0 to ~80 in 90 minutes

a shallow intro to deep networks

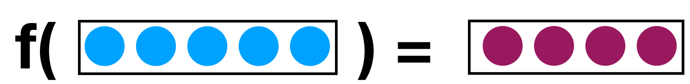
Yoav Goldberg

NLPL Winter School 2020

"I do think that most participants will know the basics about embeddings, neural networks and loss functions (although the depth of their knowledge will vary, of course)."

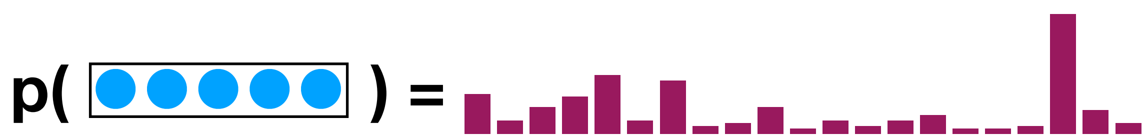
"I do think that **most** participants will know the **basics** about embeddings, neural networks and loss functions (although the depth of their knowledge will vary, of course)."

Neural Networks



functions from vectors to vectors

Neural Networks

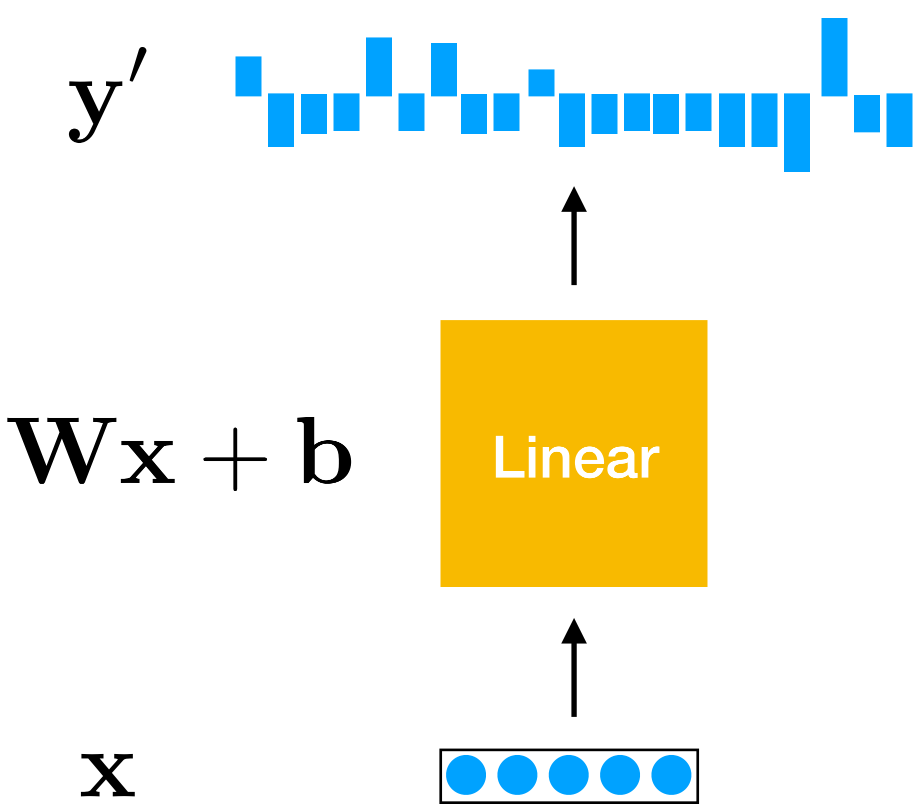


functions from vectors
to probabilities

(these are still functions from vectors to vectors)

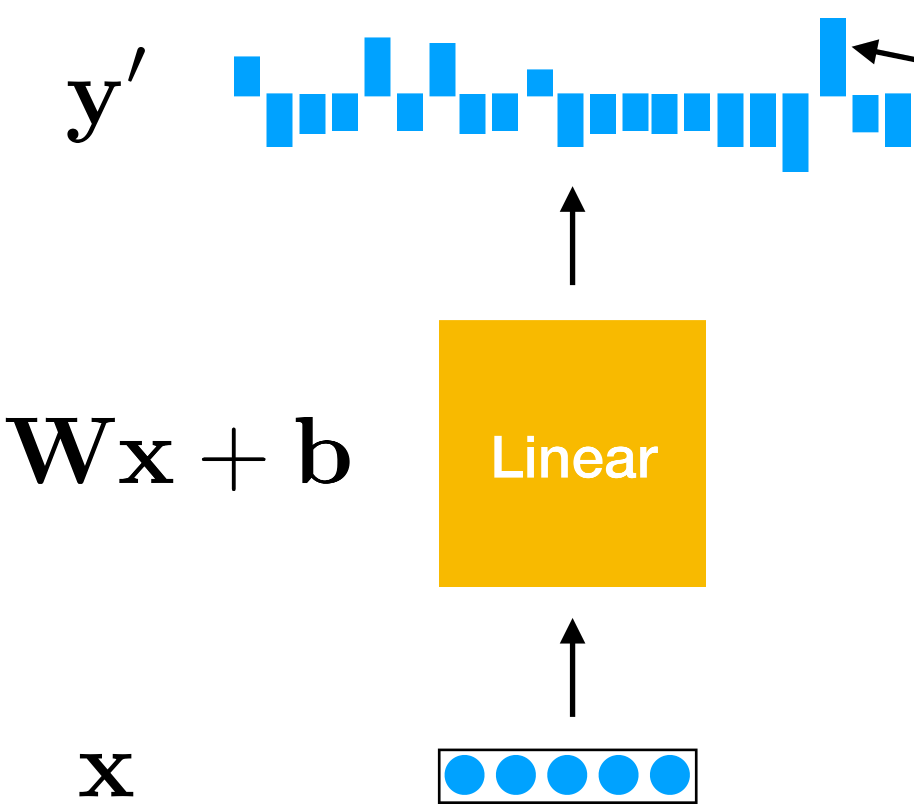
Predicting from a vector

Predict from a vector (Linear Layer)



$$predict(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Predict from a vector (Linear Layer)

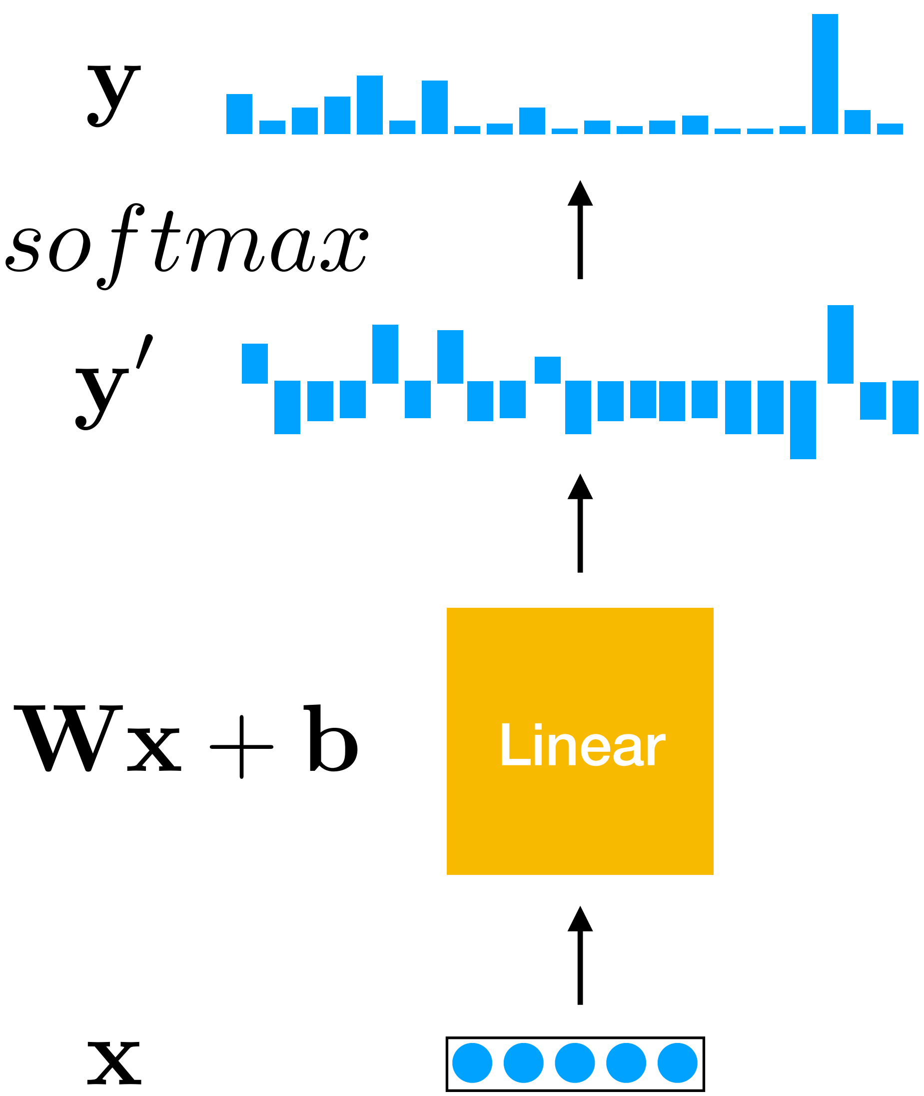


$$predict(\mathbf{x}) = \mathbf{W}\mathbf{x}+\mathbf{b}$$

$$predict(\mathbf{x}) = \arg\max_i (\mathbf{W}\mathbf{x}+\mathbf{b})_{[i]}$$

Predict from a vector (Linear Layer + softmax)

$$p(y = ? | \mathbf{x})$$



$$predict(\mathbf{x}) = softmax(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$softmax(\mathbf{x})_{[i]} = \frac{e^{\mathbf{x}_{[i]}}}{\sum_j e^{\mathbf{x}_{[j]}}}$$

Predict from a vector (Linear Layer + softmax)

$$p(y = ?|\mathbf{x})$$

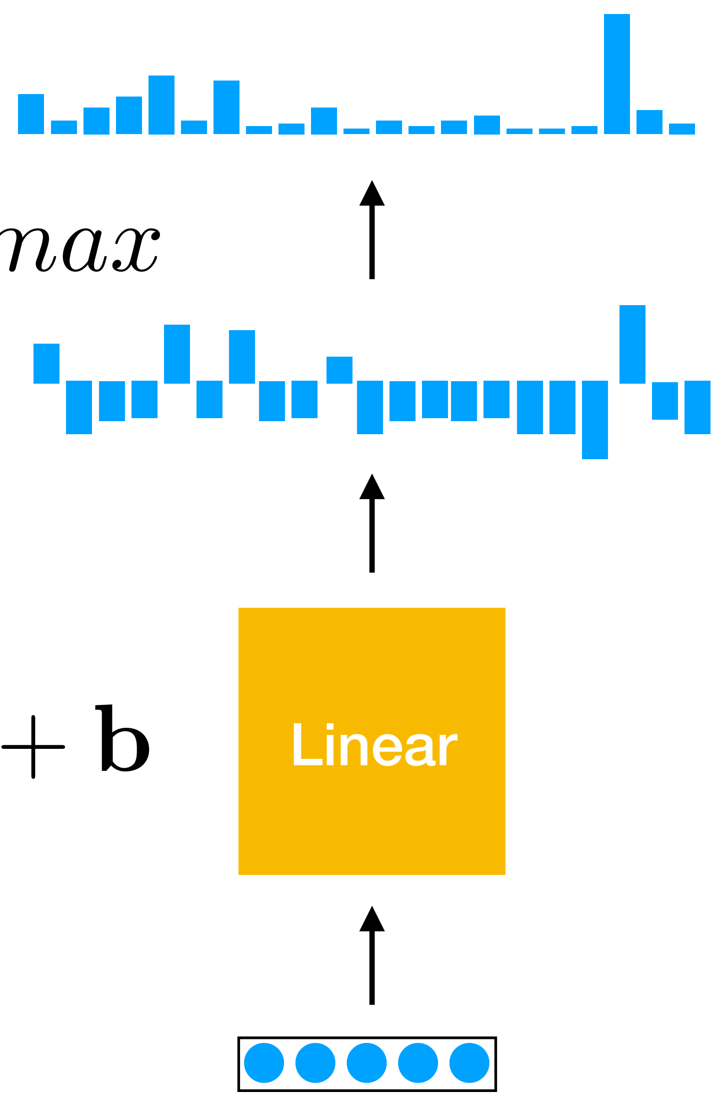
$\mathbf{y}$

$softmax$

$\mathbf{y}'$

$\mathbf{W}\mathbf{x} + \mathbf{b}$

$\mathbf{x}$



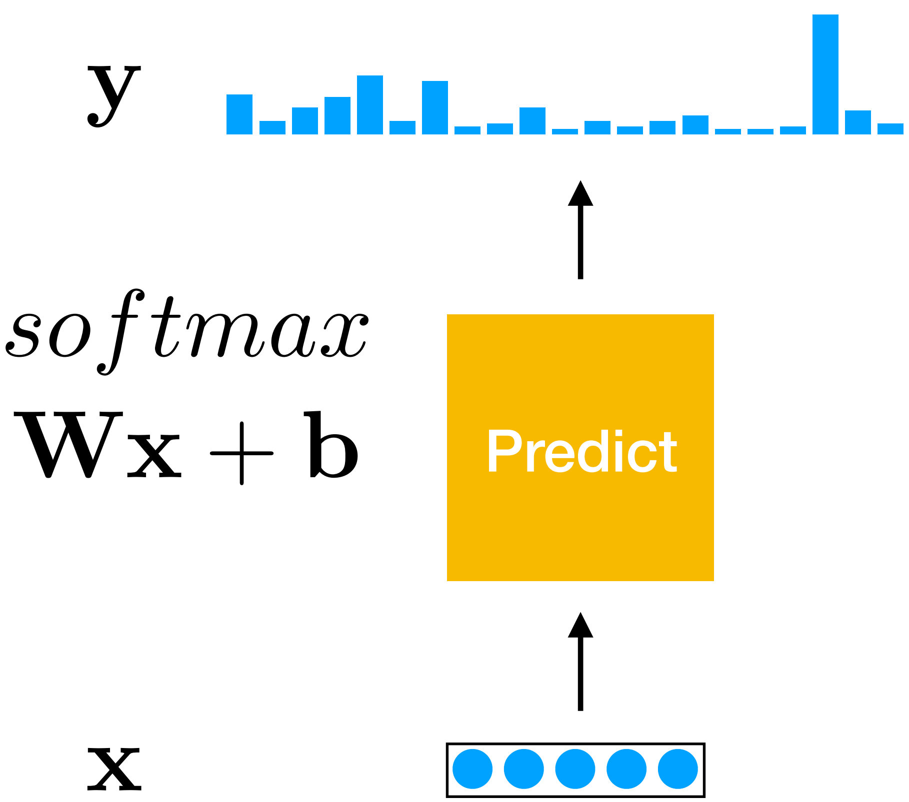
$$predict(\mathbf{x}) = softmax(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$softmax(\mathbf{x})_{[i]} = \frac{e^{\mathbf{x}_{[i]}}}{\sum_j e^{\mathbf{x}_{[j]}}}$$

(can still take the argmax, will yield same result)

Predict from a vector (Linear Layer + softmax)

$$p(y = ?|\mathbf{x})$$



$$predict(\mathbf{x}) = softmax(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$softmax(\mathbf{x})_{[i]} = \frac{e^{\mathbf{x}_{[i]}}}{\sum_j e^{\mathbf{x}_{[j]}}}$$

Training:
Learning as optimization

Data:

$$\mathbf{x_1}, \ldots, \mathbf{x_n}$$

$$\mathbf{y_1}, \ldots, \mathbf{y_n}$$

(y_i are vectors, why?)

Desired:

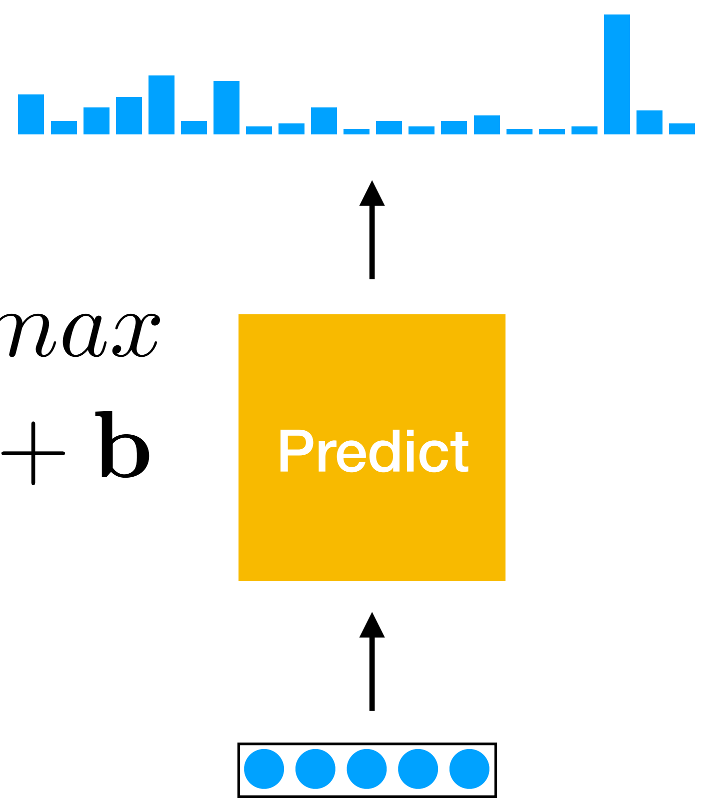
$$f_\theta(\mathbf{x})$$

"that works well"

$$\theta = \mathbf{W}, \mathbf{b}$$

$softmax$

$\mathbf{Wx} + \mathbf{b}$



- **hypothesis class**
- **parameters**
- **a search problem**

Training:
Learning as optimization

$$\mathbf{x_1}, \ldots, \mathbf{x_n}$$

$$\mathbf{y_1}, \ldots, \mathbf{y_n}$$

Desired:

$f_\theta(\mathbf{x})$ **"that works well"**

$$\mathbf{Y} = \mathbf{y_1}, \ldots, \mathbf{y_n}$$

$$\hat{\mathbf{Y}}_\theta = f_\theta(\mathbf{x_1}), \ldots, f_\theta(\mathbf{x_n})$$

$$\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}_\theta)$$

loss function

Training:
Learning as optimization

$$\mathbf{x_1}, \dots, \mathbf{x_n}$$

$$\mathbf{y_1}, \dots, \mathbf{y_n}$$

Desired:

$f_\theta(\mathbf{x})$ **"that works well"**

$$\mathbf{Y} = \mathbf{y_1}, \dots, \mathbf{y_n}$$

$$\hat{\mathbf{Y}}_\theta = f_\theta(\mathbf{x_1}), \dots, f_\theta(\mathbf{x_n})$$

$$\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}_\theta) \propto \sum_{i=1}^{n} \ell(\mathbf{y_i}, f_\theta(\mathbf{x_i}))$$

loss function

decomposed over items

Training: Learning as optimization

$$\arg\min_{\theta} \mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}_{\theta})$$

solved with gradient based methods

Desired:

$f_{\theta}(\mathbf{x})$ **"that works well"**

$$\mathbf{Y} = \mathbf{y_1}, \ldots, \mathbf{y_n}$$

$$\hat{\mathbf{Y}}_{\theta} = f_{\theta}(\mathbf{x_1}), \ldots, f_{\theta}(\mathbf{x_n})$$

$$\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}_{\theta}) \propto \sum_{i=1}^{n} \ell(\mathbf{y_i}, f_{\theta}(\mathbf{x_i}))$$

loss function

decomposed over items

Training: cross-entropy loss

$$\arg\min_{\theta} \mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}_{\theta}) \propto \sum_{i=1}^{n} \ell(\mathbf{y_i}, f_{\theta}(\mathbf{x_i}))$$

When prediction are "probabilities" $\hat{\mathbf{y}}_{[k]} = P(y = k|\mathbf{x})$

$$\ell_{\text{cross-ent}} = -\sum_{k} \mathbf{y}_{[k]} \log \hat{\mathbf{y}}_{[k]}$$

for "hard" (0 or 1) labels: $\boxed{\ell_{\text{cross-ent}} = -\log \hat{\mathbf{y}}_{[t]}}$

Training: cross-entropy loss

other loss functions are available. but not today.

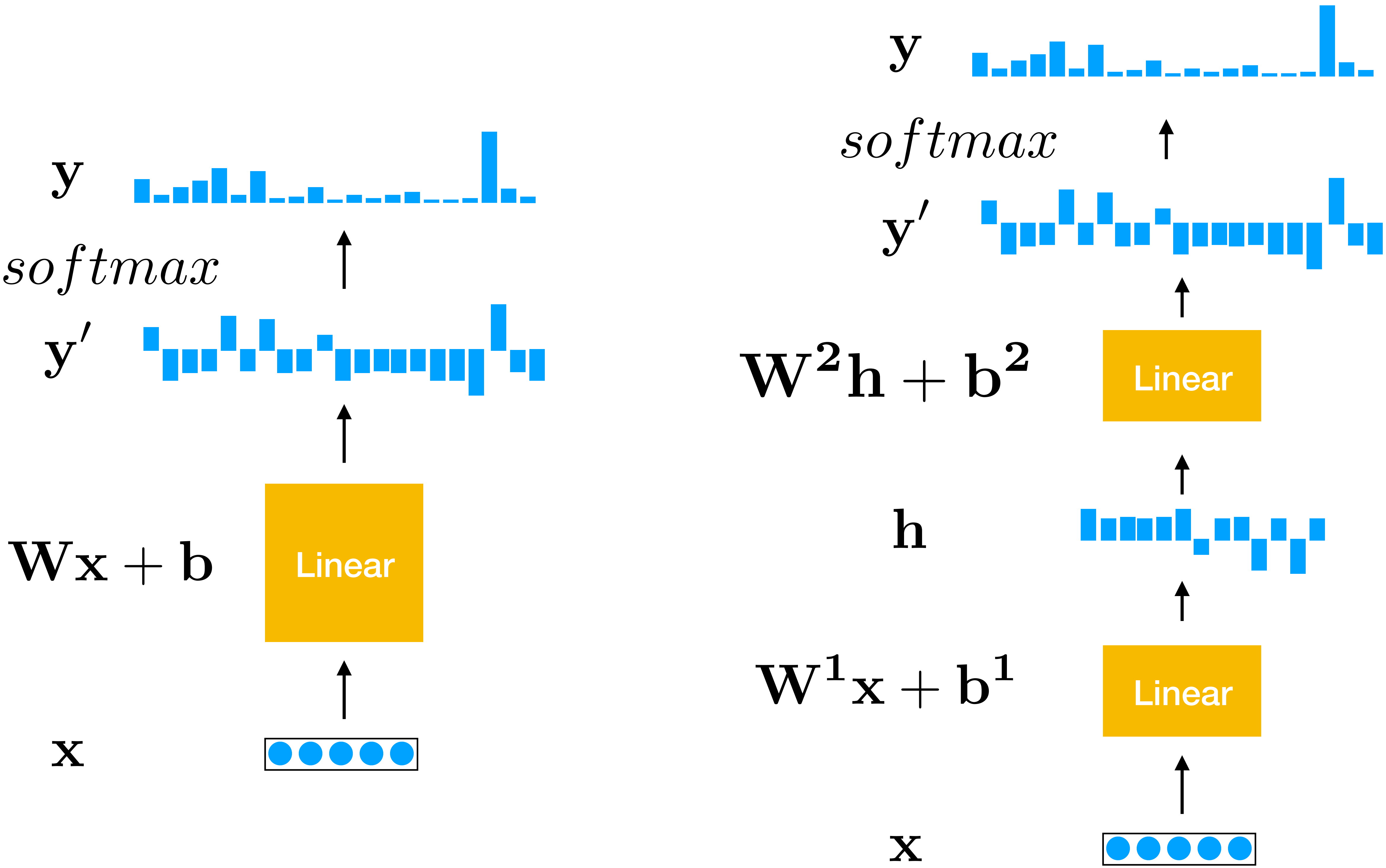
$$\arg\min_{\theta} \mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}_{\theta}) \propto \sum_{i=1}^{n} \ell(\mathbf{y_i}, f_{\theta}(\mathbf{x_i}))$$

When prediction are "probabilities" $\hat{\mathbf{y}}_{[k]} = P(y = k|\mathbf{x})$

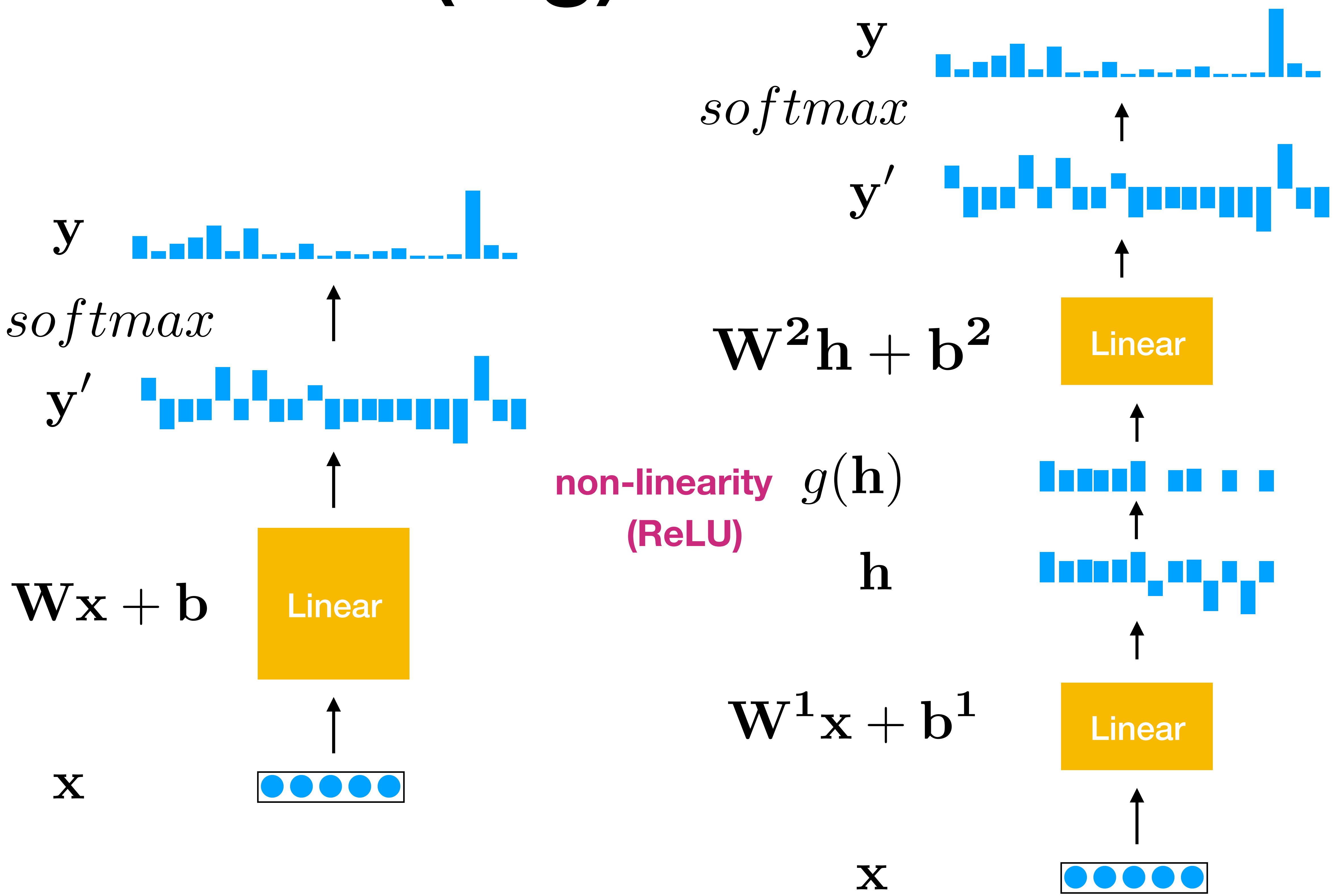
$$\ell_{\text{cross-ent}} = -\sum_{k} \mathbf{y}_{[k]} \log \hat{\mathbf{y}}_{[k]}$$

for "hard" (0 or 1) labels: $\boxed{\ell_{\text{cross-ent}} = -\log \hat{\mathbf{y}}_{[t]}}$

Hypothesis classes: from (log) linear to MLP

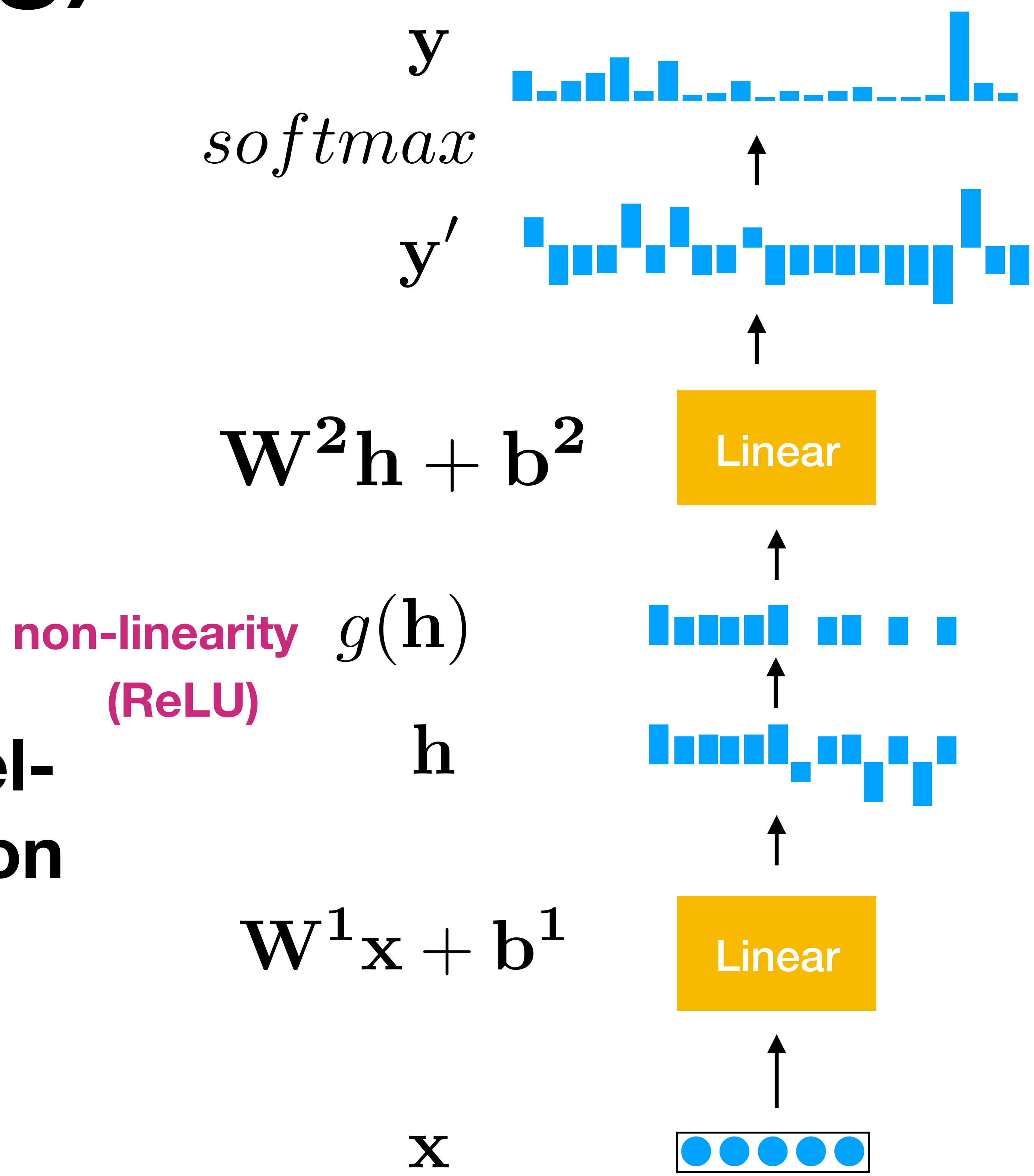


Hypothesis classes: from (log) linear to MLP



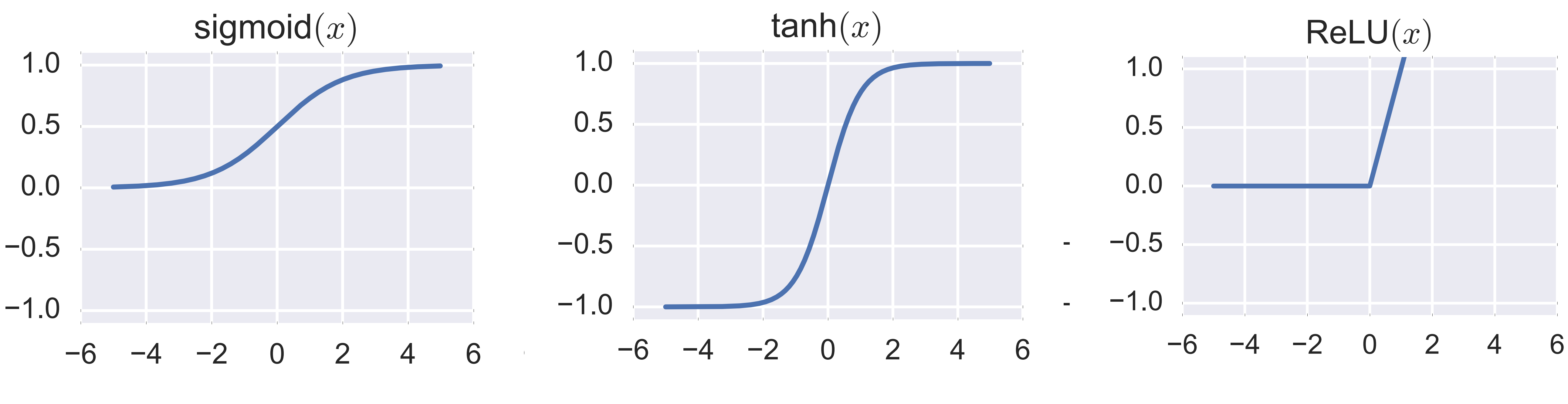
Hypothesis classes: from (log) linear to MLP

MLP (multi-layer perceptron) is strictly more powerful than linear.
Can learn any borel-measurable function (if large enough)

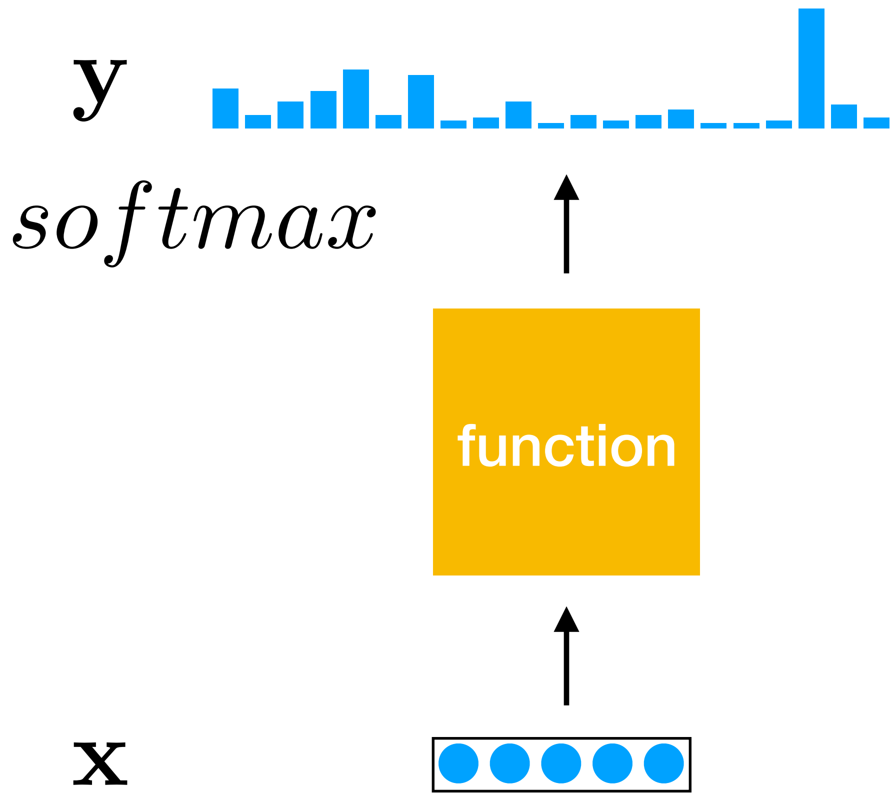


Non-linearities / Activations

the common ones



Neural Network



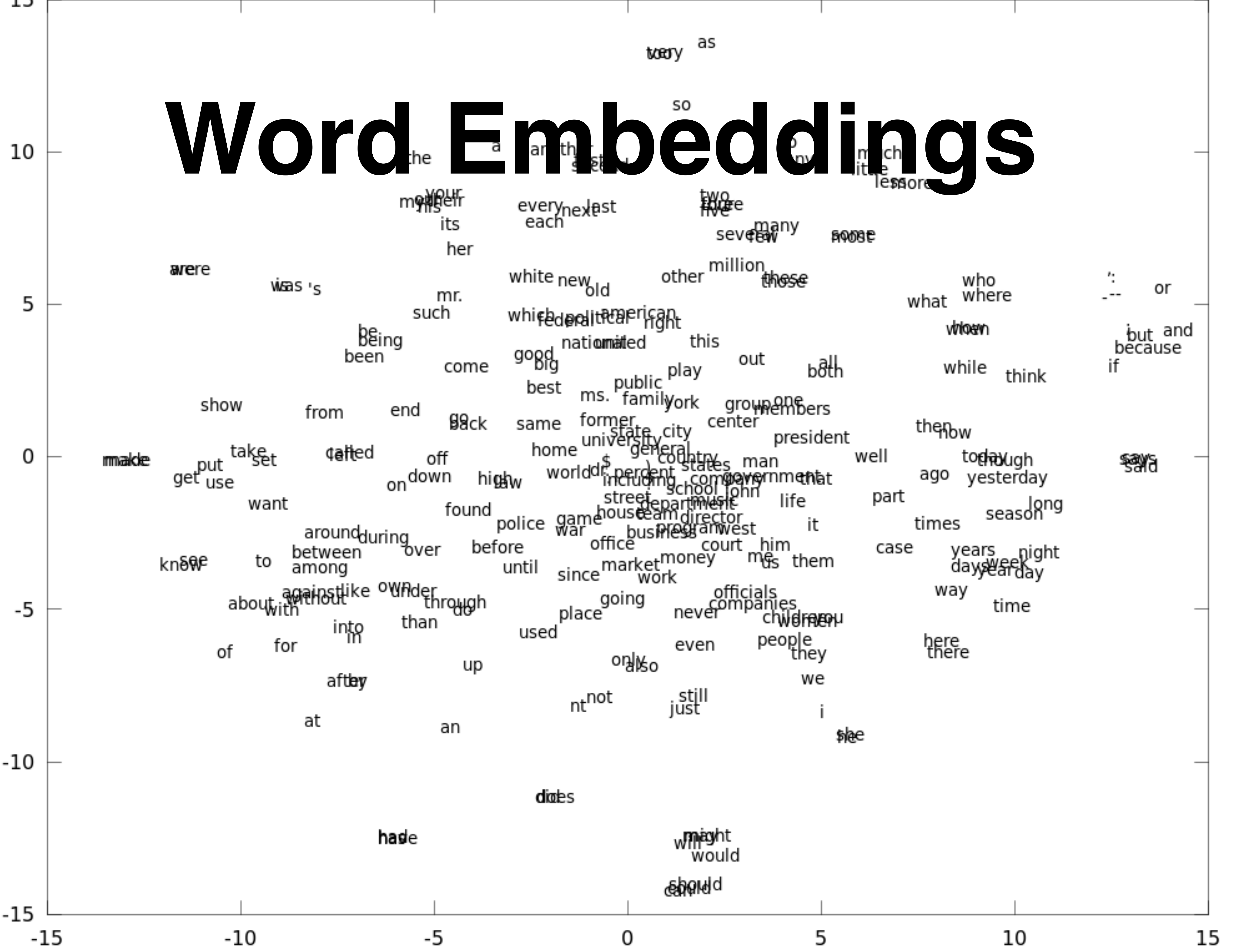
what is x?

Predicting from words

Neural **NLP** Building Blocks

- Word Embeddings: translate a word to a vector.
- Ways of combining vectors.

Word Embeddings



Word Embeddings

- Translate each word in the (fixed) vocabulary to a vector.
    - Typical dimensions: 100-300
    - Translation is done using a lookup table.
    - Can be "pre-trained" (word2vec, glove)
- Dealing with "infinite" vocabularies:
    - {characters}, {word pieces, bpe}, {fastText}

Word Embeddings

- {characters}, {word pieces, bpe}, {fastText}

dinosaur = d i n o s a u r

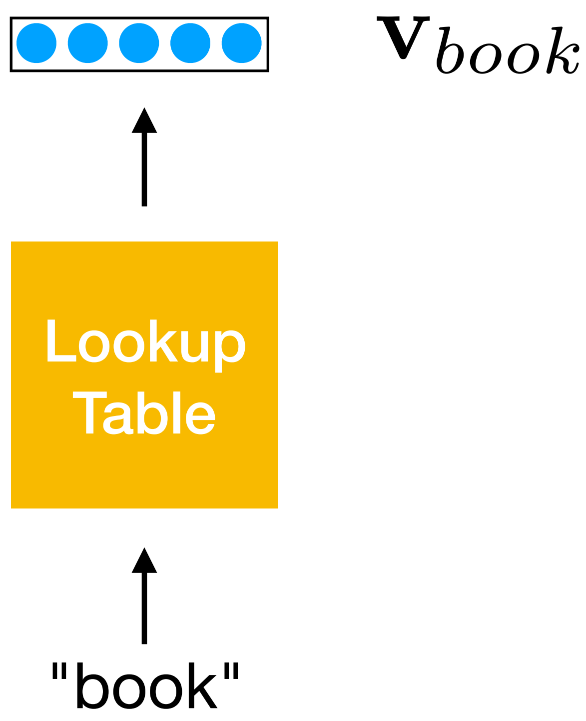
dinosaur = dino #sa #ur

dinosaur =

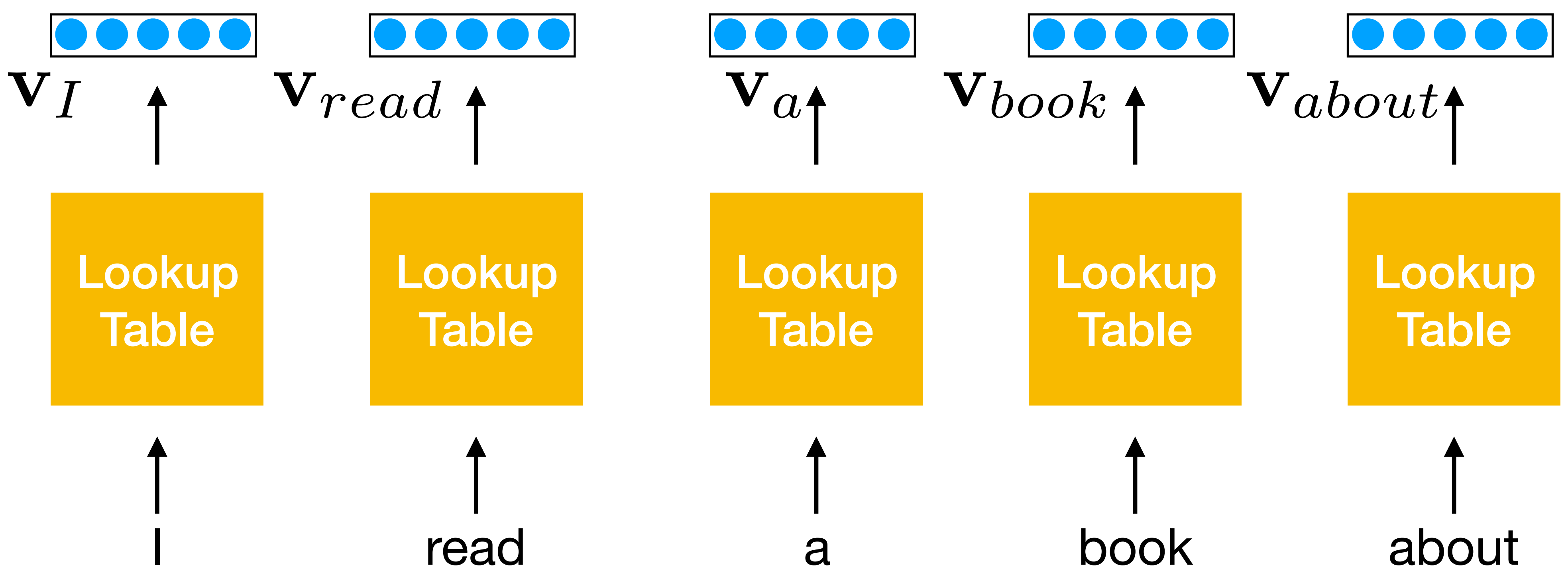
dinosa + inosau + nosaur +
dino + inos + nosa + osau + saur
+ din + ino + nos + osa + sau + aur

Word Embeddings

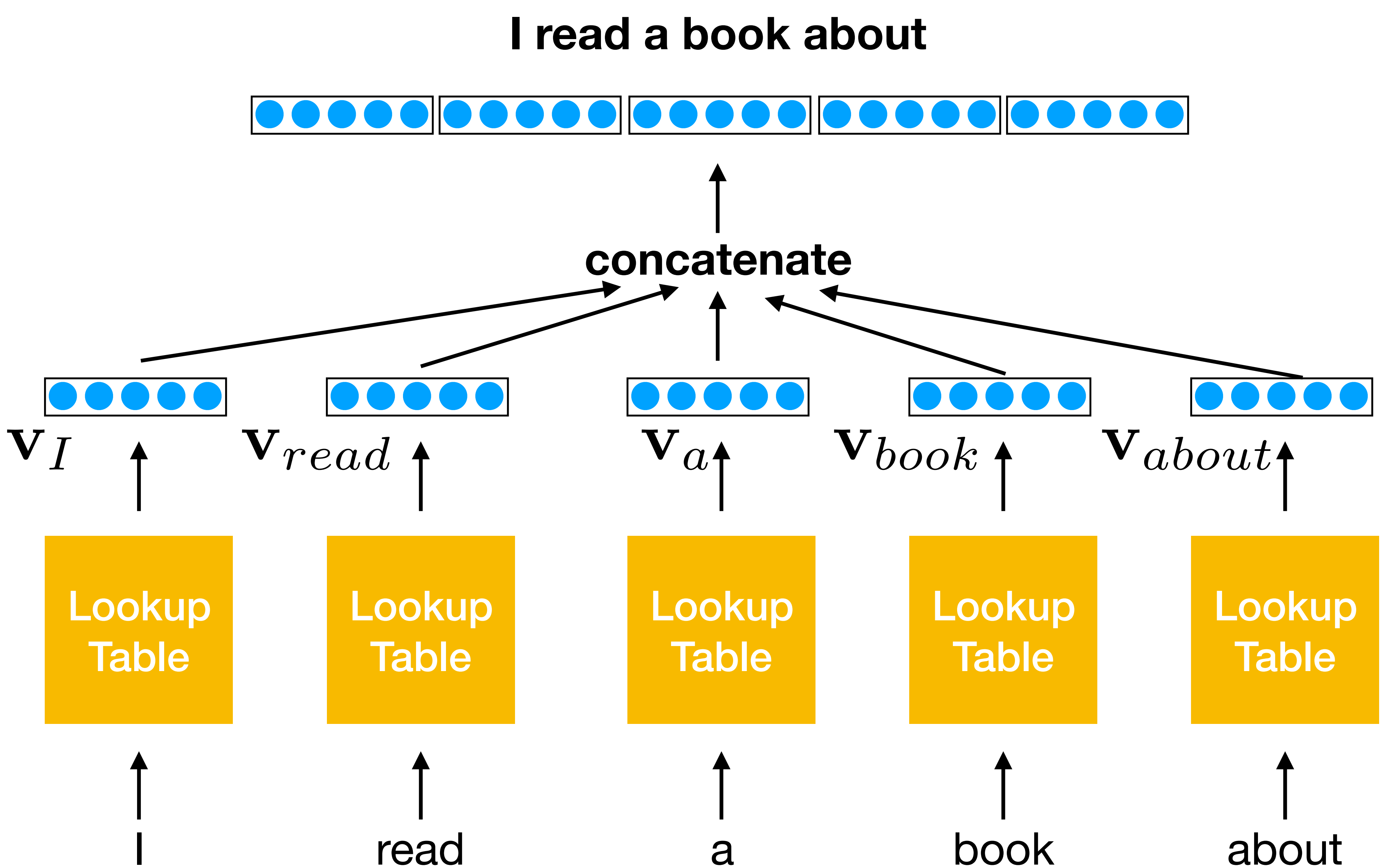
$$\mathbf{v}_{book} = \mathbf{E}[book]$$



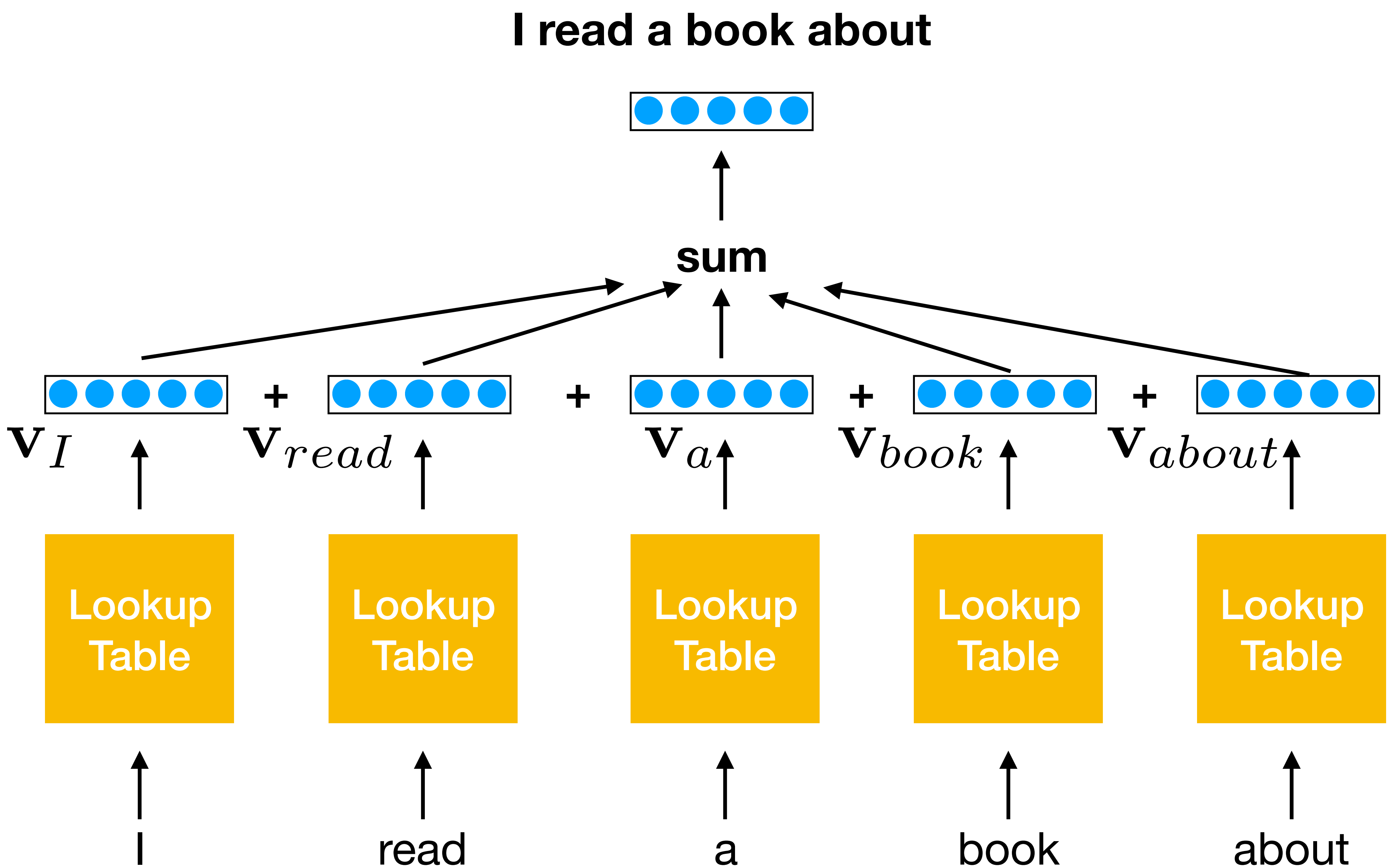
Combining Vectors



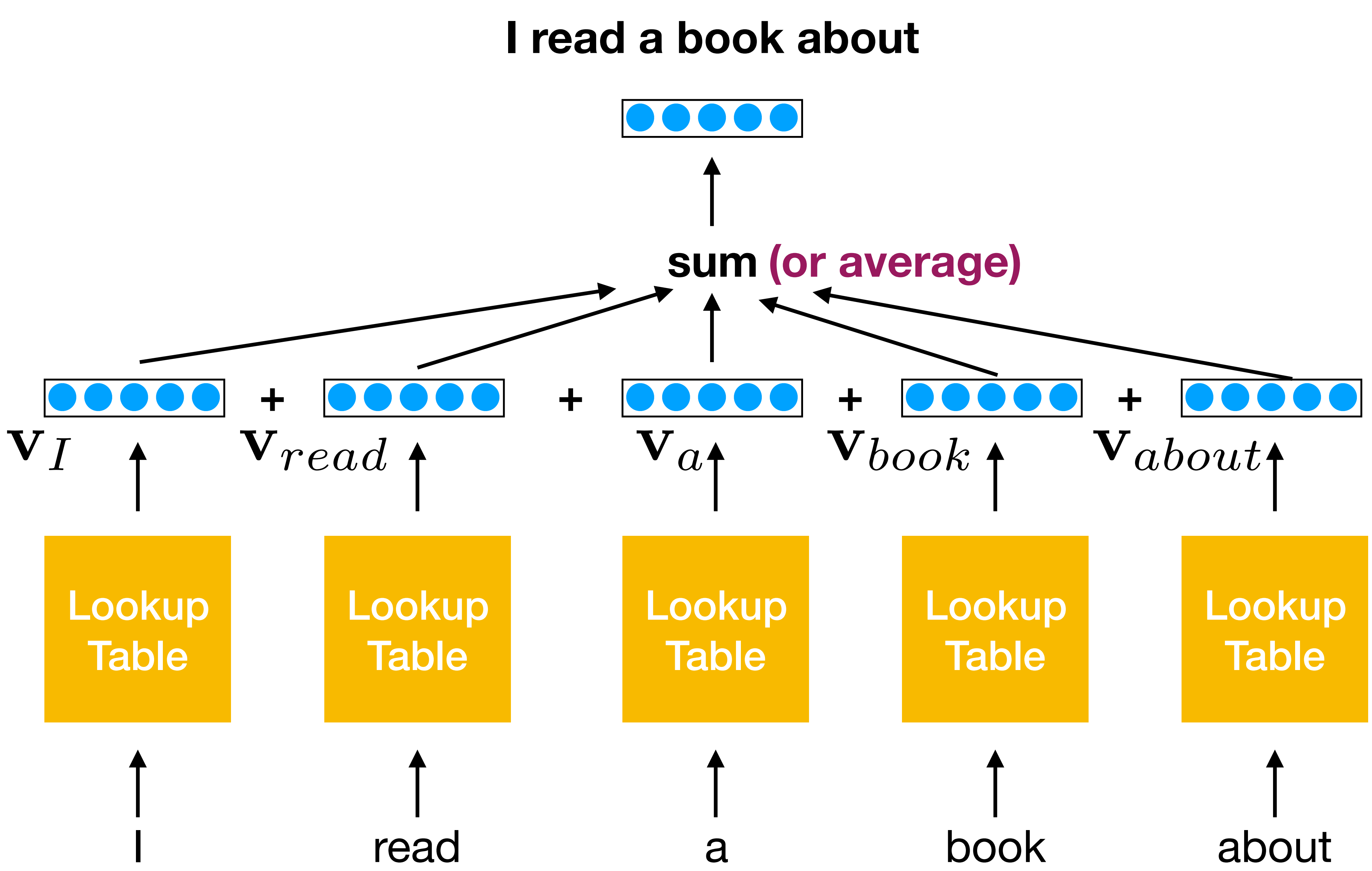
Combining Vectors



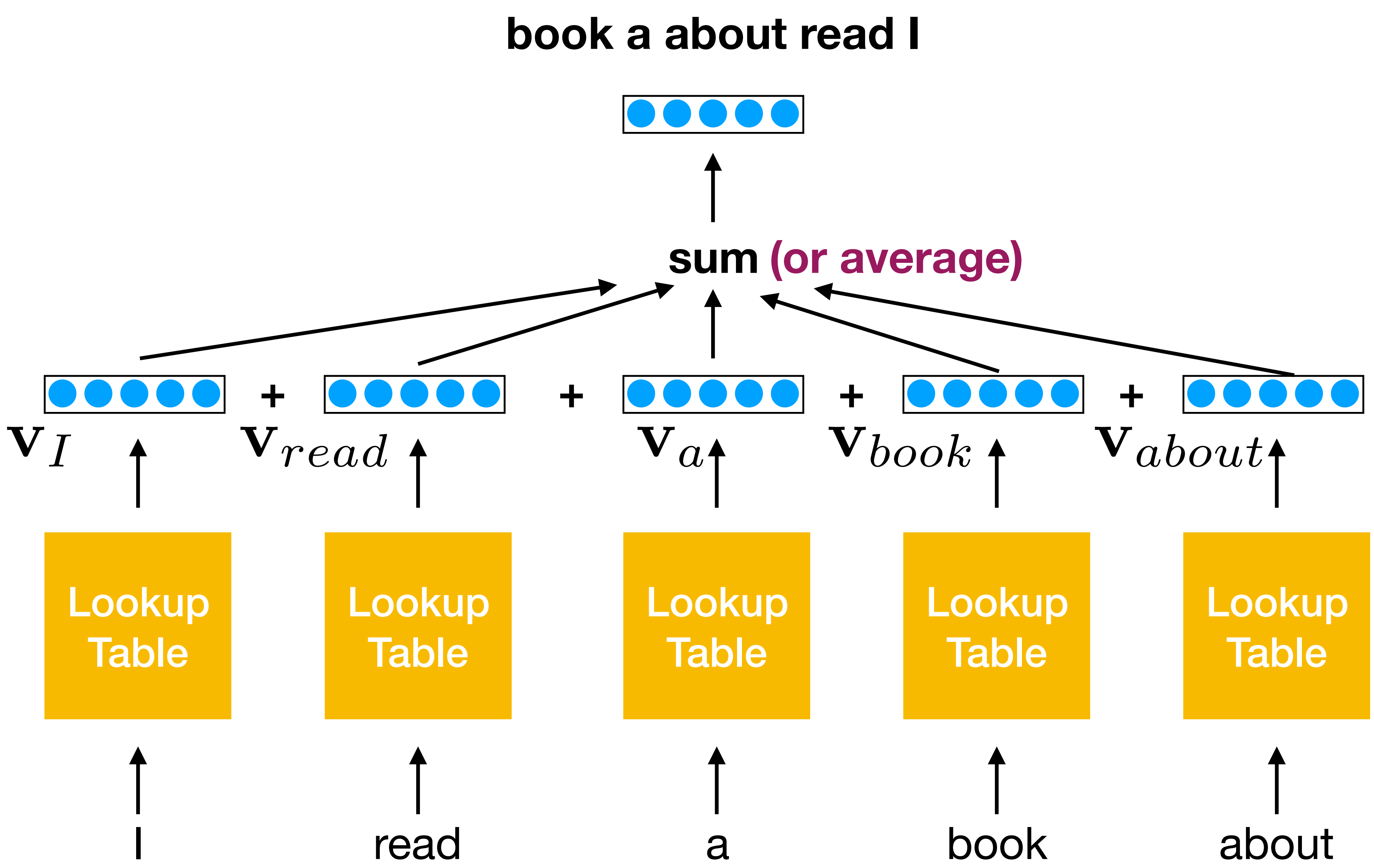
Combining Vectors



Combining Vectors



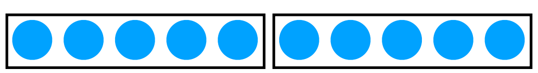
Combining Vectors



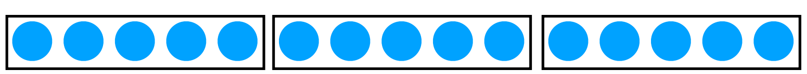
Combining Vectors

Concatenate

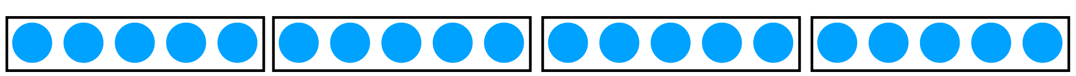
I read



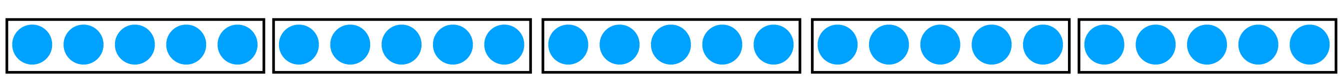
I read a



I read a book



I read a book about

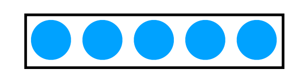


more words = longer vectors

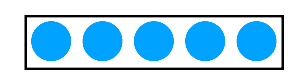
Sum (or average)

