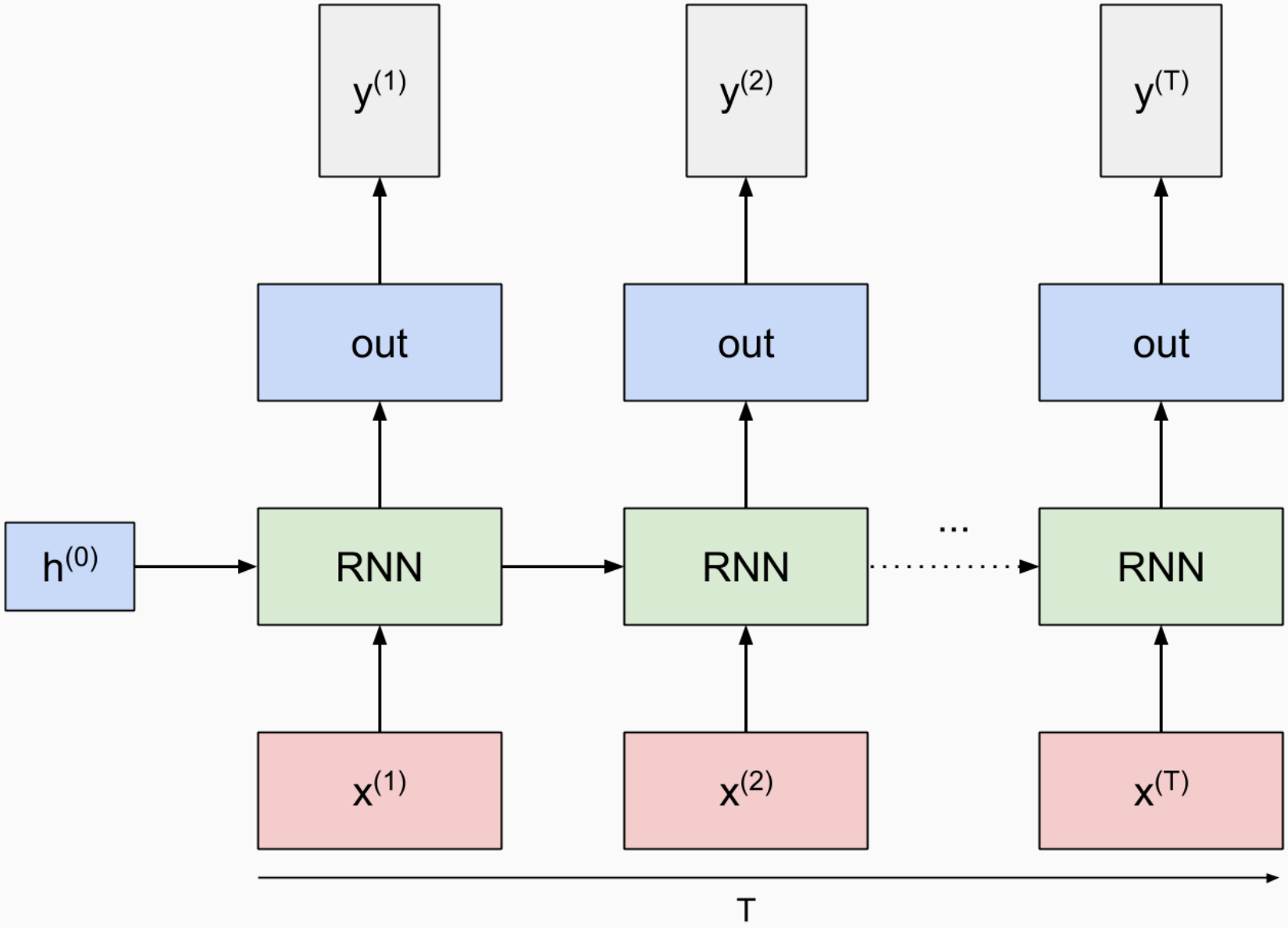
Advanced recurrent models

Martin Tutek, Petra Bevandić, Josip Šarić, Siniša Šegvić

2024.

Recap

Plain recurrent model (RNN denotes recurrent cells)



Basic recurrent cell

Update of the hidden state of the recurrent cell:

$$h^{(t)} = \tanh(\underbrace{W_{hh}h^{(t-1)} + W_{xh}x^{(t)} + b_h}_{a^{(t)}})$$

Output projection

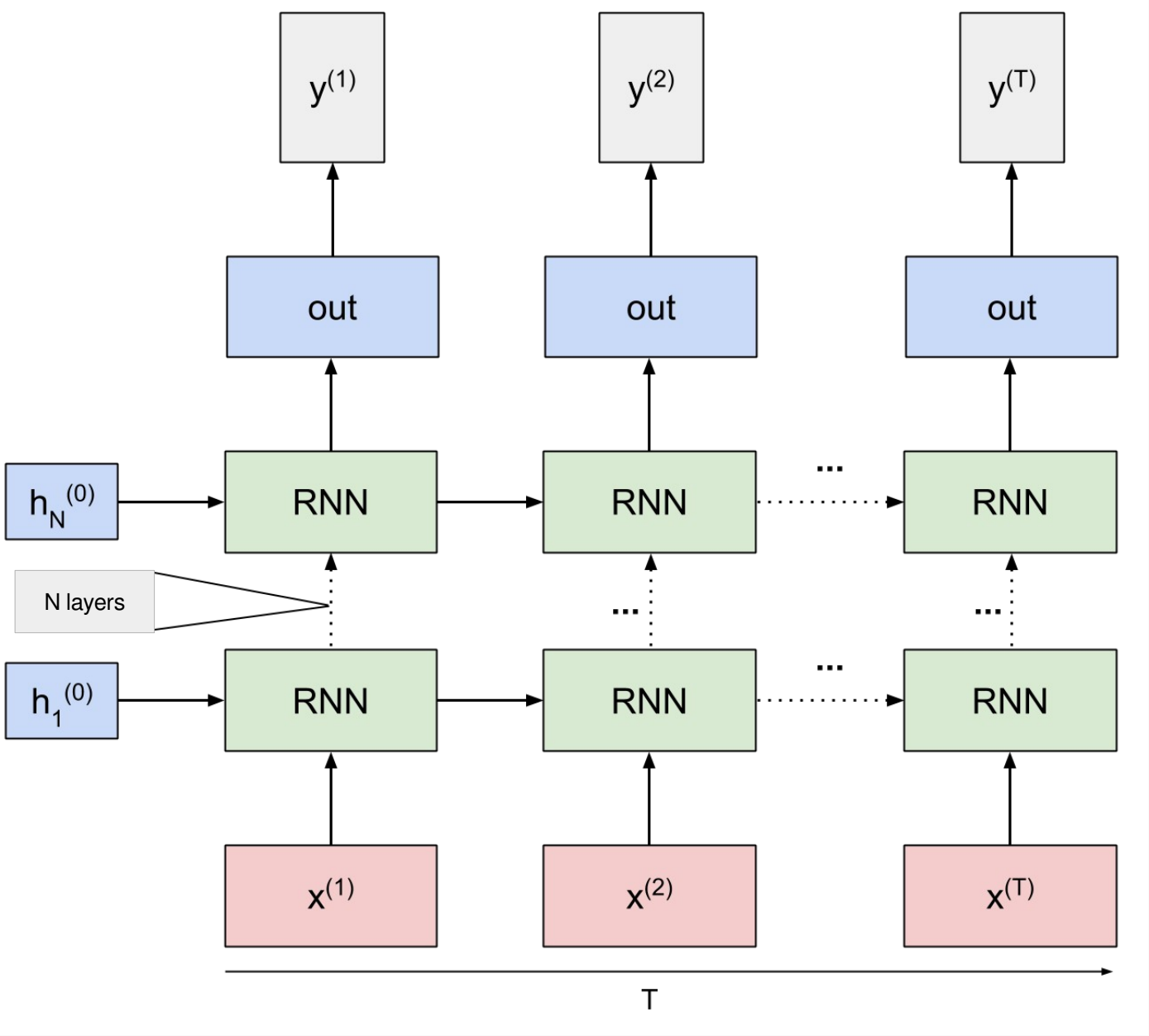
$$o^{(t)} = W_{hy}h^{(t)} + b_o \tag{1}$$

The plain formulation processes the whole sequence by measn of three weight matrices and two bias vectors:

- often insufficient for learning complex dependencies among the sequence elements
- this can be addressed by introducing latent recurrent layers between the input and the predictions!

Deep recurrent models

Deep (multi-layer) recurrent model



Deep recurrent model

Can you observe a problem in the precedent figure?

- $x^{(t)}$ and $h^{(t)}$ may have different dimensions
- dimensionality of $W_{xh} \in \mathbb{R}^{h \times h}$ may differ across layers

Layer $n = 1$:

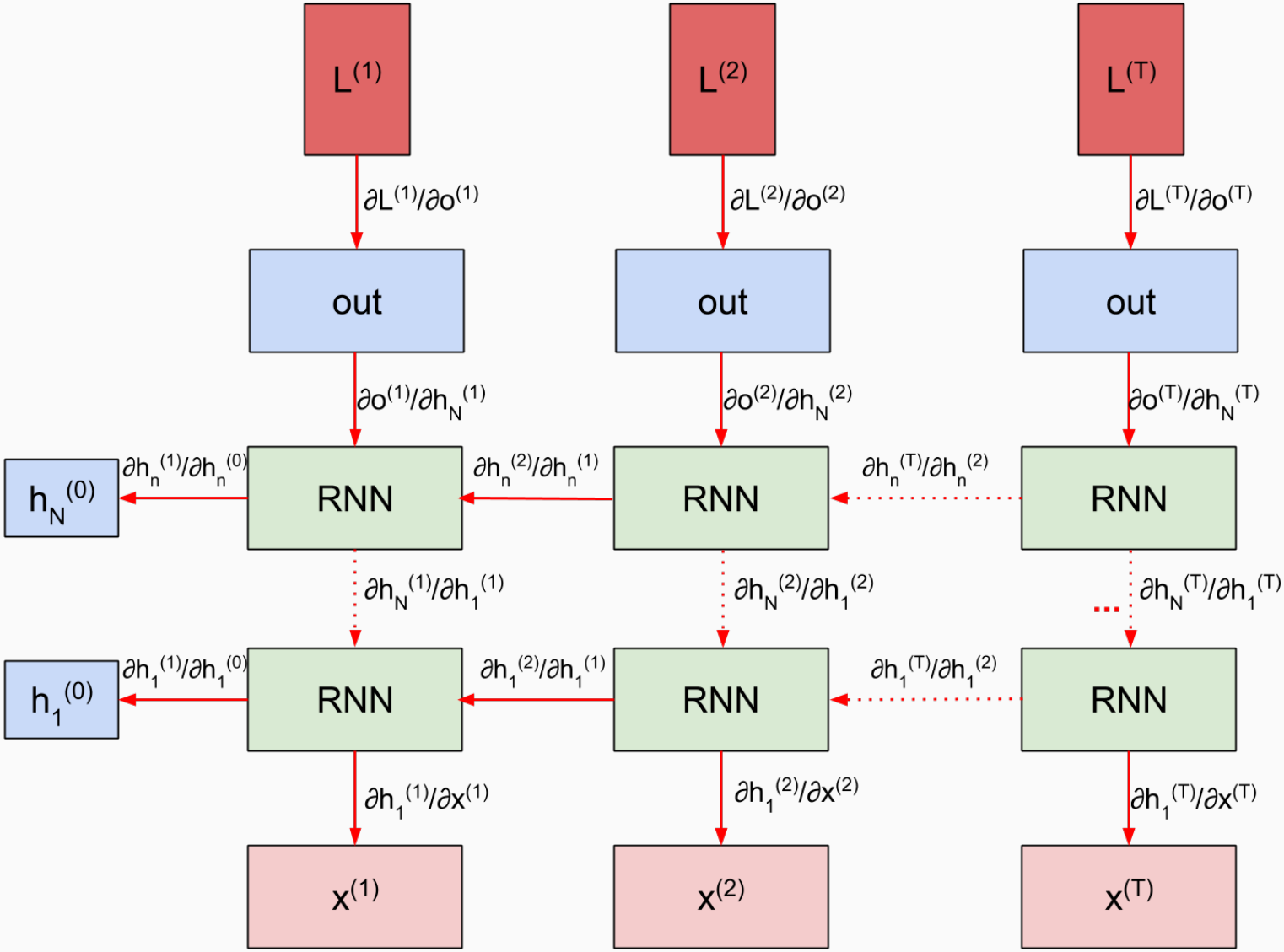
$$h_n^{(t)} = \tanh(\underbrace{W_{nhh} h_n^{(t-1)} + W_{nxh} x^{(t)} + b_{nh}}_{a_n^{(t)}}) \tag{2}$$

Layers $n > 1$:

$$h_n^{(t)} = \tanh(\underbrace{W_{nhh} h_n^{(t-1)} + W_{nxh} h_{n-1}^{(t)} + b_{nh}}_{a_n^{(t)}}) \tag{3}$$

[!!] Recent deep learning frameworks offer recurrent cells that adapt the matrix shapes automatically.

Deep recurrent model: backprop



Deep recurrent model: summary

Recurrent models extend through depth (vertically) and time (horizontally):

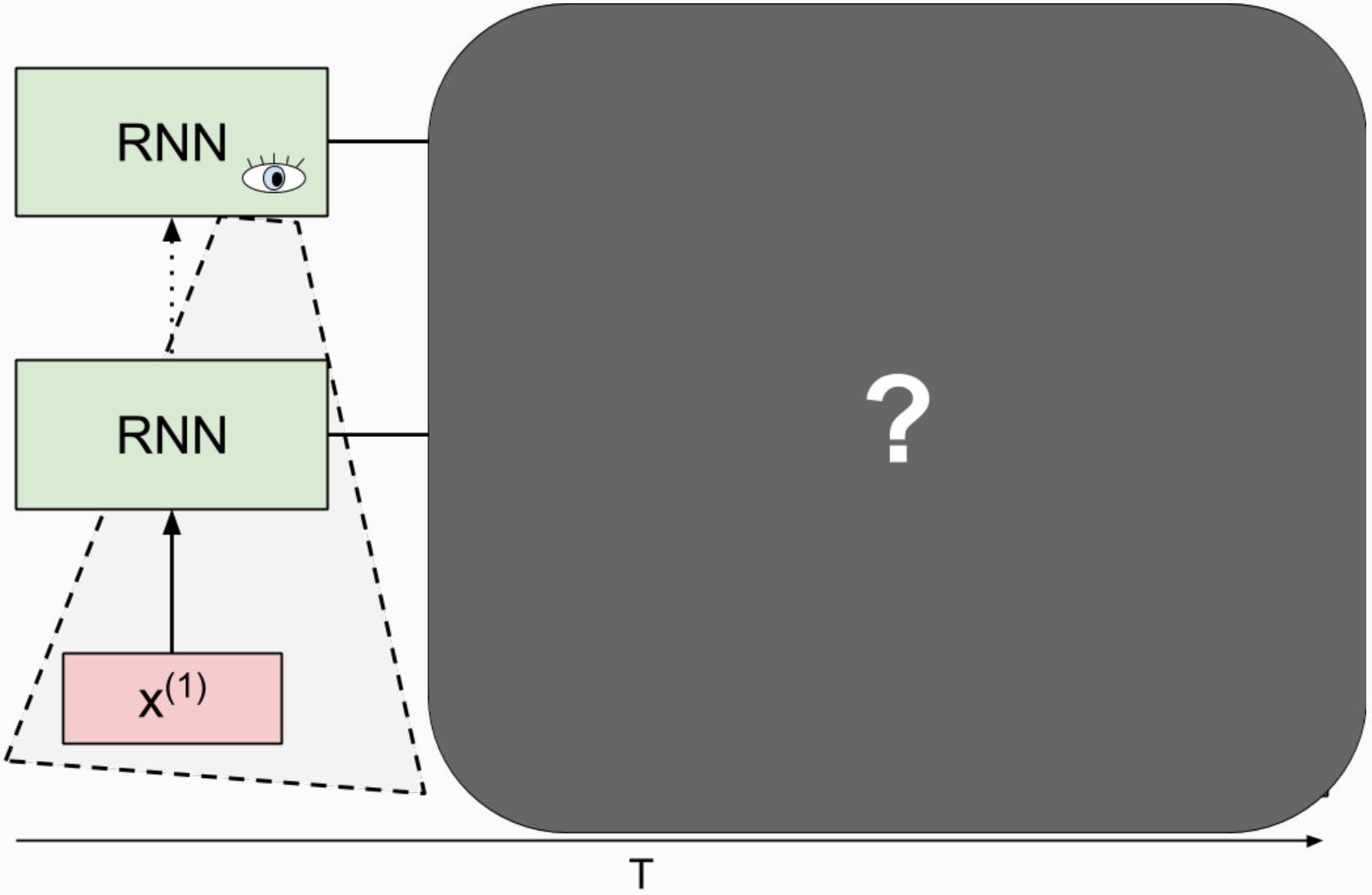
- practical configurations involve 4 to 8 layers depending on the quantity of training data
- more than 8 subsequent recurrent layers do not lead to significant performance improvements (even in the case of advanced cells)

Different layers may have different dimensionalities:

- this does not complicate the implementation
- layers are typically configured through constructor arguments of the chosen recurrent cell.

