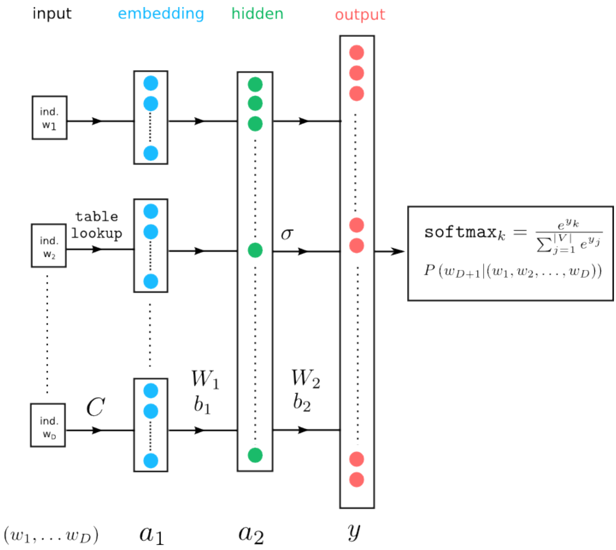
DD2417
8a: A simple neural language model

Johan Boye, KTH

Language modeling



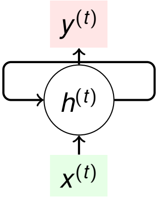
DD2417
8b: Recurrent neural networks (RNNs)

Johan Boye, KTH

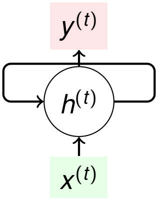
Recurrent neural networks (RNNs)

A recurrent neural network (RNN), the network maintains a hidden state, which is updated as a function of the input and the last hidden state.

The output is computed as a function of the hidden state.



Recurrent neural networks (RNNs)



The hidden state is updated as a function of the input and last hidden state:

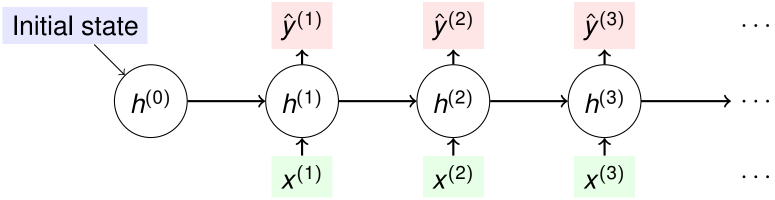
$$h^{(t)} = g(W_{hh}h^{(t-1)} + W_{xh}x^{(t)} + b^{(t)})$$

The output is a function of the current state:

$$\hat{y}^{(t)} = f(W_{hy}h^{(t)})$$

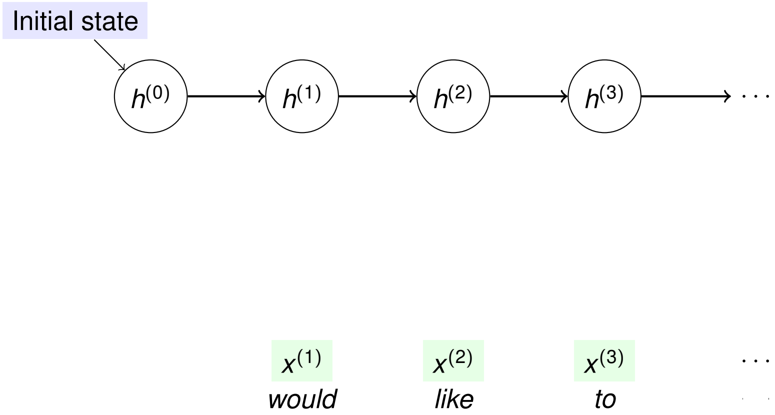
(f and g are non-linear activation functions)

Unrolling RNNs

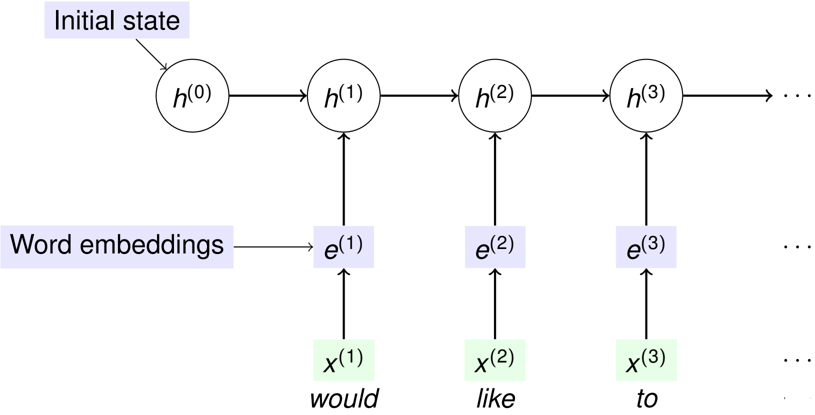


Unrolling = displaying RNNs with every time step separately.

