10-701: Introduction to Machine Learning

Lecture 12 - Recurrent Neural Networks

Hoda Heidari

* Slides adopted from F24 offering of 10701 by Henry Chai.

Convolutional Neural Networks

- Neural networks are frequently applied to inputs with some inherent spatial structure, e.g., images
- Idea: use the first few layers to identify relevant macro-features, e.g., edges
- Insight: for spatially-structured inputs, many useful macro-features are shift or location-invariant, e.g., an edge in the upper left corner of a picture looks like an edge in the center
- Strategy: learn a *filter* for macro-feature detection in a small window and apply it over the entire image

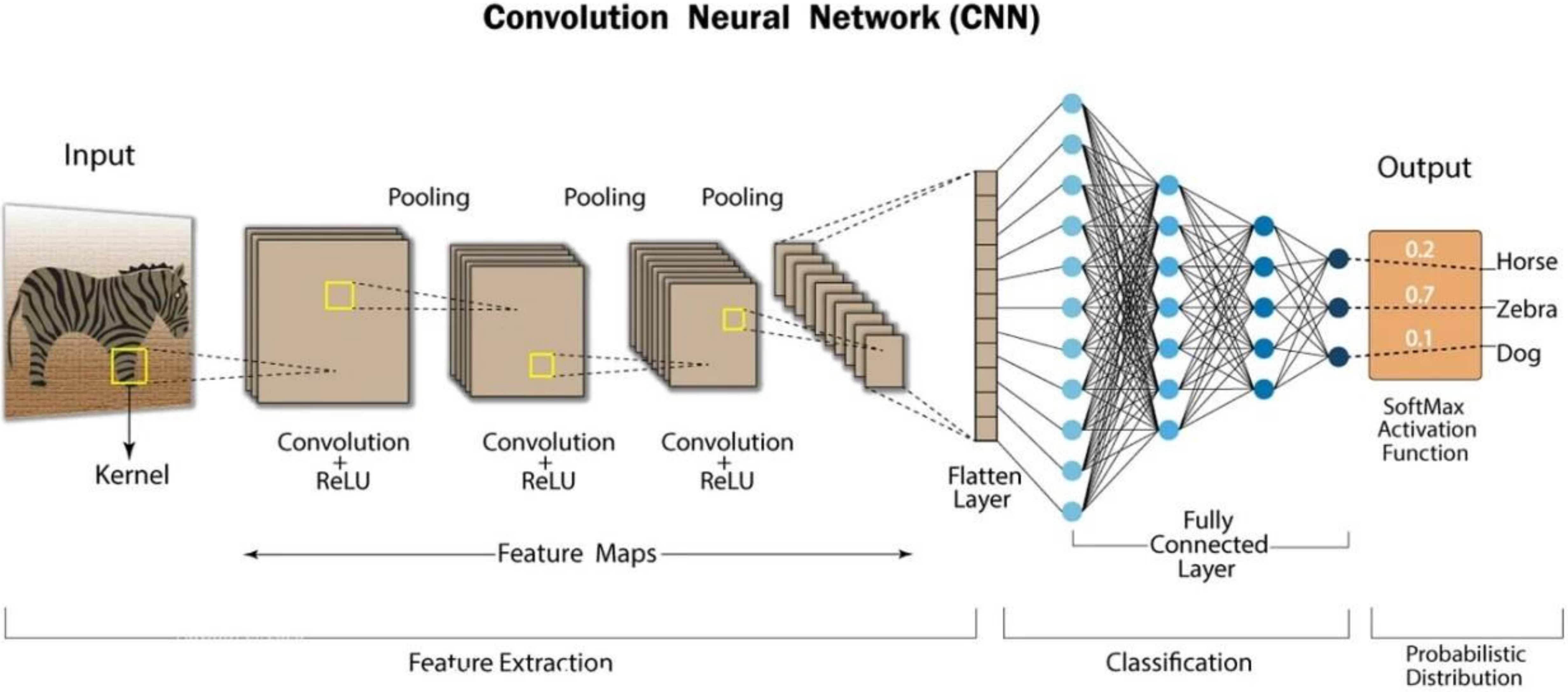


Image from: https://www.linkedin.com/pulse/what-convolutional-neural-network-cnn-deep-learning-nafiz-shahriar/

Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A **filter/kernel** is just a small matrix that is convolved with same-sized sections of the image matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 4 & 4 & 2 & 0 \\ 0 & 1 & 3 & 3 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A **filter/kernel** is just a small matrix that is convolved with same-sized sections of the image matrix

$$
\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 4 & 4 & 2 & 0 \\ 0 & 1 & 3 & 3 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}
$$

$$
(0 * 0) + (0 * 1) + (0 * 0) + (0 * 1) + (1 * -4) \\ + (2 * 1) + (0 * 0) + (2 * 1) + (4 * 0) = 0
$$

Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A **filter/kernel** is just a small matrix that is convolved with same-sized sections of the image matrix

$$\begin{array}{|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 2 & 2 & 1 & 0 \\ \hline 0 & 2 & 4 & 4 & 2 & 0 \\ \hline 0 & 1 & 3 & 3 & 1 & 0 \\ \hline 0 & 1 & 2 & 3 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & -4 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline 0 & -1 & & \\ \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array}$$

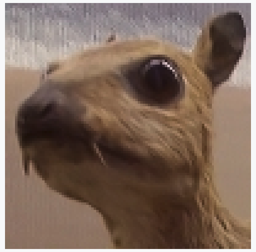
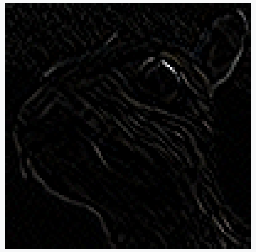
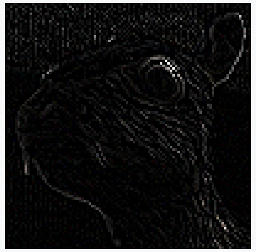
$$(0 * 0) + (0 * 1) + (0 * 0) + (1 * 1) + (2 * -4) + (2 * 1) + (2 * 0) + (4 * 1) + (4 * 0) = -1$$

Convolutional Filters

- Images can be represented as matrices: each element corresponds to a pixel and its value is the intensity
- A **filter/kernel** is just a small matrix that is convolved with same-sized sections of the image matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 4 & 4 & 2 & 0 \\ 0 & 1 & 3 & 3 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & -1 & 0 \\ -2 & -5 & -5 & -2 \\ 2 & -2 & -1 & 3 \\ -1 & 0 & -5 & 0 \end{bmatrix}$$

Convolutional Filters

Operation	Kernel ω	Image result g(x,y)
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	1 $\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	2 $\begin{bmatrix} \cancel{0} & 1 & \cancel{0} \\ 1 & -4 & 1 \\ \cancel{0} & 1 & \cancel{0} \end{bmatrix}$	
	3 $\begin{bmatrix} -1 & -1 & -1 \\ -1 & \boxed{8} & -1 \\ -1 & -1 & -1 \end{bmatrix}$	

Source: https://en.wikipedia.org/wiki/Kernel_(image_processing)

More Filters

Operation	Kernel ω	Image result g(x,y)
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

Source: https://en.wikipedia.org/wiki/Kernel_(image_processing)

Convolutional Filters

- Convolutions can be represented by a feed forward neural network where:
    1. Nodes in the input layer are only connected to some nodes in the next layer but not all nodes.
    2. Many of the weights have the same value.

0	0	0	0	0	0
0	1	2	2	1	0
0	2	4	4	2	0
0	1	3	3	1	0
0	1	2	3	1	0
0	0	1	1	0	0

0	-1	-1	0
-2	-5	-5	-2
2	-2	-1	3
-1	0	-5	0

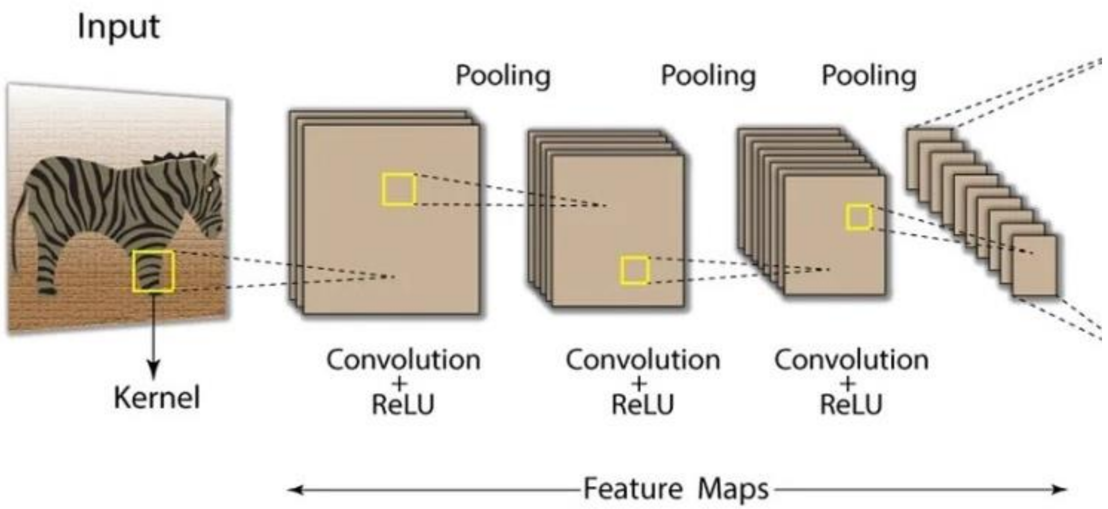
- Many fewer weights than a fully connected layer!
- **Convolution weights are learned using gradient descent/ backpropagation, not prespecified**

Convolutional Filters: Padding

- What if relevant features exist at the border of our image?
- Add zeros around the image to allow for the filter to be applied "everywhere" e.g. a *padding* of 1 with a 3x3 filter preserves image size and allows every pixel to be the center

$$\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&1&2&2&1&0&0\\ 0&0&2&4&4&2&0&0\\ 0&0&1&3&3&1&0&0\\ 0&0&1&2&3&1&0&0\\ 0&0&0&1&1&0&0&0\\ 0&0&0&0&0&0&0&0 \end{bmatrix} * \begin{bmatrix} 0&1&0\\ 1&-4&1\\ 0&1&0 \end{bmatrix} = \begin{bmatrix} 0&1&2&2&1&0\\ 1&0&-1&-1&0&1\\ 2&-2&-5&-5&-2&2\\ 1&2&-2&-1&3&1\\ 1&-1&0&-5&0&1\\ 0&2&-1&0&2&0 \end{bmatrix}$$

Downsampling



- **Idea:** reduce the spatial size of the feature maps to
  - cut down the number of parameters and computations in later layers
  - reduce the risk of overfitting
  - make the model less sensitive to small shifts in input

Downsampling: Stride

- Only apply the convolution to some subset of the image

  e.g., every other column and row = a *stride* of 2

$$\begin{array}{|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 2 & 2 & 1 & 0 \\ \hline 0 & 2 & 4 & 4 & 2 & 0 \\ \hline 0 & 1 & 3 & 3 & 1 & 0 \\ \hline 0 & 1 & 2 & 3 & 1 & 0 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 1 & -2 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline -2 & & \\ \hline & & \\ \hline & & \\ \hline \end{array}$$

Downsampling: Stride

- Only apply the convolution to some subset of the image

  e.g., every other column and row = a *stride* of 2

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 4 & 4 & 2 & 0 \\ 0 & 1 & 3 & 3 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -2 & \\ & & \\ & & \end{bmatrix}$$

Downsampling: Stride

- Only apply the convolution to some subset of the image

  e.g., every other column and row = a *stride* of 2

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 4 & 4 & 2 & 0 \\ 0 & 1 & 3 & 3 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -2 & 1 \\ & & \\ & & \end{bmatrix}$$

Downsampling: Stride

- Only apply the convolution to some subset of the image

  e.g., every other column and row = a *stride* of 2

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 4 & 4 & 2 & 0 \\ 0 & 1 & 3 & 3 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -2 & 1 \\ 0 & & \\ & & \end{bmatrix}$$

