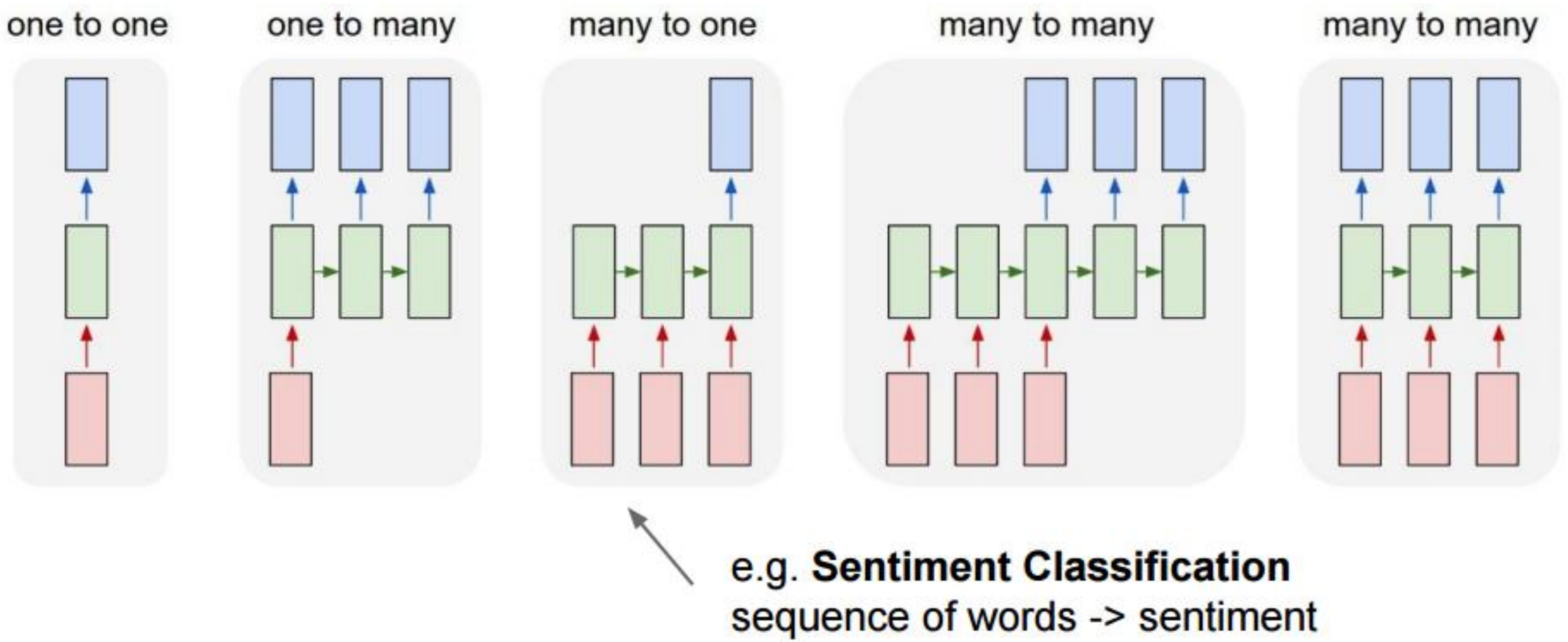
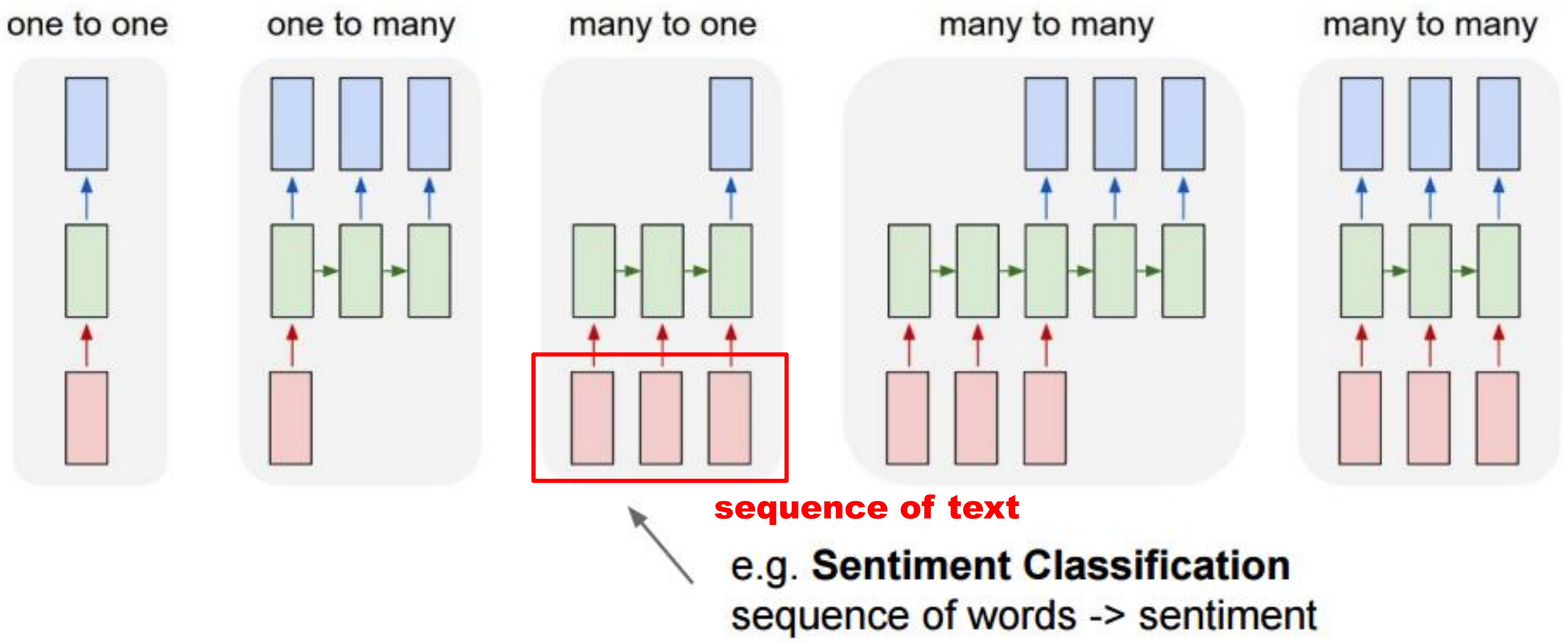
Types of RNNs

