Language Models

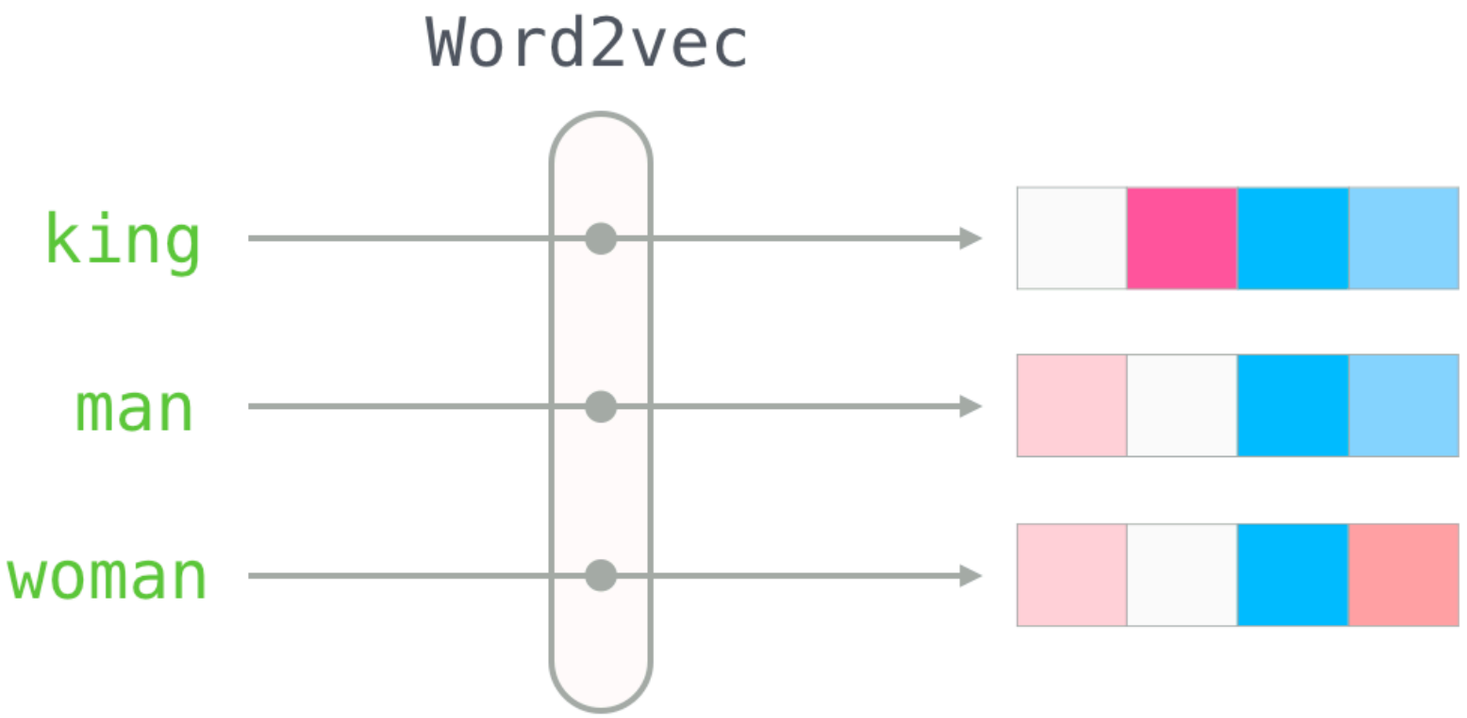
Boris Zubarev

@bobazooba

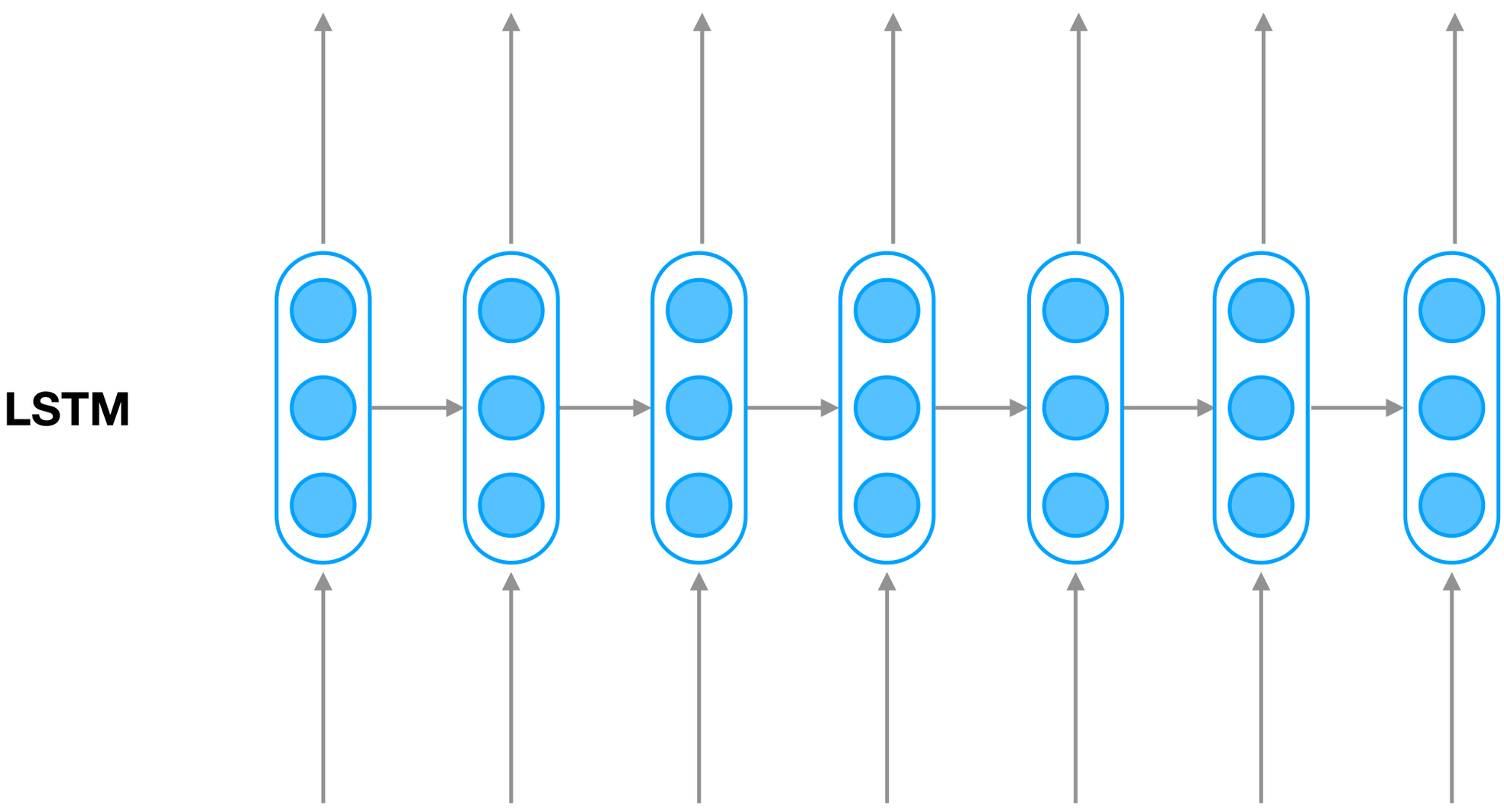
Word Embeddings

w2v, glove, etc

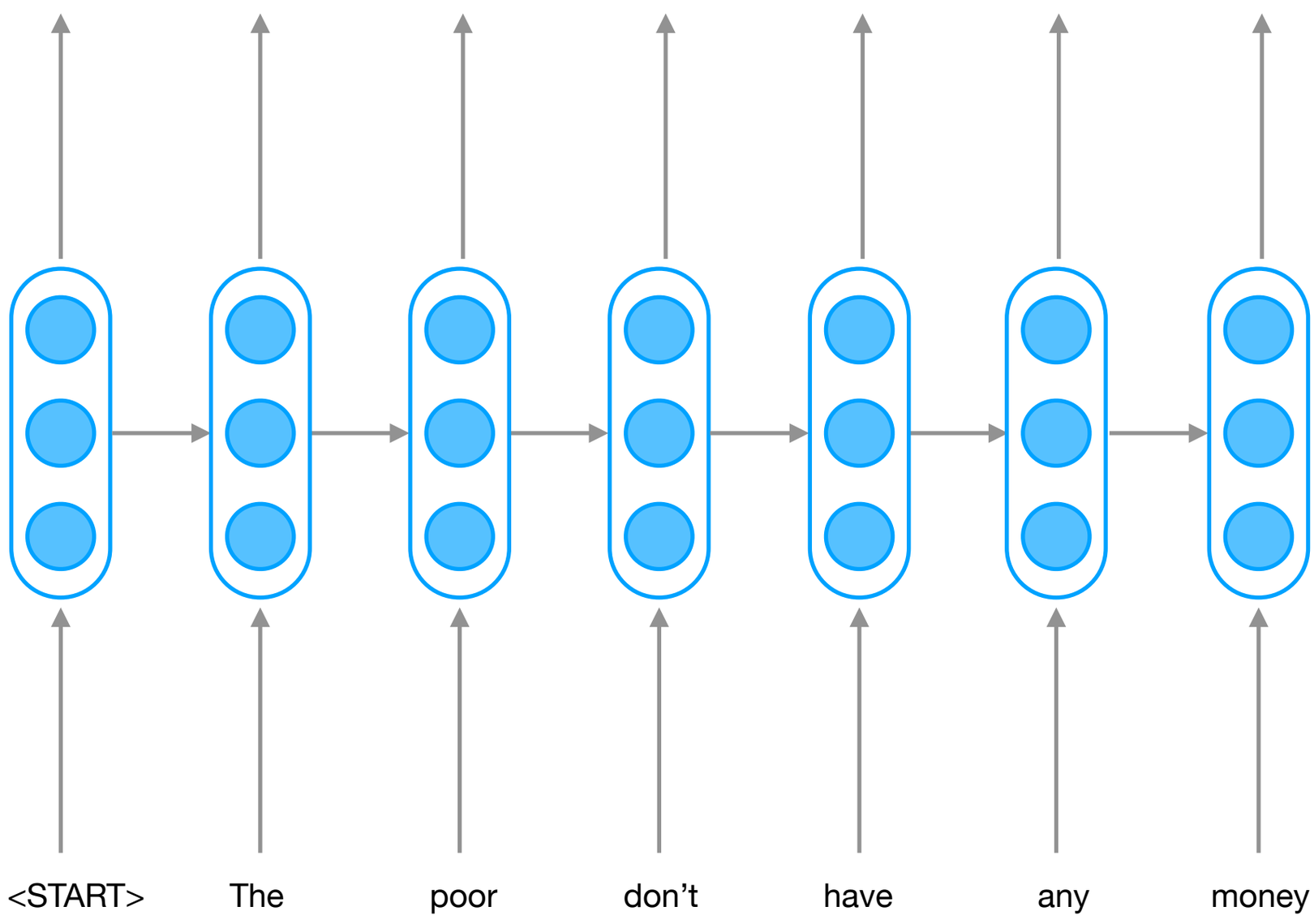
- Just key—value storage at inference
- Don’t change from relationships with other words in the current text



