Convolutional Neural Networks (CNN): HIgh Level

TL;DR

- A single unit in Fully Connected (FC) Layer identifies the presence/absence of a single feature spanning the **entire** input
- A single "kernel" in a Convolutional Layer identifies the presence/absence of a single feature
    - Whose size is a fraction of the entire input
    - At **each** sub-span of the input

Example

- FC: is the input image the digit "8"
- CNN: are there one or more small "8"'s in the input image

Convolutional Neural Networks: Introduction

We have seen how the Fully Connected (FC) layer performs a template-matching on the layer's inputs.

$$\mathbf{y}_{(l)} = a_{(l)}(\mathbf{W}_{(l)}\mathbf{y}_{(l-1)} + b)$$

- Each element of $\mathbf{y}_{(l-1)}$ is independent
  - there is no relationship between $\mathbf{y}_{(l-1),j}$ and $\mathbf{y}_{(l-1),j+1}$
  - even though they are adjacent in the vector ordering

To see the lack of relationship:

Let perm be a random ordering of the integers in the range $[1 \ldots n]$.

Then

- $\mathbf{x}[\mathrm{perm}]$ is a permutation of input $\mathbf{x}$
- $\Theta[\mathrm{perm}]$ is the corresponding permutation of parametrs Θ.

$$\Theta^T \cdot \mathbf{x} = \mathbf{x}[\mathrm{perm}] \cdot \Theta[\mathrm{perm}]$$

So a FC layer cannot take advantage of any explicit ordering among the input elements

- timeseries of prices
- adjacent pixels in an image

Another issue with an FC layer:

- The "template" $\mathbf{W}_{(l)}$ matches the full length of the input $\mathbf{y}_{(l-1)}$

There are cases where we might want to discover a feature

- whose length is less than n
- that occurs *anywhere* in the input, rather than at a fixed location

For example

- a spike in a timeseries
- the presence of an "eye" in an image

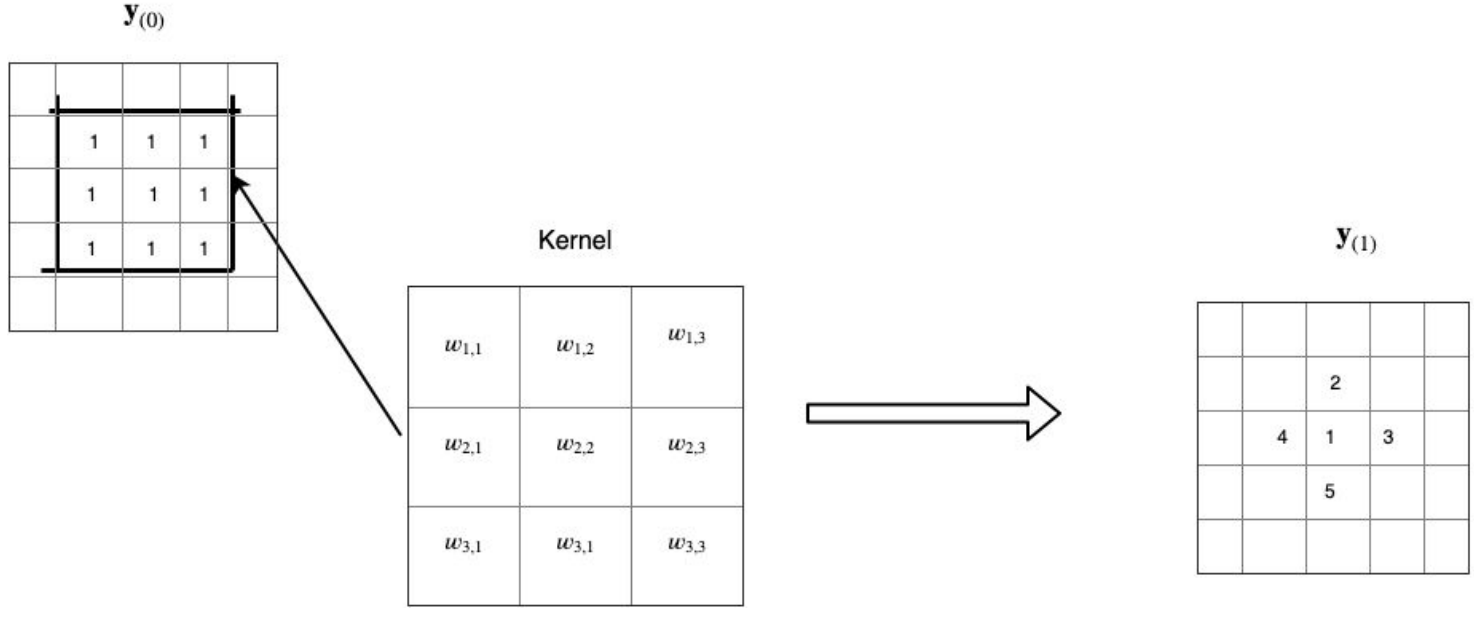
Both these questions motivate the notion of convolutional matching