- many-to-one



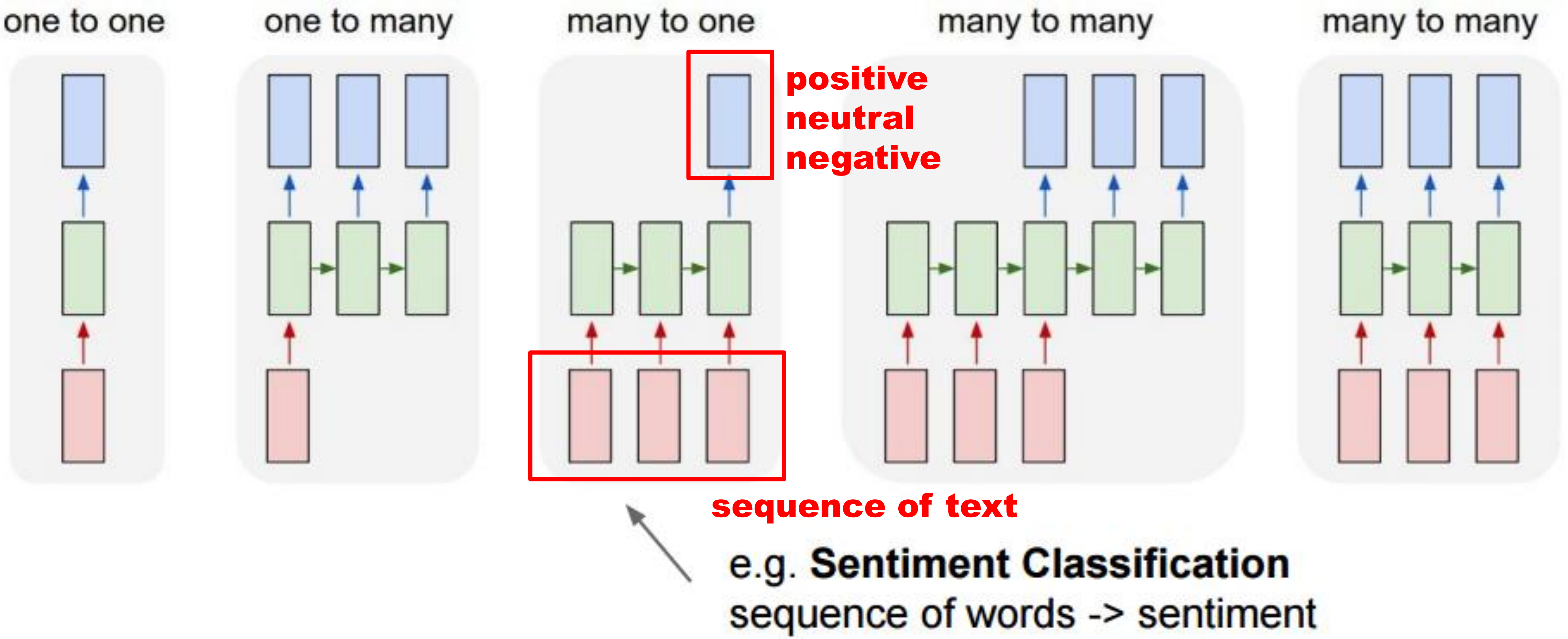
Types of RNNs

- many-to-one



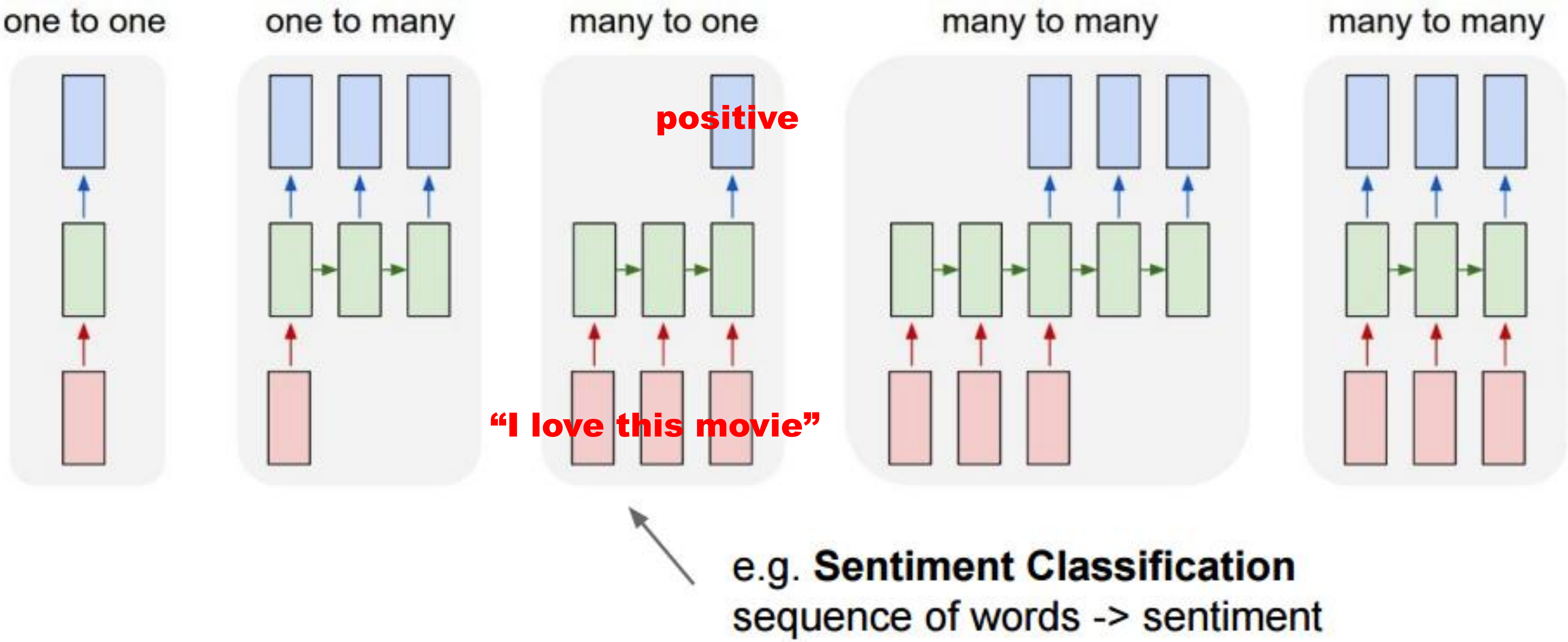
Types of RNNs

- many-to-one



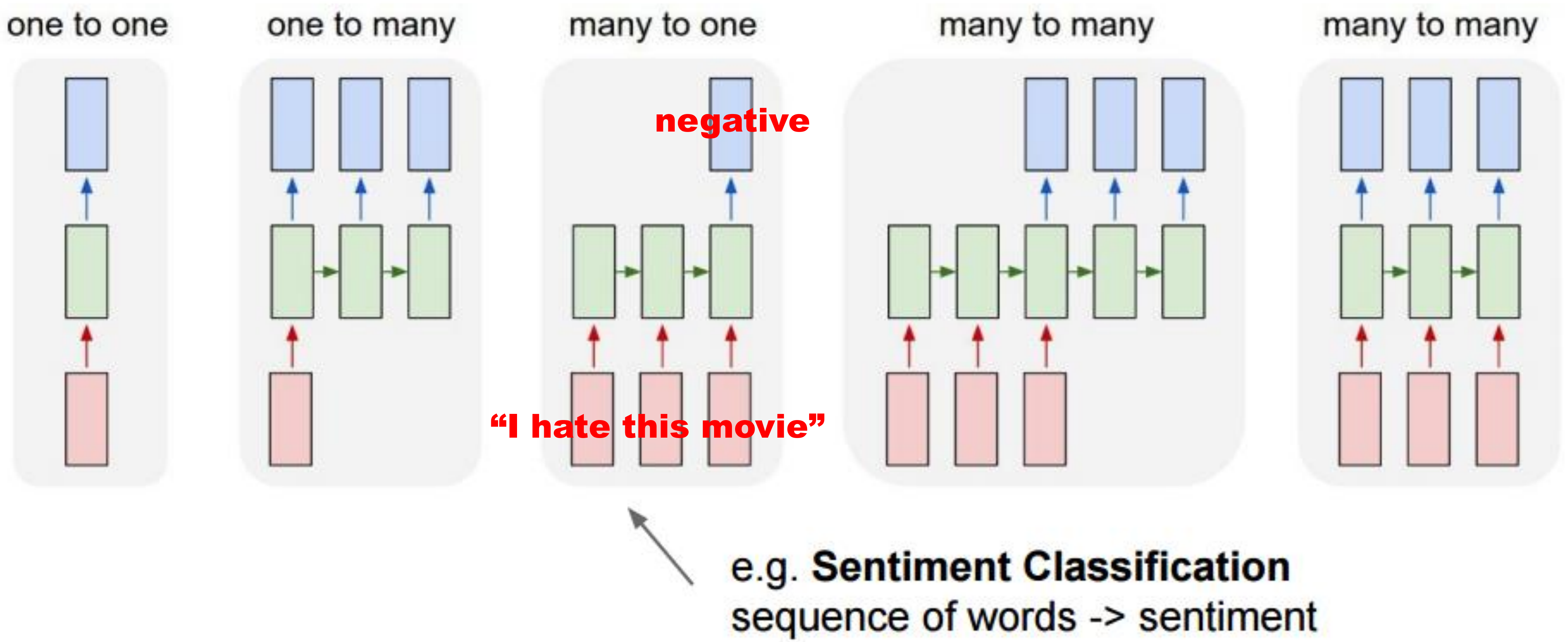
Types of RNNs

- many-to-one



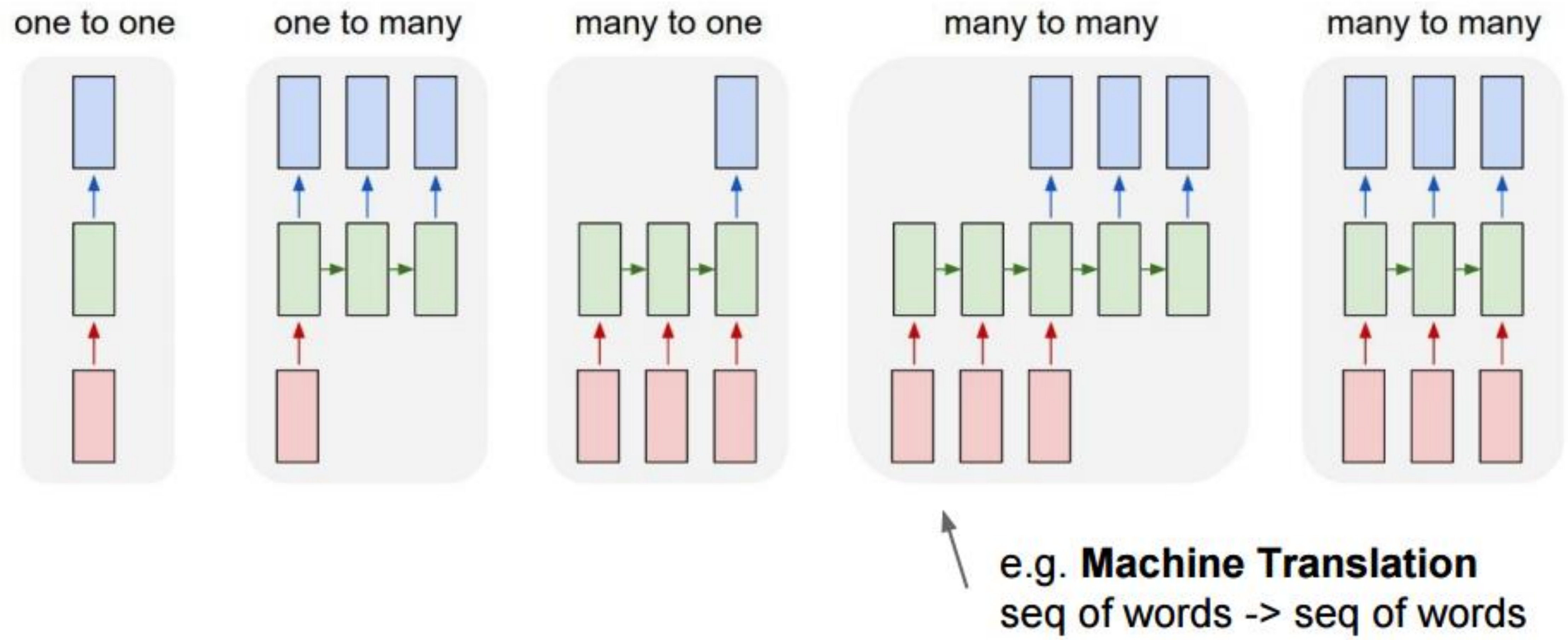
Types of RNNs

- many-to-one



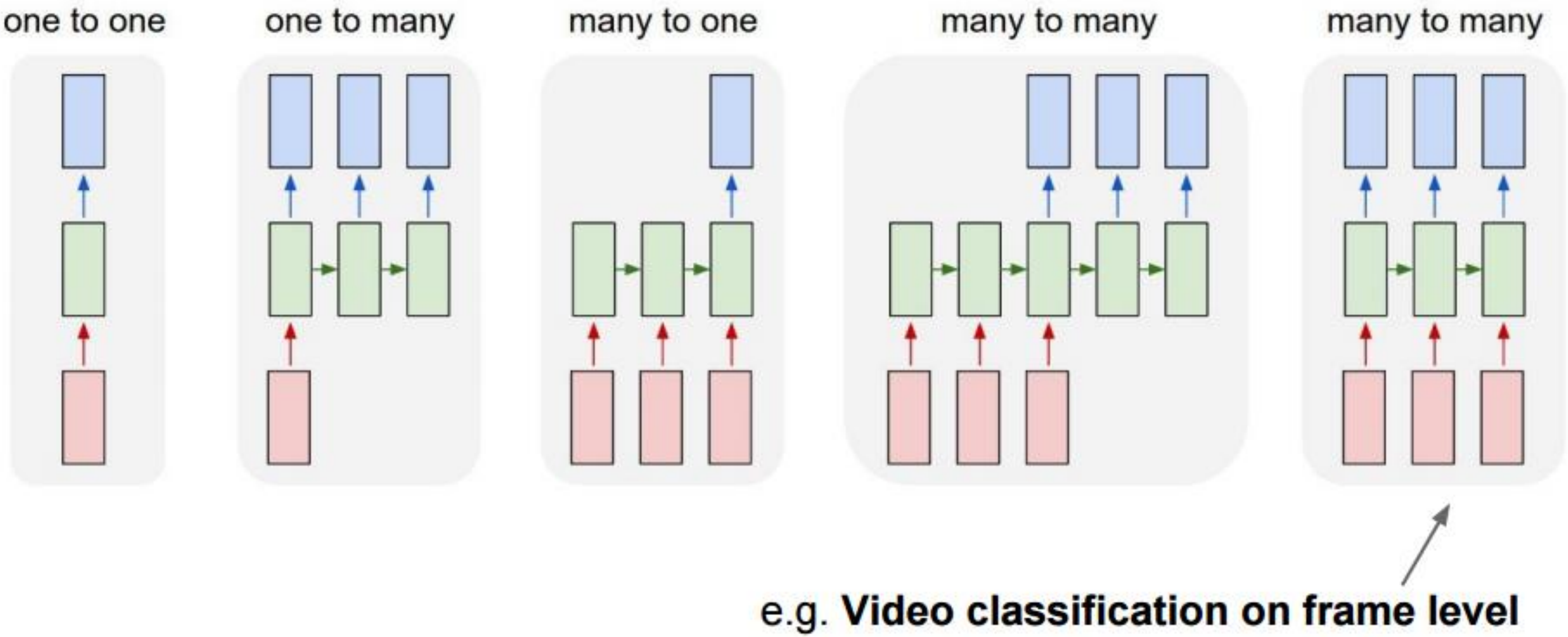
Types of RNNs

- Sequence-to sequence



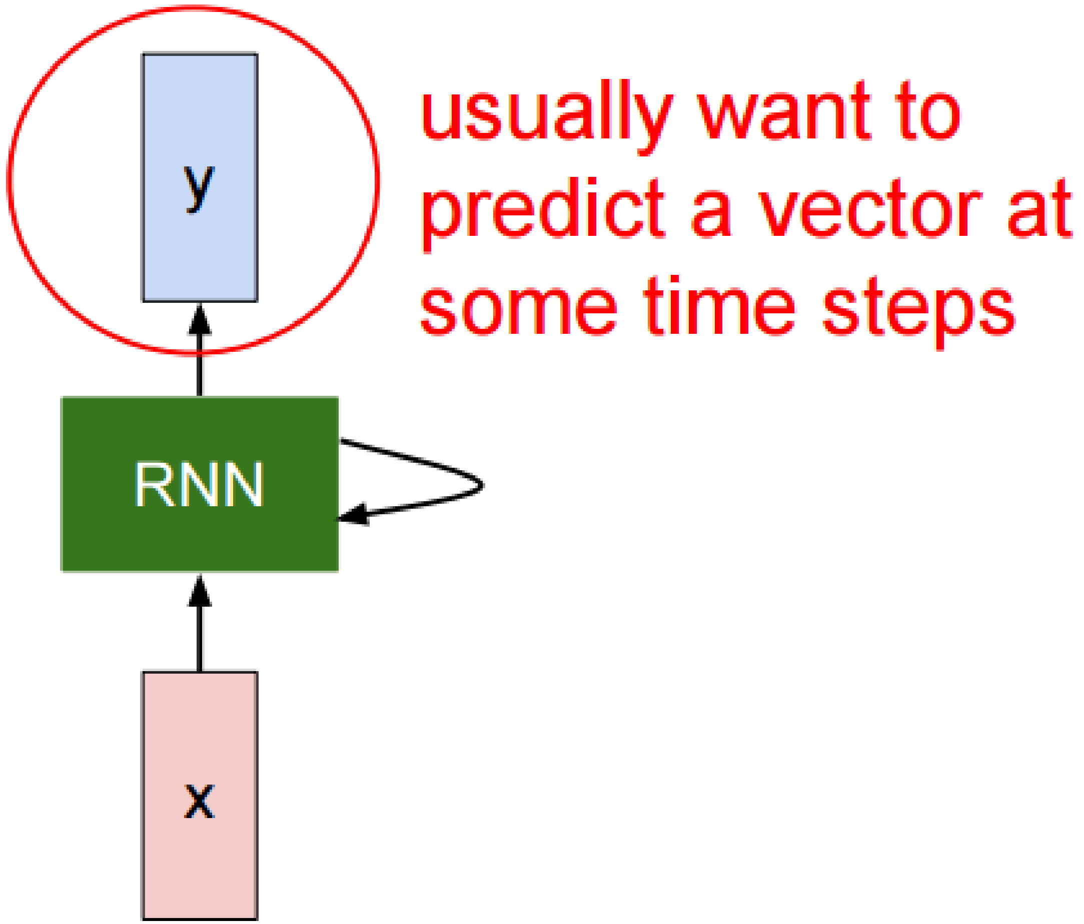
Types of RNNs

- Sequence-to sequence



Recurrent Neural Network

- Inputs and outputs of RNNs (rolled version)



Recurrent Neural Network

- How to calculate the hidden state of RNNs

We can process a sequence of vectors **x** by applying a recurrence formula at every time step:

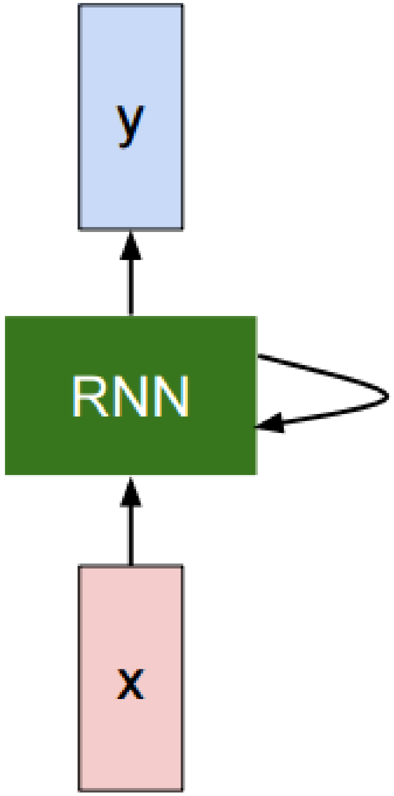
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

input vector at some time step

some function with parameters W



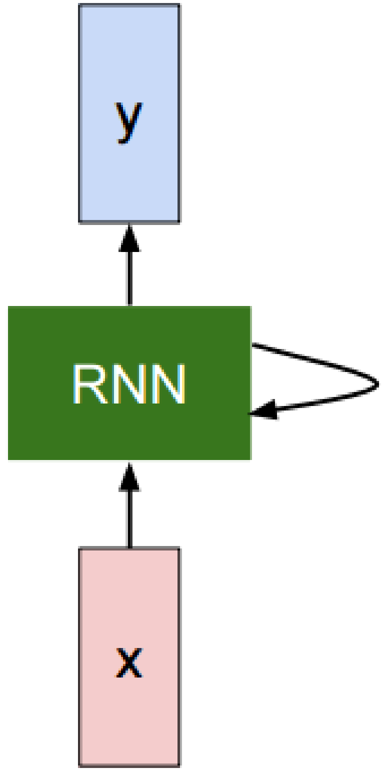
Recurrent Neural Network

- How to calculate the hidden state of RNNs

We can process a sequence of vectors **x** by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

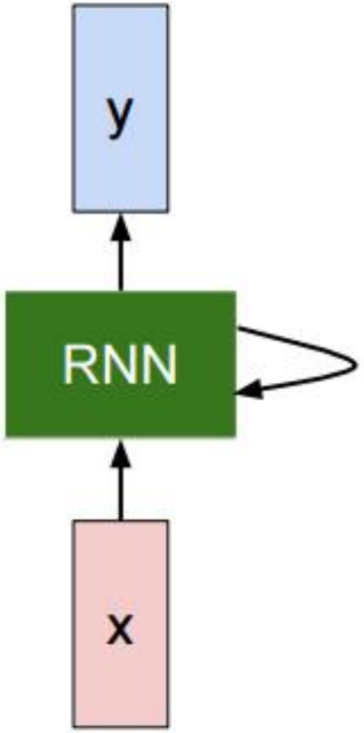
Notice: the same function and the same set of parameters are used at every time step.