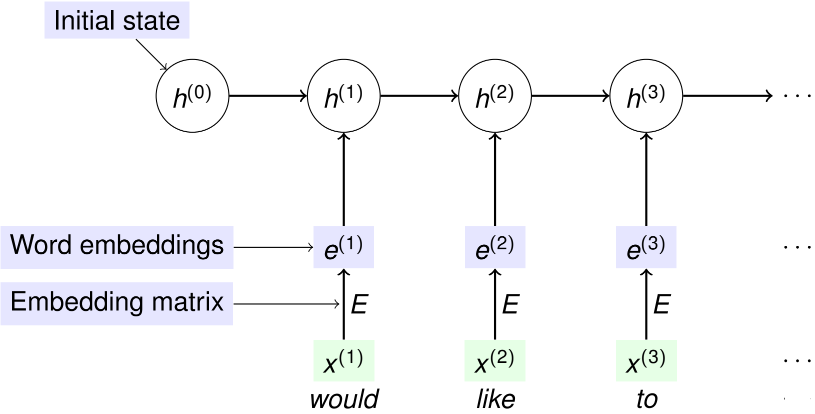
Language RNNs



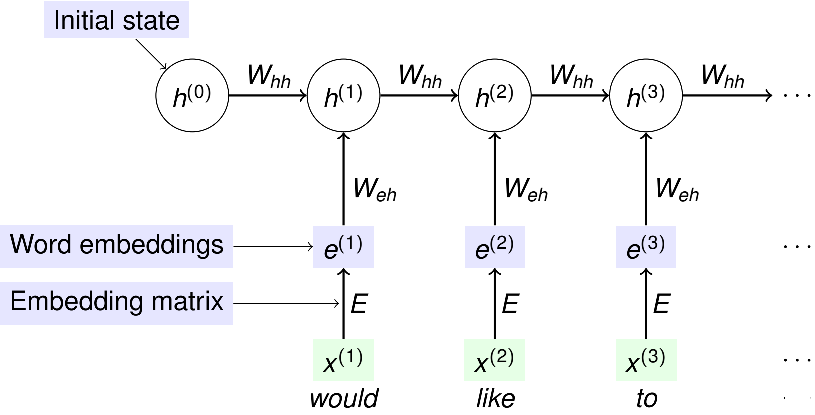
Language RNNs



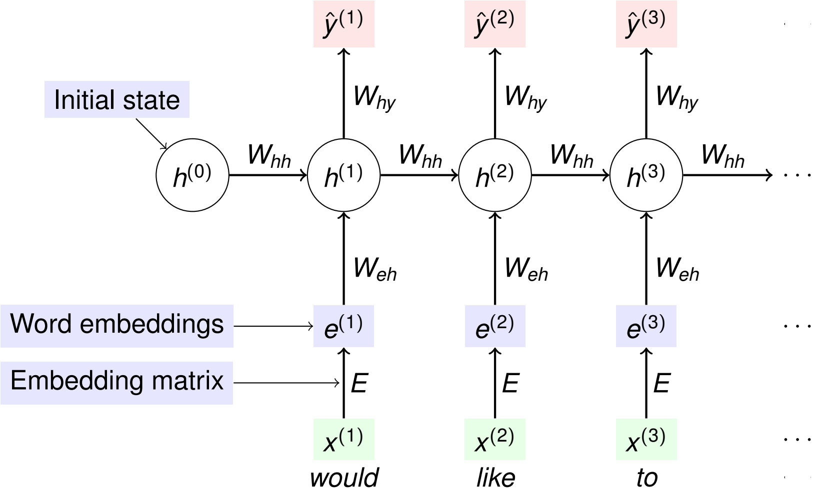
Language RNNs



Language RNNs



Language RNNs



RNNs for language processing

Some nice things:

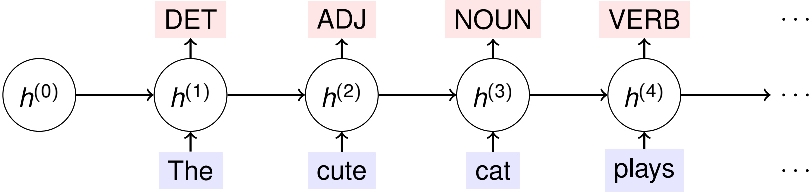
- Can handle arbitrarily long sequences
- Model size is independent on sequence length.
- Can (potentially) remember things from way back.

But:

- Can be slow to train.
- Non-parallelizable.
- Vanilla RNNs don't actually remember long-term dependencies so well (but there are extensions that do).