"cbow"

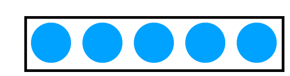
I read



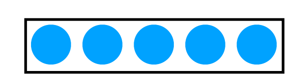
I read a



I read a book

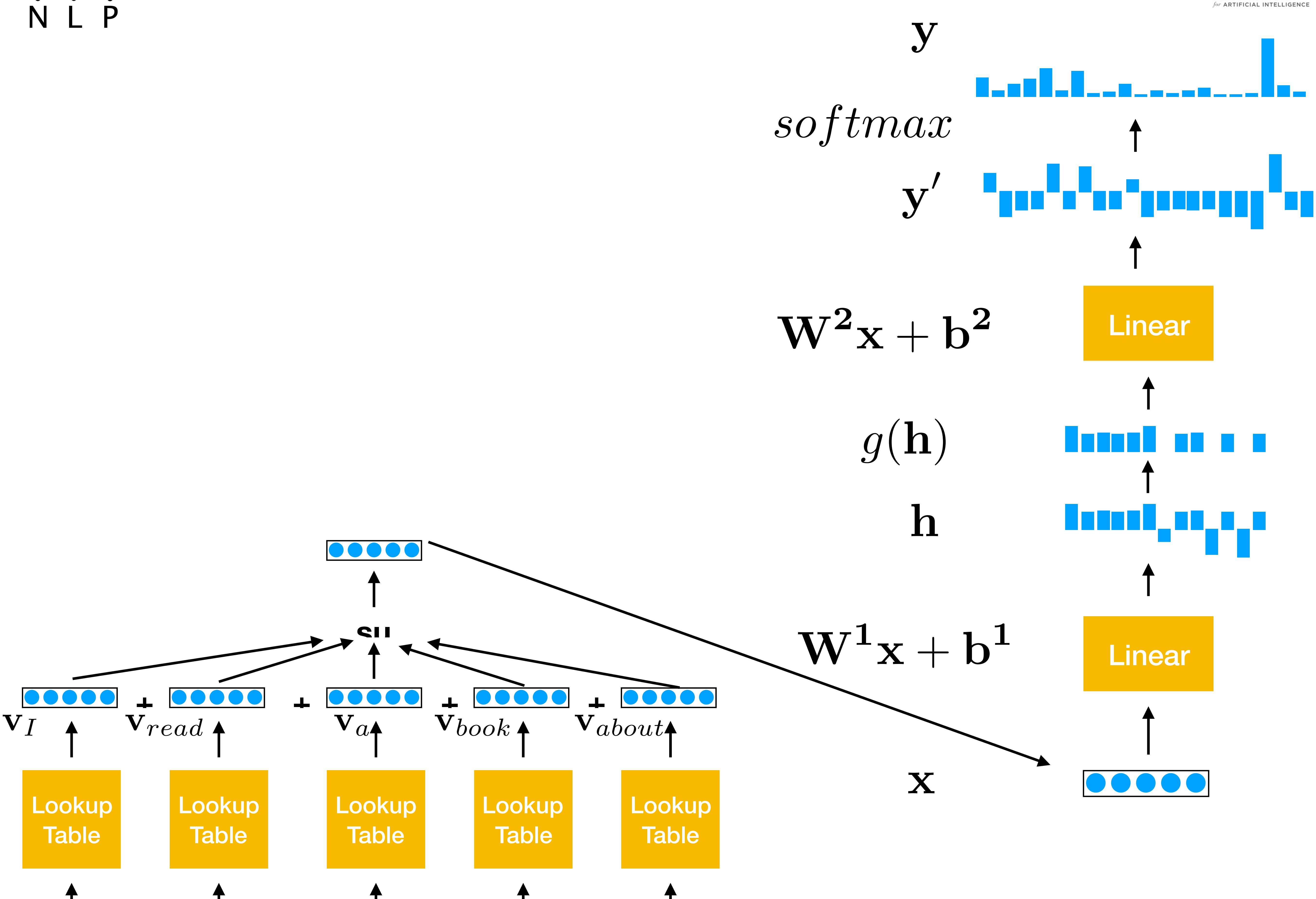


I read a book about



I book a read about
book about read I a
I a about book read
a read about book I
...

order invariant


$\mathbf{y}$
$softmax$
$\mathbf{y}'$
$\mathbf{W^2x + b^2}$
Linear
$g(\mathbf{h})$
$\mathbf{h}$
$\mathbf{W^1x + b^1}$
Linear
$\mathbf{x}$
$\mathbf{v}_I$ + $\mathbf{v}_{read}$ + $\mathbf{v}_a$ + $\mathbf{v}_{book}$ + $\mathbf{v}_{about}$
Lookup Table
Lookup Table
Lookup Table
Lookup Table
Lookup Table

The Computation Graph

Gradient-based training

- Computing the gradients:
  - The network (and loss calculation) is a mathematical function.

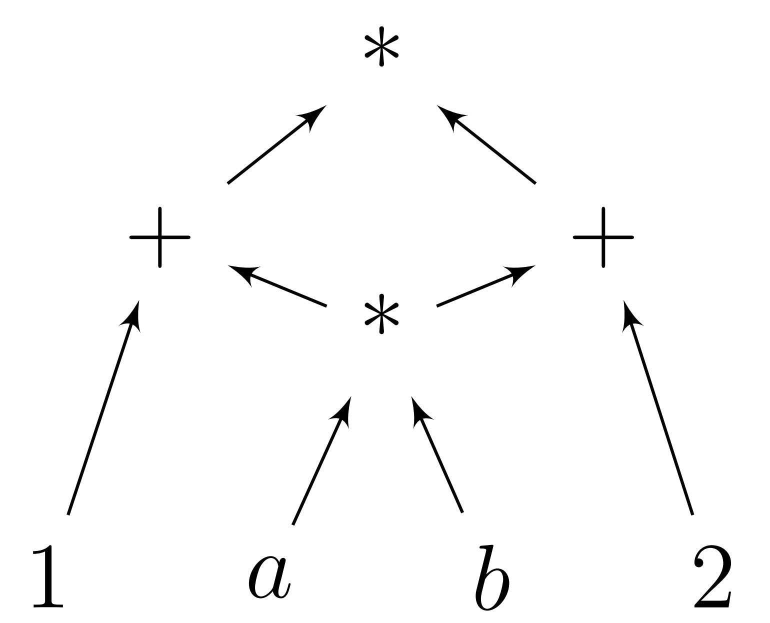
$$\ell(x, k) = -log(softmax(\mathbf{W}^3 g^2(\mathbf{W}^2 g^1(\mathbf{W}^1 x + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)[k])$$

  - Calculus rules apply.
  - (a bit hairy, but carefully follow the chain rule and you'll get there)

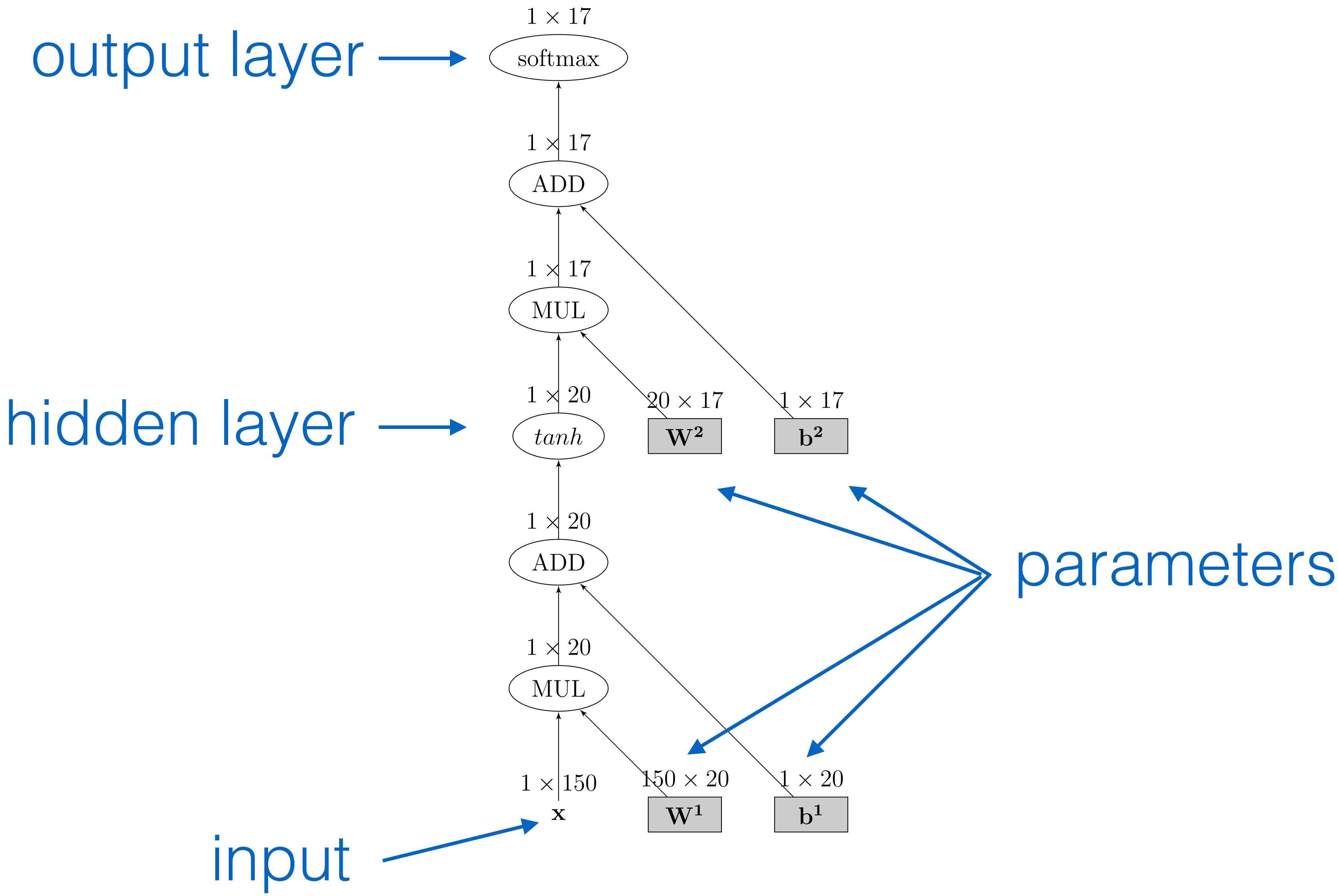
The Computation Graph (CG)

- a DAG.
- Leafs are inputs (or parameters).
- Nodes are operators (functions).
- Edges are results (values).
- Can be built for any function.

$$(a * b + 1) * (a * b + 2)$$

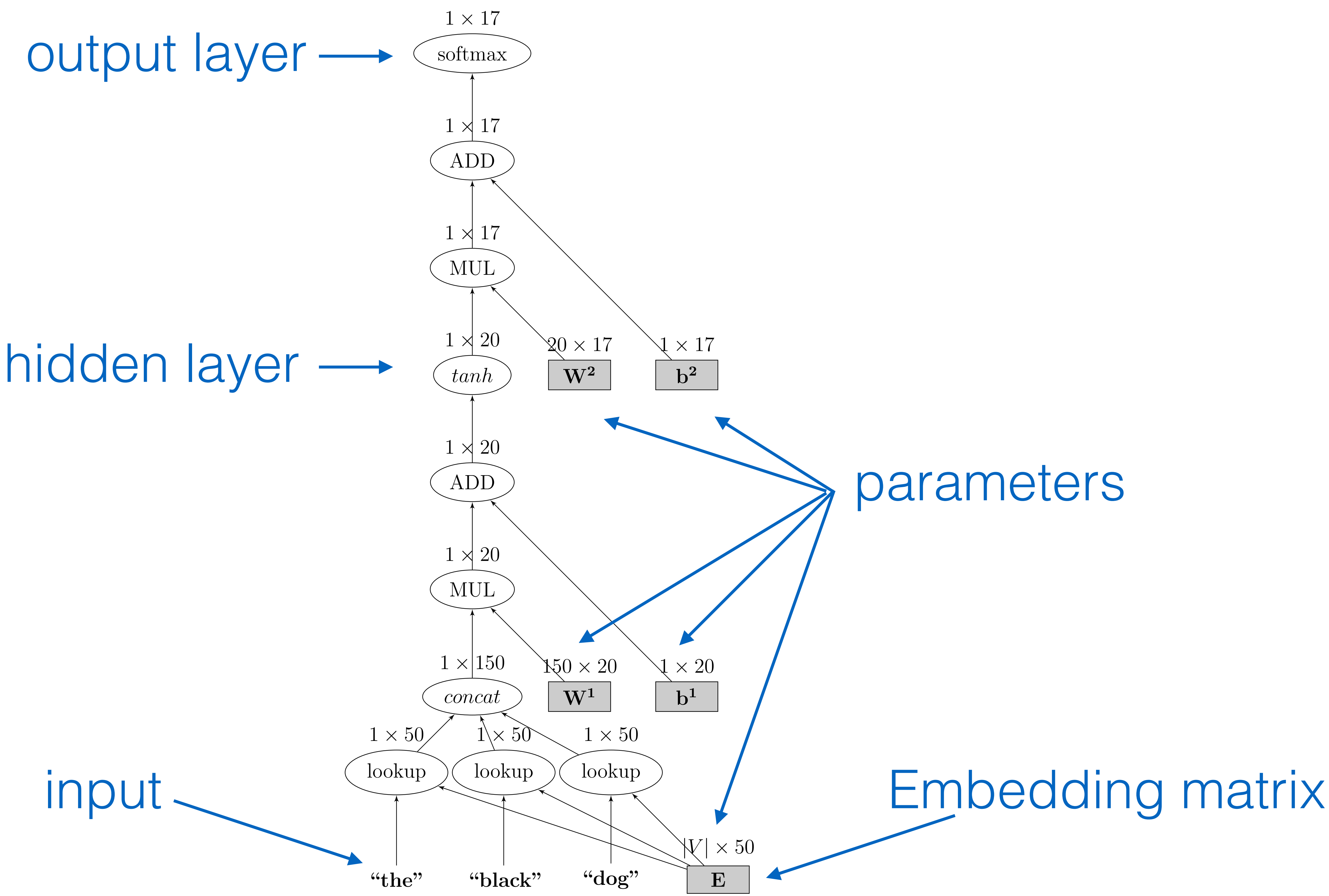


MLP_1

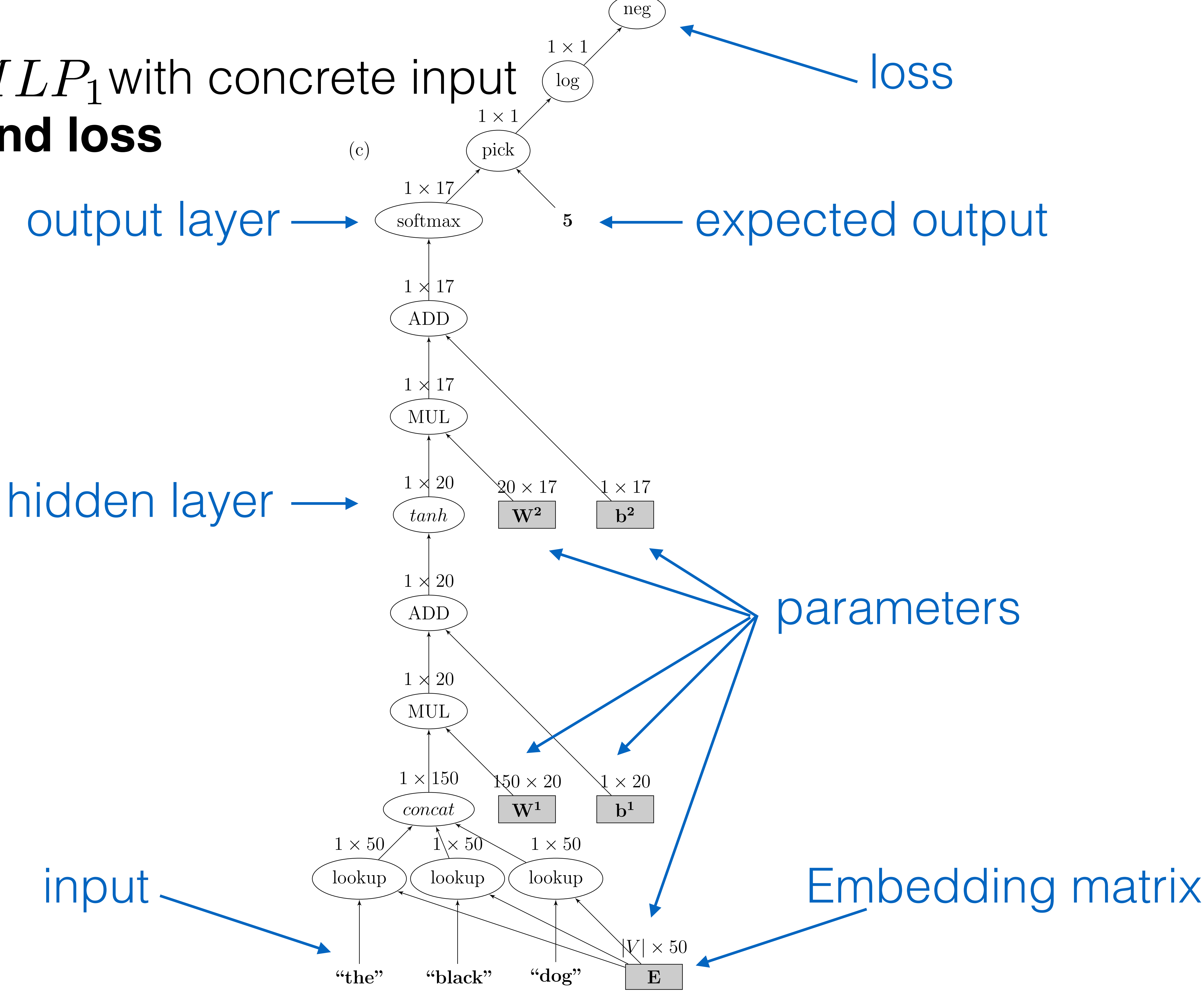


output layer
hidden layer
input
parameters
1 × 17
softmax
1 × 17
ADD
1 × 17
MUL
1 × 20
tanh
20 × 17
W²
1 × 17
b²
1 × 20
ADD
1 × 20
MUL
1 × 150
x
150 × 20
W¹
1 × 20
b¹

MLP_1 with concrete input



MLP_1 with concrete input **and loss**

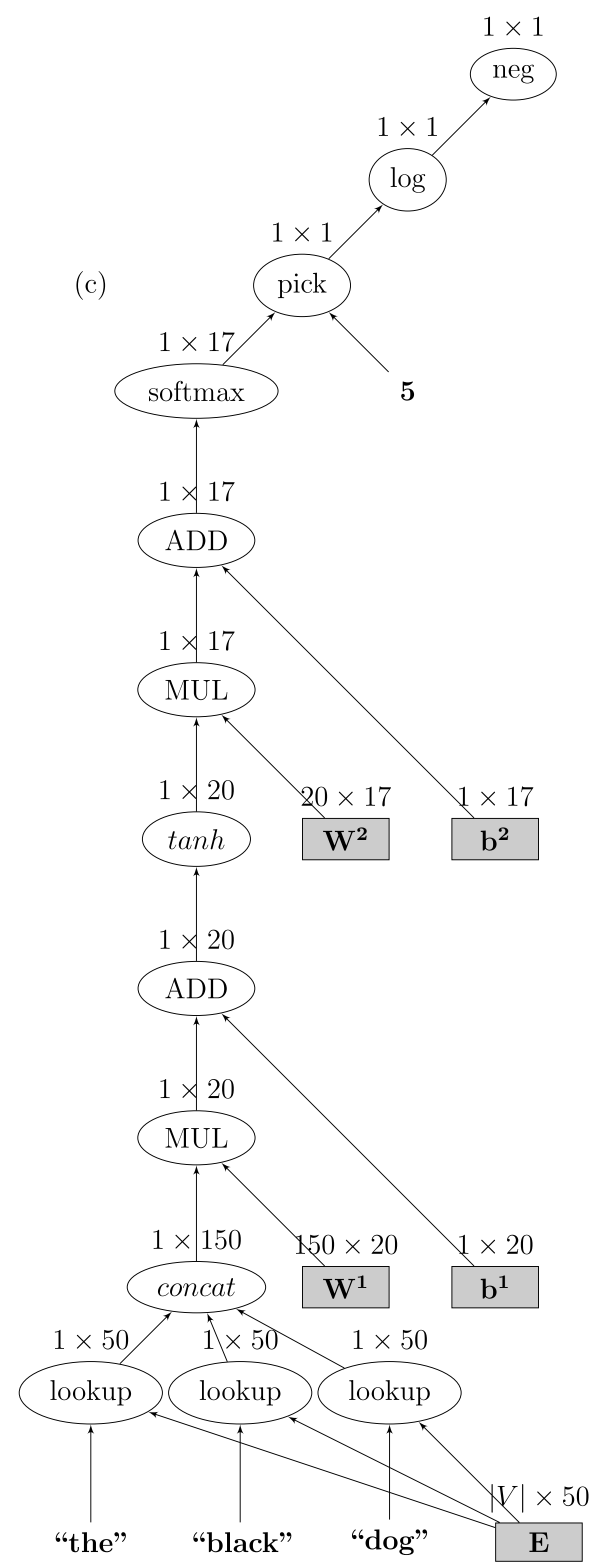


- Create a graph for each training example.

- Once graph is built, we have two essential algorithms:

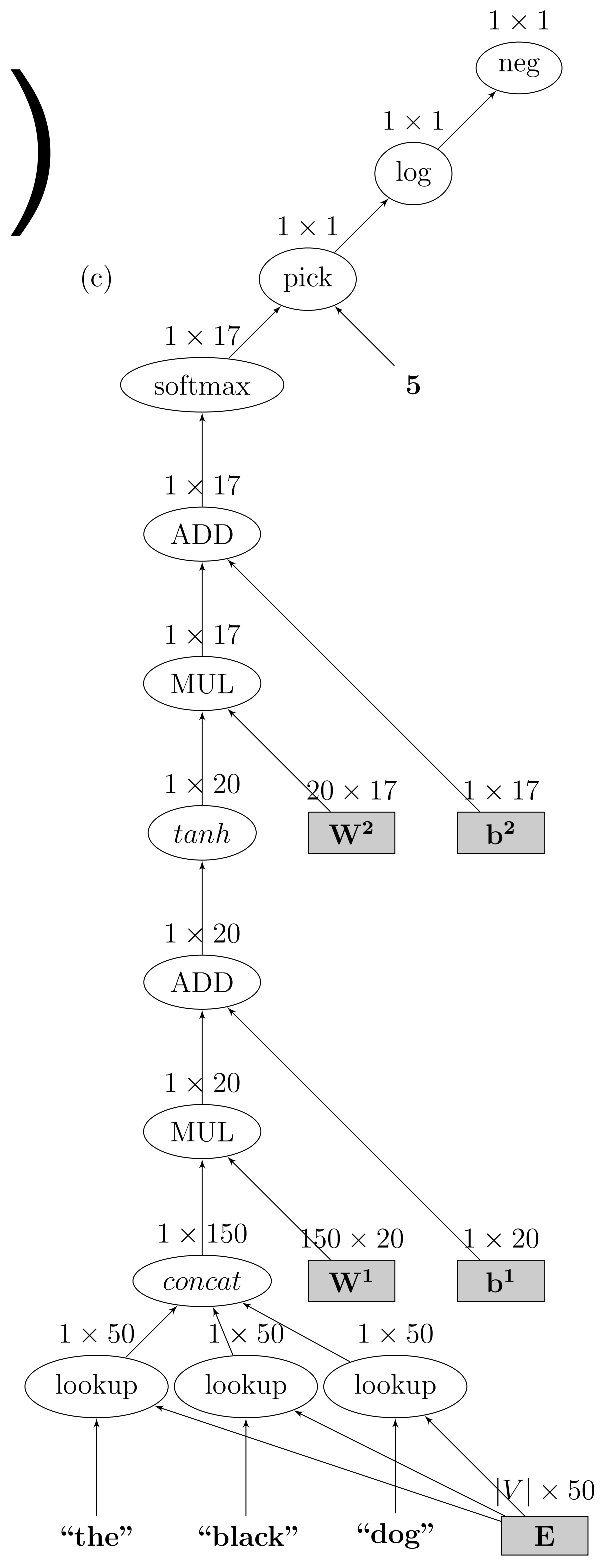
  - **Forward:**
    compute all values.

  - **Backward (backprop):**
    compute all gradients.



Computing the Gradients (backprop)

- Consider the chain-rule (example on blackboard)
- Each node needs to know how to:
  - Compute forward.
  - Compute its **local** gradient.



The Python Neural Networks Toolkits Landscape (partial)

theano

TensorFlow

K

Chainer

dy/net

PYTORCH

The Python Neural Networks Toolkits Landscape (partial)

high-level

theano

TensorFlow

K

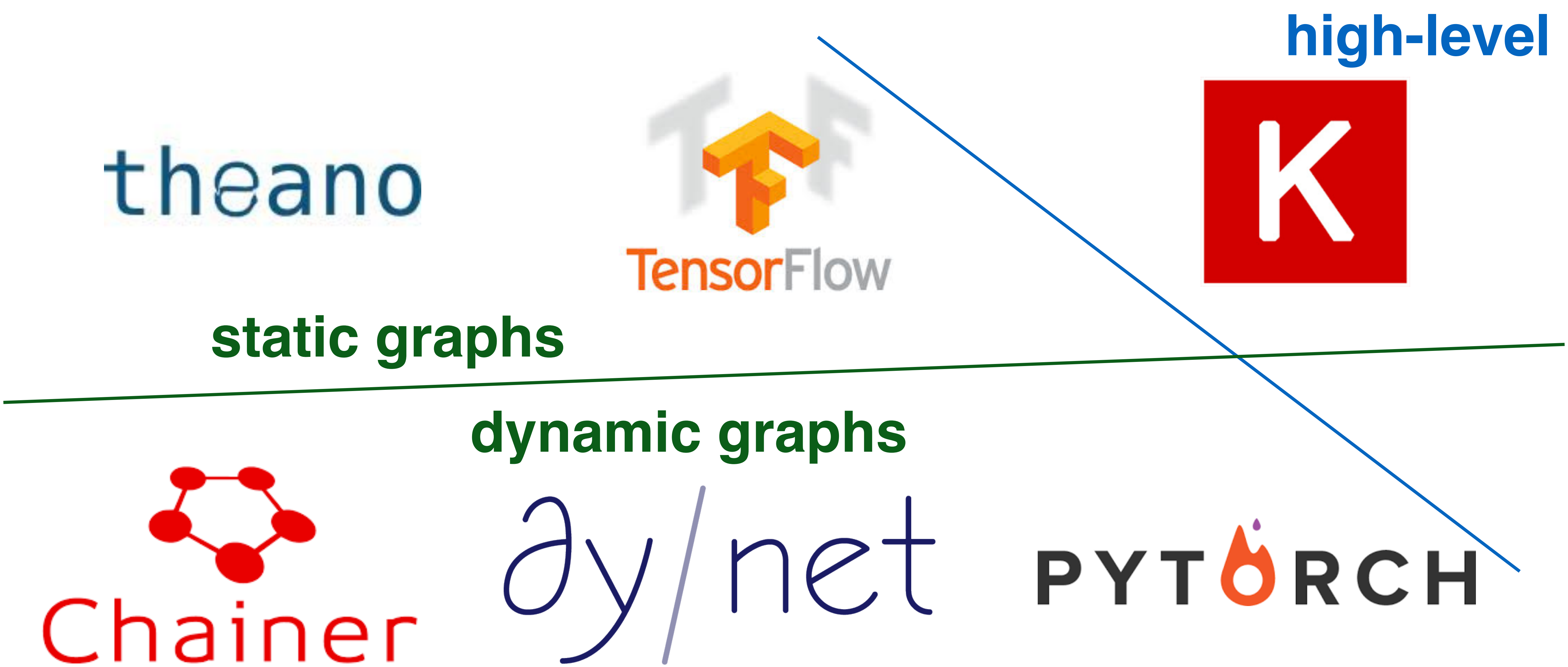
low-level

Chainer

dy/net

PYTORCH

The Python Neural Networks Toolkits Landscape (partial)



The Python Neural Networks Toolkits Landscape (partial)

high-level

theano

TensorFlow

K

static graphs

dynamic graphs

Chainer

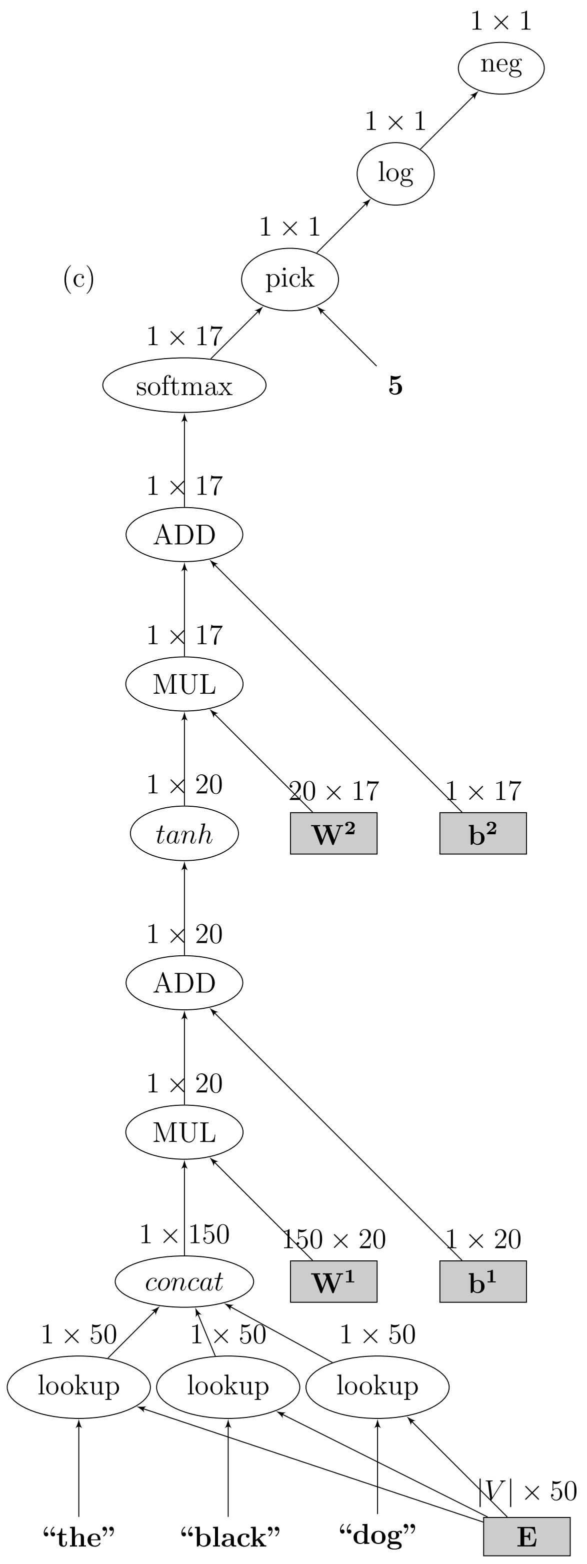
dynet

PYTORCH

- fast also on CPU
- automatic batching

Network Training algorithm:

- For each training example (or mini-batch):
  - Create graph for computing loss.
  - Compute loss (**forward**).
  - Compute gradients (**backwards**).
  - Update model parameters.



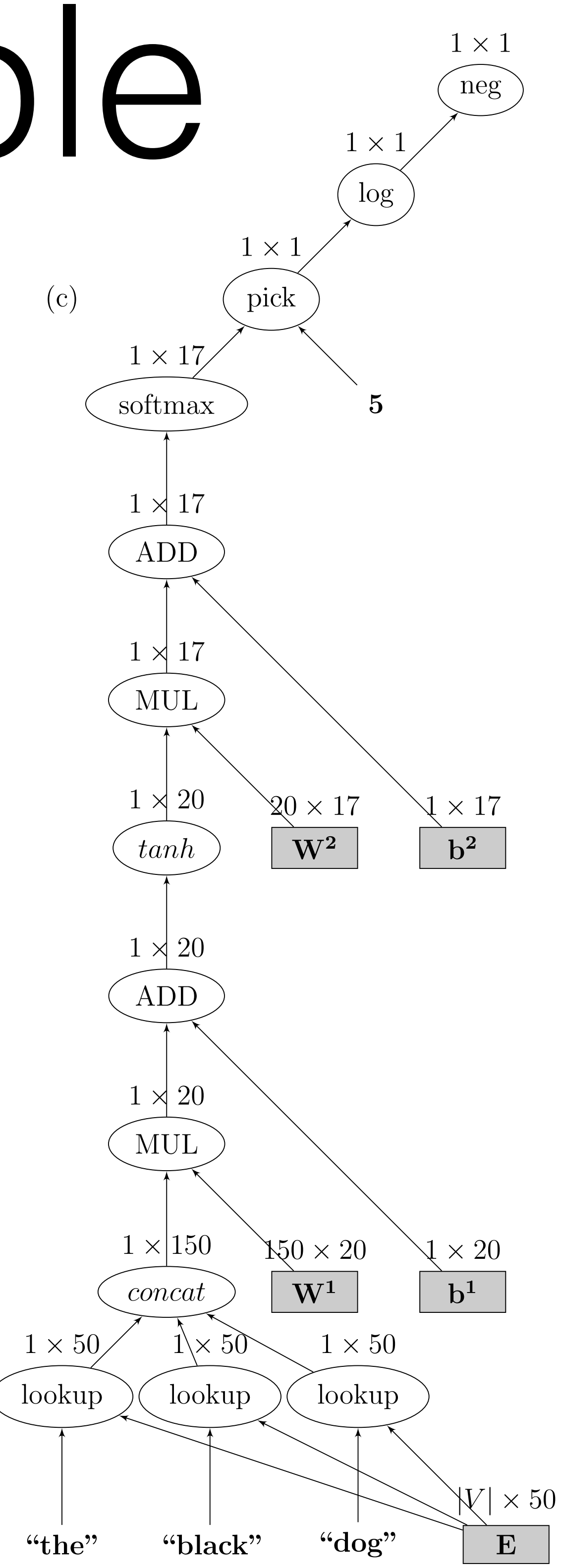
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



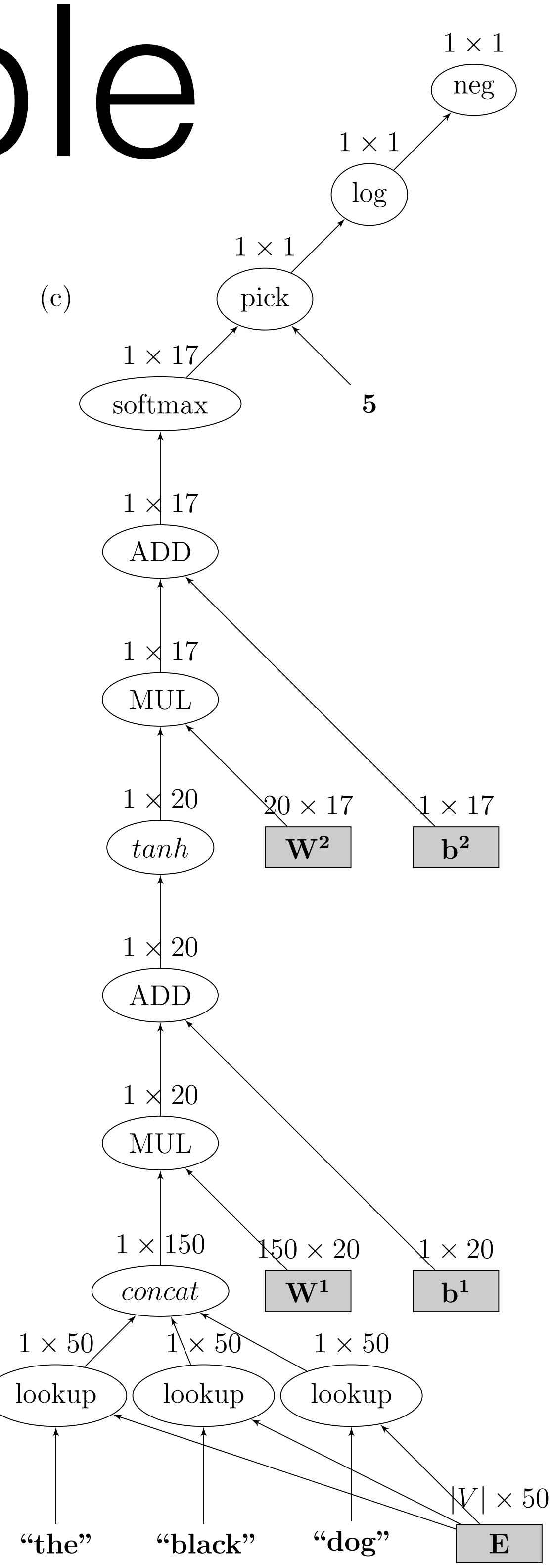
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



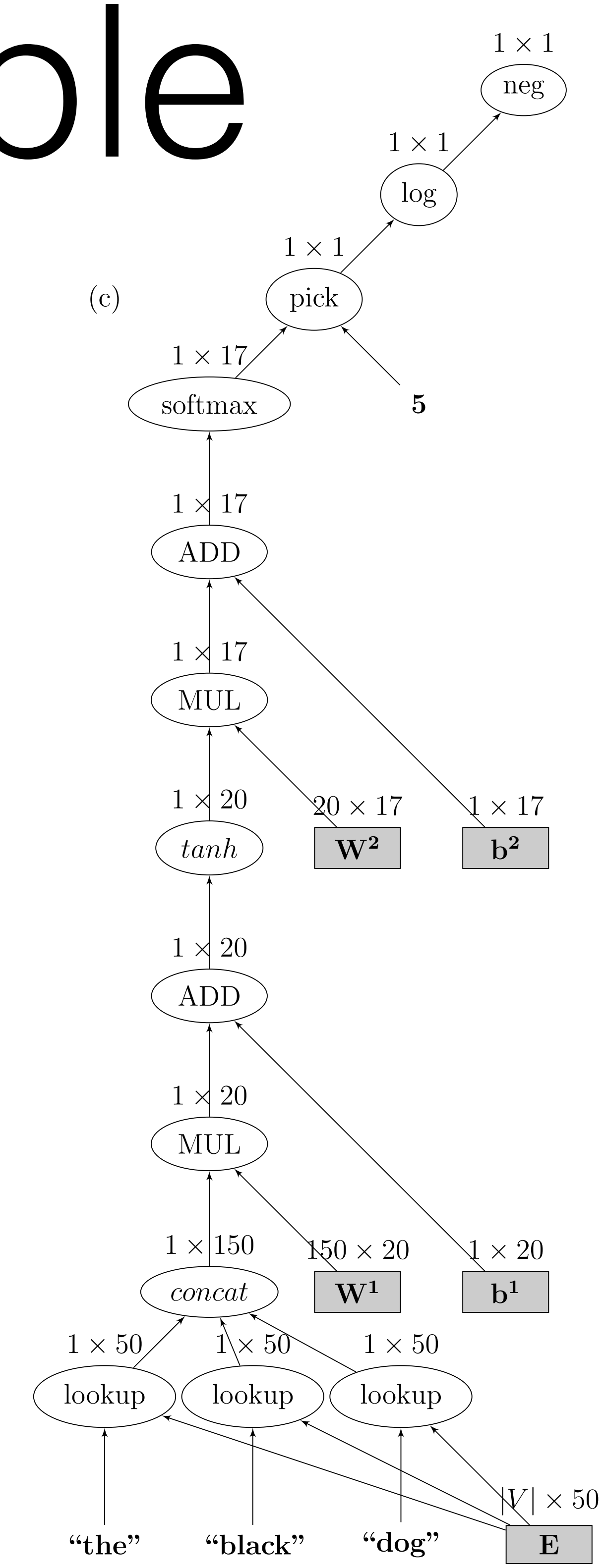
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



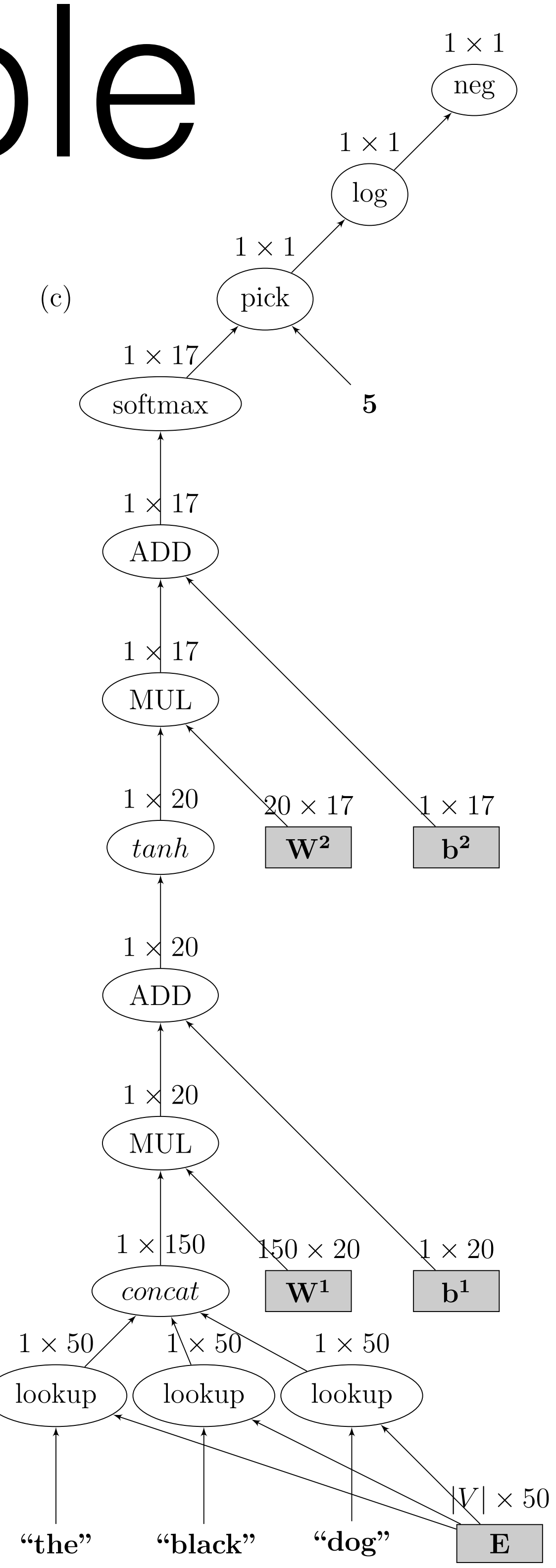
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



