

d_k-dimensional vectors

Computing Self-Attention, Step One: Compute Key, Query, and Value Vectors

d_x-dimensional embeddings

$d_k \times d_x$-dimensional Weight Matrices

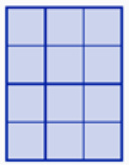
d_k-dimensional vectors

Computing Self-Attention, Step One: Compute Key, Query, and Value Vectors

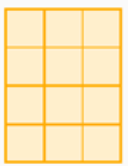
d_x-dimensional embeddings

$d_k \times d_x$-dimensional Weight Matrices

d_k-dimensional vectors

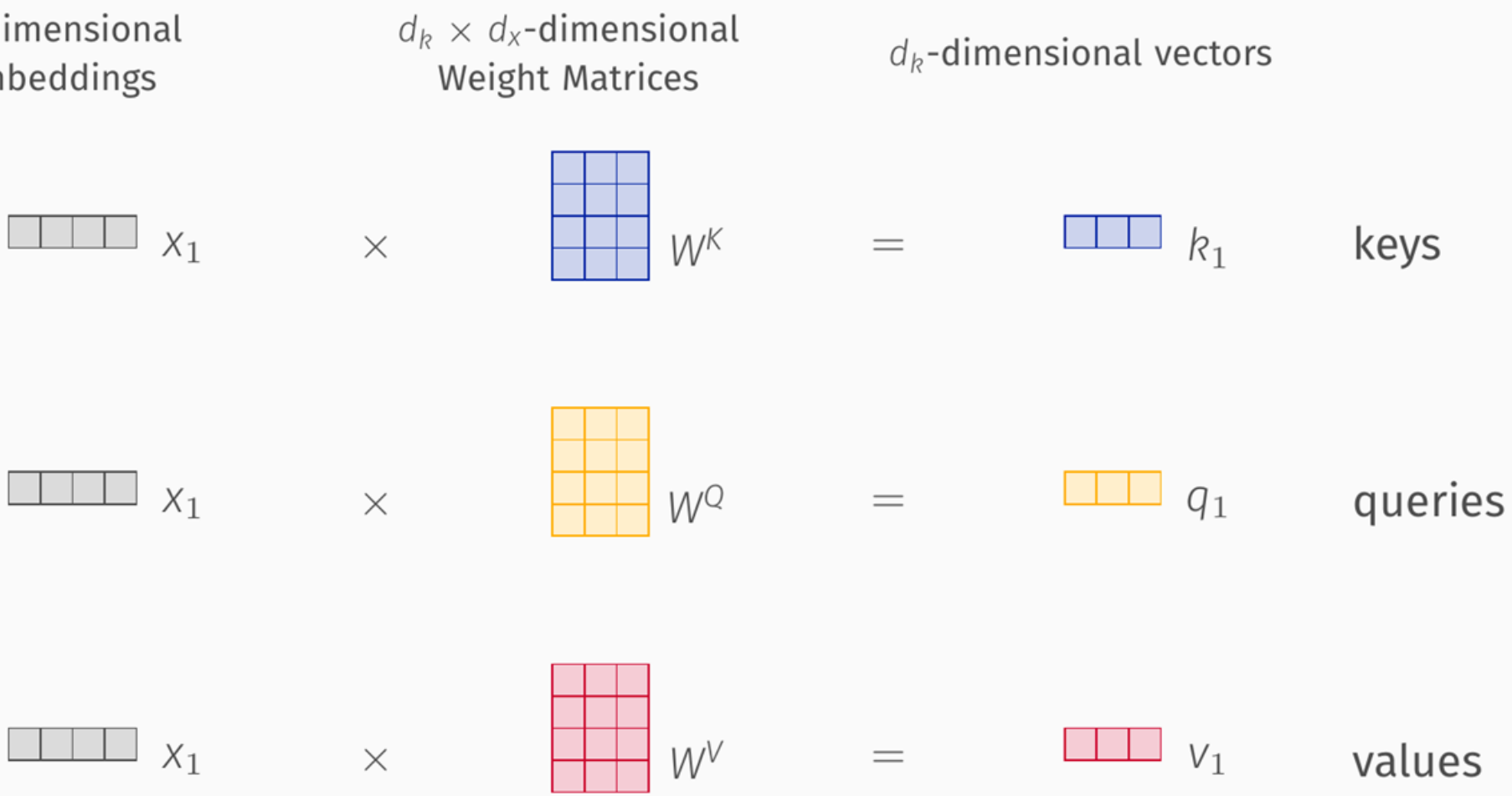

x_1 $\times$

W^K $=$

k_1 keys


x_1 $\times$

W^Q $=$

q_1 queries

Computing Self-Attention, Step One: Compute Key, Query, and Value Vectors



Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

query-key dot product

divide by $\sqrt{d_k}$

softmax

$\times$ value

sum

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We
	x_1
query-key dot product	$q_1 \cdot k_1 = 13$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$
softmax	
$\times$ value	
sum	

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash
	x_1	x_2
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$
softmax		
$\times$ value		
sum		

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our
	x_1	x_2	x_3
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$
softmax			
$\times$ value			
sum			

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our	cats
	x_1	x_2	x_3	x_4
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$	$q_1 \cdot k_4 = 12$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$	$\frac{12}{\sqrt{64}} = 1.5$
softmax				
$\times$ value				
sum				

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our	cats
	x_1	x_2	x_3	x_4
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$	$q_1 \cdot k_4 = 12$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$	$\frac{12}{\sqrt{64}} = 1.5$
softmax	0.12	0.48	0.29	0.10
$\times$ value				
sum				

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our	cats
	x_1	x_2	x_3	x_4
