ATTENTION AND TRANSFORMERS

Sergey Nikolenko

Harbour Space University

June 17, 2026

Random facts:

- on June 17, 1631, Mumtaz Mahal died giving birth to her fourteenth child, and her grief-stricken husband, the Mughal emperor Shah Jahan, spent the next two decades building her tomb, the Taj Mahal
- on June 17, 1885, the Statue of Liberty arrived in New York Harbor from France, dismantled into 350 pieces packed in 214 crates; it then sat boxed up for months while Joseph Pulitzer's newspaper shamed the public into paying for the pedestal it was to stand on
- on June 17, 1972, five White House operatives were arrested for burgling the offices of the Democratic National Committee during an attempt by members of the administration of Richard Nixon to illegally wiretap the political opposition (the Watergate scandal)
- on June 17, 1939, France carried out its last public guillotining, of the murderer Eugen Weidmann at Versailles; the crowd was so unruly and the smuggled film footage so scandalous that all further executions were moved behind prison walls
- on June 17, 1972, five men were caught breaking into the Democratic National Committee headquarters at the Watergate complex in Washington; what was first dismissed as a ``third-rate burglary'' would two years later force a sitting U.S. president to resign

LSTM and GRU

LSTM

- Vanishing gradients: we have to multiply by W every time.
- This makes it impossible to have long-term memory.

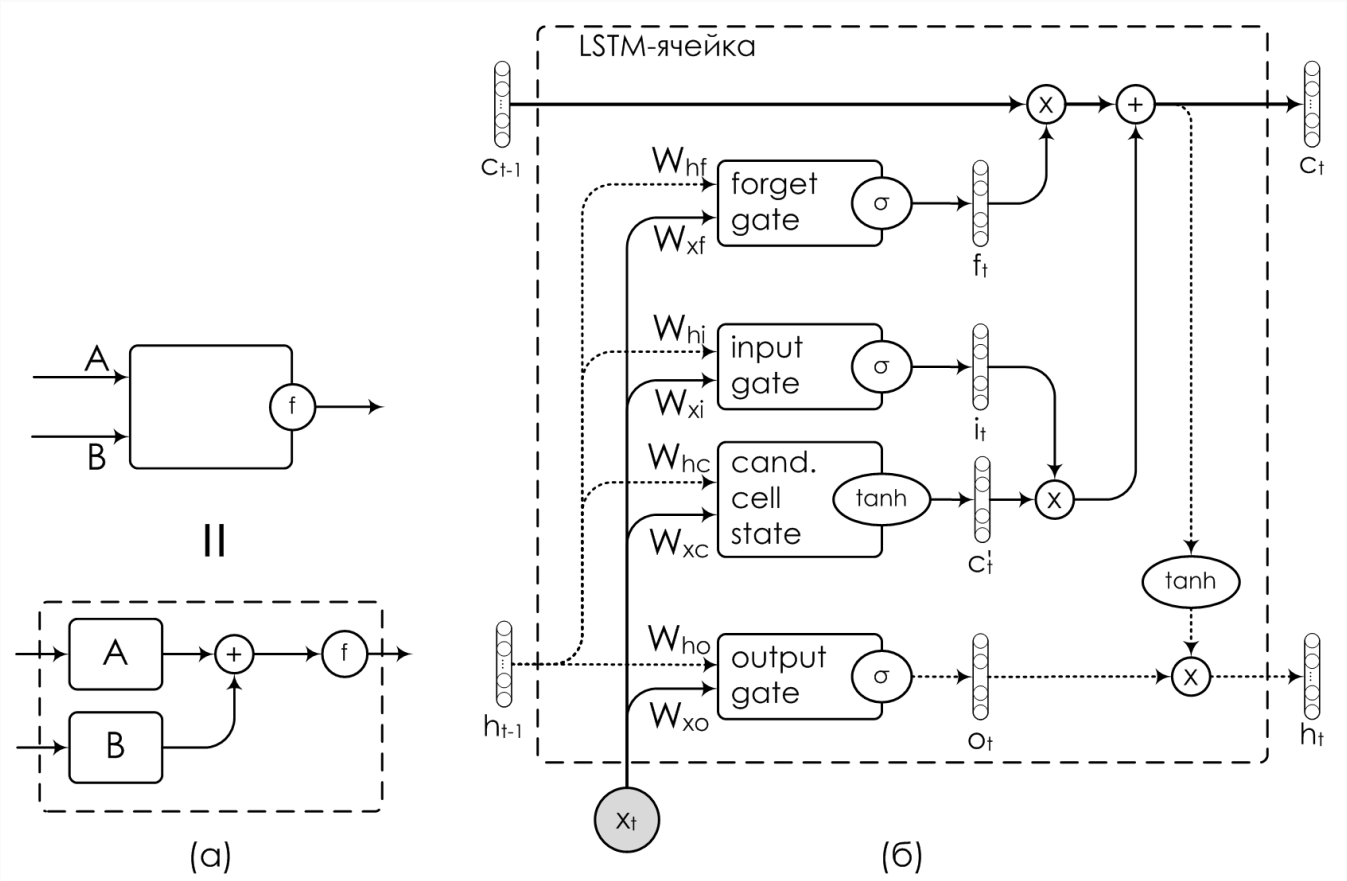
- What do we do?..

LSTM

- ...we have seen the basic idea in ResNet: we have to provide the "constant error carousel", a shortcut for the gradients.
- Idea from the mid-1990s (Schmidhuber): let's construct RNNs from a more complex unit that has the shortcut and controls memory explicitly.
- LSTM (long short-term memory).

LSTM

- "Vanilla" LSTM: $\mathbf{c}_t$ is the cell state, and $\mathbf{h}_t$ is the hidden state.
- Input gate and forget gate control whether we change $\mathbf{c}_t$ to the new candidate cell state.



LSTM

- Formally:

$$
\begin{aligned}
\mathbf{c}'_t &= \tanh\left(W_{xc}\mathbf{x}_t + W_{hc}\mathbf{h}_{t-1} + \mathbf{b}_{c'}\right) && \textit{candidate cell state} \\
\mathbf{i}_t &= \sigma\left(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i\right) && \textit{input gate} \\
\mathbf{f}_t &= \sigma\left(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f\right) && \textit{forget gate} \\
\mathbf{o}_t &= \sigma\left(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o\right) && \textit{output gate} \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{c}'_t, && \textit{cell state} \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) && \textit{block output}
\end{aligned}
$$

- So the LSTM cell is able to control the cell state with the hidden state and weights.
- Very flexible, and if the forget gate is closed ($\mathbf{f}_t = 1$), we have

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{c}'_t, \text{ so } \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = 1.$$

- The constant error carousel!

LSTM

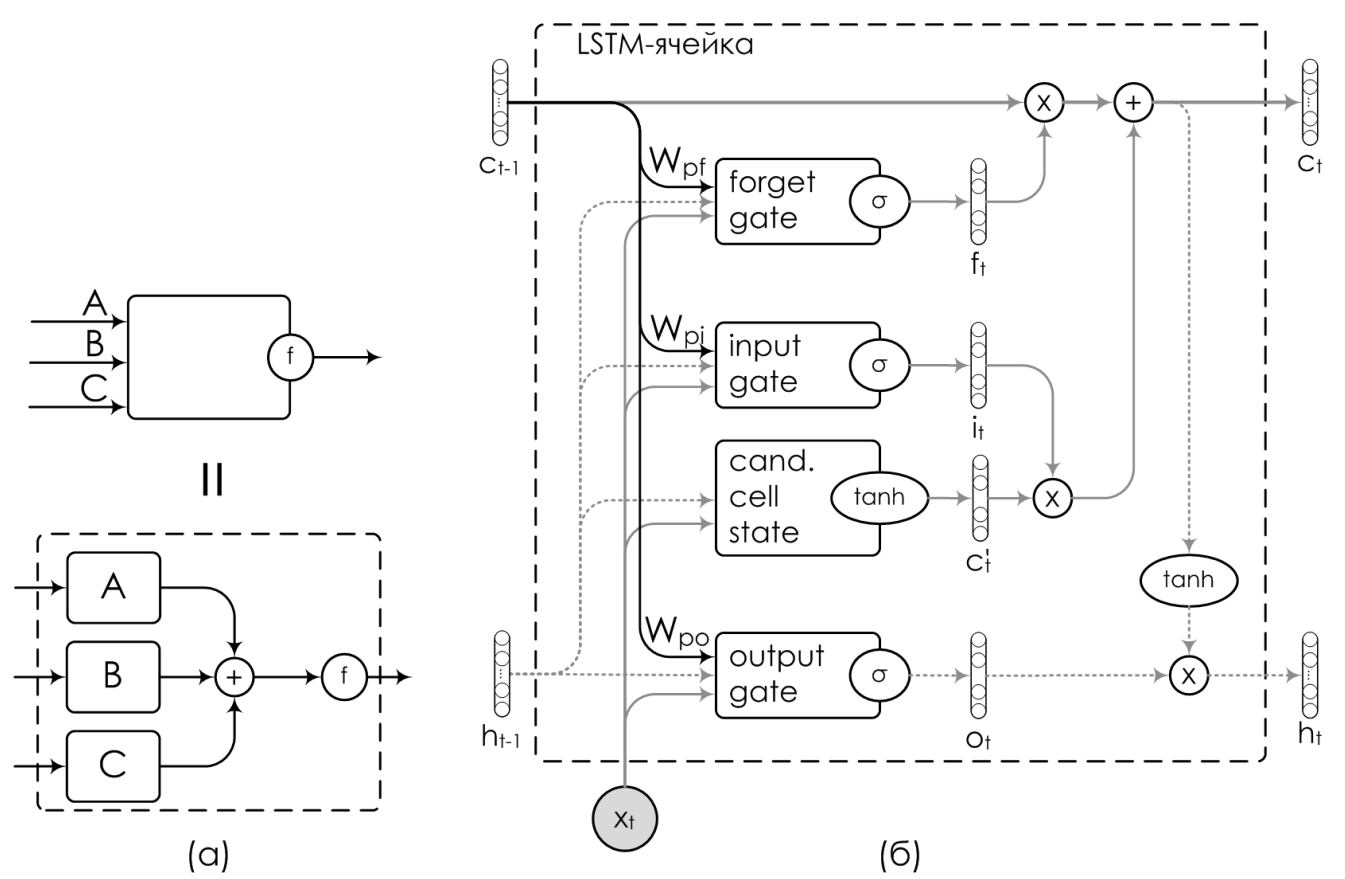
- LSTM was developed in mid-1990s (Hochreiter and Schmidhuber, 1995; 1997).
- Completely in its modern form in (Gers, Schmidhuber, 2000).
- One problem: we want to control $\mathbf{c}$, but it's unavailable for the gates! They only see $\mathbf{h}_{t-1}$, which is

$$\mathbf{h}_{t-1} = \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}).$$

- So if the output gate is closed the behaviour of LSTM does not depend on cell state at all.
- This is not good...

LSTM

- ...so of course we add a few more weight matrices (peepholes).



LSTM

- Formally:

$$\mathbf{i}_t = \sigma\left(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + W_{pi}\mathbf{c}_{t-1} + \mathbf{b}_i\right)$$
$$\mathbf{f}_t = \sigma\left(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + W_{pf}\mathbf{c}_{t-1} + \mathbf{b}_f\right)$$
$$\mathbf{o}_t = \sigma\left(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + W_{po}\mathbf{c}_{t-1} + \mathbf{b}_o\right)$$

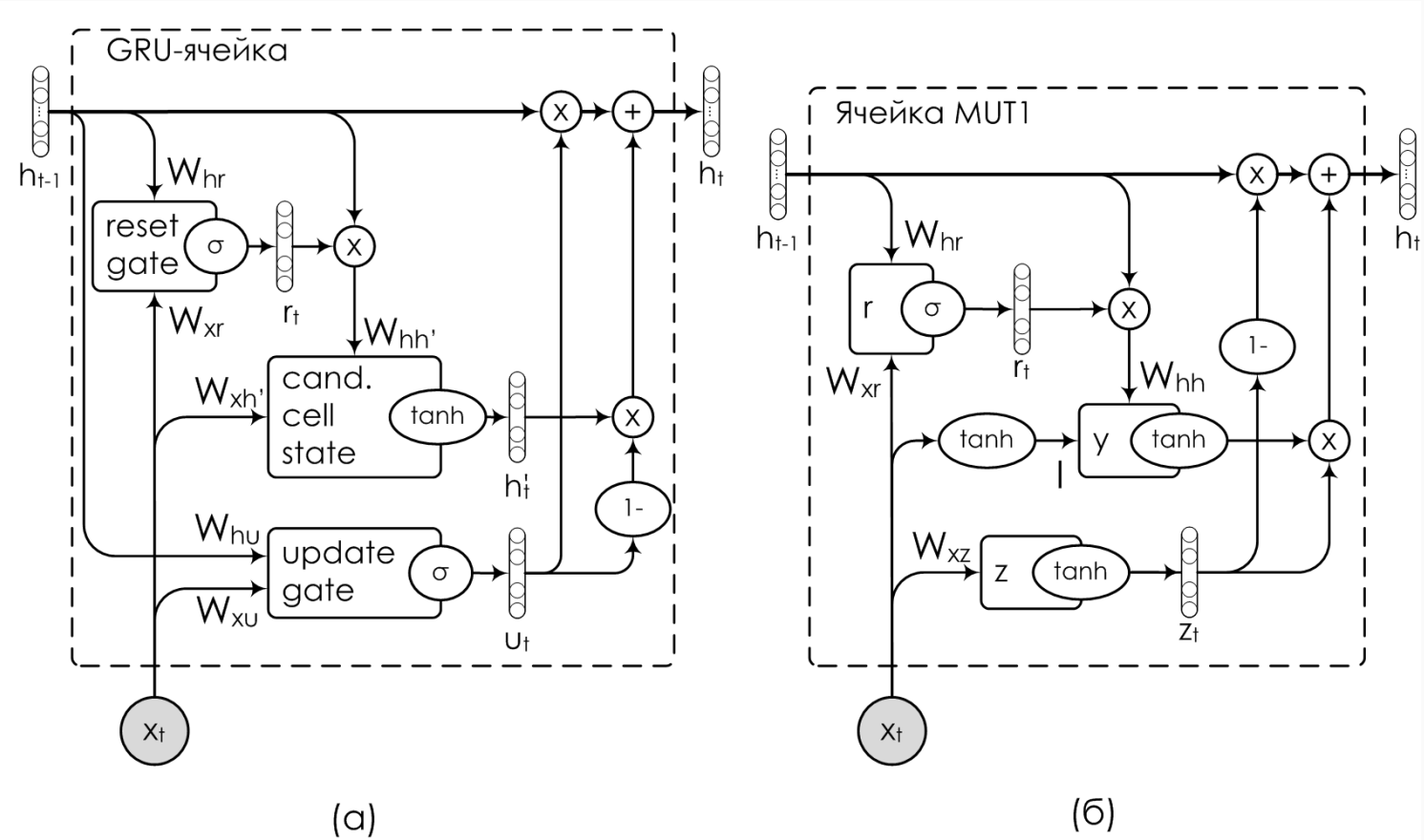
- Huge number of LSTM variations: we can basically remove any kind of gate, add or remove each peephole, add or remove activation functions etc.
- How do we choose?

LSTM

- «LSTM: a Search Space Odyssey» (Greff et al., 2015).
- Shows an experimental comparison of many variations.
- In particular, some significantly simpler architectures (with one less gate!) did not lose to the vanilla LSTM much.
- This brings us to...

GRU

- ...Gated Recurrent Units (GRU); developed in (Cho et al., 2014).
- Also implements the constant error carousel, but simpler than LSTM.



GRU

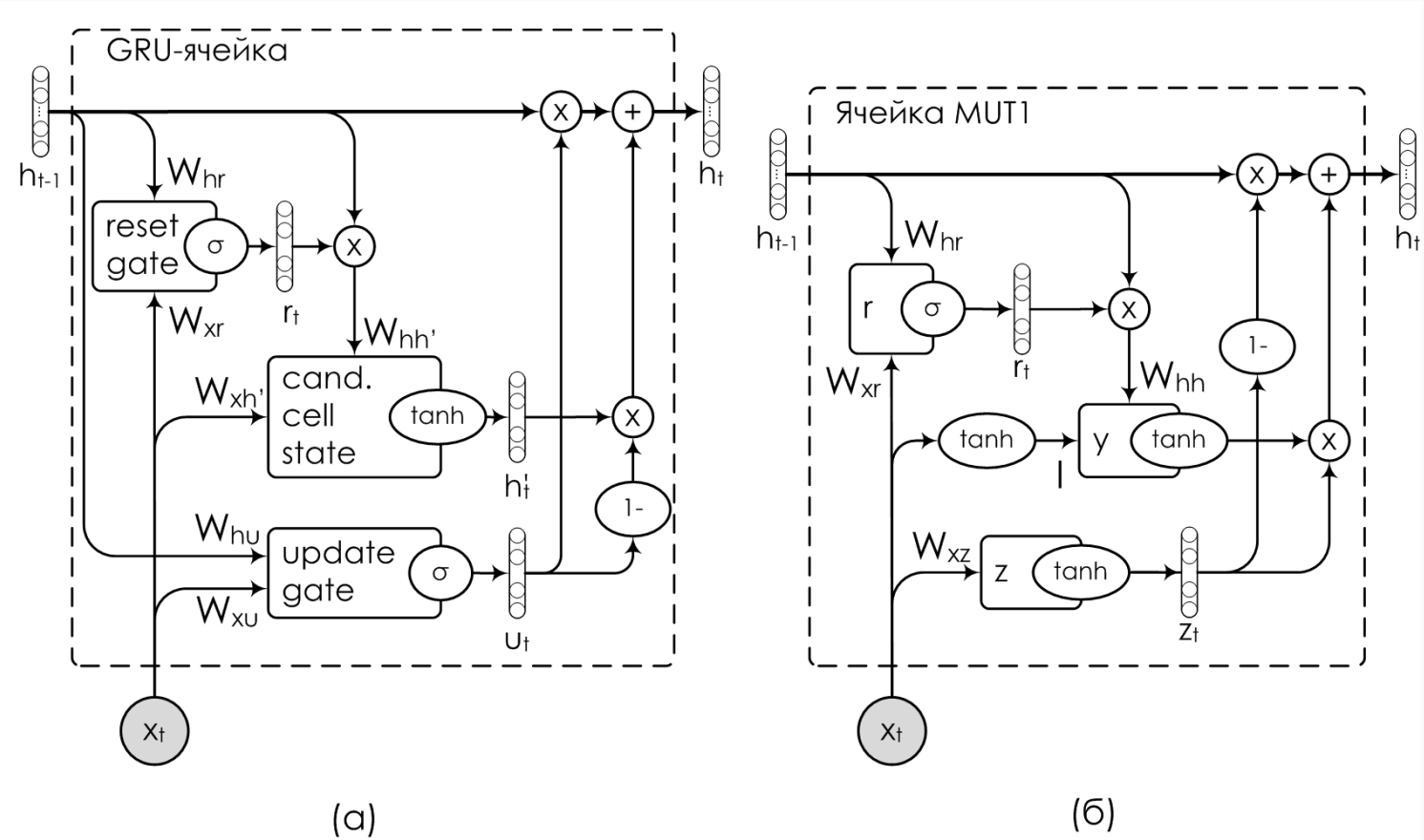
- Formally:

$$\begin{aligned}
\mathbf{u}_t &= \sigma(W_{xu}\mathbf{x}_t + W_{hu}\mathbf{h}_{t-1} + \mathbf{b}_u) \\
\mathbf{r}_t &= \sigma(W_{xr}\mathbf{x}_t + W_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r) \\
\mathbf{h}'_t &= \tanh(W_{xh'}\mathbf{x}_t + W_{hh'}(\mathbf{r}_t \odot \mathbf{h}_{t-1})) \\
\mathbf{h}_t &= (1 - \mathbf{u}_t) \odot \mathbf{h}'_t + \mathbf{u}_t \odot \mathbf{h}_{t-1}
\end{aligned}$$

- Update gate and reset gate; there is no distinction between $\mathbf{c}_t$ and $\mathbf{h}_t$.
- Fewer matrices (6 vs. 8 or 11 with peepholes), fewer weights, but only very slightly worse than LSTM.
- So you can fit more GRUs and have better networks on a given computational budget.