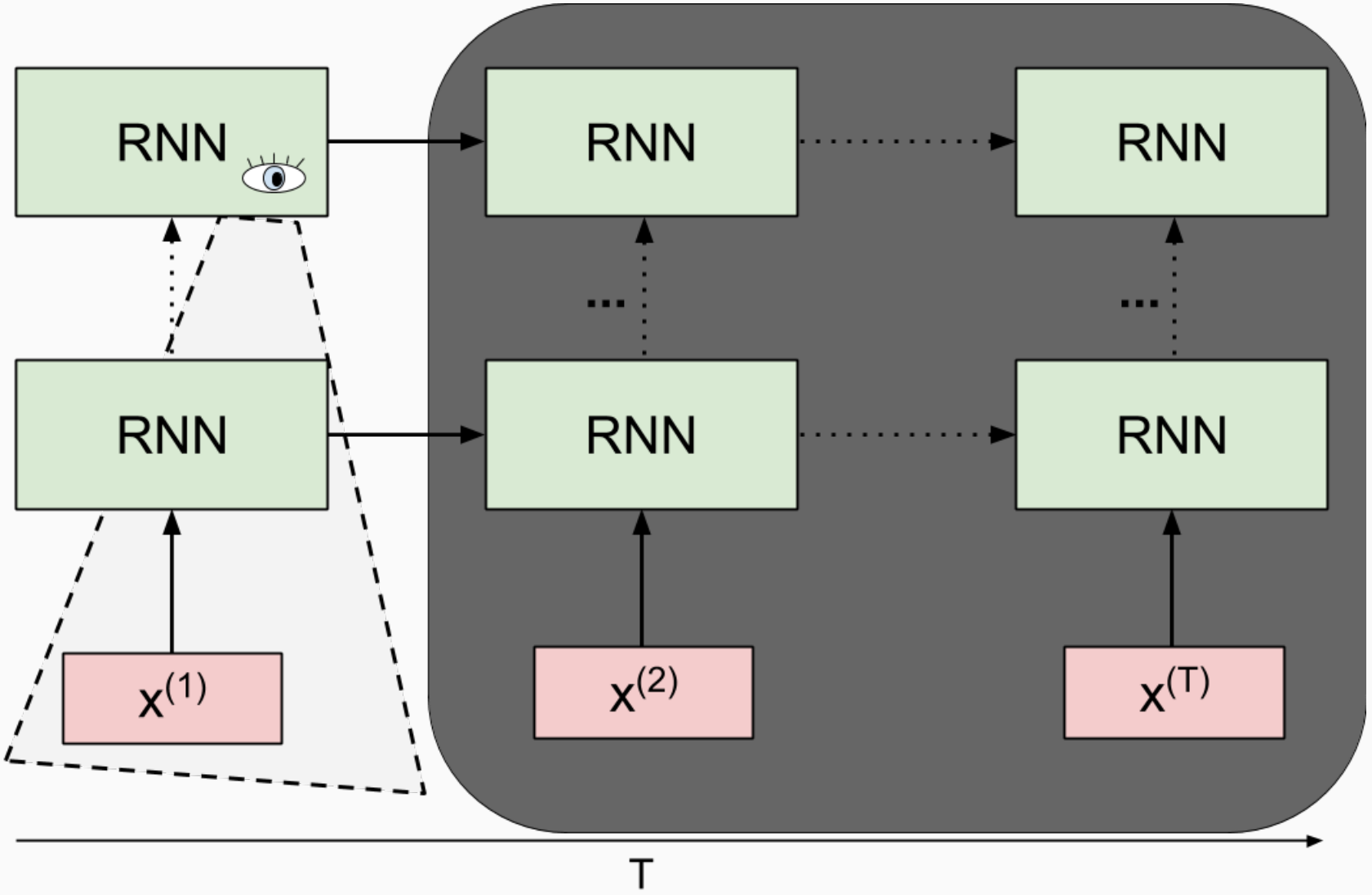
Troubleshooting

What is the **receptive field** of a recurrent cell?



Troubleshooting

What is the **receptive field** of a recurrent cell?



Troubleshooting: receptive field

Each recurrent cell (in any layer, at time t) observes only $x^{(t)} \leq t$:

- prediction at time t is conditioned only by observed inputs!
- if the problem at hand does not imply hiding the *future* context, we would like to allow the model to observe the **whole** sequence prior to making the prediction

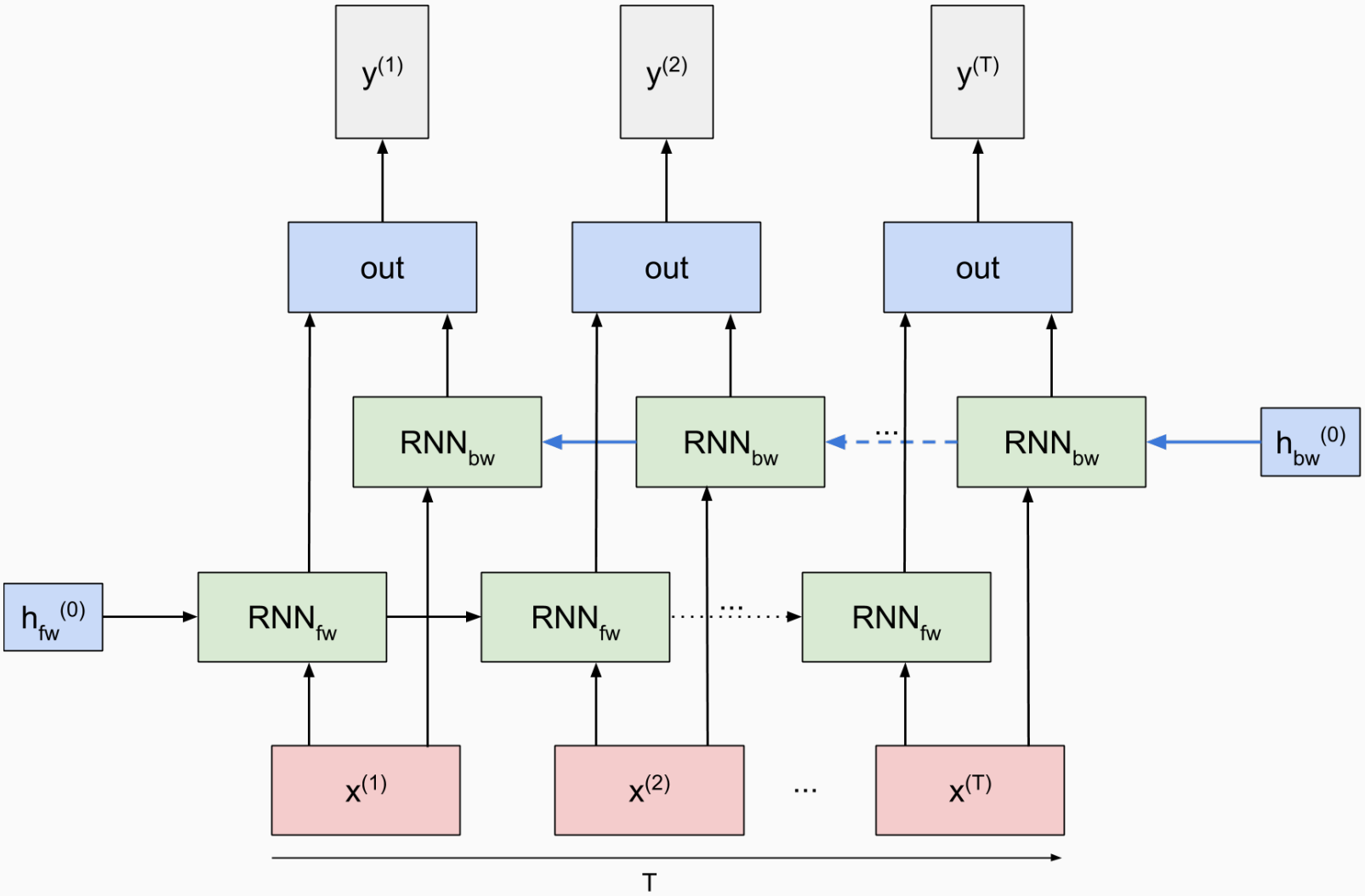
Idea: if the hidden state $h^{(t)}$ of a cell that reads from left to right sees $x^{(t)} \leq t$, then a cell that reads in the opposite direction sees the remaining inputs $x^{(t)} > t$

- together, these two cells observe the whole sequence

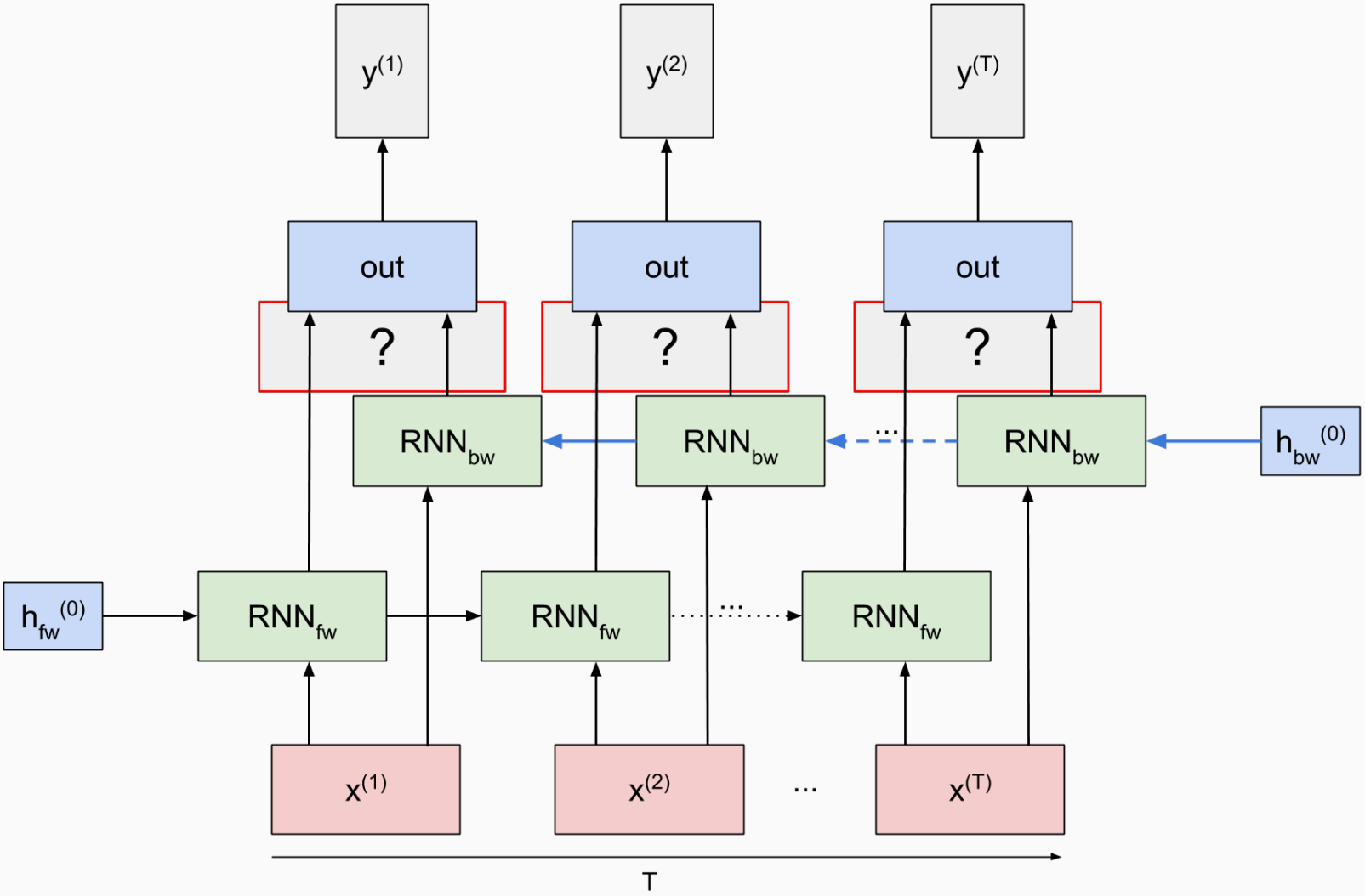
Bidirectional recurrent models

Bidirectional recurrent model

We add an **independent** recurrent model ($\overleftarrow{RNN}$) that operates in the **opposite** direction with respect to the original model ($\overrightarrow{RNN}$)



How to aggregate the hidden states?



Bidirectional recurrent model

Bidirectional recurrent cell (BiRNN) consists of two unidirectional models that operate in opposite directions:

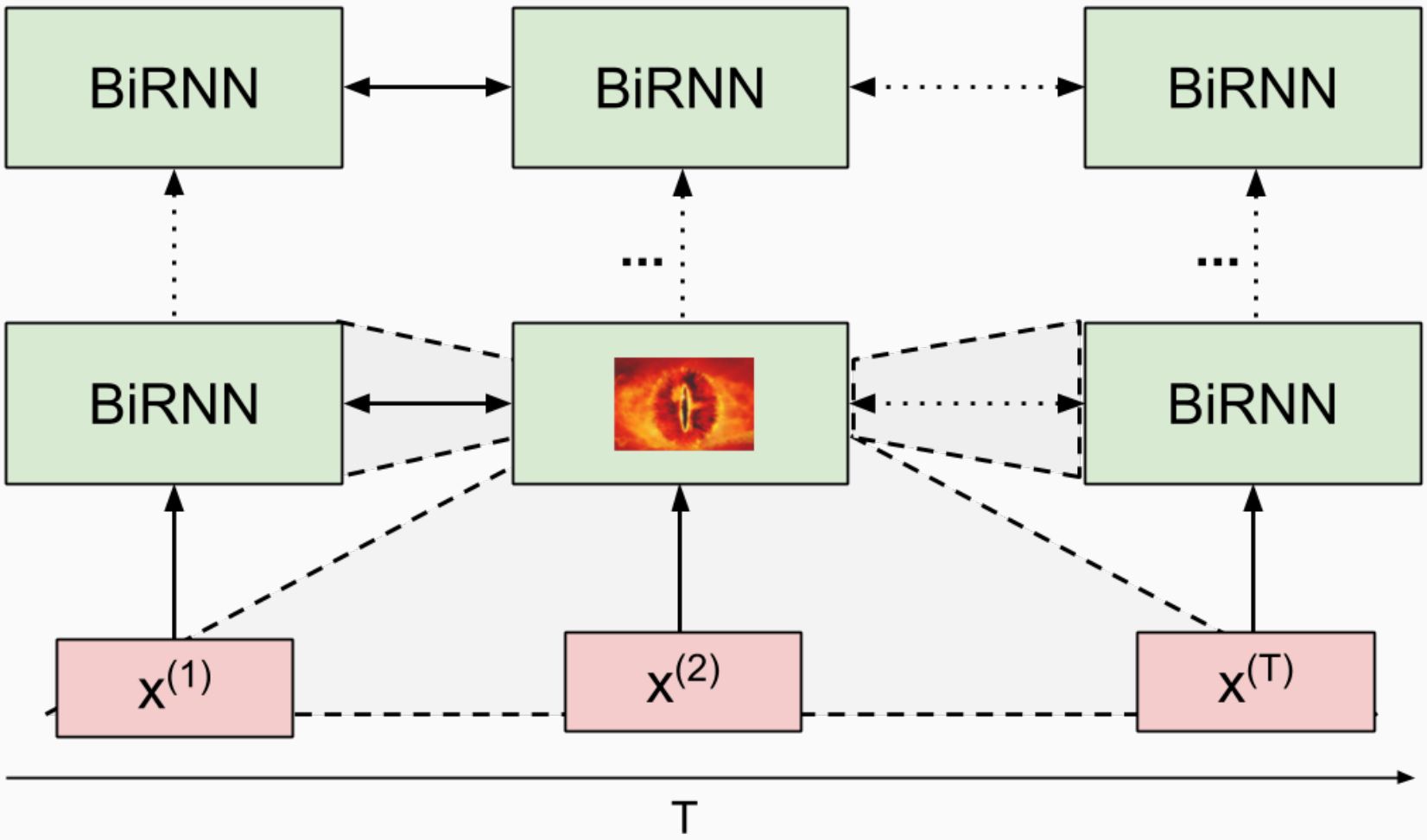
- $\overrightarrow{RNN}$ reads from left to right
- $\overleftarrow{RNN}$ reads from right to left

How to combine the hidden states?