Downsampling: Stride

- Only apply the convolution to some subset of the image

  e.g., every other column and row = a *stride* of 2

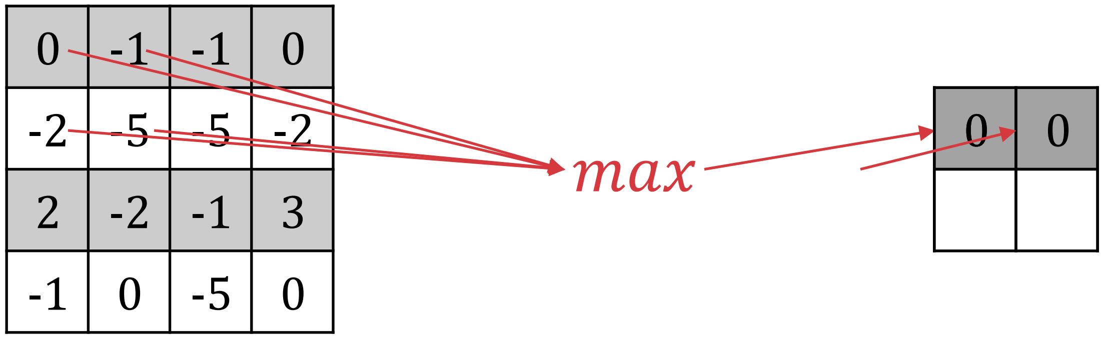
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 4 & 4 & 2 & 0 \\ 0 & 1 & 3 & 3 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -2 & 1 \\ 0 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

- Reduces the dimensionality of the input to subsequent layers and thus, the number of weights to be learned

- Many relevant macro-features will tend to span large portions of the image, so taking strides with the convolution tends not to miss out on too much

Downsampling: Pooling

- Combine multiple adjacent nodes into a single node

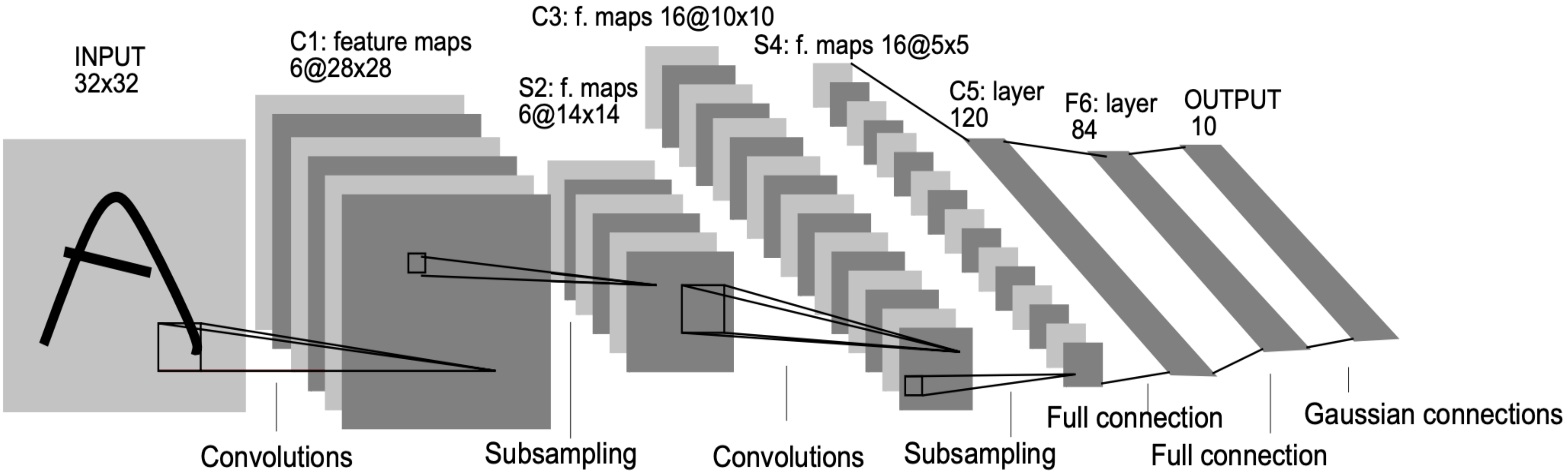


Downsampling: Pooling

- Combine multiple adjacent nodes into a single node

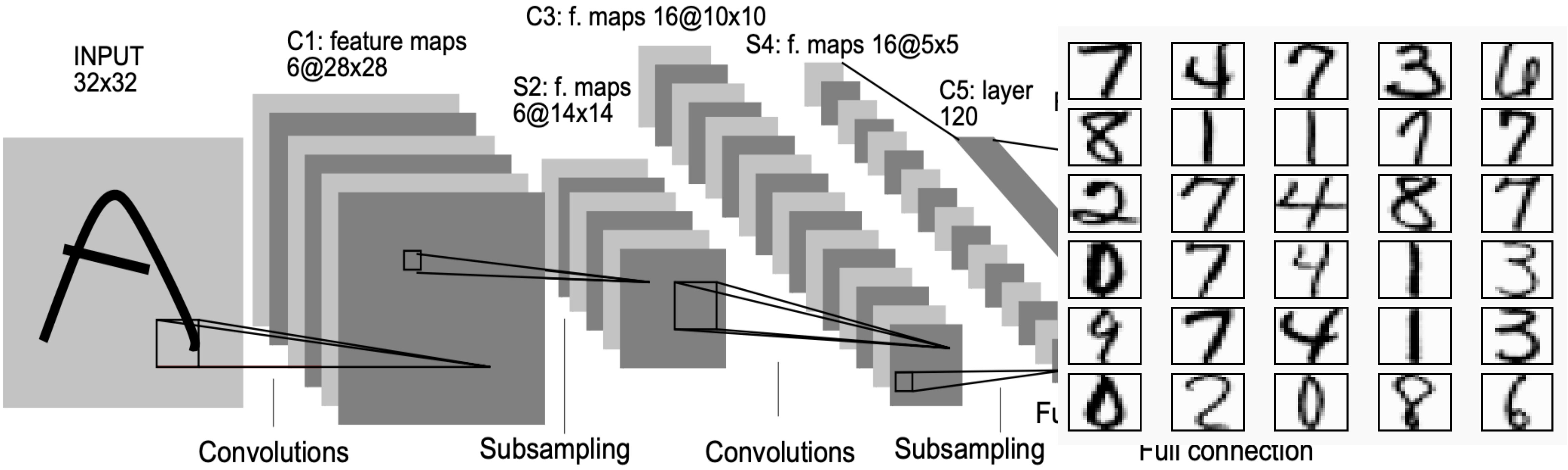
$$\begin{bmatrix} 0 & -1 & -1 & 0 \\ -2 & -5 & -5 & -2 \\ 2 & -2 & -1 & 3 \\ -1 & 0 & -5 & 0 \end{bmatrix} \xrightarrow{\textit{max pooling}} \begin{bmatrix} 0 & 0 \\ 2 & 3 \end{bmatrix}$$

- **Max Pooling** keeps the strongest activation in each region, focusing on the most prominent features.
- **Average Pooling** computes the average of the region, providing a smoother, more generalized representation.



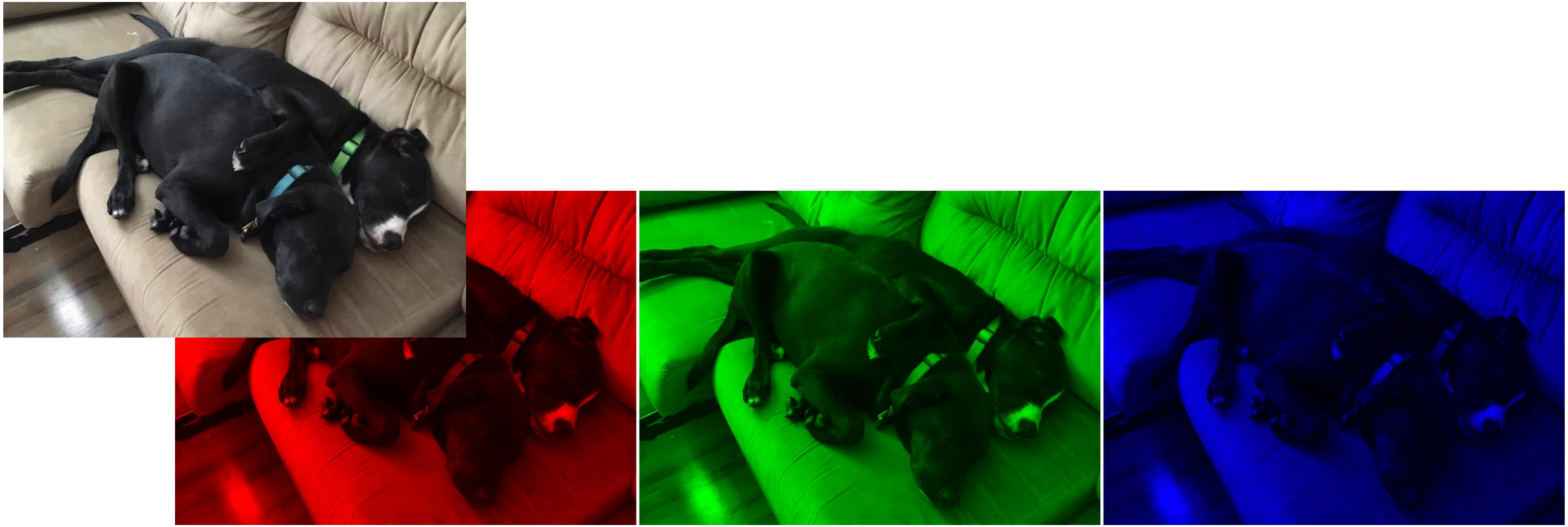
LeNet (LeCun et al., 1998)

Source: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf

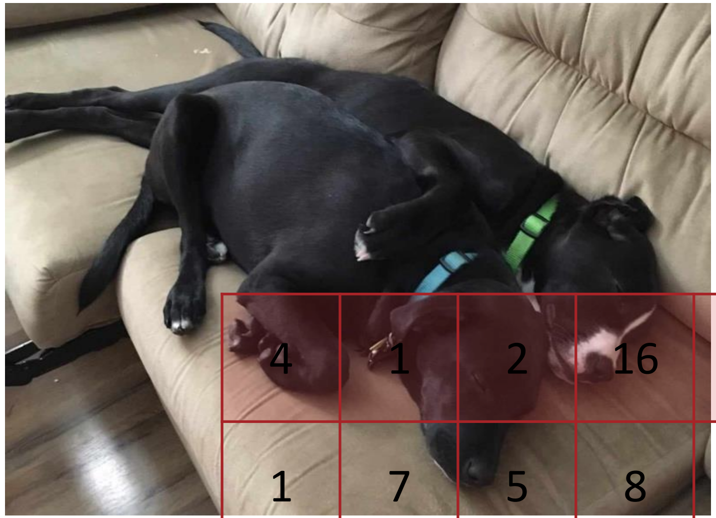


- One of the earliest, most famous deep learning models – achieved remarkable performance at handwritten digit recognition (< 1% test error rate on MNIST)
- Used sigmoid (or logistic) activation functions between layers and mean-pooling, both of which are pretty uncommon in modern architectures

Source: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf



Colored Images and Channels



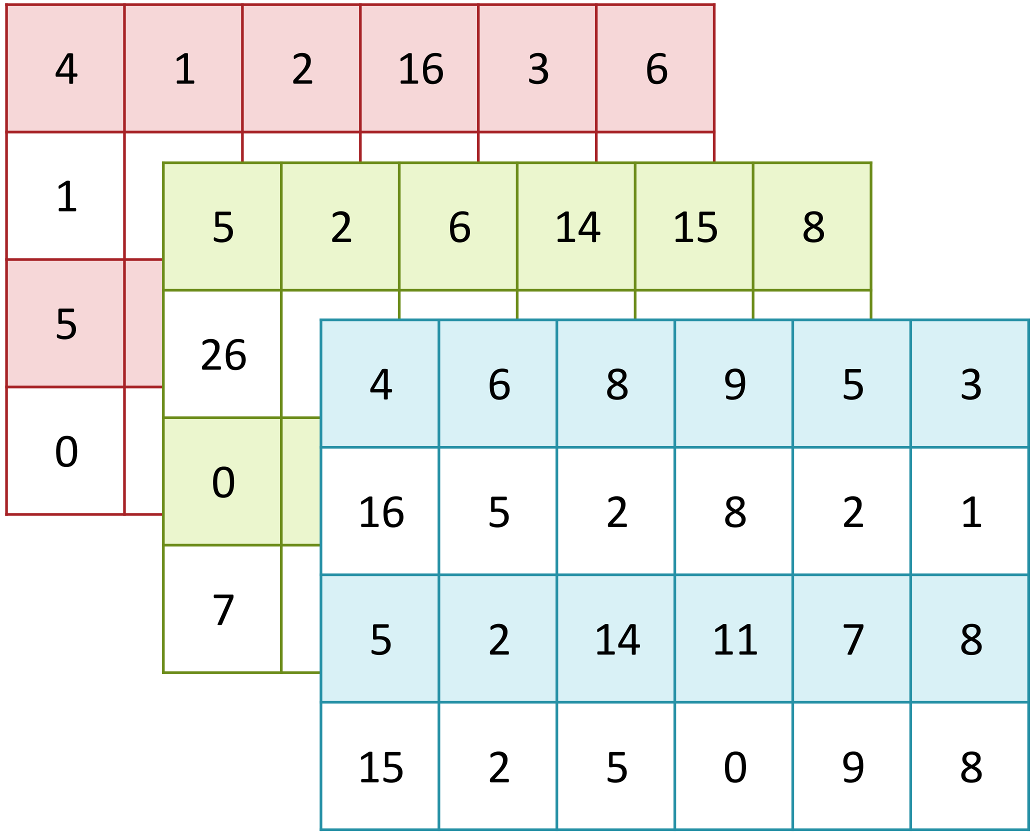
4	1	2	16	3	6
1	7	5	8	19	27
5	2	5	12	17	8
0	4	9	9	6	11

