midterm review

CS 585, Fall 2019

Introduction to Natural Language Processing

http://people.cs.umass.edu/~miyyer/cs585/

Mohit Iyyer

College of Information and Computer Sciences

University of Massachusetts Amherst

questions from last time…

- grading of HW2 / milestone1 in progress
- midterm!!!!

text classification

- input: some text **x** (e.g., sentence, document)
- output: a label **y** (from a finite label set)
- goal: learn a mapping function *f* from **x** to **y**

fyi: basically every NLP problem reduces to learning a mapping function with various definitions of **x** and **y**!

f can be hand-designed rules

- if “won \$10,000,000” in **x**, **y** = **spam**
- if “CS585 Fall 2019” in **x**, **y** = **not spam**

what are the drawbacks of this method?

f can be **learned** from data

- given **training data** (already-labeled **x,y** pairs) learn *f* by maximizing the likelihood of the training data

- this is known as **supervised learning**

naive Bayes

- represents input text as a bag of words
- assumption: each word is independent of all other words
- given labeled data, we can use naive Bayes to estimate probabilities for unlabeled data
- **goal:** infer probability distribution that generated the labeled data for each label

class conditional probabilities

Bayes rule (ex: x = sentence, y = label in {pos, neg})

$$\underbrace{p(y \mid x)}_{\text{posterior}} = \frac{\overbrace{p(y)}^{\text{prior}} \cdot \overbrace{P(x \mid y)}^{\text{likelihood}}}{p(x)}$$

our predicted label is the one with the highest posterior probability, i.e.,

$$\hat{y} = \arg\max_{y \in Y} p(y) \cdot P(x \mid y)$$

n-gram LMs

goal: assign probability to a piece of text

- why would we ever want to do this?

- translation:
    - P(i flew to the movies) <<<<< P(i went to the movies)

- speech recognition:
    - P(i saw a van) >>>>> P(eyes awe of an)

Probabilistic Language Modeling

- Goal: compute the probability of a sentence or sequence of words:

  $P(W) = P(w_1,w_2,w_3,w_4,w_5 \ldots w_n)$

- Related task: probability of an upcoming word:

  $P(w_5|w_1,w_2,w_3,w_4)$

- A model that computes either of these:

  $P(W)$ or $P(w_n|w_1,w_2 \ldots w_{n-1})$ is called a **language model** or **LM**

Markov Assumption

$$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i \mid w_{i-k} \ldots w_{i-1})$$

- In other words, we approximate each component in the product

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-k} \ldots w_{i-1})$$

Estimating bigram probabilities

- The Maximum Likelihood Estimate (MLE)
  - relative frequency based on the empirical counts on a training set

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

$$P(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

c — count

Perplexity

The best language model is one that best predicts an unseen test set

- Gives the highest P(sentence)

Perplexity is the inverse probability of the test set, normalized by the number of words:

$$PP(W) = P(w_1 w_2 ... w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}}$$

Chain rule:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

For bigrams:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$$

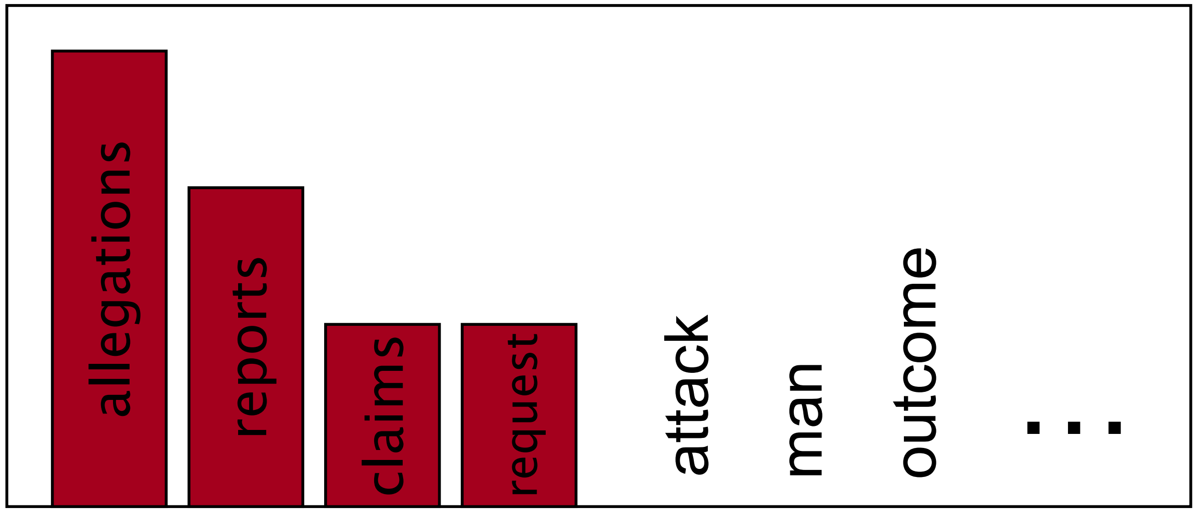
Minimizing perplexity is the same as maximizing probability

The intuition of smoothing (from Dan Klein)

- When we have sparse statistics:

  P(w | denied the)
  3 allegations
  2 reports
  1 claims
  1 request

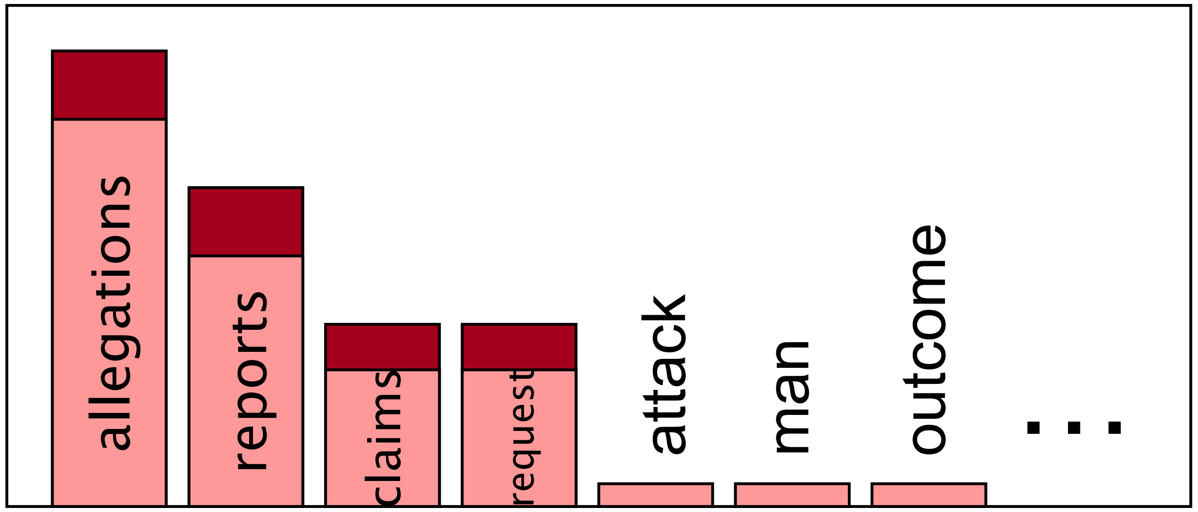
  7 total



- Steal probability mass to generalize better

  P(w | denied the)
  2.5 allegations
  1.5 reports
  0.5 claims
  0.5 request
  2 other

  7 total



Add-1 estimation is a blunt instrument

- So add-1 isn’t used for N-grams:
  - We’ll see better methods
- But add-1 is used to smooth other NLP models
  - For text classification
  - In domains where the number of zeros isn’t so huge.

Absolute discounting: just subtract a little from each count

- Suppose we wanted to subtract a little from a count of 4 to save probability mass for the zeros
- How much to subtract ?
- Church and Gale (1991)'s clever idea
- Divide up 22 million words of AP Newswire
  - Training and held-out set
  - for each bigram in the training set
  - see the actual count in the held-out set!

Bigram count in training	Bigram count in heldout set
0	.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

log-linear LMs (and more generally, logistic regression)

The General Problem

- We have some **input domain** $\mathcal{X}$
- Have a finite **label set** $\mathcal{Y}$
- Aim is to provide a **conditional probability** $P(y \mid x)$ for any x, y where $x \in \mathcal{X}$, $y \in \mathcal{Y}$

Language Modeling

- x is a "history" $w_1, w_2, \ldots w_{i-1}$, e.g.,

  Third, the notion "grammatical in English" cannot be identified in any way with the notion "high order of statistical approximation to English". It is fair to assume that neither sentence (1) nor (2) (nor indeed any part of these sentences) has ever occurred in an English discourse. Hence, in any statistical

- y is an "outcome" w_i

Feature Vector Representations

- Aim is to provide a conditional probability $P(y \mid x)$ for "decision" y given "history" x

A feature is some function $\phi(x)$; in LMs $\phi(\mathit{context})$.

Features are often binary *indicators*; i.e. $\phi(x) \in \{0,1\}$

If you have *m* features, you can form a **feature vector** $\mathbf{x} \in \mathbb{R}^{\mathbf{m}}$

what could be some useful indicator features for language modeling?

given features **x**, how do we predict the next word *y*?

$$s = Wx + b$$

score vector $s \in \mathbb{R}^{|V|}$

features $x \in \mathbb{R}^{m}$

weight matrix $W \in \mathbb{R}^{|V| \times m}$

each row of W contains weights for a (word *y*, **x**) pair

how do we obtain probabilities?

$$s = Wx + b$$

score vector $s \in \mathbb{R}^{|V|}$

features $x \in \mathbb{R}^{m}$

weight matrix $W \in \mathbb{R}^{|V| \times m}$

$$p_i = \frac{e^{s_i}}{\sum_j e^{s_j}}; p = \text{softmax}(s)$$

"Log-linear" ?

$$\log p(y \mid x, W) \propto W_y x$$

why is this true?

$$p(y \mid x, W) = \frac{e^{W_y x}}{\sum_{y' \in V} e^{W_{y'} x}}$$

$$\log p(y \mid x, W) = W_y x - \log \sum_{y' \in V} e^{W_{y'} x}$$

linear in weights and features…

… except for this!

known as log-sum-exp,

very important for these models

what do we have left?

- how do we find the optimal values of **W** and **b** for our language modeling problem?
- gradient descent! this involves computing:

1. a *loss function*, which tells us how good the current values of **W** and **b** are on our training data
2. the partial derivatives $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$

first, an aside: what is the bias **b**?

- Let's say we have a feature that is always set to 1 regardless of what the input text is.
- This is clearly not an informative feature. However, let's say it was the only one I had…

first, how many weights do I need to learn for this feature?

okay… what is the best set of weights for it?

Training with softmax and cross-entropy error

- For each training example {x,y}, our objective is to maximize the probability of the correct class y

- Hence, we minimize the negative log probability of that class:

$$L = -\log p(y \mid x, W) = -\log\left(\frac{e^{W_y x}}{\sum_{y' \in V} e^{W_{y'} x}}\right)$$

Background: Why "Cross entropy" error

- Assuming a ground truth (or gold or target) probability distribution that is 1 at the right class and 0 everywhere else: p = [0,…,0,1,0,…0] and our computed probability is q, then the cross entropy is:

$$H(p,q) = -\sum_{w \in V} p(w) \log q(w)$$

- **Because of one-hot p, the only term left is the negative log probability of the true class**

let's say I also have the derivatives

$$\frac{\partial L}{\partial W} \qquad\qquad \frac{\partial L}{\partial b}$$

- the partial derivatives tell us how the loss changes given a change in the corresponding parameter
- we can thus take steps in the *negative* direction of the gradient to *minimize* the loss function

word embeddings

why do neural networks work better?

- multiple layer and *nonlinearities* allow feature combinations that a linear model can't get
  - e.g., XOR function
- the learned representations of words and contexts are tuned to the prediction problem
  - unlike one-hot vectors

why use vectors to encode meaning?

- computing the similarity between two words (or phrases, or documents) is *extremely* useful for many NLP tasks

- Q: how **tall** is Mount Everest?

  A: The official **height** of Mount Everest is 29029 ft

one-hot vectors

- we've already seen these before in bag-of-words models (e.g., naive Bayes)!
- represent each word as a vector of zeros with a single 1 identifying the index of the word

vocabulary
i
hate
love
the
movie
film

movie = <0, 0, 0, 0, 1, 0>

film = <0, 0, 0, 0, 0, 1>

what are the issues of representing a word this way?

all words are equally (dis)similar!

movie = <0, 0, 0, 0, 1, 0>

film = <0, 0, 0, 0, 0, 1>

dot product is zero!

these vectors are **orthogonal**

how can we compute a vector representation such that the dot product correlates with word similarity?

Word2vec

- Instead of **counting** how often each word *w* occurs near "*apricot*"
- Train a classifier on a binary **prediction** task:
  - Is *w* likely to show up near "*apricot*"?

- We don't actually care about this task
  - But we'll take the learned classifier weights as the word embeddings

Setup

Let's represent words as vectors of some length (say 300), randomly initialized.

So we start with 300 * V random parameters

Over the entire training set, we'd like to adjust those word vectors such that we

- Maximize the similarity of the target word, context word pairs (t,c) drawn from the positive data
- Minimize the similarity of the (t,c) pairs drawn from the negative data.

Skip-Gram Training Data

Training sentence:

... lemon, a tablespoon of **apricot** jam a pinch ...

c1 c2 target c3 c4

Asssume context words are those in +/- 2 word window

Skip-Gram Goal

Given a tuple (t,c) = target, context

- (*apricot*, *jam*)
- (*apricot*, *aardvark*)

Return probability that c is a real context word:

$P(+|t,c)$

$$P(-|t,c) = 1 - P(+|t,c)$$

How to compute $p(+|t,c)$?

Intuition:

- Words are likely to appear near similar words
- Model similarity with dot-product!
- Similarity(t,c) $\propto t \cdot c$

t and **c** here are vectors for **t**arget and **c**ontext!

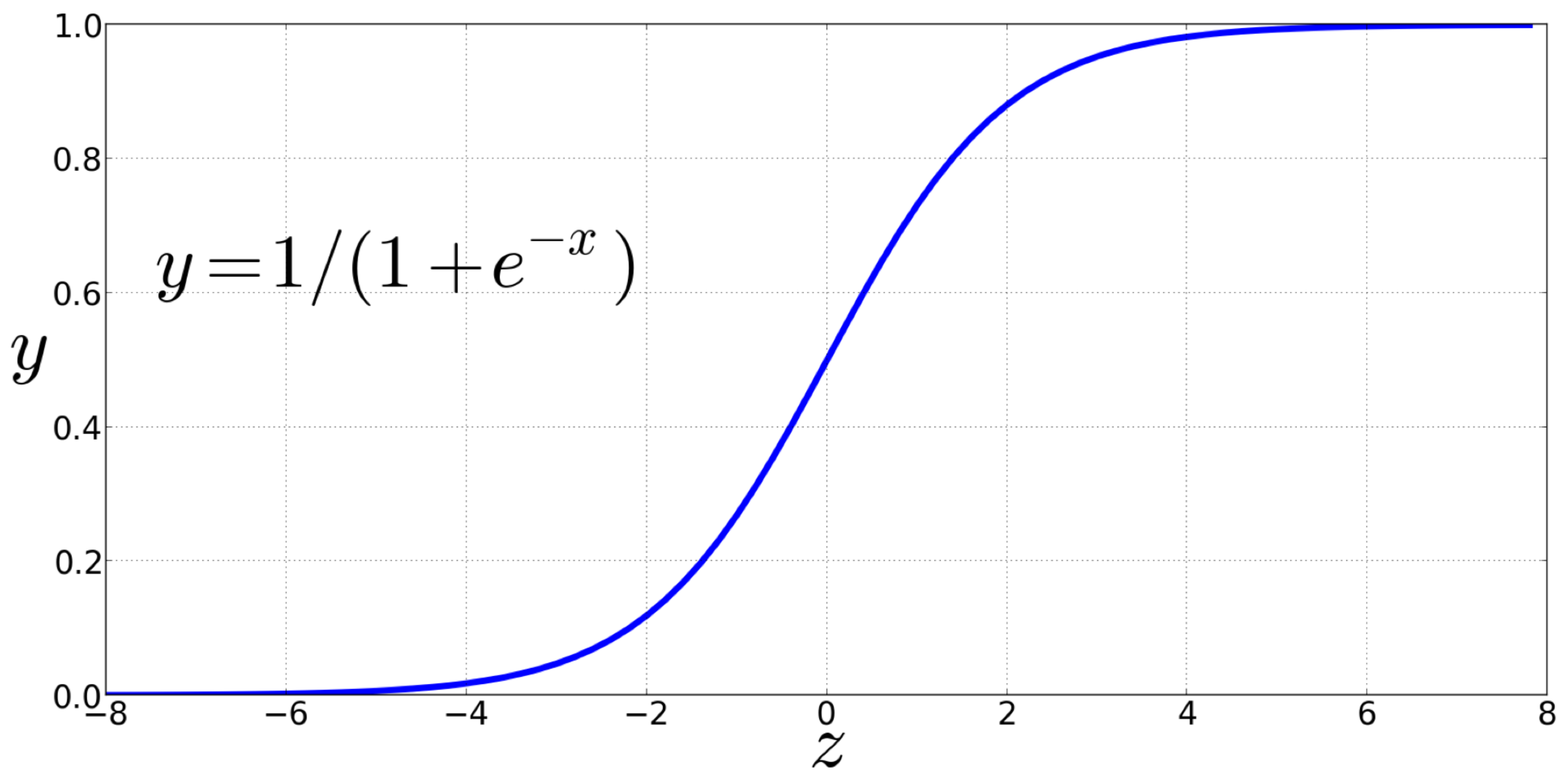
Problem:

- *Dot product is not a probability!*
  - *(Neither is cosine)*

Turning dot product into a probability

The sigmoid lies between 0 and 1:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Turning dot product into a probability

think back to last class… what are our features and weights here???

both target and context vectors are *learned*, so we have no explicit featurization!

$$P(+|t,c) = \frac{1}{1+e^{-t\cdot c}}$$

$$P(-|t,c) = 1 - P(+|t,c)$$
$$= \frac{e^{-t\cdot c}}{1+e^{-t\cdot c}}$$

Learning the classifier

Iterative process.