- small templates
- that are slid over the entire input

By sliding the pattern over the entire input

- we can detect the existence of the feature somewhere in the input
- we can localize its location

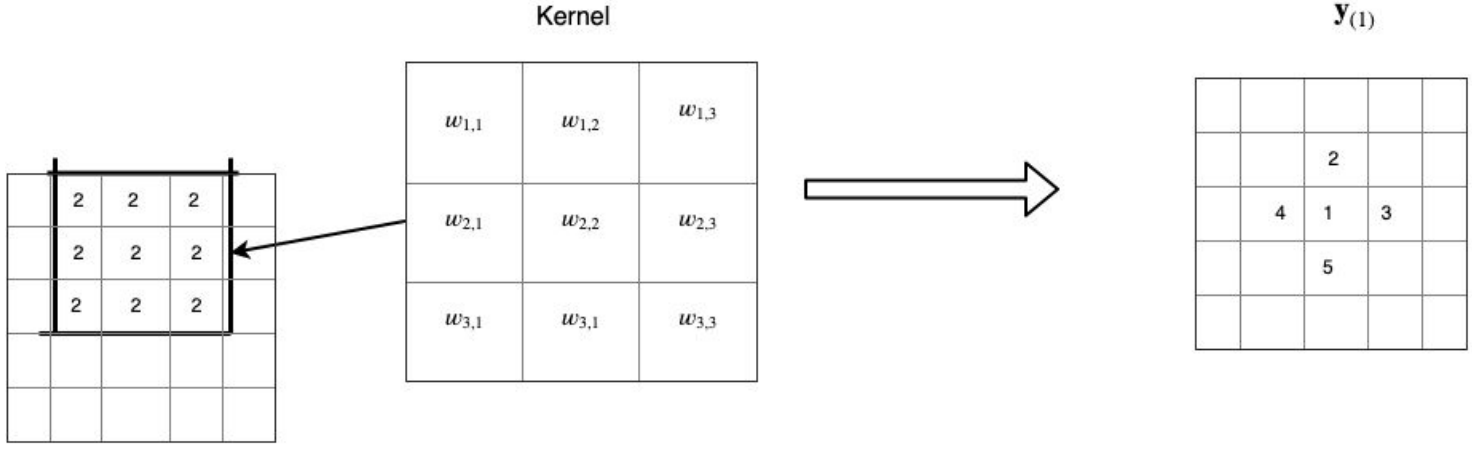
CNN convolution



- We place (3×3) Kernel (weight matrix) on the inputs ($\mathbf{y}_{(0)}$, output of layer 0)
- Performs dot product
- Produces Layer 1 output ($\mathbf{y}_{(1)}$) feature labelled 1

Slide the Kernel up and repeat:

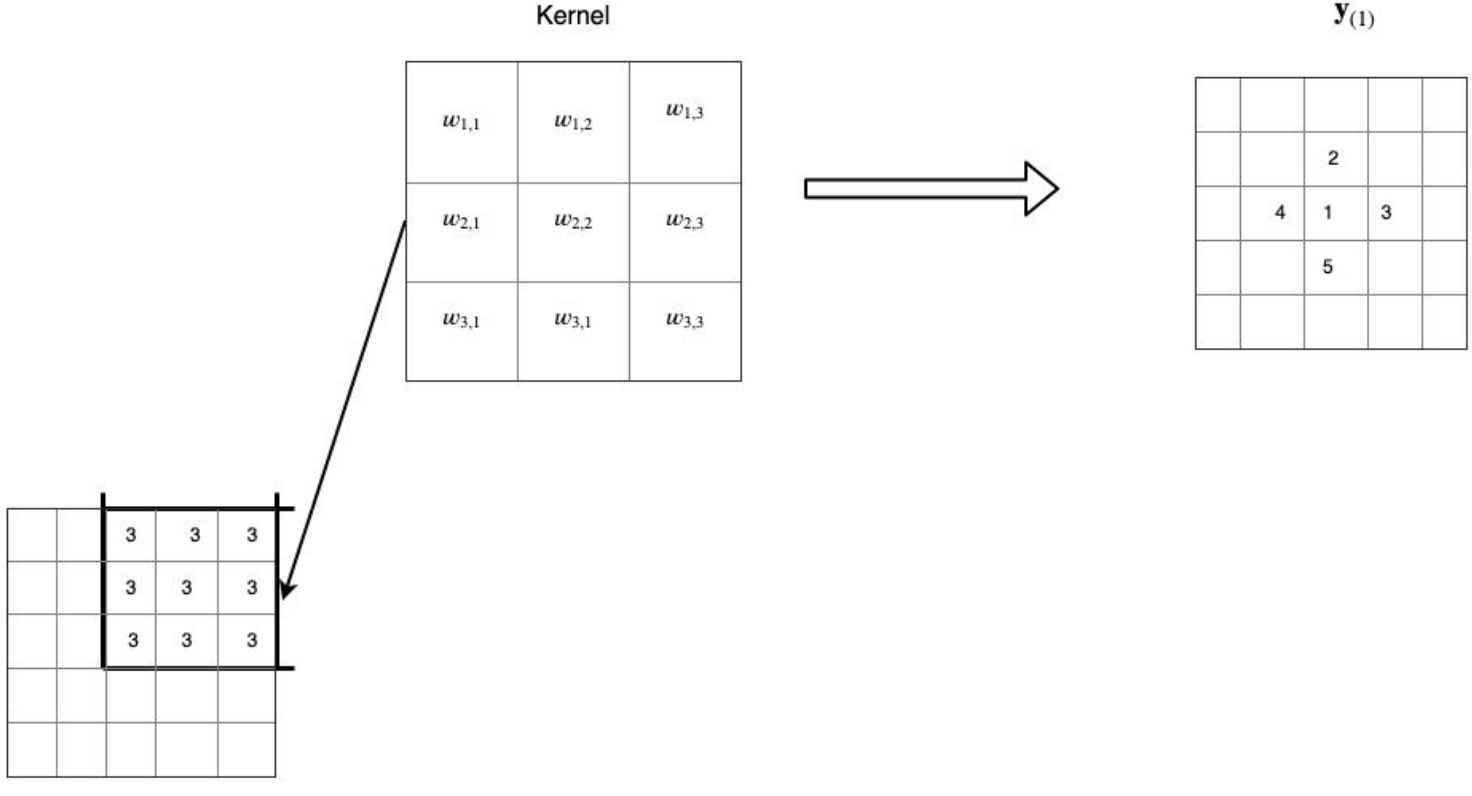
$\mathbf{y}_{(0)}$


Kernel
$\mathbf{y}_{(1)}$
2 2 2
2 2 2
2 2 2
$w_{1,1}$ $w_{1,2}$ $w_{1,3}$
$w_{2,1}$ $w_{2,2}$ $w_{2,3}$
$w_{3,1}$ $w_{3,1}$ $w_{3,3}$
2
4 1 3
5

- The dot product *using the identical kernel weights* produces output ($\mathbf{y}_{(1)}$) feature labelled 2

Repeat, centering the Kernel over each feature in $\mathbf{y}_{(0)}$:

$\mathbf{y}_{(0)}$


Kernel
$w_{1,1}$ $w_{1,2}$ $w_{1,3}$
$w_{2,1}$ $w_{2,2}$ $w_{2,3}$
$w_{3,1}$ $w_{3,1}$ $w_{3,3}$
3 3 3
3 3 3
3 3 3
$\mathbf{y}_{(1)}$
2
4 1 3
5

The output of a convolution is of similar size as the input

- detects a specific feature *at each input location*

So, for example, if there are

- three spikes: will detect the "is a spike" feature in 3 locations
- two eyes: will detect the "is an eye" feature in two locations

The convolution operation is like

- creating a small (size of template) FC layer
- that is applied to each location

So it "fully connects" *neighboring inputs* rather than the *entire input* and thus takes advantage of ordering present in the input.

