Learning from Data

Lecture 4: RNN, LSTM, Seq2Seq

Rik van Noord - 25 September 2023

Assignment 1 feedback

Nothing personal

Read the assignment well

- Check carefully what you are expected to do/answer
  - Not sentiment analysis!
- A few good reports that missed relatively easy points
- Key points are added throughout the assignments

Read the assignment well

- Check carefully what you are expected to do/answer
  - Not sentiment analysis!
- A few good reports that missed relatively easy points
- Key points are added throughout the assignments

Github:

- OK to use for submission, but remember to not make changes after
- Always send in a ZIP file with all code anyway
- Check what you are actually sending in

General advice

- What information is relevant?
  - “The data also came with a document ID that we do not use”

General advice

- What information is relevant?
  - “The data also came with a document ID that we do not use”
- Make sure the readers knows with what settings you obtained certain results
  - In the text, and preferably also in the caption of the table

General advice

- What information is relevant?
  - “The data also came with a document ID that we do not use”
- Make sure the readers knows with what settings you obtained certain results
  - In the text, and preferably also in the caption of the table
- If you include a table or figure, you need to discuss it
  - Also tell us what is happening, not just “The results are in Table 4” and then move on

General advice

- What information is relevant?
  - “The data also came with a document ID that we do not use”
- Make sure the readers knows with what settings you obtained certain results
  - In the text, and preferably also in the caption of the table
- If you include a table or figure, you need to discuss it
  - Also tell us what is happening, not just “The results are in Table 4” and then move on
- Consider if adding a table or figure really adds anything
  - Do you really need 6 confusion matrices?

General advice

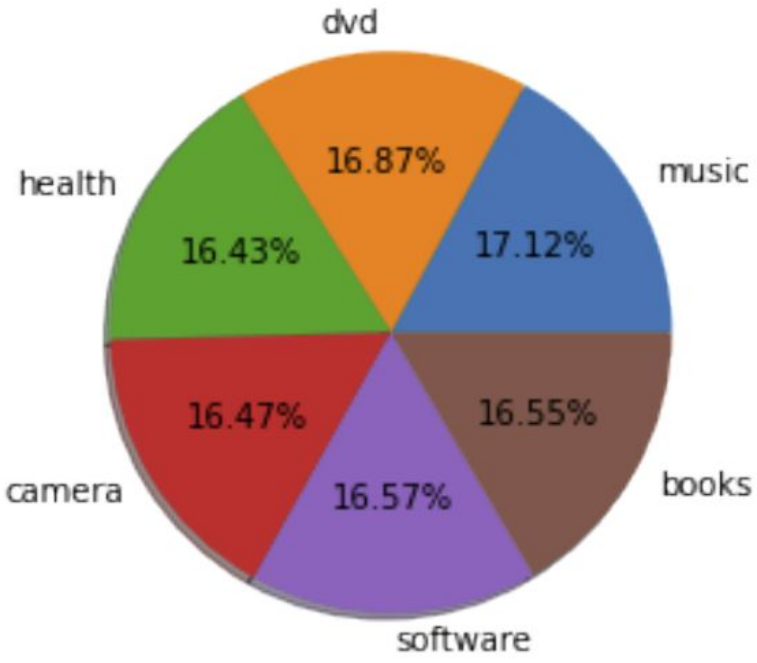
- What information is relevant?
  - “The data also came with a document ID that we do not use”
- Make sure the readers knows with what settings you obtained certain results
  - In the text, and preferably also in the caption of the table
- If you include a table or figure, you need to discuss it
  - Also tell us what is happening, not just “The results are in Table 4” and then move on
- Consider if adding a table or figure really adds anything
  - Do you really need 6 confusion matrices?
- You should comment on whether small differences seem meaningful
  - Is 0.1% accuracy improvement really an improvement?

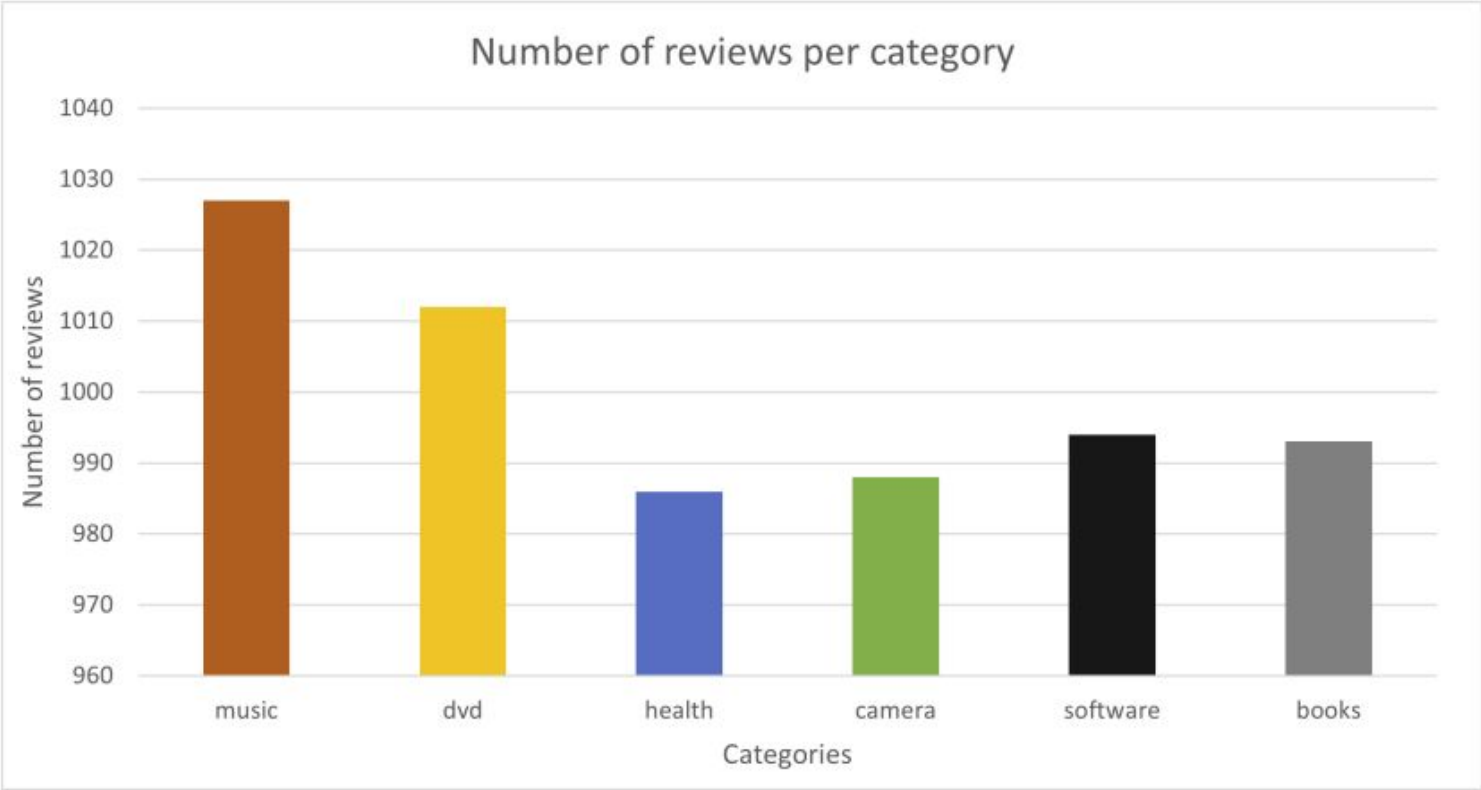
General advice

- What information is relevant?
  - “The data also came with a document ID that we do not use”
- Make sure the readers knows with what settings you obtained certain results
  - In the text, and preferably also in the caption of the table
- If you include a table or figure, you need to discuss it
  - Also tell us what is happening, not just “The results are in Table 4” and then move on
- Consider if adding a table or figure really adds anything
  - Do you really need 6 confusion matrices?
- You should comment on whether small differences seem meaningful
  - Is 0.1% accuracy improvement really an improvement?
- sklearn is the tool, not the method or algorithm
  - No need to describe exactly how to run your code

Data visualization

- Think about what the best option is
  - Often not a bar or pie chart
- Some bad examples will follow


dvd
16.87%
music
17.12%
health
16.43%
16.47%
camera
16.57%
software
16.55%
books


Number of reviews per category
1040
1030
1020
1010
1000
990
980
970
960
Number of reviews
music
dvd
health
camera
software
books
Categories

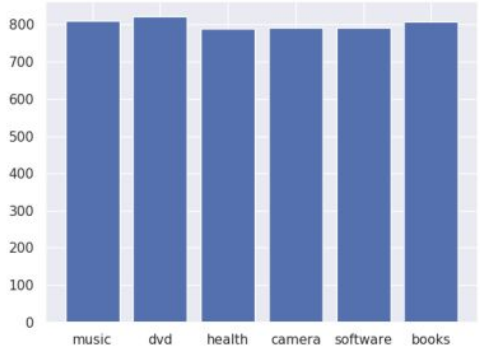


Figure 1: Distribution of the reviews among the topics (training set)

Topic	Count
books	807
camera	790
dvd	821
health	789
music	809
software	790
Total	6000

Table 1: Distribution of the reviews among the topics (training set)

Formatting

- Tables need to be self explanatory, not just caption it “Results of our model”
  - Captions need to end with punctuation
- References to tables/sections/figures are with a capital
  - “See Figure 3”, “This is explained in Section 4”, etc
- Write numbers with a comma: “6,000” instead of “6000”
- Take care of formatting the appendix
  - I’ve seen a 6 page Appendix that would have fit in 3
- No print screens of a terminal, please just create a table
- Check final PDF
  - One group had an empty references section

Table 3: Classification Metrics

	Accuracy without parameters	Best setting
Naive Bayes	89.33%	92%
Decision Trees	77%	79%
Random Forests	85.5%	85.8%
KNN	40.66%	77.66%
SVM	81.66%	92%

Table 4: Accuracy Comparison

This is the lowest accuracy of any of the algorithms or features tested, however the python script for POS tagging might not be completely accurate or fully working which limits the results. The final accuracy in POS tagging may improve if changes to the code are implemented.

4 Discussion/Conclusion

Examining the results from comparatively analyzing our 5 different models showed that the overall accuracy for NB and SVM with adjusted parameters were the highest. The settings we used for both were

Good example

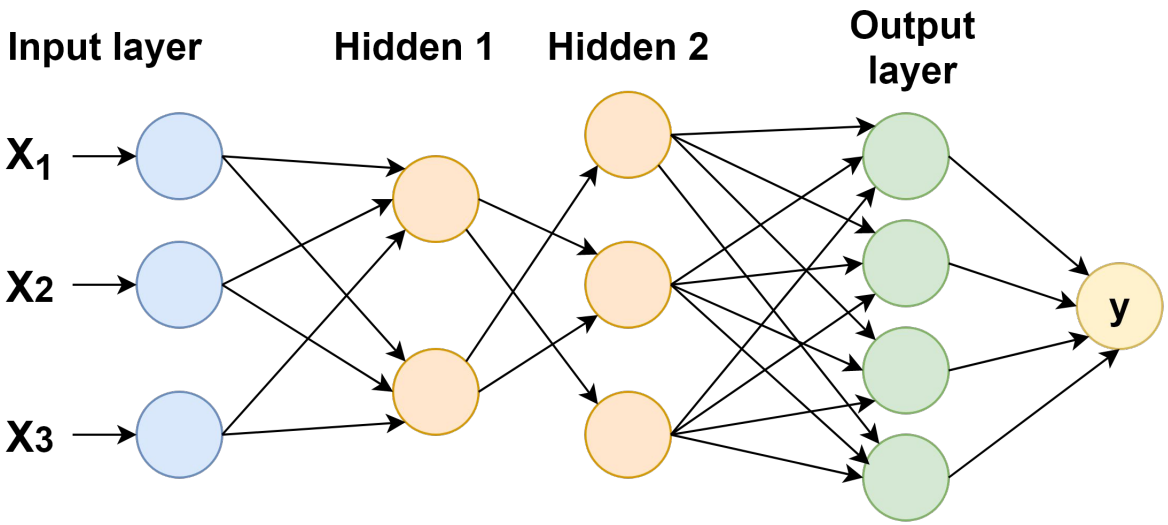
	Train	Dev	Test
Books	659	180	154
Camera	650	170	168
DVD	704	145	163
Health	660	158	168
Music	666	186	175
Software	661	161	172
Total	4,000	1,000	1,000

Today - start of deep learning in NLP

- Assignment feedback
- Recap of Neural Networks
- Recurrent Neural Networks
- LSTMs
- Sequence-to-sequence models (with attention)

Really, this (and next) lecture could be a full course. I will give you highlights, but it might feel like not all details are explained. I will add a reading list on Brightspace.

Last week: multi-layer perceptron



$y = g_2(g_1(W_2(xW_1)))W_3$ ⇨ W_1, W_2 and W_3 are learnable weight matrices

⇨ g_1 and g_2 are non-linear functions

Last week: learning the weights

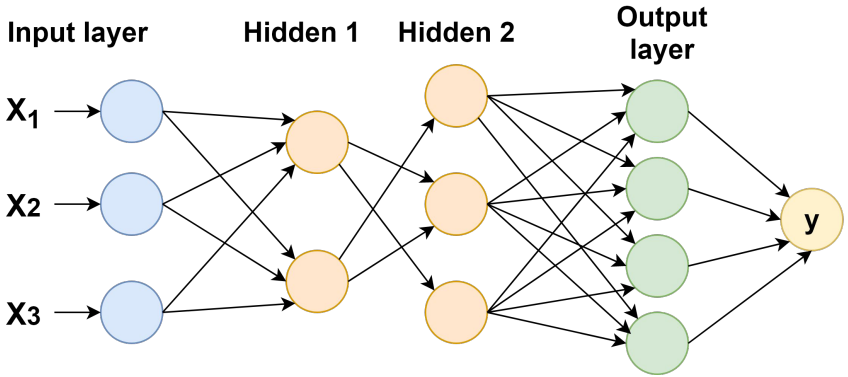
- Changing the weights is how **learning** takes place
- **Intuition:**
  - Pass instance through the network
  - Measure how far off we are (**loss function**)
  - Update weights so that we move in the right direction
  - Loop over instances multiple times until we are happy
  - Save weights as final model so we can predict on unseen data

This **intuition** will stay the same, even though the networks get more complicated.