We'll start with 0 or random weights

Then adjust the word weights to

- make the positive pairs more likely
- and the negative pairs less likely

over the entire training set

guess what algorithm we'll use to make this happen?

neural LMs

A fixed-window neural Language Model

output distribution

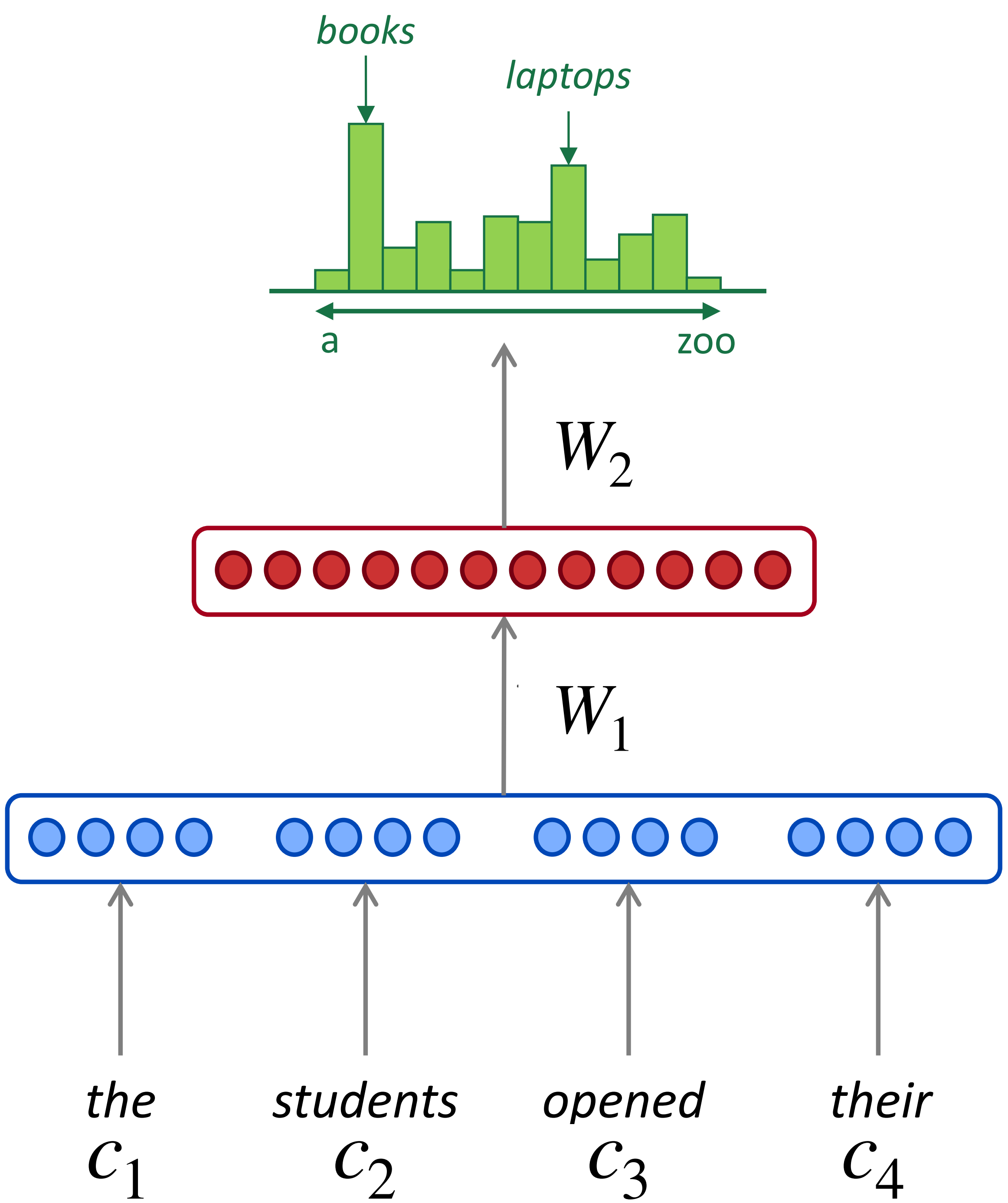
$$\hat{y} = \mathrm{softmax}(W_2 h + b_2)$$

hidden layer

$$h = f(W_1 c + b_1)$$

concatenated word embeddings

$$c = [c_1; c_2; c_3; c_4]$$



A RNN Language Model

output distribution

$$\hat{y} = \text{softmax}(W_2 h^{(t)} + b_2)$$

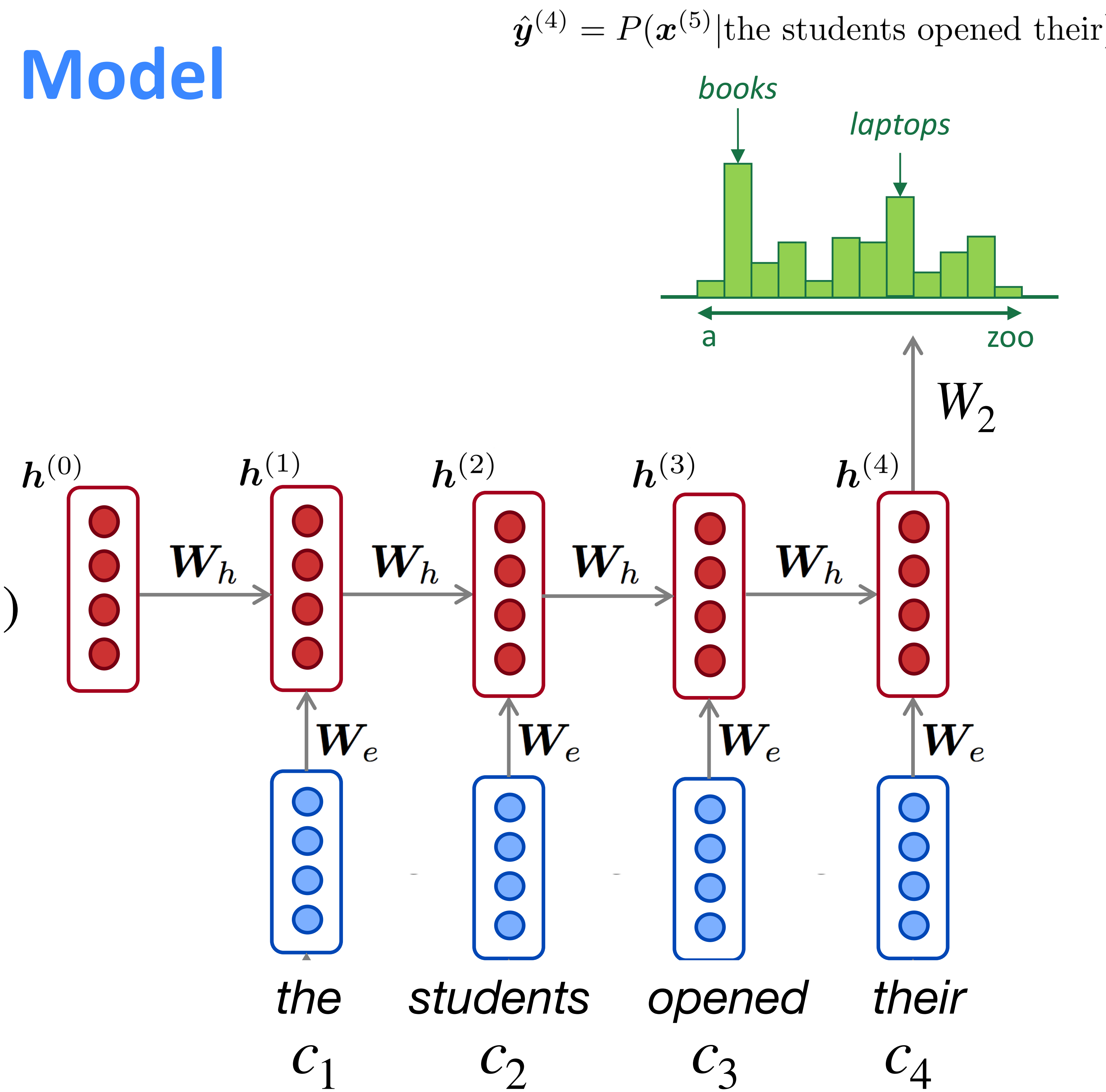
hidden states

$$h^{(t)} = f(W_h h^{(t-1)} + W_e c_t + b_1)$$

$h^{(0)}$ is initial hidden state!

word embeddings

$$c_1, c_2, c_3, c_4$$



$$\hat{\boldsymbol{y}}^{(4)} = P(\boldsymbol{x}^{(5)}|\text{the students opened their})$$

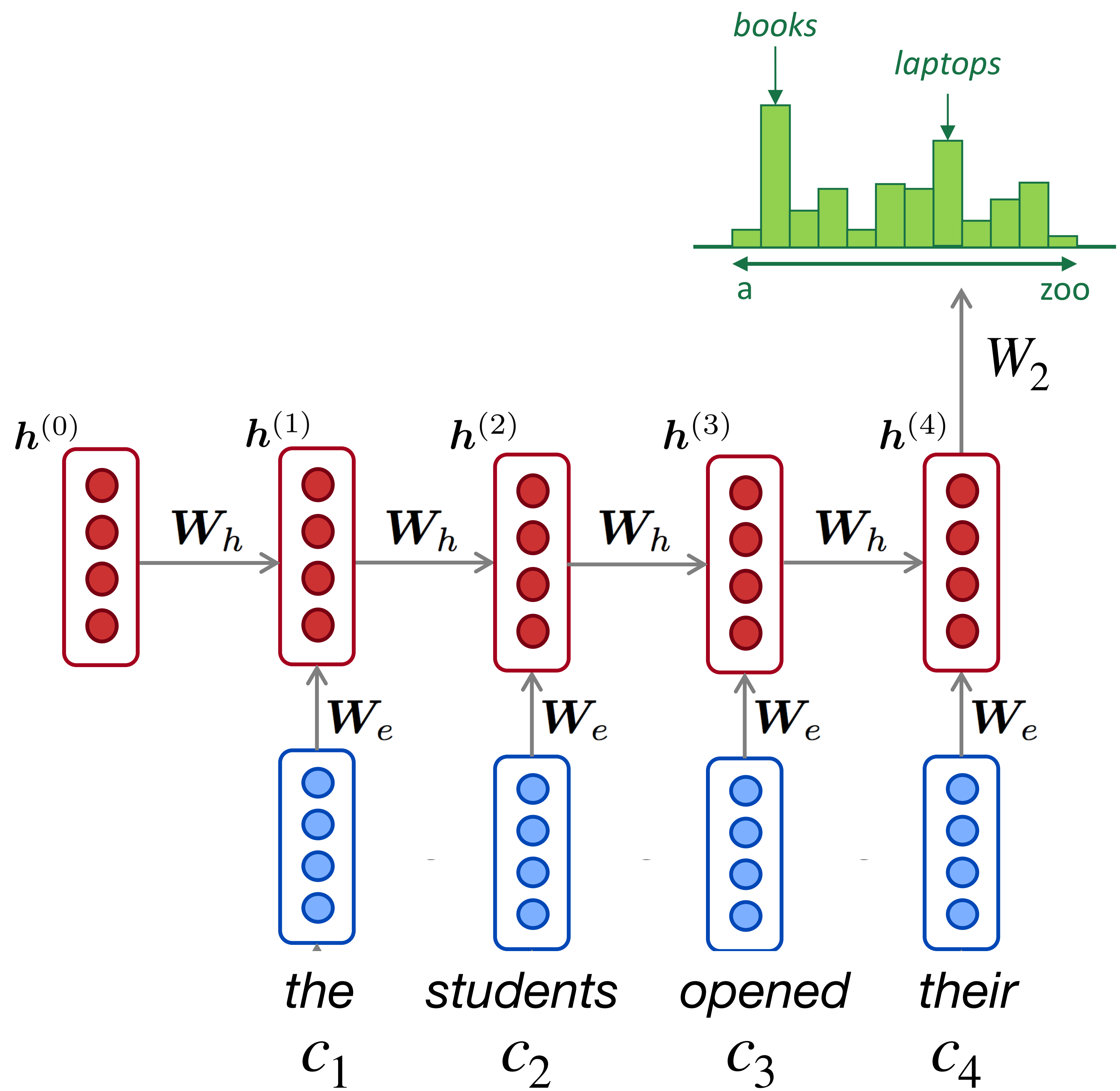
why is this good?

RNN **Advantages**:

- Can process any length input
- Model size doesn't increase for longer input
- Computation for step *t* can (in theory) use information from many steps back
- Weights are shared across timesteps → representations are shared

RNN **Disadvantages**:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back



Training a RNN Language Model

- Get a big corpus of text which is a sequence of words $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\boldsymbol{y}}^{(t)}$ for *every step t.*
  - i.e. predict probability dist of *every word*, given words so far

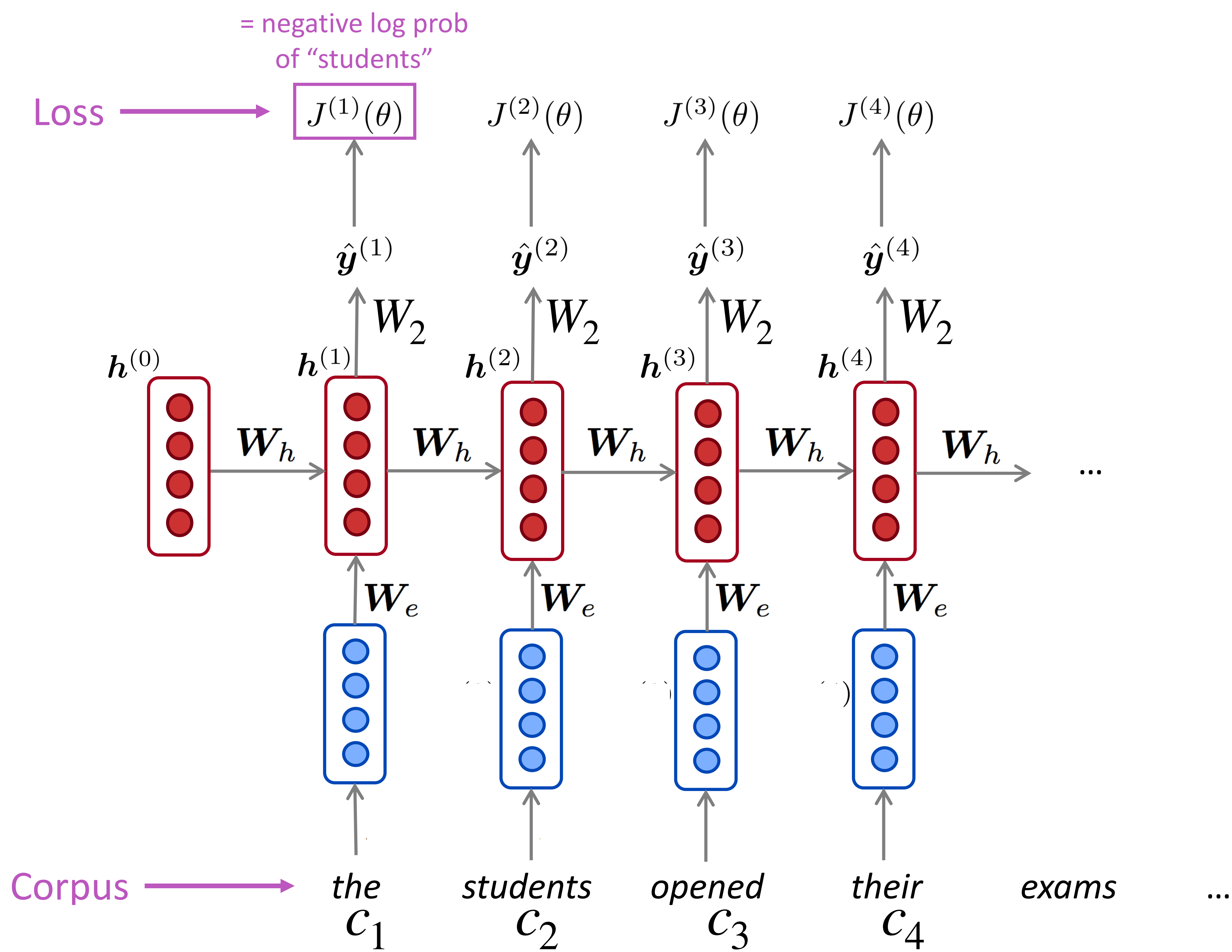
- Loss function on step *t* is usual cross-entropy between our predicted probability distribution $\hat{\boldsymbol{y}}^{(t)}$, and the true next word $\boldsymbol{y}^{(t)} = \boldsymbol{x}^{(t+1)}$:

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} \log \hat{y}_j^{(t)}$$

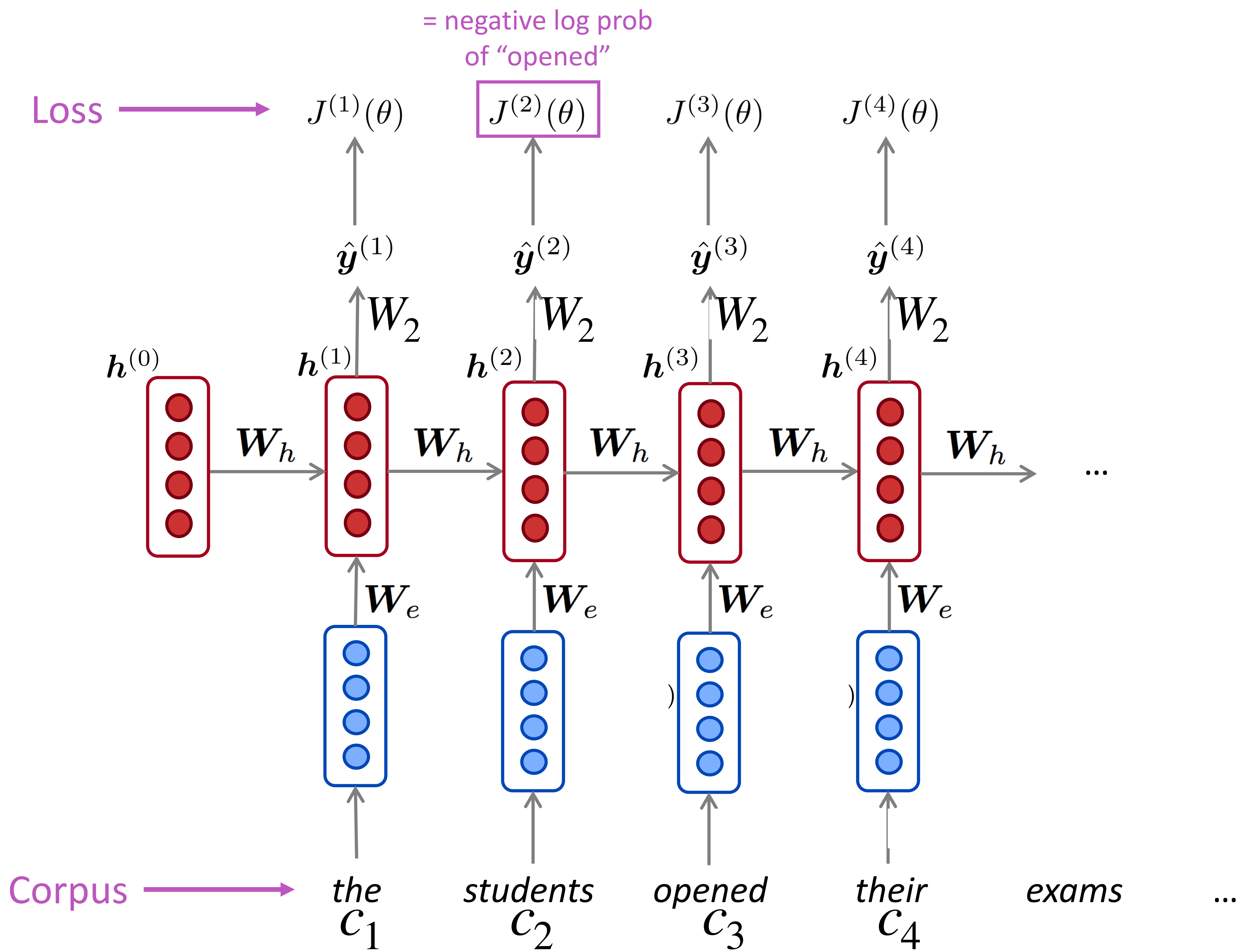
- Average this to get overall loss for entire training set:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)$$

