Lecture 2: Introduction

CIS 7000: Trustworthy Machine Learning

Spring 2024

Agenda

- **Neural networks**
  - PyTorch
  - CNNs, RNNs, and transformers

Pytorch

- Open source packages have helped democratize deep learning

Pytorch

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```

Common parent class: nn.Module

Constructor: Defining layers of the network

```python
class Net(nn.Module):
    def __init__(self, in_features=10, num_classes=2, hidden_features=20):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.fc2 = nn.Linear(hidden_features, num_classes)

    def forward(self, x):
        x1 = self.fc1(x)
        x2 = F.relu(x1)
        x3 = self.fc2(x2)
        log_prob = F.log_softmax(x3, dim=1)

        return log_prob
```

Forward propagation

What about backward propagation?

Pytorch

- Open source packages have helped democratize deep learning

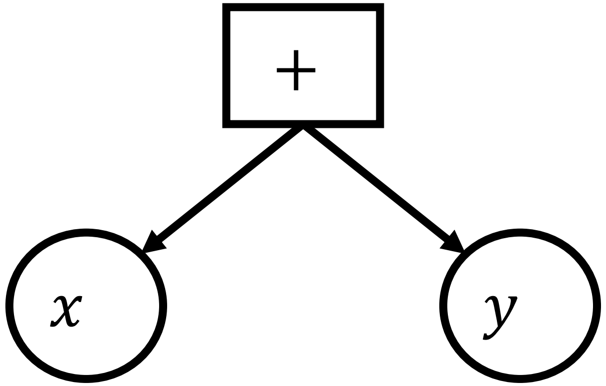
- Backpropagation implemented for all neural network architectures
  - Most modern libraries, including Tensorflow, Mxnet, Caffe, Pytorch, and Jax
  - Only need gradients of new layers

- **Basic Idea:** Provide model family as sequence of functions $[f_1, \ldots, f_m]$
  - What about more general compositions?
  - **Solution:** Composition of functions can be represented as graphs!

Computation Graphs

- The **tensor** datatype represents a **computation graph**
  - **Not just a numpy array!**
  - Instead, performing the computation produces a numpy array

- **Example:**
  - Suppose x is tensor that evaluates to $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
  - Suppose y is a tensor evaluates to $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
  - Then, $x + y$ is a tensor that evaluates to $\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$

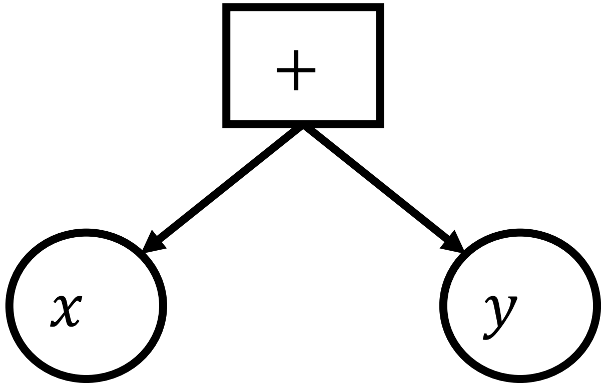


Toy Implementation of Computation Graphs

```
class Constant(tensor):
    def __init__(self, val):
        self.val = val
    def backpropagate(self):
        ...

class Add(tensor):
    def __init__(self, t1, t2):
        self.t1 = t1
        self.t2 = t2
        self.val = self.t1.val + self.t2.val
    def backpropagate(self):
        ...
```

```
x = Constant(np.array([[1, 0], [0, 1]])
y = Constant(np.array([[1, 1], [1, 0]])
z = x + y  # z has type Add
```

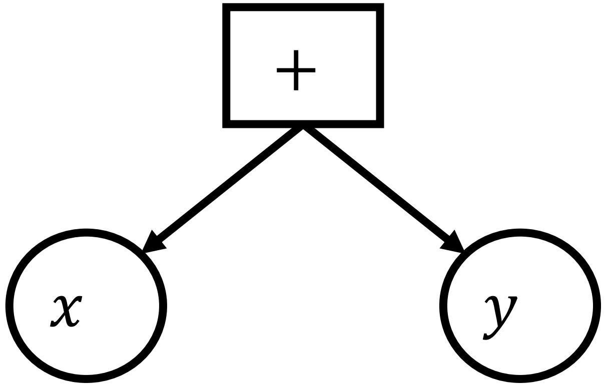


Toy Implementation of Computation Graphs

```
class Constant(tensor):
        def __init__(self, val):
                self.val = val
        def backpropagate(self):
                ...


class Add(tensor):
        def __init__(self, t1, t2):
                self.t1 = t1
                self.t2 = t2
                self.val = self.t1.val + self.t2.val
        def backpropagate(self):
                ...
```

```
x = Constant(np.array([[1, 0], [0, 1]])
y = Constant(np.array([[1, 1], [1, 0]])
z = x + x + y  # Z has type Add
```



Computation Graphs

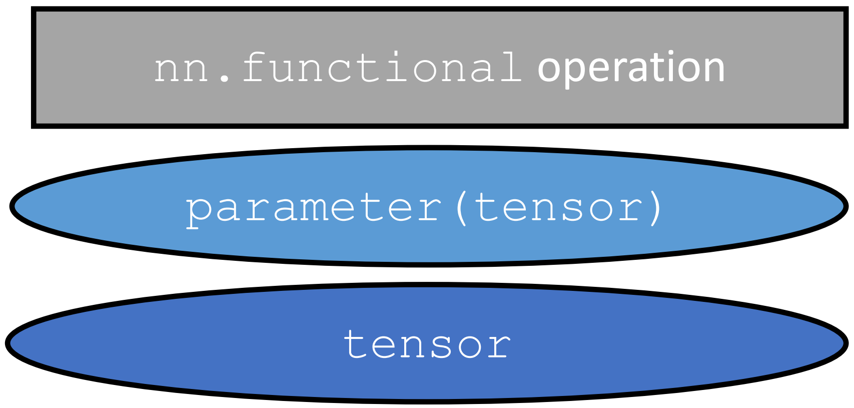
- Layers are implemented as tensors
  - **Examples:** addition, multiplication, ReLU, sigmoid, softmax, matrix multiplication/linear layers, MSE, logistic NLL, concatenation, etc.
  - You can also implement your own by providing forward pass and derivatives

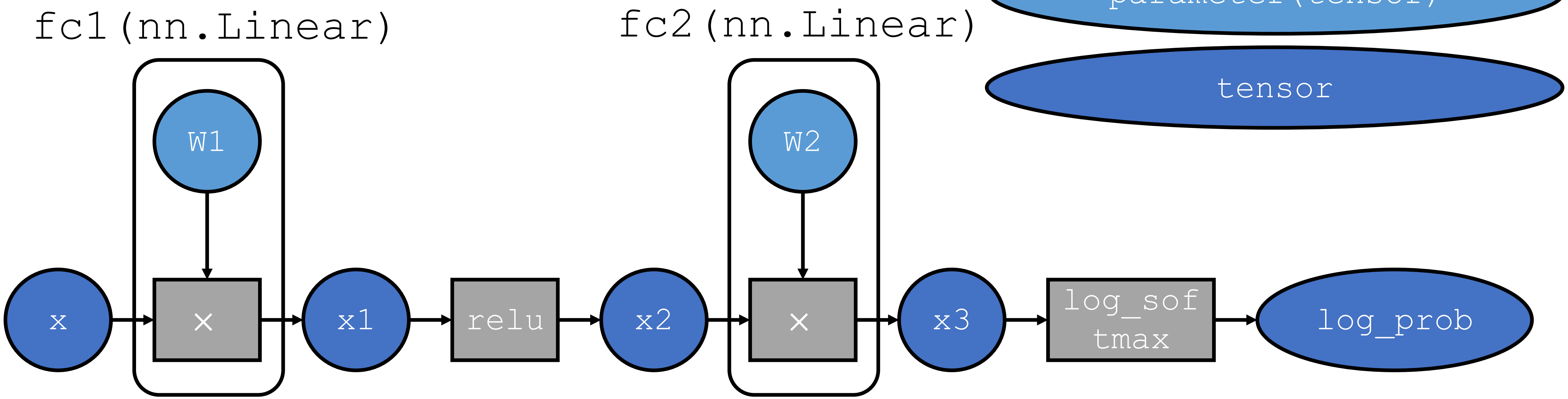
- Tensors can be composed together to form neural networks

Computation Graphs

- **Forward propagation:** Values are evaluated as they are constructed
- **Backpropagation:** Automatically compute derivative of scalar with respect to all parameters based on derivatives of layers
  - `x.backwards()`
  - Does not perform any gradient updates!

Computation Graphs






```python
    def forward(self, x):
        x1 = self.fc1(x)
        x2 = F.relu(x1)
        x3 = self.fc2(x2)
        log_prob = F.log_softmax(x3, dim=1)

        return log_prob
```

Pytorch Training Loop

```python
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

Looping over mini-batches

Zero out all old gradients

Runs forward pass model.forward(data)

Loss computation

Backpropagation

Gradient step

Pytorch Training Loop

```python
def main():
    torch.manual_seed(1)
    device = torch.device("cuda")
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=64, shuffle=True)

    model = Net().to(device)
    optimizer = optim.Adam(model.parameters(), lr=1e-4)
    sc                                       e=1, gamma=0.9)
    for epoch in range(1, 15):
        train(model, device, train_loader, optimizer, epoch)
        scheduler.step()
```

Load dataset

Loop over epochs (full passes over data)

Minibatch SGD for one epoch

Update base learning rate

Pytorch Model

- To use your model (once it has been trained):