5	2	6	14	15	8
26	3	6	8	4	9
0	15	24	6	1	8
7	4	9	5	24	17

4	6	8	9	5	3
16	5	2	8	2	1
5	2	14	11	7	8
15	2	5	0	9	8

- An image can be represented as the sum of red, green and blue pixel intensities
- Each color corresponds to a *channel*

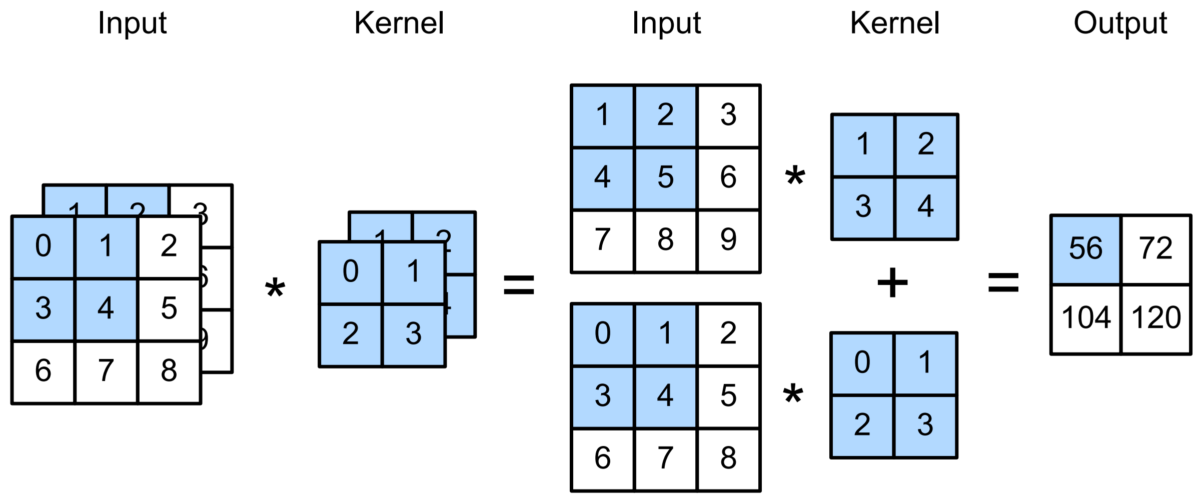
Example: $3 \times 4 \times 6$ tensor



- An image can be represented as a *tensor* or multidimensional array

Convolutions on Multiple Input Channels

- Given multiple input channels, we can specify a filter for each one and sum the results to get a 2-D output tensor

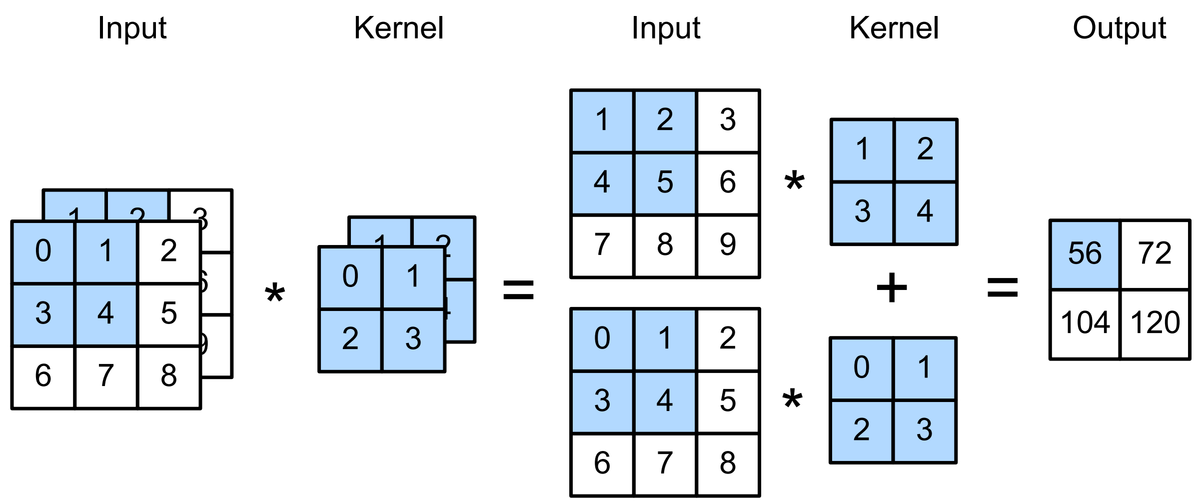


- For c channels and $h \times w$-sized filters, we have $chw + c$ learnable parameters (each filter has a bias term)

Source: http://preview.d2l.ai/d2l-en/master/chapter_convolutional-neural-networks/channels.html

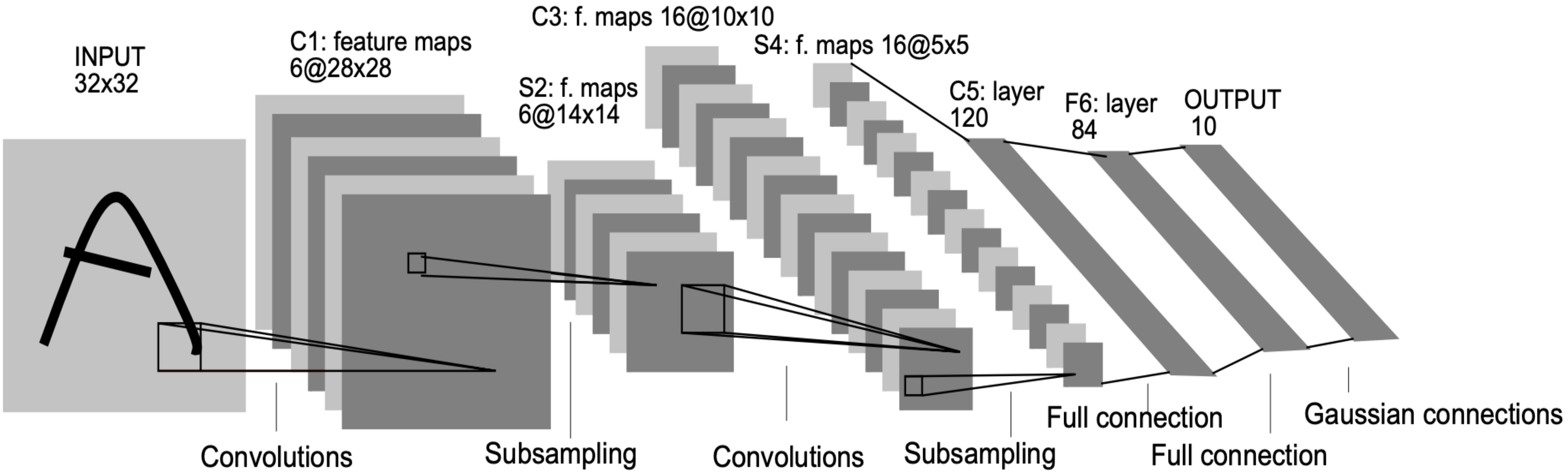
Convolutions on Multiple Input Channels

- Given multiple input channels, we can specify a filter for each one and sum the results to get a 2-D output tensor



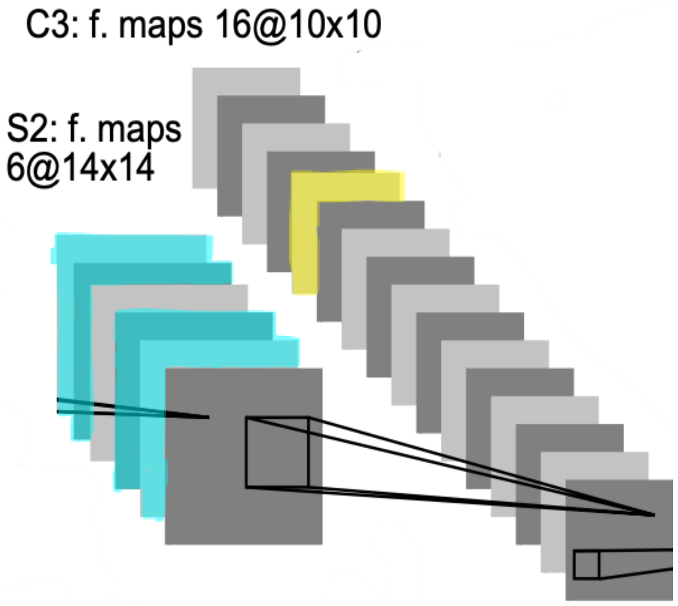
- Questions:
  1. Why might we want a different filter for each input?
  2. Why do we combine them together into a single output channel?

Source: http://preview.d2l.ai/d2l-en/master/chapter_convolutional-neural-networks/channels.html



- Channels in hidden layers correspond to different macro-features, which we might want to manipulate differently → one filter per channel

Source: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf



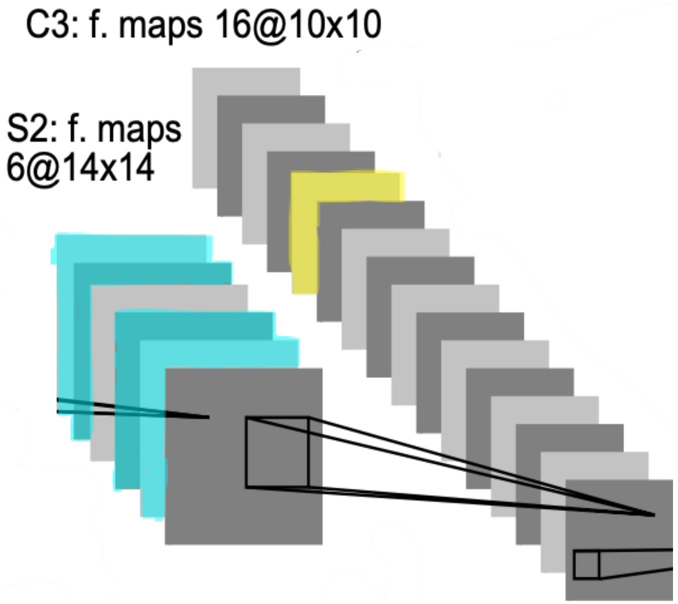
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

- We can combine these macro-features into a new, interesting, "higher-level" feature
  - But we don't always need to combine all of them!
  - Different combinations → multiple output channels
  - Common pattern: more output channels and smaller outputs in deeper layers

Source: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

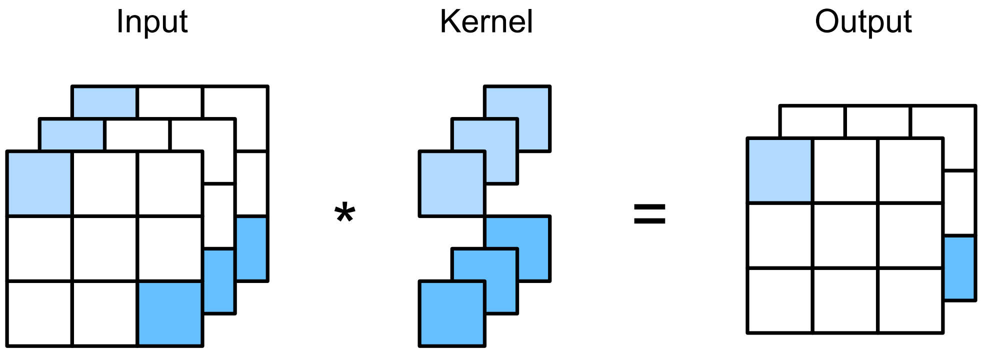
Each column indicates which feature map in S2 are combined by the units in a particular feature map of C3.

Okay, but what if our layers become too big in the channel dimension?