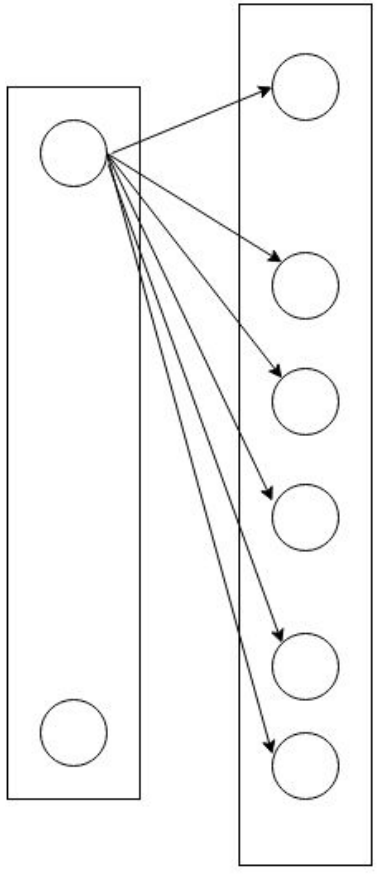
- The template is called a *kernel* or *filter*
- The output of a convolution is called a *feature map*

Pre-Deep Learning: manually specified filters have a rich history for image recognition.

Let's see some in action to get a better intuition.

Fully Connected vs Convolution

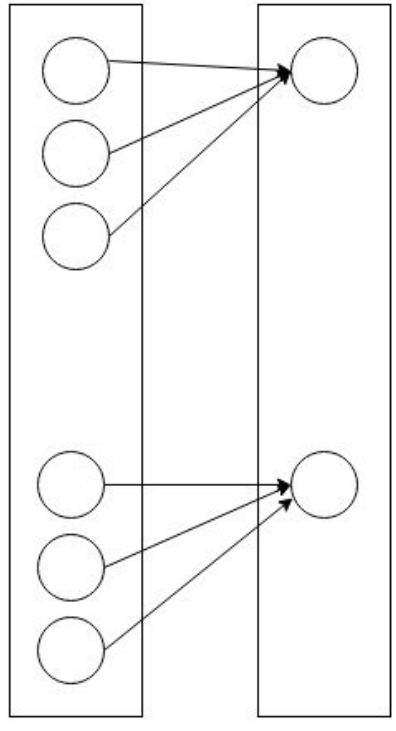
Fully Connected Layer



$n_{(l-1)}$ $n_{(l)}$

$n_{(l-1)} \times n_{(l)}$ weights

Convolutional Layer



$n_{(l-1)}$ $n_{(l)} = n_{(l-1)}$

$f = 3$ weights

- Fully Connected Layer
    - each of the $n_{(l-1)}$ units of layer $(l-1)$ connected to each of the $n_{(l)}$ units of layer (l)
    - $n_{(l-1)} * n_{(l)}$ weights total
- Convolutional Layer with 1D convolution, filter size 3
    - groups of 3 units of layer $(l-1)$ connected to *individual* units of layer (l)
    - using the *same* 3 weights
    - 3 weights total

So a Convolutional Layer can use *many fewer* weights/parameters than a Fully Connected Layer.

As we will see, this enables us to create *many* separate convolutions in a single layer

CNN advantages/disadvantages

Advantages

- Translational invariance
    - feature can be anywhere
- Locality
    - feature depends on nearby features, not the entire set of features
    - reduced number of parameters compared to a Fully Connected layer

Disadvantages

- Output feature map is roughly same size as input
    - lots of computation to compute a single output feature
        - one per feature of input map
    - higher computation cost
        - training and inference
- Translational invariance not always a positive

Multiple kernels

We have thus far seen a *single* kernel, applied to a $N = 2$ dimensional input $\mathbf{y}_{(l-1)}$.

The output $\mathbf{y}_{(l)}$ is an N dimensional feature map that identifies the presence/absence of a feature at each element of $\mathbf{y}_{(l-1)}$.

Why not use *multiple* kernels to identify *multiple* features ?