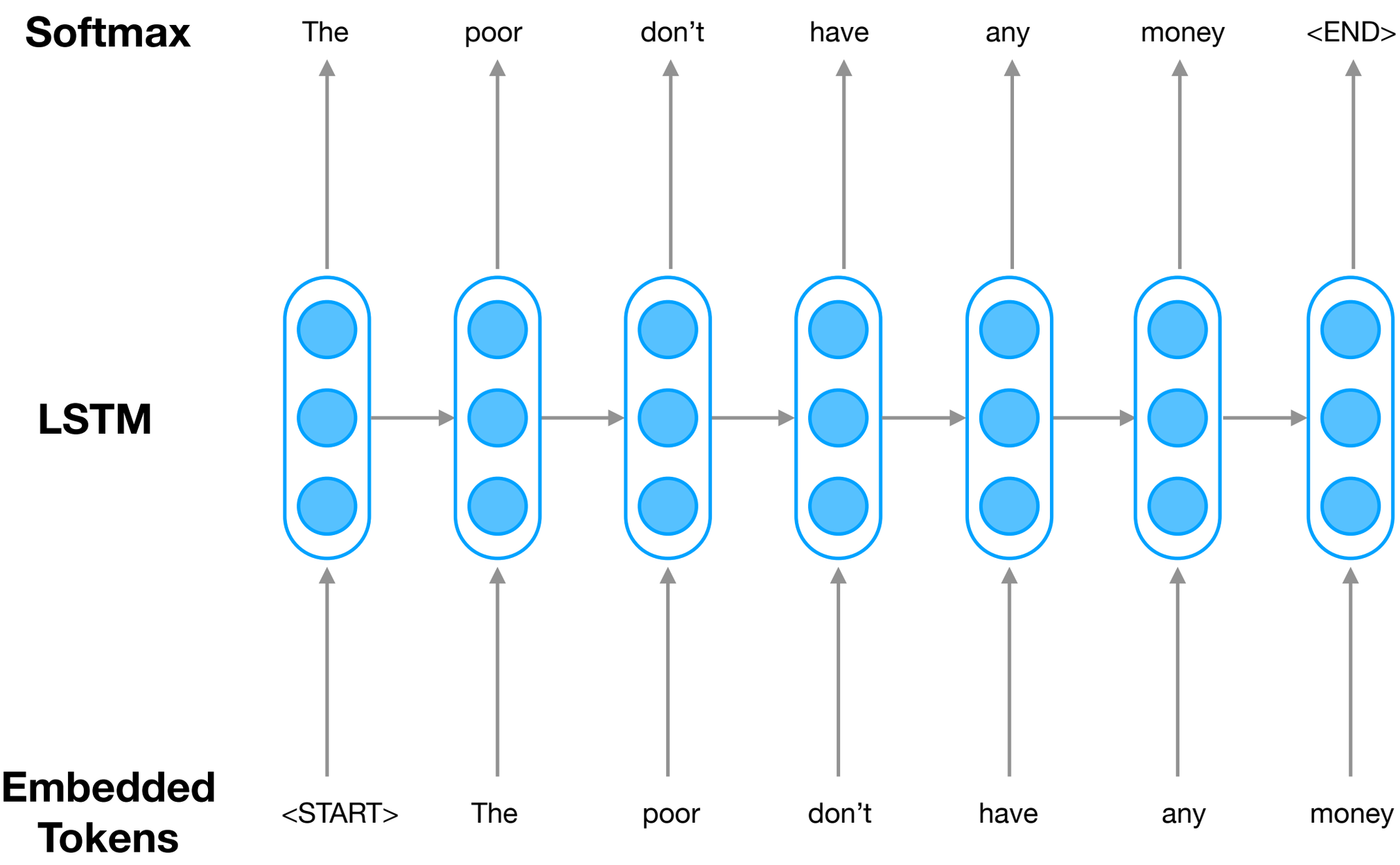
Language Model



Language Model



Language Model



Language Model

Inference

LSTM

Embedded Tokens

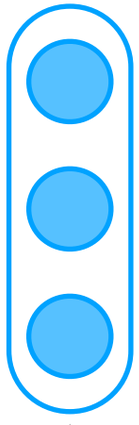
<START>

Language Model

Inference

Softmax

LSTM

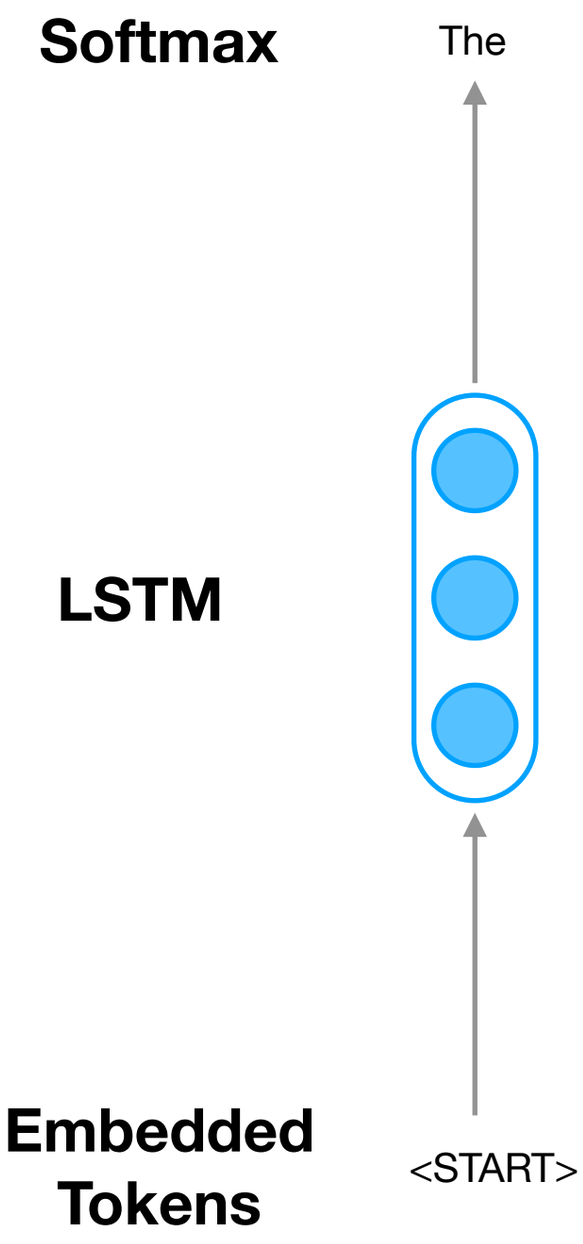


Embedded Tokens

<START>

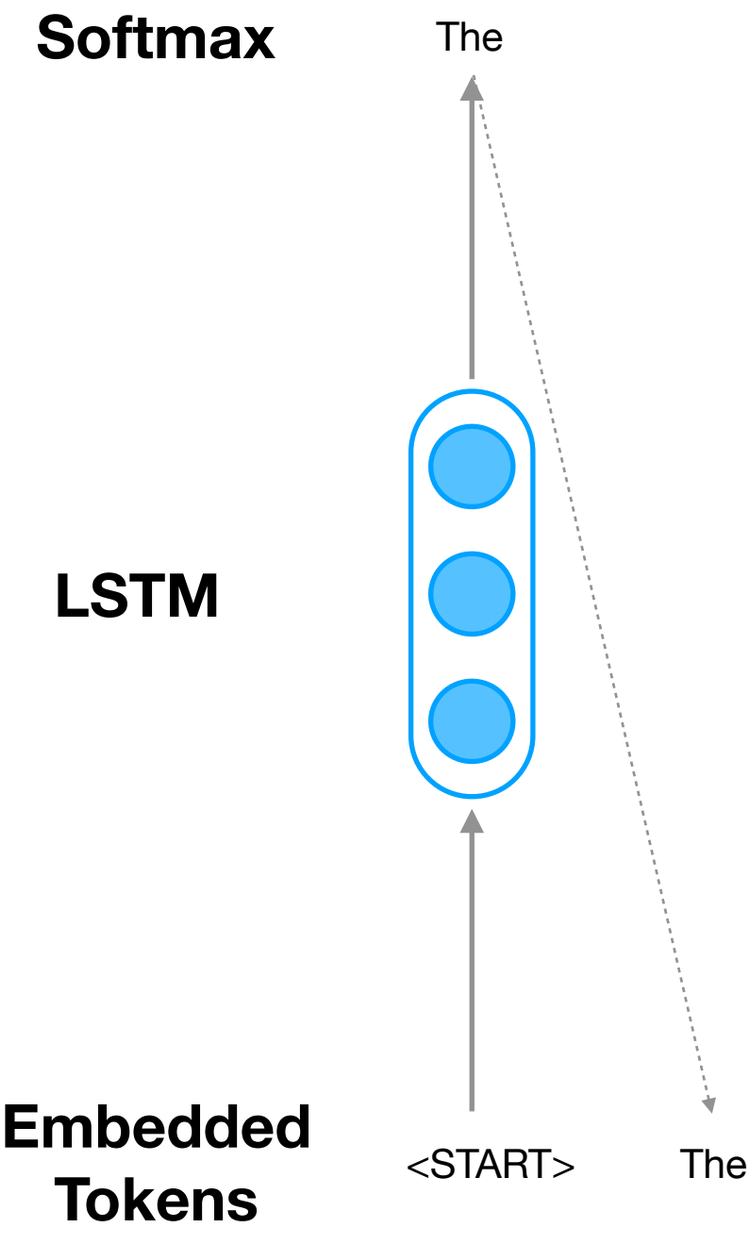
Language Model

Inference



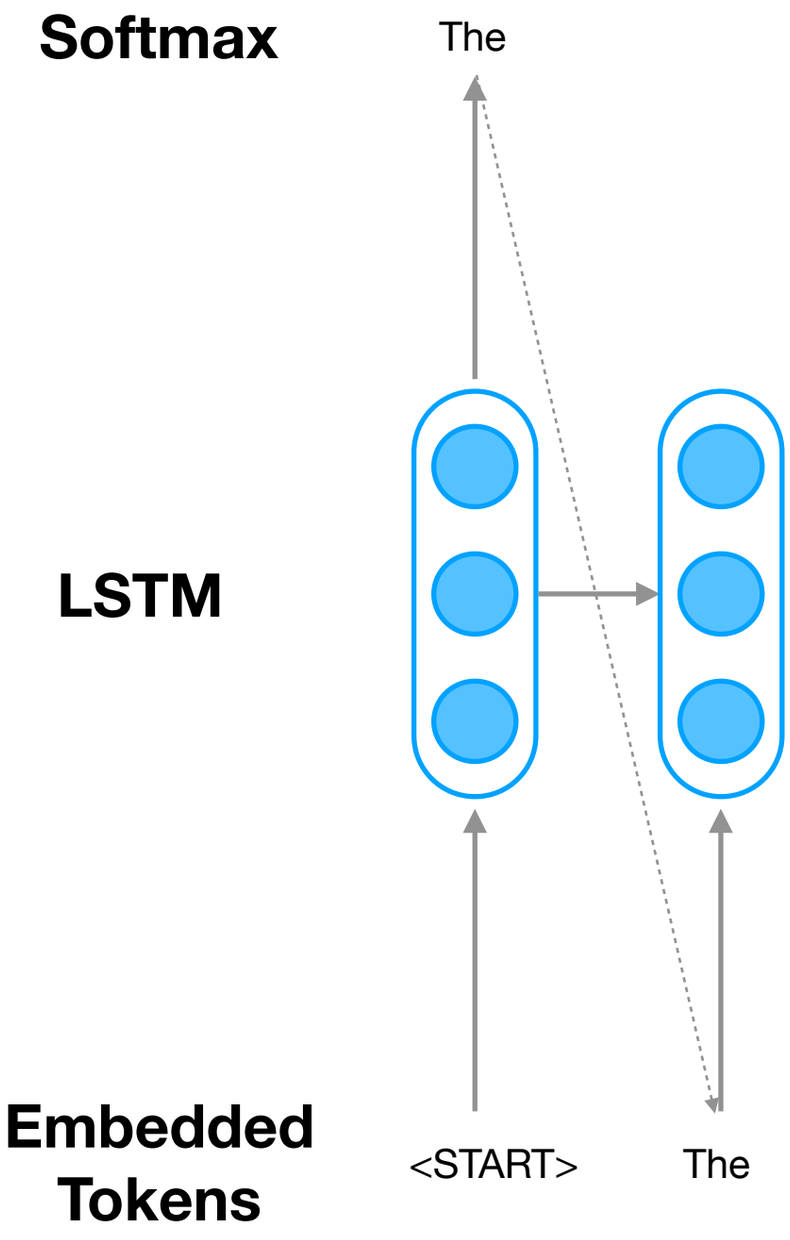
Language Model

Inference



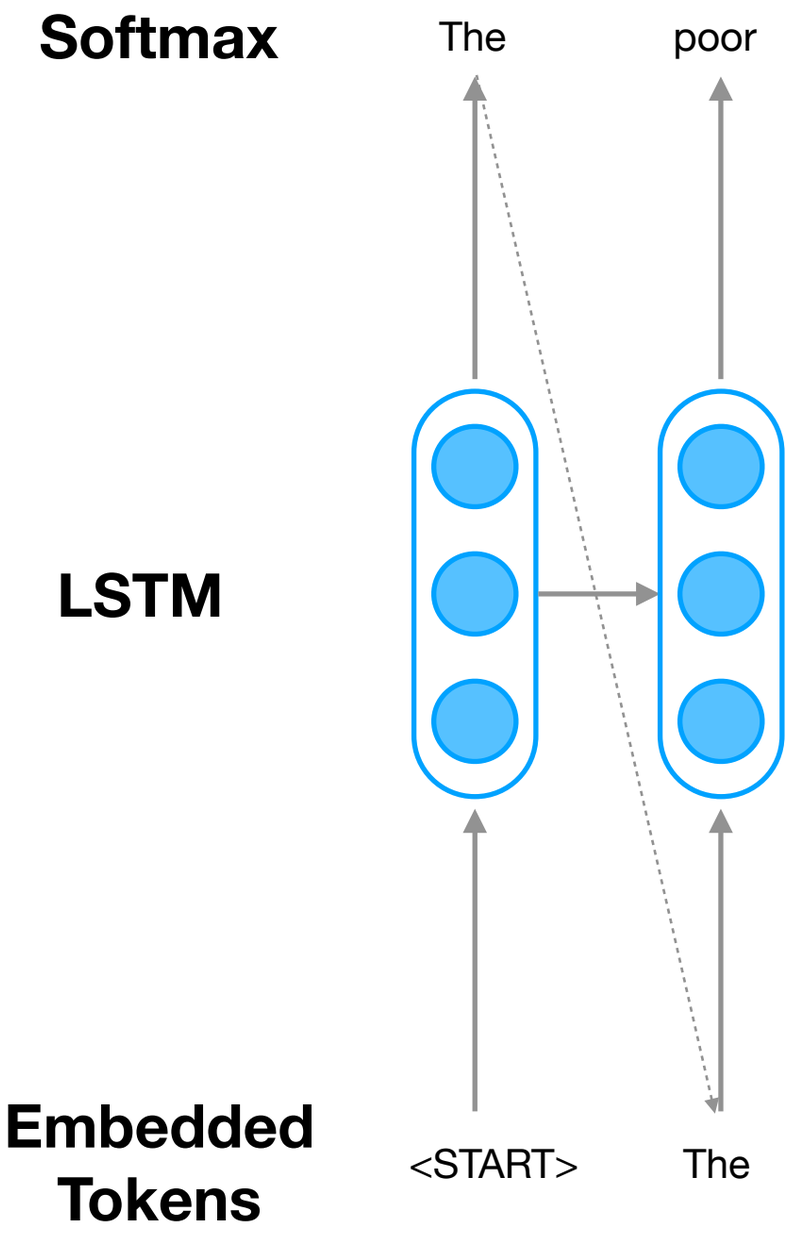
Language Model

Inference



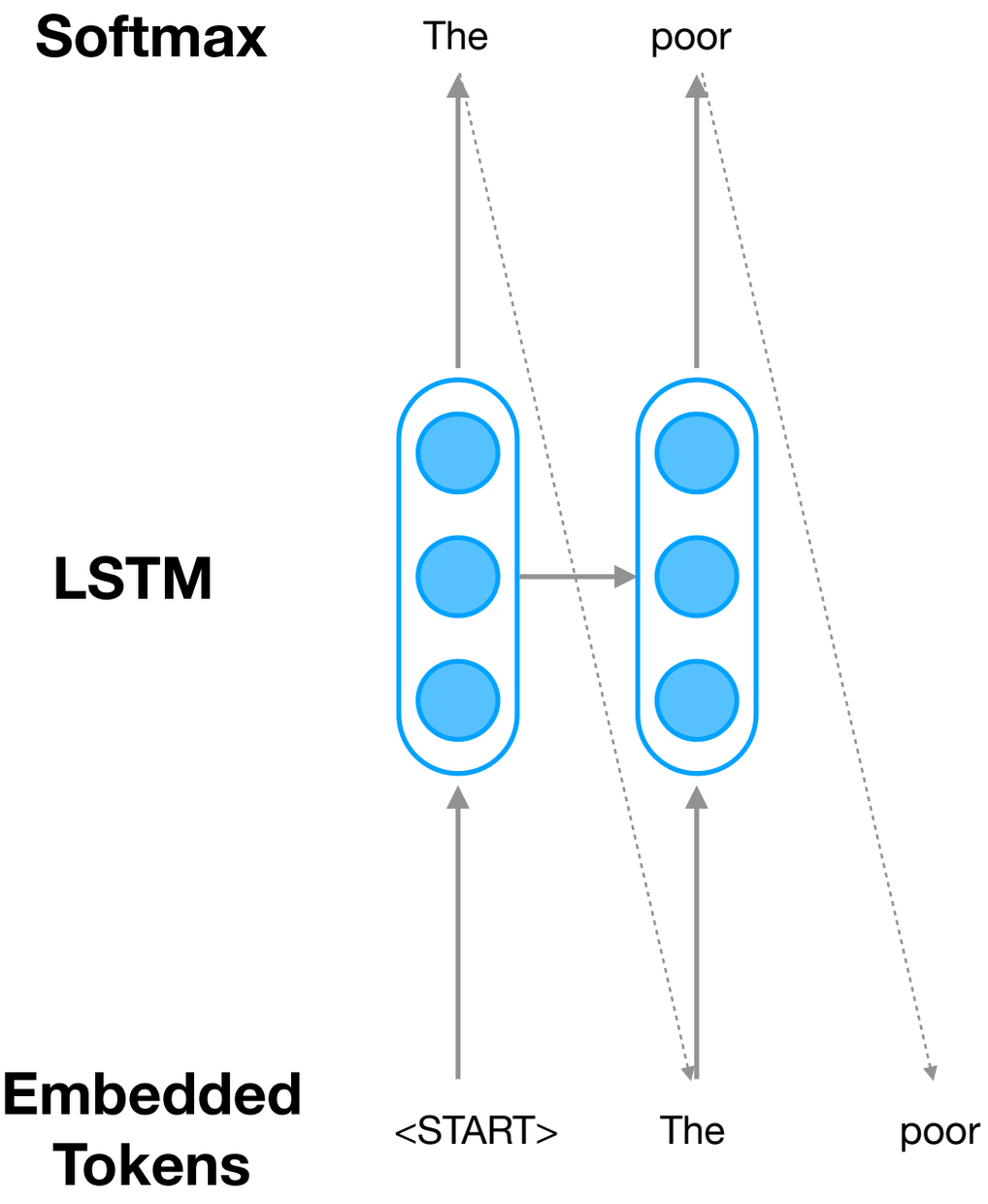
Language Model

Inference



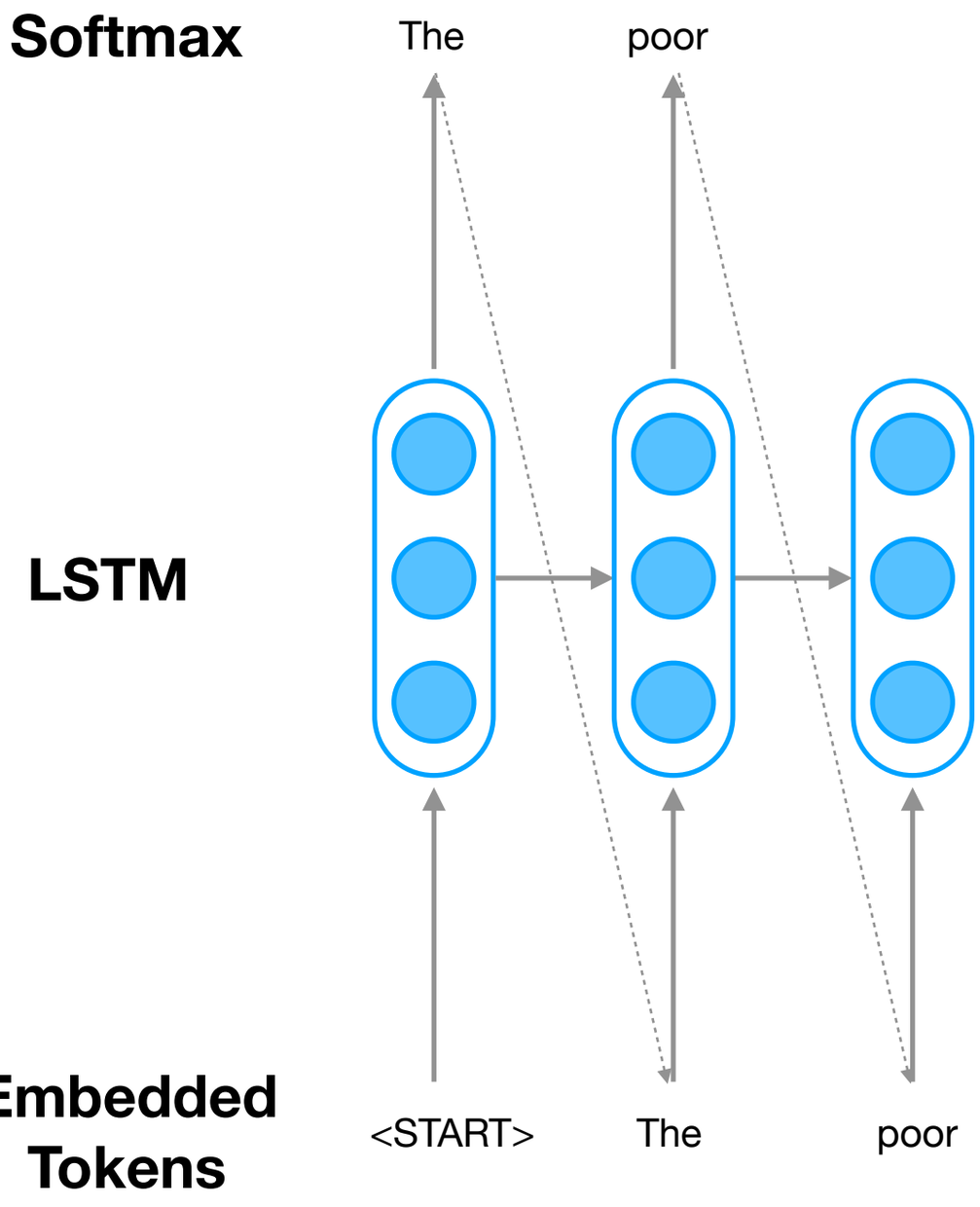
Language Model

Inference



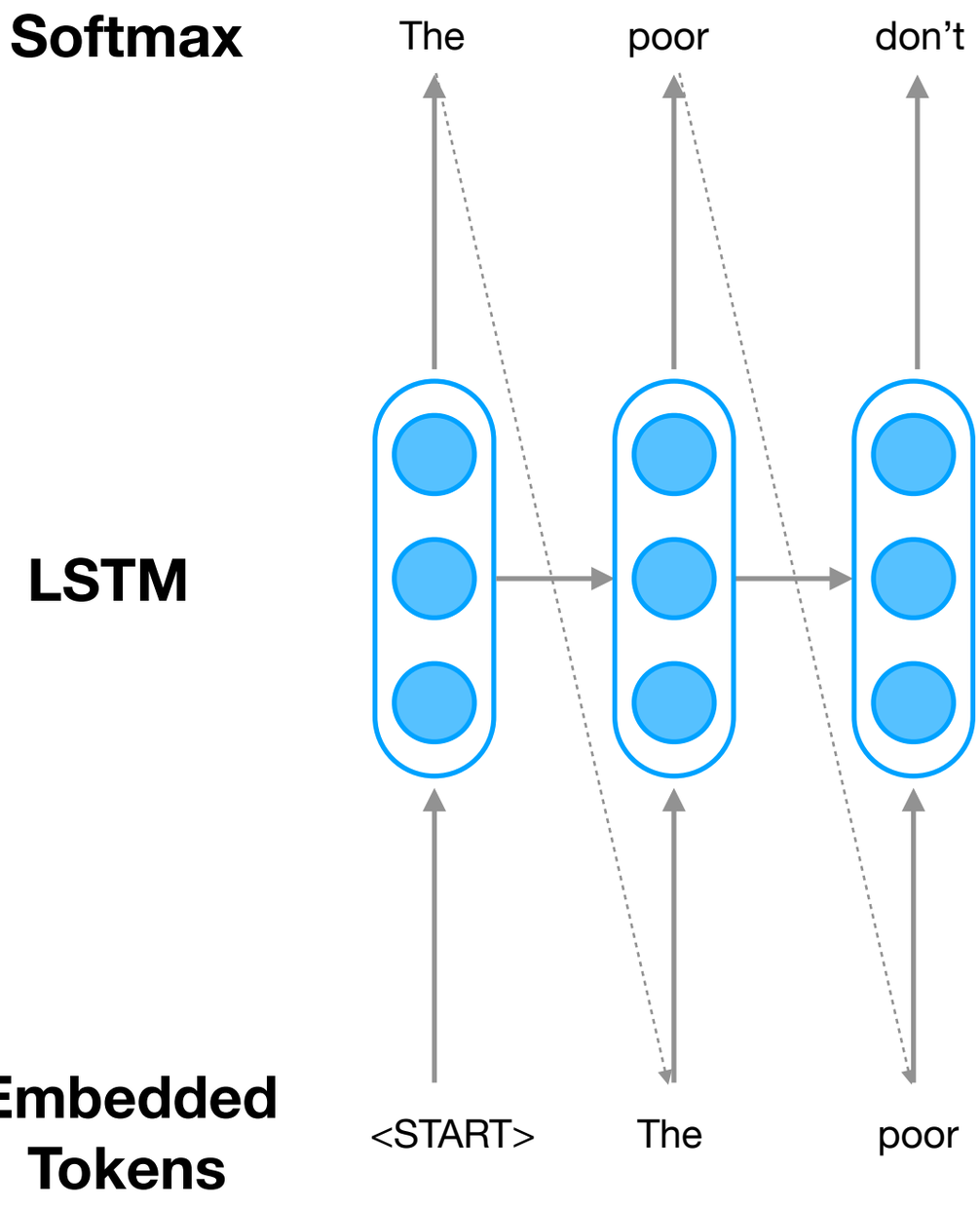
Language Model

Inference



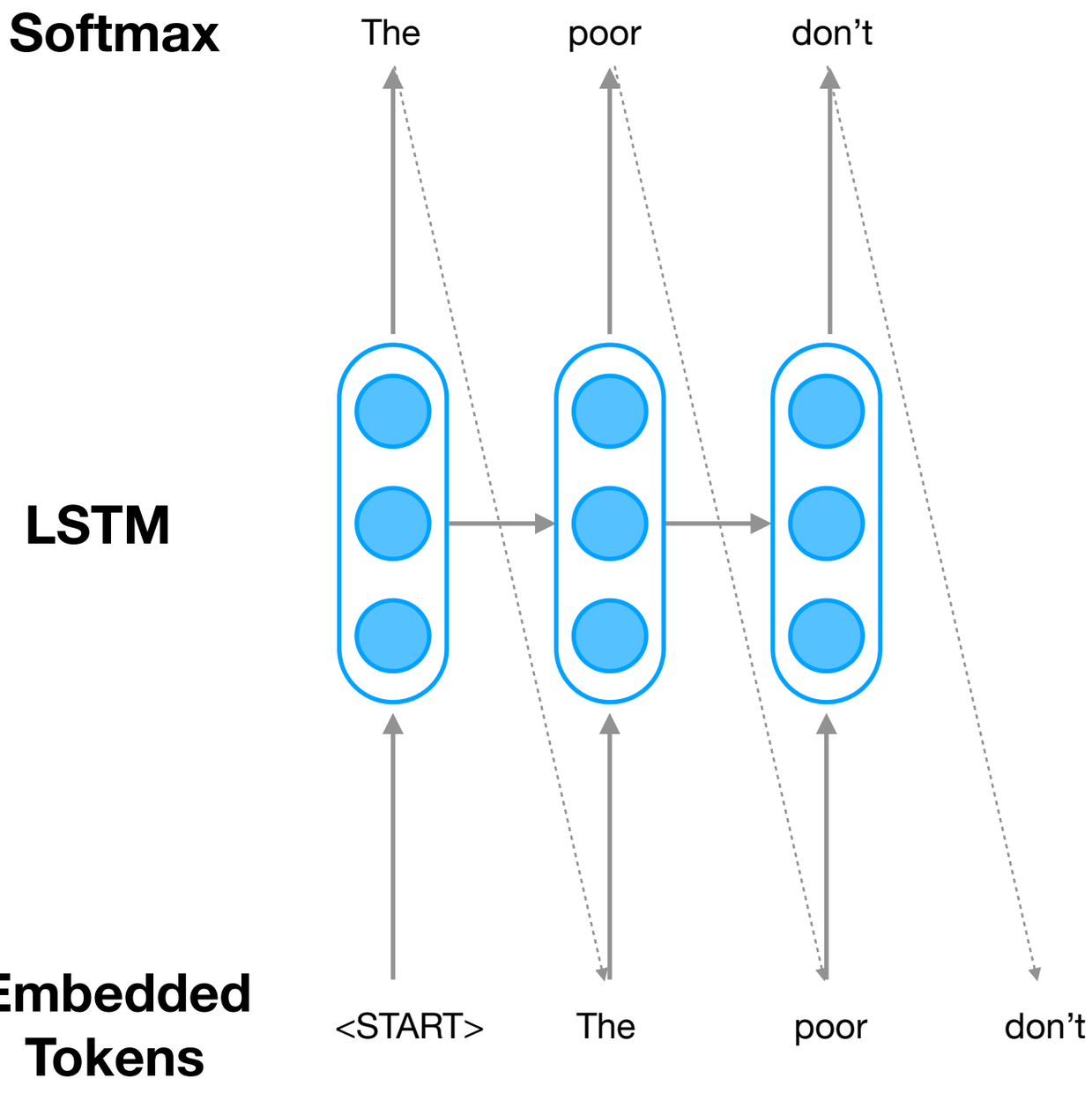
Language Model

Inference



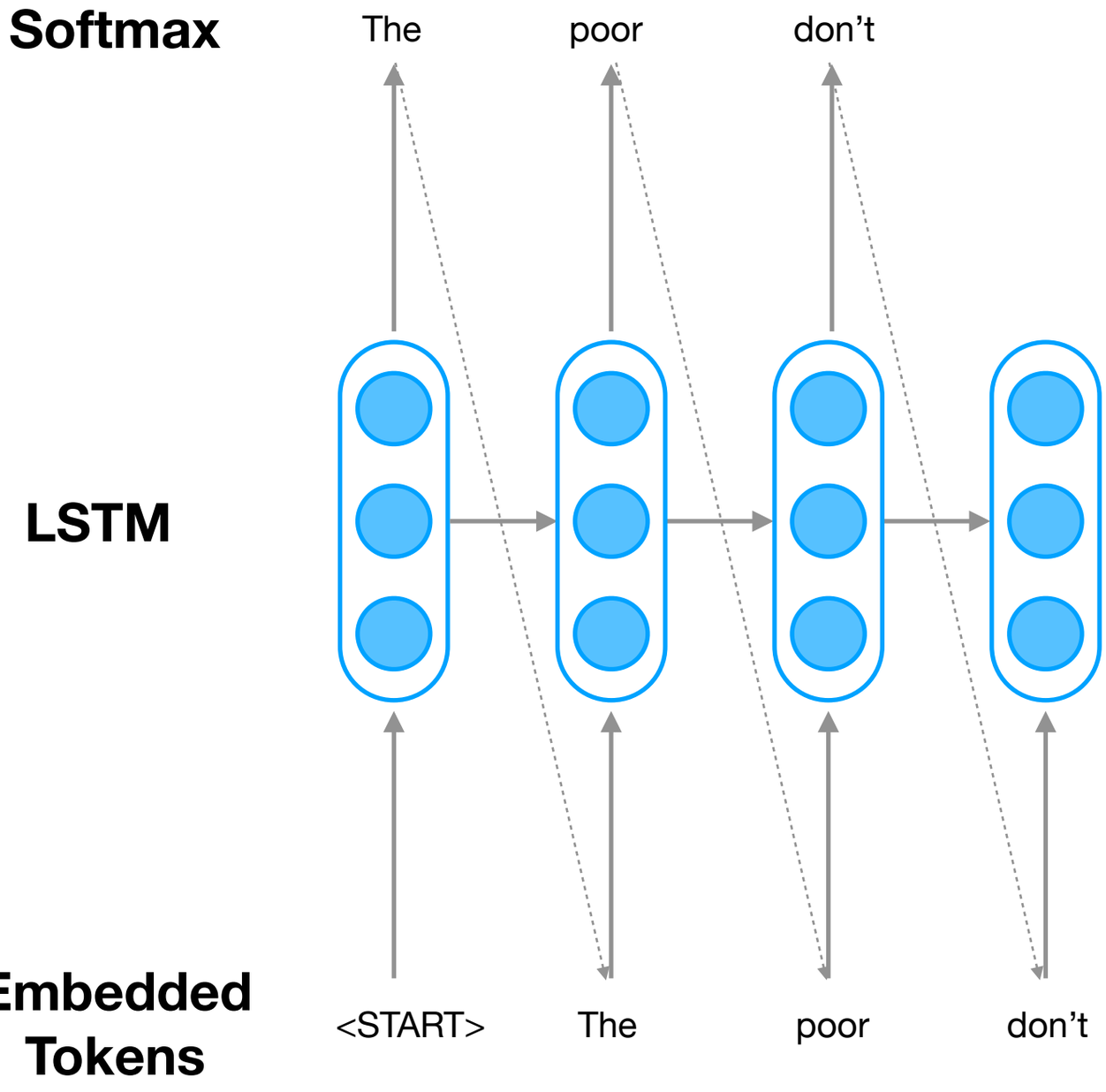
Language Model

Inference



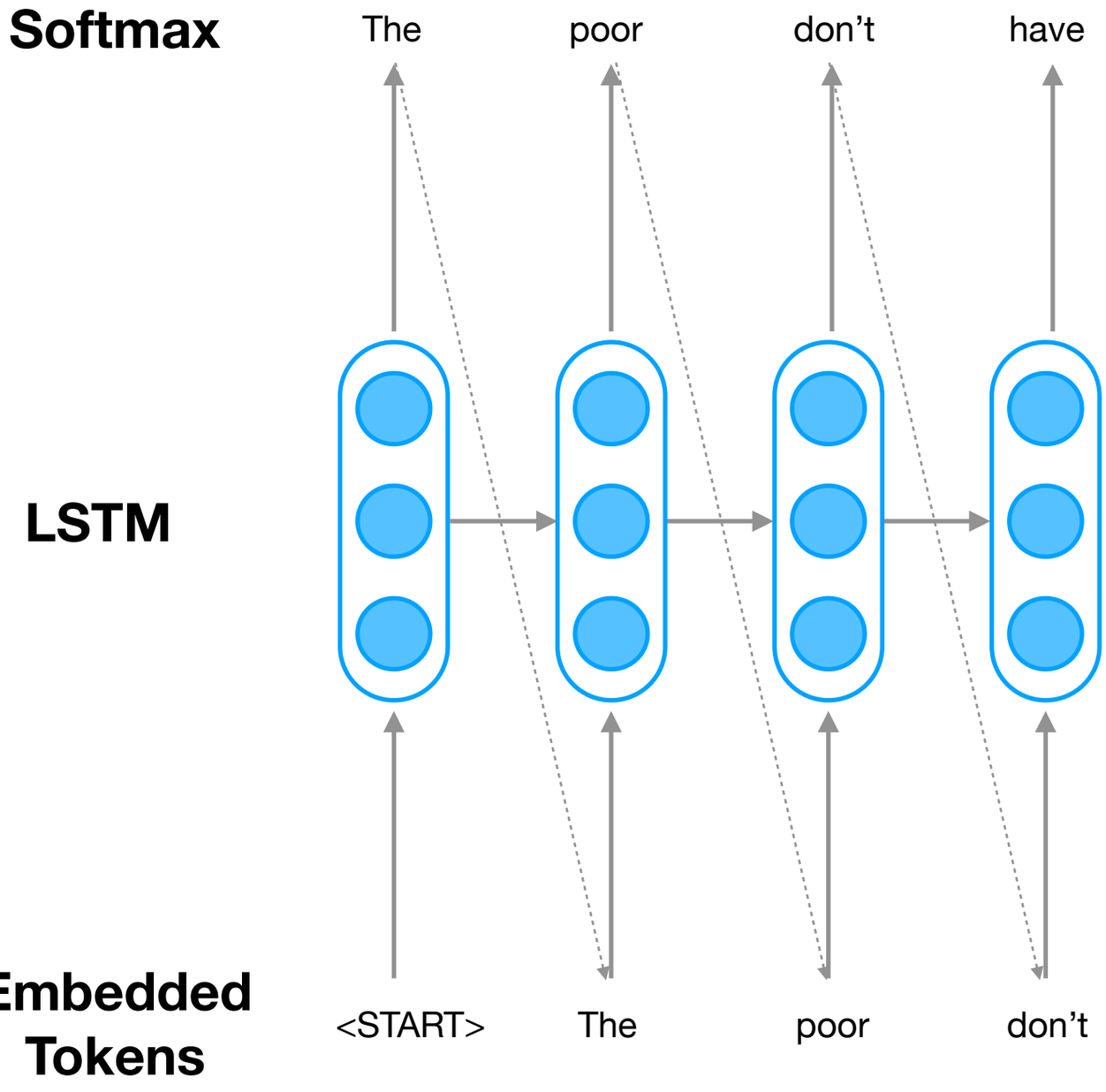
Language Model

Inference



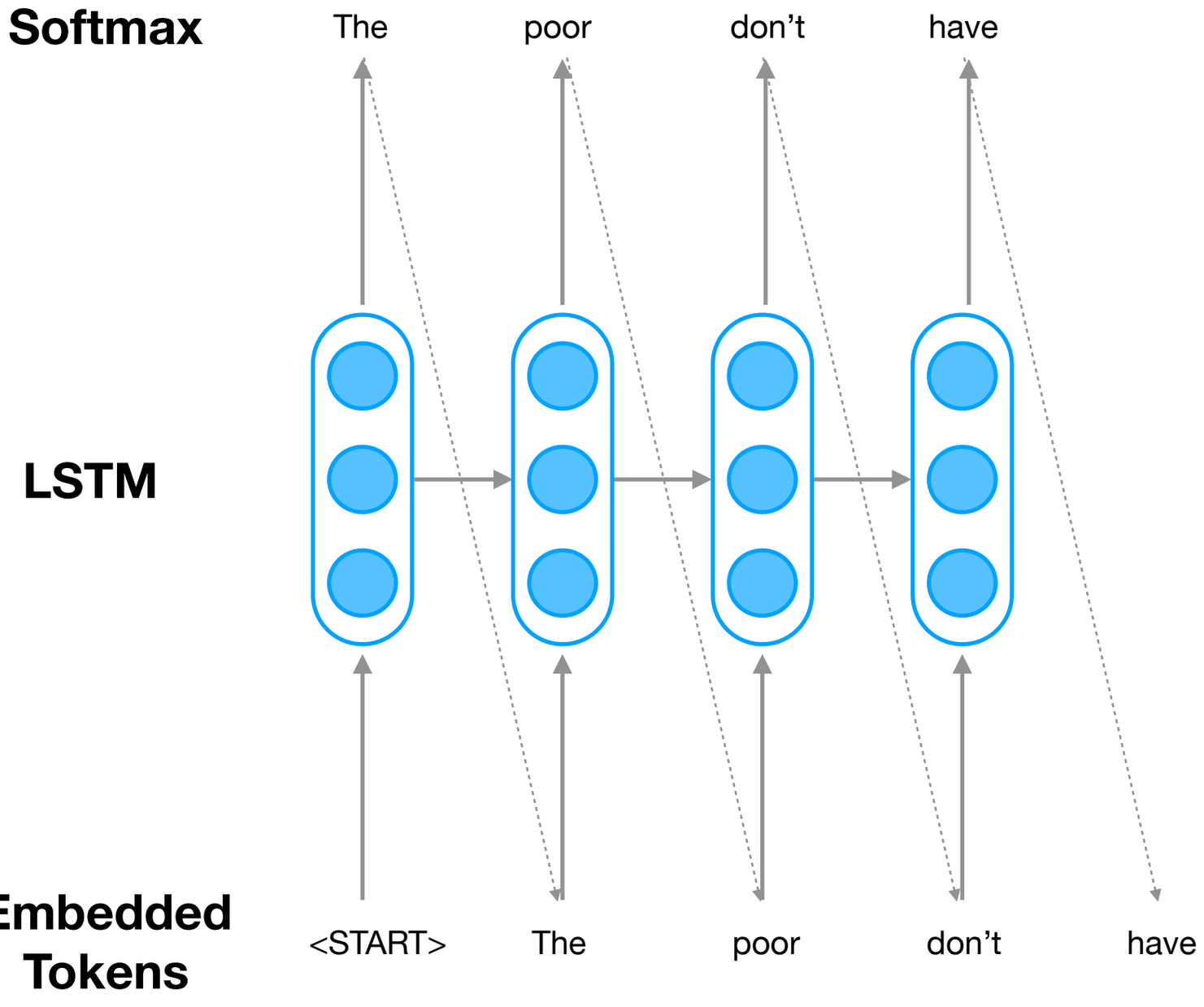
Language Model

Inference



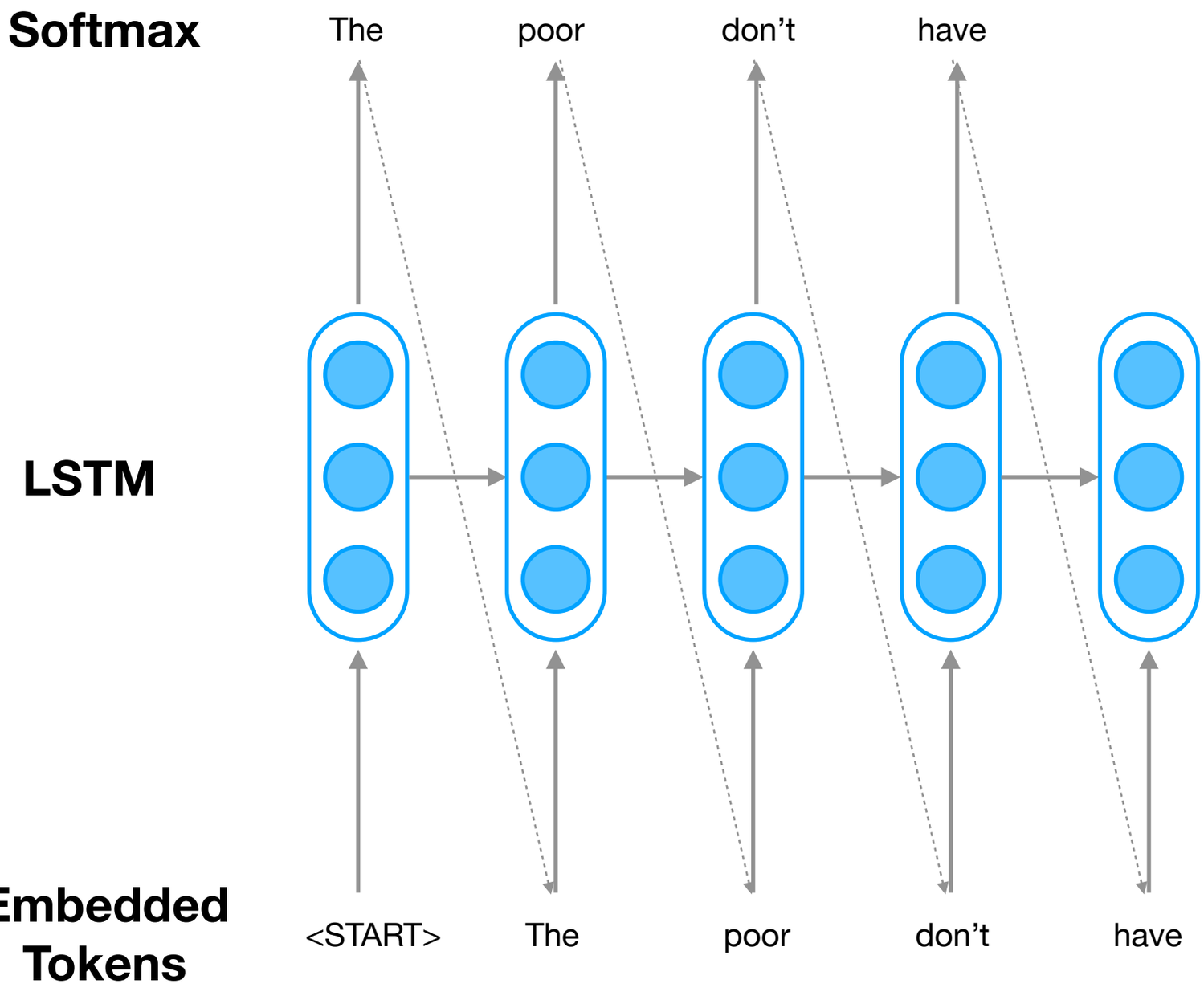
Language Model

Inference



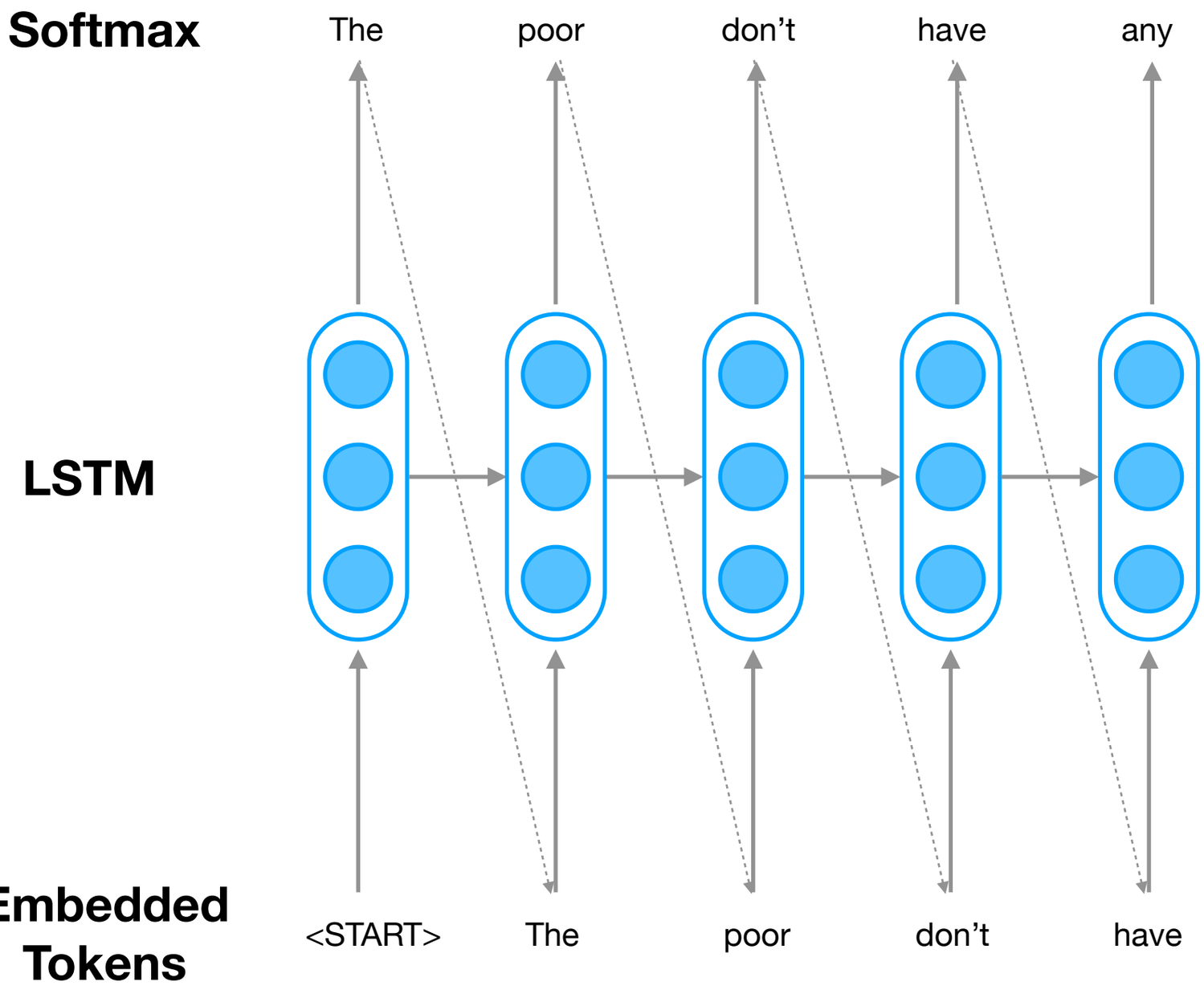
Language Model

Inference



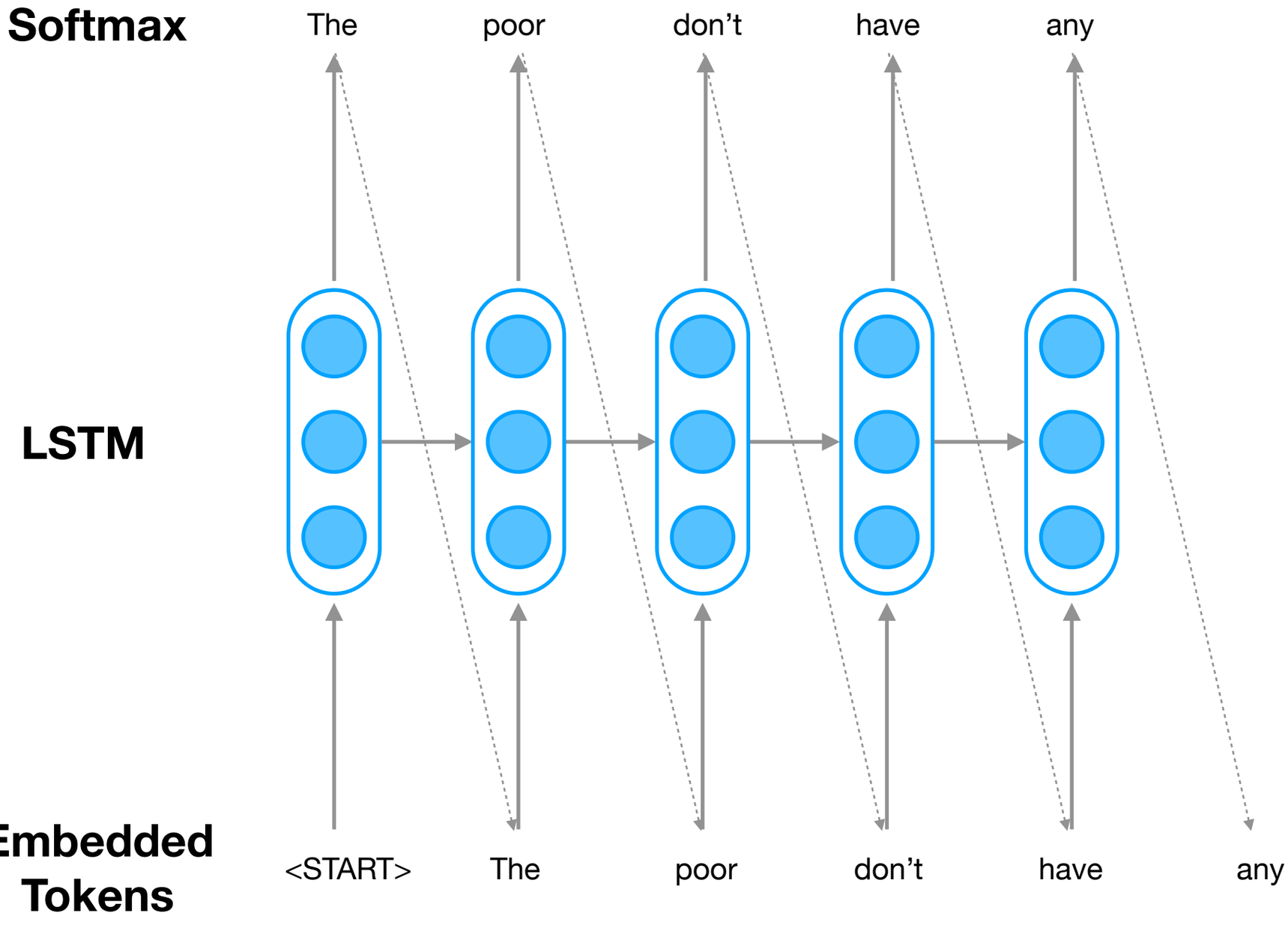
Language Model

Inference



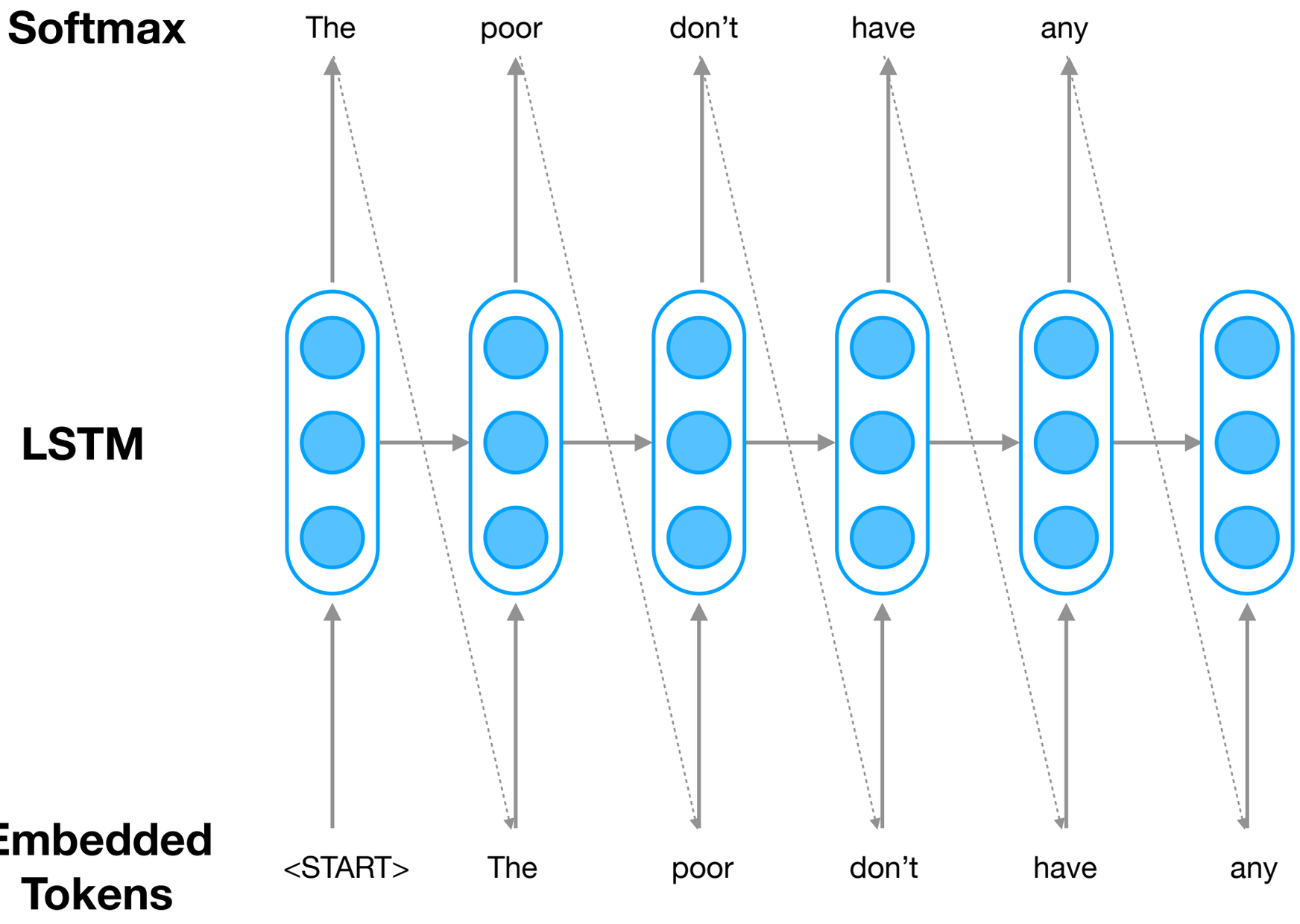
Language Model

Inference



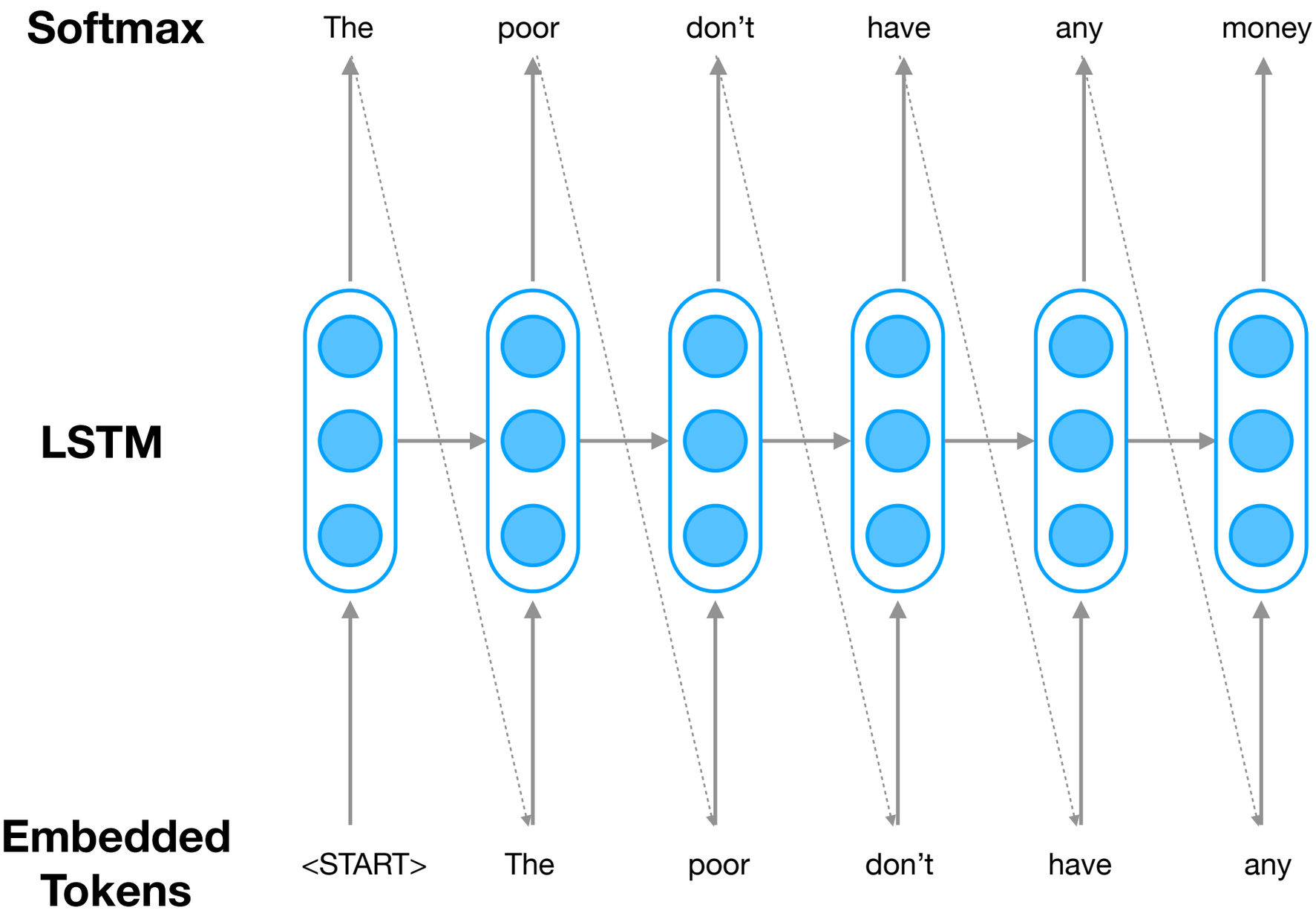
Language Model

Inference



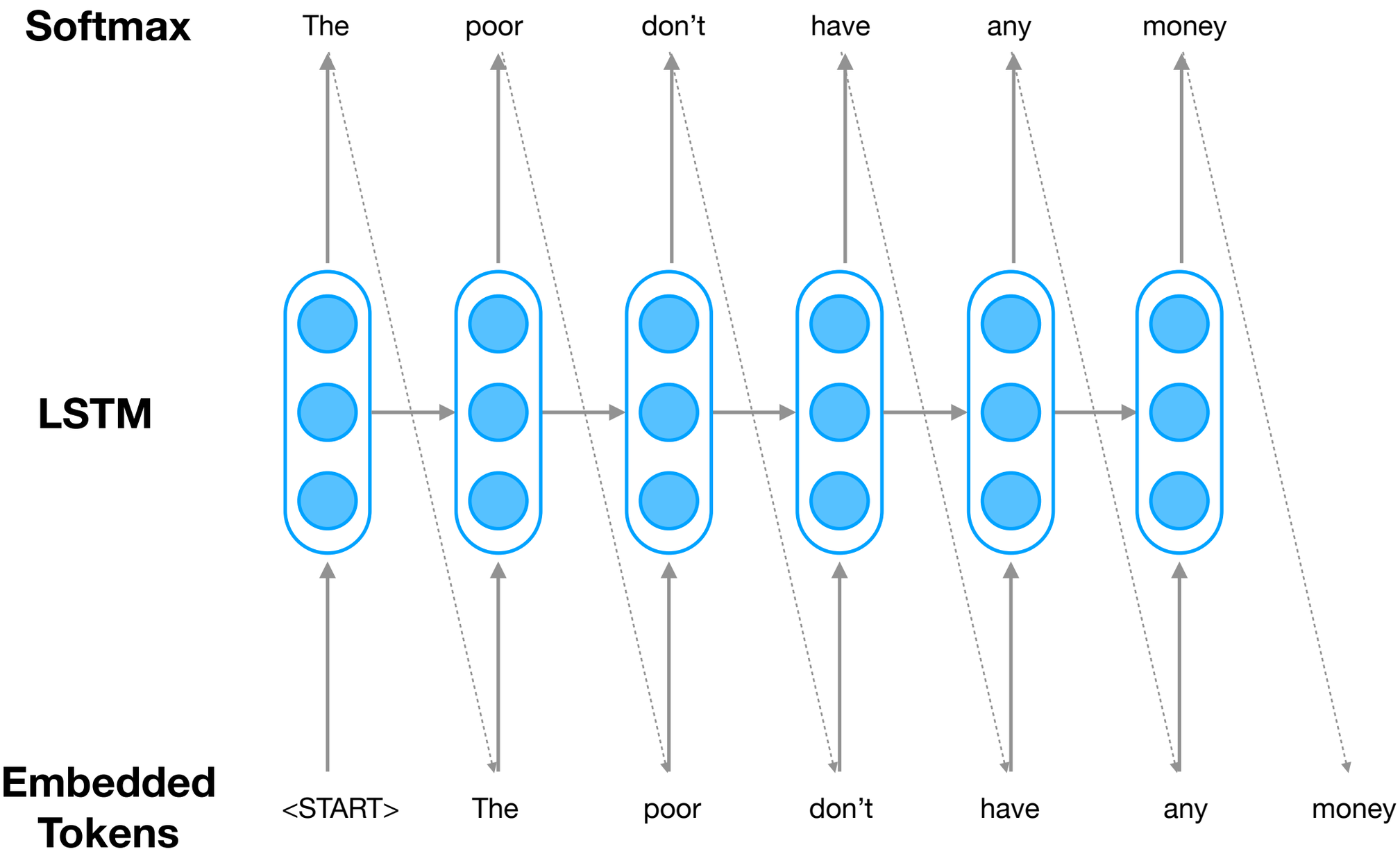
Language Model