```
label = model(input)
```

Agenda

- **Neural networks**
  - PyTorch
  - CNNs, RNNs, and transformers

Images as 2D Arrays

- Grayscale image is a 2D array of pixel values

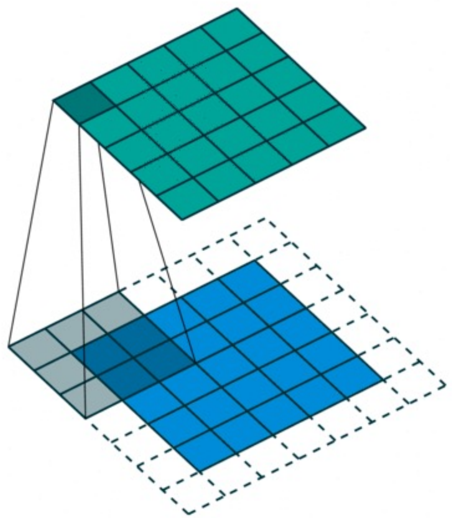
- Color images are 3D array
  - 3rd dimension is color (e.g., RGB)
  - Called "channels"

0	3	2	5	4	7	6	9	8
3	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
5	2	3	0	1	2	3	4	5
4	3	2	1	0	3	2	5	4
7	4	5	2	3	0	1	2	3
6	5	4	3	2	1	0	3	2
9	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

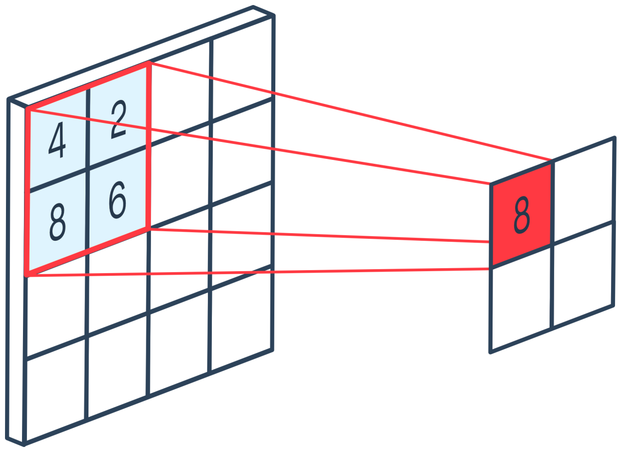
Source: S. Narasimhan, S. Lazebnik

Structure in Images

- Use layers that capture structure



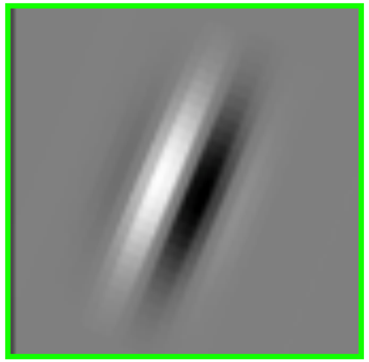
Convolution layers
(Capture equivariance)



Pooling layers
(Capture invariance)

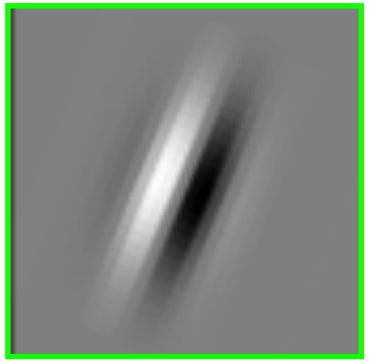
https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d
https://peltarion.com/static/2d_max_pooling_pa1.png

Convolution Filters



graphic credit: S. Lazebnik

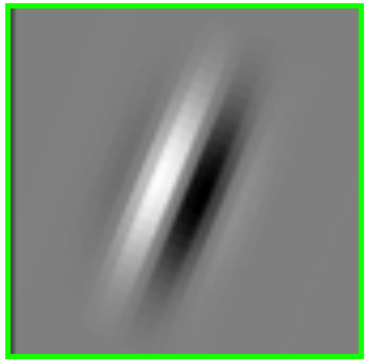
Convolution Filters



$$\text{output}[0,0] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[0+\tau, 0+\gamma]$$

graphic credit: S. Lazebnik

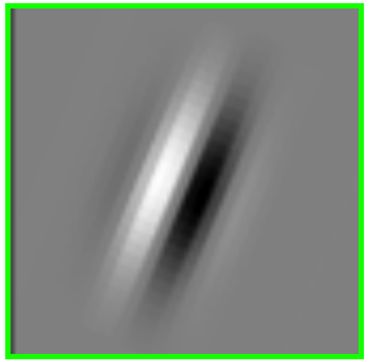
Convolution Filters



$$\text{output}[0,1] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[0+\tau, 1+\gamma]$$

graphic credit: S. Lazebnik

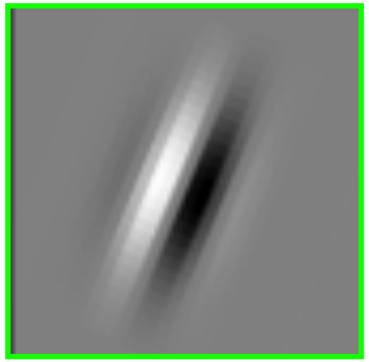
Convolution Filters



$$\text{output}[0,2] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau, \gamma] \cdot \text{image}[0 + \tau, 2 + \gamma]$$

graphic credit: S. Lazebnik

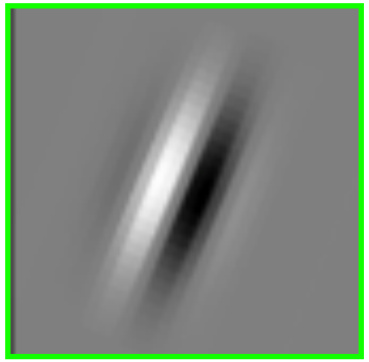
Convolution Filters

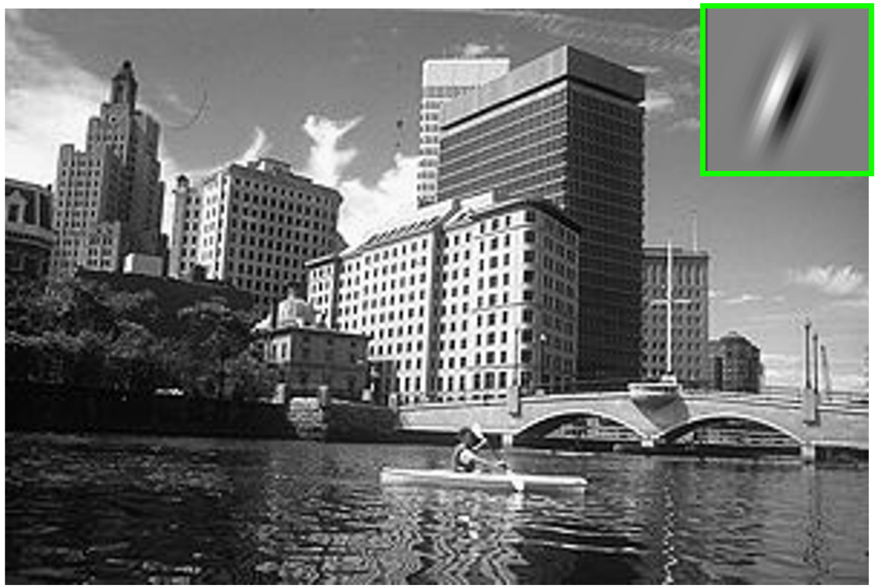


$$\text{output}[i,j] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

Convolution Filters

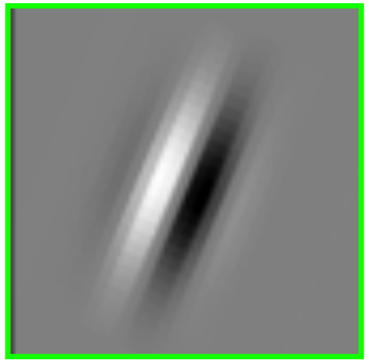