Recurrent Neural Network

- How to calculate output of RNN

The state consists of a single *"hidden"* vector **h**:



$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$y_t = W_{hy} h_t$$

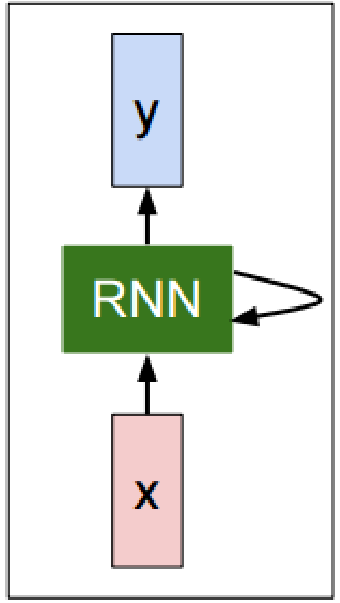
Character-level Language Model

- Example of training sequence "hello"

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
"hello"



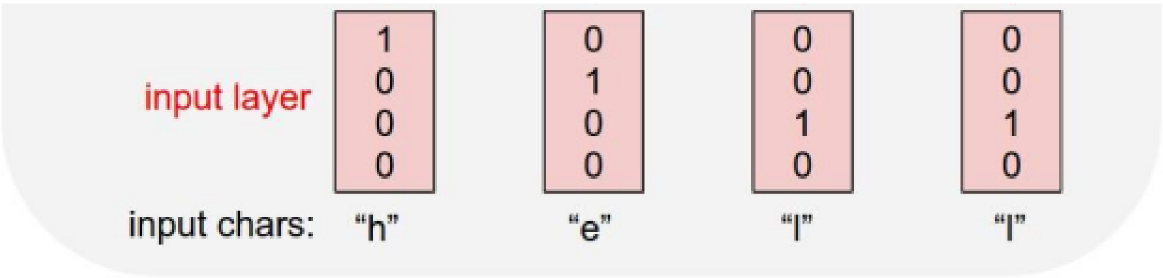
Character-level Language Model

- Example of training sequence "hello"

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
"hello"



Character-level Language Model

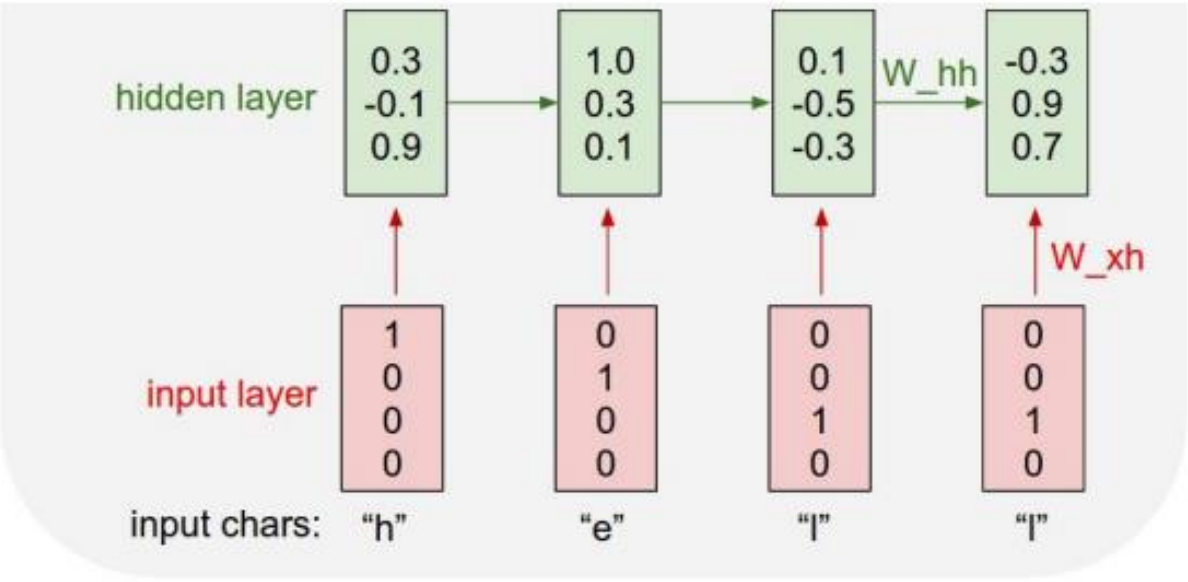
- Example of training sequence "hello"

Character-level language model example

$$\boxed{h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)}$$

Vocabulary:
[h,e,l,o]

Example training sequence:
"hello"



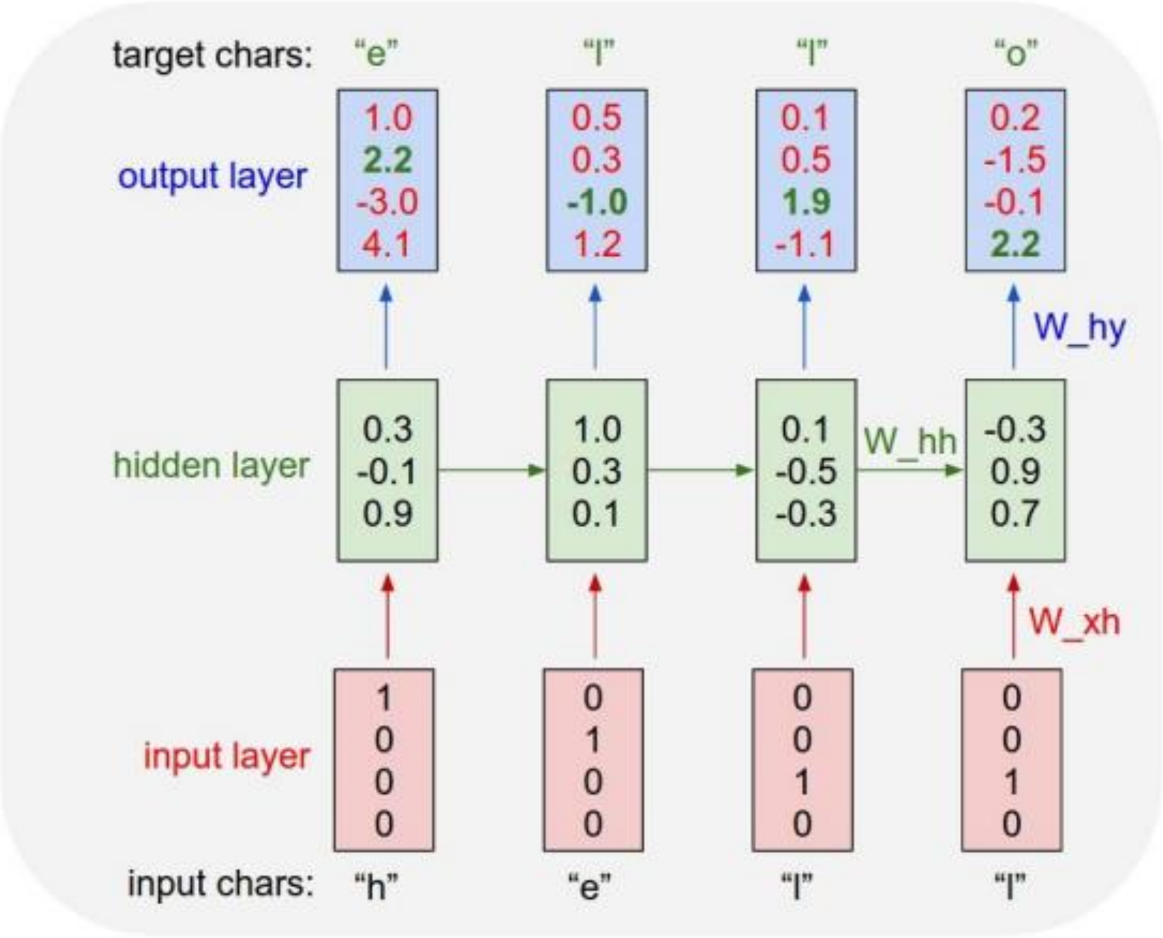
Character-level Language Model

- Example of training sequence “hello”

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”



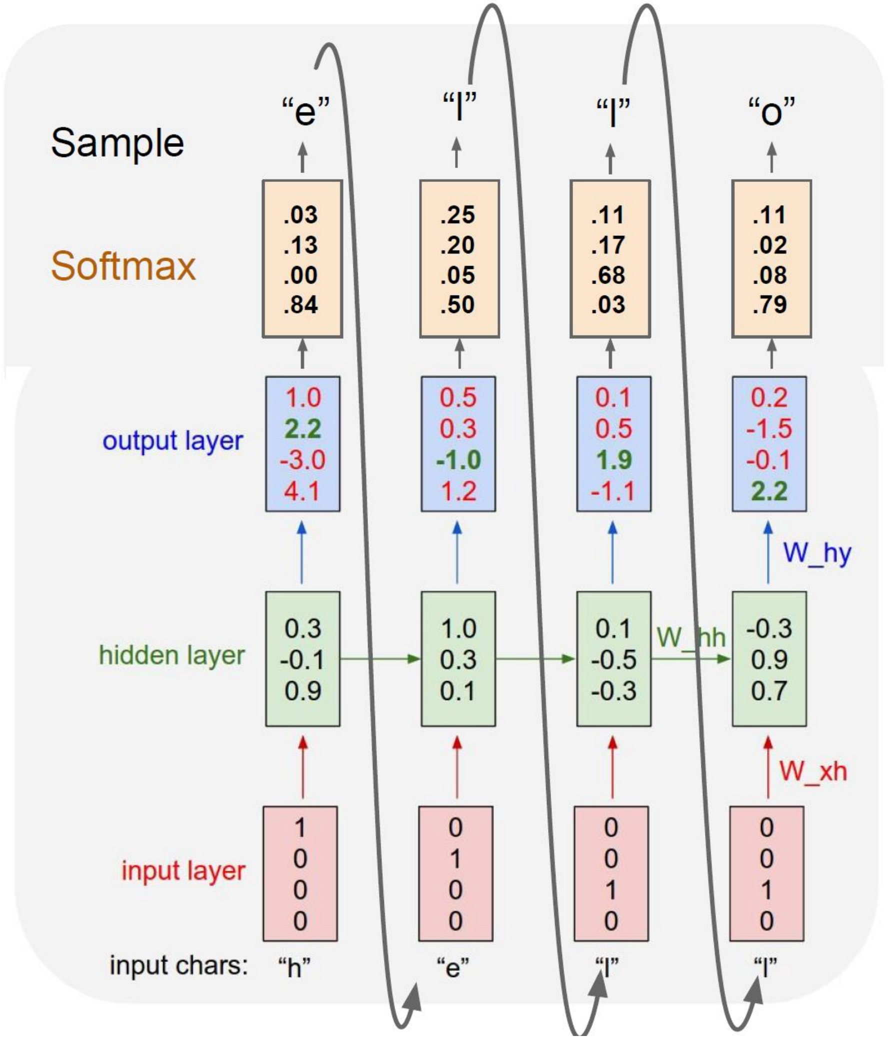
Character-level Language Model

- Example of training sequence "hello"

Example: Character-level Language Model Sampling

Vocabulary: [h,e,l,o]