Inference



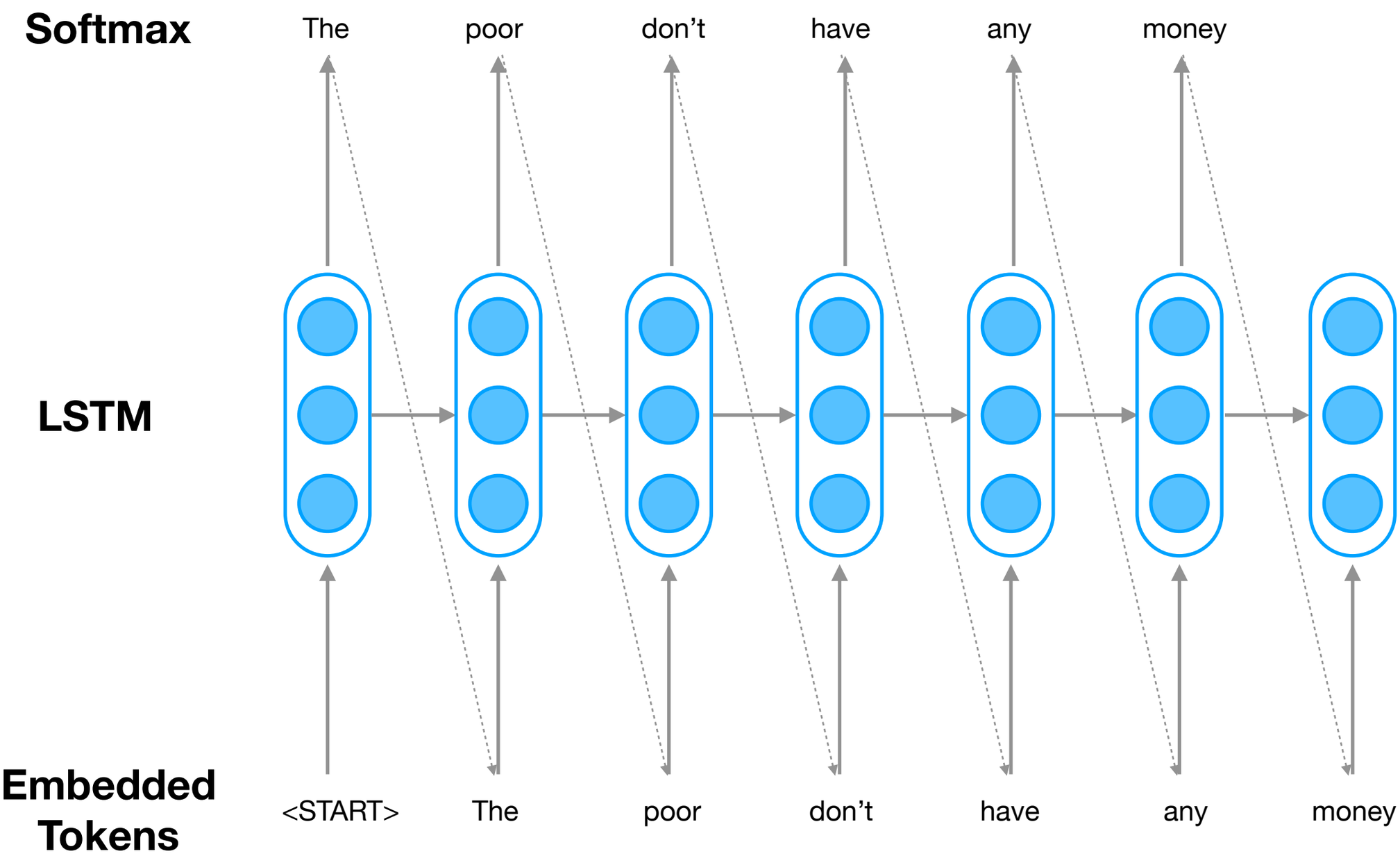
Language Model

Inference



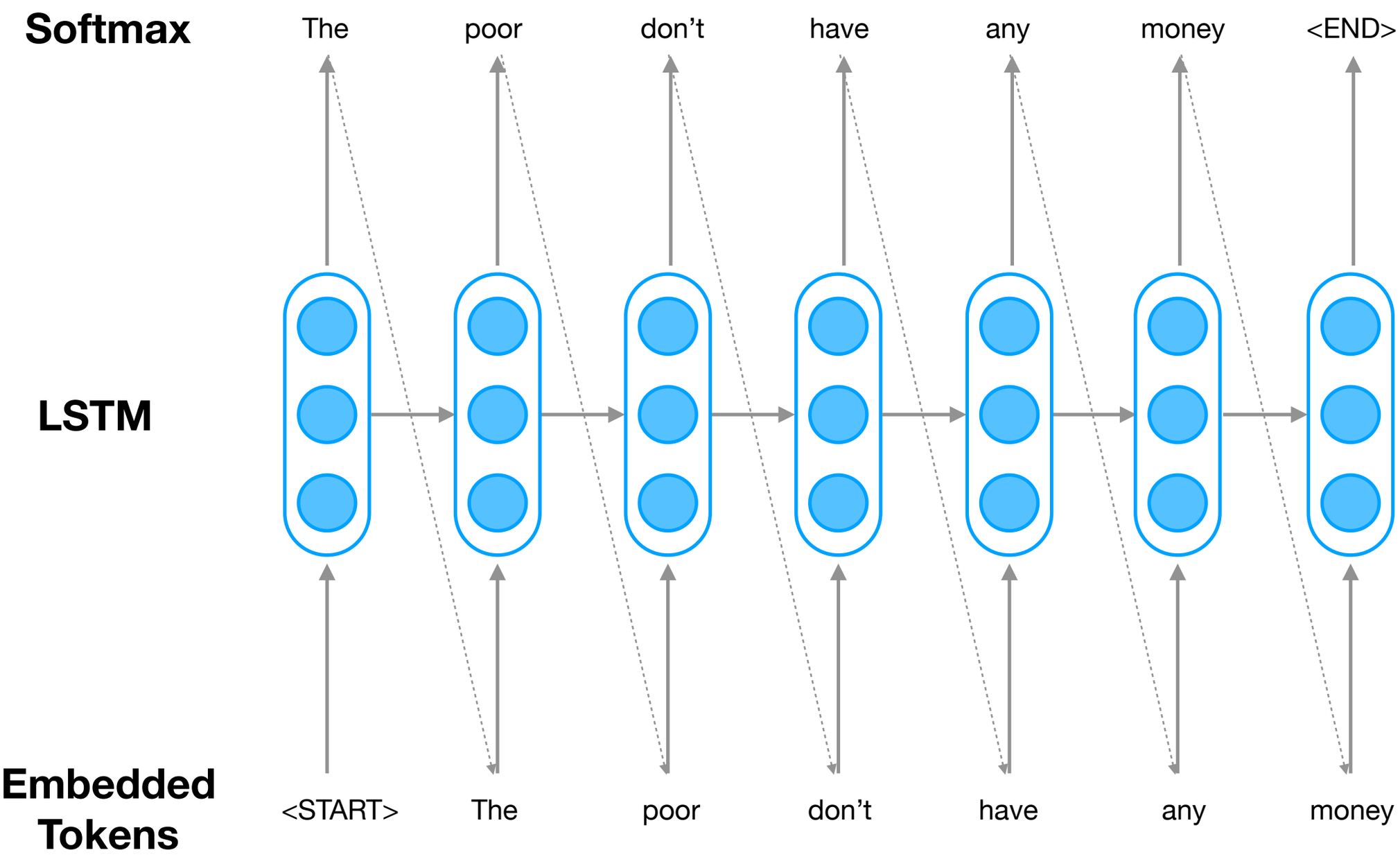
Language Model

Inference



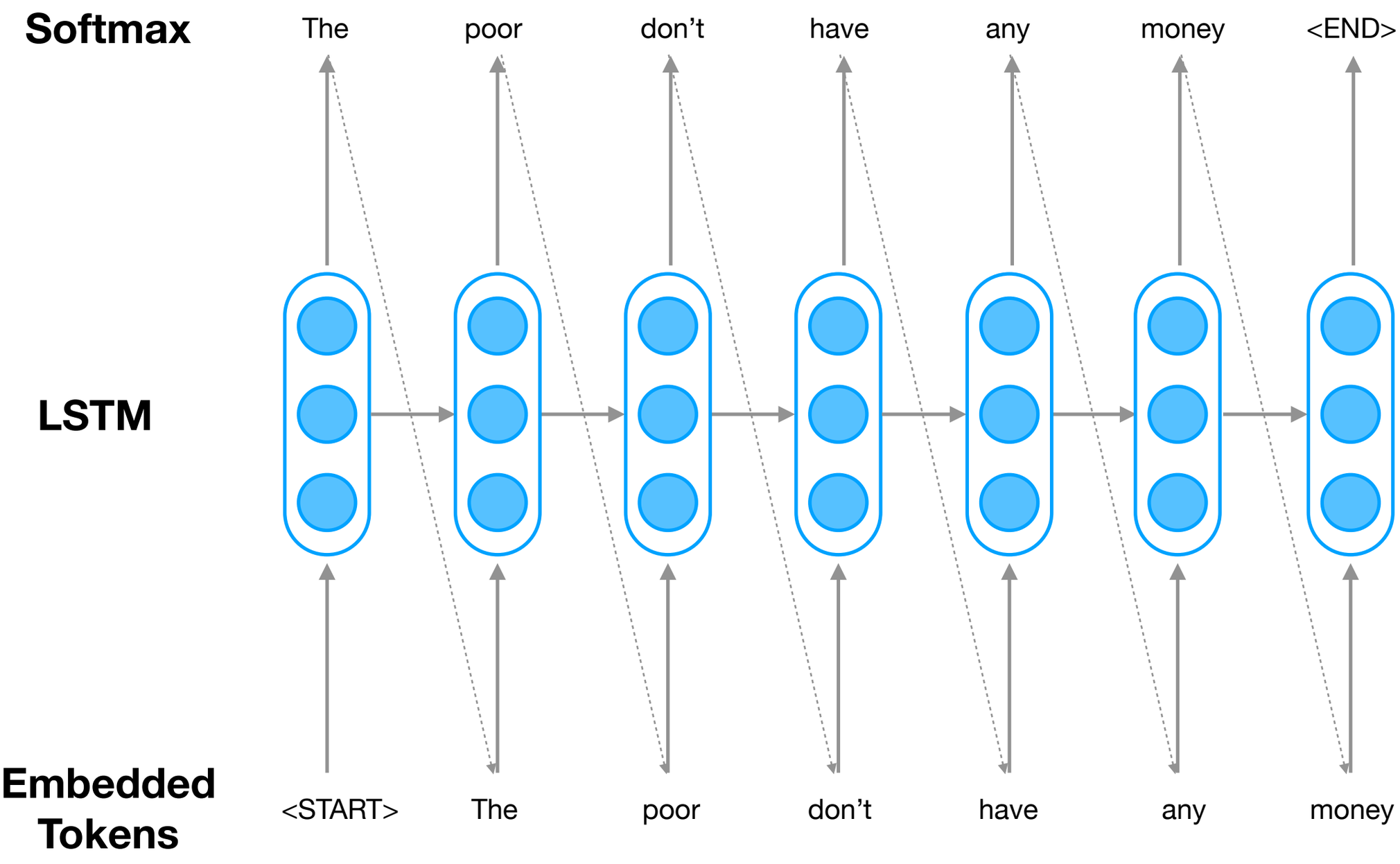
Language Model

Inference



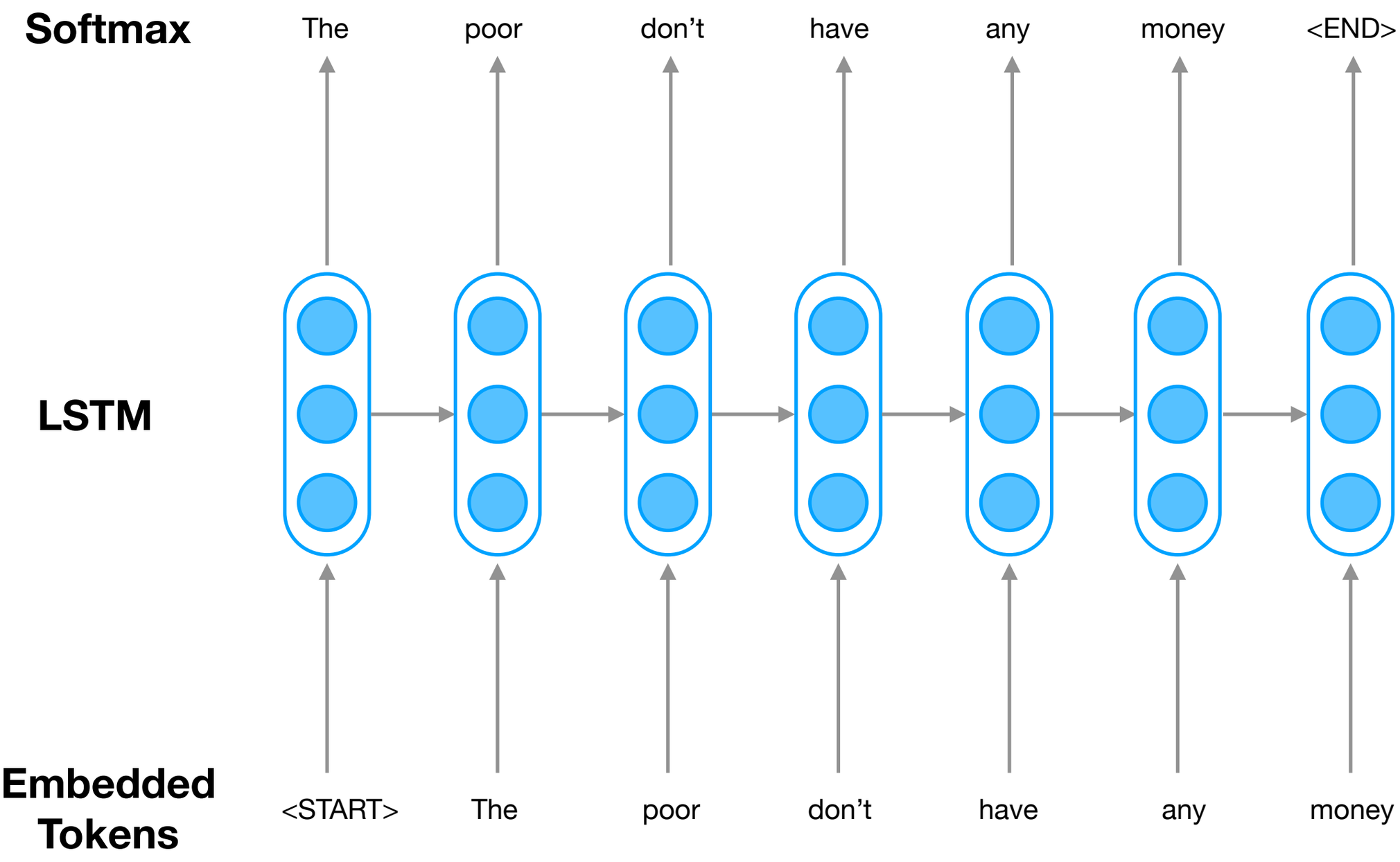
Language Model

$$p(\mathbf{x}) = \prod_i p(x|x_{<i}) = p(x_0)p(x_1|x_0)p(x_2|x_0, x_1)...$$

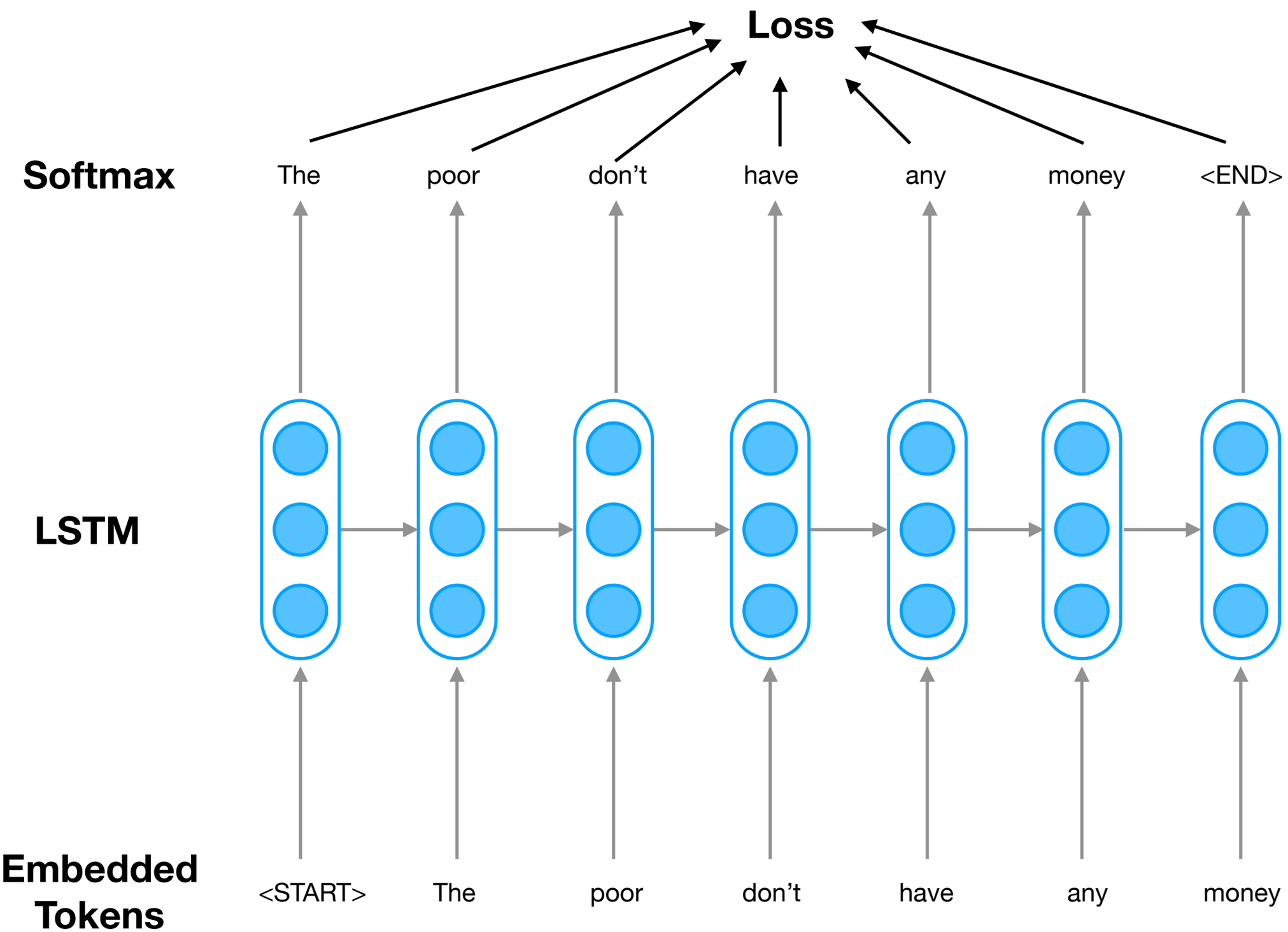


Language Model

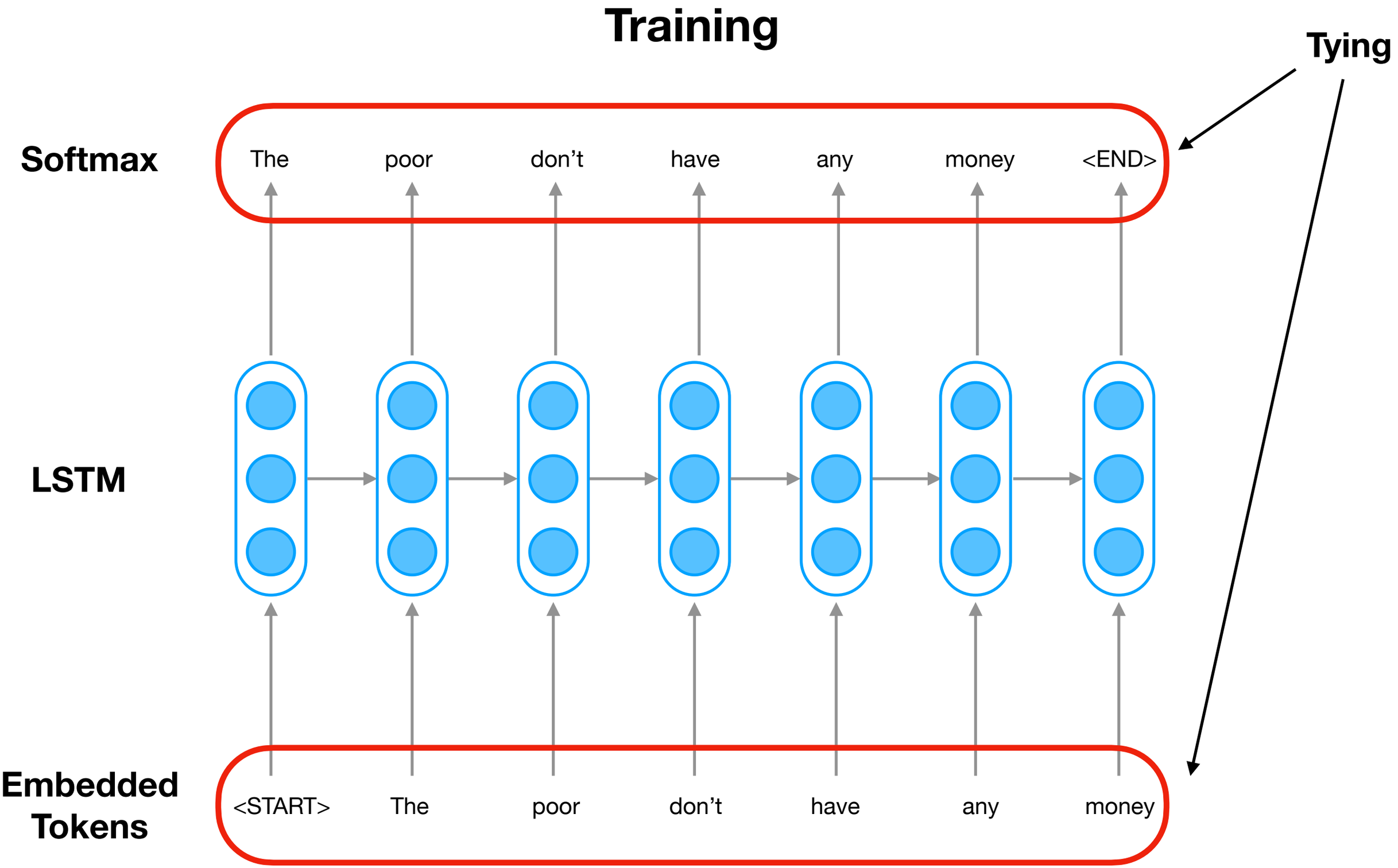
Training



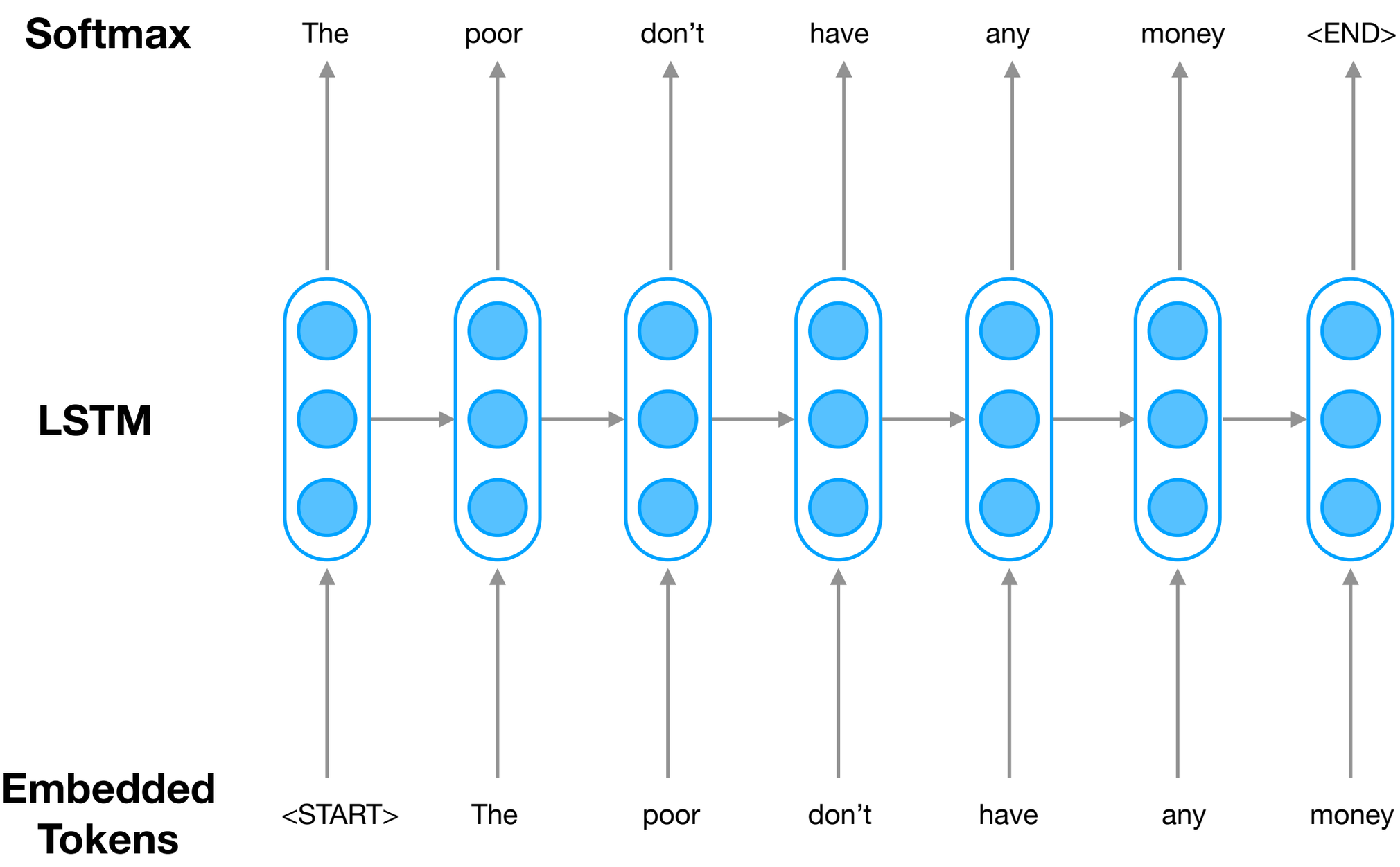
Language Model



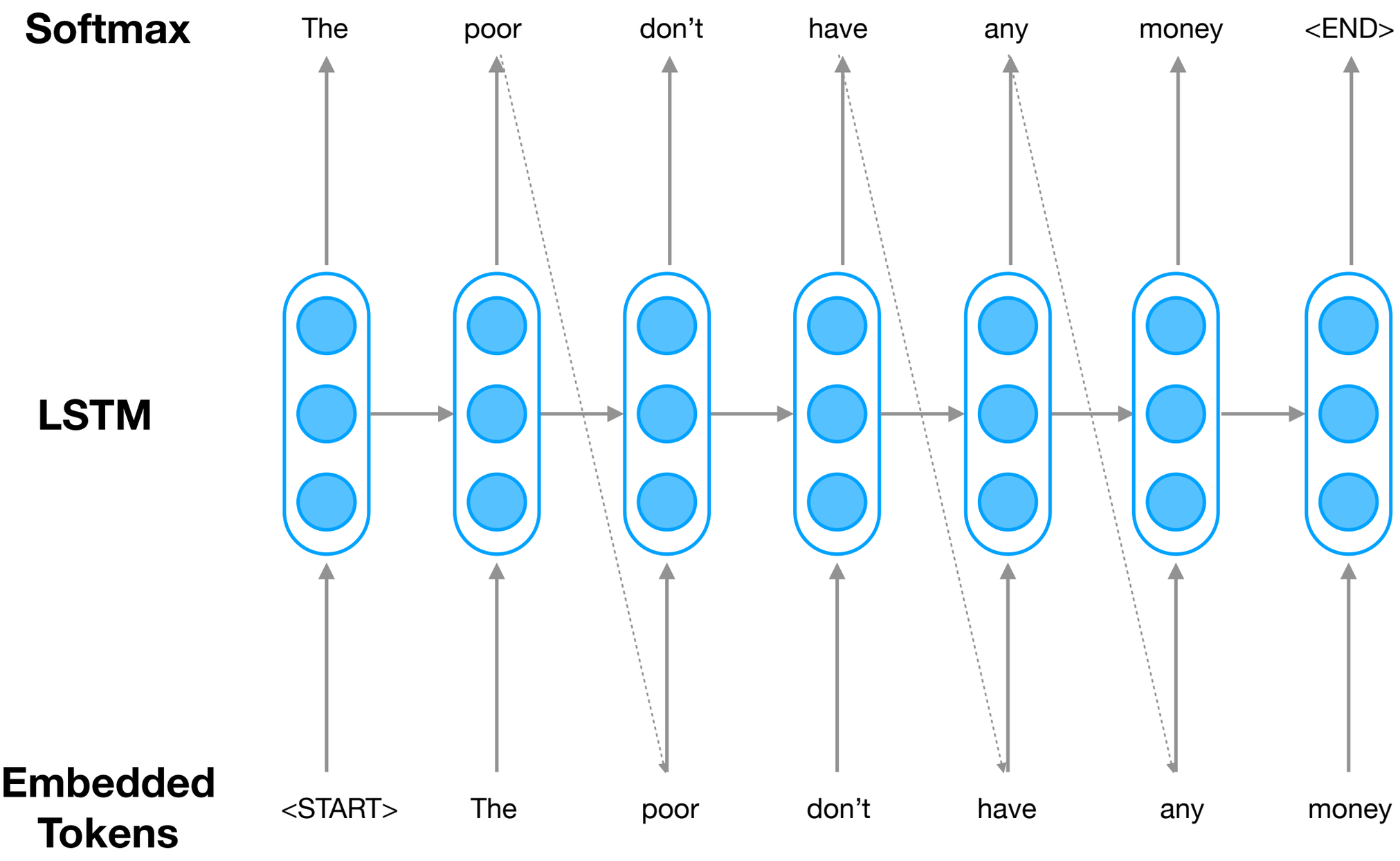
Language Model



Teacher Forcing



Teacher Forcing



Language Model

<START>	The	poor	don’t	have	any	money	<END>

Language Model

X	<START>
Y	The

Language Model

X	<START>	The
Y		poor

Language Model

X	<START>	The	poor	
Y				don’t

Language Model

X	<START>	The	poor	don’t	
Y					have

Language Model

X	<START>	The	poor	don’t	have	
Y						any

Language Model

X	<START>	The	poor	don’t	have	any	
Y							money

Language Model

X	<START>	The	poor	don’t	have	any	money		
Y									<END>

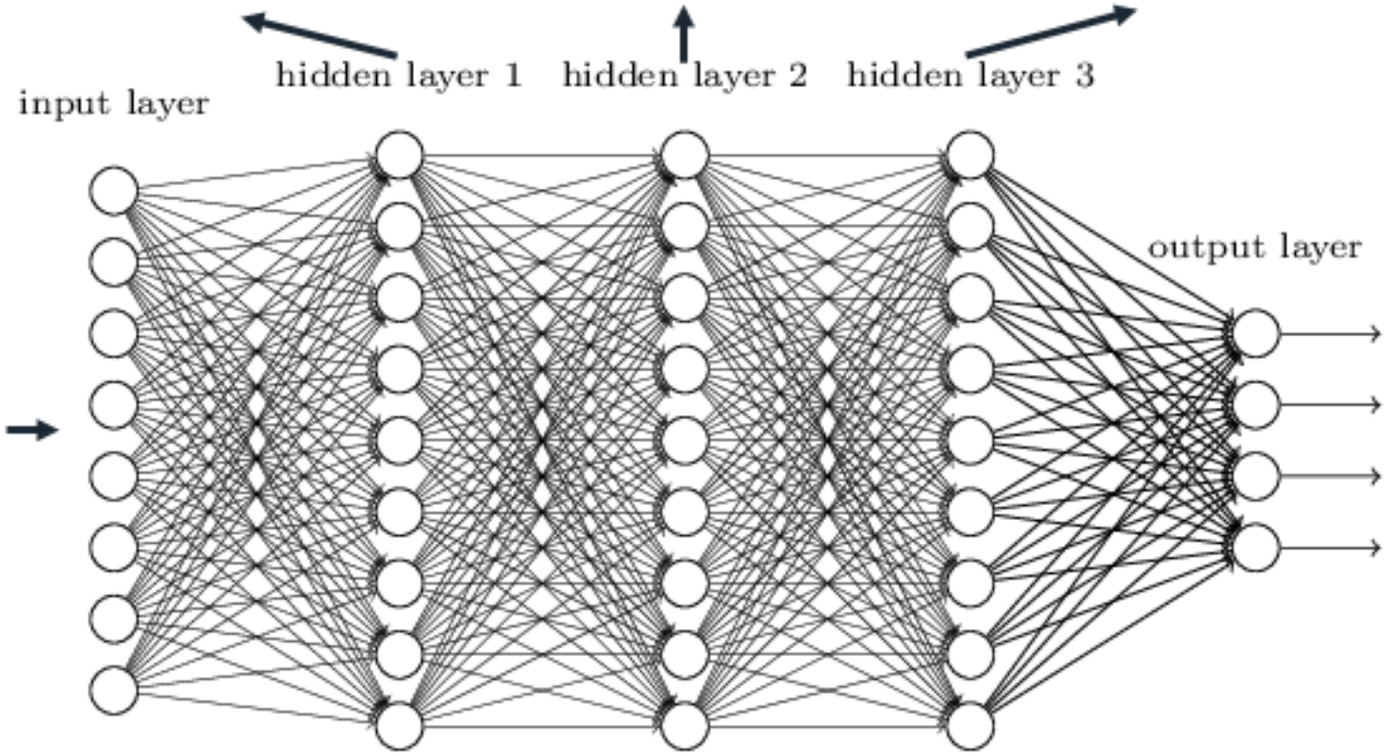
Language Model

X	<START>	The	poor	don’t	have	any	money	
Y								<END>

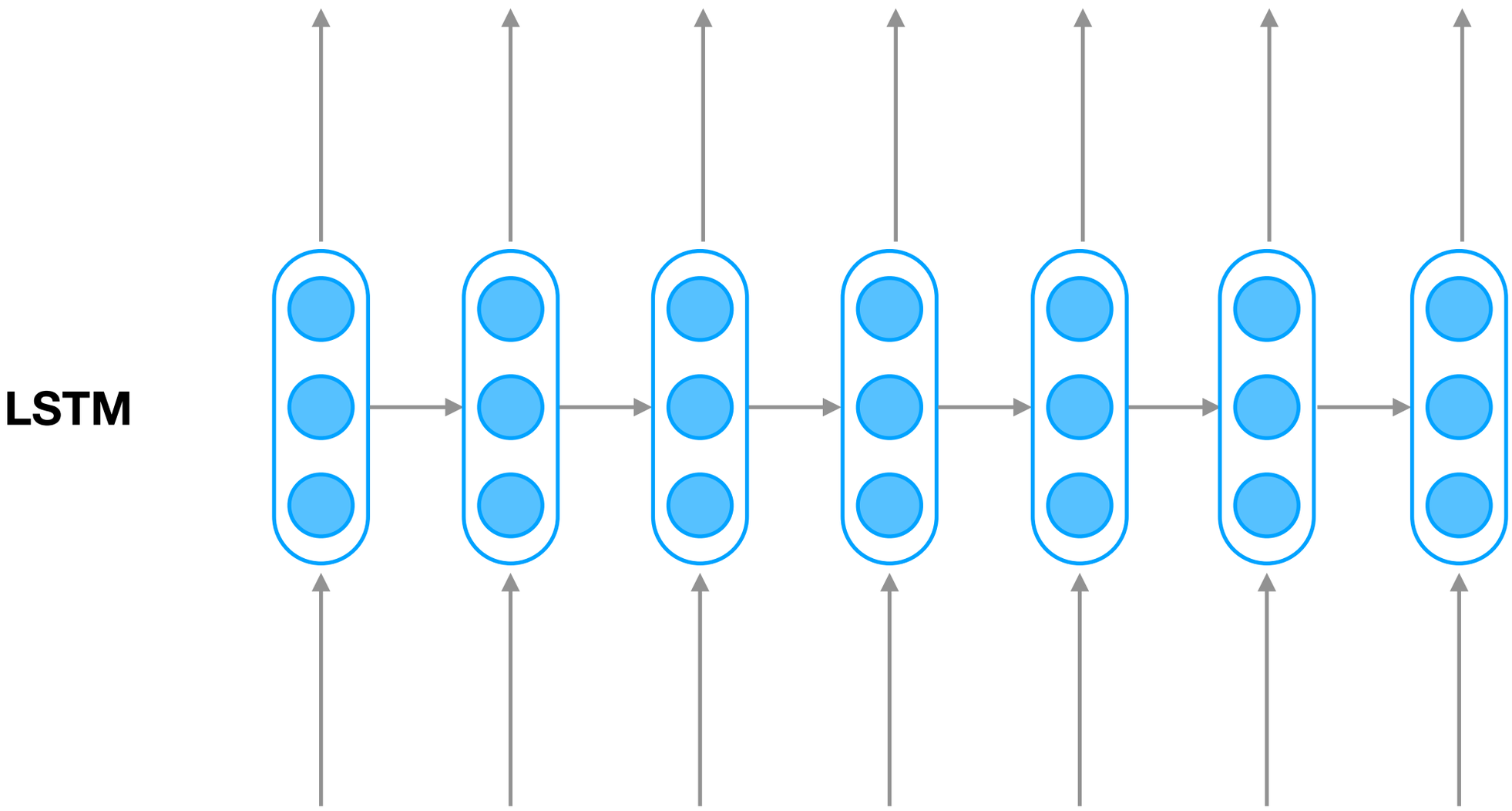
Sum of Losses

Transfer Learning

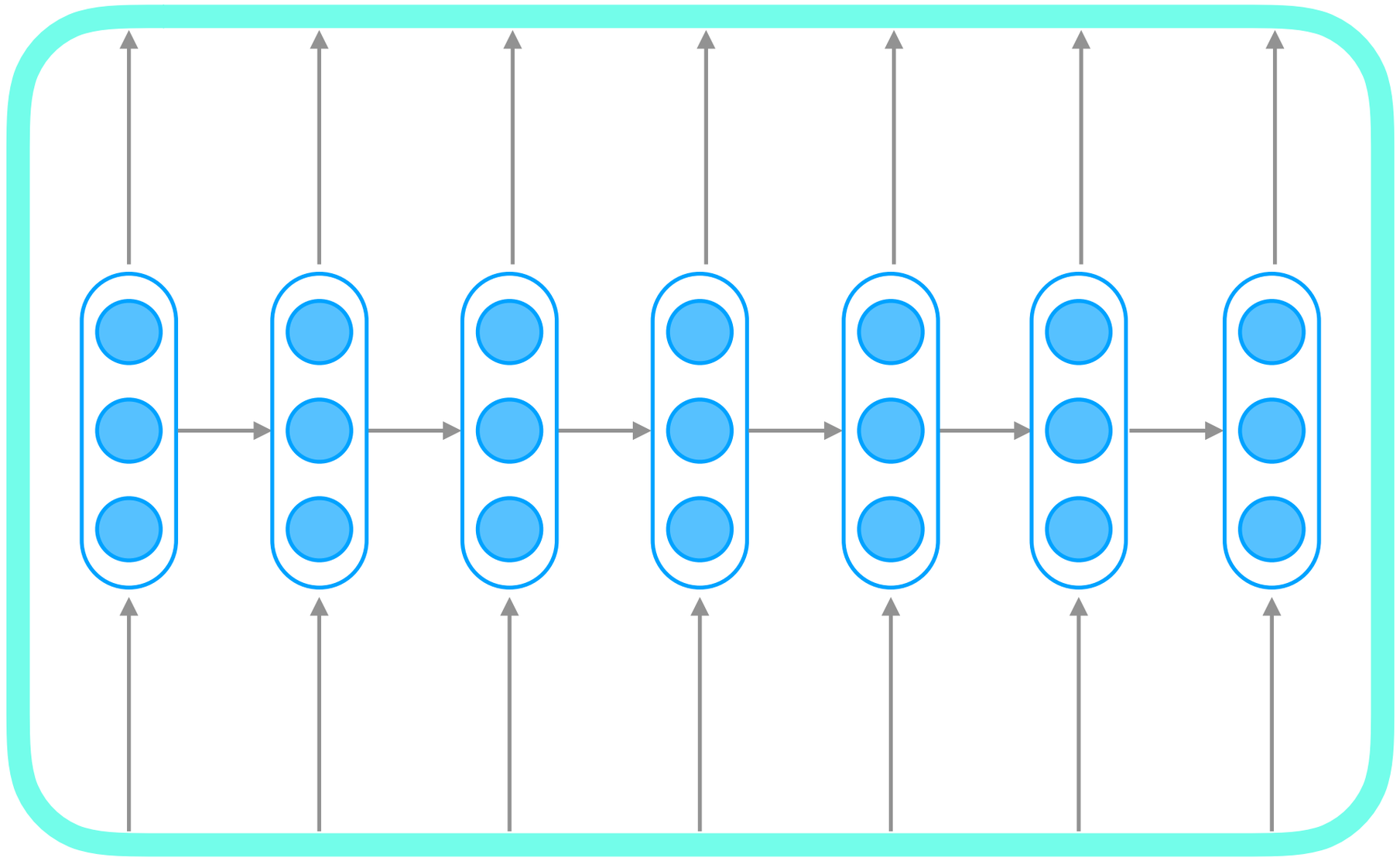
CNN Intuition



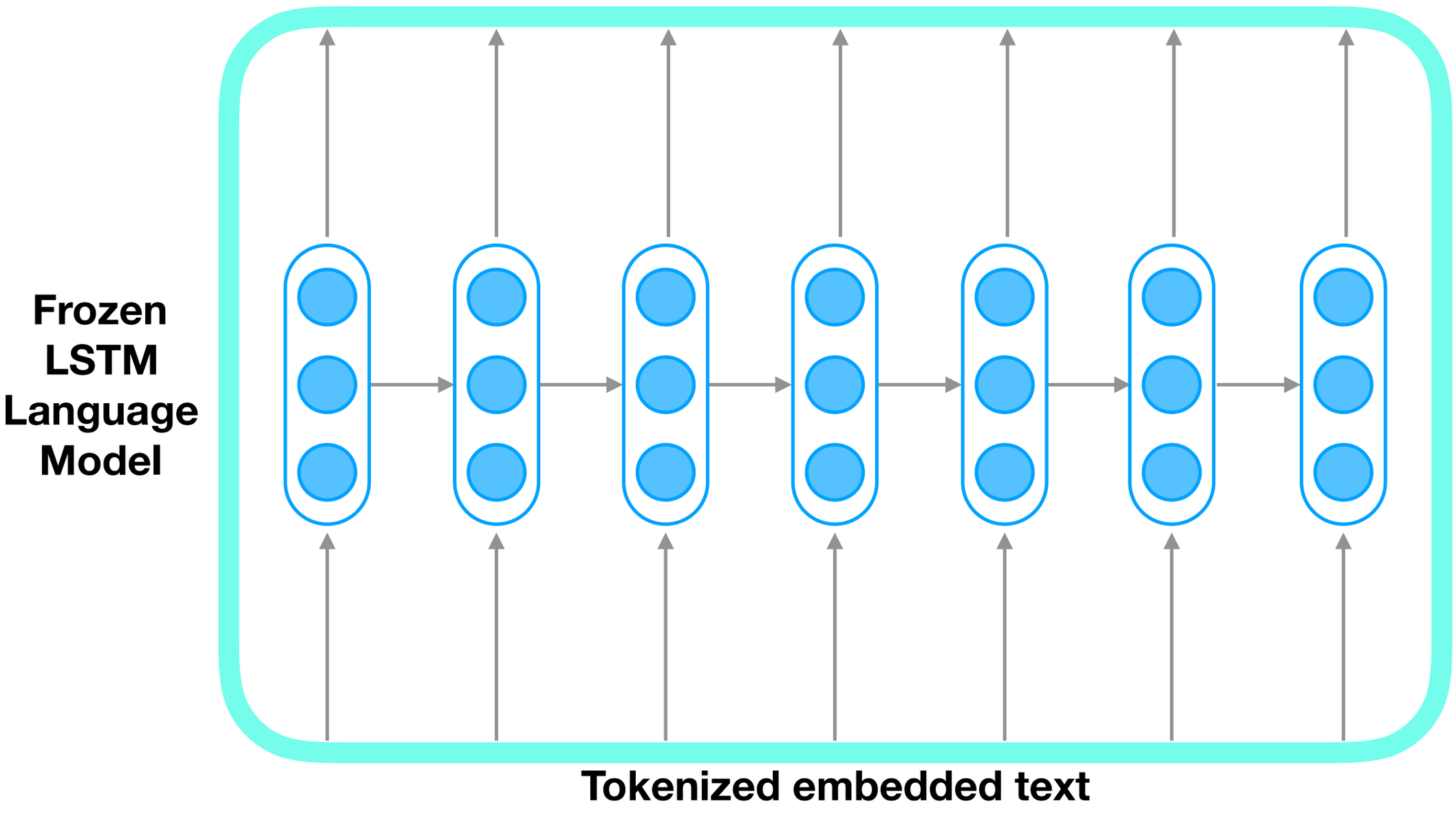
NLP LM Transfer Learning



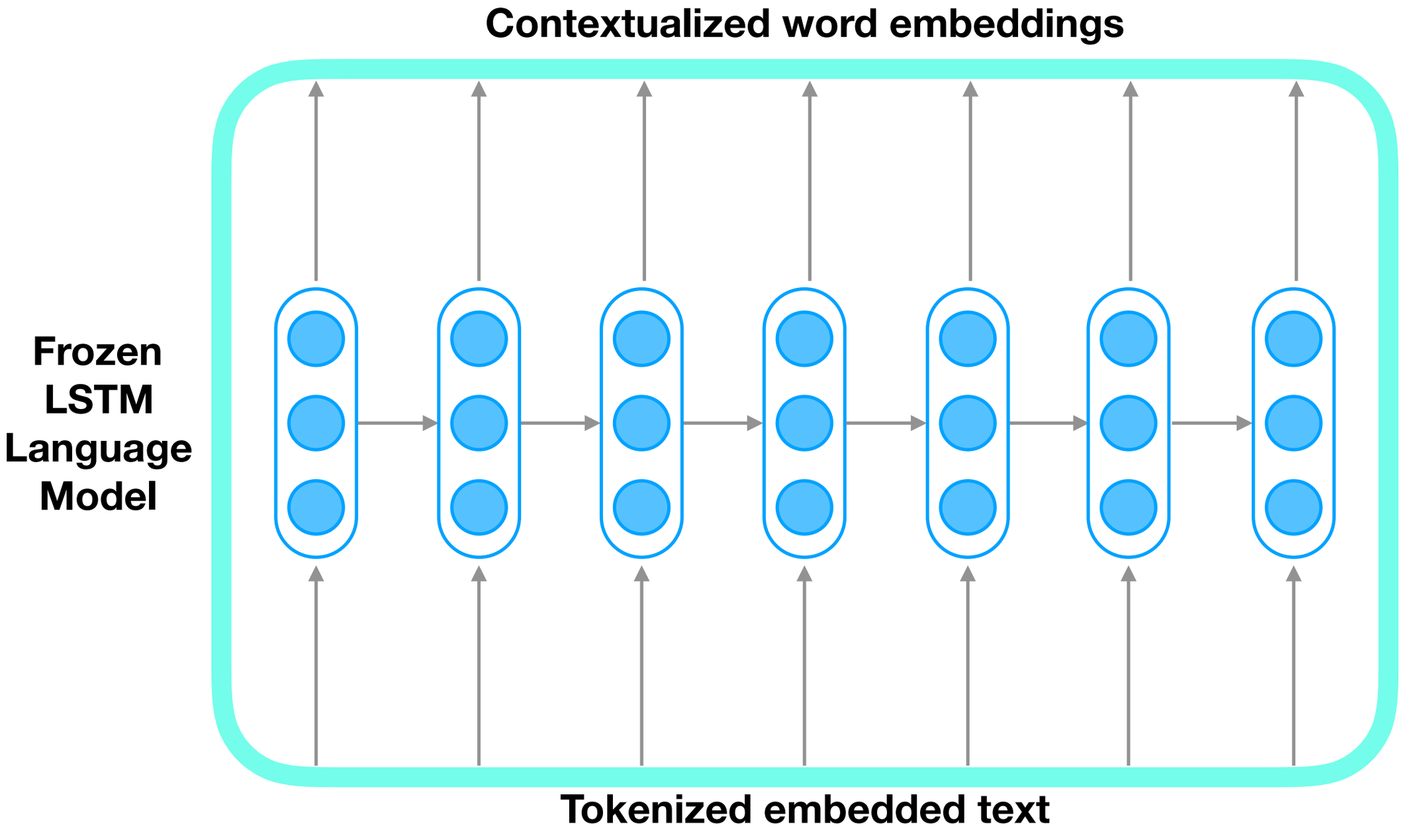