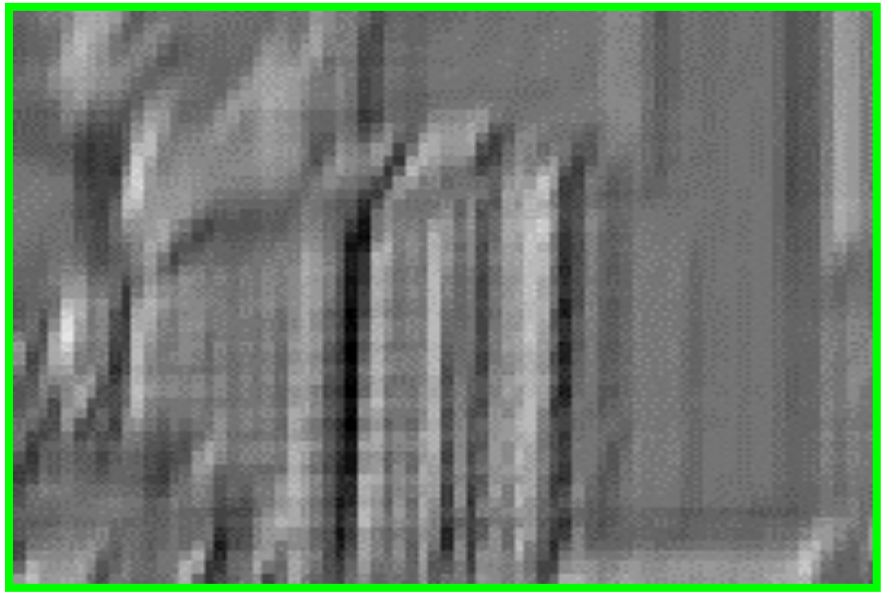


$$\text{output}[i,j] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

Convolution Filters





$$\text{output}[i,j] = \sum_{\tau=0}^{k-1} \sum_{\gamma=0}^{k-1} \text{filter}[\tau,\gamma] \cdot \text{image}[i+\tau, j+\gamma]$$

graphic credit: S. Lazebnik

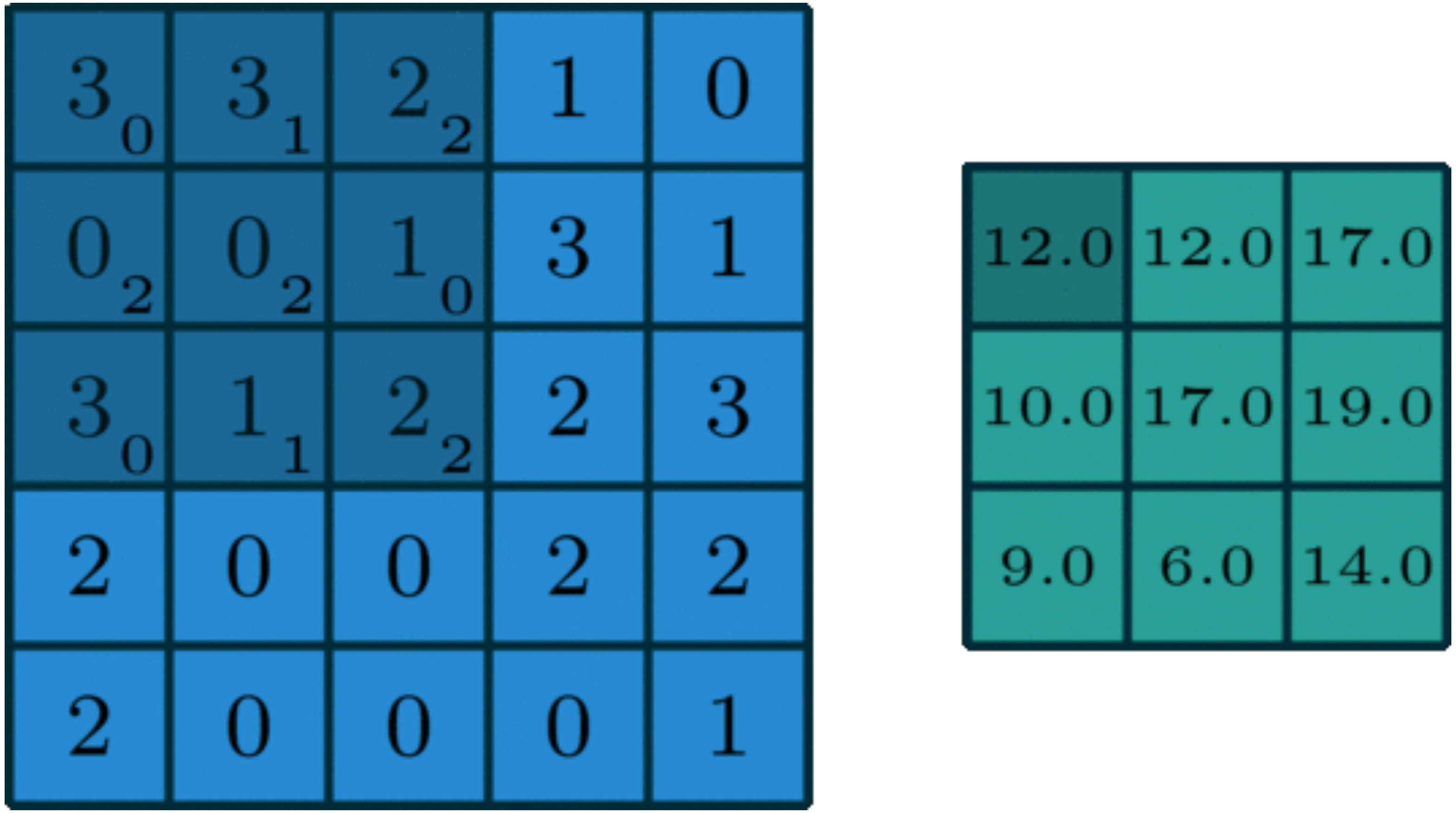
2D Convolution Filters

- **Given:**
  - A 2D input x
  - A 2D $h \times w$ kernel k

- The 2D convolution is:

$$y[s,t] = \sum_{\tau=0}^{h-1} \sum_{\gamma=0}^{w-1} k[\tau,\gamma] \cdot x[s+\tau, t+\gamma]$$

2D Convolution Filters



2D Convolution Filters

-1	-1	-1
2	2	2
-1	-1	-1

Horizontal lines

-1	2	-1
-1	2	-1
-1	2	-1

Vertical lines

-1	-1	2
-1	2	-1
2	-1	-1

45 degree lines

2	-1	-1
-1	2	-1
-1	-1	2

135 degree lines

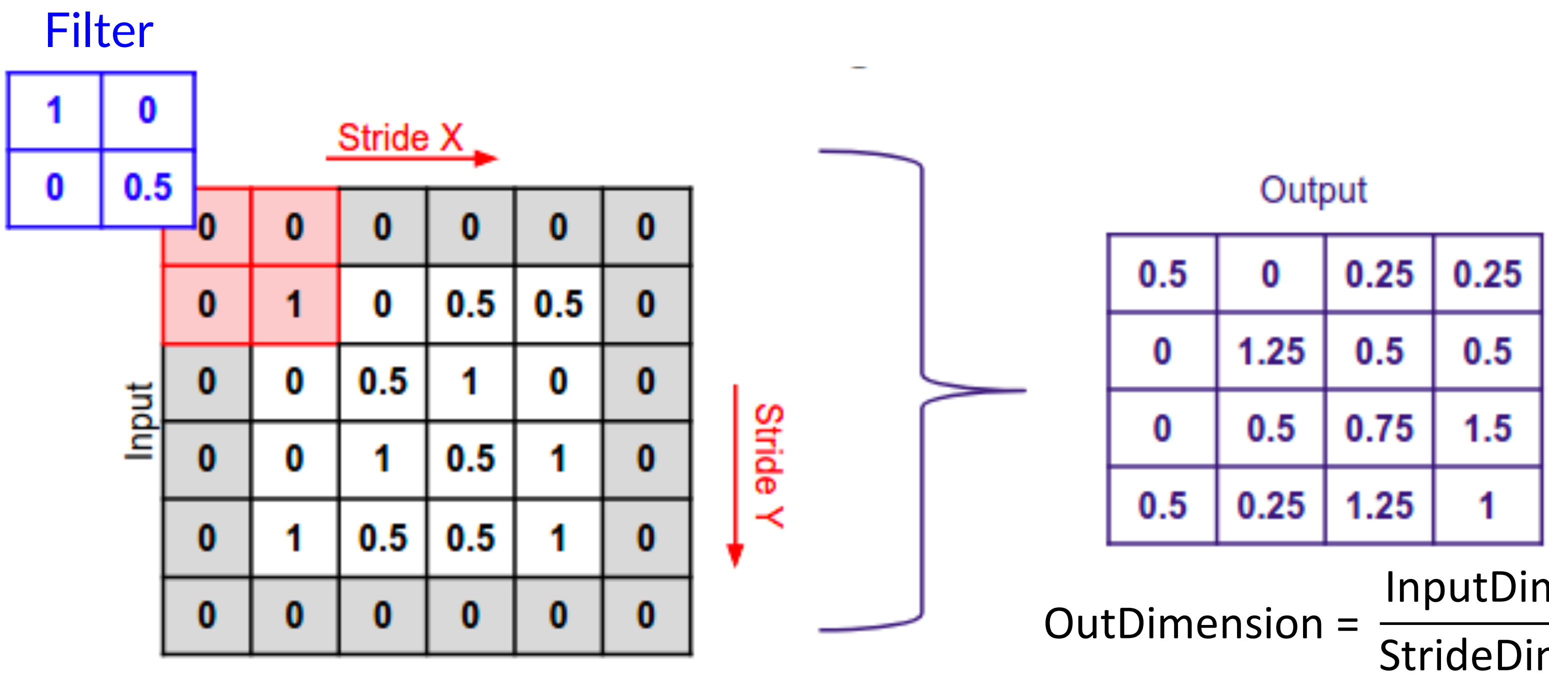
Example Edge Detection Kernels

Result of Convolution with Horizontal Kernel

https://aishack.in/tutorials/image-convolution-examples/

Convolution Layer Parameters

- **Stride:** How many pixels to skip (if any)
  - **Default:** Stride of 1 (no skipping)



Filter:

1	0
0	0.5

Input:

0	0	0	0	0	0
0	1	0	0.5	0.5	0
0	0	0.5	1	0	0
0	0	1	0.5	1	0
0	1	0.5	0.5	1	0
0	0	0	0	0	0

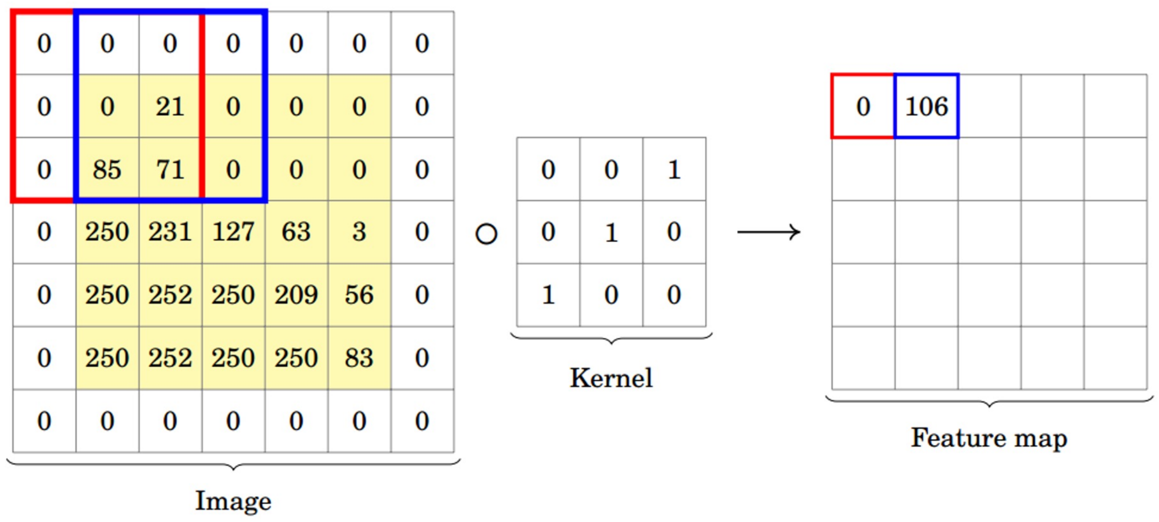
0.5	0	0.25	0.25
0	1.25	0.5	0.5
0	0.5	0.75	1.5
0.5	0.25	1.25	1

$$\text{OutDimension} = \frac{\text{InputDimension}}{\text{StrideDimension}}$$

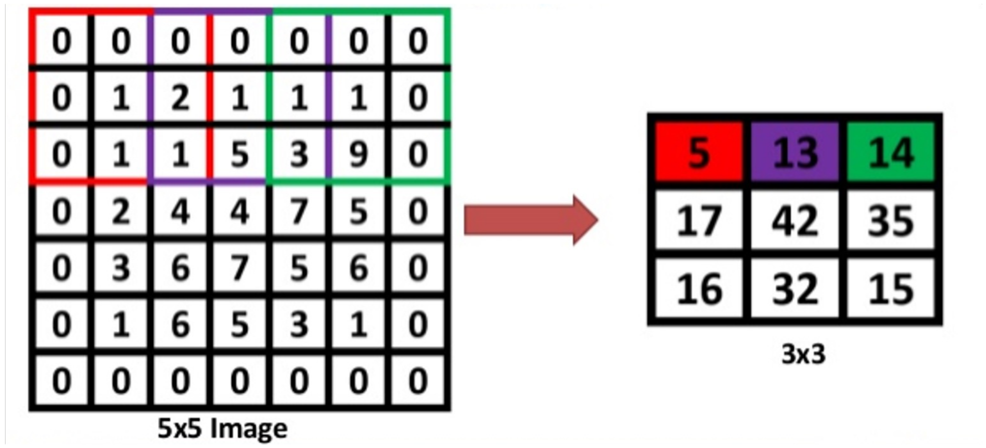
https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7

Convolution Layer Parameters

- **Padding:** Add zeros to edges of image to capture ends
  - **Default:** No padding



stride = 1, zero-padding = 1



stride = 2, zero-padding = 1

https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7

Convolution Layer Parameters

- **Summary:** Hyperparameters
  - Kernel size
  - Stride
  - Amount of zero-padding