GRU

- There are other variations too.
- (Józefowicz, Zaremba, Sutskever, 2015): huge experimental comparison, with an evolutionary approach.
- Identified three interesting architectures.



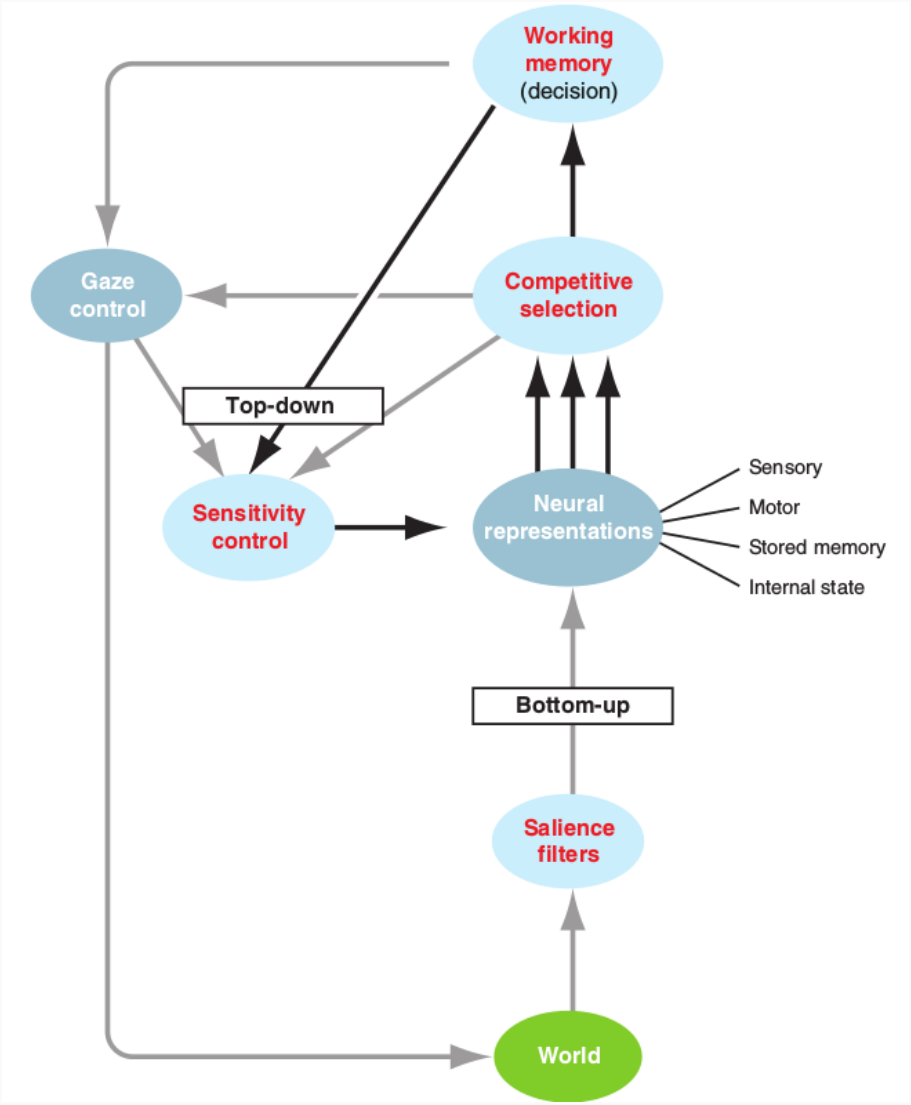
What is attention?

Attention

- You are paying *attention* to me now, right?
- What does it mean?..
- Images from the retina go through something like a CNN, but then we pay attention to a part of it and don't care about the rest. What does it mean?
- Turns out it's a pretty difficult question.
- A.R. Luria: attention, memory, and cognitive functions.

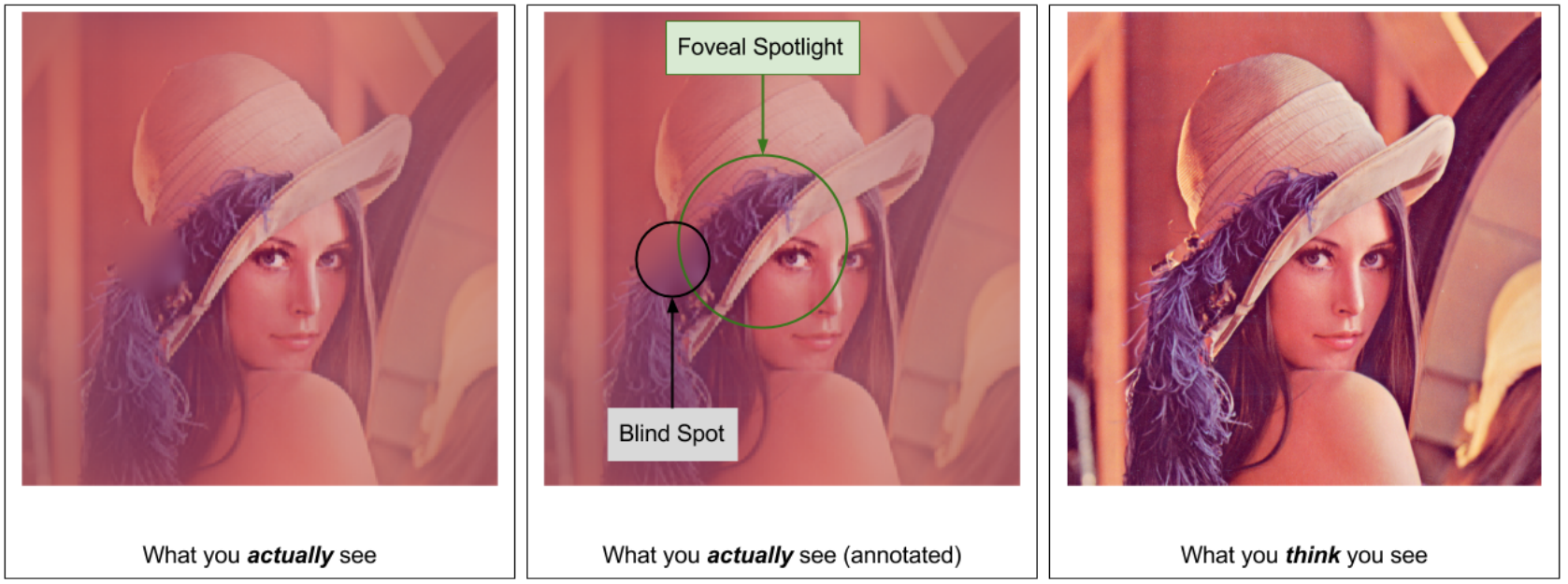
Attention

- In reality it's mostly about the interaction with working memory (Knudsen, 2007):



Attention

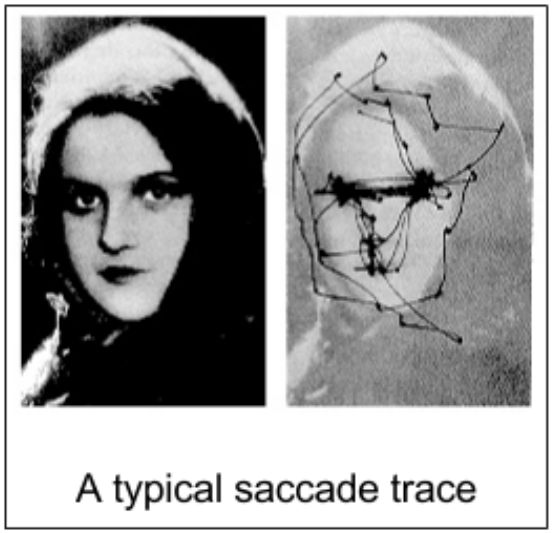
- How do we implement this in ANNs? Especially “conscious” attention.
- But “unconscious” attention as well; e.g., with images we don’t see much at any given moment of time.
- The fovea is small:



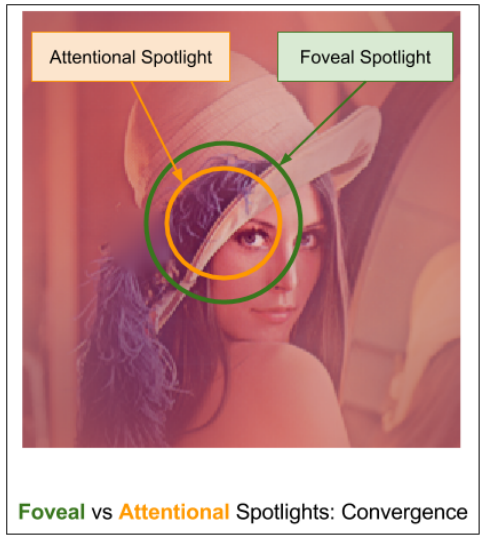
- Simons, Chabris, 1999: `https://www.youtube.com/watch?v=vJG698U2Mvo`

Attention

- Our eyes work in *saccades*, and this doesn't always help either:



A typical saccade trace

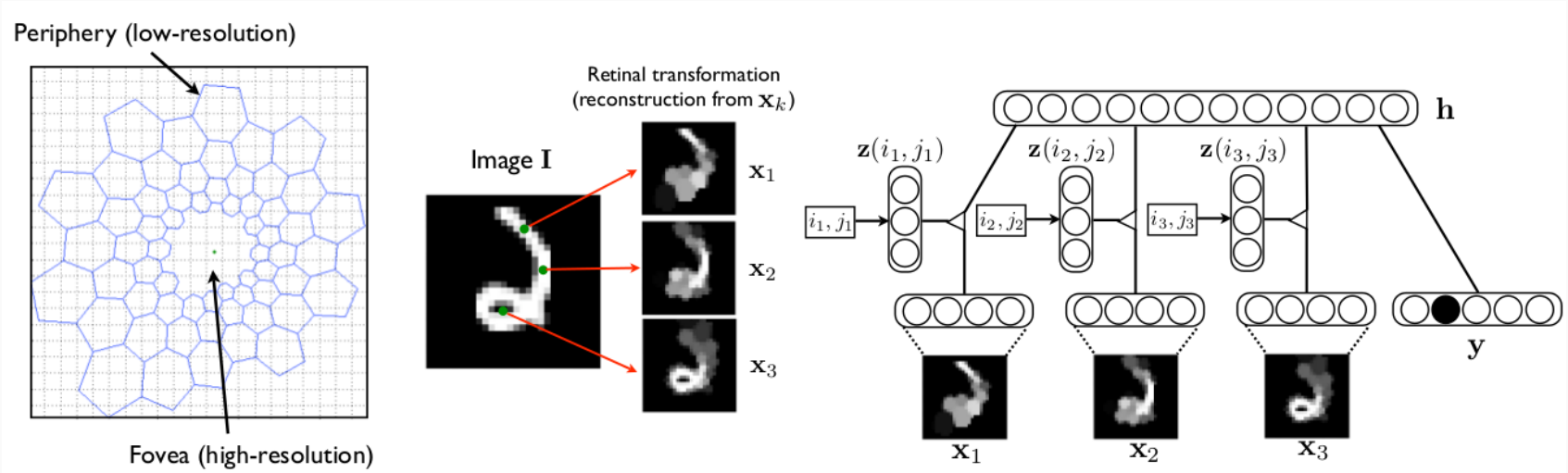


Foveal vs Attentional Spotlights: Convergence

Foveal vs Attentional Spotlights: Divergence

Foveal glimpses

- A neural network also might want to consciously understand “what to look at”.
- One of the first works (Larochelle, Hinton, 2010):

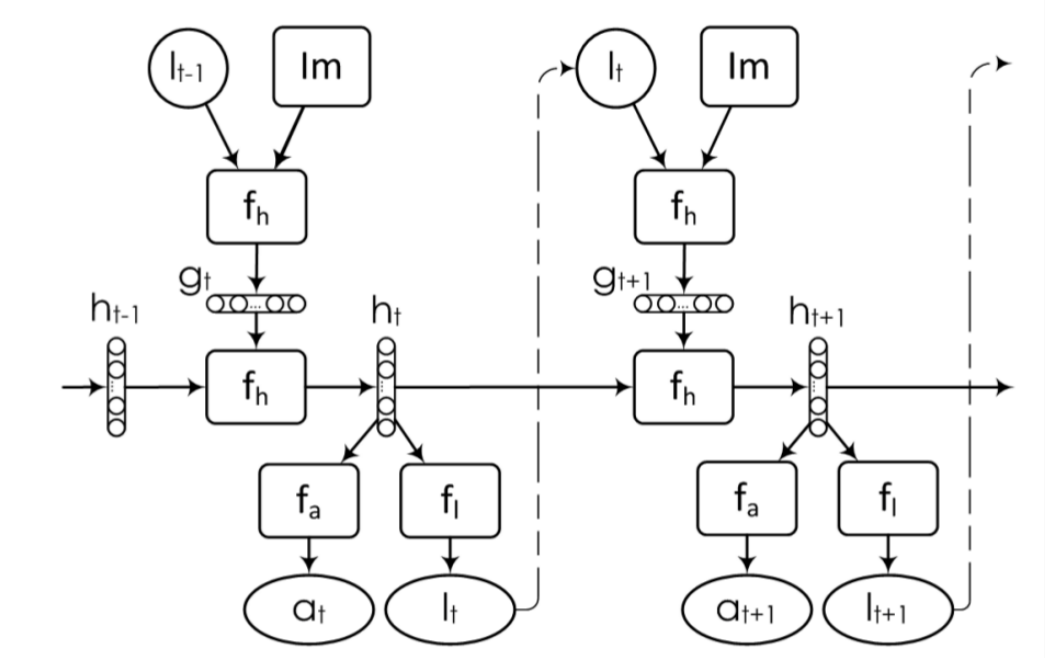


- They model the positions of saccades and learn to do a sequence of glimpses.
- A sequence means...

Recurrent visual attention

Recurrent visual attention

- It all really happened in (Mnih et al., 2014), “Recurrent Models of Visual Attention”:
  - from the previous vector $\mathbf{h}_{t-1}$ and new “glimpse” location $_t$ the funciton f_g makes $\mathbf{g}_t$, the input for step t;
  - from $\mathbf{h}_{t-1}$ and $\mathbf{g}_t$ function f_h obtains $\mathbf{h}_t$;
  - from it, the “action” $a_t = f_a(\mathbf{h}_t)$ and the next “glimpse” location $_{t+1} = f_l(\mathbf{h}_t)$.



Recurrent visual attention

- This is where a lot of different ML ideas come together; formal model:

$$
\begin{aligned}
\mathbf{g}_t &= f_g(\mathbf{x}_t, \mathbf{l}_{t-1}; \theta_g), \\
\mathbf{h}_t &= f_h(\mathbf{h}_{t-1}, \mathbf{g}_t; \theta_h), \\
\mathbf{l}_t &\sim p(\cdot \mid f_l(\mathbf{h}_t; \theta_l)), \\
a_t &\sim p(\cdot \mid f_a(\mathbf{h}_t; \theta_a)).
\end{aligned}
$$

- After an action we get a new observation $\mathbf{x}_{t+1}$ and reward r_t that we'll get at the end, for correct classification.
- What does it remind of?..

Recurrent visual attention

- ...that's right, reinforcement learning!
- We need to train a stochastic policy $\pi((\mathbf{l}_t, a_t) \mid \mathbf{s}_{1:t}; \theta)$ that would output the next action by history.
- Here π is defined via an RNN, and we need to optimize

$$J(\theta) = \mathbb{E}_{p(\mathbf{s}_{1:T};\theta)}[R] = \mathbb{E}_{p(\mathbf{s}_{1:T};\theta)}\left[\sum_{t=1}^{T} r_t\right].$$

- We won't go into details, but RL is all about optimizing these seemingly very complicated functions; here we can use the REINFORCE algorithm and other policy gradient approaches.

Recurrent visual attention

- Results:

(a) Translated MNIST inputs.

(b) Cluttered Translated MNIST inputs.

(a) 28x28 MNIST

Model	Error
FC, 2 layers (256 hiddens each)	1.69%
Convolutional, 2 layers	1.21%
RAM, 2 glimpses, 8×8, 1 scale	3.79%
RAM, 3 glimpses, 8×8, 1 scale	1.51%
RAM, 4 glimpses, 8×8, 1 scale	1.54%
RAM, 5 glimpses, 8×8, 1 scale	1.34%
RAM, 6 glimpses, 8×8, 1 scale	1.12%
RAM, 7 glimpses, 8×8, 1 scale	**1.07%**

(b) 60x60 Translated MNIST

Model	Error
FC, 2 layers (64 hiddens each)	6.42%
FC, 2 layers (256 hiddens each)	2.63%
Convolutional, 2 layers	1.62%
RAM, 4 glimpses, 12×12, 3 scales	1.54%
RAM, 6 glimpses, 12×12, 3 scales	**1.22%**
RAM, 8 glimpses, 12×12, 3 scales	**1.2%**

Recurrent visual attention

- Results:

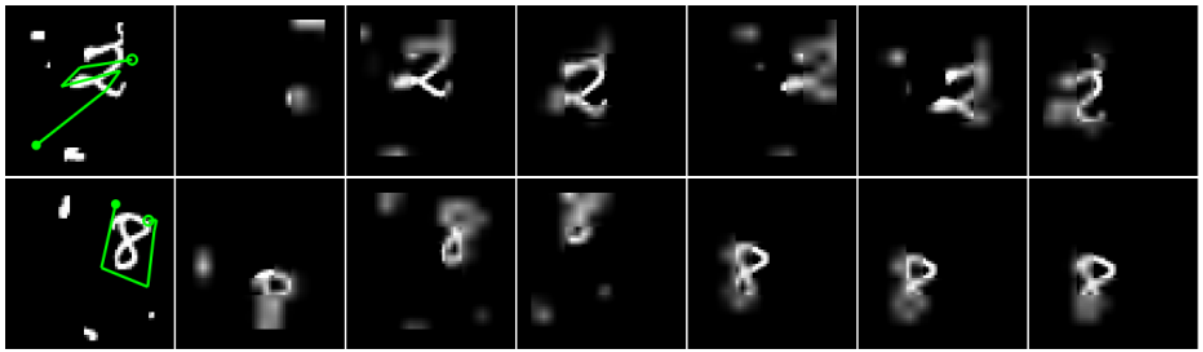
(a) 60x60 Cluttered Translated MNIST

Model	Error
FC, 2 layers (64 hiddens each)	28.58%
FC, 2 layers (256 hiddens each)	11.96%
Convolutional, 2 layers	8.09%
RAM, 4 glimpses, 12 × 12, 3 scales	4.96%
RAM, 6 glimpses, 12 × 12, 3 scales	4.08%
RAM, 8 glimpses, 12 × 12, 3 scales	4.04%
RAM, 8 random glimpses	14.4%