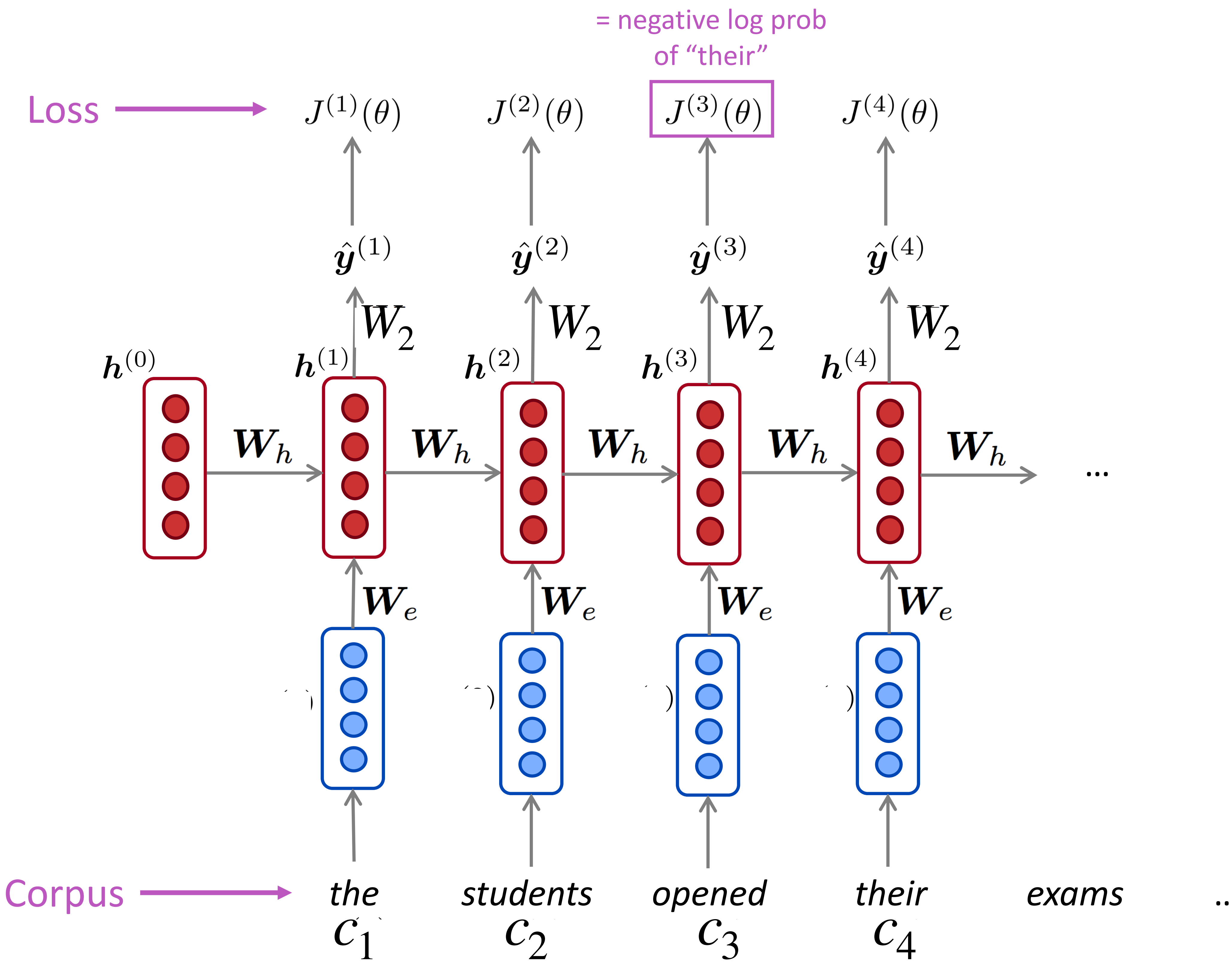
Training a RNN Language Model



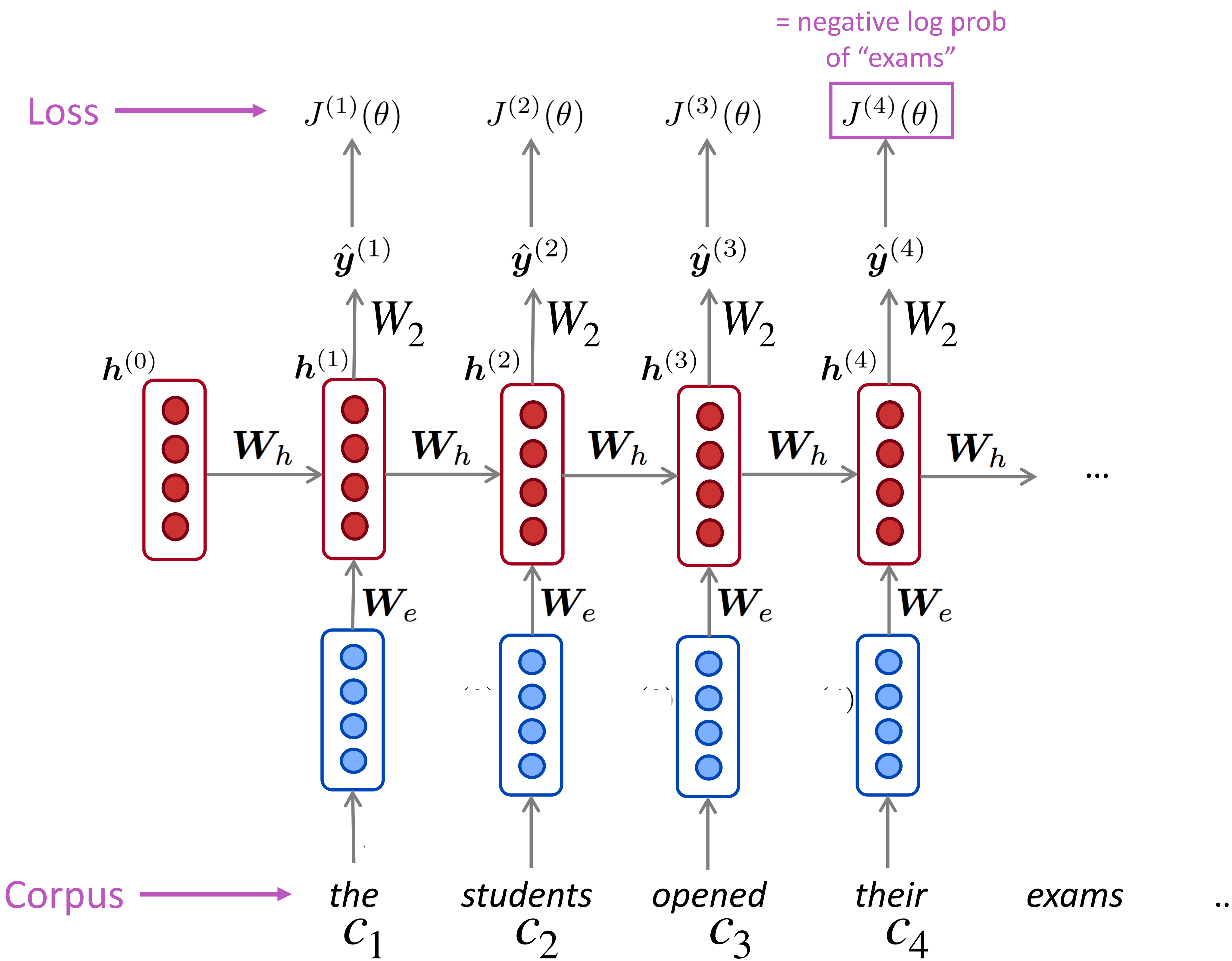
Training a RNN Language Model



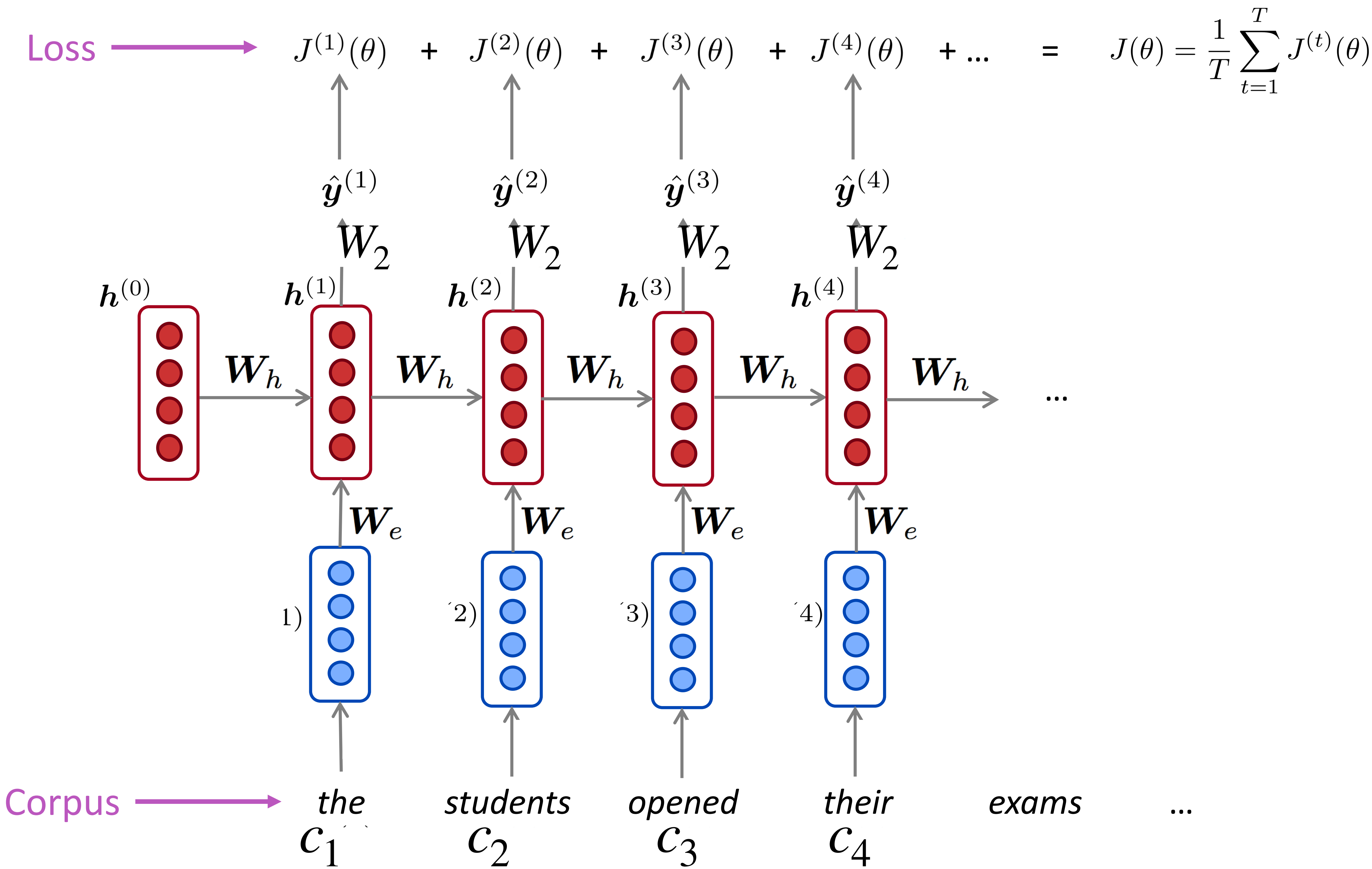
Training a RNN Language Model



Training a RNN Language Model



Training a RNN Language Model



Training a RNN Language Model

- However: Computing loss and gradients across entire corpus is too expensive!
- Recall: Stochastic Gradient Descent allows us to compute loss and gradients for small chunk of data, and update.

- → In practice, consider $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$ as a sentence

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$$

- Compute loss $J(\theta)$ for a sentence (actually usually a batch of sentences), compute gradients and update weights. Repeat.

okay… enough with the unconditional LMs. let's switch to conditional LMs!

we'll start with *machine translation*

today: neural MT

- we'll use French (*f*) to English (*e*) as a running example
- **goal**: given French sentence *f* with tokens f_1, f_2, … f_n produce English translation *e* with tokens e_1, e_2, … e_m

is *n* always equal to *m*?

- **real goal**: compute $\arg\max_e p(e \mid f)$

today: neural MT

- let's use an NN to directly model $p(e \mid f)$

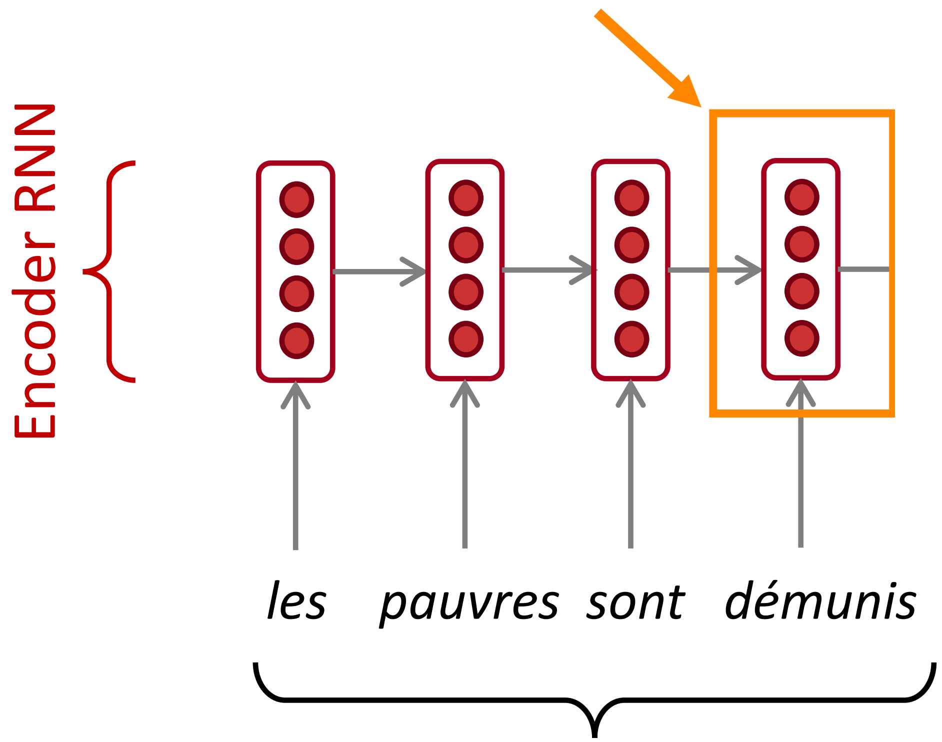
$$
\begin{aligned}
p(e \mid f) &= p(e_1, e_2, \ldots, e_m \mid f) \\
&= p(e_1 \mid f) \cdot p(e_2 \mid e_1, f) \cdot p(e_3 \mid e_2, e_1, f) \cdot \ldots \\
&= \prod_{i=1}^{m} p(e_i \mid e_1, \ldots, e_{i-1}, f)
\end{aligned}
$$

how does this formulation relate to the language models we discussed previously?

Neural Machine Translation (NMT)

The sequence-to-sequence model

Encoding of the source sentence.
Provides initial hidden state
for Decoder RNN.

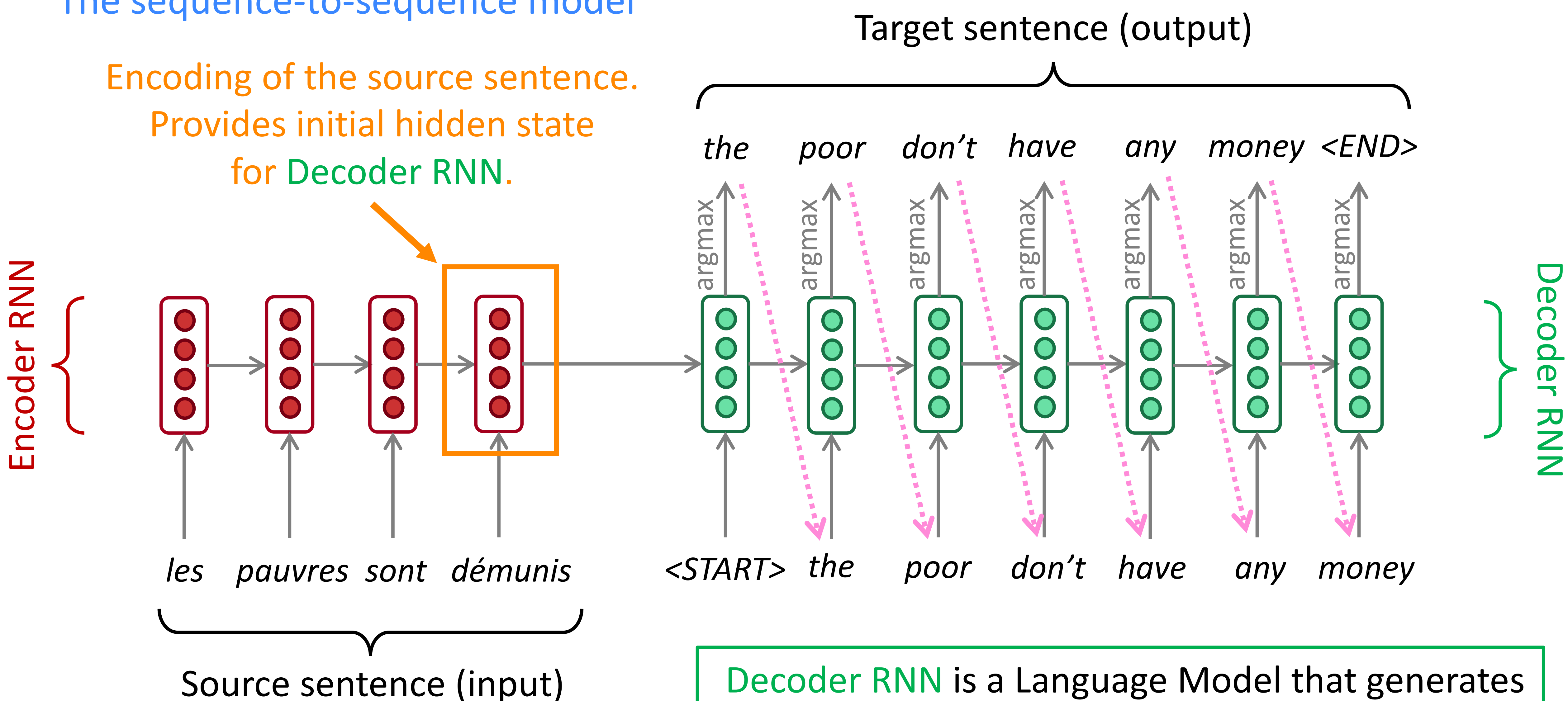


Source sentence (input)

Encoder RNN produces an encoding of the source sentence.