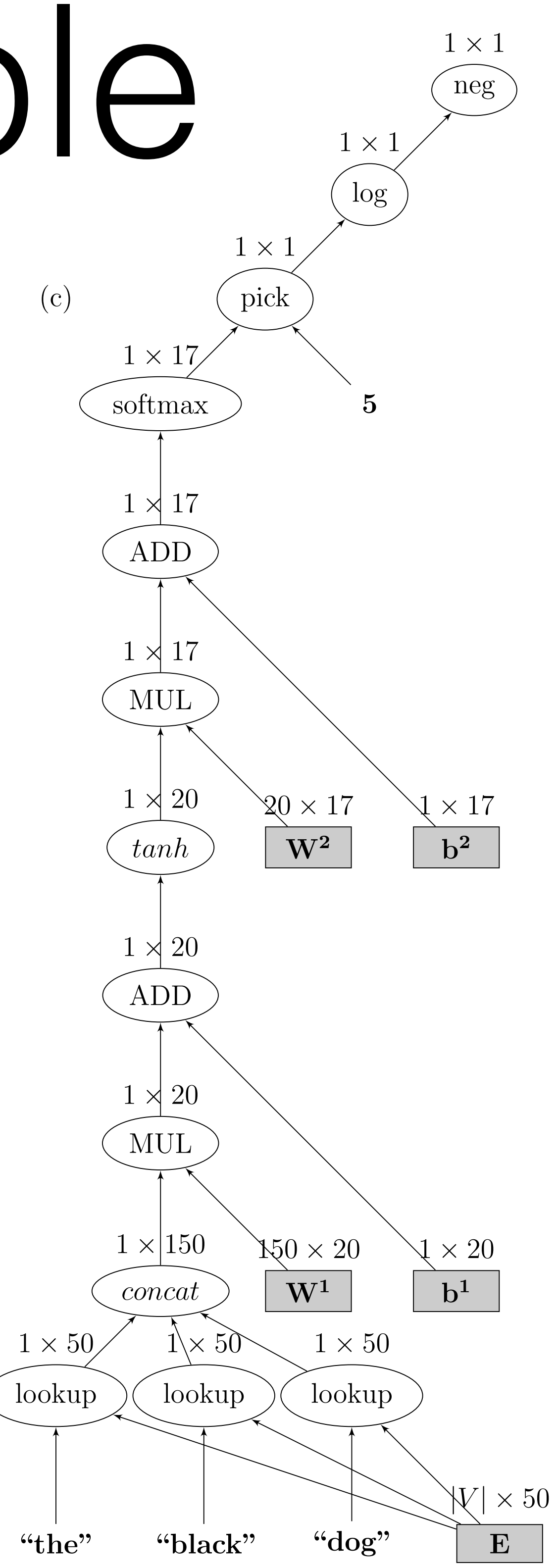
DyNet Example

```
# model initialization.
model = Model()
mW1 = model.add_parameters((20,150))
mb1 = model.add_parameters(20)
mW2 = model.add_parameters((17,20))
mb2 = model.add_parameters(17)
lookup = model.add_lookup_parameters((100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(mW1)
b1 = parameter(mb1)
W2 = parameter(mW2)
b2 = parameter(mb2)
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup[get_index("the")]
vblack = lookup[get_index("black")]
vdog   = lookup[get_index("dog")]

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))
```

```
loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```



BIU NLP

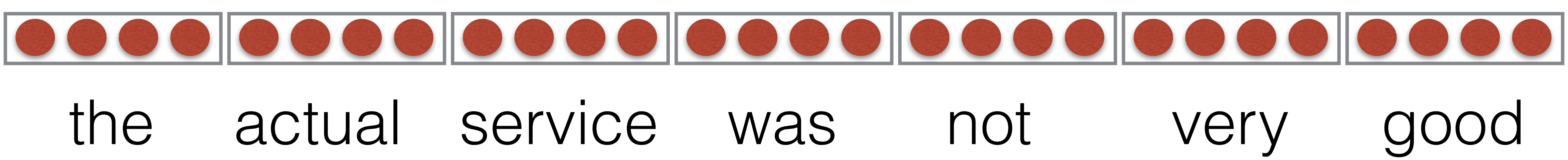
AI2
ALLEN INSTITUTE for ARTIFICIAL INTELLIGENCE

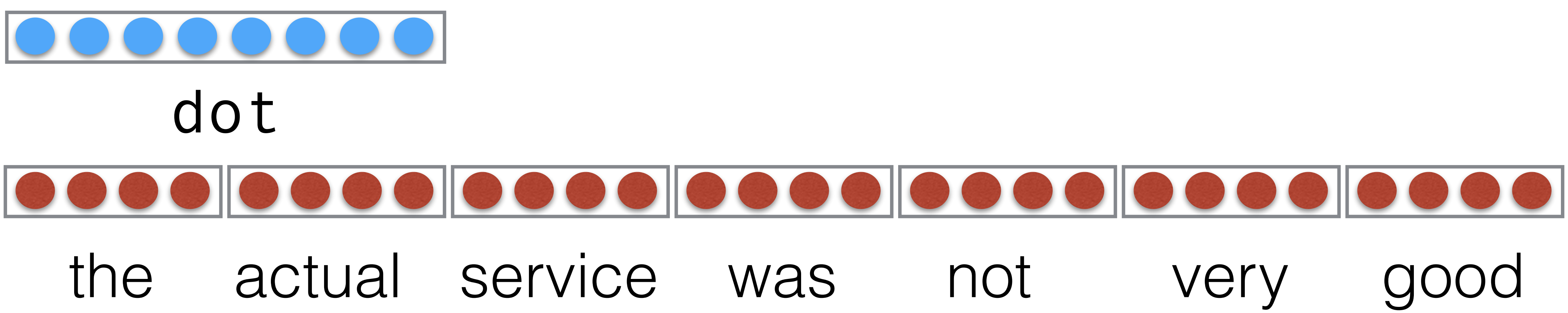
Back to Combining Vectors

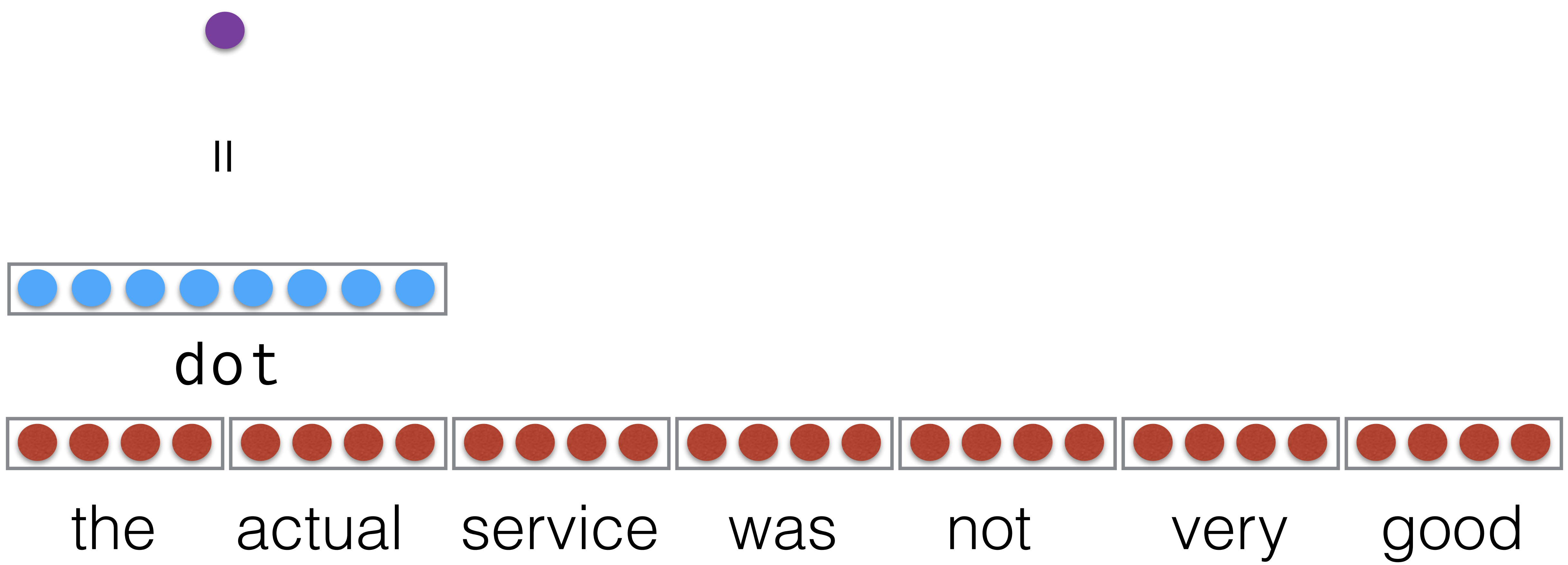
ConvNets

- "bags of ngrams".
- Useful!

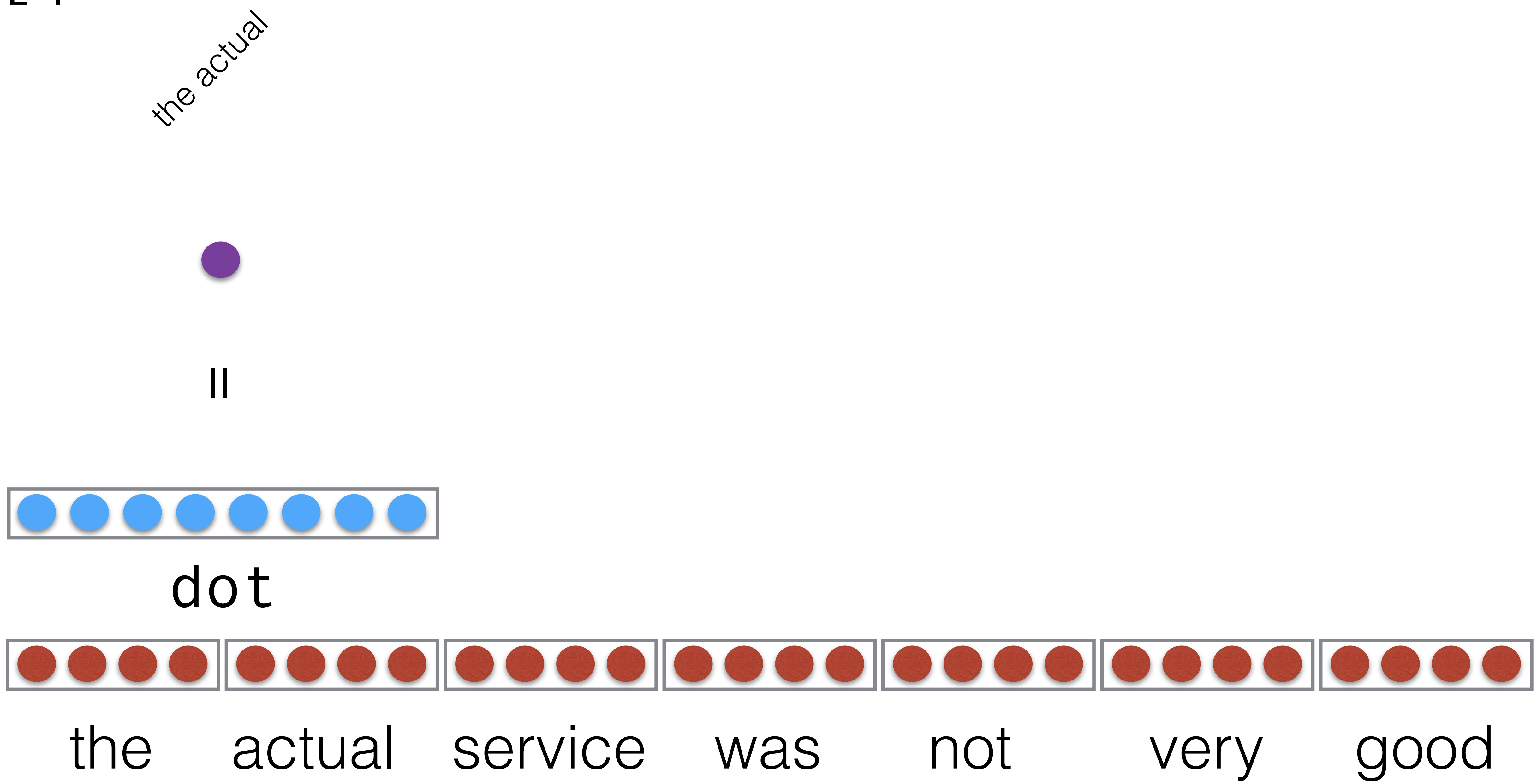
(we'll probably skip them today)

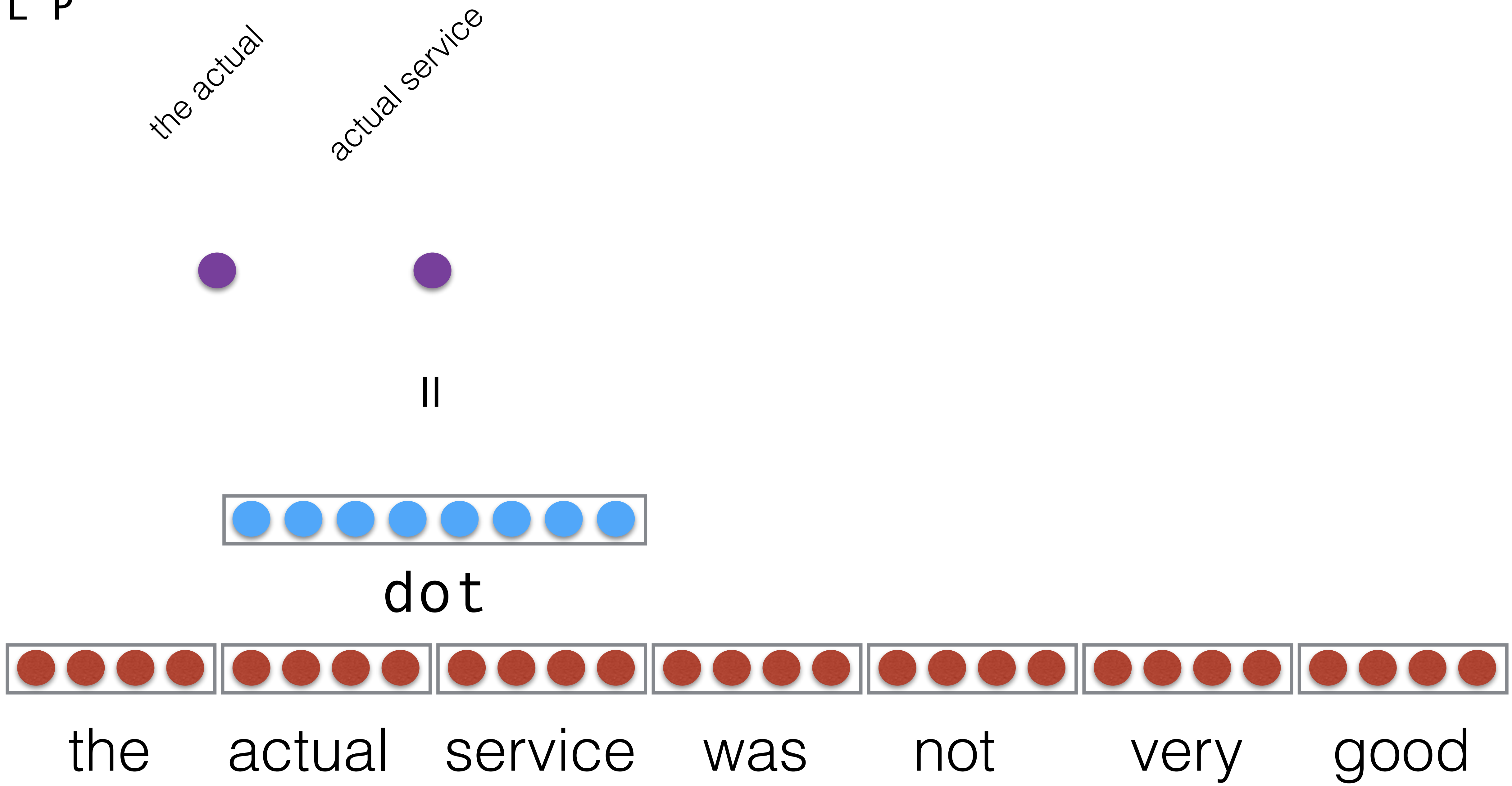

the
actual
service
was
not
very
good


dot
the
actual
service
was
not
very
good

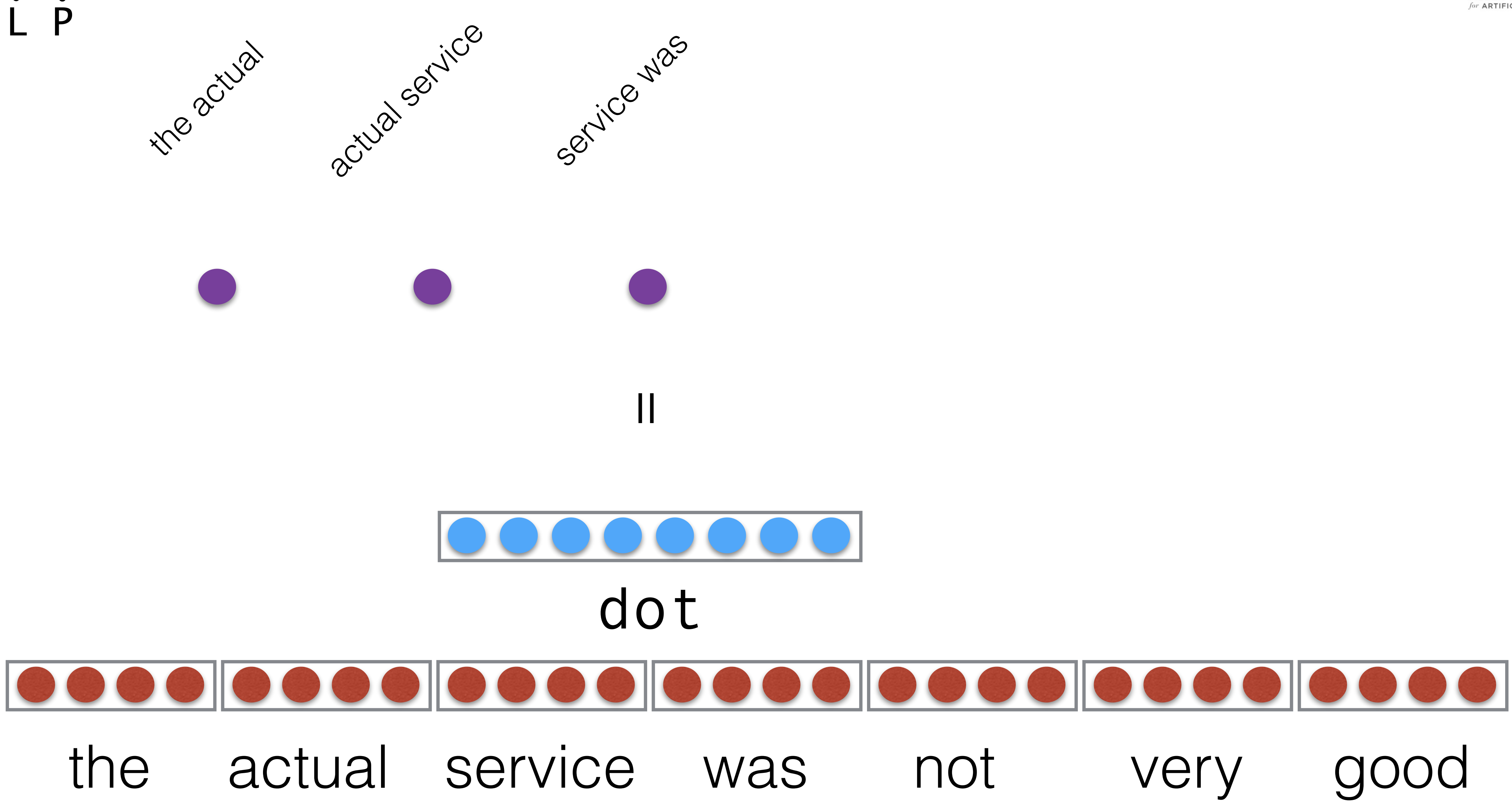


=
dot
the
actual
service
was
not
very
good

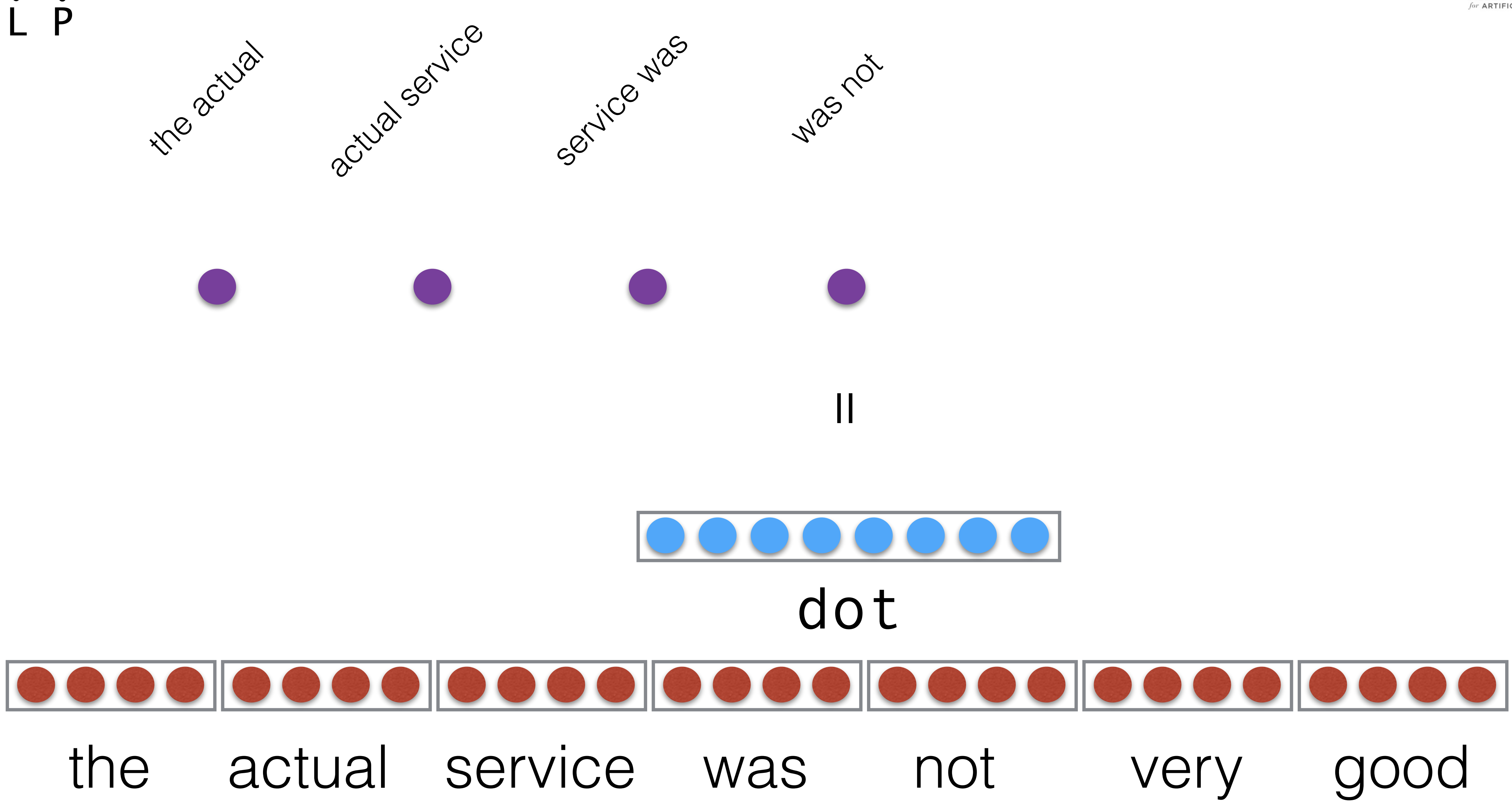

the actual
=
dot
the
actual
service
was
not
very
good

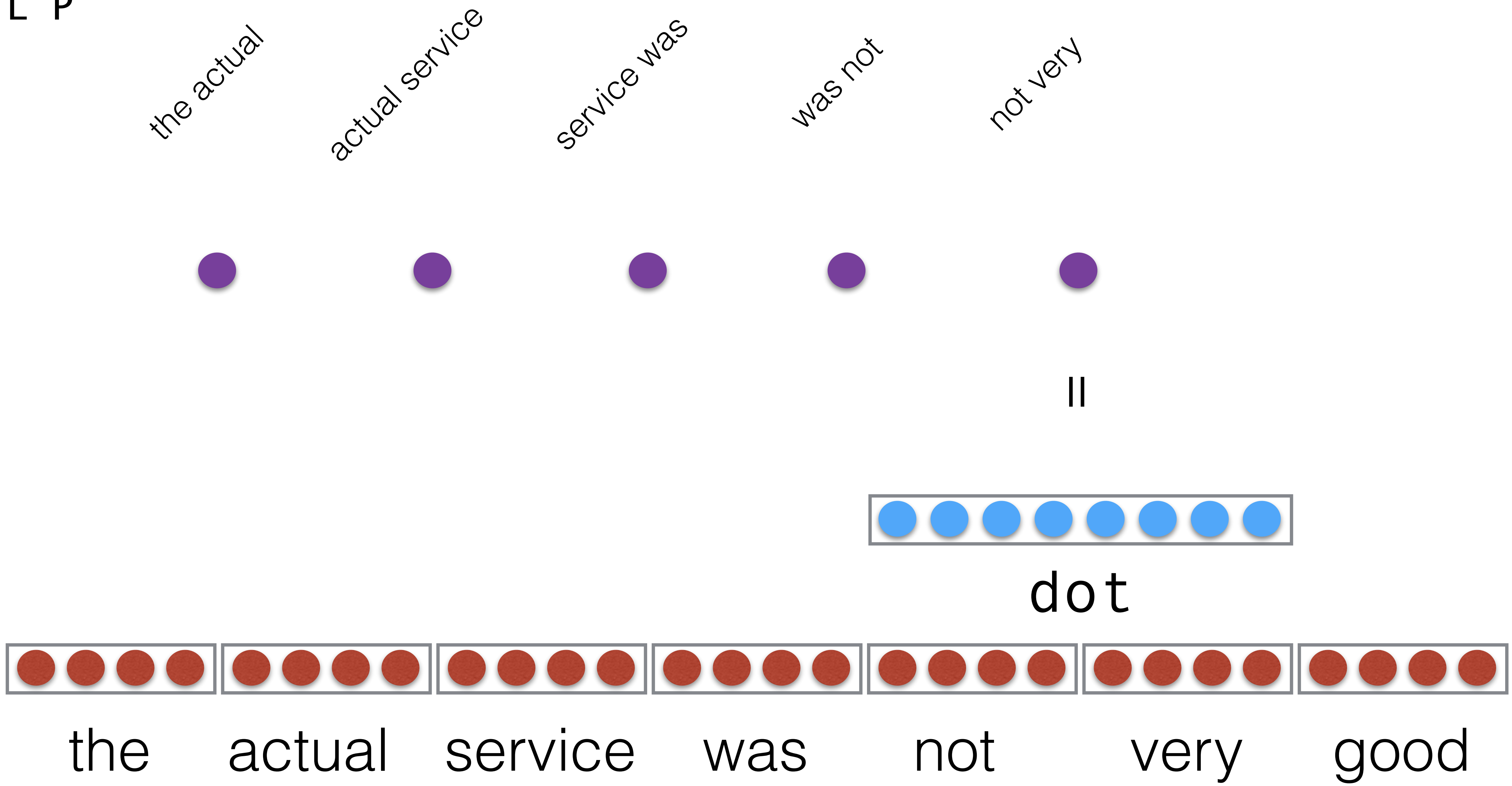


the actual
actual service
=
dot
the
actual
service
was
not
very
good

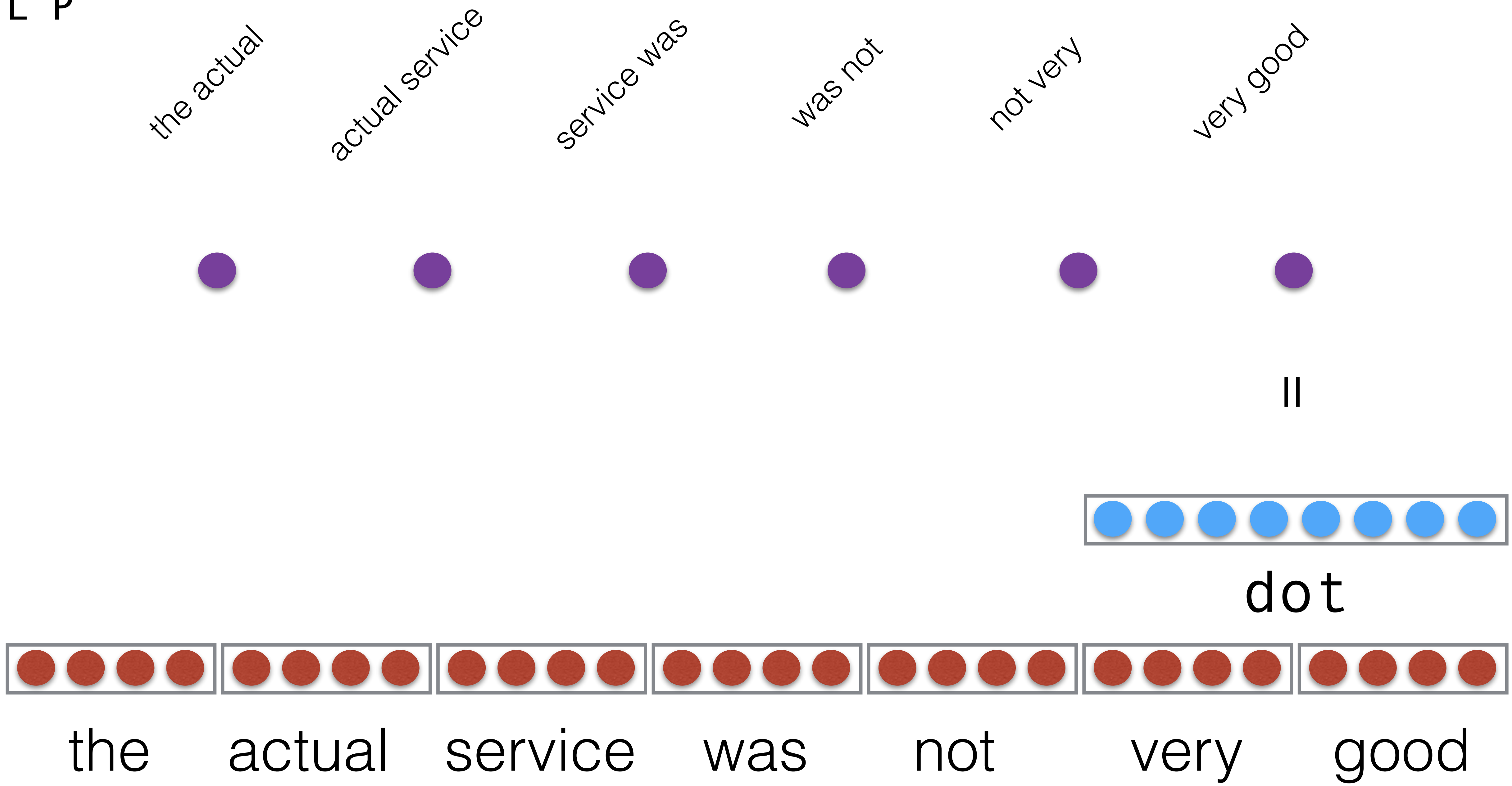


the actual
actual service
service was
=
dot
the
actual
service
was
not
very
good

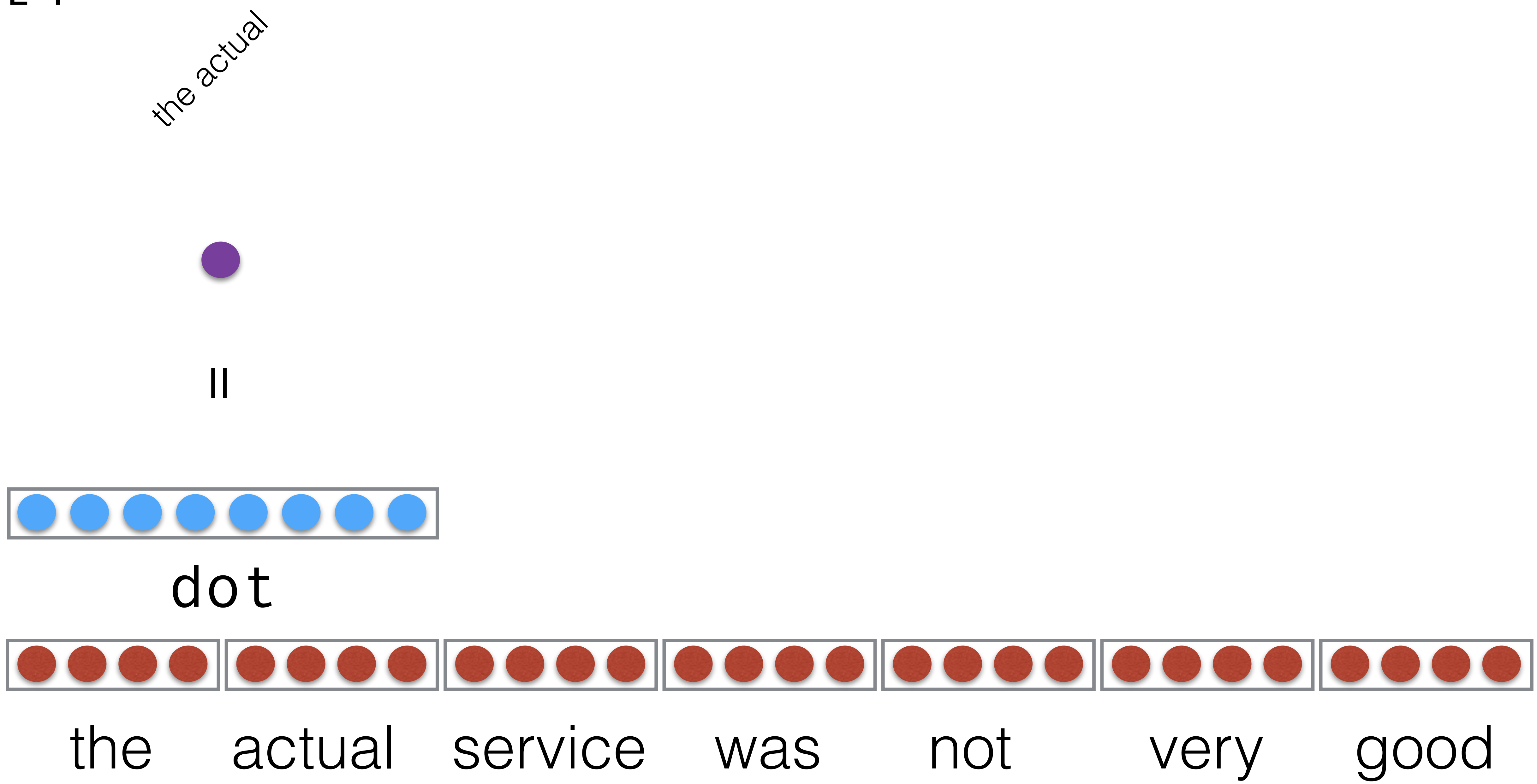

the actual
actual service
service was
was not
=
dot
the
actual
service
was
not
very
good



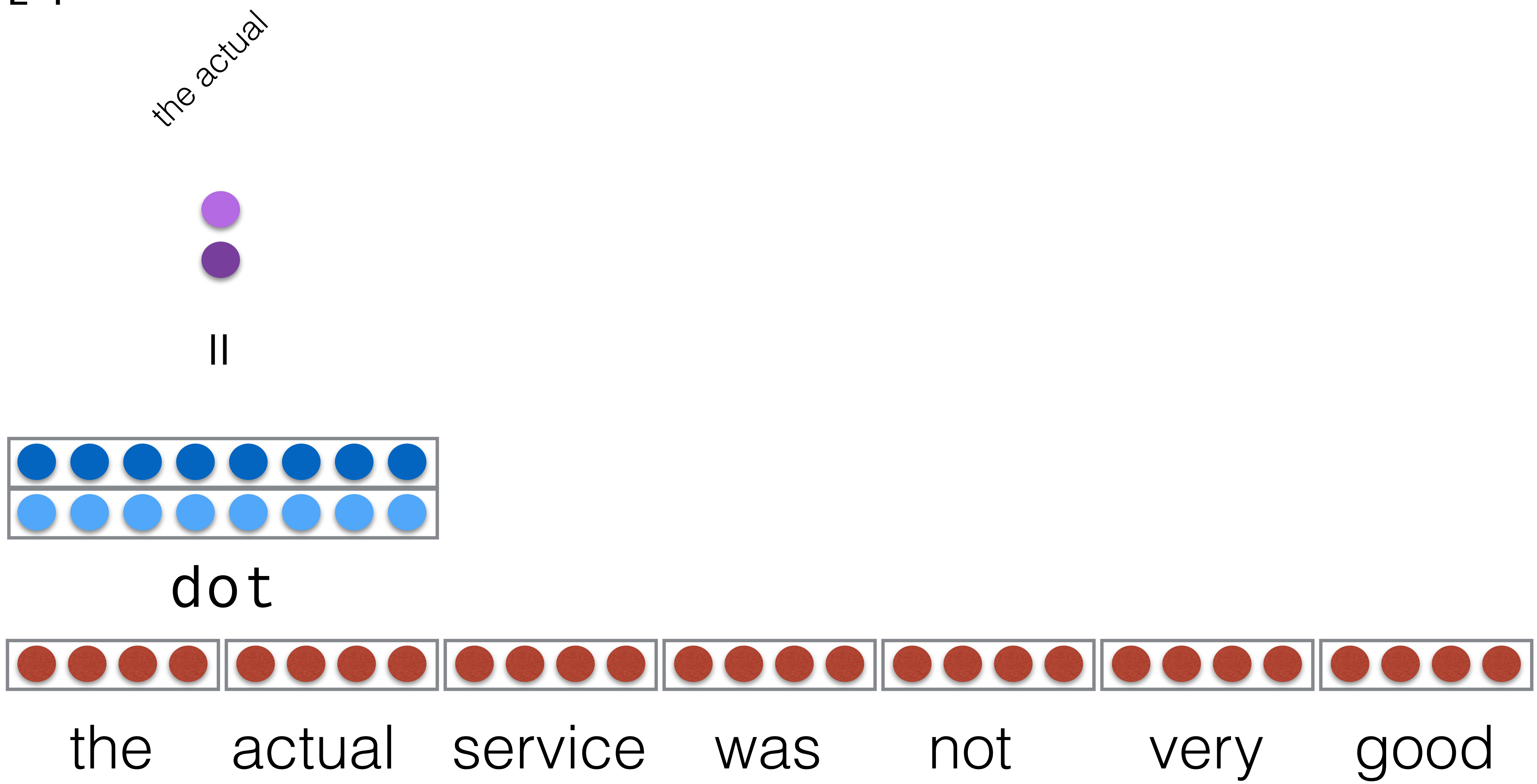
the actual
actual service
service was
was not
not very
=
dot
the
actual
service
was
not
very
good



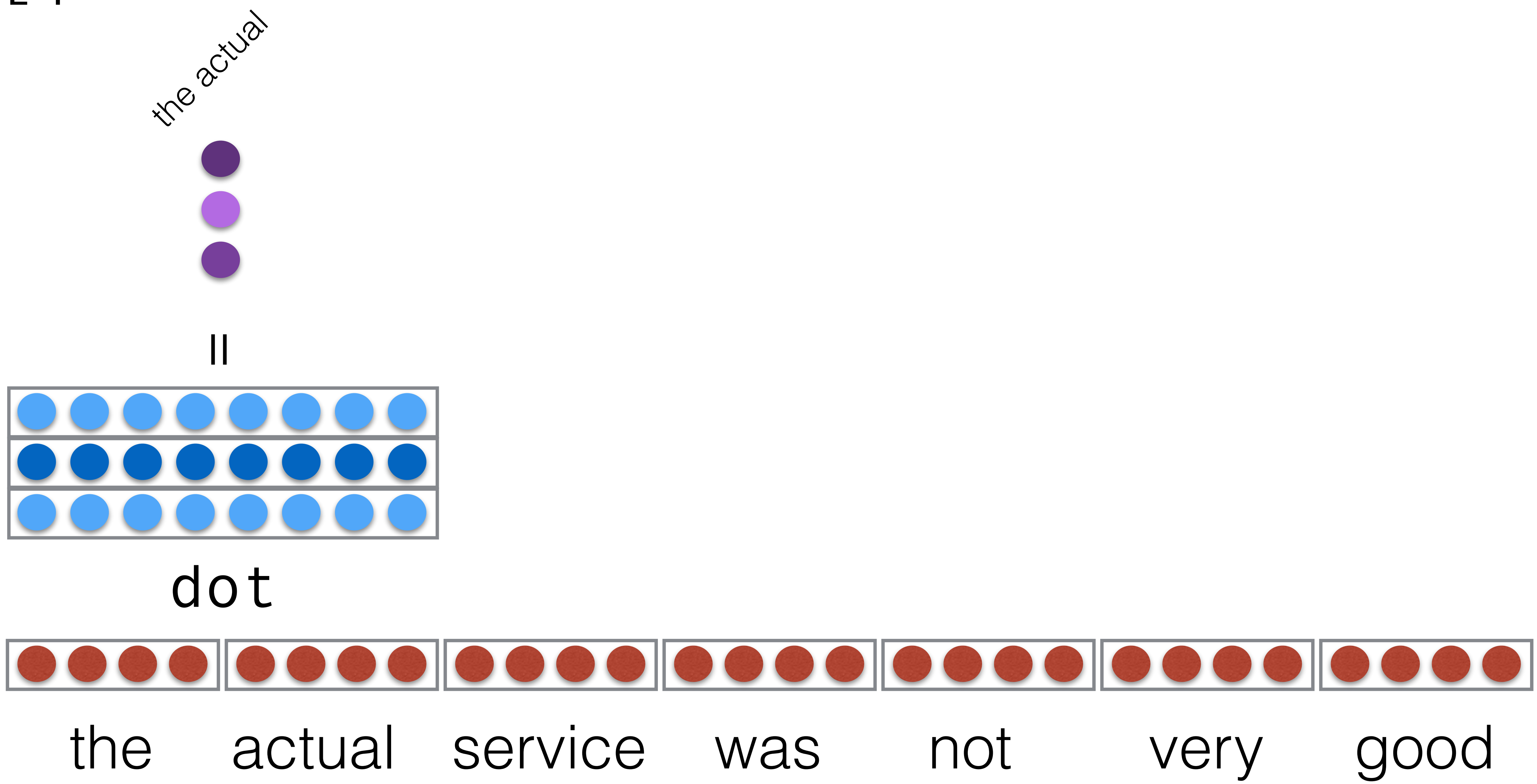
the actual
actual service
service was
was not
not very
very good
=
dot
the
actual
service
was
not
very
good



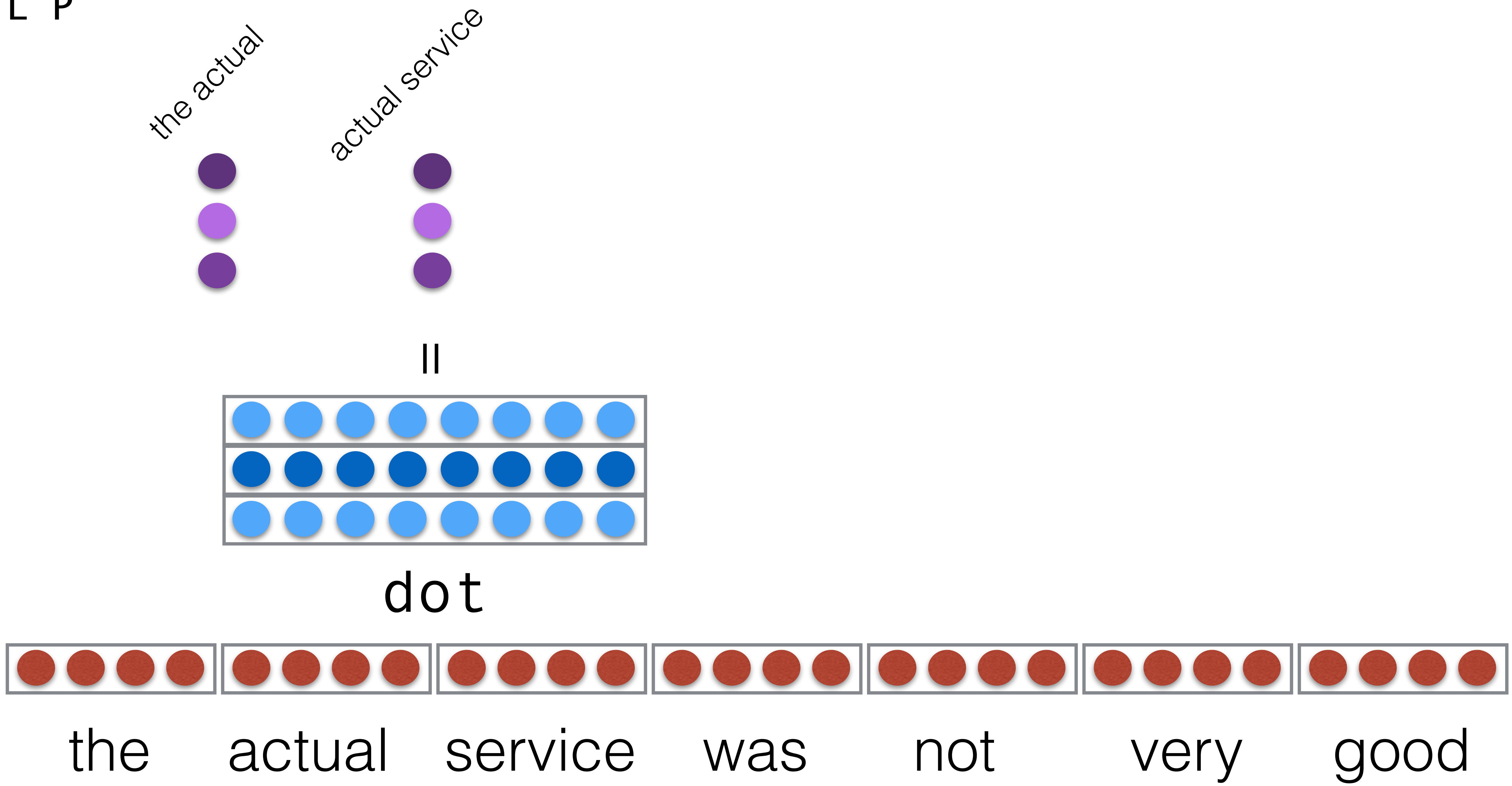
the actual
=
dot
the
actual
service
was
not
very
good

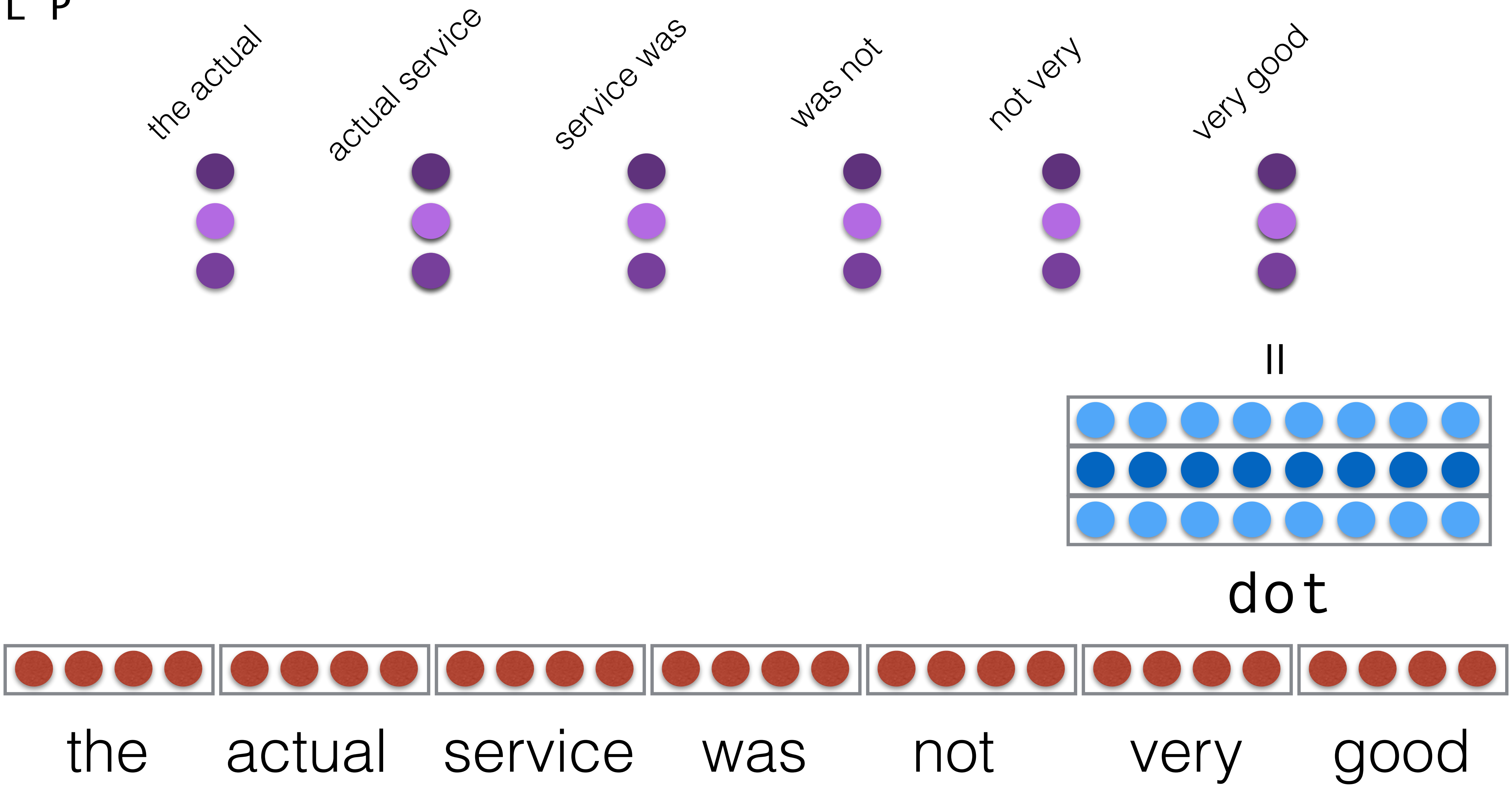


the actual
=
dot
the
actual
service
was
not
very
good

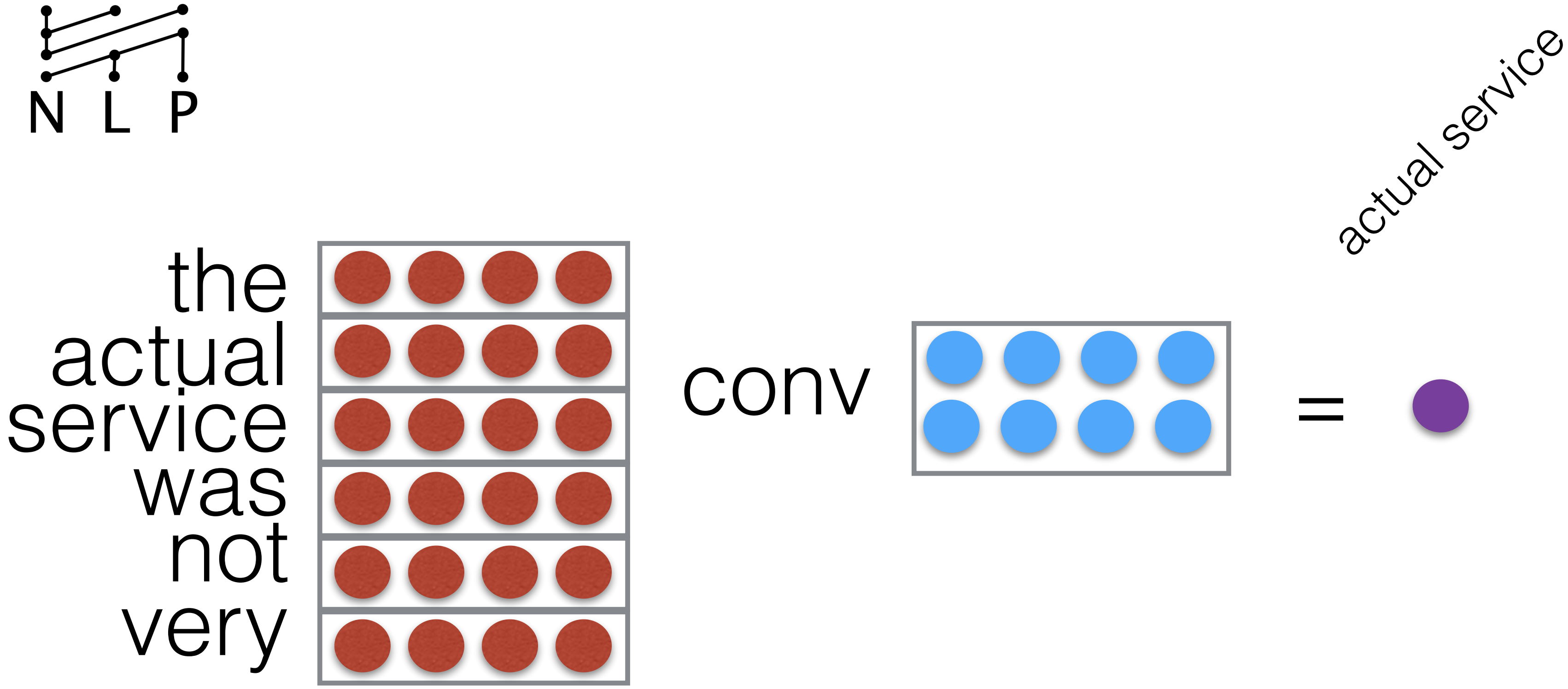


the actual
=
dot
the
actual
service
was
not
very
good

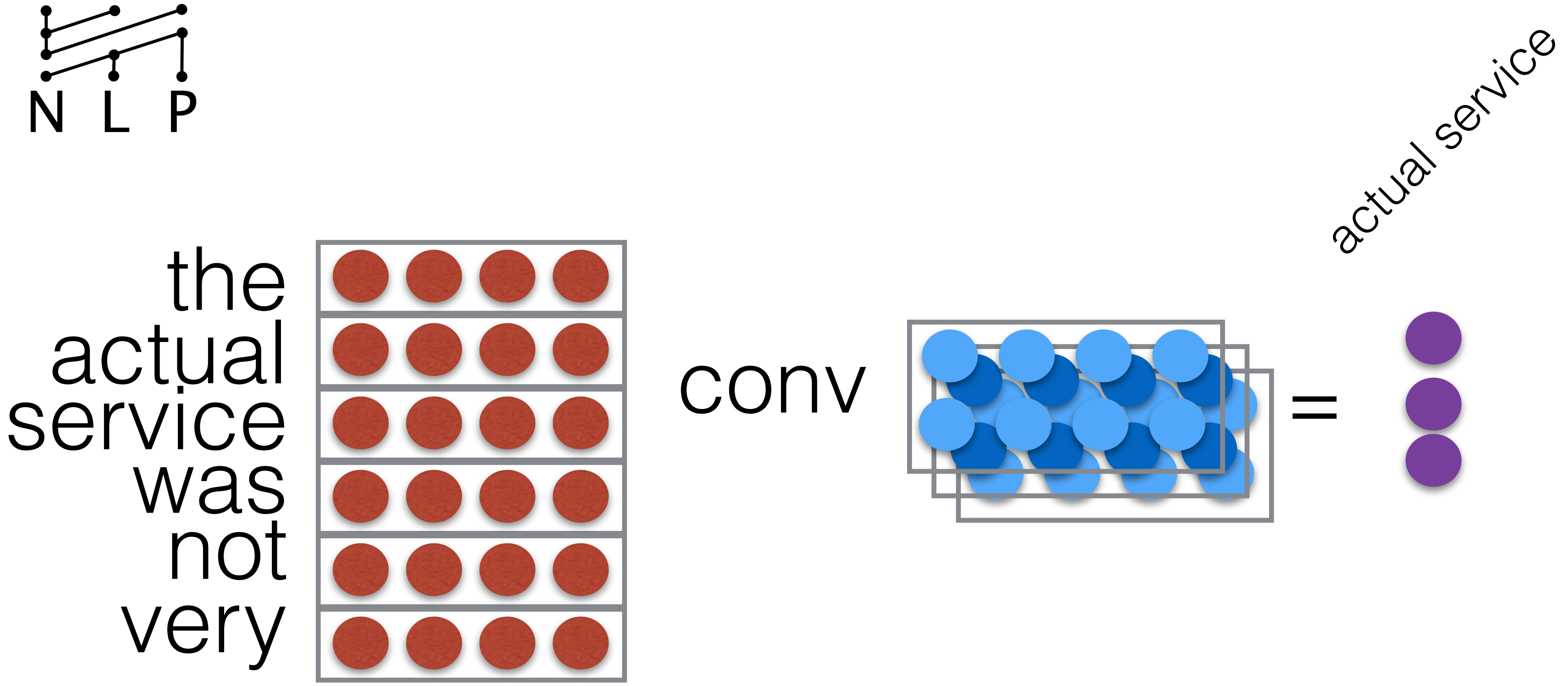

the actual
actual service
=
dot
the
actual
service
was
not
very
good



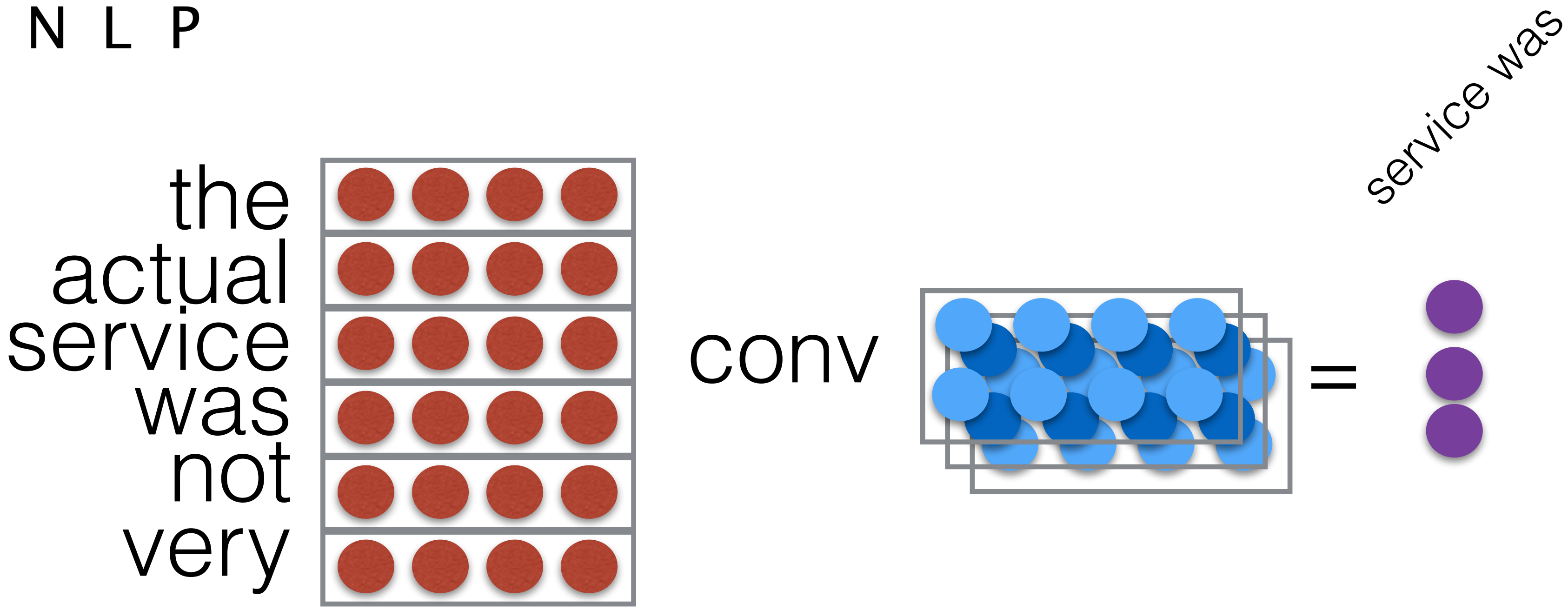
the actual
actual service
service was
was not
not very
very good
=
dot
the
actual
service
was
not
very
good



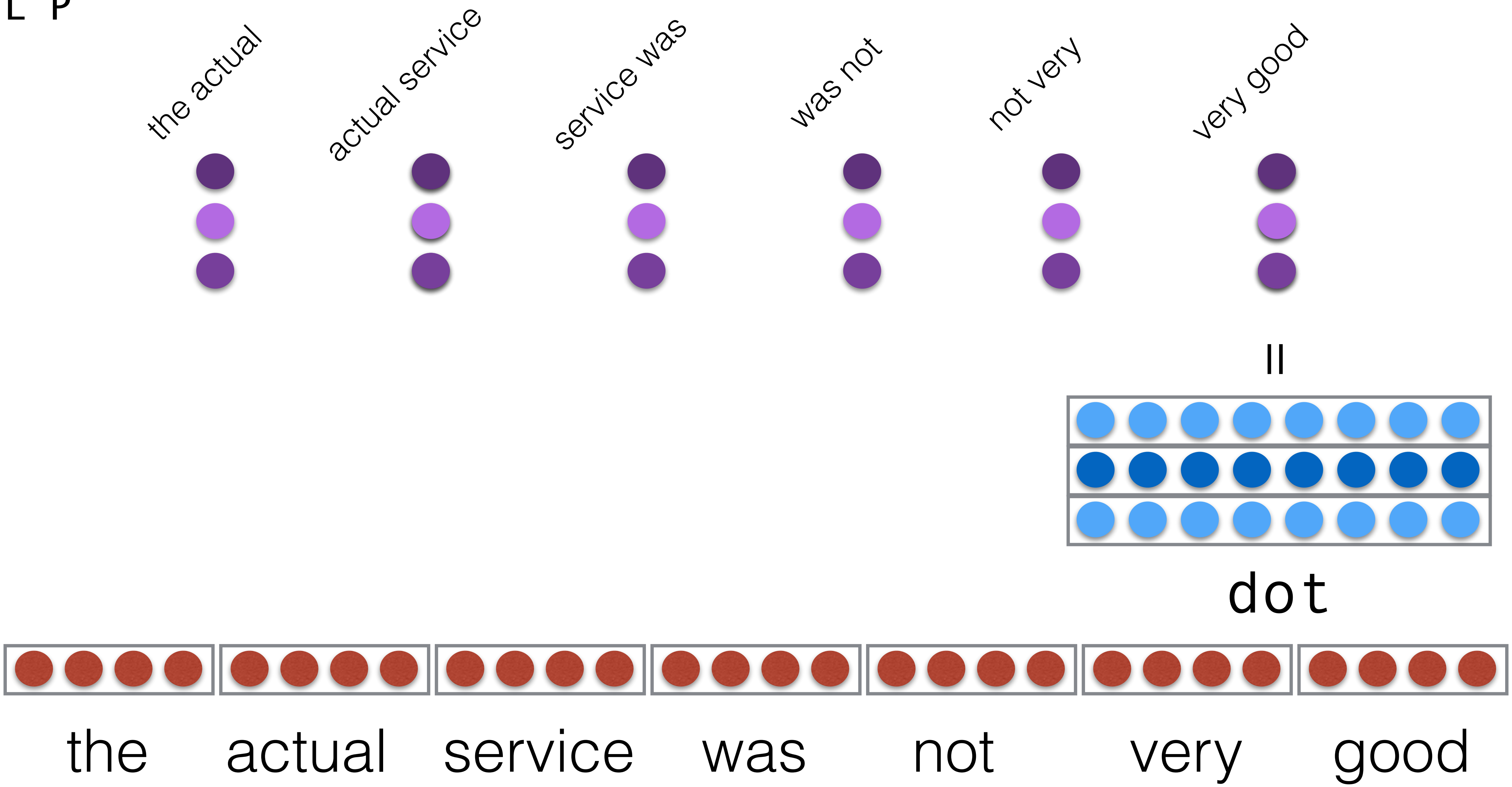
(another way to represent text convolutions)



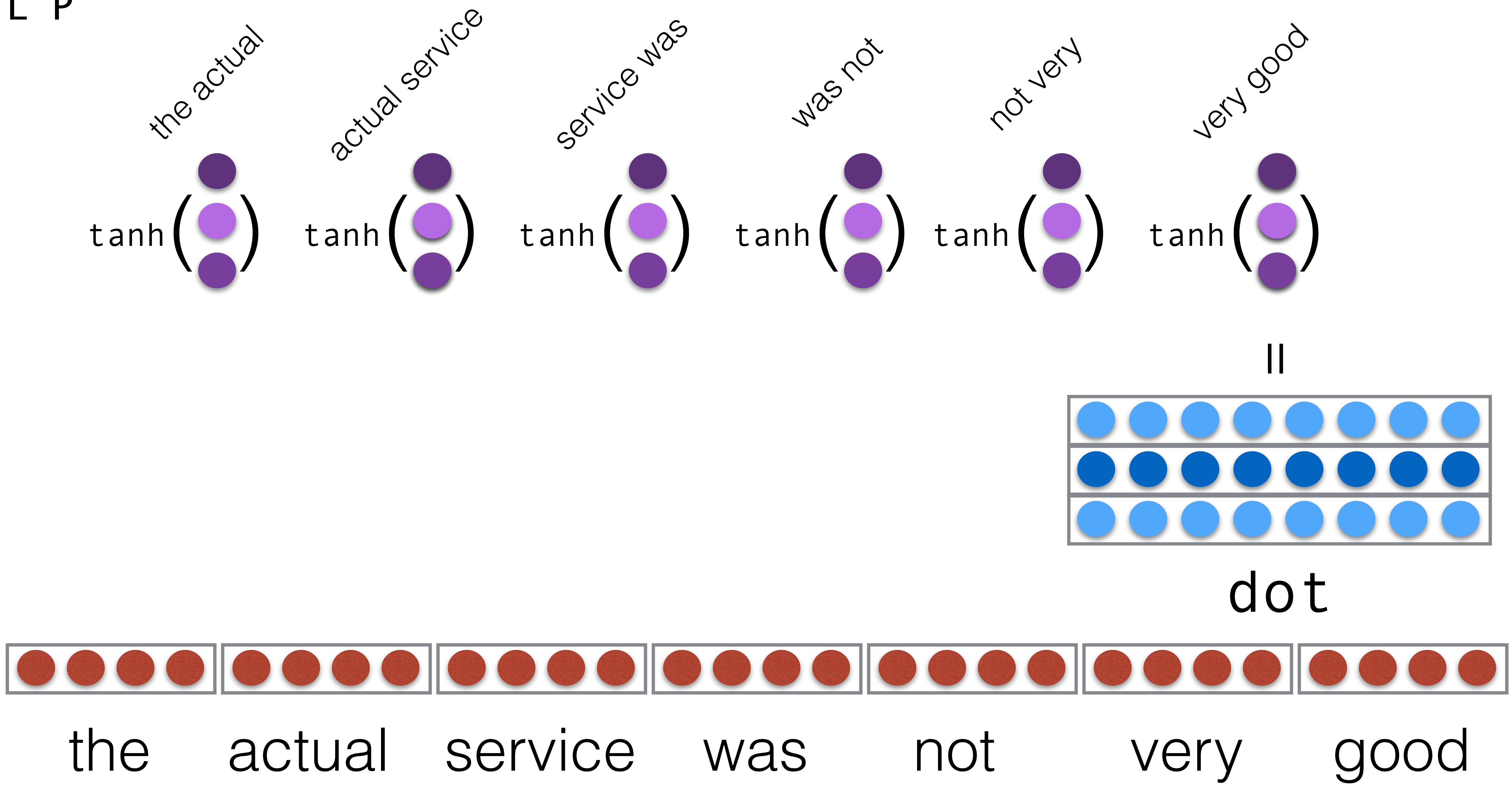
(another way to represent text convolutions)



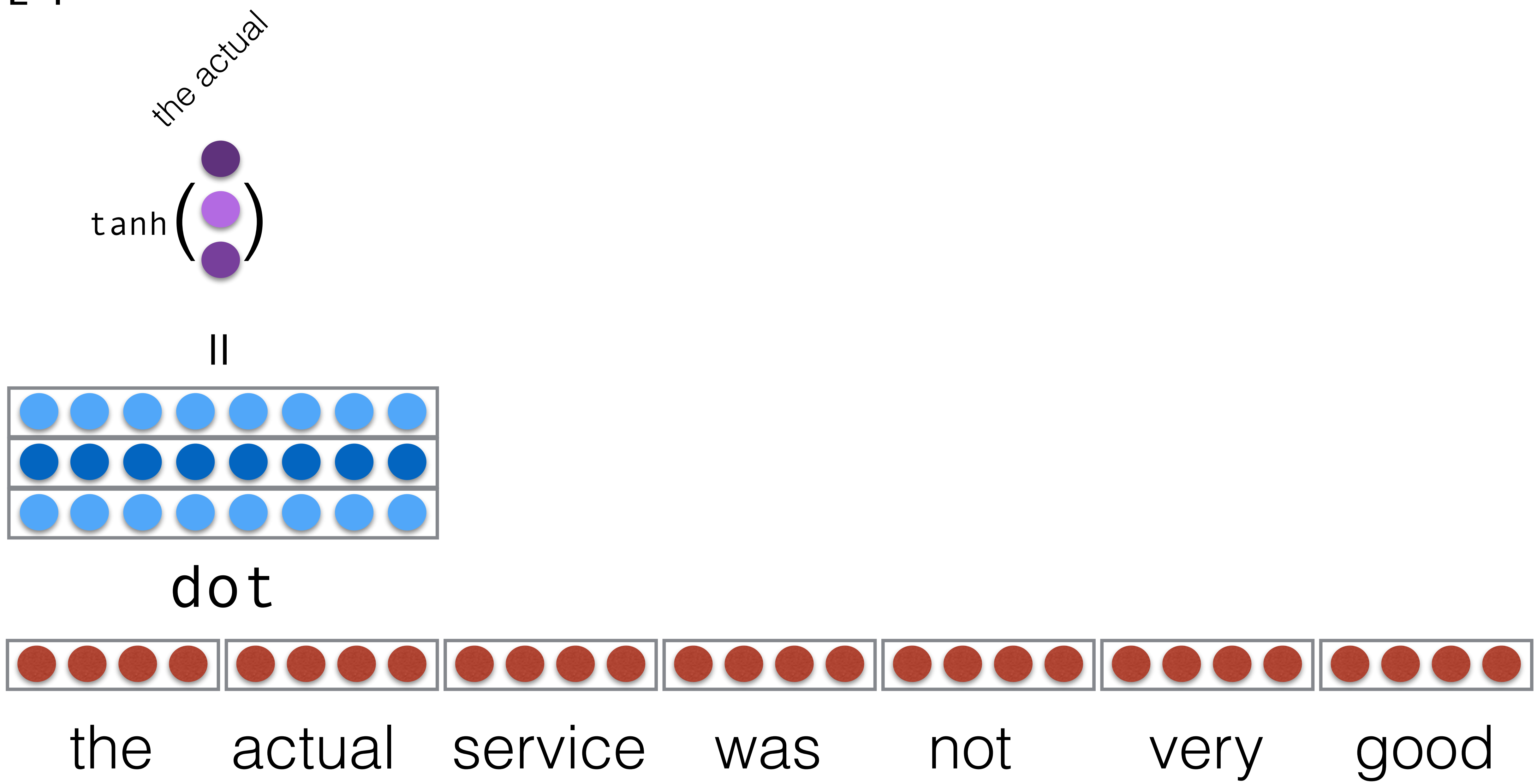
(another way to represent text convolutions)



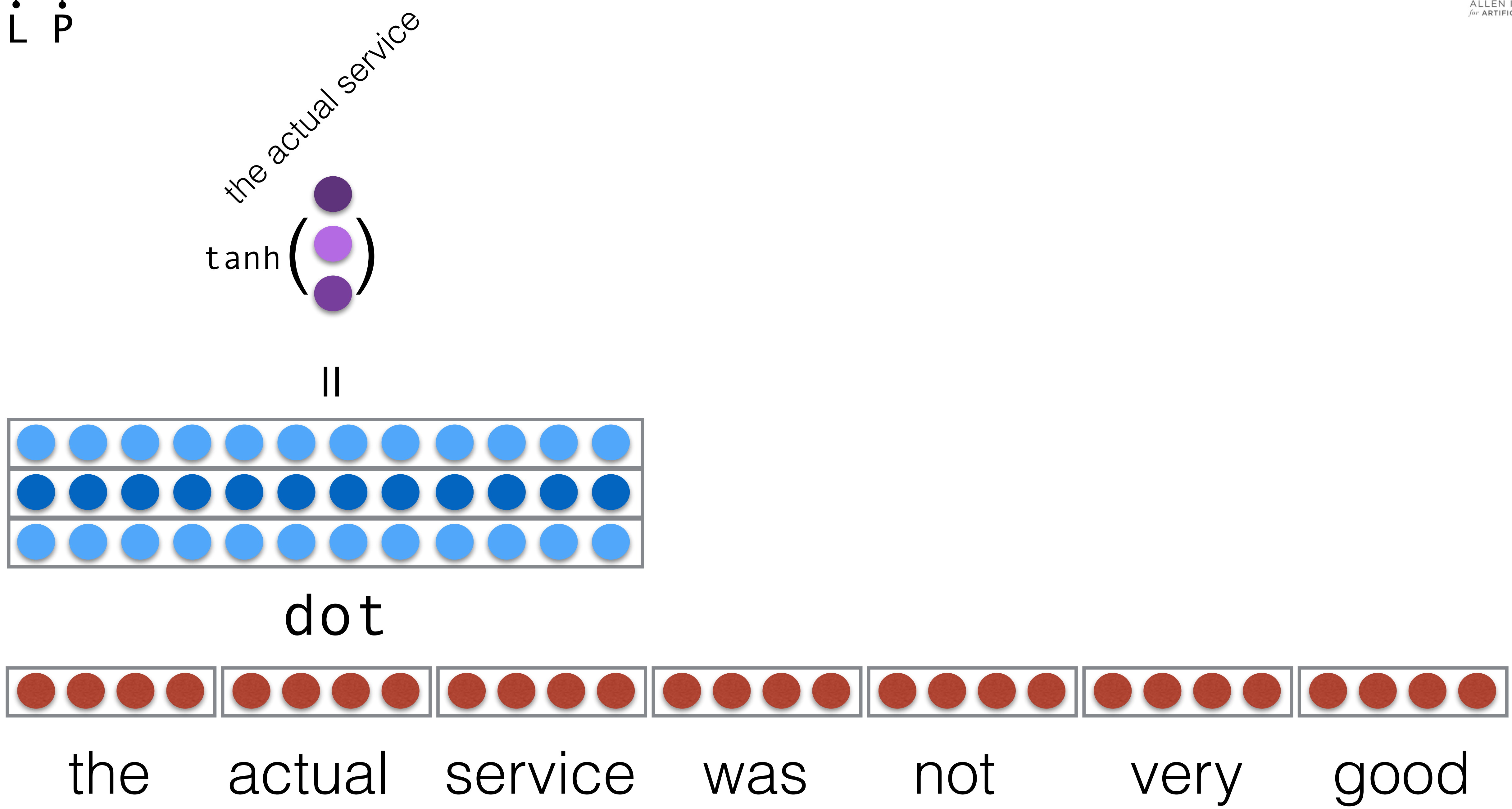
**(we'll focus on the 1-d view here,
but remember they are equivalent)**



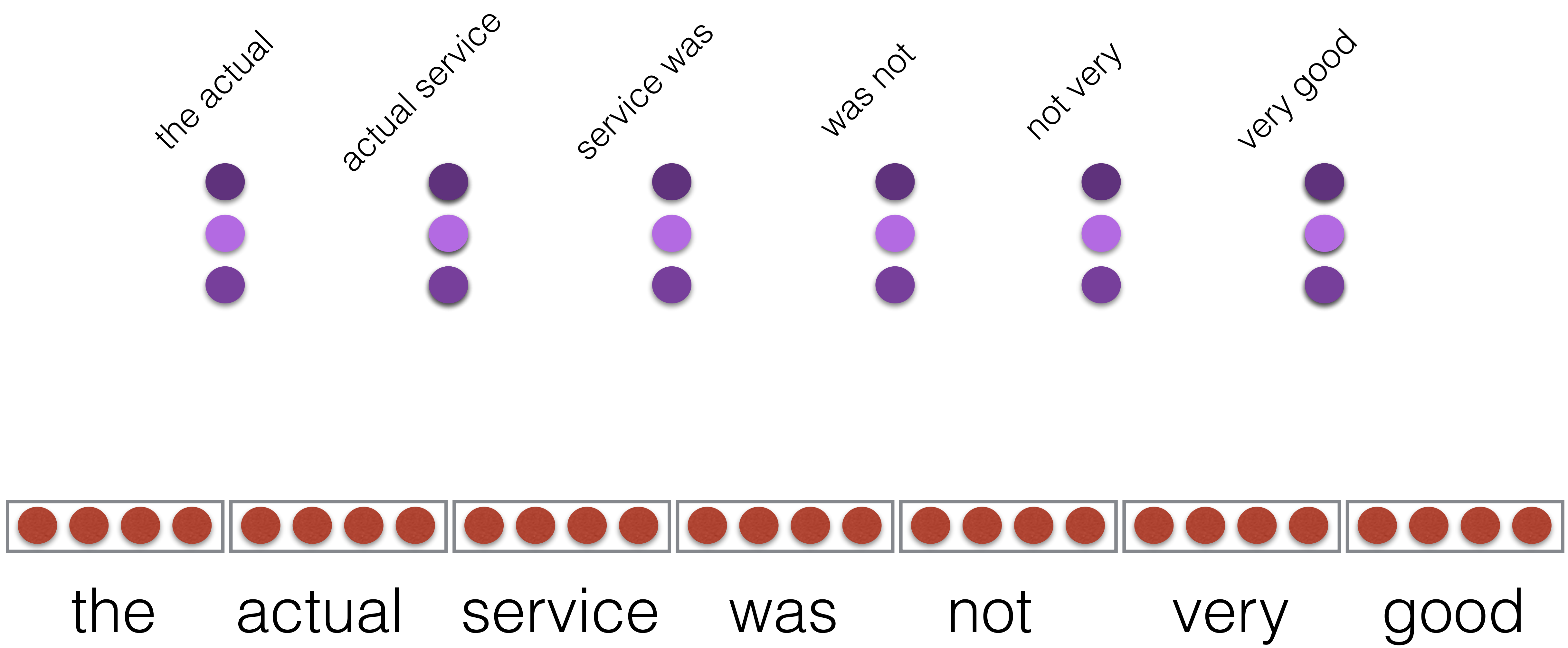
(usually also add non linearity)



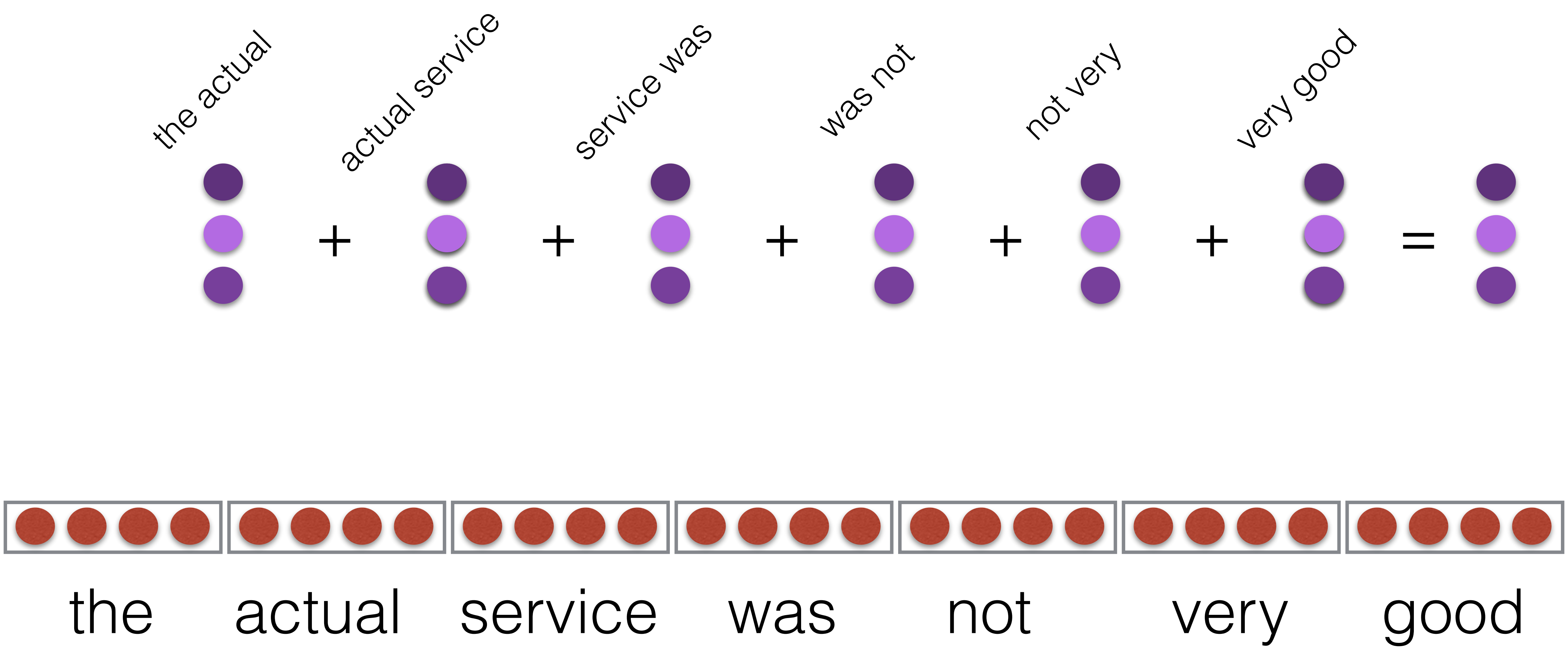
(can have larger filters)



(can have larger filters)



we have the ngram vectors. now what?