Last week: creating a neural network

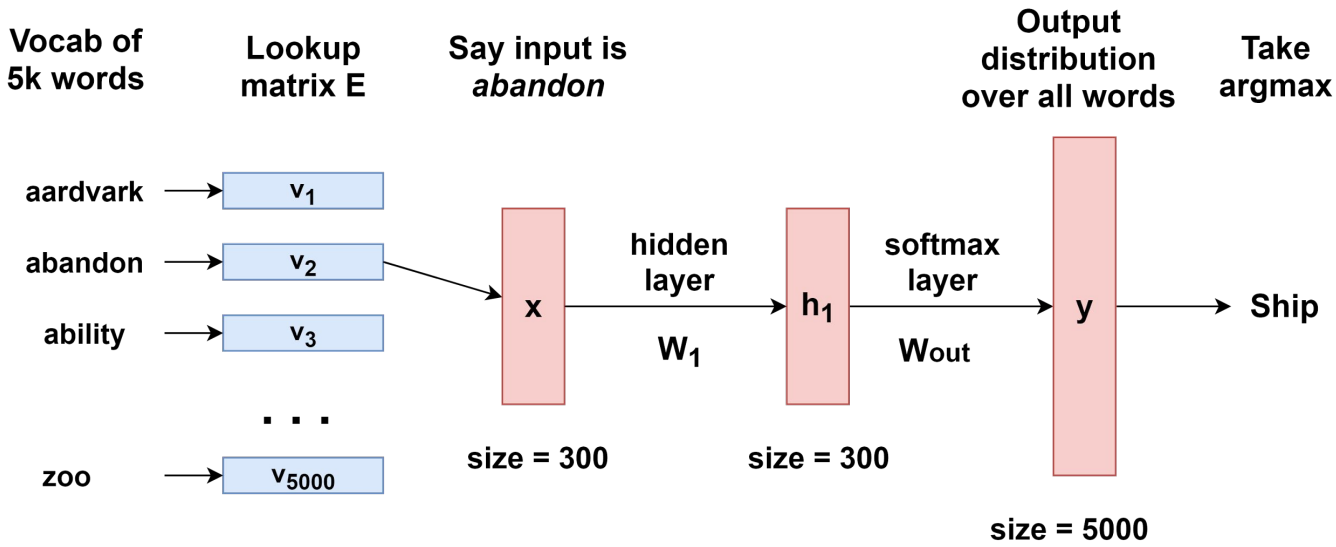
To create a neural net, you have to specify:

- Input features and output categories
- Number of hidden layers
- Activation function (sigmoid, tanh)
- Loss function (MSE, Cross entropy)
- Optimizer (SGD, Adadelta, Adam)
- Where and if to add dropout
- When to stop training



Of course, you don't always know what works in advance. **Experiment!**

Last week: training word embeddings



Each word has an associated **dense vector** that we can use for our own tasks!

Problem

We have a problem: each word has an associated vector, but the input to our neural network also is a single vector. What do we do now?

Problem

We have a problem: each word has an associated vector, but the input to our neural network also is a single vector. What do we do now?

- Concatenating the vectors: does not work for variable input length

Problem

We have a problem: each word has an associated vector, but the input to our neural network also is a single vector. What do we do now?

- Concatenating the vectors: does not work for variable input length
- **Averaging** or **max pooling** vectors
  - Could work, but especially for long sentences we lose a lot of information
  - Also, some words are more important than others

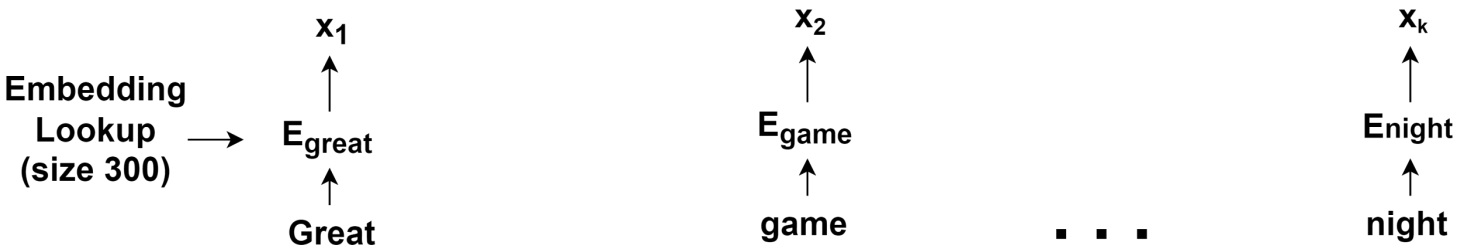
Problem

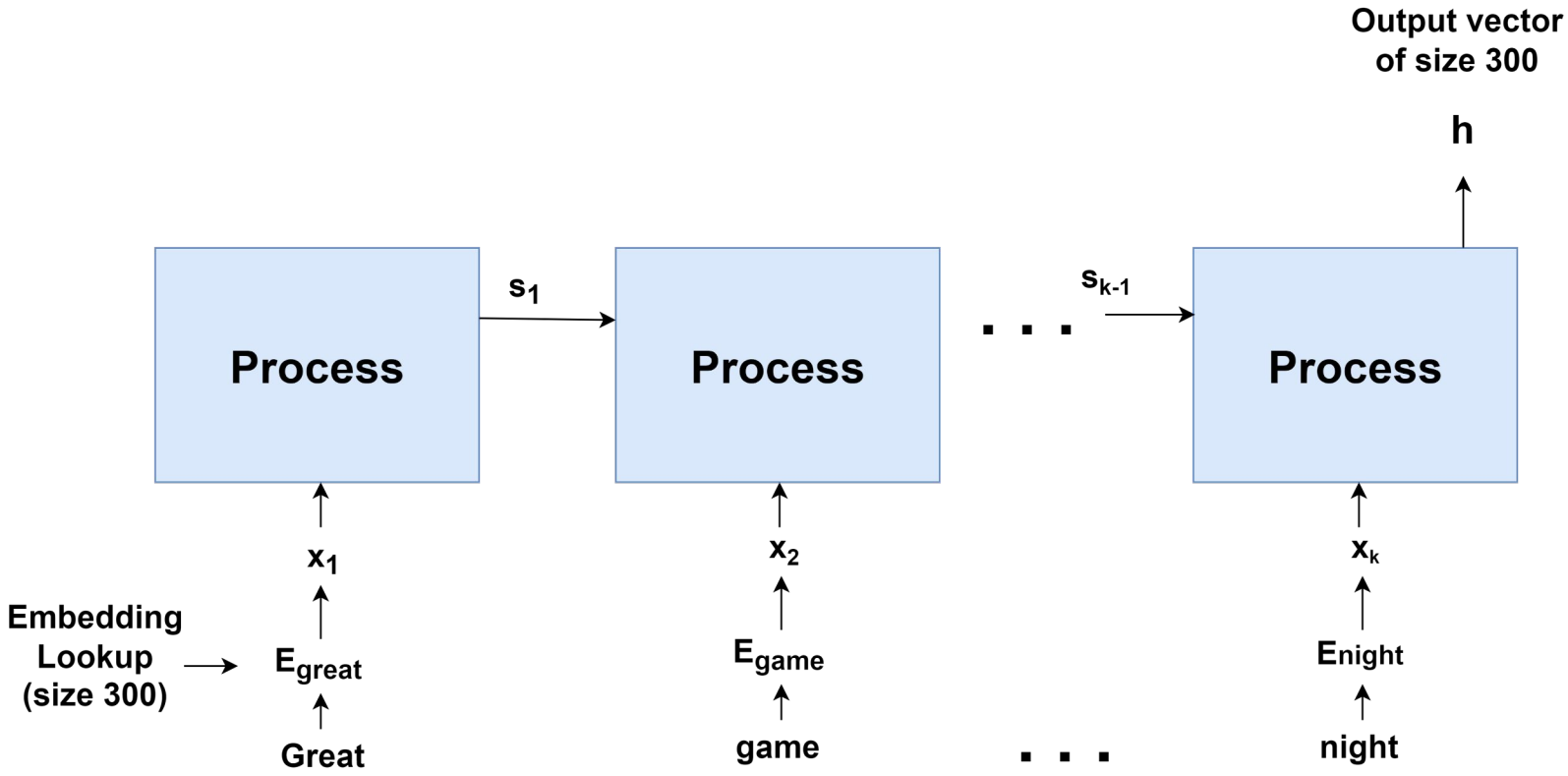
We have a problem: each word has an associated vector, but the input to our neural network also is a single vector. What do we do now?

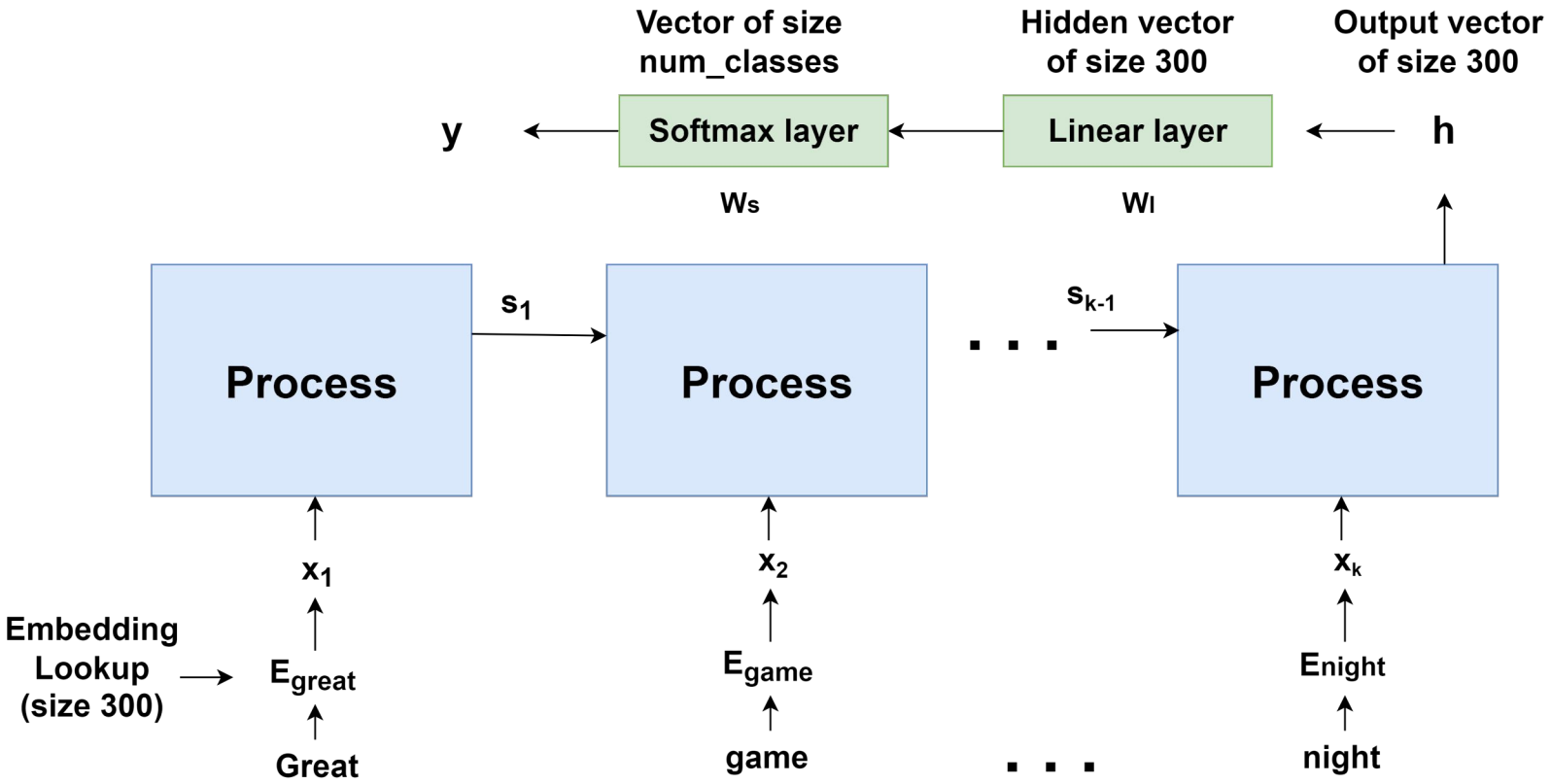
- Concatenating the vectors: does not work for variable input length
- **Averaging** or **max pooling** vectors
  - Could work, but especially for long sentences we lose a lot of information
  - Also, some words are more important than others

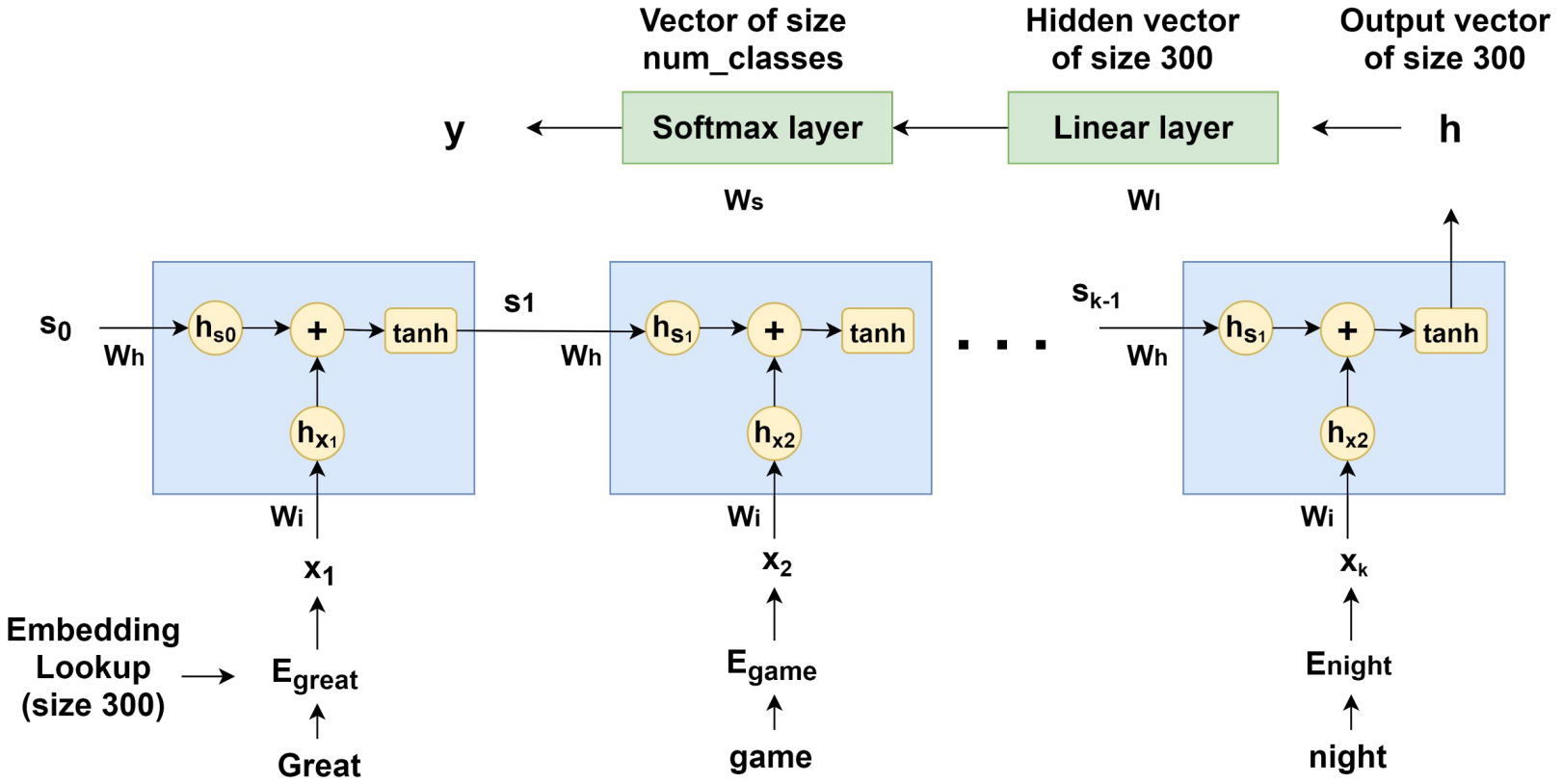
We would like a neural network that can take a **sequence of vectors** as input and automatically **learn** what vectors are important

Recurrent Neural Networks


Embedding
Lookup
(size 300)
x1
Egreat
Great
x2
Egame
game
xk
Enight
night


Output vector
of size 300
h
Process
Process
Process
s_1
s_{k-1}
x_1
x_2
x_k
Embedding
Lookup
(size 300)
E_{great}
E_{game}
E_{night}
Great
game
night


Vector of size num_classes
Hidden vector of size 300
Output vector of size 300
y
Softmax layer
Linear layer
h
Ws
Wl
Process
Process
Process
s1
sk-1
x1
x2
xk
Embedding Lookup (size 300)
Egreat
Great
Egame
game
Enight
night


Vector of size num_classes
Hidden vector of size 300
Output vector of size 300
y
Softmax layer
Linear layer
h
Ws
Wl
s0
Wh
hs0
+
tanh
hx1
Wi
x1
Embedding Lookup (size 300)
Egreat
Great
s1
Wh
hs1
+
tanh
hx2
Wi
x2
Egame
game
. . .
sk-1
Wh
hs1
+
tanh
hx2
Wi
xk
Enight
night

Recurrent Neural Network

- **Recurrent:** current state $\mathbf{s}_i$ is dependent on all previous states **s**
- Each state puts the input vector through a linear layer
- Then these input vectors are added together and put through a tanh

Recurrent Neural Network

- **Recurrent:** current state $\mathbf{s_i}$ is dependent on all previous states **s**
- Each state puts the input vector through a linear layer
- Then these input vectors are added together and put through a tanh

Formula: $\text{RNN}(x_j, s_{j-1}) = \tanh(s_{j-1}W_h + x_jW_i)$

Note the recurrency: to get vector s_{j-1} we have to calculate $\text{RNN}(x_{j-1}, s_{j-2})$!