Source: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf

Downsampling: 1×1 Convolutions

- Convolutional layers can be represented as 4-D tensors of size $c_o \times c_i \times h \times w$ where c_o is the number of output channels and c_i is the number of input channels
- Layers of size $c_o \times c_i \times 1 \times 1$ can condense many input channels into fewer output channels (if $c_o < c_i$)



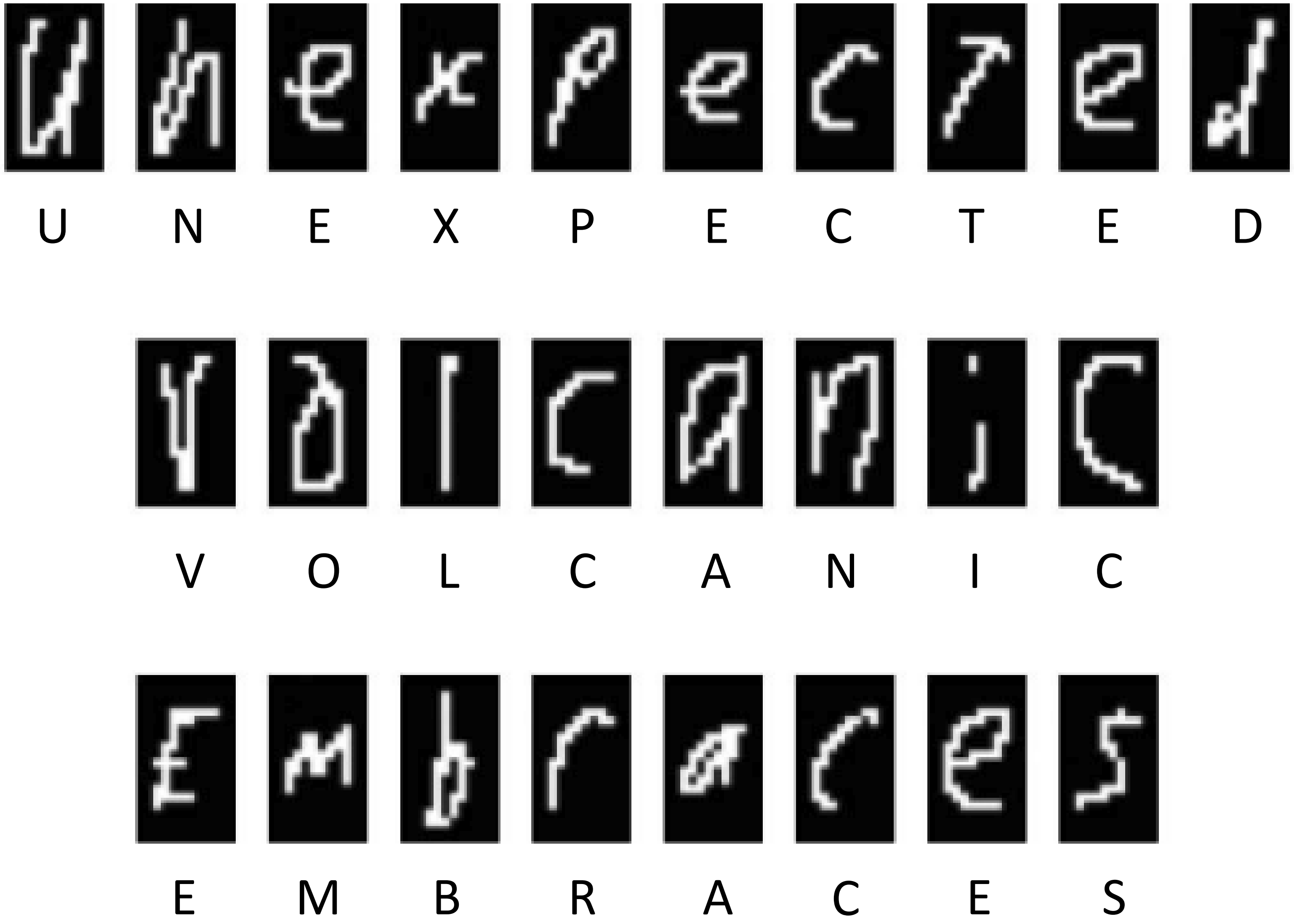
- Practical note: 1×1 convolutions are typically followed by a nonlinear activation function; otherwise, they could simply be folded into other convolutions

Source: http://preview.d2l.ai/d2l-en/master/chapter_convolutional-neural-networks/channels.html

Key Takeaways

- Convolutional neural networks use convolutions to learn macro-features.
  - Can be thought of as slight modifications to the fully-connected feed-forward neural network.
  - Can still be learned using SGD.
  - Padding is used to preserve spatial dimensions.
  - Pooling, stride and 1×1 convolutions are used to downsample intermediate representations.

Example: Handwriting Recognition

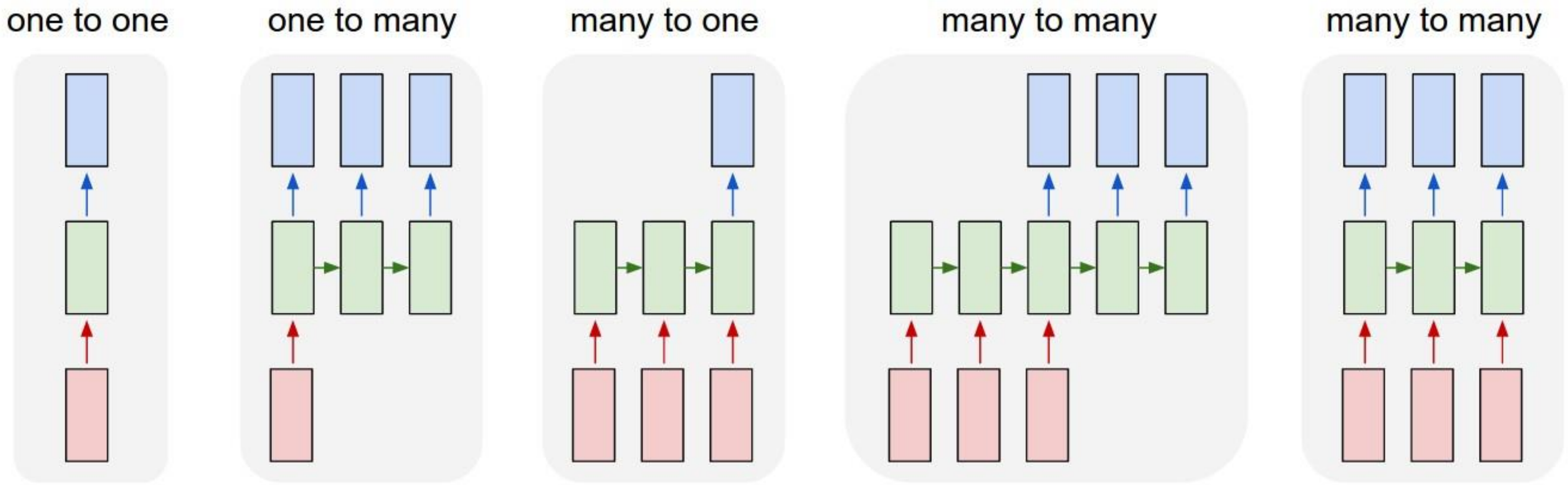


Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

$$\boldsymbol{y}^{(i)} = \left[\boldsymbol{y}_1^{(i)}, \boldsymbol{y}_2^{(i)}, \dots, \boldsymbol{y}_{T_i}^{(i)}\right]$$

Outputs, $y^{(i)}$

Inputs, $\boldsymbol{x}^{(i)}$



$$\boldsymbol{x}^{(i)} = \left[\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \dots, \boldsymbol{x}_{T_i}^{(i)}\right]$$

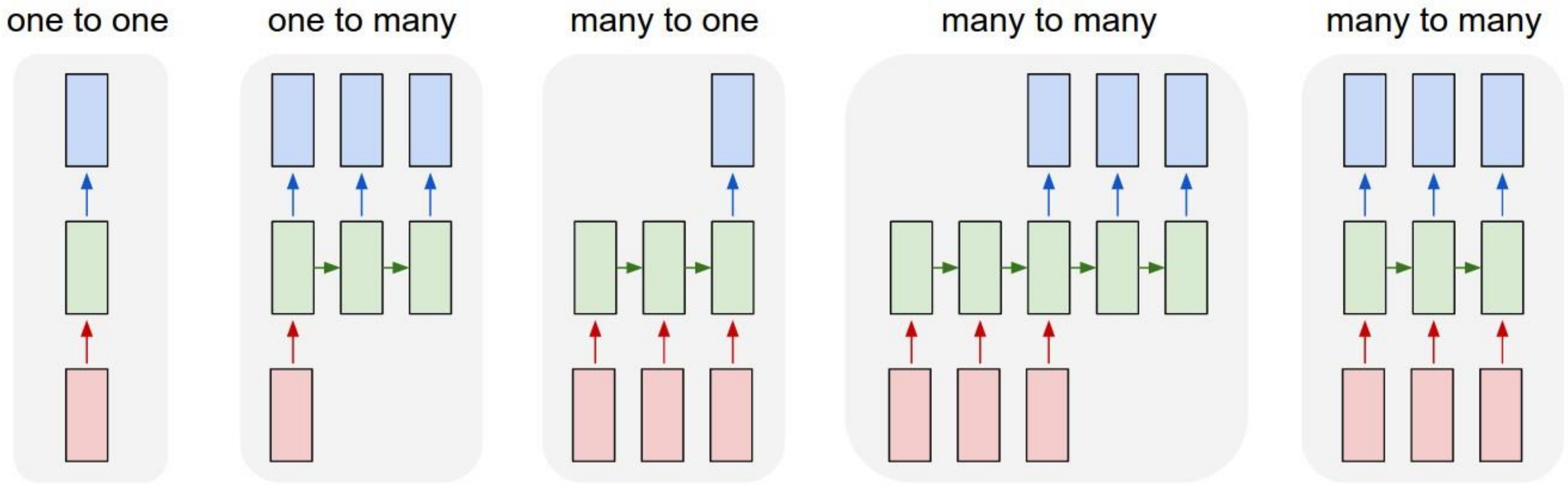
Sequential Data

Source: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

$$\boldsymbol{y}^{(i)} = \left[\boldsymbol{y}_1^{(i)}, \boldsymbol{y}_2^{(i)}, \ldots, \boldsymbol{y}_{T_i}^{(i)}\right]$$

Outputs, $y^{(i)}$

Inputs, $\boldsymbol{x}^{(i)}$



$$\boldsymbol{x}^{(i)} = \left[\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right]$$

Poll:

formulate a hand-written digit recognition task

Source: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Recurrent Neural Networks

- Neural networks are frequently applied to inputs with some inherent **temporal or sequential** structure (e.g., text or video) of **variable length**
- Idea: use the information from previous parts of the input to inform subsequent predictions
- Insight: the hidden layers learn a useful representation (relative to the task)
- Approach: incorporate the output from earlier hidden layers into later ones.

Recurrent Neural Networks

- Data points consists of (input **sequence**, label **sequence**) pairs, potentially of **varying lengths**

$$\mathcal{D} = \left\{\left(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\right)\right\}_{n=1}^{N}$$

$$\boldsymbol{x}^{(n)} = \left[\boldsymbol{x}_1^{(n)}, \dots, \boldsymbol{x}_{T_n}^{(n)}\right]$$

$$\boldsymbol{y}^{(n)} = \left[\boldsymbol{y}_1^{(n)}, \dots, \boldsymbol{y}_{T_n}^{(n)}\right]$$

Recurrent Neural Networks

- RNNs process inputs one time step at a time, using **recurrence**:

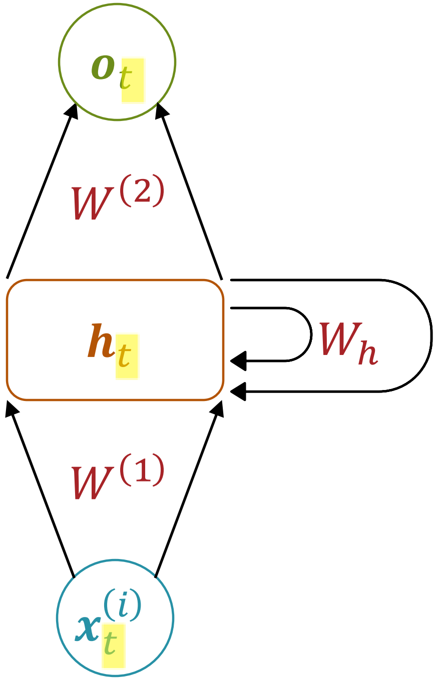
  $\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h\boldsymbol{h}_{t-1}\right)\right]^T$ and $\boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(2)}\boldsymbol{h}_t\right)$

  Where $\boldsymbol{h}_t$ serves as a summary or latent representation of the sequence up to time t.

  - The same parameters $W^{(1)}$, W_h and $W^{(2)}$ are reused at every step.
  - We can unroll the RNN for as many time steps as the sequence requires.
  - So, at training and inference time, the RNN can run for different numbers of steps depending on the input length.