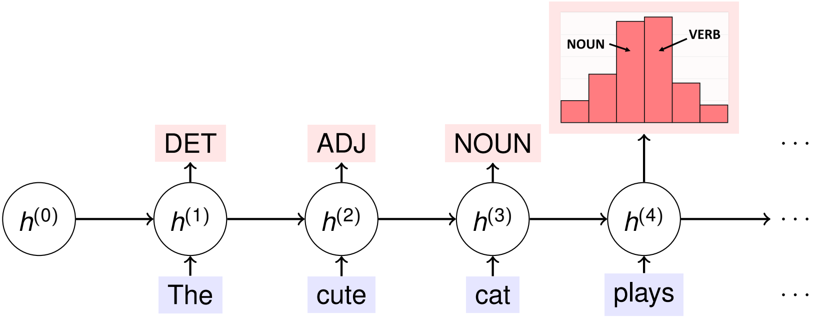
Labeling words in a sequence

RNNs can be used for labeling words in a sequence (e.g. POS tagging).

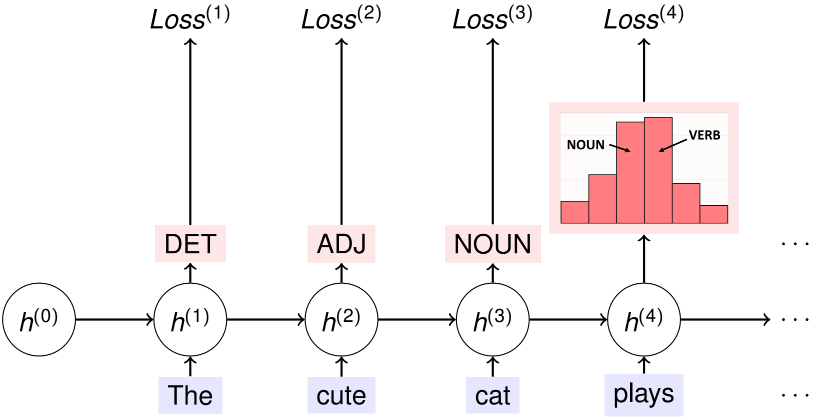


Labeling words in a sequence

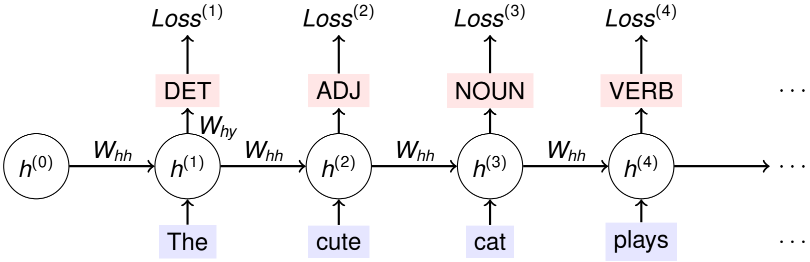
RNNs can be used for labeling words in a sequence (e.g. POS tagging).



Training


$Loss^{(1)}$
$Loss^{(2)}$
$Loss^{(3)}$
$Loss^{(4)}$
NOUN
VERB
DET
ADJ
NOUN
...
$h^{(0)}$
$h^{(1)}$
$h^{(2)}$
$h^{(3)}$
$h^{(4)}$
...
The
cute
cat
plays
...

Training

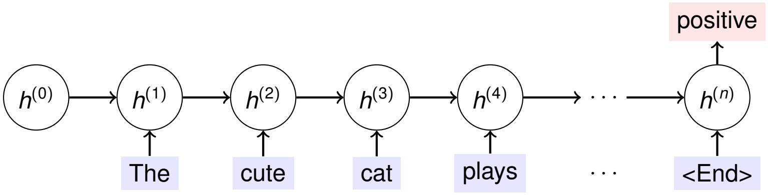


Need to compute the gradient of the loss in each timestep w.r.t. all trainable parameters, e.g. $\frac{\partial Loss^{(i)}}{\partial W_{hh}}$.

The backpropagation-through-time (BPTT) algorithm solves this problem.

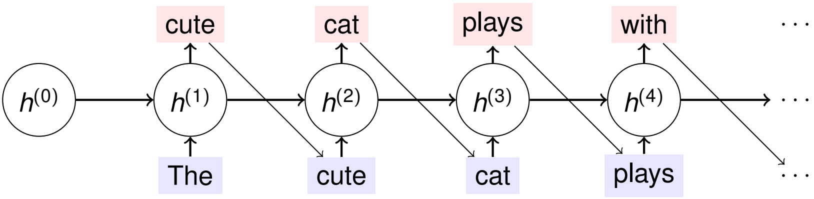
Sequence classification

RNNs can be used to classify the entire sequence, e.g. sentiment analysis.



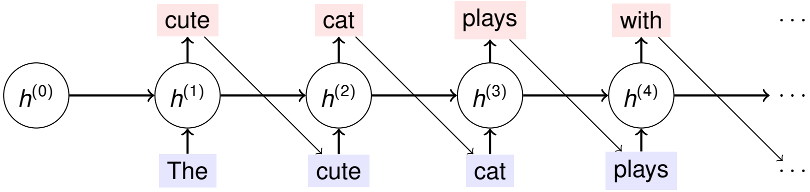
Language models

RNNs can be used to predict/generate the next word.



