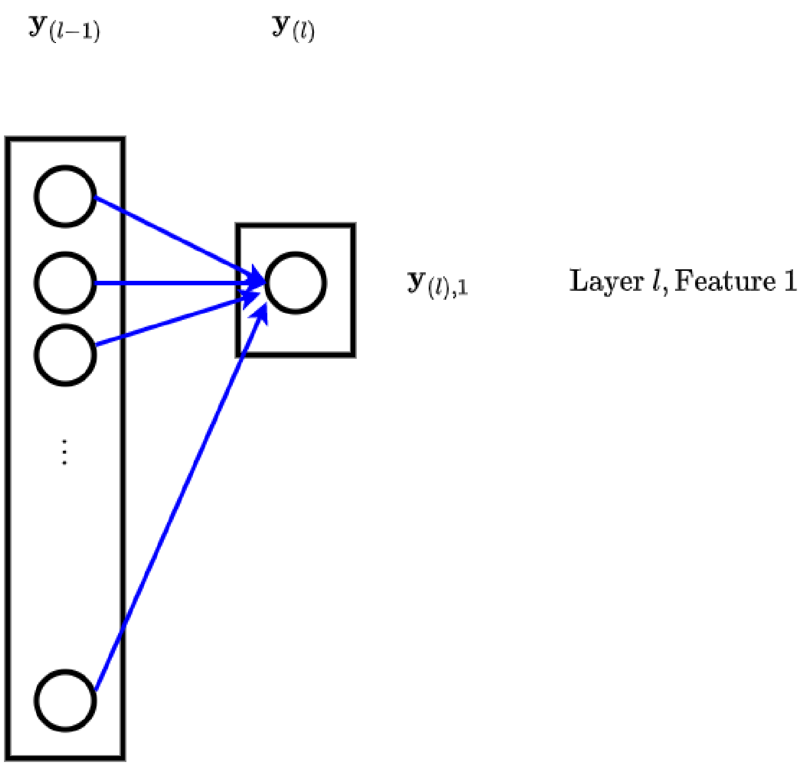
Convolutional Neural Networks

A Fully Connected/Dense Layer with a single unit producing a single feature at layer l computes

$$\mathbf{y}_{(l),1} = a_{(l)}(\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l),1})$$

Fully connected, single feature



That is:

- It recognizes one new synthetic feature
- In the entirety ("fully" connected) of $\mathbf{y}_{(l-1)}$
- Using pattern $\mathbf{W}_{(l),1}$ (same size as $\mathbf{y}_{(l-1)}$)
- To reduce $\mathbf{y}_{(l-1)}$ to a single feature.

The pattern being matched spans the entirety of the input

- Might it be useful to recognize a smaller feature that spanned only *part* of the input ?
- What if this smaller feature could occur *anywhere* in the input rather than at a fixed location ?

For example

- A "spike" in a time series
- The eye in a face

A pattern whose length was that of the entire input could recognize the smaller feature only in a *specific* place

This motivates some of the key ideas behind a Convolutional Layer.

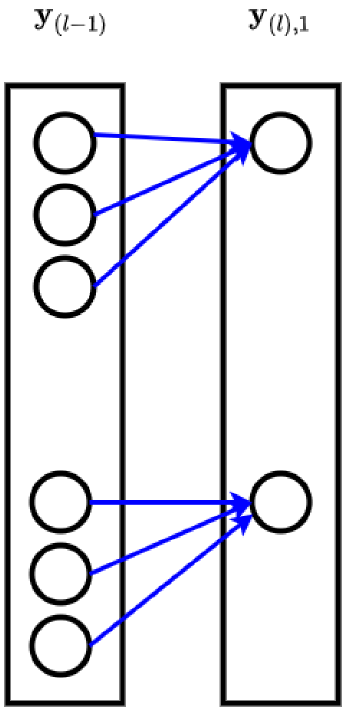
- Recognize smaller features within the whole
- Using small patterns
- That are "slid" over the entire input
- Localizing the specific part of the input containing the smaller feature

The spatial dimension

Here is the connectivity diagram of a Convolutional Layer producing a **single** feature at layer l

- Using a pattern of length 3
- Eventually we will show how to produce *multiple* features
- Hence the subscript "1" in $\mathbf{y}_{(l),1}$ to denote the first output feature
- The output $\mathbf{y}_{(l),1}$ is called a *feature map* as it attempts to match a feature at each input location

Convolutional layer, single feature



We really need to make the shapes of the vectors more precise.