  - Output channels
- Together, these determine the relationship between the input tensor shape and the output tensor shape
- Typically, also use a single bias term for each convolution filter

Convolution Layers

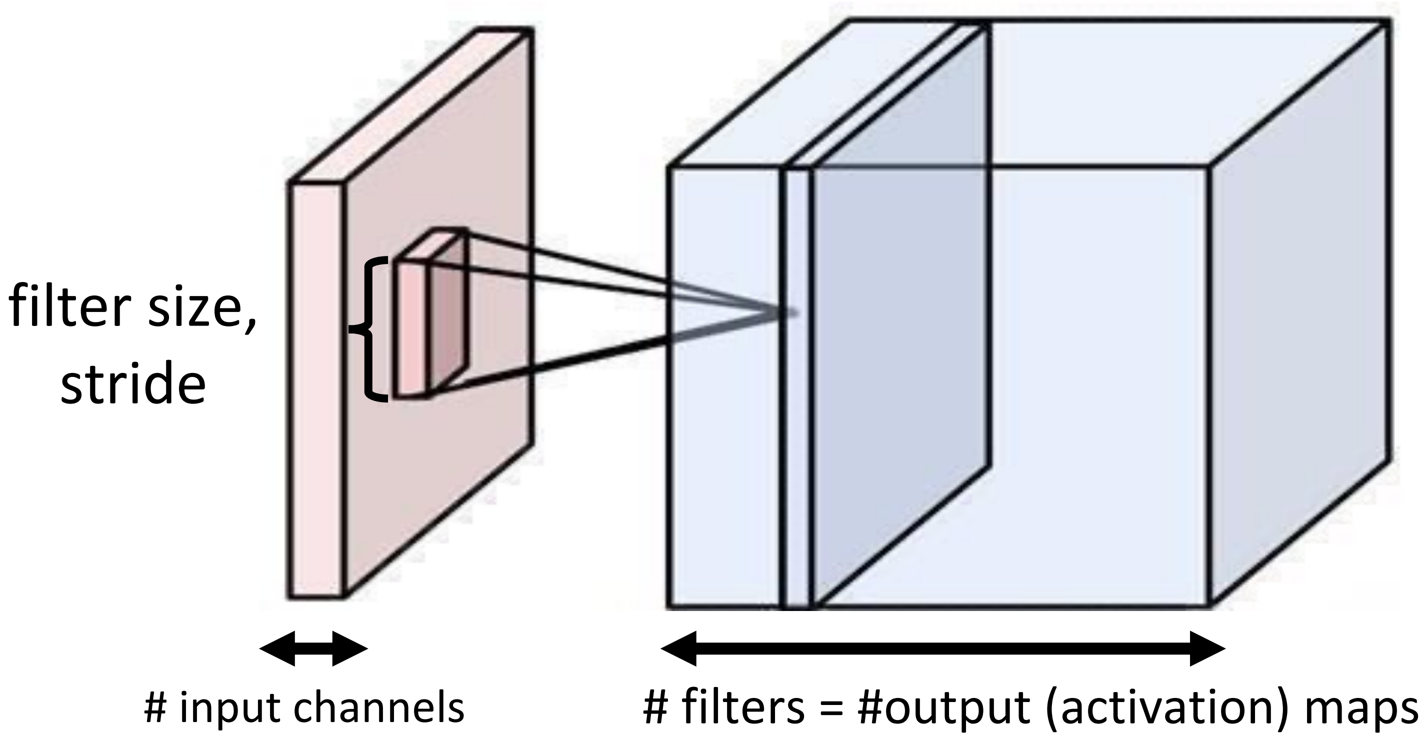
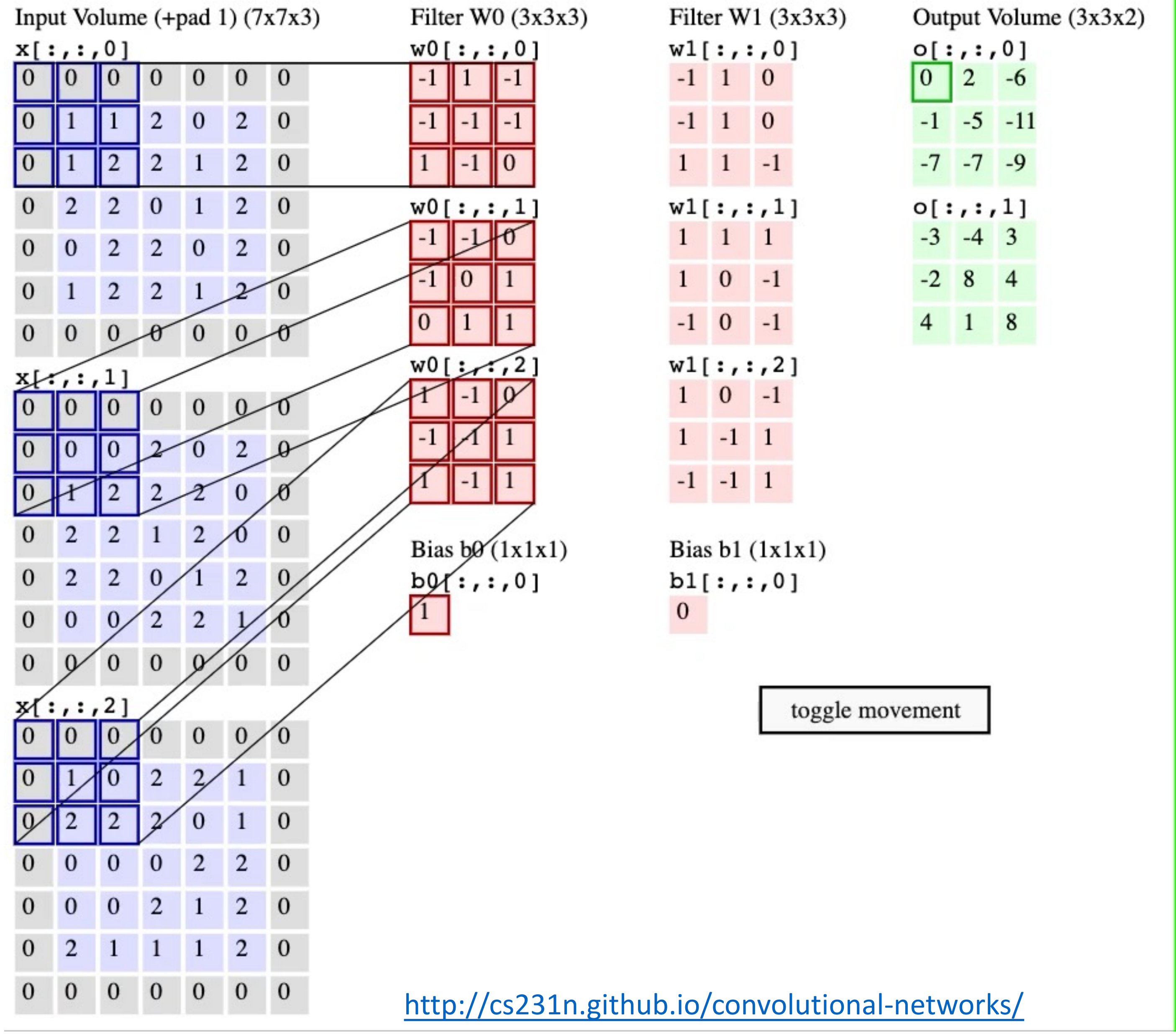


Image credit: A. Karpathy Slide credit: Jia-Bin Huang

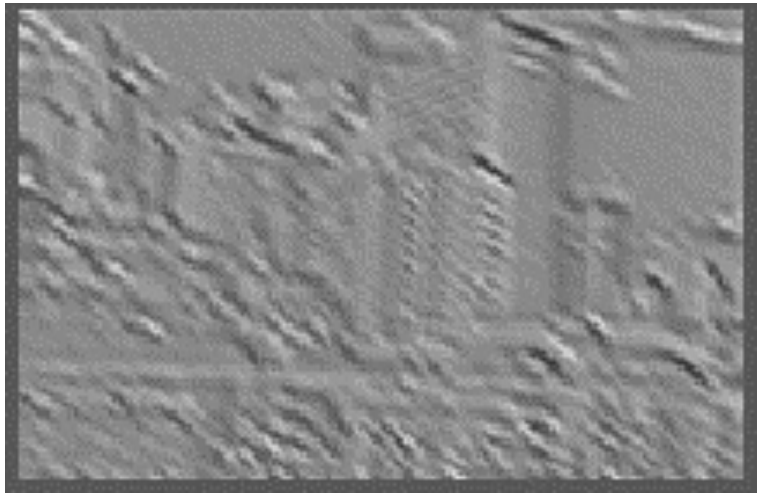
Example

- Kernel size 3, stride 2, padding 1
- 3 input channels
  - Hence kernel size 3×3×3
- 2 output channels
  - Hence 2 kernels
- Total # of parameters:
  - $(3\times3\times3+1)\times2 \;=\; 56$

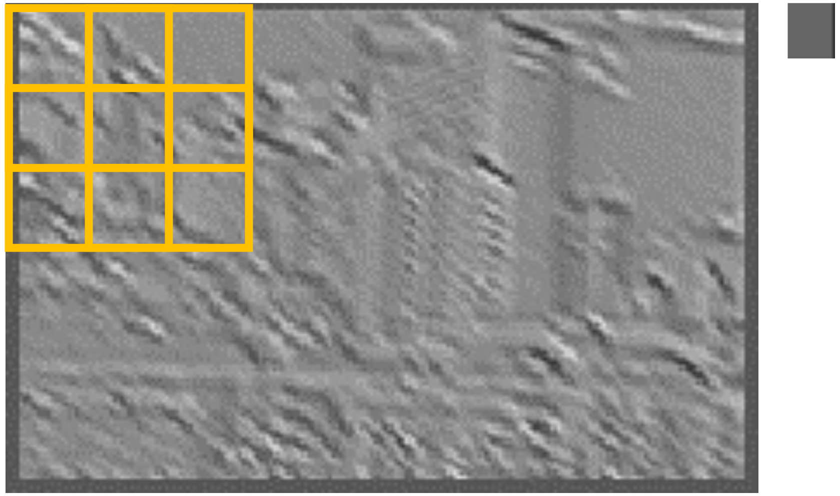


http://cs231n.github.io/convolutional-networks/

Pooling Layers

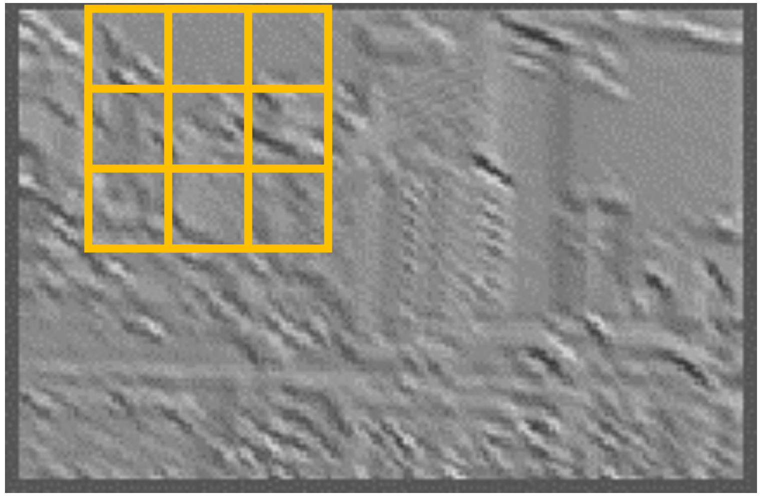


Pooling Layers



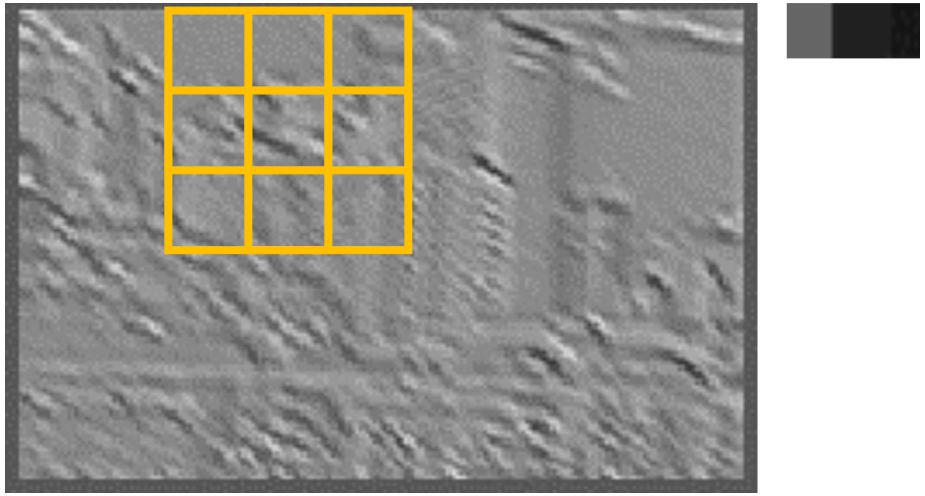
$$\text{output}[0,0] = \max_{0 \le \tau < k} \max_{0 \le \gamma < k} \text{image}[0+\tau, 0+\gamma]$$

Pooling Layers



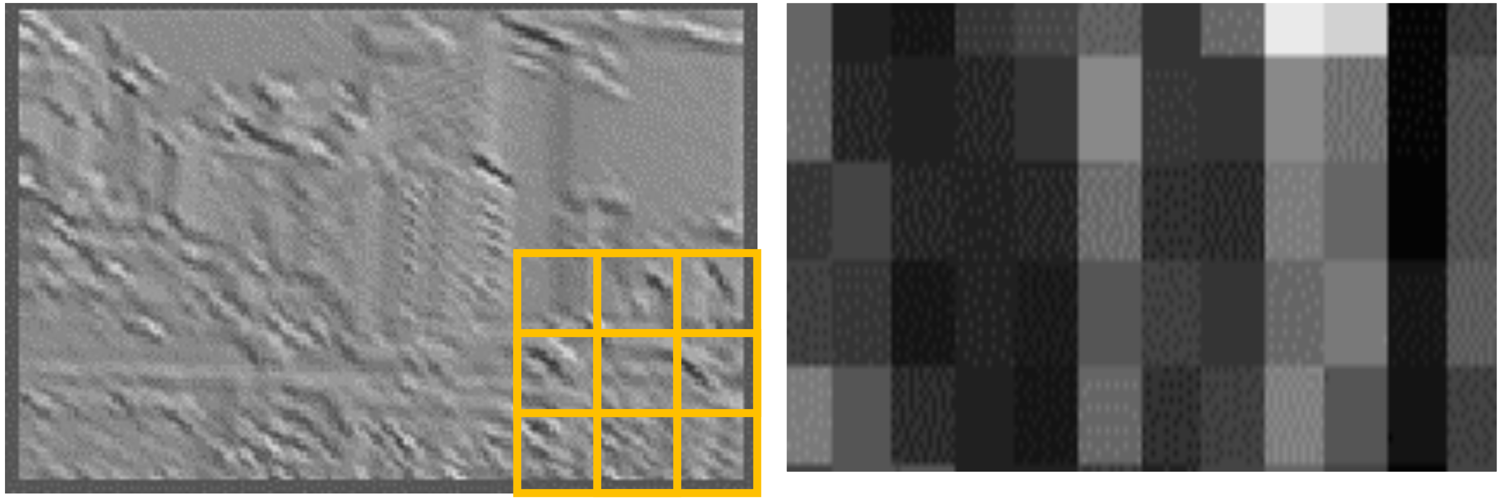
$$\text{output}[0,1] = \max_{0 \le \tau < k} \max_{0 \le \gamma < k} \text{image}[0 + \tau, 1 + \gamma]$$

Pooling Layers



$$\text{output}[0,2] = \max_{0 \le \tau < k} \max_{0 \le \gamma < k} \text{image}[0+\tau, 2+\gamma]$$

Pooling Layers



$$\text{output}[i,j] = \max_{0 \le \tau < k} \max_{0 \le \gamma < k} \text{image}[i+\tau, j+\gamma]$$

Pooling Layers

- **Summary:** Hyperparameters
  - Kernel size
  - Stride (usually >1)
  - Amount of zero-padding
  - Pooling function (almost always "max")

- Together, these determine the relationship between the input tensor shape and the output tensor shape

- **Note:** Unlike convolution, pooling operates on channels separately
  - Thus, n input channels → n output channels

Example Architecture: AlexNet

- **ImageNet dataset**
  - 1000 class image classification problem (e.g., grey fox, tabby cat, barber chair)
