Convolutional Neural Networks

① Let's say somebody asks you to implement a function f(b) which takes a bitstring b as its argument and returns true if bitstring b contains the substring "10"

I guess you could write:

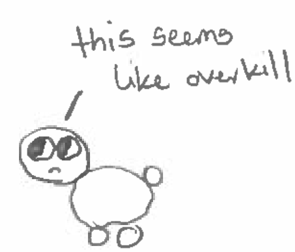
```
def f(b):
    return ("10" in b)
```

But why go to all that trouble when you could train a neural network to do it?

---

② First we collect some training data:

positive examples	negative examples
1011	0001
0110	0000
0100	0111
1110	1111
1010	0011



Convolutional Neural Networks

③ This is a hard problem, so let's begin by just solving it for bitstrings of length 4.

Our evidence variables will just be the bits in the bitstring; while our response will be whether "10" appears in the bitstring:

X (evidence vars)				y (response)
x_1 (bit1)	x_2 (bit2)	x_3 (bit3)	x_4 (bit4)	
1	0	1	1	1
0	0	0	1	0
0	1	1	0	1
0	0	0	0	0
1	1	1	0	1
1	1	1	1	0

Convolutional Neural Networks

④ We'll derive three new evidence variables $\dot{x}_1, \dot{x}_2, \dot{x}_3$ which indicate whether "10" appears starting at position 1, 2, or 3 in the bitstring:

$$\dot{x}_1 = a(x_1 - x_2)$$
$$\dot{x}_2 = a(x_2 - x_3)$$
$$\dot{x}_3 = a(x_3 - x_4)$$

where a is the ReLU function.

i.e. $a(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$

⑤ We can rewrite these as "activated" dot products:

$$\dot{x}_1 = a\left(\begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \right)$$

$$\dot{x}_2 = a\left(\begin{bmatrix} 0 \\ 1 \\ -1 \\ 0 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \right)$$

$$\dot{x}_3 = a\left(\begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \right)$$

Convolutional Neural Networks

(6) Finally, we can compute whether "10" appears in the bitstring as the sum of our derived features:

$$\ddot{\pi} = \dot{x}_1 + \dot{x}_2 + \dot{x}_3$$

Or, as a dot product:

$$\ddot{\pi} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}^T \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix}$$

---

(7) Good! We've built a neural network for determining whether "10" appears in a 4-bit bitstring:

$$\begin{tikzcd}
 & \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & -1 \end{bmatrix} \arrow[d] & & \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \arrow[d] \\
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \arrow[r] & \begin{bmatrix} \dot{\pi}_1 \\ \dot{\pi}_2 \\ \pi_3 \end{bmatrix} \arrow[r] & \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} \arrow[r] & \ddot{\pi}
\end{tikzcd}$$

Convolutional Neural Networks

⑧ So rather than set the weights ourselves, we can train the following neural network:

$$\begin{bmatrix} \dot{w}_{11} & \dot{w}_{12} & \dot{w}_{13} \\ \dot{w}_{21} & \dot{w}_{22} & \dot{w}_{23} \\ \dot{w}_{31} & \dot{w}_{32} & \dot{w}_{33} \\ \dot{w}_{41} & \dot{w}_{42} & \dot{w}_{43} \end{bmatrix} \qquad \begin{bmatrix} \ddot{w}_{11} \\ \ddot{w}_{21} \\ \ddot{w}_{31} \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \longrightarrow \begin{bmatrix} \dot{\pi}_1 \\ \dot{\pi}_2 \\ \dot{\pi}_3 \end{bmatrix} \longrightarrow \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} \longrightarrow \ddot{\pi}$$

⑨ But if we know upfront that our "codestring" 10 has length 2, then we don't really need to train 12 different weights in $\dot{W}$. There's a pattern:

$$\dot{W} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & -1 \end{bmatrix}$$

we're looking for the same thing at 3 possible locations in the string

CONVOLUTIONAL NEURAL NETWORKS

(10) In other words, we're applying some "detector function" f at every starting point of a 2-bit bitstring:

$$\underbrace{\begin{bmatrix} c_1 & 0 & 0 \\ c_2 & c_1 & 0 \\ 0 & c_2 & c_1 \\ 0 & 0 & c_2 \end{bmatrix}}_{\mathring{W}}{}^{T} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

$$= \begin{bmatrix} c_1 x_1 + c_2 x_2 \\ c_1 x_2 + c_2 x_3 \\ c_1 x_3 + c_2 x_4 \end{bmatrix}$$

$$= \begin{bmatrix} f(x_1, x_2) \\ f(x_2, x_3) \\ f(x_3, x_4) \end{bmatrix}$$

---

(11) So maybe we can get away with only training 2 parameters (c_1 and c_2) instead all 12 parameters of $\mathring{W}$.