At test-time sample characters one at a time, feed back to model



min-char-rnn.py

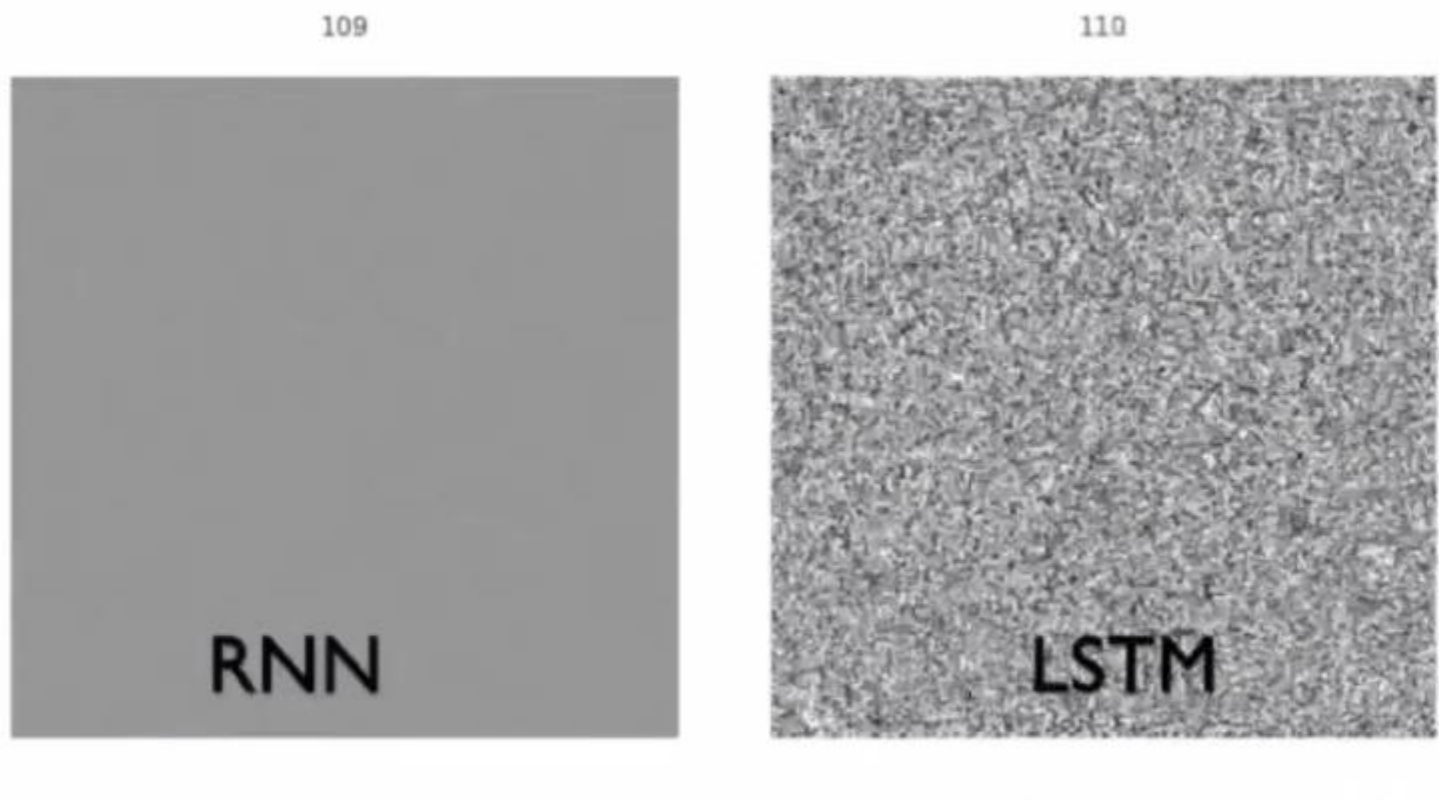
min-char-rnn.py gist: 112 lines of Python

```python
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
"""
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }

# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]

def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in xrange(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes

n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

(https://gist.github.com/karpathy/d4dee566867f8291f086)

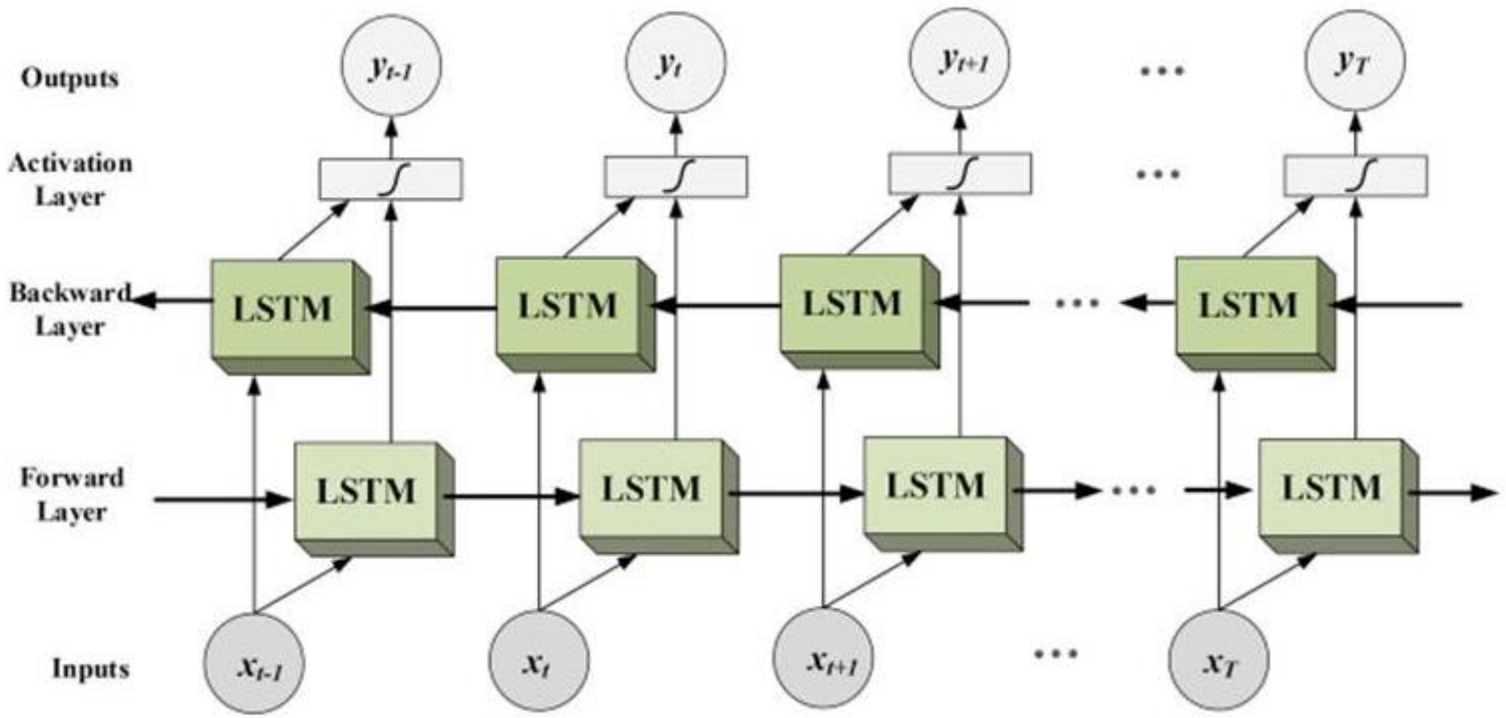
Vanishing Gradient Problem

- The reason why the vanishing gradient problem is important:



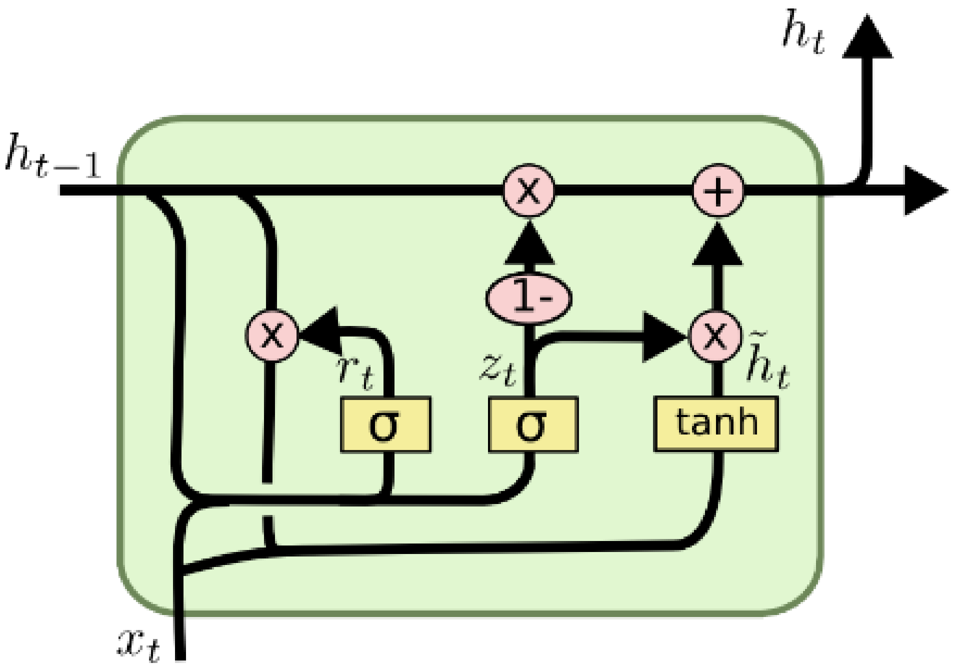
[RNN vs LSTM: Vanishing Gradients - GIF on Imgur]

Bidirectional RNN



GRU

- What is GRU(Gated Recurrent Unit)?



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

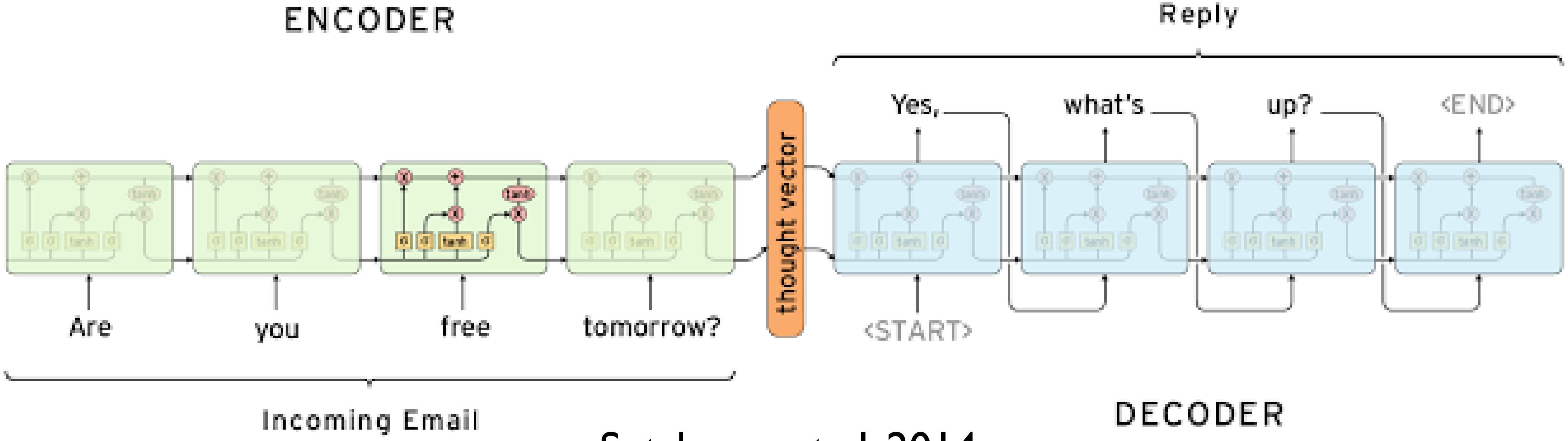