Recurrent Neural Networks

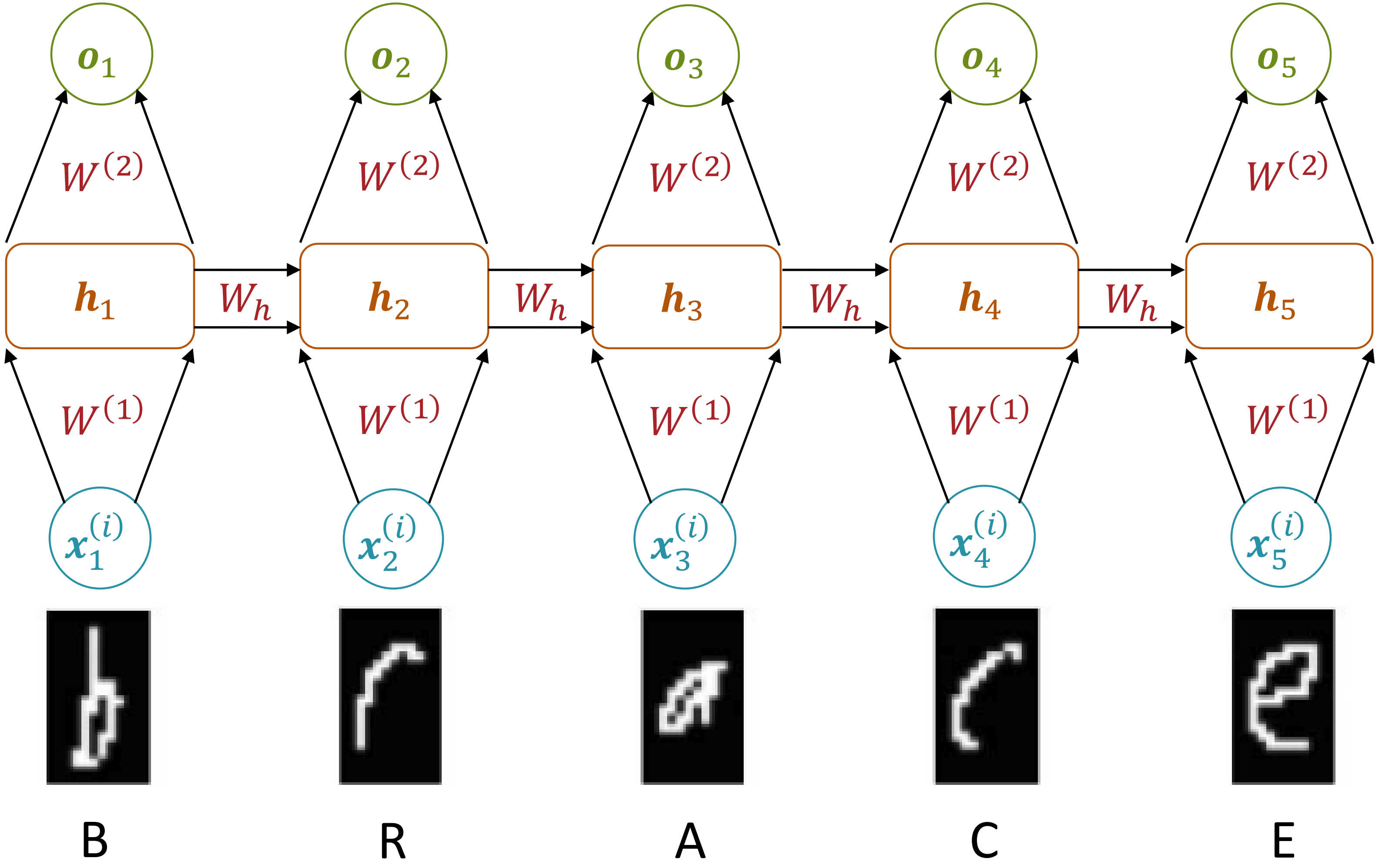
$$\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h \boldsymbol{h}_{t-1}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(2)}\boldsymbol{h}_t\right)$$



- This model requires an initial value for the hidden representation, $\boldsymbol{h}_0$, typically a vector of all zeros

Unrolling Recurrent Neural Networks

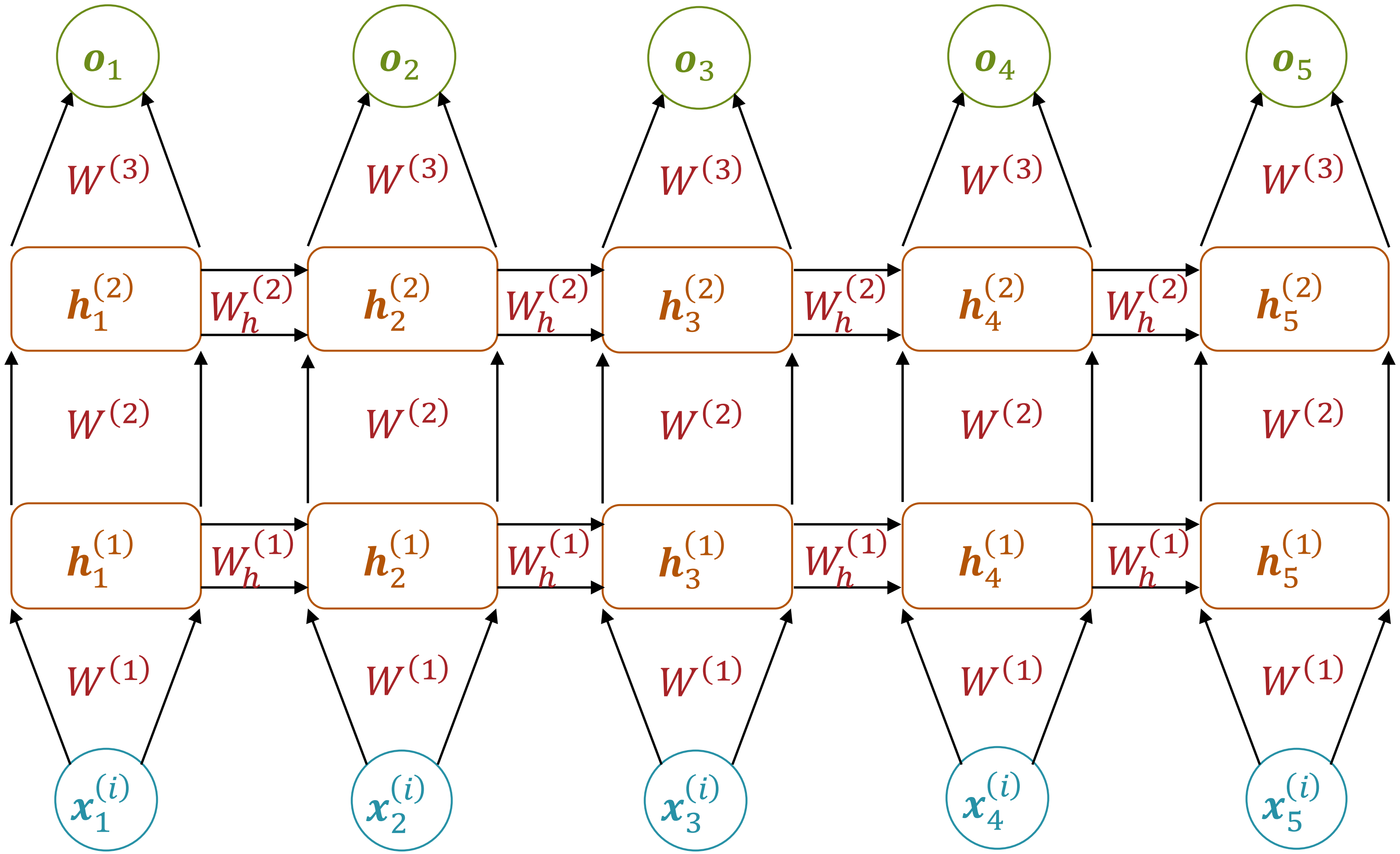
$$\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h\boldsymbol{h}_{t-1}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta(W^{(2)}\boldsymbol{h}_t)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

Deep Recurrent Neural Networks

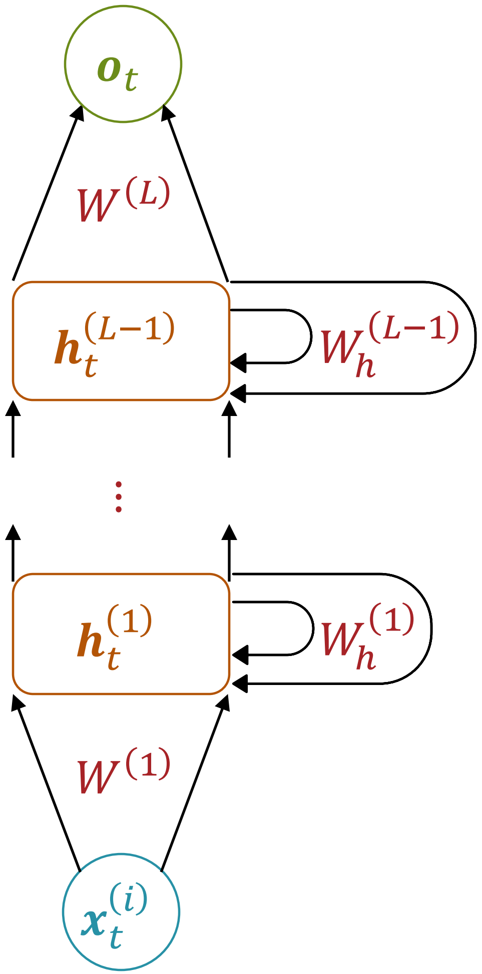
$$\boldsymbol{h}_t^{(l)} = \left[1, \theta\left(W^{(l)}\boldsymbol{h}_t^{(l-1)} + W_h^{(l)}\boldsymbol{h}_{t-1}^{(l)}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(L)}\boldsymbol{h}_t^{(L-1)}\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

Deep Recurrent Neural Networks

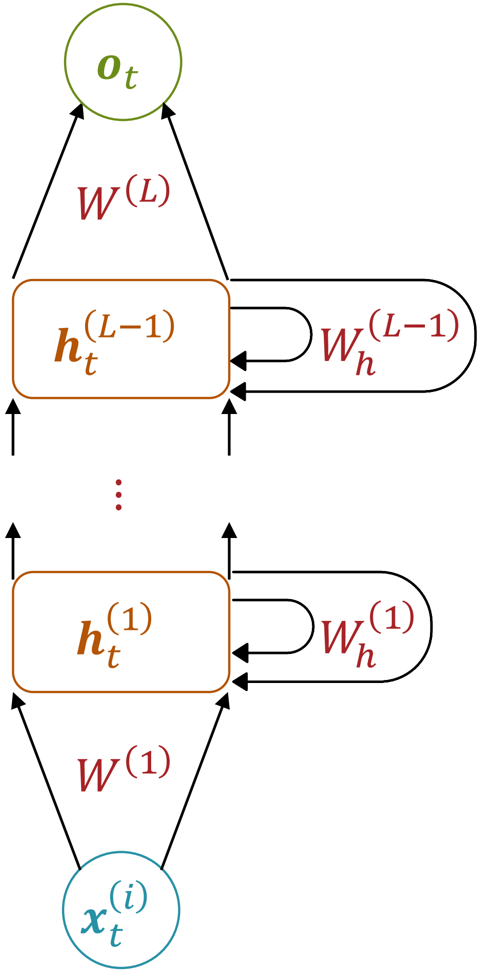
$$\boldsymbol{h}_t^{(l)} = \left[1, \theta\left(W^{(l)}\boldsymbol{h}_t^{(l-1)} + W_h^{(l)}\boldsymbol{h}_{t-1}^{(l)}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(L)}\boldsymbol{h}_t^{(L-1)}\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

But why do we only pass information forward? What if later time steps have useful information as well?