1. concatenation:

$$h^{(t)} = [\overrightarrow{h}^{(t)}, \overleftarrow{h}^{(t)}]$$

   - this doubles the input dimensionality of the next layer
   - default behaviour in existing implementations
2. mean pooling
3. arbitrary (parameterized) function


BiRNN
BiRNN
BiRNN
BiRNN
BiRNN
x(1)
x(2)
x(T)
T

Bidirectional recurrent models: summary

Bidirectional models consist of two recurrent models that iterirate in opposite directions:

- concatenation of the two states allow the next layer to receive the state that depends on all inputs

Concatenation increases the dimensionality of the subsequent layer:

- default behaviour
- alternatives: mean pooling, pooling + projection, ...

We must consider whether the model **is allowed** to access all input data (forecasting vs dense prediction).

Troubleshooting recurrent models

Vanishing and exploding gradients

Recurrent models are susceptible to vanishing and exploding gradients:

- caused by parameter sharing through subsequent operations
- more precisely: repeated multiplication by W_{hh}

Reminder:

$$h^{(t)} = \tanh(W_{hh}h^{(t-1)} + W_{xh}x^{(t)} + b_h)$$

We first consider the *scalar* context ($w_{hh}, w_{xh}, b_h, h, x \in \mathbb{R}$):

$$h^{(t)} = \tanh(\underbrace{w_{hh}h^{(t-1)} + w_{xh}x^{(t)} + b_h}_{a^{(t)}})$$

Vanishing and exploding gradients (scalar context)

Equation of the recurrent cell (reminder):

$$h^{(t)} = \tanh(\underbrace{w_{hh}h^{(t-1)} + w_{xh}x^{(t)} + b_h}_{a^{(t)}})$$

Consider the gradient between subsequent hidden states:

$$\begin{aligned}\frac{\partial h^{(t)}}{\partial h^{(t-1)}} &= \frac{\partial h^{(t)}}{\partial a^{(t)}}\frac{\partial a^{(t)}}{\partial h^{(t-1)}} \\ &= \frac{\partial \tanh(a^{(t)})}{\partial a^{(t)}} w_{hh} \\ &= (1 - \tanh^2(a^{(t)}))w_{hh}\end{aligned}$$

Note that the tanh derivative is limited to unit interval:

$$\tanh(x) \in (-1, 1)$$

$$\frac{\partial \tanh(x)}{\partial x} = (1 - \tanh^2(x)) \in (0, 1)$$

Vanishing and exploding gradients (scalar context, 2)

The gradient between subsequent hidden states (reminder):

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = (1 - \tanh^2(a^{(t)}))w_{hh}$$

Let us apply the following substitution:

$$\gamma_t = \partial \tanh(x)/\partial x\big|_{a^{(t)}} = 1 - \tanh^2(a^{(t)}) < 1$$

Consider the gradient towards the state $h^{(t_0)}$:

$$\begin{aligned}\frac{\partial h^{(T)}}{\partial h^{(t_0)}} &= \prod_{t=T}^{t_0+1} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} = \frac{\partial \tanh(x)}{\partial x}\Big|_{a^{(t)}} w_{hh} \\ &= \prod_{t=T}^{t_0+1} \gamma_t w_{hh} \\ &= (\overline{\gamma} w_{hh})^{T-t_0}\end{aligned}$$

Vanishing and exploding gradients (scalar context, 3)

Gradient towards $h^{(t_0)}$ (reminder):

$$\frac{\partial h^{(T)}}{\partial h^{(t_0)}} = (\overline{\gamma} w_{hh})^{T-t_0}$$

When we have long sequences, $T - t_0 \gg 0$, the gradients may explode, vanish or be stable depending on $\overline{\gamma} w_{hh}$:

$$(\overline{\gamma} w_{hh})^{T-t_0} \to \begin{cases} \infty & \text{if } \overline{\gamma} w_{hh} > 1 \quad \text{(explodes)} \\ 0 & \text{if } \overline{\gamma} w_{hh} < 1 \quad \text{(vanishes)} \\ 1 & \text{if } \overline{\gamma} w_{hh} \approx 1 \quad \text{(stable)} \end{cases}$$

If we assume $\overline{\gamma} = 1$, then the above conditions apply to the **parameter** w_{hh}.

We proceed by repeating the analysis in the vector context.

Vanishing and exploding gradients: spectral norm

We consider the following properties of the spectral norm of a square matrix A:

- norm of the product is less than or equal the product of the norms (this holds for all matrix norms):

$$\|AB\| \leq \|A\| \, \|B\|$$

- the spectral norm is equal to the largest singular value of A
  - or, equivalently, square root of the largest eigenvalue of $A^\top A$
- the spectral norm is the natural norm induced by the L2-norm:

$$\|Ax\| \leq \|A\| \, \|x\|$$

Vanishing and exploding gradients (vector context, 1)

Update of the hidden state of the recurrent cell (reminder):

$$h^{(t)} = \tanh(\underbrace{W_{hh}h^{(t-1)} + W_{xh}x^{(t)} + b_h}_{a^{(t)}})$$

We are looking at the gradient between two subsequent states:

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = \frac{\partial h^{(t)}}{\partial a^{(t)}} W_{hh}$$

We note that the gradient magnitude is bounded:

$$\left\| \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \right\| \leq \left\| \frac{\partial h^{(t)}}{\partial a^{(t)}} \right\| \|W_{hh}\| \leq \gamma_{\max}^{t} \lambda_1$$

- λ_1 ... the greatest singular value of W_{hh}
- $\gamma_{\max}^{t}$ ... the upper bound of $\max(\frac{\partial \tanh(a^{(t)})}{\partial a^{(t)}})$

Vanishing and exploding gradients (vector context, 2)

We extend the last equation over several time-steps:

$$\frac{\partial h^{(T)}}{\partial h^{(t_0)}} \leq (\overline{\gamma_{\max}}\lambda_1)^{T-t_0}$$

For long sequences ($T - t_0 \gg 0$) the gradient may explode, vanish or be stable depending on $\overline{\gamma_{\max}}\lambda_1$:

$$(\overline{\gamma_{\max}}\lambda_1)^{T-t_0} \to \begin{cases} \infty & \text{if } \overline{\gamma_{\max}}\lambda_1 > 1 \quad \text{(explode)} \\ 0 & \text{if } \overline{\gamma_{\max}}\lambda_1 < 1 \quad \text{(vanish)} \\ 1 & \text{if } \overline{\gamma_{\max}}\lambda_1 \approx 1 \quad \text{(stable)} \end{cases}$$

- Matrix W_{hh} must fulfill **strict requirements** in order to ensure smooth optimization
- For a more detailed analysis refer to [pascanu13icml]:
  Razvan Pascanu, Tomás Mikolov, Yoshua Bengio: On the difficulty of training recurrent neural networks. ICML 2013.

Long-term dependencies

Recurrent models consistently underperform on long sequences:

- the problem occurs due to numerical unstability of the gradients due to repeated multiplication with W_{hh}:

  Y Bengio, PY Simard, P Frasconi: Learning long-term dependencies with gradient descent is difficult, IEEE TNN 1994.

Symptoms can be alleviated by:

- ensuring moderate singular values of the recurrent connection:

  M Arjovsky, A Shah, Y Bengio: Unitary Evolution Recurrent Neural Networks. ICML 2016

A solution: express the recurrent connection without multiplication with a learned matrix.

Recurrent cell with long-term memory (LSTM)

Long short-term memory

The **hidden state** $h^{(t)}$ is repurposed: it captures short-term dependencies towards the output and the candidate cell state.

The **cell** state $c^{(t)}$ (new!) holds the long-term information up to now.

The **candidate cell state** $\hat{c}^{(t)}$ (new!) holds a possible contribution to the cell-state $c^{(t)}$ given the current input $x^{(t)}$.

Gates $f^{(t)}$ and $i^{(t)}$ (sigmoid vectors, new!) determine dimensions of $c^{(t)}$:

- that should be forgotten (*forget gate* f)
- that should be updated with respect to $x^{(t)}$ (*input gate* i).

LSTM cell-state update eliminates arbitrary matrix multiplication from the recurrent path:

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$

Long short term memory (2)

LSTM cell-state update (reminder):

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$

The $f^{(t)}$ gate **forgets** some information from the previous state:

$$f^{(t)} = \sigma(W_{fhh} h^{(t-1)} + W_{fxh} x^{(t)} + b_{fh}) = \sigma(a_f^{(t)})$$

The input gate $i^{(t)}$ lets through only a **subset** of the input information:

$$i^{(t)} = \sigma(W_{ihh} h^{(t-1)} + W_{ixh} x^{(t)} + b_{ih}) = \sigma(a_i^{(t)})$$

Each gate has its own set of parameters W_{hh}, W_{xh}, b_h.

Long short term memory (3)

LSTM update involves **Hadamard products** ($\odot$) with sigmoid gates:

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$

Hadamard product ($\odot$) corresponds to element-wise multiplication:

$$a \odot b = \begin{pmatrix} a_0 b_0 \\ \dots \\ a_i b_i \end{pmatrix}$$

The purpose of the two gates is to filter information ($\sigma : \mathbb{R} \to (0, 1)$):

- sigmoid function has a probabilistic interpretation: **the amount information that we wish to keep**.

Limiting $f^{(t)}$ and $i^{(t)}$ between $(0, 1)$ *eliminates* exploding gradients

- this interval is open in theory ($\sigma(x) < 1 \quad \forall x$)...
- ...but closed in practice due to finite precision in exp(-x).

Long short term memory (4)

LSTM cell state update (reminder):

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$

The candidate cell state $\hat{c}$ attends to to the **hidden state** and the current input:

$$\hat{c}^{(t)} = \tanh(W_{chh} h^{(t-1)} + W_{cxh} x^{(t)} + b_{ch}) = \tanh(a_c^{(t)})$$

Note that our notation is a bit different than in the book:

- in the book: $s^{(t)} := c^{(t)}; g^{(t)} := i^{(t)}; q^{(t)} := o^{(t)}$

Instead of the **aesthetic** substitution $\hat{c}^{(t)}$, the book inlines the corresponding expression into the cell-state equation:

$$s^{(t)} = f^{(t)} s^{(t-1)} + g^{(t)} \left(\tanh(W h^{(t-1)} + U x^{(t)} + b_s) \right)$$

Long short term memory (5)

LSTM equations (reminder):

$$\hat{c}^{(t)} = \tanh(W_{chh}h^{(t-1)} + W_{cxh}x^{(t)} + b_{ch}) = \tanh(a_c^{(t)})$$
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$

The hidden state modulates the cell state with the *output gate*:

$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$$

The output gate has an independent set of parameters:

$$o^{(t)} = \sigma(W_{ohh}h^{(t-1)} + W_{oxh}x^{(t)} + b_{oh}) = \sigma(a_o^{(t)})$$

Long short term memory (summary, 1)

LSTM equations encourage a moderate magnitude of the cell state, and enforce that the hidden state is strictly finite:

$$\hat{c}^{(t)} = \tanh(W_{chh}h^{(t-1)} + W_{cxh}x^{(t)} + b_{ch}) = \tanh(a_c^{(t)})$$
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$
$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$$

These properties are expressed by means of the three sigmoid gates that operate upon the hidden state and the current input:

$$o^{(t)} = \sigma(W_{ohh}h^{(t-1)} + W_{oxh}x^{(t)} + b_{oh}) = \sigma(a_o^{(t)})$$
$$f^{(t)} = \sigma(W_{fhh}h^{(t-1)} + W_{fxh}x^{(t)} + b_{fh}) = \sigma(a_f^{(t)})$$
$$i^{(t)} = \sigma(W_{ihh}h^{(t-1)} + W_{ixh}x^{(t)} + b_{ih}) = \sigma(a_i^{(t)})$$

Long short term memory (summary, 2)

LSTM separates the long-term memory from the output delivery:

- this unburdens $h^{(t)}$, which has to do both tasks in the regular recurrent cell

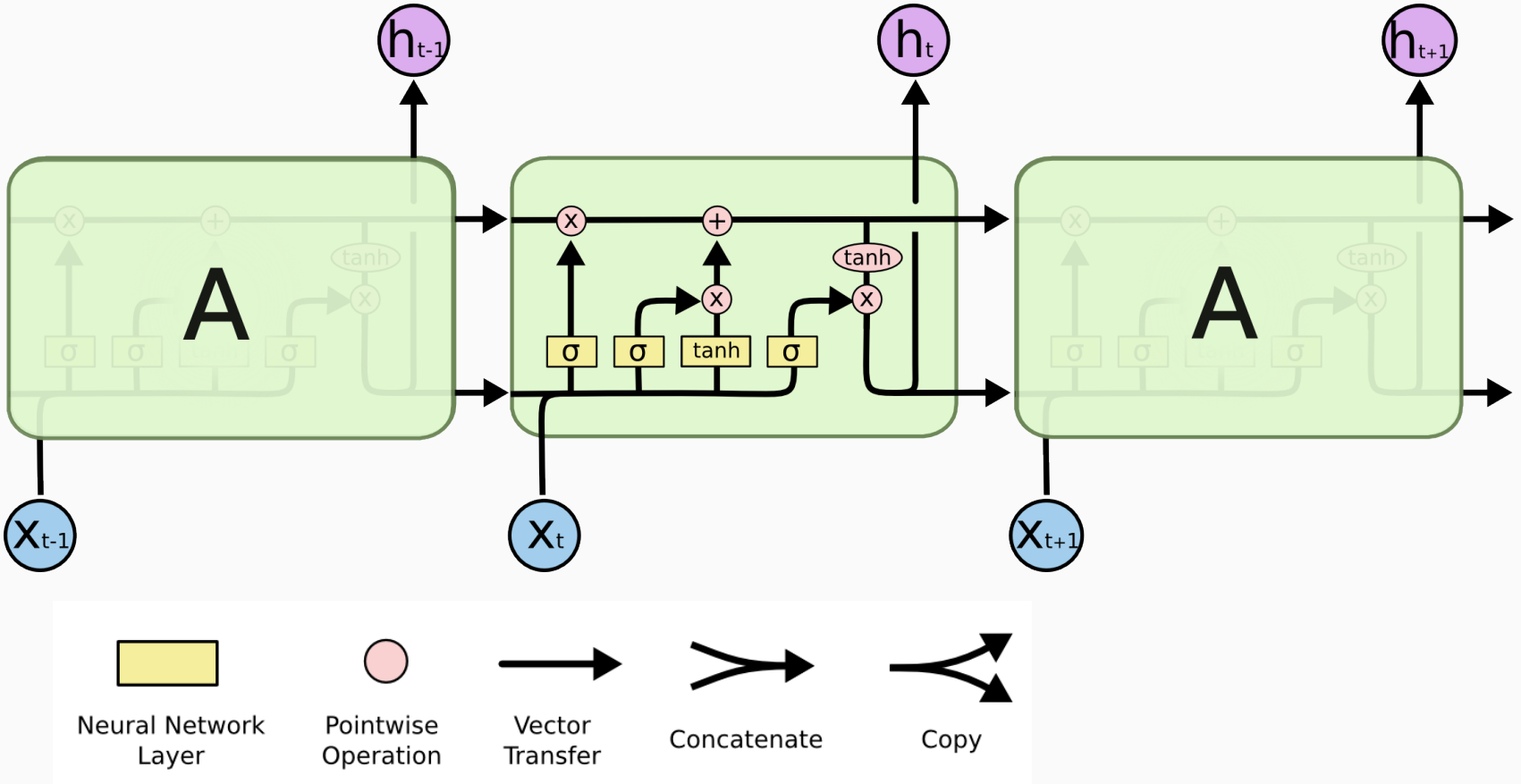
The cell state $c^{(t)}$ keeps count of the long-term information

- we ensure it is **hard** to change its value)

Cell-space evolution involves **four** times more parameters than in a regular RNN cell.