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

[Understanding LSTM Networks -- colah's blog]

Seq2Seq Model

- It takes a sequence of words as input and gives a sequence of words as output.
- It composed of an encoder and a decoder.
- Many NLP applications exist, e.g., machine translation, dialog systems, and so on.

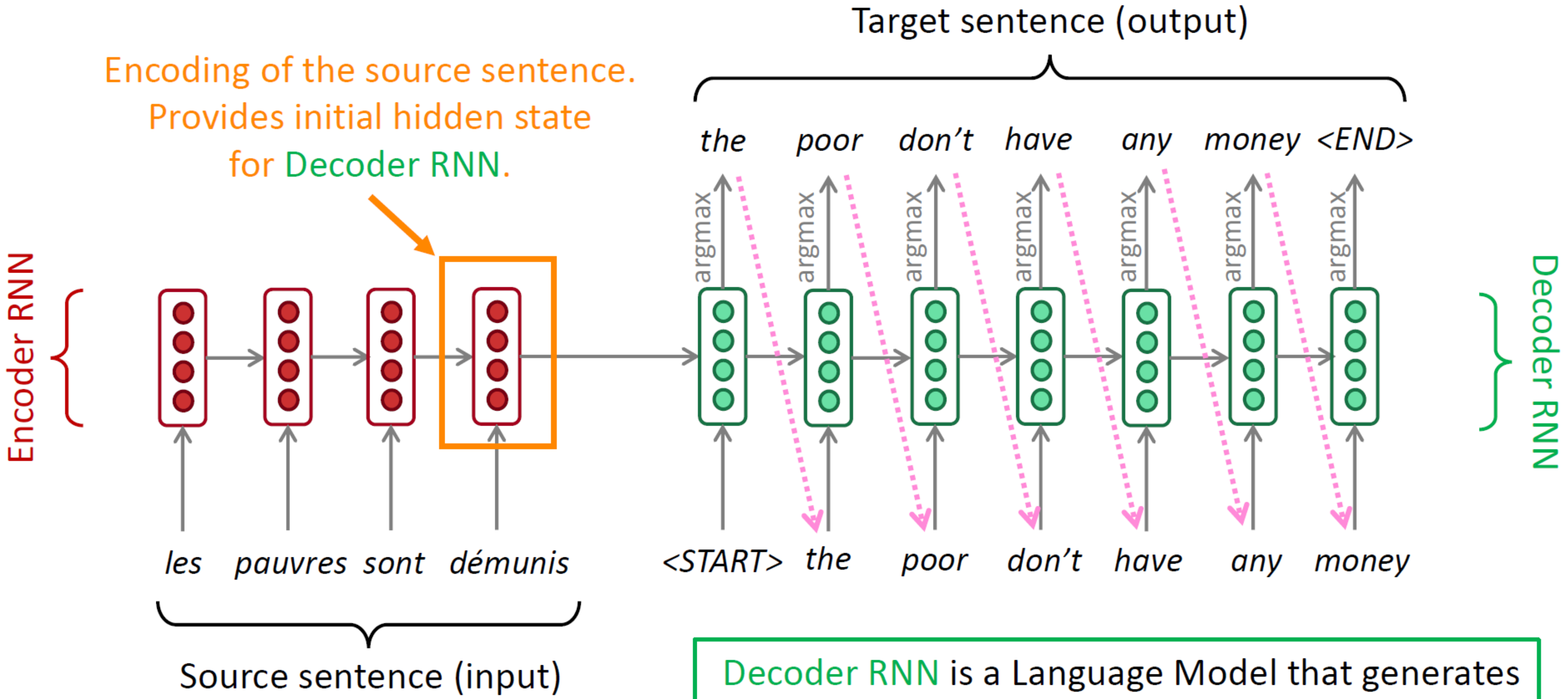


Sutskever et al. 2014

"Sequence to Sequence Learning with Neural Networks"

Encode source into fixed length vector, use it as initial recurrent state for target decoder model

Seq2Seq for Machine Translation

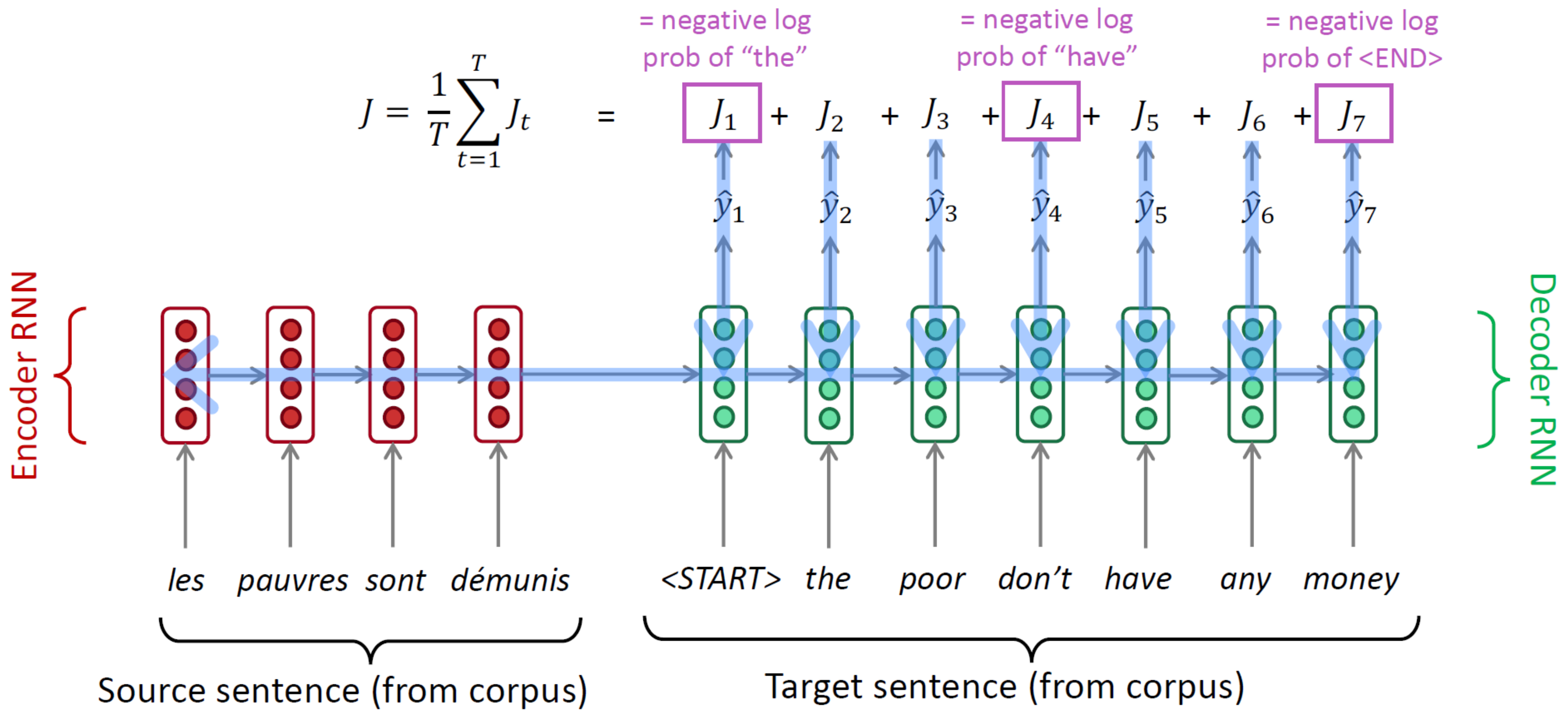


Decoder RNN is a Language Model that generates target sentence conditioned on encoding.

Encoder RNN produces an encoding of the source sentence.

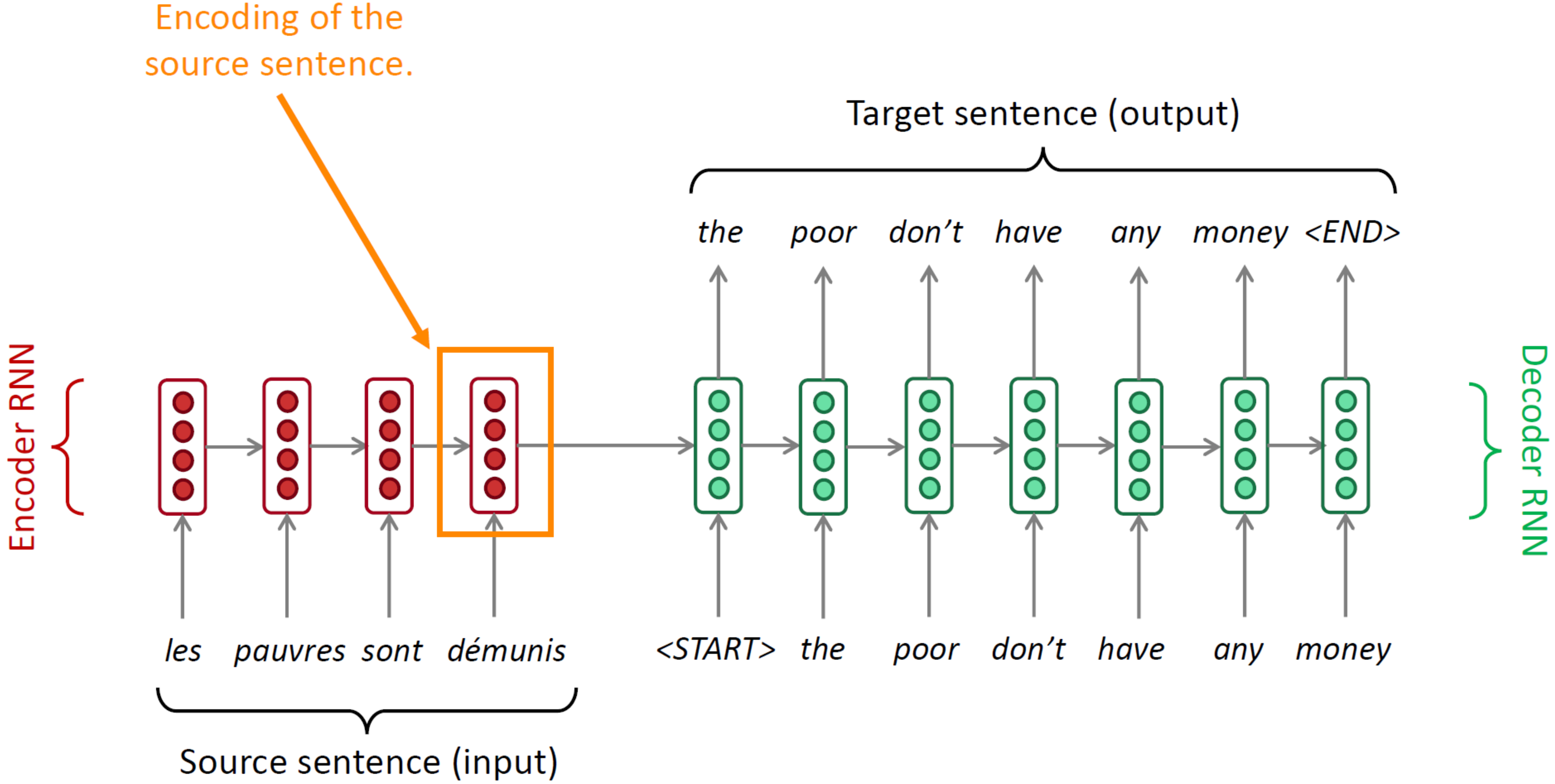
Note: This diagram shows **test time** behavior: decoder output is fed in ⋯⋯> as next step's input

Seq2Seq Model: Training



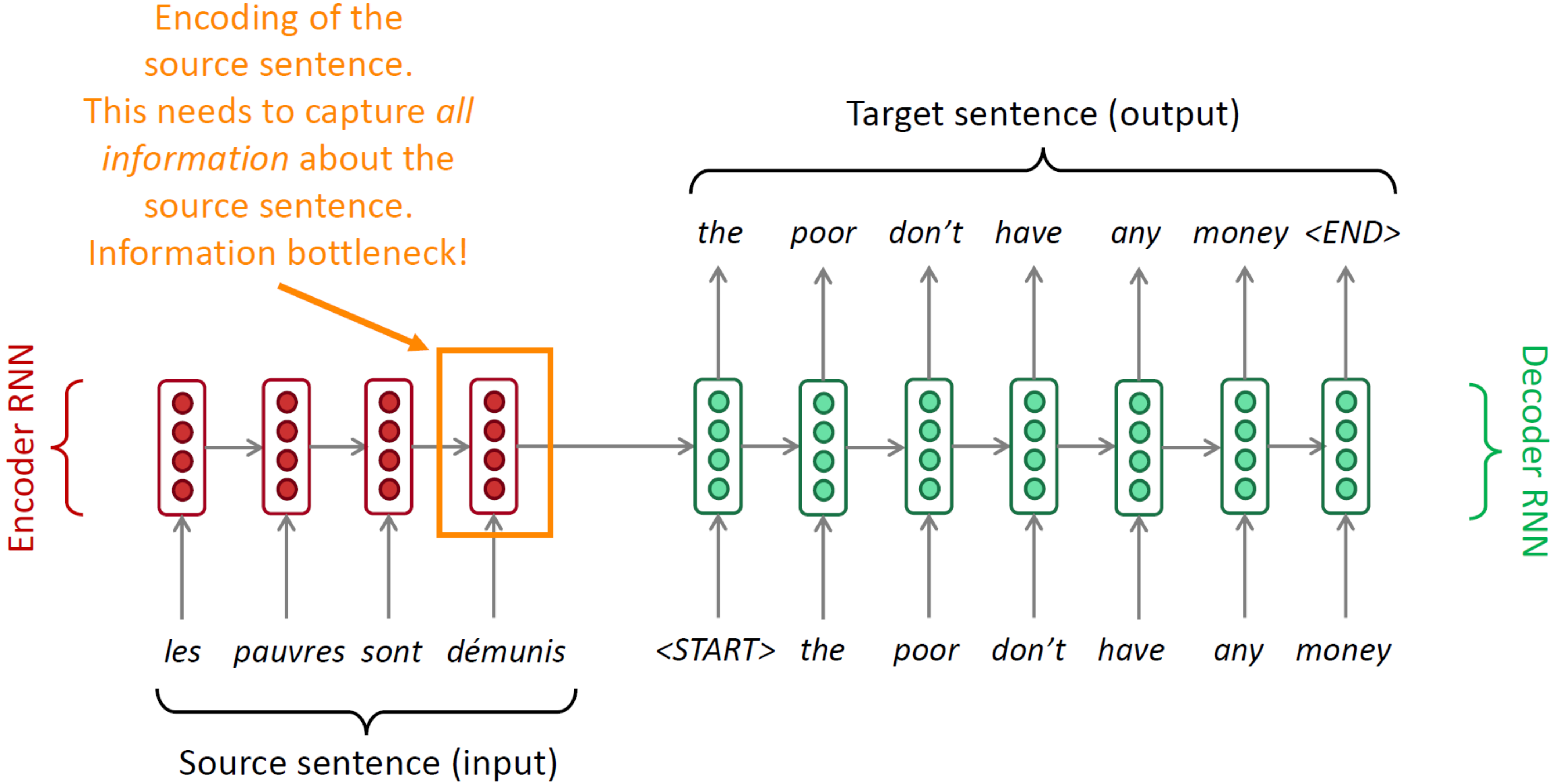