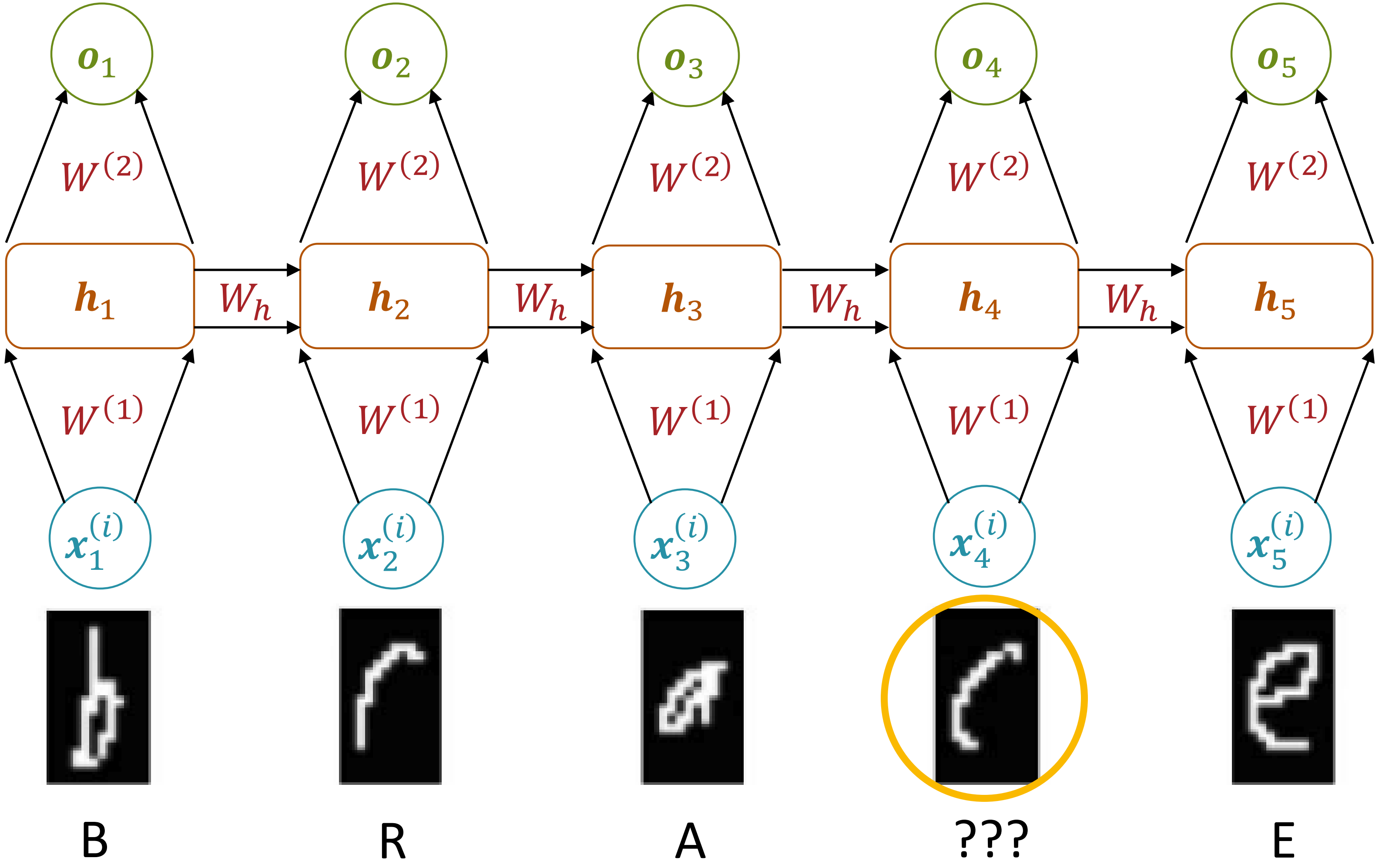
$$\boldsymbol{h}_t^{(l)} = \left[1, \theta\left(W^{(l)}\boldsymbol{h}_t^{(l-1)} + W_h^{(l)}\boldsymbol{h}_{t-1}^{(l)}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(L)}\boldsymbol{h}_t^{(L-1)}\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

But why do we only pass information forward? What if later time steps have useful information as well?

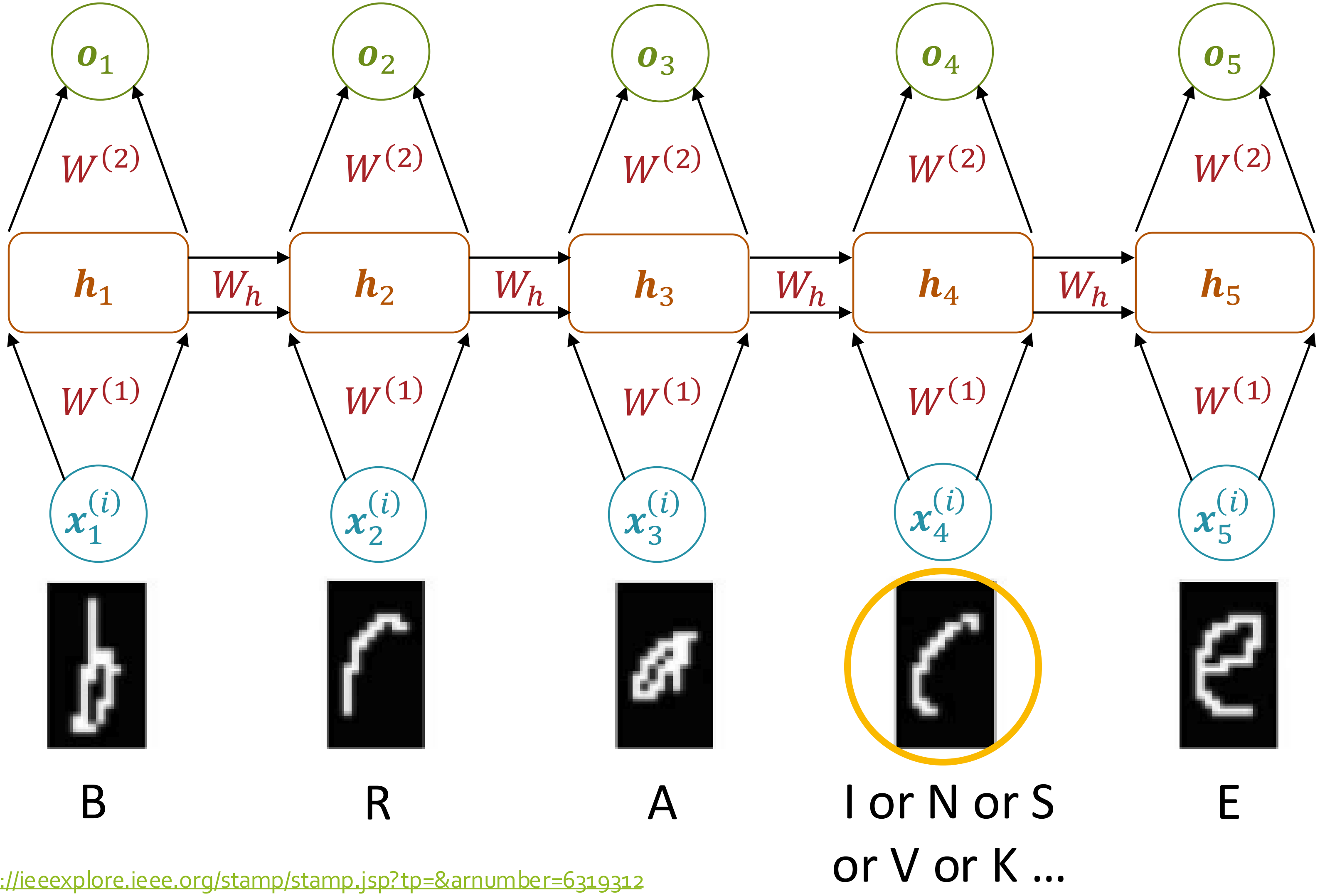
$$\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h \boldsymbol{h}_{t-1}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(2)}\boldsymbol{h}_t\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

But why do we only pass information forward? What if later time steps have useful information as well?

$$\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h \boldsymbol{h}_{t-1}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(2)}\boldsymbol{h}_t\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

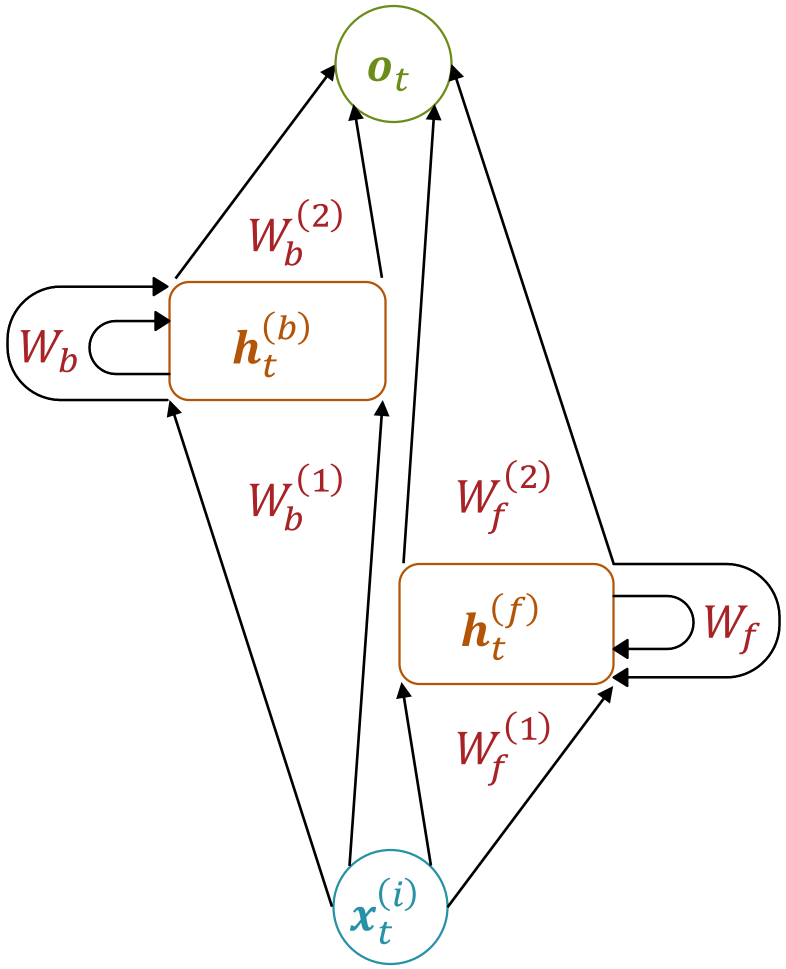
Bidirectional Recurrent Neural Networks

- Bidirectional Recurrent Neural Networks (BiRNNs) capture **context from both the past and the future** of a sequence.
- A BiRNN has two RNNs:
  - one $\boldsymbol{h}_t^{(f)}$ processes the sequence **forward in time**
  - one $\boldsymbol{h}_t^{(b)}$ processes it **backward in time**
  - The combination contains information from the entire sequence centered around position t.

Bidirectional Recurrent Neural Networks

$\boldsymbol{h}_t^{(f)} = \left[1, \theta\left(W_f^{(1)}\boldsymbol{x}_t^{(i)} + W_f\boldsymbol{h}_{t-1}\right)\right]^T$ and $\boldsymbol{h}_t^{(b)} = \left[1, \theta\left(W_b^{(1)}\boldsymbol{x}_t^{(i)} + W_b\boldsymbol{h}_{t+1}\right)\right]^T$

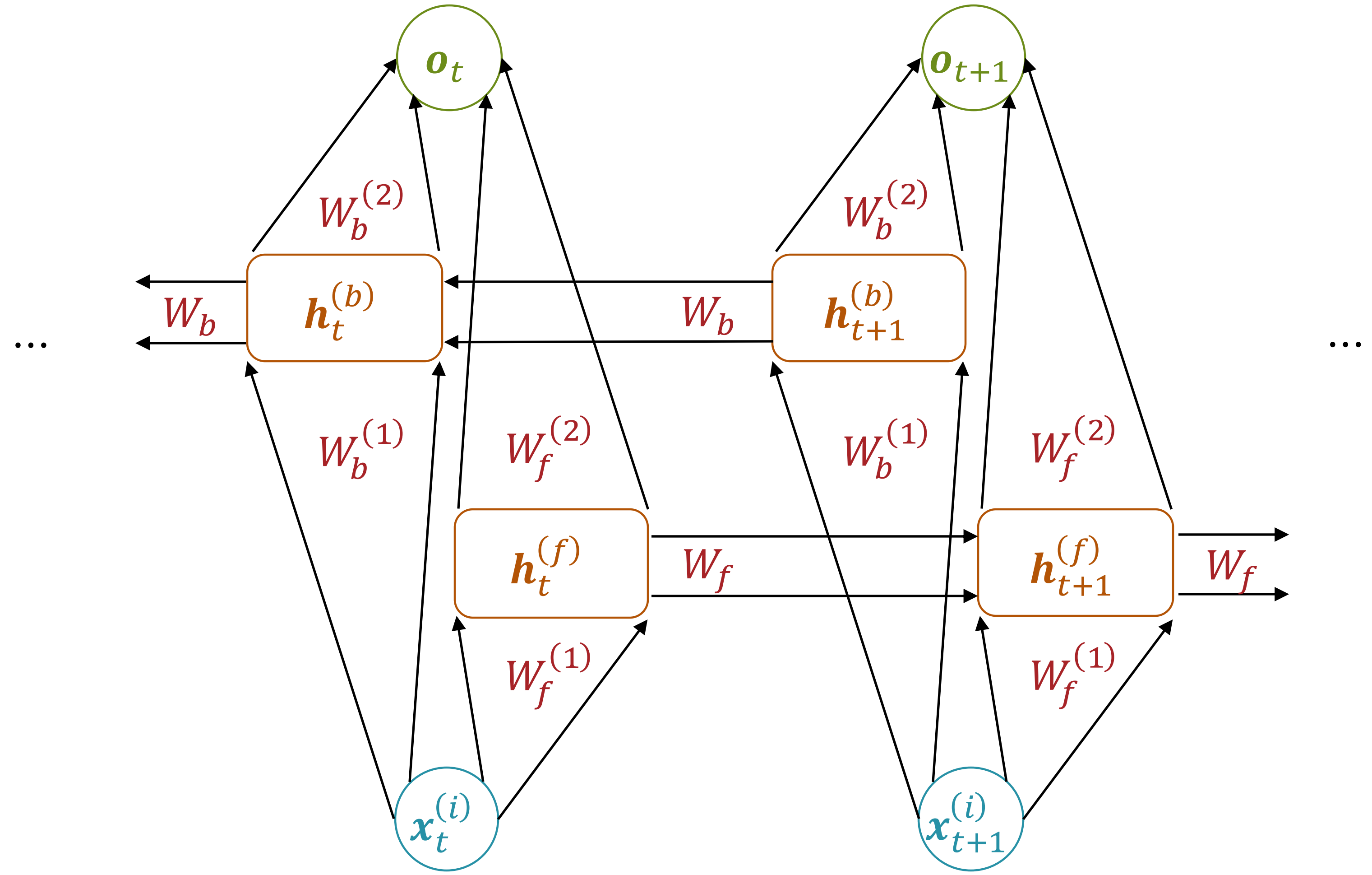
$$\boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W_f^{(2)}\boldsymbol{h}_t^{(f)} + W_b^{(2)}\boldsymbol{h}_t^{(b)}\right)$$