Language models

RNNs can be used to predict/generate the next word.

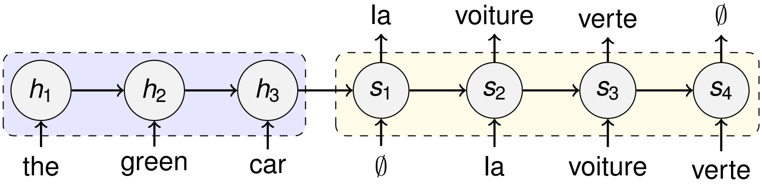


This is an autoregressive model: it takes its own previous output into account when producing the next output.

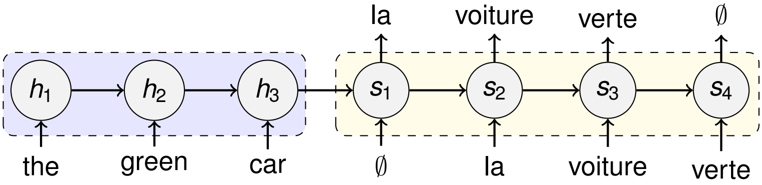
Training can be done by inputting the produced output or the correct output of the preceding time step.

The latter is called teacher forcing.

Translation



Translation



This is an example of an encoder-decoder architecture.

The hidden state $h^{(n)}$ (hopefully) encodes all necessary information about the source sentence, which can then be decoded into the target sentence.

In practice, this scheme needs to be extended (more on this later).

RNNs in Pytorch

In Pytorch, the hidden states are the outputs.

```
import torch
import torch.nn as nn
input_size = 5
hidden_size = 3
rnn = nn.RNN(input_size,hidden_size, bidirectional=False)

sequence_length = 10
x = torch.rand(sequence_length, input_size)
hidden_states, last_hidden_state = rnn(x)
print( hidden_states.shape, last_hidden_state.shape )
```

will print out

```
torch.Size([10, 3]) torch.Size([1, 3])
```

RNNs in Pytorch

You can process a batch of inputs at the same time.

```
import torch
import torch.nn as nn
input_size = 5
hidden_size = 3
rnn = nn.RNN(input_size,hidden_size, batch_first=True)

sequence_length = 10
batch_size = 32
x = torch.rand(batch_size, sequence_length, input_size)
hidden_states, last_hidden_state = rnn(x)
print( hidden_states.shape, last_hidden_state.shape )
```

will print out

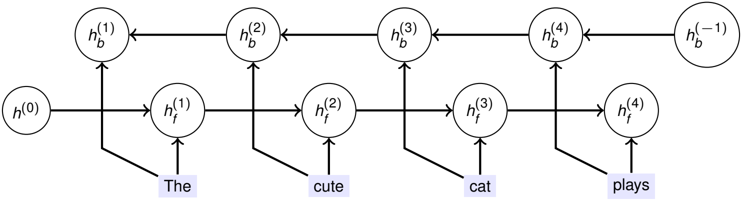
```
torch.Size([32, 10, 3]) torch.Size([1, 32, 3])
```

DD2417
8c: Extentions of RNNs

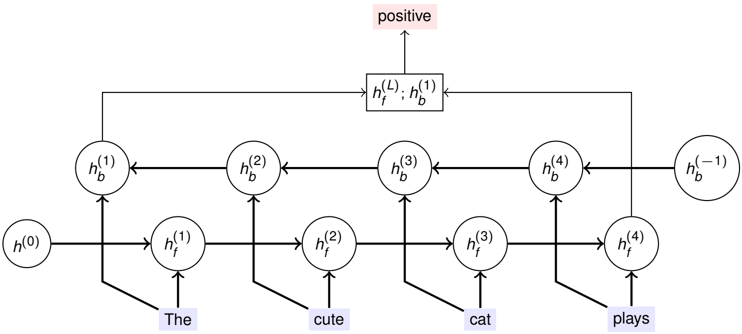
Johan Boye, KTH

Bi-directional RNNs

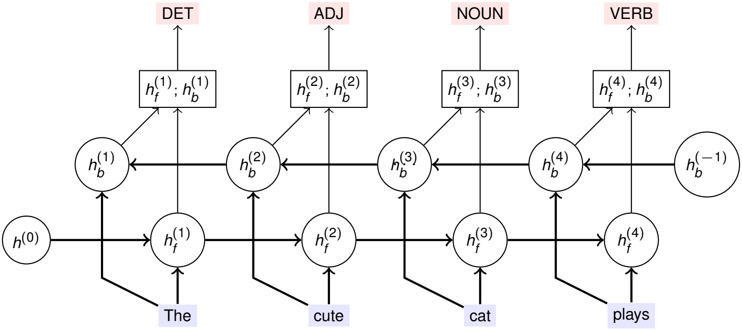
Often it is useful to get information both from the left and the right context.