  - >1M image-label pairs gathered from internet and crowdsourced labels

- **AlexNet Architecture (Krizhevsky 2012)**
  - Historically important architecture
  - Image classification network (~60M parameters)
  - Trained using GPUs on ImageNet dataset
  - Huge improvement in performance compared to prior state-of-the-art

Example Architecture: AlexNet

Fully connected (i.e., linear) layers

output

fc, 1000

fc, 4096

fc, 4096

Input

3x3 conv, 256, pool/2

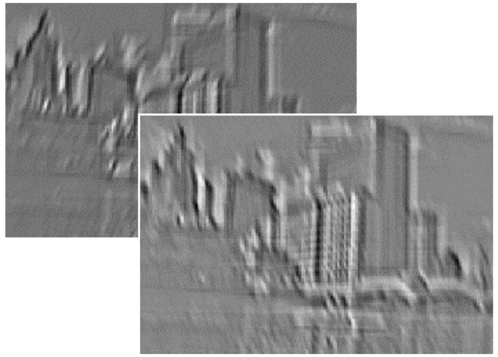
3x3 conv, 384

3x3 conv, 384

5x5 conv, 256, pool/2

11x11 conv, 96, /4, pool/2

input



Local Response Normalization

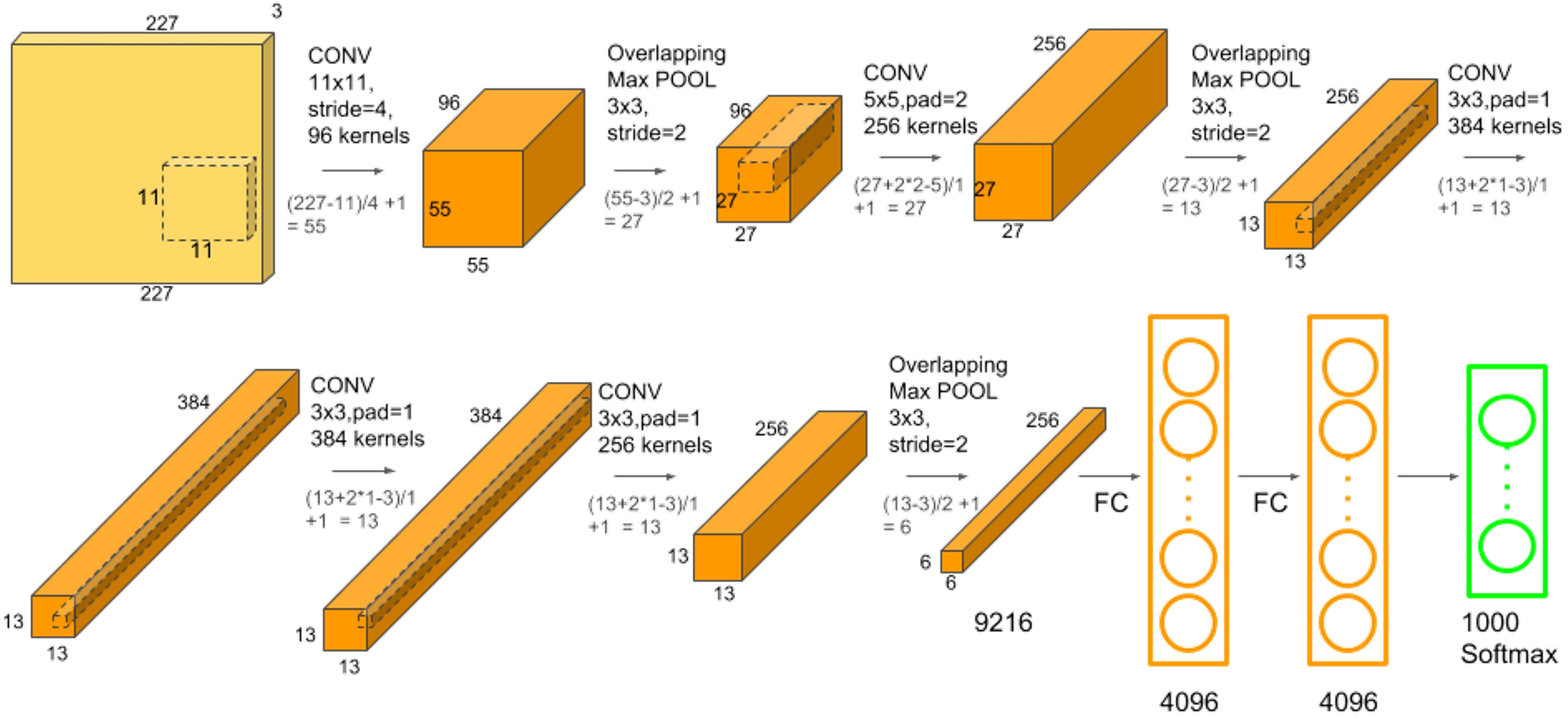
Pooling (kernel size 3, stride 2, no padding)

ReLU Activation

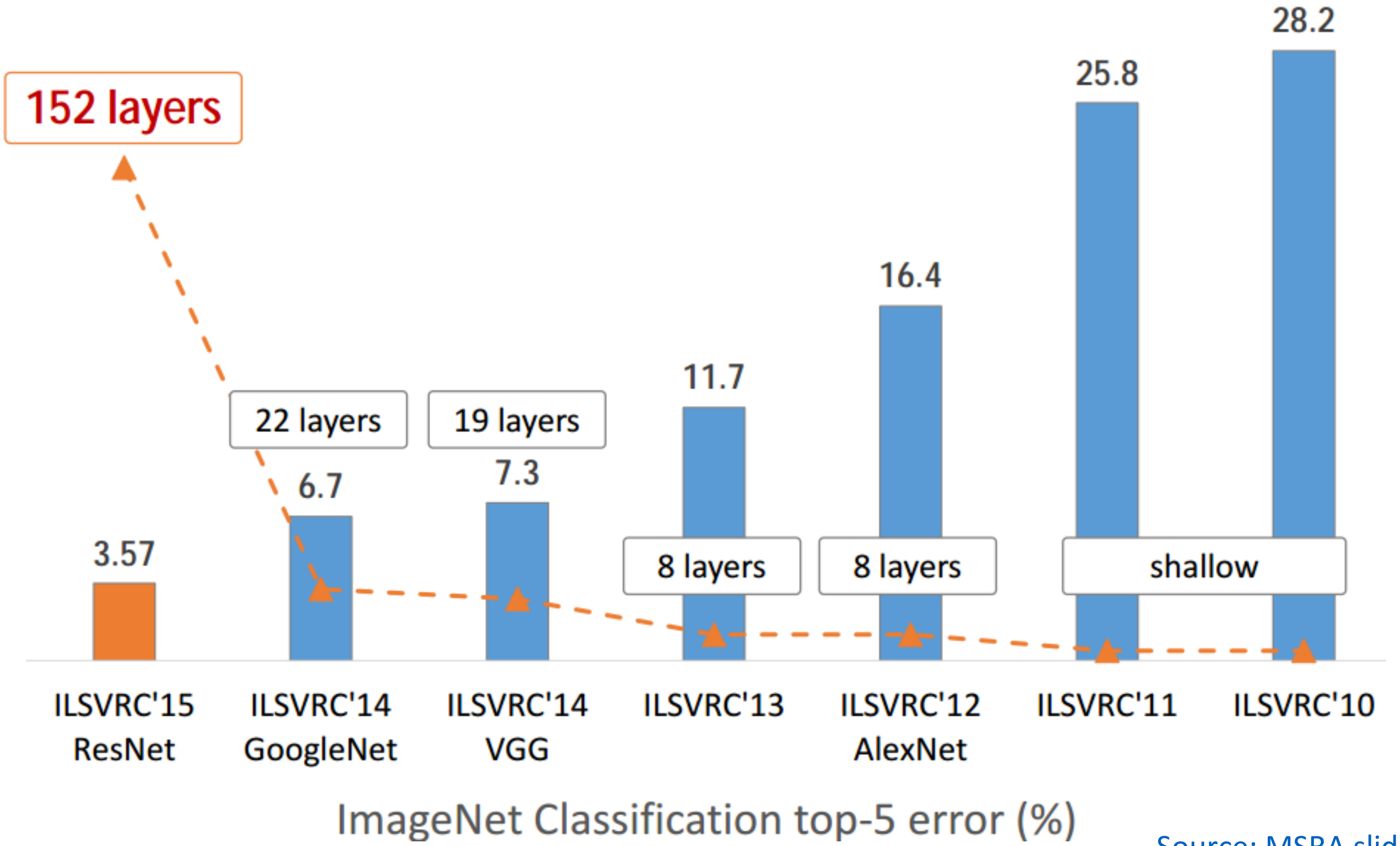
Convolution (kernel size 11, stride 4, 96 output channels, no padding)

slide credit: S. Lazebnik

Example Architecture: AlexNet



Evolution of Neural Networks



Source: MSRA slides at ILSVRC15

Evolution of Neural Networks

AlexNet, 8 layers
(ILSVRC 2012)
~60M params

VGG, 19 layers
(ILSVRC 2014)
~140M params

ResNet, 152 layers
(ILSVRC 2015)

Less computation
in forward pass
than VGGNet!

Back to 60M params

GoogleNet, 22 layers
(ILSVRC 2014)
~5M params

ICCV15

[Source: MSRA slides at ILSVRC15]

Agenda

- **Neural networks**
  - PyTorch
  - CNNs, RNNs, and transformers

- **Distribution shift robustness**
  - Basic examples and definitions

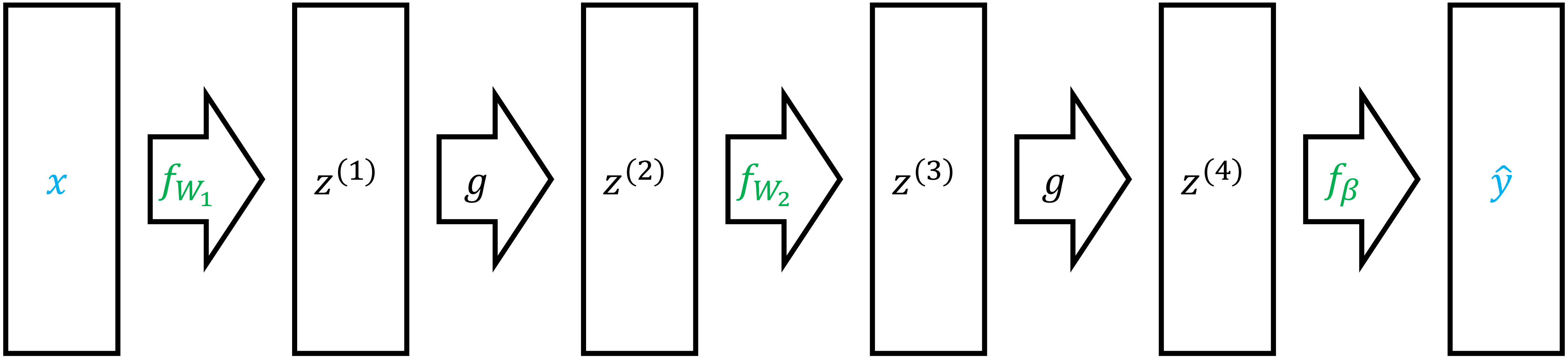
Recurrent Neural Networks

- Handle inputs/outputs that are **sequences**

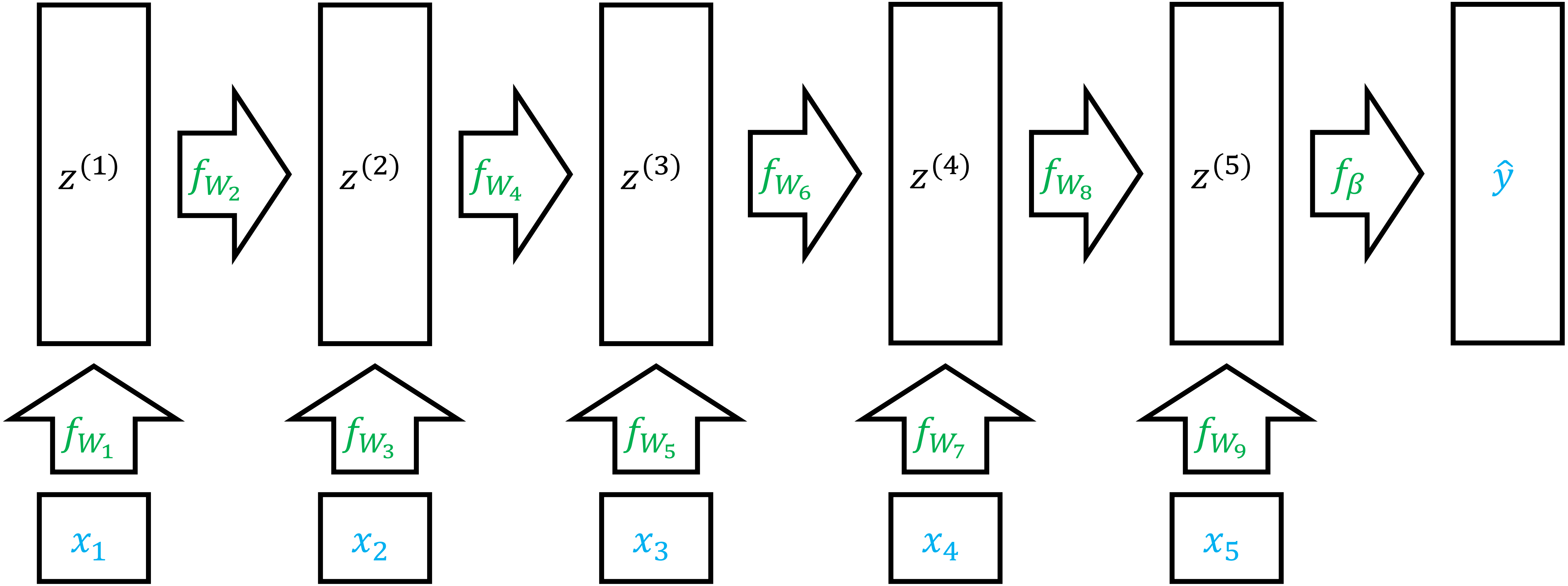
- **Naïve strategy**
  - Pad inputs to fixed length and use feedforward network
  - **Ignores temporal structure**

- **Recurrent neural networks (RNNs):** Process input sequentially

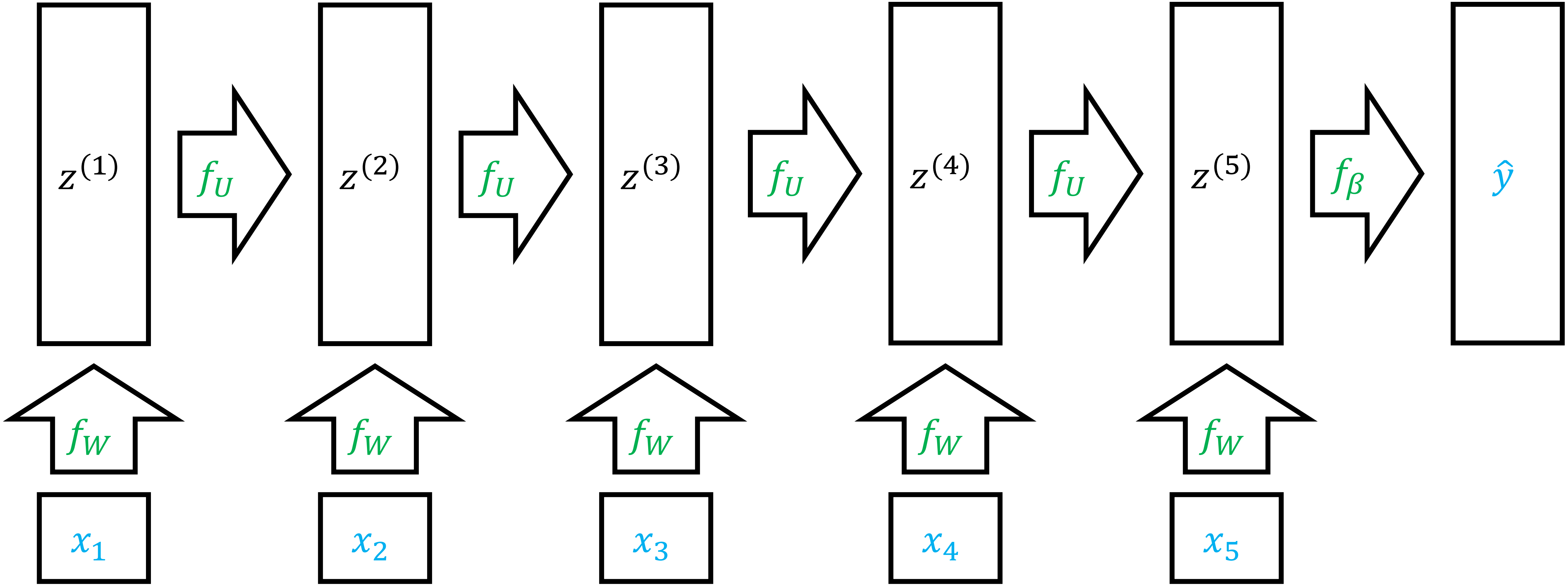
Feedforward Neural Networks