- Let Convolutional Layer l have $n_{(l),1}$ kernels
- The output is $n_{(l),1}$ feature maps, one per kernel, identifying the presence/absence of one feature each

This is similar in concept to a Fully Connected layer

- Let FC layer l have $n_{(l)}$ units/neurons
- The output is a vector of $n_{(l)}$ features

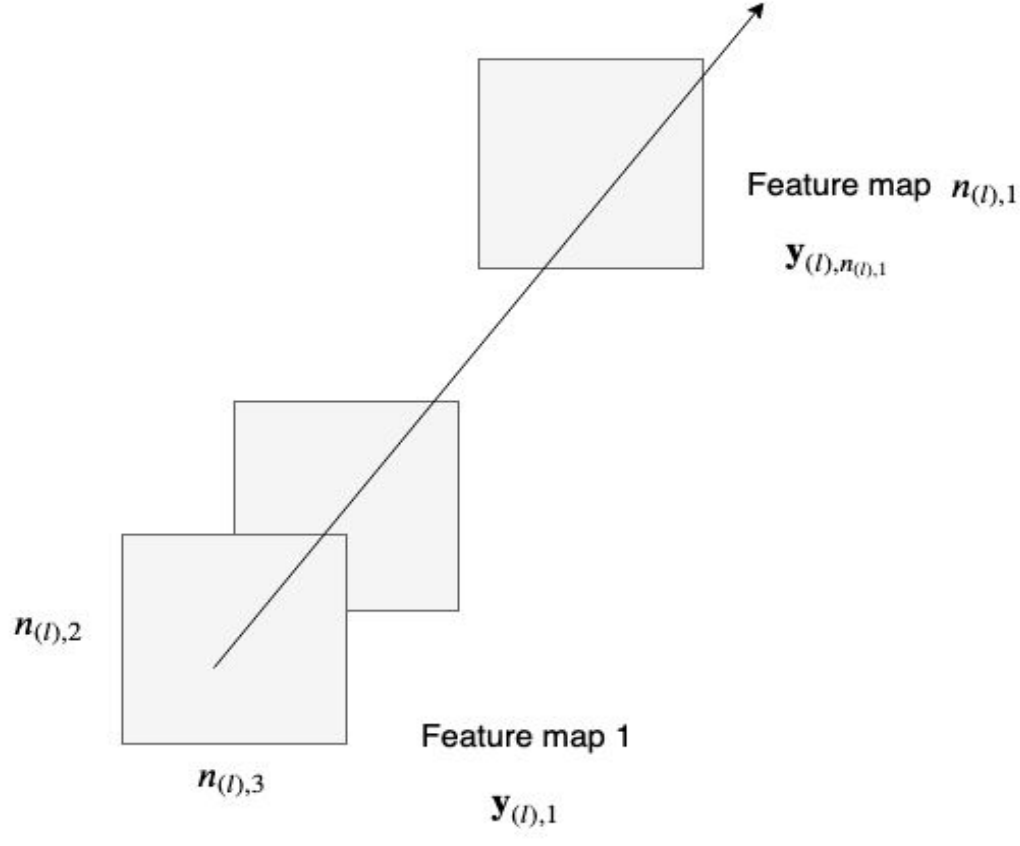
Let $\left(n_{(l-1),1} \times n_{(l-1),2}\right)$ denote the shape of $\mathbf{y}_{(l-1)}$.

If Convolutional Layer l has $n_{(l),1}$ kernels

- the output shape is $\left(n_{(l),1} \times n_{(l-1),1} \times n_{(l-1),2}\right)$

That is, the input is replicated $n_{(l)}$ times, one per kernel of layer l.

CNN convolution



It also means that the matrix $\mathbf{k}_{(l)}$ representing the kernels at layer l has the same number of dimensions as the output

- the first dimension is the number of kernels
- the rest of the dimensions are the size of each kernel

Higher dimensional input ($N > 2$)

After applying $n_{(l),1}$ kernels to $N = 2$ dimensional input $\mathbf{y}_{(l-1)}$ of shape $(n_{(l-1),1} \times n_{(l-1),2})$ we get a three dimensional output of shape $(n_{(l),1} \times n_{(l-1),1} \times n_{(l-1),2})$.