(b) 100x100 Cluttered Translated MNIST

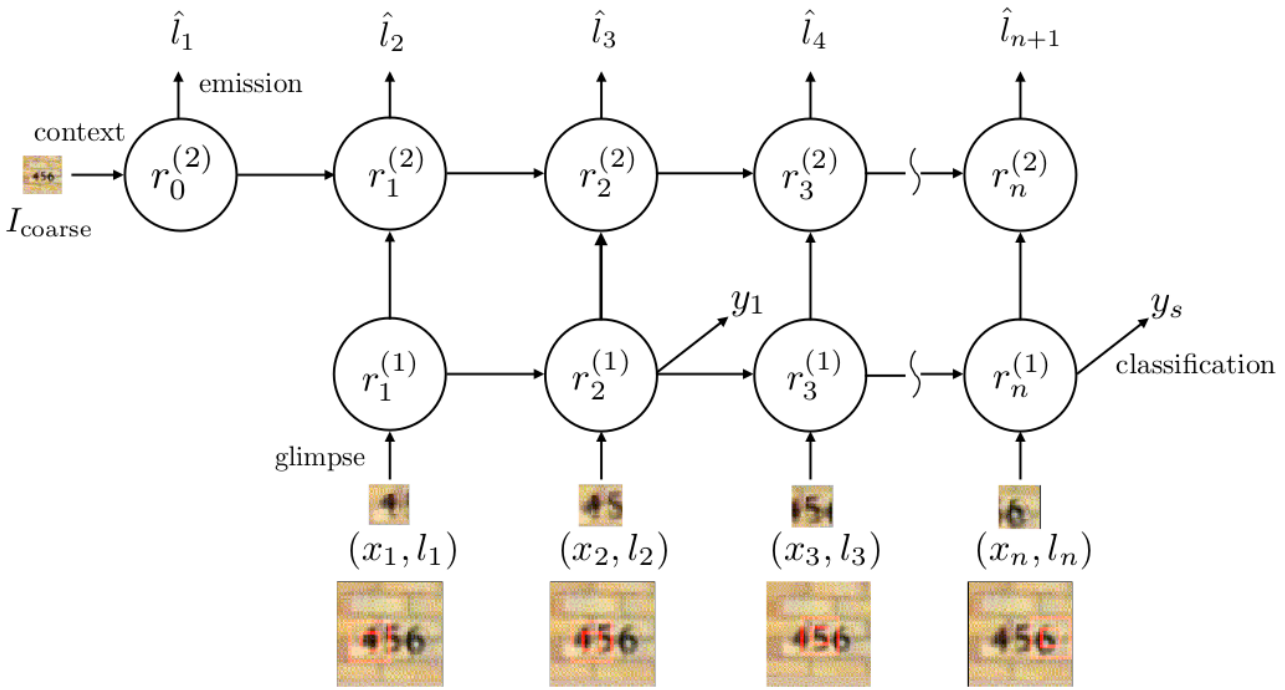
Model	Error
Convolutional, 2 layers	14.35%
RAM, 4 glimpses, 12 × 12, 4 scales	9.41%
RAM, 6 glimpses, 12 × 12, 4 scales	8.31%
RAM, 8 glimpses, 12 × 12, 4 scales	8.11%
RAM, 8 random glimpses	28.4%

- Here is attention walking around the picture:



Recurrent visual attention

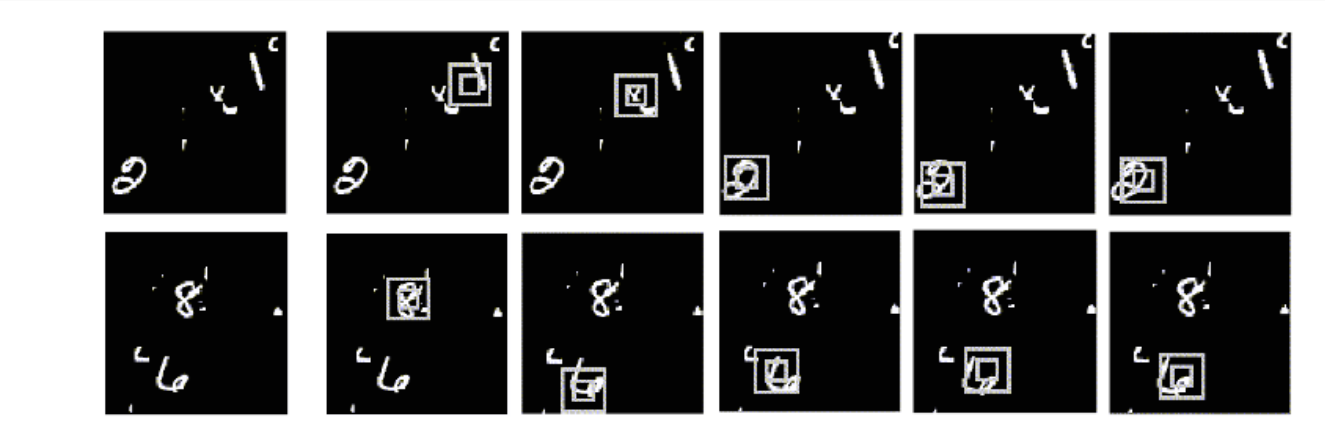
- In the next work Ba et al. (2015) made a deep model:



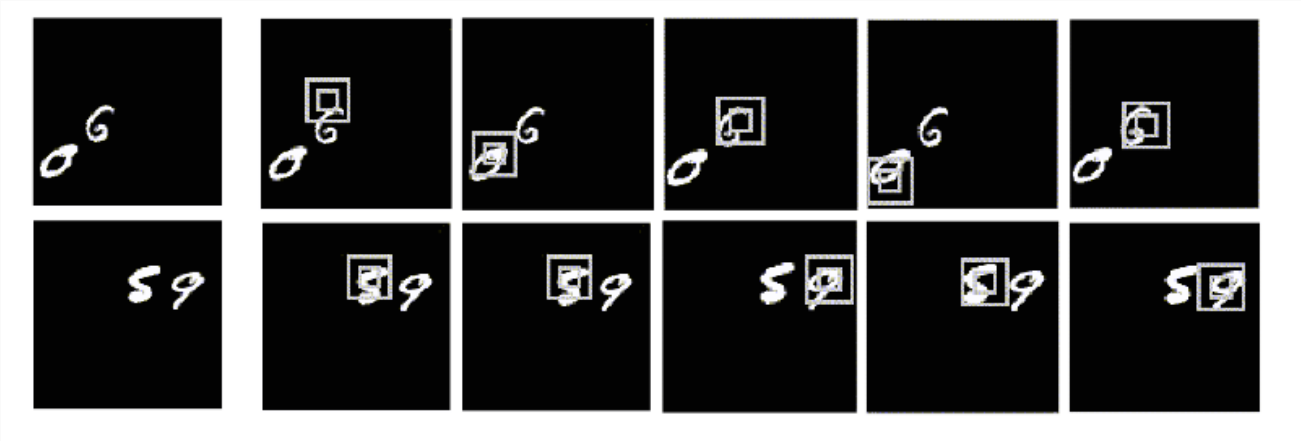
- And trained it differently, with variational approximations (definitely won't go there now, but the point is it's all used in DL too).

Recurrent visual attention

- Recognizing two numbers:

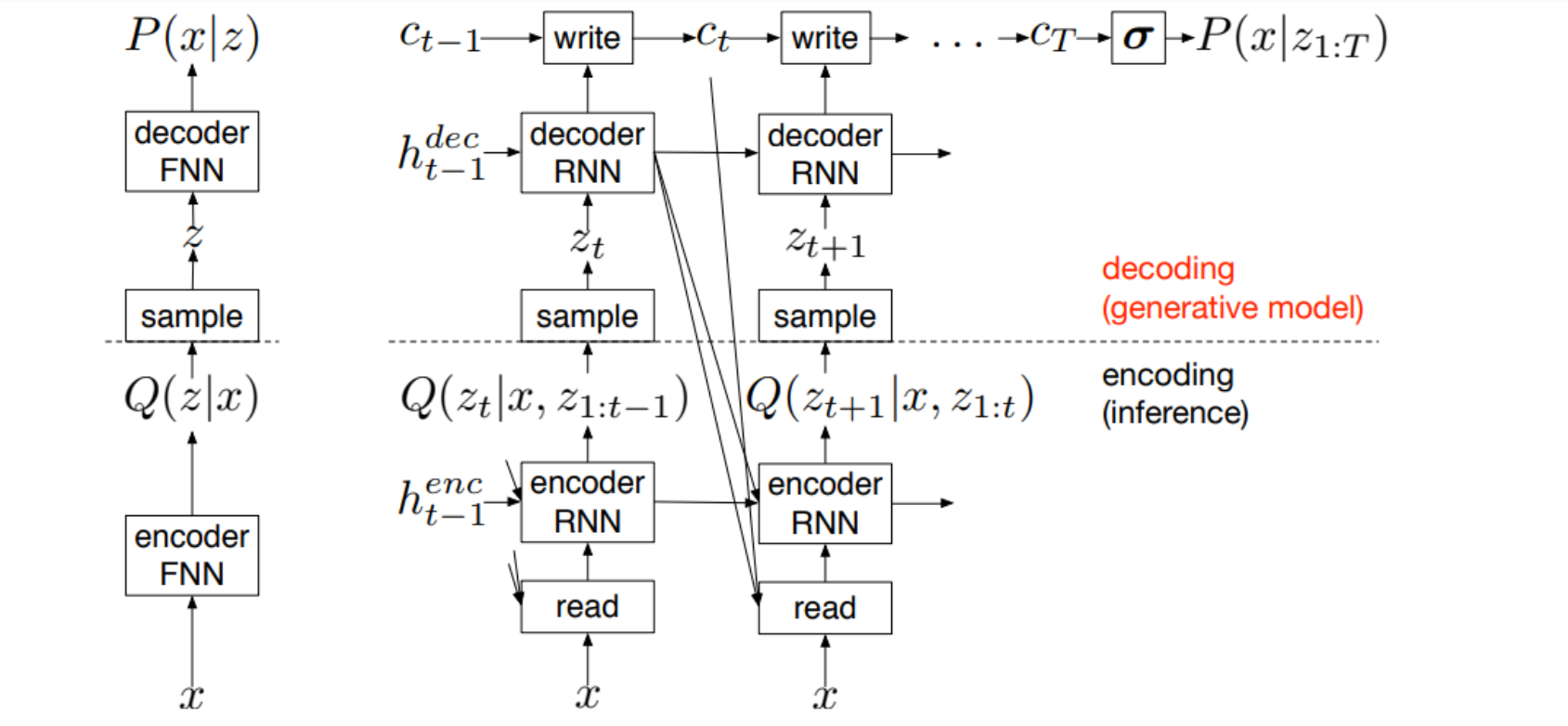


- Interestingly, addition works differently:



Recurrent visual attention

- (Gregor et al., 2015): DRAW: A Recurrent Neural Network For Image Generation.
- Next step in visual attention development, a variation of the variational autoencoder:



Encoder-decoder and attention

Machine translation

- Machine translation is an excellent test problem
- In essence, it's a sequence-to-sequence problem, so it's natural to use RNNs:
  - RNN models the sequence $X = (x_1, x_2, \dots, x_T)$ as

$$p(x_1), p(x_2 \mid x_1), \dots, p(x_T \mid x_{<T}) = p(x_T \mid x_{T-1}, \dots, x_1),$$

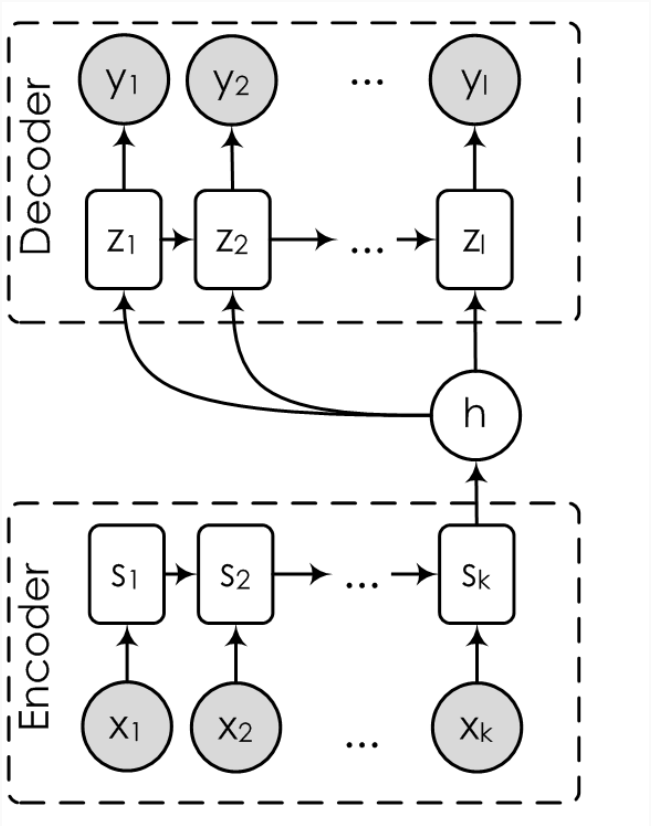
    and now $p(X)$ is just

$$p(X) = p(x_1)p(x_2 \mid x_1) \dots p(x_k \mid x_{<k}) \dots p(x_T \mid x_{<T});$$

  - that's how RNNs are used in language models: predict the next word from the previous word and hidden state.
- How do we apply this idea to translation?

Encoder-decoder architectures

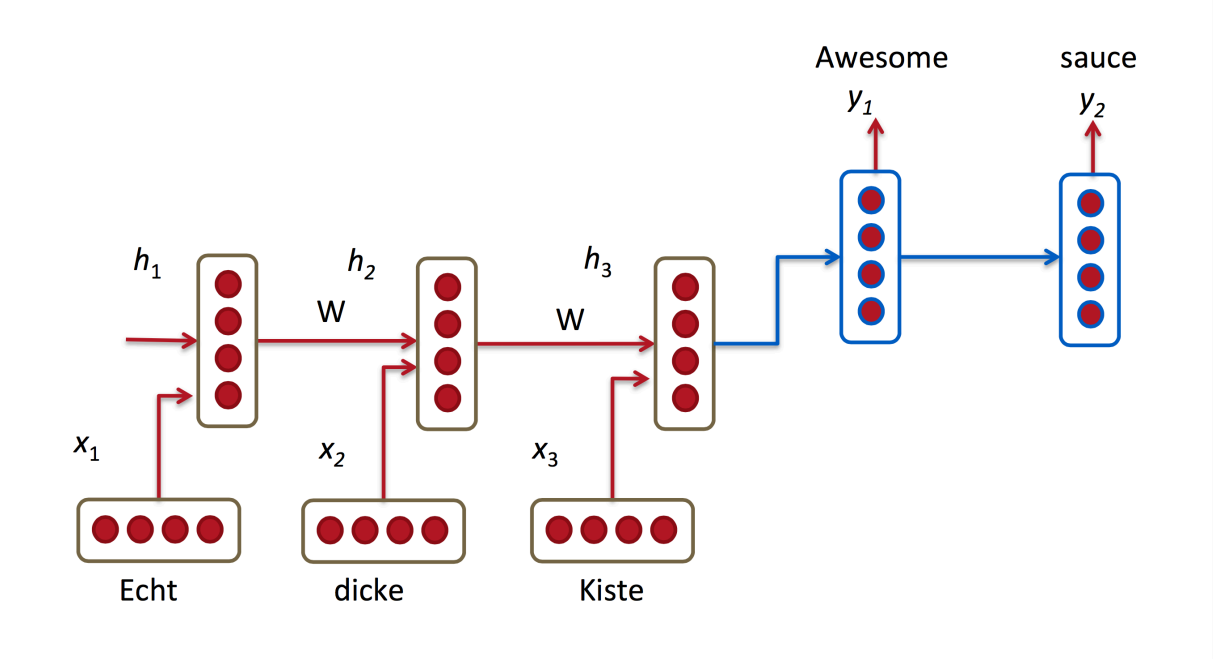
- Encoder-decoder architectures (Sutskever et al., 2014; Cho et al., 2014):



- First encode, then decode back.

Encoder-decoder architectures

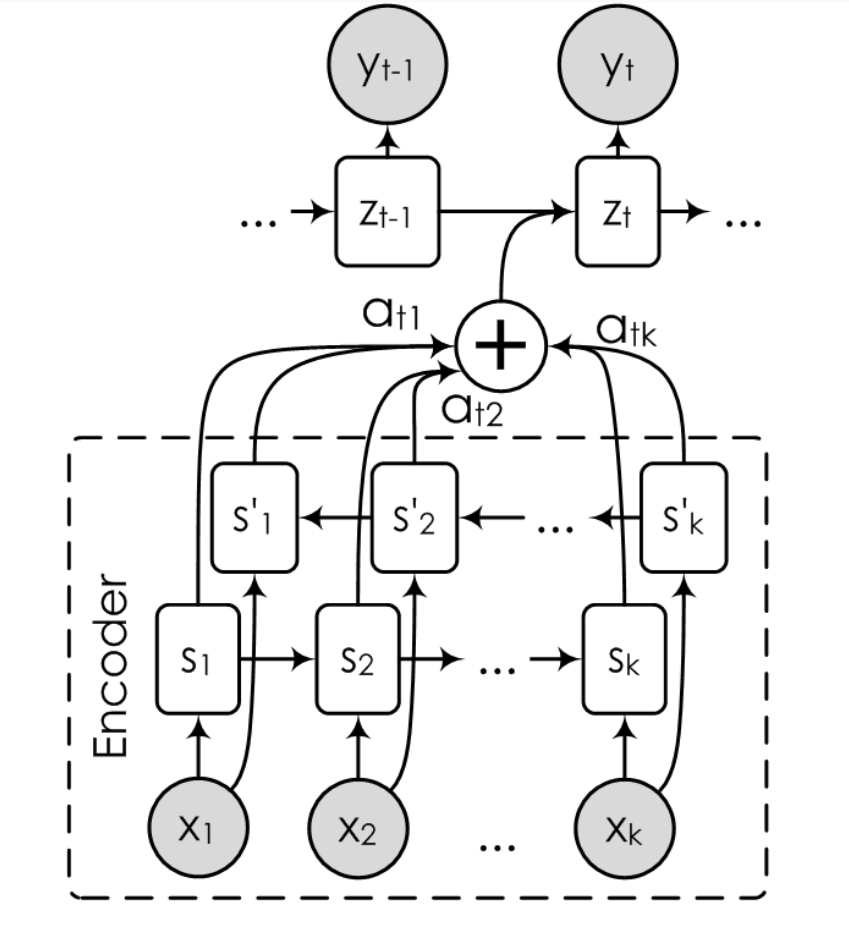
- The same can work with machine translation:



- Problem: we need to compress everything down to a single vector.
- It won't work with long sequences, even long-ish individual sentences.