- The vectors depicted now have 2 (or more) dimensions
- In our case: there are 2 dimensions, one of them a singleton
- The final dimension is the *feature* dimension

In the above diagram, layers $(l-1)$ and l have dimensions are $(d_{(l)} \times 1)$

- a single feature
- at $d_{(l)} = d_{(l-1)}$ *spatial* locations

This is different than the vector of shape $(1 \times d_{(l)})$

- (Thus far, we seemingly have been equating $d_{(l)} = n_{(l)}$)
- $d_{(l)} = d_{(l-1)}$ features
- at a single spatial location

The choice of where the singleton dimension appears is sometimes a matter of interpretation.

Consider the time series of prices of a single ticker over d days.

Two representations

- $(d \times 1)$: 1 feature ("price") over d spatial ("date") locations
- $(1 \times d)$: 1 ticker with d features $(\text{price}\, 1, \ldots, \text{price}\, d)$

Note that a convolution finds small patterns in the spatial dimension, not the feature dimension

Your choice of where to place the singleton dimension thus has consequences for a Convolutional layer.

Notation

- the feature dimension will be the last index
- $n_{(l)}$ will always denote the *number of features* of a layer l
- $\mathbf{y}_{(l),,j',j}$ denotes feature j of layer l at spatial location j'

We say that the above convolutional layer l

- Maps a single feature (defined over $d_{(l)} = d_{(l-1)}$ locations) of layer $(l-1)$
- To a single feature, defined over an identical number of spatial locations in layer l

The Fully Connected layer we depicted matches a pattern over the full *feature* dimension

- There is no ordering (or spatial relationship) between features

To see this,

- Consider a vector $\mathbf{x}$ of n features (input to the Fully Connected layer)
- Let perm be permutation of the indices of $\mathbf{x}$: $[1 \ldots n]$.

If we permute both $\mathbf{x}$ and weights Θ, the dot product remains unchanged

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

But for certain types of inputs (e.g. images) it is easy to imagine that spatial locality is important.

By using a small pattern (and restricting connectivity)

- we **emphasize the importance of neighboring features over far away features**.

Mathematically, the One Dimensional Convolutional Layer (Conv1d) we have shown computes $\mathbf{y}_{(l)}$

$$\mathbf{y}_{(l),1} = \begin{pmatrix} a_{(l)} \left(\; N(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l),1}, 1) \cdot \mathbf{W}_{(l),1} \; \right) \\ a_{(l)} \left(\; N(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l),1}, 2) \cdot \mathbf{W}_{(l),1} \; \right) \\ \vdots \\ a_{(l)} \left(\; N(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l),1}, n_{(l-1)} \cdot \mathbf{W}_{(l),1} \; \right) \end{pmatrix}$$

where $N(\; \mathbf{y}_{(l-1)}, \mathbf{W}_{(l),1}, j \;)$

- selects a subsequence of $\mathbf{y}_{(l-1),\ldots,1}$ centered at $\mathbf{y}_{(l-1),j,1}$
  - Note the extra spatial dimension in the subscripting; "$\ldots$" denotes the full spatial dimension
  - Centered at the j^{th} element in the spatial dimension of feature 1 of layer $(l-1)$

Note that

- The *same* weight matrix $\mathbf{W}_{(l),1}$ is used for the first feature at *all* locations j
- The size of $\mathbf{W}_{(l),1}$ is the same as the size of the subsequence $N(\; \mathbf{y}_{(l-1)}, \mathbf{W}_{(l),1}, j)$
  - Since dot product is element-wise multiplication

So $\mathbf{W}_{(l),1}$

- Is a smaller pattern
- That is applied to *each* spatial location j in $\mathbf{y}_{(l-1)}$
- $\mathbf{y}_{(l),j,1}$ recognizes the match/non-match of the smaller first feature at the spatial locations centered at $\mathbf{y}_{(l-1),j,1}$

$\mathbf{W}_{(l),1}$ is called a convolutional *filter* or *kernel*

- We will often denote it $\mathbf{k}_{(l),1}$
- But it is just a part of the weights $\mathbf{W}$ of the multi-layer NN.
- We use $f_{(l)}$ to denote the size of the smaller pattern called the *filter size*

Note

The default activation $a_{(l)}$ in Keras is "linear"