Visualizing the LSTM

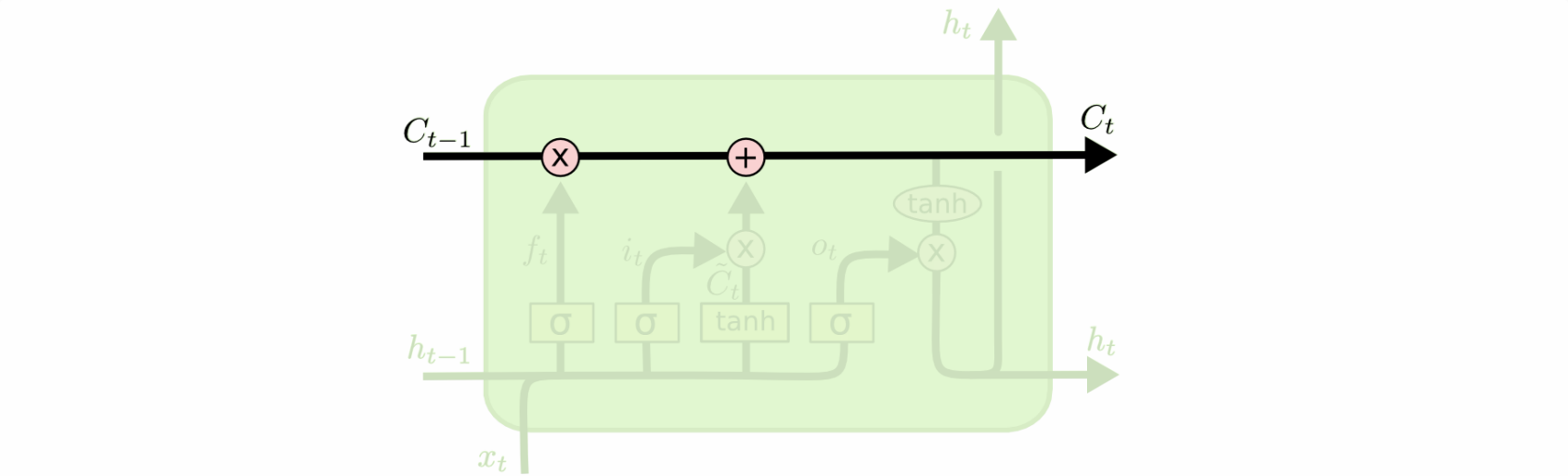
We reproduce several figures from the blog by Cristopher Olah [link]



Question: What is the order of gate names in the figure?

Visualizing the LSTM - cell state

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$

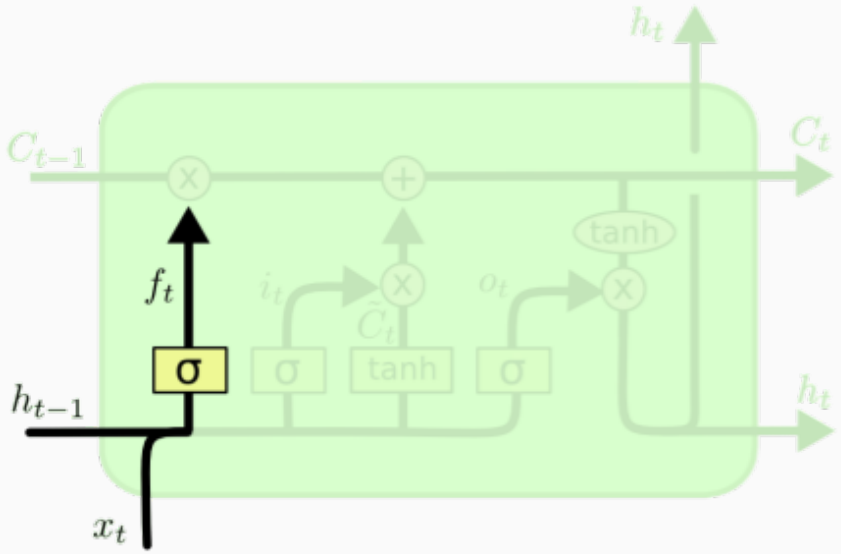


Cell state is modified with one multiplication and one summation - information flow is simple

- Cell state is **hard** to modify: the two gates have to allow the change to *"pass through"*

Visualizing the LSTM - forget gate

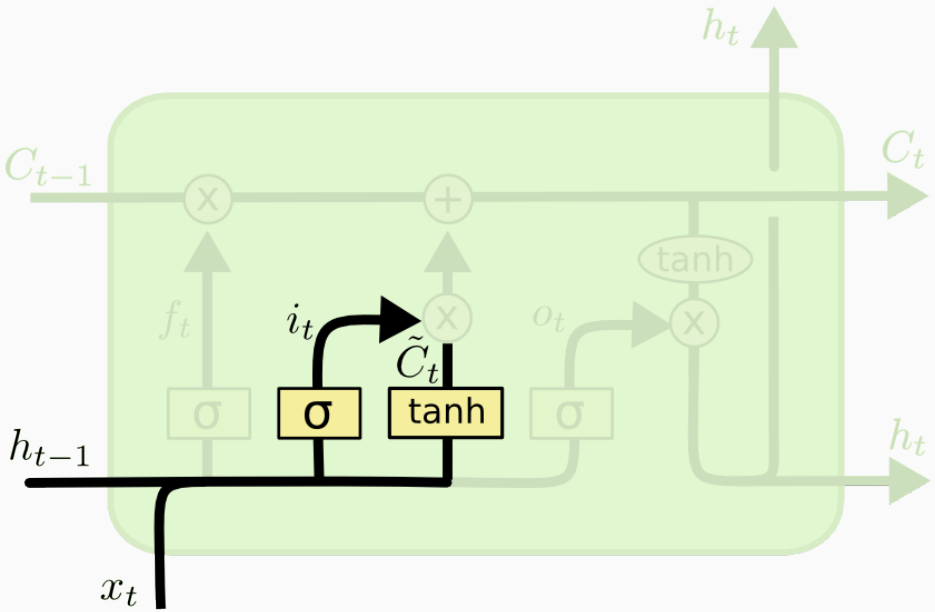
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$



$$f^{(t)} = \sigma(W_{fhh}h^{(t-1)} + W_{fxh}x^{(t)} + b_{fh})$$

Visualizing the LSTM - input gate

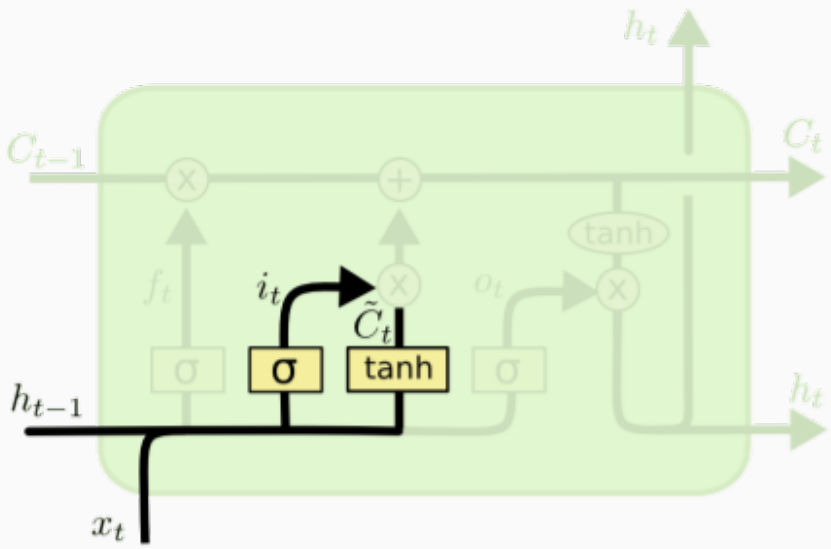
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$



$$i^{(t)} = \sigma(W_{ihh} h^{(t-1)} + W_{ixh} x^{(t)} + b_{ih})$$

$$\hat{c}^{(t)} = \tanh(W_{chh} h^{(t-1)} + W_{cxh} x^{(t)} + b_{ch})$$

Visualizing the LSTM - input gate



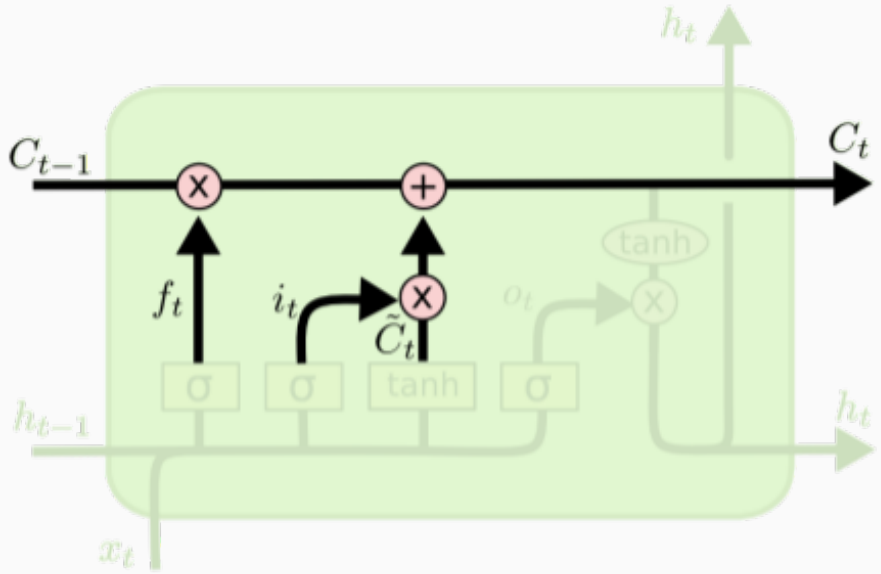
$$\hat{c}^{(t)} = \tanh(W_{chh}h^{(t-1)} + W_{cxh}x^{(t)} + b_{ch})$$

According to the literature, either a sigmoid or a hyperbolic tangent can be used to activate $\hat{c}^{(t)}$

- PyTorch and Tensorflow use $\tanh$

Visualizing the LSTM - updating the cell state

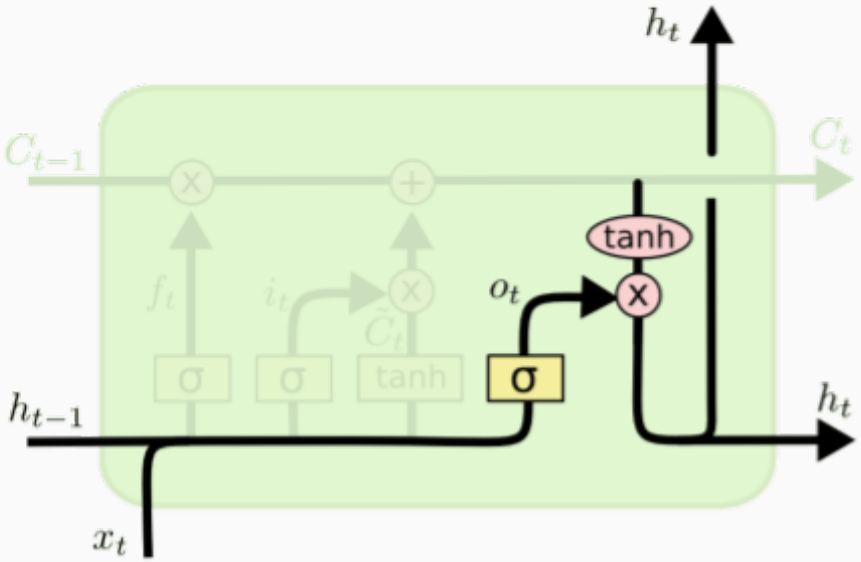
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$



Visualizing the LSTM - output gate

$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$$

$$o^{(t)} = \sigma(W_{ohh} h^{(t-1)} + W_{oxh} x^{(t)} + b_{oh})$$



LSTM as a better RNN

Basic RNNs are difficult to optimize:

- numerical instability due to repeated multiplication with W_{hh}
- the cell state has to memorize information and drive the output
- these models lose performance as sample length increases

Long short term memory cell (LSTM) alleviates these problems:

$$\hat{c}^{(t)} = \tanh(W_{chh} h^{(t-1)} + W_{cxh} x^{(t)} + b_{ch})$$
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$
$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$$

- LSTM removes matrix multiplication from the recurrent equations
- the three sigmoid gates ($f^{(t)}, i^{(t)}, o^{(t)}$) filter information and encourage moderate magnitude of the cell state
- it reduces the load on the cell state by decoupling **responsibility** for short-term action and long-term memorization.

LSTM: backprop

LSTM cell state update (reminder):

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$

Consider the gradient throught the recurrent equation:

$$\frac{\partial c^{(t)}}{\partial c^{(t-1)}} = f^{(t)} = \sigma(a_f^{(t)}) \in (0, 1)$$

The chain rule gives us:

$$\frac{\partial c^{(T)}}{\partial c^{(t_0)}} = \prod_{t=t_0}^{T} f^{(t)} \leq 1$$

It appears that the **exploding** gradients are eliminated!

- *Is that really so?*
- LSTM cell has *dual* hidden state ($c^{(t)}$, $h^{(t)}$)

LSTM: backprop (2)

LSTM hidden state update (reminder):

$$h^{(t)} = o^{(t)} \odot \tanh(c^{(t)})$$
$$o^{(t)} = \sigma(W_{ohh} h^{(t-1)} + W_{oxh} x^{(t)} + b_{oh})$$

We observe a similar pattern as in the basic RNN cell:

$$h^{(t)} = \sigma(W_{ohh} h^{(t-1)} + W_{oxh} x^{(t)} + b_{oh}) \odot \tanh(c^{(t)})$$
$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = \frac{\partial h^{(t)}}{\partial a_o^{(t)}} \frac{\partial a_o^{(t)}}{\partial h^{(t-1)}} = \frac{\partial h^{(t)}}{\partial a_o^{(t)}} W_{ohh} = \dots$$

Numerical overflow and underflow are therefore **still possible** when performing the backward pass through $h^{(t)}$:

- they are, however, relatively rare in practice

LSTM variants - peepholes

Include the cell state $c^{(t-1)}$ while calculating the gate value:

$$f^{(t)} = \sigma(\underbrace{W_{fch}c^{(t-1)} + W_{fhh}h^{(t-1)} + W_{fxh}x^{(t)} + b_{fh}}_{a_f^{*(t)}})$$

This idea can be applied to all gates!

Strength: information about the current cell state may **help** [gers2000recurrent].

Weakness: it increases the number of parameters.

Weakness: it introduces a second path for the exploding gradients to appear.

LSTM variants - fused gates

LSTM cell state update (reminder):

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \hat{c}^{(t)}$$

Idea: If some information is forgotten, it should be replaced

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + (1 - f^{(t)}) \odot \hat{c}^{(t)}$$

Strength: 25% less parameters.

Weakness: it performs worse than the standard LSTM

- more parameters help.

Weakness: it looses expressiveness

- $c^{(t)}$ is unable to accumulate evidence.

Gated recurrent unit

GRU models involve the *update gate* $u^{(t)}$ and the *reset gate* $r^{(t)}$:

$$u^{(t)} = \sigma\left(W_{uhh}h^{(t-1)} + W_{uxh}x^{(t)} + b_{uh}\right)$$
$$r^{(t)} = \sigma\left(W_{rhh}h^{(t-1)} + W_{rxh}x^{(t)} + b_{rh}\right)$$

The state update involves the candidate state $\hat{h}^{(t)}$:

$$\hat{h}^{(t)} = \sigma\left(W_{hh}\left(r^{(t)} \odot h^{(t-1)}\right) + W_{xh}x^{(t)} + b_h\right)$$
$$h^{(t)} = u^{(t)} \odot h^{(t-1)} + (1 - u^{(t)}) \odot \hat{h}^{(t)}$$

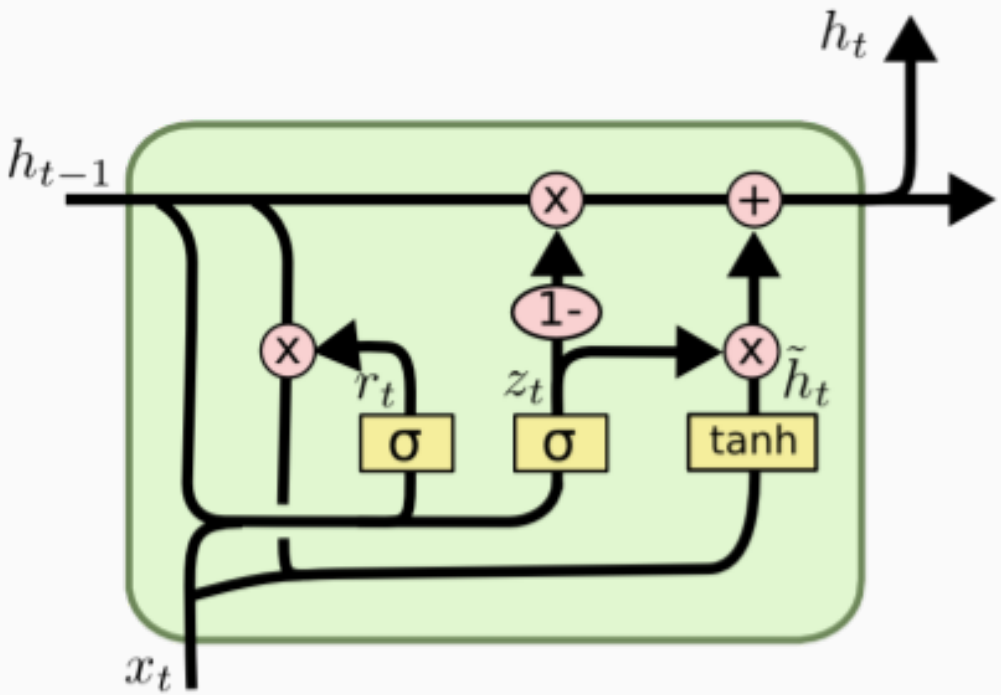
Properties of GRU in comparison with LSTM and RNN:

- single recurrent state $h^{(t)}$ with multiple responsibilities (as in RNN)
- the old state and the candidate state are blended with a single sigmoid gate (as in LSTM with fused gates)

Gated recurrent unit (2)

Gated recurrent unit, a simpler LSTM variant

- works great even though it does not detach the long-term context from driving the output
- this suggests that some intutitions about LSTM may be incomplete.