(12) This can get increasingly important as the length of the bitstring increases:

$$\left.\begin{bmatrix} \begin{bmatrix}1\\-1\end{bmatrix} & 0 & \cdots & 0 \\ & \begin{bmatrix}1\\-1\end{bmatrix} & & 0 \\ 0 & & & 0 \\ \vdots & \vdots & \ddots & \begin{bmatrix}1\\-1\end{bmatrix} \\ 0 & 0 & \cdots & \end{bmatrix}\right\} \begin{matrix}(D-1)\times(D-1)\\ \text{matrix}\end{matrix}$$

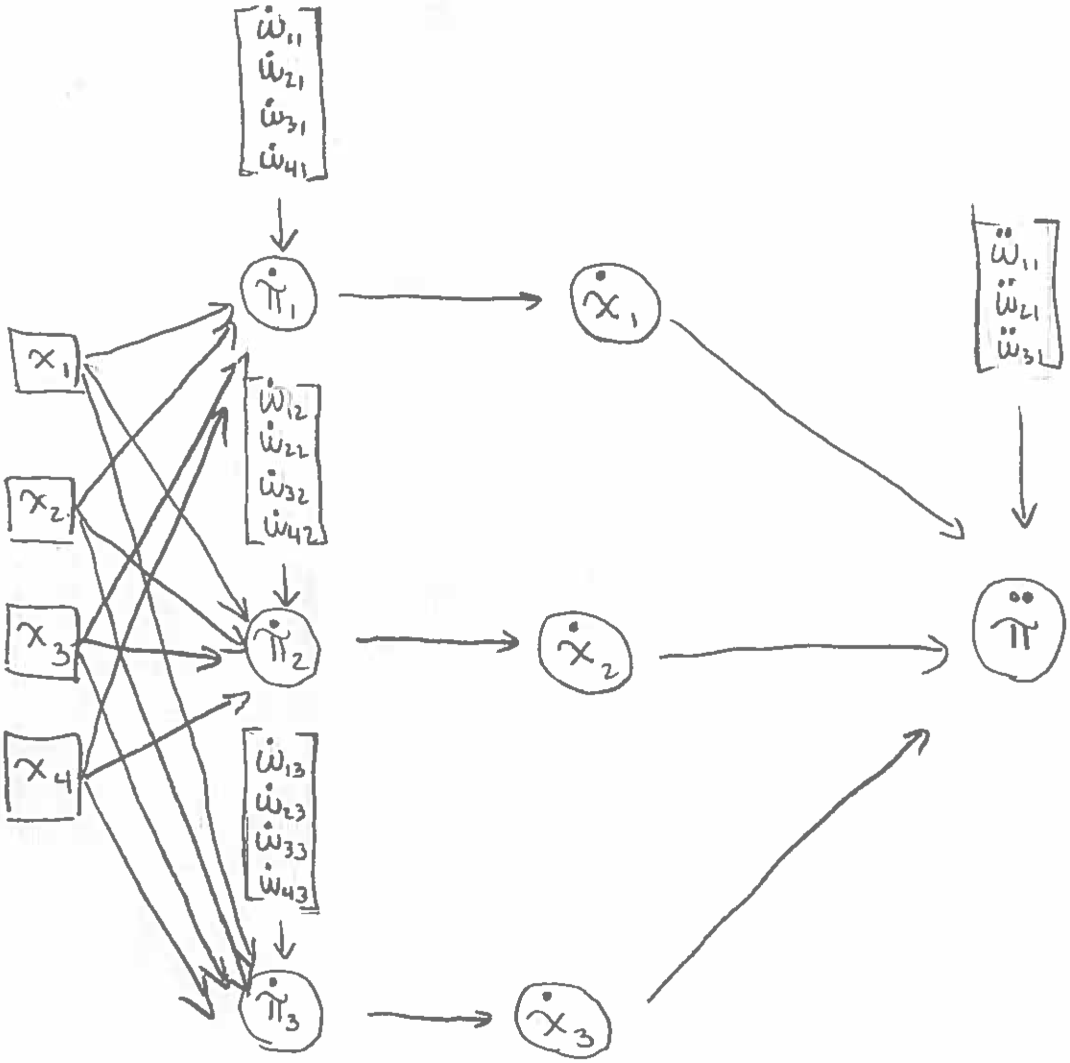
$$\downarrow$$

$$\begin{bmatrix}x_1\\ \vdots \\ x_D\end{bmatrix} \longrightarrow \begin{bmatrix}\pi_1\\ \vdots\\ \pi_{D-1}\end{bmatrix}$$

For a D-length bitstring, we would train $(D-1)^2$ parameters using our naive approach, but still only 2 parameters with our factored approach.