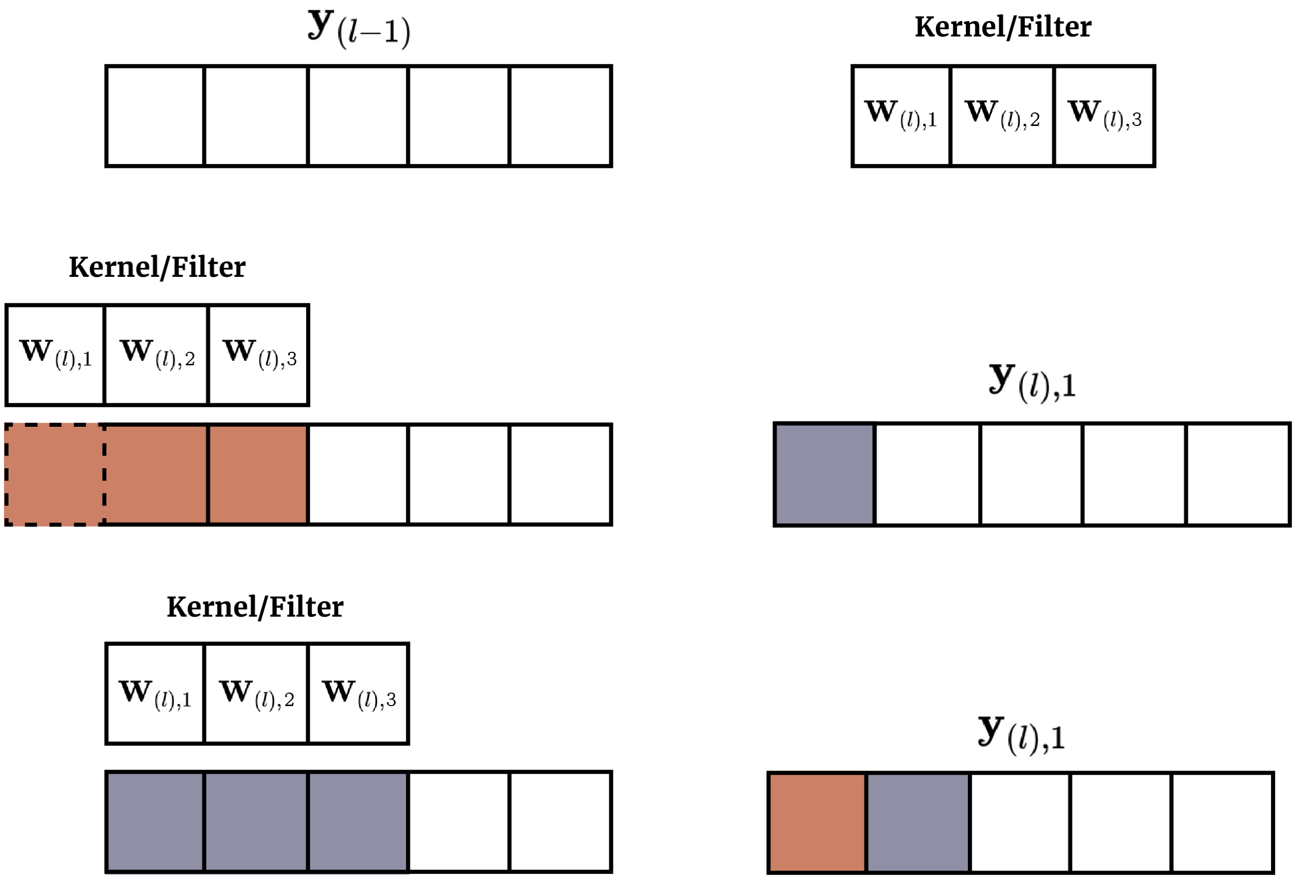
- That is: it returns the dot product input unchanged
- Always know what is the default activation for a layer; better yet: always specify !

A *Convolution* is often depicted as

- A filter/kernel
- That is slid over each location in the input
- Producing a corresponding output for that location