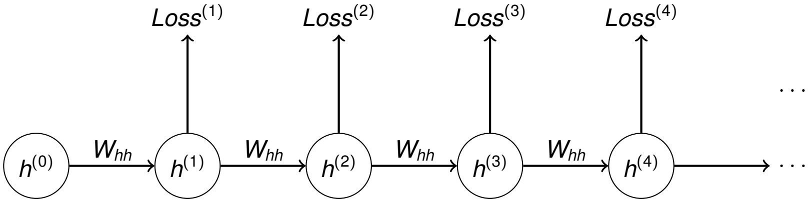
Bi-directional RNNs



Bi-directional RNNs

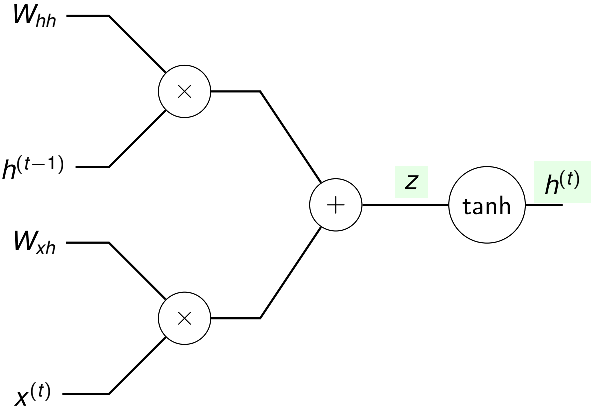


Vanishing gradient problem

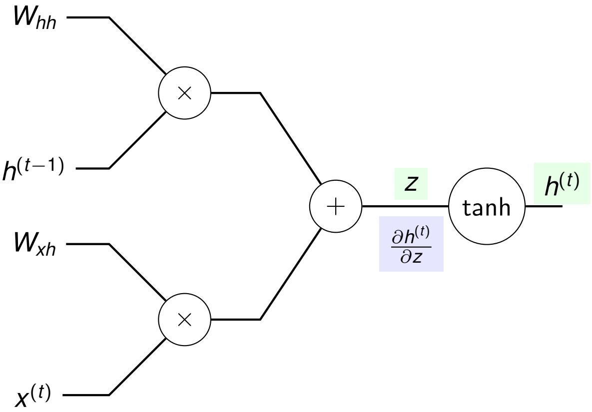


Gradient of an RNN cell

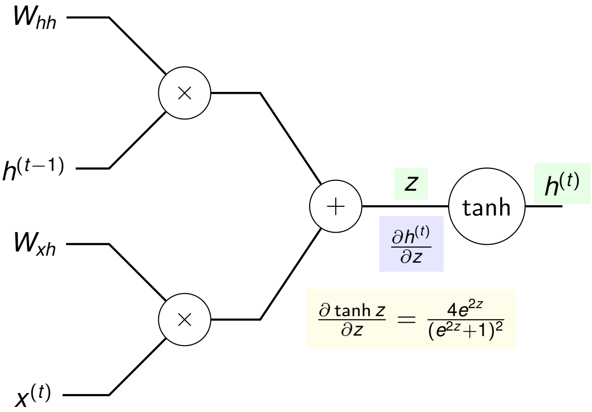
$$h^{(t)} = \tanh(W_{hh}h^{(t-1)} + W_{xh}x^{(t)})$$