Recurrent Neural Network

- **Recurrent:** current state $\mathbf{s_i}$ is dependent on all previous states **s**
- Each state puts the input vector through a linear layer
- Then these input vectors are added together and put through a tanh

Formula: $\text{RNN}(x_j, s_{j-1}) = \tanh(s_{j-1}W_h + x_jW_i)$

Note the recurrency: to get vector s_{j-1} we have to calculate $\text{RNN}(x_{j-1}, s_{j-2})$!

Important: Weights $\mathbf{W_h}$ and $\mathbf{W_i}$ are shared across all states of the network

Recurrent Neural Network

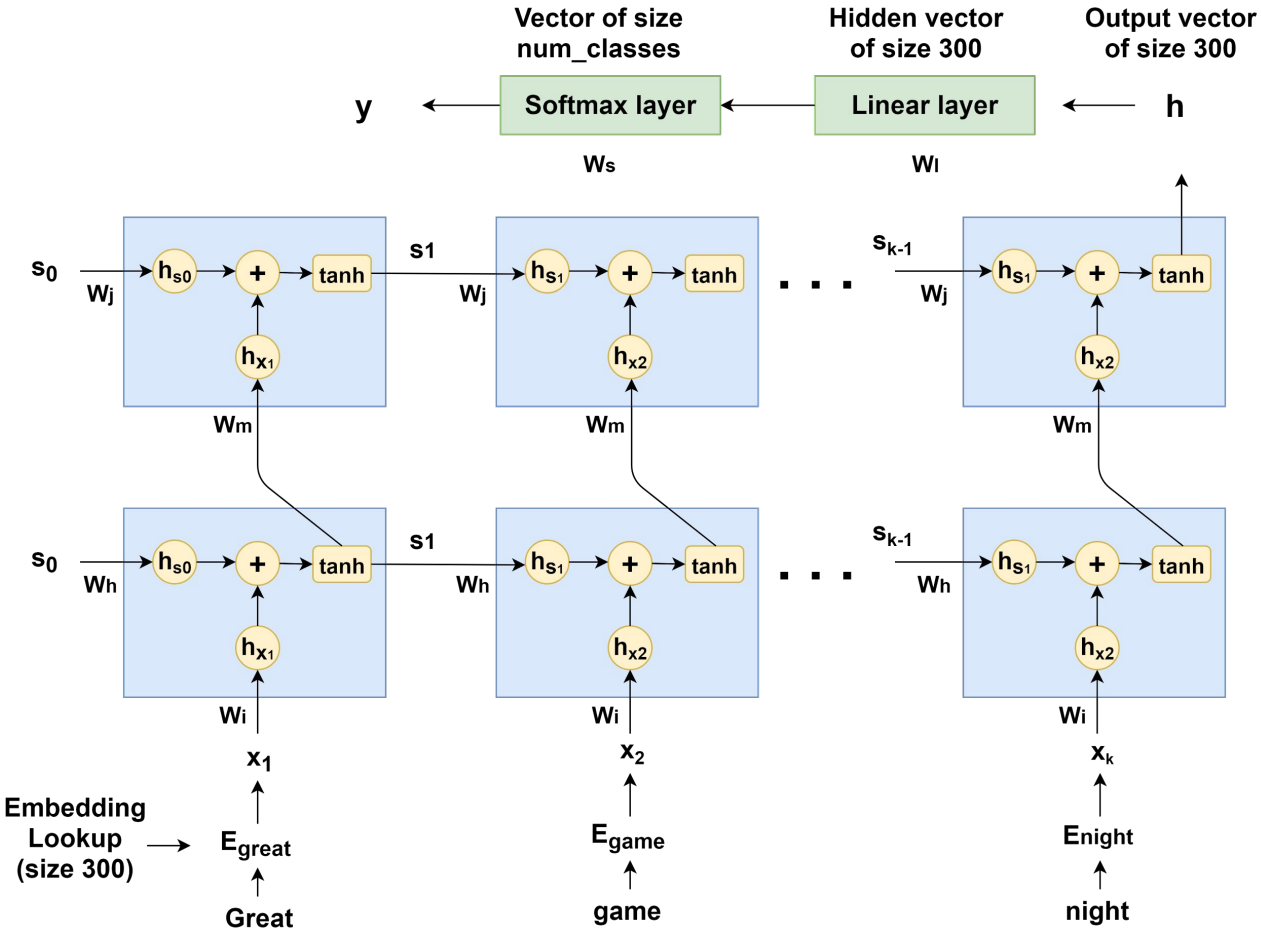
- **Recurrent:** current state $\mathbf{s_i}$ is dependent on all previous states **s**
- Each state puts the input vector through a linear layer
- Then these input vectors are added together and put through a tanh

Formula: $\text{RNN}(x_j, s_{j-1}) = \tanh(s_{j-1}W_h + x_jW_i)$

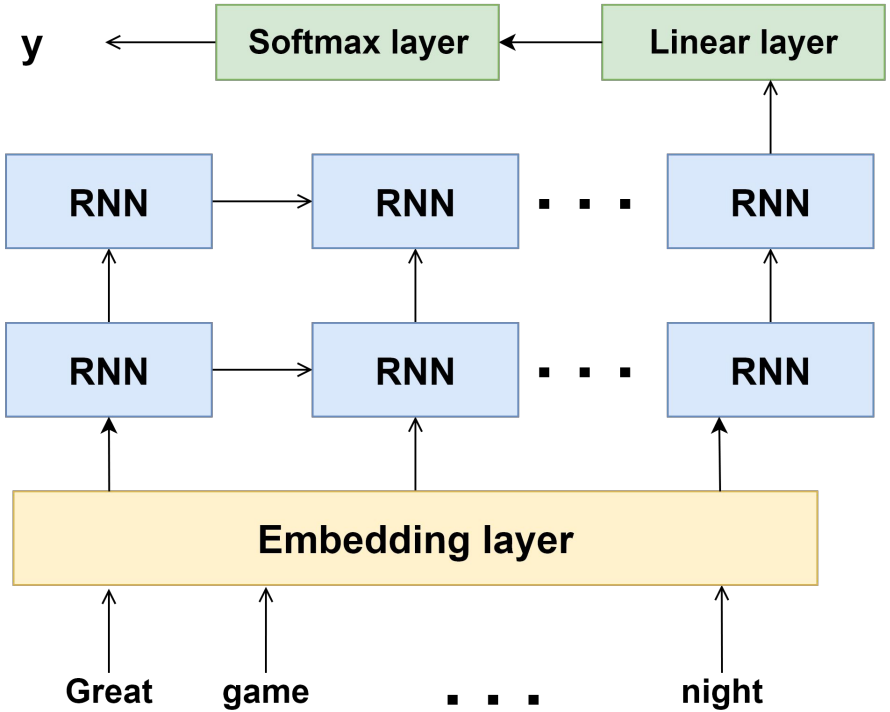
Note the recurrency: to get vector s_{j-1} we have to calculate $\text{RNN}(x_{j-1}, s_{j-2})$!

Important: Weights $\mathbf{W_h}$ and $\mathbf{W_i}$ are shared across all states of the network

It is possible to have **multiple** RNN layers!


Vector of size num_classes
Hidden vector of size 300
Output vector of size 300
y
Softmax layer
Linear layer
h
Ws
Wl
s0
Wj
hs0
+
tanh
hx1
s1
hs1
hx2
sk-1
Wm
Wh
Wi
x1
x2
xk
Embedding Lookup (size 300)
Egreat
Egame
Enight
Great
game
night

Simplified representation



RNN problem

- All information is condensed in a single vector (hidden state)
- The gradient is dependent on a lot of multiplications
- The gradient tends to either get very small, or very small
  - Small: **vanishing gradients**
  - Large: **exploding gradients**

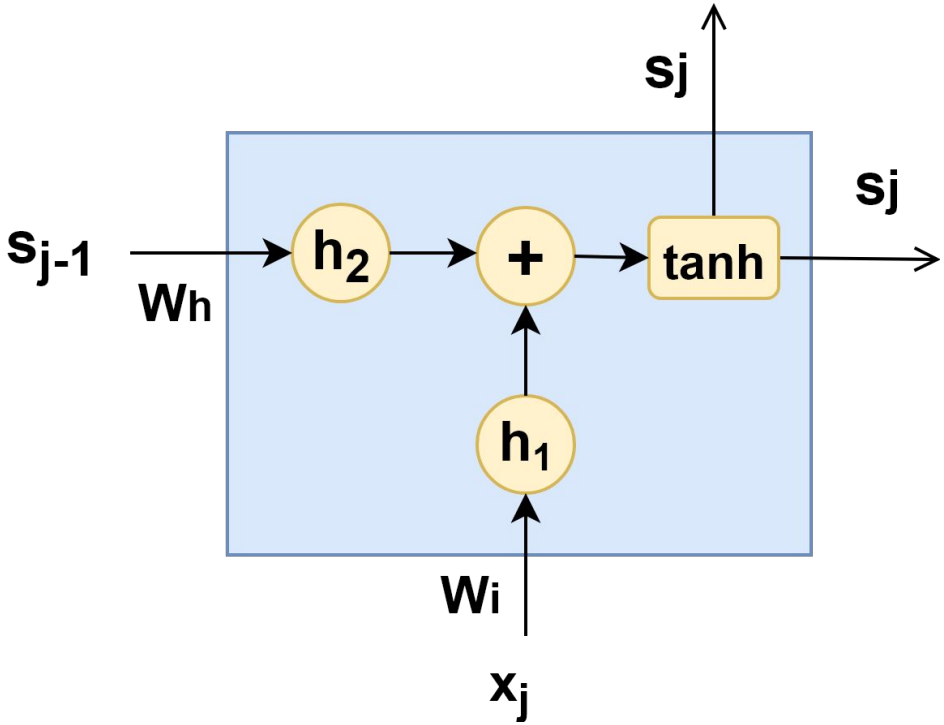
RNN problem

- All information is condensed in a single vector (hidden state)
- The gradient is dependent on a lot of multiplications
- The gradient tends to either get very small, or very small
  - Small: **vanishing gradients**
  - Large: **exploding gradients**

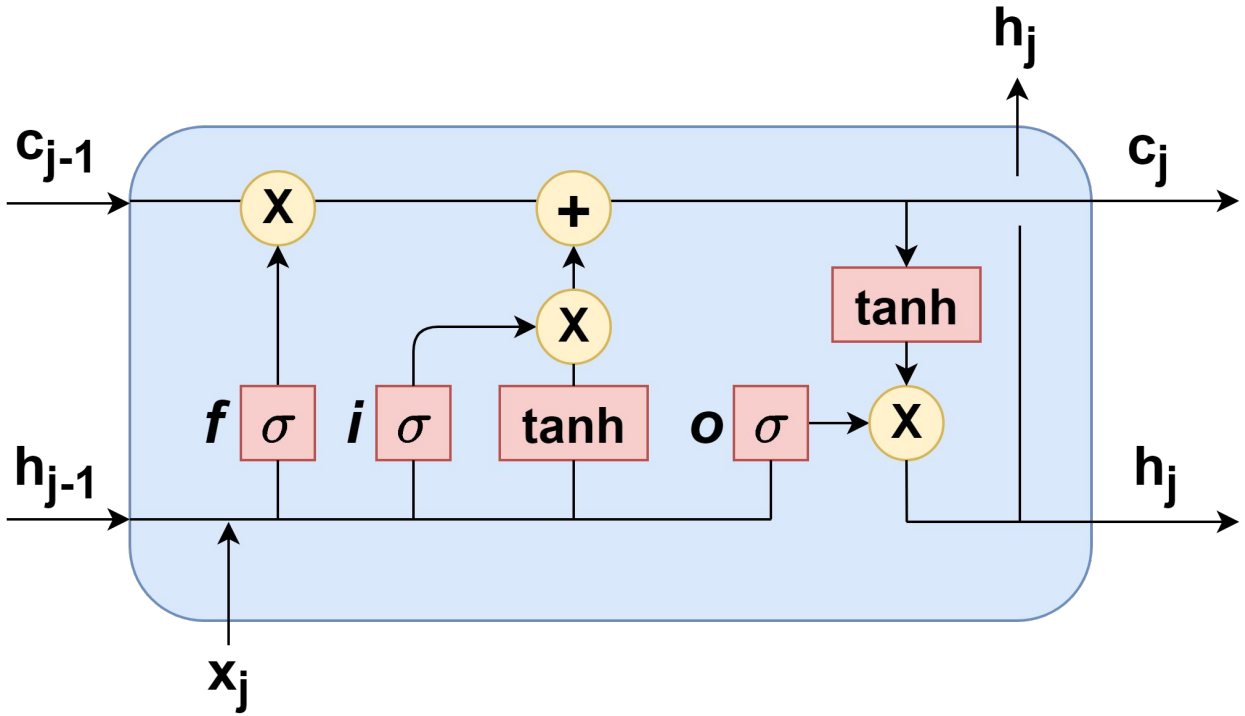
Also: the model has problems learning **long-term dependencies**. Or in other words: at the end of the sequence the model has not retained enough information of the beginning of the sequence.