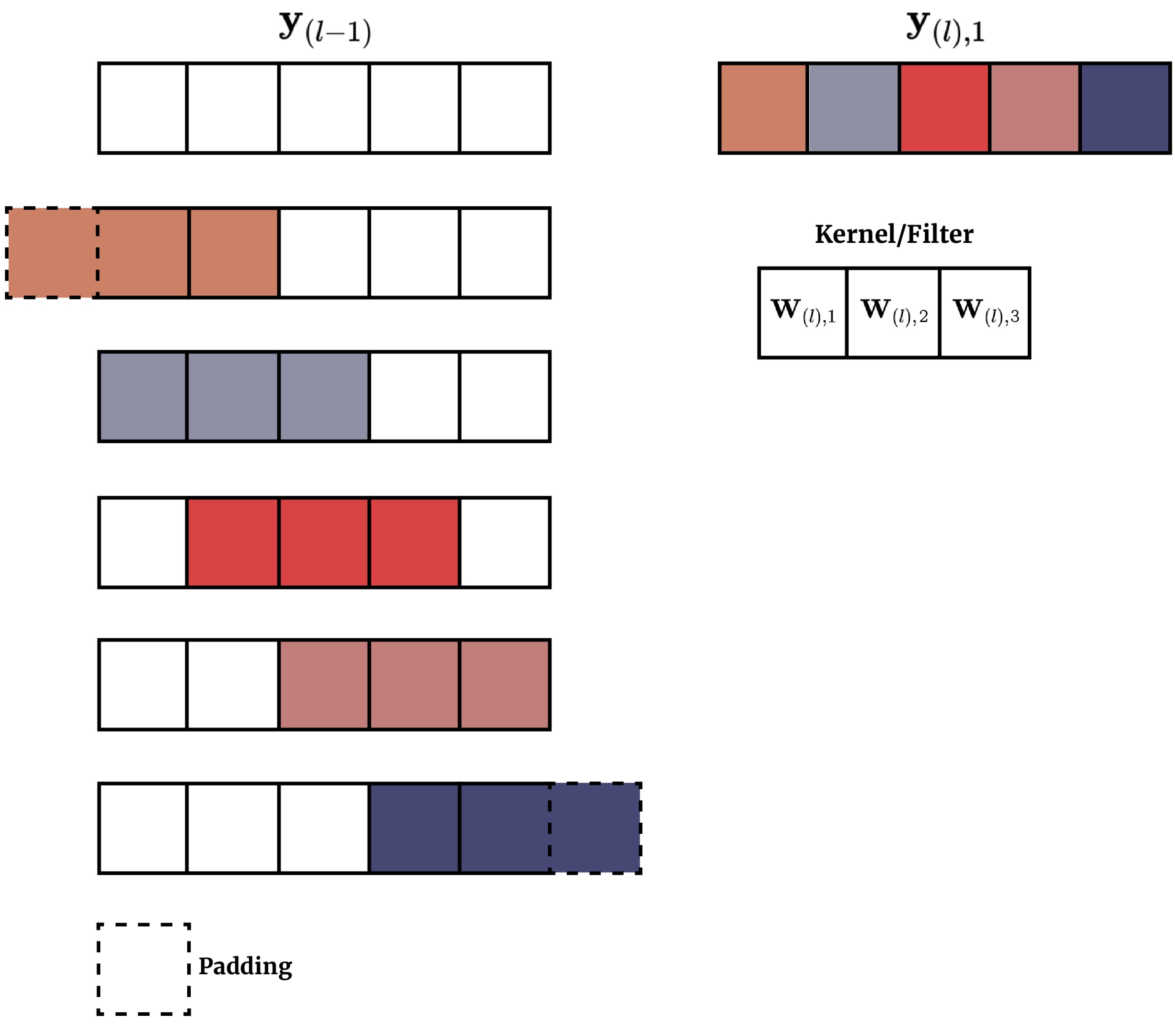
Here's a picture with a kernel of size $f_{(l)} = 3$

Conv 1D, single feature: sliding the filter



After sliding the Kernel over the whole $\mathbf{y}_{(l-1)}$ we get:

Conv 1D, single feature



Element j of output $\mathbf{y}_{(l),\ldots,1}$ (i.e., $\mathbf{y}_{(l),j,1}$)

- Is colored (e.g., $j = 1$ is colored Red)
- Is computed by applying the *same* $\mathbf{W}_{(l),1}$ to
    - The $f_{(l)}$ elements of $\mathbf{y}_{(l-1),1}$, centered at $\mathbf{y}_{(l-1),j,1}$
    - Which have the same color as the output

Note however that, at the "ends" of $\mathbf{y}_{(l-1)}$ the kernel may extend beyond the input vector.

In that case $\mathbf{y}_{(l-1)}$ may be extended with *padding* (elements with 0 value typically)