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$	$q_1 \cdot k_4 = 12$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$	$\frac{12}{\sqrt{64}} = 1.5$
softmax	0.12	0.48	0.29	0.10
$\times$ value	$0.12 \times v_1$	$0.48 \times v_2$	$0.29 \times v_3$	$0.10 \times v_4$
sum				

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our	cats
	x_1	x_2	x_3	x_4
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$	$q_1 \cdot k_4 = 12$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$	$\frac{12}{\sqrt{64}} = 1.5$
softmax	0.12	0.48	0.29	0.10
$\times$ value	$0.12 \times v_1$	$0.48 \times v_2$	$0.29 \times v_3$	$0.10 \times v_4$
sum	z_1			

Multihead Self-Attention

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\text{MultiHead}(Q, K, V) = \text{concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

In multihead attention, there are multiple attention "heads," each with their own weight matrices $\mathbf{W}^Q_i$, $\mathbf{W}^K_i$, and $\mathbf{W}^V_i$. MultiHead is the product of the output weight matrix $\mathbf{W}^O$ and the concatenation of the outputs from the *h* heads.

The Linguistic Intuition behind Multi-Head Attention

- There are many heads in an encoder or decoder (as we will discuss when we introduce the Transformer)
- There are also many relationships in language
  - Between verbs and their subjects/objects

    I passed the **test** with flying colors.

  - Between pronouns and their antecedents (the phrases they stand in for)

    Lori delivered the lecture on Thursday that **she** prepared Tuesday morning.

  - And so on…
- Multiple heads allow a model to have separate weight matrices for each kind of relationship.
- $\mathbf{W}^K$ and $\mathbf{W}^Q$ matrices are trained so that positions (e.g., words) that are in a particular relationship will have a strong association (the resulting **k** and **q** vectors will have a large dot product)
- We will talk more about this in a moment.

What are the heads looking at?