Seq2seq is optimized as a **single system.**
Backpropagation operates *"end to end"*.

Seq2Seq Model: Bottleneck Problem



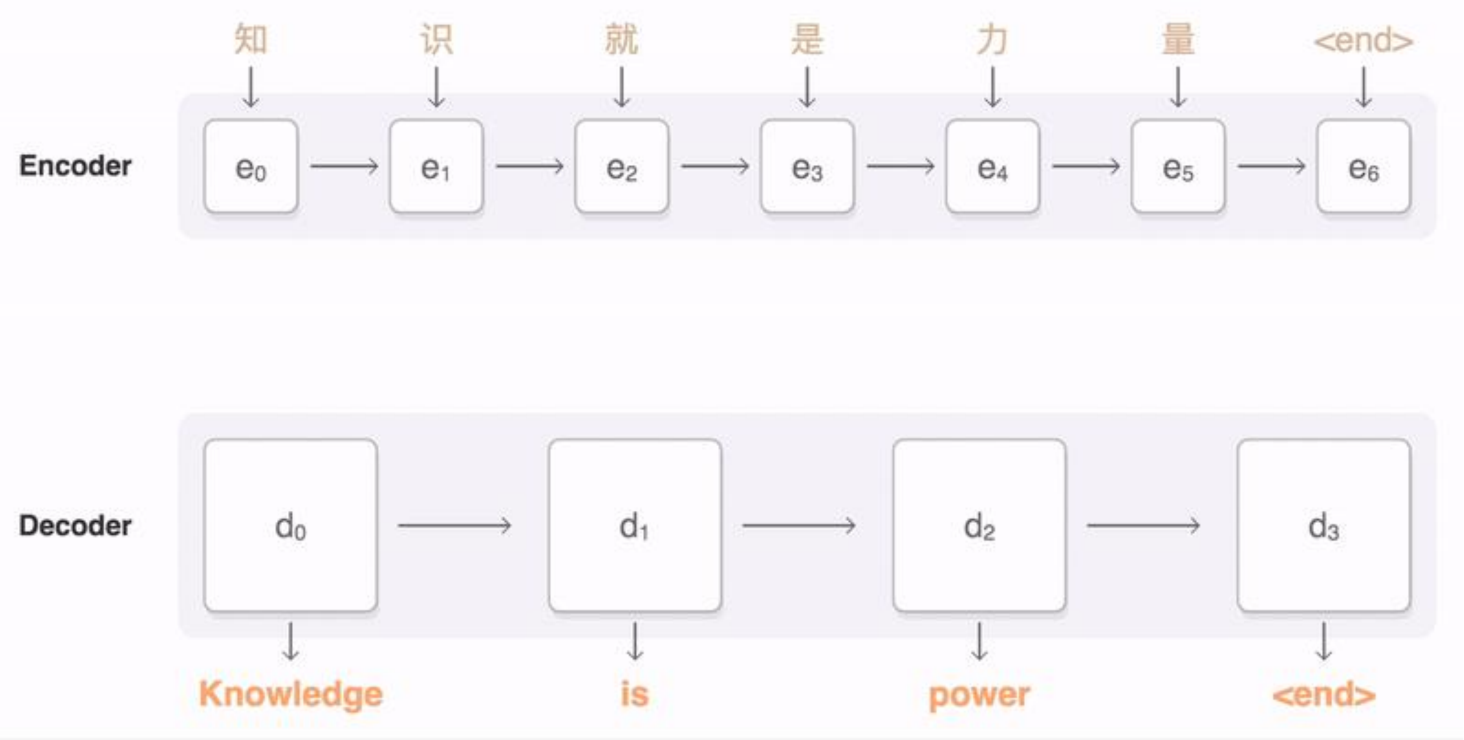
Problems with this architecture?

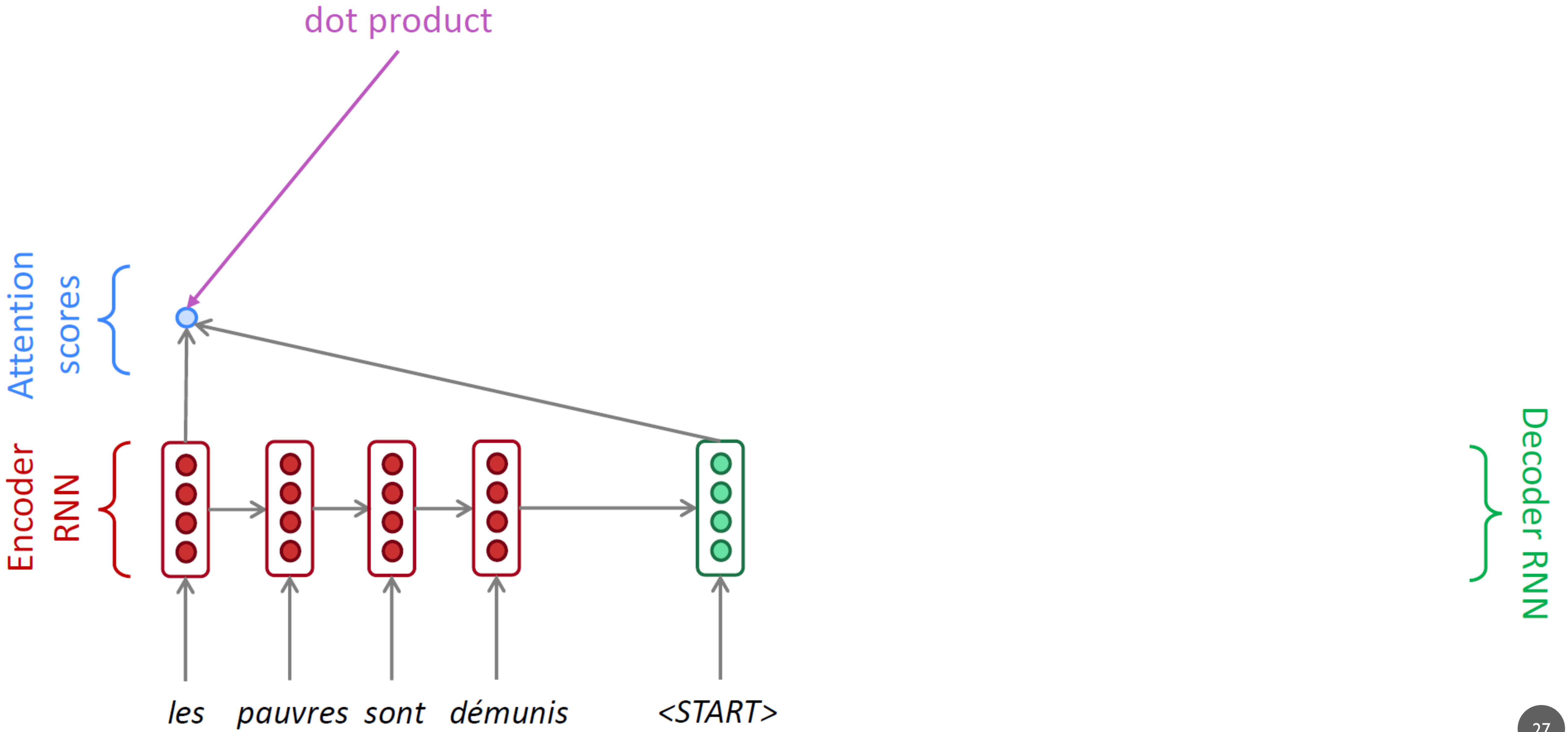
Seq2Seq Model: Bottleneck Problem

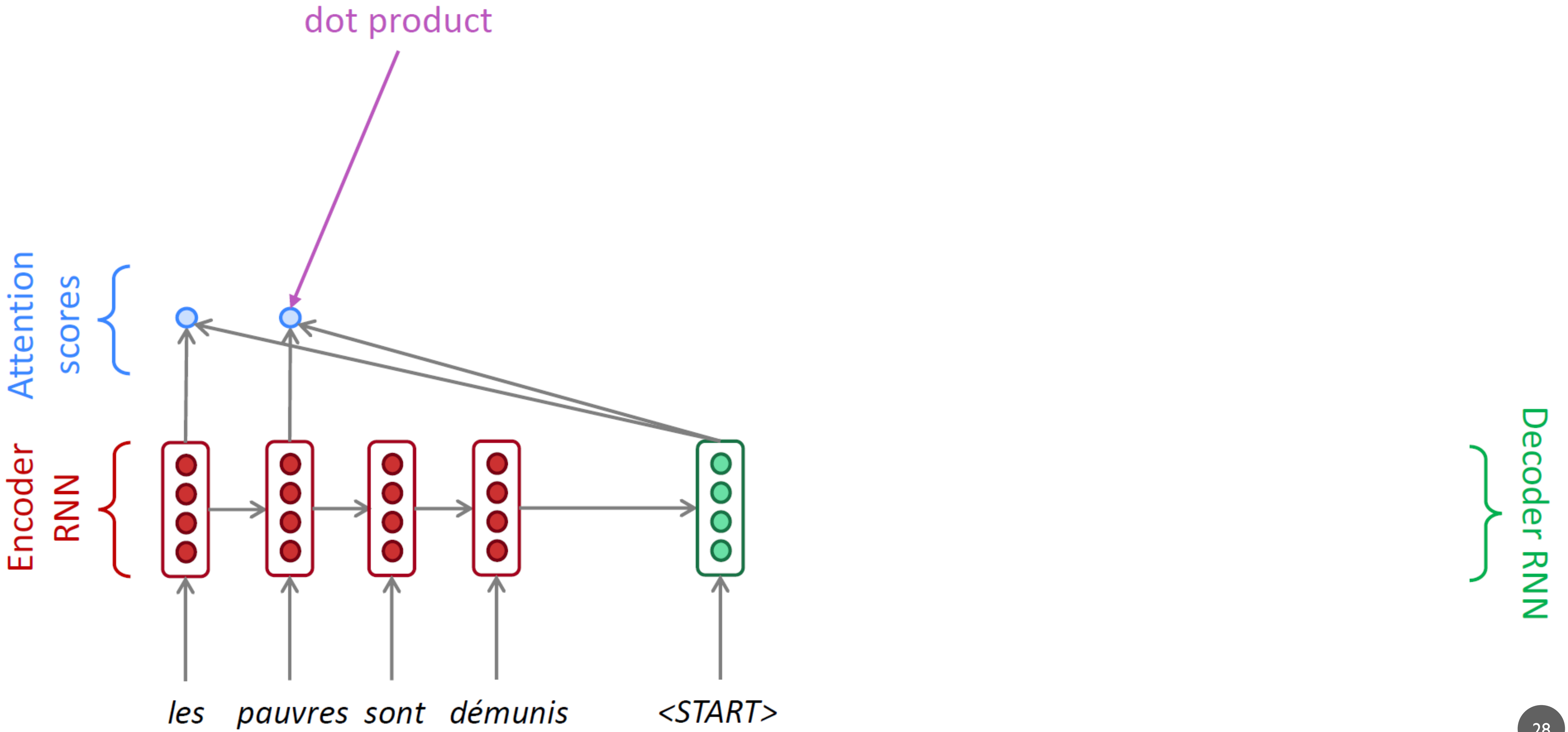


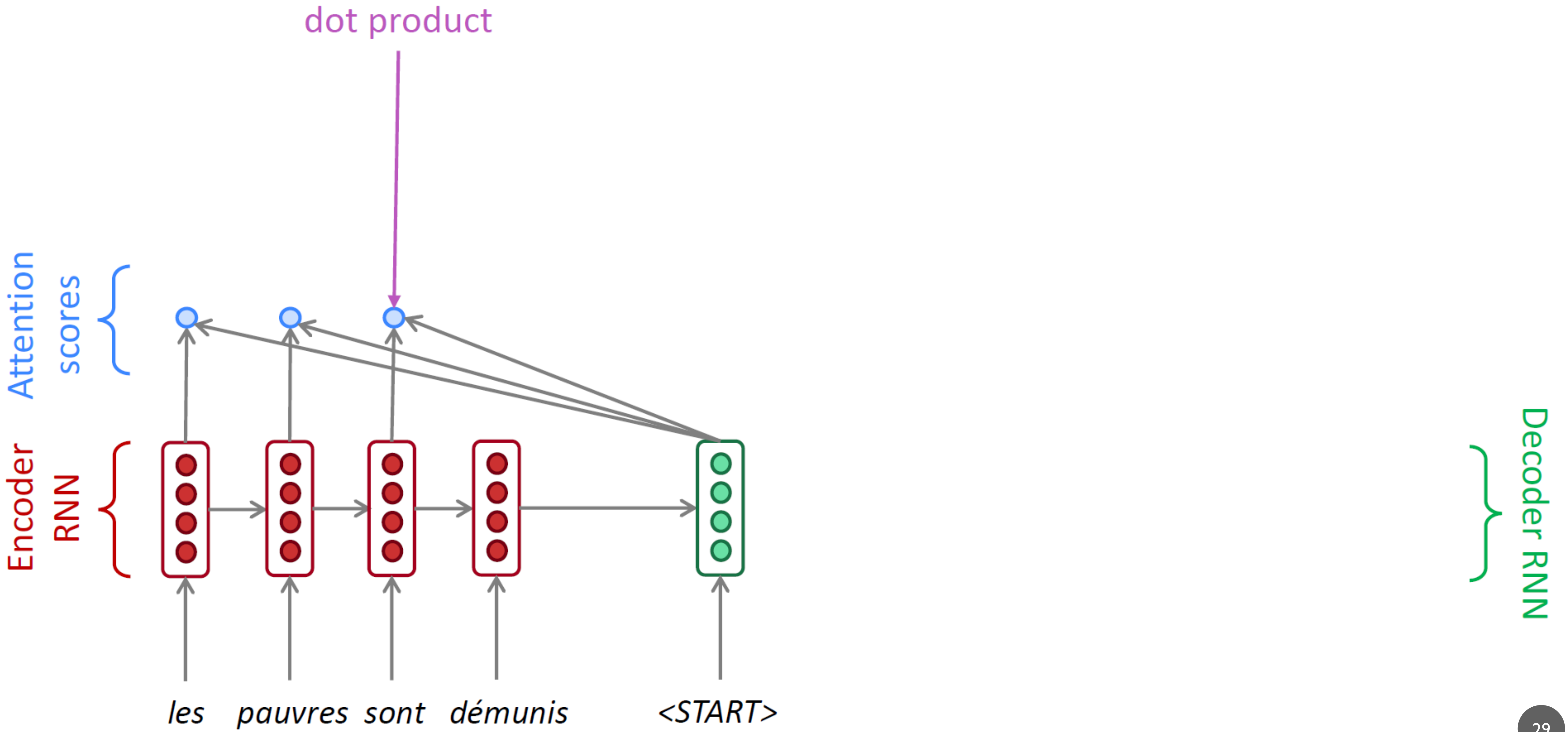
Seq2Seq with Attention

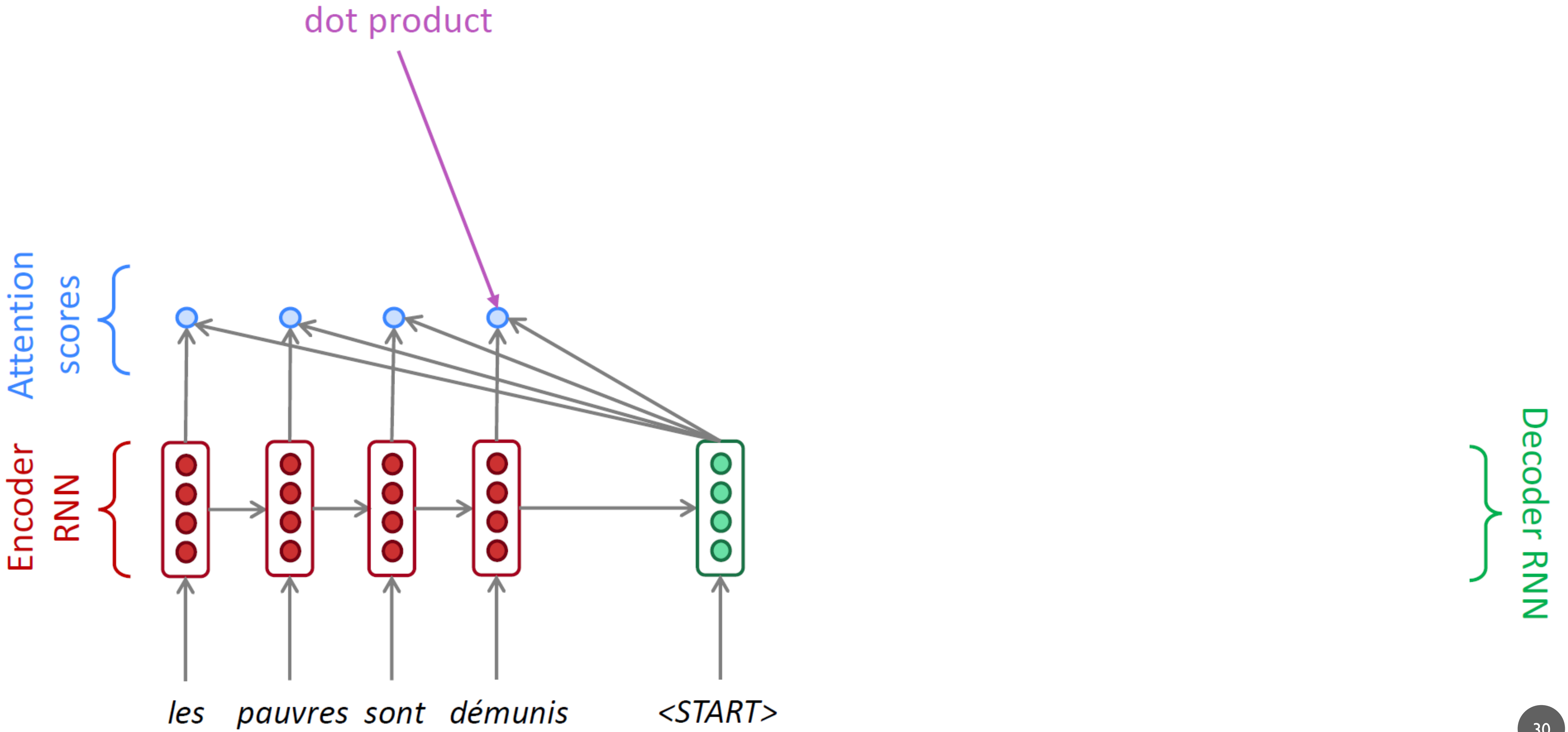
- Attention provides a solution to the bottleneck problem.
- Core idea: At each time step of the decoder, focus on a particular part of the source sequence
- Tensorflow official implementation: https://github.com/google/seq2seq

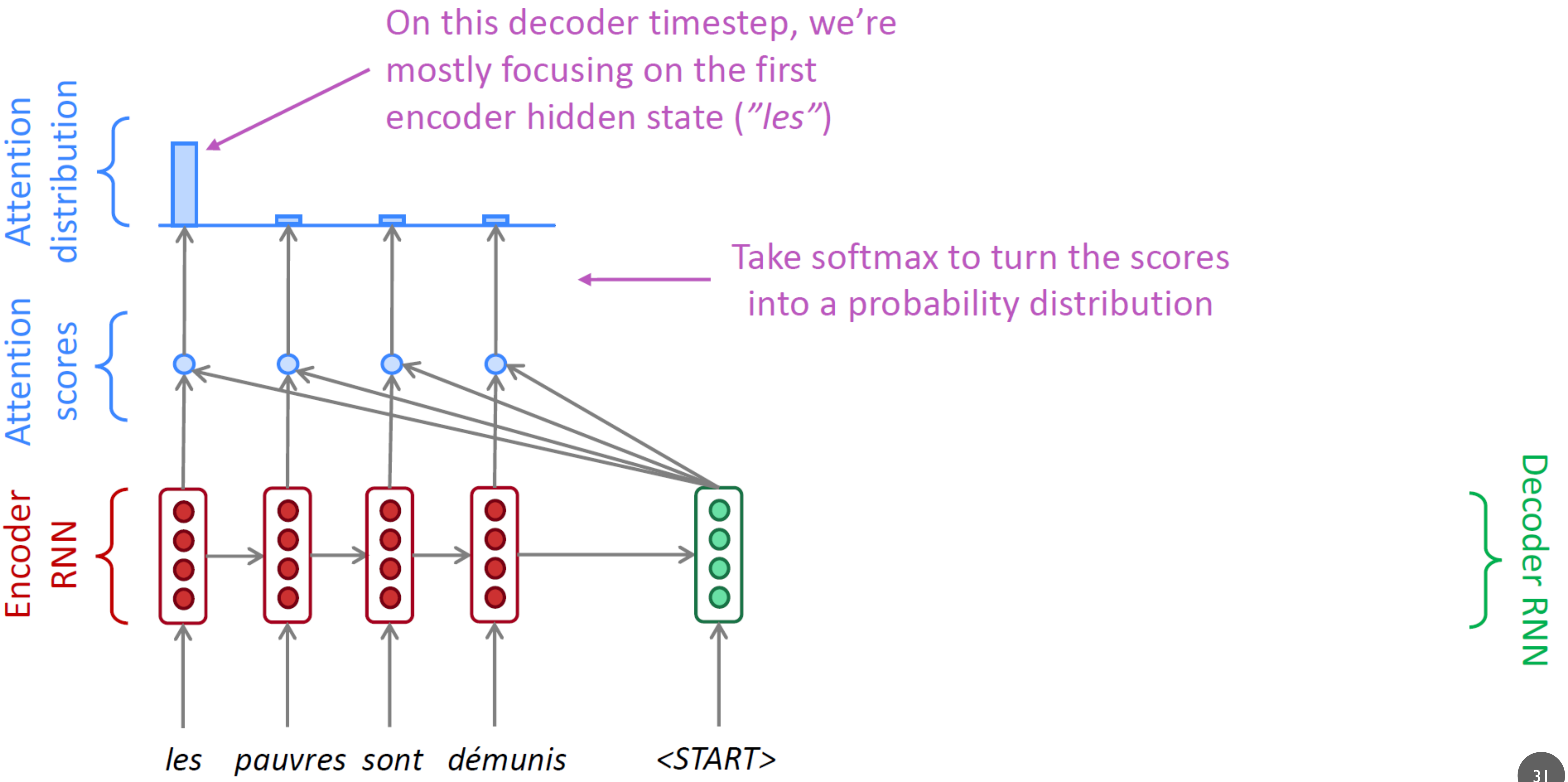


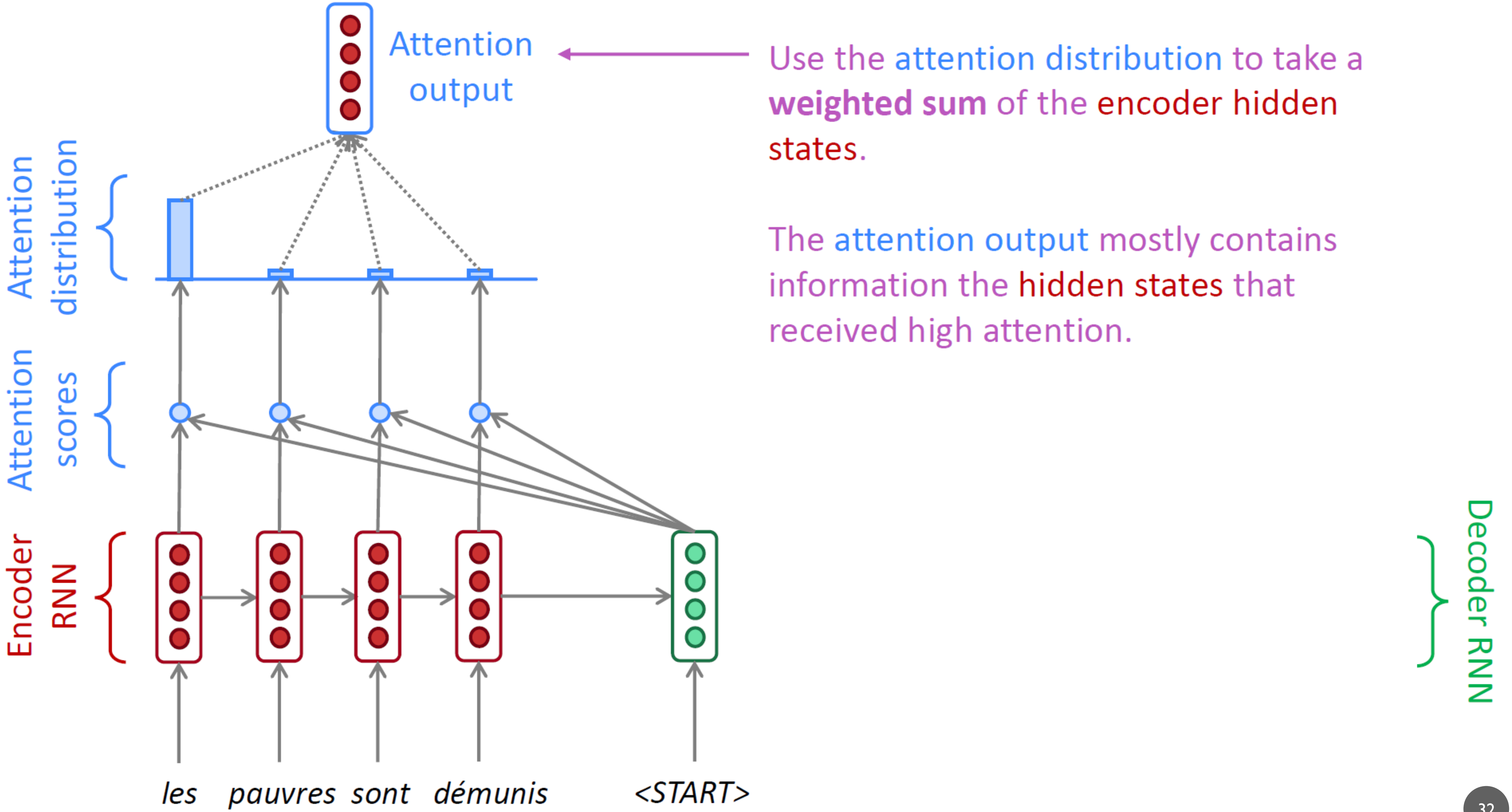

dot product
Attention scores
Encoder RNN
Decoder RNN
les pauvres sont démunis
<START>


dot product
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>


dot product
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>