Recurrent Neural Networks



Recurrent Neural Networks



Recurrent Neural Networks

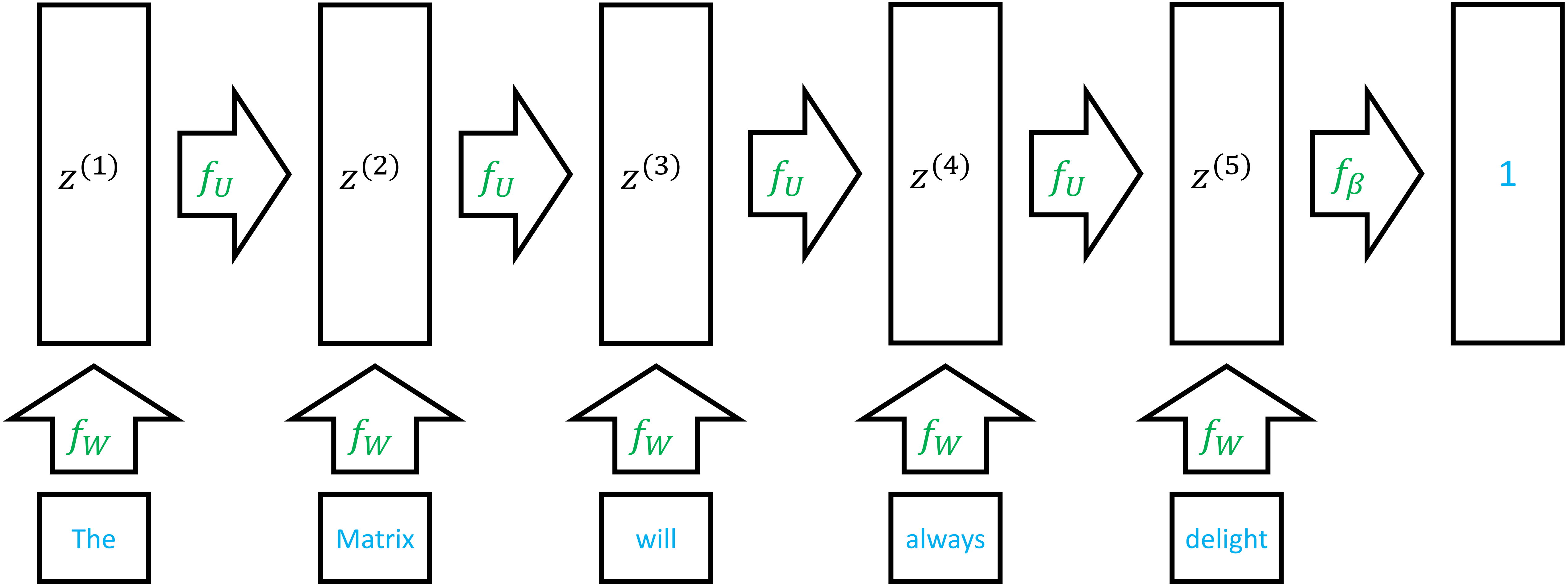
- Initialize $z^{(0)} = \vec{0}$
- Iteratively compute (for $t \in \{1, \dots, \mathrm{T}\}$):

$$z^{(t)} = g\left(W x_t + U z^{(t-1)}\right)$$

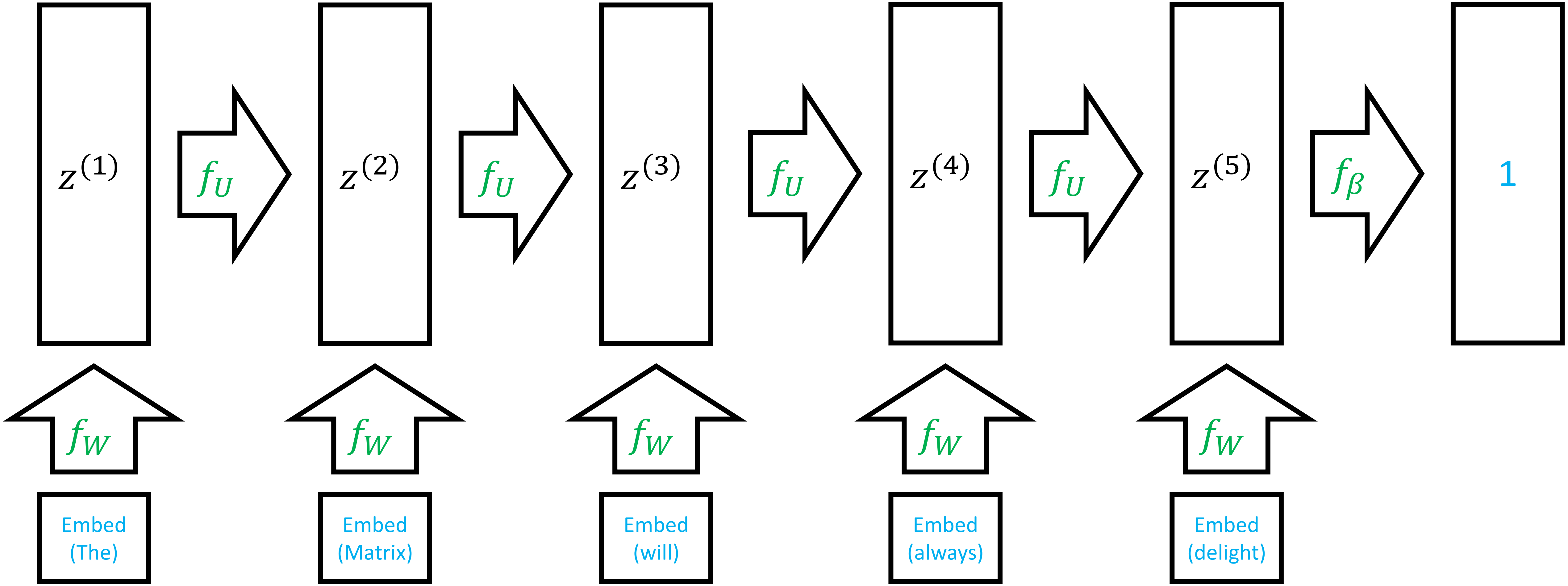
- Compute output:

$$y = \beta^\top z^{(T)}$$

Sentiment Classification



Sentiment Classification



Recurrent Neural Networks

- Initialize $z^{(0)} = \vec{0}$
- Iteratively compute (for $t \in \{1, \dots, \mathrm{T}\}$):

$$z^{(t)} = g\left(W\ \mathrm{Embed}(x_t) + Uz^{(t-1)}\right)$$

- Compute output:

$$y = \beta^\top z^{(T)}$$

Recurrent Neural Networks

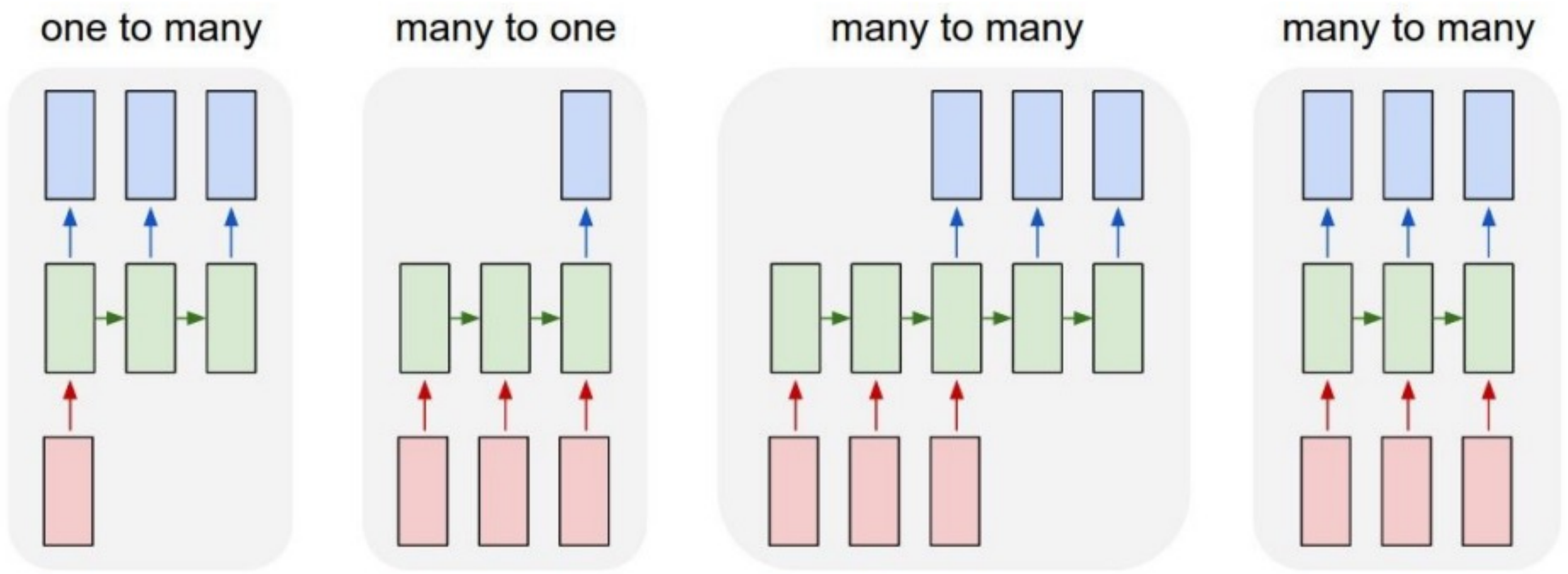
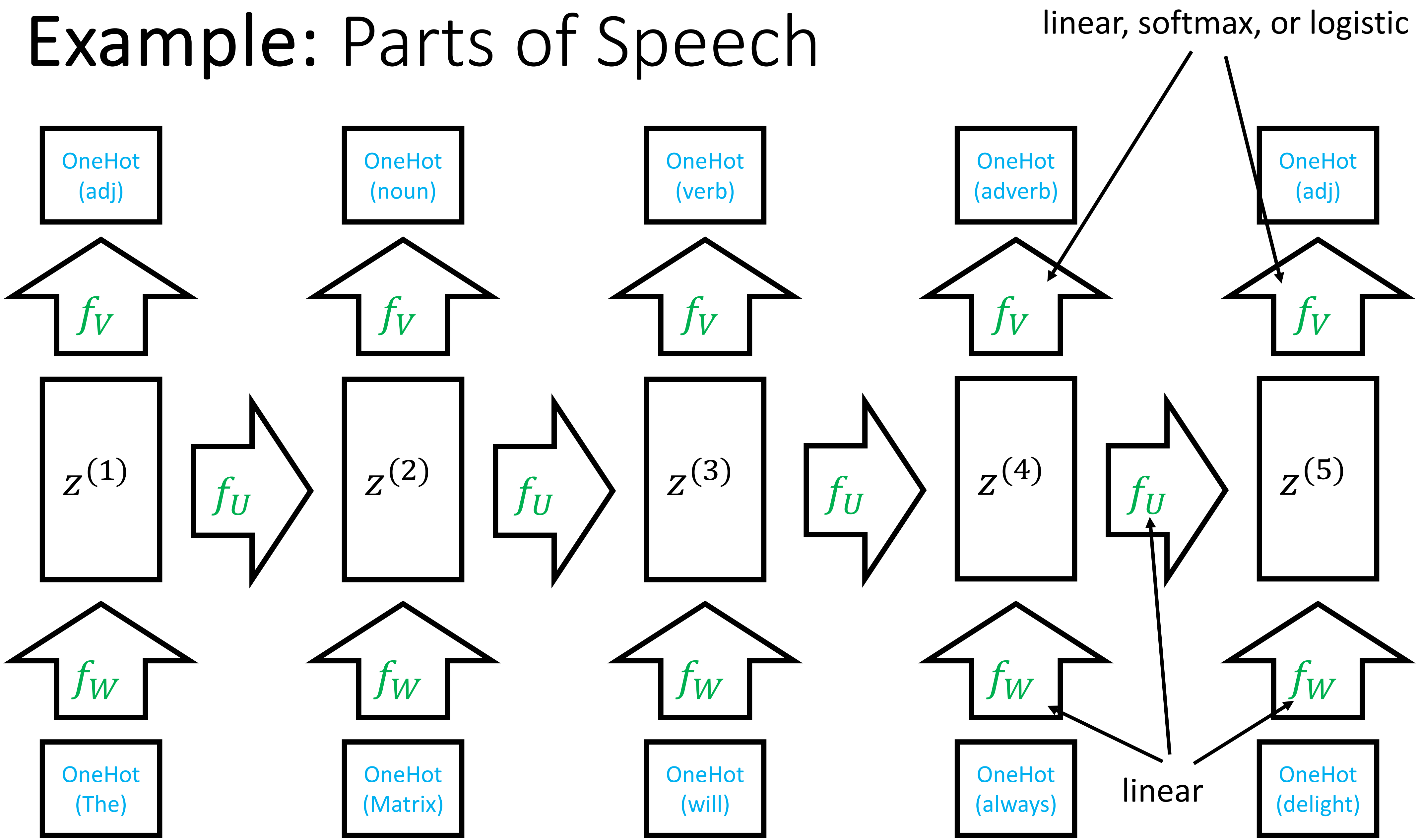


Image captioning | Sentiment prediction | Machine translation | Video captioning

Example: Parts of Speech



Training RNNs

- Backpropagation works as before
  - For shared parameters, we can show that the overall gradient is sum of gradient at each usage

- LSTM (“long short-term memory”) and GRU (“gated recurrent unit”) do clever things to better maintain hidden state

Training RNNs

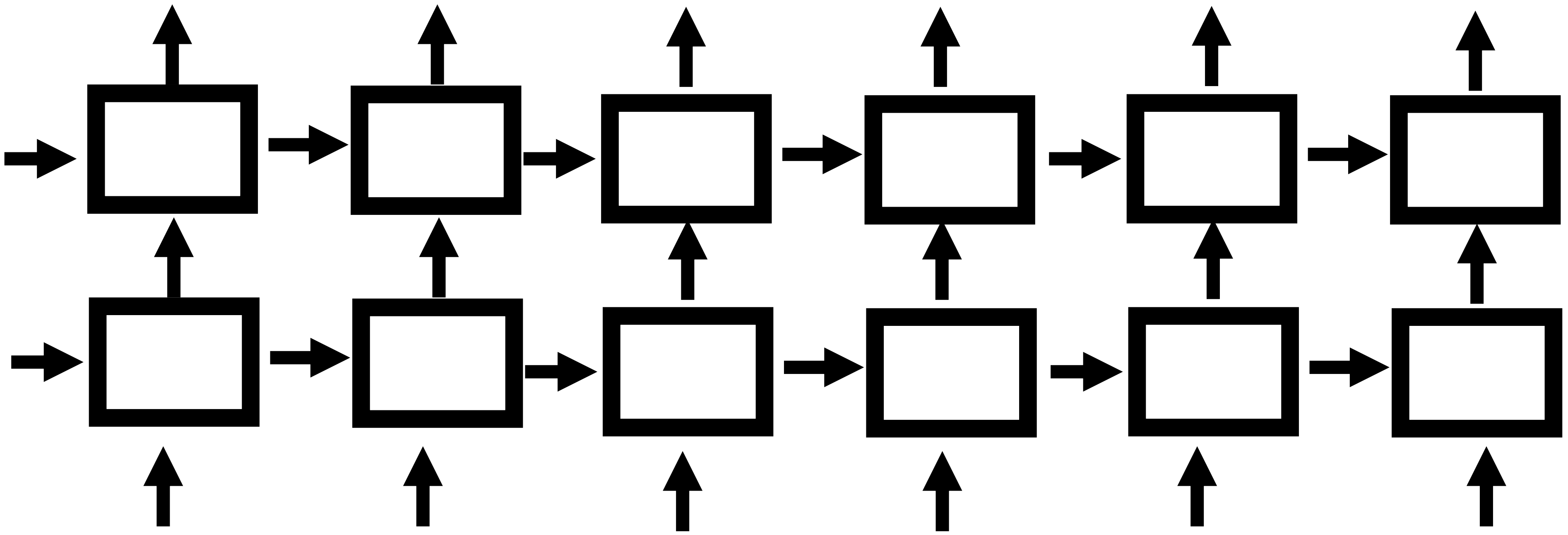
$$z_1 = g(Wx_1 + Uz_0)$$
$$z_2 = g(Wx_2 + Uz_1)$$
$$z_3 = g(Wx_3 + Uz_2)$$

$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial U} + \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial U} + \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1}\frac{\partial z_1}{\partial U}$$

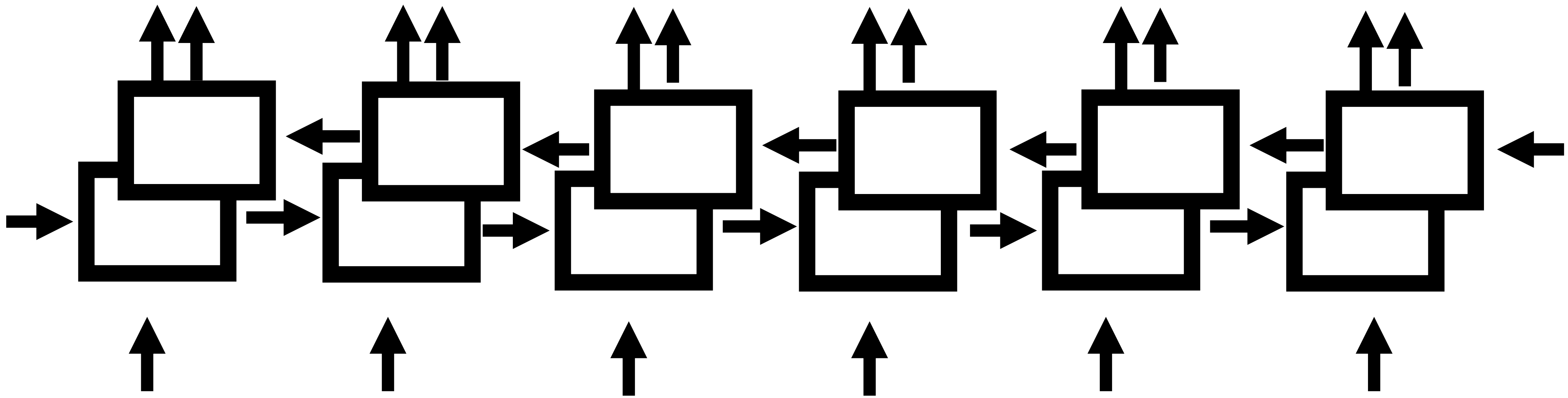
$$\longrightarrow$$

Local Contribution Historical Contribution

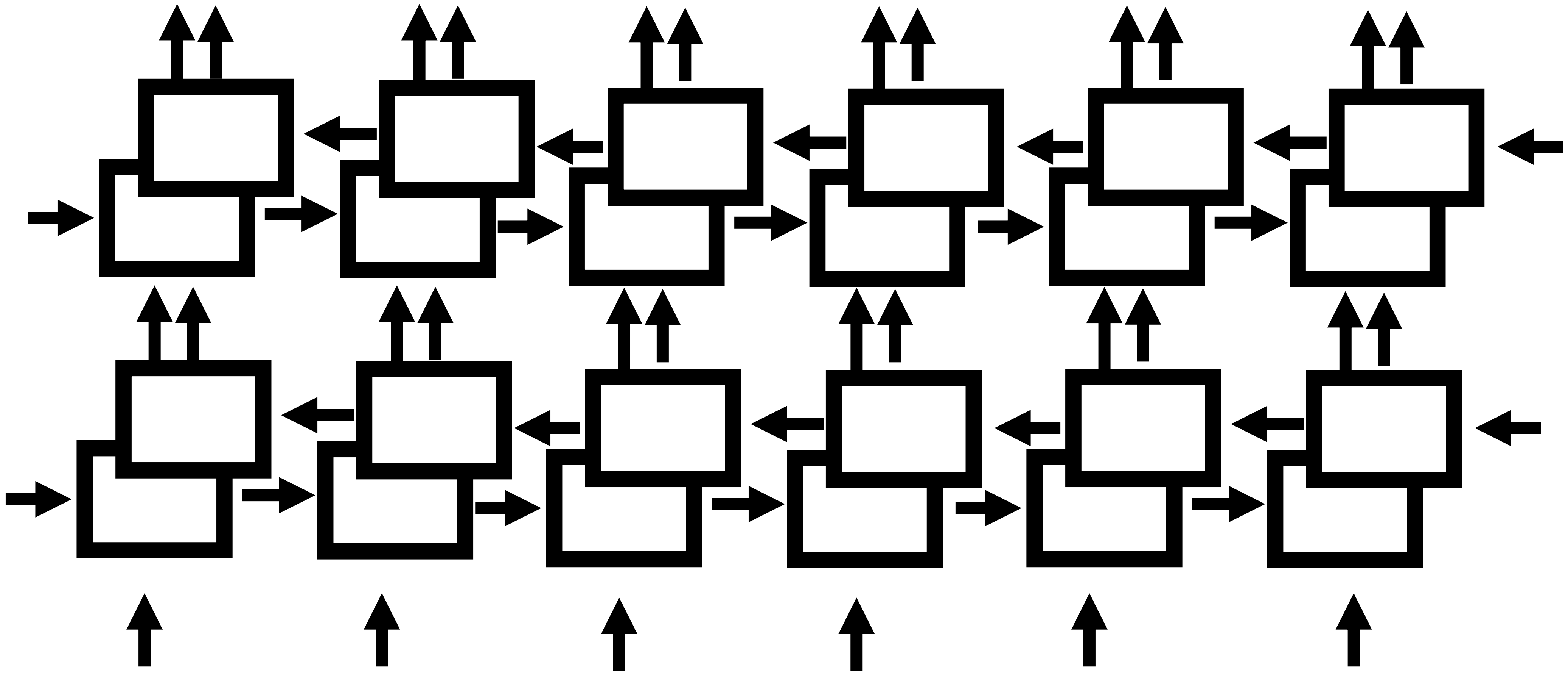
Stacked RNN



Bidirectional RNN

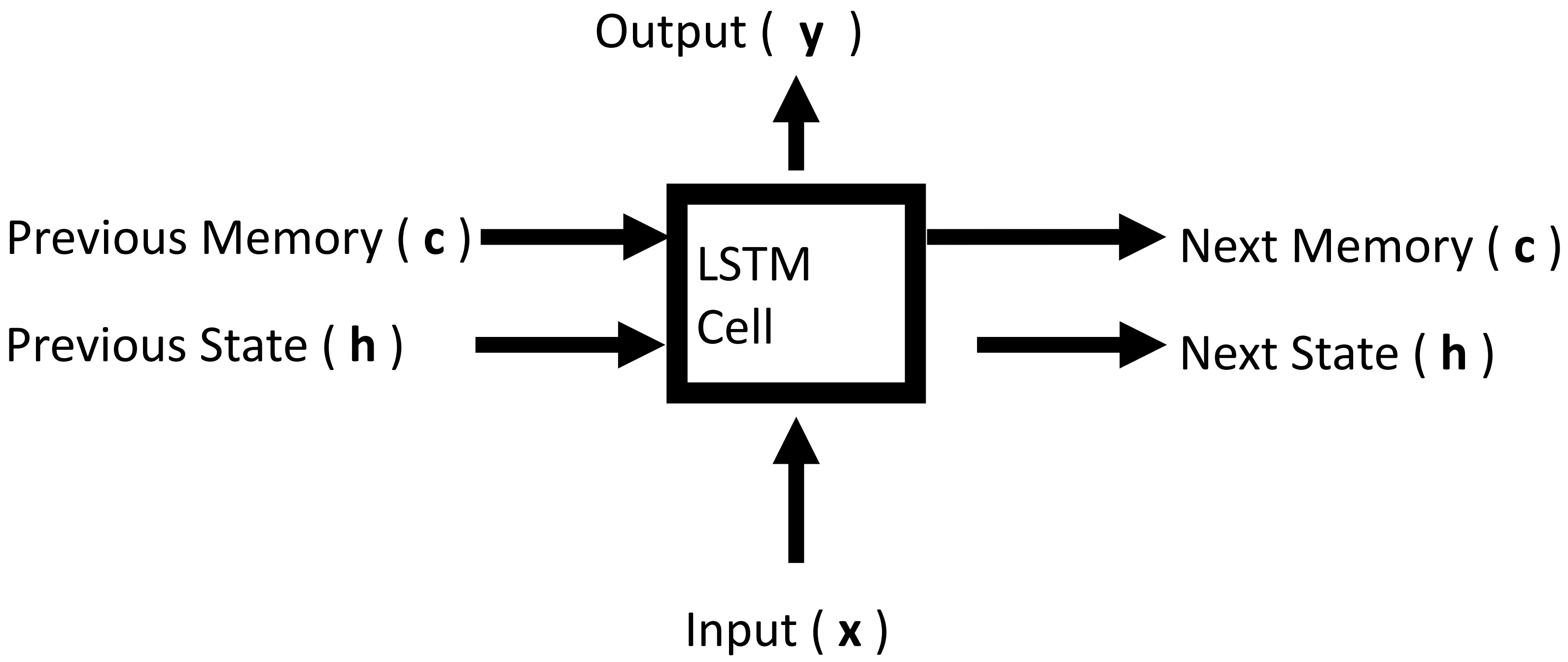


Stacked + Bidirectional RNN



Long Short Term Memory

- **Goal:** Replace some multiplicative relationships in hidden state with additive relationships