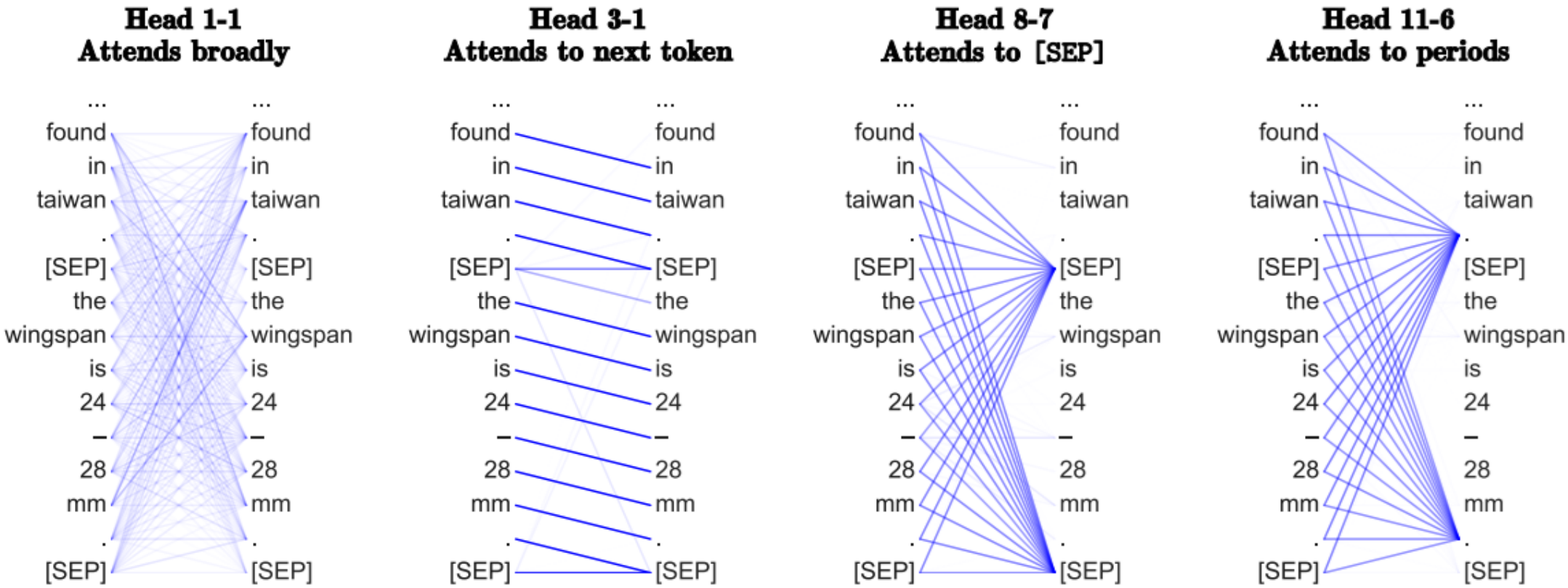


Figure 1: Examples of heads exhibiting the patterns discussed in Section 3. The darkness of a line indicates the strength of the attention weight (some attention weights are so low they are invisible).

Transformers

The Transformer Architecture: Encoders

Transformers consist of

- Stack of N encoders
- Stack of N decoders

Each **encoder** has

- A multi-head attention layer
- A feed-forward neural net

After each of these, there is an additive normalization step (which keeps the changes from from the preceding layers from being too large).