Neural Machine Translation (NMT)

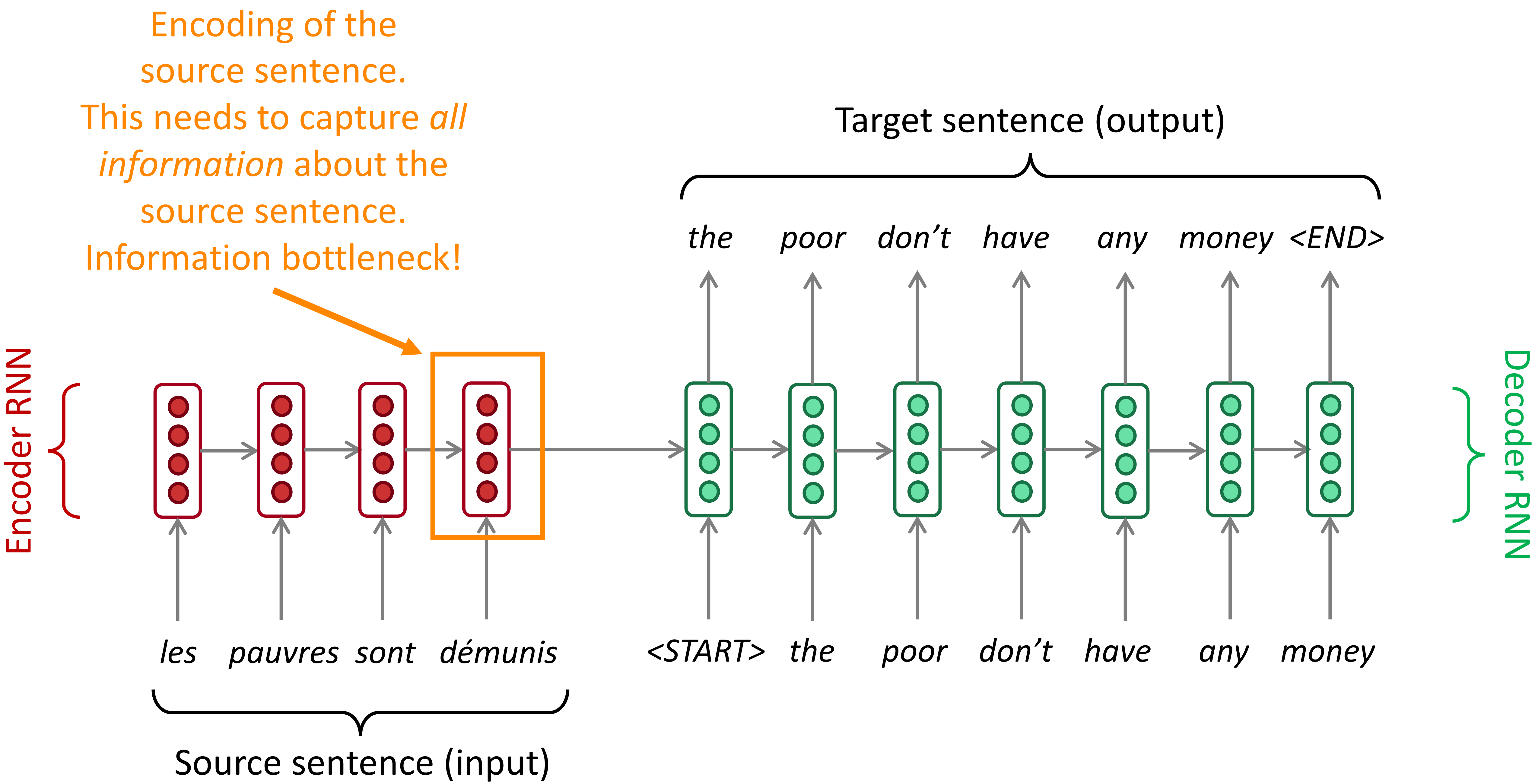
The sequence-to-sequence model



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

Encoder RNN produces an encoding of the source sentence.

Sequence-to-sequence: the bottleneck problem



The solution: **attention**

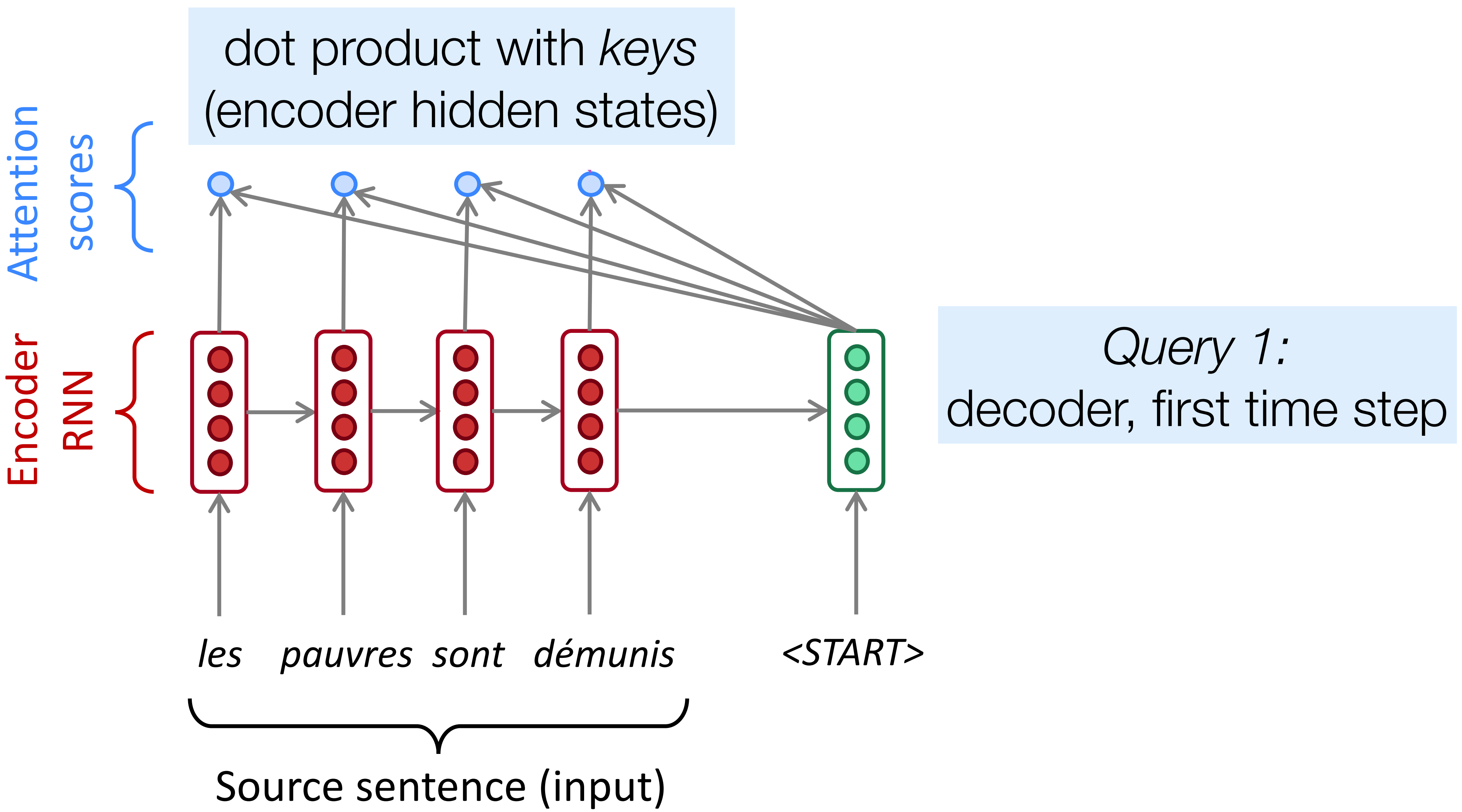
- **Attention mechanisms** (Bahdanau et al., 2015) allow the decoder to focus on a particular part of the source sequence at each time step
  - Conceptually similar to *word alignments*

How does it work?

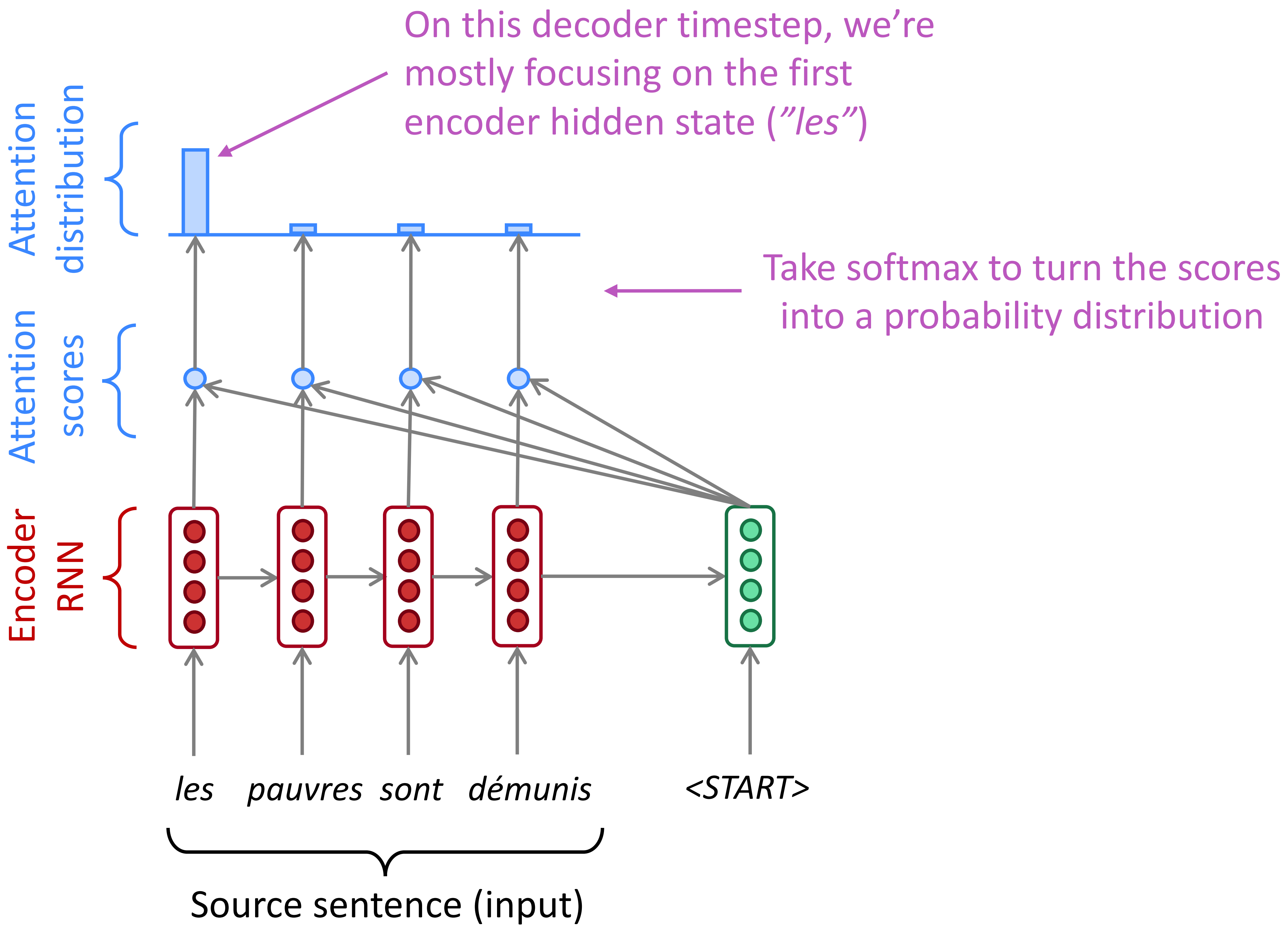
- in general, we have a single *query* vector and multiple *key* vectors. We want to score each query-key pair

in machine translation with RNNs, what are the queries and keys?

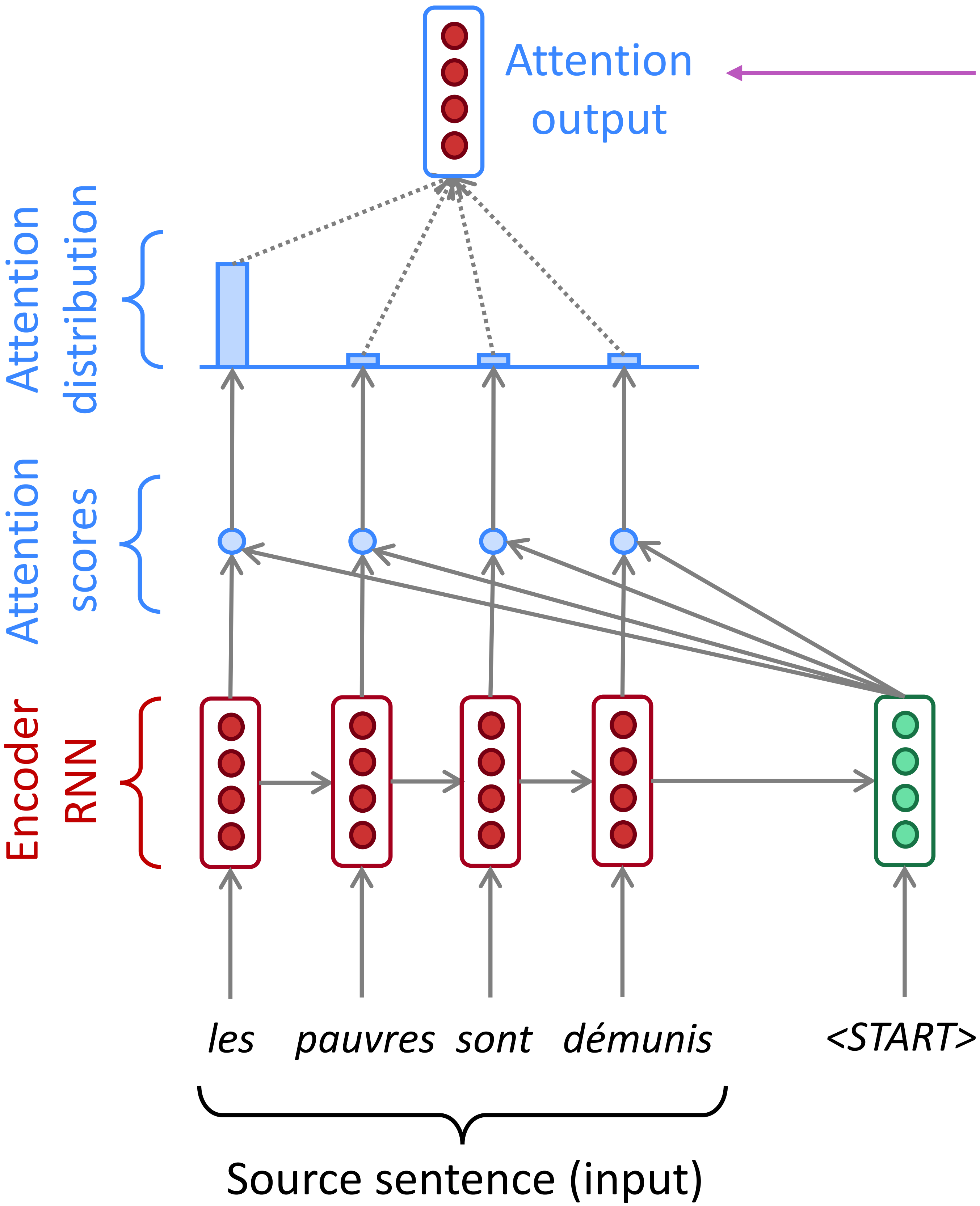
Sequence-to-sequence with attention



Sequence-to-sequence with attention



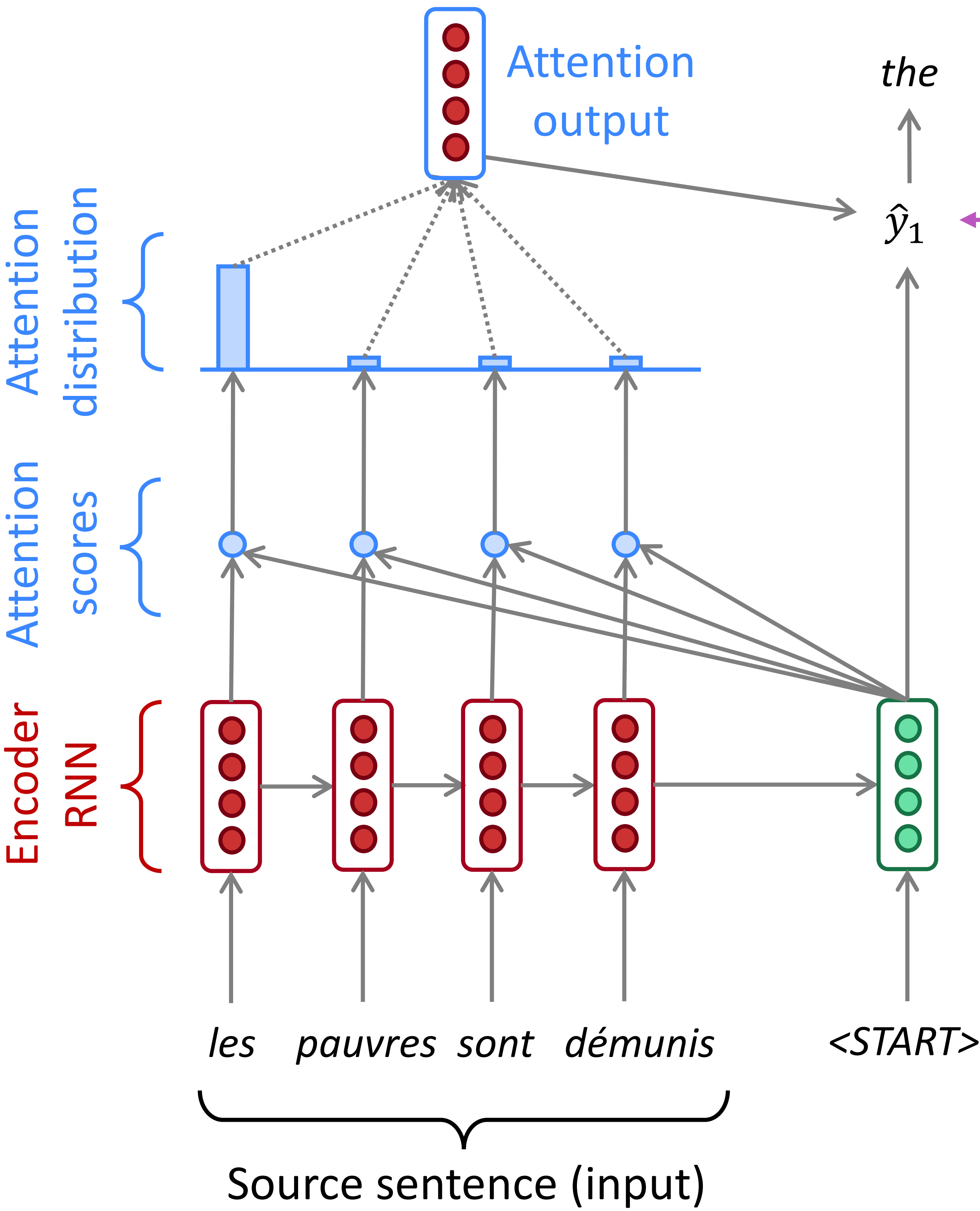
Sequence-to-sequence with attention



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

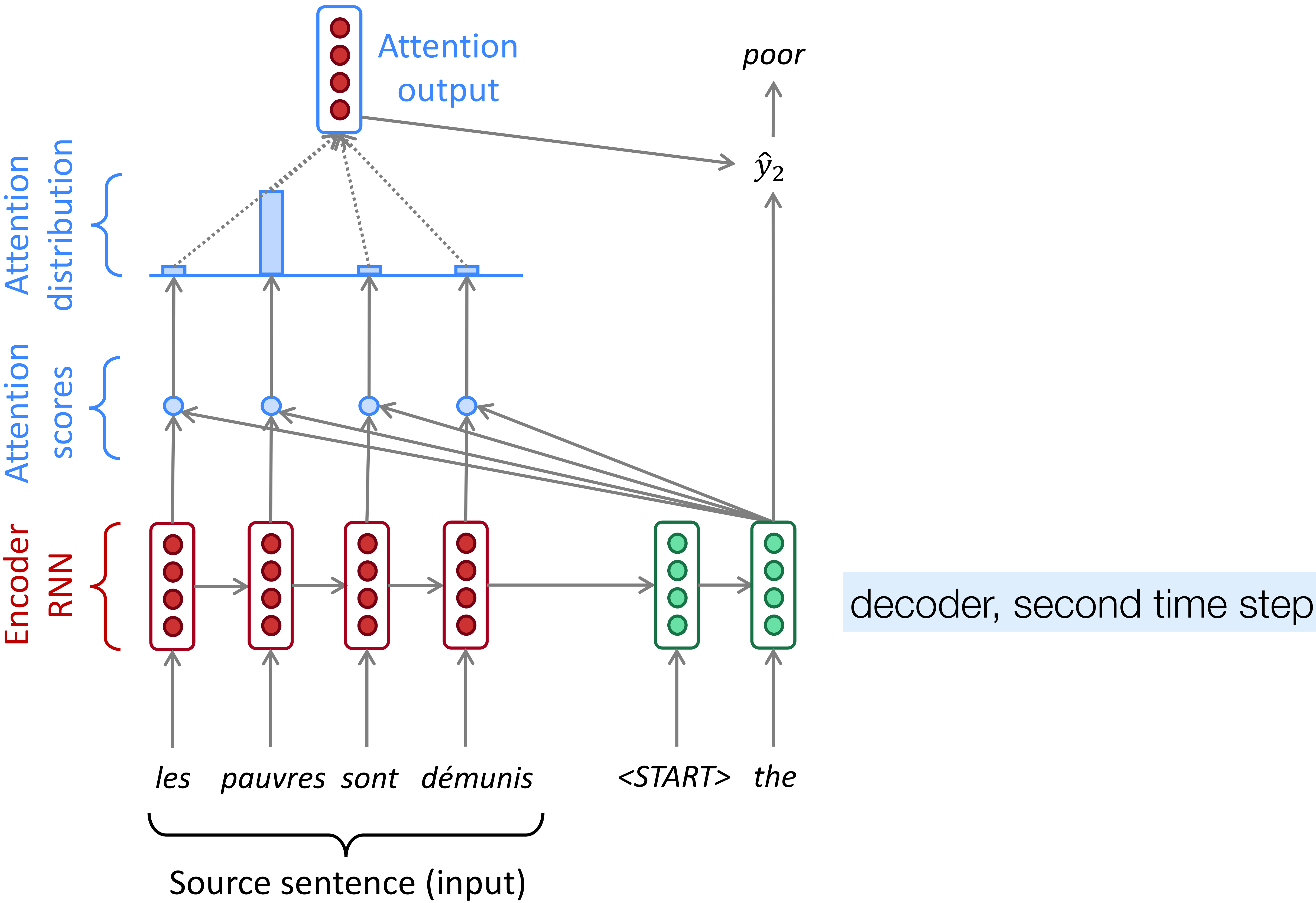
The attention output mostly contains information the hidden states that received high attention.