can do "pooling"

"Pooling"

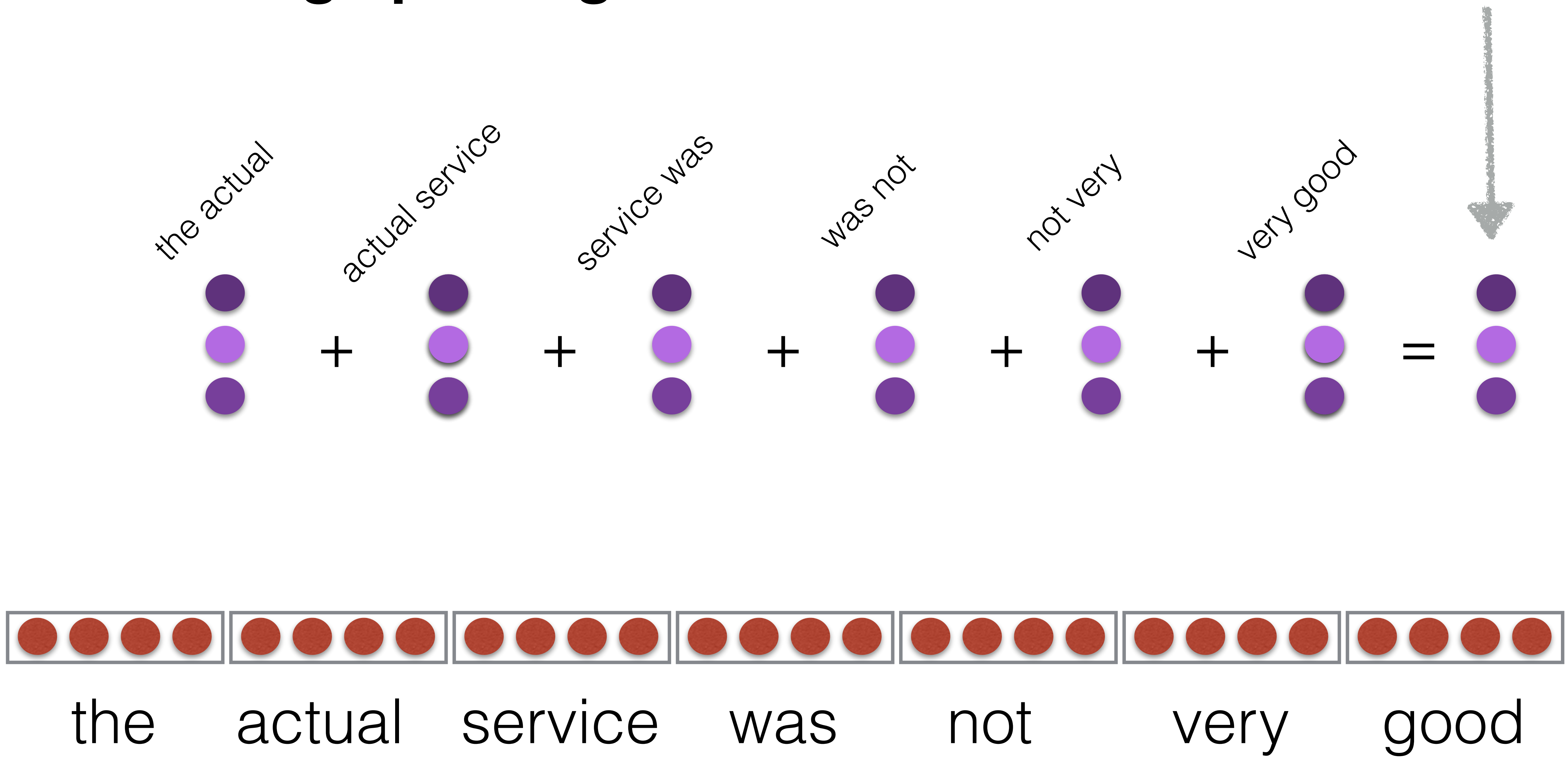
Combine K vectors into a single vector

"Pooling"

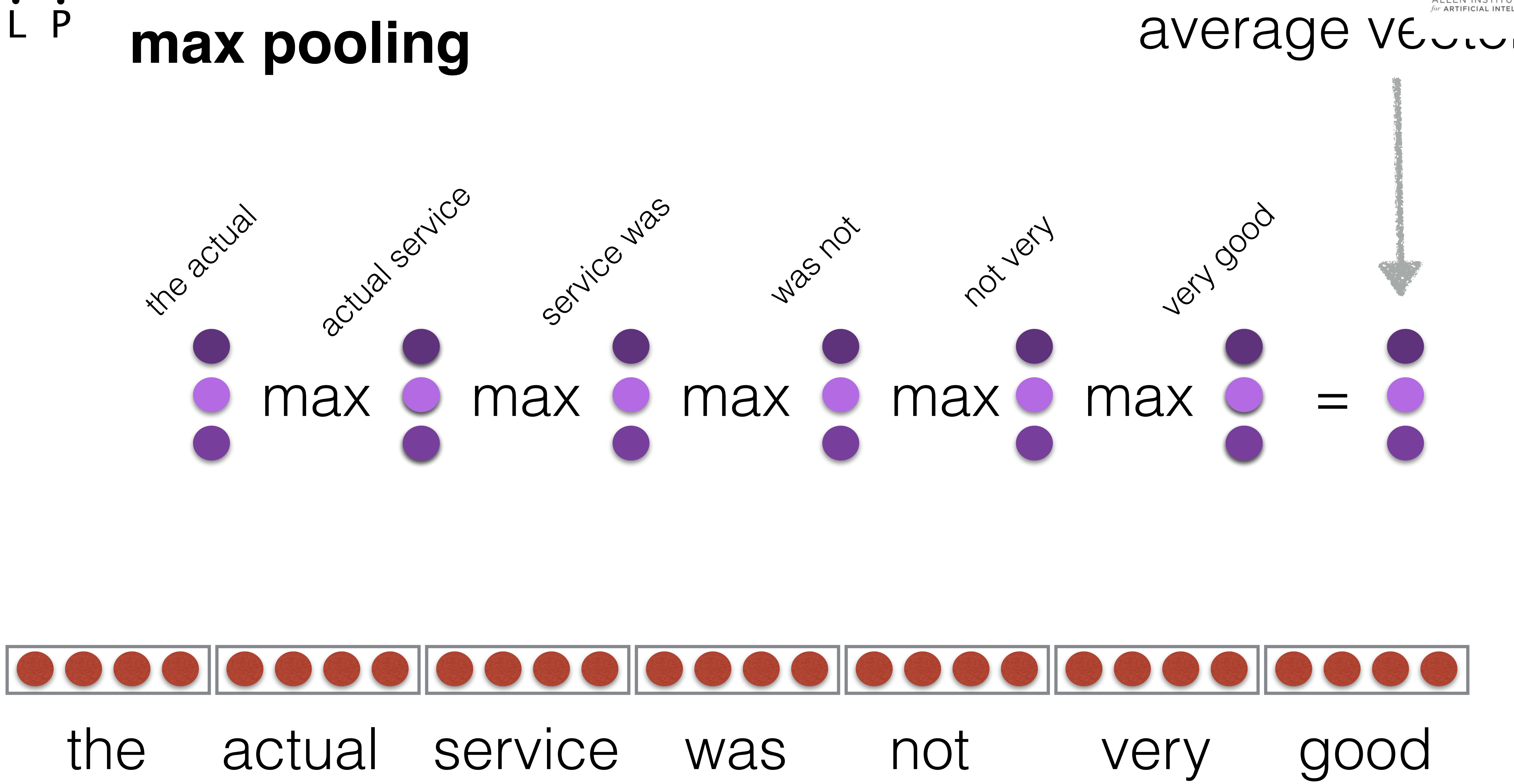
Combine K vectors into a single vector

This vector is a summary of the K vectors, and can be used for prediction.

average pooling

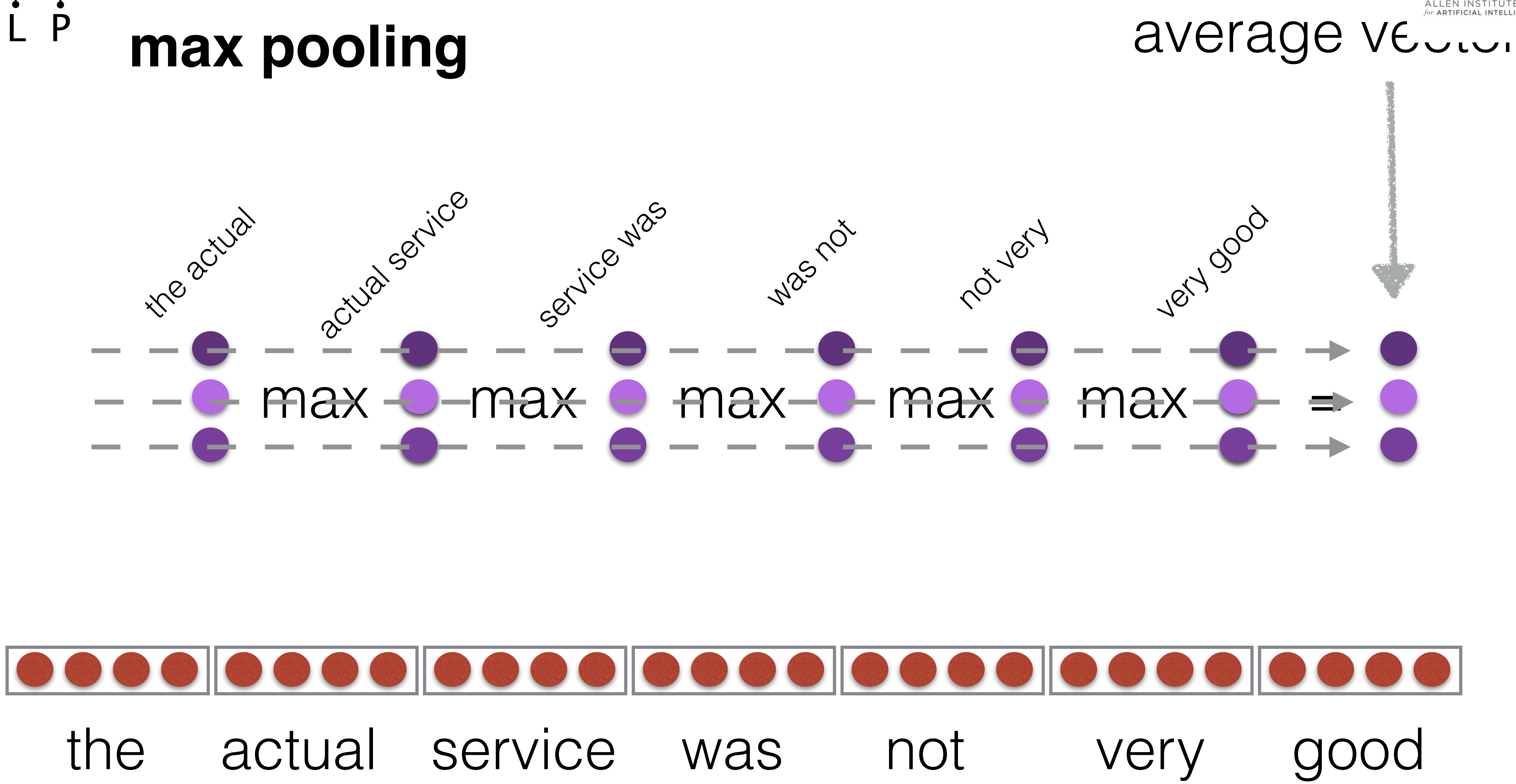


max pooling



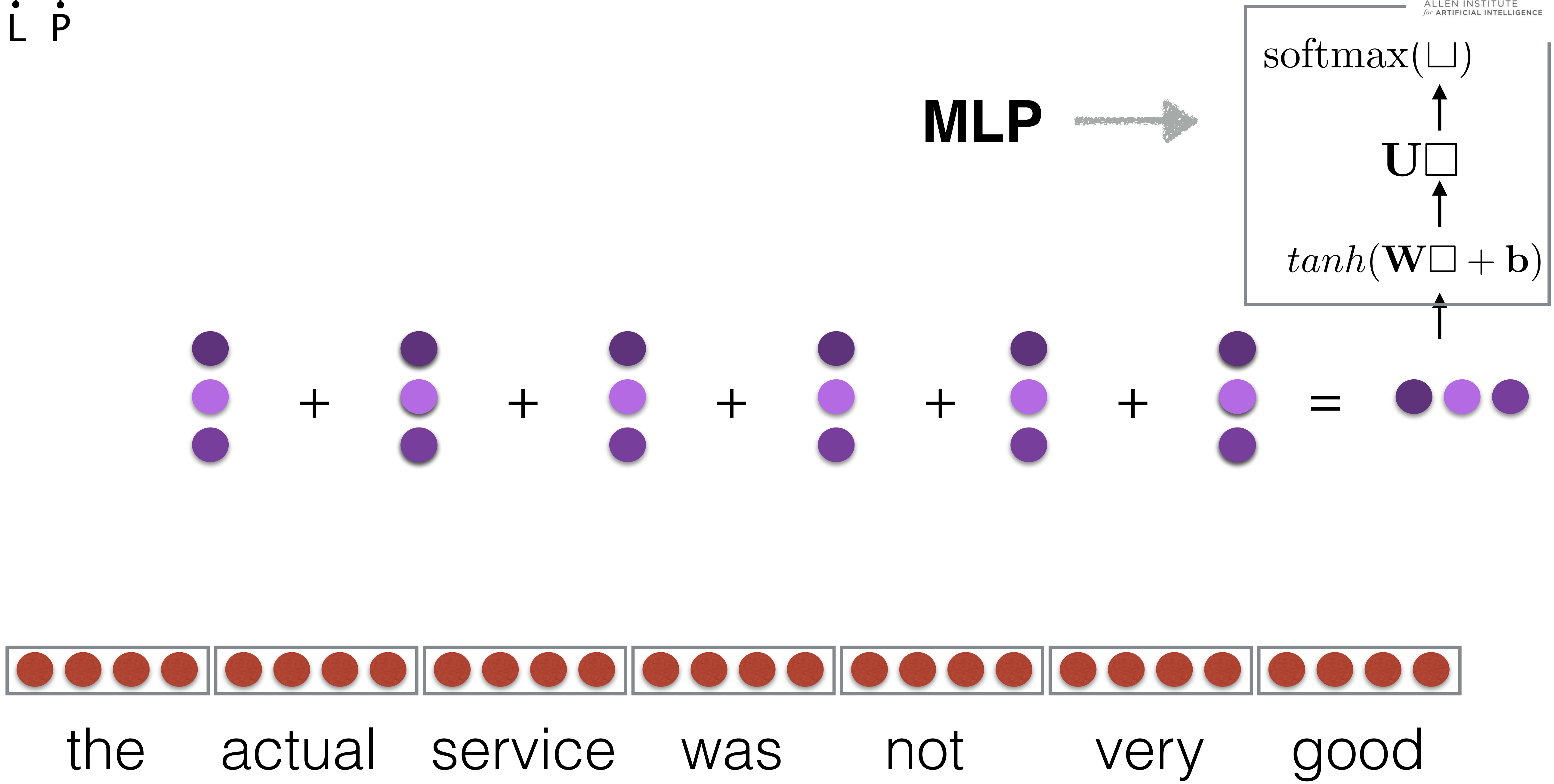
average
the actual
actual service
service was
was not
not very
very good
max
max
max
max
max
=
the
actual
service
was
not
very
good

max pooling



max over each dimension

predic



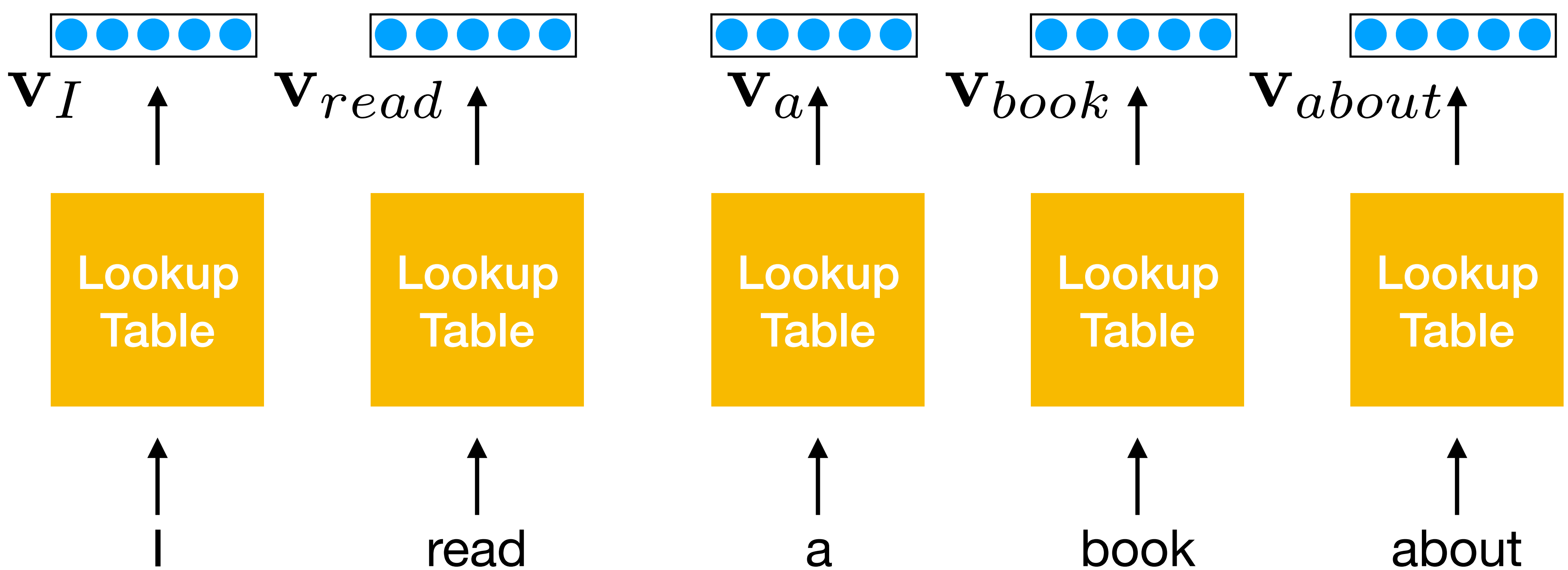
train end-to-end for some task

(train the MLP, the filter matrix, and the embeddings together)

RNNs

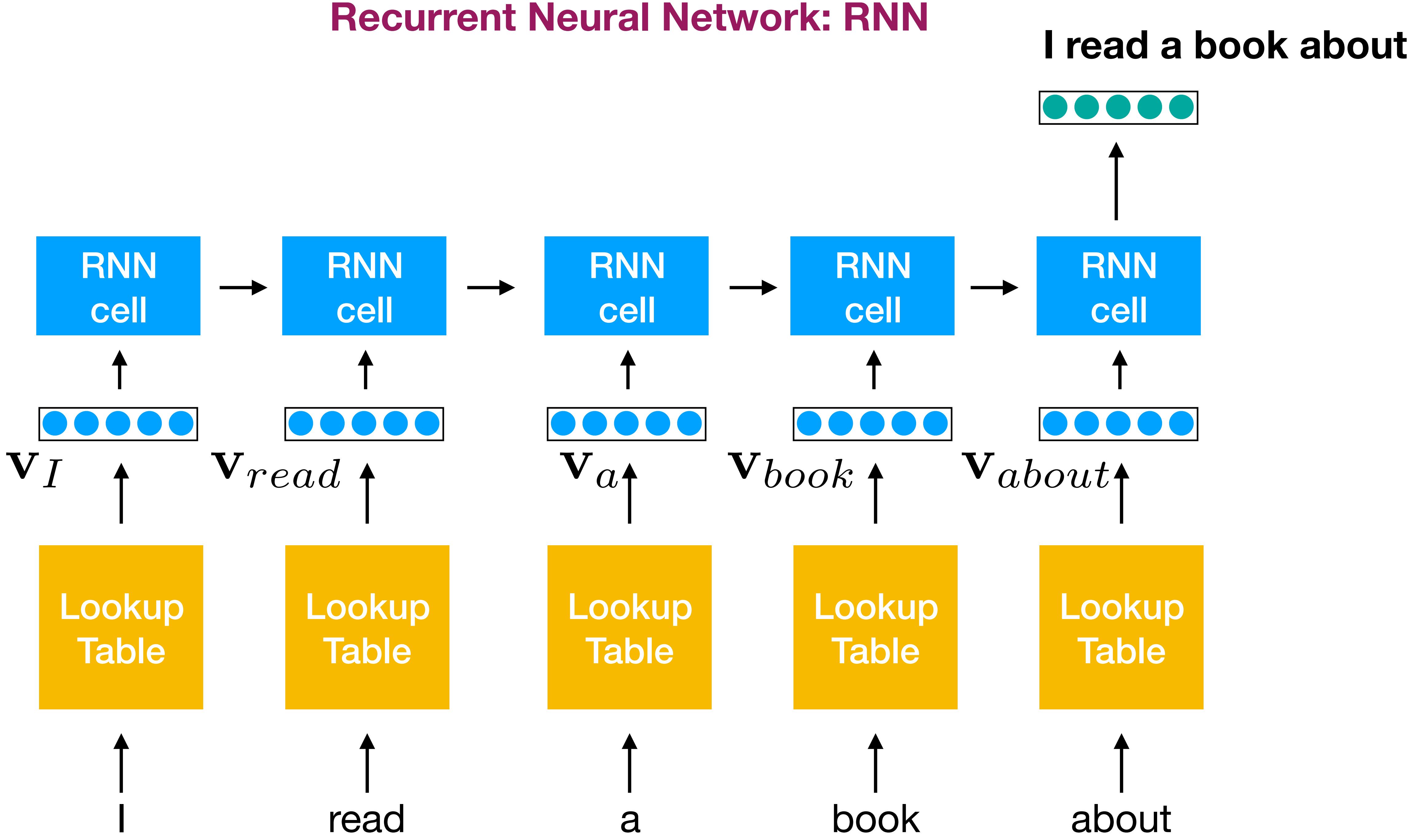
Combining Vectors

Recurrent Neural Network: RNN

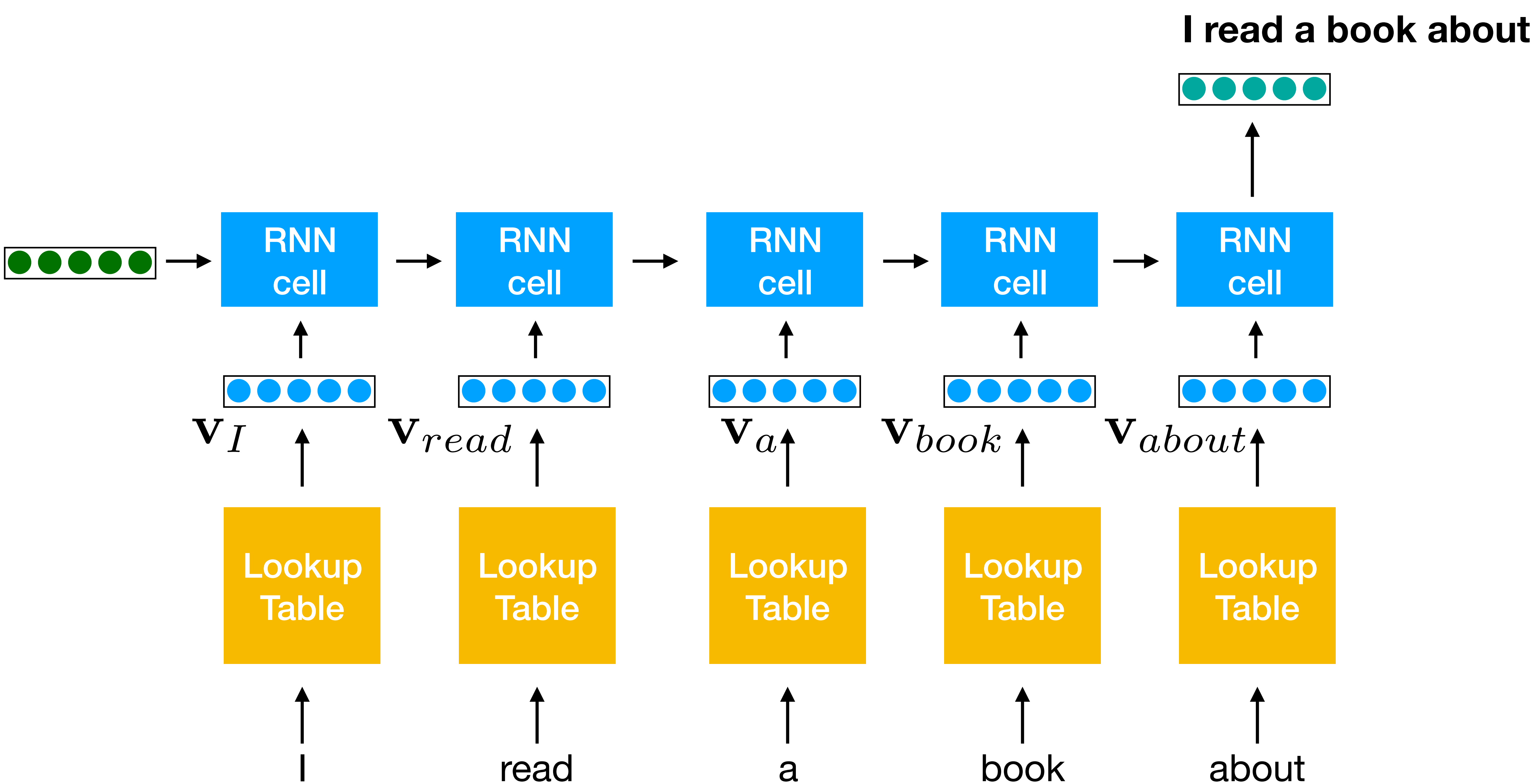


Combining Vectors

Recurrent Neural Network: RNN

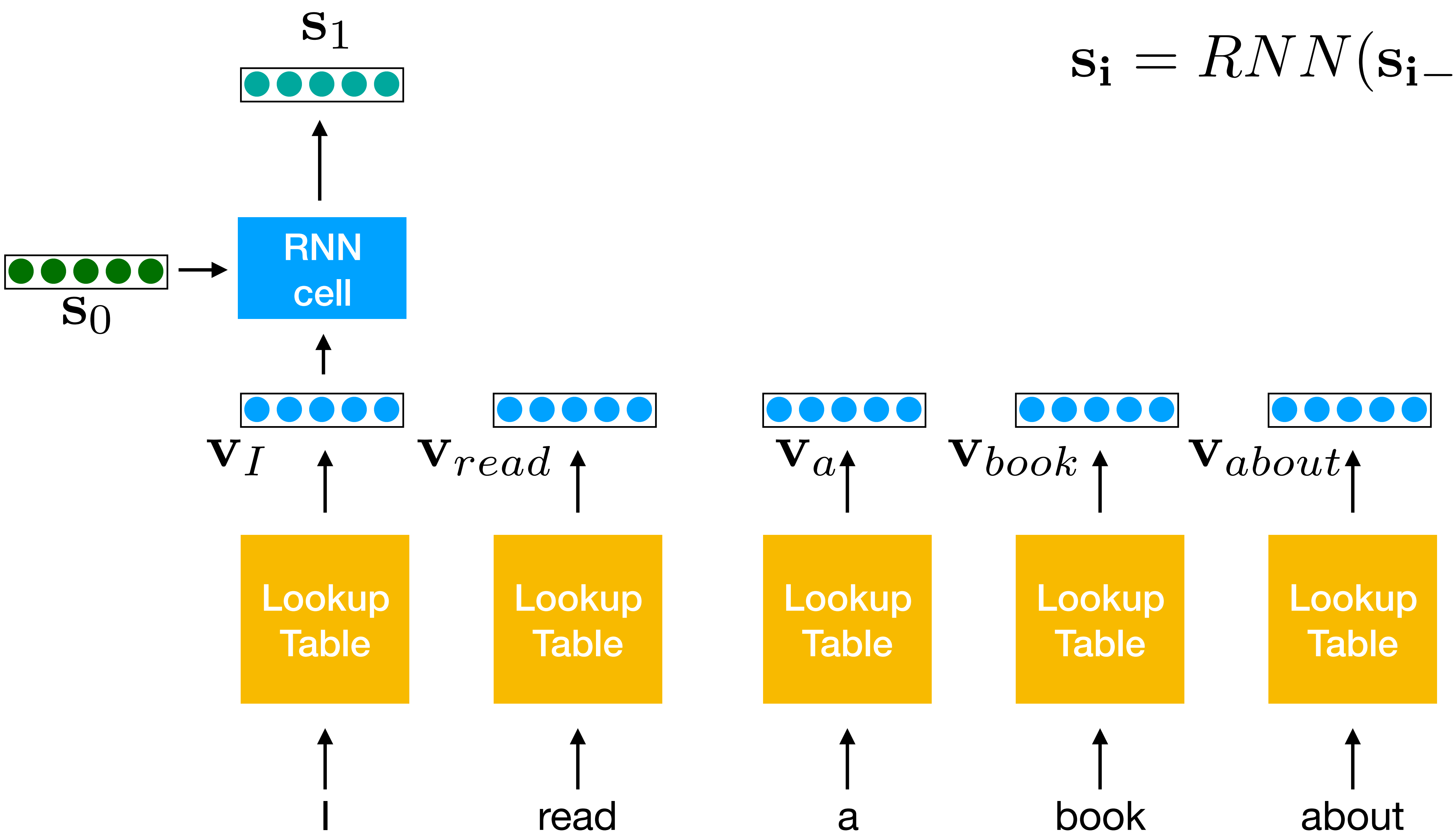


Combining Vectors



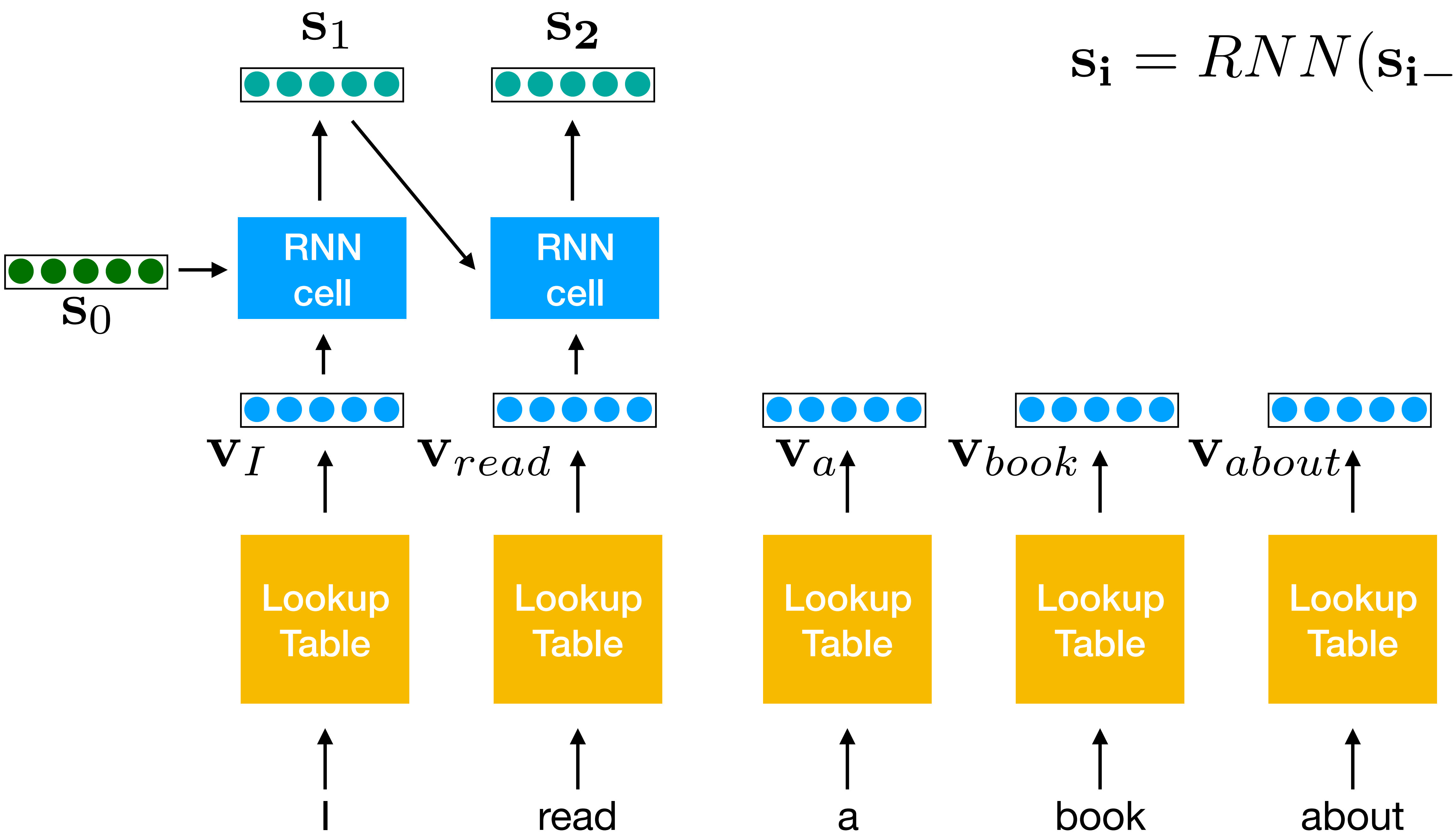
Combining Vectors

$$\mathbf{s_i} = RNN(\mathbf{s_{i-1}}, \mathbf{x_i})$$

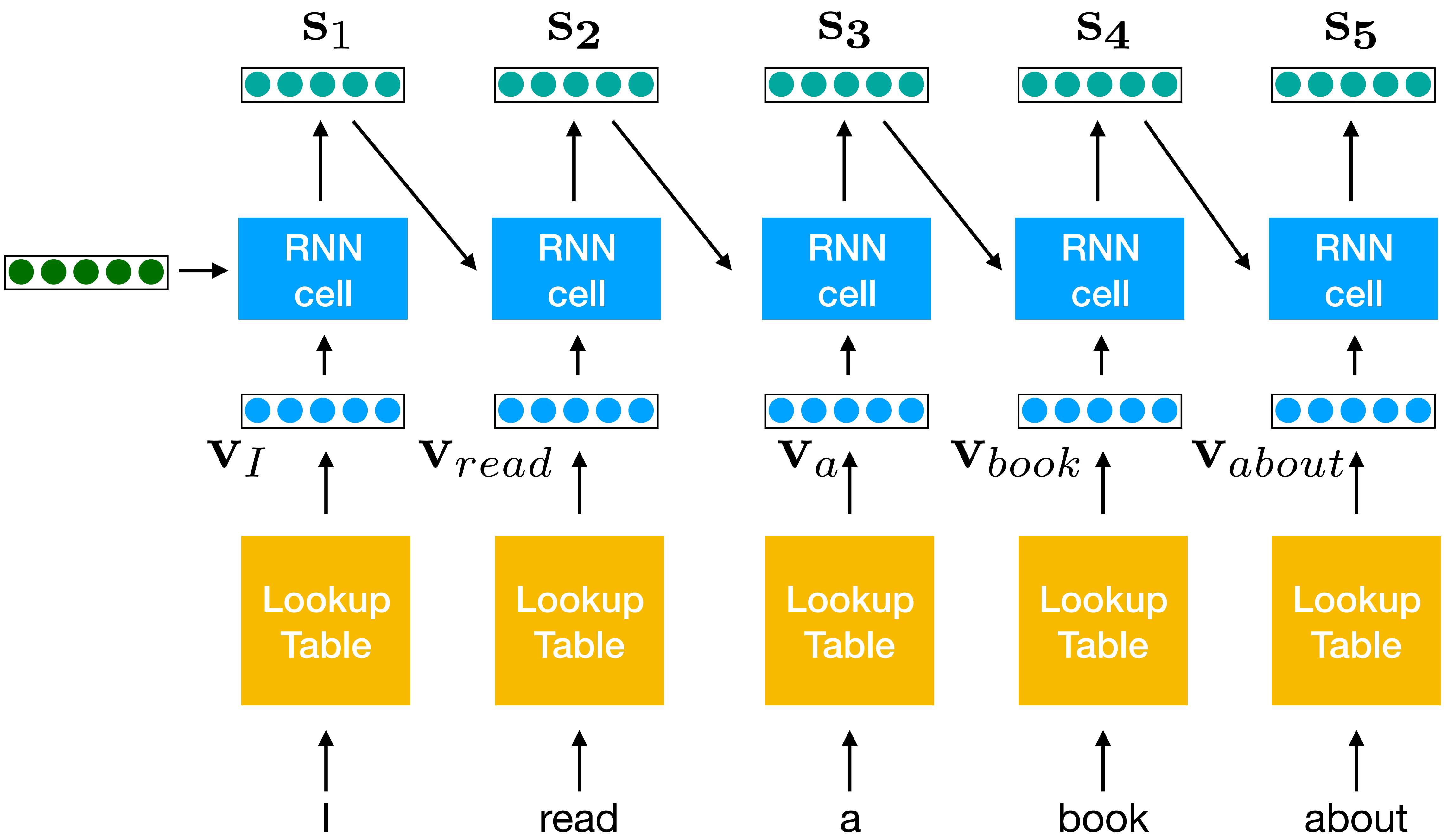


Combining Vectors

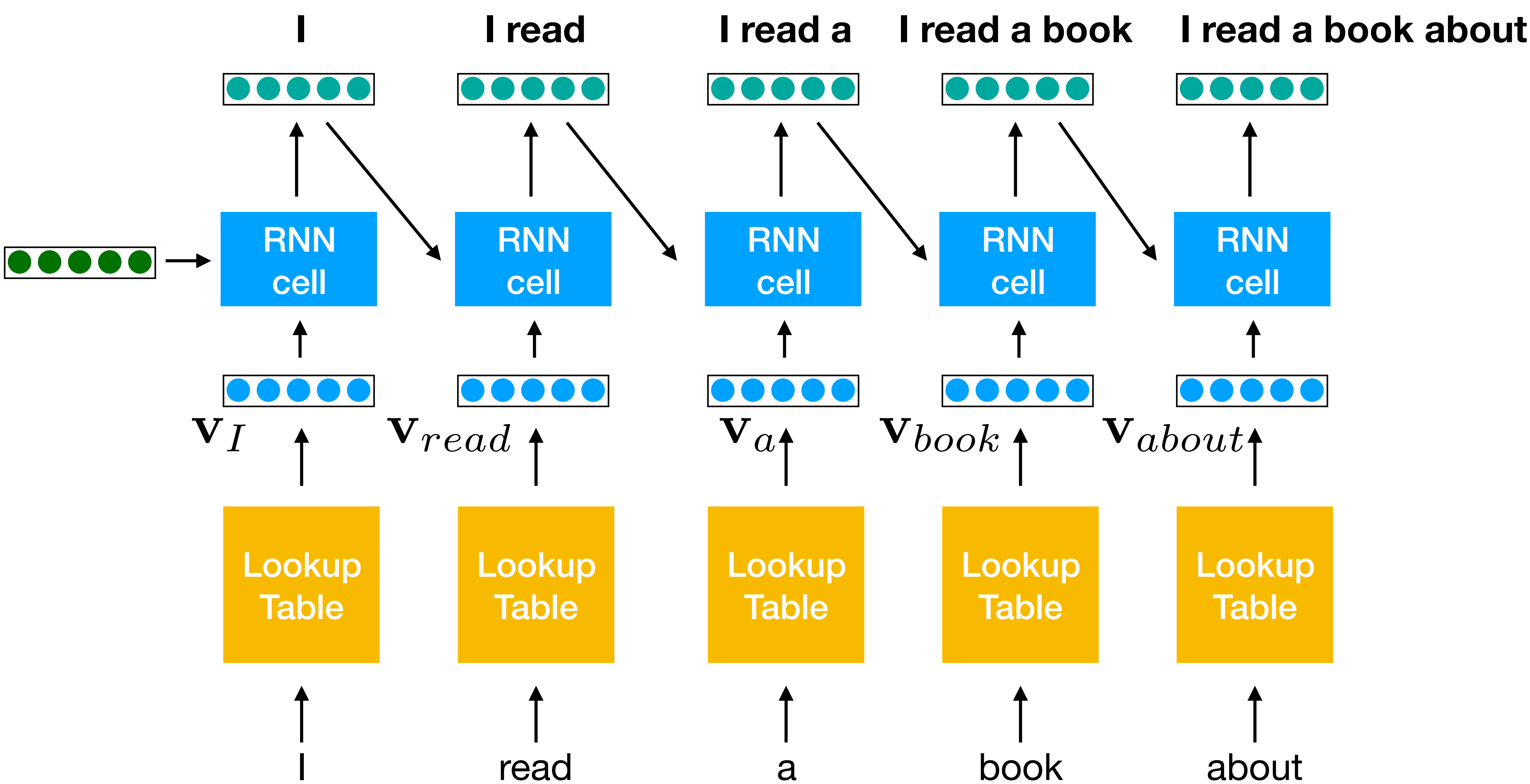
$$\mathbf{s_i} = RNN(\mathbf{s_{i-1}}, \mathbf{x_i})$$



Combining Vectors



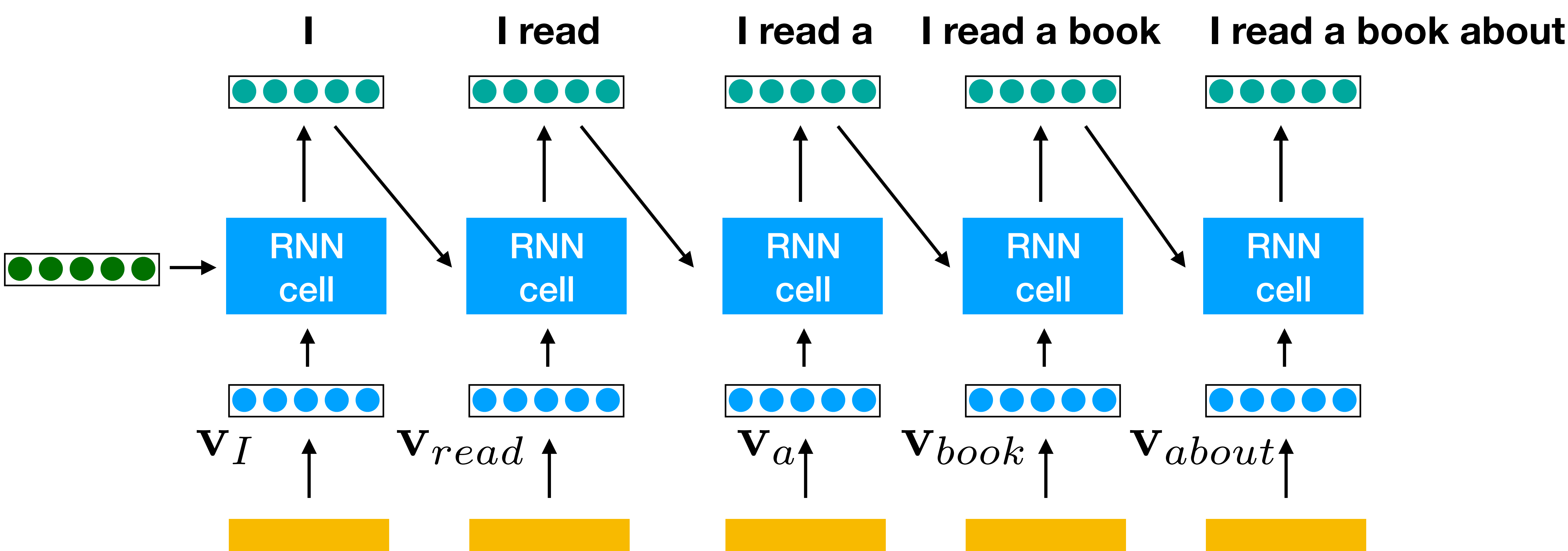
Combining Vectors



Combining Vectors

Recurrent Neural Network: RNN

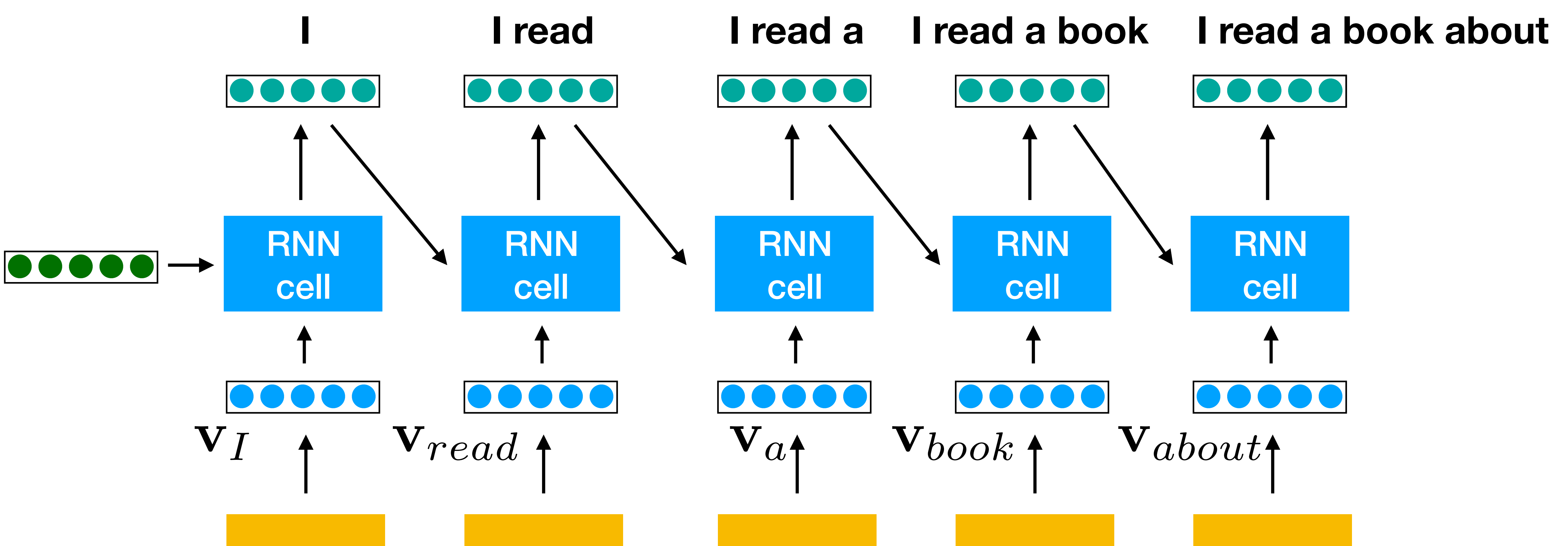
$$\mathbf{s_i} = RNN(\mathbf{s_{i-1}}, \mathbf{x_i})$$



Combining Vectors

Recurrent Neural Network: RNN

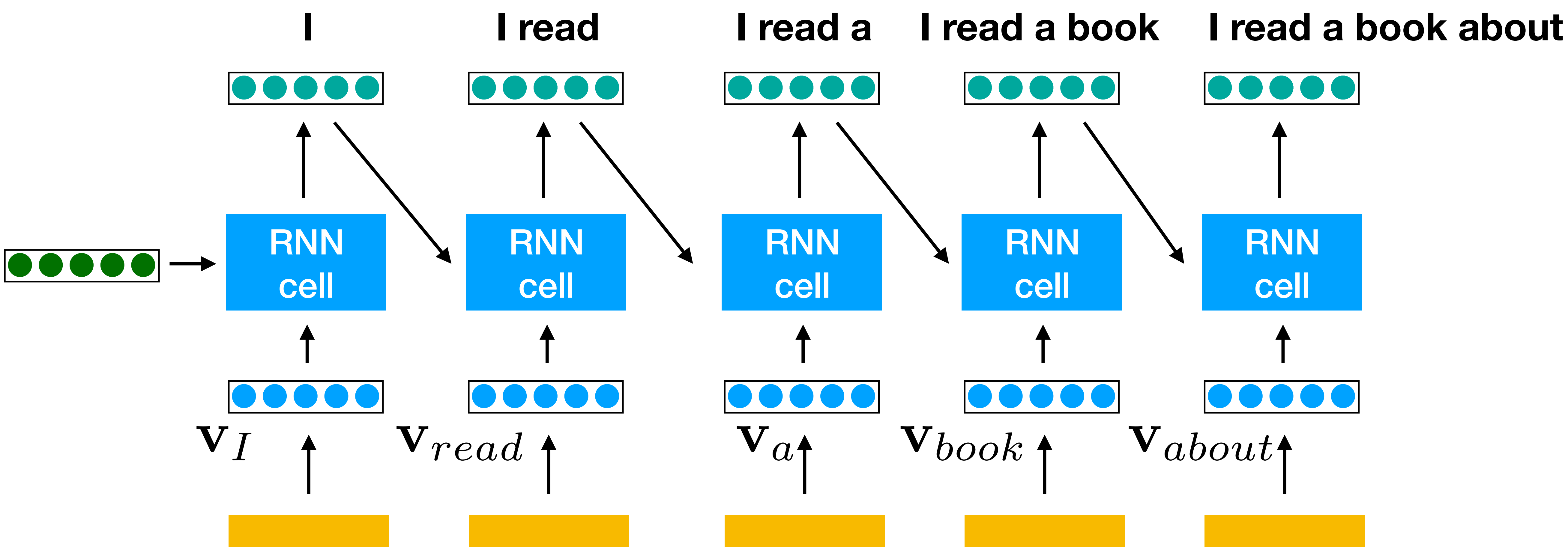
$$R_{SRNN}(\mathbf{s_{i-1}}, \mathbf{x_i}) = tanh(\mathbf{W^s} \cdot \mathbf{s_{i-1}} + \mathbf{W^x} \cdot \mathbf{x_i})$$



Combining Vectors

Recurrent Neural Network: RNN

$$
\begin{aligned}
R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) =& [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} =& \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} =& \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} =& \sigma(\mathbf{W^{xi}} \cdot \mathbf{x_j} + \mathbf{W^{hi}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{f} =& \sigma(\mathbf{W^{xf}} \cdot \mathbf{x_j} + \mathbf{W^{hf}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{o} =& \sigma(\mathbf{W^{xo}} \cdot \mathbf{x_j} + \mathbf{W^{ho}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{g} =& \tanh(\mathbf{W^{xg}} \cdot \mathbf{x_j} + \mathbf{W^{hg}} \cdot \mathbf{h_{j-1}})
\end{aligned}
$$



LSTM: differential gates

$$
\begin{aligned}
R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) =& [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} =& \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} =& \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} =& \sigma(\mathbf{W^{xi}} \cdot \mathbf{x_j} + \mathbf{W^{hi}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{f} =& \sigma(\mathbf{W^{xf}} \cdot \mathbf{x_j} + \mathbf{W^{hf}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{o} =& \sigma(\mathbf{W^{xo}} \cdot \mathbf{x_j} + \mathbf{W^{ho}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{g} =& \tanh(\mathbf{W^{xg}} \cdot \mathbf{x_j} + \mathbf{W^{hg}} \cdot \mathbf{h_{j-1}})