NLP LM Transfer Learning



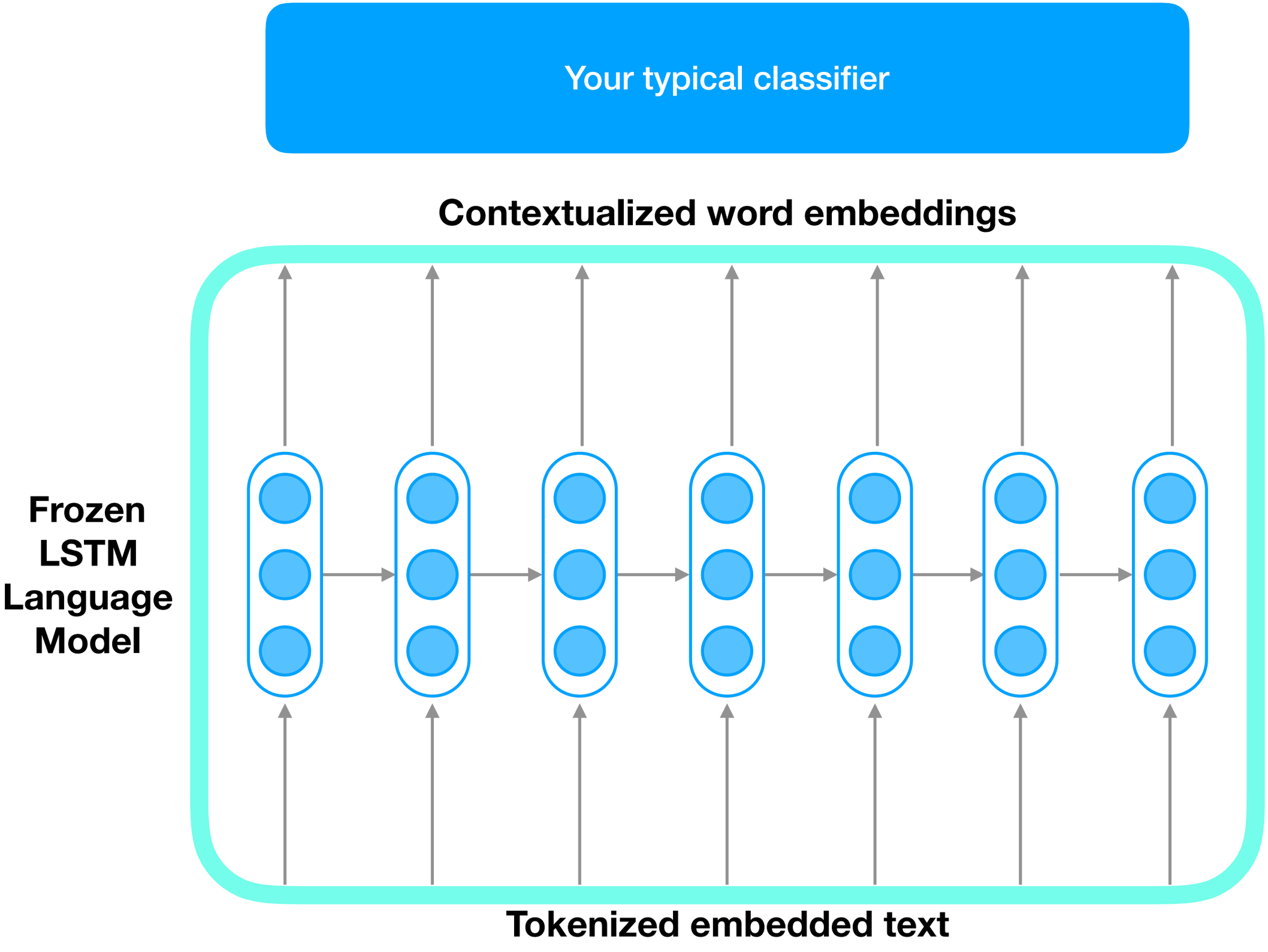
NLP LM Transfer Learning



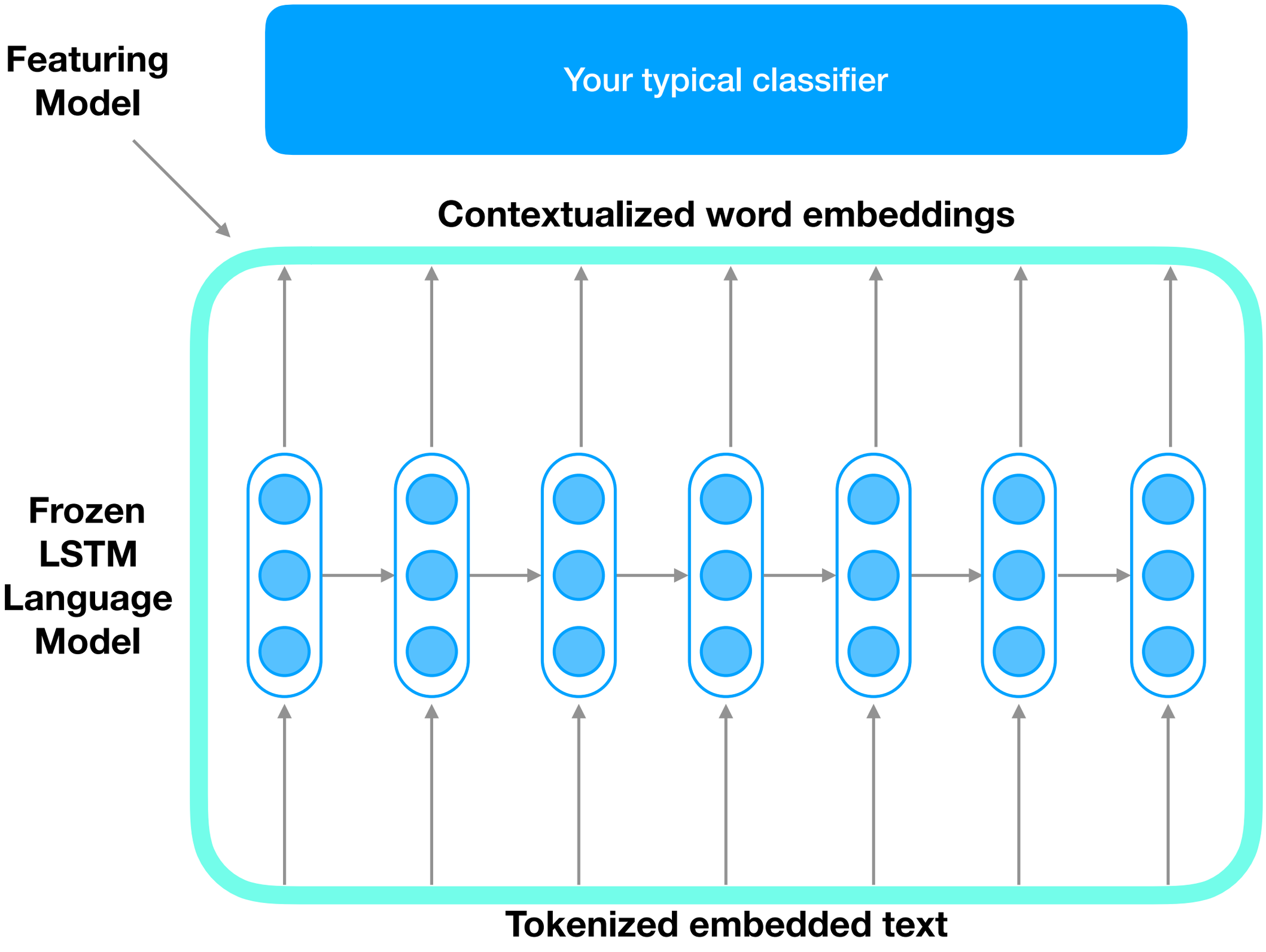
NLP LM Transfer Learning



NLP LM Transfer Learning

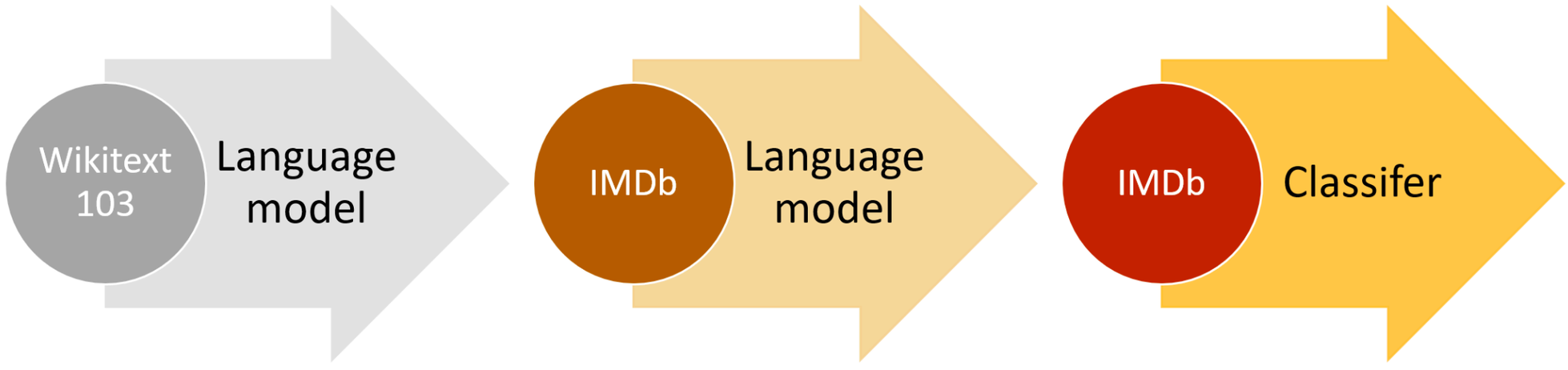


NLP LM Transfer Learning

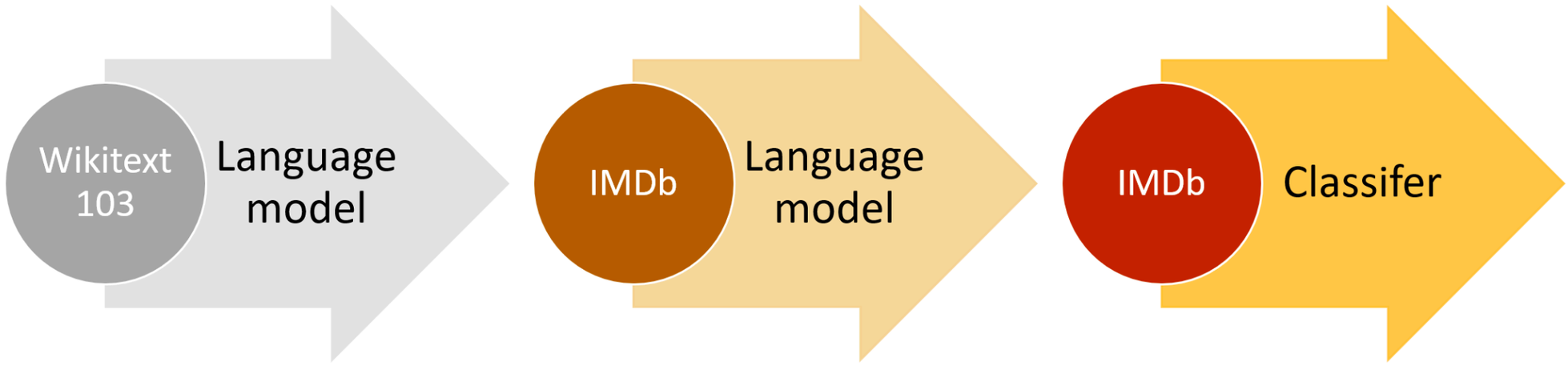


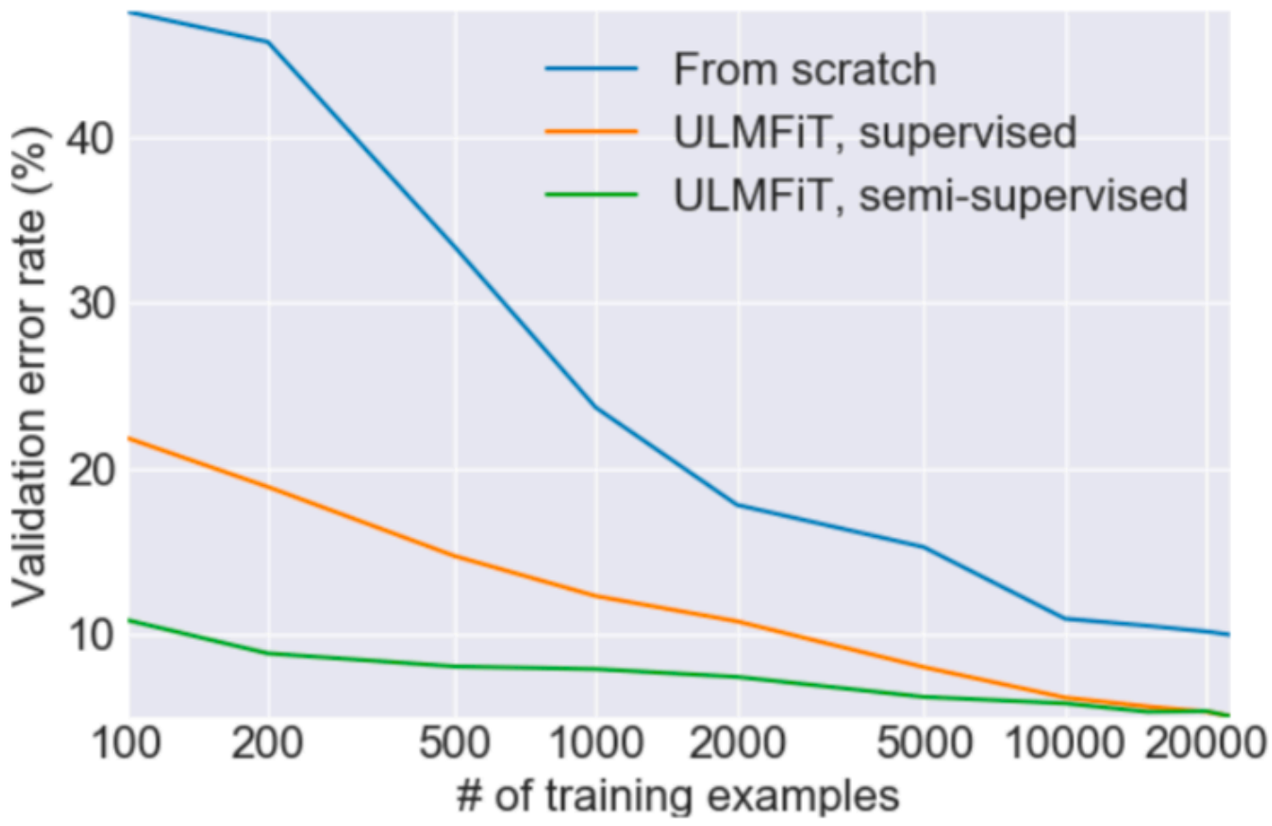
ULMFiT

ULMFiT


Wikitext 103
Language model
IMDb
Language model
IMDb
Classifer

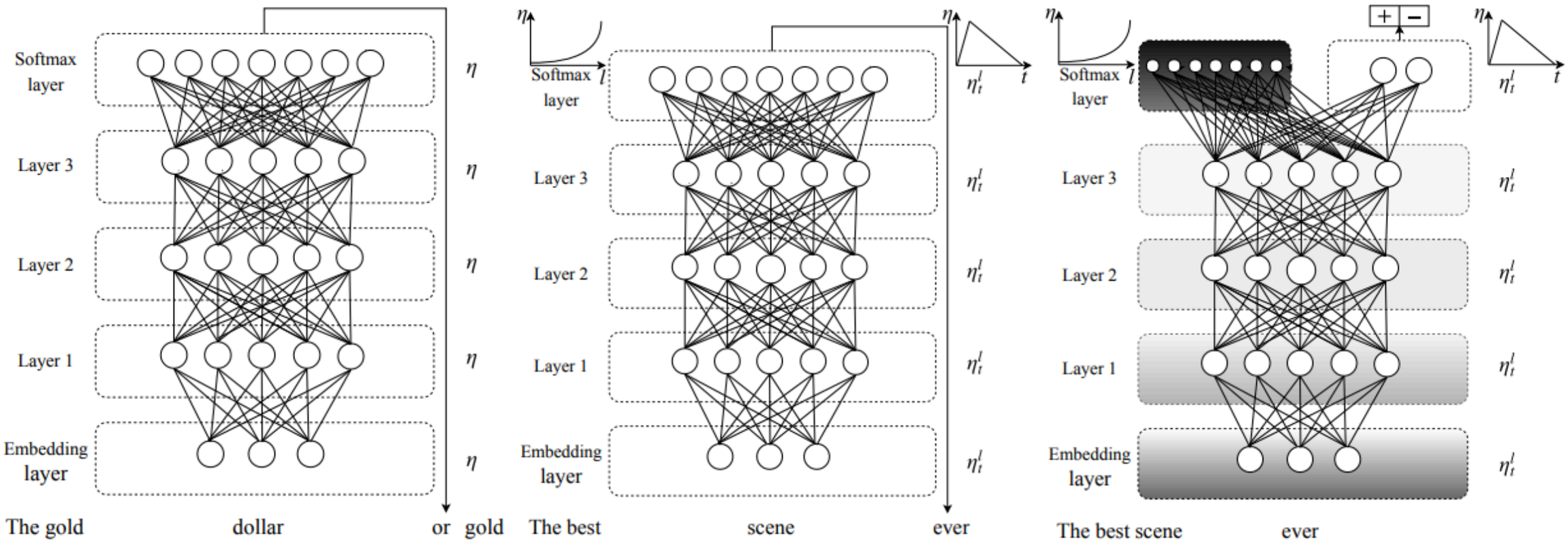
ULMFiT


Wikitext 103
Language model
IMDb
Language model
IMDb
Classifer


From scratch
ULMFiT, supervised
ULMFiT, semi-supervised
Validation error rate (%)
40
30
20
10
100
200
500
1000
2000
5000
10000
20000
# of training examples

ULMFiT

$$\mathbf{h}_c = [\mathbf{h}_T, \texttt{maxpool}(\mathbf{H}), \texttt{meanpool}(\mathbf{H})]$$



(a) LM pre-training (b) LM fine-tuning (c) Classifier fine-tuning

Figure 1: ULMFiT consists of three stages: a) The LM is trained on a general-domain corpus to capture general features of the language in different layers. b) The full LM is fine-tuned on target task data using discriminative fine-tuning ('*Discr*') and slanted triangular learning rates (STLR) to learn task-specific features. c) The classifier is fine-tuned on the target task using gradual unfreezing, '*Discr*', and STLR to preserve low-level representations and adapt high-level ones (shaded: unfreezing stages; black: frozen).