Sequence-to-sequence with attention



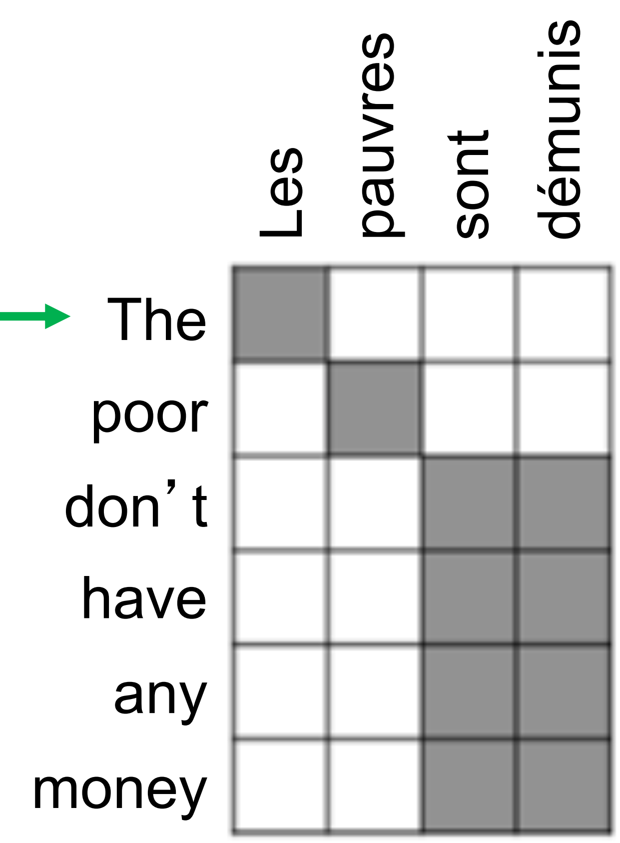
Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before

Sequence-to-sequence with attention



Attention is great

- Attention significantly improves NMT performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with vanishing gradient problem
  - Provides 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



decoding

- given that we trained a seq2seq model, how do we find the most probable English sentence?
- more concretely, how do we find

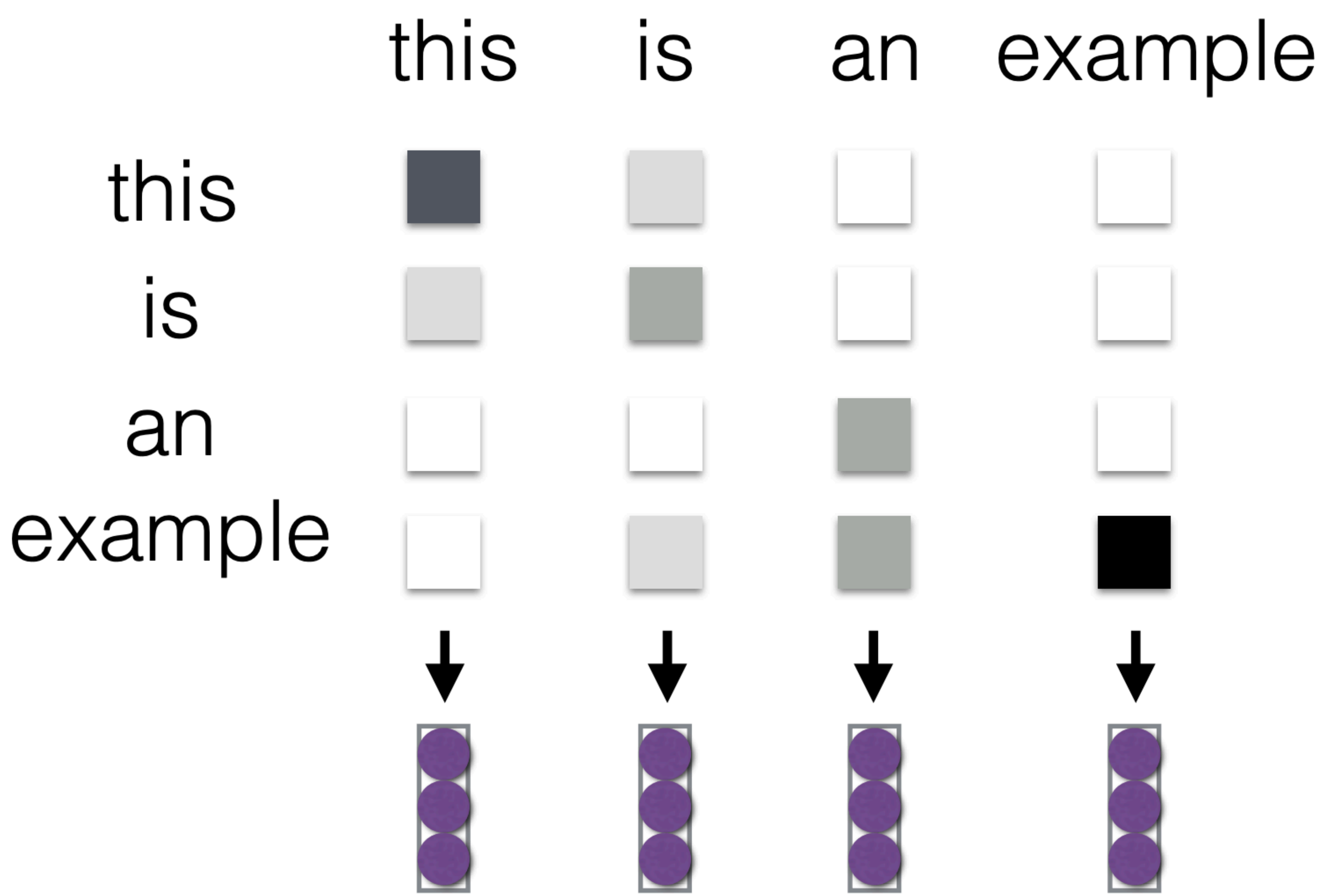
$$\arg\max \prod_{i=1}^{m} p(e_i \mid e_1, \ldots, e_{i-1}, f)$$

- can we enumerate all possible English sentences *e*?

can we just do attention and get rid of recurrence?

Self-attention as an encoder!

(core component of Transformer)

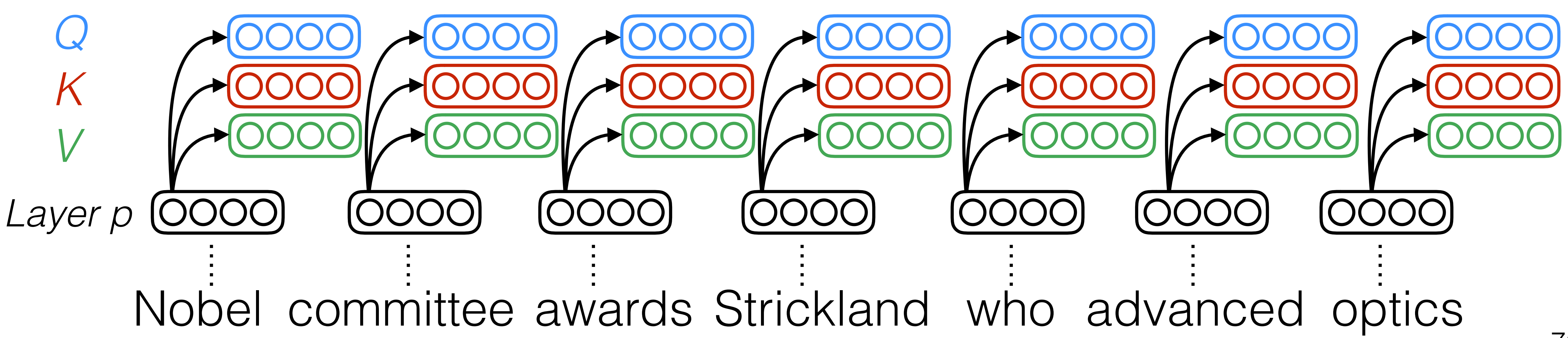


Vaswani et al., 2017

figure: Graham Neubig

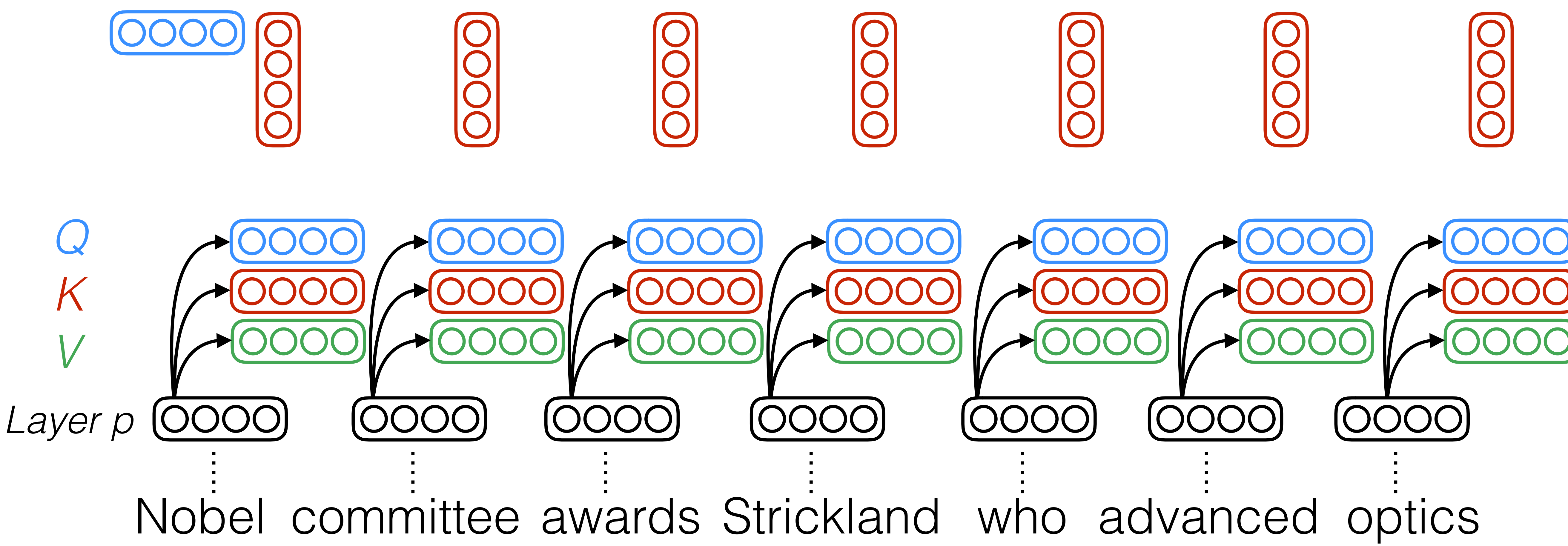
Self-attention

[Vaswani et al. 2017]

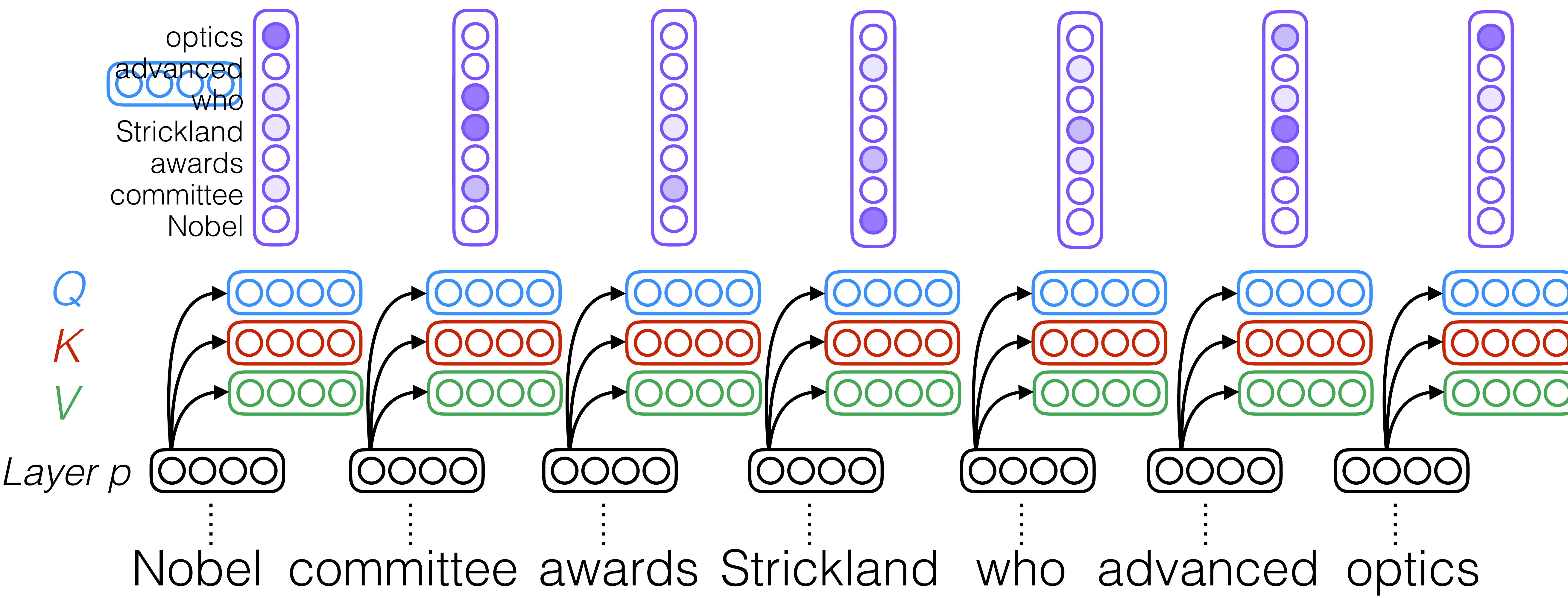


Self-attention

[Vaswani et al. 2017]

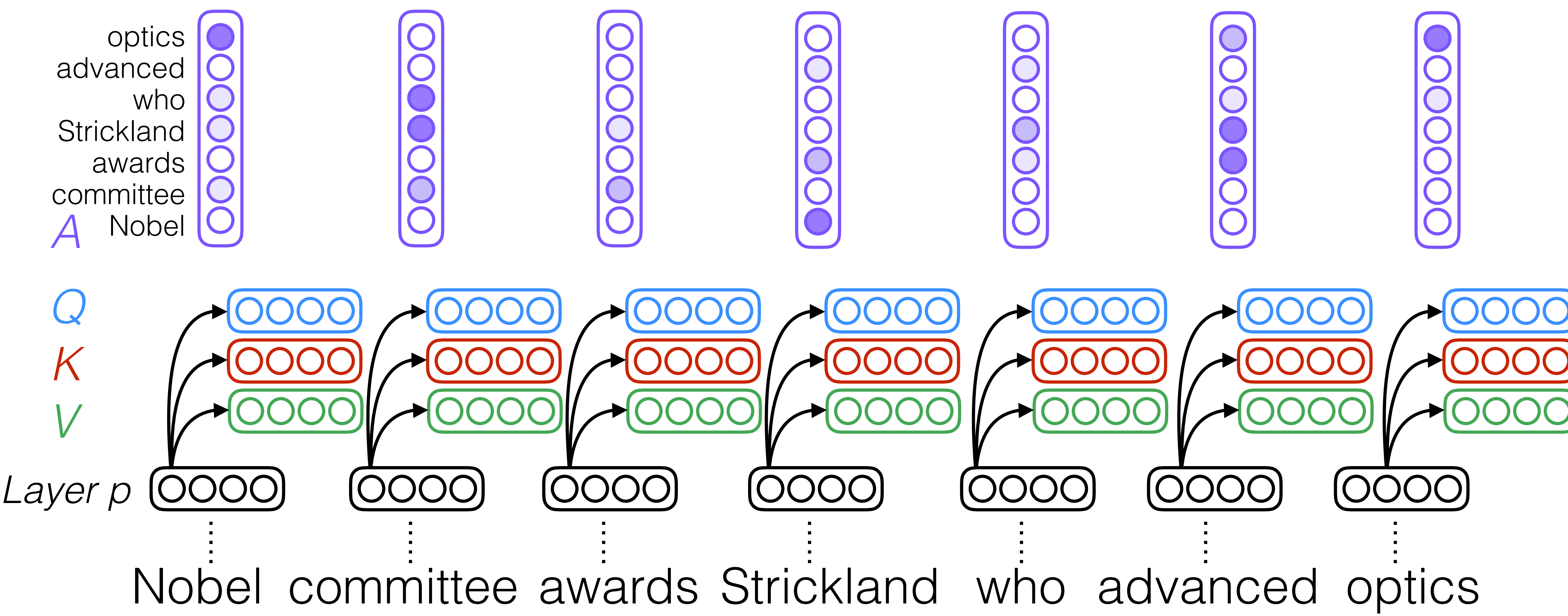


Self-attention



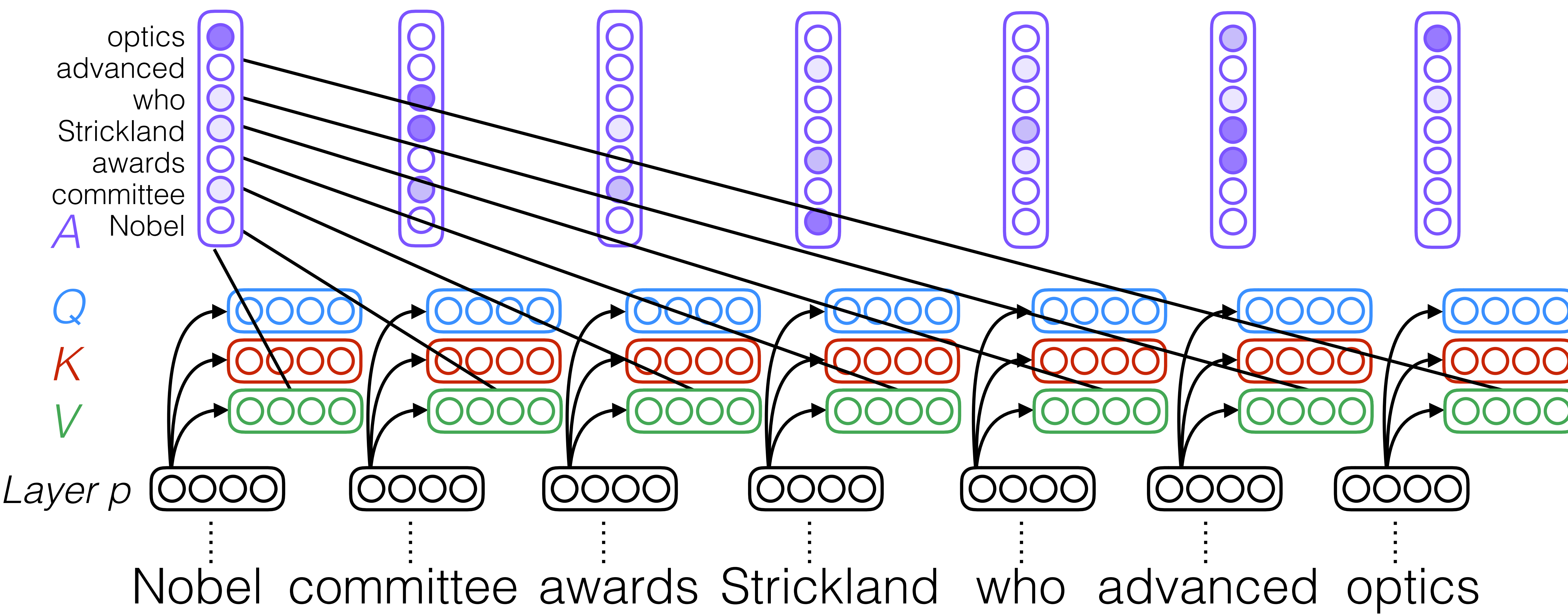
Self-attention

[Vaswani et al. 2017]