Model to get around these issues: Long Short Term Memory (LSTM)

Simple RNN cell



LSTM cell

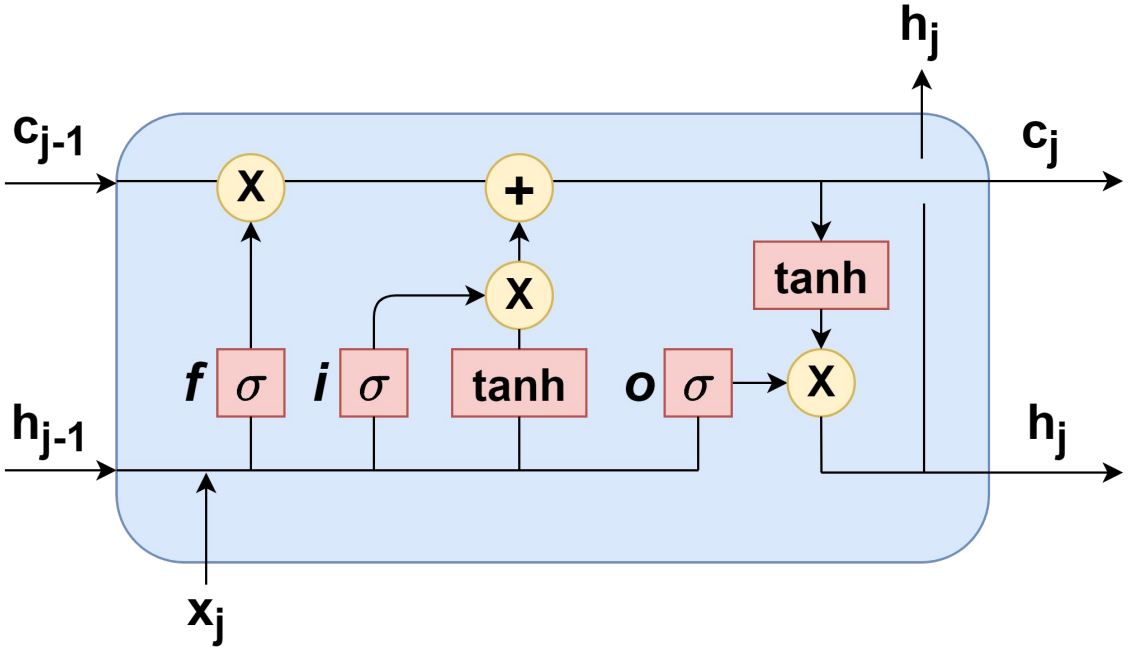


LSTM in formulas

$s_j = \text{LSTM}(s_{j-1}, x_j) = [h_j; c_j]$

$f_j = \sigma(W_f x_j + V_f h_{j-1})$
$i_j = \sigma(W_i x_j + V_i h_{j-1})$
$o_j = \sigma(W_o x_j + V_o h_{j-1})$
$z_j = \tanh(W_z x_j + V_z h_{j-1})$

$c_j = (f_j \odot c_{j-1}) + (i_j \odot z_j)$
$h_j = o_j \odot \tanh(c_j)$



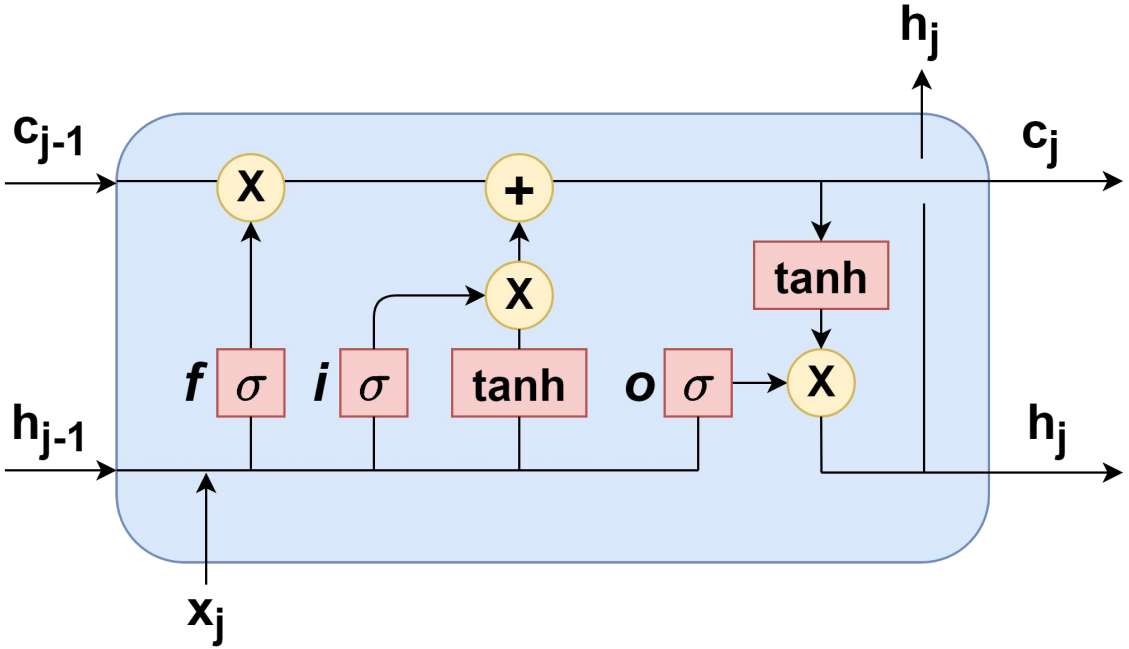
W and **V** are learnable weight matrices

LSTM in formulas

$s_j = \text{LSTM}(s_{j-1}, x_j) = [h_j; c_j]$

$f_j = \sigma(W_f \mathbf{x_j} + V_f \mathbf{h_{j-1}})$
$i_j = \sigma(W_i \mathbf{x_j} + V_i \mathbf{h_{j-1}})$
$o_j = \sigma(W_o \mathbf{x_j} + V_o \mathbf{h_{j-1}})$
$z_j = \tanh(W_z \mathbf{x_j} + V_z \mathbf{h_{j-1}})$

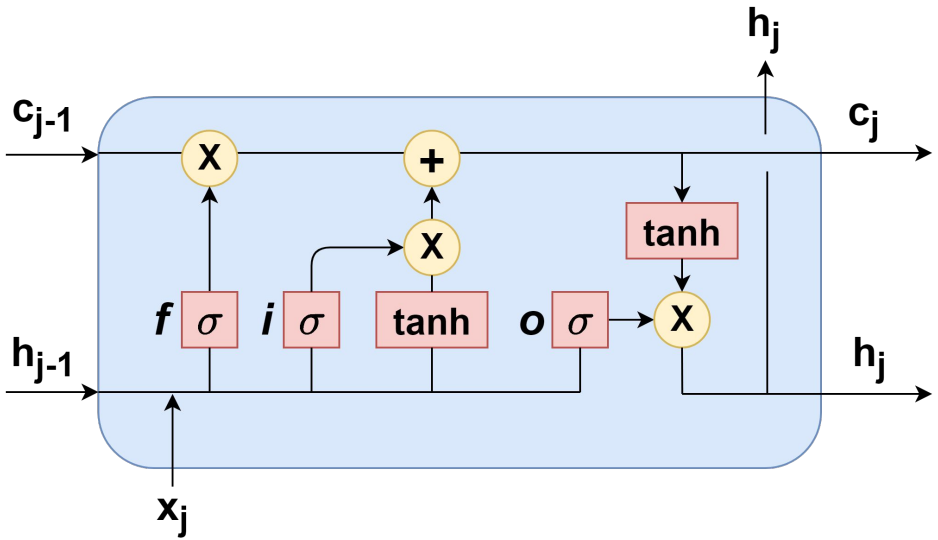
$c_j = (f_j \odot c_{j-1}) + (i_j \odot z_j)$
$h_j = o_j \odot \tanh(c_j)$



W and **V** are learnable weight matrices

LSTM in formulas

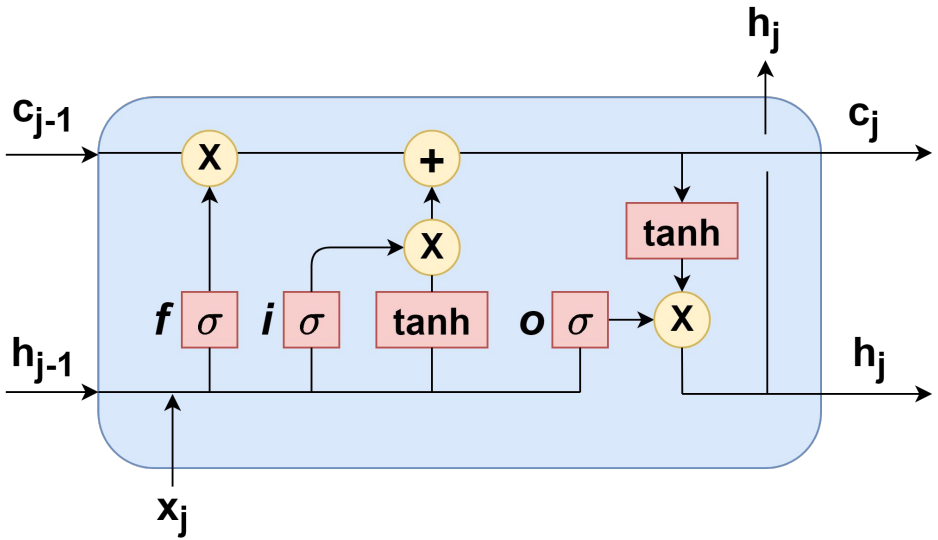
c: **cell state** - intuition is that it is similar to a long term memory



LSTM in formulas

c: **cell state** - intuition is that it is similar to a long term memory

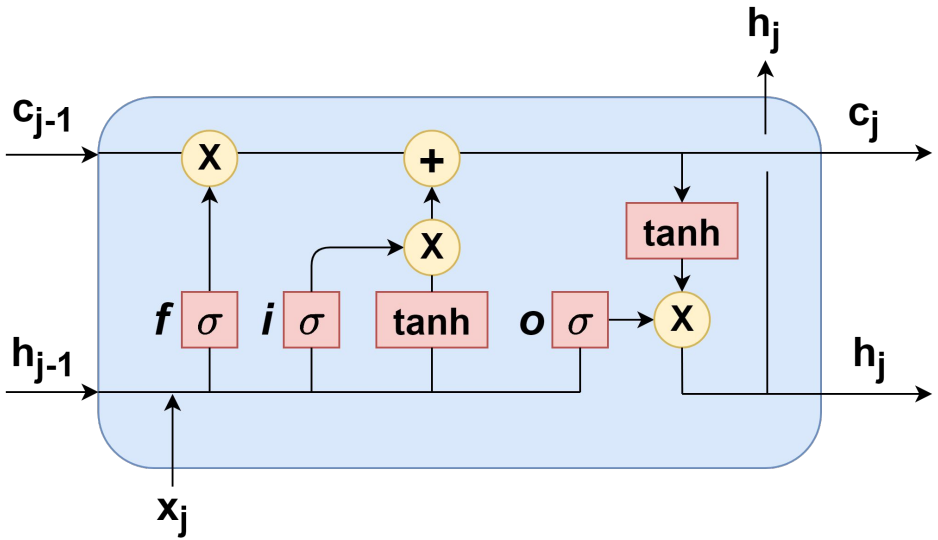
h: **hidden state**: similar to the current working memory



LSTM in formulas

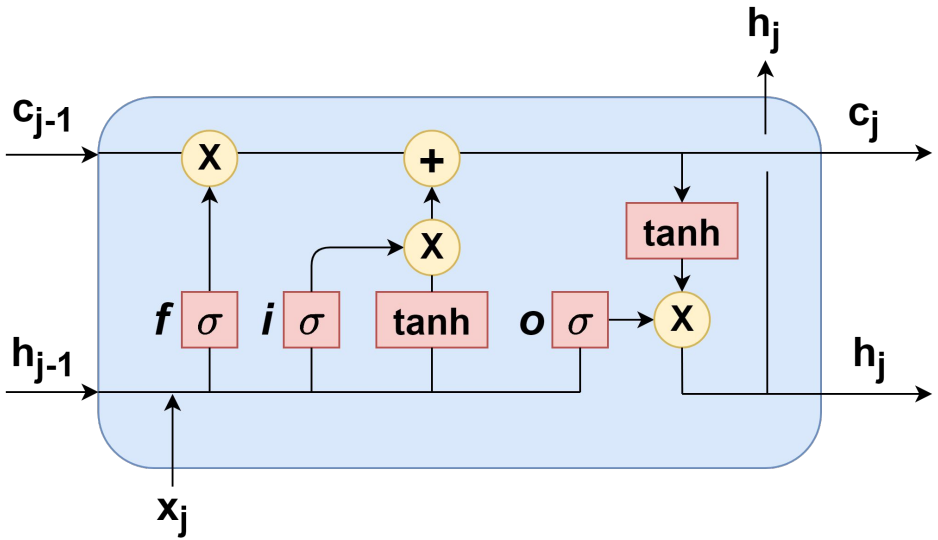
c: **cell state** - intuition is that it is similar to a long term memory
h: **hidden state**: similar to the current working memory

f, **i** and **o** are gates: they control how much information gets added to **c**



LSTM in formulas

c: **cell state** - intuition is that it is similar to a long term memory
h: **hidden state**: similar to the current working memory

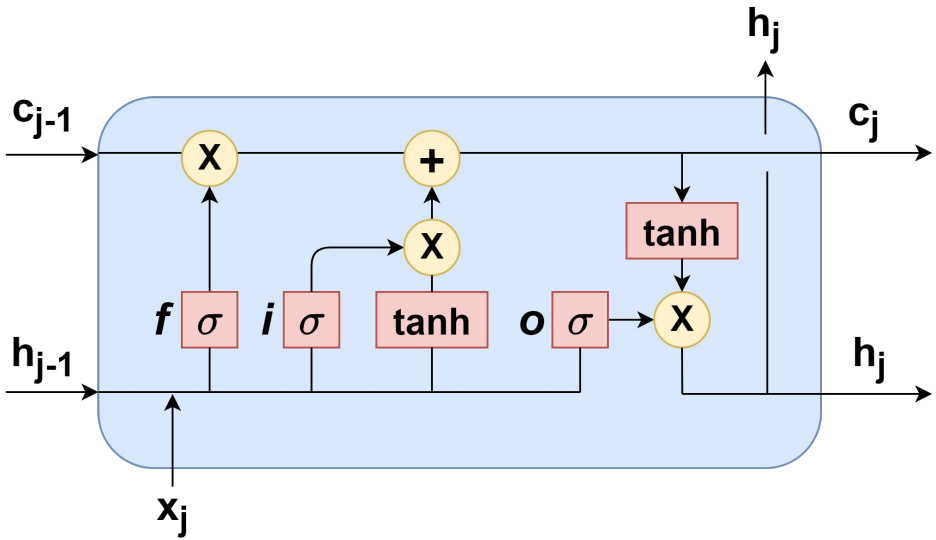


f, **i** and **o** are gates: they control how much information gets added to **c**

f: forget gate - how much of previous state c_{j-1} do we keep?

LSTM in formulas

c: **cell state** - intuition is that it is similar to a long term memory
h: **hidden state**: similar to the current working memory



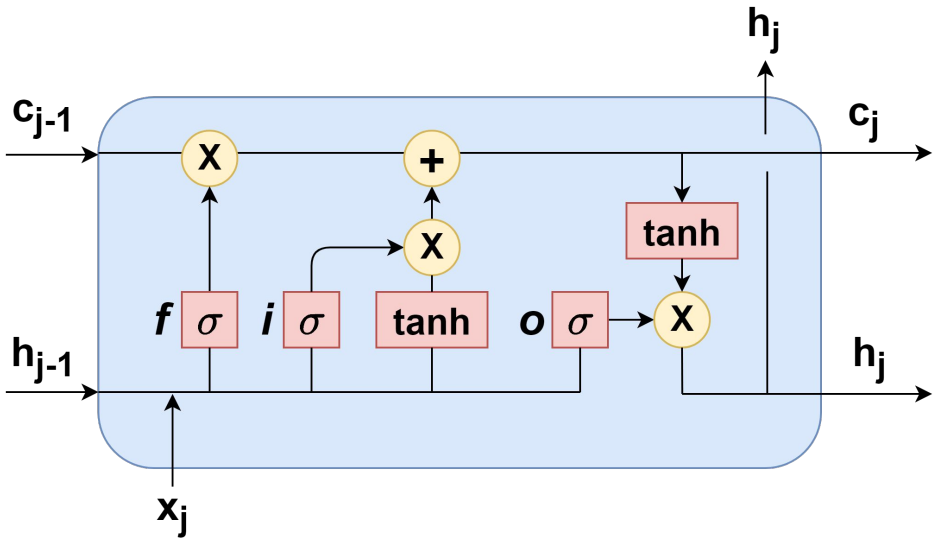
f, **i** and **o** are gates: they control how much information gets added to **c**

f: forget gate - how much of previous state c_{j-1} do we keep?
i: **input gate** - to what extent do we add new information to c_{j-1}?