Scheduling

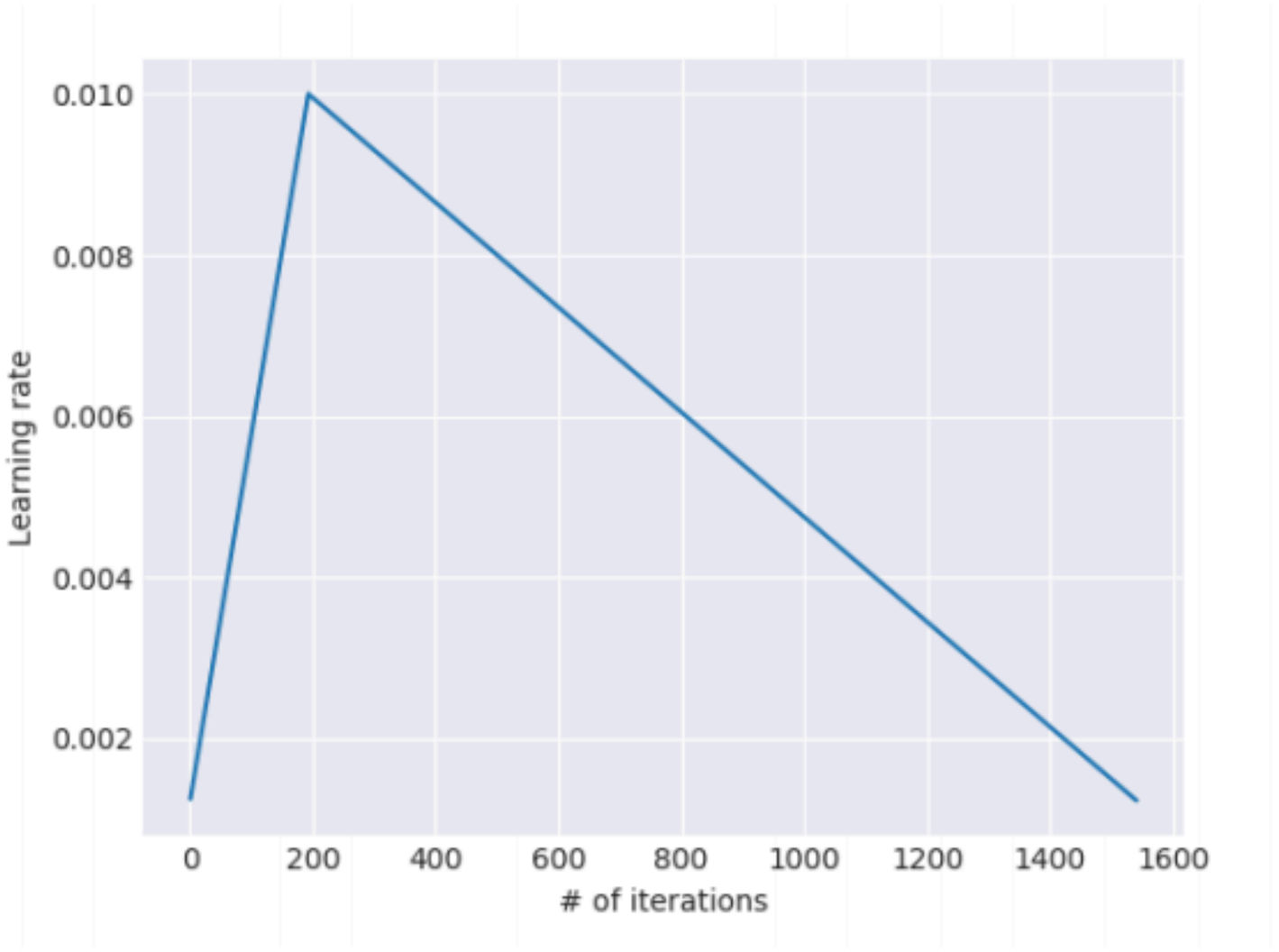
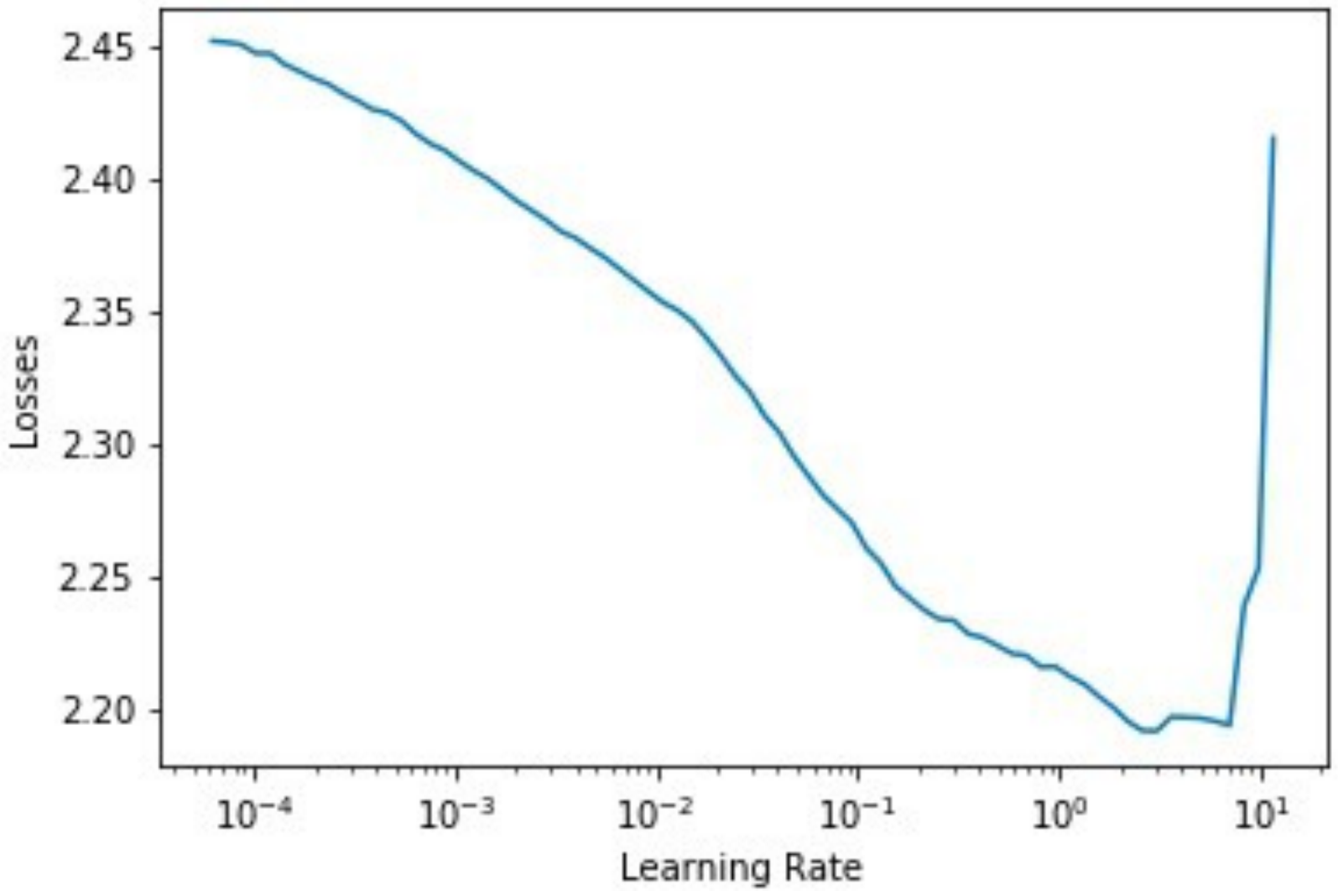


Figure 2: The slanted triangular learning rate schedule used for ULMFiT as a function of the number of training iterations.

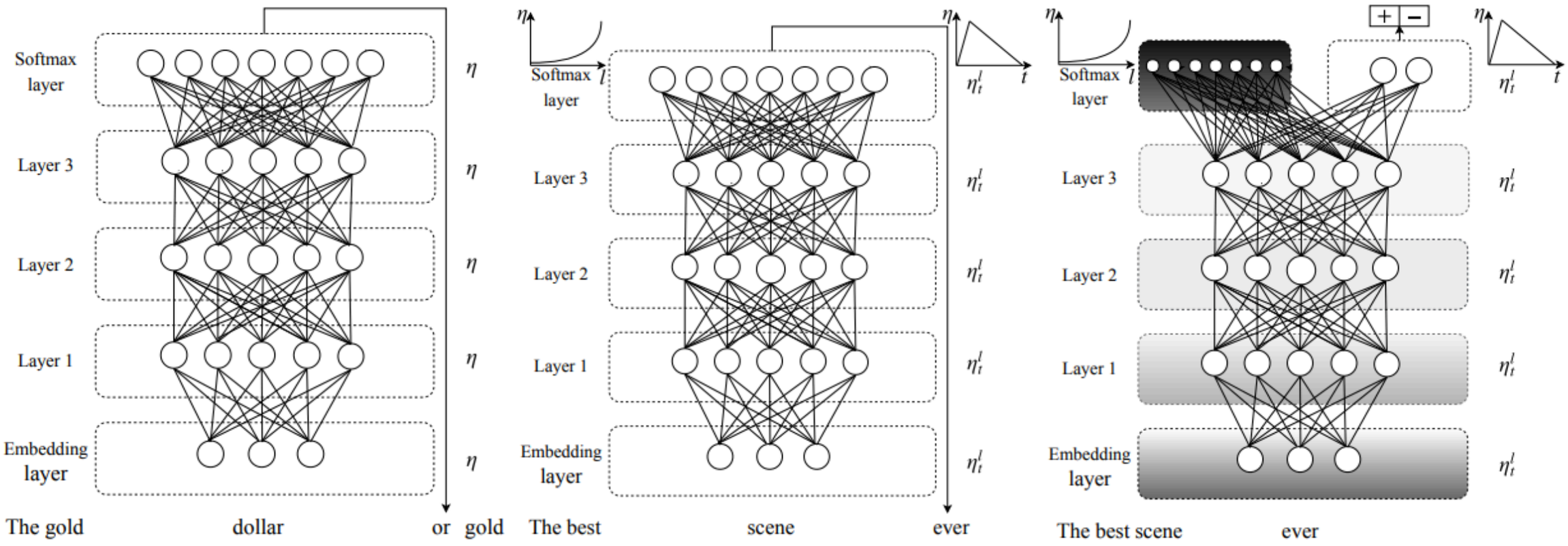
LR Finder


2.45
2.40
2.35
2.30
2.25
2.20
Losses
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
Learning Rate

ULMFiT

Gradual Unfreezing (Catastrophic Forgetting)

$$\mathbf{h}_c = [\mathbf{h}_T, \texttt{maxpool}(\mathbf{H}), \texttt{meanpool}(\mathbf{H})]$$



(a) LM pre-training (b) LM fine-tuning (c) Classifier fine-tuning

Figure 1: ULMFiT consists of three stages: a) The LM is trained on a general-domain corpus to capture general features of the language in different layers. b) The full LM is fine-tuned on target task data using discriminative fine-tuning ('*Discr*') and slanted triangular learning rates (STLR) to learn task-specific features. c) The classifier is fine-tuned on the target task using gradual unfreezing, '*Discr*', and STLR to preserve low-level representations and adapt high-level ones (shaded: unfreezing stages; black: frozen).

ULMFiT

Classifier

$$\mathbf{h}_c = [\mathbf{h}_T, \texttt{maxpool}(\mathbf{H}), \texttt{meanpool}(\mathbf{H})]$$

```
(1): PoolingLinearClassifier(
    (layers): Sequential(
      (0): BatchNorm1d(900, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (1): Dropout(p=0.4)
      (2): Linear(in_features=900, out_features=50, bias=True)
      (3): ReLU(inplace)
      (4): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): Dropout(p=0.1)
      (6): Linear(in_features=50, out_features=2, bias=True)
    )
)
```

AWD-LSTM

ENCODER DROPOUT										
	before dropout					after dropout				
	token	d1	d2	...		token	d1	d2	...	
	I	0.399	0.75379	0.62616		I	0	0	0	
	love	0.88533	0.29449	0.15856		love	0.88533	0.29449	0.15856	
	cats	0.48927	0.04071	0.21427		cats	0.48927	0.04071	0.21427	
	dogs	0.72918	0.86882	0.77136		dogs	0.72918	0.86882	0.77136	

AWD-LSTM

ENCODER DROPOUT

	before dropout				after dropout			
	token	d1	d2	...	**token**	d1	d2	...
	I	0.399	0.75379	0.62616	I	0	0	0
	love	0.88533	0.29449	0.15856	love	0.88533	0.29449	0.15856
	cats	0.48927	0.04071	0.21427	cats	0.48927	0.04071	0.21427
	dogs	0.72918	0.86882	0.77136	dogs	0.72918	0.86882	0.77136

INPUT DROPOUT

batch: [I love cats, I love dogs]

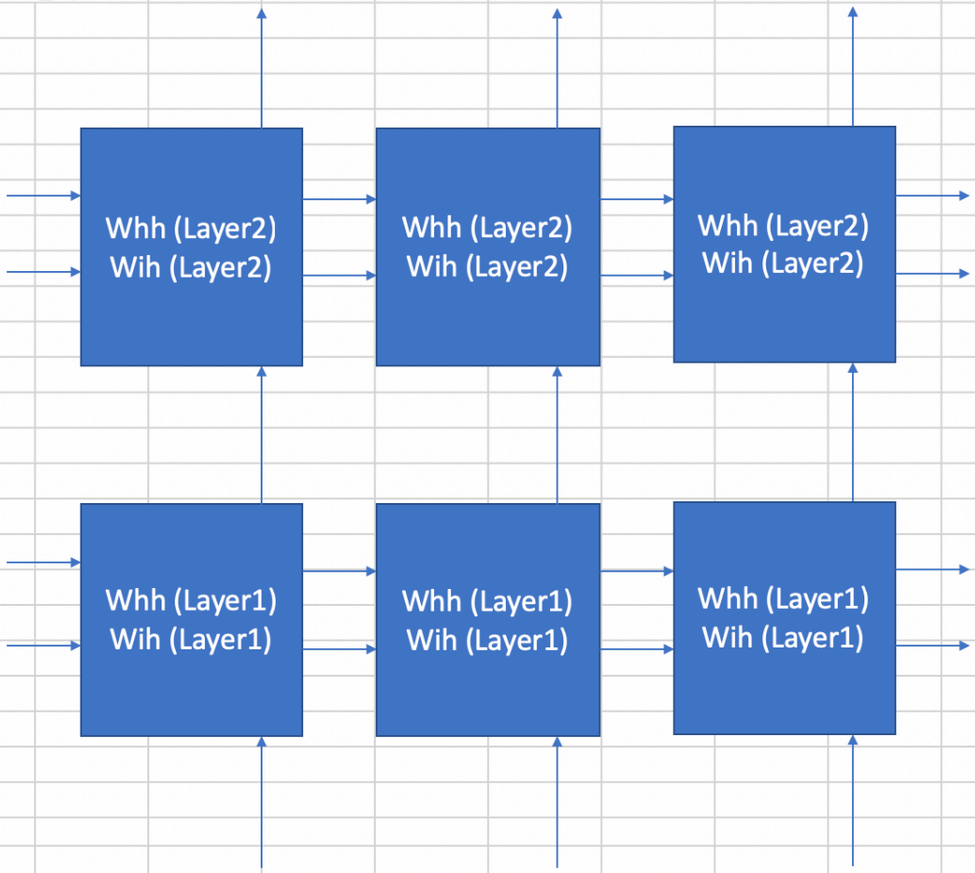
	before dropout				after dropout			
	I	0	0	0	I	0	0	0
	love	0.88533	0.29449	0.15856	love	0	0.29449	0.15856
	cats	0.48927	0.04071	0.21427	cats	0	0.04071	0.21427

	before dropout				after dropout			
	I	0	0	0	I	0	0	0
	love	0.88533	0.29449	0.15856	love	0.88533	0.29449	0
	dogs	0.72918	0.86882	0.77136	dogs	0.72918	0.86882	0

AWD-LSTM

WEIGHT DROPOUT

n_hid=5, n_layers=2



I	I
0	0
0	0
0	0

love	love
0	0.88533
0.29449	0.29449
0.15856	0

cats	dogs
0	0.72918
0.04071	0.86882
0.21427	0

before dropout

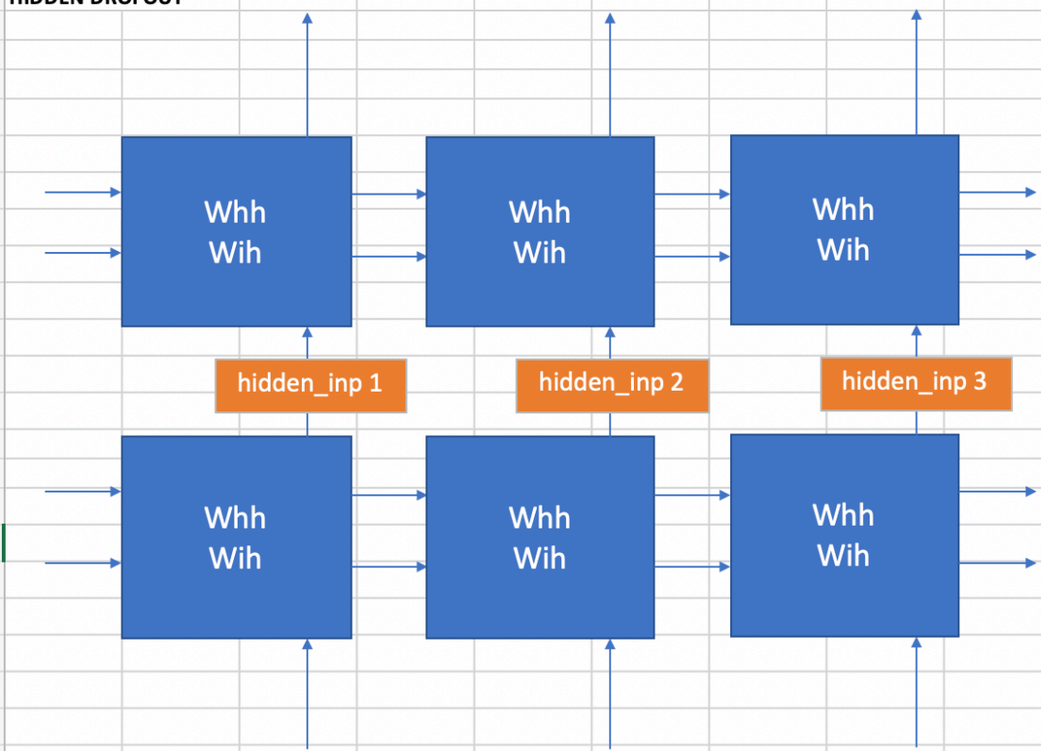
Whh (Layer1)				
0.30546	0.94294	0.66715	0.5655	0.31191
0.06246	0.36674	0.35672	0.53357	0.04652
0.84278	0.17049	0.92283	0.97827	0.67913
0.94699	0.02277	0.79537	0.09479	0.16284
0.34098	0.09256	0.69661	0.22605	0.57153
0.09455	0.78172	0.48226	0.82222	0.25151
0.92516	0.50419	0.82879	0.67049	0.84304
0.11741	0.09362	0.69367	0.5633	0.34755
0.98025	0.73131	0.21491	0.86319	0.31456
0.0289	0.90899	0.30697	0.75105	0.62107
0.96808	0.4712	0.77992	0.83567	0.09754
0.15539	0.94866	0.51977	0.51167	0.08723
0.10944	0.88881	0.74007	0.96823	0.03516
0.62014	0.4363	0.02097	0.61126	0.30466
0.10465	0.02462	0.2334	0.25372	0.54579
0.14384	0.962	0.70157	0.39285	0.41064
0.42081	0.64236	0.06941	0.41805	0.78772
0.45229	0.9871	0.49831	0.17108	0.48868
0.42142	0.66105	0.53273	0.30901	0.45095
0.33421	0.77472	0.33966	0.42693	0.4567

after dropout

Whh (Layer1)				
0.30546	0	0.66715	0.5655	0
0	0.36674	0.35672	0.53357	0.04652
0.84278	0.17049	0.92283	0.97827	0.67913
0.94699	0.02277	0.79537	0.09479	0.16284
0.34098	0.09256	0.69661	0.22605	0.57153
0	0.78172	0.48226	0.82222	0.25151
0	0.50419	0	0.67049	0.84304
0	0.09362	0.69367	0.5633	0
0.98025	0.73131	0	0	0.31456
0	0	0.30697	0.75105	0.62107
0	0	0.77992	0.83567	0.09754
0.15539	0	0.51977	0.51167	0.08723
0.10944	0.88881	0.74007	0.96823	0.03516
0.62014	0.4363	0.02097	0.61126	0.30466
0.10465	0.02462	0.2334	0	0.54579
0	0.962	0.70157	0	0.41064
0	0	0.06941	0.41805	0.78772
0.45229	0.9871	0.49831	0	0.48868
0.42142	0.66105	0.53273	0.30901	0.45095
0	0.77472	0.33966	0.42693	0.4567

AWD-LSTM

HIDDEN DROPOUT



before dropout

	I	0.11587	0.89354	0.18537	0.13159	0.04328	0.36997	0.68241	0.21505	0.07696	0.1405
	love	0.51854	0.94156	0.80541	0.49411	0.44335	0.31453	0.20678	0.95815	0.70224	0.2766
	cats	0.41255	0.70882	0.76667	0.93813	0.89034	0.1676	0.92944	0.32216	0.11675	0.99425

after dropout

hidden_inp 1	I	0.11587	0	0.18537	0.13159	0	0.36997	0.68241	0	0	0.1405
hidden_inp 2	love	0.51854	0	0.80541	0.49411	0	0.31453	0.20678	0	0	0.2766
hidden_inp 3	cats	0.41255	0	0.76667	0.93813	0	0.1676	0.92944	0	0	0.99425

before dropout

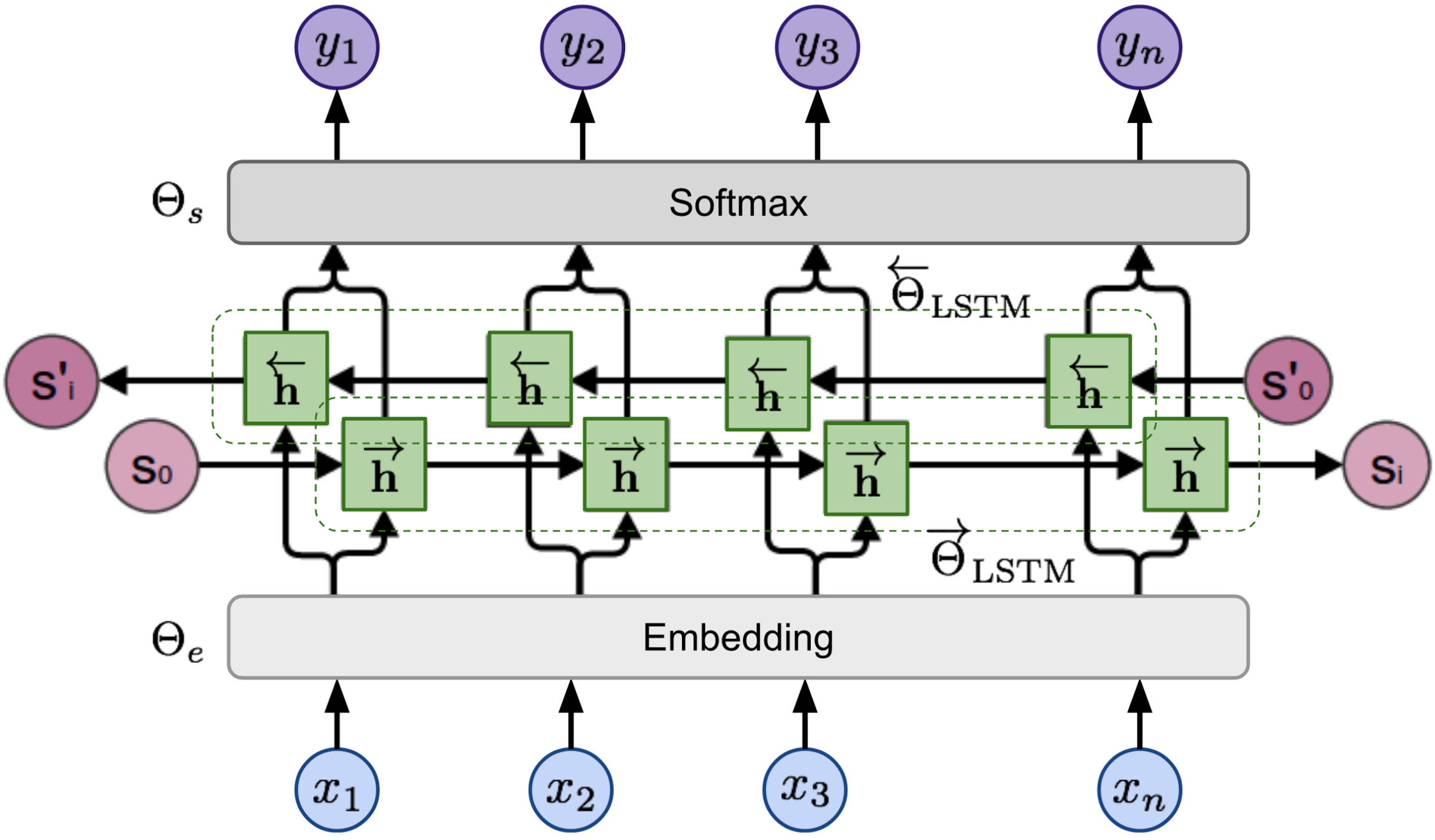
	I	0.90644	0.01673	0.38366	0.86887	0.26845	0.332	0.93804	0.34769	0.83711	0.66387
	love	0.82539	0.09143	0.32444	0.35099	0.64491	0.79199	0.24797	0.56466	0.46948	0.33358
	dogs	0.38805	0.56548	0.1424	0.31113	0.11645	0.36759	0.73068	0.05727	0.7315	0.01498

after dropout

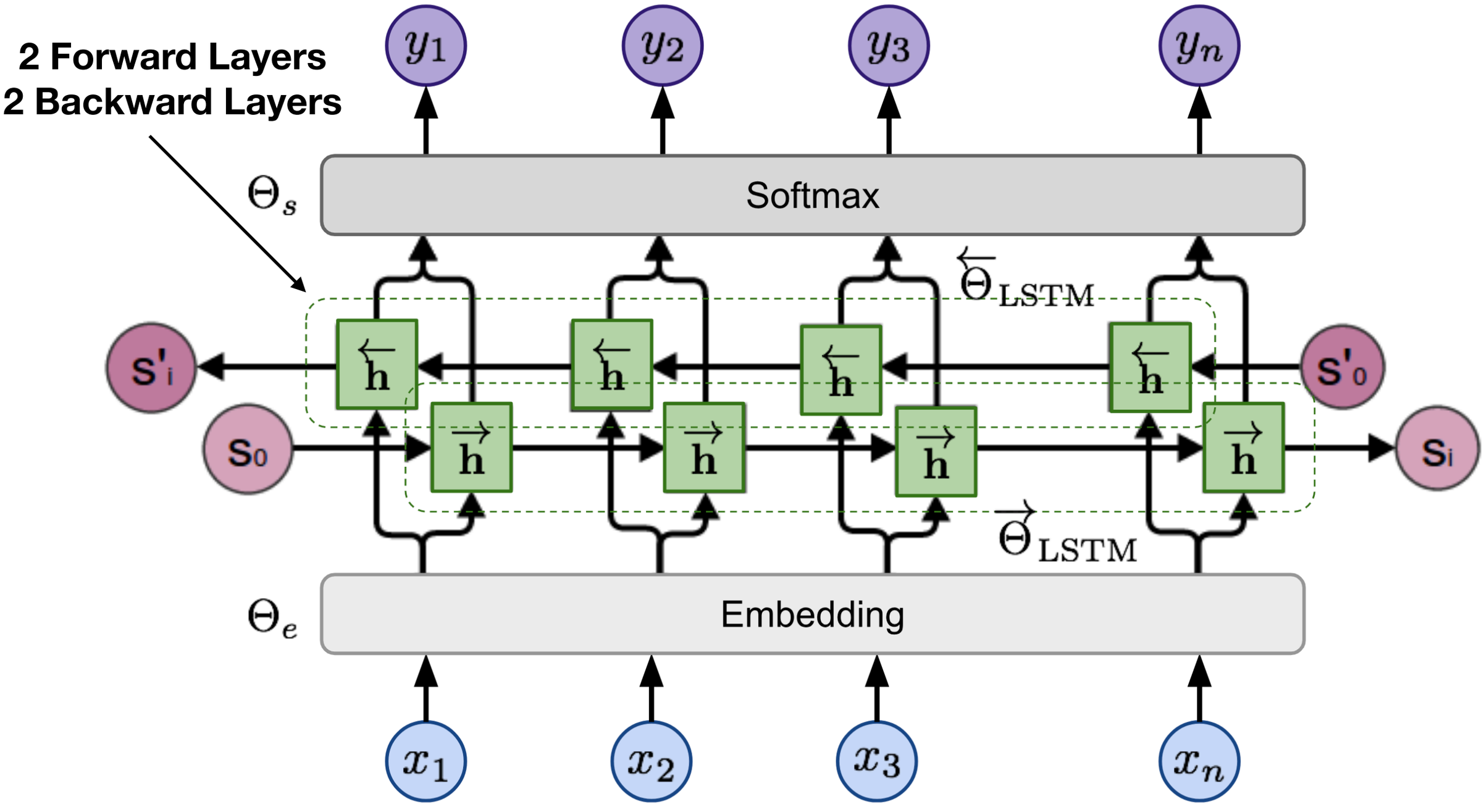
hidden_inp 1	I	0	0	0.38366	0.86887	0	0.332	0	0.34769	0.83711	0.66387
hidden_inp 2	love	0	0	0.32444	0.35099	0	0.79199	0	0.56466	0.46948	0.33358
hidden_inp 3	dogs	0	0	0.1424	0.31113	0	0.36759	0	0.05727	0.7315	0.01498

ELMo

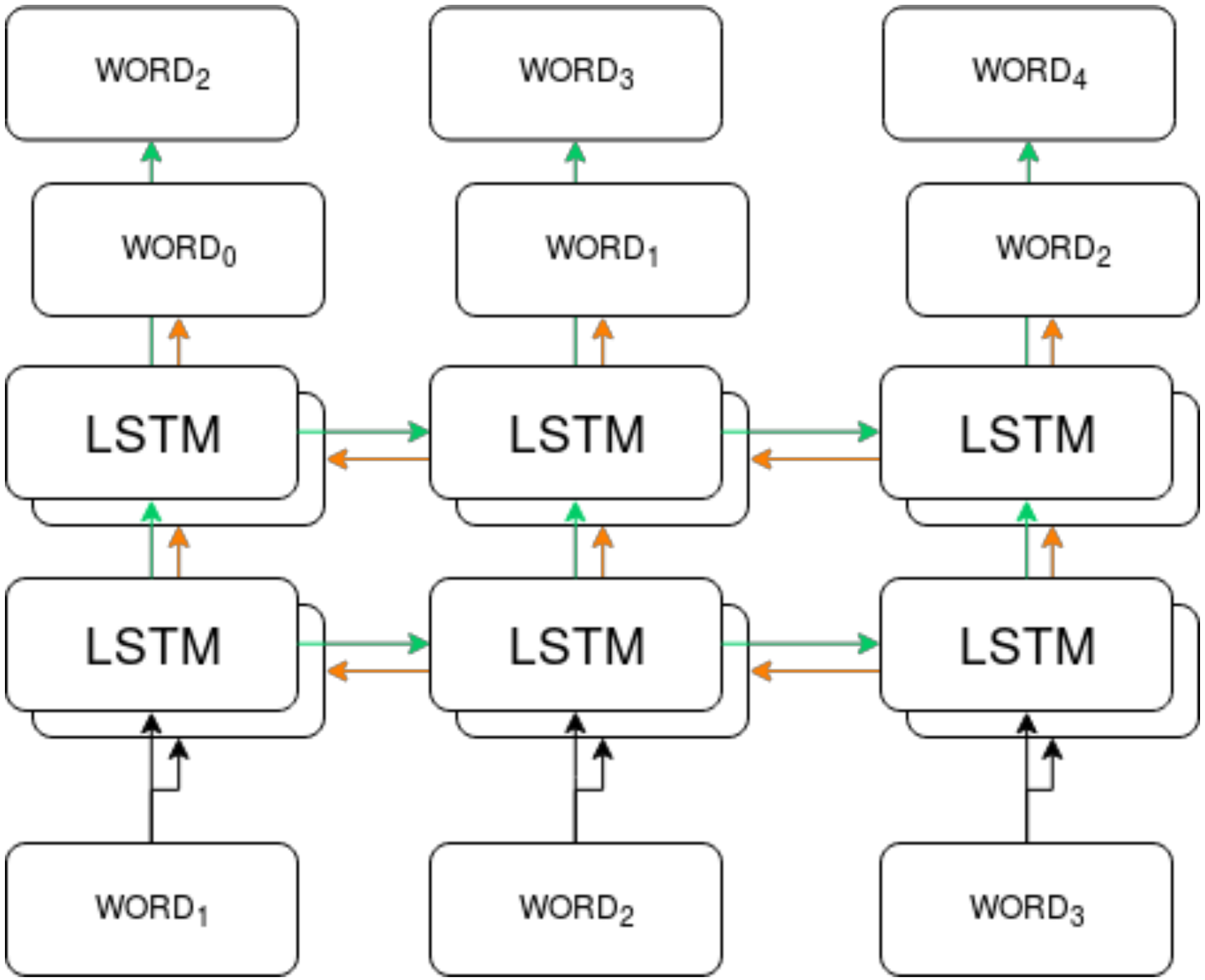
ELMo


y_1
y_2
y_3
y_n
Θ_s
Softmax
$\overleftarrow{\Theta}_{\text{LSTM}}$
s'_i
s'_0
s_0
s_i
$\overrightarrow{\Theta}_{\text{LSTM}}$
Θ_e
Embedding
x_1
x_2
x_3
x_n

ELMo


2 Forward Layers
2 Backward Layers
y_1
y_2
y_3
y_n
Θ_s
Softmax
$\overleftarrow{\Theta}_{\text{LSTM}}$
s'_i
s'_0
s_0
s_i
$\overrightarrow{\Theta}_{\text{LSTM}}$
Θ_e
Embedding
x_1
x_2
x_3
x_n

ELMo


WORD2
WORD3
WORD4
WORD0
WORD1
WORD2
LSTM
LSTM
LSTM
LSTM
LSTM
LSTM
WORD1
WORD2
WORD3

ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>

ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward								
Backward								

ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward	<START>							
Backward	<END>							

ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward	<START>	The						
Backward	<END>	money						

ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward	<START>	The	poor					
Backward	<END>	money	any					

ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward	<START>	The	poor	don’t				
Backward	<END>	money	any	have				

ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward	<START>	The	poor	don’t	have			
Backward	<END>	money	any	have	don’t			

ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward	<START>	The	poor	don’t	have	any		
Backward	<END>	money	any	have	don’t	poor		

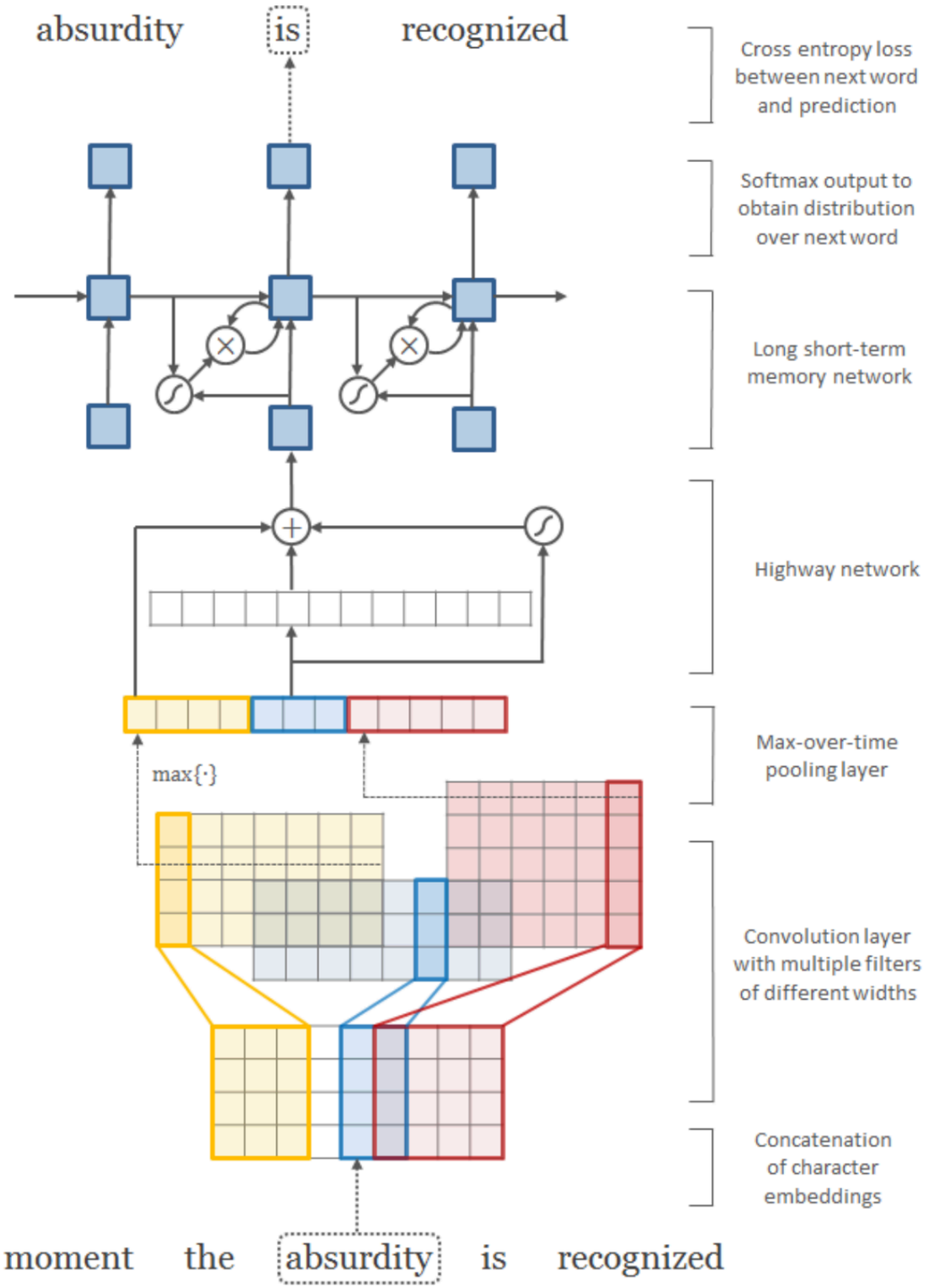
ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward	<START>	The	poor	don’t	have	any	money	
Backward	<END>	money	any	have	don’t	poor	The	