Recurrent variations: summary

Exploding and vanishing gradients may appear in LSTM-s as well

- however, they appear less frequent in practice

Success of LSTM led to development of other gated recurrent cells.

1. LSTM with peepholes:
   - they return the state information into the update equation
   - it makes sense since LSTMs do not completely circumvent exploding and vanishing gradients
2. LSTM with gate fusion
   - fused forget and input gates for improved efficiency and less parameters
3. Gated recurrent unit (GRU)
   - fused gates and changed gate semantics
   - similar performance to LSTMs in spite of coupled state vector

Analysis: Sequence-to-sequence

Machine translation problem

We wish to generate an output sequence of words for a given input sequence of words:

- the model targets output sequences of **unknown length**.
- at each output position the model outputs a categoric distribution over the output vocabulary.

Datasets: WMT, IWSLT (regularly updated):

- `https://www.statmt.org/wmt15/translation-task.html`
- `https://sites.google.com/site/iwsltevaluation2015/mt-track`

Example of input-output pair in the WMT-14 en-de dataset:

```
Parliament Does Not Support Amendment Freeing Tymoshenko
Keine befreiende Novelle für Tymoshenko durch das Parlament
```

Machine translation problem: preliminaries

Target variables:

- words in the target language: $\{keine, befreiende, Novelle, \ldots\}$
- choose the size of the target vocabulary.
- convert target variables into indices: $\{0, \ldots, V_{out}\}$

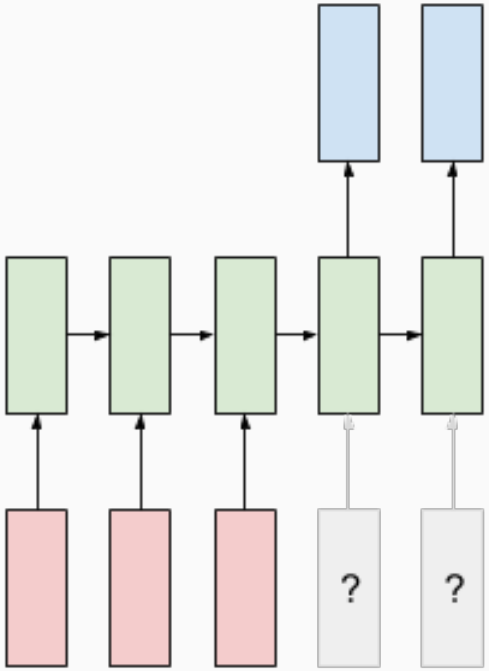
Input variables:

- we must fix the input vocabulary
- we map input words to rows of the embedding matrix.

This setup is similar to part-of-speech recognition, but:

1. we can start generation only after seeing the whole input
2. we do not know the number of output tokens
3. it is not clear which inputs determine the current output

Sequence-to-sequence



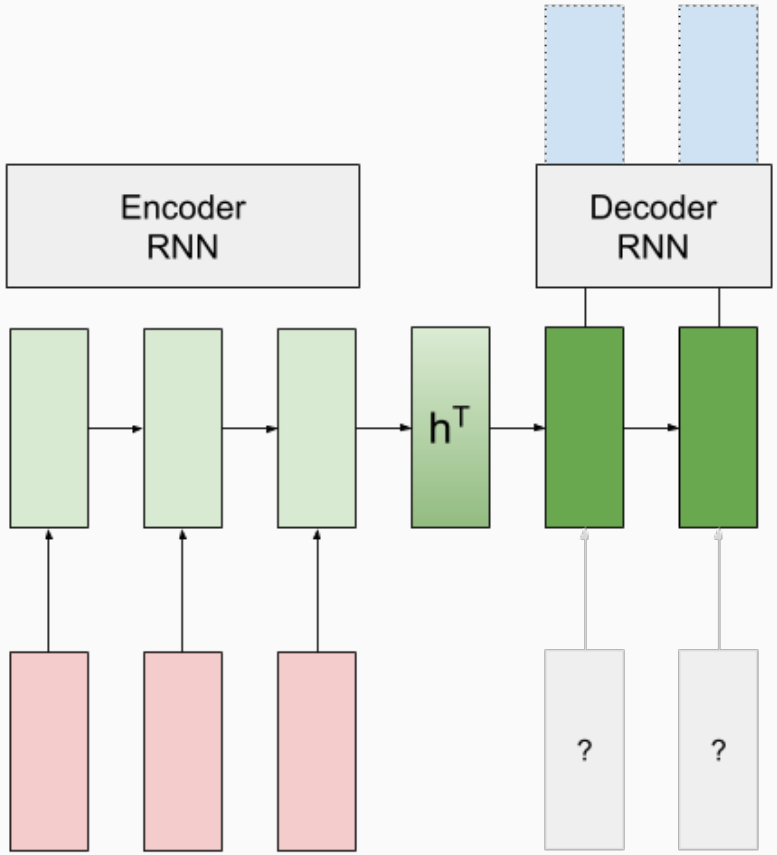
We start by formalizing the sequence-to-sequence problem (above).

Subsequently, we will present some concrete solutions.

Sequence-to-sequence: formalization

We envision a solution with two modules:

1. **encoder** ("reader"): reads the input sequence and builds the best possible hidden representation
2. **decoder** ("writer") generates the translation by consulting the encoded representation.



Sequence-to-sequence: formalization (2)

The last state of the encoder determines the first state of the decoder:

$$h_{dec}^{(0)} = f(h_{enc}^{(T)})$$

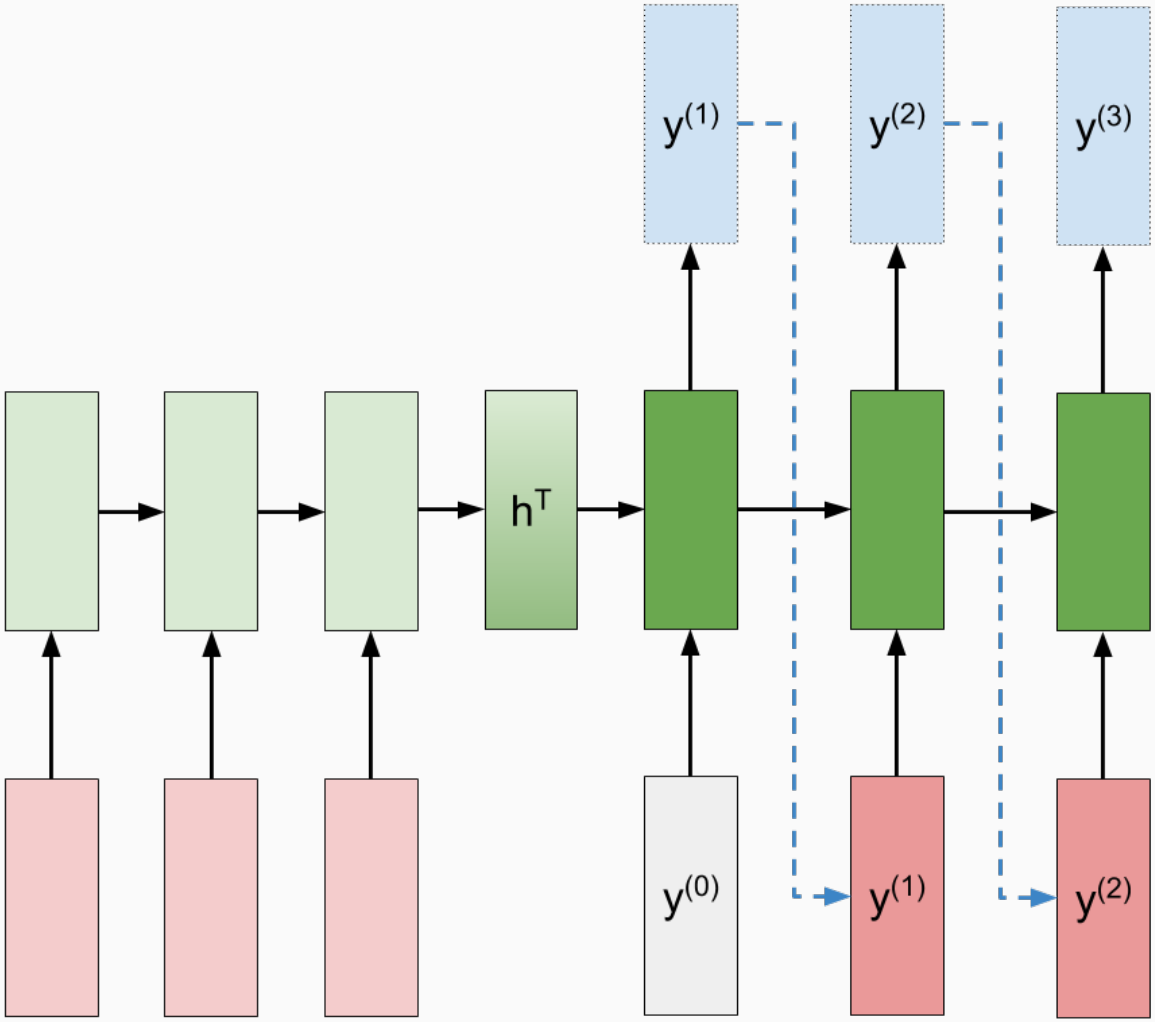
- in practice, we usually have $h_{dec}^{(0)} = h_{enc}^{(T)}$
- still, f can be any parameterized function
- this becomes necessary when $\dim(h_{dec}) \neq \dim(h_{enc})$

The encoder **does not receive** the loss directly

- the loss propagates through the entire **decoder**.

Problem: what are the inputs to the decoder module?

Sequence-to-sequence: decoder inputs



We feed the most recent generated output to each cell of the decoder.

Sequence-to-sequence: generating the output

The decoder input at timestep $t > 0$ contains the most likely decoder output from the previous timestep

- what about the timestep $t = 0$?
- the first input to the decoder is *a special symbol* that represents the start of sequence (`<sos>`).

How do we know that the output sequence is completed?

- the model signals the end of translation by outputting a *special symbol* that represents the end of sequence (`<eos>`)
- the token `<eos>` is appended at **the end** of each target sequence
- we stop the output generation when we get `<eos>` or when the generated sequence exceeds the maximum length.

Sequence-to-sequence: generating the output (2)

Sequence-to-sequence task is very hard to learn, especially in the early stages of optimization.

Hence, we often train with **teacher forcing** where the decoder inputs are obtained stochastically as:

- **groundtruth tokens** of the previous step in p training samples
- **generated outputs** of the prevous step in $1 - p$ training samples
- $p \in [0, 1]$ is a hyper parameter that starts with $p = 1$ and may decrease as the training proceeds

Sequence-to-sequence: inference

Denote the translation as a vector of word indices $(y^{(t)}, t = 1, 2, ...T(x))$.

From the viewpoint of training, the optimal translation should maximize the likelihood:

$$(y^{(t)}, t = 1, 2, ...T(x)) = \arg\max \sum_t \log P_t(Y = y^{(t)}|x)$$

Other inference criteria include linguistic tools (METEOR), evaluating the density of the translation (G-Eval) etc.

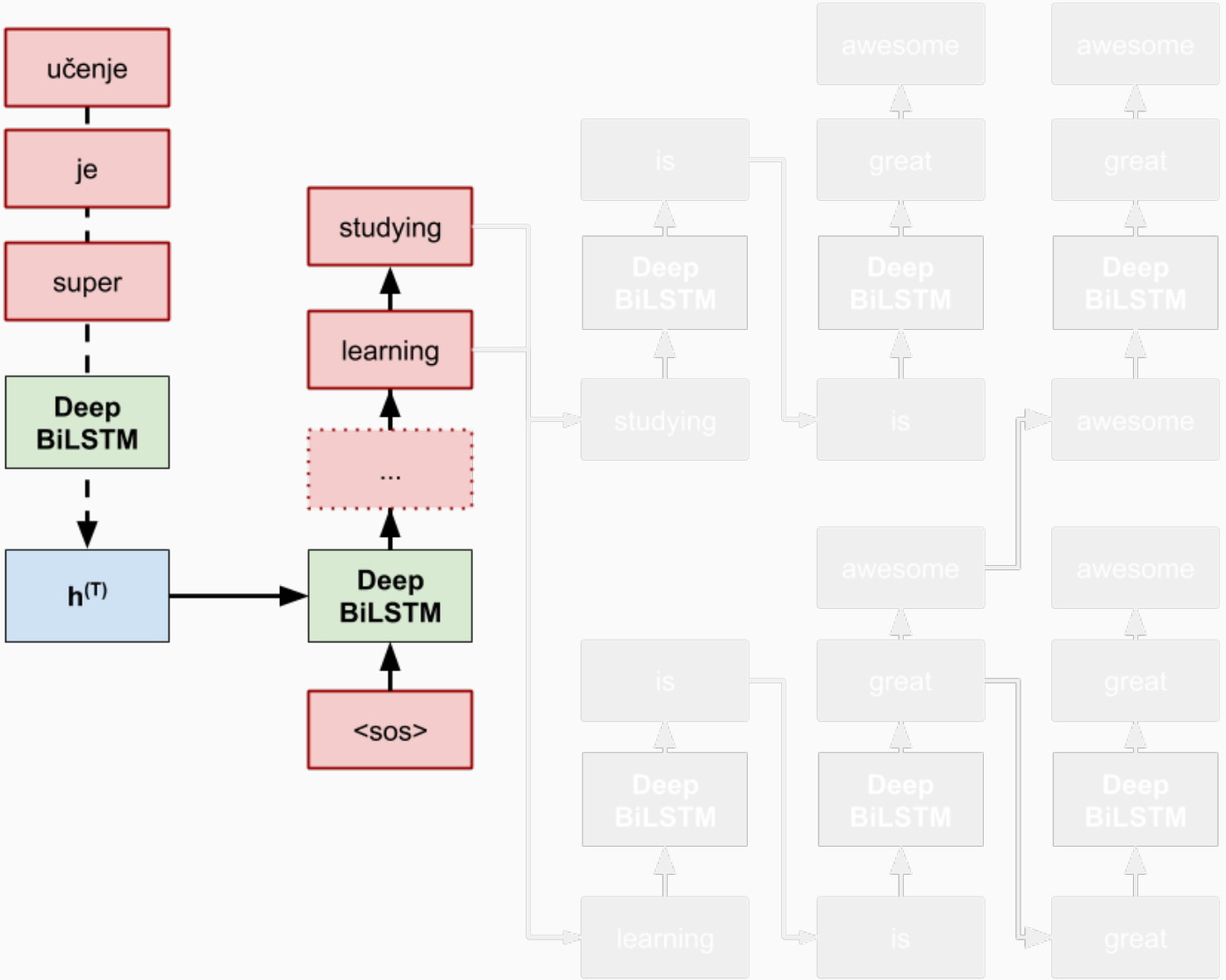
Optimal inference may involve exponential complexity.