LSTM in formulas

c: **cell state** - intuition is that it is similar to a long term memory
h: **hidden state**: similar to the current working memory

f, **i** and **o** are gates: they control how much information gets added to **c**



f: forget gate - how much of previous state c_{j-1} do we keep?
i: **input gate** - to what extent do we add new information to c_{j-1}?
o: **output gate** - what do we actually output as our new hidden state h_j?

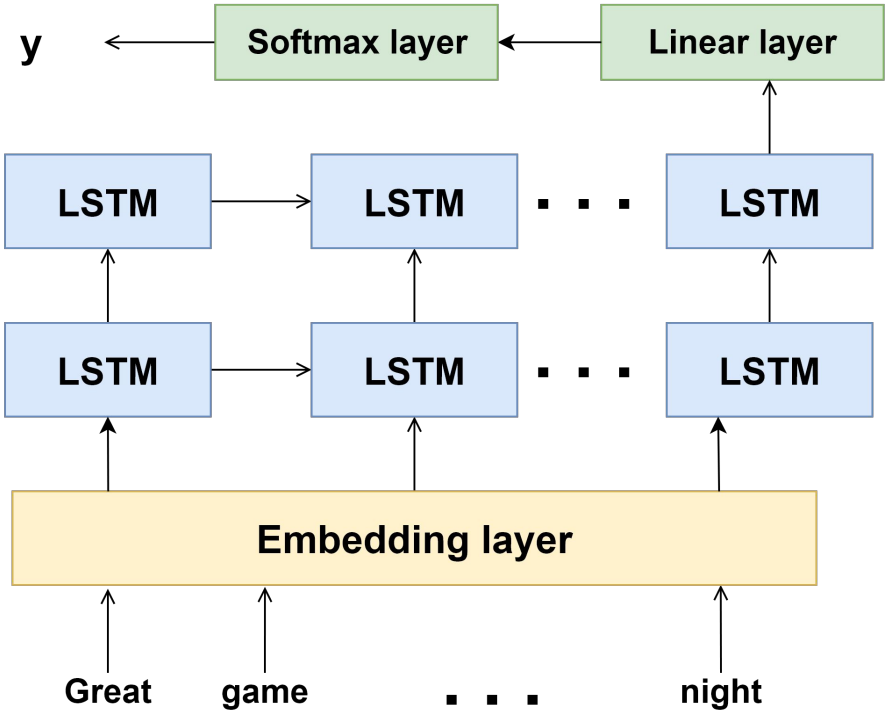
LSTM architecture

This architecture looks a bit random (or ad-hoc), but it's incredible hard to improve on

Jozefowicz et al. (2015) did an architecture search over **thousands** of RNN architectures that were similar to the LSTM architecture. In their own words:

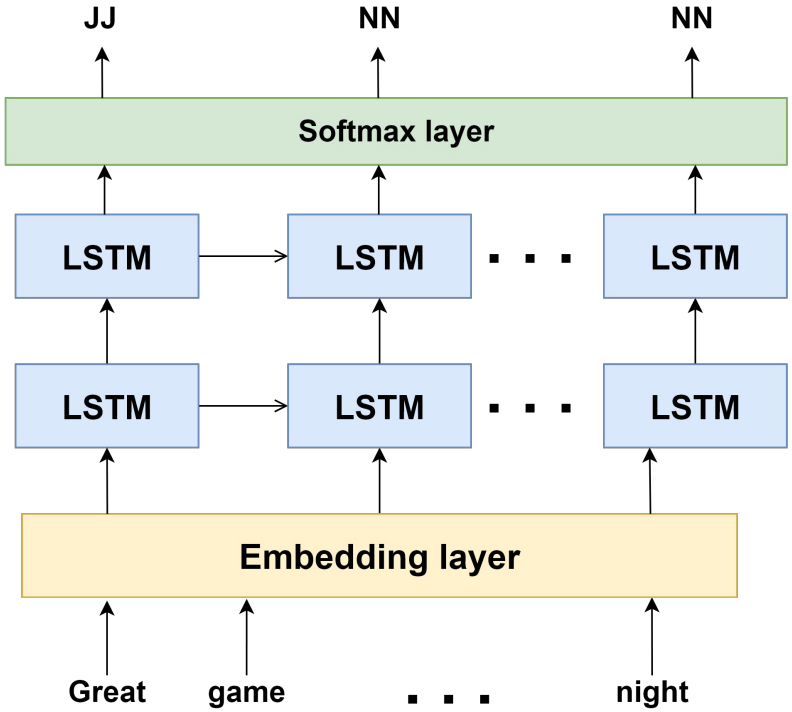
"Though there were architectures that outperformed the LSTM on some problems, we were unable to find an architecture that consistently beat the LSTM [...] in all experimental conditions"

LSTM simplified



What if I want an output for each token?

Sequence Tagging (POS)



LSTM issue

This architecture has a problem: we never look ahead to see what the future words are in the sentence. This can be quite important!

Example sentence: New ...

LSTM issue

This architecture has a problem: we never look ahead to see what the future words are in the sentence. This can be quite important!

Example sentence: New ...

- New players often underperform
- New York is a lovely city

LSTM issue

This architecture has a problem: we never look ahead to see what the future words are in the sentence. This can be quite important!

Example sentence: New ...

- New players often underperform
- New York is a lovely city

Solution:

- A simple, but effective trick: run the LSTM also right-to-left and simply concatenate the resulting state vectors: $h_i = [h_{i\text{-left-right}} ; h_{i\text{-right-left}}]$

LSTM issue

This architecture has a problem: we never look ahead to see what the future words are in the sentence. This can be quite important!

Example sentence: New ...

- New players often underperform
- New York is a lovely city

Solution:

- A simple, but effective trick: run the LSTM also right-to-left and simply concatenate the resulting state vectors: $h_i = [h_{\text{i-left-right}} ; h_{\text{i-right-left}}]$
- The right-to-left part saw **York** already before processing **New**

LSTM issue

This architecture has a problem: we never look ahead to see what the future words are in the sentence. This can be quite important!

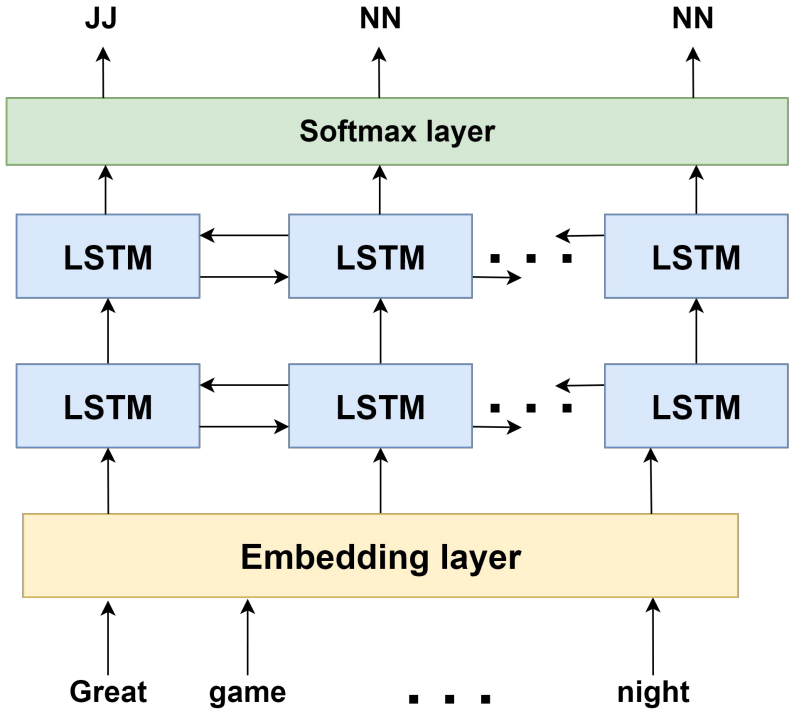
Example sentence: New ...

- New players often underperform
- New York is a lovely city

Solution:

- A simple, but effective trick: run the LSTM also right-to-left and simply concatenate the resulting state vectors: $h_i = [h_{\text{i-left-right}} ; h_{\text{i-right-left}}]$
- The right-to-left part saw **York** already before processing **New**
- Weights are **not shared** between the two states

Bi-directional LSTM (bi-LSTM)



But what if we have output of variable length?
For example machine translation?

Encoder-decoder architecture

Encoder-decoder architecture

- We use the LSTM as an **encoder**: we feed the sentence to the LSTM and use the final states $\mathbf{c}_m$ and $\mathbf{h}_m$ (for simplicity represented as $\mathbf{s}_m$) as encoded representation of the sentence

Encoder-decoder architecture

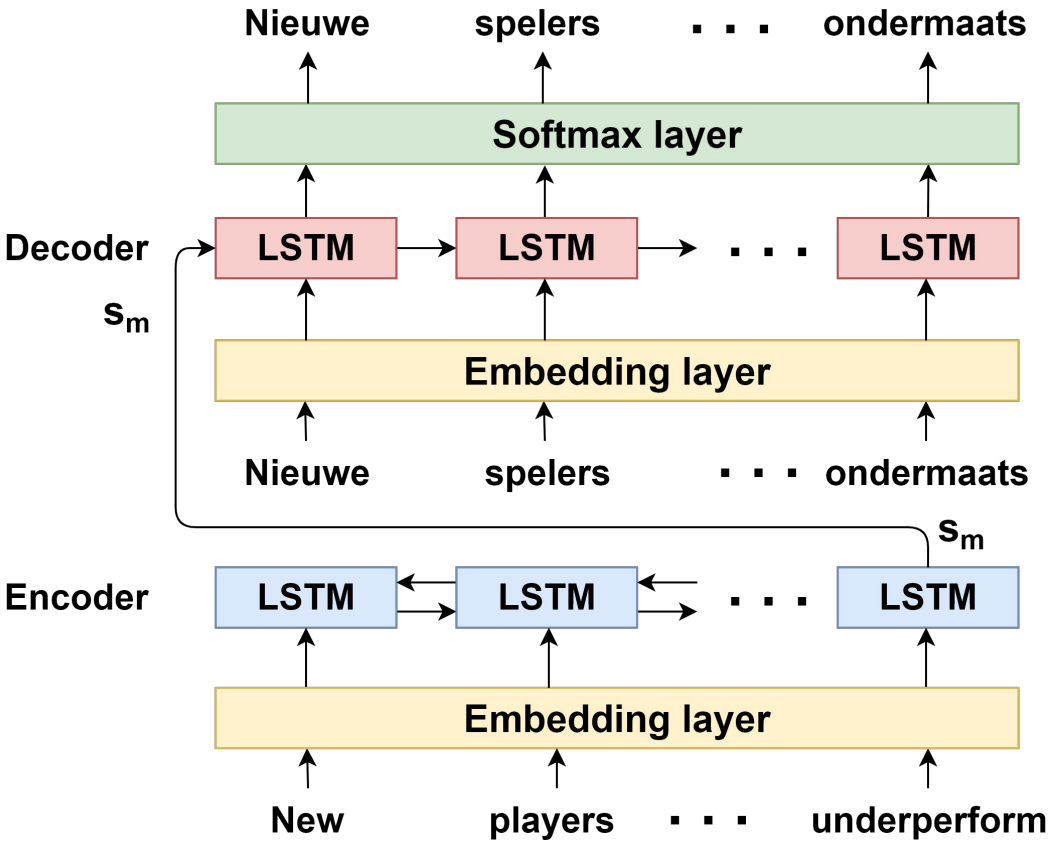
- We use the LSTM as an **encoder**: we feed the sentence to the LSTM and use the final states $\mathbf{c}_m$ and $\mathbf{h}_m$ (for simplicity represented as $\mathbf{s}_m$) as encoded representation of the sentence
- We then train a new LSTM model (**decoder**) that learns to output a sequence of words given the encoded representation $\mathbf{c}_k$ and $\mathbf{h}_k$

Encoder-decoder architecture

- We use the LSTM as an **encoder**: we feed the sentence to the LSTM and use the final states $\mathbf{c}_m$ and $\mathbf{h}_m$ (for simplicity represented as $\mathbf{s}_m$) as encoded representation of the sentence
- We then train a new LSTM model (**decoder**) that learns to output a sequence of words given the encoded representation $\mathbf{c}_k$ and $\mathbf{h}_k$
- As inputs to the decoder, we feed the embeddings of our gold standard output

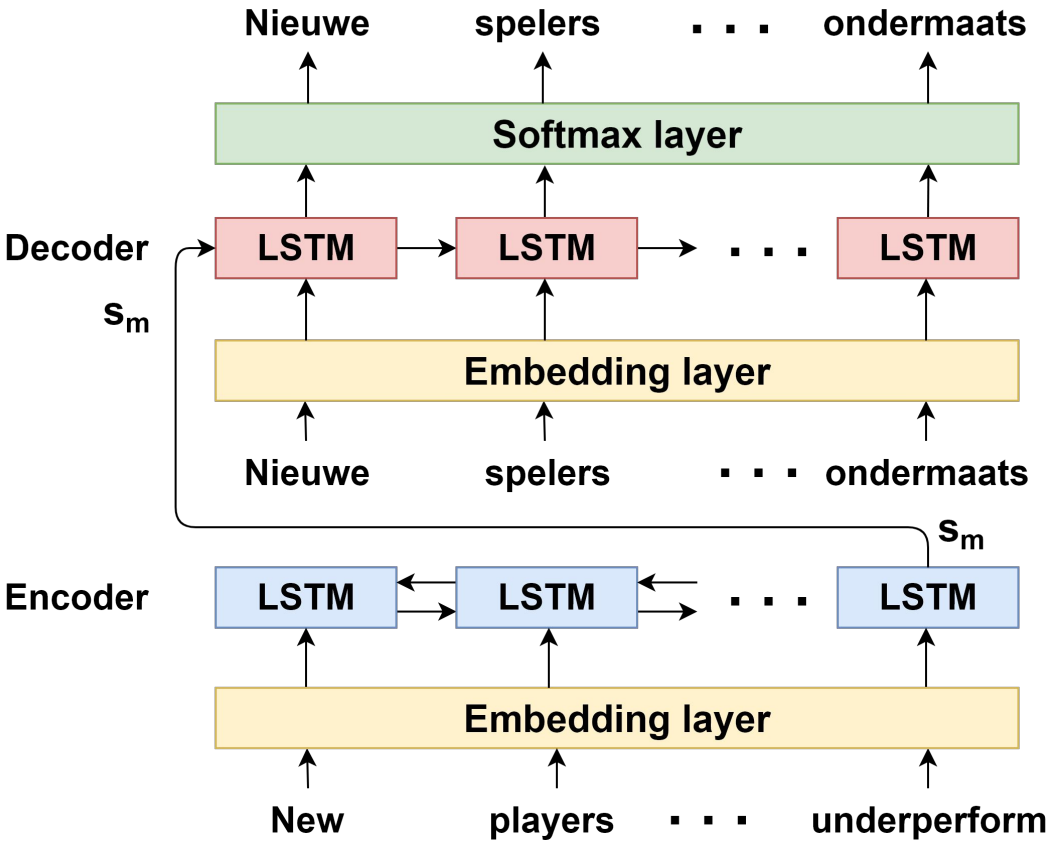
Encoder-decoder architecture

- We use the LSTM as an **encoder**: we feed the sentence to the LSTM and use the final states $\mathbf{c}_m$ and $\mathbf{h}_m$ (for simplicity represented as $\mathbf{s}_m$) as encoded representation of the sentence
- We then train a new LSTM model (**decoder**) that learns to output a sequence of words given the encoded representation $\mathbf{c}_k$ and $\mathbf{h}_k$
- As inputs to the decoder, we feed the embeddings of our gold standard output
- The decoder outputs a word for each state, which we can compare to our gold standard output (and backpropagate errors ⇒ learning)


Nieuwe
spelers
. . .
ondermaats
Softmax layer
Decoder
LSTM
LSTM
. . .
LSTM
s_m
Embedding layer
Nieuwe
spelers
. . .
ondermaats
s_m
Encoder
LSTM
LSTM
. . .
LSTM
Embedding layer
New
players
. . .
underperform

What is the problem here?