\end{aligned}
$$

better controlled memory access

LSTM: differential gates

- The main idea behind the LSTM is that you want to somehow control the "memory access".

- In a SimpleRNN:

$$R_{SRNN}(\mathbf{s_{i-1}}, \mathbf{x_i}) = tanh(\mathbf{W^s} \cdot \mathbf{s_{i-1}} + \mathbf{W^x} \cdot \mathbf{x_i})$$

read previous state memory   write new input

- All the memory gets overwritten

Vector Gates

- We'd like to:
  * Selectively read from some memory "cells".
  * Selectively write to some memory "cells".

Vector "Gates"

- We'd like to:
  * Selectively read from some memory "cells".
  * Selectively write to some memory "cells".

- A gate function:

$$\underbrace{\begin{bmatrix}0\\1\\0\\0\\0\\1\end{bmatrix}}_{\mathbf{g}} \odot \underbrace{\begin{bmatrix}10\\11\\12\\13\\14\\15\end{bmatrix}}_{\mathbf{x}}$$

(element-wise multiplication)

gate controls access

vector of values

-

Vector "Gates"

- We'd like to:
  * Selectively read from some memory "cells".
  * Selectively write to some memory "cells".

- A gate function:

$$\mathbf{s_{i-1}} \odot \mathbf{g} \qquad\qquad \mathbf{g} \in \{0,1\}^d$$

vector of values

gate controls access

-

Vector "Gates"

- Using the gate function to control access:

$$\mathbf{s_i} \leftarrow \mathbf{s_{i-1}} \odot \mathbf{g^r} + \mathbf{x_i} \odot \mathbf{g^w} \qquad \mathbf{g} \in \{0,1\}^d$$

which cells to read

which cells to write

-

Vector "Gates"

- Using the gate function to control access:

$$\mathbf{s_i} \leftarrow \mathbf{s_{i-1}} \odot \mathbf{g^r} + \mathbf{x_i} \odot \mathbf{g^w} \qquad \mathbf{g} \in \{0,1\}^d$$

which cells to read

which cells to write

- (can also tie them: $\mathbf{g^r} = 1 - \mathbf{g^w}$)

Vector "Gates"

$$\underset{\mathbf{s}'}{\begin{bmatrix}8\\11\\3\\7\\5\\15\end{bmatrix}} \leftarrow \underset{\mathbf{g}}{\begin{bmatrix}0\\1\\0\\0\\0\\1\end{bmatrix}} \odot \underset{\mathbf{x}}{\begin{bmatrix}10\\11\\12\\13\\14\\15\end{bmatrix}} + \underset{(\mathbf{1}-\mathbf{g})}{\begin{bmatrix}1\\0\\1\\1\\1\\0\end{bmatrix}} \odot \underset{\mathbf{s}}{\begin{bmatrix}8\\9\\3\\7\\5\\8\end{bmatrix}}$$

Differentiable "Gates"

- **Problem with the gates**:
  * they are fixed.
  * they don't depend on the input or the output.

Differentiable "Gates"

- **Problem with the gates**:
  * they are fixed.
  * they don't depend on the input or the output.

- Solution: make them smooth, input dependent, and trainable.

$$\mathbf{g^r} = \sigma(\mathbf{W} \cdot \mathbf{x_i} + \mathbf{U} \cdot \mathbf{s_{i-1}})$$

"almost 0"
or
"almost 1"

function of input and state

-

LSTM

(Long short-term Memory)

- The LSTM is a specific combination of gates.
-

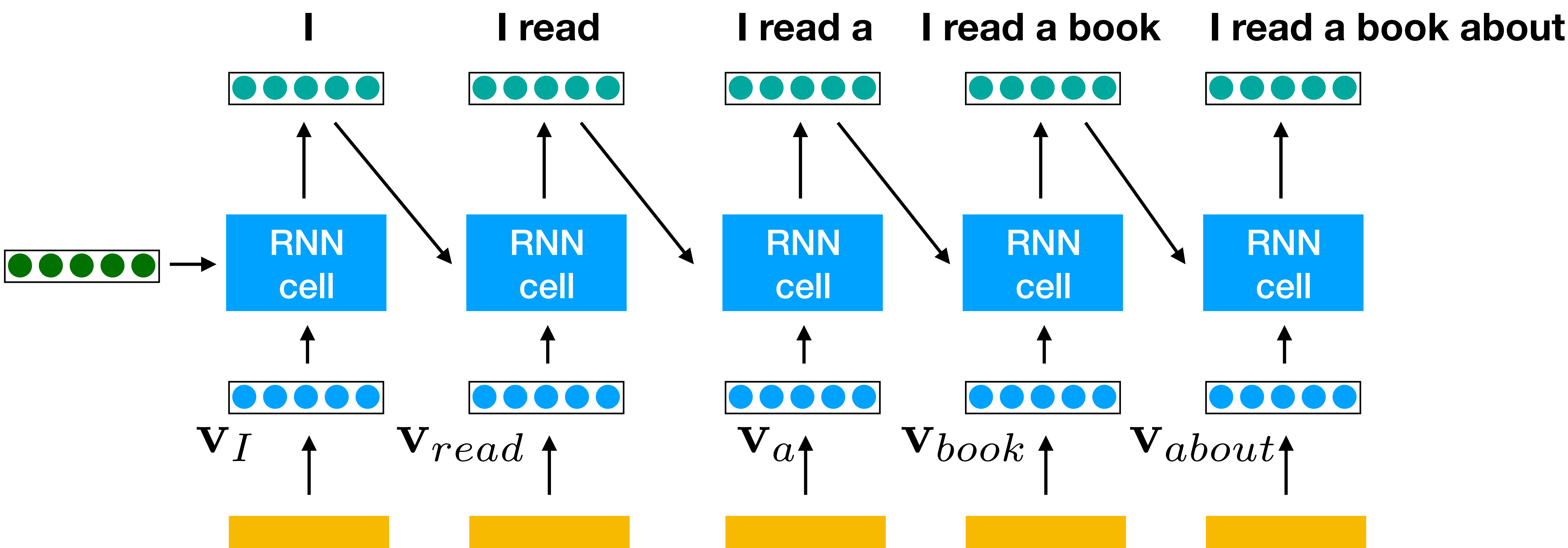
$$
\begin{aligned}
R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) =& [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} =& \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} =& \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} =& \sigma(\mathbf{W^{xi}} \cdot \mathbf{x_j} + \mathbf{W^{hi}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{f} =& \sigma(\mathbf{W^{xf}} \cdot \mathbf{x_j} + \mathbf{W^{hf}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{o} =& \sigma(\mathbf{W^{xo}} \cdot \mathbf{x_j} + \mathbf{W^{ho}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{g} =& \tanh(\mathbf{W^{xg}} \cdot \mathbf{x_j} + \mathbf{W^{hg}} \cdot \mathbf{h_{j-1}})