$$\boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W_f^{(2)}\boldsymbol{h}_t^{(f)} + W_b^{(2)}\boldsymbol{h}_t^{(b)}\right)$$

Unrolling Bidirectional Recurrent Neural Networks

$\boldsymbol{h}_t^{(f)} = \left[1, \theta\left(W_f^{(1)}\boldsymbol{x}_t^{(i)} + W_f\boldsymbol{h}_{t-1}\right)\right]^T$ and $\boldsymbol{h}_t^{(b)} = \left[1, \theta\left(W_b^{(1)}\boldsymbol{x}_t^{(i)} + W_b\boldsymbol{h}_{t+1}\right)\right]^T$



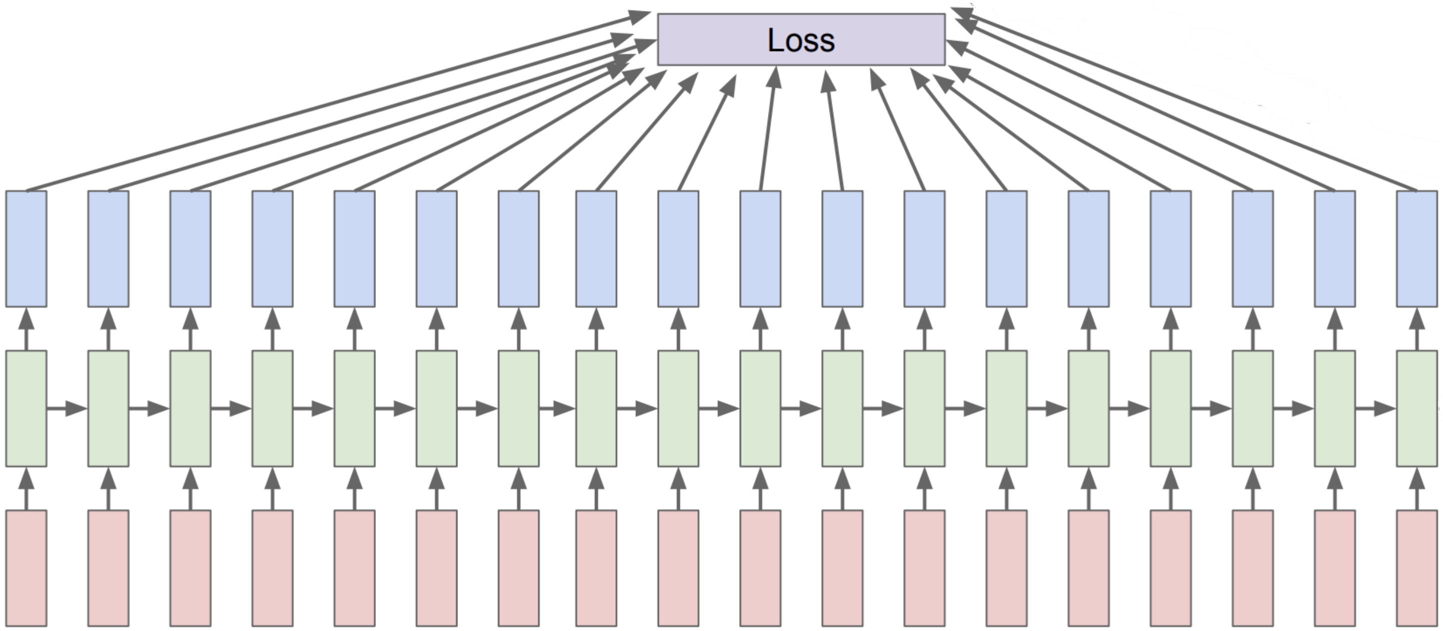
Training RNNs

- A (deep/bidirectional) RNN simply represents a (somewhat complicated) computation graph
  - Weights ($W^{(1)}$, W_h and $W^{(2)}$) are shared between different timesteps, significantly reducing the number of parameters to be learned!
- Can be trained using (stochastic) gradient descent/ backpropagation → "backpropagation through time"

Backprop Through Time

- Each hidden state h_t influences not only its immediate output y_t ,but also all future hidden states $h_{t+1}, h_{t+2}, \ldots$.
- Thus, each parameter affects the loss indirectly **through time**.
- So, during training, need to propagate the gradient back through all those time steps.
- To train the RNN, we unroll it over the sequence, treating it like a deep feed-forward network with T layers — one per time step — all sharing the same parameters.
- Then, we apply standard backpropagation over this unrolled network.

Training RNNs

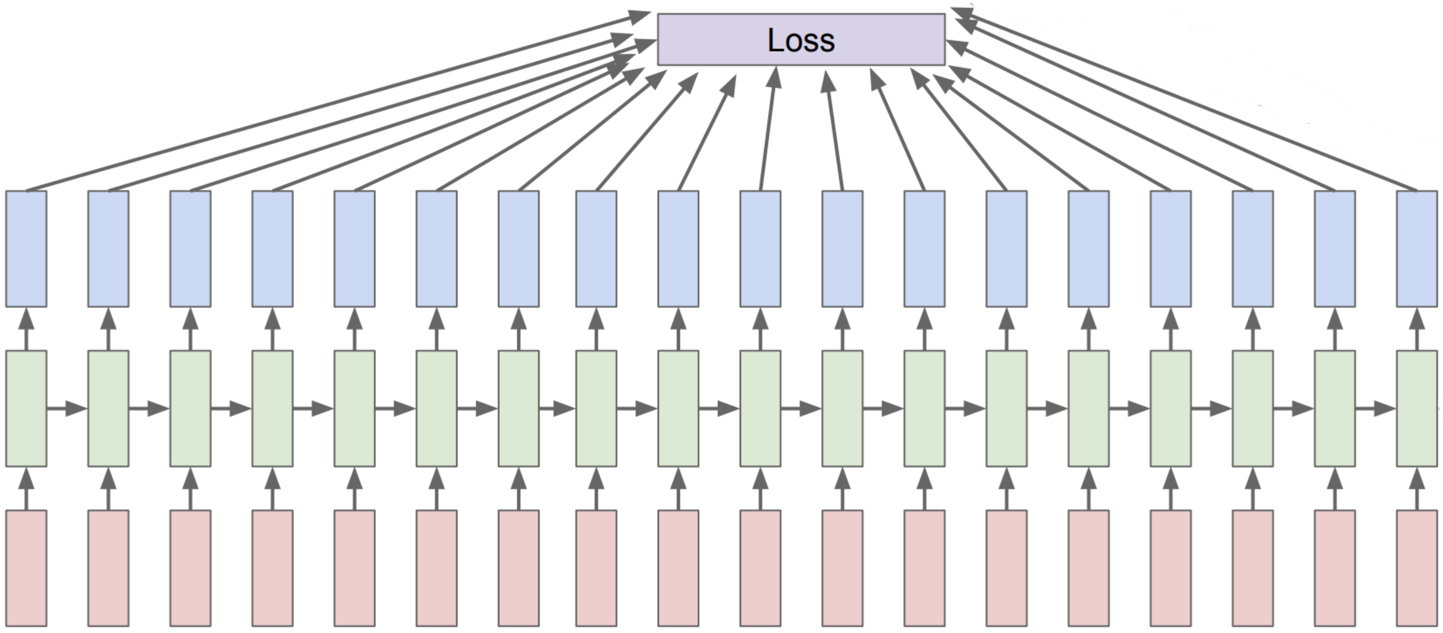


Forward pass to compute outputs and hidden representations

Backward pass to compute gradients

Source: http://cs231n.stanford.edu/slides/2023/lecture_8.pdf

Training RNNs: Challenges



Forward pass to compute outputs and hidden representations

Backward pass to compute gradients

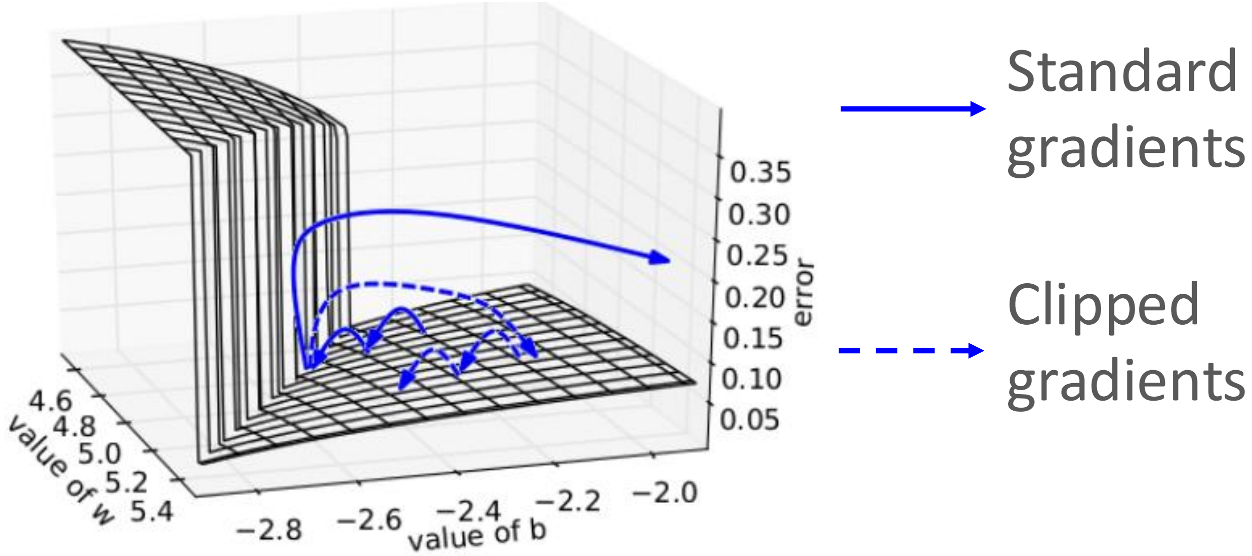
- Issue: as the sequence length grows, the gradient is more likely to explode or vanish

Source: http://cs231n.stanford.edu/slides/2023/lecture_8.pdf

Gradient Clipping (Pascanu et al., 2013)