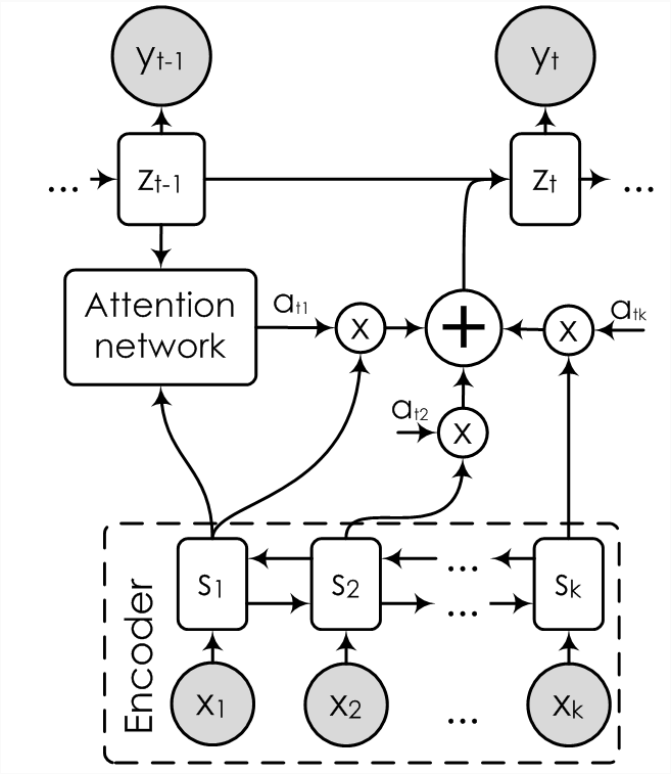
Attention in neural networks

- Solution: let's train special weights that show how important a given part of the input is for the current output.
- Direct application of the idea – bidirectional LSTM plus attention (Bahdanau et al. 2014):



Attention in neural networks

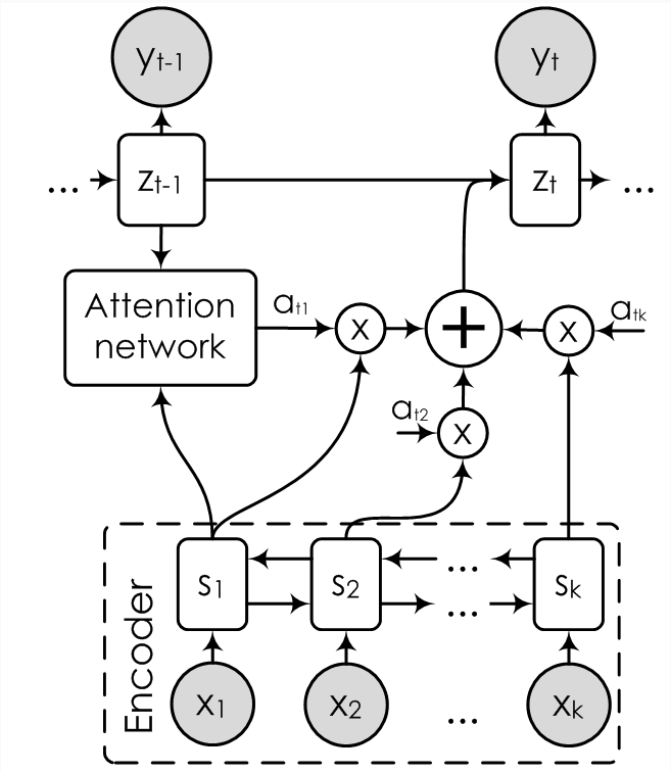
- *Soft attention* (Luong et al. 2015a; 2015b; Jean et al. 2015):
  - encoder is a bidirectional RNN, with both contexts;
  - the attention network estimates relevance – do we need to translate this word right now?



Attention in neural networks

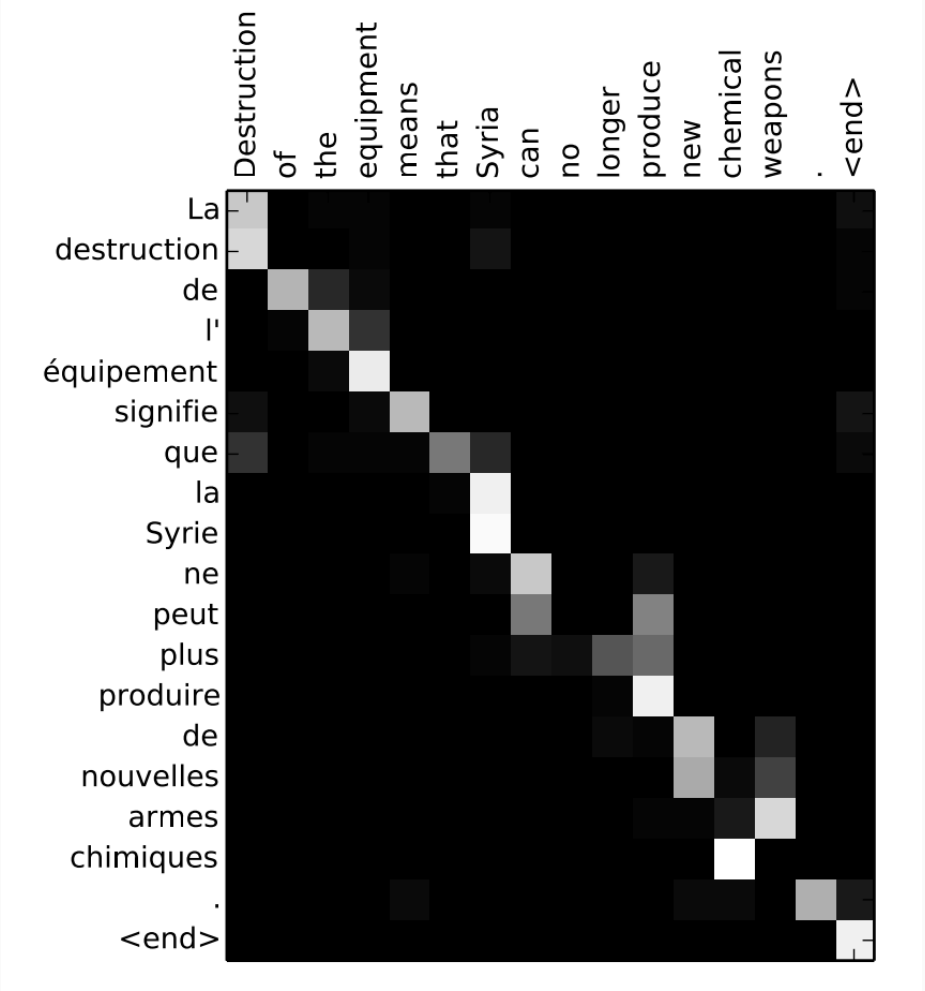
- Formally it's very simple: compute attention weights α_{tj} and re-weigh context vectors:

$$e_{tj} = a(z_{t-1}, j), \quad \alpha_{tj} = \mathrm{softmax}(e_{tj}; e_{t*}),$$
$$c_t = \sum_j \alpha_{tj} h_j, \text{ and now } z_t = f(s_{t-1}, y_{t-1}, c_i).$$



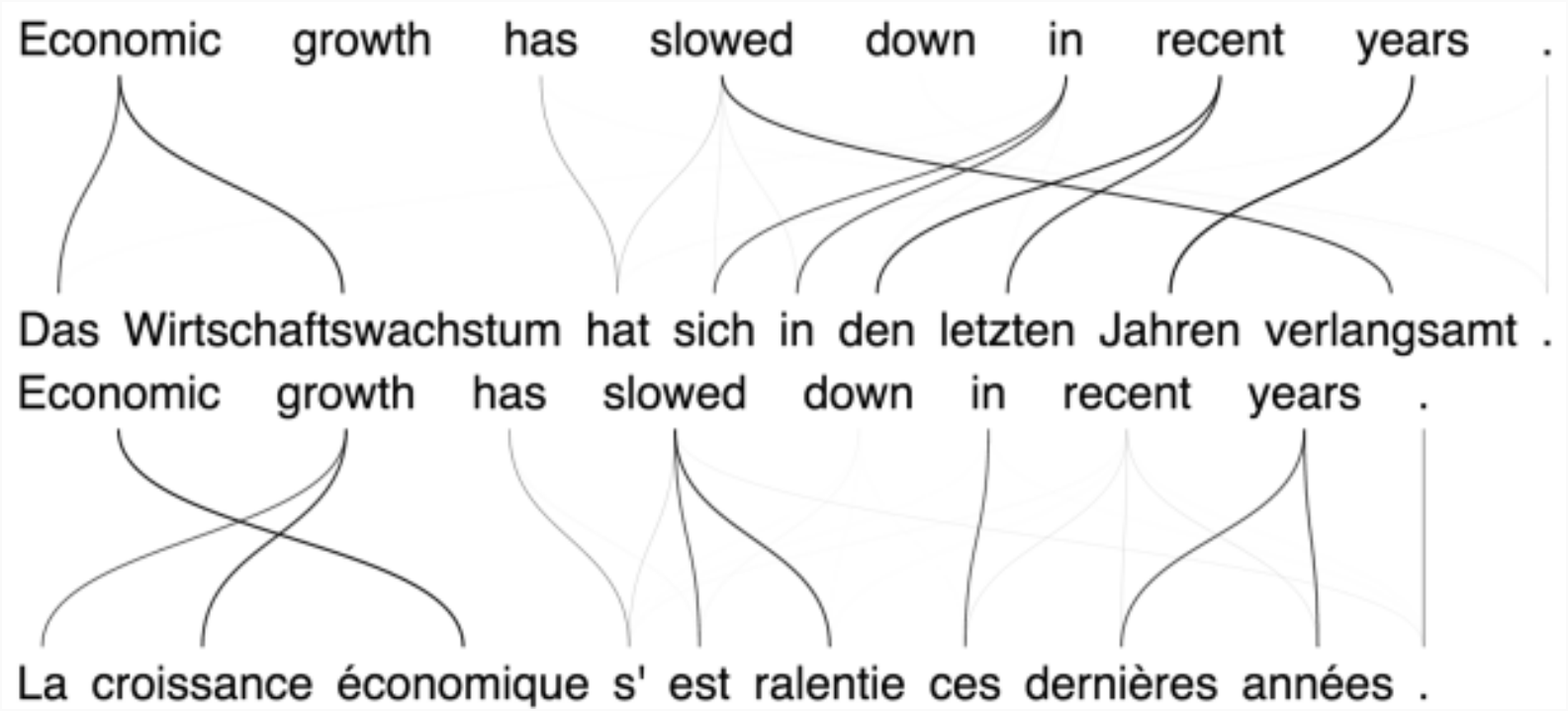
Attention in neural networks

- We can visualize what the network is looking at:



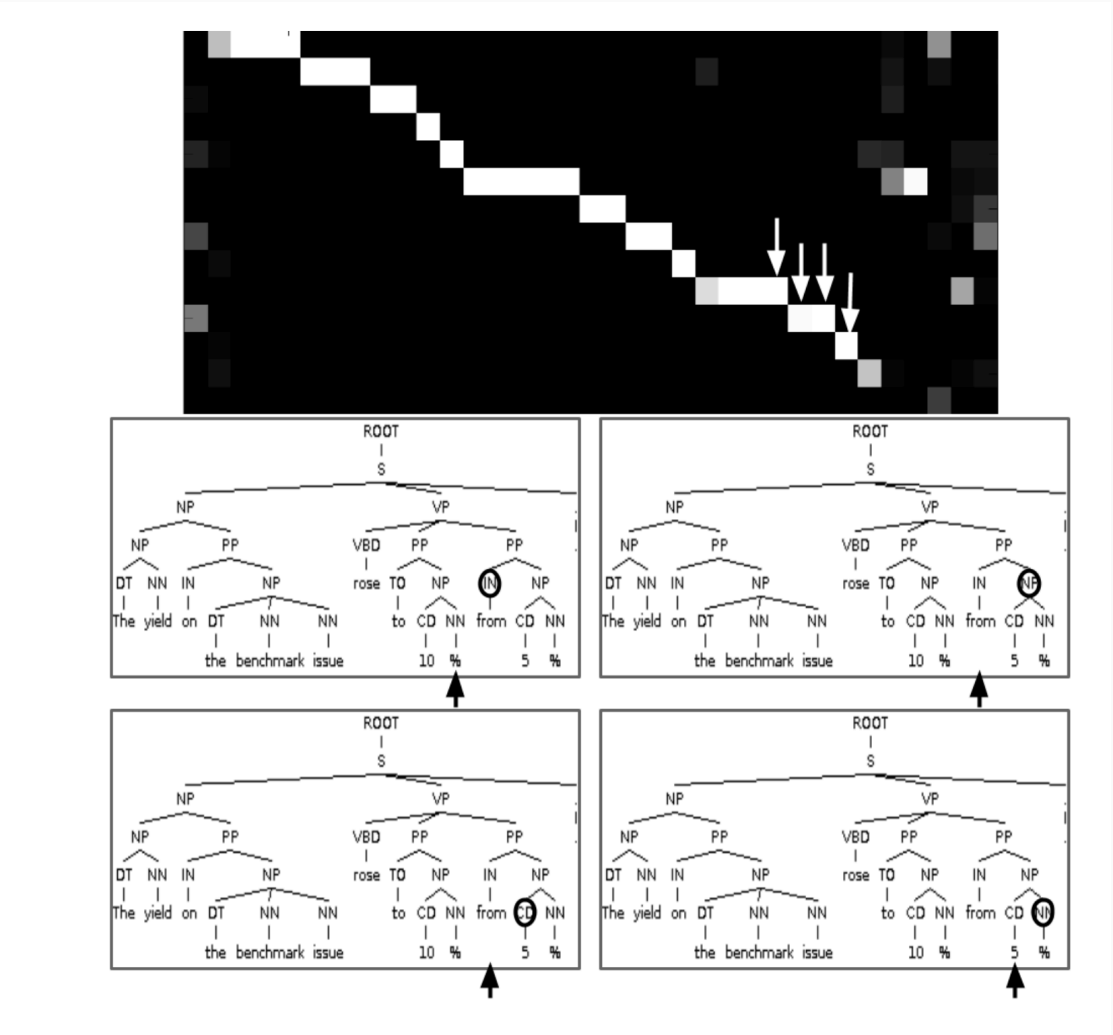
Attention in neural networks

- And we get a much better word order:



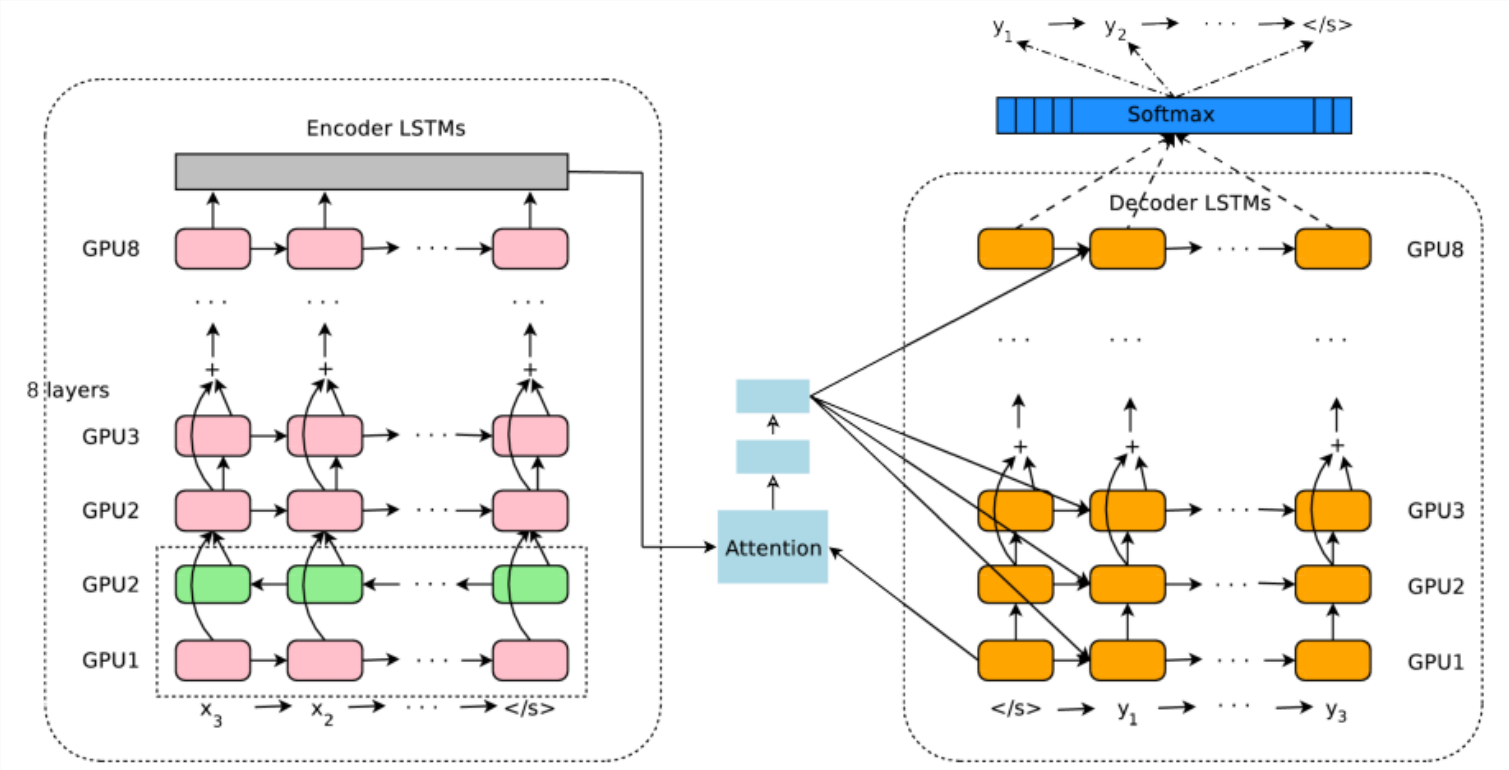
Attention in neural networks

- One unusual application – “Grammar as a Foreign Language” (Vinyals et al., 2015)



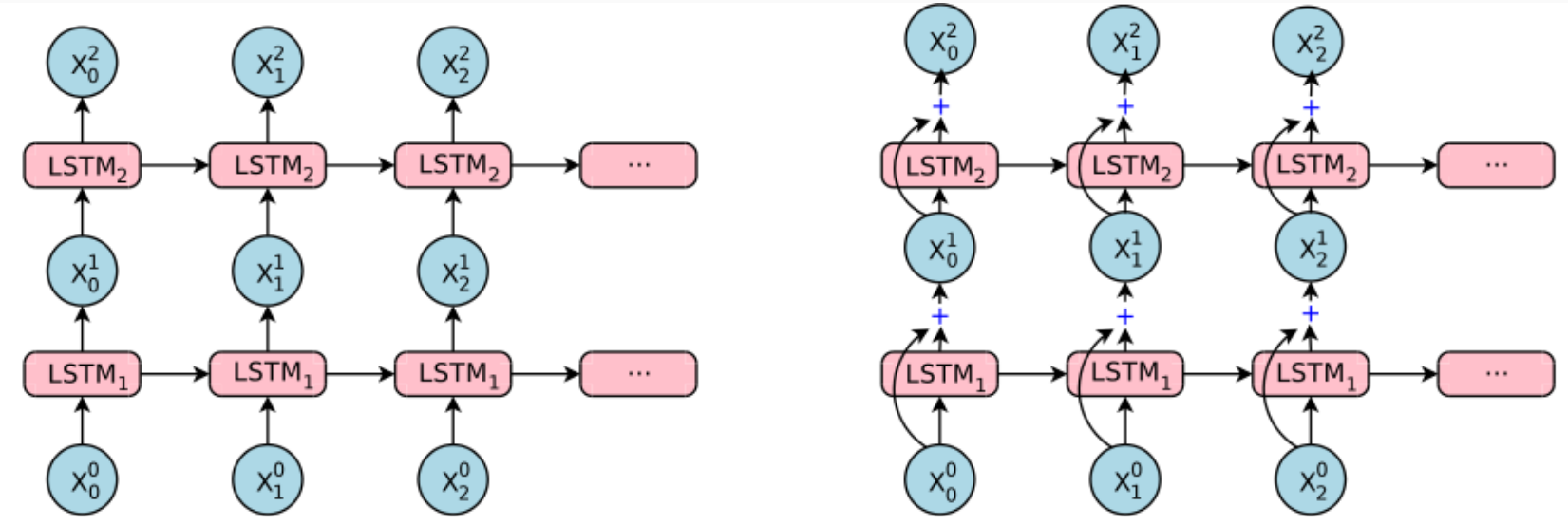
Google Translate

- (Wu et al., 2016): *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*:
  - the basic architecture of *Google Translate* was the same: encoder, decoder, attention;
  - RNNs are much deeper, 8 layers in both encoder and decoder:



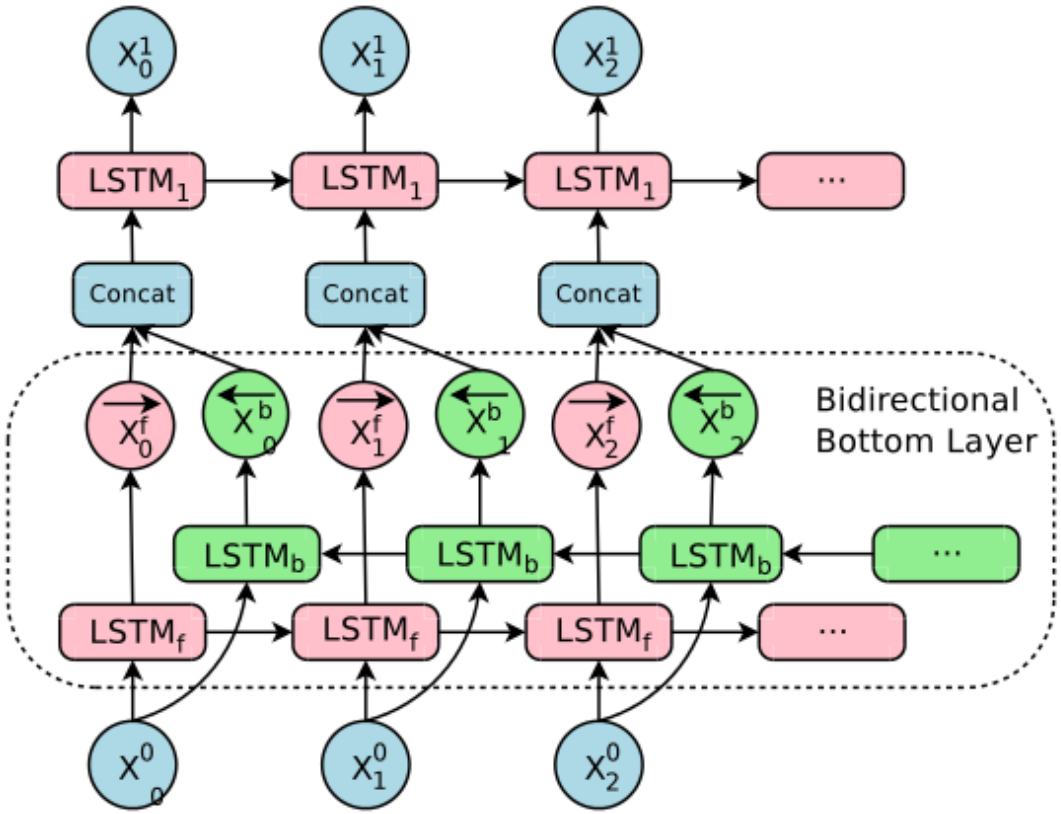
Google Translate

- (Wu et al., 2016): *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*:
  - stacked LSTM stop working beyond 4-5 layers;
  - so they add residual connections, like *ResNet*:



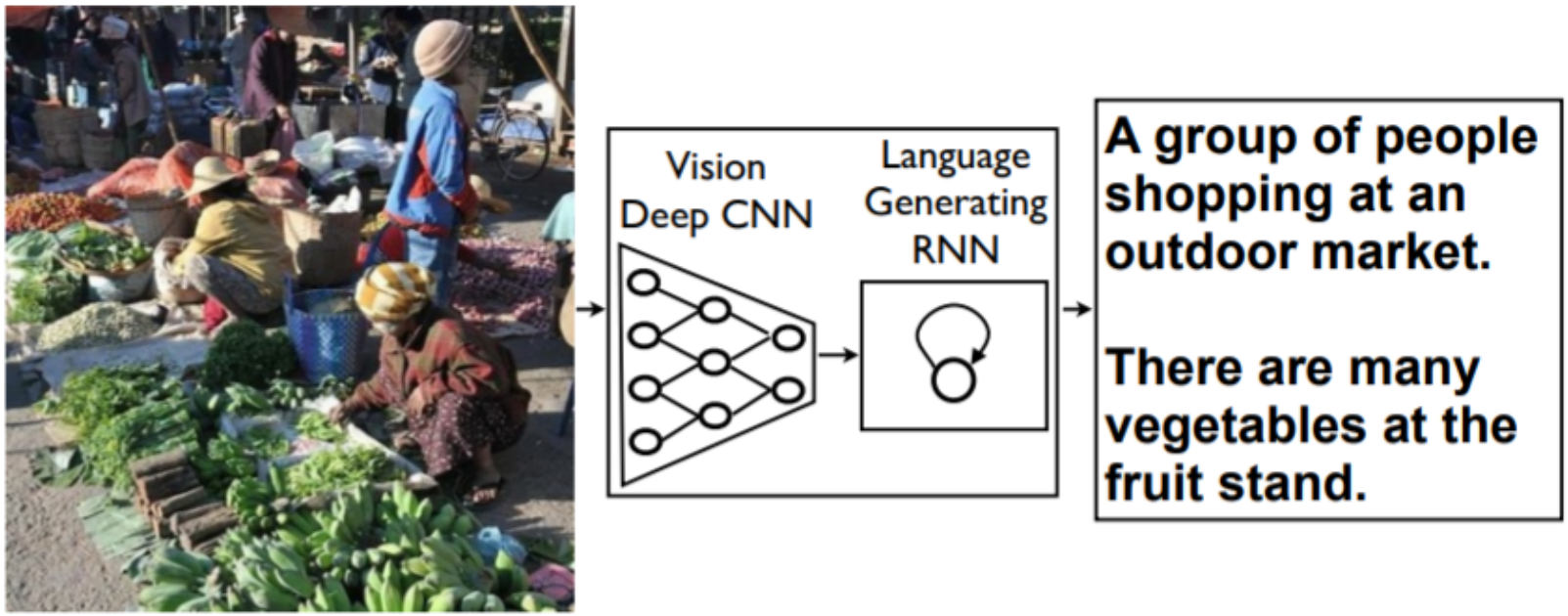
Google Translate

- (Wu et al., 2016): *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*:
  - the bottom layer is bidirectional:



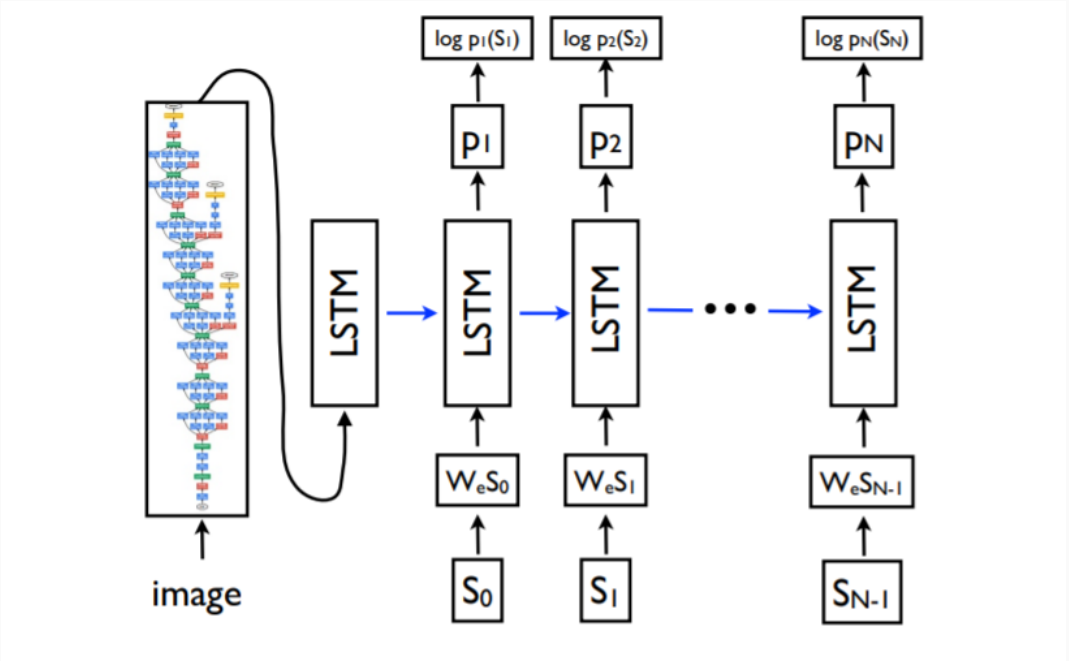
Show, Attend, and Tell

- Now let's get back to images and try to generate subtitles for them.
- First work – "Show and Tell" (Vinyals et al., 2015):



Show, Attend, and Tell

- Straightforward architecture:
  - objective function is just $\sum_{(I,S)} \log p(S \mid I; \theta)$, where I is the image and S is the description;
  - decompose and model $p(S_t \mid I, S_0, \ldots, S_{t-1})$ with an LSTM-based RNN;
  - and use a CNN to extract image features.

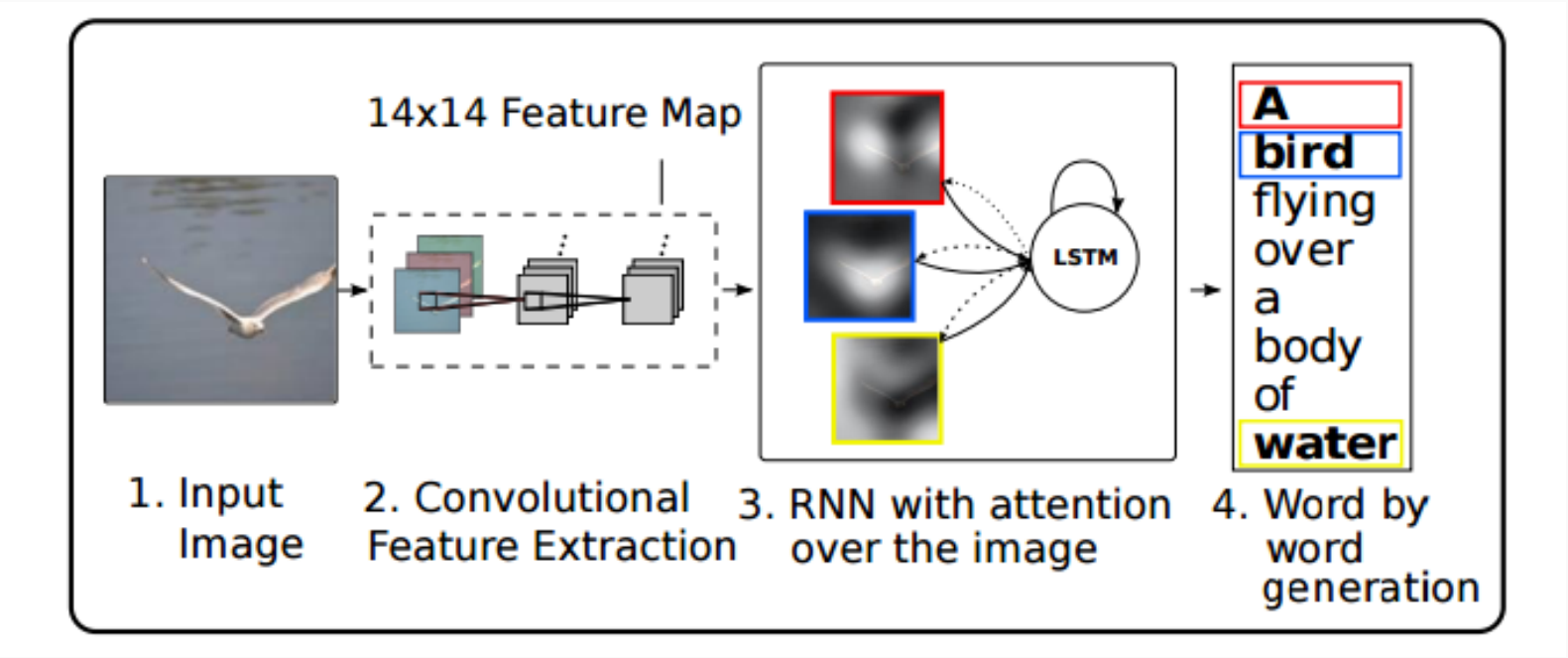


Show, Attend, and Tell

- It was reasonable but not very good:

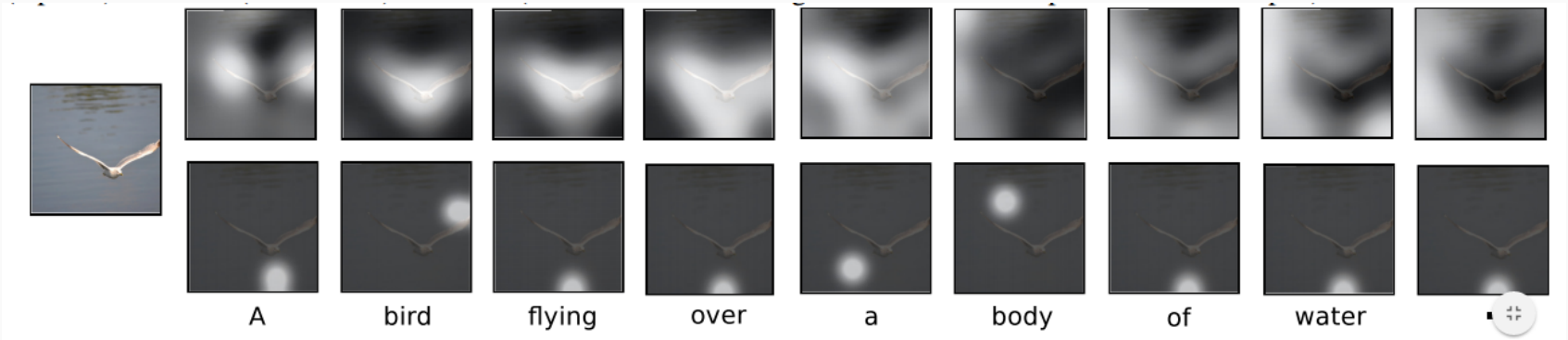
Show, Attend, and Tell

- "Show, Attend, and Tell" (Xu et al., 2015):



Show, Attend, and Tell

- Soft attention vs. hard attention:



- Soft attention – create an annotation with weights

$$\phi(\{\mathbf{a}\}_i, \{\alpha_i\}) = \sum_{i=1}^{L} \alpha_{t,i} \mathbf{a}_i.$$

Show, Attend, and Tell

- Hard attention is trained by maximizing the variational lower bound

$$L_s = \sum_s p(s \mid \mathbf{a}) \log p(\mathbf{y} \mid s, \mathbf{a}) \leq \log \sum_s p(s \mid \mathbf{a}) p(\mathbf{y} \mid s, \mathbf{a}) = \log p(\mathbf{y} \mid \mathbf{a}).$$

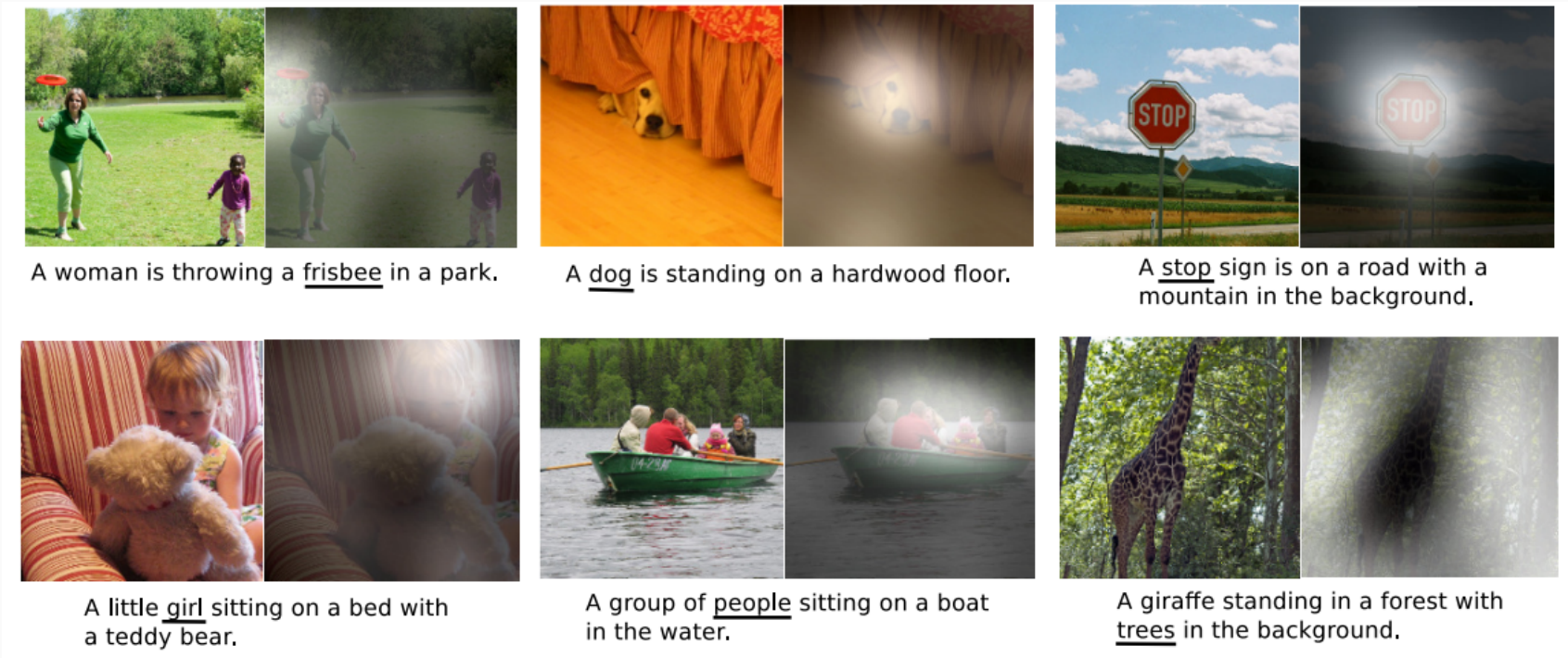
- We can take derivatives of L_s:

$$\frac{\partial L_s}{\partial W} = \sum_s p(s \mid \mathbf{a}) \left[\frac{\partial \log p(\mathbf{y} \mid s, \mathbf{a})}{\partial W} + \log p(\mathbf{y} \mid s, \mathbf{a}) \frac{\partial \log p(s \mid \mathbf{a})}{\partial W} \right].$$

- And then sample s_t with probabilities α_i and approximate the expectation with a sample.

Show, Attend, and Tell

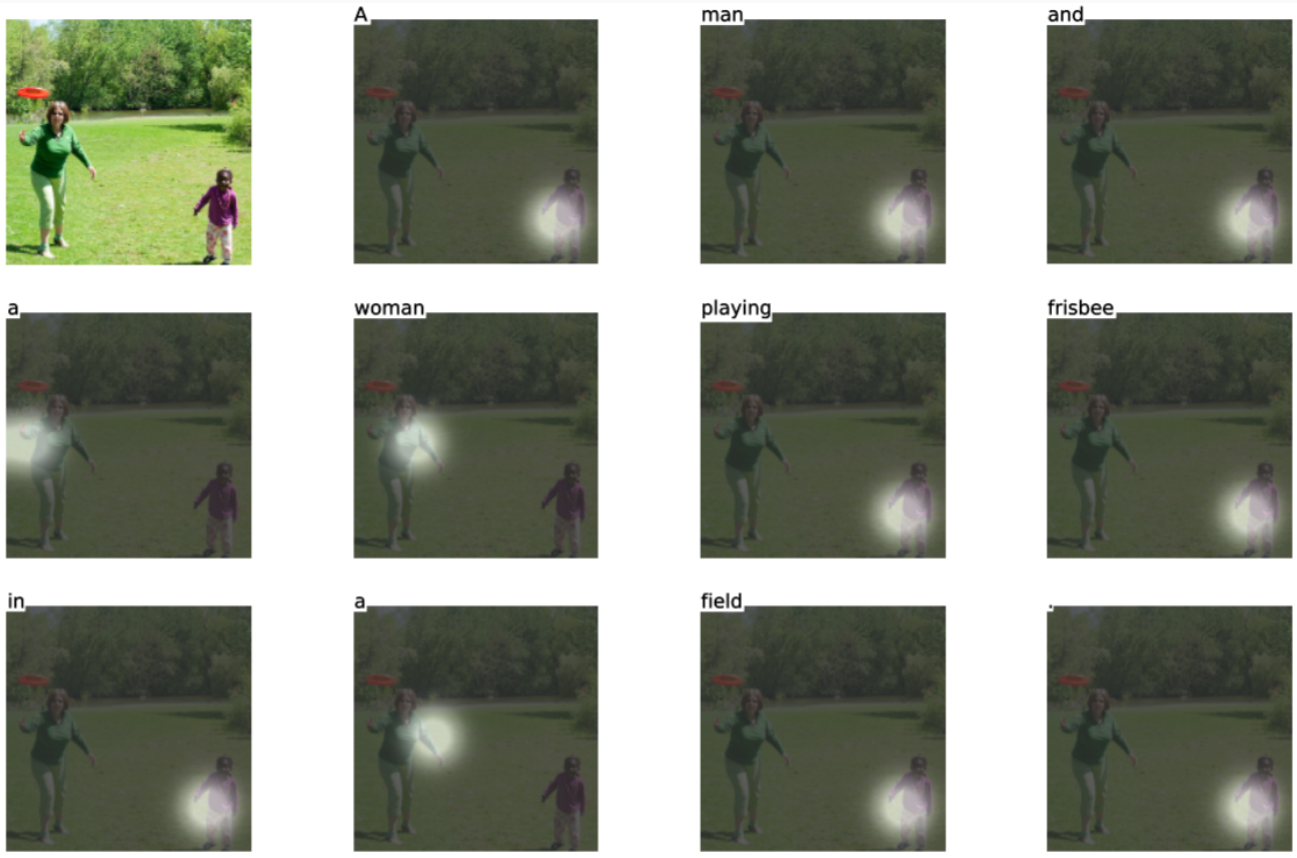
- Show, Attend, and Tell is often very good:



- And when it fails we can try to see why.