\end{aligned}
$$

$$
O_{LSTM}(\mathbf{s_j}) = O_{LSTM}([\mathbf{c_j}; \mathbf{h_j}]) = \mathbf{h_j}
$$

Combining Vectors

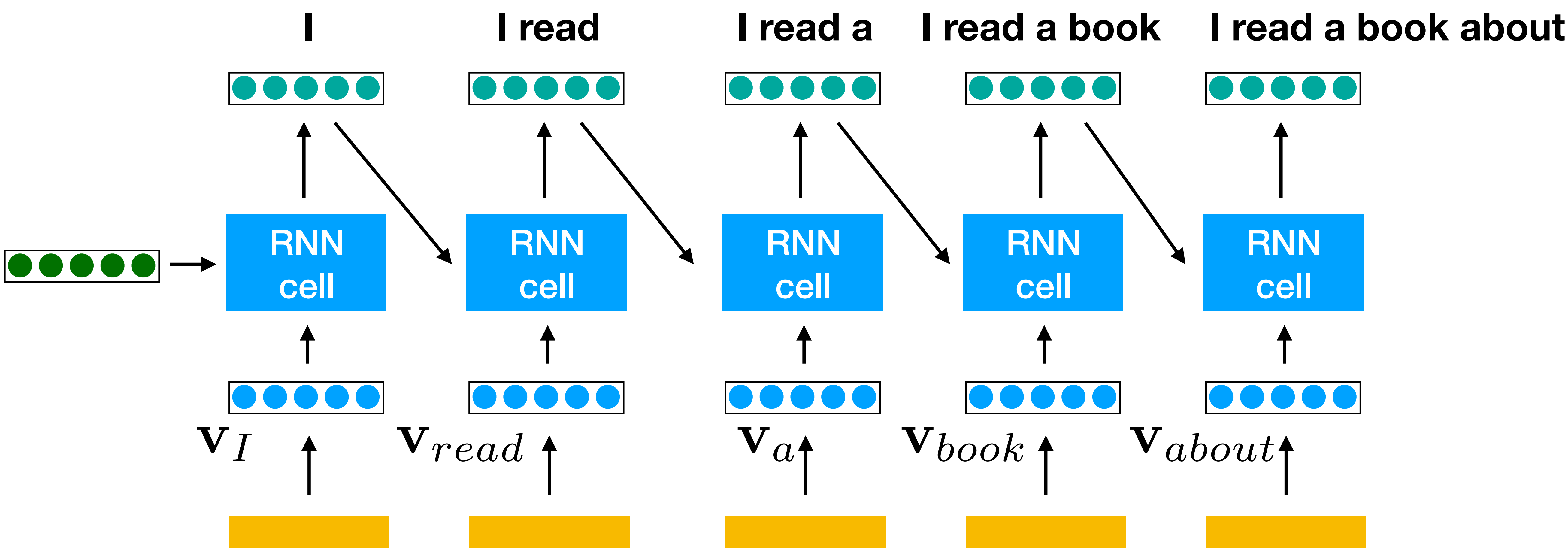
Recurrent Neural Network: RNN

$$
\begin{aligned}
R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) &= [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} &= \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} &= \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} &= \sigma(\mathbf{W^{xi}} \cdot \mathbf{x_j} + \mathbf{W^{hi}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{f} &= \sigma(\mathbf{W^{xf}} \cdot \mathbf{x_j} + \mathbf{W^{hf}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{o} &= \sigma(\mathbf{W^{xo}} \cdot \mathbf{x_j} + \mathbf{W^{ho}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{g} &= \tanh(\mathbf{W^{xg}} \cdot \mathbf{x_j} + \mathbf{W^{hg}} \cdot \mathbf{h_{j-1}})
\end{aligned}
$$



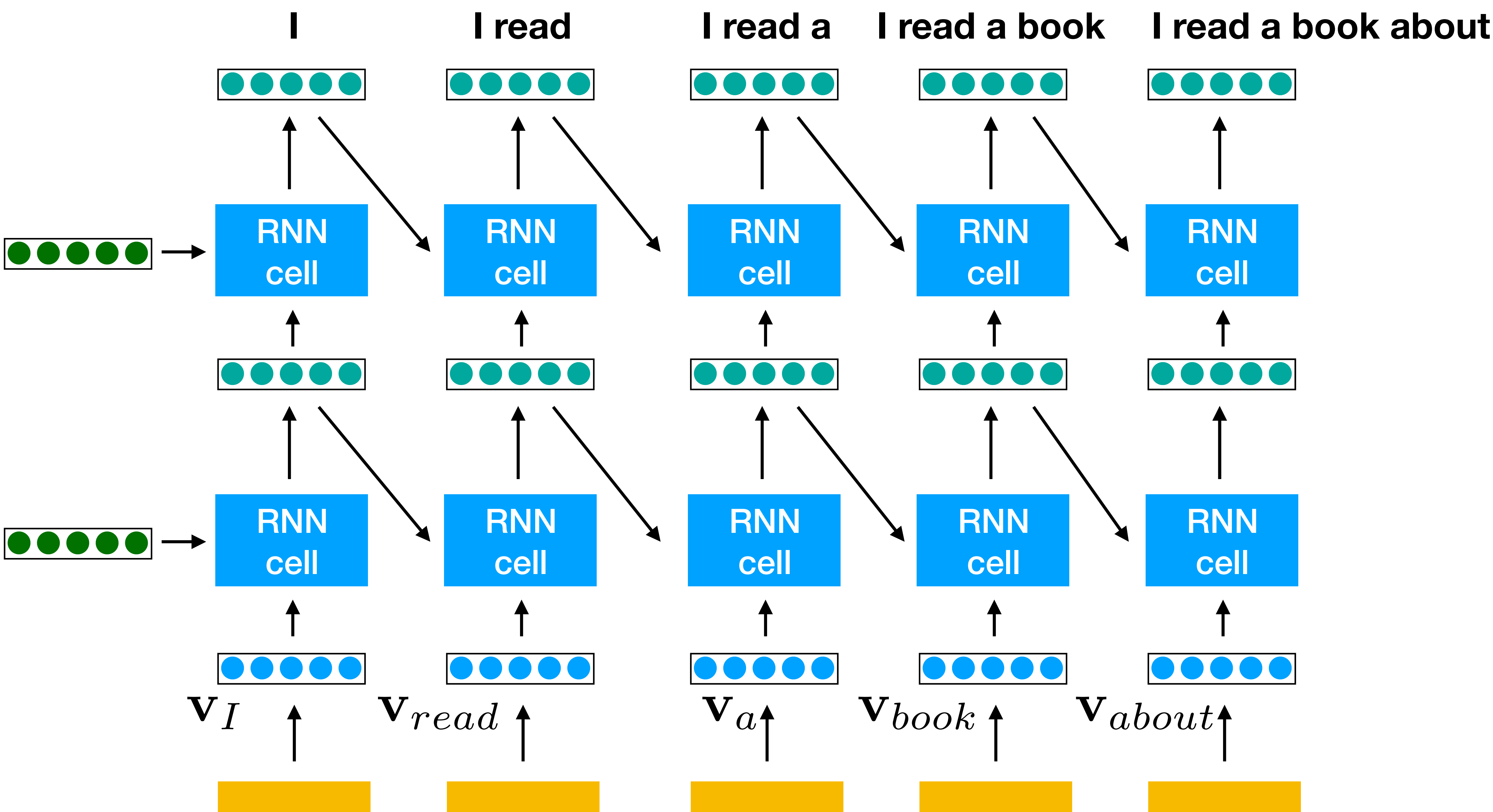
Combining Vectors

Recurrent Neural Network: RNN



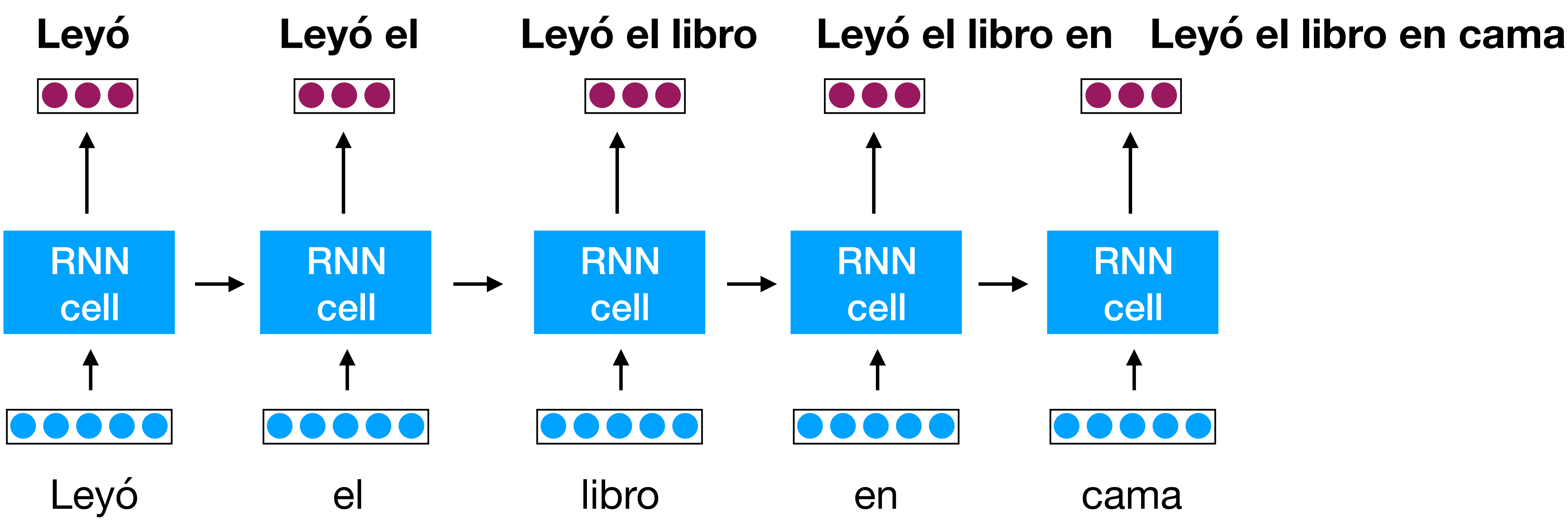
Combining Vectors

multi-layer RNN



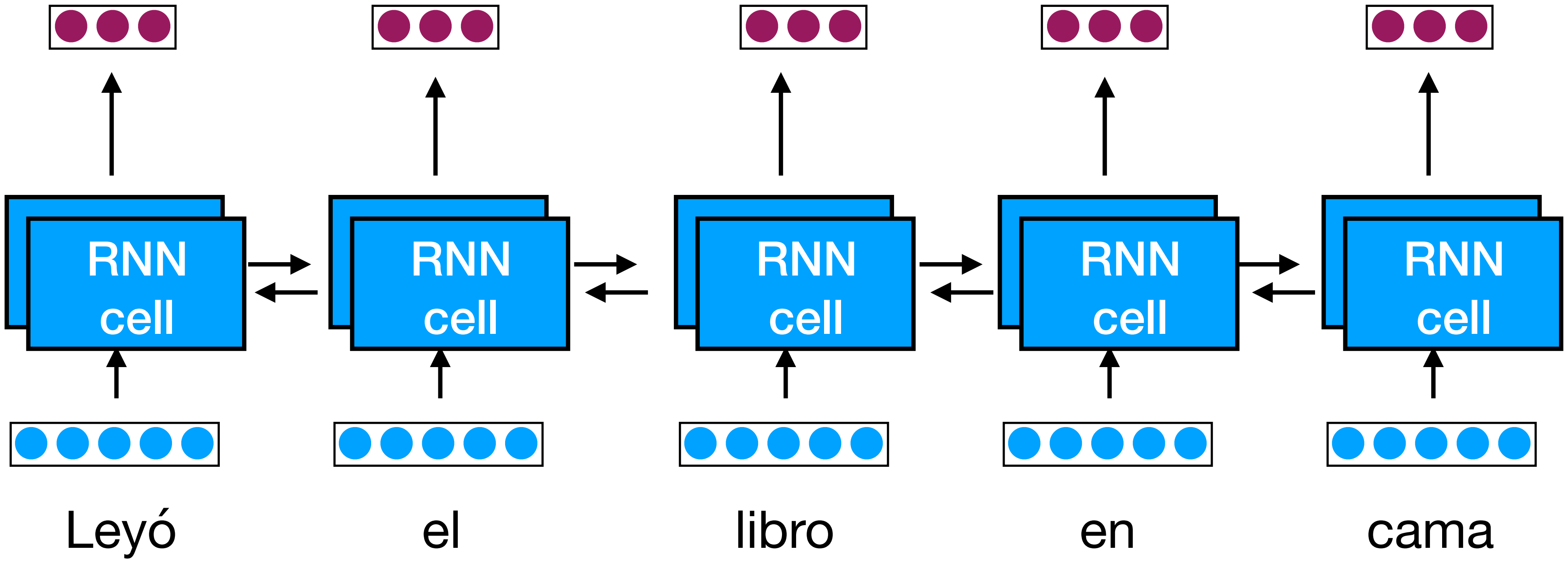
Bi-RNN

keep intermediate vectors



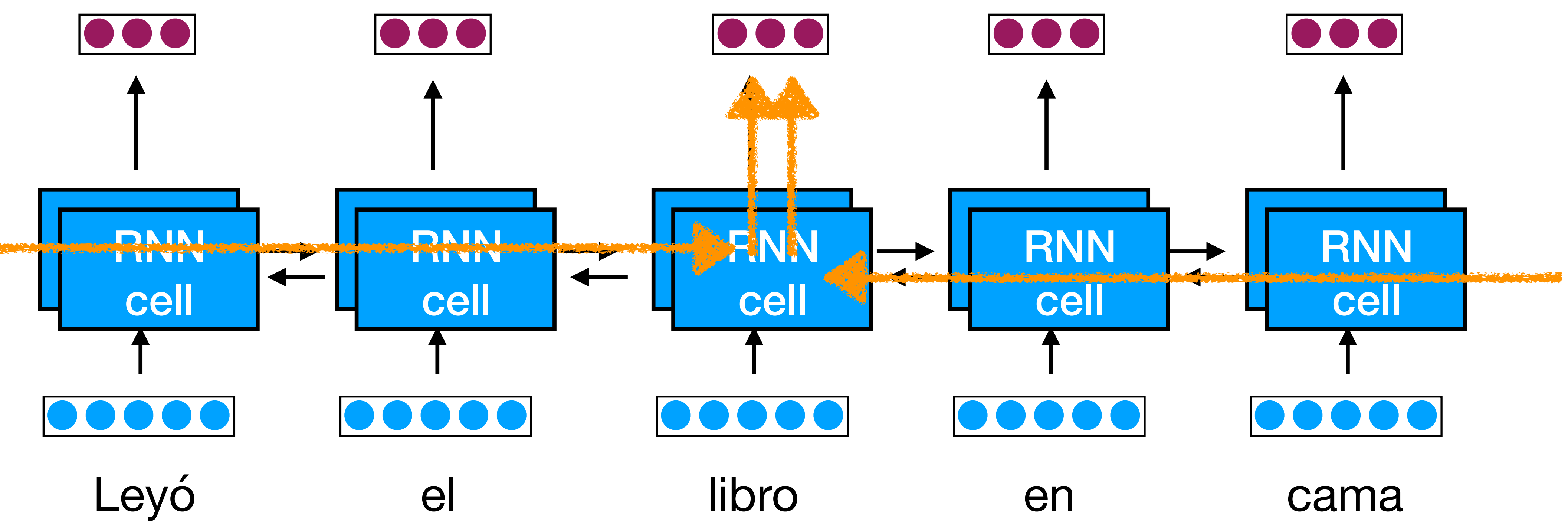
Bi-RNN

add right-to-left RNN (bi-RNN)



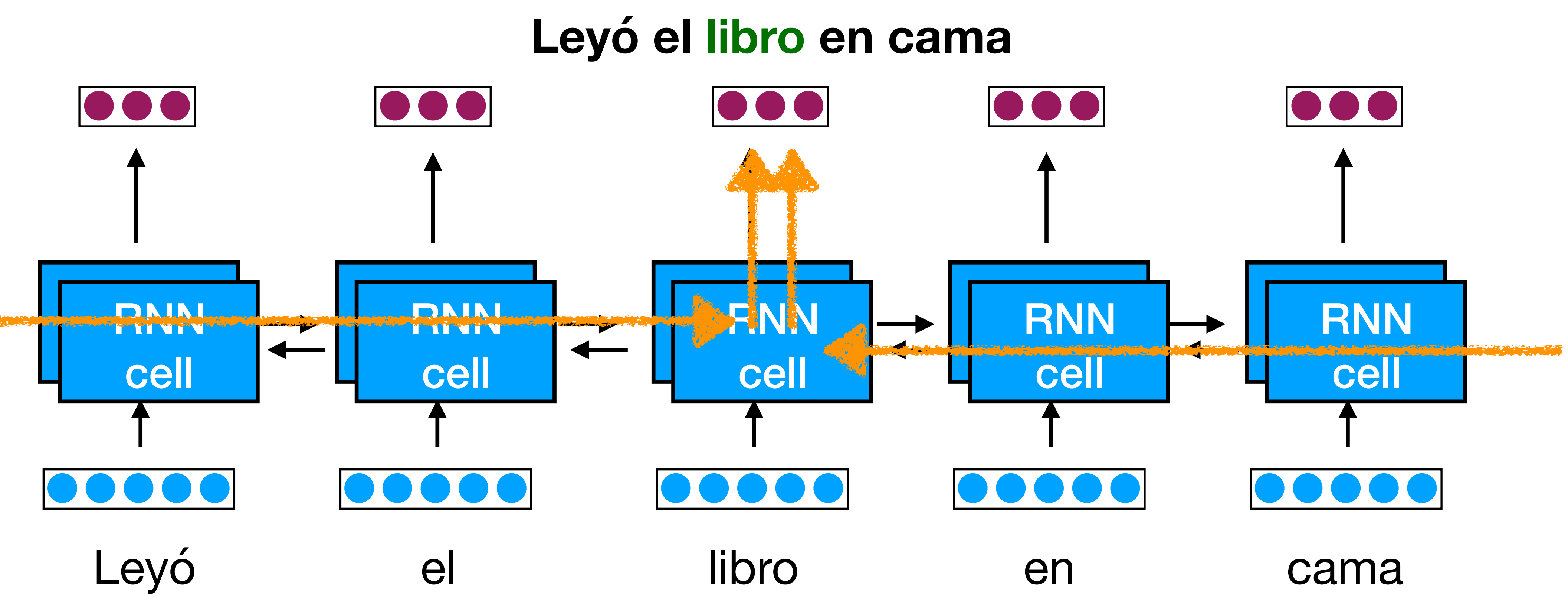
Bi-RNN

add right-to-left RNN (bi-RNN)



Bi-RNN

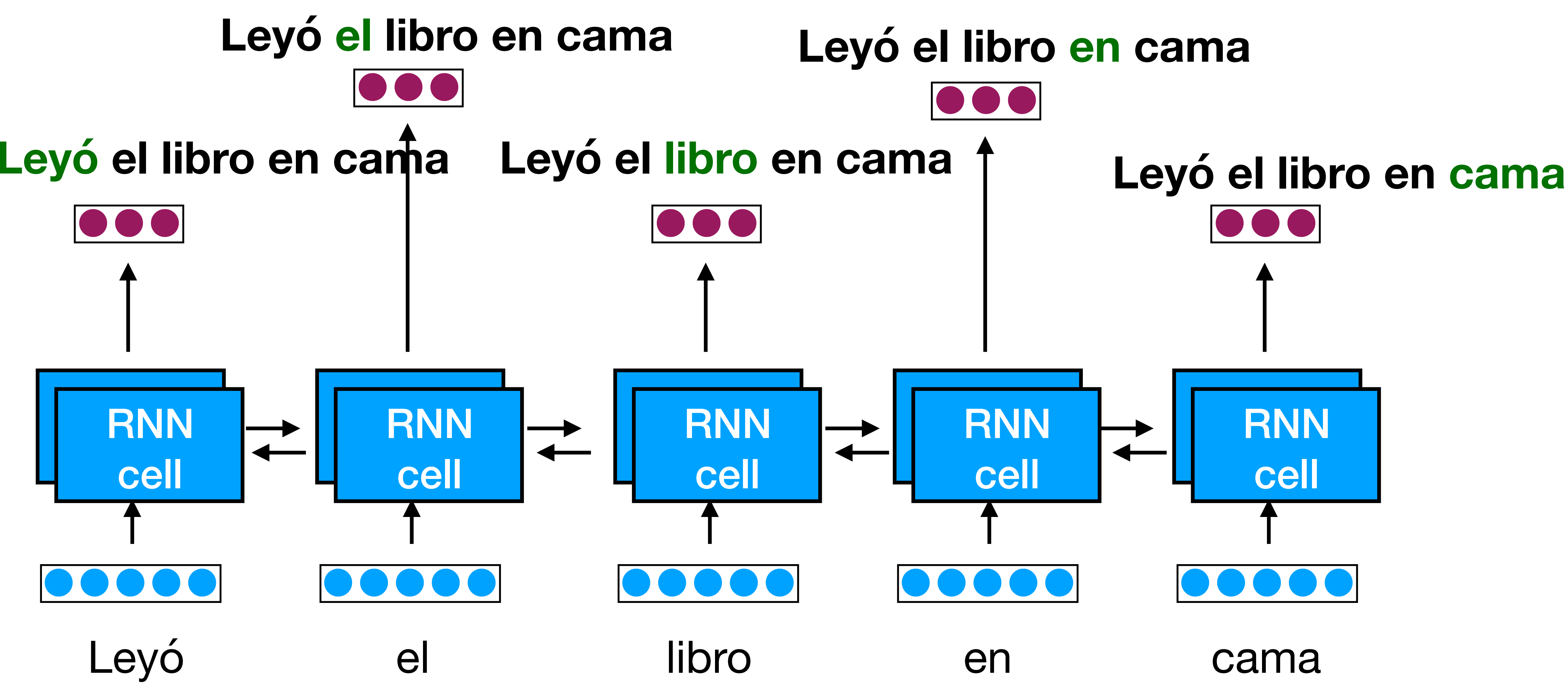
add right-to-left RNN (bi-RNN)



Bi-RNN

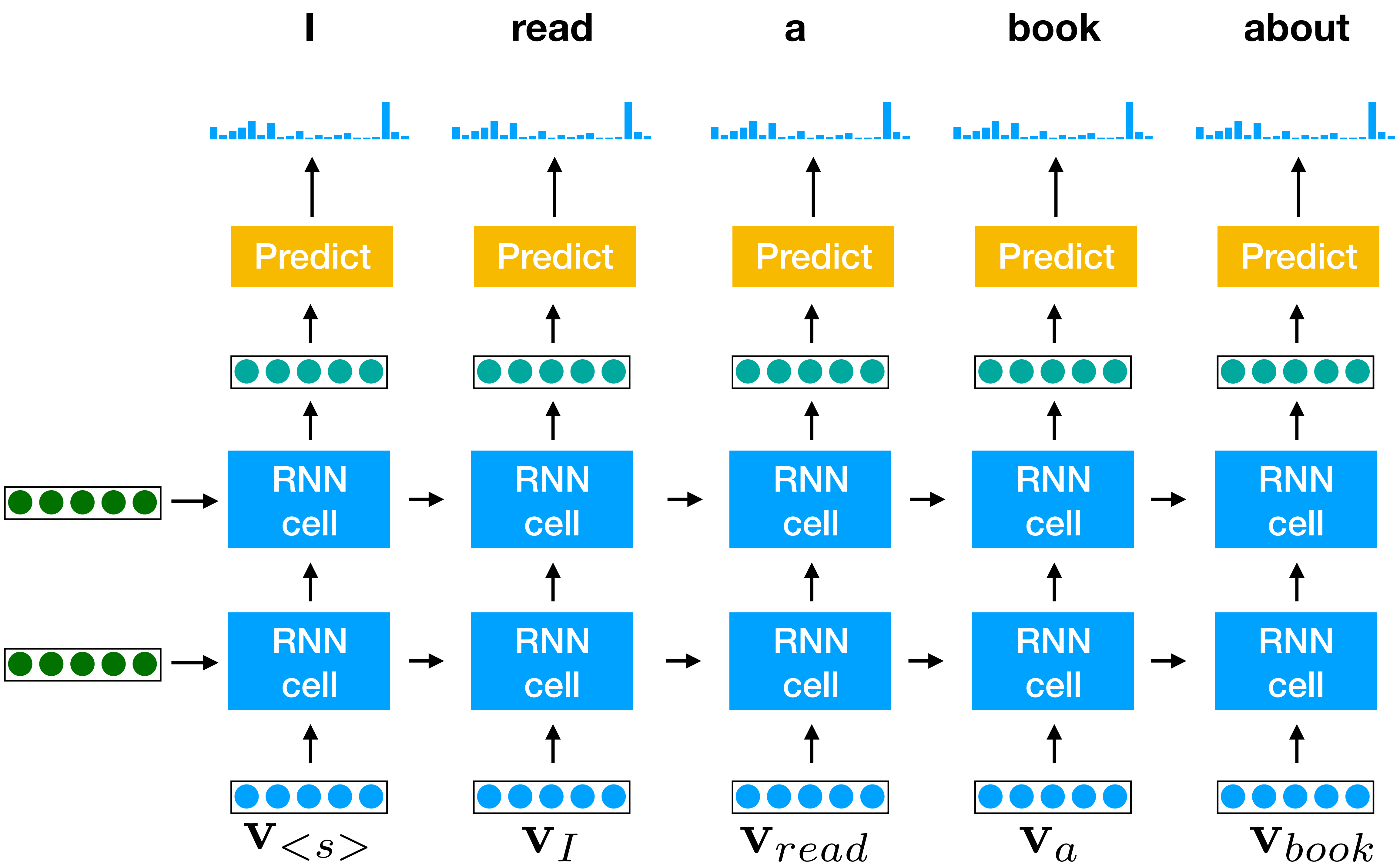
a representation of a word in context.

add right-to-left RNN (bi-RNN)



Training

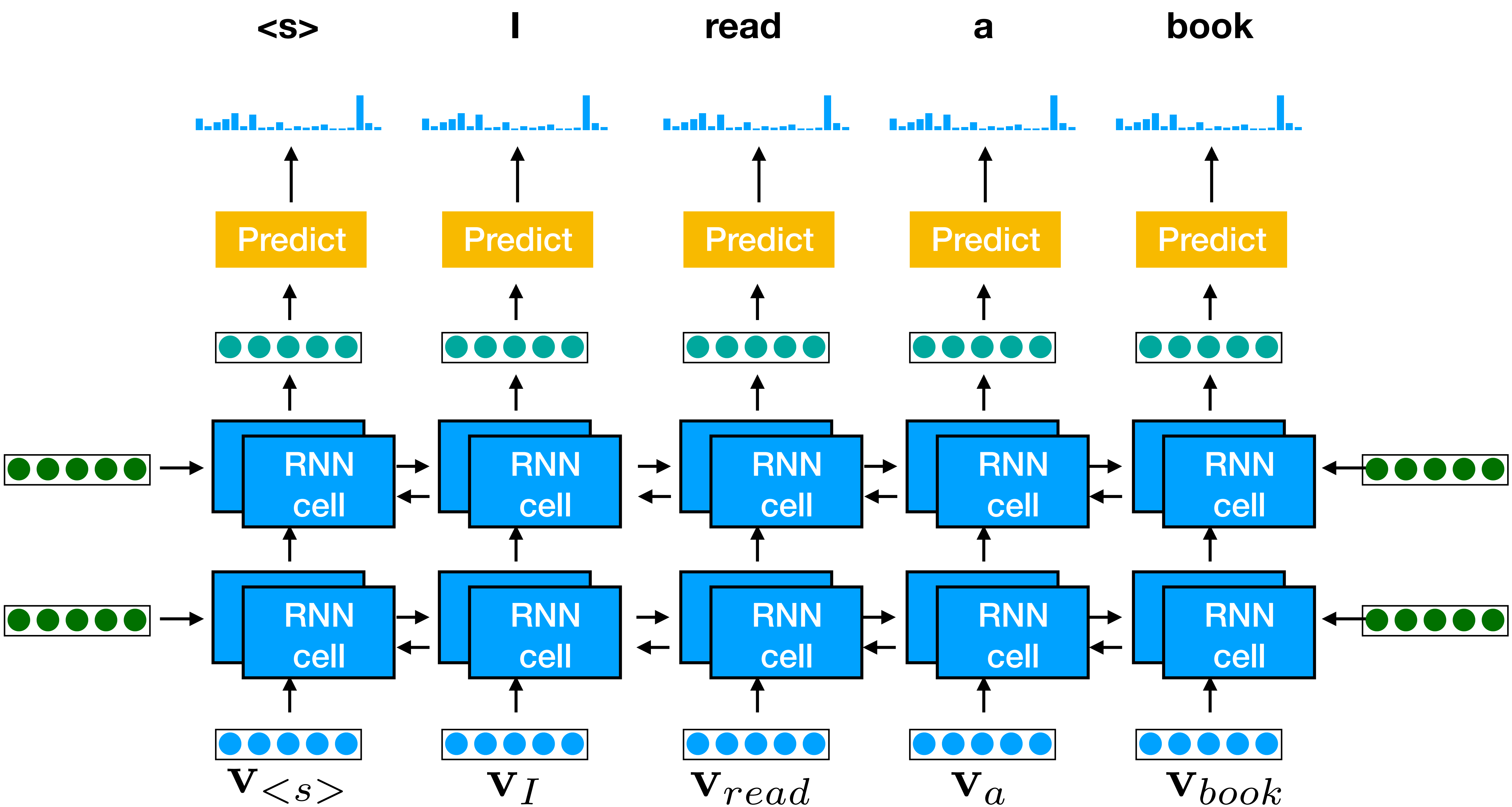
RNN



Training

bi-RNN

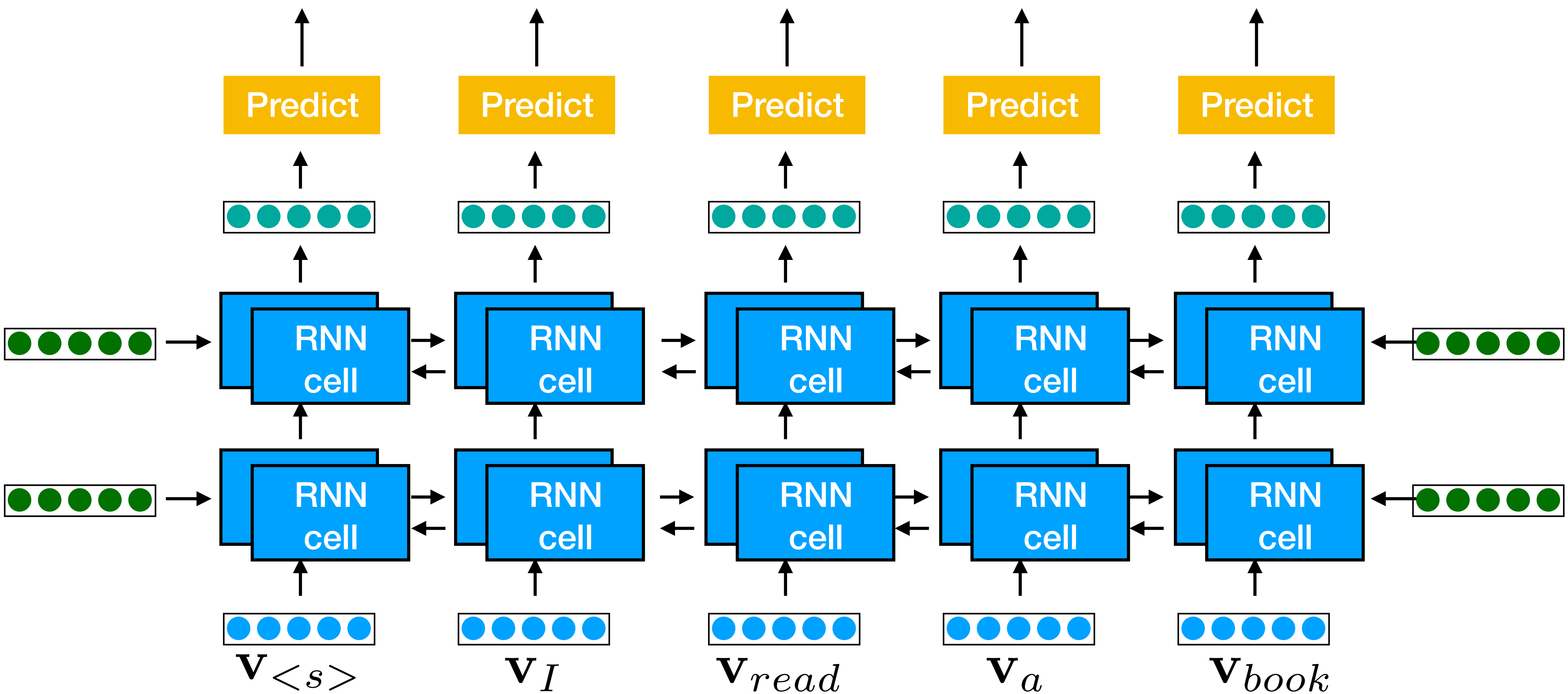
see a problem?



Training

bi-RNN

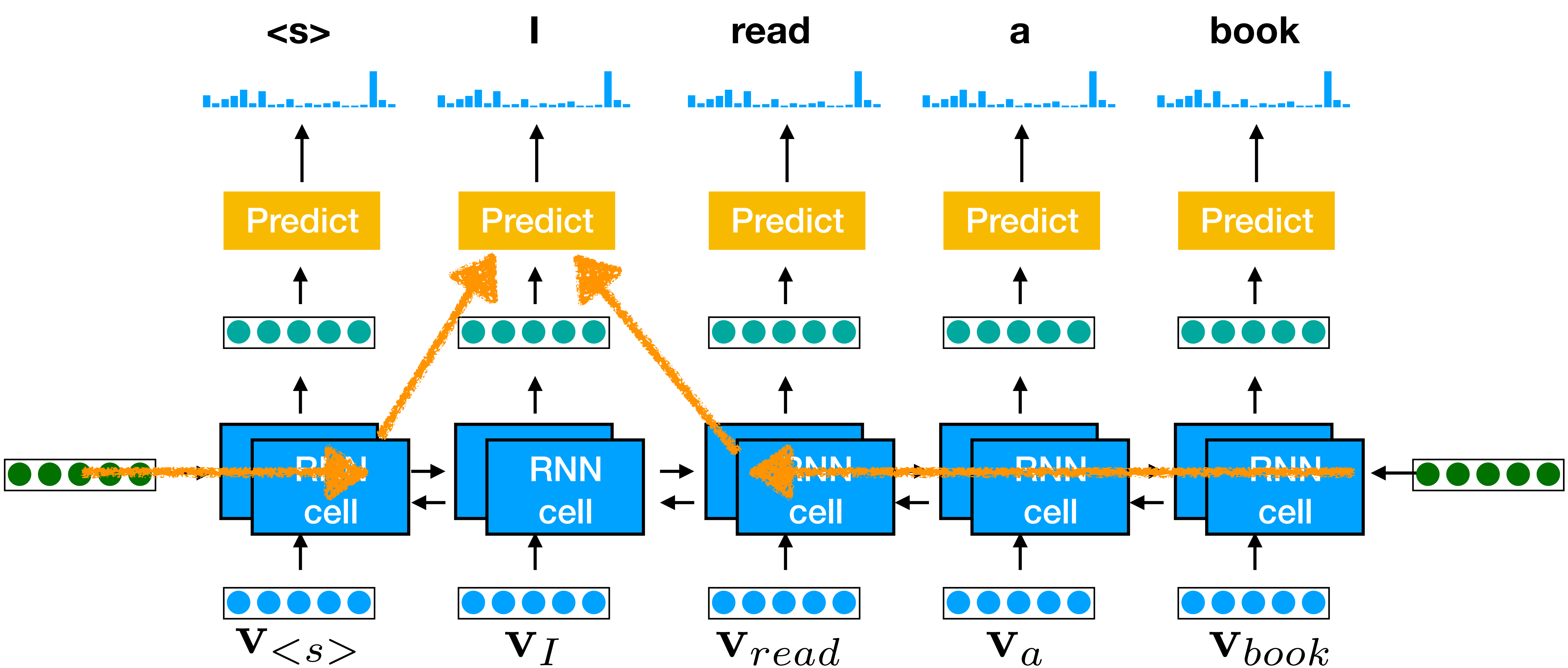
solution 1:
don't predict words.
predict tags. use as part fo larger network.



Training

bi-RNN

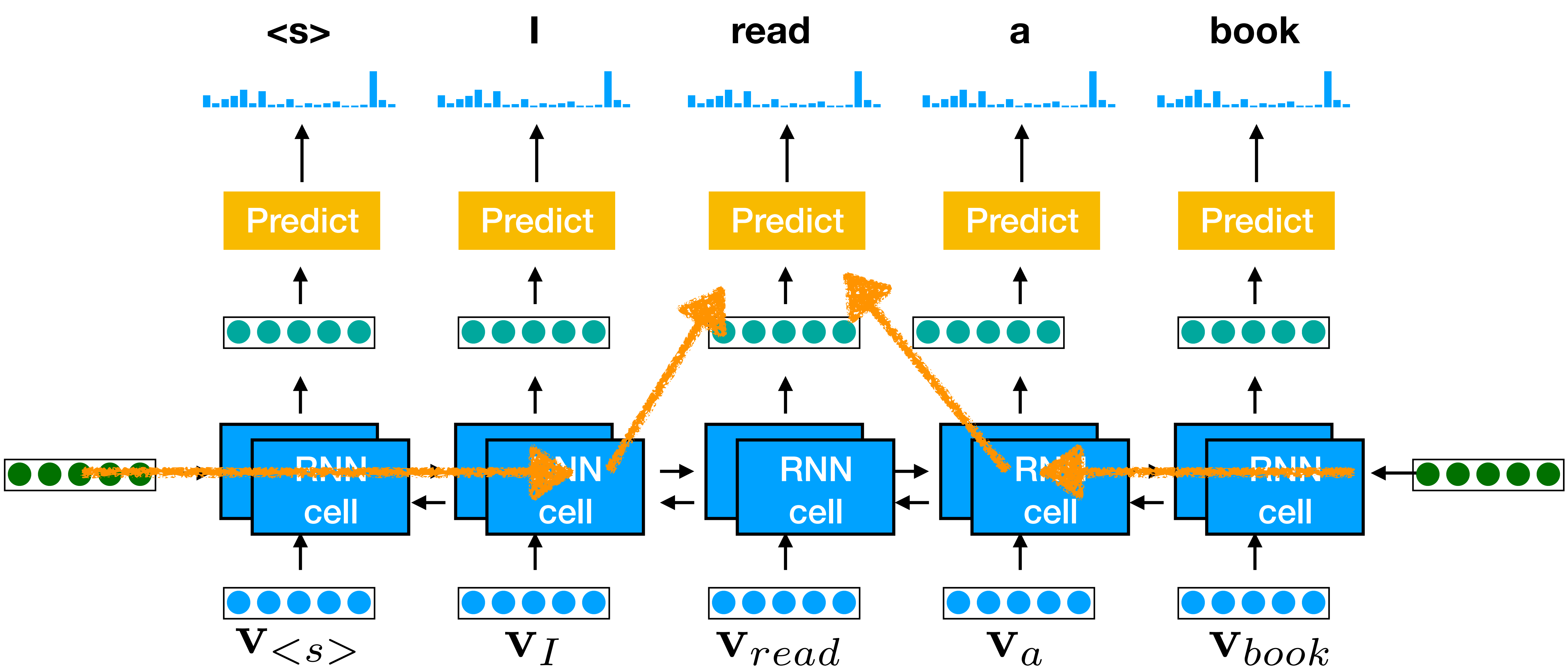
solution 2:
single layer. skip word



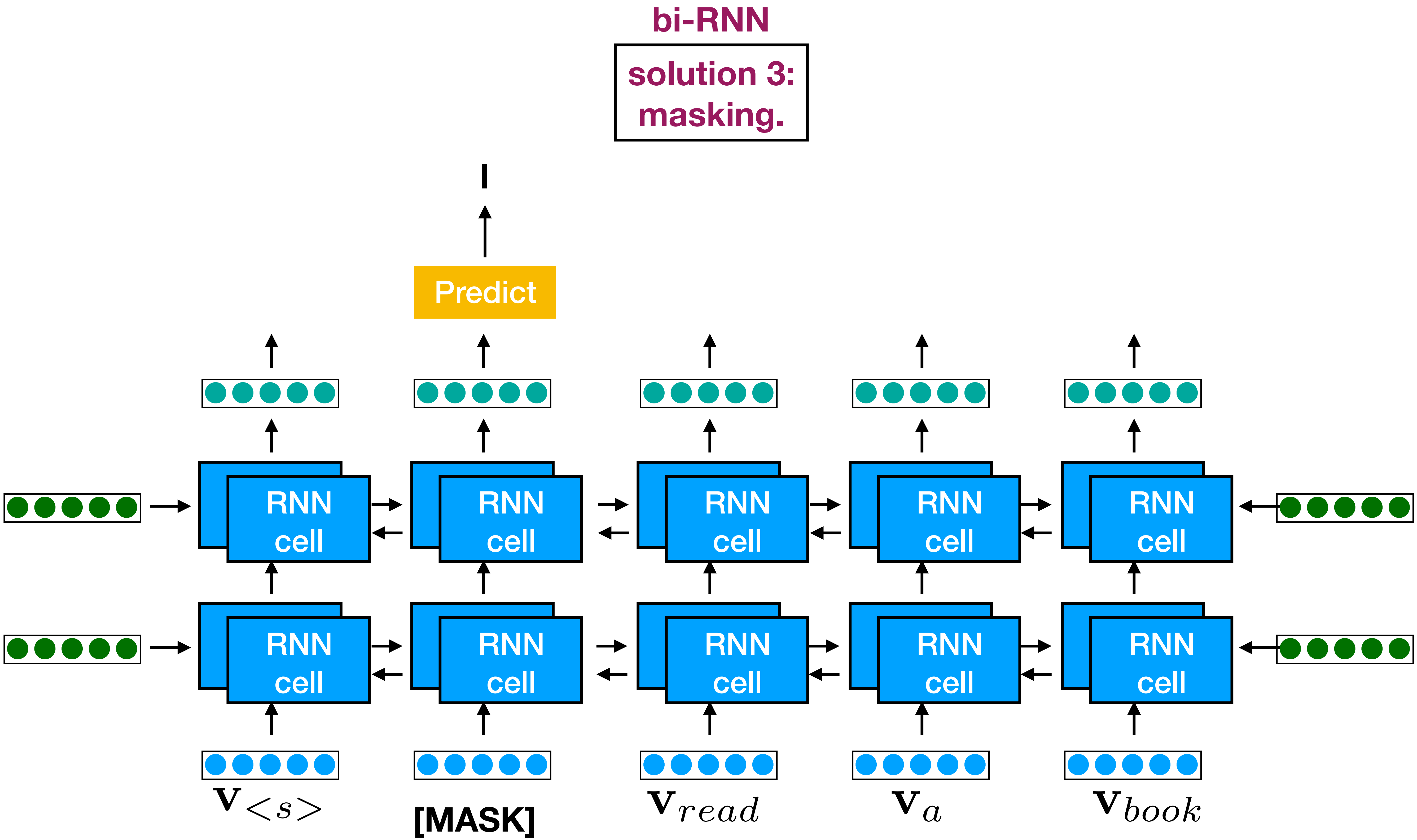
Training

bi-RNN

solution 2:
single layer. skip word

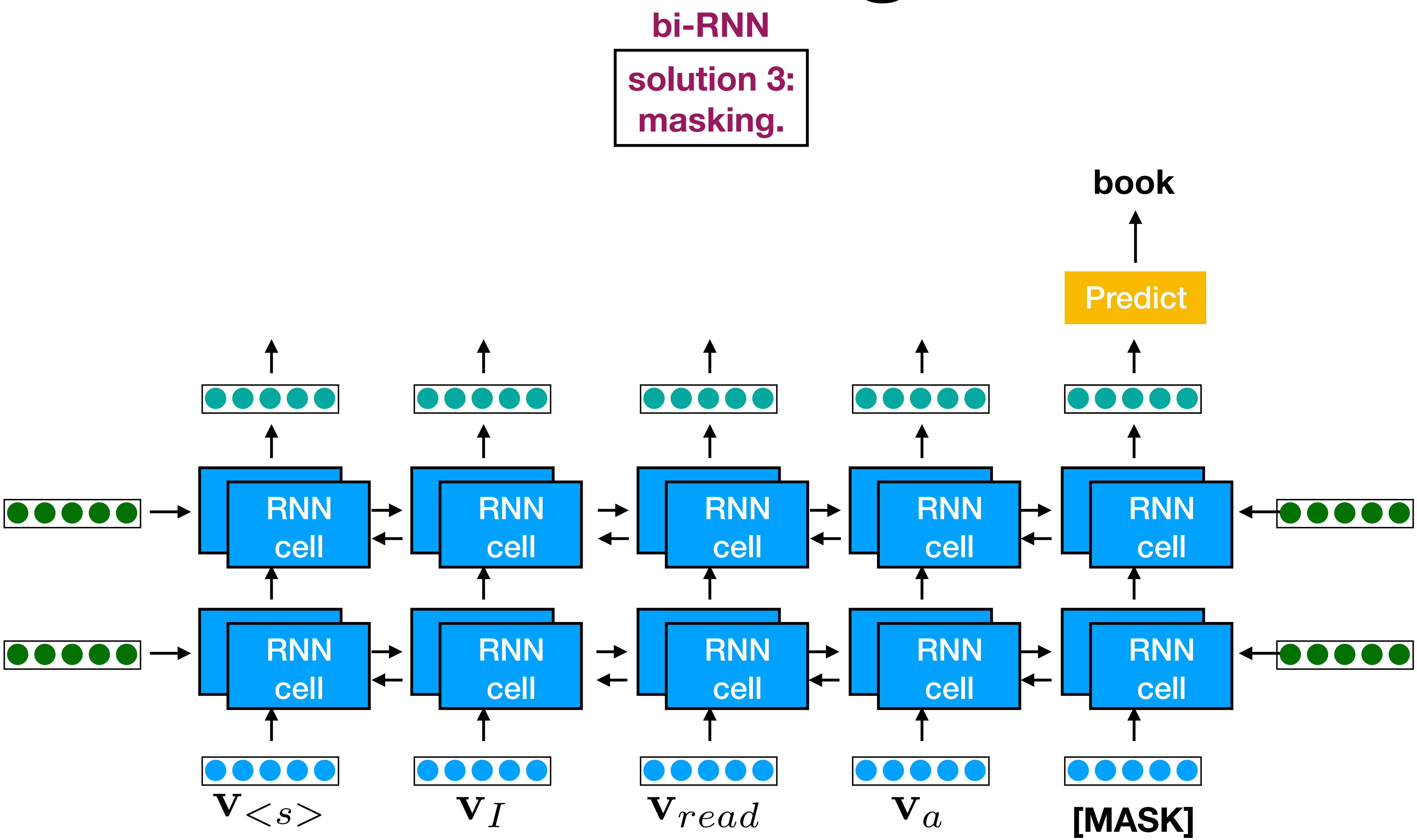


Training

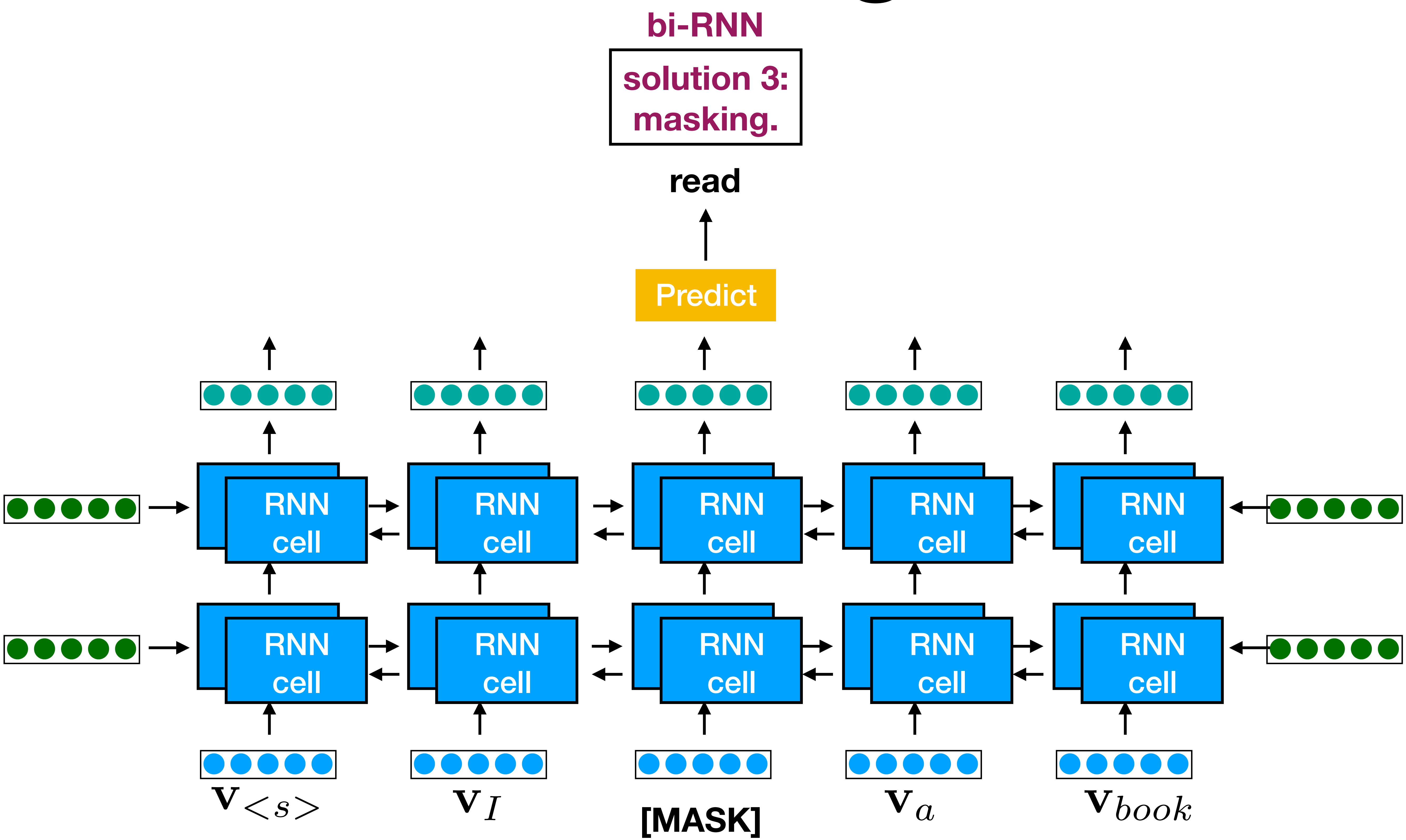


$\mathbf{v}_{<s>}$ $\mathbf{v}_{read}$ $\mathbf{v}_{a}$ $\mathbf{v}_{book}$

Training

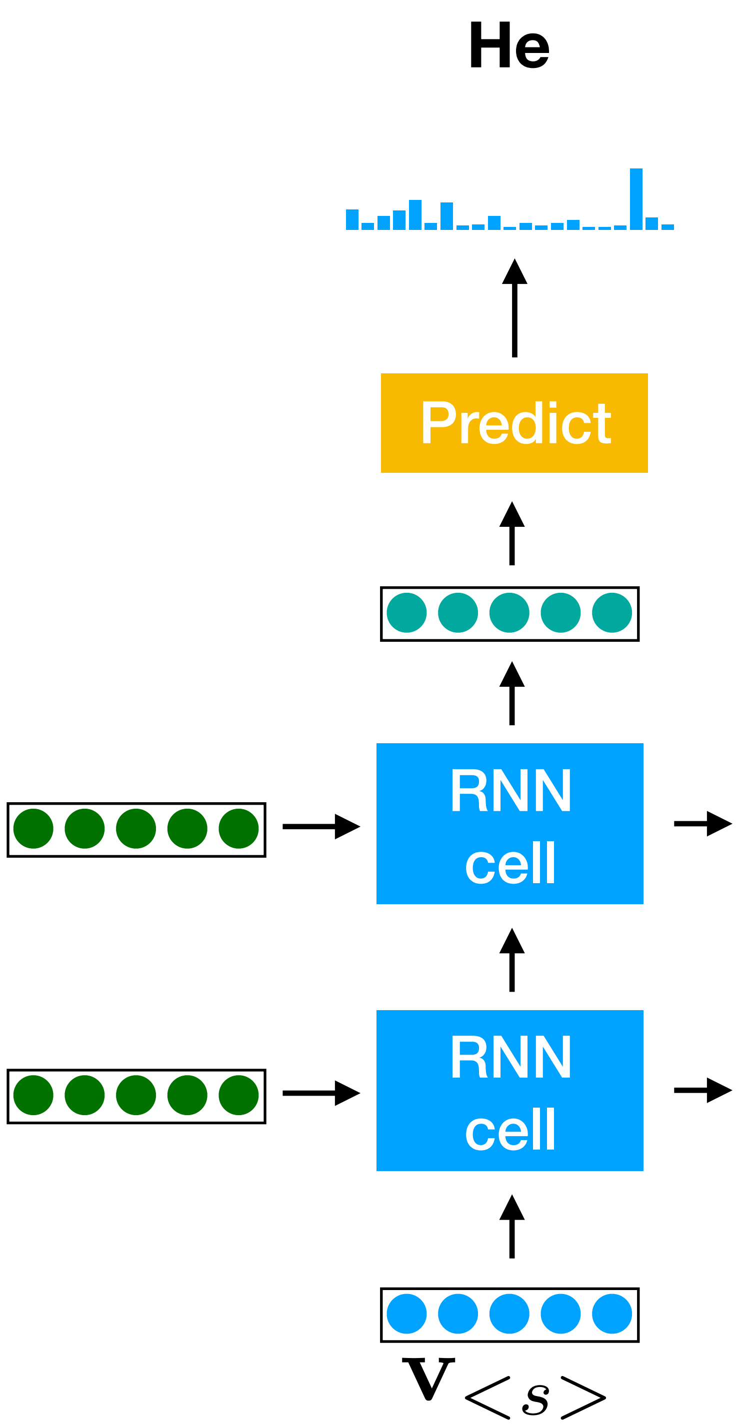

bi-RNN
solution 3:
masking.
book
Predict
RNN cell
$\mathbf{v}_{<s>}$
$\mathbf{v}_{I}$
$\mathbf{v}_{read}$
$\mathbf{v}_{a}$
[MASK]

Training


bi-RNN
solution 3:
masking.
read
Predict
RNN cell
$\mathbf{v}_{<s>}$
$\mathbf{v}_I$
[MASK]
$\mathbf{v}_a$
$\mathbf{v}_{book}$

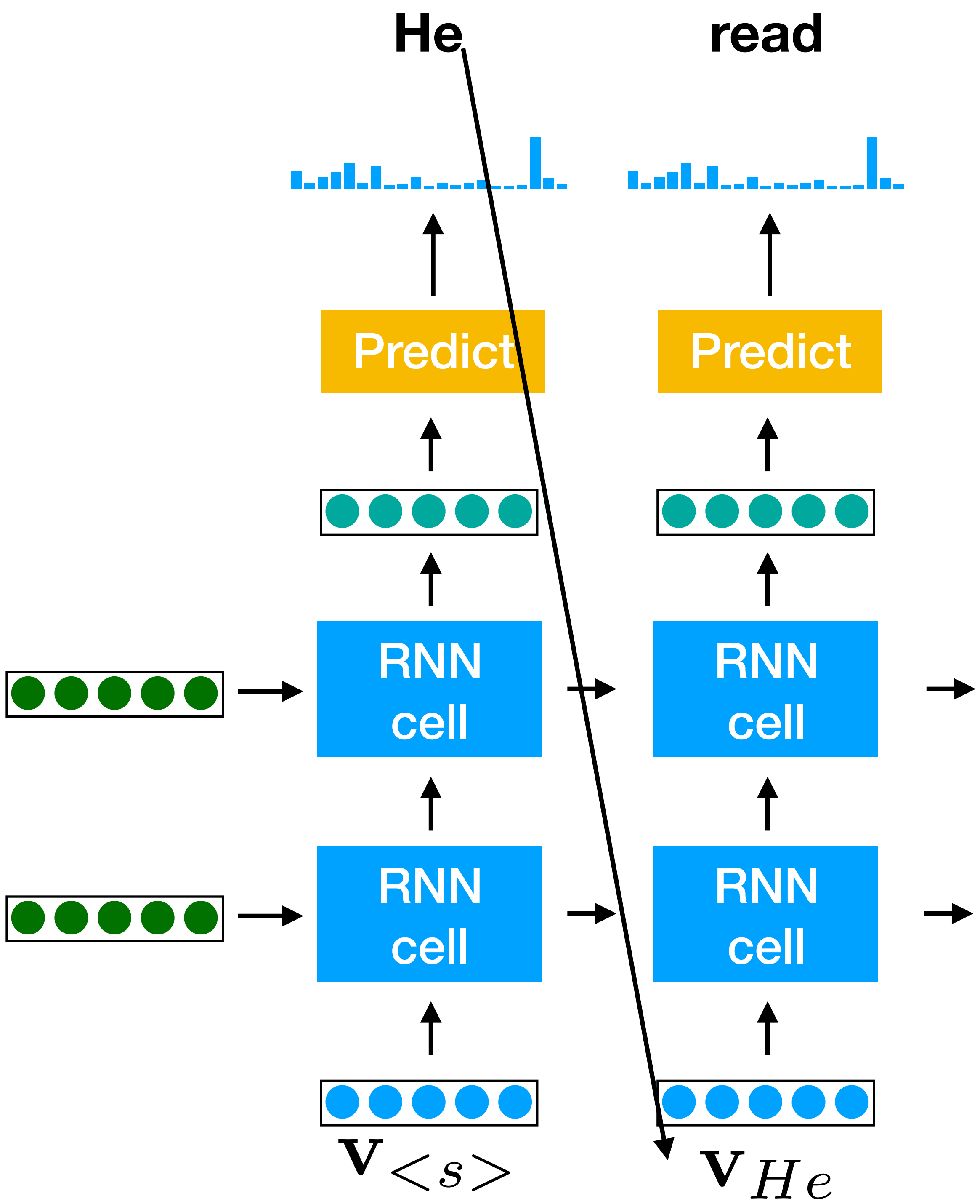
Generation

from RNN



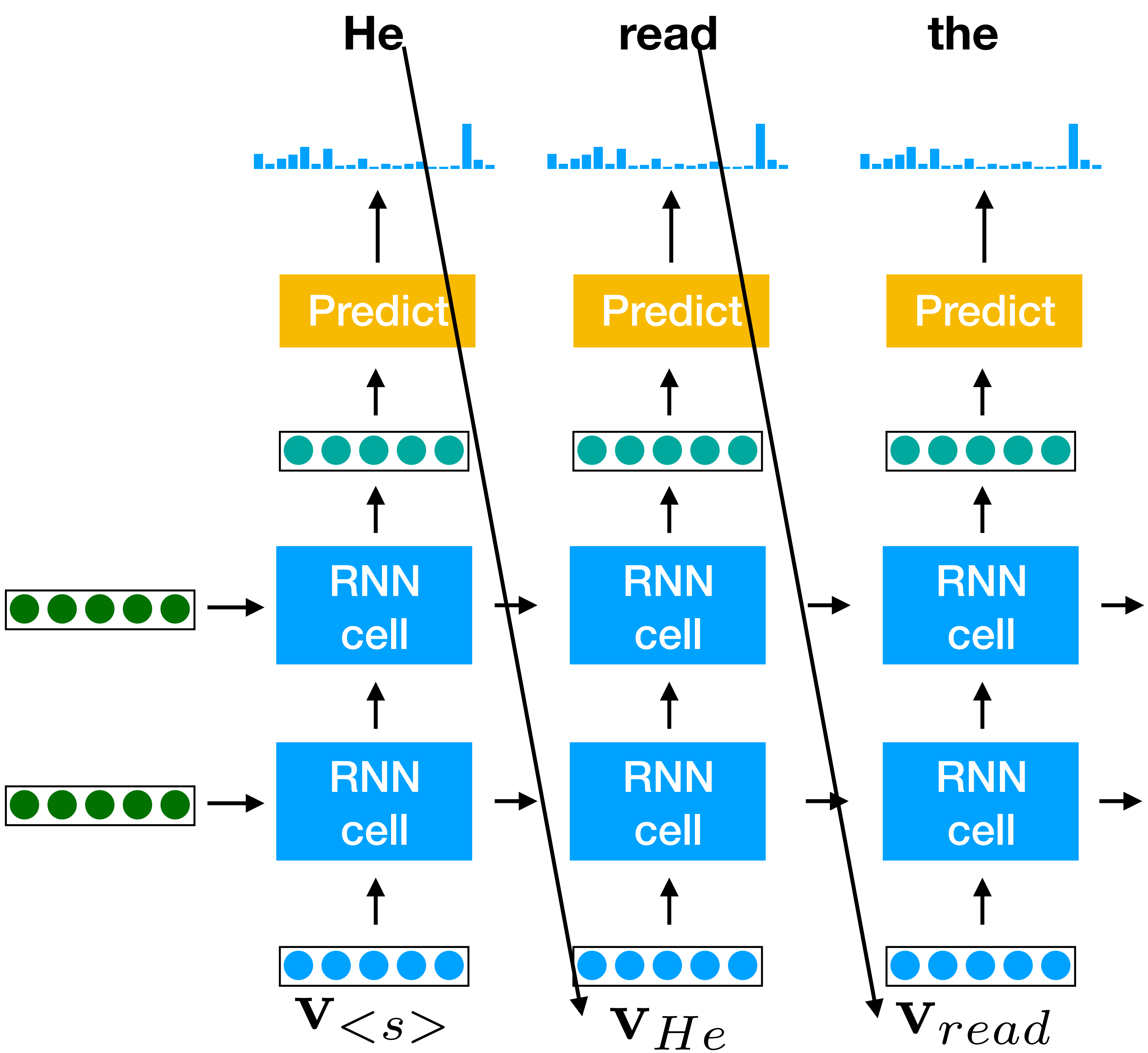
Generation

from RNN



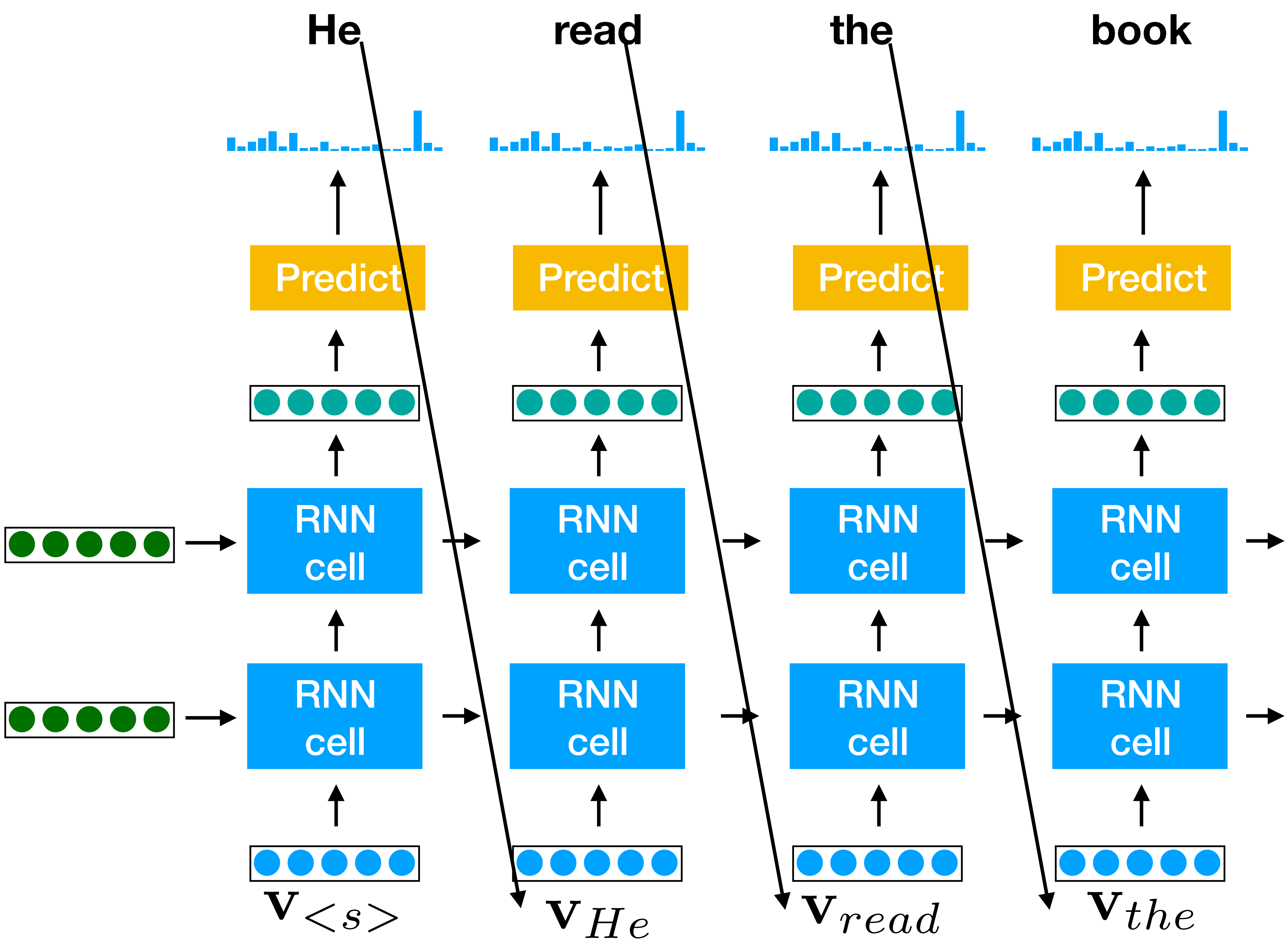
Generation

from RNN



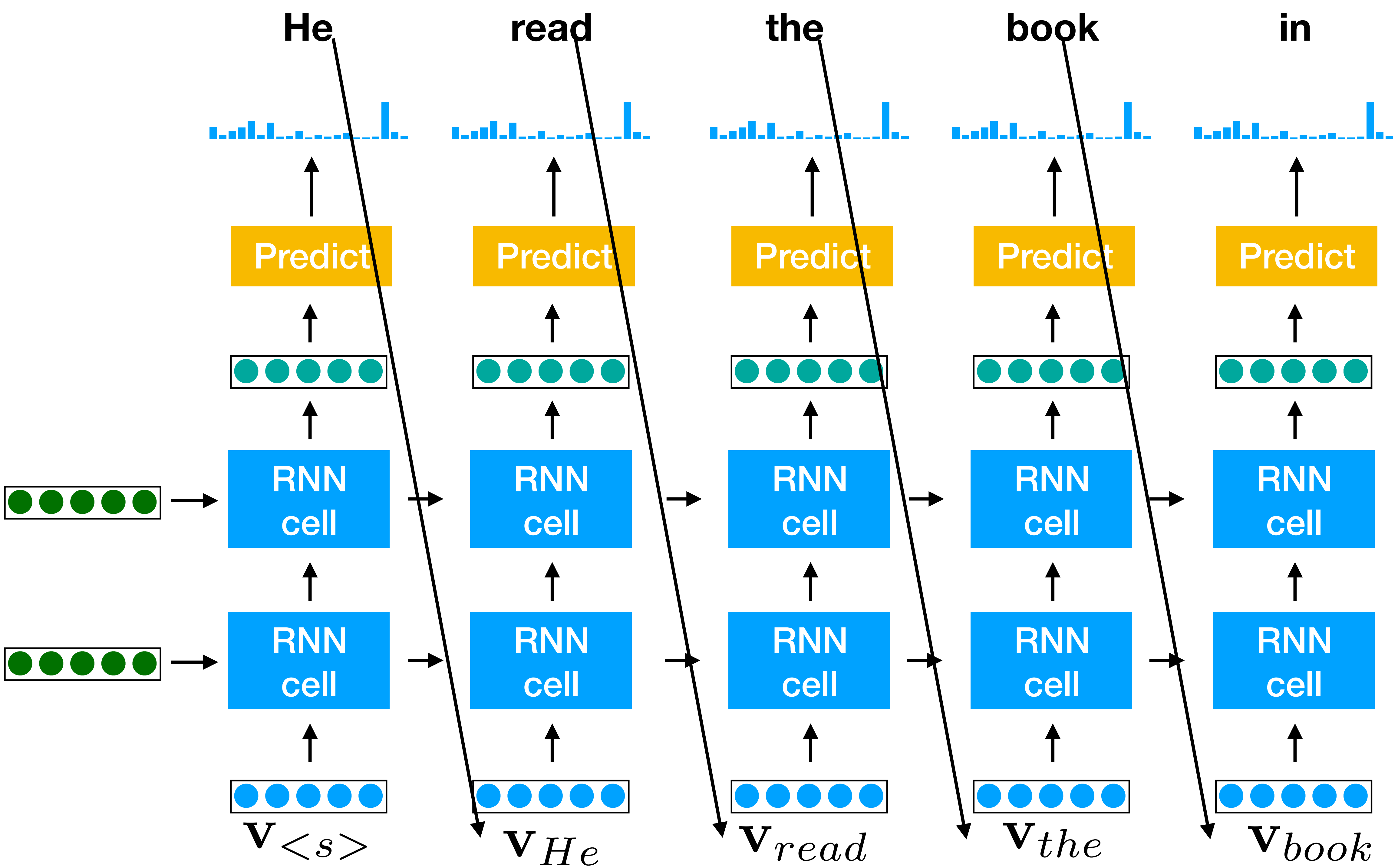
Generation

from RNN



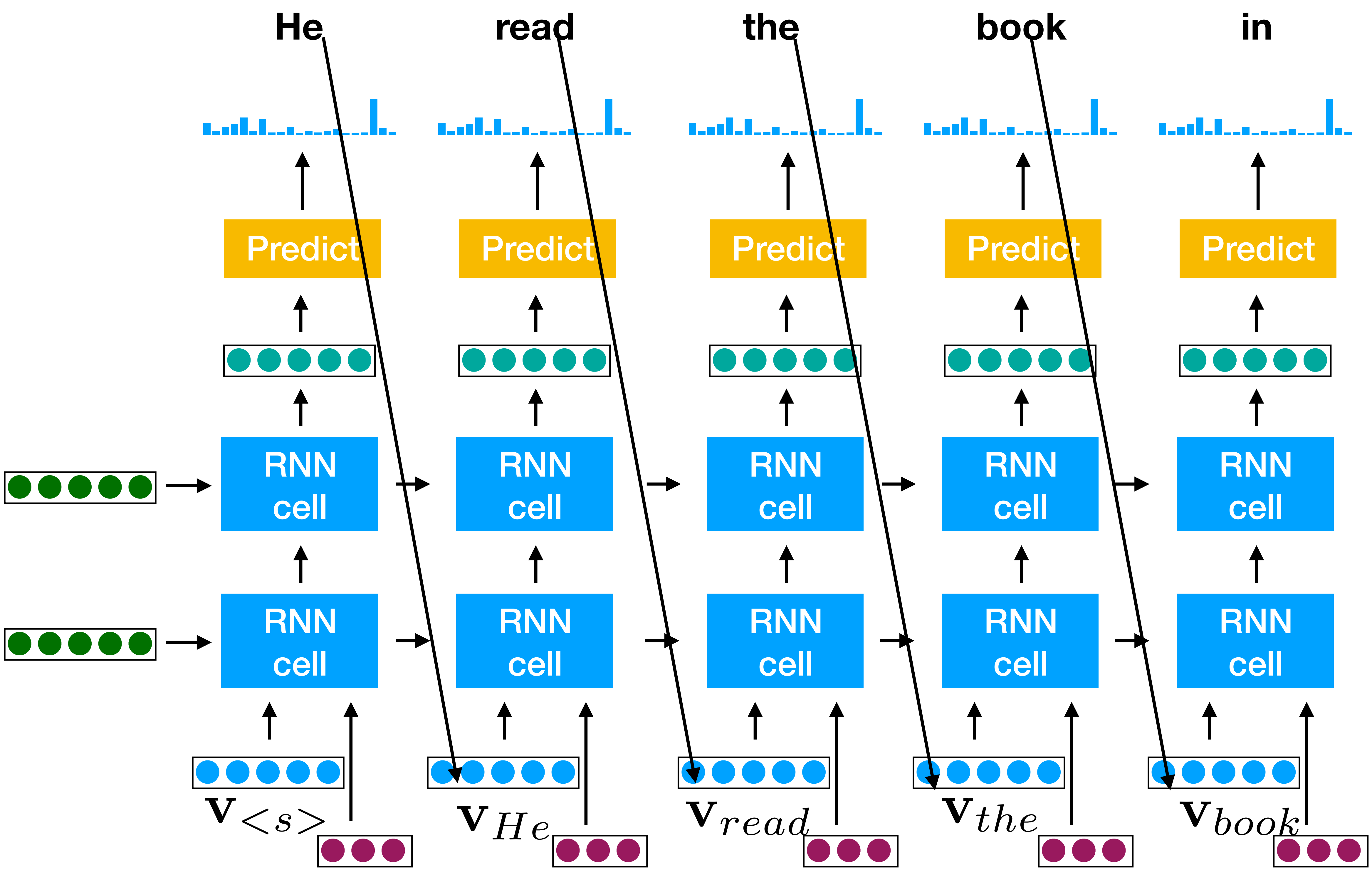
Generation

from RNN