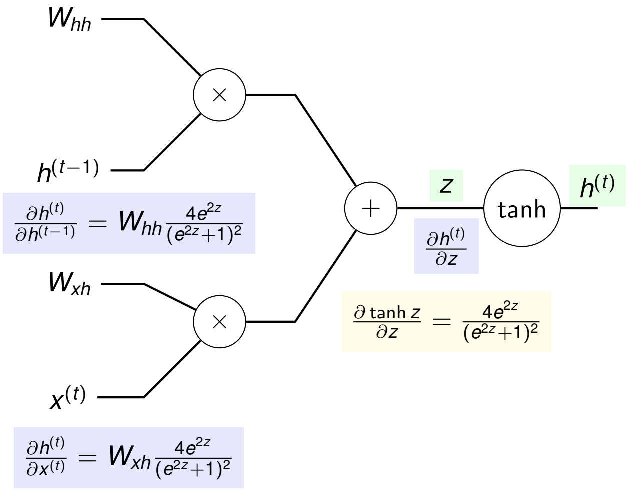
Gradient of an RNN cell



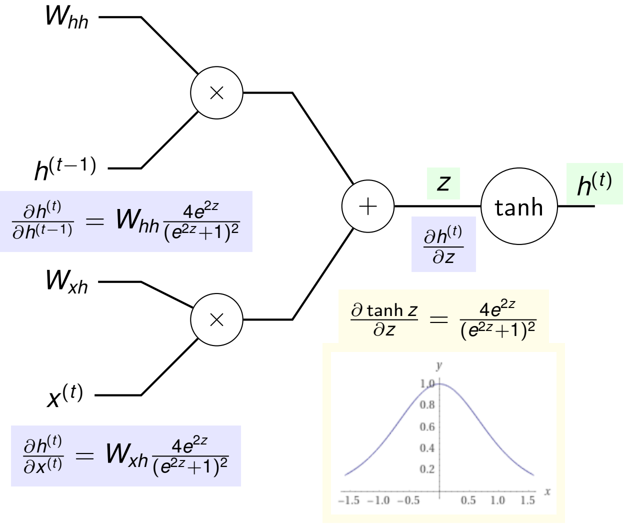
Gradient of an RNN cell



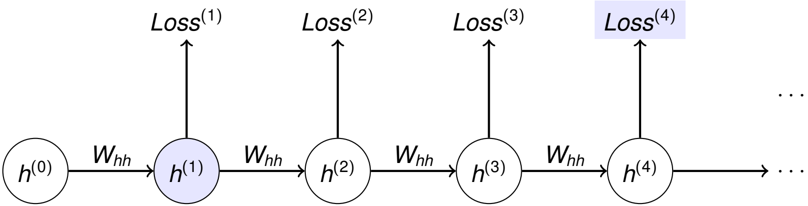
Gradient of an RNN cell



Gradient of an RNN cell

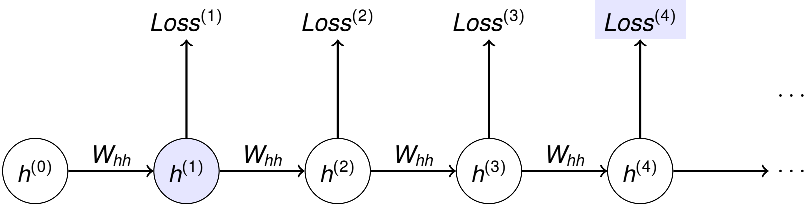


Vanishing gradient problem



$$\frac{\partial Loss^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial Loss^{(4)}}{\partial h^{(4)}}$$

Vanishing gradient problem



$$\frac{\partial Loss^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial Loss^{(4)}}{\partial h^{(4)}}$$

If all intermediate gradients are small, then the gradient signal from $Loss^{(4)}$ on $h^{(1)}$ is going to be very small...

... but the gradient signal from $Loss^{(2)}$ is going to be stronger ...

... so the RNN will have problems learning long-distance relationships!

Dependencies in text

$\boxed{\text{Apples}}$ $\boxed{\text{are}}$ tasty.

Distance: 1 word

Dependencies in text

$\boxed{\text{The search}}$ for more tasty apples $\boxed{\text{is}}$ ongoing.

Distance: 5 words

Dependencies in text

Since the reasons for requesting a leave of absence can have important implications for academic planning and external financial aid , students are encouraged ...

Distance: 20 words

Managing context in RNNs

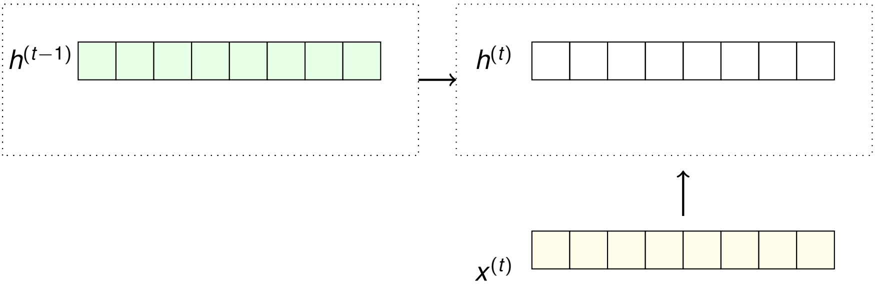
To make RNNs better capture long-distance dependencies, we can add so-called gates that better control the flow of information.

Two important suggestions:

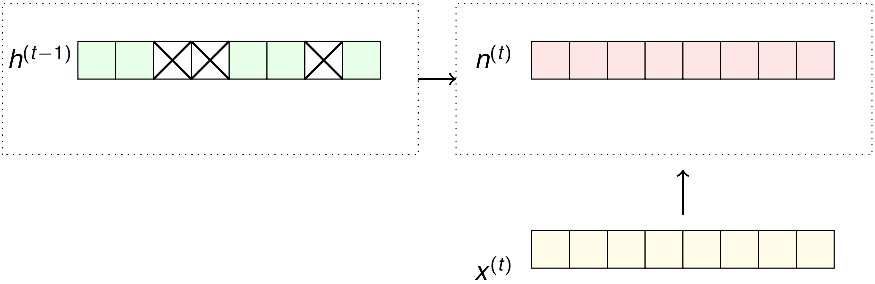
- Long Short-Term Memory (LSTM) networks (Schmidthuber and Hochreiter 1997)
- Gated Recurrent Units (GRU) networks (Cho et al. 2014)

LSTMs are more powerful, but GRUs are quicker to train and simpler to understand and implement.