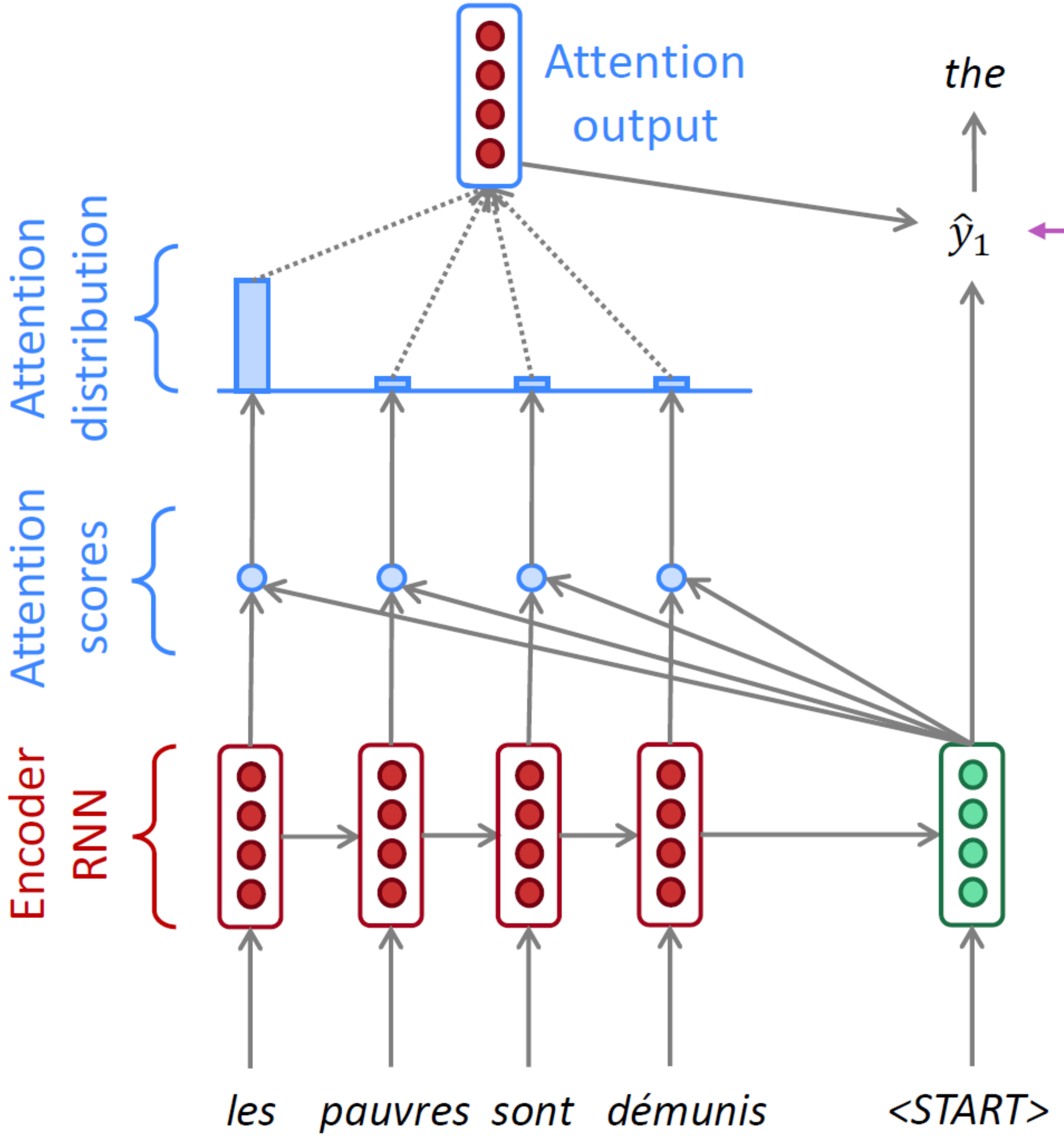

dot product
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>


On this decoder timestep, we're mostly focusing on the first encoder hidden state ("*les*")
Take softmax to turn the scores into a probability distribution
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>

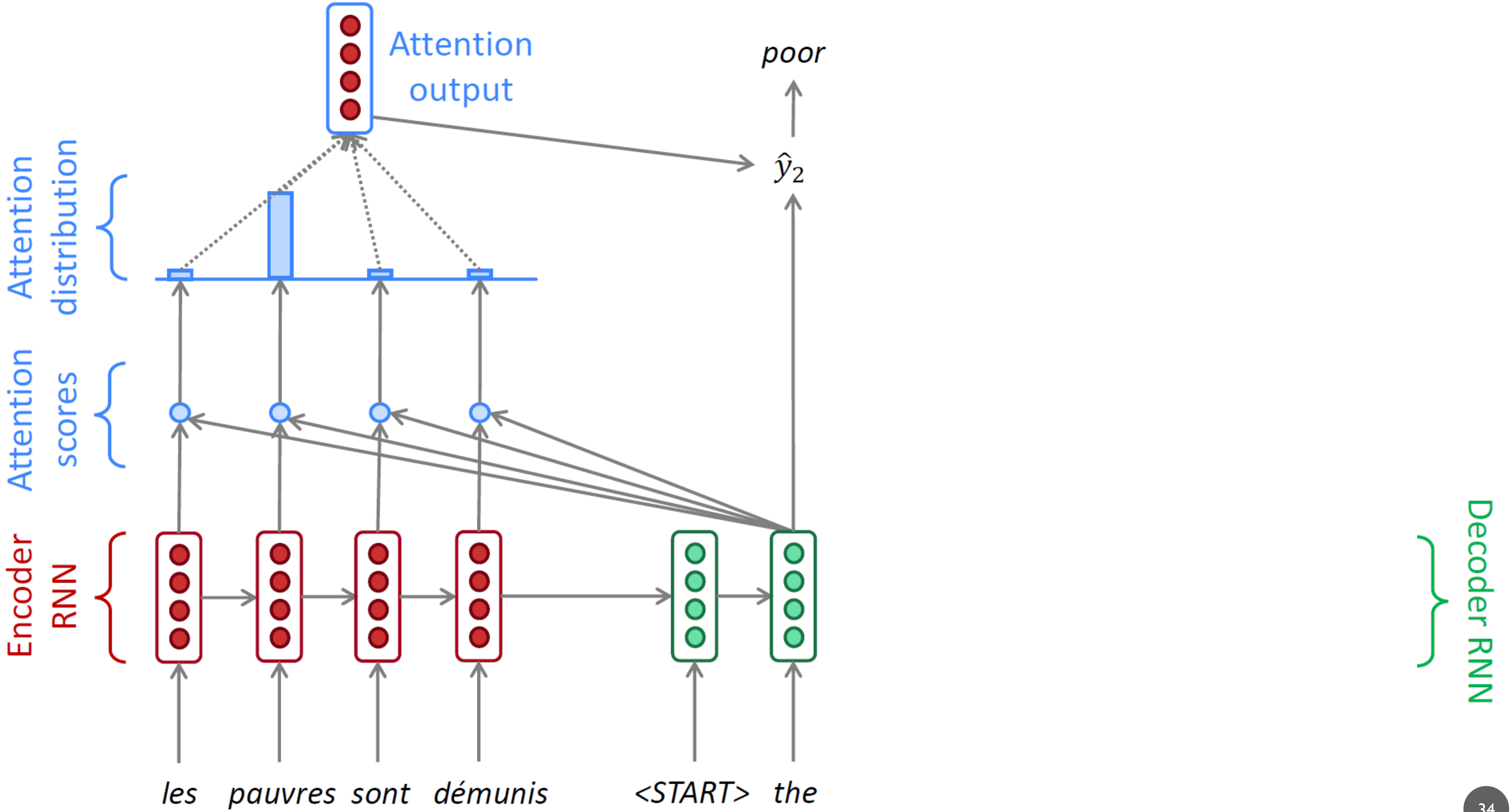


Use the attention distribution to take a **weighted sum** of the encoder hidden states.

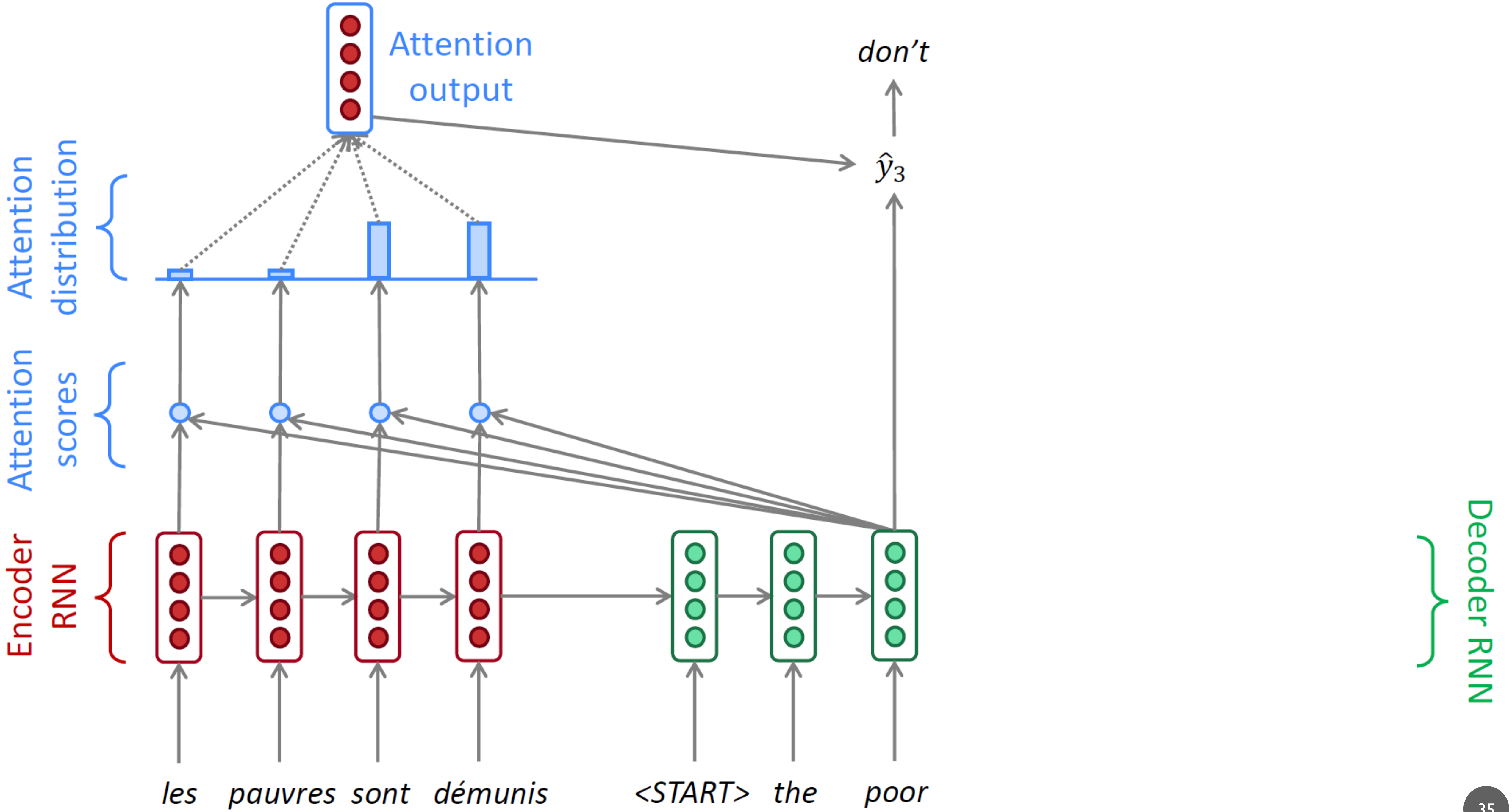
The attention output mostly contains information the hidden states that received high attention.

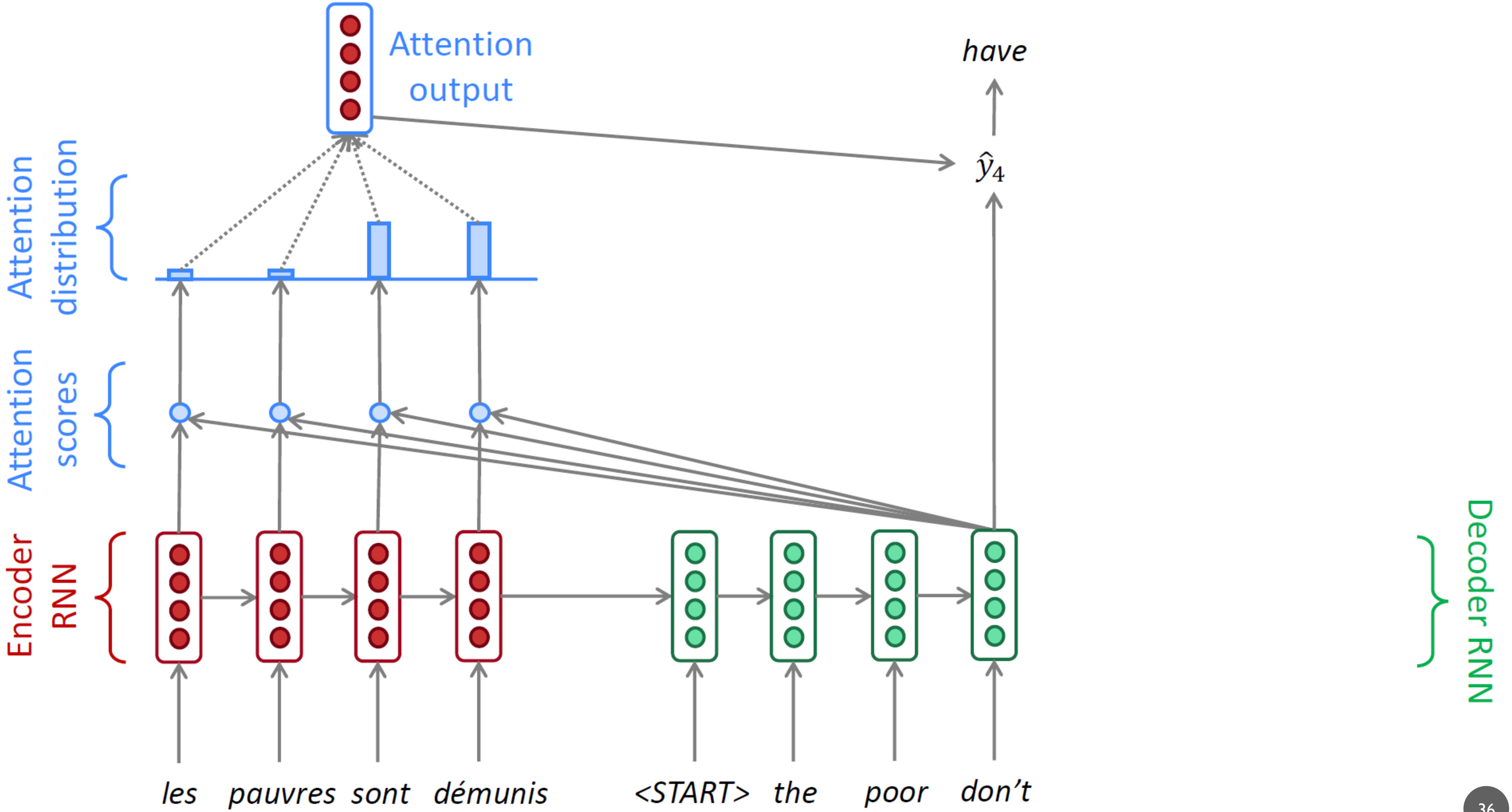


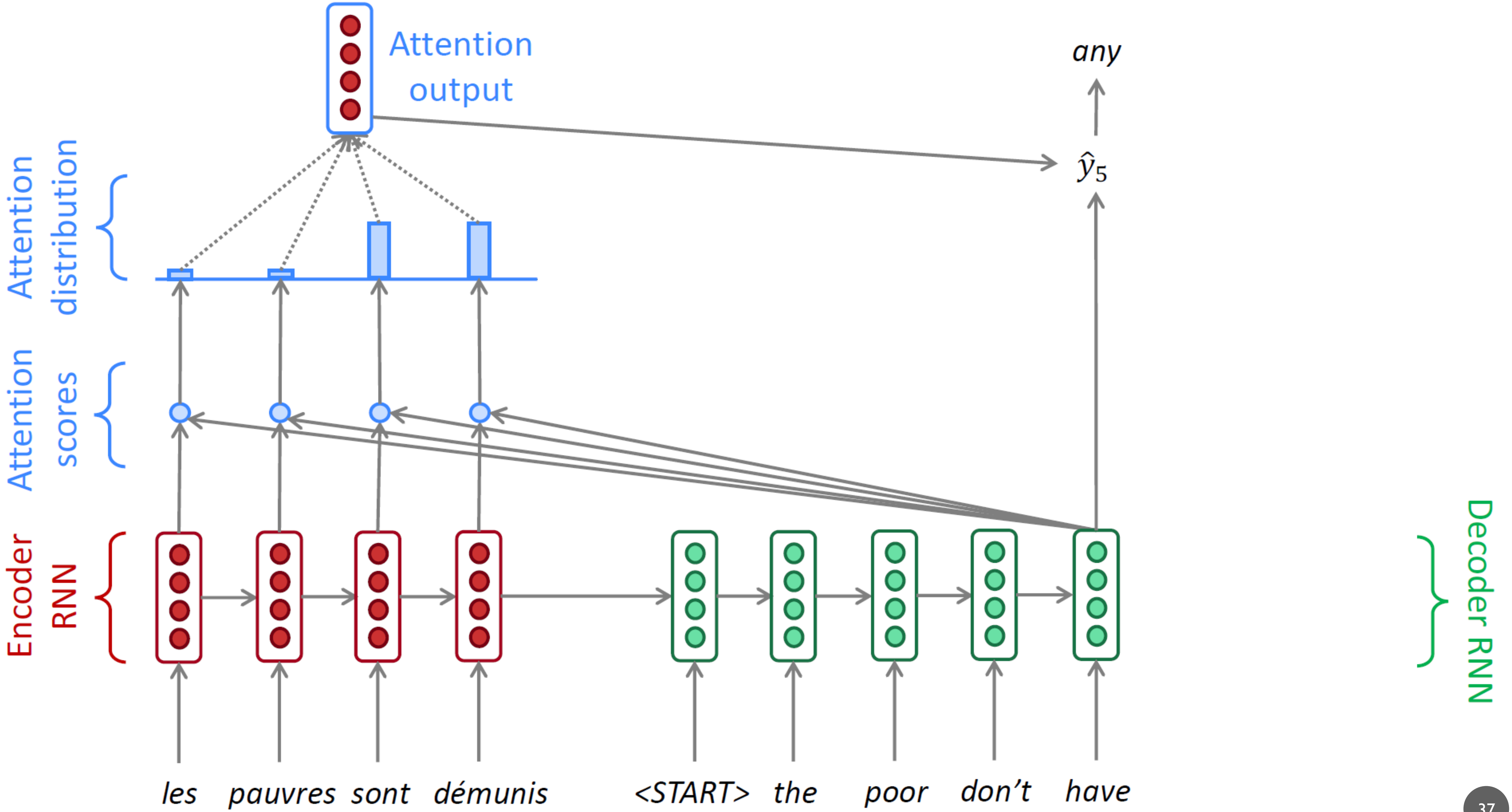
Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before



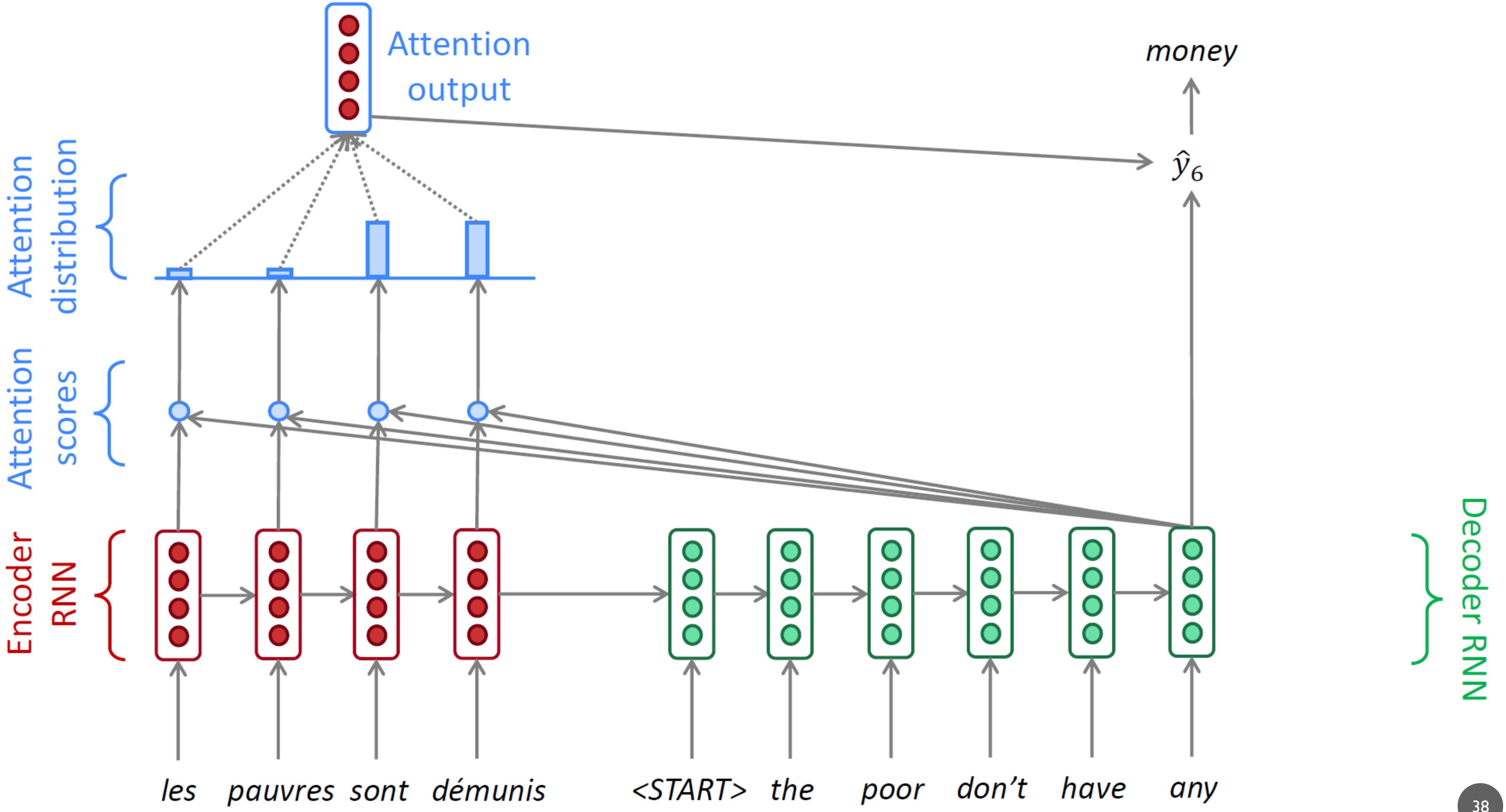
Attention output
poor
$\hat{y}_2$
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>
the


Attention output
don't
$\hat{y}_3$
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>
the
poor


Attention output
have