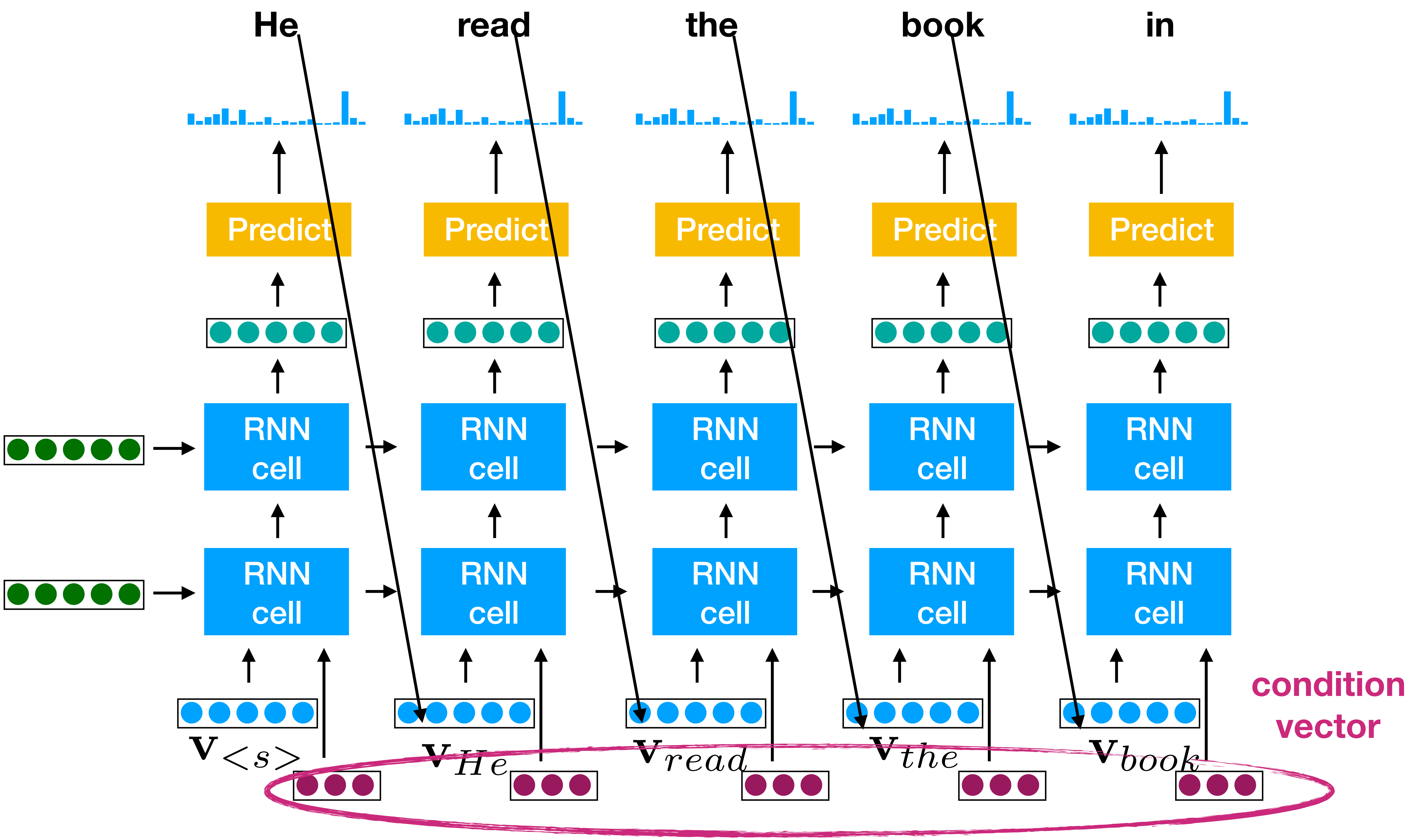


Conditioned Generation

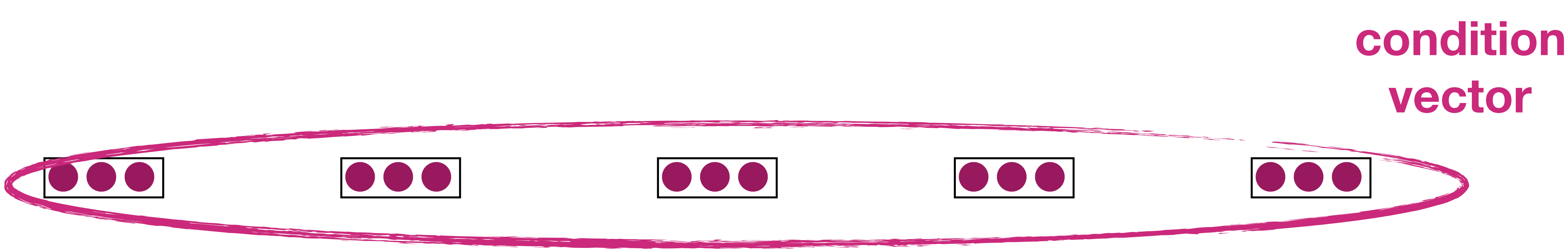
from RNN



Conditioned Generation



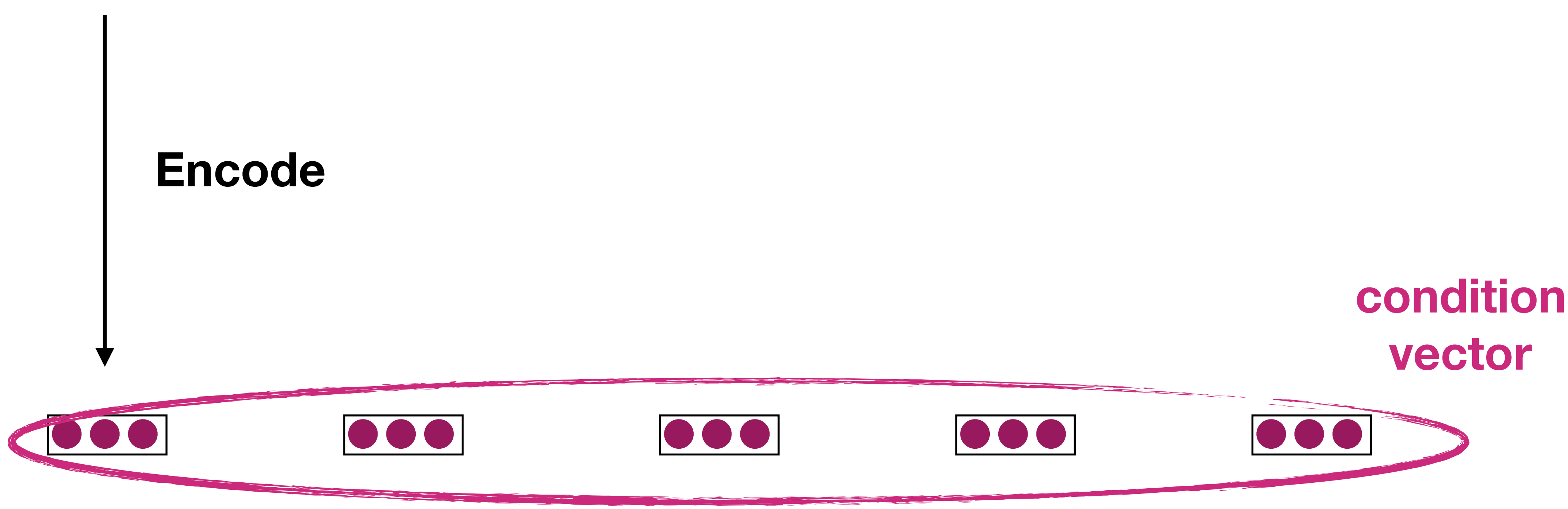
Conditioned Generation



Conditioned Generation

Table

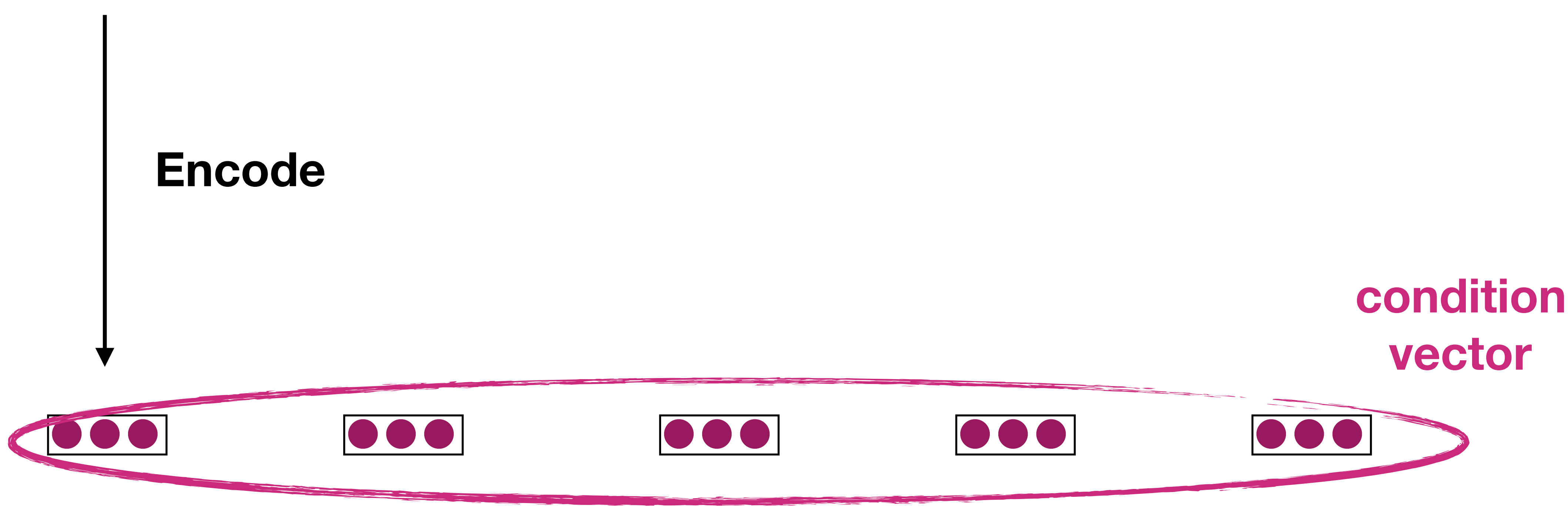
Name	Triton 52
EcoRating	A+
Family	L7



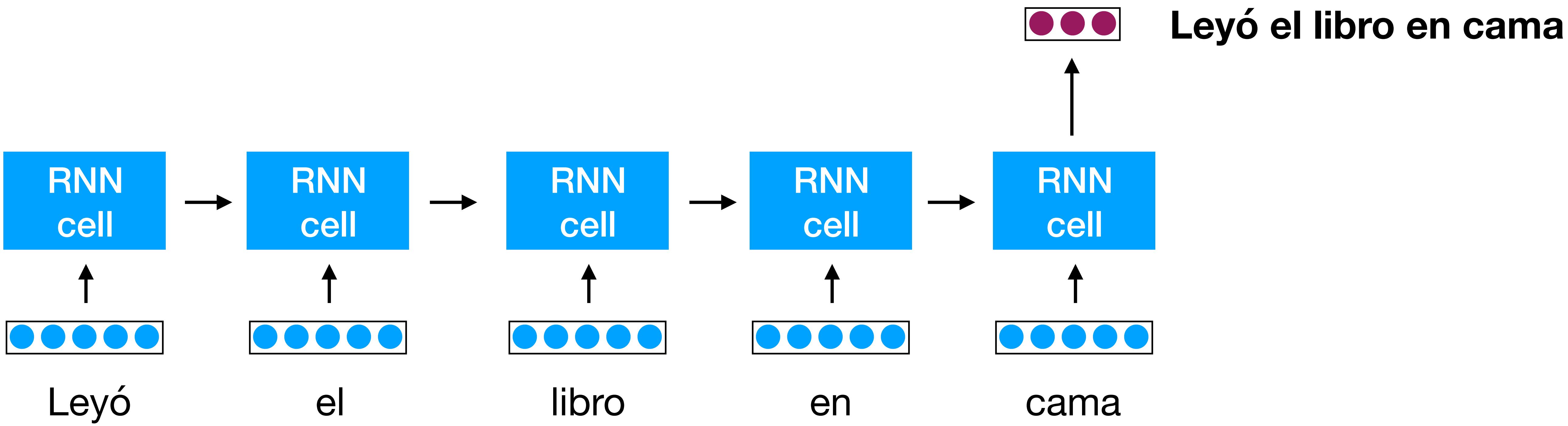
Conditioned Generation

Text

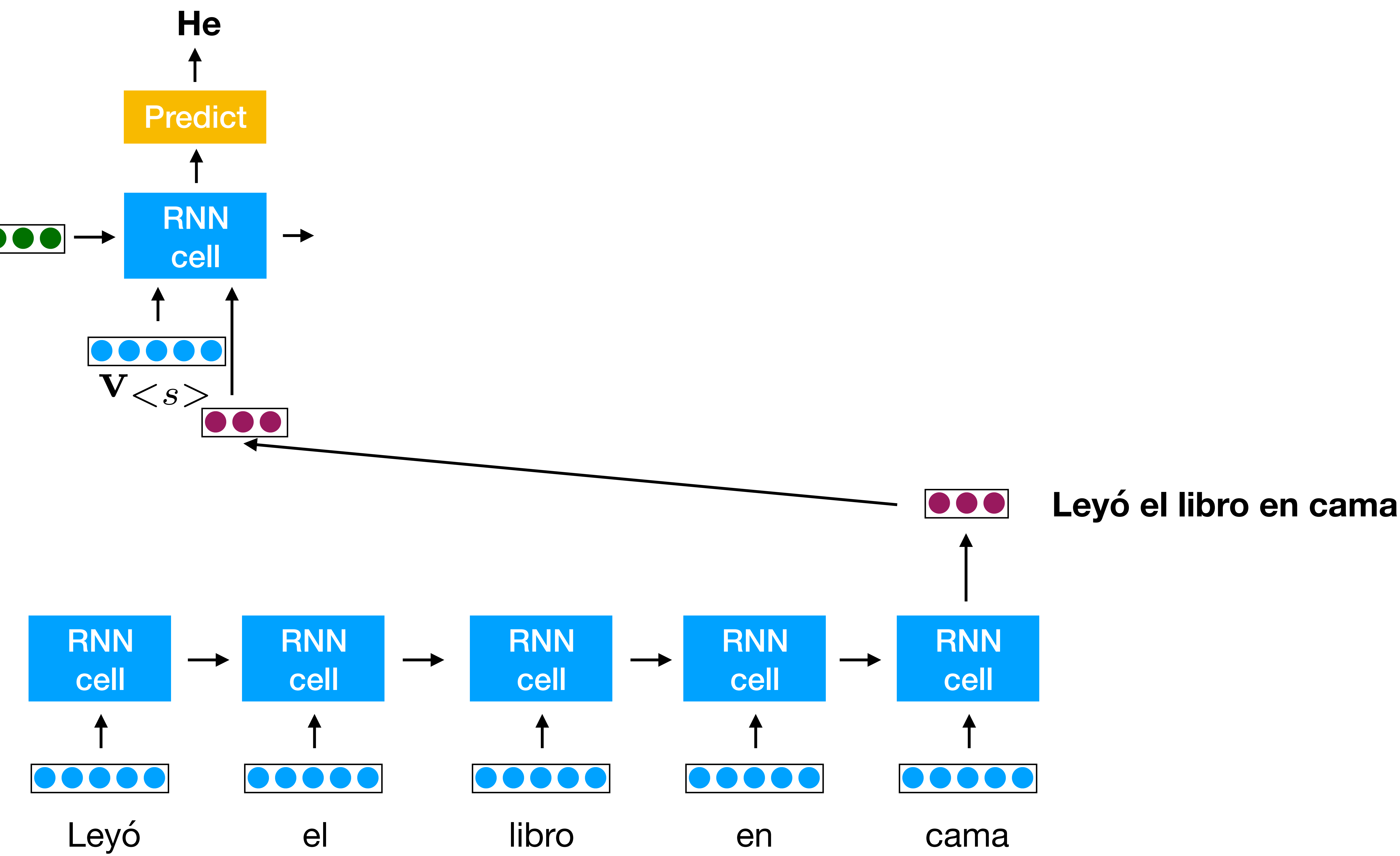
Leyó el libro en cama



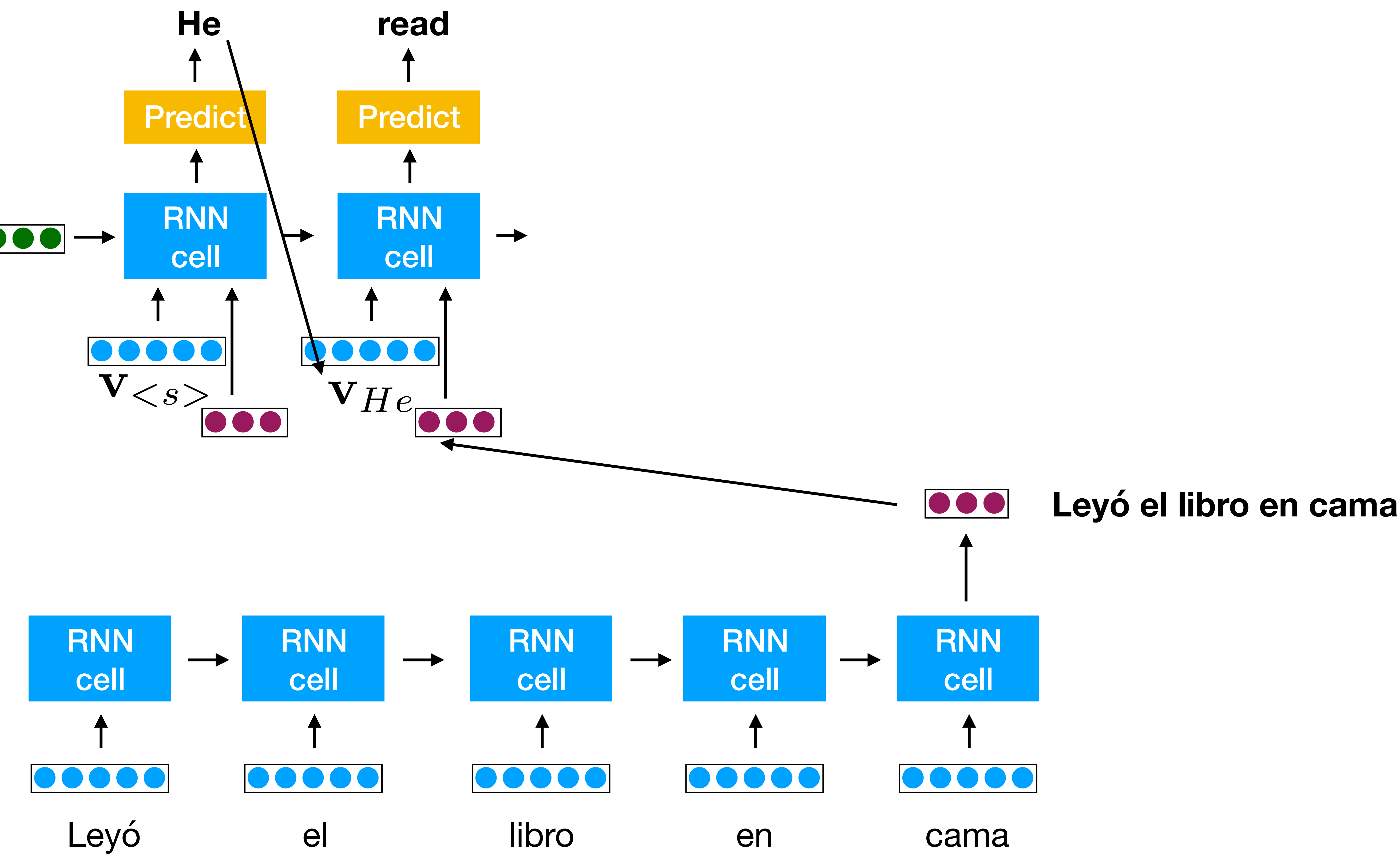
Seq2Seq


Leyó el libro en cama
RNN cell
RNN cell
RNN cell
RNN cell
RNN cell
Leyó
el
libro
en
cama

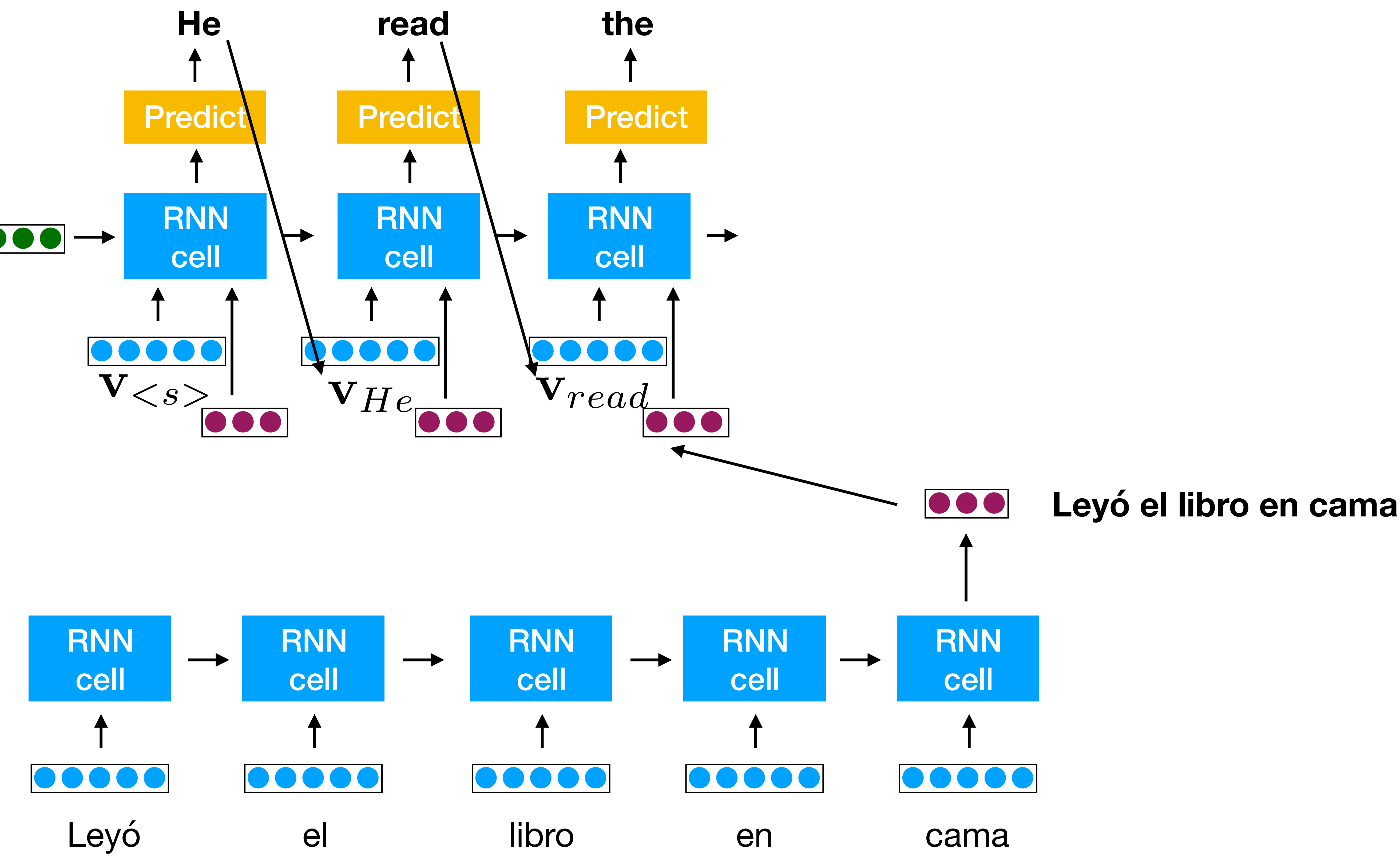
Seq2Seq


He
Predict
RNN cell
$\mathbf{v}_{<s>}$
Leyó el libro en cama
RNN cell
RNN cell
RNN cell
RNN cell
RNN cell
Leyó
el
libro
en
cama

Seq2Seq

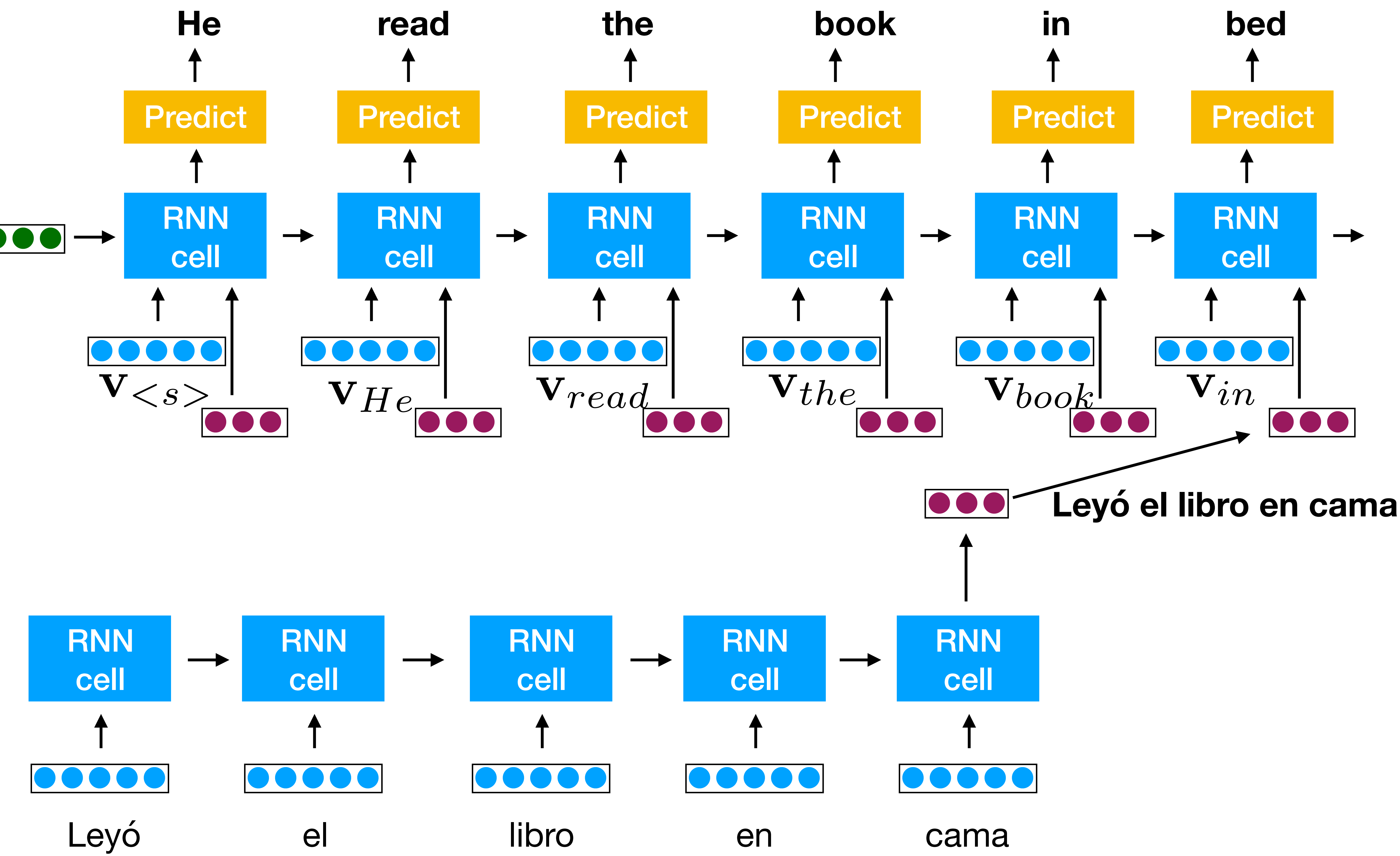

He
read
Predict
Predict
RNN cell
RNN cell
$\mathbf{v}_{<s>}$
$\mathbf{v}_{He}$
Leyó el libro en cama
RNN cell
RNN cell
RNN cell
RNN cell
RNN cell
Leyó
el
libro
en
cama

Seq2Seq



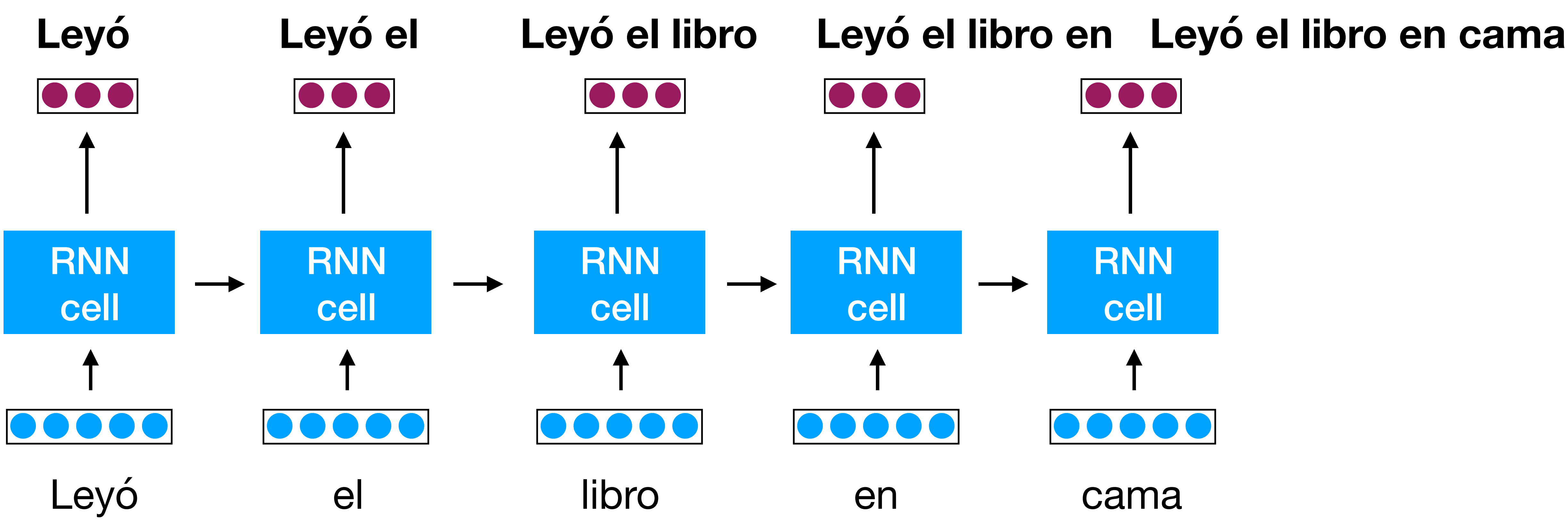
He
read
the
Predict
Predict
Predict
RNN cell
RNN cell
RNN cell
$\mathbf{v}_{<s>}$
$\mathbf{v}_{He}$
$\mathbf{v}_{read}$
Leyó el libro en cama
RNN cell
RNN cell
RNN cell
RNN cell
RNN cell
Leyó
el
libro
en
cama

Seq2Seq


He
read
the
book
in
bed
Predict
RNN cell
$\mathbf{v}_{<s>}$
$\mathbf{v}_{He}$
$\mathbf{v}_{read}$
$\mathbf{v}_{the}$
$\mathbf{v}_{book}$
$\mathbf{v}_{in}$
Leyó el libro en cama
Leyó
el
libro
en
cama

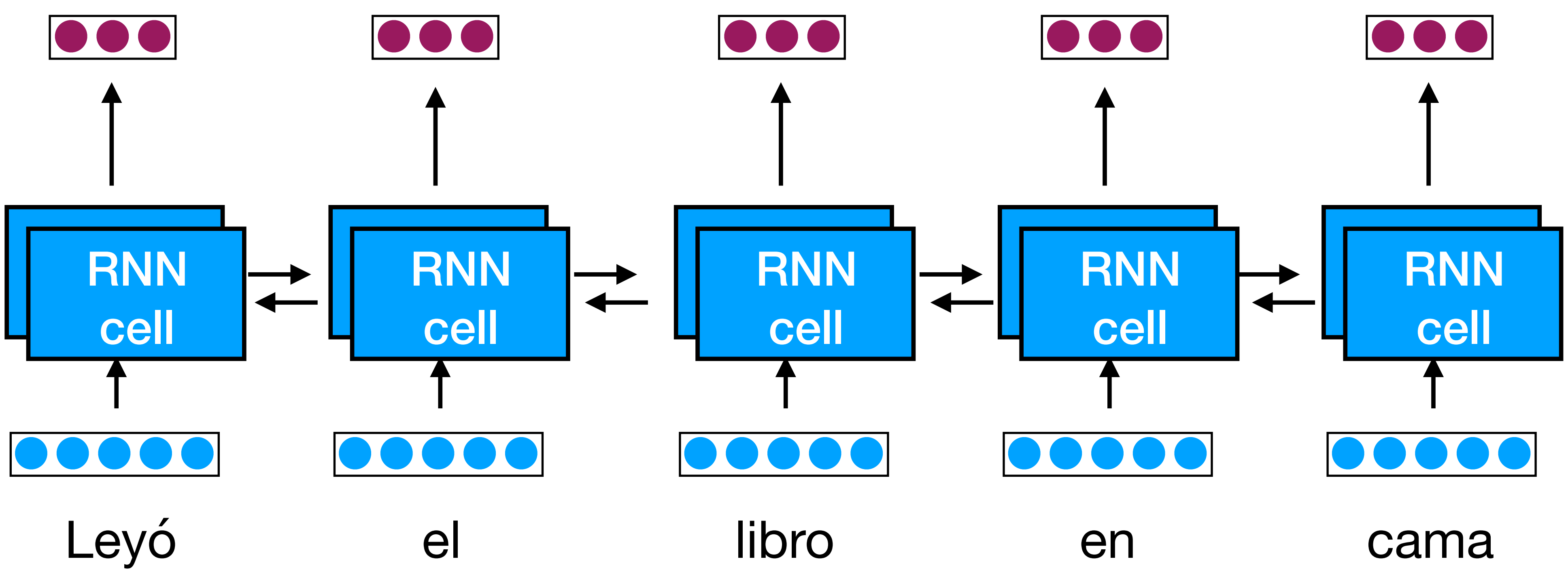
Seq2Seq + Attention

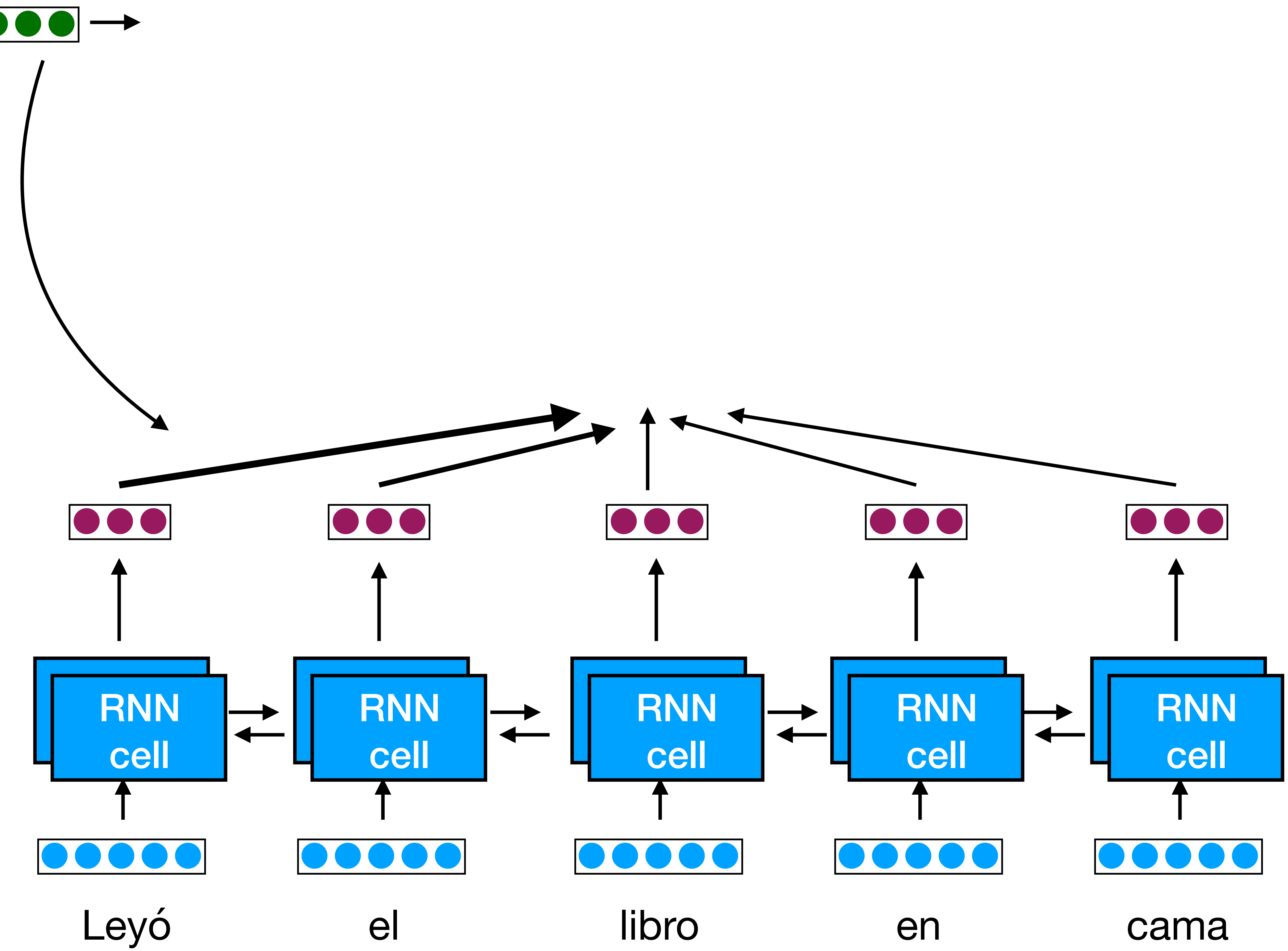
keep intermediate vectors



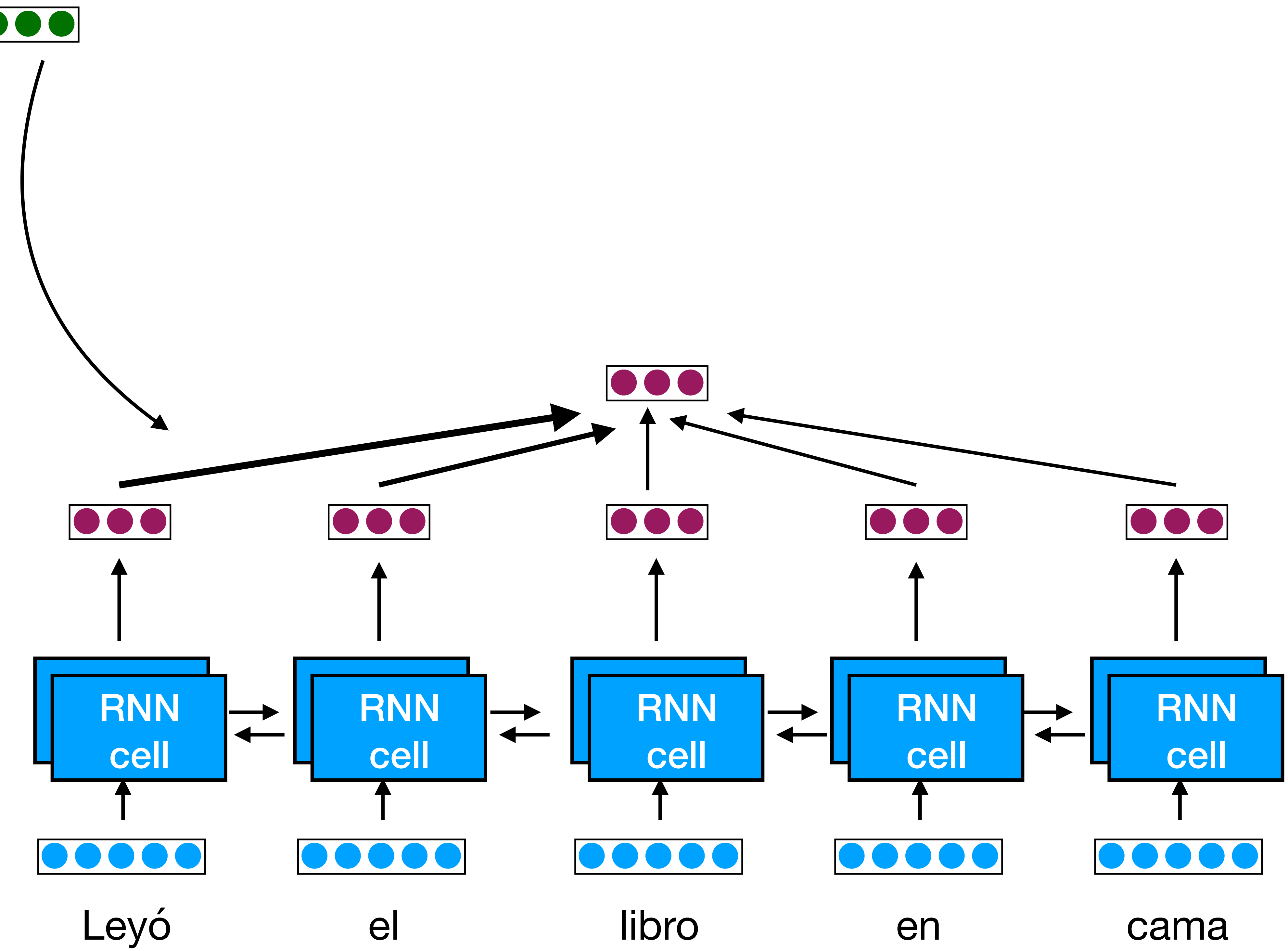
Seq2Seq + Attention

as Bi-RNN

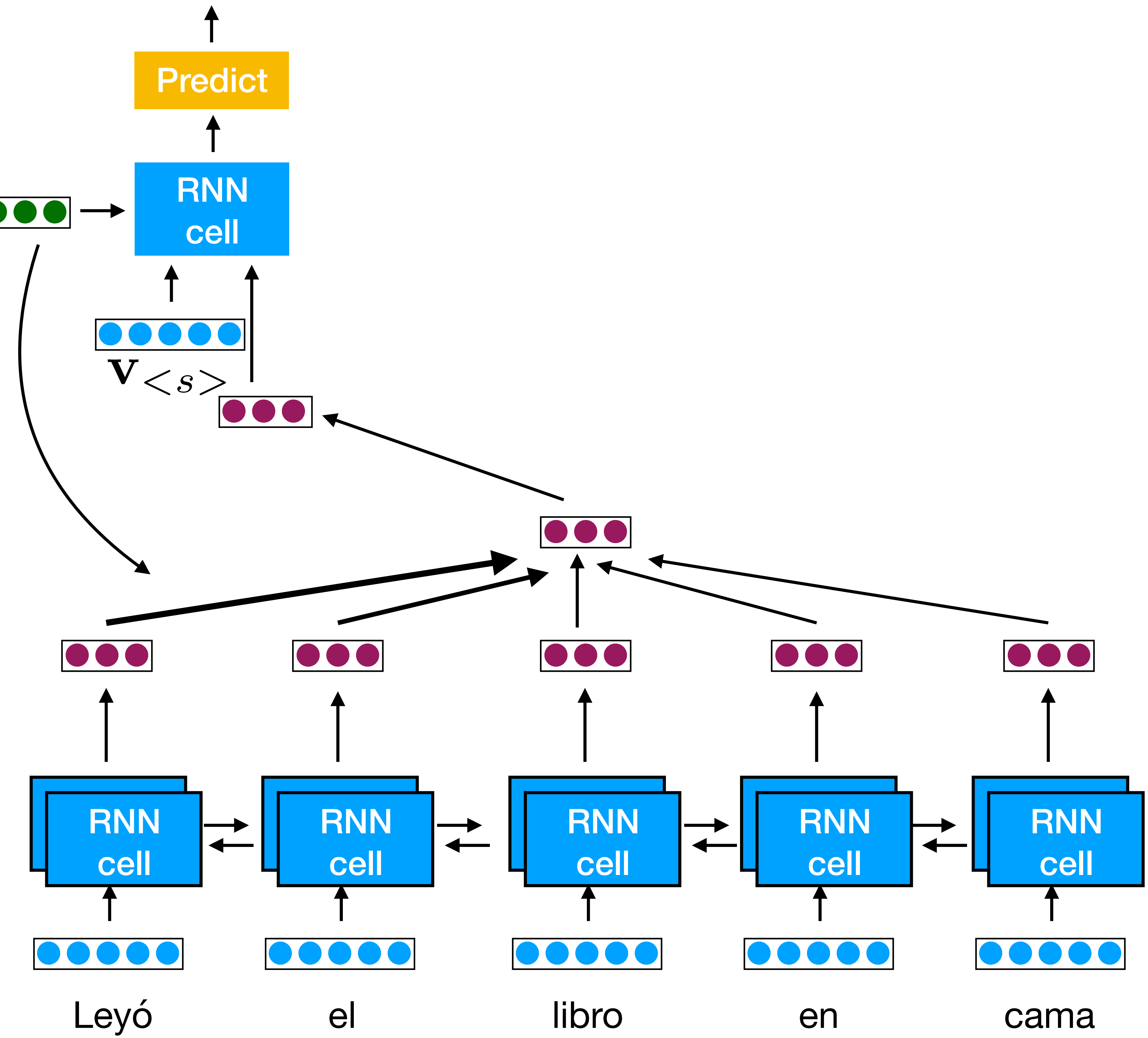




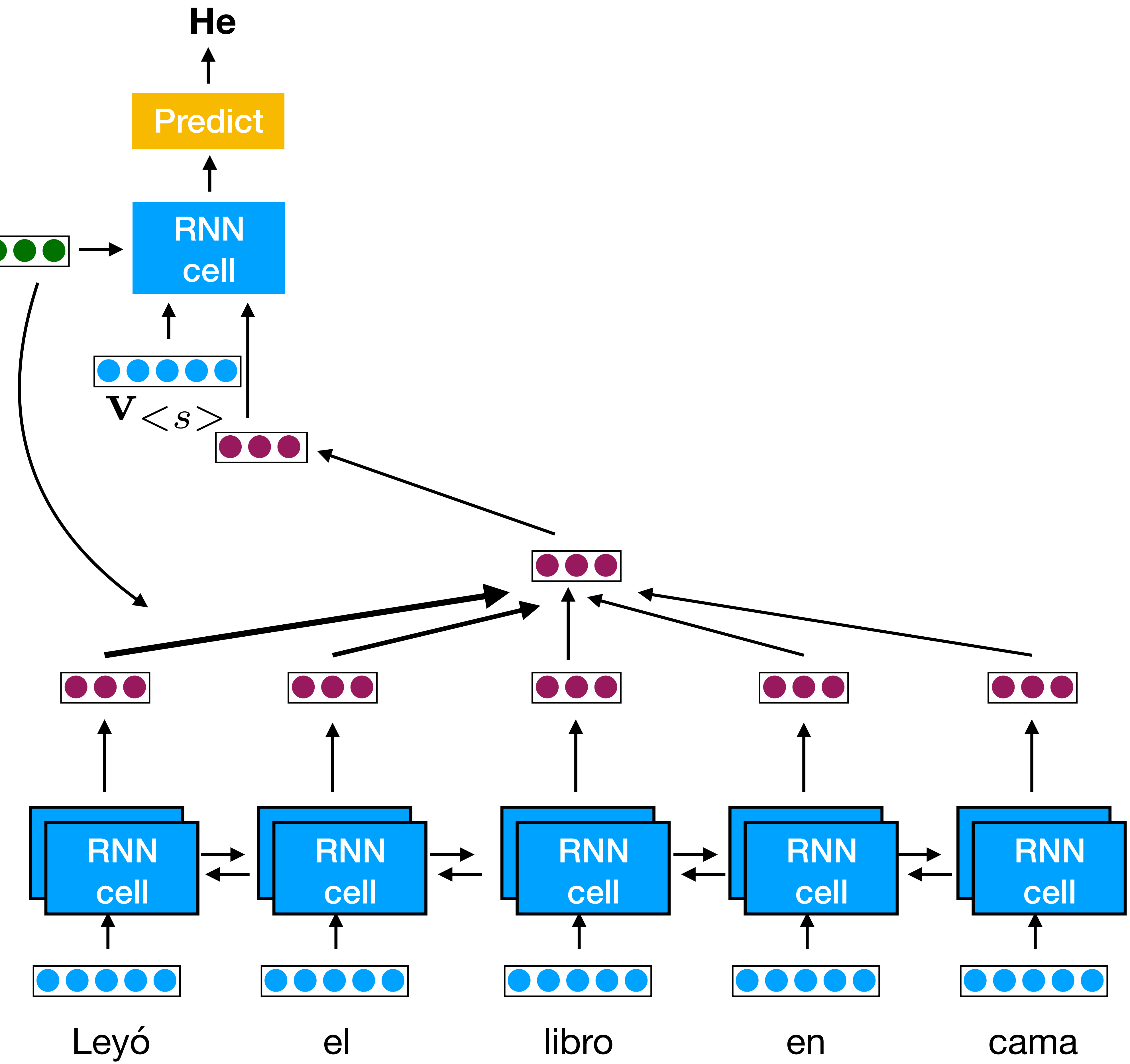
weighted sum



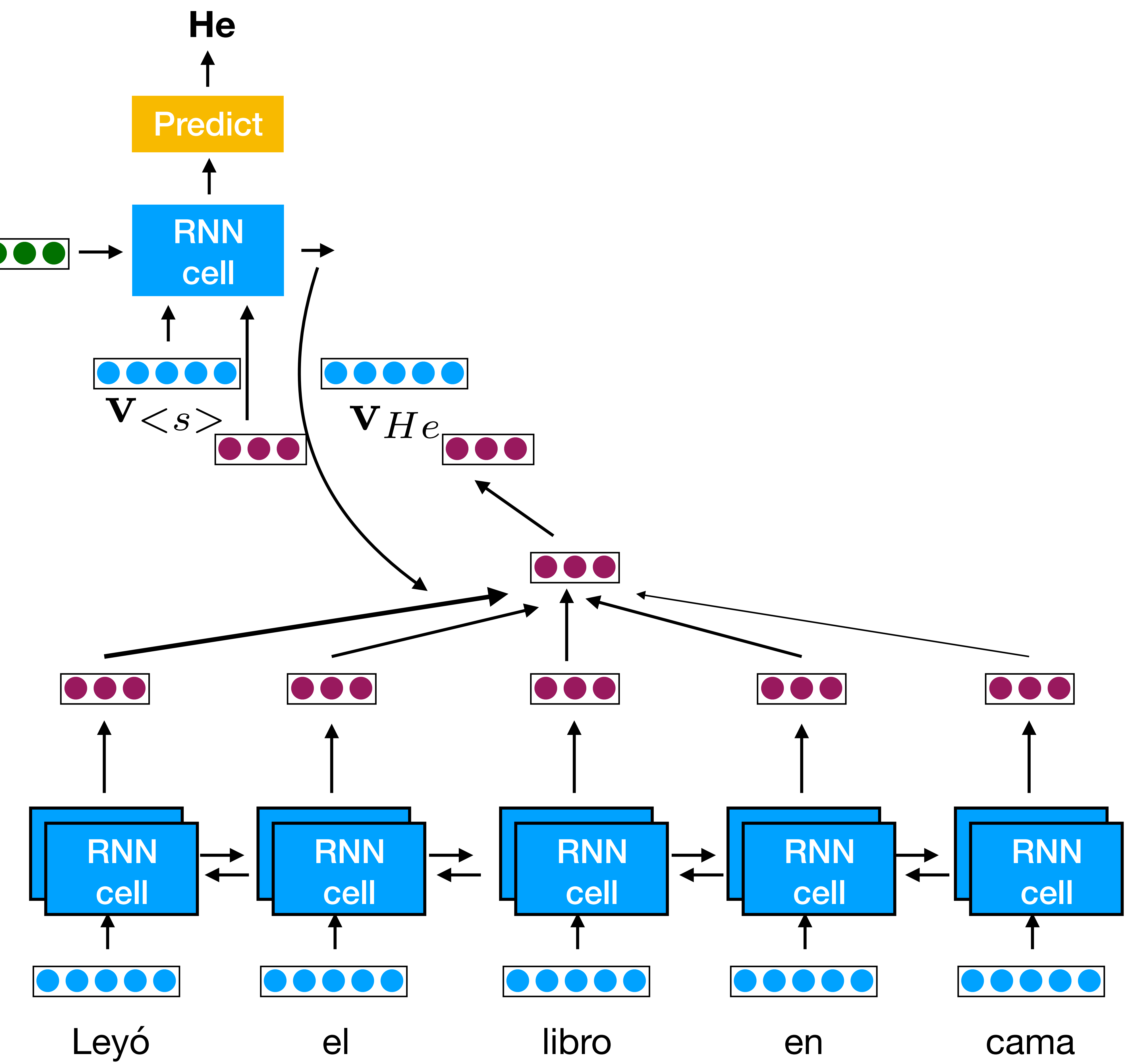
weighted sum



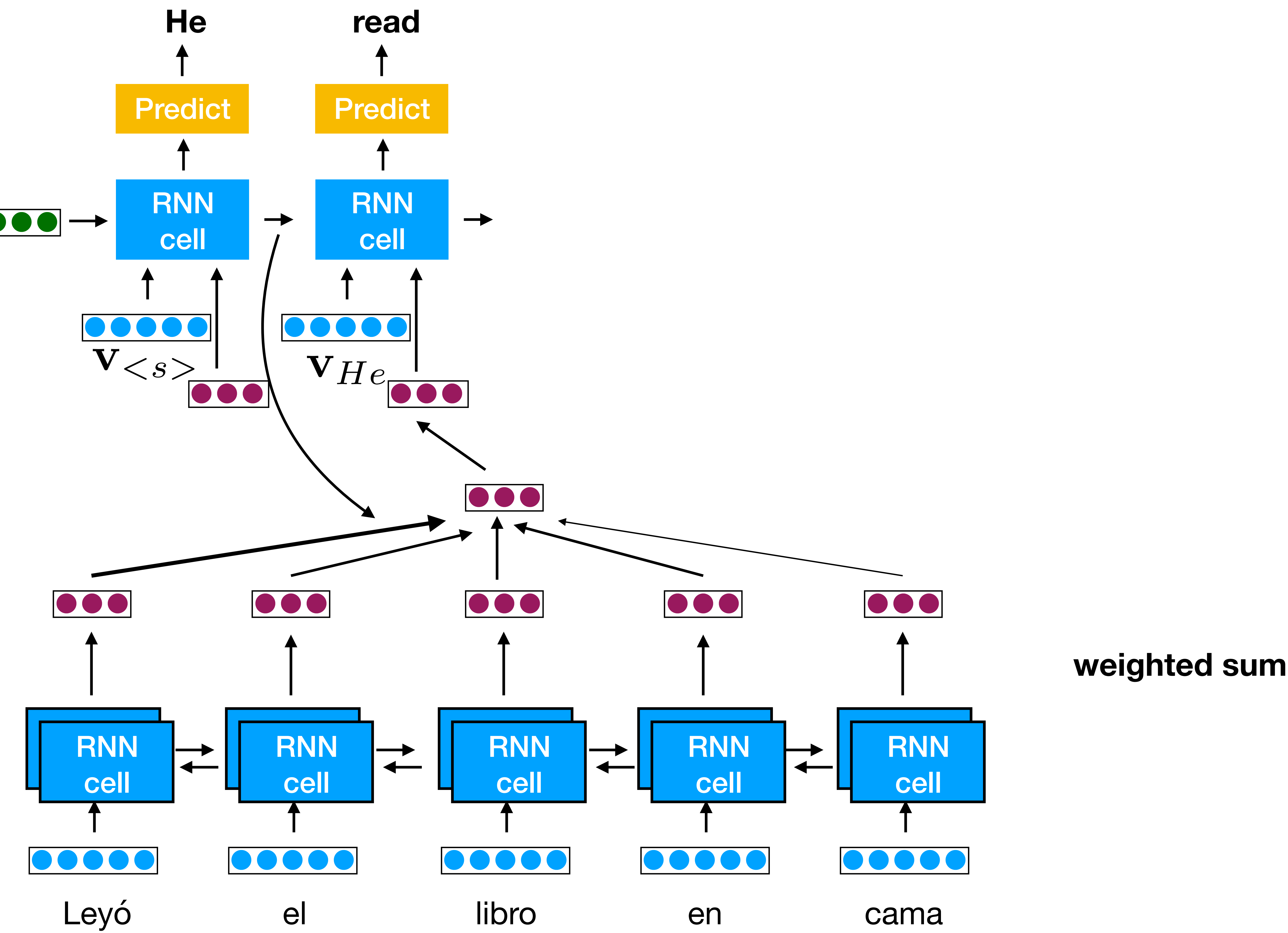
weighted sum



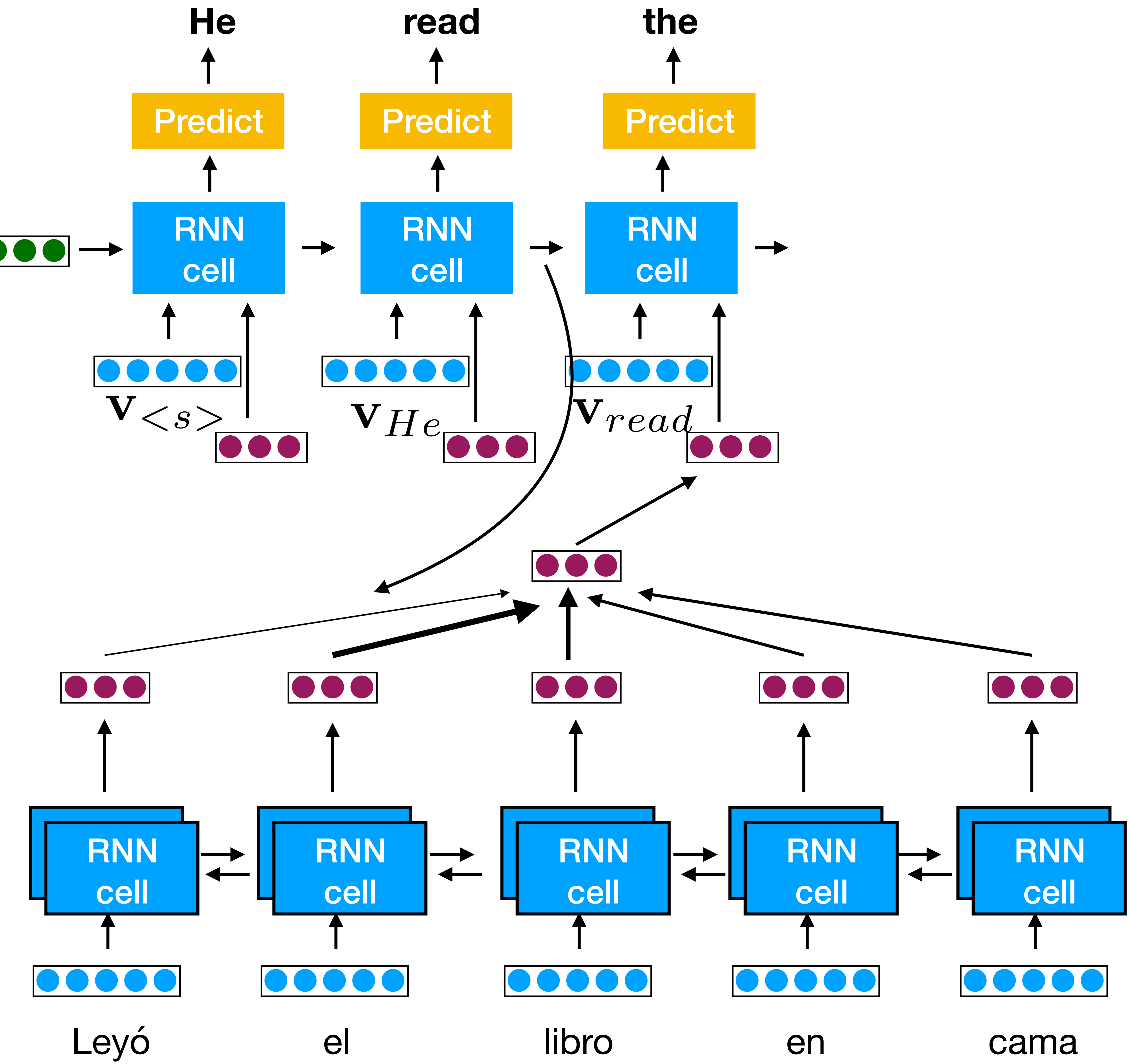
weighted sum



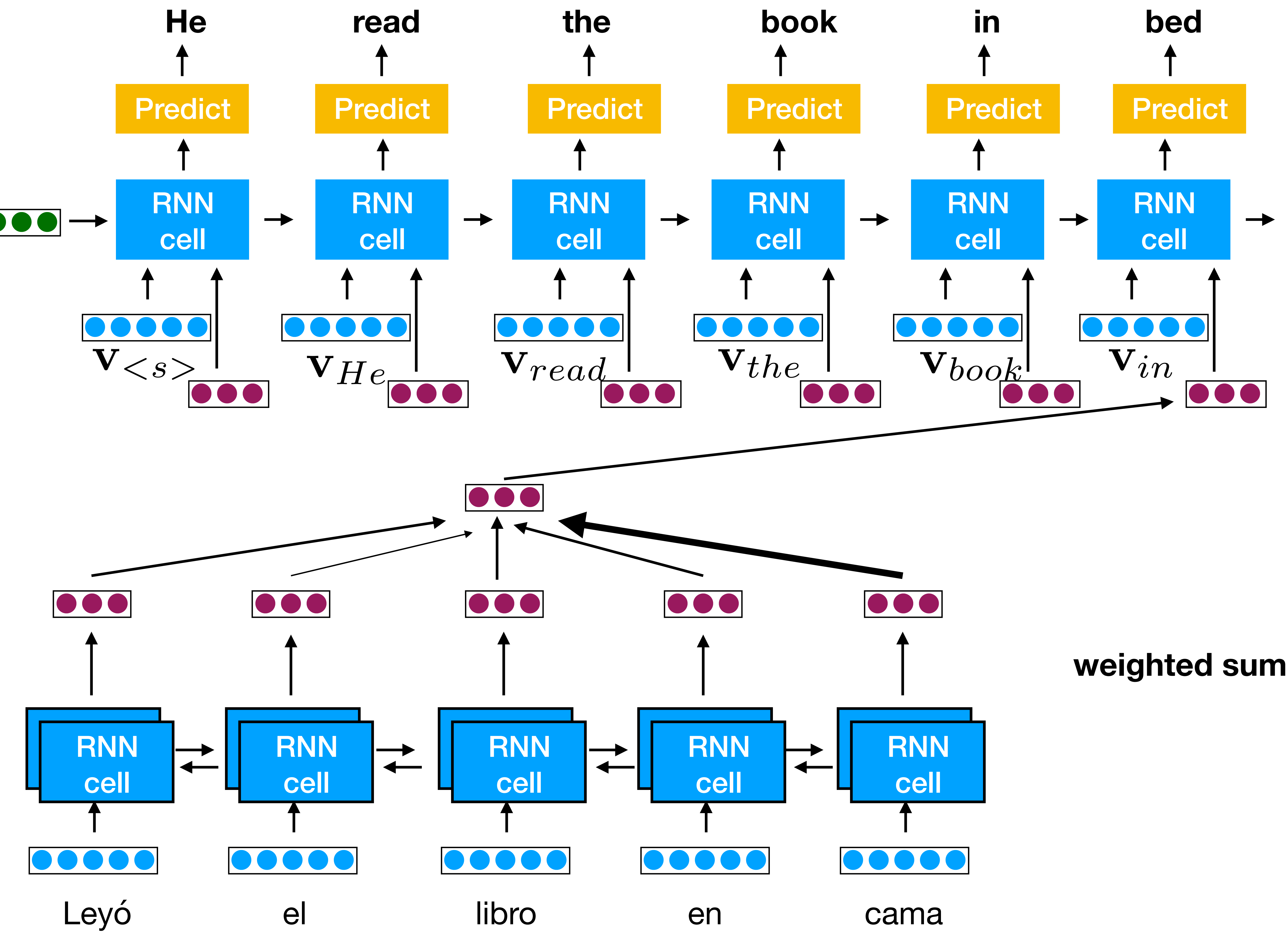
weighted sum



He
read
Predict
Predict
RNN cell
RNN cell
$\mathbf{v}_{<s>}$
$\mathbf{v}_{He}$
RNN cell
RNN cell
RNN cell
RNN cell
RNN cell
Leyó
el
libro
en
cama
weighted sum



weighted sum


He
read
the
book
in
bed
Predict
RNN cell
$\mathbf{v}_{<s>}$
$\mathbf{v}_{He}$
$\mathbf{v}_{read}$
$\mathbf{v}_{the}$
$\mathbf{v}_{book}$
$\mathbf{v}_{in}$
weighted sum
Leyó
el
libro
en
cama

Transformer

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Transformer

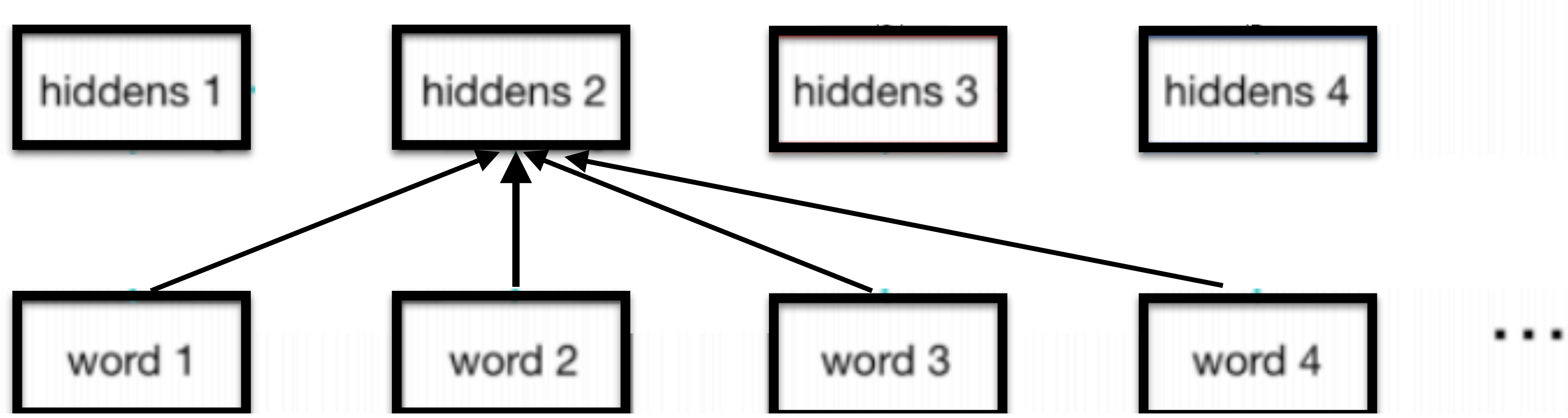
replace RNN with attention-based mechanism

- Main concepts to know:
  - Self-attention
  - Multi-head attention
- Also think about: why do this? what is the motivation?

Transformer

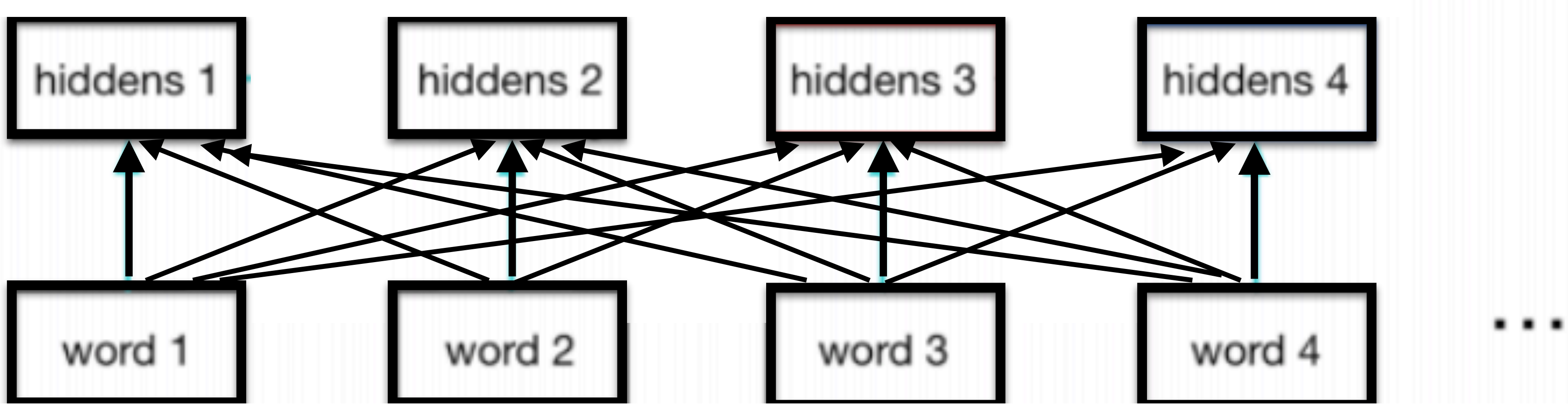
Self attention

each token attends to all tokens in previous layer



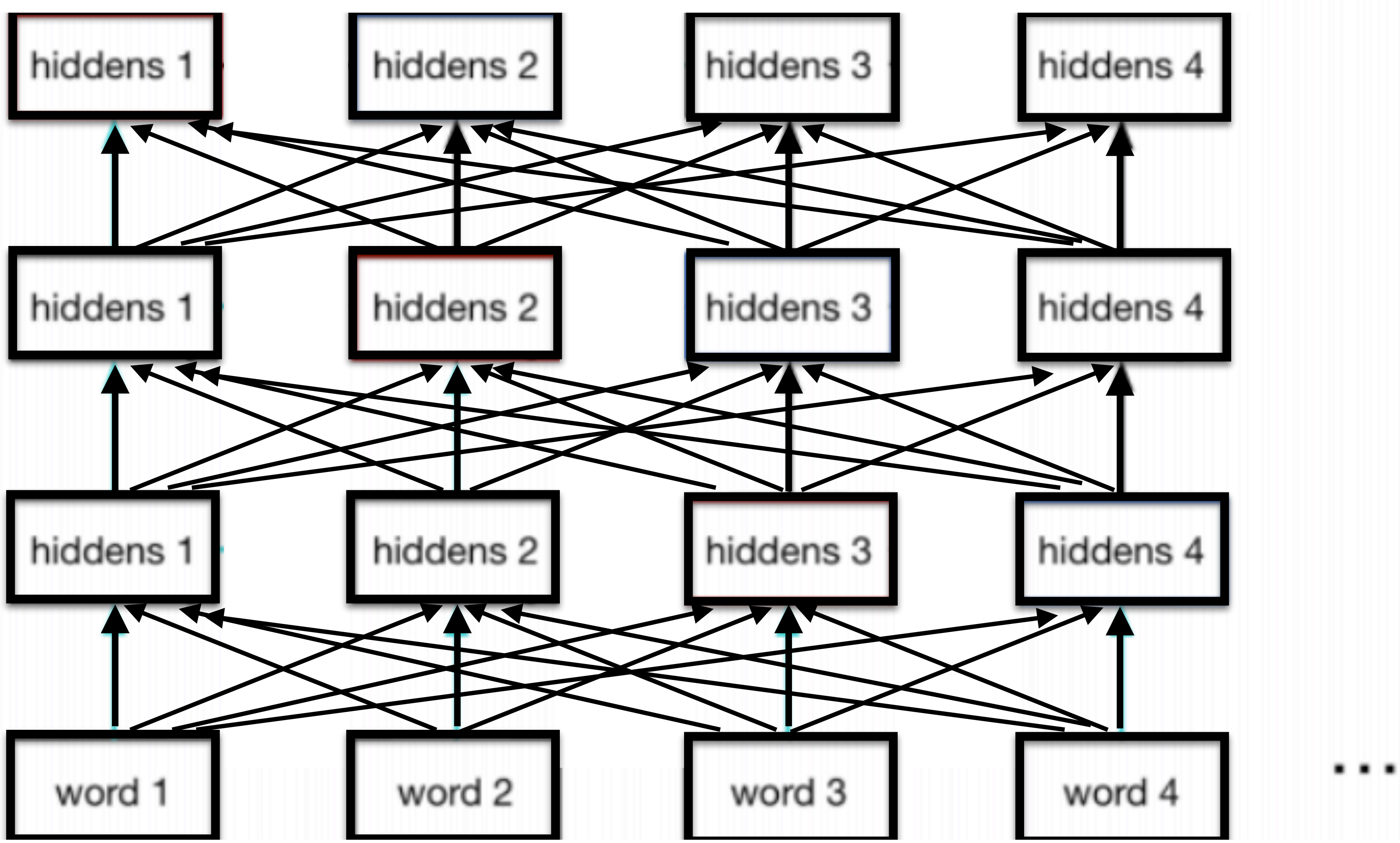
Transformer

Self attention



Transformer

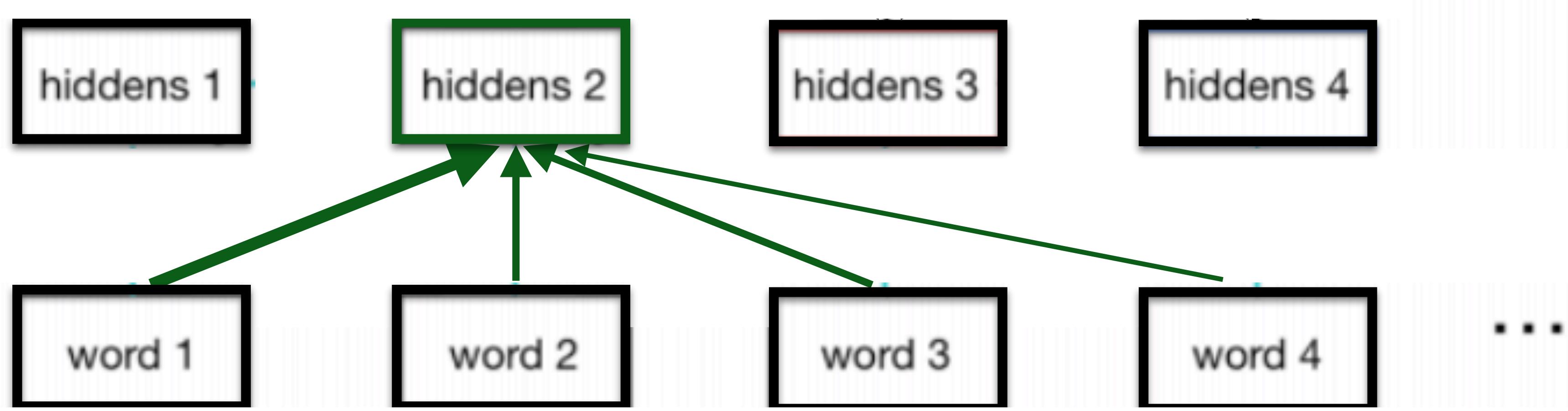
Self attention



Transformer

multi-head attention

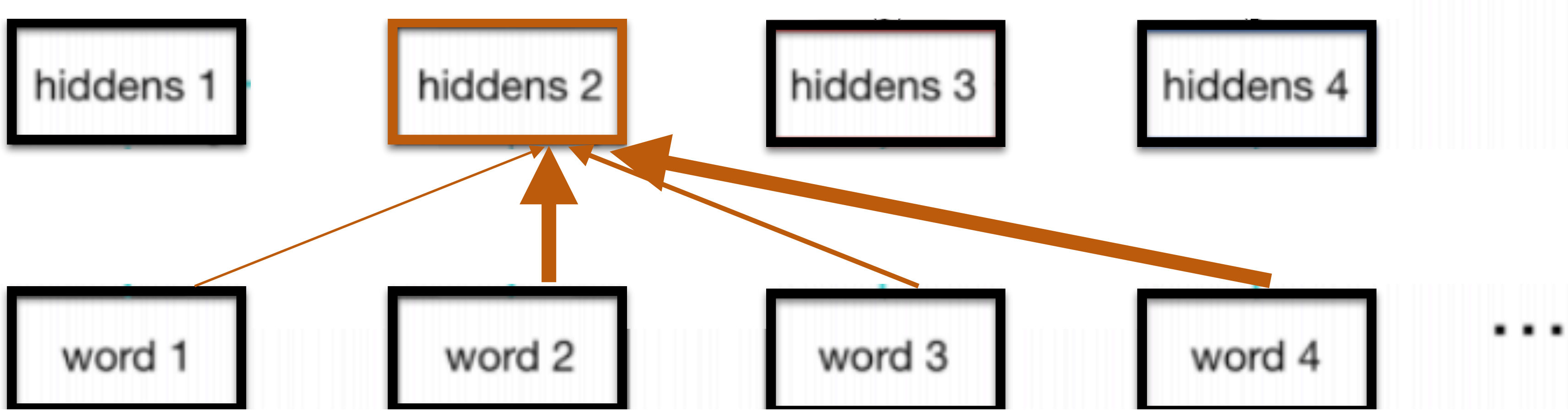
one attention pattern



Transformer

multi-head attention

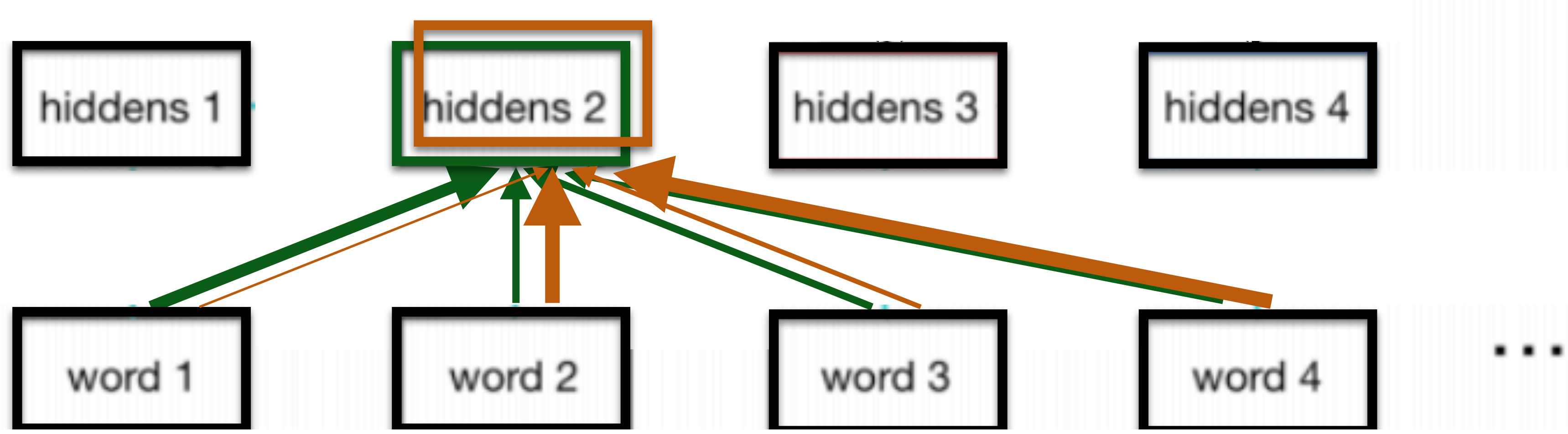
another attention pattern