This does not work for prediction!


Nieuwe
spelers
. . .
ondermaats
Softmax layer
Decoder
LSTM
LSTM
. . .
LSTM
s_m
Embedding layer
Nieuwe
spelers
. . .
ondermaats
s_m
Encoder
LSTM
LSTM
. . .
LSTM
Embedding layer
New
players
. . .
underperform

Decoder prediction

Problems during prediction:

- We have no target embedding inputs
- We don’t know when to **stop** outputting words

Decoder prediction

Problems during prediction:

- We have no target embedding inputs
- We don’t know when to **stop** outputting words

Solution:

- Shift the output: add special tokens for beginning and end of sequence
  - <BOS> and <EOS>

Decoder prediction

Problems during prediction:

- We have no target embedding inputs
- We don’t know when to **stop** outputting words

Solution:

- Shift the output: add special tokens for beginning and end of sequence
  - <BOS> and <EOS>
- Input for output token $\mathbf{t_j}$ is actually the vector of token $\mathbf{t_{j-1}}$

Decoder prediction

Problems during prediction:

- We have no target embedding inputs
- We don’t know when to **stop** outputting words

Solution:

- Shift the output: add special tokens for beginning and end of sequence
  - <BOS> and <EOS>
- Input for output token $\mathbf{t_j}$ is actually the vector of token $\mathbf{t_{j-1}}$
- During prediction, feed target vector of the predicted token $\mathbf{y_{t^{j-1}}}$

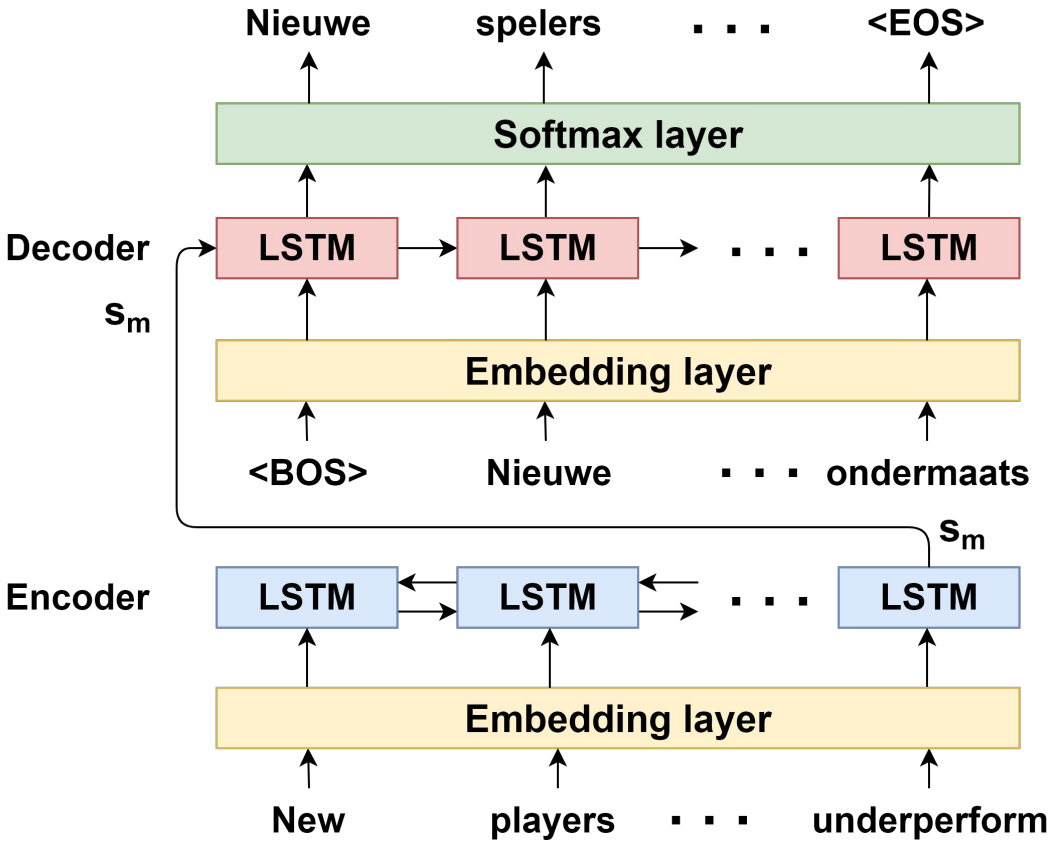
Decoder prediction

Problems during prediction:

- We have no target embedding inputs
- We don't know when to **stop** outputting words

Solution:

- Shift the output: add special tokens for beginning and end of sequence
  - <BOS> and <EOS>
- Input for output token $\mathbf{t_j}$ is actually the vector of token $\mathbf{t_{j-1}}$
- During prediction, feed target vector of the predicted token $\mathbf{y_{t^{j-1}}}$
- Once we predict <EOS>, we are **done predicting**


Nieuwe
spelers
. . .
<EOS>
Softmax layer
Decoder
LSTM
LSTM
. . .
LSTM
s_m
Embedding layer
<BOS>
Nieuwe
. . .
ondermaats
s_m
Encoder
LSTM
LSTM
. . .
LSTM
Embedding layer
New
players
. . .
underperform

These models are called sequence-to-sequence models

Decoder output

Currently, we always output the **most probable** token at each step in the decoder

However, say that for the first token we want to output we have two good options:

- p(umpires) = 0.45
- p(referees) = 0.43

Decoder output

Currently, we always output the **most probable** token at each step in the decoder

However, say that for the first token we want to output we have two good options:

- p(umpires) = 0.45
- p(referees) = 0.43

They are so close in probability that we don’t want to make a hard decision just yet!

Decoder output

Currently, we always output the **most probable** token at each step in the decoder

However, say that for the first token we want to output we have two good options:

- p(umpires) = 0.45
- p(referees) = 0.43

They are so close in probability that we don't want to make a hard decision just yet!

Preferably, we keep **both** in memory and see what we would have output given those choices. Ultimately, we can take the **sequence** that was most probable.

Decoder output

Currently, we always output the **most probable** token at each step in the decoder

However, say that for the first token we want to output we have two good options:

- p(umpires) = 0.45
- p(referees) = 0.43

They are so close in probability that we don't want to make a hard decision just yet!

Preferably, we keep **both** in memory and see what we would have output given those choices. Ultimately, we can take the **sequence** that was most probable.

This process is called **beam search**

Beam search

Problem:

Keeping all good hypotheses grows exponentially: if at each step we keep the 3 most probable tokens, we have close to 60,000 (3^{10}) hypotheses at timestep 10

Beam search

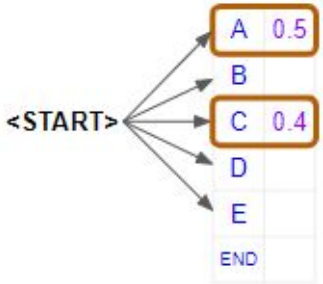
Problem:

Keeping all good hypotheses grows exponentially: if at each step we keep the 3 most probable tokens, we have close to 60,000 (3^{10}) hypotheses at timestep 10

Solution:

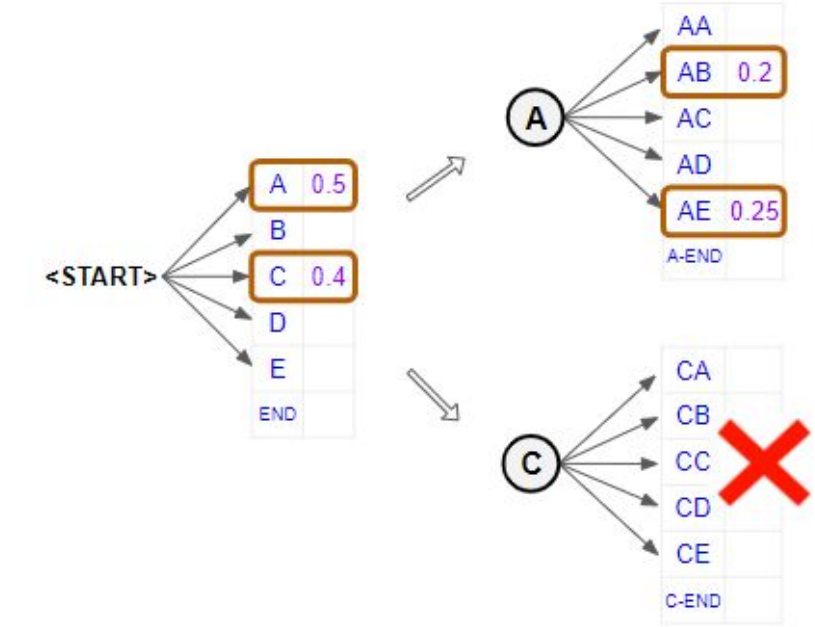
- Specify a beam size **b**: number of hypotheses we keep in memory
- At each step, prune the least likely hypotheses to keep only **b**
- Do this by simply multiplying probabilities of the current path
- Finally, select the path with the highest probability
    - Note that we do normalize by length as not to punish longer sequences

Let’s do an example
with a beam size of 2



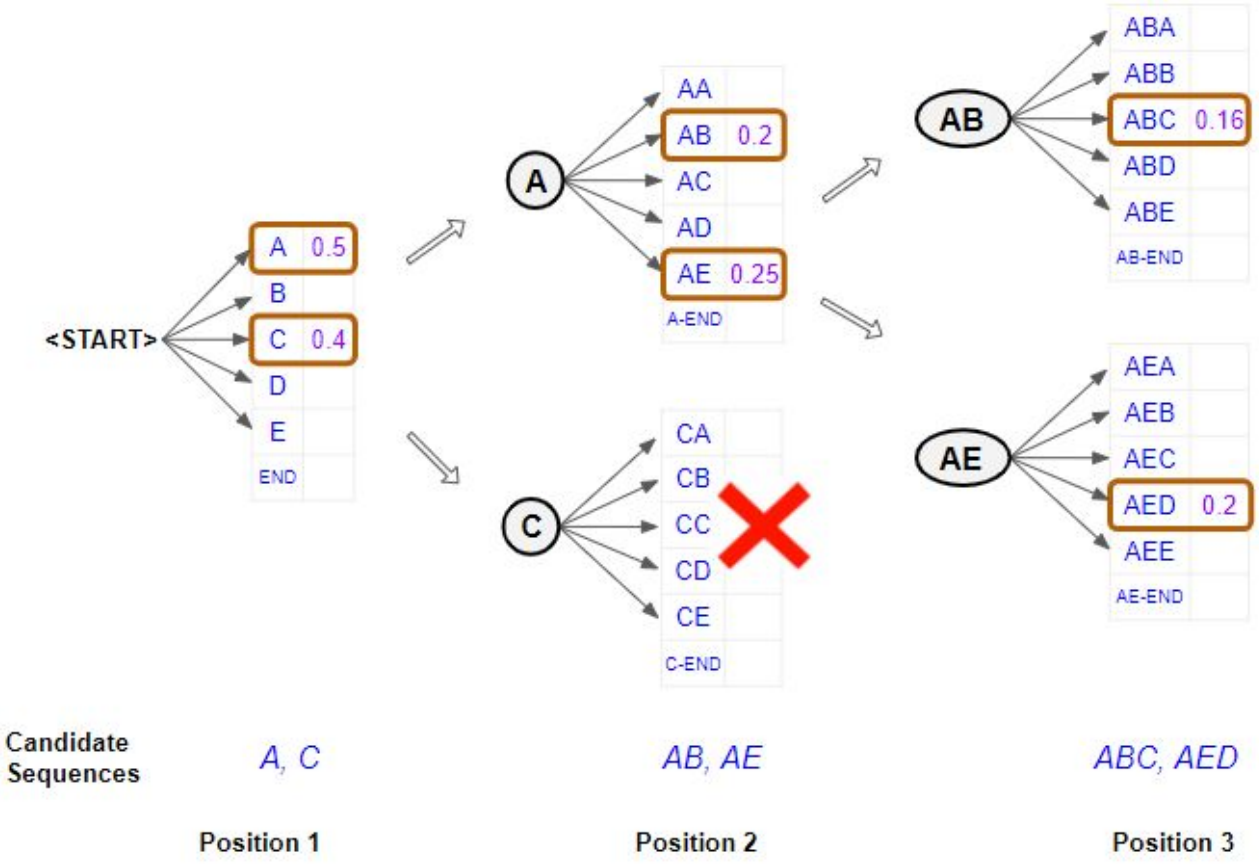
Candidate Sequences *A, C*

Position 1



Candidate Sequences *A, C* *AB, AE*

Position 1 **Position 2**


<START>
A 0.5
B
C 0.4
D
E
END
A
AA
AB 0.2
AC
AD
AE 0.25
A-END
C
CA
CB
CC
CD
CE
C-END
AB
ABA
ABB
ABC 0.16
ABD
ABE
AB-END
AE
AEA
AEB
AEC
AED 0.2
AEE
AE-END
Candidate Sequences
A, C
AB, AE
ABC, AED
Position 1
Position 2
Position 3

Problem: encoder

- But there's again a problem with this setup: all information of the input sentence has to be encoded in a **single vector**

Problem: encoder

- But there's again a problem with this setup: all information of the input sentence has to be encoded in a **single vector**
- Especially for longer sentences, how can we ever encode all relevant information **and** be able to extract this in the decoder?

Problem: encoder

- But there's again a problem with this setup: all information of the input sentence has to be encoded in a **single vector**
- Especially for longer sentences, how can we ever encode all relevant information **and** be able to extract this in the decoder?