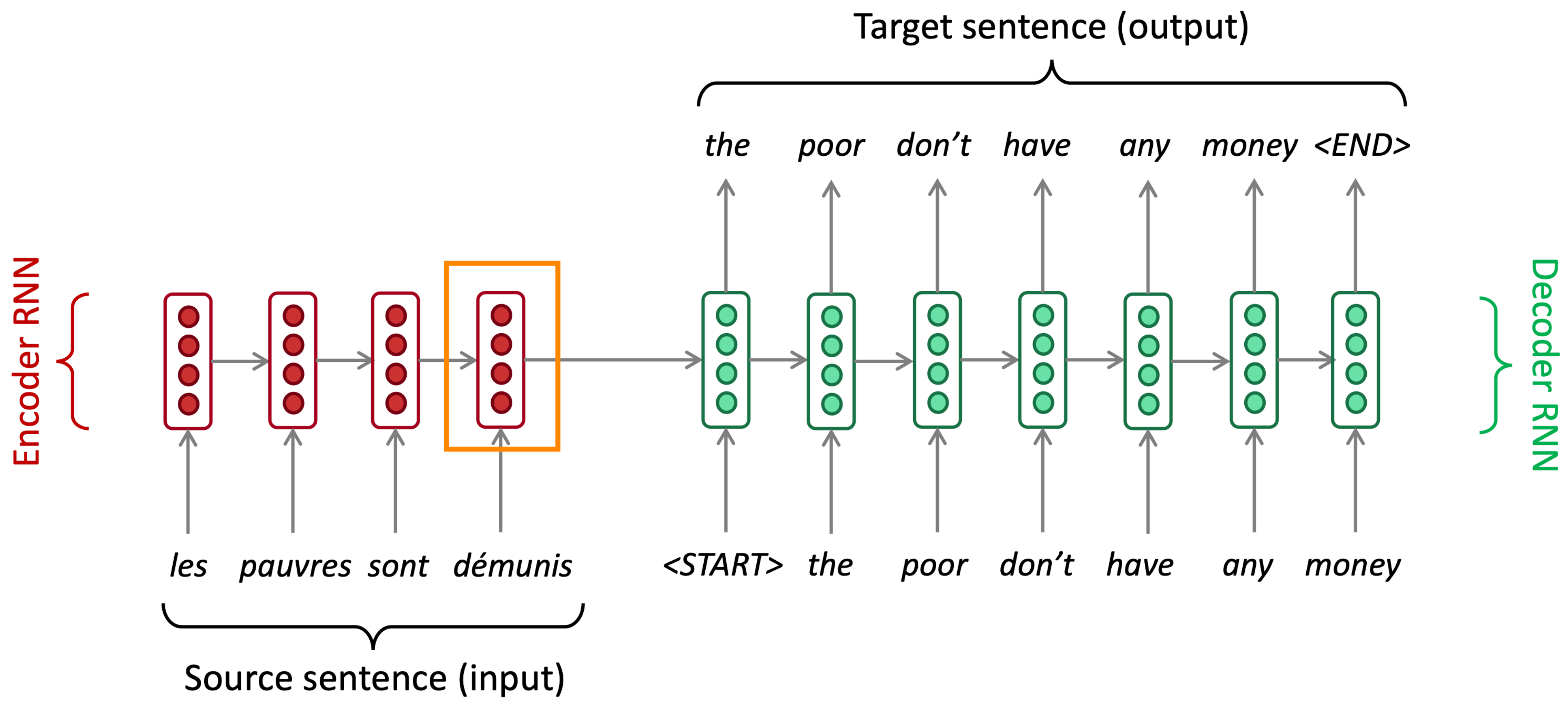
Agenda

- **Neural networks**
  - PyTorch
  - CNNs, RNNs, and transformers

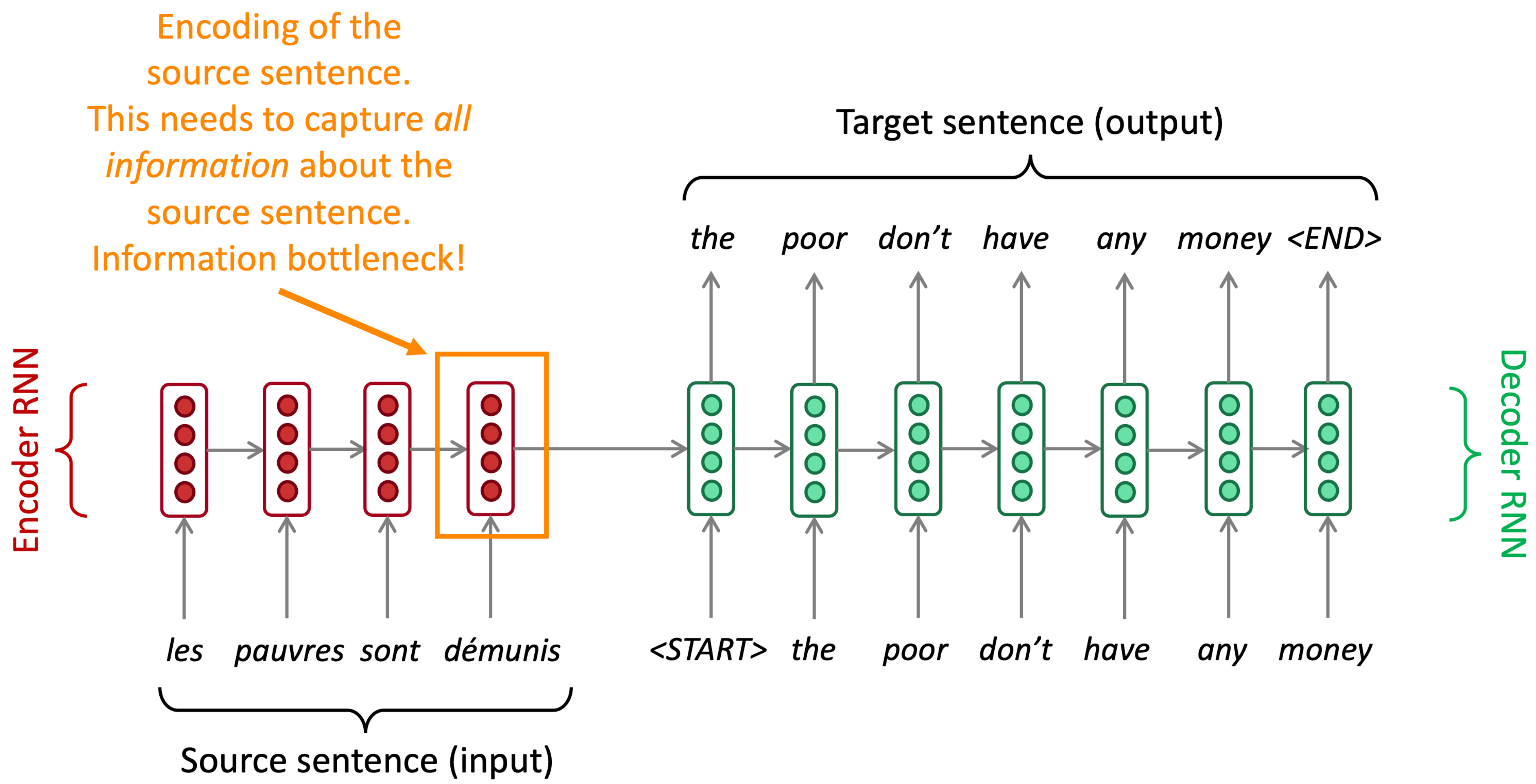
Attention

- RNNs have trouble propagating information forwards
- **Solution:** Let RNN “pay attention” to small part of past sequence

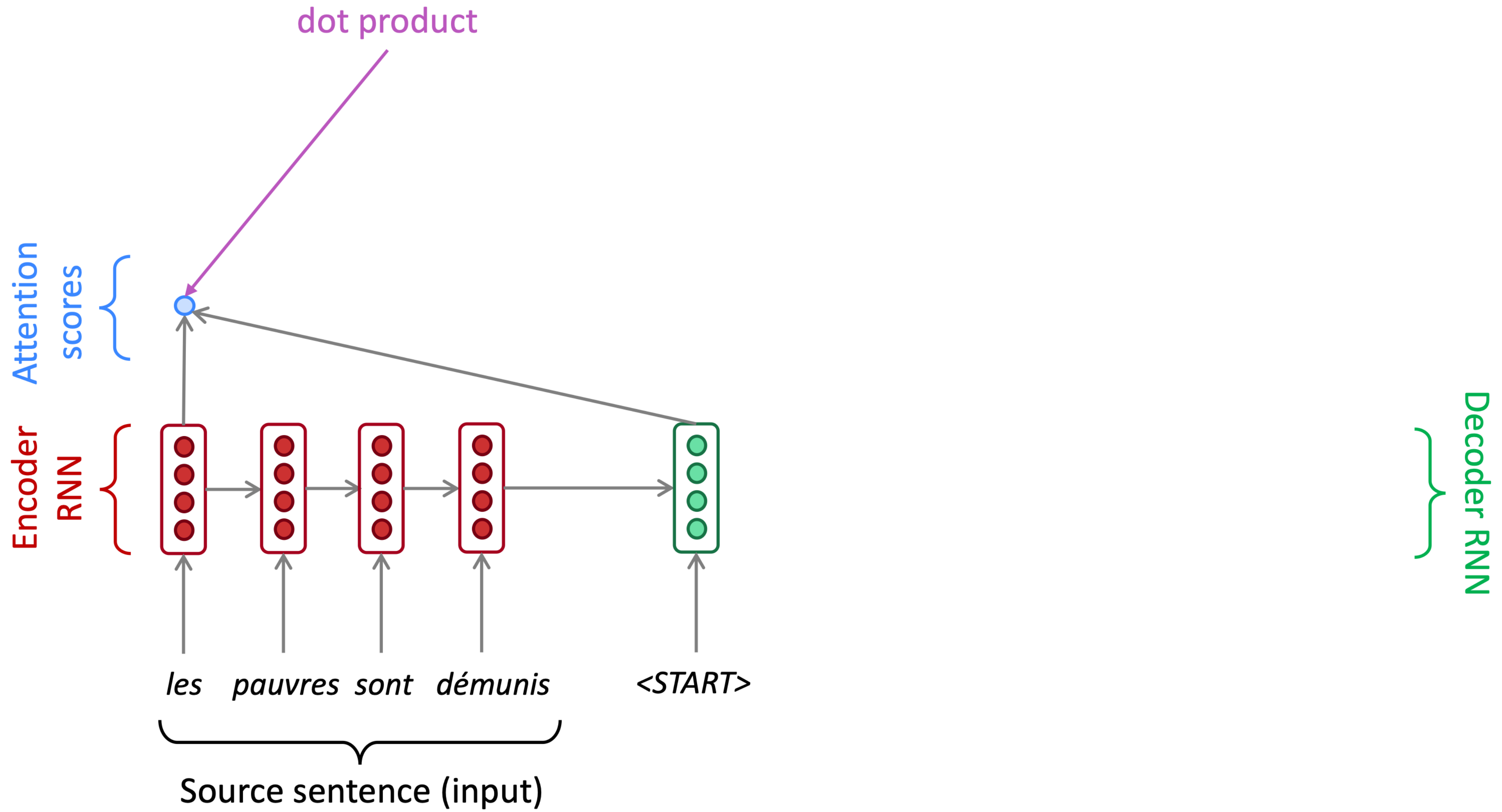
Example: Machine Translation



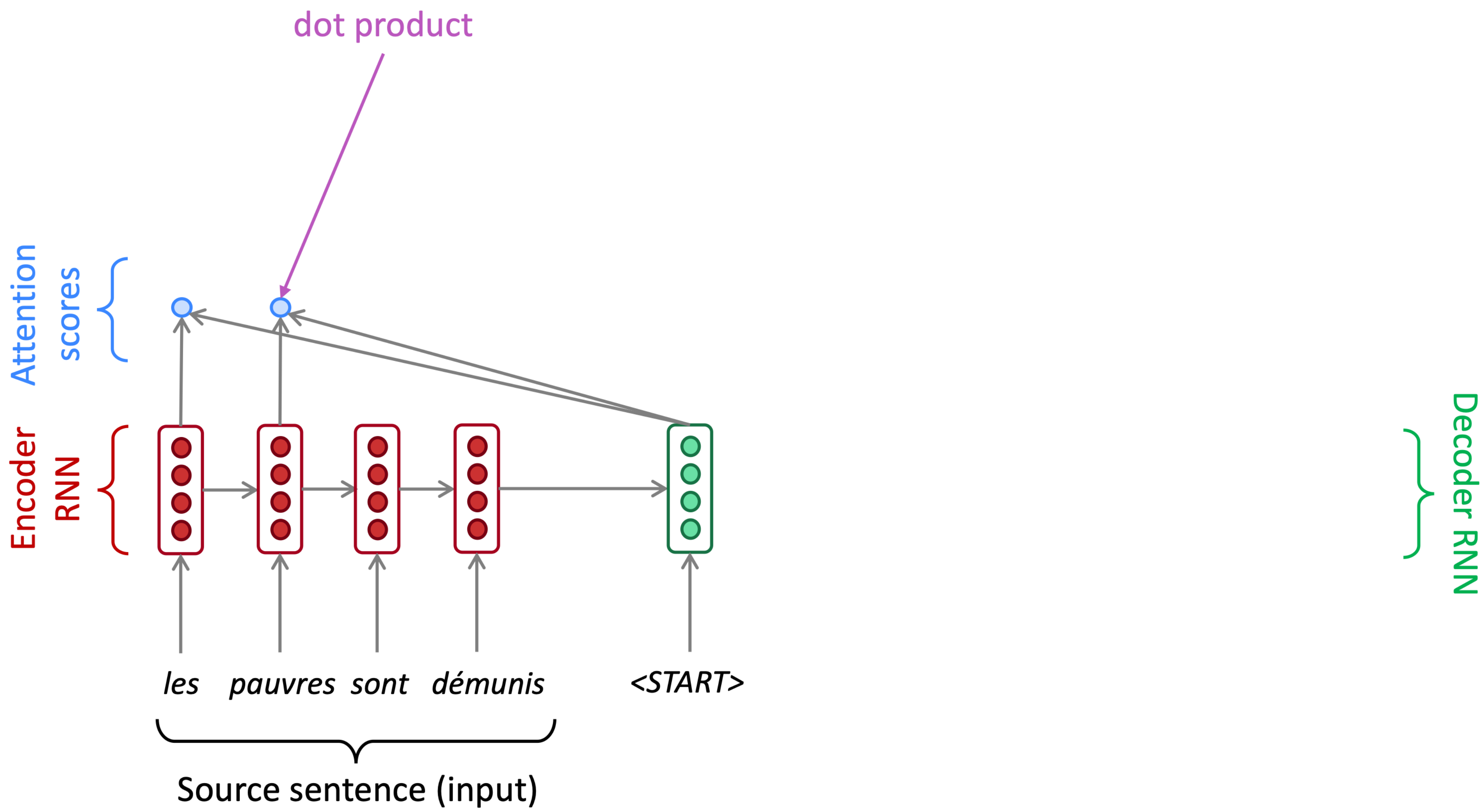
Example: Machine Translation



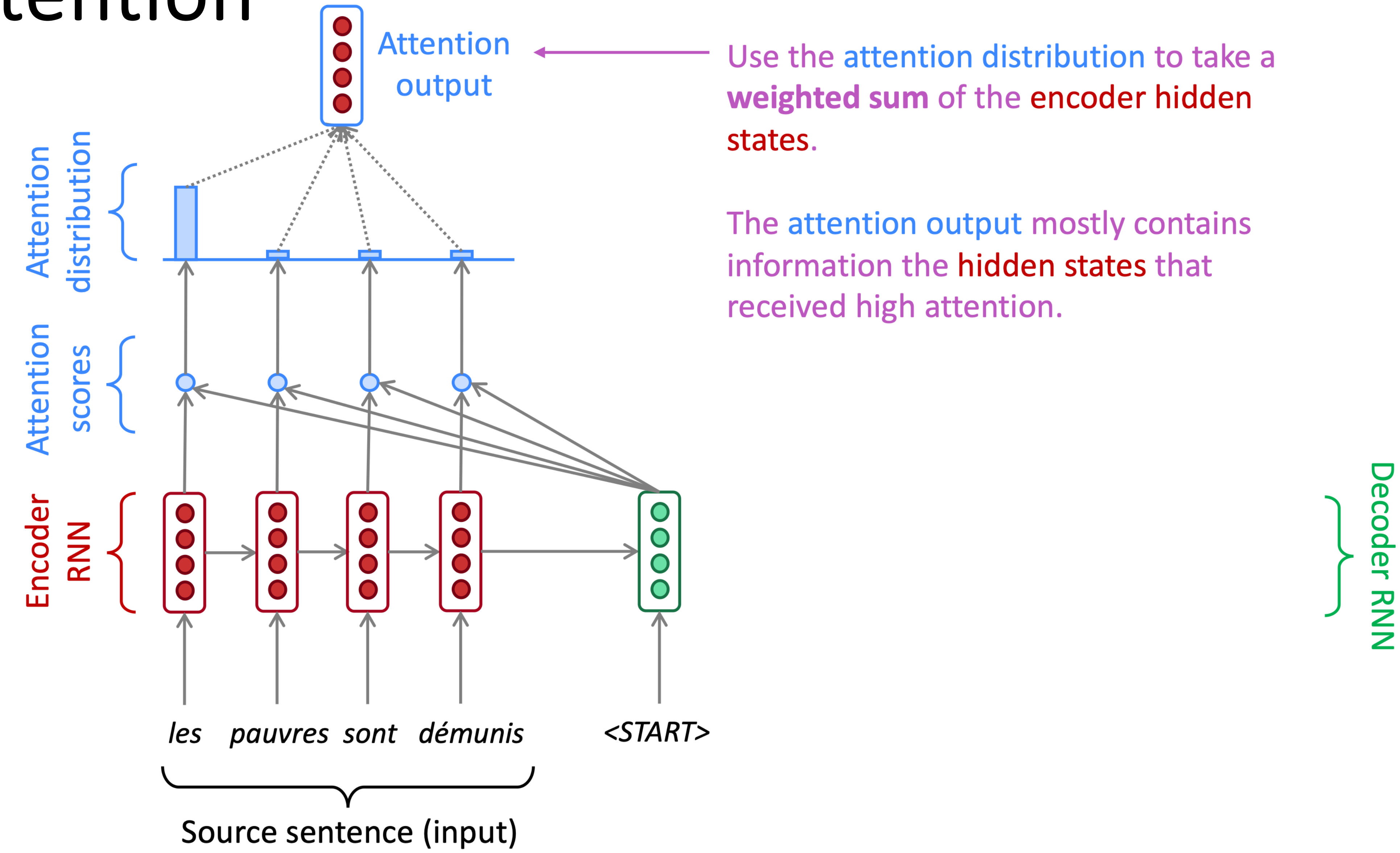
Attention


dot product
Attention scores
Encoder RNN
Decoder RNN
les
pauvres
sont
démunis
<START>
Source sentence (input)

Attention


dot product
Attention scores
Encoder RNN
Decoder RNN
les pauvres sont démunis
<START>
Source sentence (input)

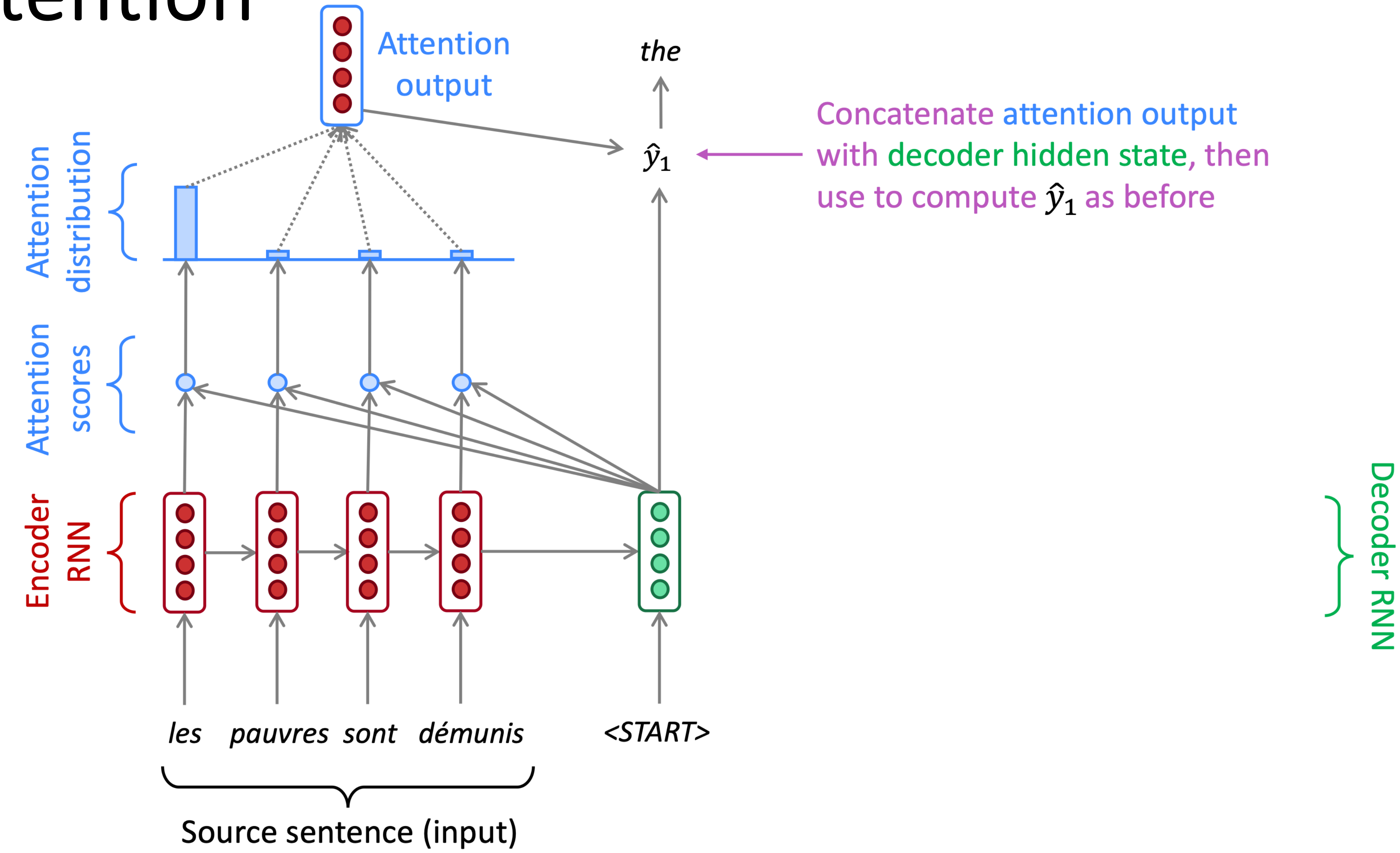
Attention



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

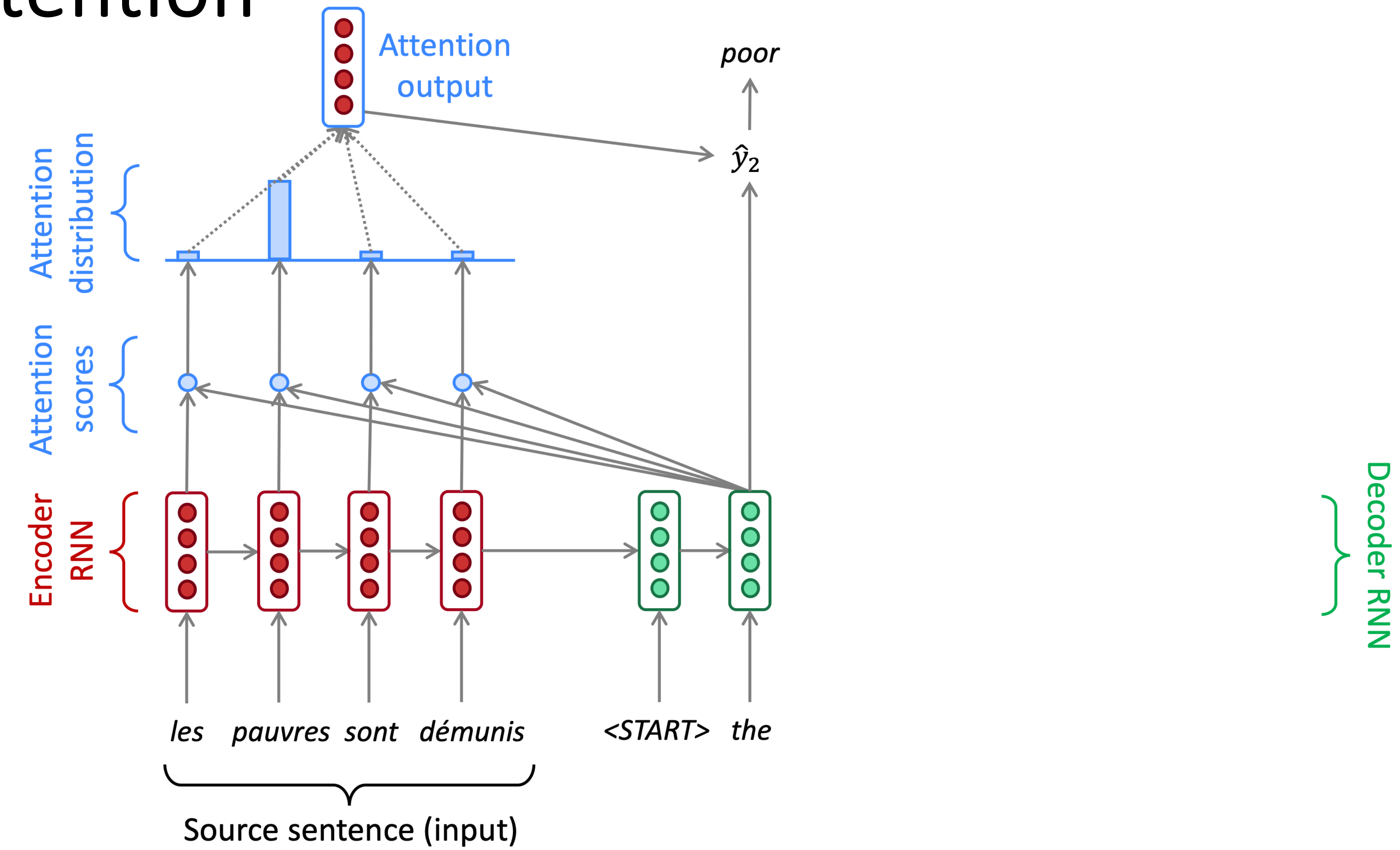
The attention output mostly contains information the hidden states that received high attention.

Attention



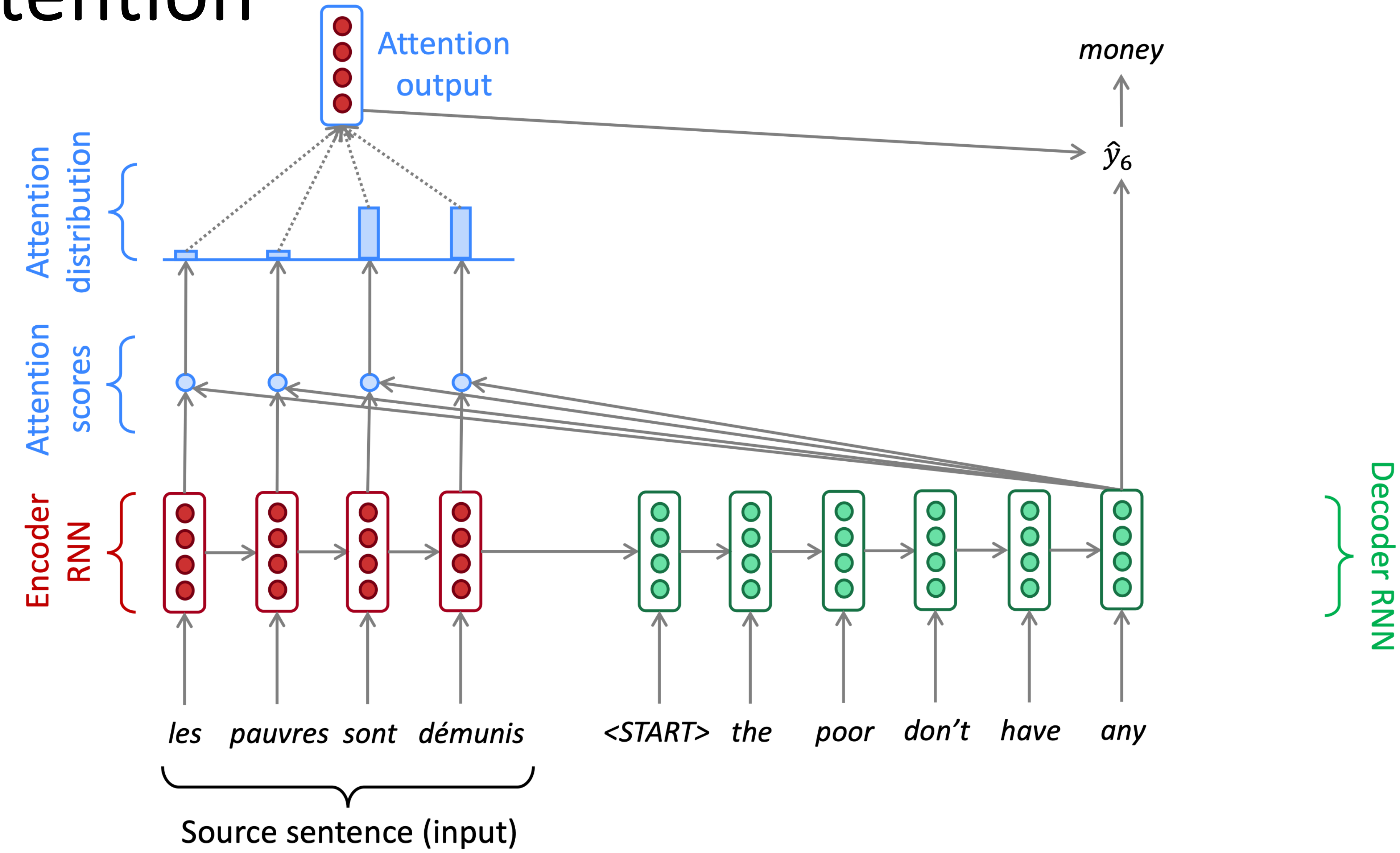
Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before

Attention



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

Attention



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
Source sentence (input)

Attention

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores $\boldsymbol{e}^t$ for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \mathrm{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

Transformers

- Composition of **self-attention layers**

- **Intuition**
  - Want sparse connection structure of CNNs, but with different structure
  - Can we **learn** the connection structure?

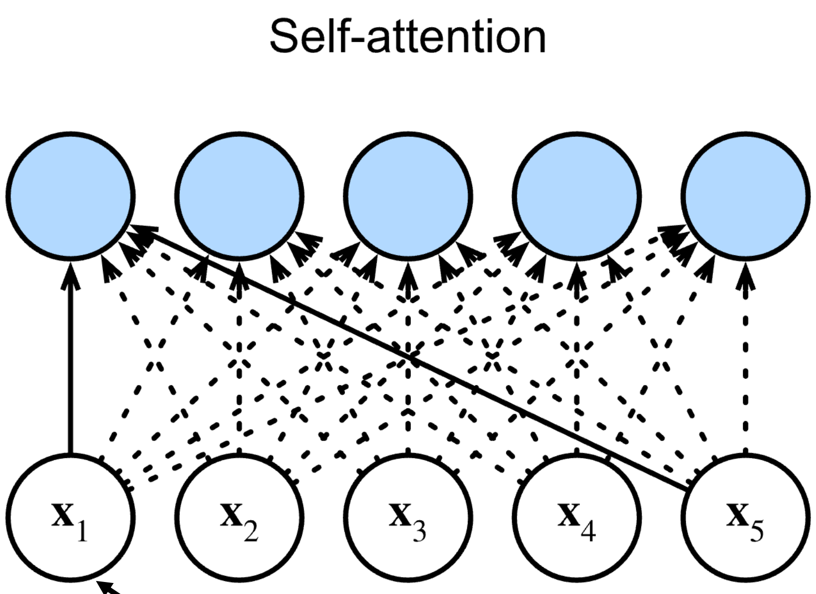
Self-Attention Layer

- **Self-attention layer:**

$$y[t] = \sum_{s=1}^{T} \text{attention}(x[s], x[t]) \cdot f(x[s])$$