What happens when input $\mathbf{y}_{(l-1)}$ is $N = 3$ dimensional of shape $(n_{(l-1),1} \times n_{(l-1),2} \times_{(l-1),3})$?

- this can easily occur in layer $(l-1)$ is a Convolutional Layer with $n_{(l-1),1}$ kernels

If $\mathbf{y}_{(l-1)}$ is the output of a Convolutional Layer

- $\mathbf{y}_{(l-1)}$ has $n_{(l-1),1}$ features over a space of shape $\left(n_{(l-1),2} \times_{(l-1),3}\right)$

Convolutional Layer l

- combines all $n_{(l-1),1}$ input features at a single "location"
- into a new synthetic *scalar* feature at the same location in output $\mathbf{y}_{(l)}$

This means the output $\mathbf{y}_{(l)}$ is of shape $\left(n_{(l),1} \times n_{(l-1),2} \times_{(l-1),3}\right)$

This implies that *each kernel* of layer l is of dimension $N_{(l-1)}$

For example: if layer $(l-1)$ has $N_{(l-1)} = 3$ dimensions

- each kernel of layer l is has 3 dimensions

Kernel shape

We see that a Convolutional Layer l transforms

- an input of dimension $(n_{(l-1),1} \times n_{(l-1),2} \times n_{(l-1),3})$
- into output with dimension $(n_{(l),1} \times n_{(l-1),2} \times n_{(l-1),3})$

If each kernel of layer l has shape $(k_1 \times k_2)$ then the kernel matrix $\mathbf{k}_{(l)}$ for layer l

- has shape $(n_{(l),1} \times k_1 \times k_2)$
- one kernel per output synthetic feature

Receptive field

The *receptive field* of a elements of a feature map

- are the Layer 0 (input) features that affect features in the map.

For ease of notation:

- we assume $N = 2$ as the dimension of the kernel
- we assume that all N dimensions of the kernel are the same ($f_{(l)}$)

So we will assume without loss of generality that

- the "height" and "width" of a single kernel kernel is $(f \times f)$
- the full dimensionality of a single layer l kernel is $(n_{(l-1),1} \times f \times f)$

Thus the receptive field of a Convolutional Layer at layer 1 is $(f \times f)$.

Increasing the Receptive Field

There are several ways to "widen" the receptive field

- Increasing $f_{(l)}$, the size of the kernel
- Stacking Convolutional Layers
- Stride
- Pooling

Striding and Pooling also have the effect of reducing the size of the output feature map.

Size of output

We can relate the size of a layer's output feature map to the size of its input feature map:

- input $W_i \times H_i \times D_i$
- N: input size $N \times N$
- F: filter size $F \times F$
- S: stride
- P: padding

No padding

- output size $((W_i - F)/S + 1) \times ((H_i - F)/S + 1) \times D_o$

Padding

Assuming full padding, a layer l Convolutional Layer with $n_{(l),1}$ kernels will have output $\mathbf{y}_{(l)}$ dimension

- $\left(n_{(l),1} \times n_{(l-1),2} \times n_{(l-1),3}\right)$
  - $n_{(l),1}$ features
  - over a spatial map of $\left(n_{(l-1),2} \times n_{(l-1),3}\right)$

That is, the number of features changes but the spatial dimension is similar to $\mathbf{y}_{(l-1)}$

Down-sampling: Why does output size matter

Convolution applies a kernel to *each* region of the input feature map (assuming $S = 1$).

Reducing the size of feature map at layer $(l - 1)$

- will reduce the number of operations performed by Convolution at layer l.

For image inputs (with thousands or millions of input features) there is a incentive to down-sample

- Speed up training
- Speed up inference

Down-sample by

- Increasing Stride
- Pooling

Number of parameters

The real power of convolution comes from using the *same* filter against all locations of the input.

As a result, the number of parameters is quite small (compared to a separate set of parameters per each input).

- Dimension of a single filter $F \times F \times D_i$
- D_o: number of output filters
- total parameters: $F * F * D_i * D_o$

Remember: there is a depth to the input and the filter applies to the entire input depth

- size of a filter $F * F * D_i$
- number of filters: D_o, one per output channel
- total: $F * F * D_i * D_o$

If we were to have a separate filter for each input location, the number of parameters would increase by a factor of $W_i * H_i$.

In [4]:
```
print("Done")
```

```
Done
```