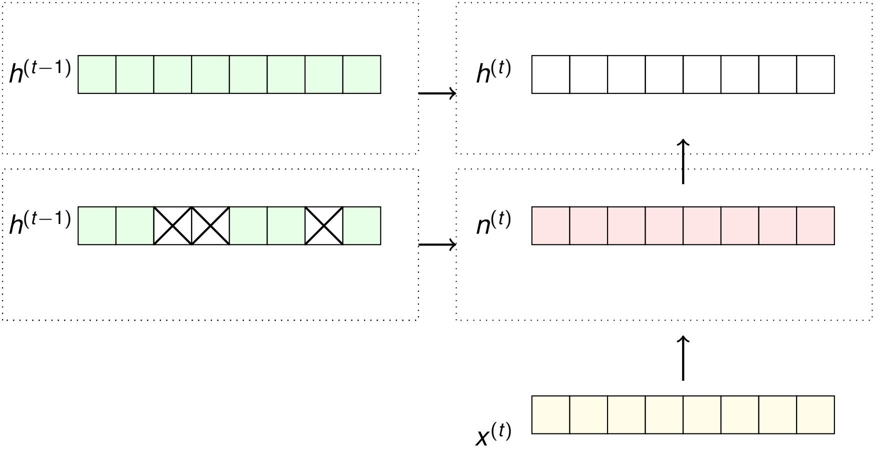
Gated Recurrent Units (GRUs)



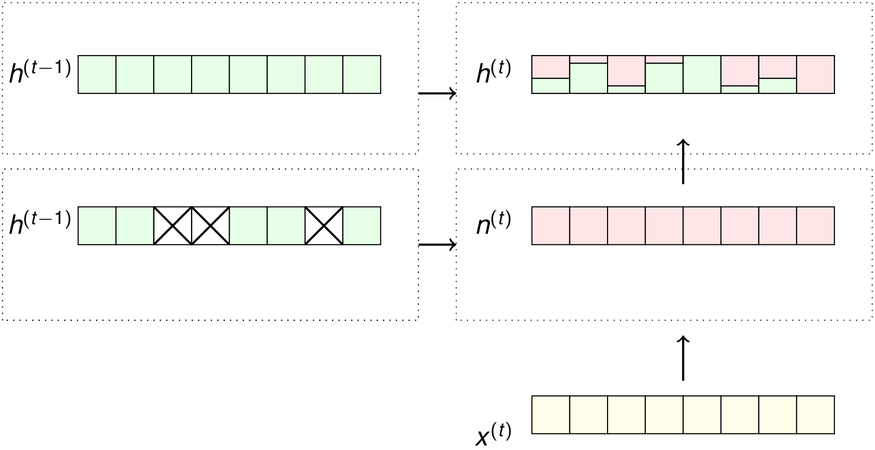
Gated Recurrent Units (GRUs)



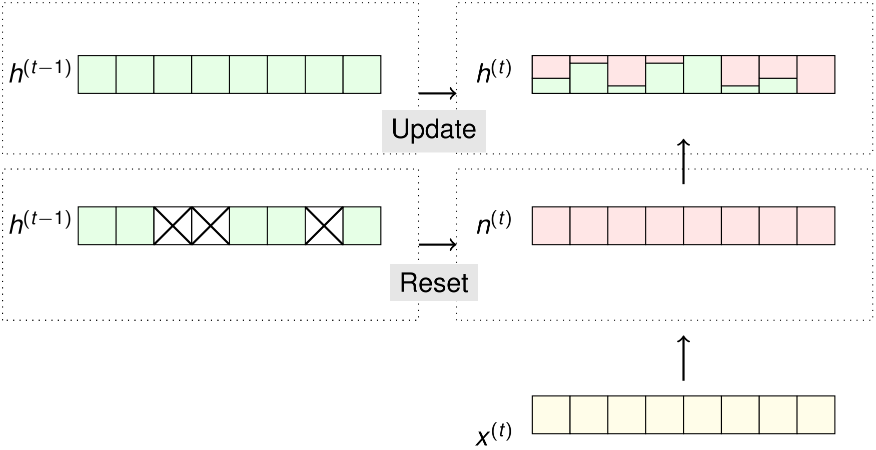
Gated Recurrent Units (GRUs)



Gated Recurrent Units (GRUs)



Gated Recurrent Units (GRUs)



Gated Recurrent Units (GRUs)

The update of the hidden states are controlled by two gates:

- the reset gate r controls what part of the previous hidden state is relevant for the current situation
- the update gate z decides what of the old previous hidden state should be retained, and what part should be updated

$$r^{(t)} = \sigma(W_{ir}x^{(t)} + W_{hr}h^{(t-1)})$$
$$z^{(t)} = \sigma(W_{iz}x^{(t)} + W_{hz}h^{(t-1)})$$

The tentative new hidden state:

$$n^{(t)} = \tanh(W_{in}x^{(t)} + r^{(t)} \odot W_{hn}h^{(t-1)})$$

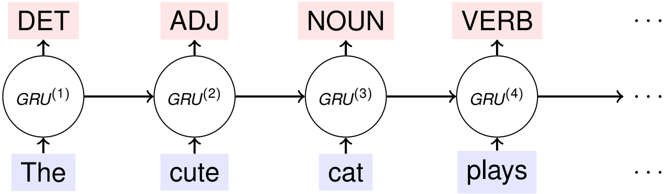
The new hidden state:

$$h^{(t)} = (1 - z^{(t)}) \odot n^{(t)} + z^{(t)} \odot h^{(t-1)}$$

($\odot$ = component-wise multiplication)

GRU networks

Gated Recurrent Units can be then be used in RNNs instead of (just) hidden states.