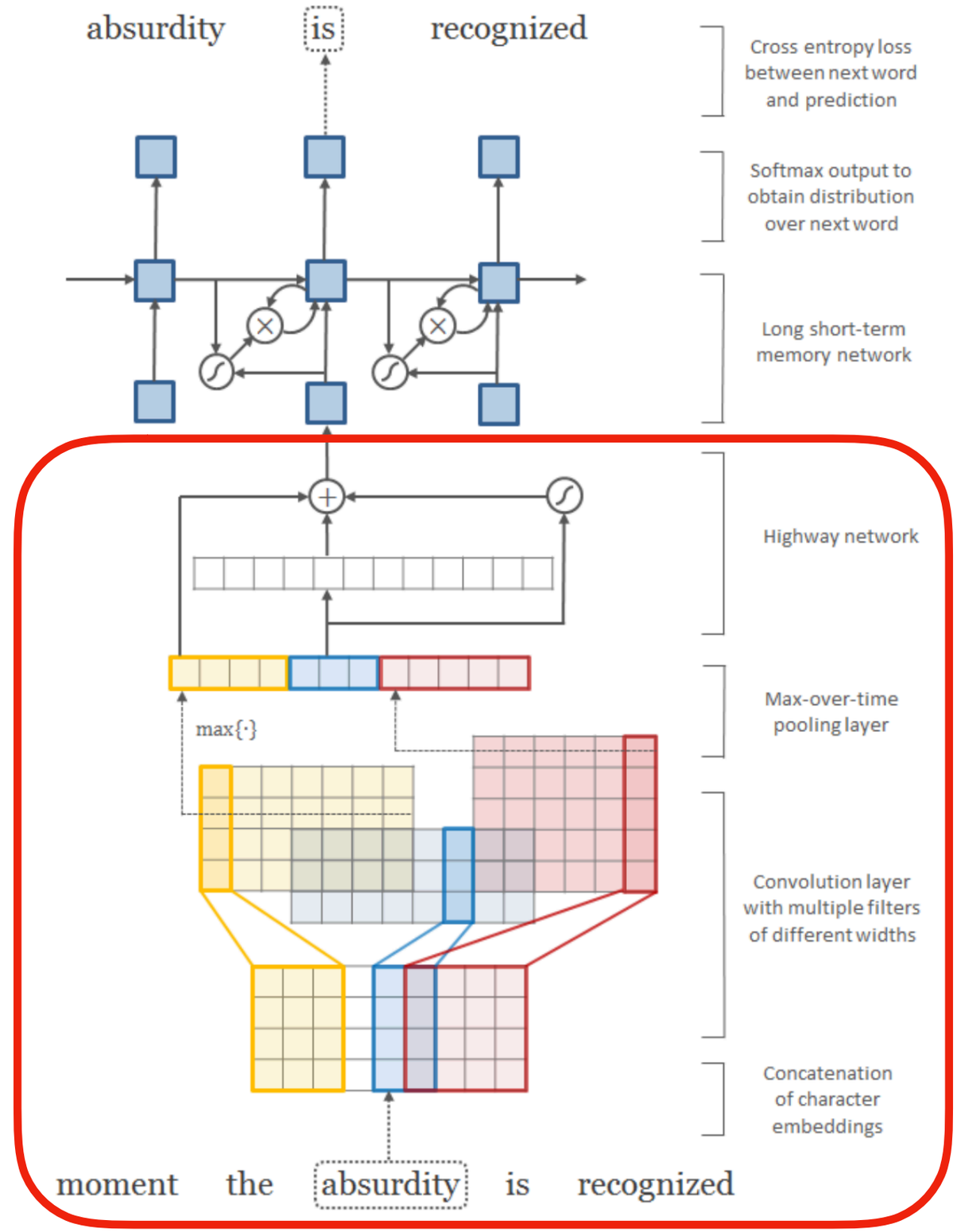
ELMo

Source	<START>	The	poor	don’t	have	any	money	<END>
Forward	<START>	The	poor	don’t	have	any	money	<END>
Backward	<END>	money	any	have	don’t	poor	The	<START>

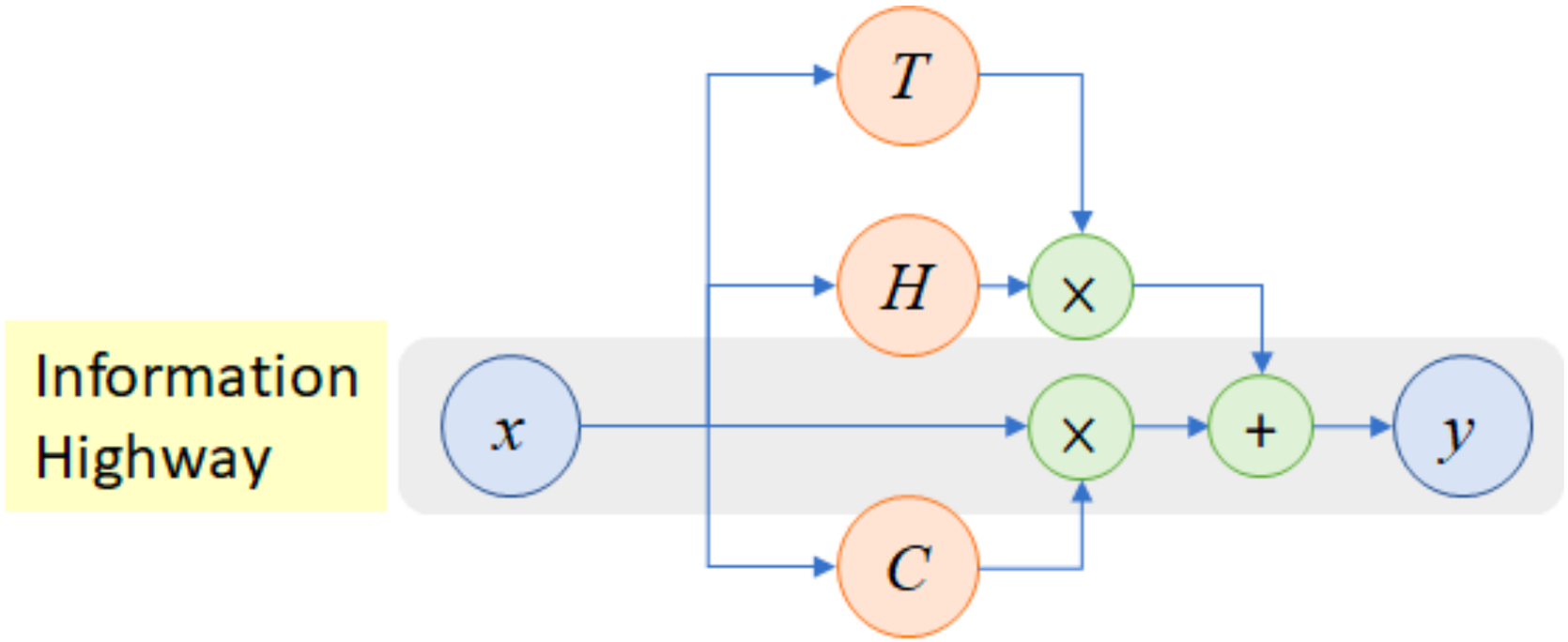
Character-Aware Neural Language Models



Character-Aware Neural Language Models

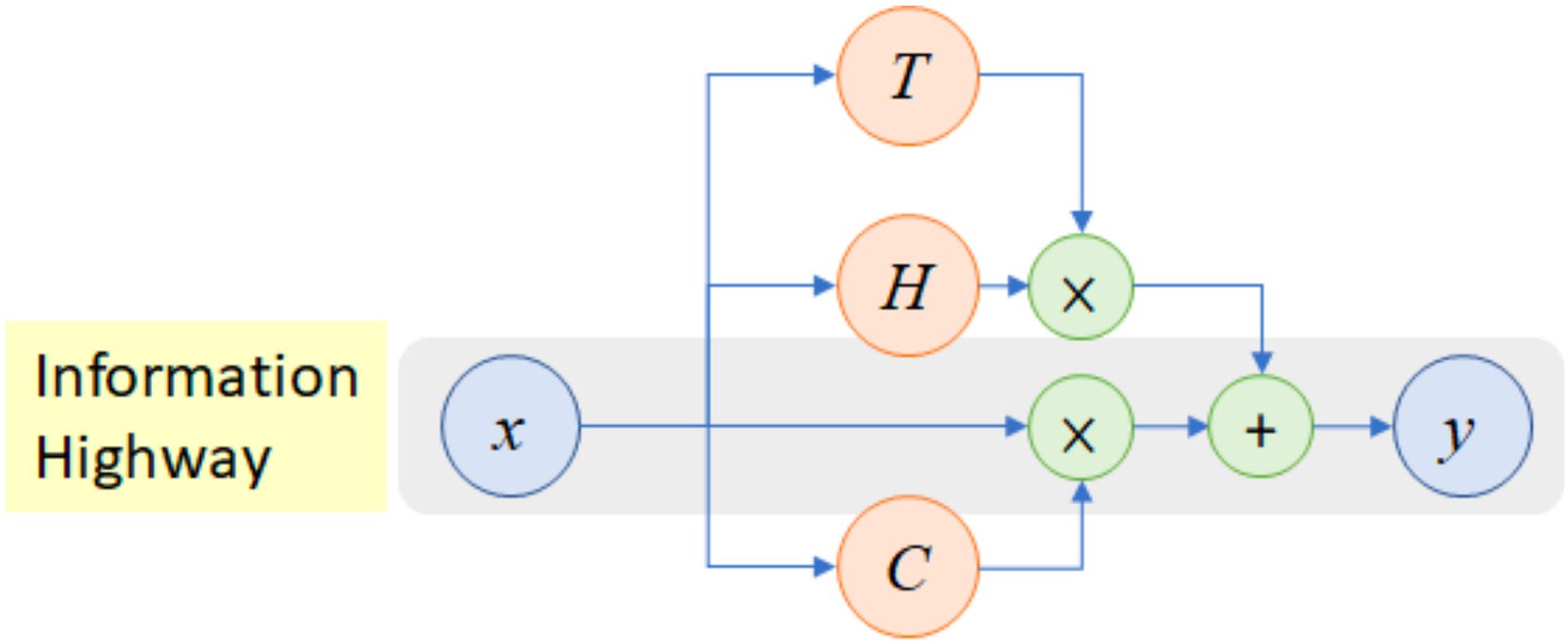


Highway Network



$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot C(\mathbf{x}, \mathbf{W_C}).$$

Highway Network



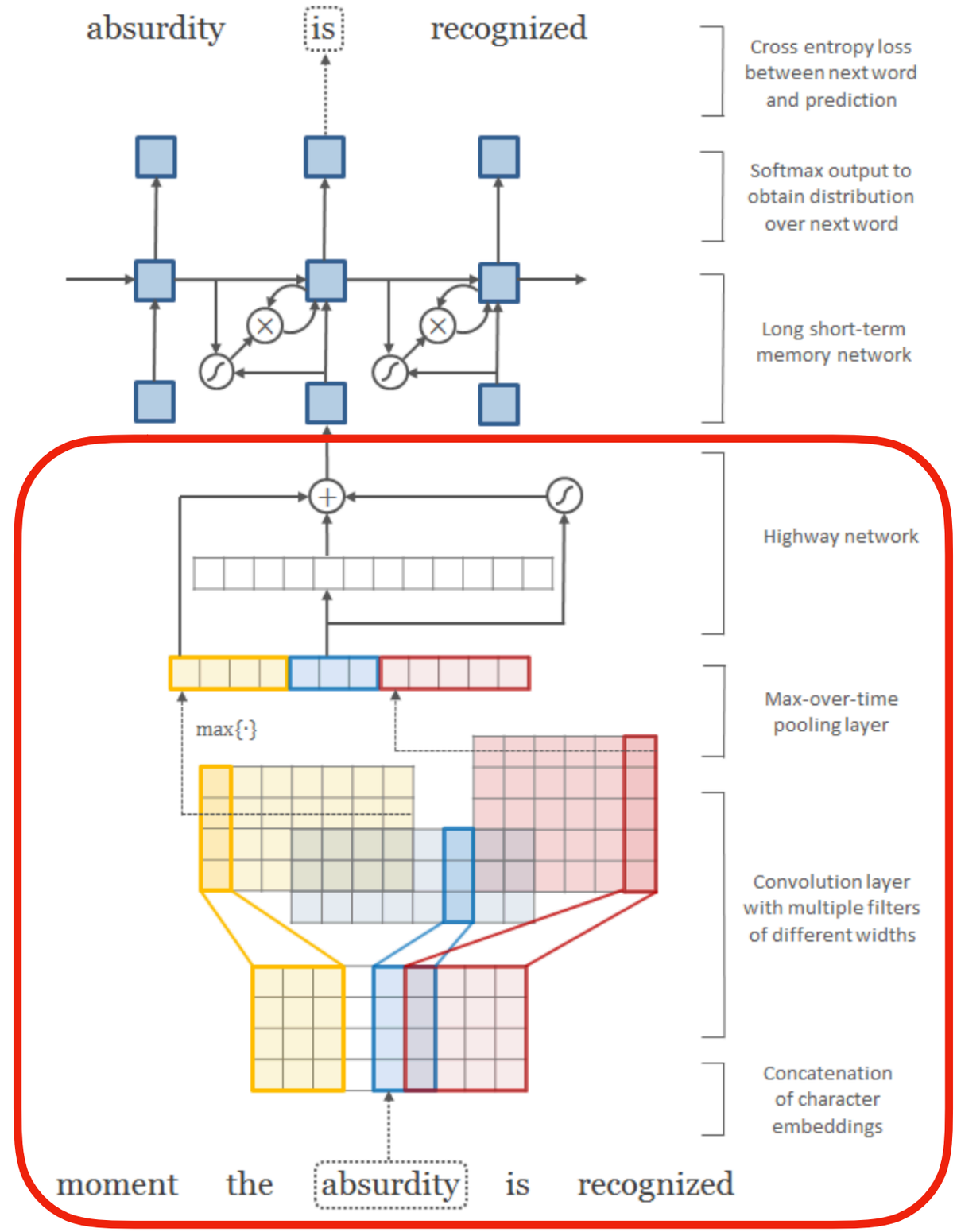
$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot C(\mathbf{x}, \mathbf{W_C}).$$

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot (1 - T(\mathbf{x}, \mathbf{W_T})).$$

Highway Network

```python
class Highway(BaseModule):

    def __init__(self, in_features, out_features, activation_function=torch.nn.ReLU()):

        super(Highway, self).__init__()

        self.nonlinear = torch.nn.Linear(in_features, out_features)
        self.linear = torch.nn.Linear(in_features, out_features)
        self.gate = torch.nn.Linear(in_features, out_features)

        self.gate_function = torch.nn.Sigmoid()
        self.activation_function = activation_function

    def forward(self, x):
        """
        :param x: tensor with shape of [batch_size, size]
        :return: tensor with shape of [batch_size, size]
        applies σ(x) ⊙ (f(G(x))) + (1 - σ(x)) ⊙ (Q(x)) transformation | G and Q is affine transformation,
        f is non-linear transformation, σ(x) is affine transformation with sigmoid non-linearition
        and ⊙ is element-wise multiplication
        """

        gate = self.gate(x)
        gate = self.gate_function(gate)

        nonlinear = self.nonlinear(x)
        nonlinear = self.activation_function(nonlinear)

        linear = self.linear(x)

        x = gate * nonlinear + (1 - gate) * linear

        return x
```

Character-Aware Neural Language Models

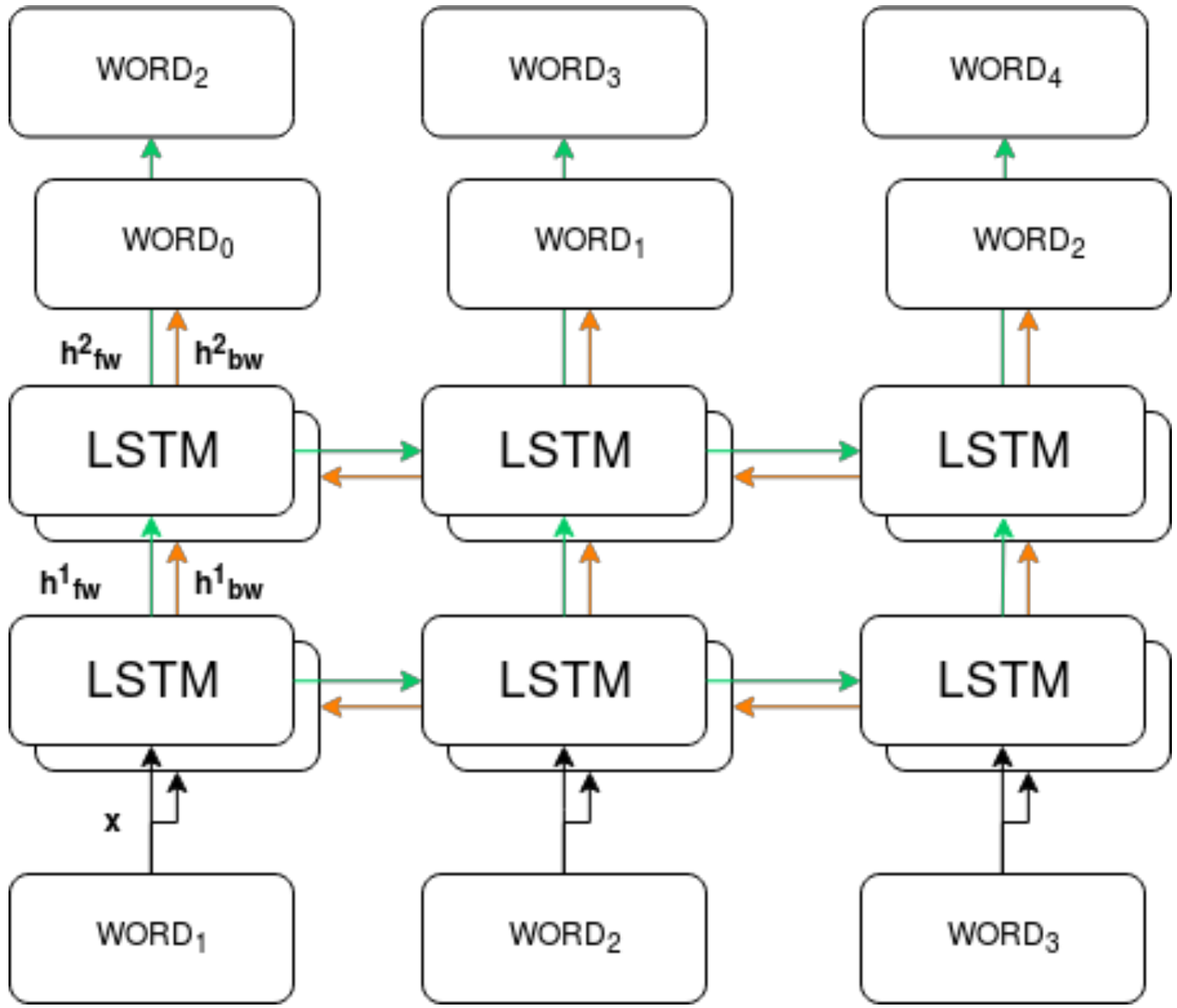


Gated Linear Unit

$$(X*W+b) \otimes sigmoid(X*V+c)$$

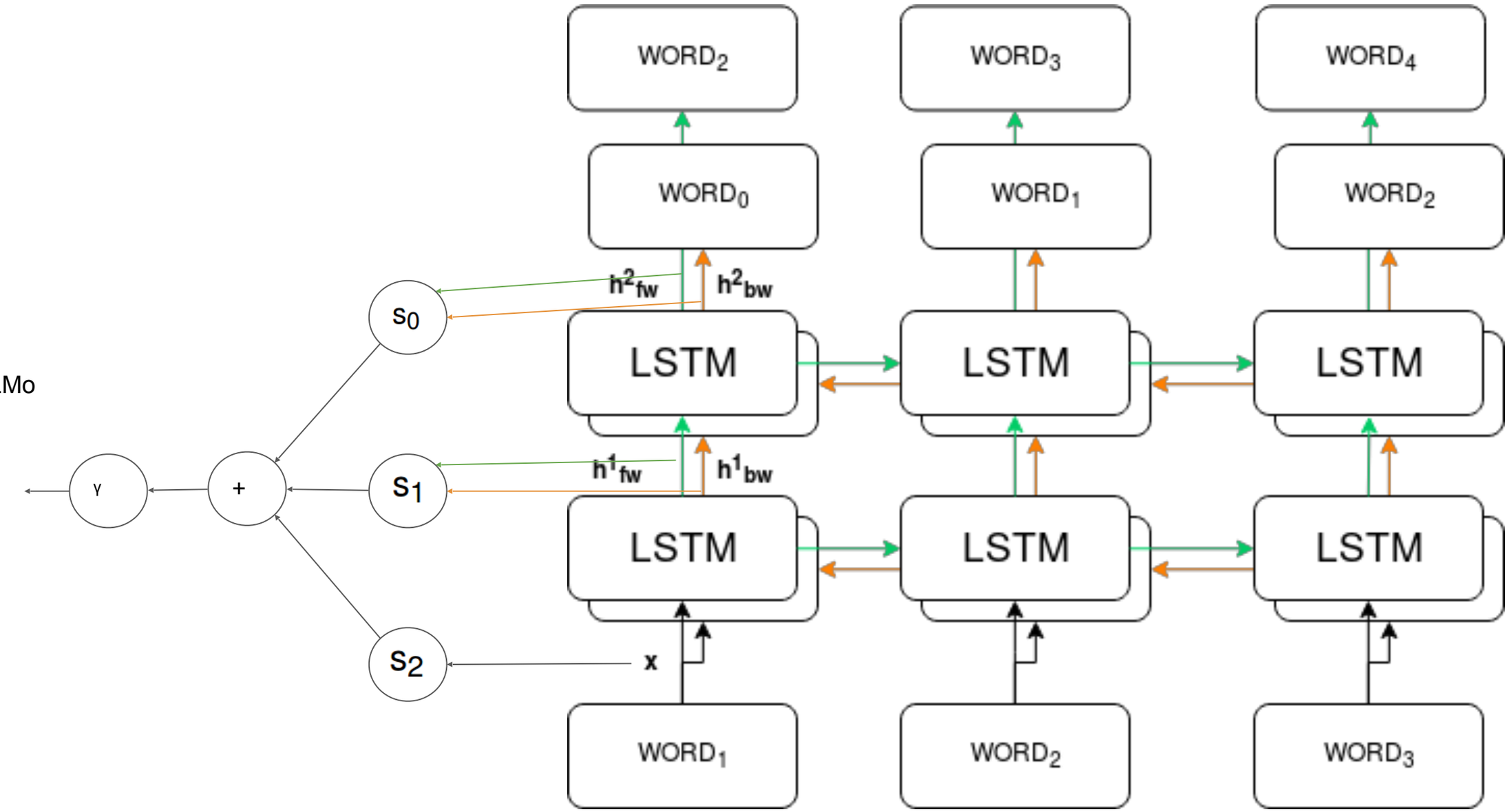
```python
class GatedLinearUnit(BaseModule):
    """
    Implementation of this https://arxiv.org/pdf/1612.08083.pdf
    """

    def __init__(self, in_features, out_features):

        super().__init__()

        self.linear = torch.nn.Linear(in_features=in_features, out_features=out_features)
        self.gate = torch.nn.Linear(in_features=in_features, out_features=out_features)

    def forward(self, x):

        x = self.linear(x)
        gate = self.linear(x)

        x = x * torch.sigmoid(gate)

        return x
```

ELMo


WORD2
WORD3
WORD4
WORD0
WORD1
WORD2
h2fw
h2bw
LSTM
LSTM
LSTM
h1fw
h1bw
LSTM
LSTM
LSTM
x
WORD1
WORD2
WORD3

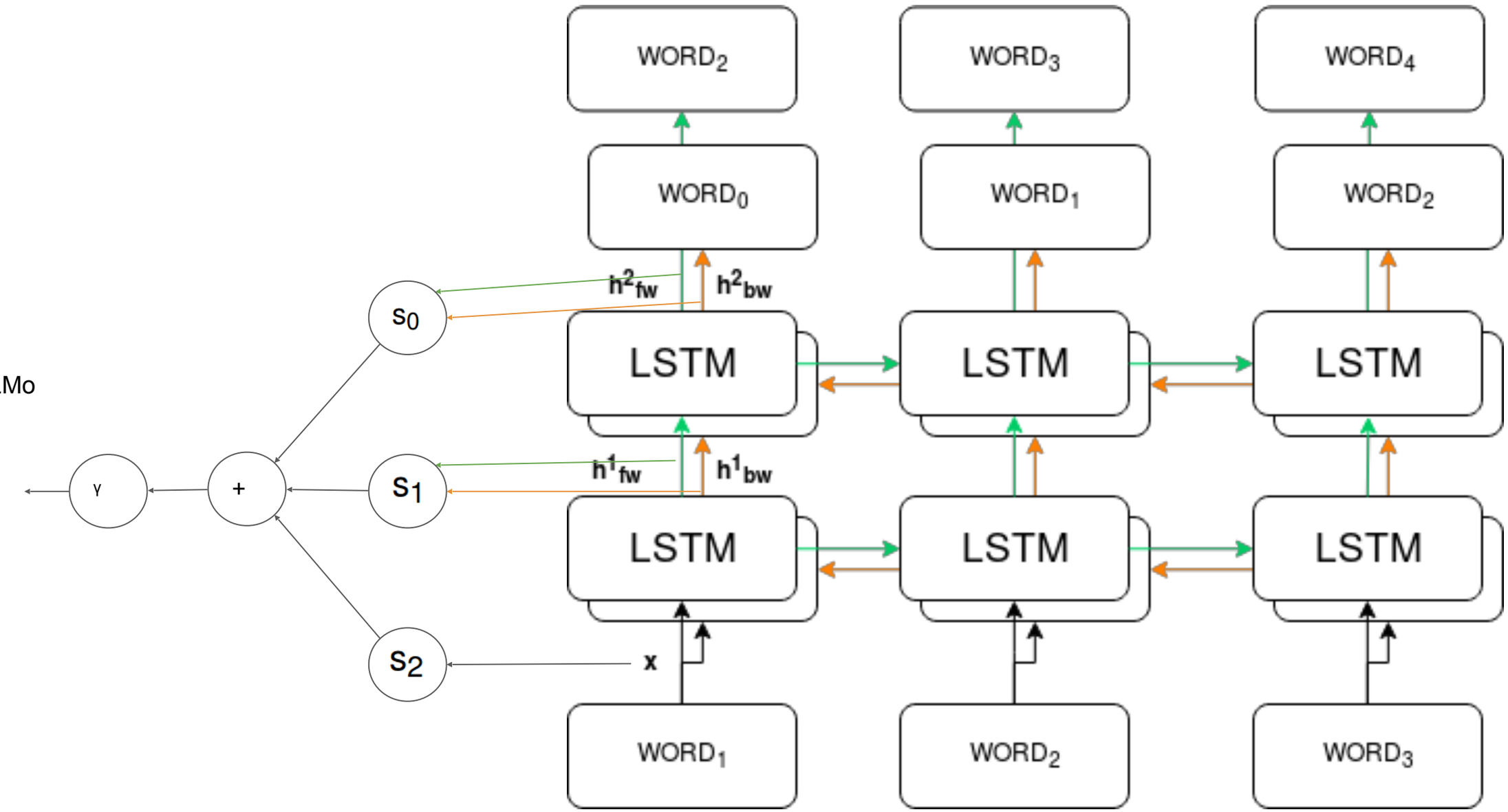
ELMo

ELMo


WORD2
WORD3
WORD4
WORD0
WORD1
WORD2
h2fw
h2bw
s0
LSTM
LSTM
LSTM
y
+
s1
h1fw
h1bw
LSTM
LSTM
LSTM
s2
x
WORD1
WORD2
WORD3

ELMo

ELMo



$$
\begin{aligned}
R_k &= \{\mathbf{x}_k^{LM}, \overrightarrow{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, \ldots, L\} \\
&= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, \ldots, L\},
\end{aligned}
$$

$$
\mathbf{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,j}^{LM}.
$$

ELMo

Results

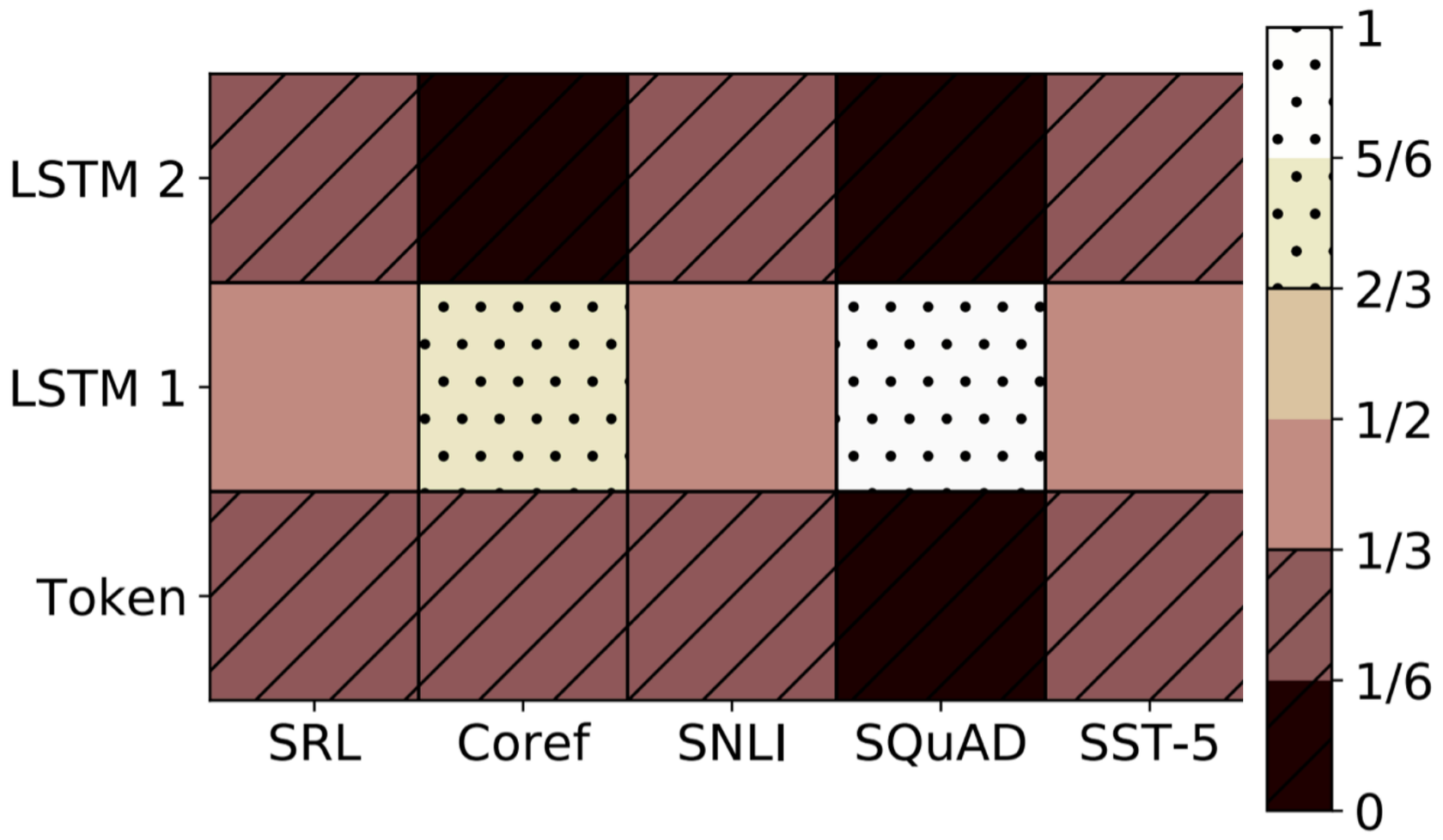
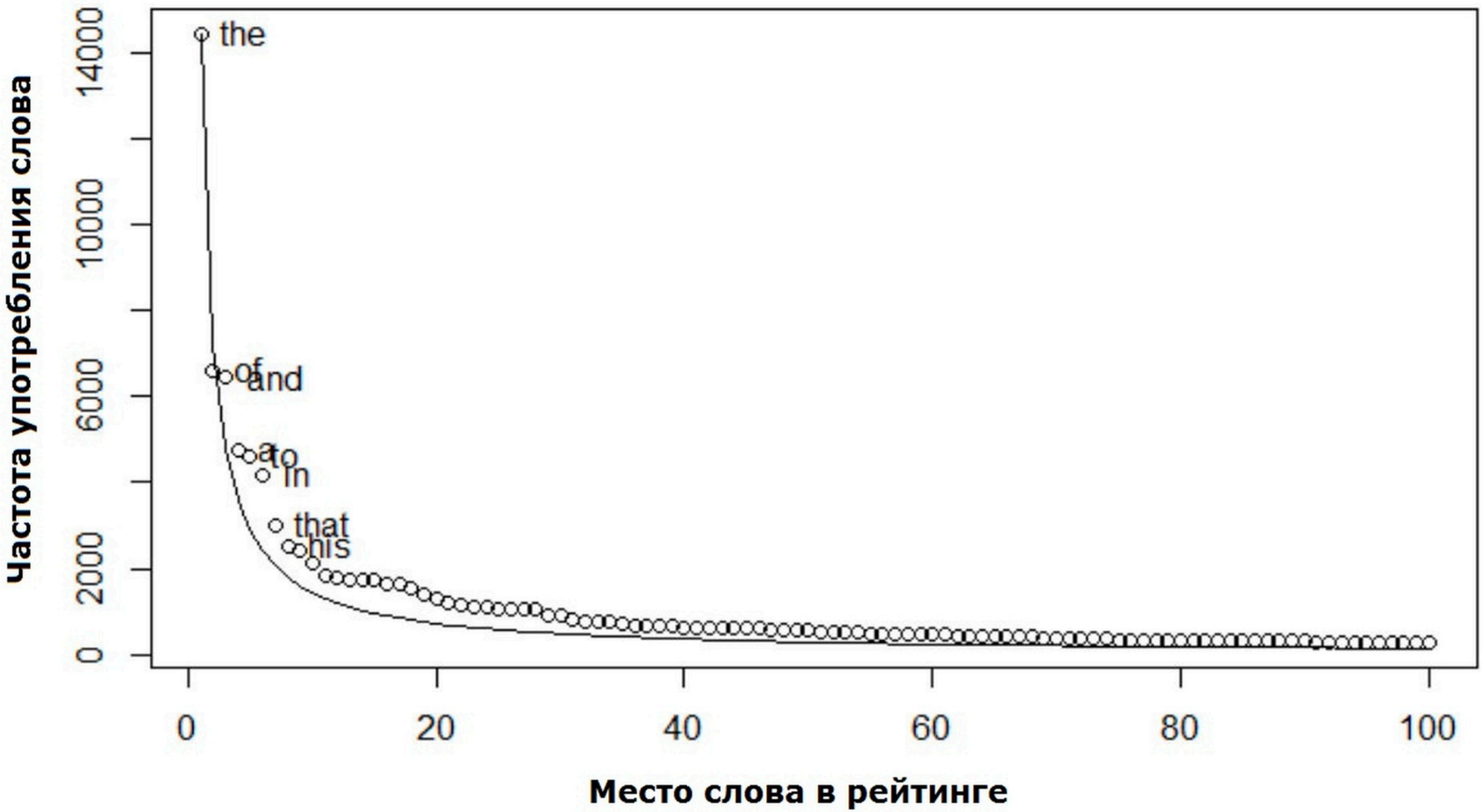


Figure 2: Visualization of softmax normalized biLM layer weights across tasks and ELMo locations. Normalized weights less then $1/3$ are hatched with horizontal lines and those greater then $2/3$ are speckled.