Famous quote by Ray Mooney: you can't cram the meaning of whole f*cking sentence in a single f*cking vector

Attention

- Very effective solution: **attention**

Attention

- Very effective solution: **attention**
- **Intuition:** each decoder state has access to **all** encoder states

Attention

- Very effective solution: **attention**
- **Intuition:** each decoder state has access to **all** encoder states
- We learn which encoder states are most important (for the current state) and use this to construct our decoder state vector

Attention

- Very effective solution: **attention**
- **Intuition:** each decoder state has access to **all** encoder states
- We learn which encoder states are most important (for the current state) and use this to construct our decoder state vector
- We pay **attention** to the encoder states that are relevant for the current state

Intuition: get a single vector based on all encoder states that takes into account what encoder states are most relevant for the current decoder state by taking a weighted average of them

Simplified process

Say I have three vectors for my input words:

- v_1= [4, 10, 15], v_2 = [15, 6, 9], v_3 = [12, 5, 3]

Simplified process

Say I have three vectors for my input words:

- v_1= [4, 10, 15], v_2 = [15, 6, 9], v_3 = [12, 5, 3]
- I “pay attention” to v_1 for 50%, v_2 for 30% and v_3 for 20%

Simplified process

Say I have three vectors for my input words:

- v_1= [4, 10, 15], v_2 = [15, 6, 9], v_3 = [12, 5, 3]
- I “pay attention” to v_1 for 50%, v_2 for 30% and v_3 for 20%

Weighted vector:

- First element: 0.5 * 4 + 0.3 * 15 + 0.2 * 12 = 8.9

Simplified process

Say I have three vectors for my input words:

- v_1= [4, 10, 15], v_2 = [15, 6, 9], v_3 = [12, 5, 3]
- I “pay attention” to v_1 for **50**%, v_2 for **30**% and v_3 for **20**%

Weighted vector:

- First element: **0.5** * 4 + **0.3** * 15 + **0.2** * 12 **=** 8.9

Simplified process

Say I have three vectors for my input words:

- v_1= [**4**, 10, 15], v_2 = [**15**, 6, 9], v_3 = [**12**, 5, 3]
- I “pay attention” to v_1 for **50**%, v_2 for **30**% and v_3 for **20**%

Weighted vector:

- First element: **0.5** * **4** + **0.3** * **15** + **0.2** * **12** = 8.9

Simplified process

Say I have three vectors for my input words:

- v_1= [**4**, **10**, 15], v_2 = [**15**, **6**, 9], v_3 = [**12**, **5**, 3]
- I “pay attention” to v_1 for **50**%, v_2 for **30**% and v_3 for **20**%

Weighted vector:

- First element: **0.5** * **4** + **0.3** * **15** + **0.2** * **12** = 8.9
- Second element: **0.5** * **10** + **0.3** * **6** + **0.2** * **5** = 7.8

Simplified process

Say I have three vectors for my input words:

- v_1= [**4**, **10**, **15**], v_2 = [**15**, **6**, **9**], v_3 = [**12**, **5**, **3**]
- I “pay attention” to v_1 for **50**%, v_2 for **30**% and v_3 for **20**%

Weighted vector:

- First element: **0.5** * **4** + **0.3** * **15** + **0.2** * **12** = 8.9
- Second element: **0.5** * **10** + **0.3** * **6** + **0.2** * **5** = 7.8
- Third element: **0.5** * **15** + **0.3** * **9** + **0.2** * **3** = 10.8

Simplified process

Say I have three vectors for my input words:

- v_1= [**4**, **10**, **15**], v_2 = [**15**, **6**, **9**], v_3 = [**12**, **5**, **3**]
- I “pay attention” to v_1 for **50**%, v_2 for **30**% and v_3 for **20**%

Weighted vector:

- First element: **0.5** * **4** + **0.3** * **15** + **0.2** * **12** = 8.9
- Second element: **0.5** * **10** + **0.3** * **6** + **0.2** * **5** = 7.8
- Third element: **0.5** * **15** + **0.3** * **9** + **0.2** * **3** = 10.8

Final weighted vector: [8.9, 7.8, 10.8]

Attention process

- Attention input: a decoder state $\mathbf{h_t}$ and all encoder states $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$

Attention process

- Attention input: a decoder state $\mathbf{h_t}$ and all encoder states $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$
- For each encoder state $\mathbf{s_k}$ compute the relevance of it to $\mathbf{h_t}$
  - We apply an attention function that returns a scalar: score($\mathbf{s_k}$, $\mathbf{h_t}$)

Attention process

- Attention input: a decoder state $\mathbf{h_t}$ and all encoder states $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$
- For each encoder state $\mathbf{s_k}$ compute the relevance of it to $\mathbf{h_t}$
  - We apply an attention function that returns a scalar: score($\mathbf{s_k}$, $\mathbf{h_t}$)
- This gives us a vector (scores for $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$) on which we apply a **softmax**

Attention process

- Attention input: a decoder state $\mathbf{h_t}$ and all encoder states $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$
- For each encoder state $\mathbf{s_k}$ compute the relevance of it to $\mathbf{h_t}$
  - We apply an attention function that returns a scalar: score($\mathbf{s_k}$, $\mathbf{h_t}$)
- This gives us a vector (scores for $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$) on which we apply a **softmax**
- Now we have a vector that tells us the importance of each state **s**

Attention process

- Attention input: a decoder state $\mathbf{h_t}$ and all encoder states $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$
- For each encoder state $\mathbf{s_k}$ compute the relevance of it to $\mathbf{h_t}$
  - We apply an attention function that returns a scalar: score($\mathbf{s_k}$, $\mathbf{h_t}$)
- This gives us a vector (scores for $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$) on which we apply a **softmax**
- Now we have a vector that tells us the importance of each state **s**
- We use this vector to weigh all encoder states and add them together

Attention process

- Attention input: a decoder state $\mathbf{h_t}$ and all encoder states $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$
- For each encoder state $\mathbf{s_k}$ compute the relevance of it to $\mathbf{h_t}$
  - We apply an attention function that returns a scalar: score($\mathbf{s_k}$, $\mathbf{h_t}$)
- This gives us a vector (scores for $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$) on which we apply a **softmax**
- Now we have a vector that tells us the importance of each state **s**
- We use this vector to weigh all encoder states and add them together
- Result: an **attention vector $\mathbf{a_t}$** based on all encoder states

Attention process

- Attention input: a decoder state $\mathbf{h_t}$ and all encoder states $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$
- For each encoder state $\mathbf{s_k}$ compute the relevance of it to $\mathbf{h_t}$
  - We apply an attention function that returns a scalar: score($\mathbf{s_k}$, $\mathbf{h_t}$)
- This gives us a vector (scores for $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$) on which we apply a **softmax**
- Now we have a vector that tells us the importance of each state **s**
- We use this vector to weigh all encoder states and add them together
- Result: an **attention vector $\mathbf{a_t}$** based on all encoder states
- Concatenate this vector to the current hidden state vector $\mathbf{h_t}$

Attention process

- Attention input: a decoder state $\mathbf{h_t}$ and all encoder states $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$
- For each encoder state $\mathbf{s_k}$ compute the relevance of it to $\mathbf{h_t}$
  - We apply an attention function that returns a scalar: score($\mathbf{s_k}$, $\mathbf{h_t}$)
- This gives us a vector (scores for $\mathbf{s_1}$, $\mathbf{s_2}$, …, $\mathbf{s_m}$) on which we apply a **softmax**
- Now we have a vector that tells us the importance of each state **s**
- We use this vector to weigh all encoder states and add them together
- Result: an **attention vector $\mathbf{a_t}$** based on all encoder states
- Concatenate this vector to the current hidden state vector $\mathbf{h_t}$
- Continue the process as before

Intuition: get a single vector based on all encoder states that takes into account what encoder states are most relevant for the current decoder state by taking a weighted average of them

Attention calculations

Attention output

$$c^{(t)} = a_1^{(t)} s_1 + a_2^{(t)} s_2 + \cdots + a_m^{(t)} s_m = \sum_{k=1}^{m} a_k^{(t)} s_k$$

"source context for decoder step t"

(weighted sum)

Attention weights

$$a_k^{(t)} = \frac{\exp(\text{score}(h_t, s_k))}{\sum_{i=1}^{m} \exp(\text{score}(h_t, s_i))}, \text{k} = 1..\text{m}$$

"attention weight for source token k at decoder step t"

(softmax)

Attention scores

$$\text{score}(h_t, s_k), \text{k} = 1..\text{m}$$

"How relevant is source token k for target step t?"

Attention input

$s_1, s_2, \ldots, s_m$ all encoder states

h_t one decoder state

Source: https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

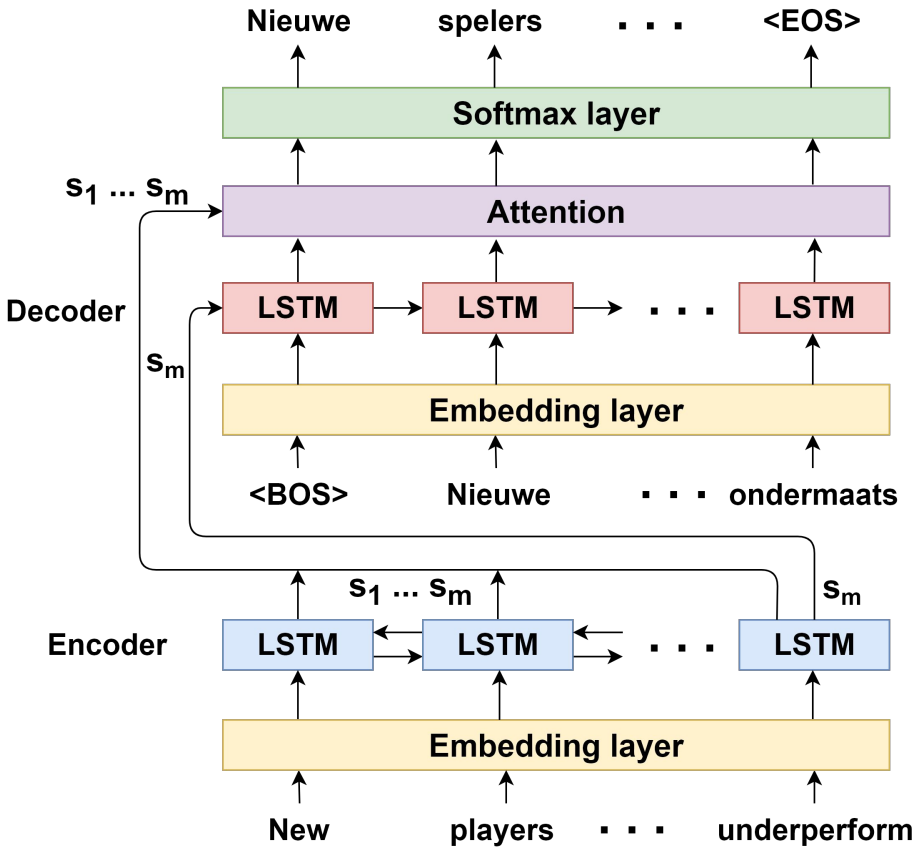
Attention score

- We have yet to define the function score($\mathbf{s_k}$, $\mathbf{h_t}$)
- There are multiple variants that usually work well
  - **Dot-product:** score($\mathbf{s_k}$, $\mathbf{h_t}$) = $h_t s_k$
  - **Bi-linear:** score($\mathbf{s_k}$, $\mathbf{h_t}$) = $h_t W_1 s_k$
  - **MLP:** score($\mathbf{s_k}$, $\mathbf{h_t}$) = $w_2 \tanh(W_1[s_k; h_t])$

Attention score

- We have yet to define the function score($\mathbf{s}_k$, $\mathbf{h}_t$)
- There are multiple variants that usually work well
  - **Dot-product:** $\text{score}(\mathbf{s}_k, \mathbf{h}_t) = h_t s_k$
  - **Bi-linear:** $\text{score}(\mathbf{s}_k, \mathbf{h}_t) = h_t W_1 s_k$
  - **MLP:** $\text{score}(\mathbf{s}_k, \mathbf{h}_t) = w_2 \tanh(W_1[s_k; h_t])$

The nice thing about dot-product is that it doesn't introduce any extra weights that need to be learned. It turns out we can automatically learn weights for the other layers so that they work well with dot-product attention!


Nieuwe
spelers
. . .
<EOS>
Softmax layer
$s_1 \dots s_m$
Attention
Decoder
LSTM
LSTM
. . .
LSTM
s_m
Embedding layer
<BOS>
Nieuwe
. . .
ondermaats
$s_1 \dots s_m$
s_m
Encoder
LSTM
LSTM
. . .
LSTM
Embedding layer
New
players
. . .
underperform

Impact on NLP

- LSTMs and sequence-to-sequence models had a **huge** impact on the field

Impact on NLP

- LSTMs and sequence-to-sequence models had a **huge** impact on the field
- **Many advantages**: no hand-written rules, no explicit features, no word lists, etc
  - Remember the first assignment?

Impact on NLP

- LSTMs and sequence-to-sequence models had a **huge** impact on the field
- **Many advantages**: no hand-written rules, no explicit features, no word lists, etc
  - Remember the first assignment?
- These models can be trained **end-to-end**: no intermediate steps needed

Impact on NLP

- LSTMs and sequence-to-sequence models had a **huge** impact on the field
- **Many advantages**: no hand-written rules, no explicit features, no word lists, etc
  - Remember the first assignment?
- These models can be trained **end-to-end**: no intermediate steps needed
- These models can easily take advantage of pretrained word embeddings

Impact on NLP

- LSTMs and sequence-to-sequence models had a **huge** impact on the field
- **Many advantages**: no hand-written rules, no explicit features, no word lists, etc
  - Remember the first assignment?
- These models can be trained **end-to-end**: no intermediate steps needed
- These models can easily take advantage of pretrained word embeddings

Deep learning wave: almost all NLP tasks could be modeled by using one of these models, lots and lots of papers with state-of-the-art results

Improving on word embeddings

Improving word embeddings

Even though pretrained word embeddings work great, they still have a **huge flaw**: they have a **fixed vector** per individual word and do not take **context** into account

Improving word embeddings

Even though pretrained word embeddings work great, they still have a **huge flaw**: they have a **fixed vector** per individual word and do not take **context** into account

- I have accidentally **set** myself on fire
- He won the first **set**

Improving word embeddings

Even though pretrained word embeddings work great, they still have a **huge flaw**: they have a **fixed vector** per individual word and do not take **context** into account

- I have accidentally **set** myself on fire
- He won the first **set**

Surely we must be able to do better than a fixed vector per word!

ELMo: Deep Contextualized Word Representations

- Use a deep LSTM model for language modelling
  - Remember: predict the next word given the previous words
  - Train this model on millions and millions of sentences

ELMo: Deep Contextualized Word Representations

- Use a deep LSTM model for language modelling
  - Remember: predict the next word given the previous words
  - Train this model on millions and millions of sentences
- LSTM hidden states (vectors) per input word depend on the previous words
- Run a bi-directional LSTM to fully exploit this context

ELMo: Deep Contextualized Word Representations

- Use a deep LSTM model for language modelling
  - Remember: predict the next word given the previous words
  - Train this model on millions and millions of sentences
- LSTM hidden states (vectors) per input word depend on the previous words
- Run a bi-directional LSTM to fully exploit this context
- Instead of keeping a single weight matrix, keep the **full model**!