$\hat{y}_4$
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>
the
poor
don't



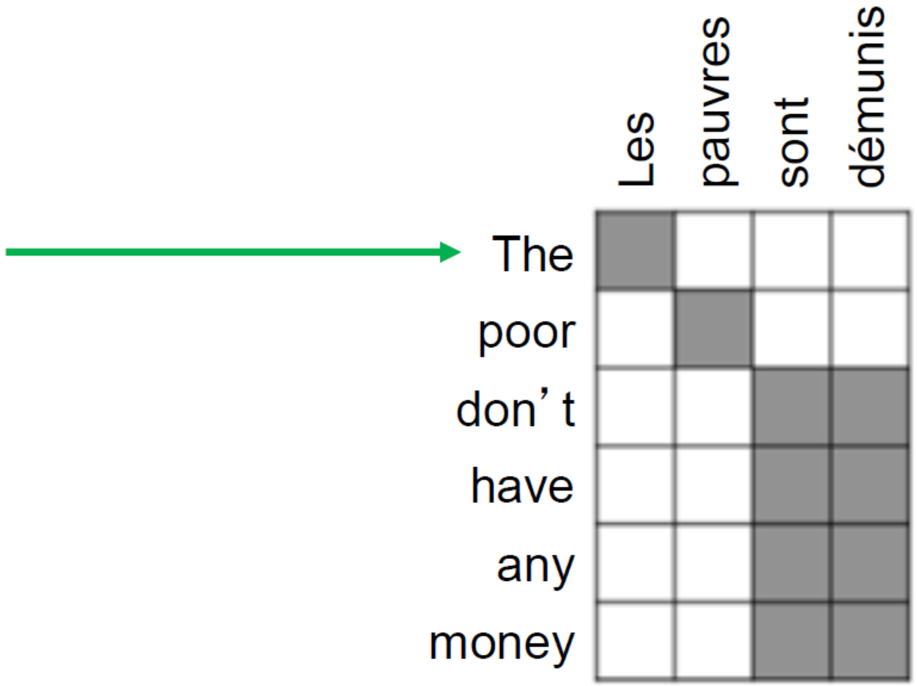
Attention output
any
$\hat{y}_5$
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>
the
poor
don't
have



Attention output
money
$\hat{y}_6$
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>
the
poor
don't
have
any

Attention is Great!

- Attention significantly improves NMT performance
  - It is useful to allow the decoder to focus on particular parts of the source
- Attention solves the bottleneck problem
  - Attention allows the decoder to look directly at source; bypass the bottleneck
- Attention helps with vanishing gradient problem
  - Provides a shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - We get alignment for free!
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself



Attention Examples in Machine Translation

- It properly learns grammatical orders of words
- It skips unnecessary words such as an article