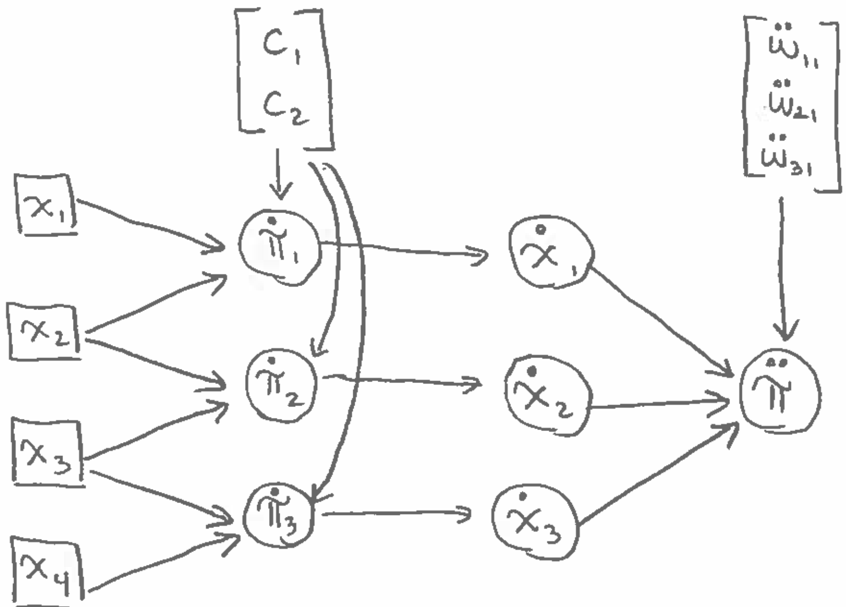
Convolutional Neural Networks

13 Going back to the 4-bit case, let's expand the fully-connected feedforward neural network:



CONVOLUTIONAL NEURAL NETWORKS

(14) Our factored alternative looks as follows:



(15) In general, having fewer parameters to train gives us:

- faster training
- less risk of overfitting

Of course, this will only work if the assumption behind our factoring actually holds, i.e. we're looking for some "local" substring of size 2.

(10) The impact of this change on backpropagation is relatively minor. Recall that:

$$\frac{\partial \ddot{\pi}}{\partial \dot{\omega}_{ij}} = \frac{\partial \ddot{\pi}}{\partial \dot{\pi}_j} \cdot \frac{\partial \dot{\pi}_j}{\partial \dot{\omega}_{ij}} \quad \left[\text{b/c } \dot{\pi}_j \text{ separates } \ddot{\pi} \text{ from } \dot{\omega}_{ij}\right]$$

$$= \frac{\partial \ddot{\pi}}{\partial \dot{\pi}_j} \cdot x_i$$

for the fully-connected feedforward neural network.

For the factored network, the main difference is that no single $\dot{\pi}_j$ separates a "shared weight" c_i from $\ddot{\pi}$.

So:

$$\frac{\partial \ddot{\pi}}{\partial c_i} = \sum_{h=1}^{H} \frac{\partial \ddot{\pi}}{\partial \dot{\pi}_h} \cdot \frac{\partial \dot{\pi}_h}{\partial c_i} \quad \left[\text{b/c } \{\dot{\pi}_1, \ldots, \dot{\pi}_H\} \text{ separates } \ddot{\pi} \text{ from } c_j\right]$$

$$= \sum_{h=1}^{H} \frac{\partial \ddot{\pi}}{\partial \dot{\pi}_h} \cdot \frac{\partial (c_1 x_h + c_2 x_{h+1})}{\partial c_i}$$

$$= \sum_{h=1}^{H} \frac{\partial \ddot{\pi}}{\partial \dot{\pi}_h} x_{h+(i-1)}$$

Convolutional Neural Networks

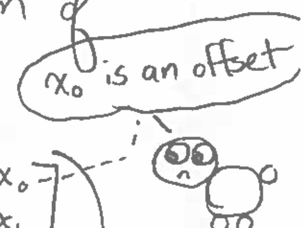
(17) Now, suppose we wanted to detect whether a 6-bit bitstring contained the substring "1001" or "0110".

positive examples	negative examples
100100	110100
010110	000001
010010	110111
011001	000111