ELMo: Deep Contextualized Word Representations

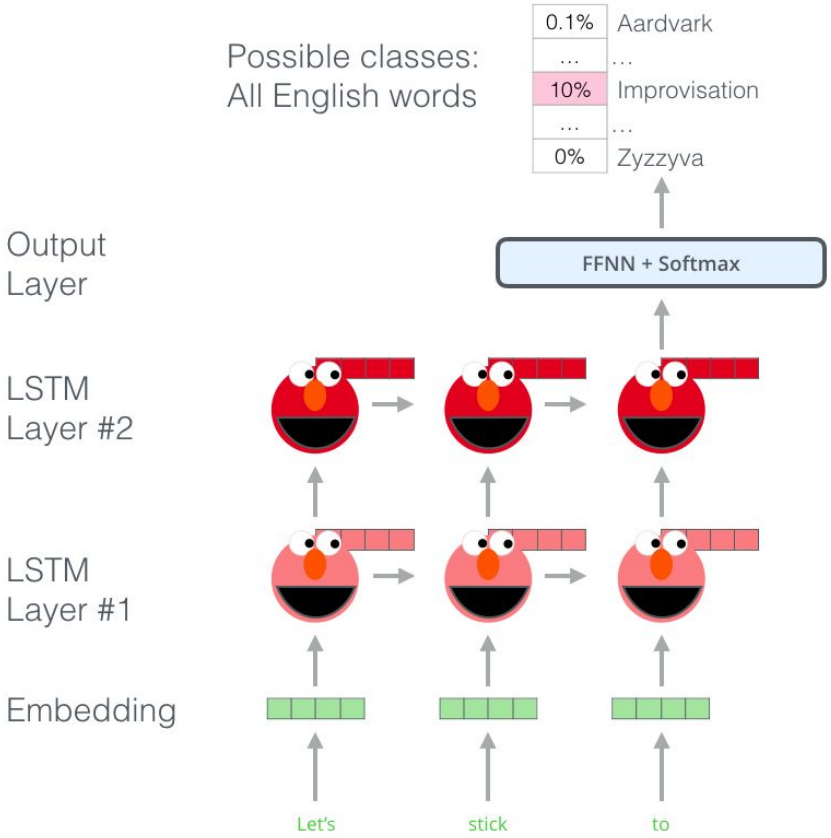
- Use a deep LSTM model for language modelling
  - Remember: predict the next word given the previous words
  - Train this model on millions and millions of sentences
- LSTM hidden states (vectors) per input word depend on the previous words
- Run a bi-directional LSTM to fully exploit this context
- Instead of keeping a single weight matrix, keep the **full model**!
- Use this full model to calculate **input vectors** for input words in our own task

ELMo: Deep Contextualized Word Representations

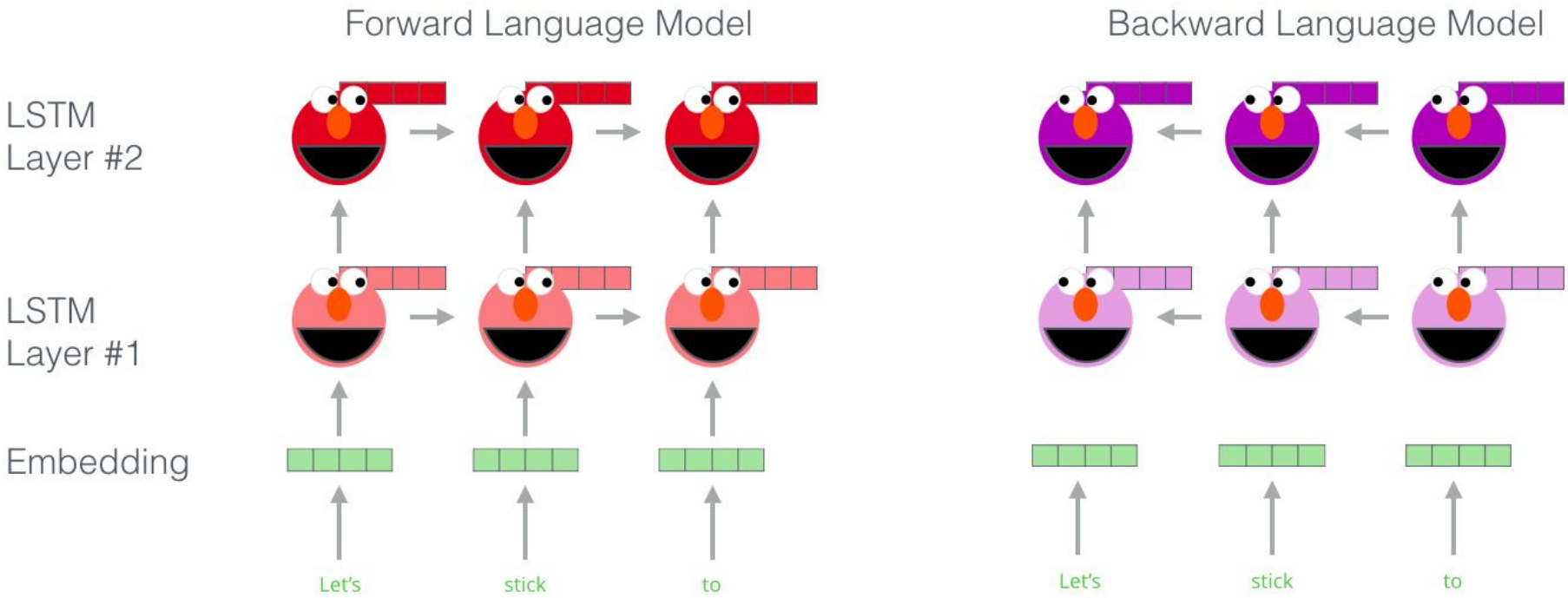
- Use a deep LSTM model for language modelling
  - Remember: predict the next word given the previous words
  - Train this model on millions and millions of sentences
- LSTM hidden states (vectors) per input word depend on the previous words
- Run a bi-directional LSTM to fully exploit this context
- Instead of keeping a single weight matrix, keep the **full model**!
- Use this full model to calculate **input vectors** for input words in our own task

Intuition: use the pretrained LSTM model to initialize our own embeddings. Since the LSTM takes context into account, so do our input vectors now!

Source: https://jalammar.github.io/illustrated-bert/


Possible classes:
All English words
0.1%
Aardvark
...
...
10%
Improvisation
...
...
0%
Zyzzyva
Output Layer
FFNN + Softmax
LSTM Layer #2
LSTM Layer #1
Embedding
Let's
stick
to

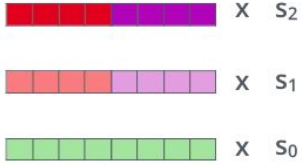
Embedding of “stick” in “Let’s stick to” - Step #1



Embedding of “stick” in “Let’s stick to” - Step #2

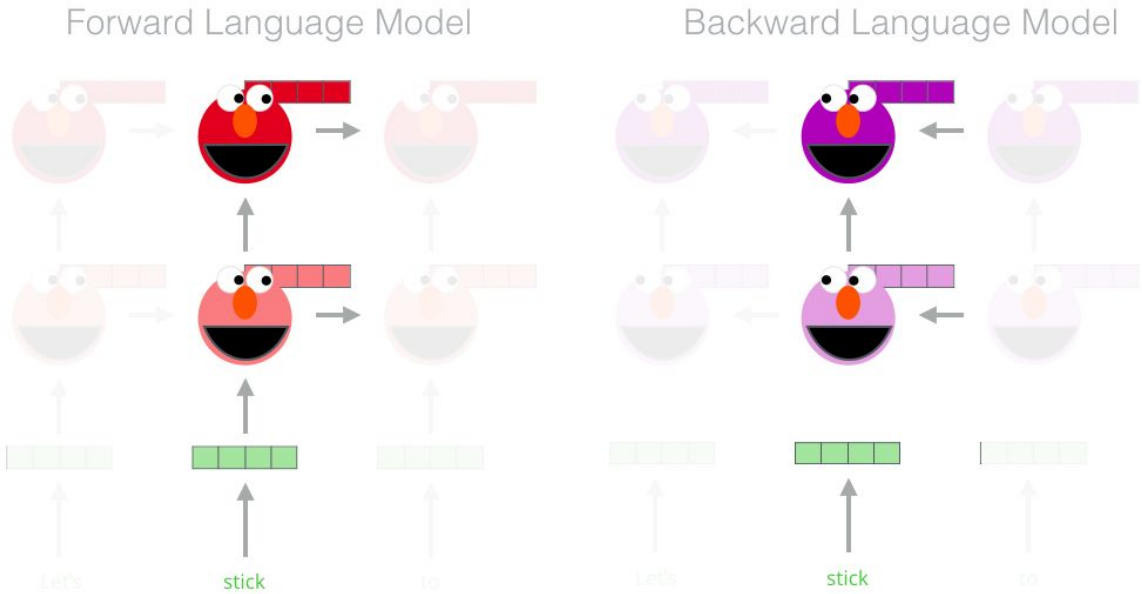
1- Concatenate hidden layers

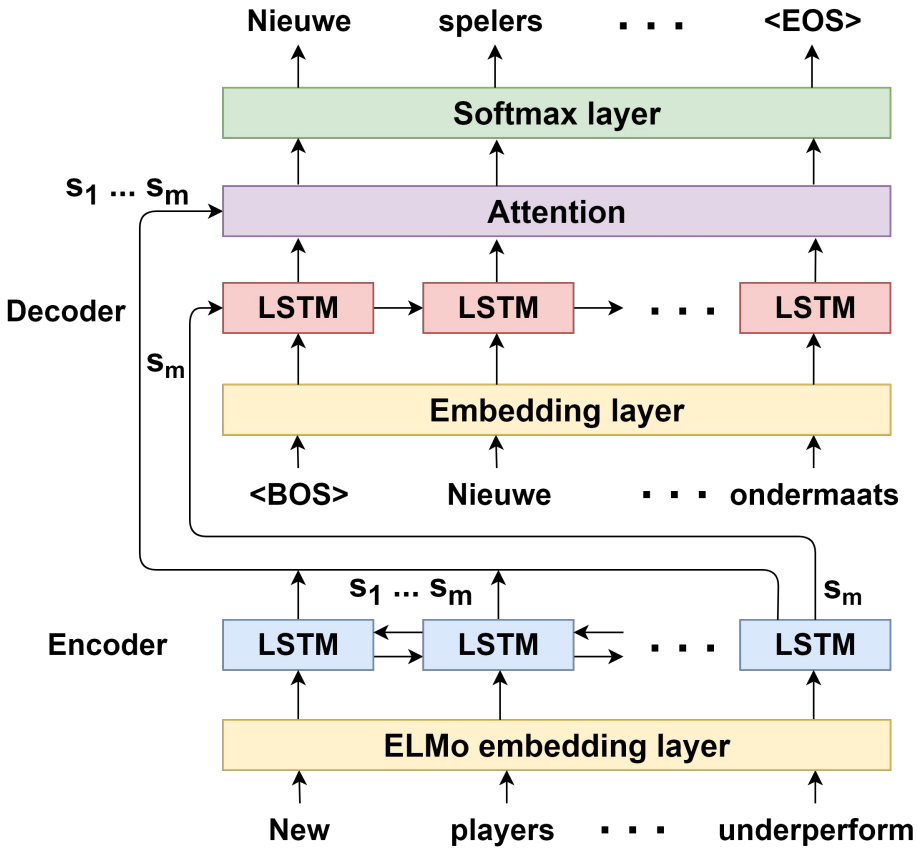
2- Multiply each vector by a weight based on the task

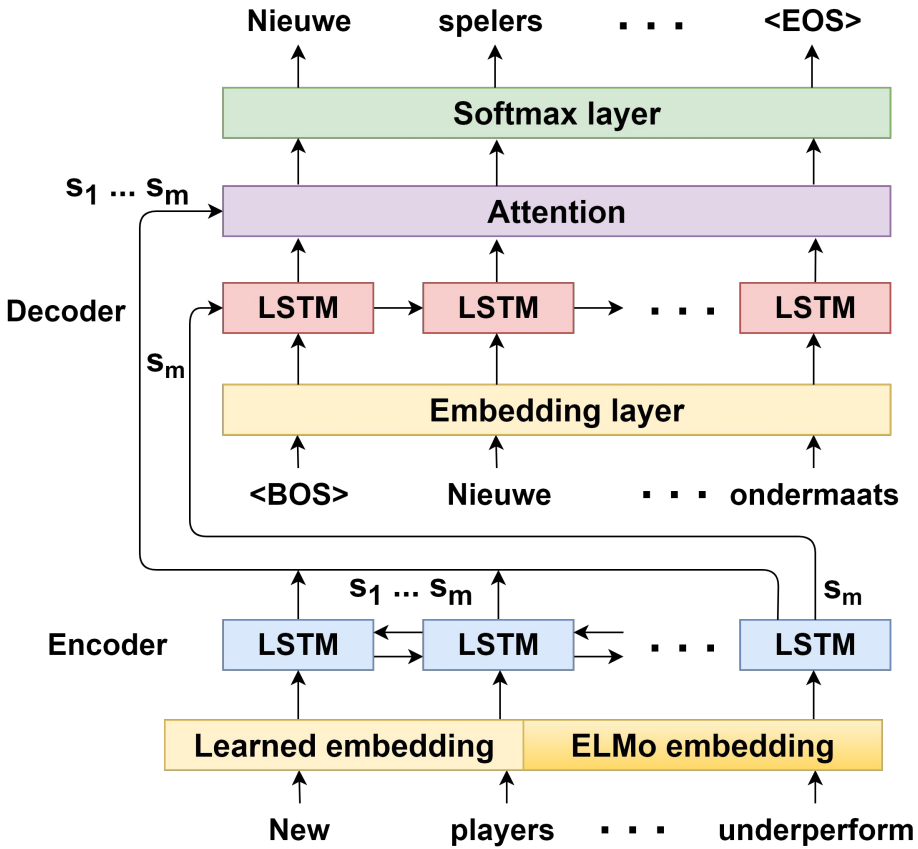


3- Sum the (now weighted) vectors

ELMo embedding of “stick” for this task in this context




Nieuwe
spelers
. . .
<EOS>
Softmax layer
s1 ... sm
Attention
Decoder
LSTM
LSTM
. . .
LSTM
sm
Embedding layer
<BOS>
Nieuwe
. . .
ondermaats
s1 ... sm
sm
Encoder
LSTM
LSTM
. . .
LSTM
ELMo embedding layer
New
players
. . .
underperform


Nieuwe
spelers
. . .
<EOS>
Softmax layer
s1 ... sm
Attention
Decoder
LSTM
LSTM
. . .
LSTM
sm
Embedding layer
<BOS>
Nieuwe
. . .
ondermaats
s1 ... sm
sm
Encoder
LSTM
LSTM
. . .
LSTM
Learned embedding
ELMo embedding
New
players
. . .
underperform

Why does this work so well?

Transfer learning: the model learns how to do task X well (in this case language modelling). This knowledge can then **transfer** towards different tasks. The weights of the model have acquired some sort of general “understanding” of language.

Why does this work so well?

Transfer learning: the model learns how to do task X well (in this case language modelling). This knowledge can then **transfer** towards different tasks. The weights of the model have acquired some sort of general “understanding” of language.

Intuition: our model weights for hidden (LSTM) layers are randomly initialized.
Of course, they start out being far from perfect. Training a model on a different task might also not be perfect, but it’s easy to do better than a **random** initialization!

Why does this work so well?

Transfer learning: the model learns how to do task X well (in this case language modelling). This knowledge can then **transfer** towards different tasks. The weights of the model have acquired some sort of general “understanding” of language.

Intuition: our model weights for hidden (LSTM) layers are randomly initialized.
Of course, they start out being far from perfect. Training a model on a different task might also not be perfect, but it’s easy to do better than a **random** initialization!

Word embeddings: the embedding layer should not be randomly initialized
Transfer learning: as much weights as possible should not be randomly initialized

Assignment 3

- In groups of 3. Write a report structured like a research paper again
- Experiment with LSTMs and pretrained word embeddings
- Experiment with fine-tuning pretrained language models
  - This will all be explained next week!
  - Work with Google Colab for access to GPU
- Extra requirement: paragraph on who did what
  - Potentially grades can be changed based on this information
- Please check Brightspace for a long list of papers and resources

Deadline: 16 October 10:59