Show, Attend, and Tell

- Examples – hard attention:



(a) A man and a woman playing frisbee in a field.

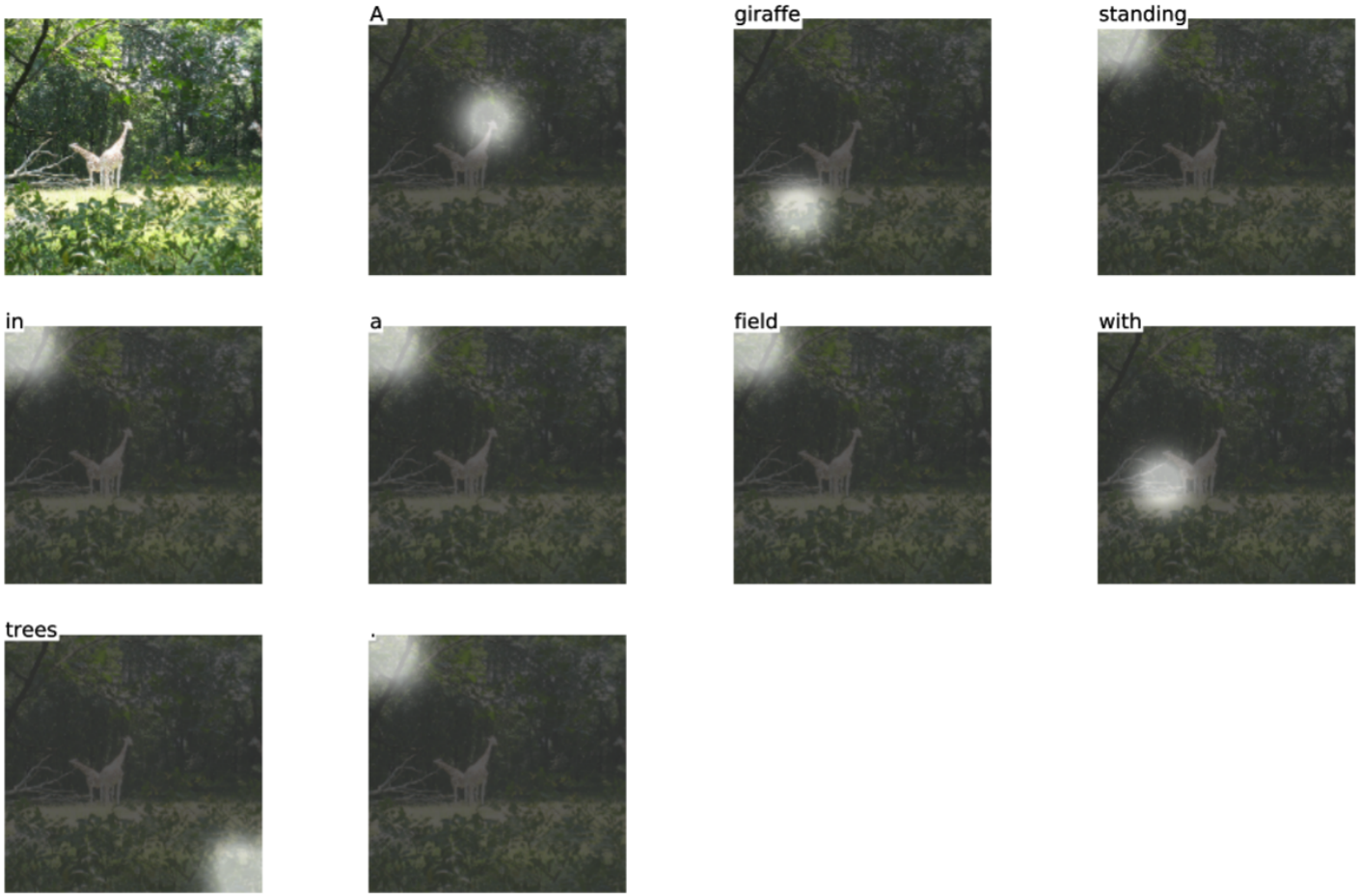
Show, Attend, and Tell

- Examples – soft attention:

(b) A woman is throwing a frisbee in a park.

Show, Attend, and Tell

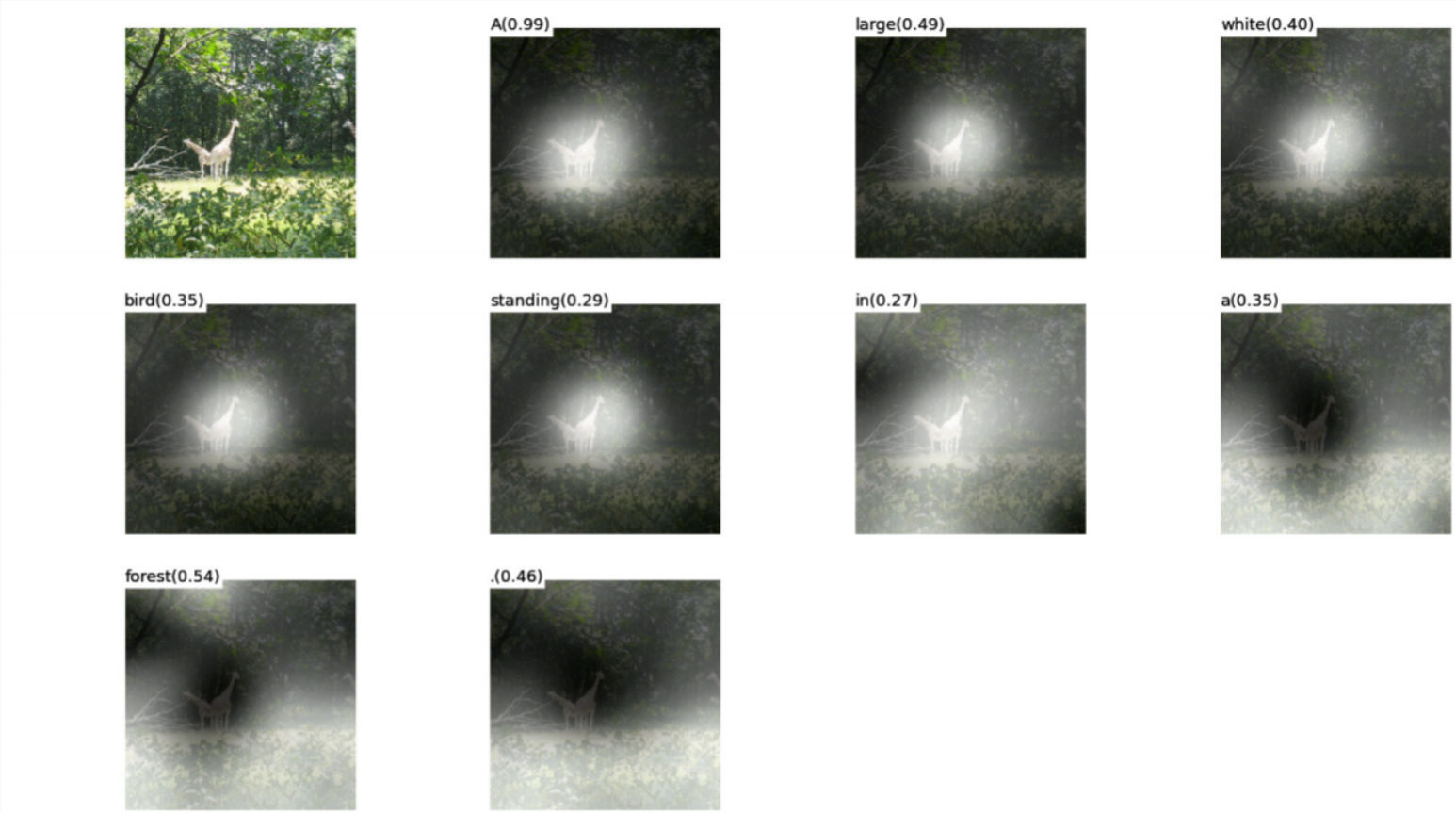
- Examples – hard attention:



(a) A giraffe standing in the field with trees.

Show, Attend, and Tell

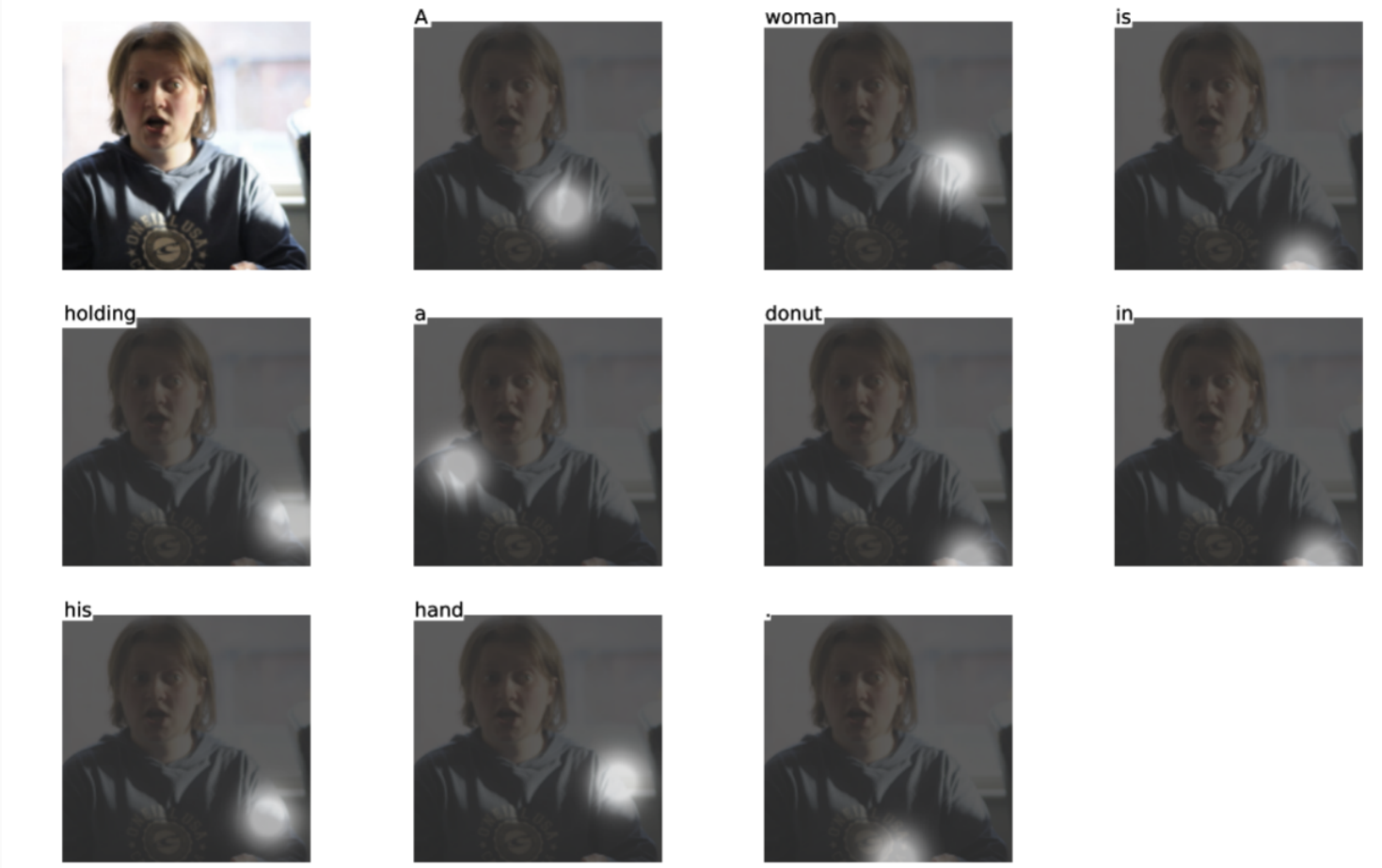
- Examples – soft attention:



(b) A large white bird standing in a forest.

Show, Attend, and Tell

- Examples – hard attention:



(a) A woman is holding a donut in his hand.

Show, Attend, and Tell

- Examples – soft attention:

(b) A woman holding a clock in her hand.

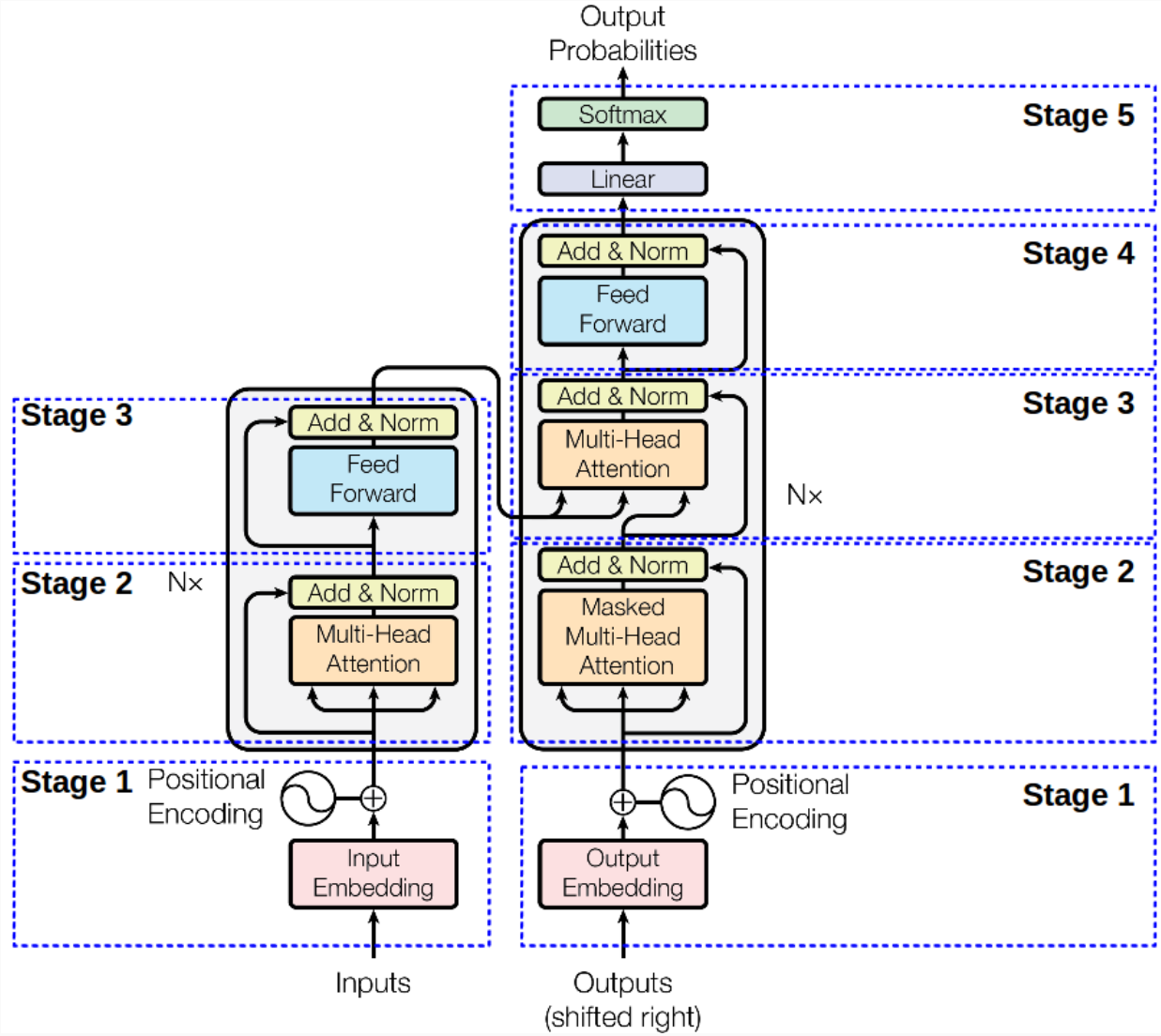
Transformers are all you need

Transformer

- In 2017, it turned out that attention-based encoder-decoder architectures can be both easier and more interesting
- *Attention is all you need* (Vaswani et al., 2017)
- The main idea, *self-attention*, turns out to be very fruitful for various *seq2seq* problems
- The primary motivation is to try and avoid encoding sequences with a fixed size vector

Transformer

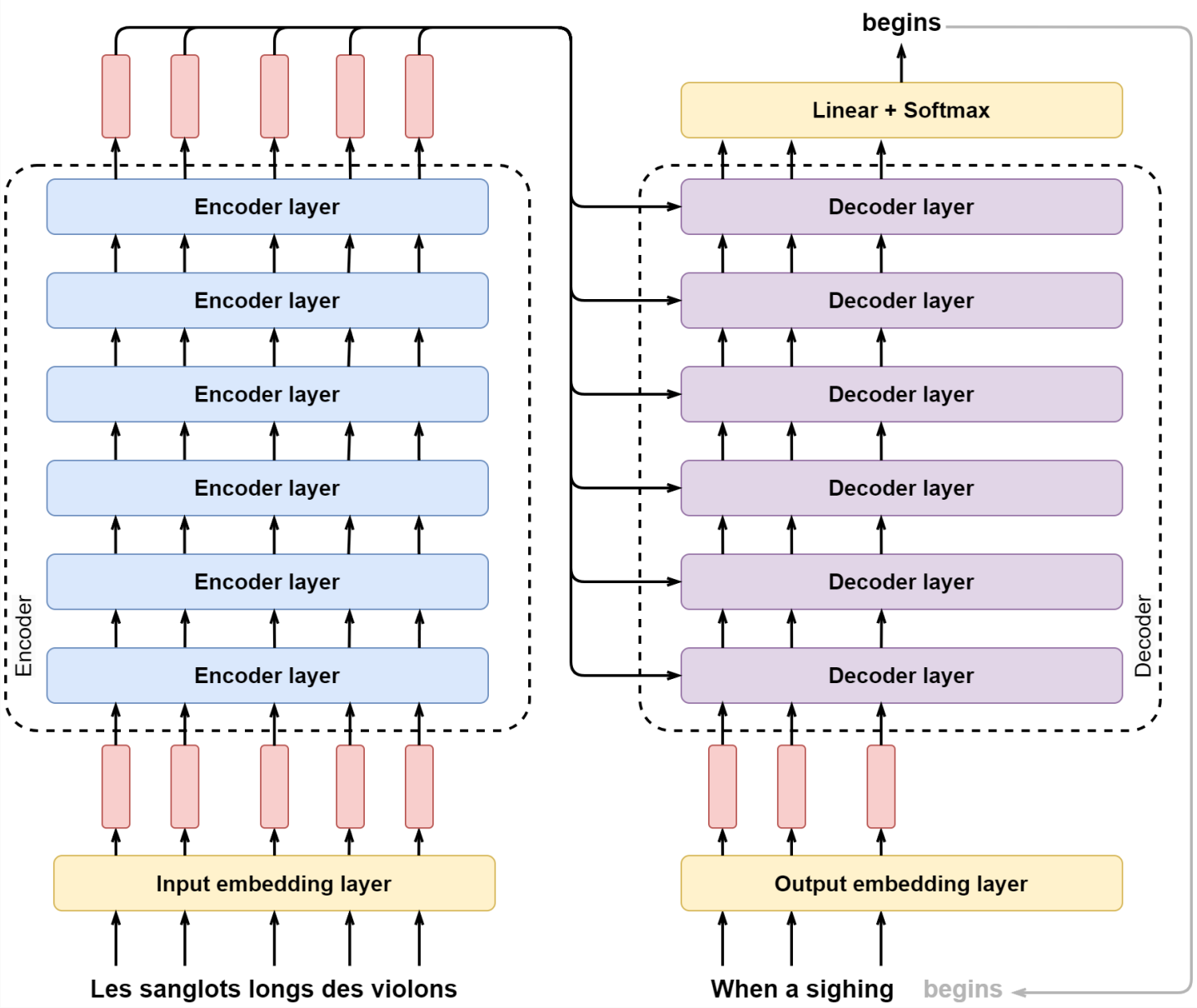
- General flowchart:



- Now in more detail...