Tokenization

Закон Ципфа



Tokenization

Word level

\+ Small text length
\- Big vocabulary size
\- OOV

Character level

\- Long text
\+ Small vocabulary size
\+ Almost no OOV

Tokenization

Word level

\+ Small text length
\- Big vocabulary size
\- OOV

Character level

\- Long text
\+ Small vocabulary size
\+ Almost no OOV

1. Word level

i'm a second year student in an ivy league school ->

["'i'm", 'a', 'second', 'year', 'student', 'in', 'an', 'ivy', 'league', ‘school’]

2. Character level

['i', "'", 'm', ' ', 'a', ' ', 's', 'e', 'c', 'o', 'n', 'd', ' ', 'y', 'e', 'a', 'r', ' ', 's', 't', 'u', 'd', 'e', 'n', 't', ' ', 'i', 'n', ' ', 'a', 'n', ' ', 'i', 'v', 'y', ' ', 'l', 'e', 'a', 'g', 'u', 'e', ' ', 's', 'c', 'h', 'o', 'o', 'l']

Tokenization

Word level

\+ Small text length
\- Big vocabulary size
\- OOV

Character level

\- Long text
\+ Small vocabulary size
\+ Almost no OOV

BPE

I saw a girl with a telescope. ->

['__I', '__saw', '__a', '__girl', '__with', '__a', '__', 'te', 'le', 's', 'c', 'o', 'pe', '.']

опубликовано видео убитого саудовского журналиста джамаля хашкуджи ->

['__опубликовано', '__видео', '__убитого', '__саудов', 'ского', '__журналиста', '__джама', 'ля', '__ха', 'шку', 'джи']

BPE

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

---

r ·	→	r·
l o	→	lo
lo w	→	low
e r·	→	er·

Figure 1: BPE merge operations learned from dictionary {‘low’, ‘lowest’, ‘newer’, ‘wider’}.

BPE

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

r ·	→	r·
l o	→	lo
lo w	→	low
e r·	→	er·

Figure 1: BPE merge operations learned from dictionary {‘low’, ‘lowest’, ‘newer’, ‘wider’}.

- learning
  - word:freq : {low:5, lowest:2, newer:6, wider:3}
  - marge & count
    1. ‘r’ ‘</w>’ : 9 → marge’r</w>’
    2. ‘e’ ‘r</w>’ : 9 →marge’er</w>’
    3. ‘l’ ‘o’ : 7 →marge’lo’
    4. ‘lo’ ‘w’ : 7 →marge’low’

→ OOV : ‘lower’ segmented ‘low er</w>’

BPE

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

r · → r·
l o → lo
lo w → low
e r· → er·

Figure 1: BPE merge operations learned from dictionary {'low', 'lowest', 'newer', 'wider'}.

- learning
  - word:freq : {low:5, lowest:2, newer:6, wider:3}
  - marge & count
    1. 'r' '</w>' : 9 → marge'r</w>'
    2. 'e' 'r</w>' : 9 →marge'er</w>'
    3. 'l' 'o' : 7 →marge'lo'
    4. 'lo' 'w' : 7 →marge'low'

→ OOV : 'lower' segmented 'low er</w>'

Vocabulary sizes:
5000, 10000, 15000, …, 50000

BPE

SentencePiece

SentencePiece is an unsupervised text tokenizer and detokenizer mainly for Neural Network-based text generation systems where the vocabulary size is predetermined prior to the neural model training. SentencePiece implements **subword units** (e.g., **byte-pair-encoding (BPE)** [Sennrich et al.]) and **unigram language model** [Kudo.]) with the extension of direct training from raw sentences. SentencePiece allows us to make a purely end-to-end system that does not depend on language-specific pre/postprocessing.

This is not an official Google product.

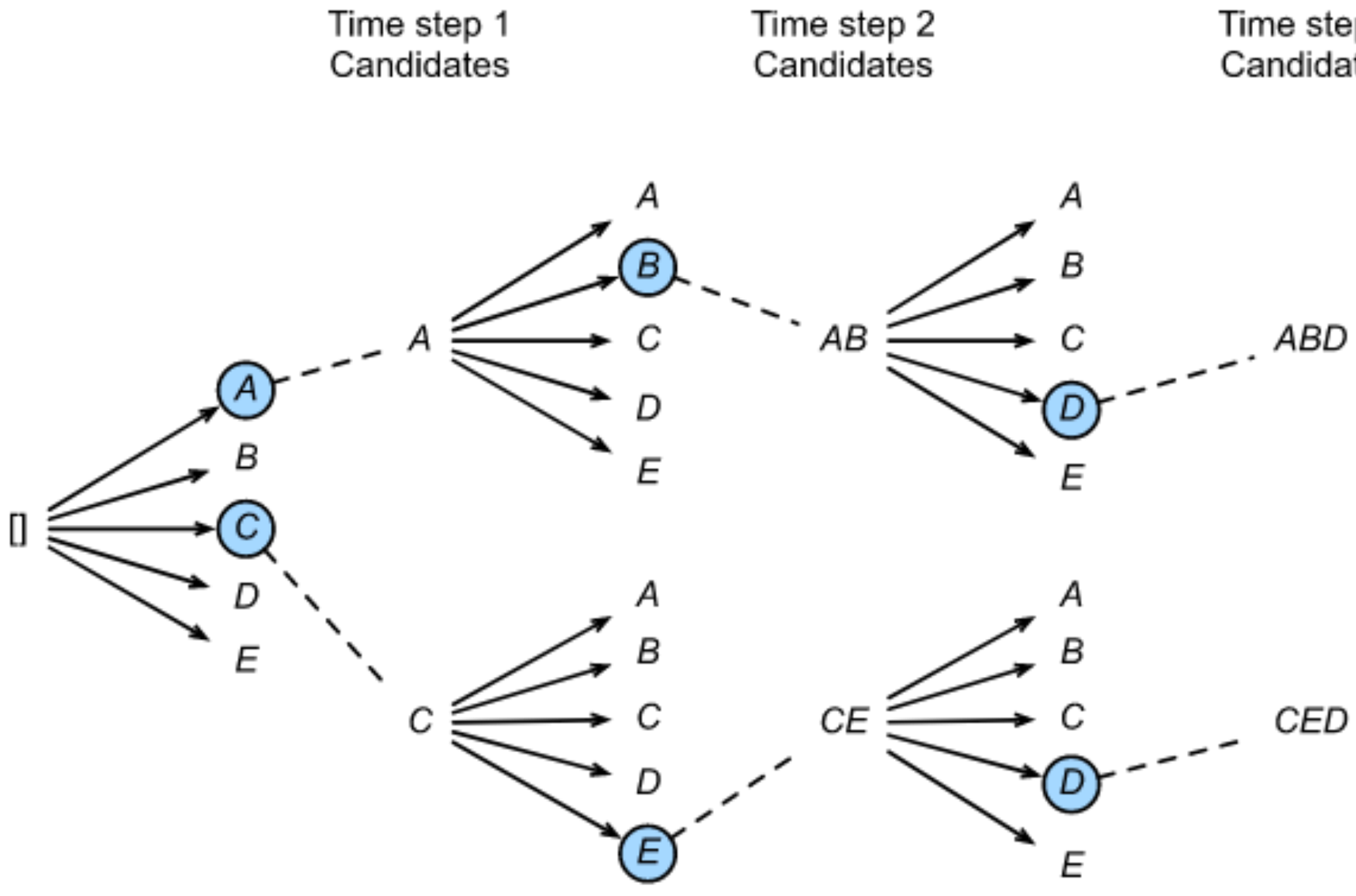
YouTokenToMe

YouTokenToMe is an unsupervised text tokenizer focused on computational efficiency. It currently implements fast Byte Pair Encoding (BPE) [Sennrich et al.]. Our implementation is much faster in training and tokenization than both fastBPE and SentencePiece. In some test cases, it is 90 times faster. Check out our benchmark results.

Key advantages:

- Multithreading for training and tokenization
- The algorithm has `O(N)` complexity, where `N` is the length of training data
- Highly efficient implementation in C++
- Python wrapper and command-line interface

Beam Search


Time step 1
Candidates
Time step 2
Candidates
Time step 3
Candidates
[]
A
B
C
D
E
A
B
C
D
E
AB
CE
ABD
CED

Beam Search

$$\arg\max_{y} \prod_{t=1}^{T_y} P(y^{<t>} | x, y^{<1>}, \ldots, y^{<t-1>})$$

Beam Search

$$p(\mathbf{x}) = \prod_i p(x|x_{<i}) = p(x_0)p(x_1|x_0)p(x_2|x_0, x_1)...$$

$$\arg\max_y \prod_{t=1}^{T_y} P(y^{<t>}|x, y^{<1>}, \ldots, y^{<t-1>})$$

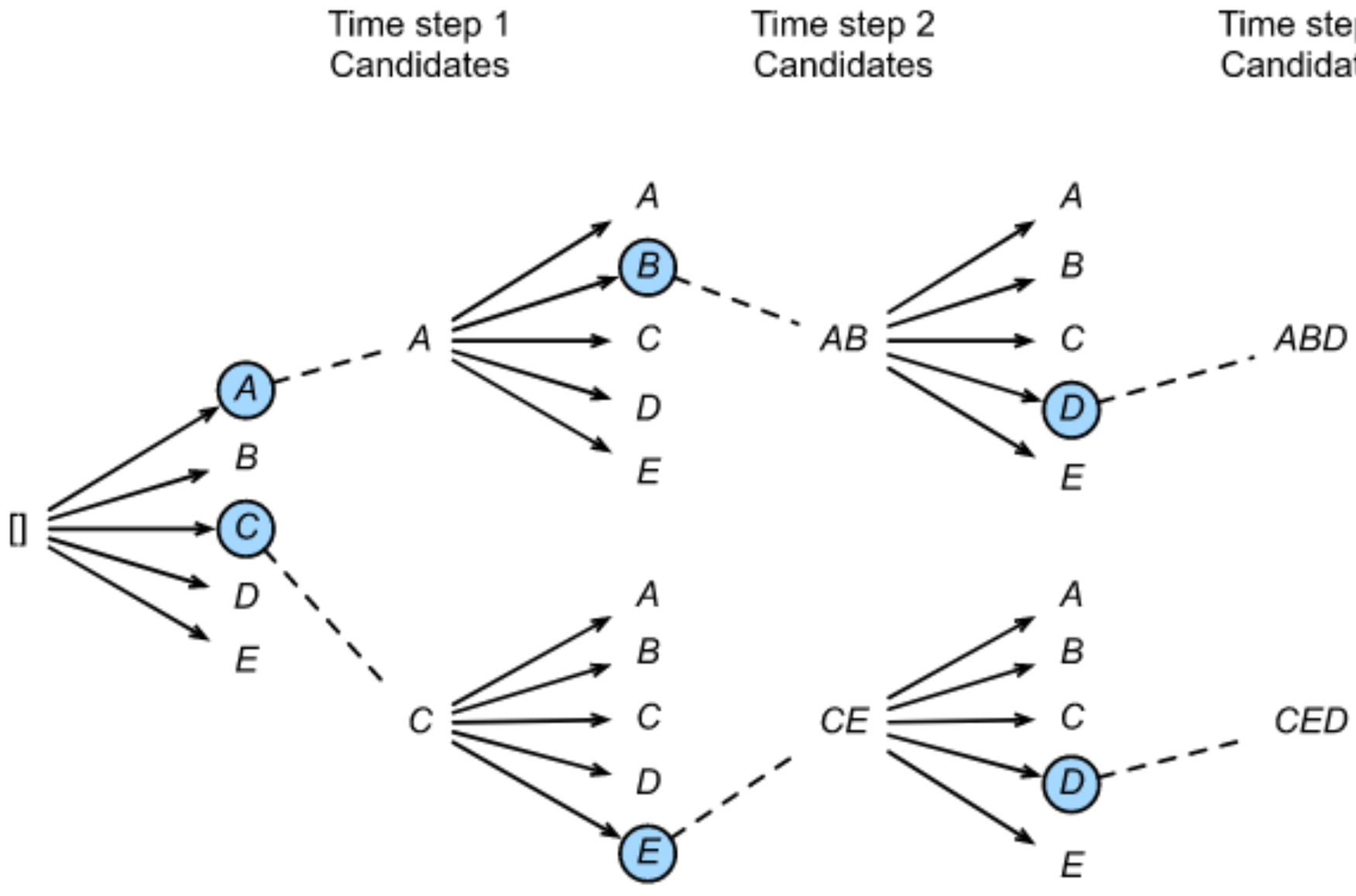
Beam Search

$$p(\mathbf{x}) = \prod_i p(x|x_{<i}) = p(x_0)p(x_1|x_0)p(x_2|x_0, x_1)...$$

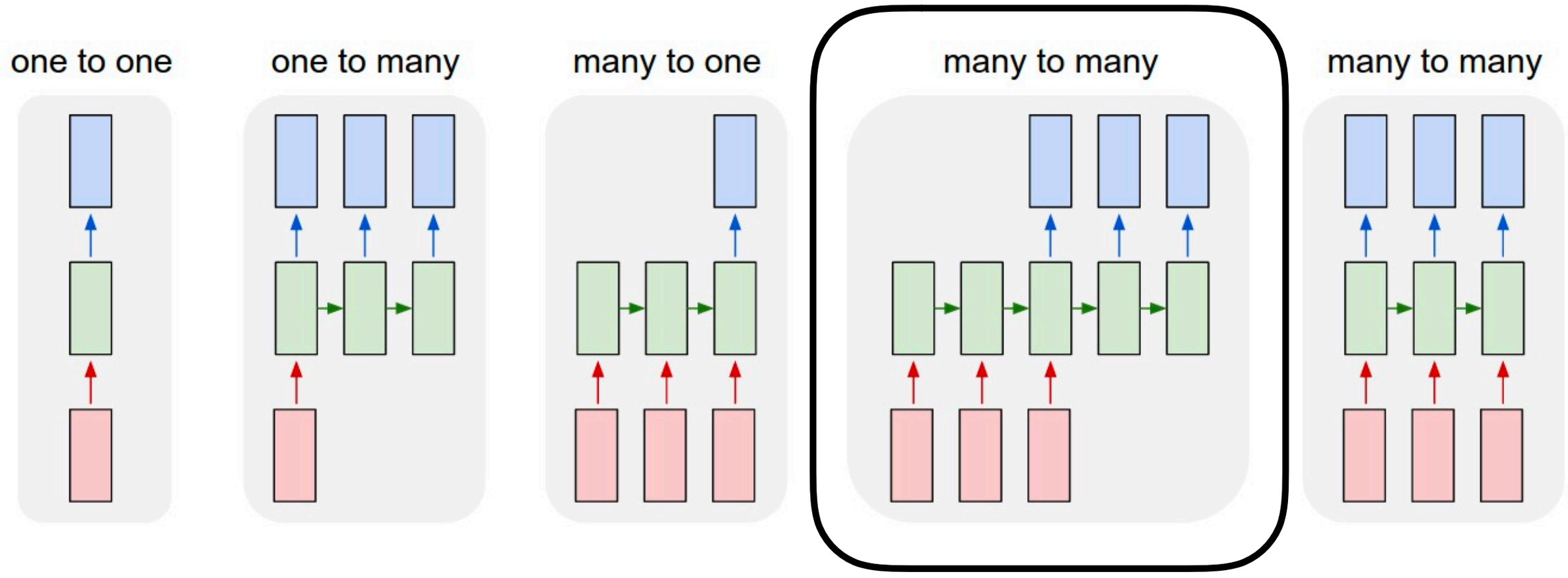
$$\arg\max_y \prod_{t=1}^{T_y} P(y^{<t>} | x, y^{<1>}, \ldots, y^{<t-1>})$$

$$\arg\max_y \sum_{y=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \ldots, y^{<t-1>})$$

Beam Search


Time step 1
Candidates
Time step 2
Candidates
Time step 3
Candidates
[]
A
B
C
D
E
A
B
C
D
E
AB
CE
ABD
CED