  - Input first processed by local layer f
  - All inputs can affect $y[t]$
  - But weighted by $\text{attention}(x[s], x[t])$

- Resembles convolution but connection is learned instead of hardcoded



vector, not a single component!

Figure credit to d2l.ai

Self-Attention Layer

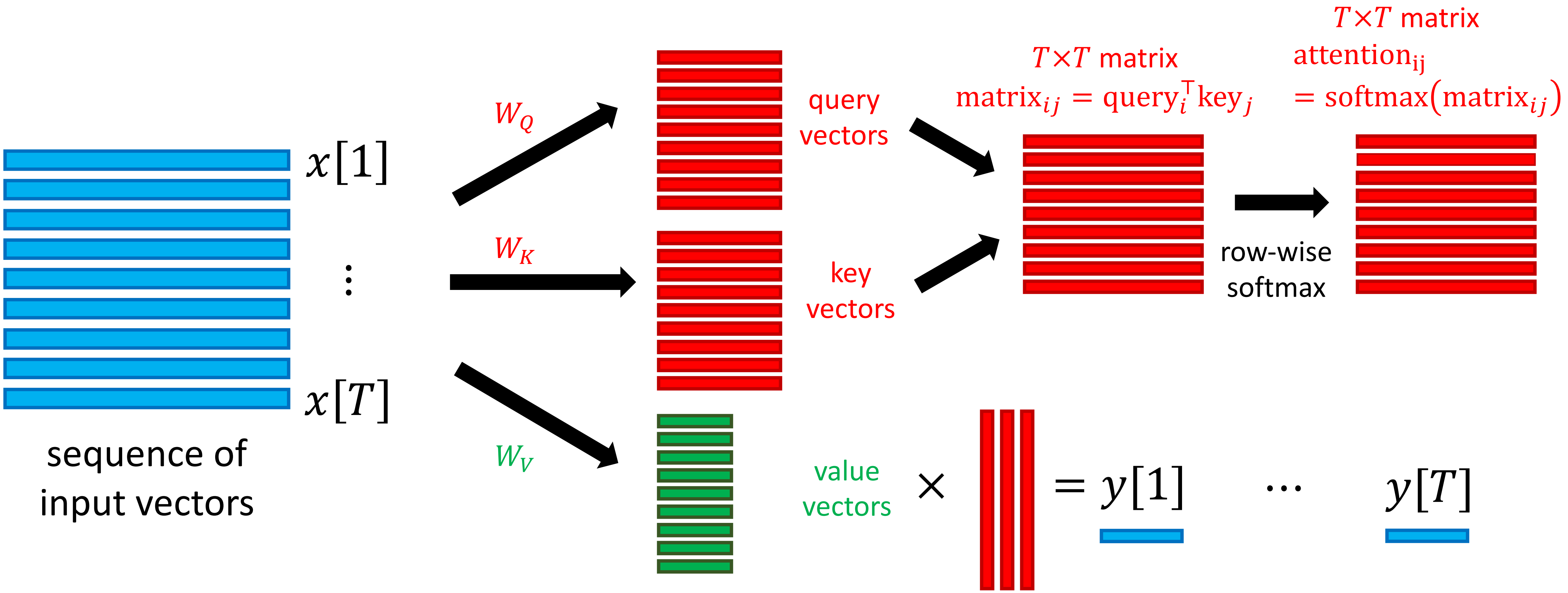
- **Self-attention layer:**

$$y[t] = \sum_{s=1}^{T} \text{softmax}([\text{query}(x[t])^\top \text{key}(x[s])]) \cdot \text{value}(x[s])$$

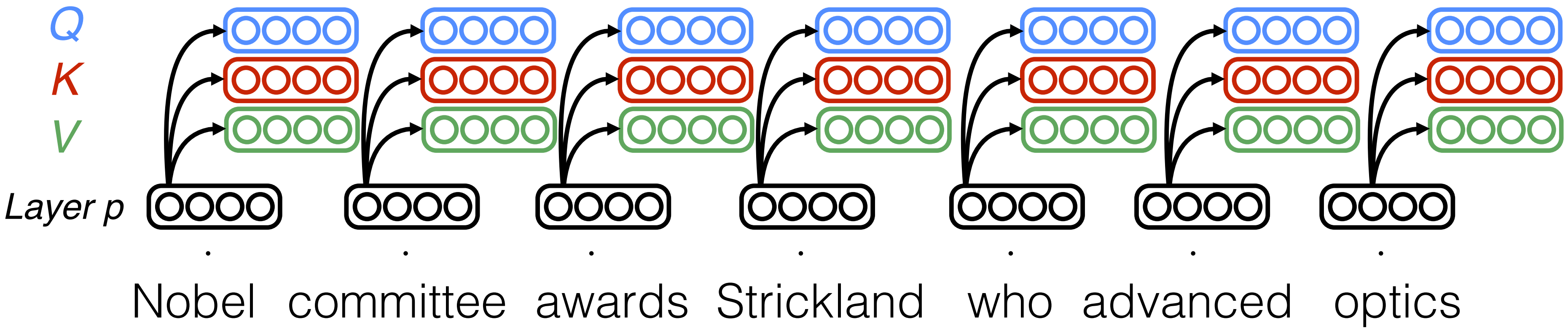
- Here, we have (learnable parameters are W_Q, W_K, and W_V):

$$\text{query}(x[s]) = W_Q x[s]$$
$$\text{key}(x[s]) = W_K x[s]$$
$$\text{value}(x[s]) = W_V x[s]$$

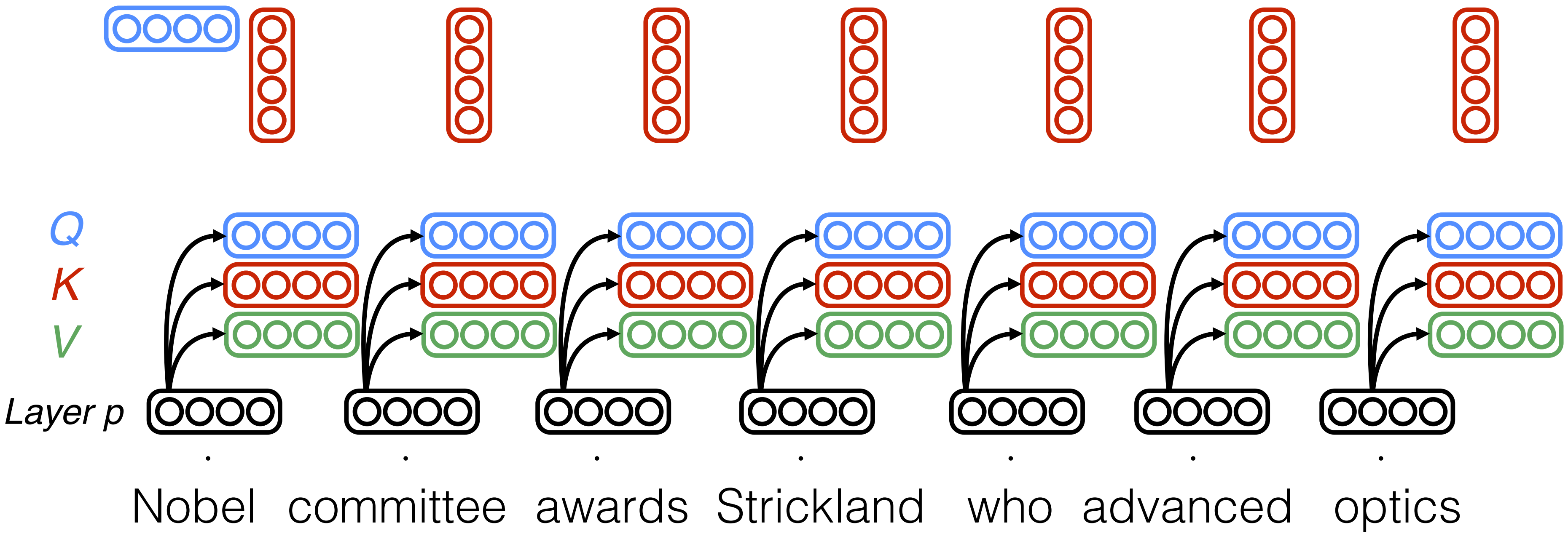
Self-Attention Layer



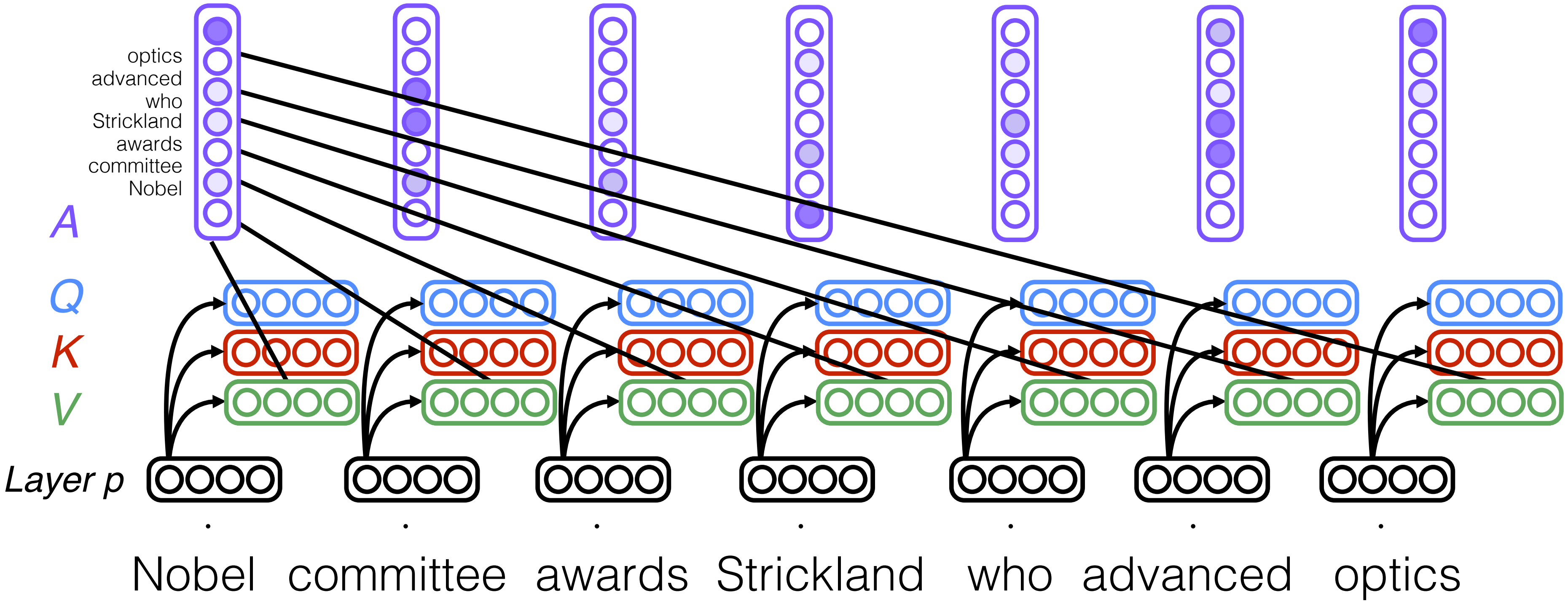
Self-Attention Layer



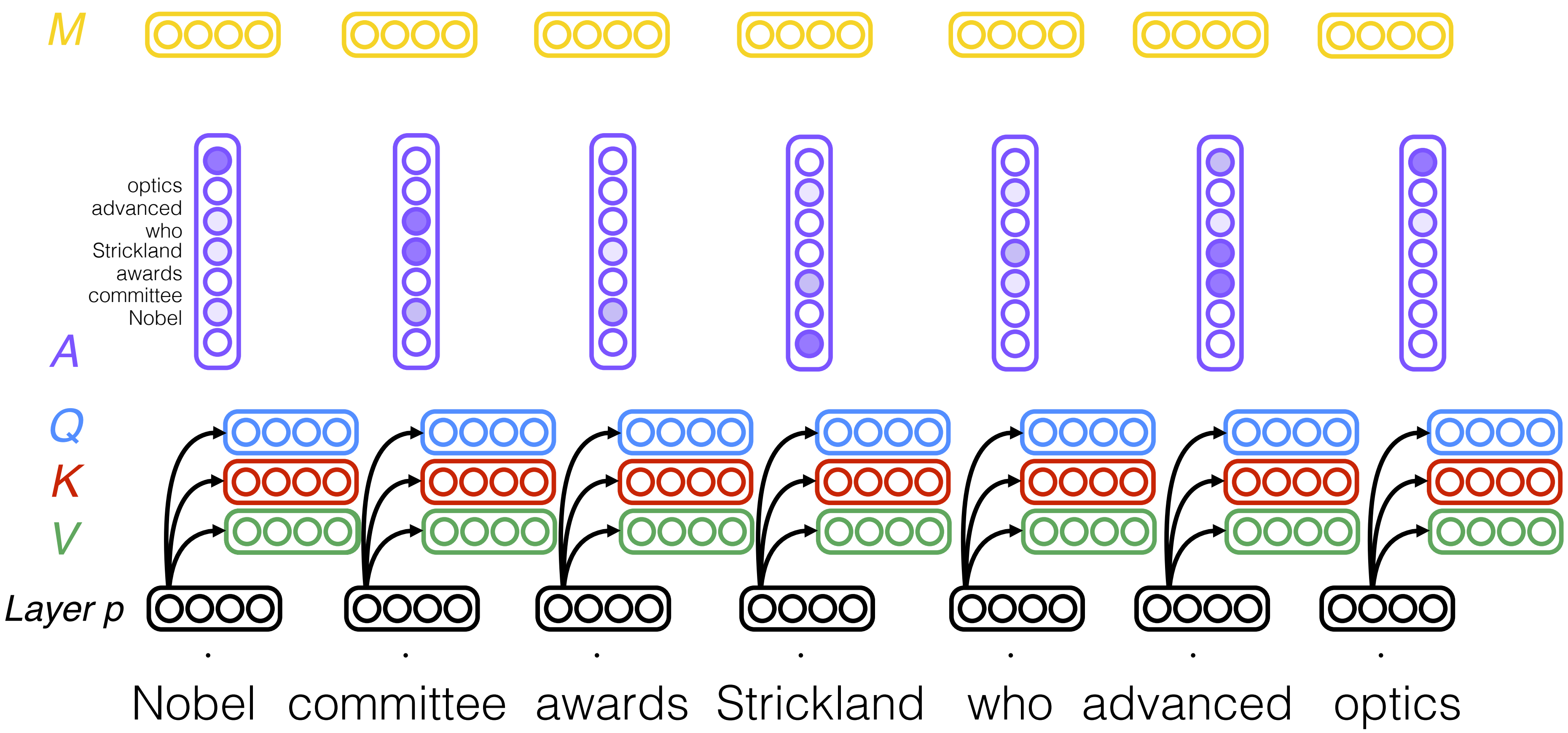
Self-Attention Layer



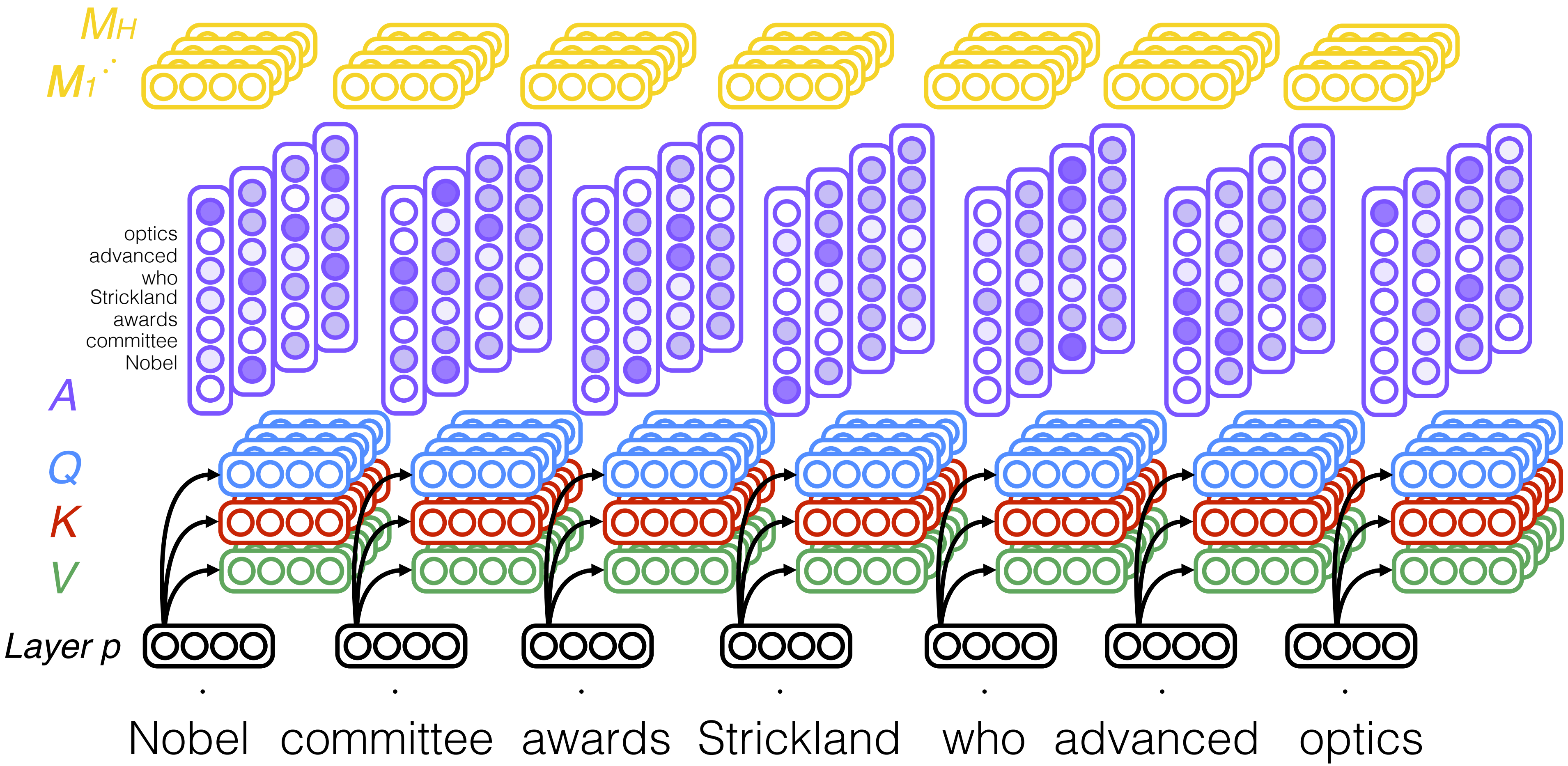
Self-Attention Layer



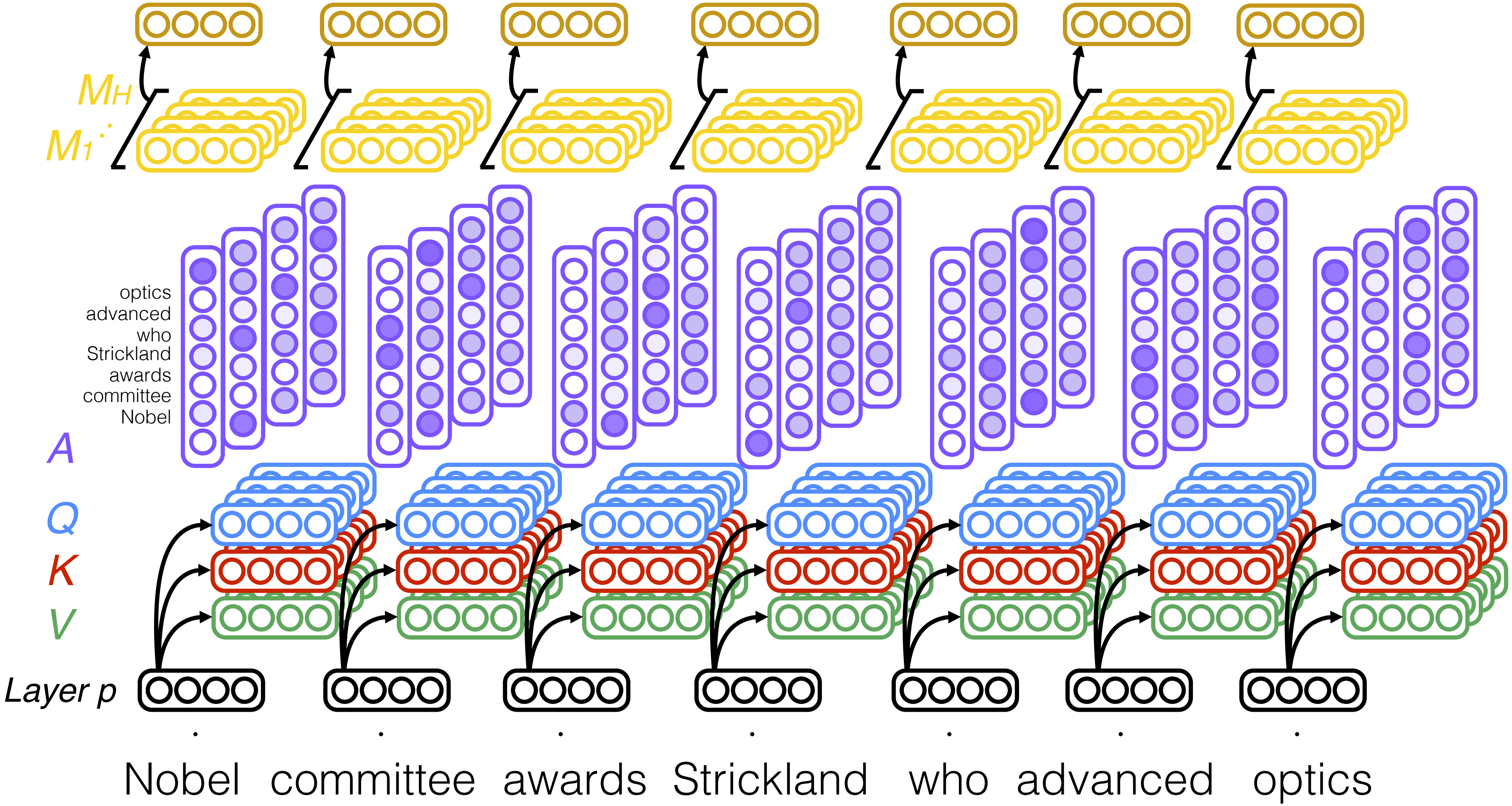
Self-Attention Layer

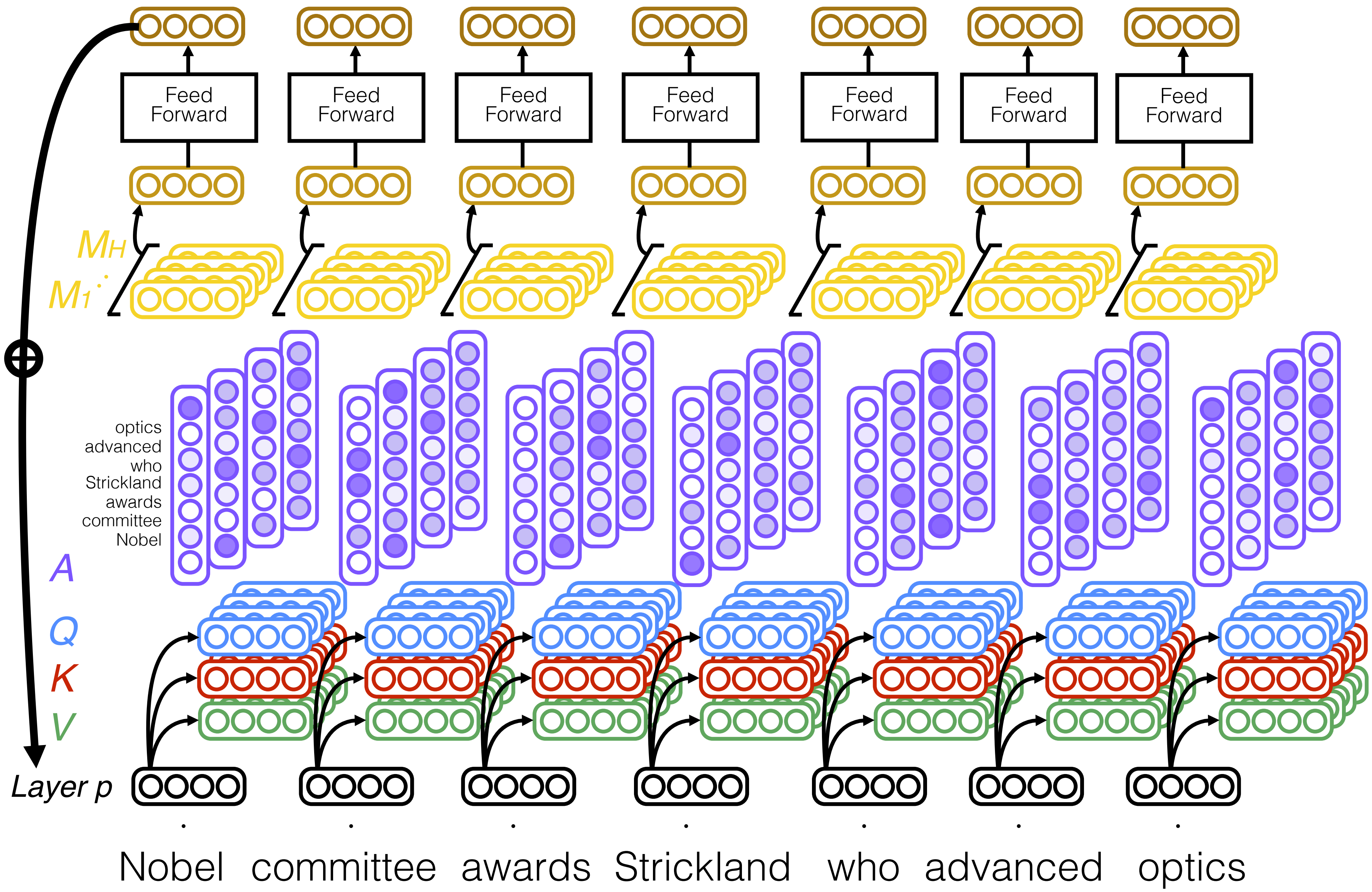


Multi-Head Self-Attention



Multi-Head Self-Attention





Feed Forward
M_H
M_1
optics
advanced
who
Strickland
awards
committee
Nobel
A
Q
K
V
Layer p
Nobel committee awards Strickland who advanced optics

Transformers

- Stack self-attention layers to form a neural network architecture
- **Examples:**
  - **BERT:** Bidirectional transformer similar to ELMo, useful for prediction
  - **GPT:** Unidirectional model suited to text generation
- **Aside:** Self-attention layers subsume convolutional layers
  - Use "positional encodings" as auxiliary input so each input knows its position
  - https://d2l.ai/chapter_attention-mechanisms/self-attention-and-positional-encoding.html#
  - Then, the attention mechanism can learn convolutional connection structure