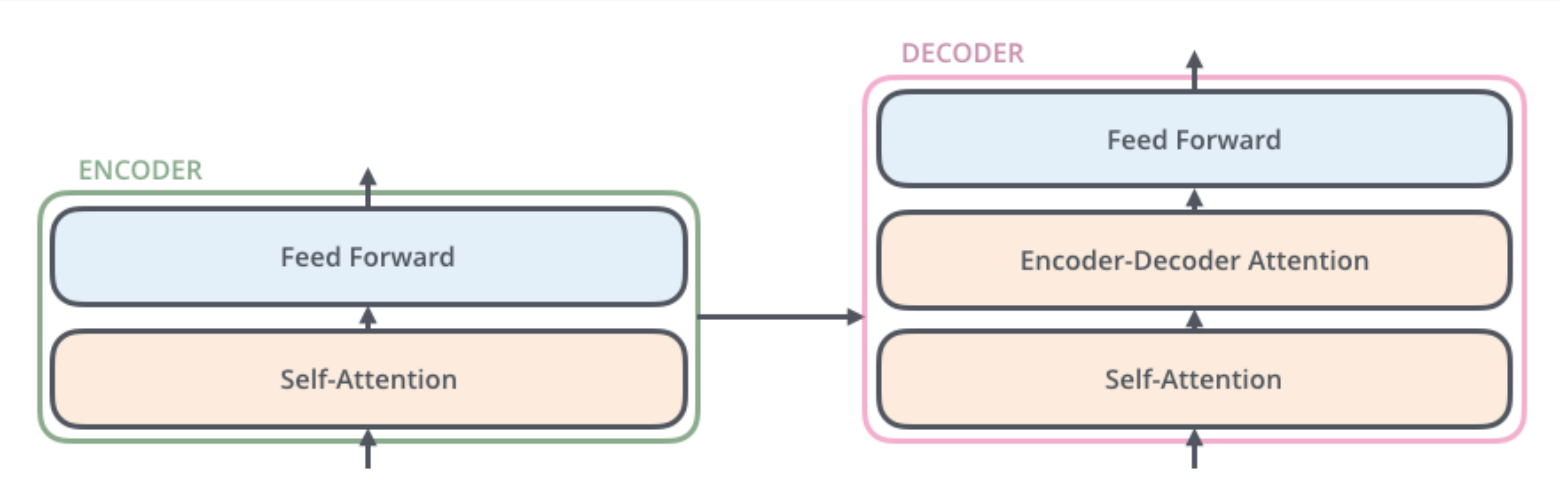
Transformer

- The essence is, again, an encoder-decoder architecture:



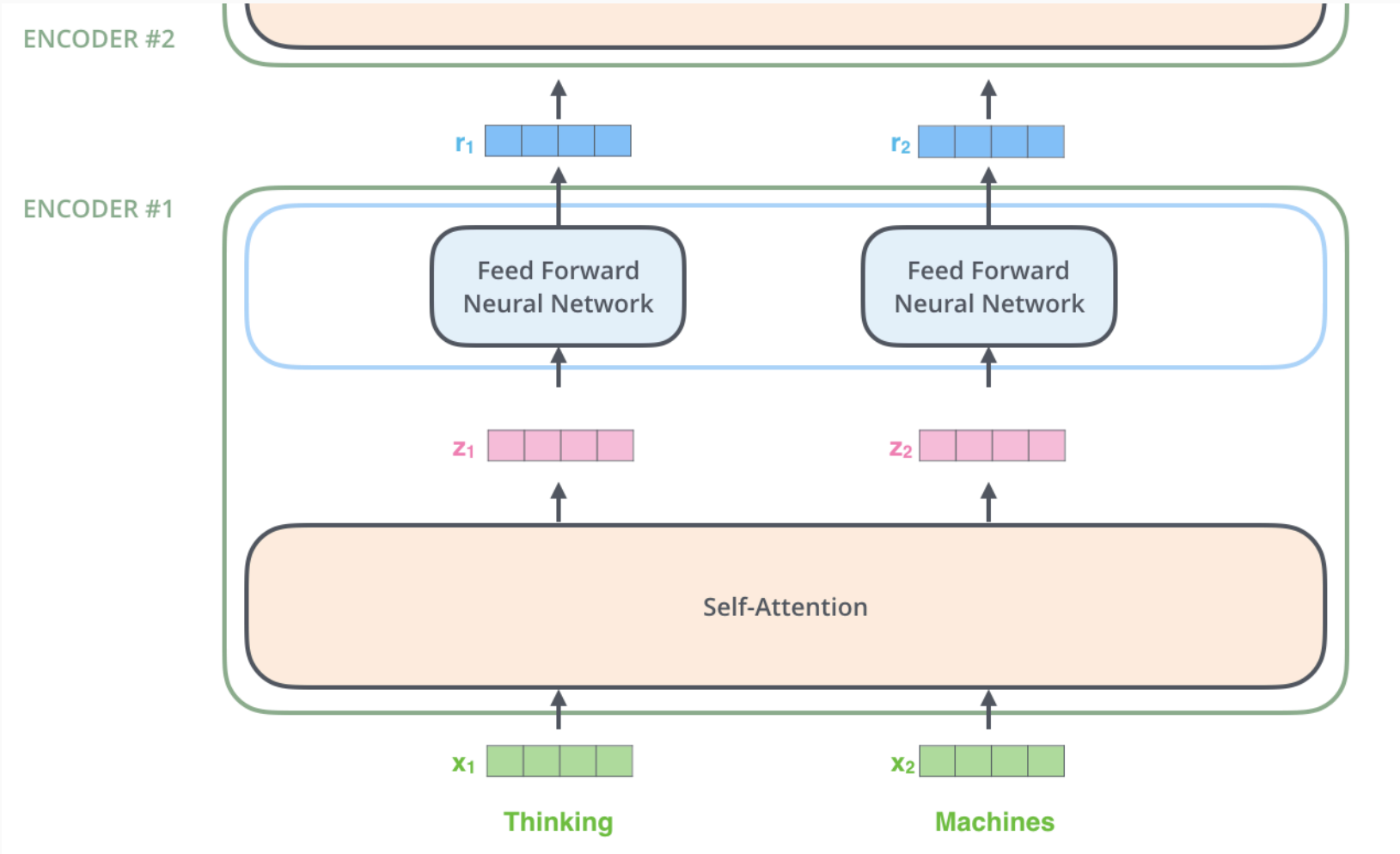
Transformer

- Each block consists of a self-attention layer followed by a feedforward layer that is applied independently to each input position:



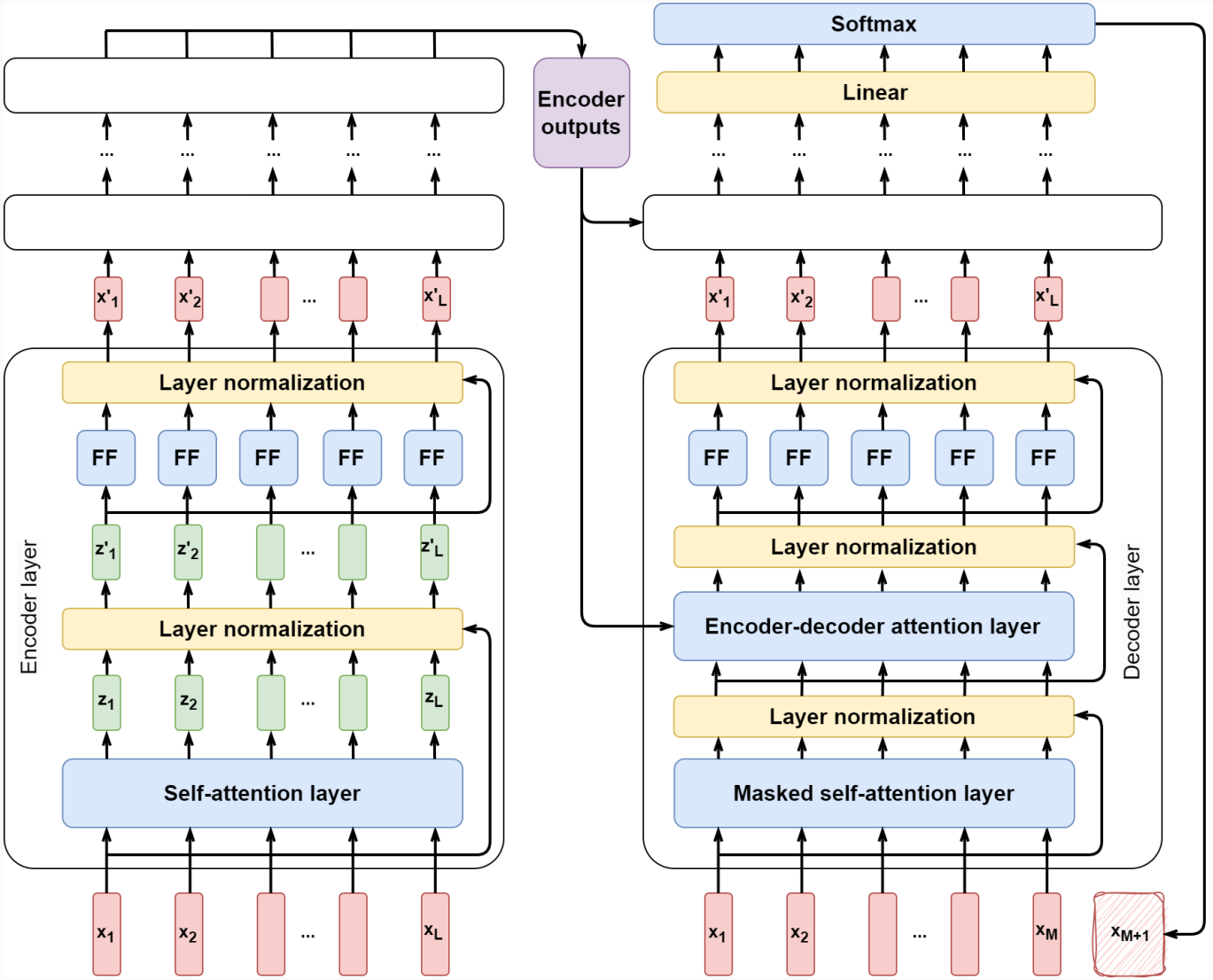
Transformer

- Words are represented by vectors, and the feedforward layer does everything in parallel:



Transformer

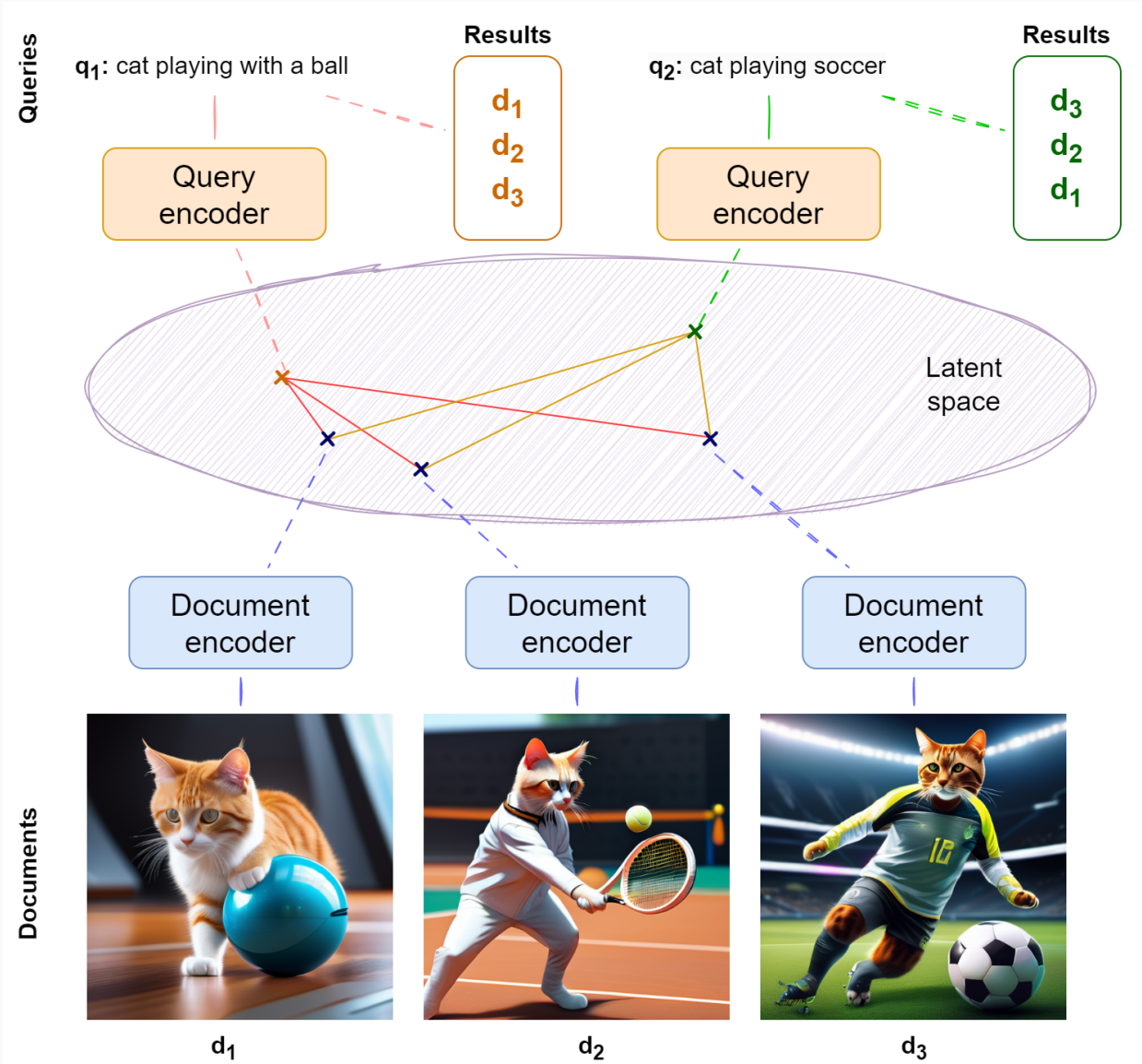
- Overall more detailed picture:



- But what is self-attention?

Self-attention

- To understand self-attention, let us begin with information retrieval.



Self-attention

- Both queries and documents share the same latent space, although the ways of encoding them into this latent space may be different (after all, even if both queries and documents are texts they have very different properties)
- A search query is put through a query encoder to get to the latent space
- Documents (represented by images in the figure) are also represented in the same latent space via a different encoder
- To find the most relevant documents, we simply find the nearest neighbors for the query in the latent space among the documents; one often assumes that the latent space is linear and the distance metric there is just the scalar product of vectors, $\mathrm{dist}(q, d) = \mathrm{Enc}_q(q)^\top \mathrm{Enc}_d(d)$.

Self-attention

- In self-attention, this intuition comes alive very abstractly. The layer receives as input a sequence of vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_L$, i.e., a matrix $X \in \mathbb{R}^{d \times L}$.
- Queries, keys, and documents come from $\mathbf{x}_i$ themselves:
  - multiplying $\mathbf{x}_1$ by a weight matrix W^Q, we get the *query* vector $\mathbf{q}_1 = W^Q \mathbf{x}_1$; note that the dimension q of the query vectors, $\mathbf{q}_i \in \mathbb{R}^q$, may be different from (usually lower than) the input dimension d, $\mathbf{x}_i \in \mathbb{R}^d$;
  - multiplying $\mathbf{x}_i$ by a weight matrix W^K, we get the *key* vectors $\mathbf{k}_i = W^K \mathbf{x}_i$ for $i = 1, \ldots, L$; since we want queries and keys to inhabit the same latent space, we have the keys with the same dimension as queries, $\mathbf{k}_i \in \mathbb{R}^q$, so $W^V \in \mathbb{R}^{q \times d}$;
  - the third weight matrix W^V gets us *value* vectors $\mathbf{v}_i = W^V \mathbf{x}_i$ for $i = 1, \ldots, L$; these are the *documents* that we will "retrieve" by their keys $\mathbf{k}_i$; formally we have a different dimension v for the values, $\mathbf{v}_i \in \mathbb{R}^v$ and $W^V \in \mathbb{R}^{v \times d}$; in practice usually $v = q$.