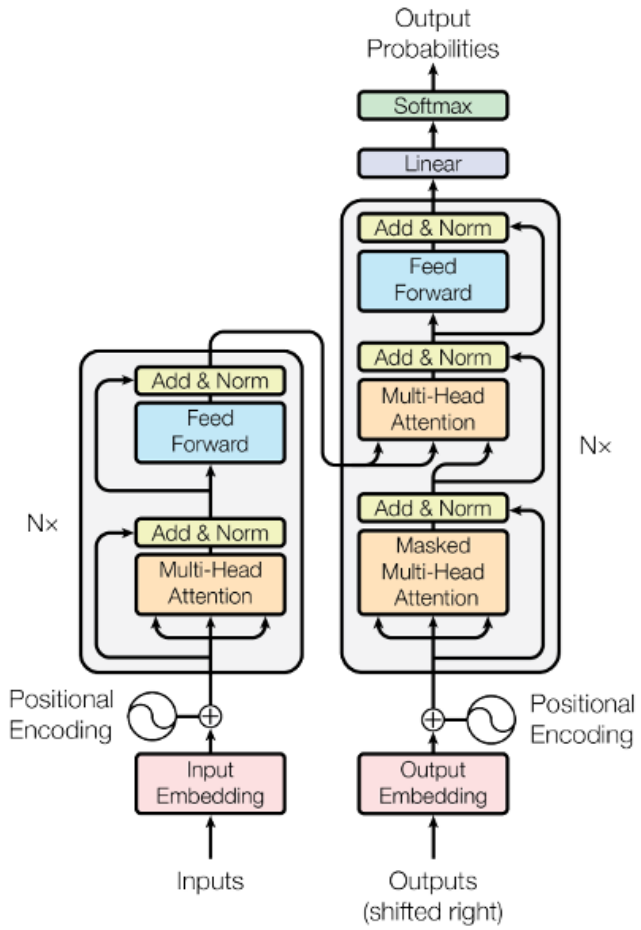


Figure 1: The Transformer - model architecture.

The Transformer Architecture: Decoders

The **decoders** consist of a masked multi-head attention layer, a multi-head attention layer, and a feed-forward neural net (with additive normalization).

The outputs pass through a linear layer and softmax.

Outputs from the encoder are inputs to the multi-headed attention layer.

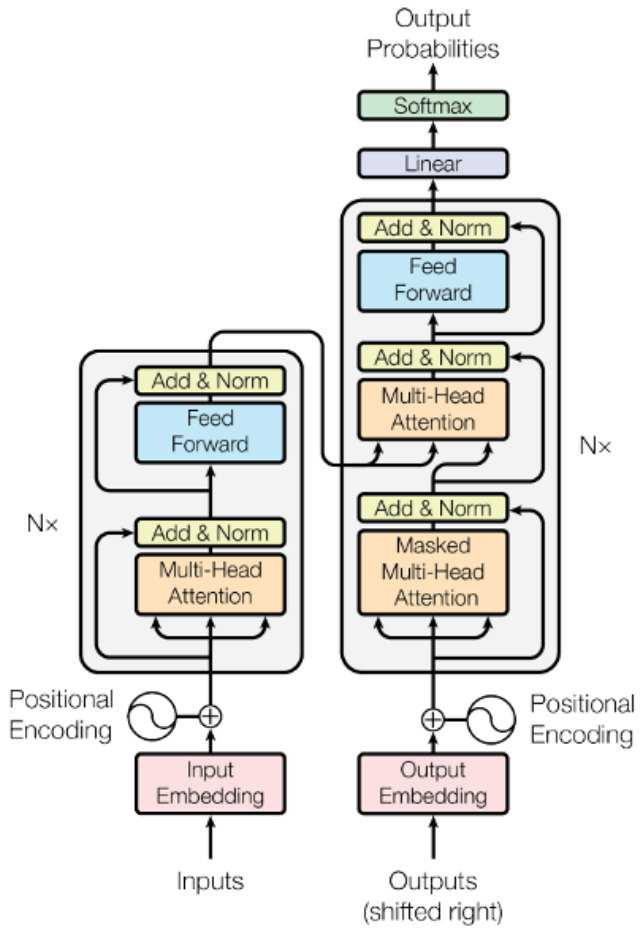


Figure 1: The Transformer - model architecture.