Sequence-to-sequence: inference (2)

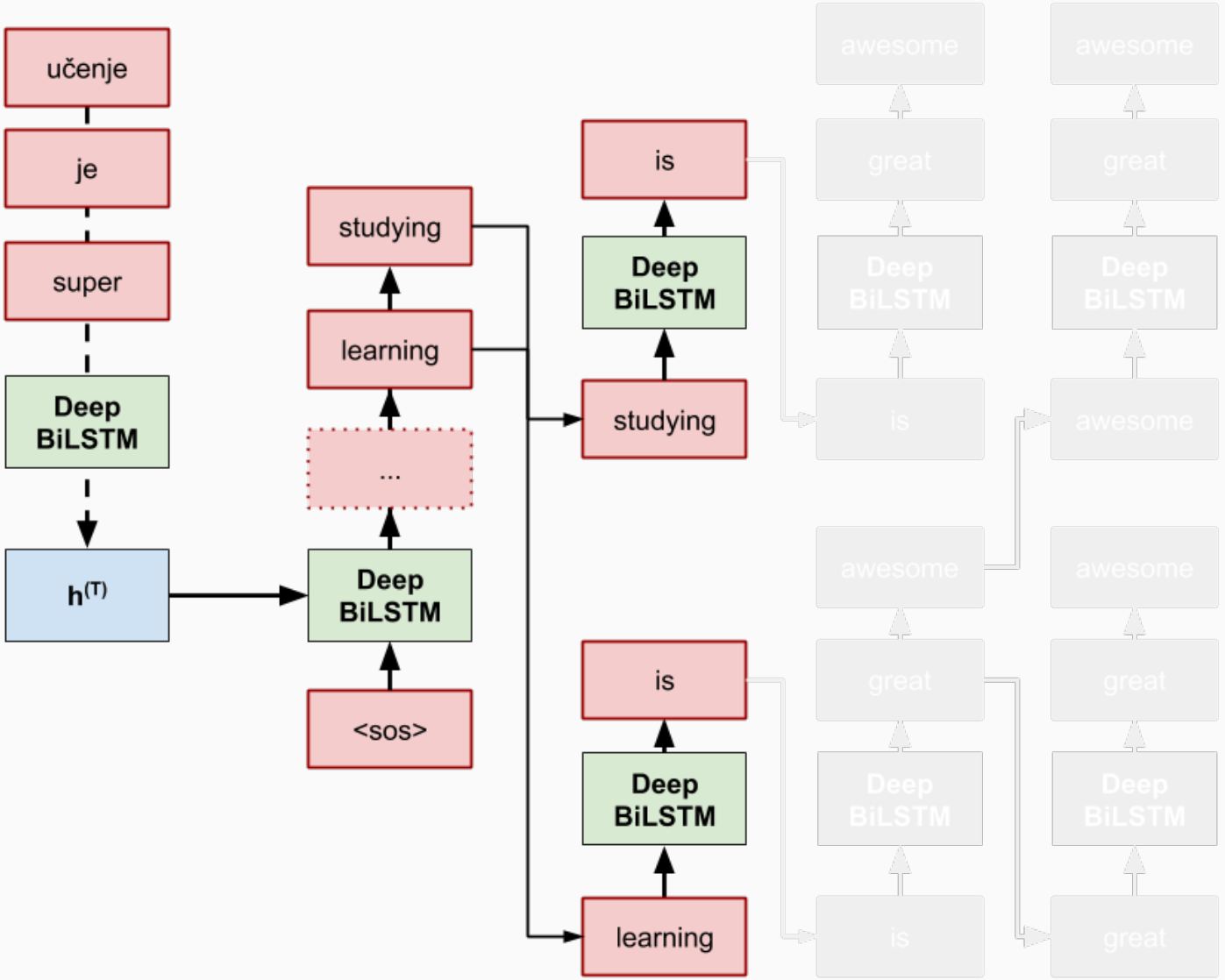
Target sequence generation approaches during inference:

1. greedy approach: take the most probable word in each step:
   - as in other greedy approaches this may be suboptimal
   - it is possible that the best translation does not contain the most probable word in each step
   - moreover, this approach is not good for sampling as it generates **deterministic sequences**
2. random sampling according to probability (roulette wheel selection):
   - this injects randomness into sequence generation and **encourages variability** of the output
   - it is not clear wheather we really desire to have **more than zero** probability for choosing a bad word
3. focused sampling by beam search:
   - consider k currently best scoring outputs in each set
   - hyper-parameter k denotes the beam width

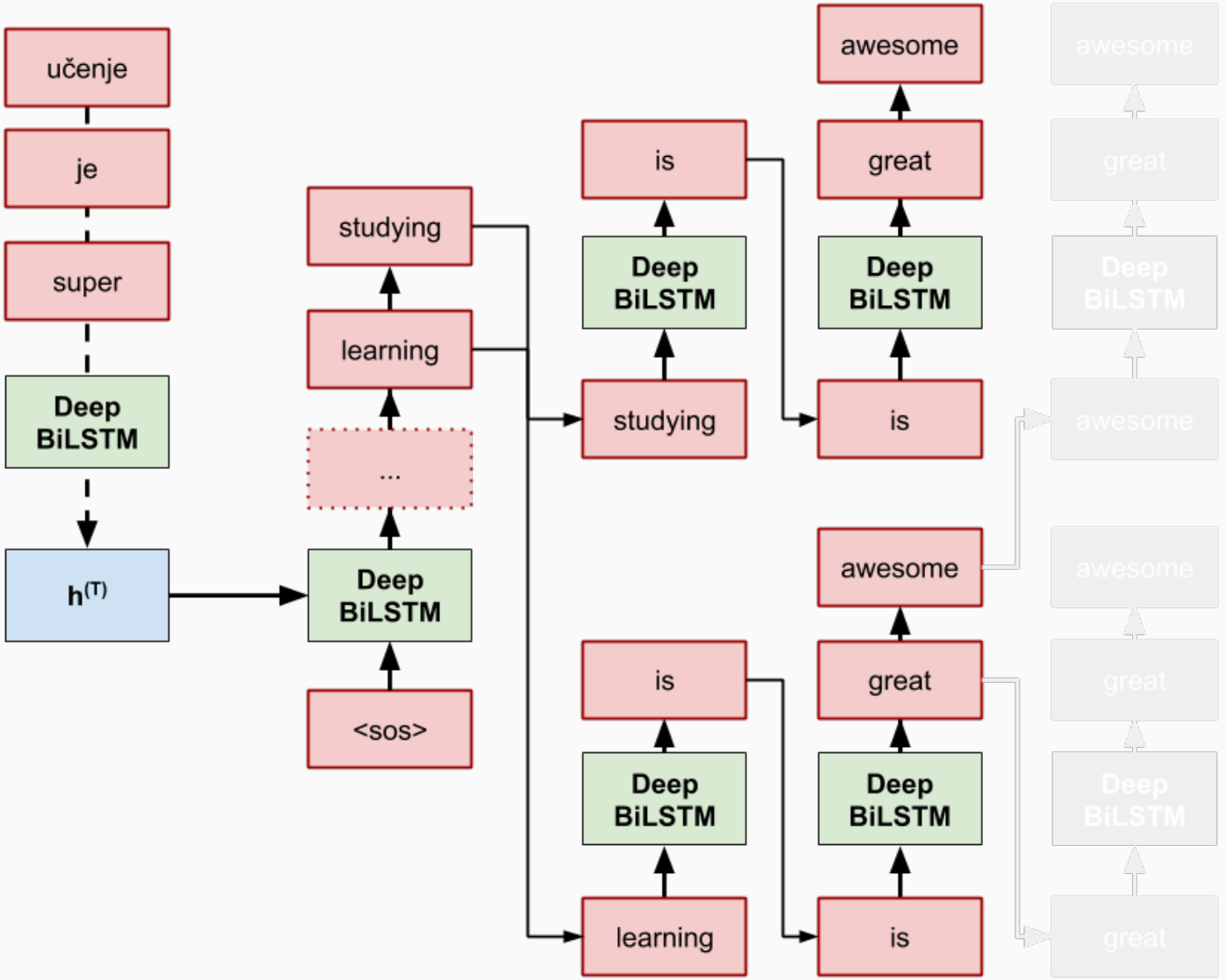
Sequence-to-sequence: beam search



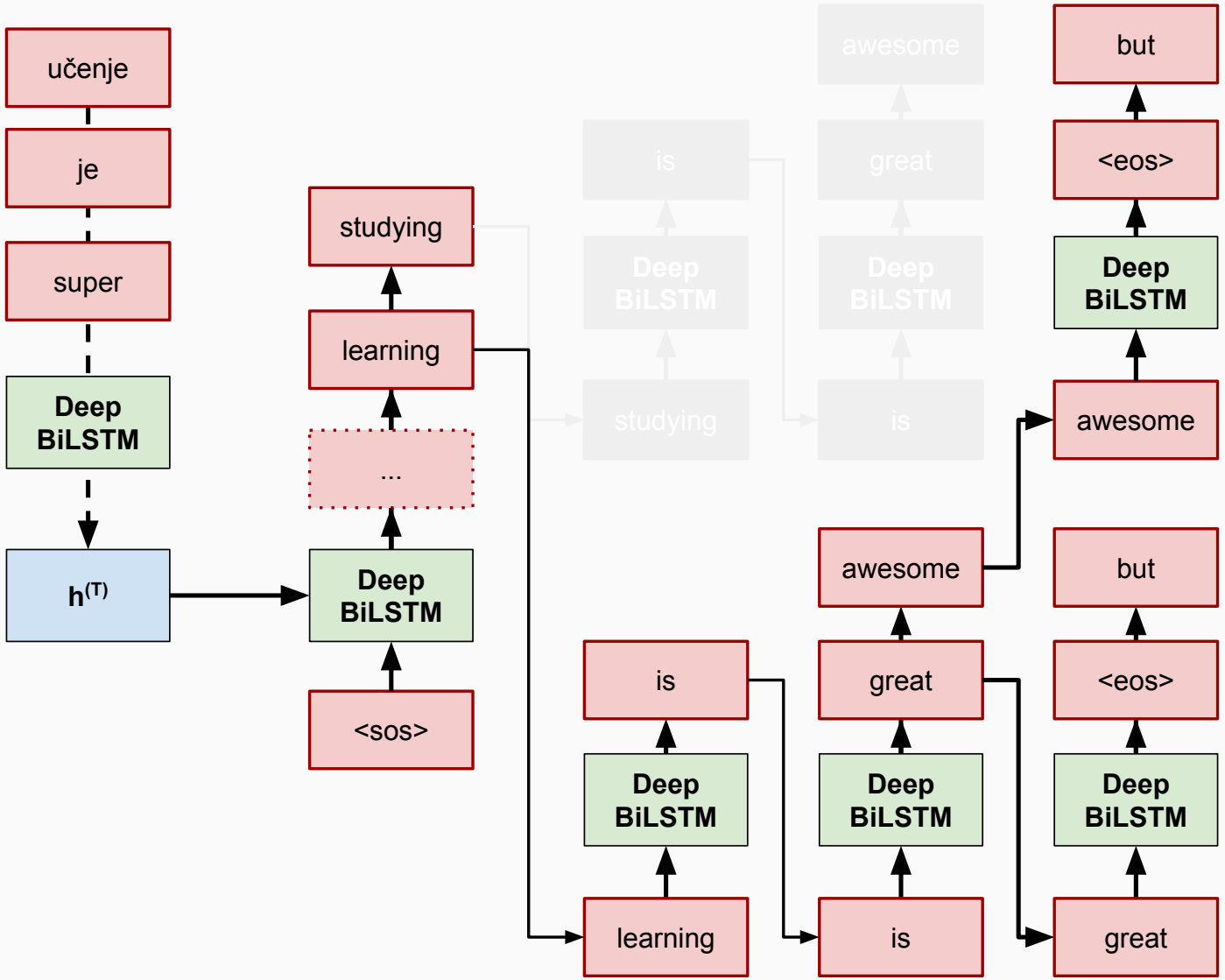
Sequence-to-sequence: vizualizing the beam search



Sequence-to-sequence: vizualizing the beam search



Sequence-to-sequence: vizualizing the beam search



Sequence-to-sequence: summary

Task complexity arises from the variable length of the target sequence:

- instead of “simple” context-based classification, the model has to learn to generate entire sequences.

We can approach this problem by separating it into **i)** reading the input sequence and **ii)** generating the output sequence:

- encoder and decoder have separate parameters
- they are learned together from end to end
- the same approach suitable for txt-to-img and img-to-txt translation.

Downside: multilingual support requires a quadratic number of models

- must train one model for each pair of languages
- some contemporaneous approaches address this task by

Sequence-to-sequence: summary (2)

Early training phase is problematic:

- early training can be improved with **teacher forcing** ("cheat-notes") in some percentage of training samples

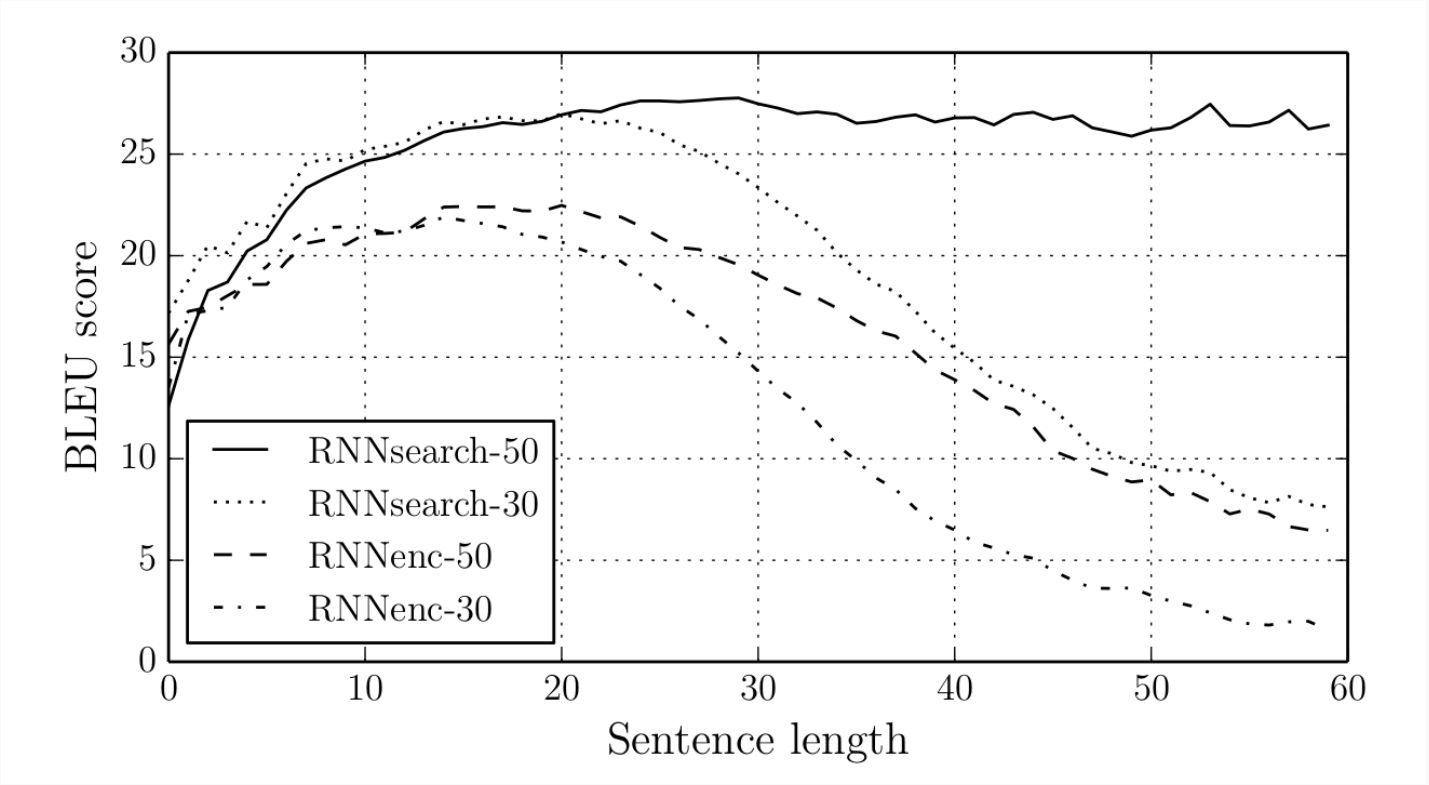
Sequence generation is hard:

- it has to maximize the probability of the entire **sequence** instead of probability of individual components
- this can be approached with **beam search** that tracks k most probable sequences in each step of the generation process

Attention

Attention

Success of machine translation for different sentence lengths. RNNsearch models use attention. Figure from [bahdanau14iclr].