(18) We could do something similar in which we create new evidence variables that indicate whether "1001" (respectively, "0110") starts at a particular position of the bitstring:

to identify "1001"

$$\begin{cases} \dot{x}_1 = a(-1 + x_1 - x_2 - x_3 + x_4) = a\left(\begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}\right) \\ \dot{x}_2 = a(-1 + x_2 - x_3 - x_4 + x_5) \\ \dot{x}_3 = a(-1 + x_3 - x_4 - x_5 + x_6) \end{cases}$$



to identify "0110"

$$\begin{cases} \dot{x}_4 = a(-1 - x_1 + x_2 + x_3 - x_4) = a\left(\begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}\right) \\ \dot{x}_5 = a(-1 - x_2 + x_3 + x_4 - x_5) \\ \dot{x}_6 = a(-1 - x_3 + x_4 + x_5 - x_6) \end{cases}$$

(19) This strategy gives us the following neural network for detecting "1001" or "0110":

$$\underbrace{\begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ -1 & 1 & 0 & 1 & -1 & 0 \\ -1 & -1 & 1 & 1 & 1 & -1 \\ 1 & -1 & -1 & -1 & 1 & 1 \\ 0 & 1 & -1 & 0 & -1 & 1 \\ 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix}}_{\dot{W}} \qquad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \xrightarrow{\dot{W}} \begin{bmatrix} \dot{\pi}_1 \\ \dot{\pi}_2 \\ \dot{\pi}_3 \\ \dot{\pi}_4 \\ \dot{\pi}_5 \\ \dot{\pi}_6 \end{bmatrix} \longrightarrow \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \end{bmatrix} \longrightarrow \ddot{\pi}$$

positive iff the string contains 1001 or 0110

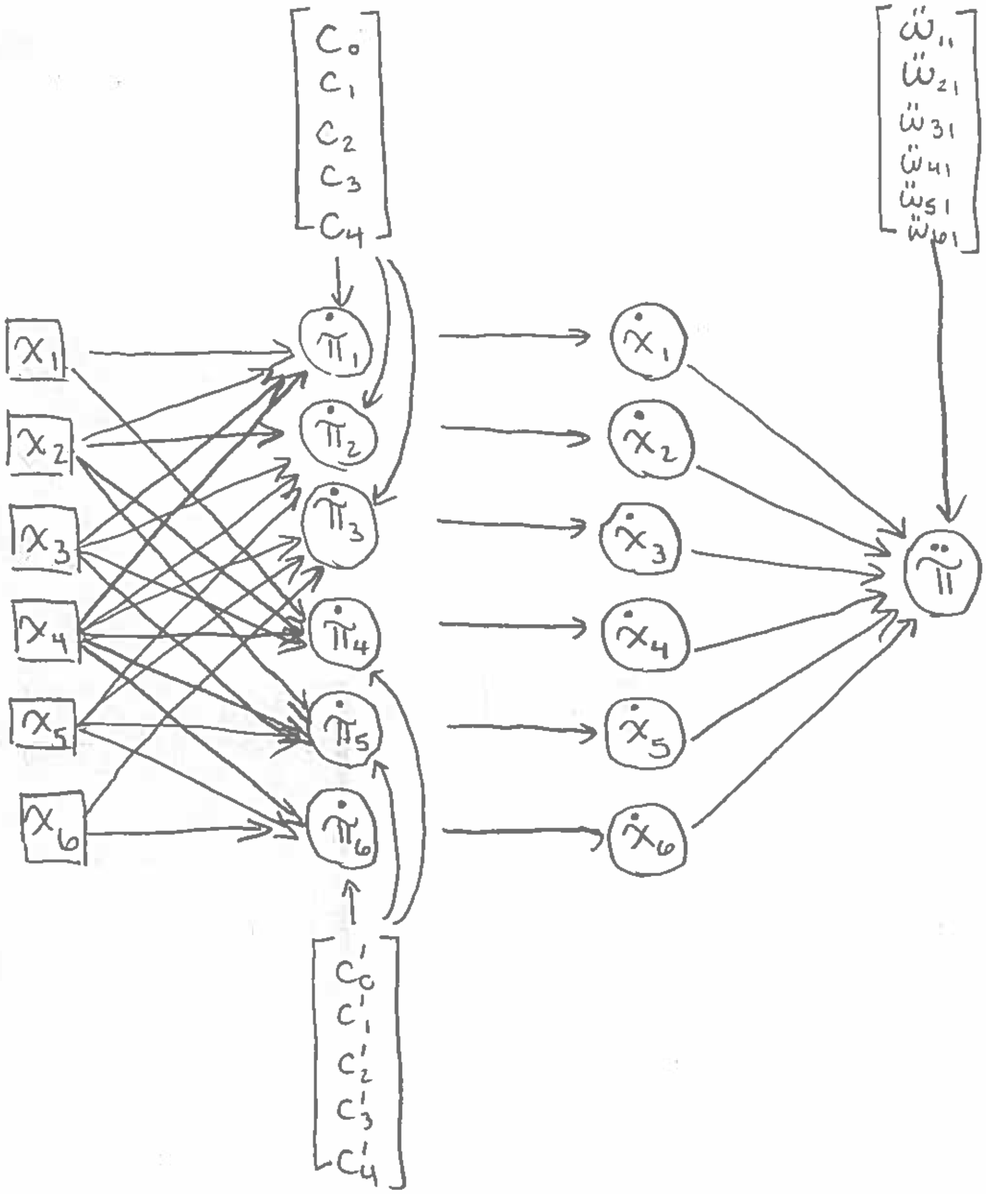
(20) Again, $\dot{W}$ can be factorized:

$$\dot{W} = \begin{bmatrix} c_0 & c_0 & c_0 & c_0' & c_0' & c_0' \\ c_1 & 0 & 0 & c_1' & 0 & 0 \\ c_2 & c_1 & 0 & c_2' & c_1' & 0 \\ c_3 & c_2 & c_1 & c_3' & c_2' & c_1' \\ c_4 & c_3 & c_2 & c_4' & c_3' & c_2' \\ 0 & c_4 & c_3 & 0 & c_4' & c_3' \\ 0 & 0 & c_4 & 0 & 0 & c_4' \end{bmatrix}$$

which gives us only 10 parameters to train, rather than 42 (the size of $\dot{W}$).

CONVOLUTIONAL NEURAL NETWORKS

21) The factored neural network looks like this:

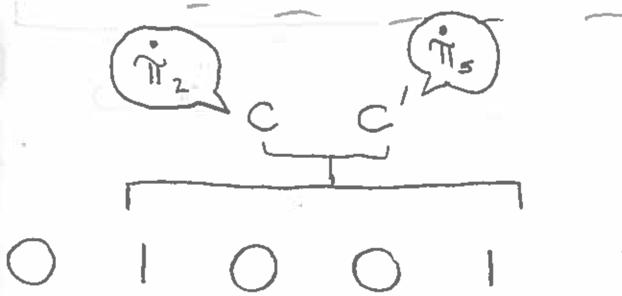


CONVOLUTIONAL NEURAL NETWORKS

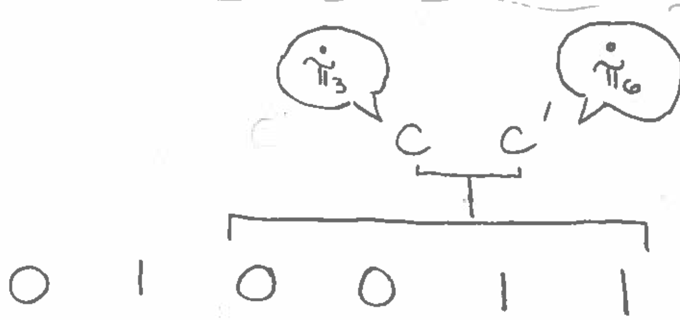
(22) To summarize, we are sliding our convolution "kernels" $\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$ and $\begin{bmatrix} c_1' \\ c_2' \\ c_3' \\ c_4' \end{bmatrix}$ across the input bitstring to detect the substrings 1001 and 0110, respectively

first:

then:



finally:



CONVOLUTIONAL NEURAL NETWORKS

(23) When designing a convolutional layer, there are several aspects to consider:

- Kernel size: In the first example, we had a kernel size of 2 (we were looking for substrings of length 2). In the second, we had a kernel size of 4 (we were looking for substrings of length 4). The kernel size is the dimension of each convolution kernel (vector).
- number of kernels: In the first example, we had 1 kernel (which identified the substring 10). In the second, we had 2 kernels (which identified substrings 0110 and 1001).
- stride: This is how much we advance the kernels. Both examples used a stride of 1, meaning that they applied each kernel to position 1, 2, 3, etc. of the input string. A stride of 2, on the other hand, would apply each kernel to positions 1, 3, 5, etc. (effectively skipping substrings that start at even positions).

CONVOLUTIONAL NEURAL NETWORKS

(24) Exercise: Consider a convolutional layer with 1 kernel of size k. If the input bitstring has length n, what is the dimension of $\dot{\pi}$?