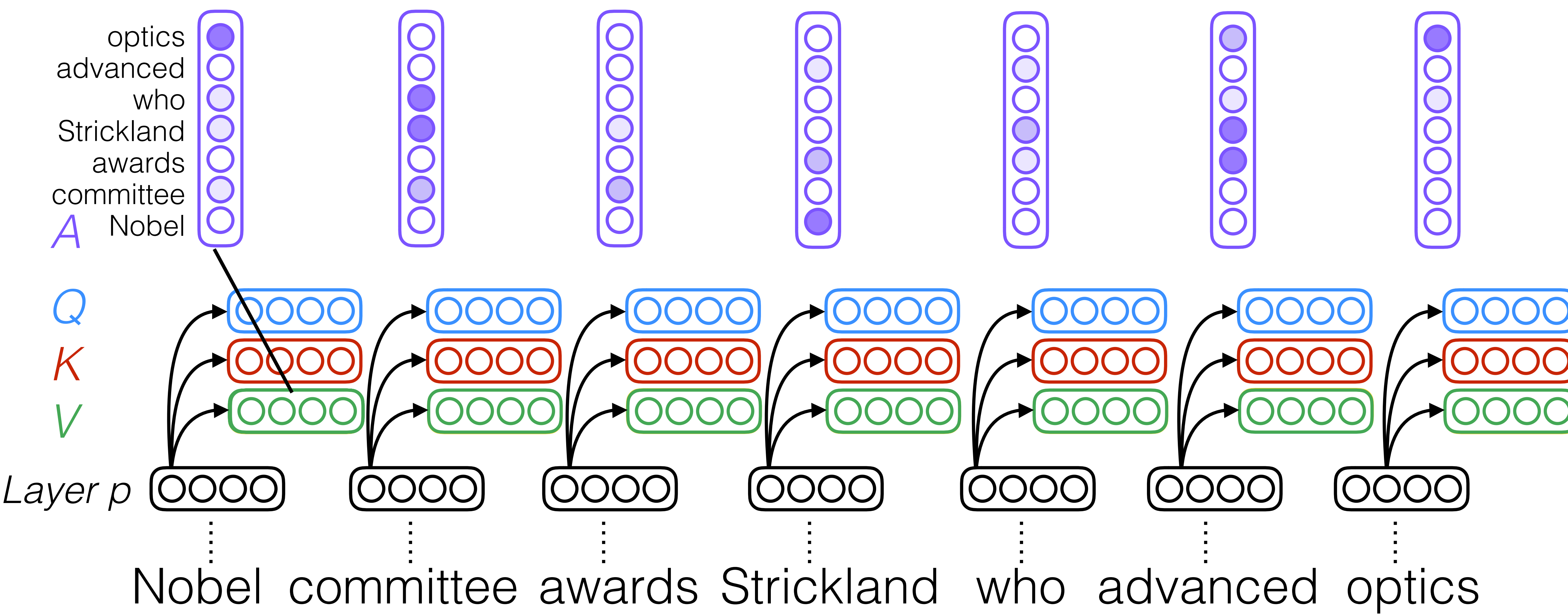
[Vaswani et al. 2017]

Self-attention

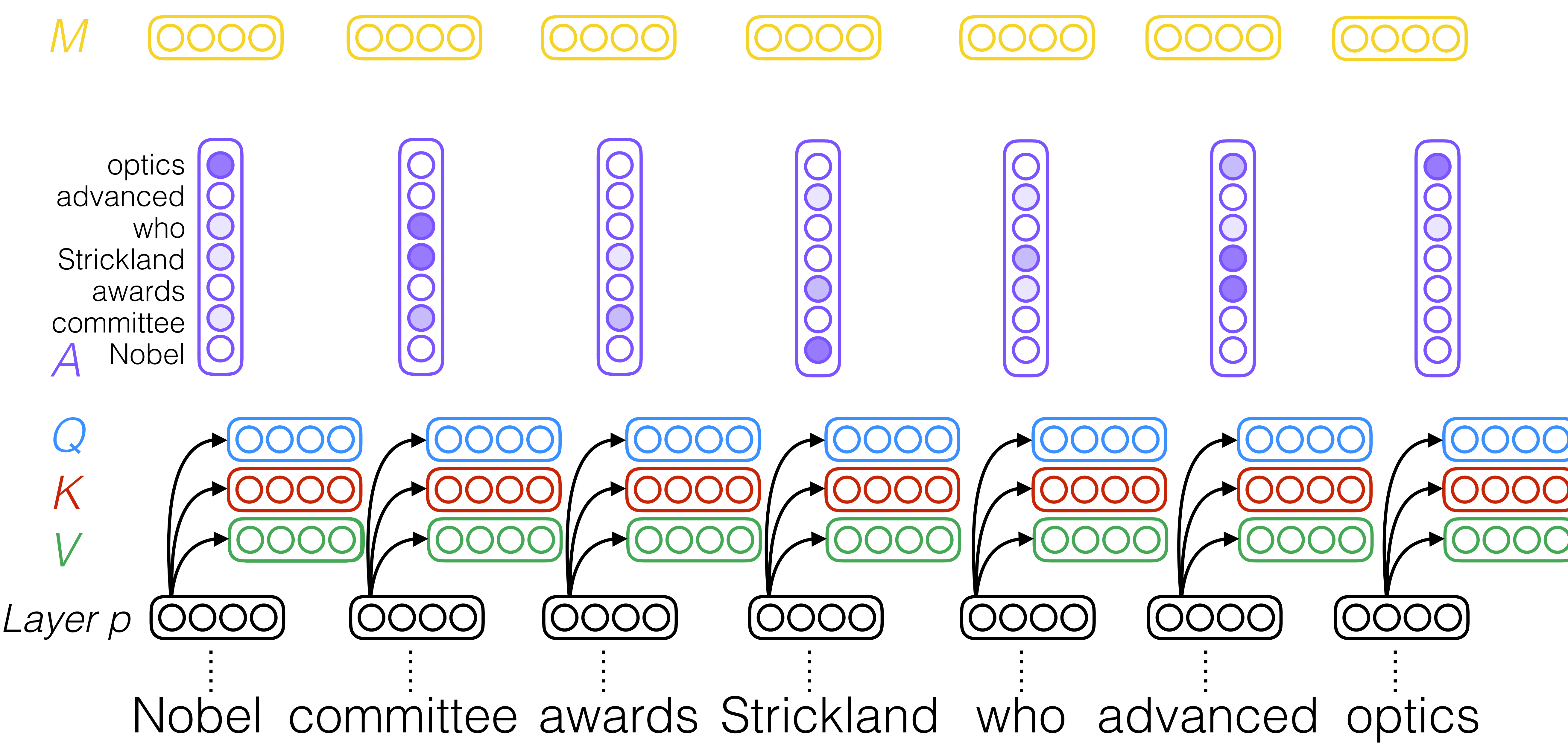


[Vaswani et al. 2017]

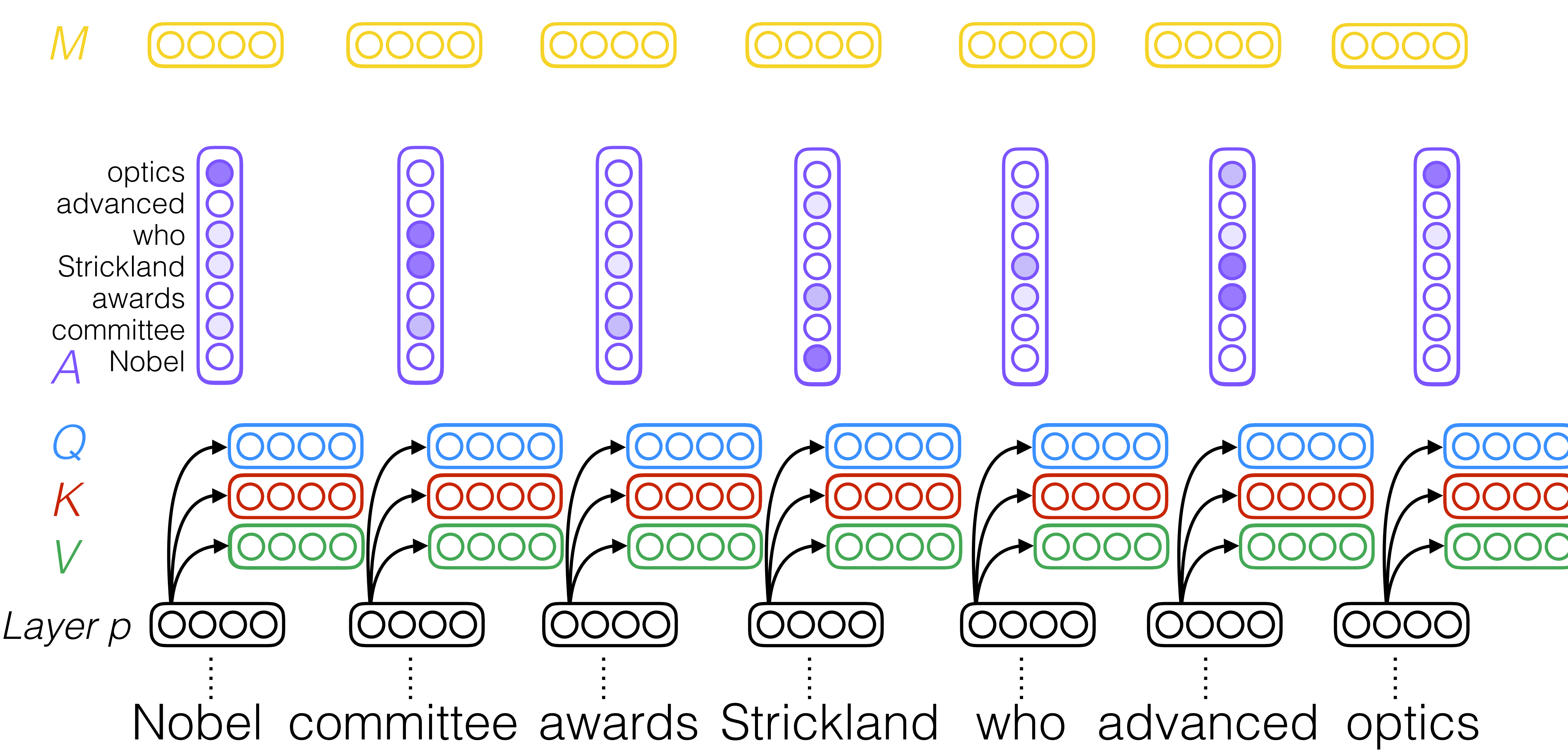
Self-attention



Self-attention

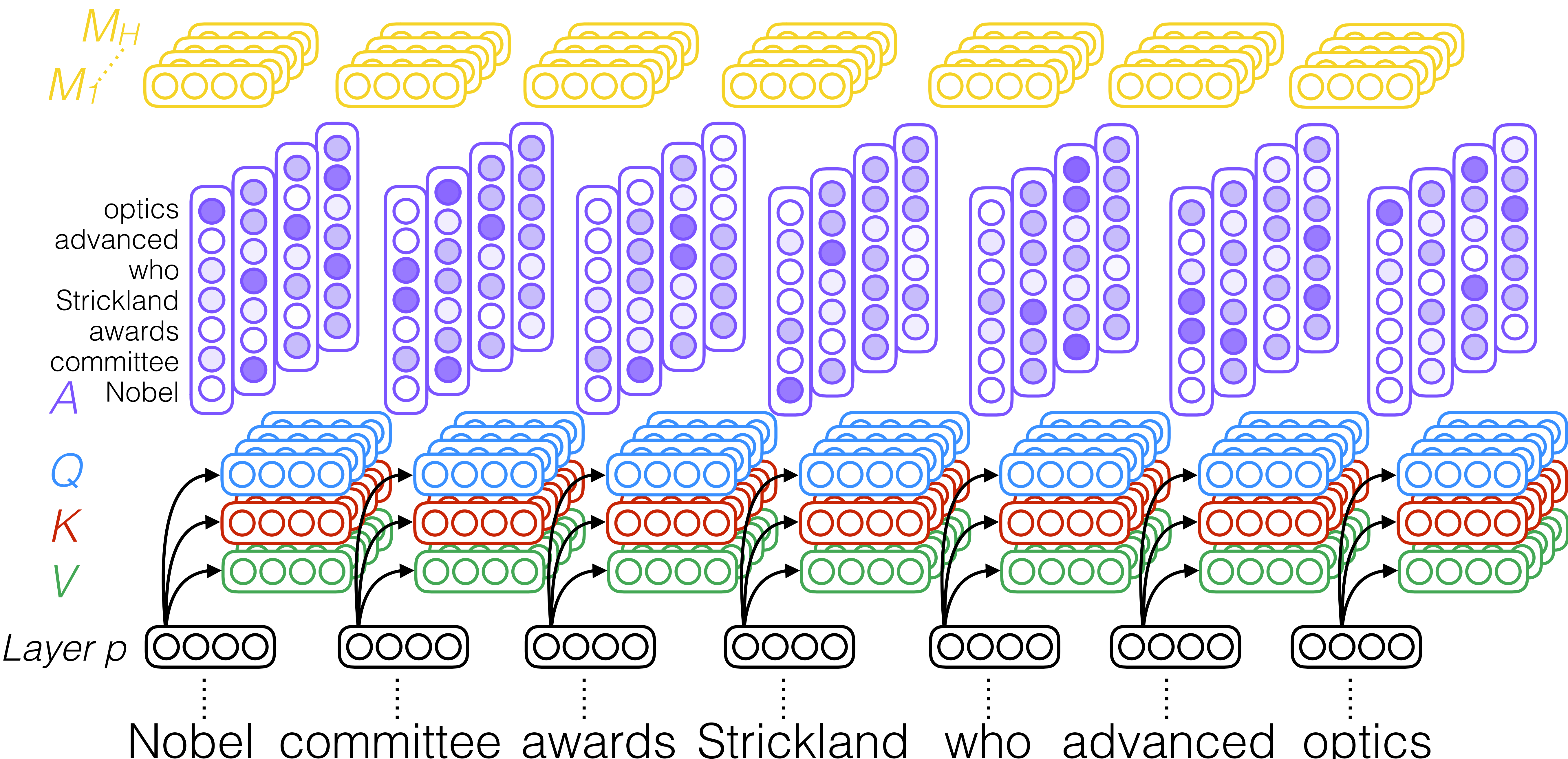


Self-attention

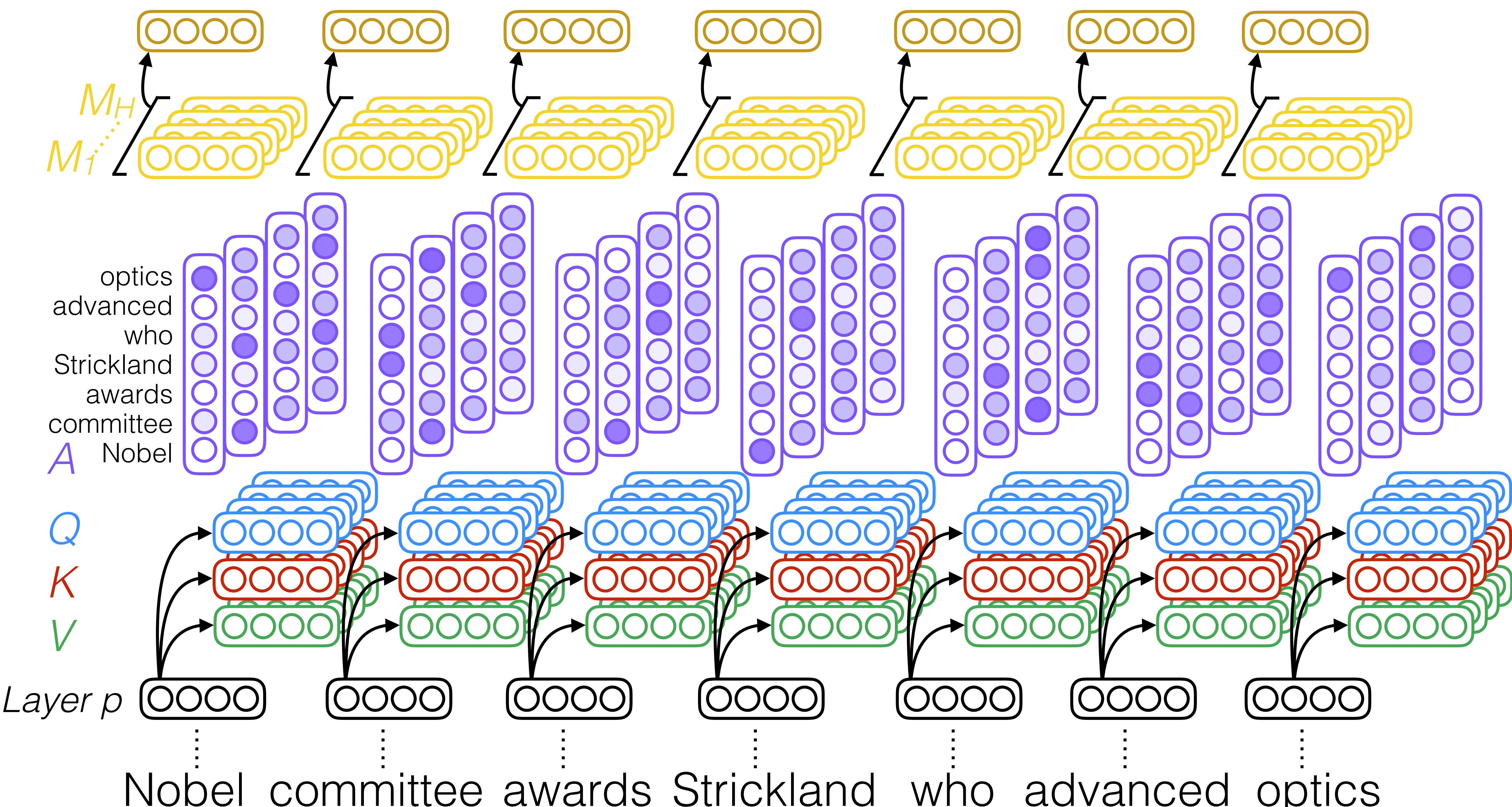


[Vaswani et al. 2017]