Transformer

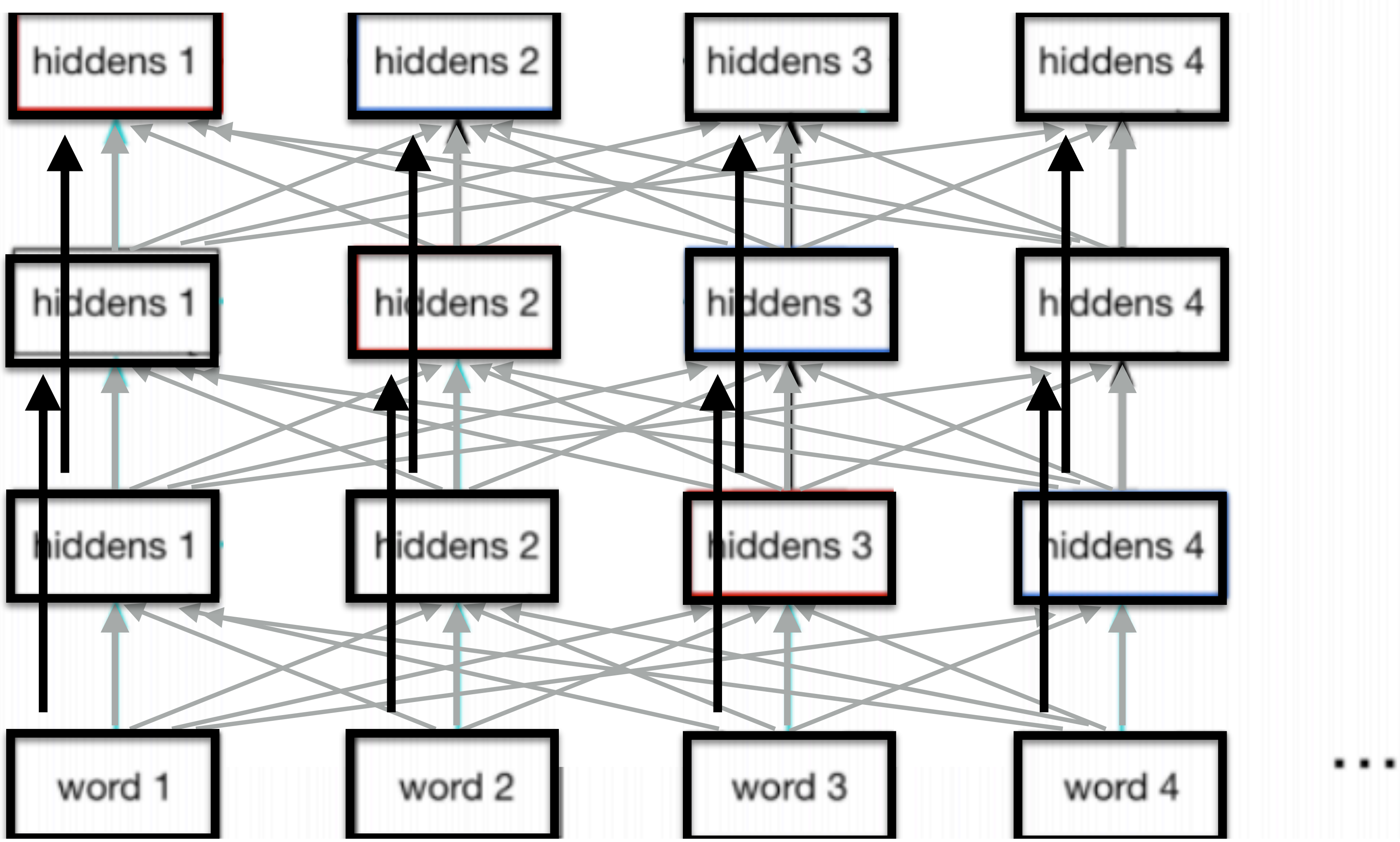
multi-head attention

why chose if we can just have several?



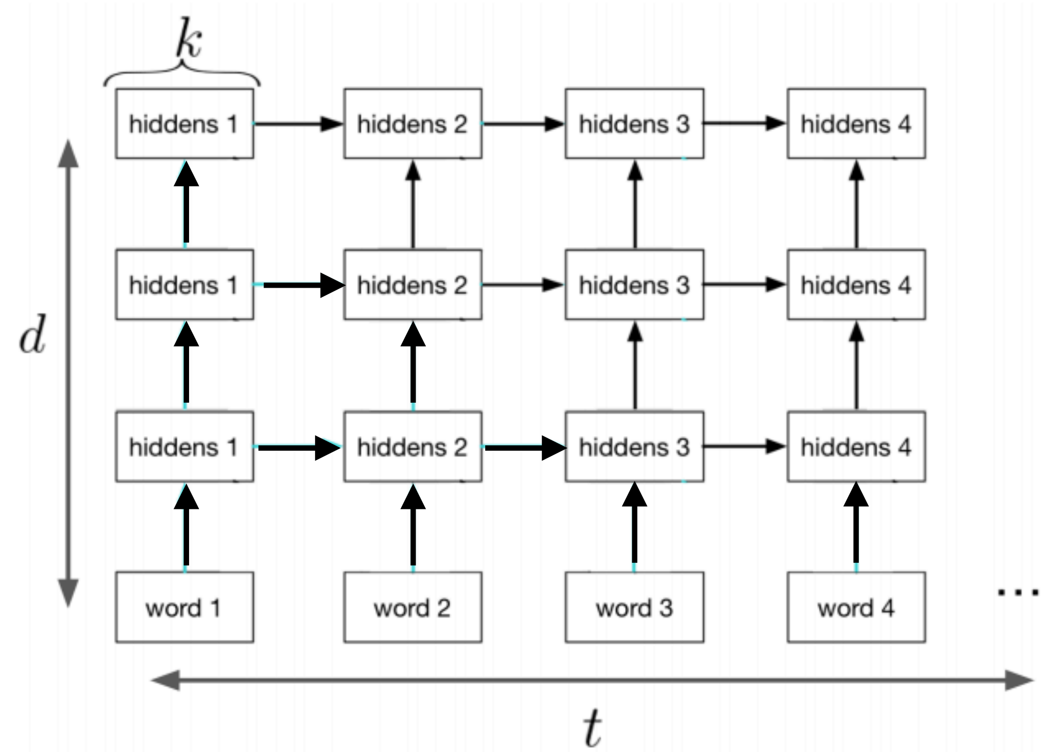
Transformer

Skip connections



Cost vs Opportunity

- Consider a standard d layer RNN from Lecture 13 with k hidden units, training on a sequence of length t.

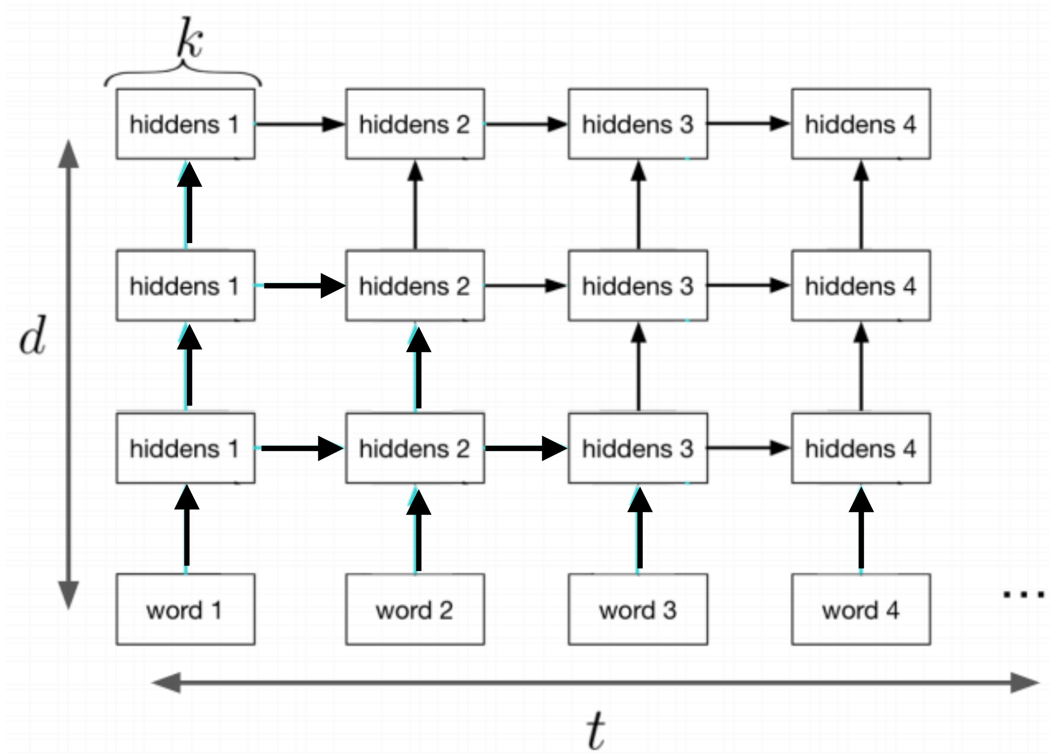


- There are k^2 connections for each hidden-to-hidden connection. A total of $t \times k^2 \times d$ connections.
- We need to store all $t \times k \times d$ hidden units during training.
- Only $k \times d$ hidden units need to be stored at test time.

http://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec16.pdf

Cost vs Opportunity

- Consider a standard d layer RNN from Lecture 13 with k hidden units, training on a sequence of length t.

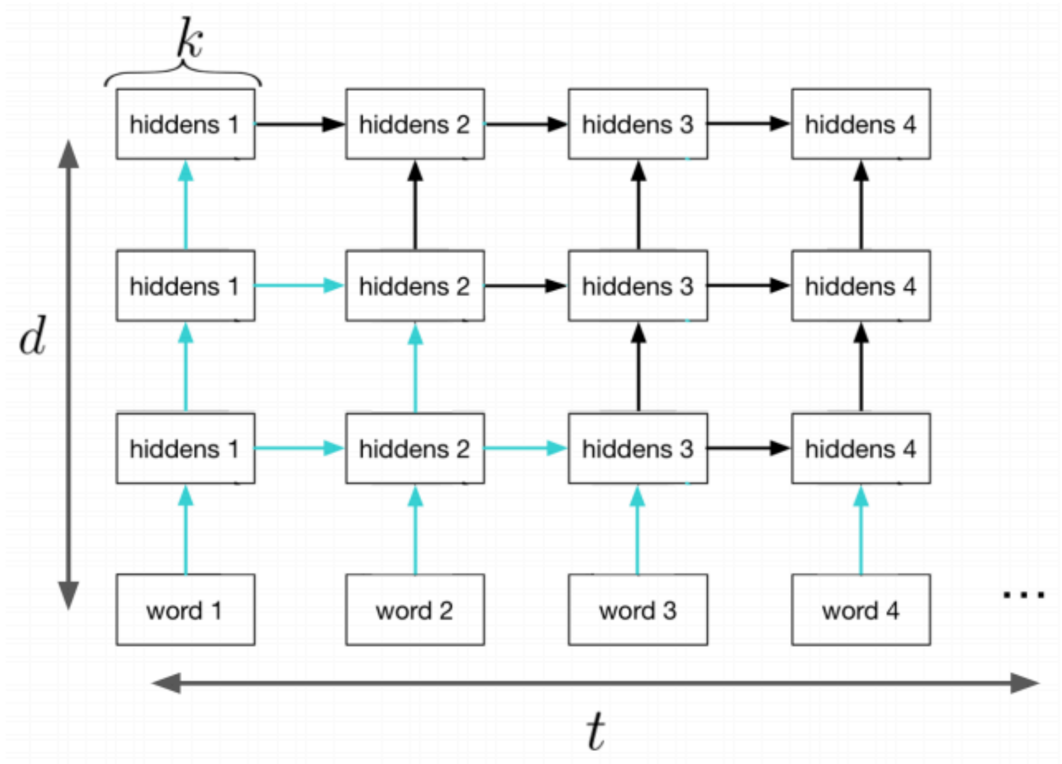


- Which hidden layers can be computed in parallel in this RNN?

http://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec16.pdf

Cost vs Opportunity

- Consider a standard d layer RNN from Lecture 13 with k hidden units, training on a sequence of length t.

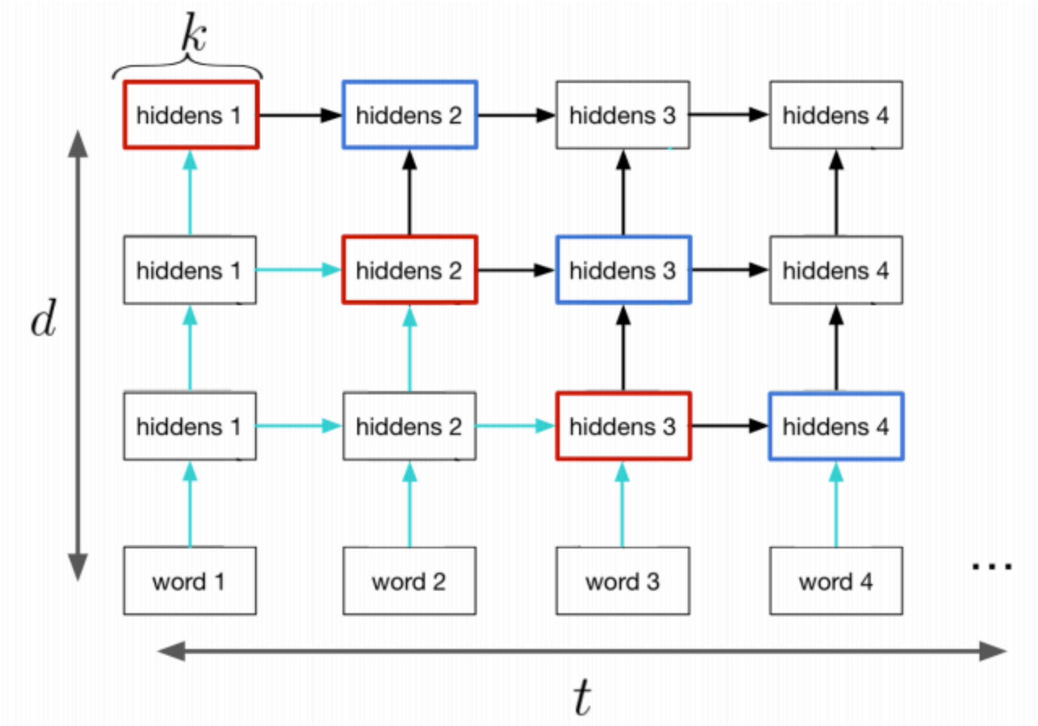


- Which hidden layers can be computed in parallel in this RNN?

http://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec16.pdf

Cost vs Opportunity

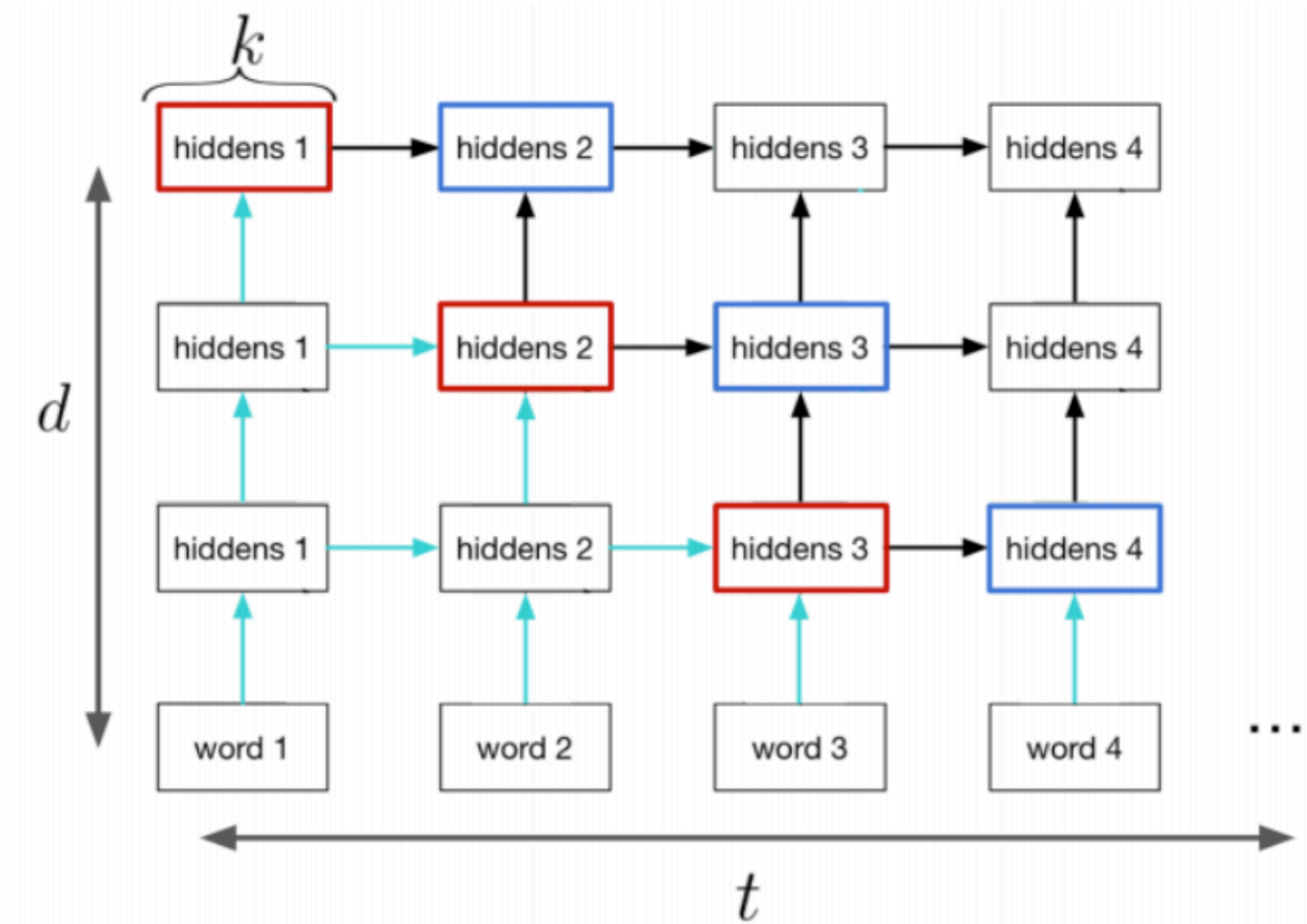
- Consider a standard d layer RNN from Lecture 13 with k hidden units, training on a sequence of length t.



- Both the input embeddings and the outputs of an RNN can be computed in parallel.
- The blue hidden units are independent given the red.
- The numer of sequential operation is still propotional to t.

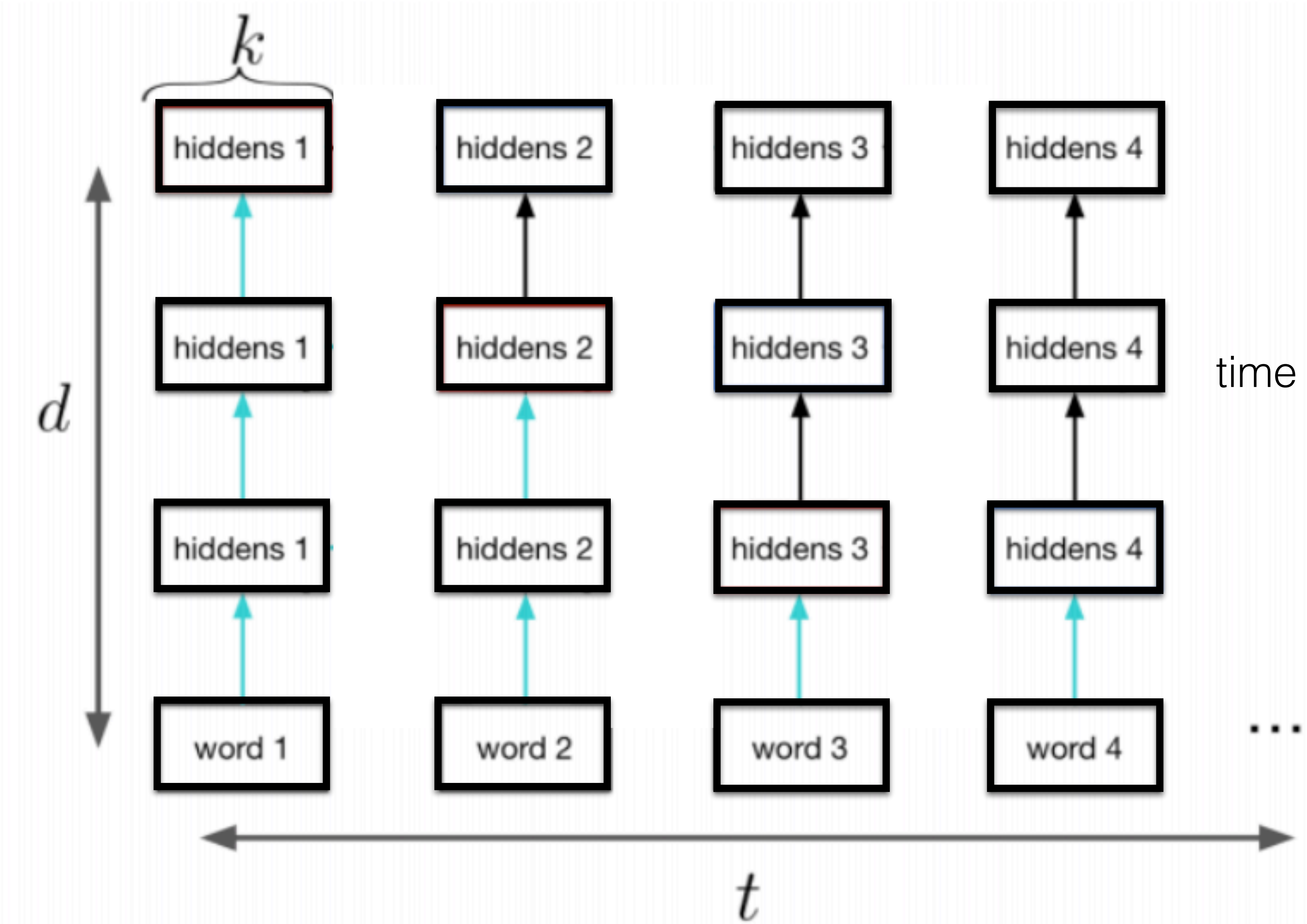
Cost vs Opportunity

RNN to Self-attention



Cost vs Opportunity

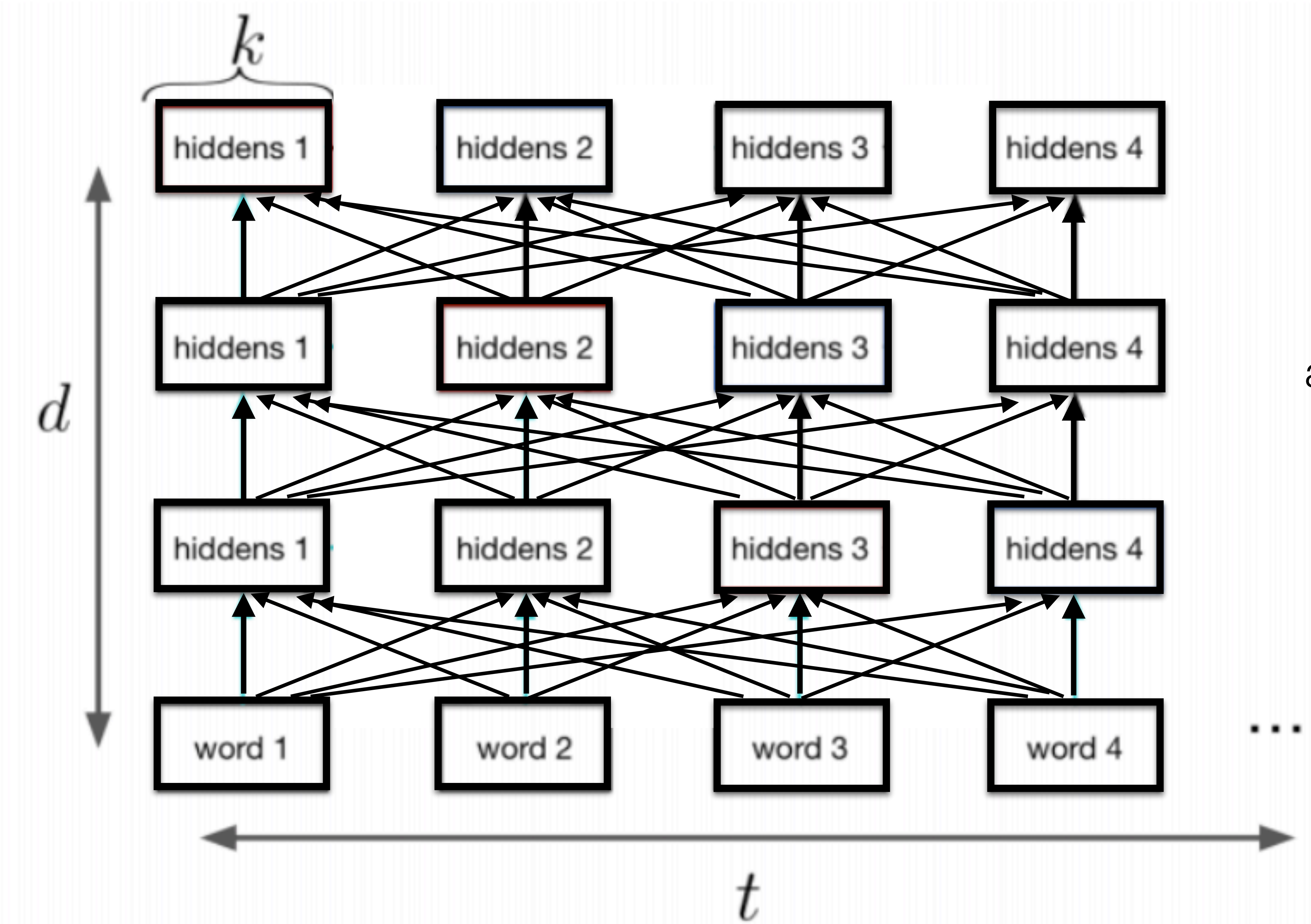
RNN to Self-attention



drop
time dependence

Cost vs Opportunity

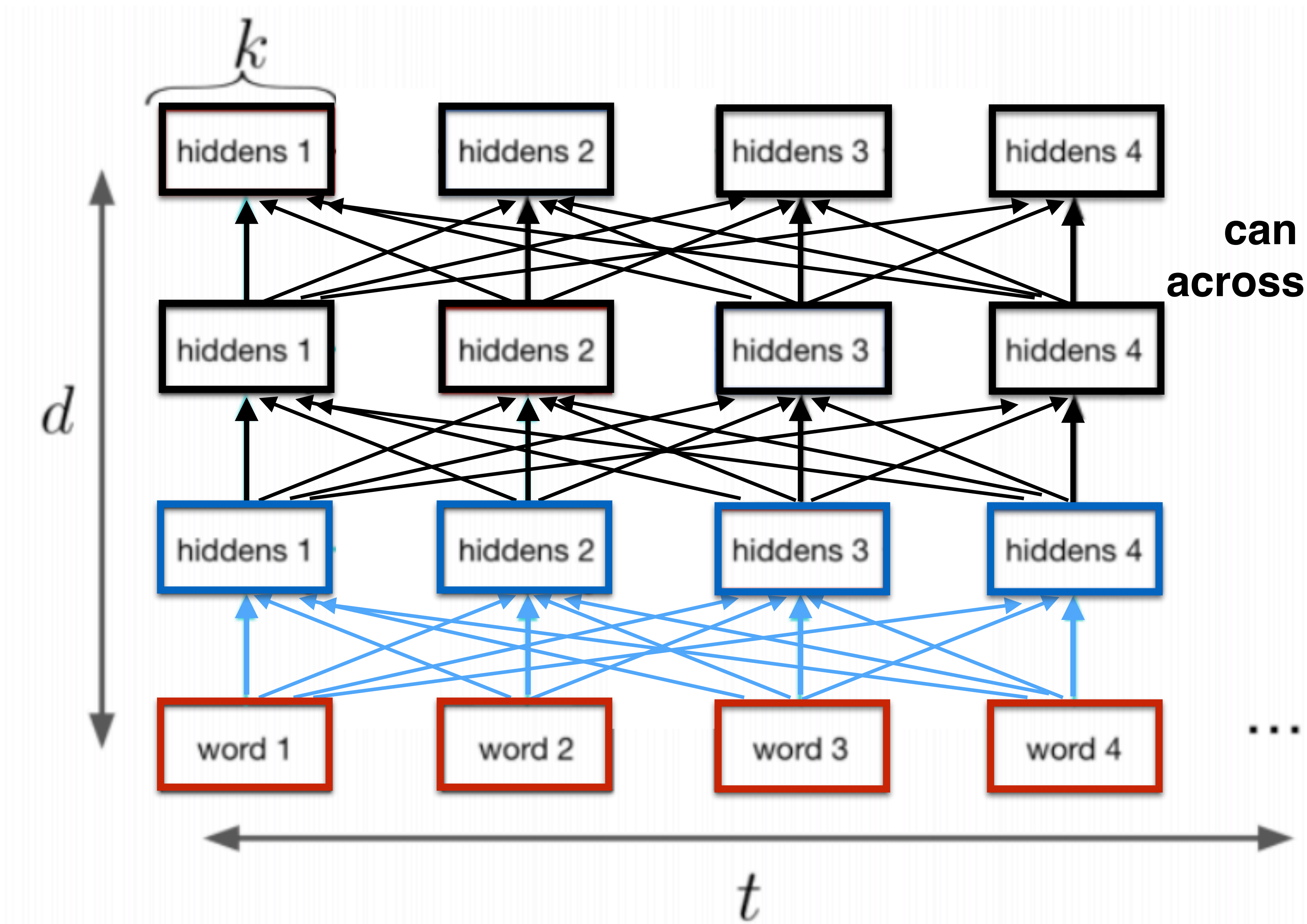
RNN to Self-attention



add attention

Cost vs Opportunity

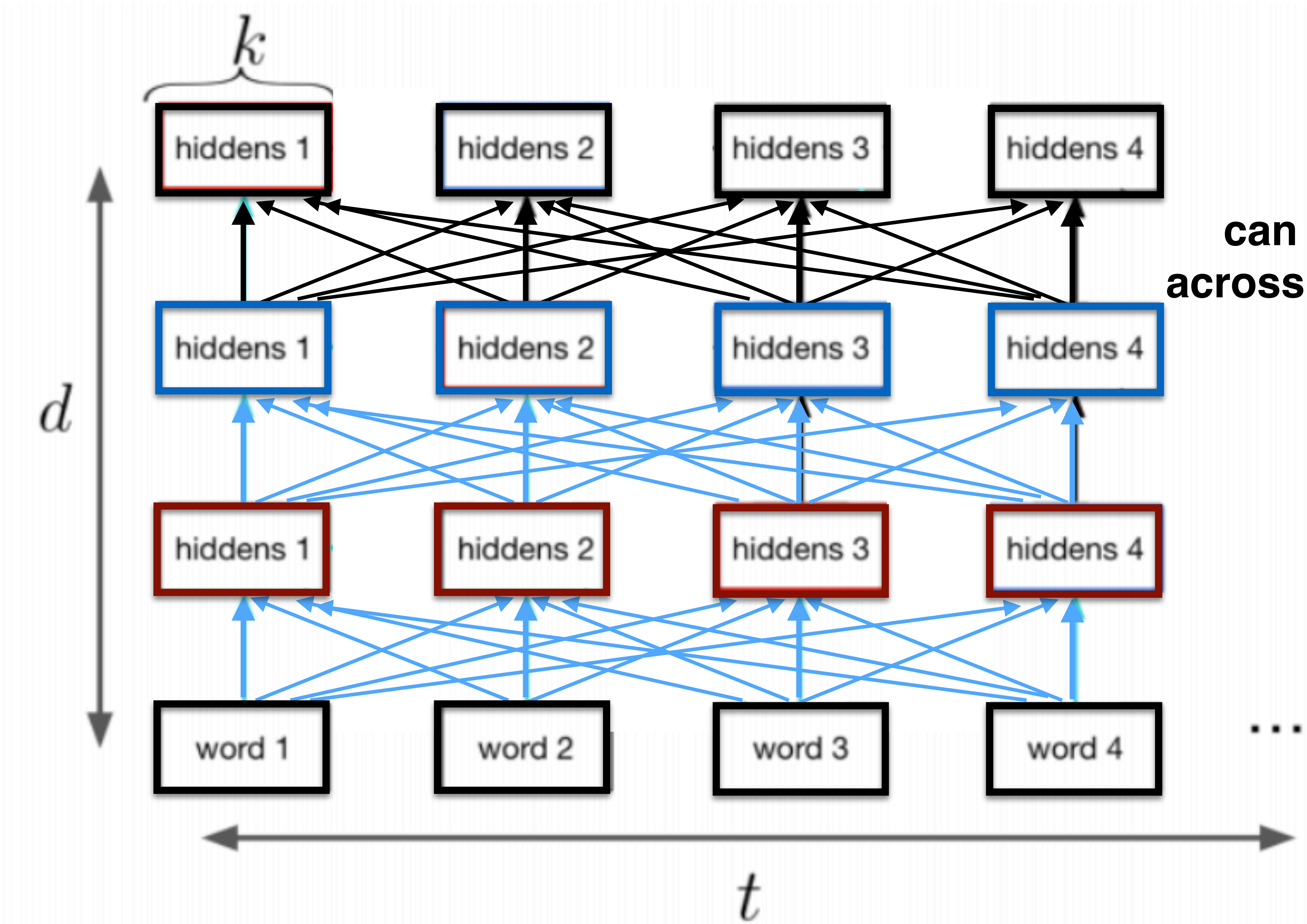
RNN to Self-attention



can parallelize across all sequence

Cost vs Opportunity

RNN to Self-attention

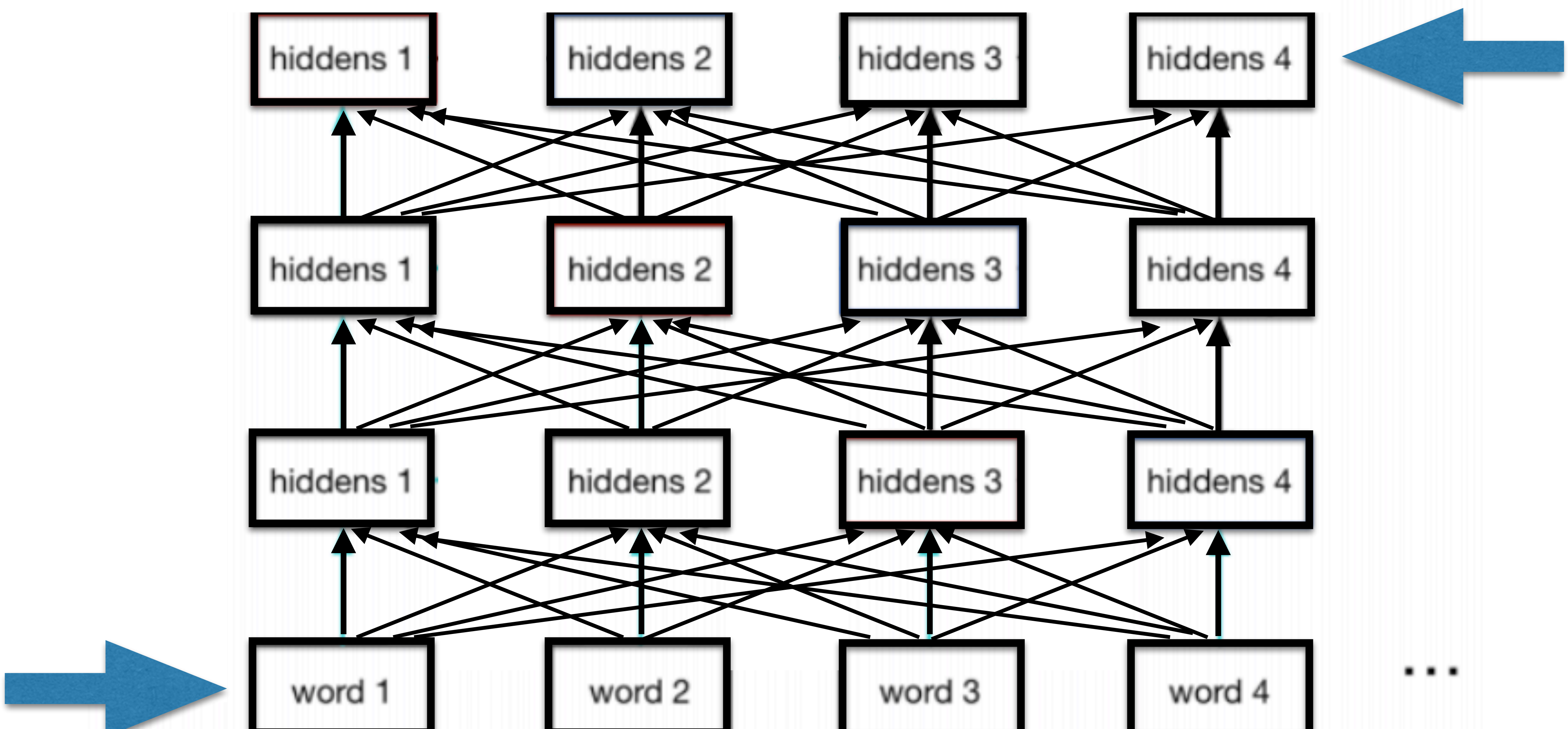


can parallelize across all sequence

Transformer

Information flow

how do we pass information between the blue arrows?

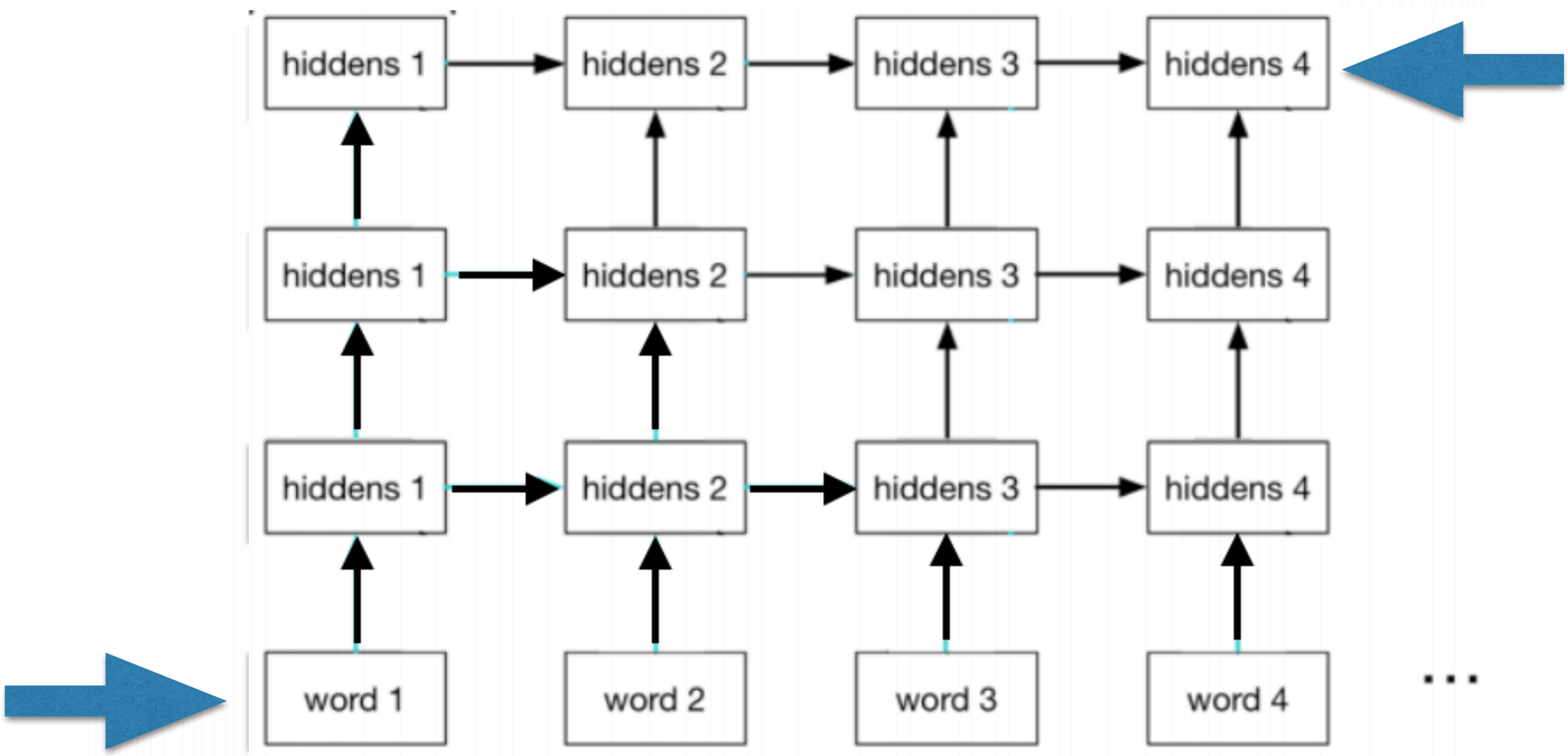


Transformer

vs RNN case

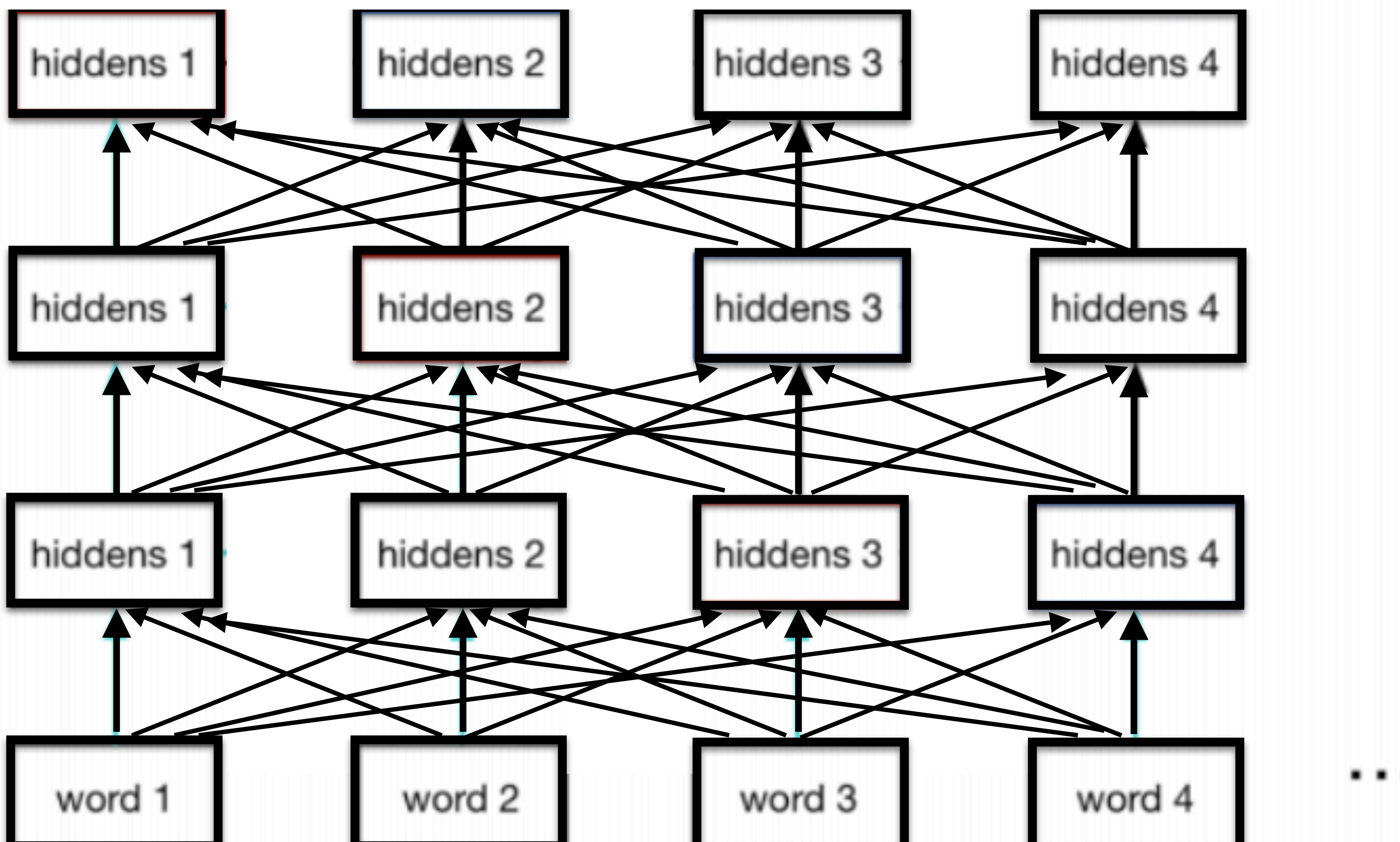
Information flow

how do we pass information between the blue arrows?



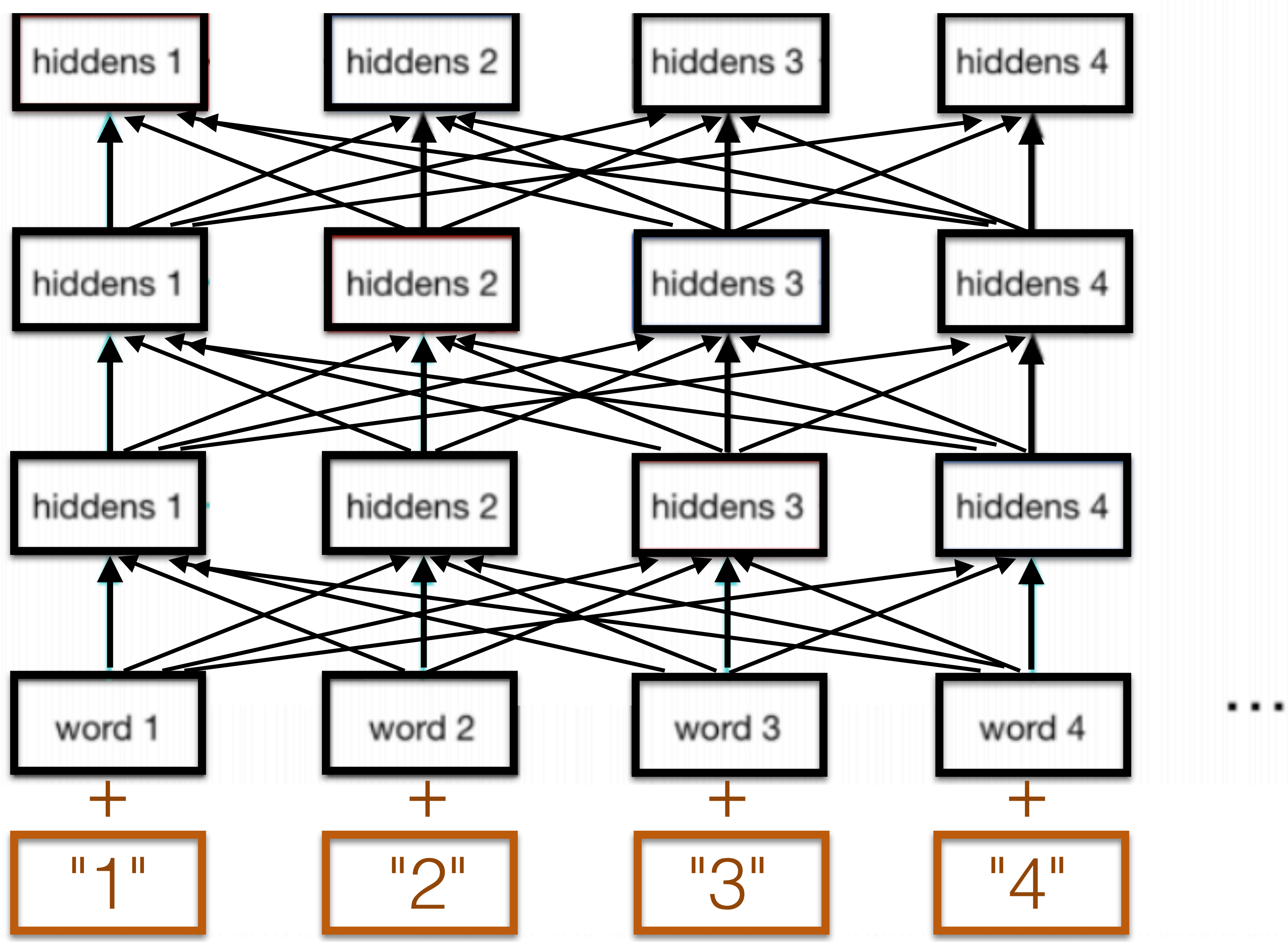
Transformer

Positional information

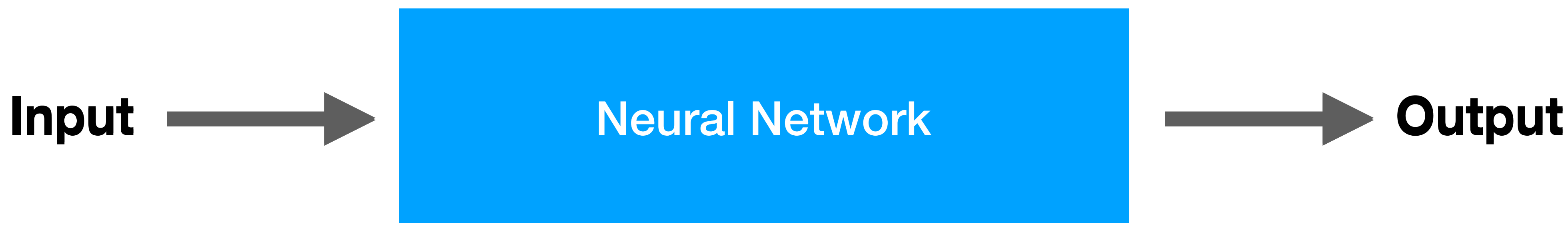


Transformer

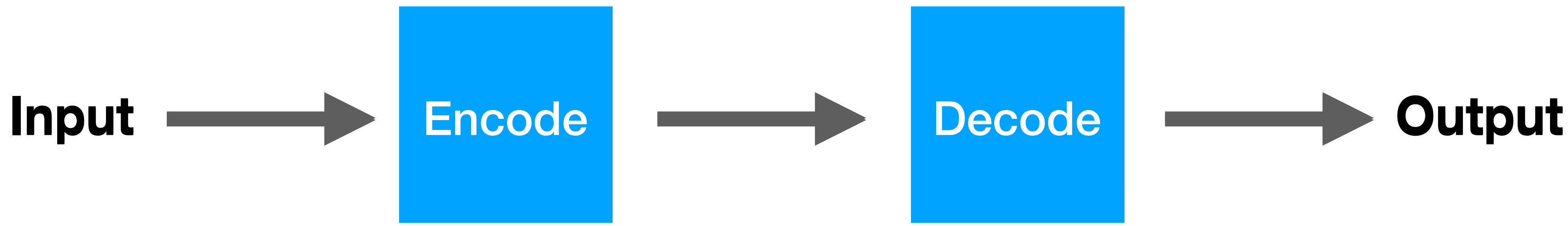
Positional information



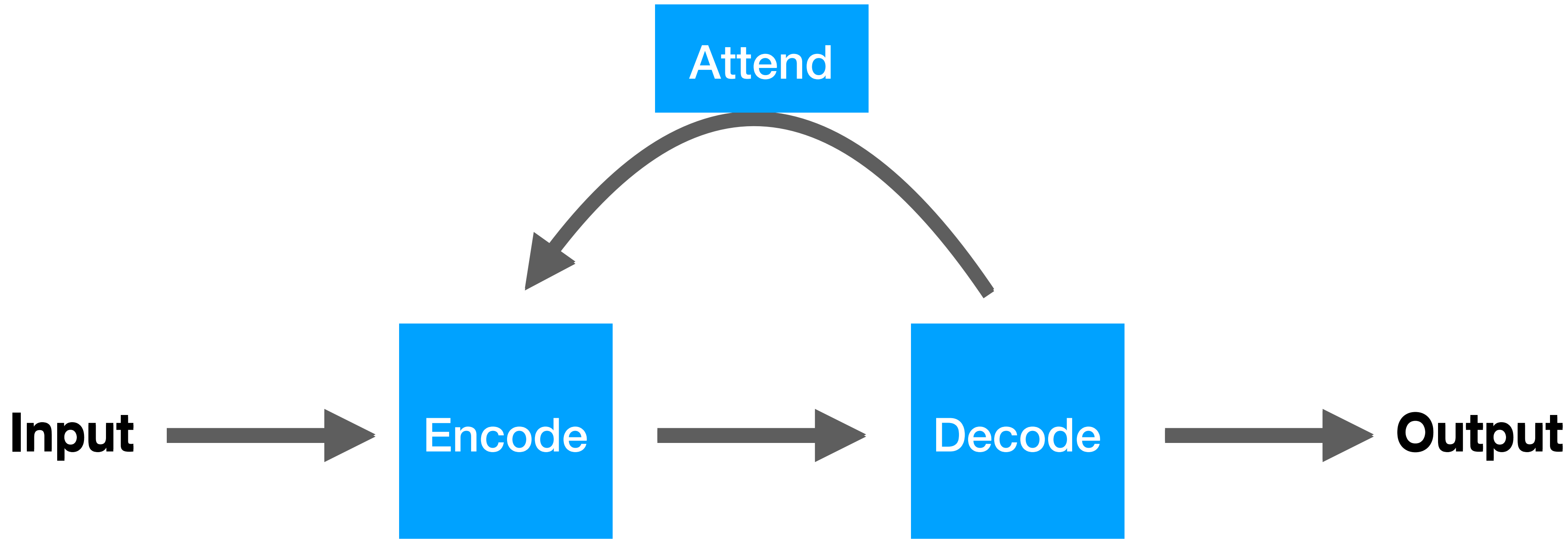
Neural NLP



The basic abstraction



The basic abstraction



Encoder abstarctions

symbol	→	Encode	→ vector	"embeddings"
n vectors	→	Encode	→ vector	Sum CNN + pooling RNN
n vectors	→	Encode	→ n vectors	RNN (~) Bi-RNN Transformer

Decoders

Linear, MLP (predict)	at single vector	one prediction
	at each position	input length

RNN

RNN + Attention

(Attention) Transformer

arbitrary length