Advantages and Disadvantages of Transformers

Transformers have several advantages over Simple RNNs and LSTMs

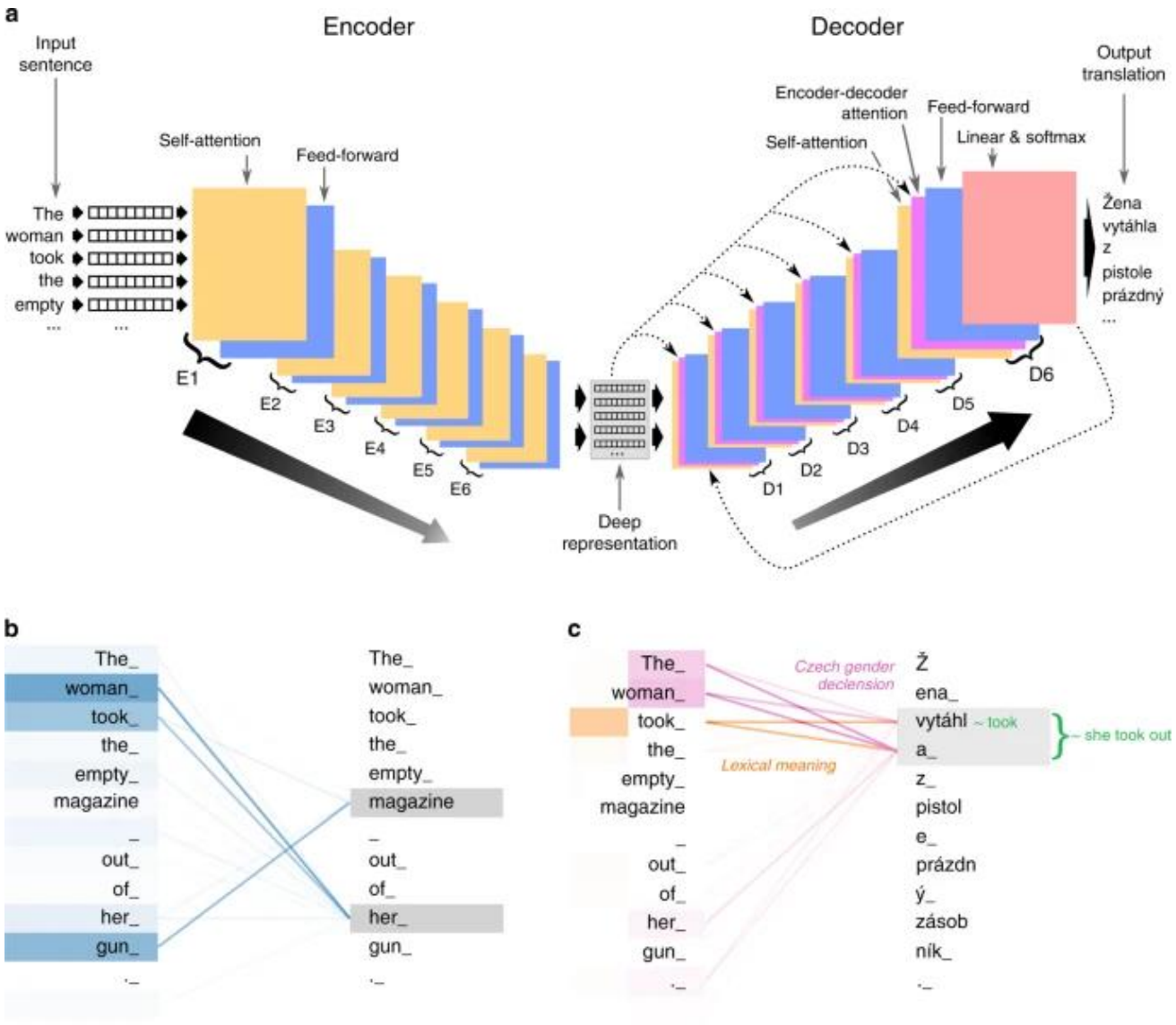
- **No issue with directionality**—all positions are connected to all other positions via attention
- **No vanishing gradients**—long distance relationships are, in principle, indistinguishable from other relationships
- **Parallelizable**—attention can be computed in parallel (no recurrence)

They have one particular disadvantage:

- **Lots of parameters**—lots of encoders/decoders with lots of heads, each with $\mathbf{W}^Q$, $\mathbf{W}^K$, and $\mathbf{W}^V$, each with lots of dimensions (not to speak of the FFNNs). **They take lots of data to train!**