Sequence to sequence



Sequence to sequence

Source sentence

Sequence to sequence

Les pauvres sont démunis

Source sentence

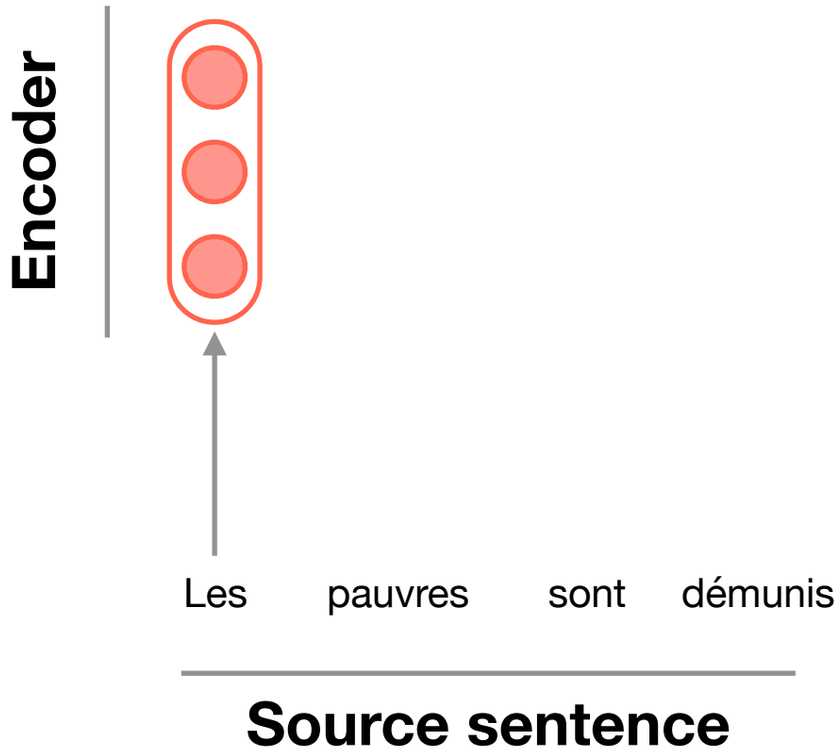
Sequence to sequence

Encoder

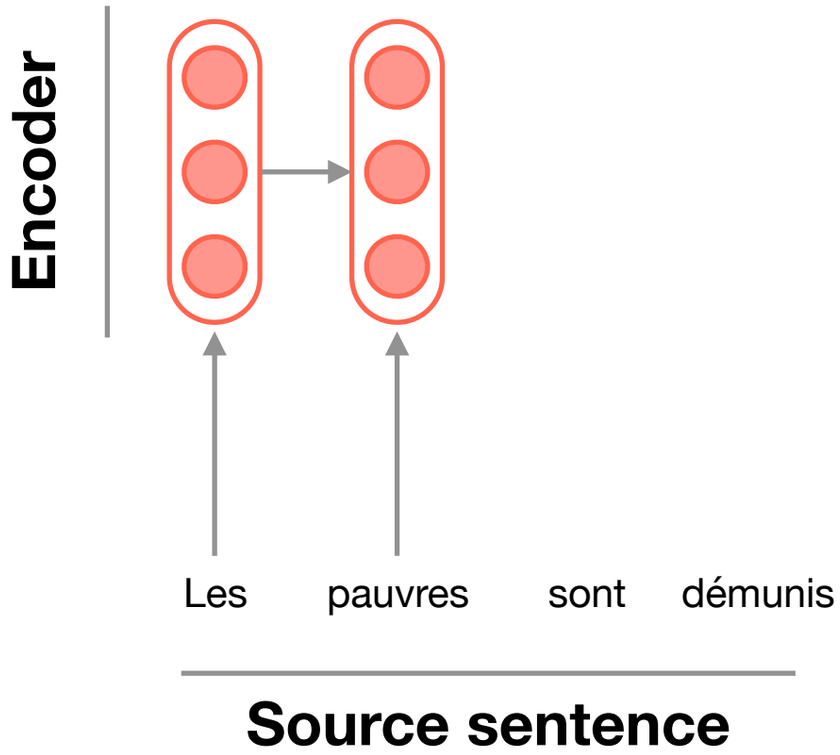
Les pauvres sont démunis

Source sentence

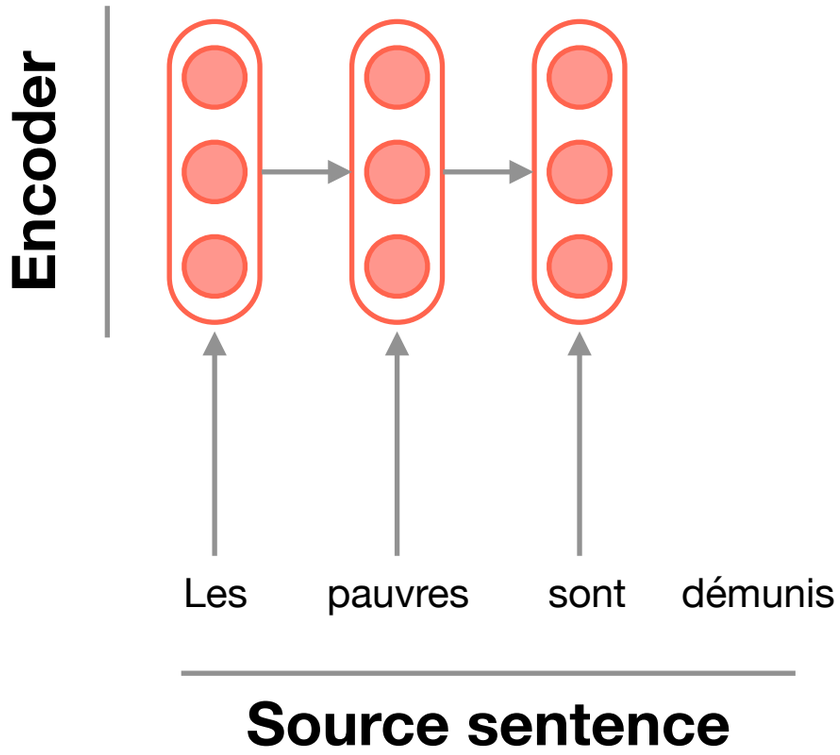
Sequence to sequence



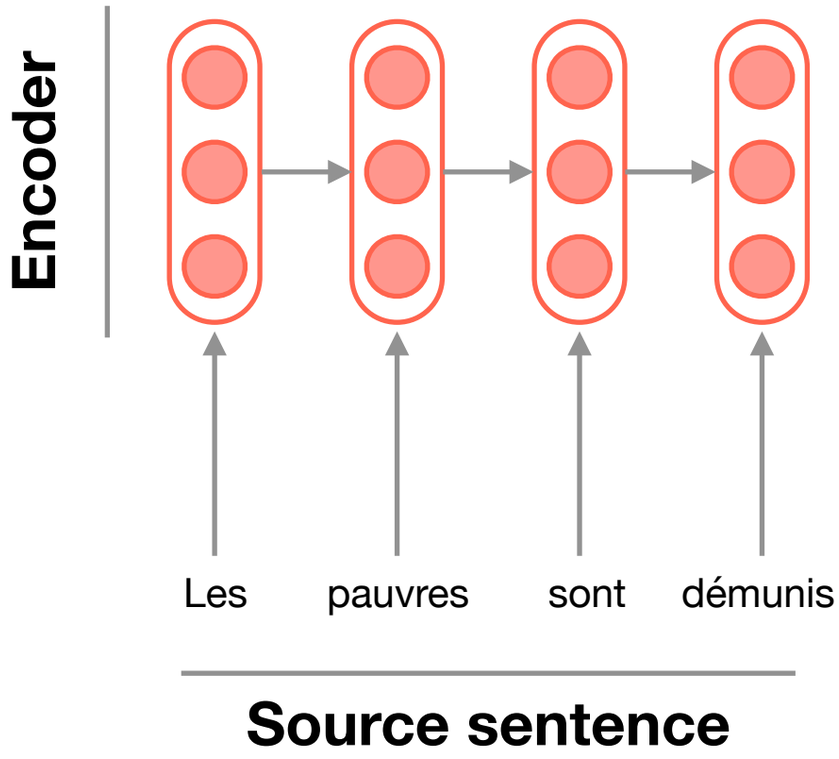
Sequence to sequence



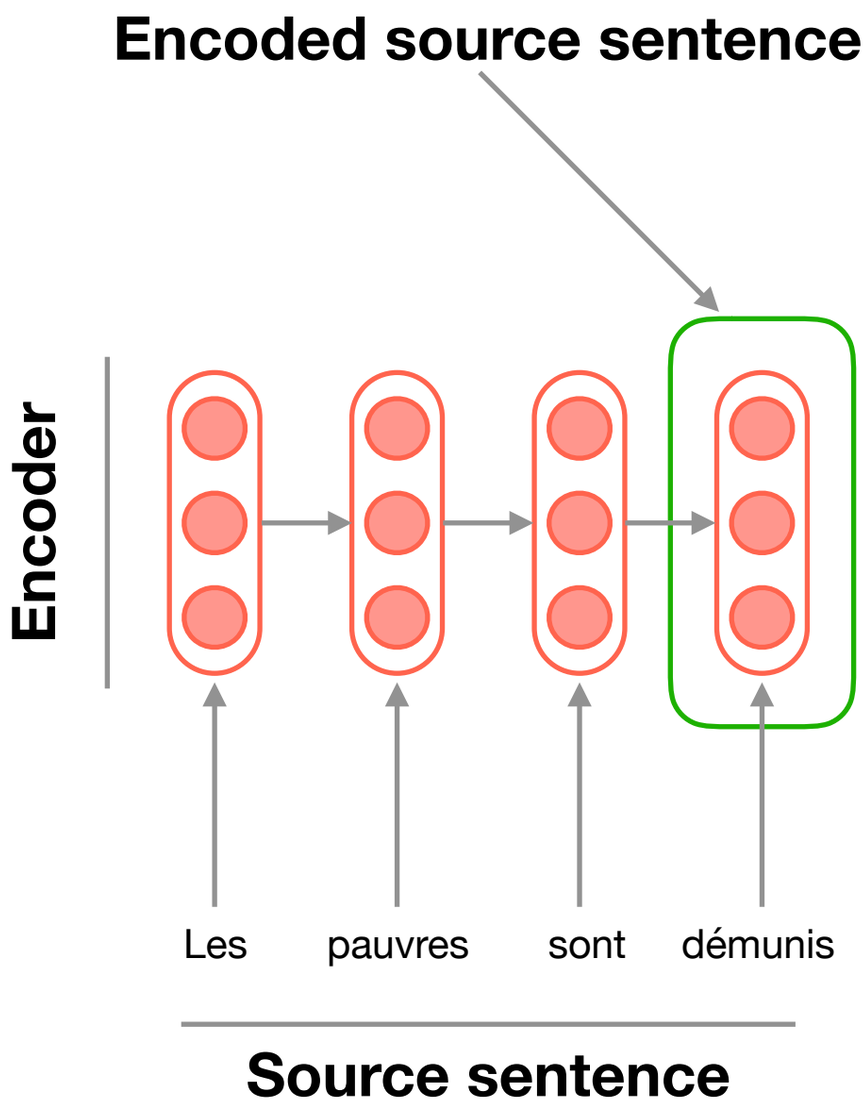
Sequence to sequence



Sequence to sequence

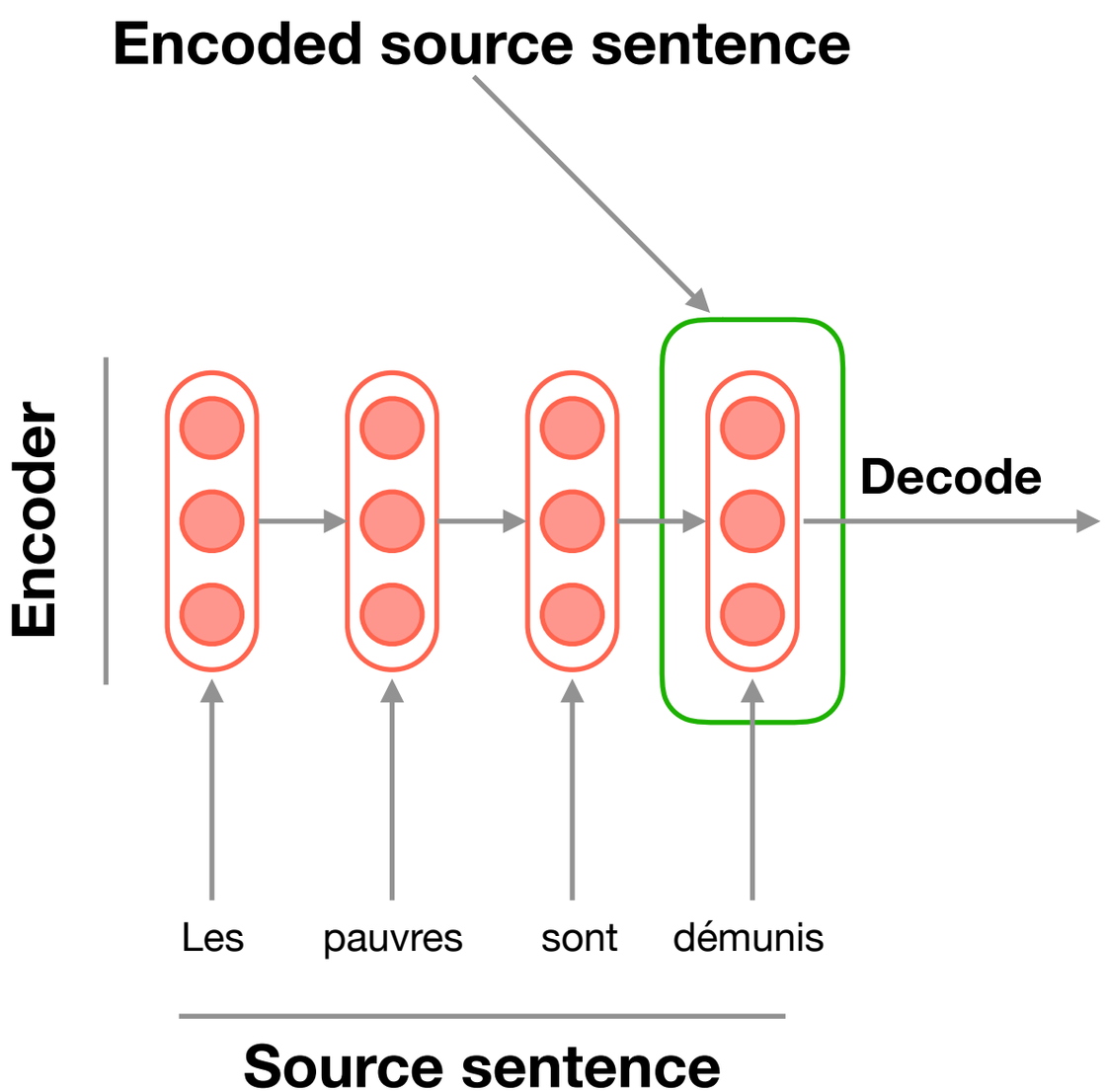


Sequence to sequence



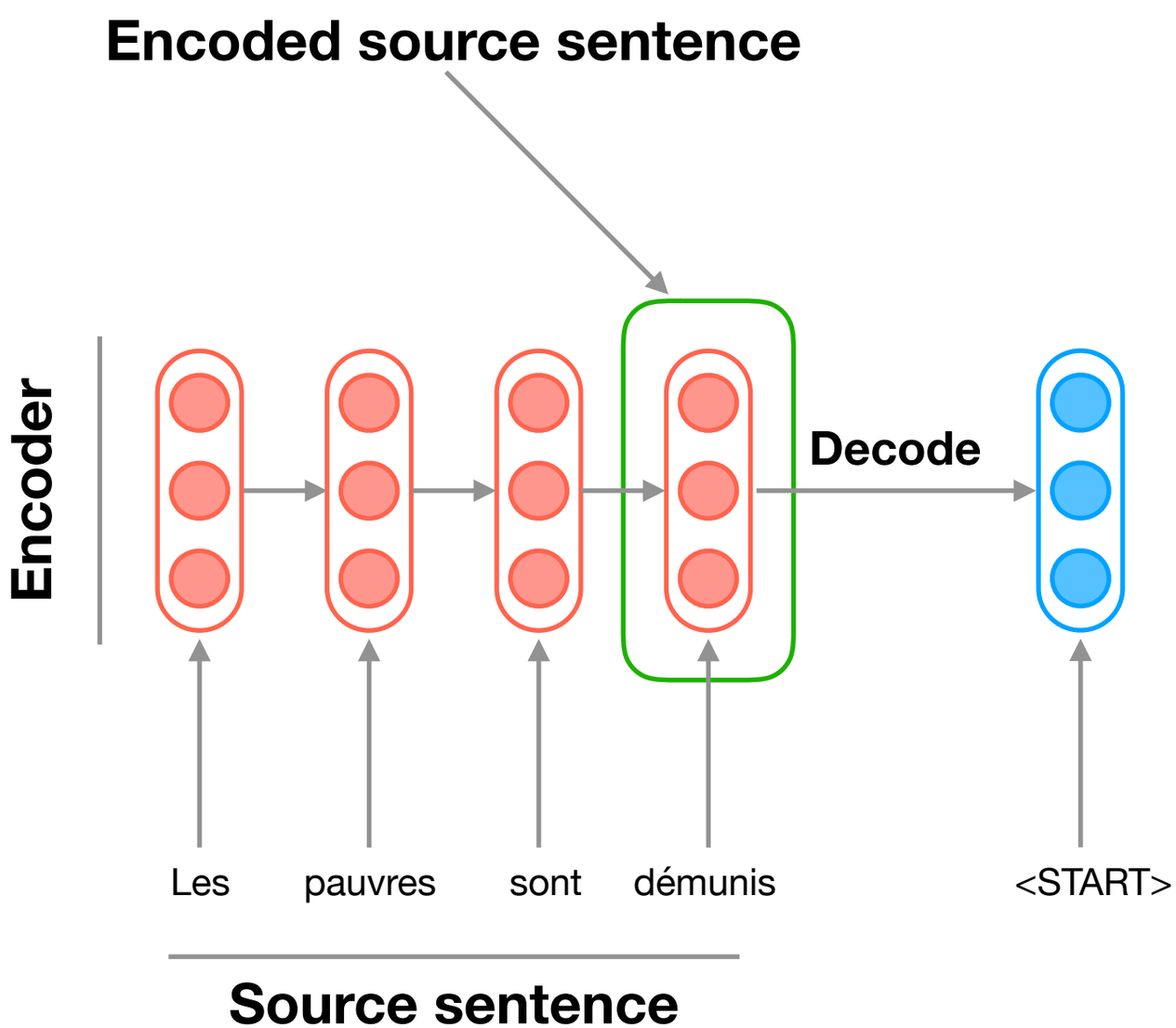
Sequence to sequence

Inference



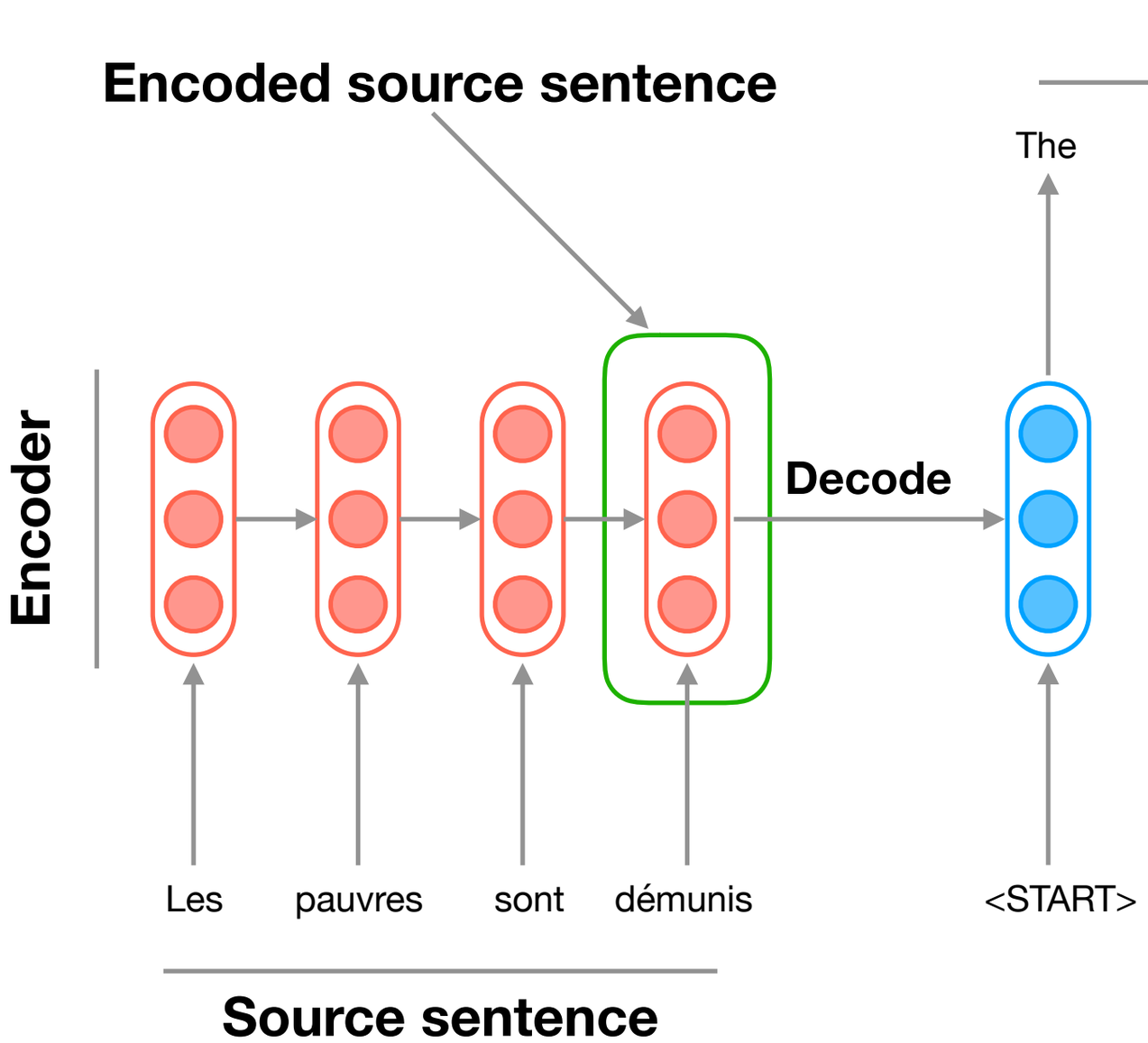
Sequence to sequence

Inference



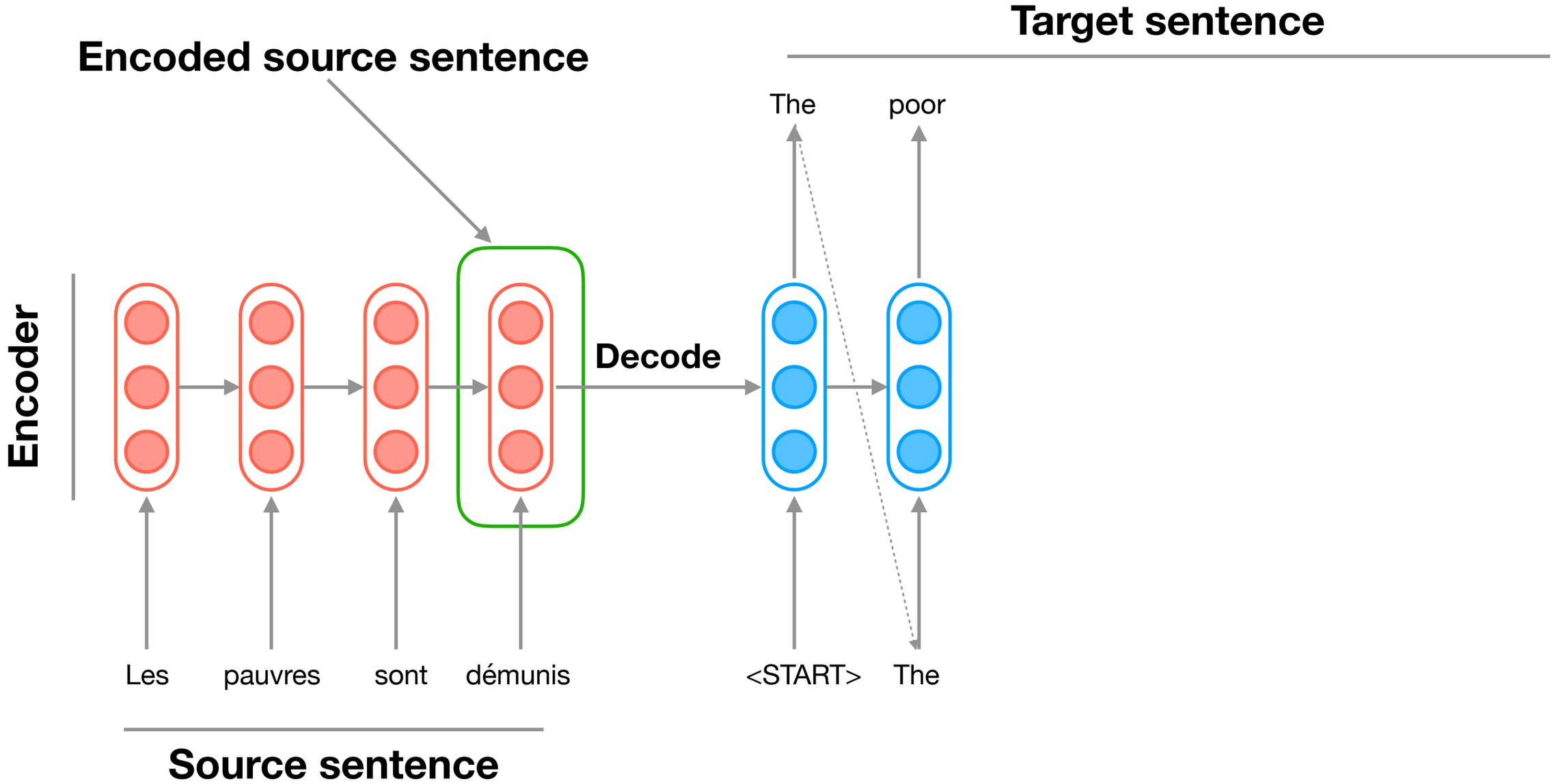
Sequence to sequence

Inference



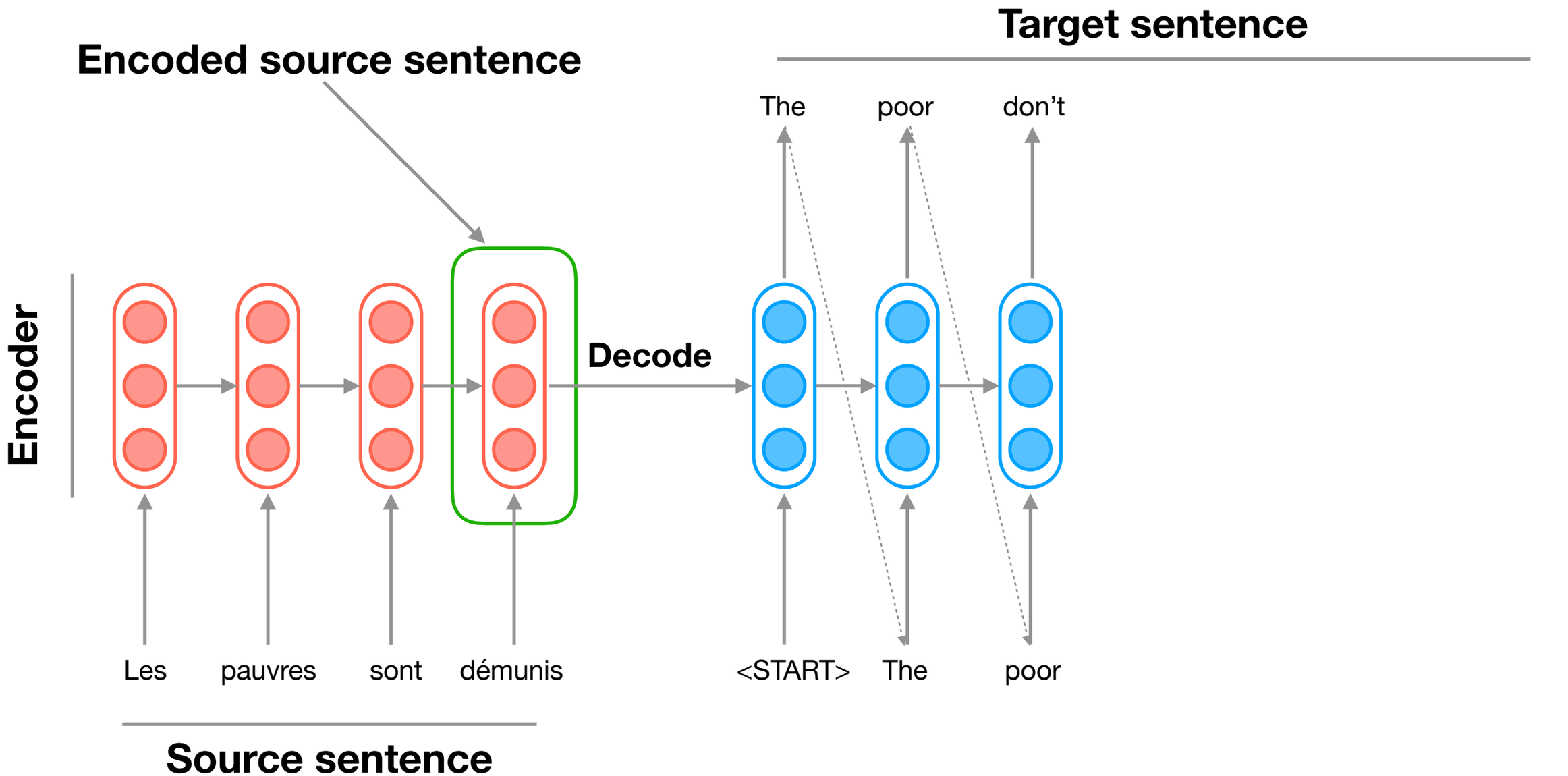
Sequence to sequence

Inference



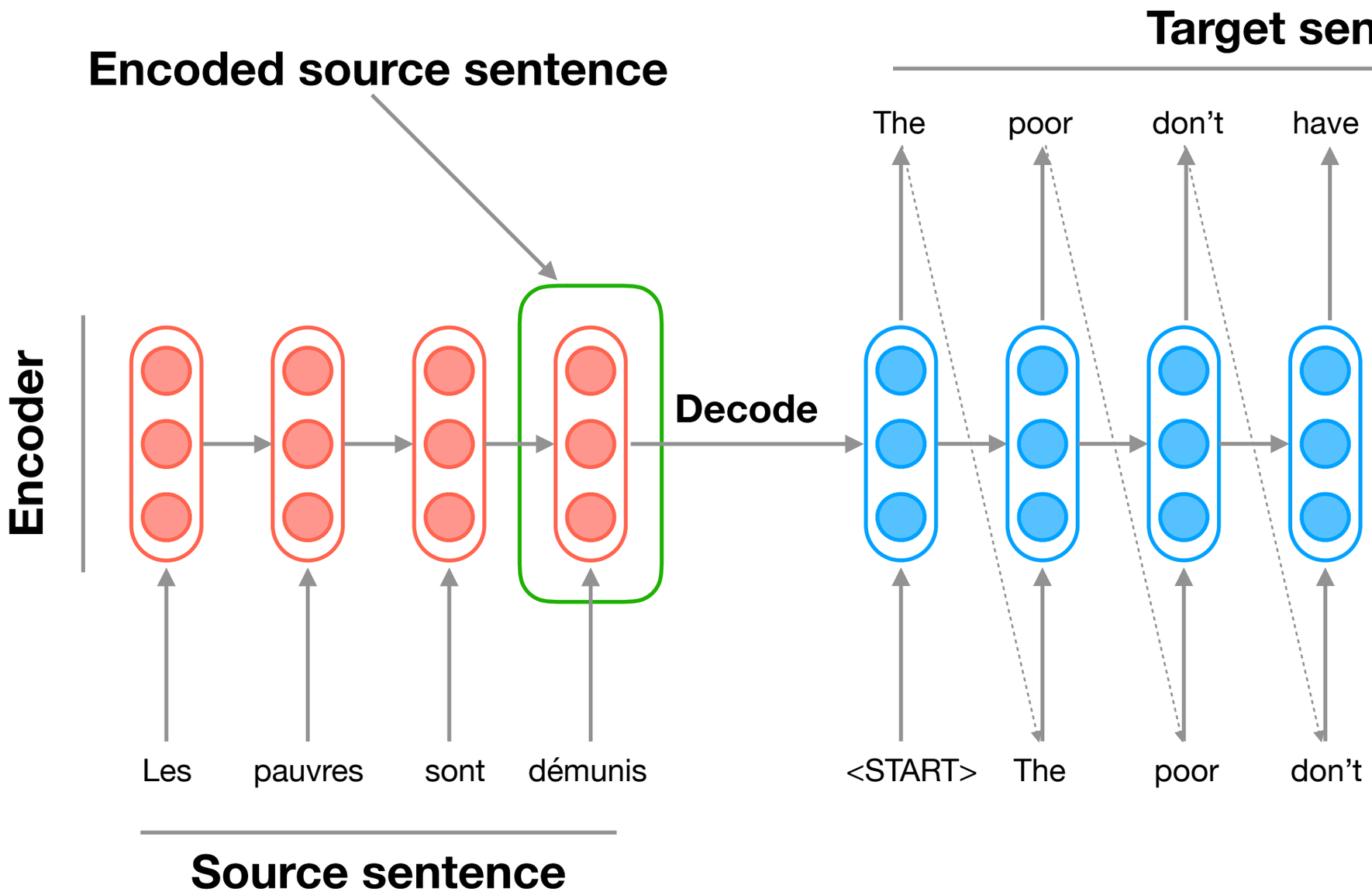
Sequence to sequence

Inference



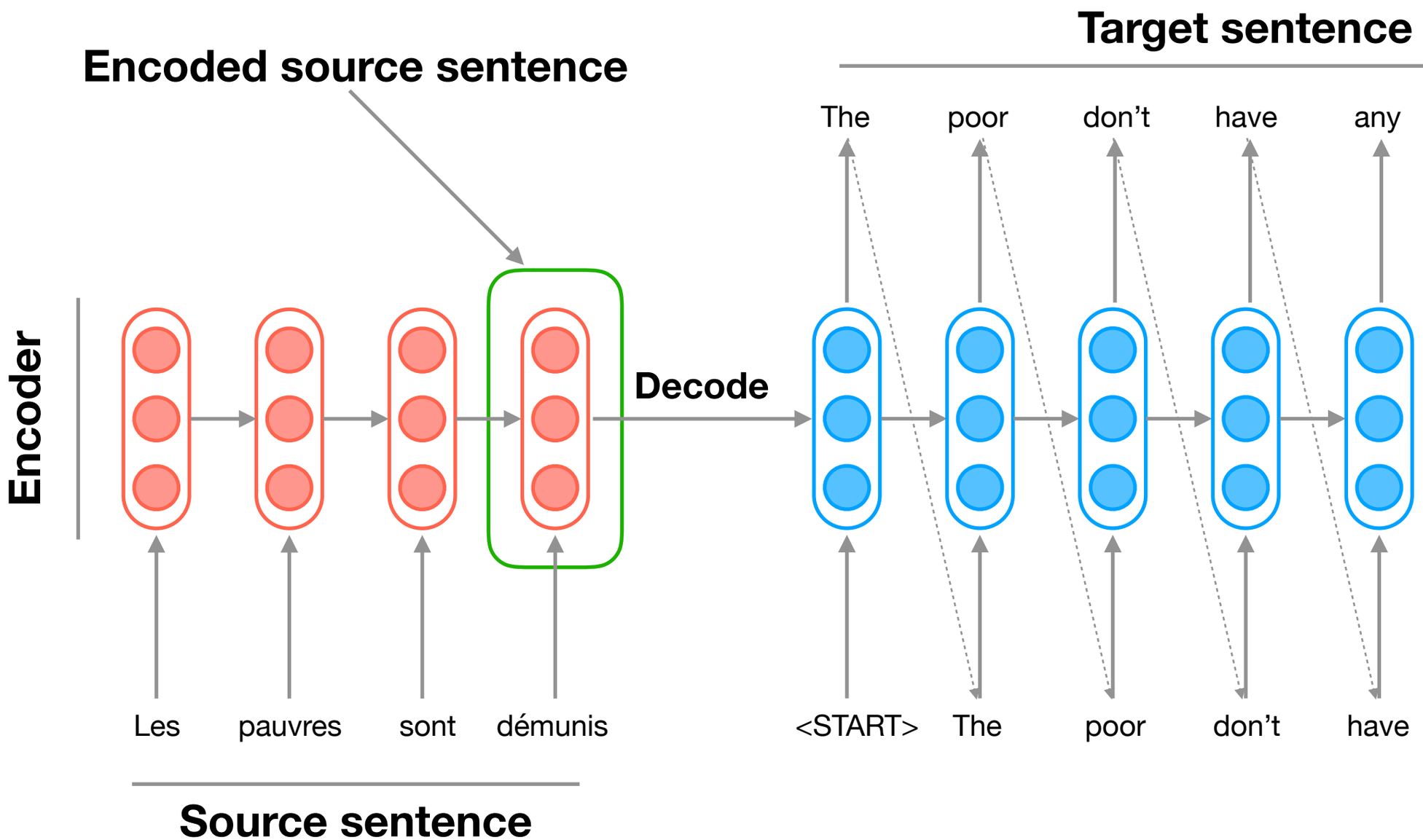
Sequence to sequence

Inference



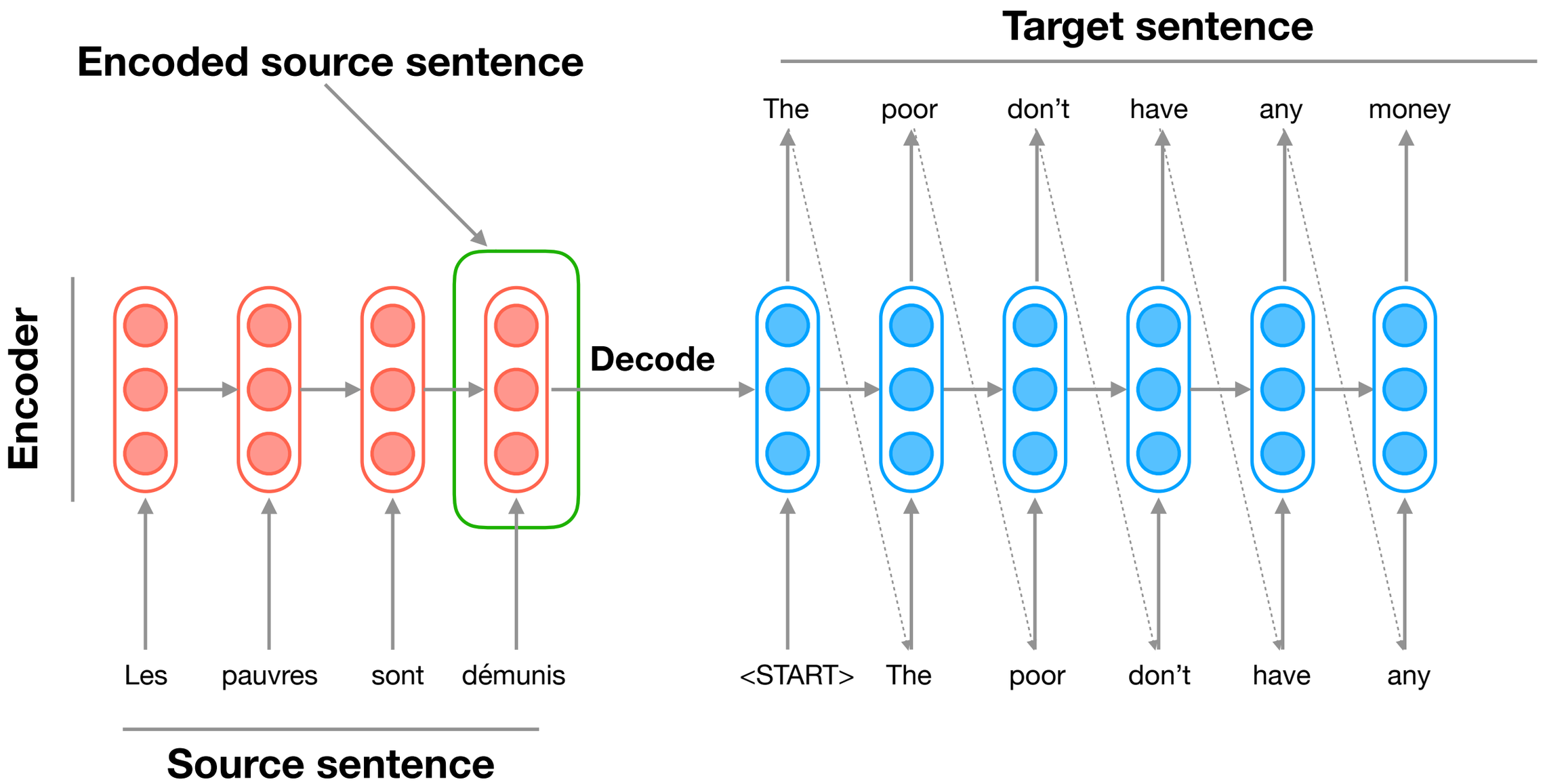
Sequence to sequence

Inference



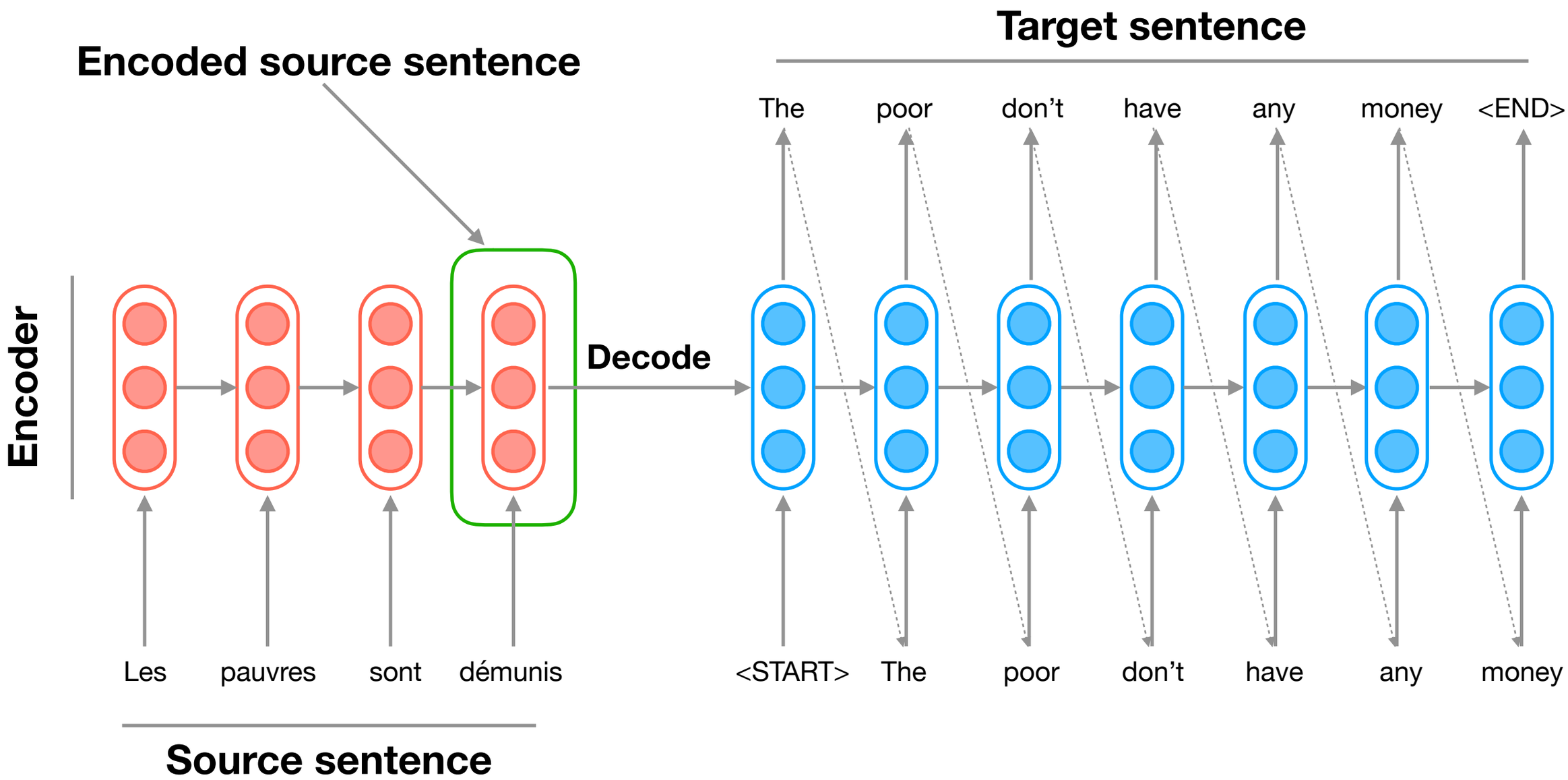
Sequence to sequence

Inference



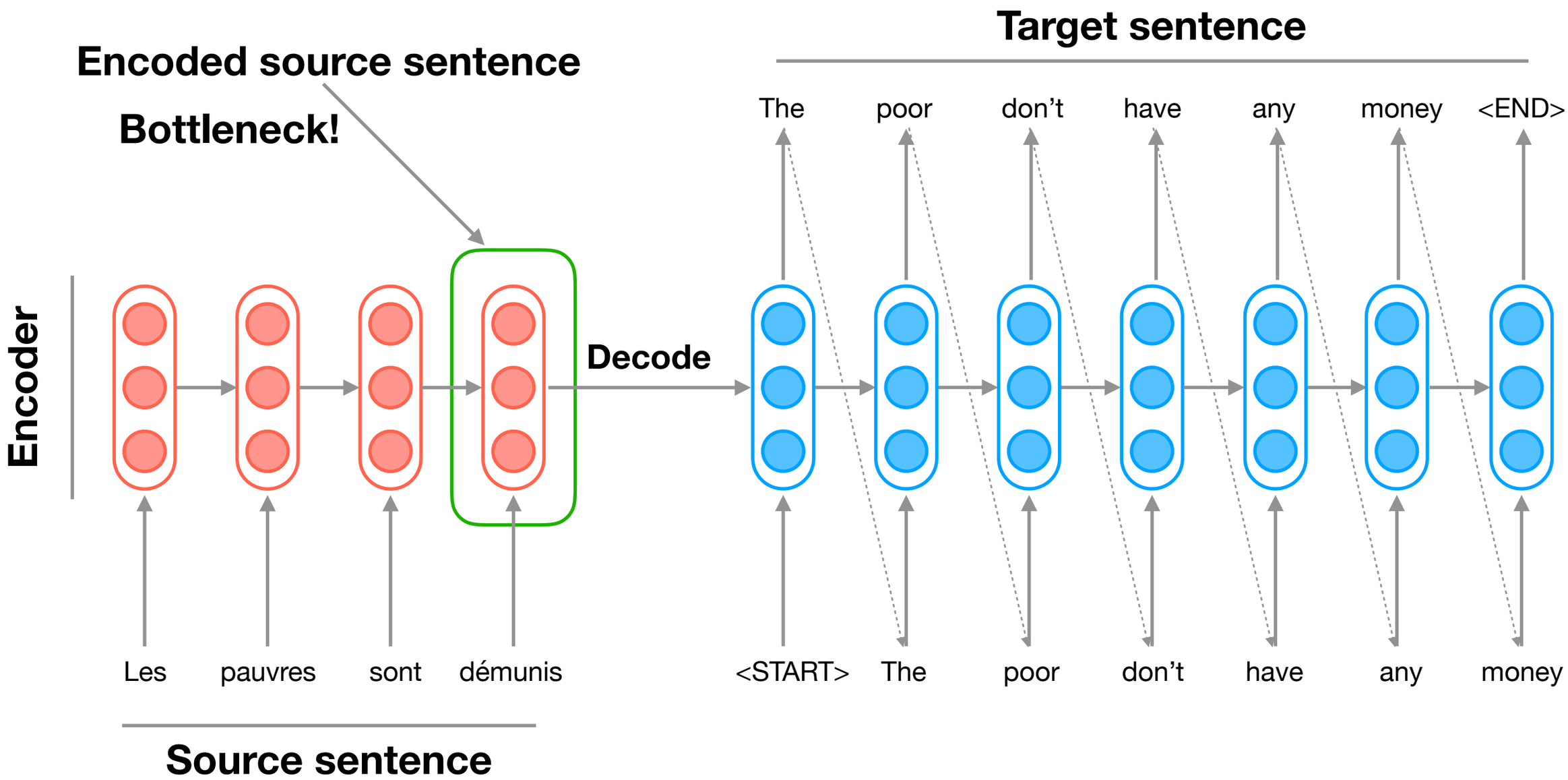
Sequence to sequence

Inference



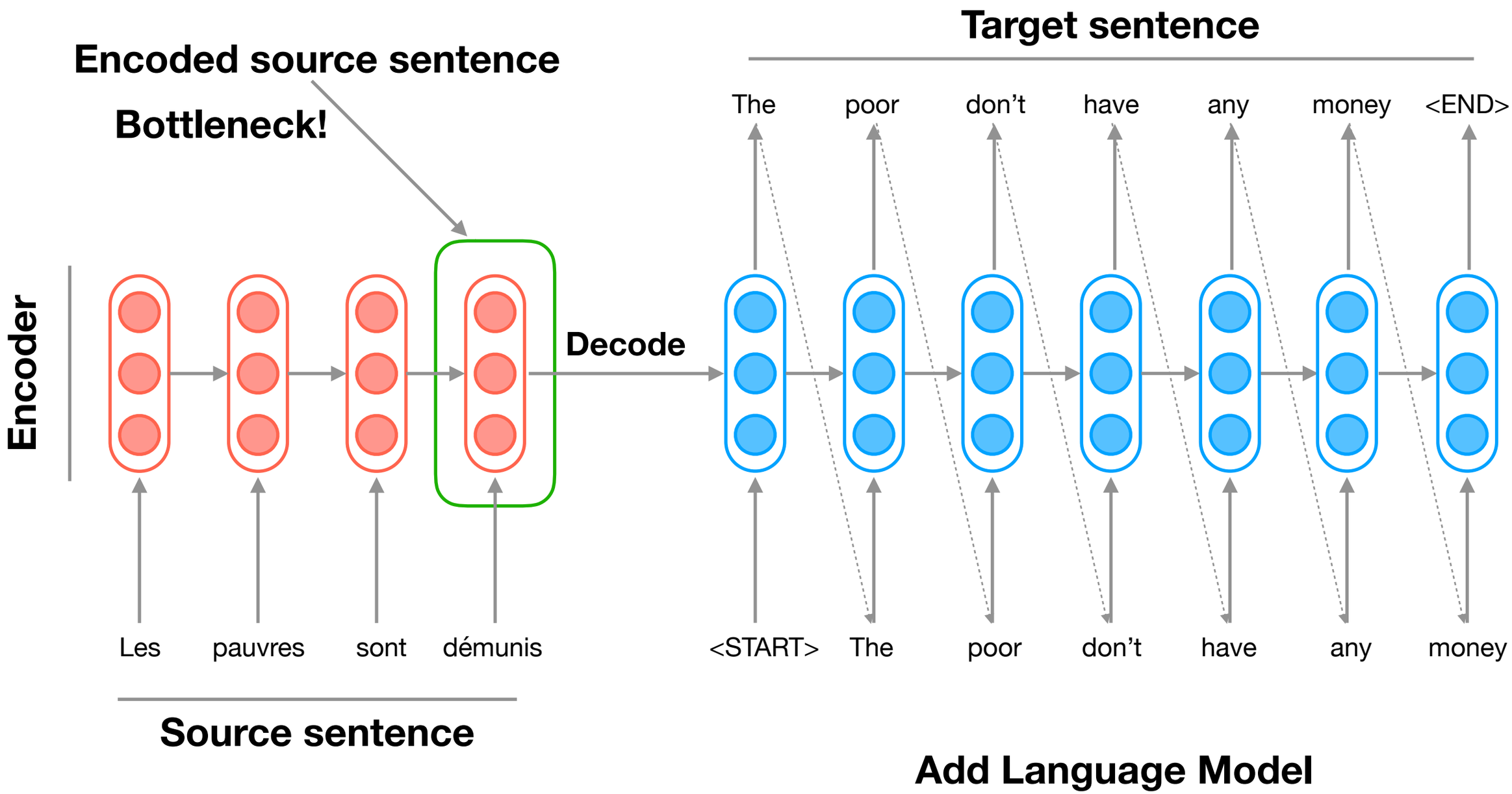
Sequence to sequence

Inference



Sequence to sequence

Inference



Contextualized Word Vectors

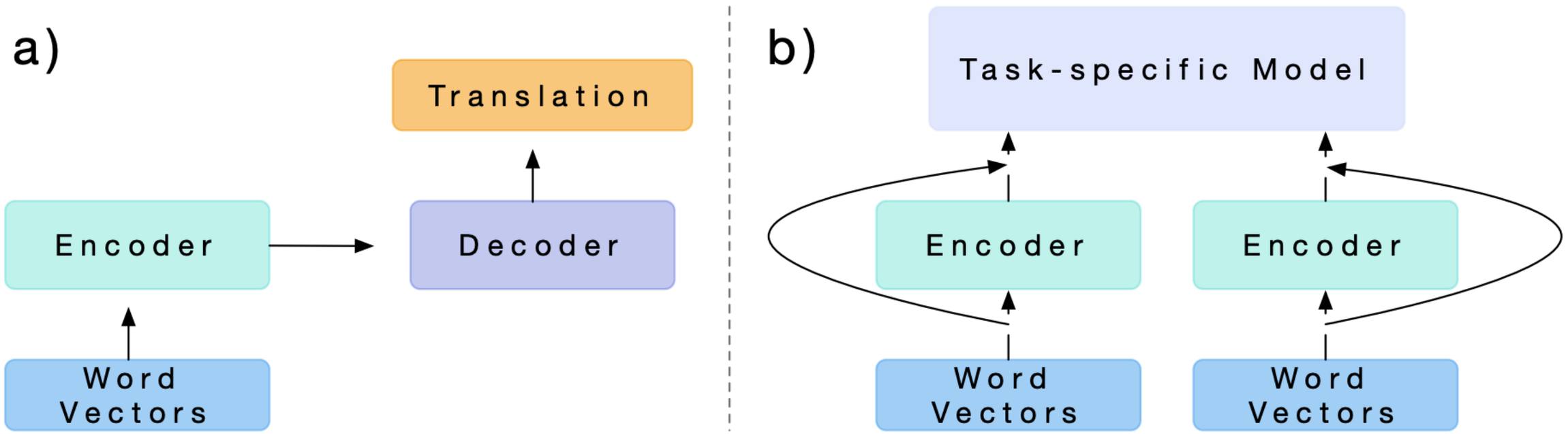


Figure 1: We a) train a two-layer, bidirectional LSTM as the encoder of an attentional sequence-to-sequence model for machine translation and b) use it to provide context for other NLP models.

Thanks for your Attention!

Boris Zubarev

@bobazooba