GRUs in Pytorch

GRUs are used exactly as RNNs in Pytorch

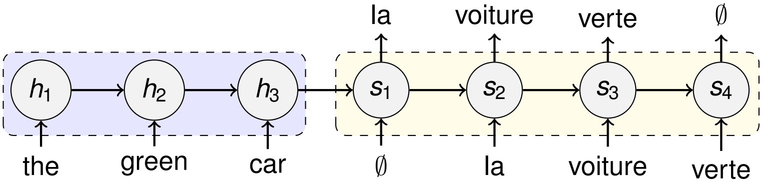
```
import torch
import torch.nn as nn
input_size = 5
hidden_size = 3
rnn = nn.GRU(input_size,hidden_size, batch_first=True)

sequence_length = 10
batch_size = 32
x = torch.rand(batch_size, sequence_length, input_size)
hidden_states, last_hidden_state = rnn(x)
print( hidden_states.shape, last_hidden_state.shape )
```

will print out

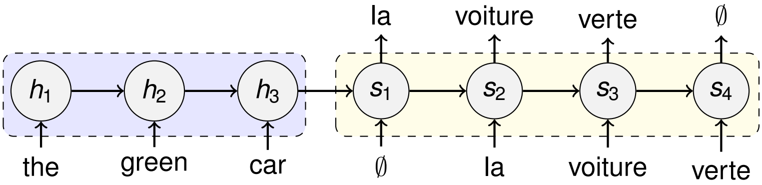
```
torch.Size([32, 10, 3]) torch.Size([1, 32, 3])
```

Encoder-decoder revisited

Sutskever et al. (2014) suggested that translation can be done with an *encoder-decoder* architecture.



Encoder-decoder revisited

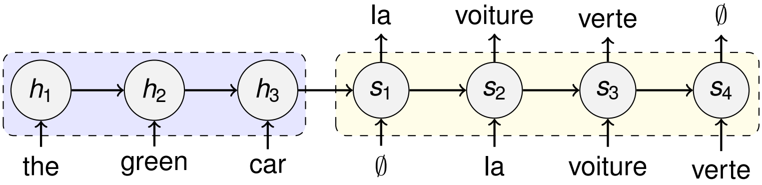


h_3 is an *encoding* of the English sentence, which is then *decoded* into the French translation.

Potential problem: h_3 is not large enough to contain all necessary information.

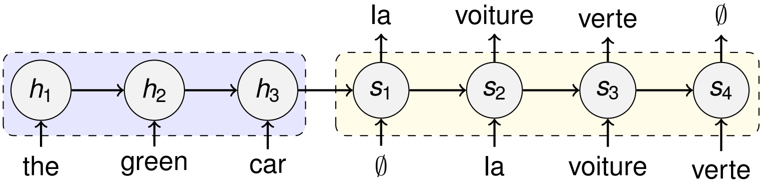
Solution: Use all h_i as input to the decoder.

Alignment/Attention mechanism



	la	voiture	verte
the	0.88	0.01	0.01
green	0.01	0.01	0.89
car	0.11	0.98	0.10

Alignment/Attention mechanism



	s_1	s_2	s_3
h_1	0.88	0.01	0.01
h_2	0.01	0.01	0.89
h_3	0.11	0.98	0.10

Alignment/Attention mechanism

	la (s_1)	voiture (s_2)	verte (s_3)
the (h_1)	0.88	0.01	0.01
green (h_2)	0.01	0.01	0.89
car (h_3)	0.11	0.98	0.10

In ordinary RNNs, s_i is computed as a function of x_i and s_{i-1}.

Now we want to compute s_i from x_i and a *context* c_i.

- c_i is a function of s_{i-1}, and of all encoder states h_j *in proportion to how important they are for generating* s_i
- the function computing c_i is *learnable*

Alignment/Attention mechanism

We will follow the ideas of Bahdanau et al (2014).

First define $e_{ij} = v^\top \tanh(Wh_j + Us_{i-1})$
where v, W, and U are trainable matrices.

Turn e_i into a probability distribution: $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ij})}$

The context for output word i takes the input words into account in proportion to how important they are:

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

Finally, the new hidden state is generated as:

$s_i = \mathrm{GRU}(x_i, c_i)$