Transformers Applied to MT

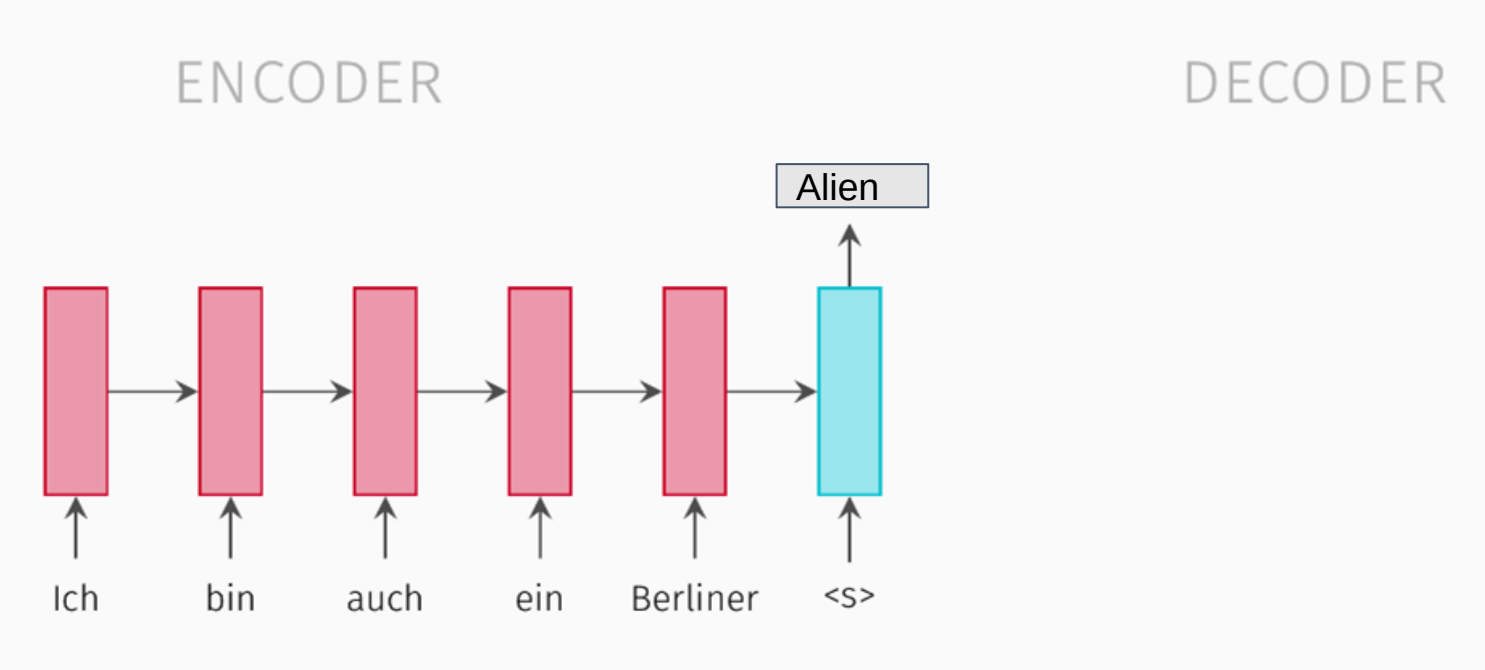
One Approach to MT using Transformers



- Inputs (tokens from source sentence) are embedded (not shown: positional encoding) and fed to the first encoder
- Each encoder feeds its output to the subsequent encoder
- From last encoder: “deep representation” (tensor)
- Each decoder receives input from:
  - preceding decoder (after D1)
  - Deep representation
- Last encoder feeds output to linear & softmax layer, resulting in a probability distribution over output translations
- Feedback from last layer to first decoder

Teacher Forcing

Why Teacher Forcing



What if the output of the decoder is wrong?

- Error propagates in consecutive timesteps of the decoder
- Increases the time taken for model training to reach acceptable performance

How to Tackle this? Teacher Forcing!

Based of a weighted coin toss, we feed the actual previous target outputs (or expected correct inputs) as inputs to the decoder at every timestep, rather than previous output of the decoder.

As the model learns, it will generate the right target outputs as expected

When do you use teacher enforcing?

To be implemented in HW6

Questions?