Multi-head self-attention

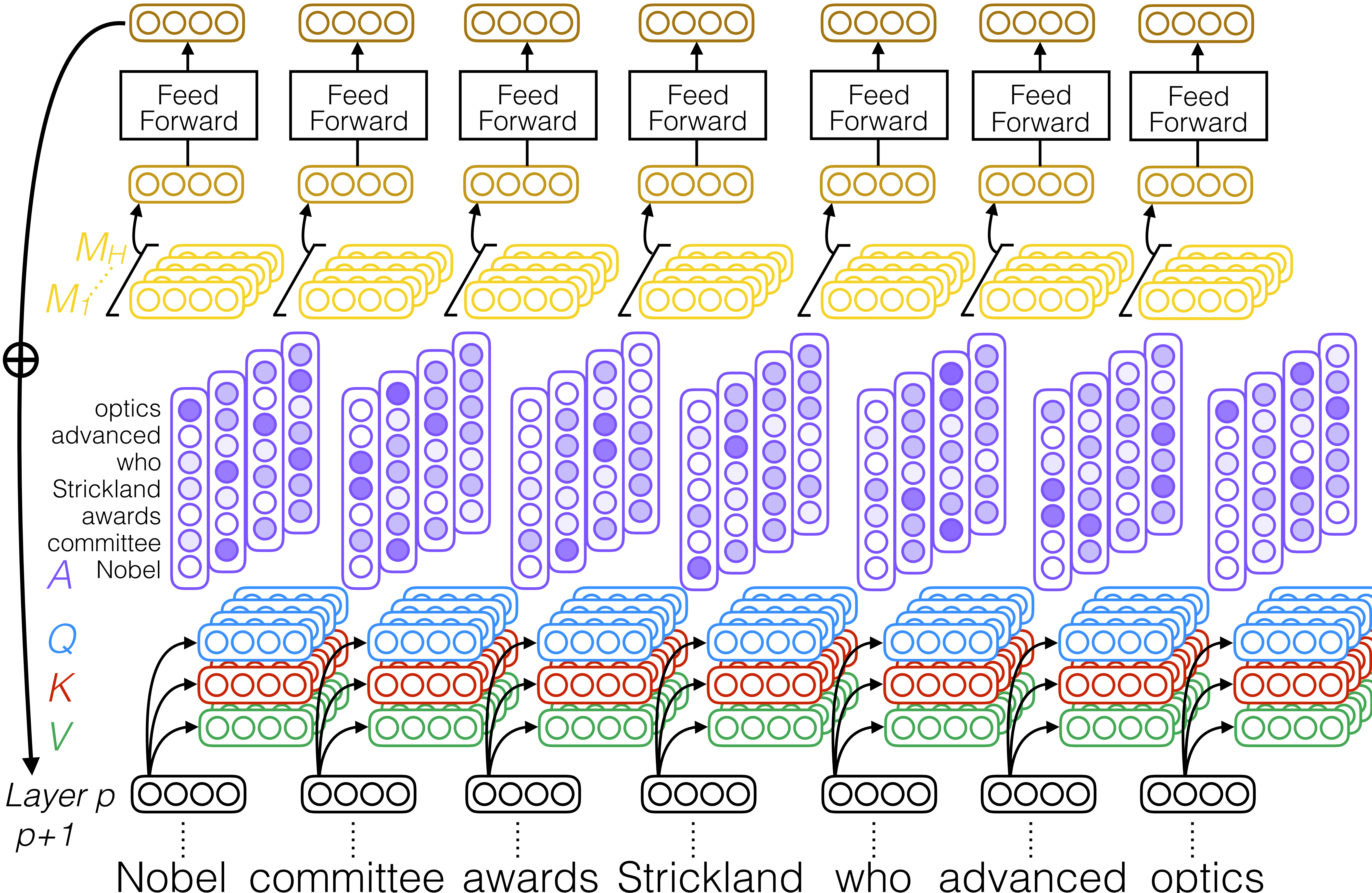


Multi-head self-attention

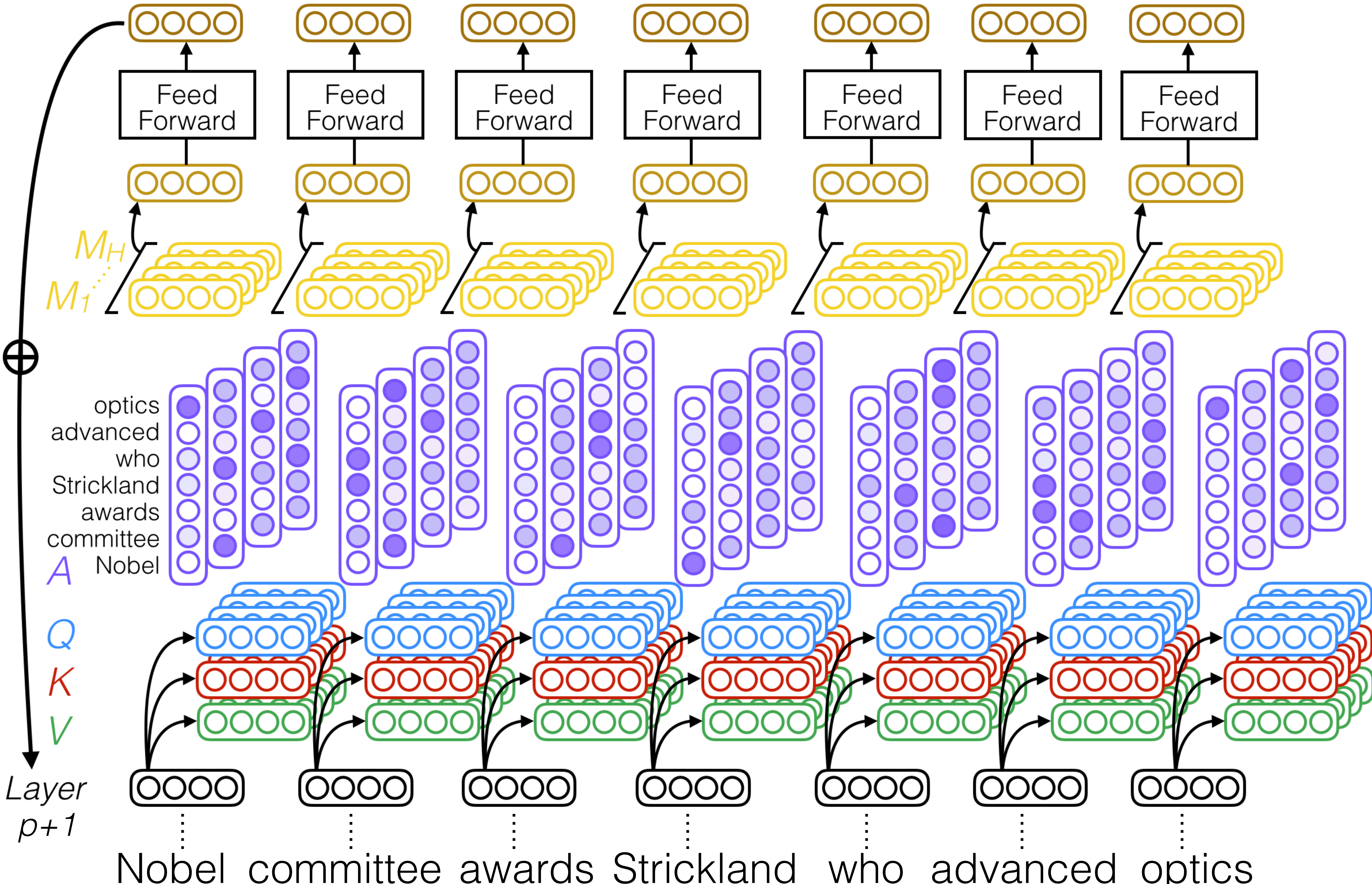


[Vaswani et al. 2017]

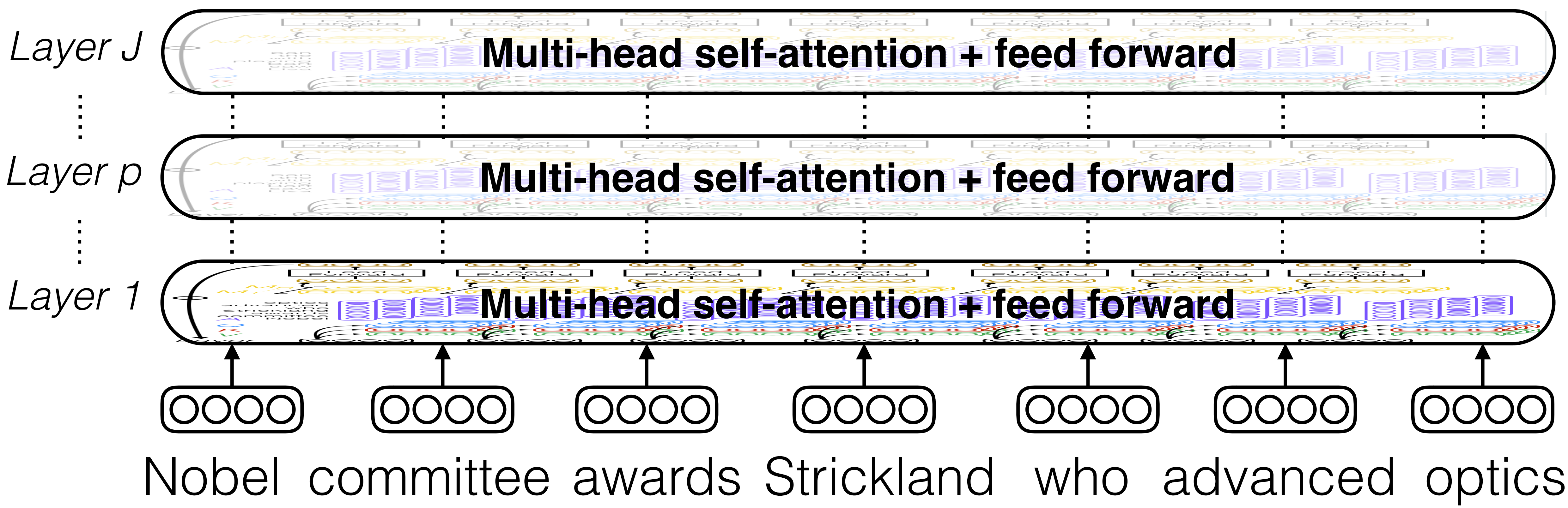
Multi-head self-attention



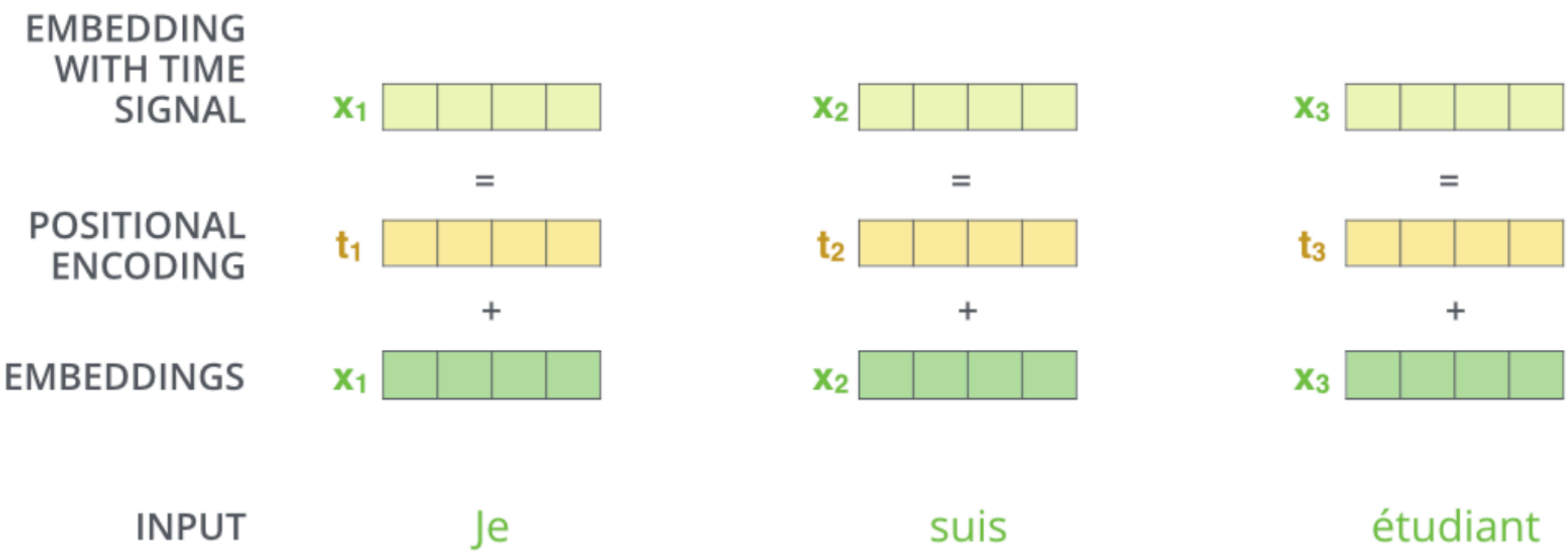
Multi-head self-attention



Multi-head self-attention

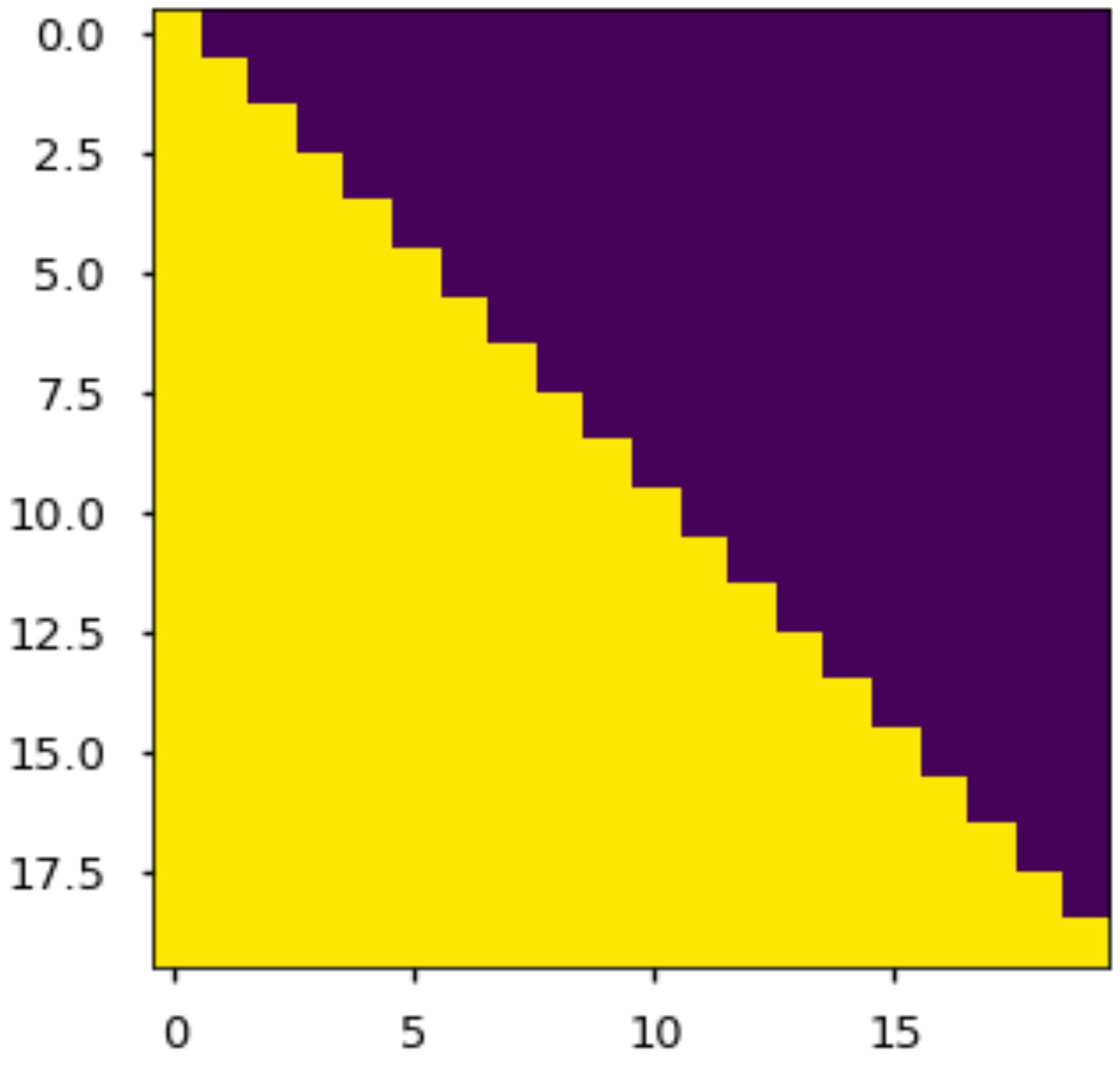


Positional encoding



Last major missing piece:

- Decoder self-attention masking



Byte pair encoding (BPE)

- Deal with rare words / large vocabulary by instead using *subword* tokenization

system	sentence
source	health research institutes
reference	Gesundheitsforschungsinstitute
WDict	Forschungsinstitute
C2-50k	Fo|rs|ch|un|gs|in|st|it|ut|io|ne|n
BPE-60k	Gesundheits|forsch|ungsinstitu|ten
BPE-J90k	Gesundheits|forsch|ungsin|stitute
source	asinine situation
reference	dumme Situation
WDict	asinine situation $\rightarrow$ UNK $\rightarrow$ asinine
C2-50k	as|in|in|e situation $\rightarrow$ As|in|en|si|tu|at|io|n
BPE-60k	as|in|ine situation $\rightarrow$ A|in|line-|Situation
BPE-J90K	as|in|ine situation $\rightarrow$ As|in|in-|Situation

Sennrich et al., ACL 2016

transfer learning

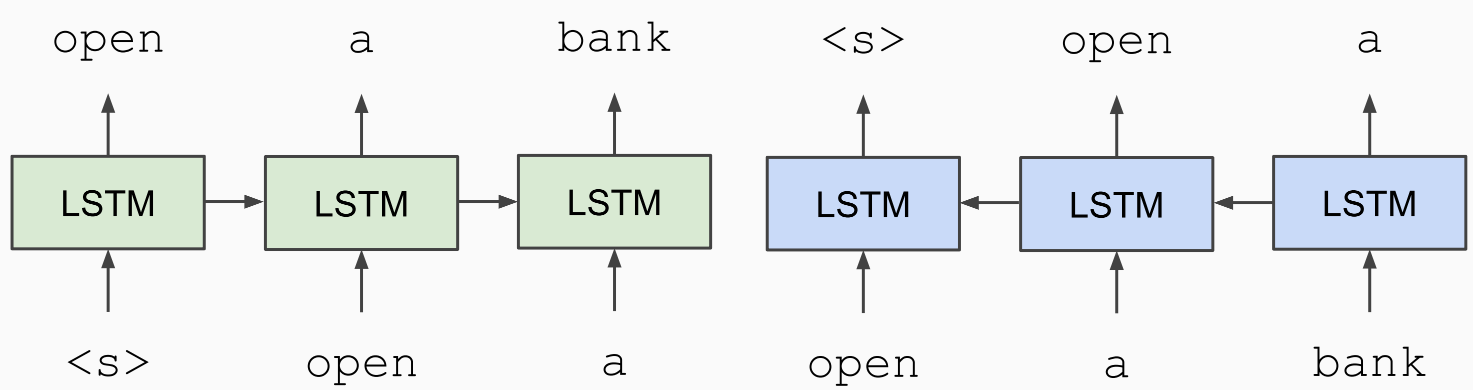
What is transfer learning?