Self-attention

- Then we do the retrieval part, computing attention scores as scalar products between queries and documents.
- We need to rescale $\mathbf{q}_i^\top \mathbf{v}_j$, dividing it by $\sqrt{q}$, and get the scores through softmax to turn them into probabilities:

$$\alpha_{ij} = \mathrm{softmax}\left(\frac{1}{\sqrt{q}}\mathbf{q}_i K^\top\right)_j,$$

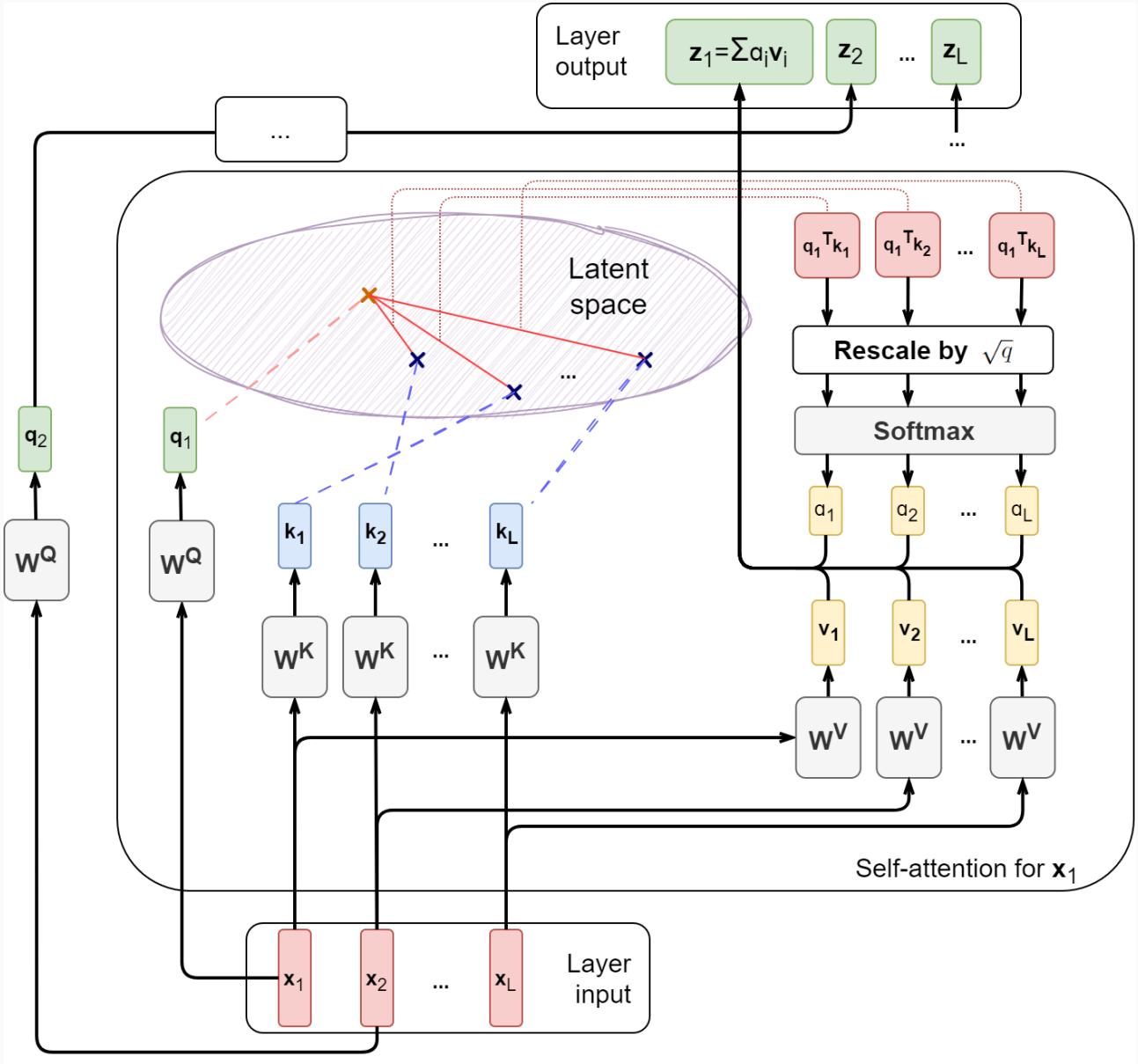
  where $K \in \mathbb{R}^{q\times L}$ are all the keys combined in a matrix, $K = W^K X$.
- Then we use the result as coefficients for a convex combination of *values* $\mathbf{v}_j$:

$$\mathbf{z}_i = \mathrm{softmax}\left(\frac{1}{\sqrt{q}}\mathbf{q}_i K^\top\right) V,$$

  where $V \in \mathbb{R}^{q\times L}$ are the values, $V = W^V X$.

Self-attention

- Overall structure of a self-attention layer:



Self-attention

- And that's it for the intuition! We can combine the computation of each $\mathbf{z}_i$ into a single formula in matrix form:

$$Z = \text{softmax}\left(\frac{1}{\sqrt{q}} QK^\top\right) V.$$

- But this is only one way to "look at" the input vectors! What we have defined now is not the full self-attention layer but only a single *self-attention head*.
- We parallelize these computations along H different heads, using different weight matrices $W_1^Q, \ldots W_H^Q$, $W_1^K, \ldots W_H^K$, and $W_1^V, \ldots W_H^V$. This is known as *multi-head attention*.

Self-attention

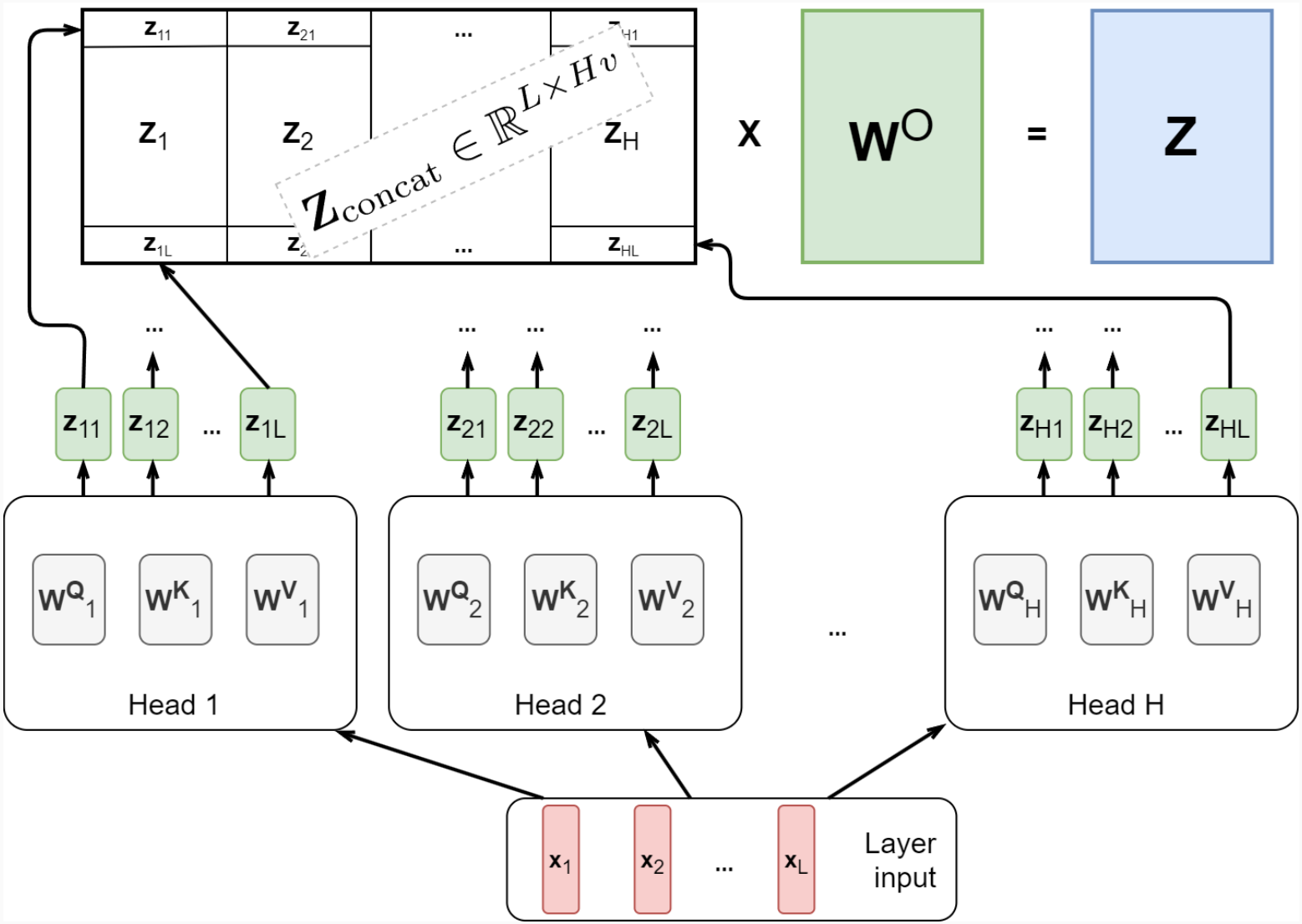
- After H parallel heads we get $v \times L$ output matrices $Z_1, Z_2, \ldots, Z_H$, concatenate them all into $Z_{\text{concat}} \in \mathbb{R}^{L \times Hv}$, and add another weight matrix $W^O \in \mathbb{R}^{Hv \times d}$ to compress the result back to the necessary dimension:

$$Z = \left(Z_{\text{concat}} W^O\right)^\top.$$

- With W^O, the self-attention layer can mix up representations obtained from different attention heads; this is also an important way to add more flexibility and expressiveness to the architecture.

Self-attention

- Multi-head attention:



- This is the most important part, but we are not yet done.

The decoder side

- The decoder part has two more layer types: *masked self-attention* and *encoder-decoder attention*.
- Masked attention basically means that since the decoder works autoregressively, it should not peek at the tokens that it has not produced yet.
- To do that, it's easier to just input the whole sequence and mask future positions inside self-attention layers themselves.
- Formally this means that we set the softmax arguments to $-\infty$ for future tokens, which means that their attention weights will always be zero.

The decoder side

- Encoder-decoder attention is a variation on the self-attention mechanism that conditions itself on the results of the encoder.
- Sounds difficult… but in fact it's an almost trivial modification! We have the exact same self-attention layer but use different vectors as input:
  - to create the queries, we use vectors from the previous layer, i.e., current representations of already generated output tokens;
  - but for the "documents" in our "retrieval" task, i.e., for the key and value vectors, we use the vectors from the decoder.
- Informally, this means that we are doing "retrieval" on the encoder's output with queries made of already produced tokens.
- Formally, all dimensions match nicely: there are L terms in the softmax argument for each vector, but the number of queries and hence number of outputs matches the number of inputs.

Tokenization

- We use an embedding layer but how do we split text into tokens?
- *Byte-pair encoding*: an interesting idea based on Huffman coding.
- Let's begin by building a vocabulary:

$$\{\text{cat} : 10, \text{pet} : 12, \text{mat} : 5, \text{rat} : 8, \text{eats} : 4\}.$$

Original words:	cat		pet		mat		rat	eats
Frequencies:	10		12		5		8	4
Characters:	c	a	t	p	e	m	r	s
Frequencies:	10	27	39	12	16	5	8	4
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts
Frequencies:	10	**27**	12	12	5	8	4	4

Tokenization

- Then split it into symbols and count pairs of symbols:

$$\{\text{cat} : 10, \text{pet} : 12, \text{mat} : 5, \text{rat} : 8, \text{eats} : 4\},$$

$$\{c : 10, a : 27, t : 39, p : 12, e : 16, m : 5, r : 8, s : 4\},$$

$$\{ca : 10, at : 27, pe : 12, et : 12, ma : 5, ra : 8, ea : 4, ts : 4\}.$$

Original words:	cat		pet		mat		rat	eats
Frequencies:	10		12		5		8	4
Characters:	c	a	t	p	e	m	r	s
Frequencies:	10	27	39	12	16	5	8	4
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts
Frequencies:	10	**27**	12	12	5	8	4	4

Tokenization

- Now take the most frequent pair ("*at*") and re-encode it with a single new symbol (*Z*) that is added to the vocabulary:

$$\{\mathrm{cZ} : 10, \mathrm{pet} : 12, \mathrm{mZ} : 5, \mathrm{rZ} : 8, \mathrm{eZs} : 4\},$$
$$\{c : 10, Z : 27, t : 12, p : 12, e : 16, m : 5, r : 8, s : 4\},$$
$$\{cZ : 10, pe : 12, et : 12, mZ : 5, rZ : 8, eZ : 4, Zs : 4\}.$$

Original words:	cat		pet		mat		rat		eats
Frequencies:	10		12		5		8		4
Characters:	c	a	t	p	e	m	r	s	
Frequencies:	10	27	39	12	16	5	8	4	
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts	
Frequencies:	10	**27**	12	12	5	8	4	4	

Tokenization

- Now we can choose the new most frequent pair—in this case *pe* or *et*—and replace it with another new symbol, say *Y*:

$$\{\mathrm{cZ}:10,\mathrm{Yt}:12,\mathrm{mZ}:5,\mathrm{rZ}:8,\mathrm{eZs}:4\},$$
$$\{c:10,Z:27,Y:12,t:12,e:4,m:5,r:8,s:4\},$$
$$\{cZ:10,Yt:12,mZ:5,rZ:8,eZ:4,Zs:4\}.$$

- And so on, until we reach the necessary vocabulary size.

Original words:	cat		pet		mat		rat	eats
Frequencies:	10		12		5		8	4
Characters:	c	a	t	p	e	m	r	s
Frequencies:	10	27	39	12	16	5	8	4
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts
Frequencies:	10	**27**	12	12	5	8	4	4

Positional encoding

- In the Transformer, every input token can “look at” any other input token *directly*, with no regard for the distance between them in the input sequence.
- This is great compared to RNNs!
- But it means that we have a fixed bounded *context window* and, moreover, since attention weights cover all tokens uniformly, we *lose the sequence*: a self-attention layer does not know that some tokens are “next to each other” or “closer in the input sequence”.
- We need to give the Transformer an idea of the input sequence artificially; this is done via *positional encodings*.

Positional encoding

- Positional encodings are vectors added to the embedding that reflect where in the sequence the current token is.
- How could we encode the position? We could have a number that is increasing with the position but it is hard to get right:
  - if we use an increasing sequence like $1, 2, 3, ...$, it will not generalize to sequences longer than the usual length in the training set, and the network's behaviour would be much less well defined for large values;
  - if we use a given interval, say $[0, 1]$, and break it down into the necessary number of pieces, then the positional encoding would have no idea how many words are actually there between two tokens: the distance from 0 to $\frac{1}{2}$ could be 1 token or 100.

Positional encoding

- So we use something like positional numbers (where each digit forms a periodic function with different periods), but with continuous periodic functions, sine waves:

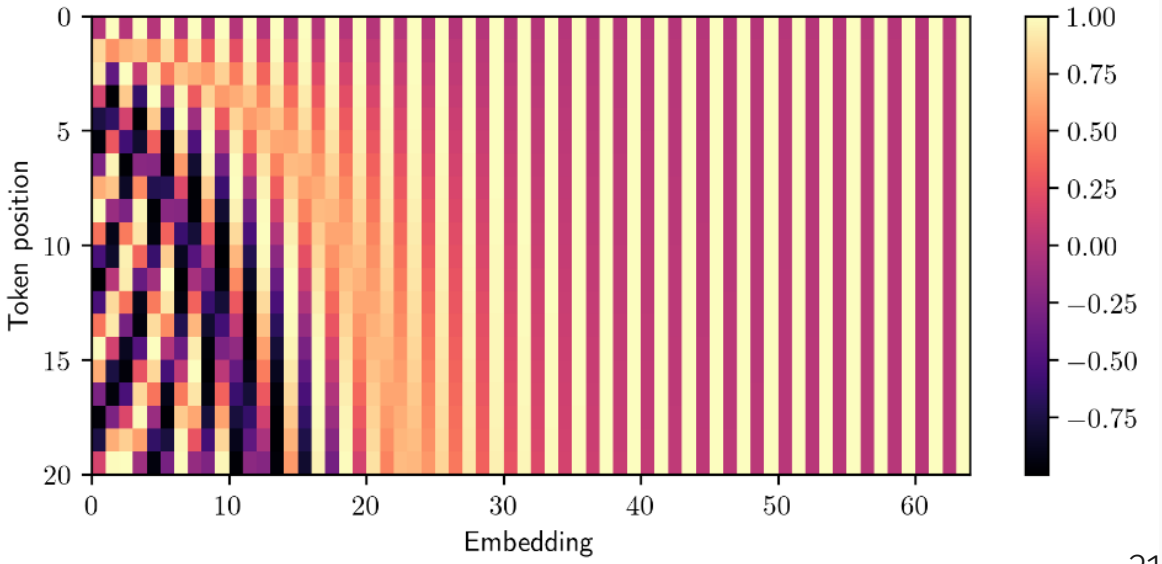
$$\mathrm{PE}\,(\mathrm{pos}, 2i) = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right), \quad \mathrm{PE}\,(\mathrm{pos}, 2i+1) = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right),$$

where pos is the token position, d is the embedding dimension.

Binary numbers

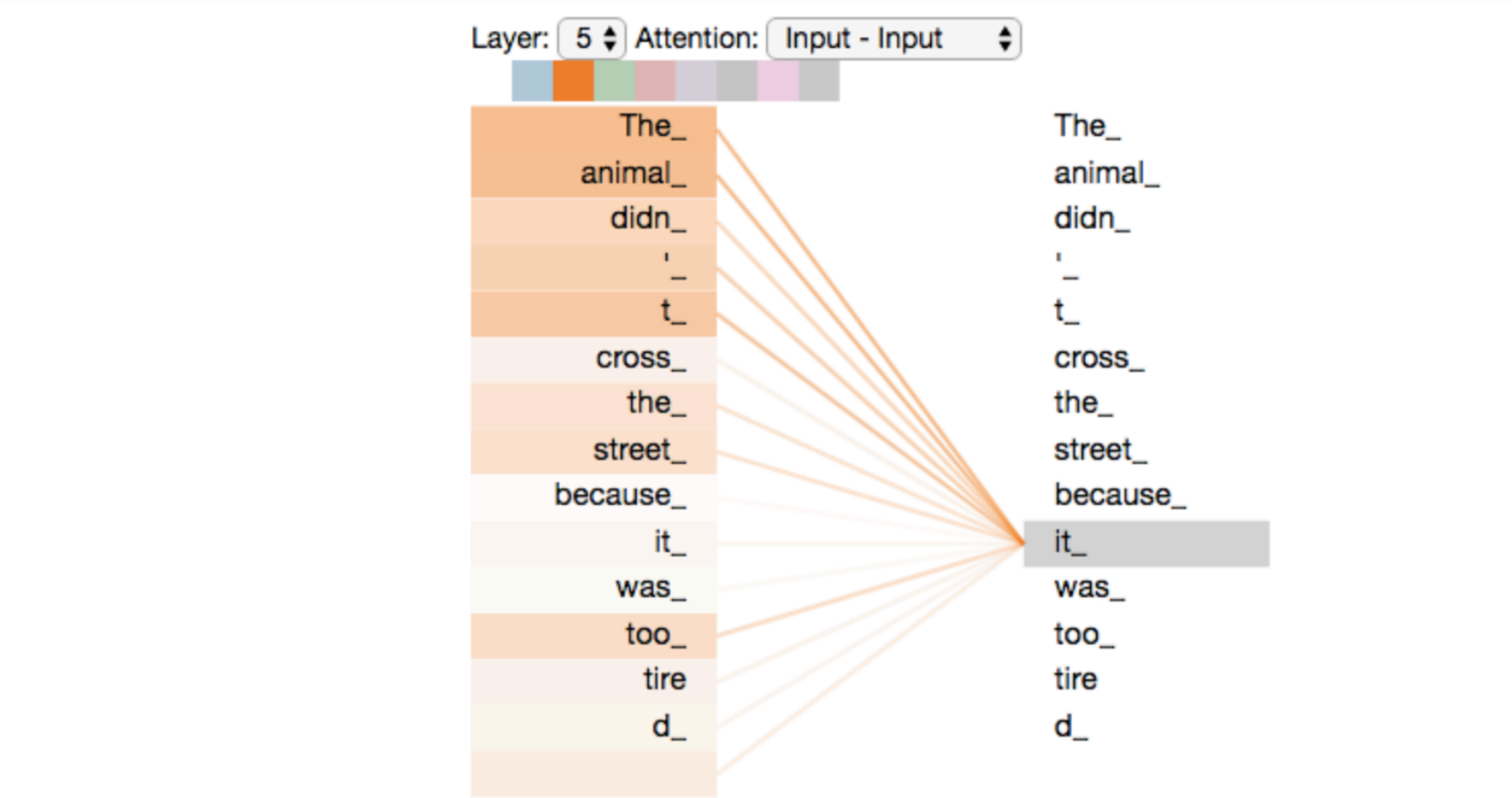
0000	1000
0001	1001
0010	1010
0011	1011
0100	1100
0101	1101
0110	1110
0111	1111

Positional encodings



Transformer results

- As a result, the Transformer trains weights for how other words participate in processing the current word:



Transformer results

- Self-attention is important: first, computationally; second, it reduces path lengths between words; third, potentially adds interpretability:

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Transformer results

- Works better, trains 100x faster:

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

Thank you for your attention!