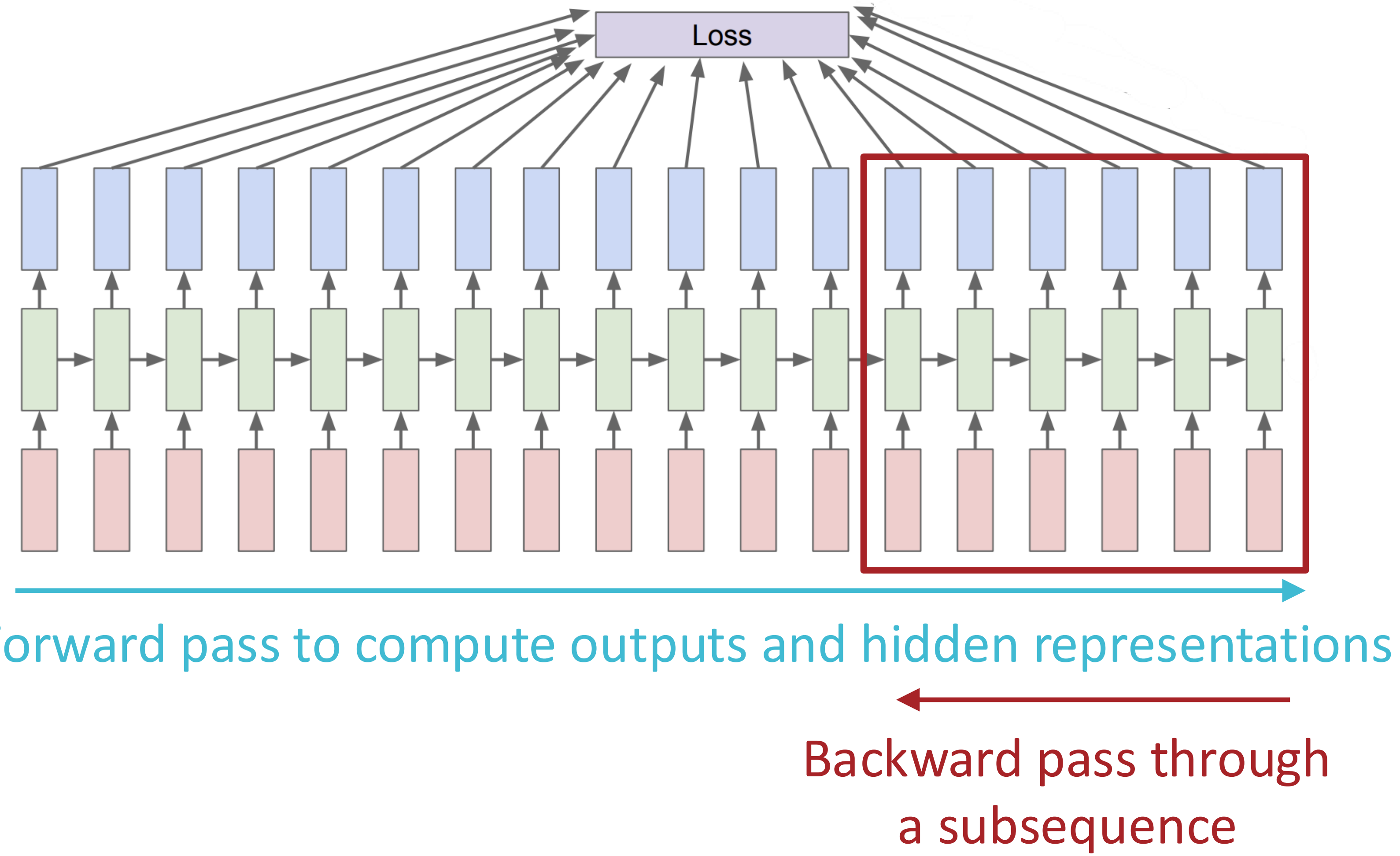
- Common strategy to deal with exploding gradients: if the magnitude of the gradient ever exceeds some threshold, simply scale it down to the threshold

$$G = \begin{cases} \nabla_W \ell^{(i)} & \text{if } \left\|\nabla_W \ell^{(i)}\right\|_2 \leq \tau \\ \left(\dfrac{\tau}{\left\|\nabla_W \ell^{(i)}\right\|_2}\right) \nabla_W \ell^{(i)} & \text{otherwise} \end{cases}$$



Source: https://arxiv.org/pdf/1211.5063.pdf

Truncated Backpropagation Through Time



- Idea: limit the number of time steps to backprop through

Source: http://cs231n.stanford.edu/slides/2023/lecture_8.pdf

Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

- LSTM networks address the vanishing gradient problem by replacing hidden layers with *memory cells*
- Each cell still computes a hidden representation $\boldsymbol{h}_t$ but also maintains a separate internal *state,* C_t
- The flow of information through a cell is manipulated by three *gates*:
  - An input gate, I_t, that controls how much the state looks like the normal RNN hidden layer
  - An output gate, O_t, that "releases" the hidden representation to later timesteps
  - A forget gate, F_t, that determines if the previous memory cell's state affects the current internal state

Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

- LSTM networks address the vanishing gradient problem by replacing hidden layers with *memory cells*
- Each cell still computes a hidden representation $\boldsymbol{h}_t$ but also maintains a separate internal *state*, C_t
- Gates are implemented as sigmoids: a value of 0 would be a fully closed gate and 1 would be fully open

$$I_t = \sigma\left(W_{ix}\boldsymbol{x}_t^{(i)} + W_{ih}\boldsymbol{h}_{t-1}\right)$$

$$O_t = \sigma\left(W_{ox}\boldsymbol{x}_t^{(i)} + W_{oh}\boldsymbol{h}_{t-1}\right)$$

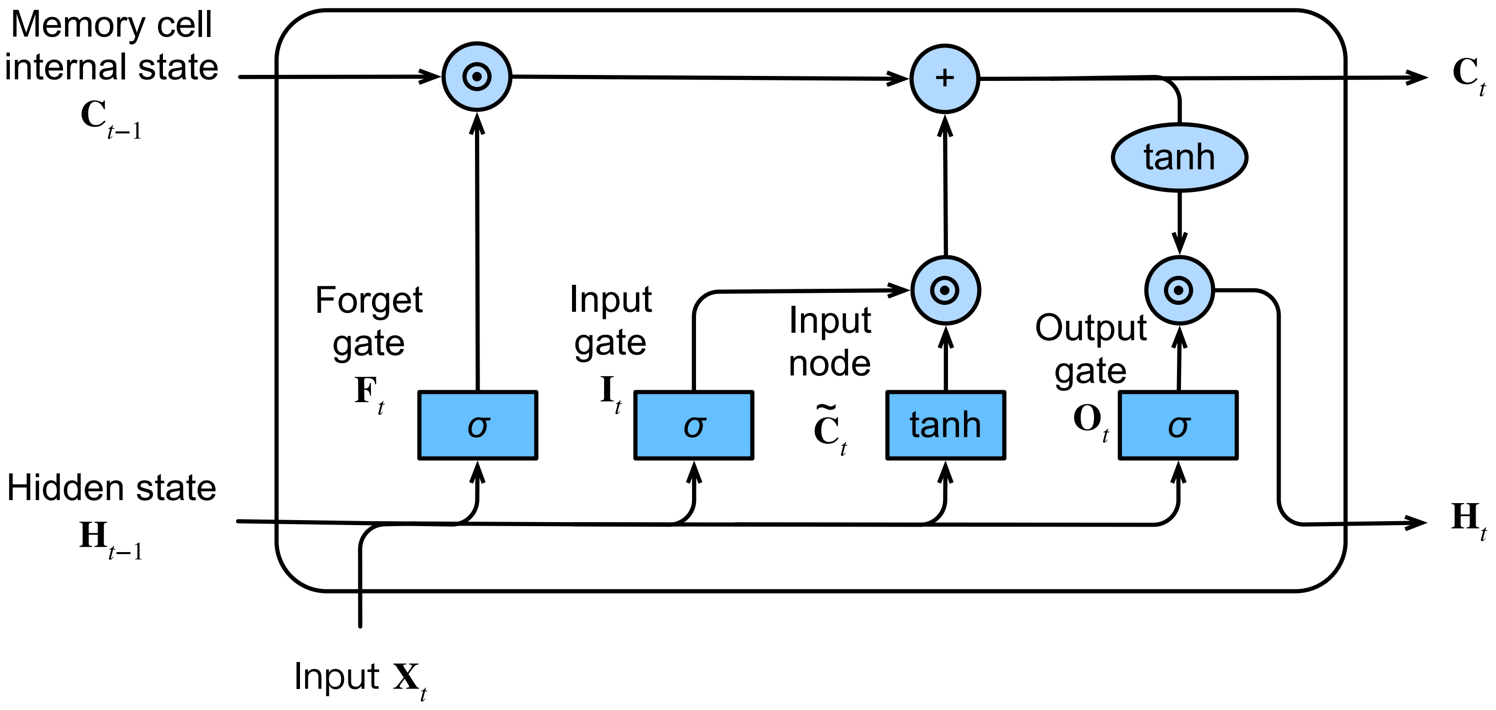
$$F_t = \sigma\left(W_{fx}\boldsymbol{x}_t^{(i)} + W_{fh}\boldsymbol{h}_{t-1}\right)$$

$$C_t = F_t \odot C_{t-1} + I_t \odot \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h\boldsymbol{h}_{t-1}\right)$$

$$\boldsymbol{h}_t = C_t \odot O_t$$

Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

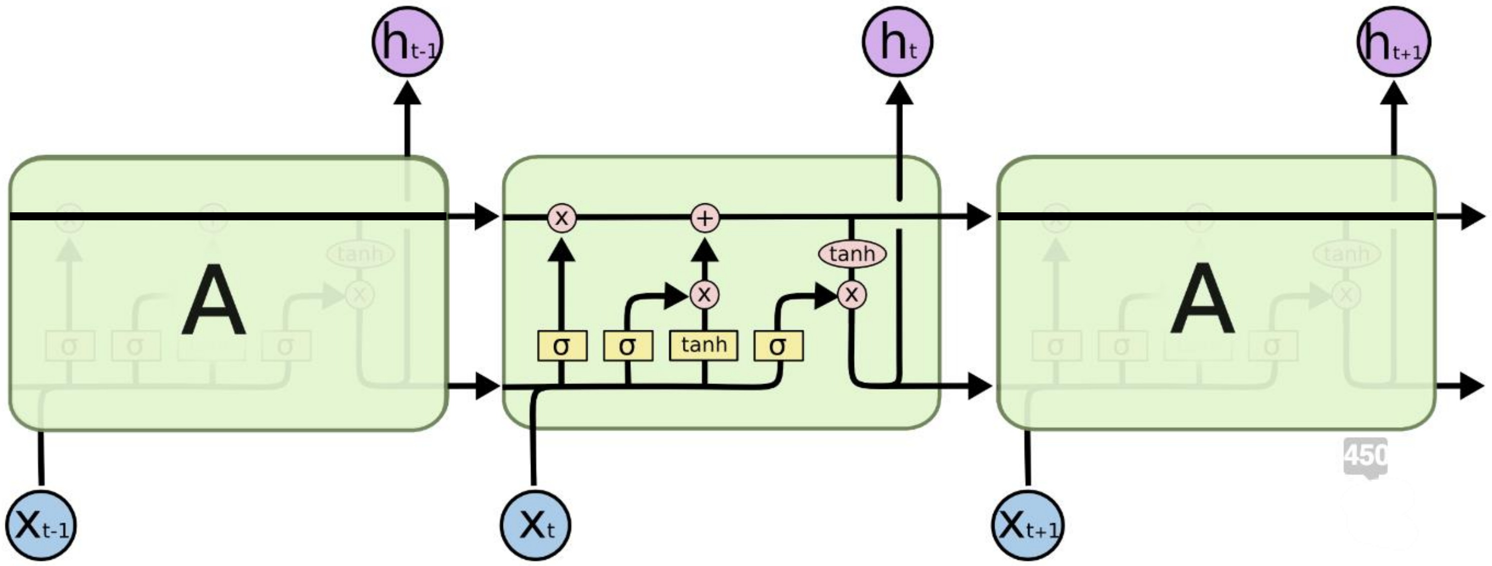
- LSTM networks address the vanishing gradient problem by replacing hidden layers with *memory cells*
- Each cell still computes a hidden representation H_t but also maintains a separate internal *state*, C_t



Source: https://d2l.ai/chapter_recurrent-modern/lstm.html

Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

- LSTM networks address the vanishing gradient problem by replacing hidden layers with *memory cells*
- Each cell still computes a hidden representation $\boldsymbol{h}_t$ but also maintains a separate internal *state*, C_t



- The internal state allows information to move through time without needing to affect the hidden representations!

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Applications of LSTMs

2018: OpenAI used LSTM trained by policy gradients to beat humans in the complex video game of Dota 2,[11] and to control a human-like robot hand that manipulates physical objects with unprecedented dexterity.[10][54]

2019: DeepMind used LSTM trained by policy gradients to excel at the complex video game of Starcraft II.[12][54]

Source: https://en.wikipedia.org/wiki/Long_short-term_memory

Key Takeaways

- Recurrent neural networks use contextual information to reason about sequential data.
    - Can still be learned using backpropagation → backpropagation through time.
    - Susceptible to exploding/vanishing gradients for long training sequences.
    - LSTMs allow contextual information to reach later timesteps without directly affecting intermediate hidden representations.