Even with state-of-the-art recurrent cells, the translation success visibly deteriorates as the length of sentence increases

- this suggests that recurrent models have a poor memory.

Attention

Motivation:

- *"When I'm translating a sentence, I pay special attention to the word I'm presently translating. When I'm transcribing an audio recording, I listen carefully to the segment I'm actively writing down. And if you ask me to describe the room I'm sitting in, I'll glance around at the objects I'm describing as I do so."*

Our hidden representations are **not** perfect (they have a limited size).

If a cell can not remember all relevant information, could it at least recall where to find it? This idea leads to a new module (layer):

- denote the hidden representation as a **query**
- denote previous hidden representations as **keys** (memory)
- determine *similarity* between the query and all keys
- aggregate representations from the encoder through **weighted pooling** where the weights correspond to the similarity.

Attention: the baseline formulation

Assume a trainable similarity score between two vectors:

$$s = \text{sim}(q, k), \qquad s \in \mathbb{R}, q \in \mathbb{R}^q, k \in \mathbb{R}^k \ ;$$

here $q = h_{dec}^{(t)}$ and $k = h_{enc}^{(t')}$ correspond to the hidden states.

We assess the similarity of the query with respect to *all* keys:

$$s^{(t)} = \text{sim}(q^{(t)}, K), \quad s^{(t)} \in \mathbb{R}^T, K = [k^{(1)}, \ldots, k^{(T)}] \ .$$

The similarities are normalized to a probability distribution:

$$\alpha^{(t)} = \text{softmax}(s^{(t)}) \ .$$

The attention output is a weighted pool of the hidden encoder states:

$$\text{out}_{\text{attn}}^{(t)} = \sum_{t'}^{T} \alpha_{t'}^{(t)} k^{(t')} \ .$$

Attention: baseline formulation

The output is a weighted sum of encoder hidden states

- this vector is concatenated to the decoder state before a word is generated (**it is not used inside the recurrent cells**)

$$h_{dec}^{*(t)} = [h_{dec}^{(t)}; \text{out}_{\text{attn}}^{(t)}] \ .$$

How to formulate similarity?

1. differentiable module with parameters W_1 (matrix) and w_2 (vector) [bahdanau14iclr]:

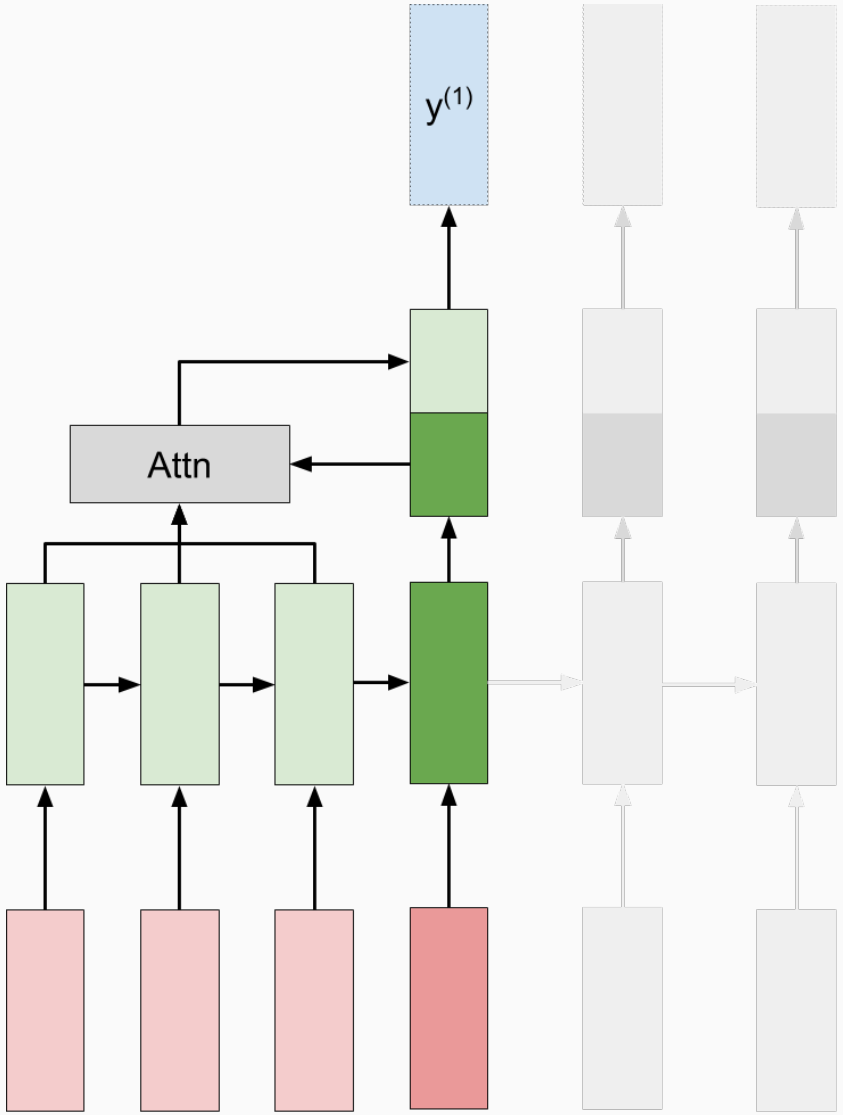
$$s^{(t)} = w_2^\top \cdot \tanh(W_1 \cdot [q^{(t)}; k^{(t')}]) \ .$$

2. Scalar product (condition: $\dim(q) = \dim(k)$):

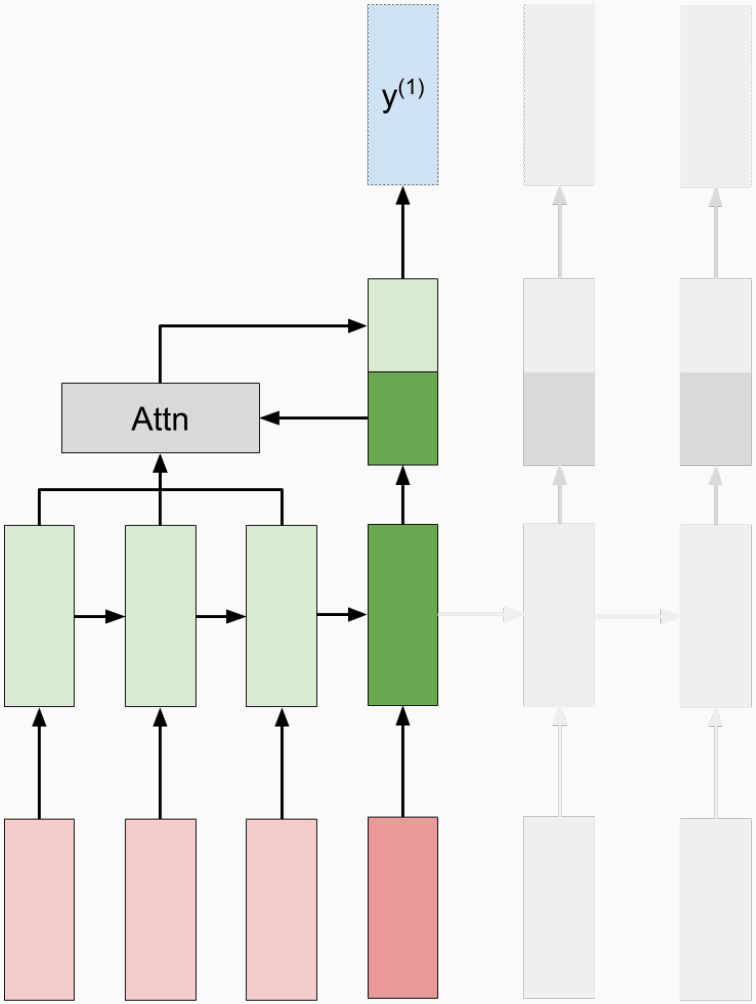
$$s^{(t)} = \frac{q^{(t)\top} \cdot k^{(t')}}{\sqrt{\dim(k)}} \ .$$

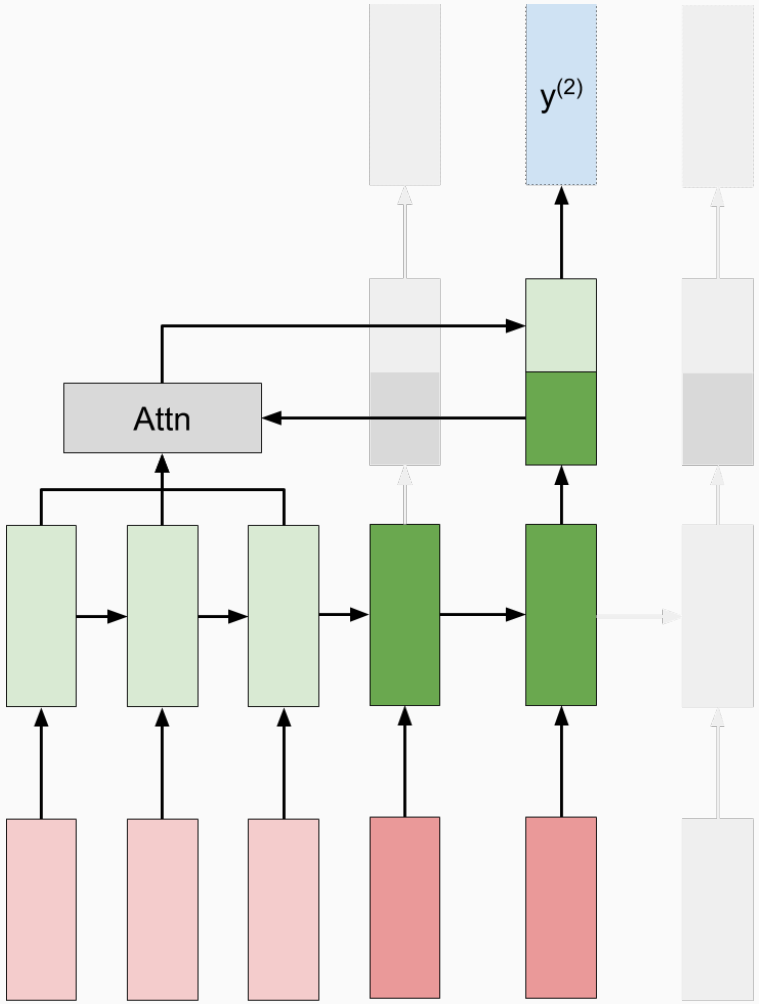
   - why do we scale with dimension size k?

Attention: visualization

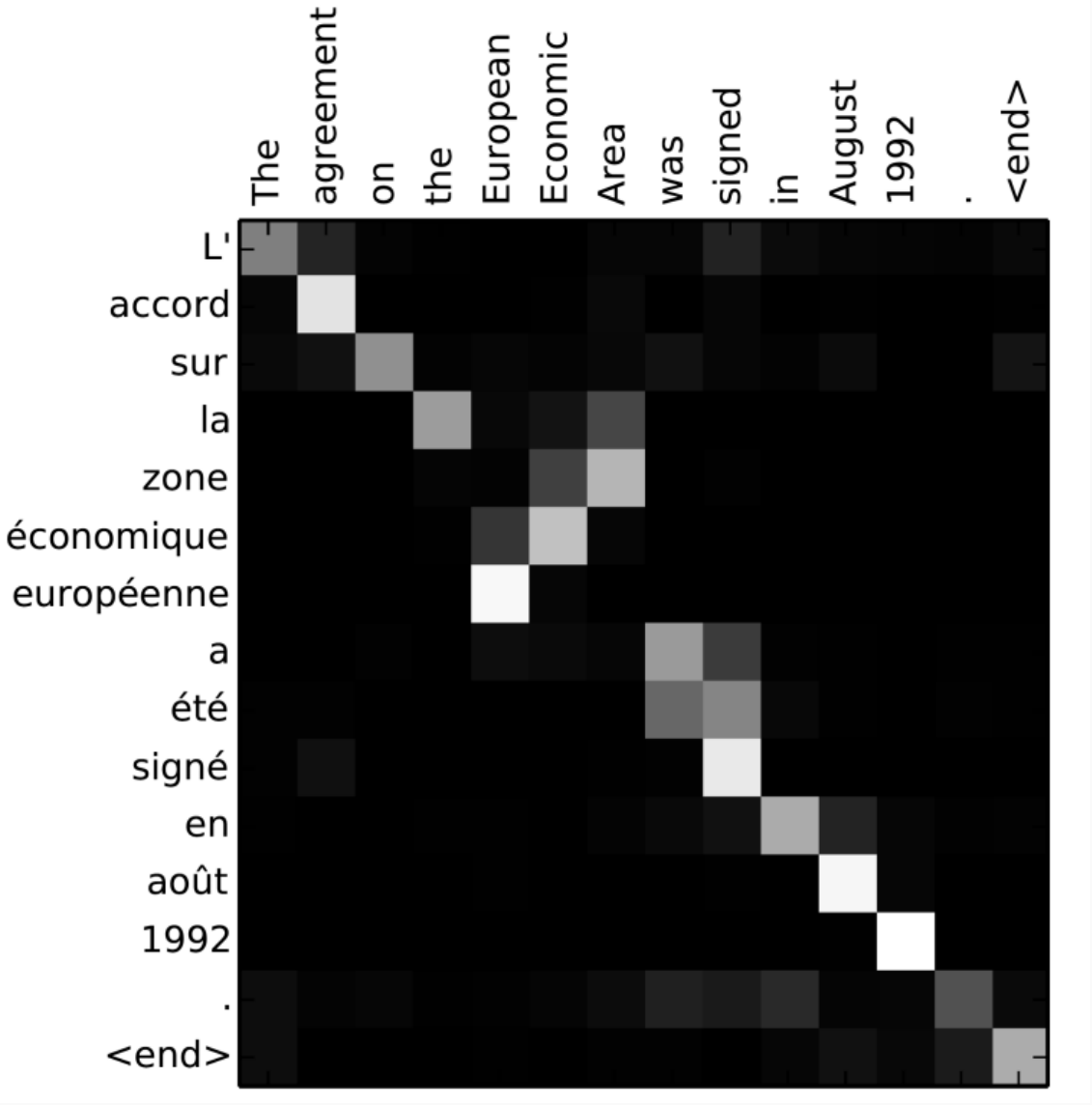


Attention: graphic overview





Attention: visualization of similarity



Similarity between the hidden encoder and decoder states for French to English translation.

Attention: extended formulation

We can decouple retreival from content generation by introducing **values**:

$$k^{(t')} = f_k(h_{enc}^{(t')}), \quad v^{(t')} = f_v(h_{enc}^{(t')}) \ .$$

- f_k and f_v transform a hidden vector into **keys** and **values**
- in practice, f_k and f_v are projections.

$$k^{(t')} = W_k h_{enc}^{(t')}, \qquad v^{(t')} = W_v h_{enc}^{(t')} \ .$$

Extended attention:

$$\alpha^{(t)} = \text{softmax}(\text{sim}(q^{(t)}, K)),$$

$$\text{out}_{\text{attn}}^{(t)} = \sum_{t'}^{T} \alpha_{t'}^{(t)} v^{(t')} \ .$$

This formulation goes beyond sequence-to-sequence translation.

Attention: summary

Our best recurrent models still struggle with long sentences

Hence we introduce **attention** to model long-distance connections

- attention is a weighted pool of hidden states
- the weights model **similarity** with respect to some query
  - information relevance depends on the current need
- in recurrent sequence-to-sequence approaches, **query** is the current hidden state of the decoder, while we attend to the hidden states of the **encoder**
- extended variants may be applied to sequence classification and dense prediction

Attention: summary (2)

Ways to define similarity:

- Bahdanau attention: differentiable module operates on a concatenation of the query and the key
- scalar-product attention: *direct* comparison of a (projected) query with a (projected) key

Attention has been used with practically all RNN variants since its introduction.

Attention is a critical component of contemporaneous deep learning approaches.

Self-attention

Self-attention in sequence classification

If the query comes from the **same** representation as the keys, the attention $\text{sim}(k_i, K)$ may approach one-hot vector e_i:

- we can avoid this with **learned queries** as we show below
- note that computer vision does not appear to suffer from this problem!