- **In our context:** take a network trained on a task for which it is easy to generate labels, and adapt it to a different task for which it is harder.
- **In computer vision:** train a CNN on ImageNet, transfer its representations to every other CV task
- **In NLP:** train a really big language model on billions of words, transfer to every NLP task!

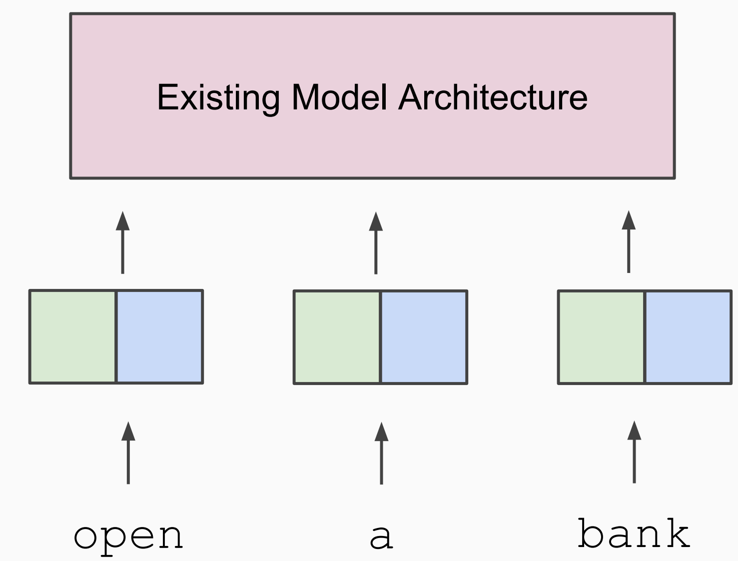
History of Contextual Representations

- *ELMo: Deep Contextual Word Embeddings*, AI2 & University of Washington, 2017

Train Separate Left-to-Right and Right-to-Left LMs

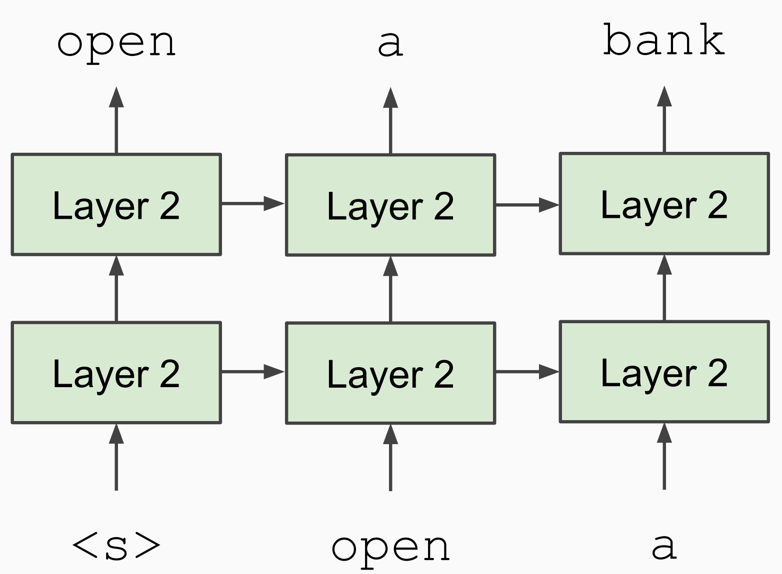


Apply as "Pre-trained Embeddings"

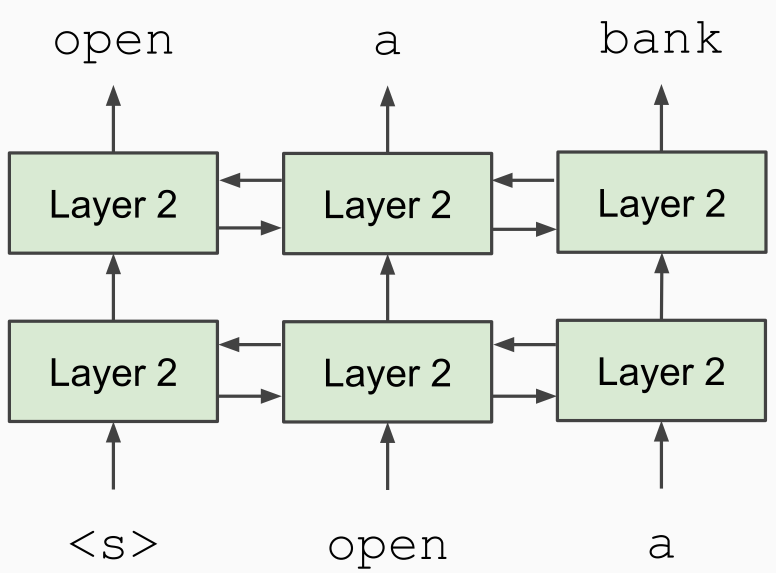


Unidirectional vs. Bidirectional Models

Unidirectional context
Build representation incrementally



Bidirectional context
Words can “see themselves”



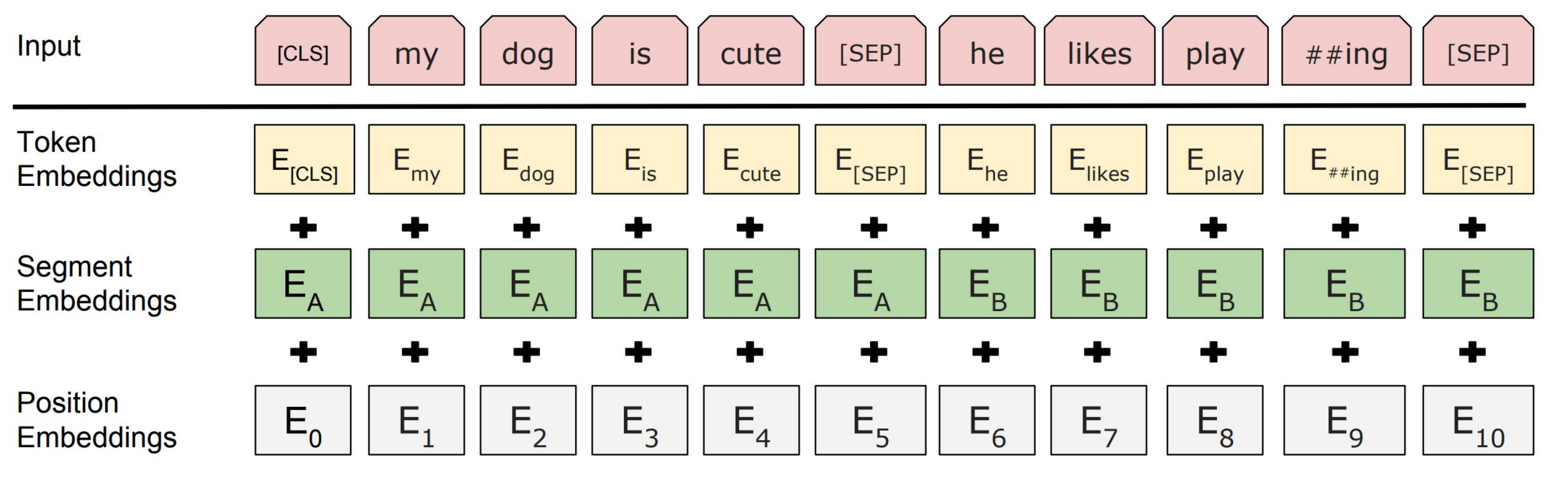
Masked LM

- **Solution:** Mask out *k*% of the input words, and then predict the masked words
  - We always use $k = 15\%$

```
                       store                gallon
                         ↑                     ↑
the man went to the [MASK] to buy a [MASK] of milk
```

What are the pros and cons of increasing *k*?

Input Representation

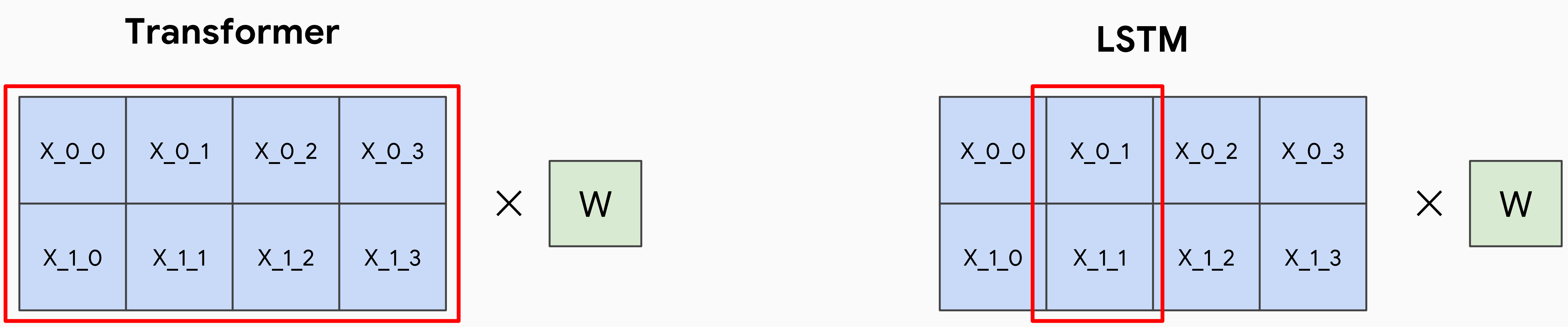


- Use 30,000 WordPiece vocabulary on input.
- Each token is sum of three embeddings
- Single sequence is much more efficient.

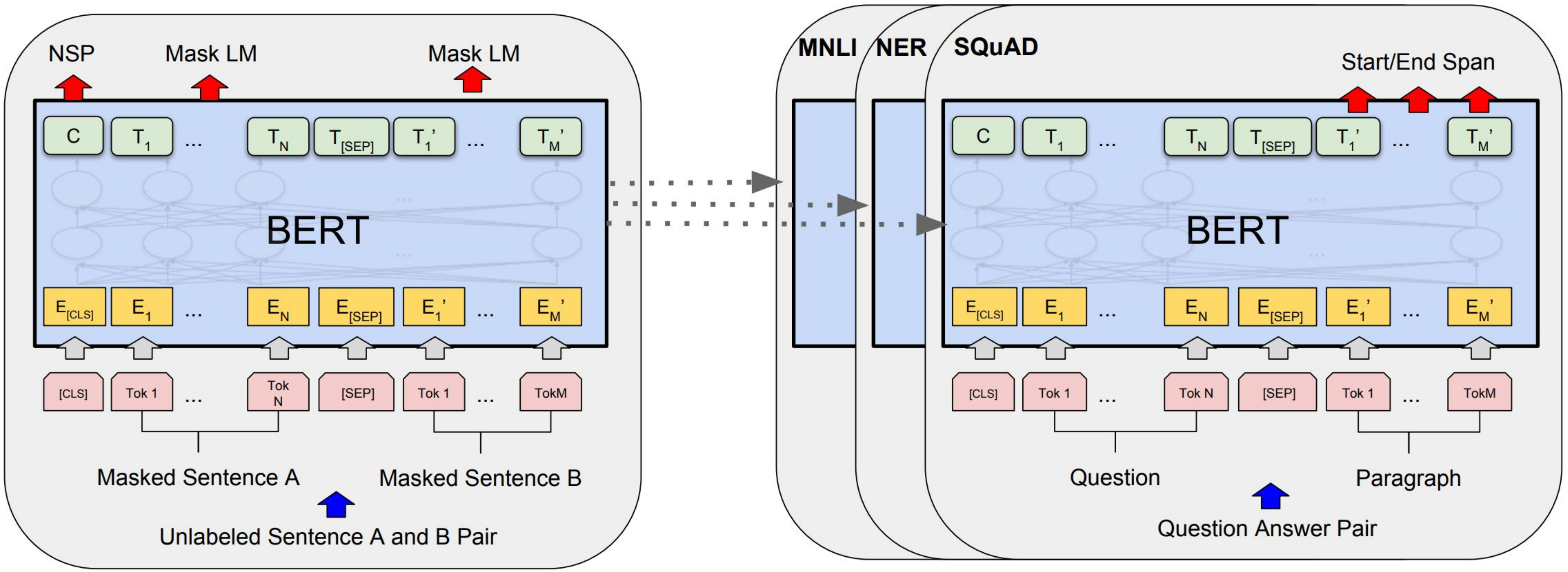
Model Architecture

- Empirical advantages of Transformer vs. LSTM:

1. Self-attention == no locality bias
   - Long-distance context has “equal opportunity”
2. Single multiplication per layer == efficiency on TPU
   - Effective batch size is number of *words*, not *sequences*



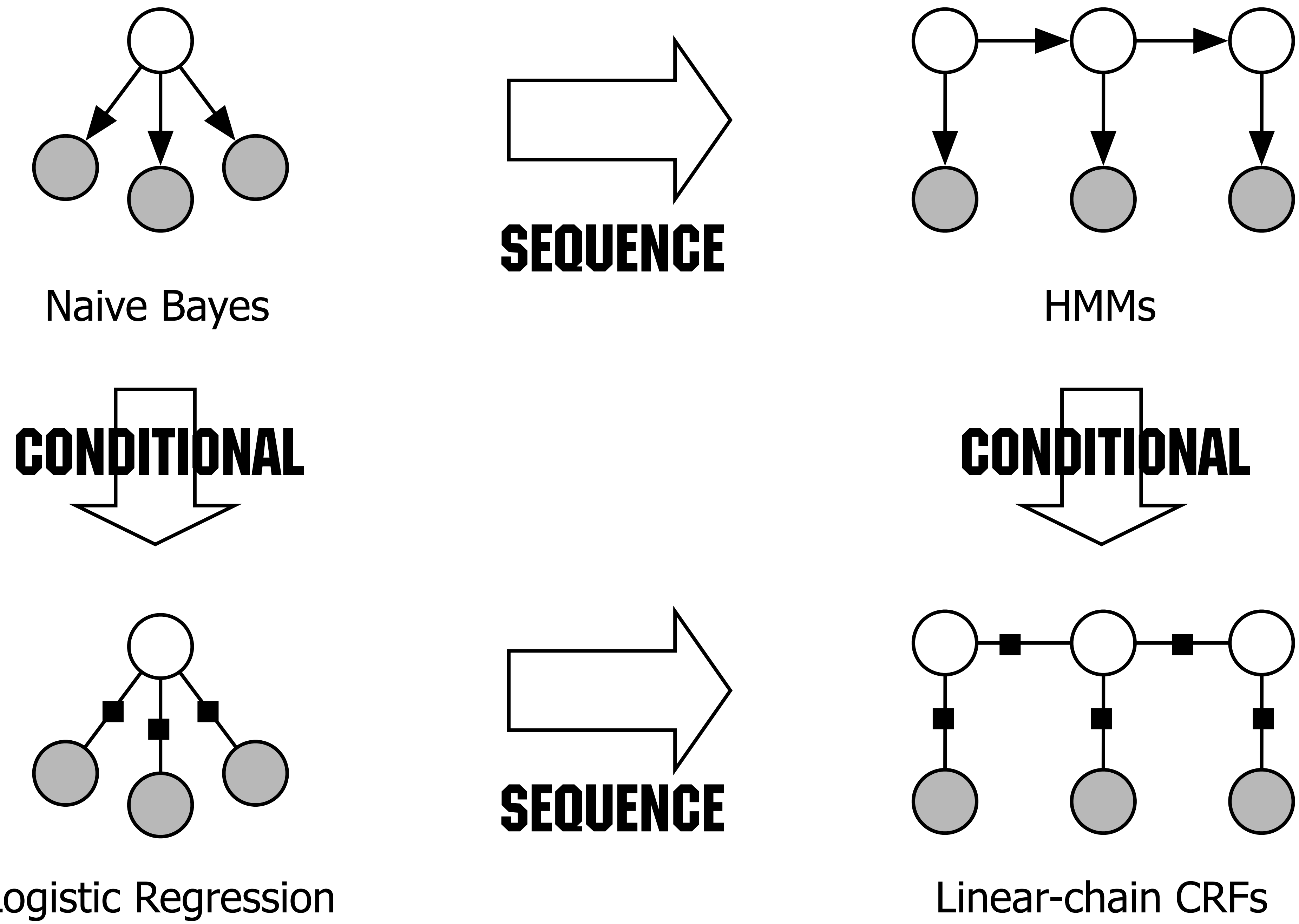
Fine-Tuning Procedure



Pre-training
Fine-Tuning

HMMs / sequence modeling

These are all **log-linear** models



HMM Definition

Assume K parts of speech, a lexicon size of V, a series of observations $\{x_1, \dots, x_N\}$, and a series of unobserved states $\{z_1, \dots, z_N\}$.

- π A distribution over start states (vector of length K):
  $\pi_i = p(z_1 = i)$
- θ Transition matrix (matrix of size K by K):
  $\theta_{i,j} = p(z_n = j | z_{n-1} = i)$
- β An emission matrix (matrix of size K by V):
  $\beta_{j,w} = p(x_n = w | z_n = j)$

Two problems: How do we move from data to a model? (Estimation) How do we move from a model and unlabled data to labeled data? (Inference)

Training Sentences

x = tokens
z = POS tags

x	here	come	old	flattop
z	MOD	V	MOD	N

a	crowd	of	people	stopped	and	stared
DET	N	PREP	N	V	CONJ	V

gotta	get	you	into	my	life
V	V	PRO	PREP	PRO	V

and	I	love	her
CONJ	PRO	V	PRO

Training Sentences

here	come	old	flattop
MOD	V	MOD	N

a	crowd	of	people	stopped	and	stared
DET	N	PREP	N	V	CONJ	V

gotta	get	you	into	my	life
V	V	PRO	PREP	PRO	N

and	I	love	her
CONJ	PRO	V	PRO

Training Sentences

here	come	old	flattop
MOD	V	MOD	N

a	crowd	of	people	stopped	and	stared
DET	N	PREP	N	V	CONJ	V

gotta	get	you	into	my	life
V	V	PRO	PREP	PRO	N

and	I	love	her
CONJ	PRO	V	PRO

Viterbi Algorithm

- Given an unobserved sequence of length L, $\{x_1, \ldots, x_L\}$, we want to find a sequence $\{z_1 \ldots z_L\}$ with the highest probability.
- It's impossible to compute K^L possibilities.
- So, we use dynamic programming to compute most likely tags for each token subsequence from 0 to t that ends in state k.
- Memoization: fill a table of solutions of sub-problems
- Solve larger problems by composing sub-solutions
- Base case:

$$\delta_1(k) = \pi_k \beta_{k,x_i} \tag{1}$$

- Recursion:

$$\delta_n(k) = \max_j (\delta_{n-1}(j)\theta_{j,k}) \beta_{k,x_n} \tag{2}$$