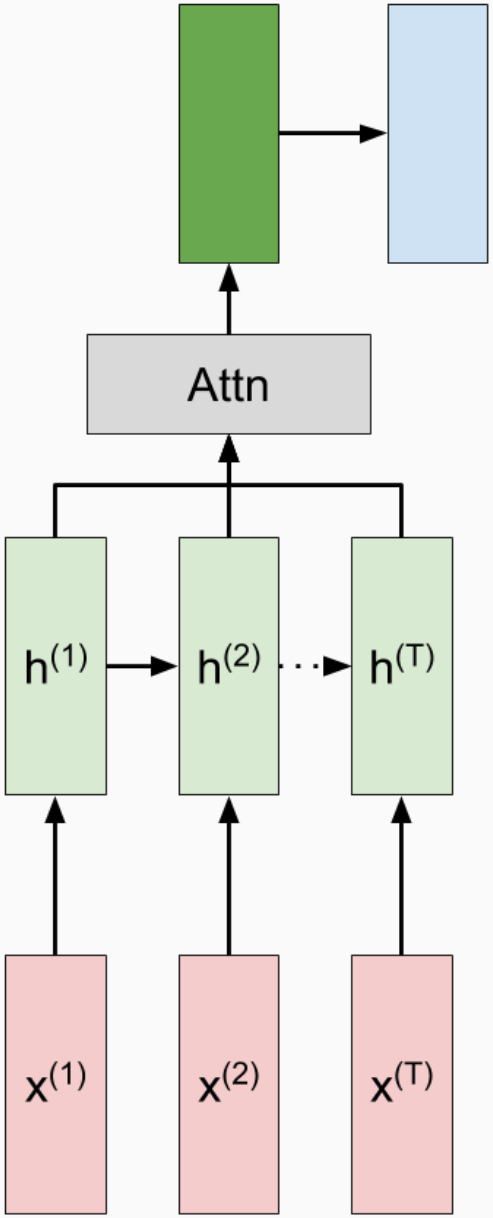
Self-attention with learned queries q_ϕ:

$$\hat{\alpha} = \text{softmax}(\text{sim}(q_\phi, K)),$$

$$\text{out}_{attn} = \sum_t^T \hat{\alpha}_t v^{(t)} \ .$$

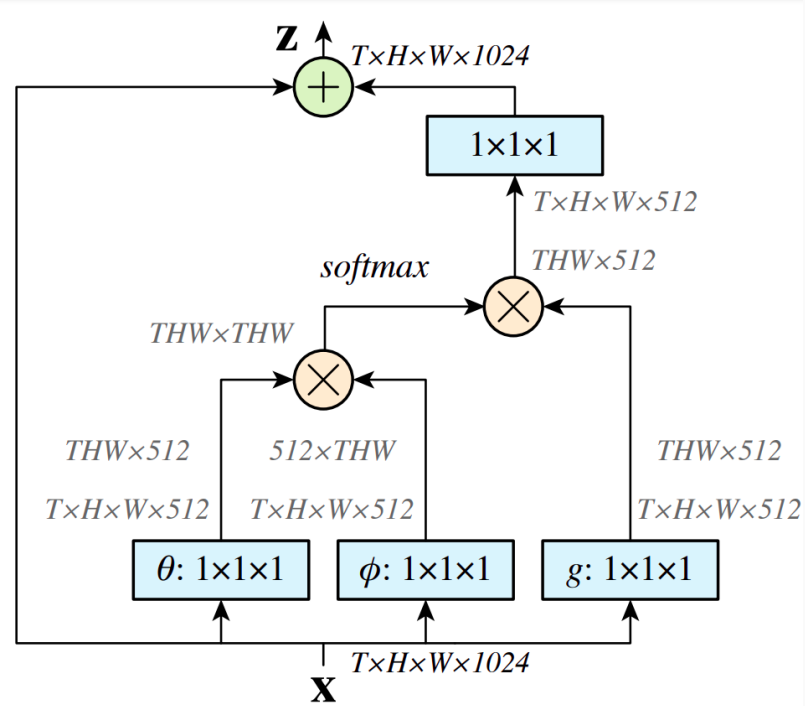
Intuitively, the learned queries correspond to abstract concepts such as *formal writing*, *slang*, *football*, *middle east*, etc.

Self-attention in sequence classification: visualization



Self-attention in computer vision

Some computer vision algorithms capture **long-range** dependencies by extended self-attention without learned-queries.



[wang18cvpr]

Input: abstract representation X

- 4th-o tensor T×H×W×1024
- can be viewed as THW×1024.
- H - height, W - width, T - time

Output: representation Z with improved long-distance connectivity

Input X is projected onto queries (θ), keys (ϕ) and values (g).

Each spatio-temporal feature $x_i \in R^{1024}$ is both a query and a key.

The similarity matrix A (THW × THW) compares queries with keys.

Self-attention in computer vision (details)

The matrix A can be obtained through matrix multiplication:

- other formulations of similarity are easily plugged-in.

$$A = (W_\theta X^\top)^\top \cdot (W_\phi X^\top),$$
$$= (X W_\theta^\top) \cdot (W_\phi X^\top) \ .$$

The weight matrix α is obtained by activating rows of A with softmax.

- A_{ij} reflects similarity of the query $W_\theta x_i$ wrt the key $W_\phi x_j$

$$\alpha = \text{softmax}(A, axis = 1) \ .$$

‘ Outputs $Z = \{z_i\}$ are linear combinations of values $V = g(x_i)$:

- of course, the weights correspond to the elements of α

$$z_i = \sum_j \alpha_{ij} \cdot g(x_j) \ .$$

Attention is … all we need?

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

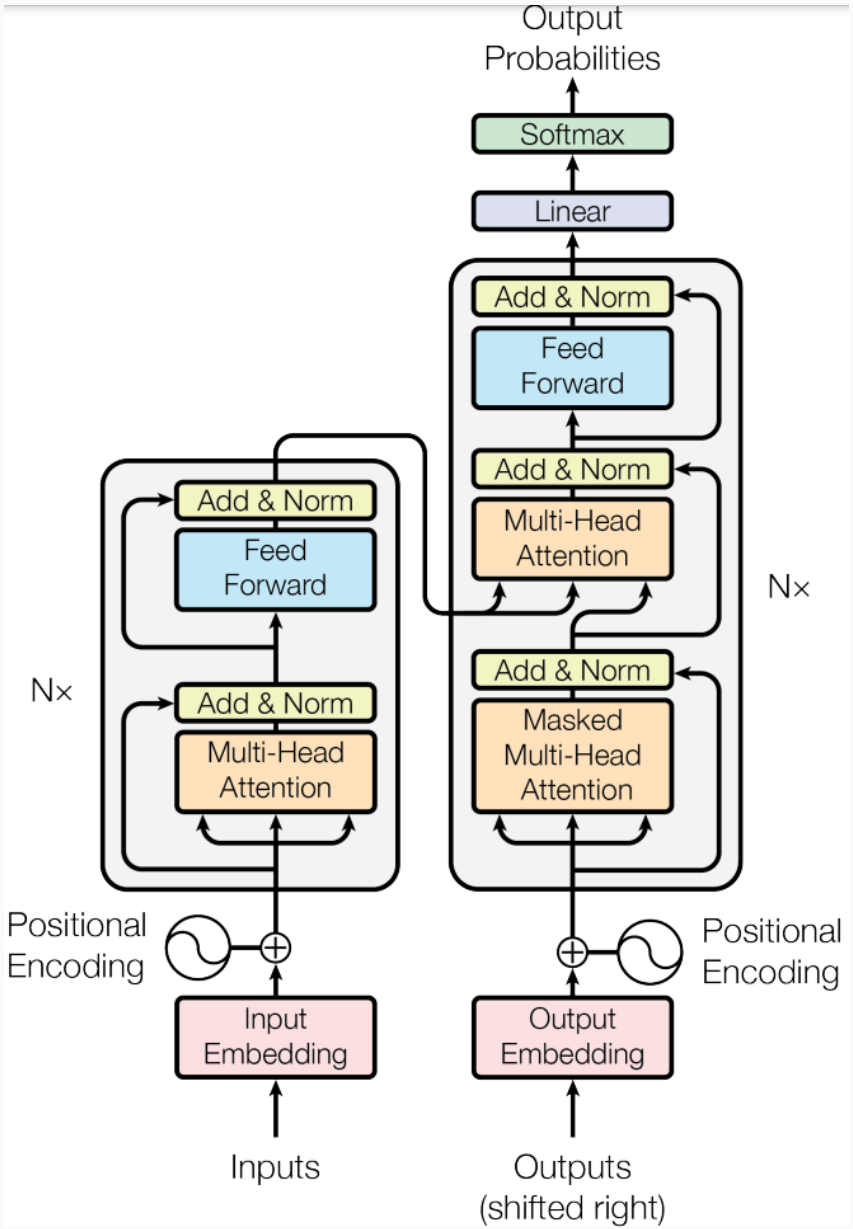
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

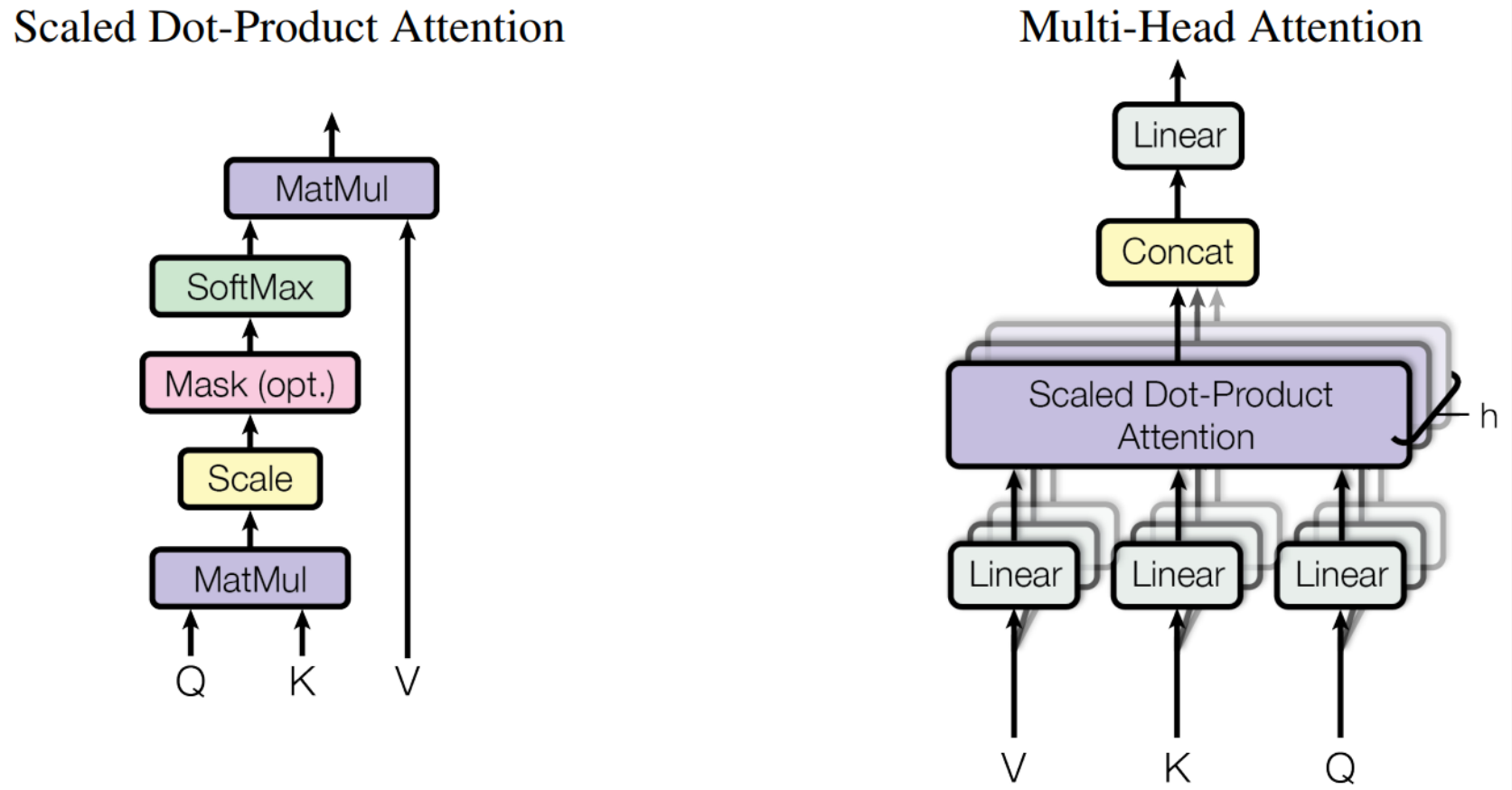
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions

Models based exclusively on attention

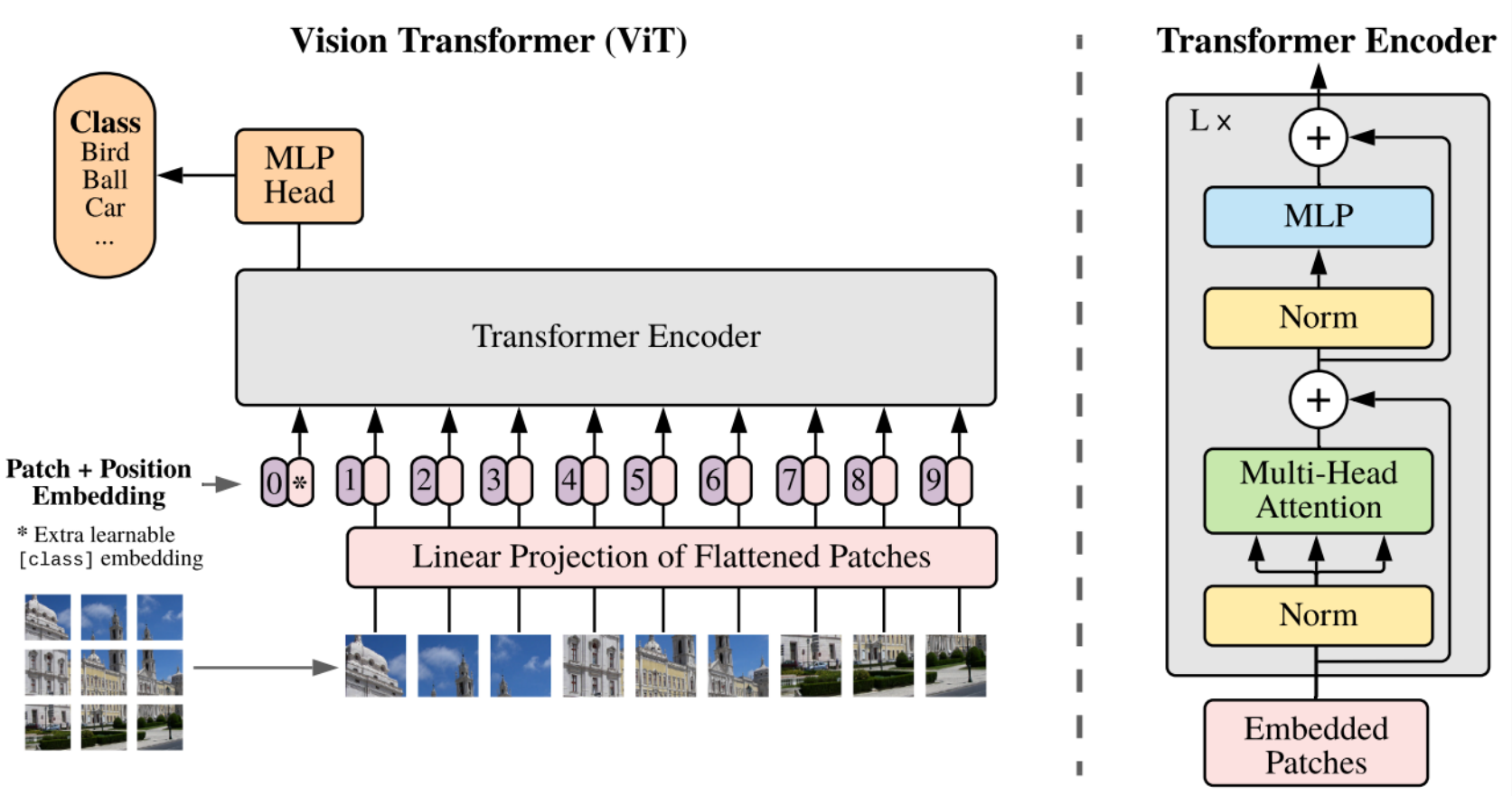


Models based exclusively on attention



Annotated paper with code samples:
http://nlp.seas.harvard.edu/2018/04/03/attention.html

Models based exclusively on attention



Questions?

Questions? :)

In the book

- Relevant chapters
  - 10.1, 10.2, 10.3, 10.4, 10.5, 10.7, 10.10, 10.11