Conv2d in action

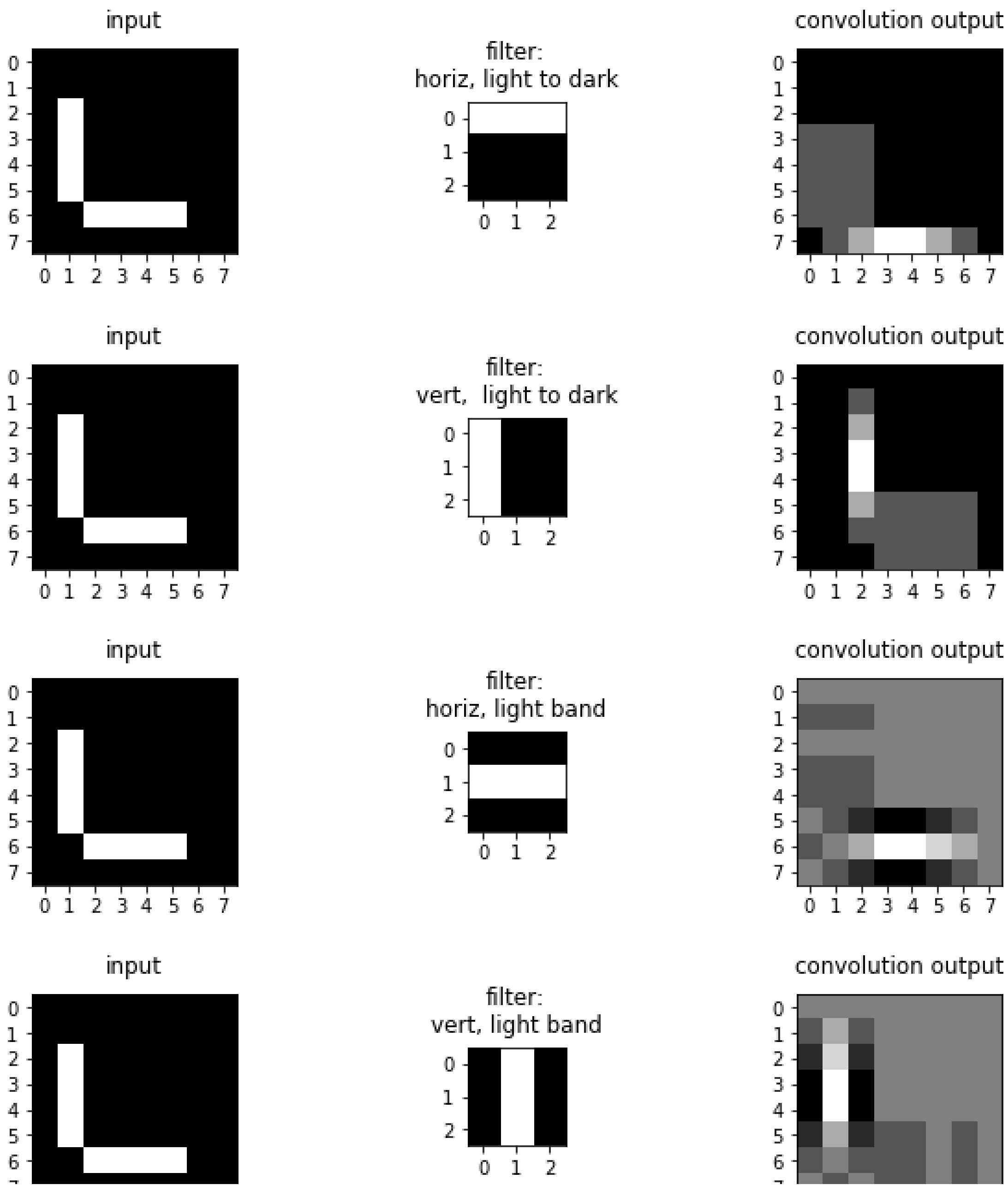
Pre-Deep Learning: manually specified filters have a rich history for image recognition.

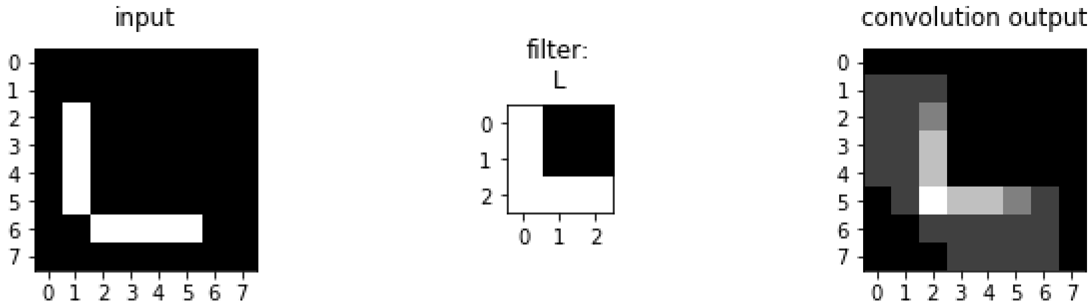
Here is a list of manually constructed kernels (templates) that have proven useful

- list of filter matrices (https://en.wikipedia.org/wiki/Kernel_(image_processing))

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

In [4]:
```
_= cnnh.plot_convs()
```




input
filter:
L
convolution output

- A bright element in the output indicates a high, positive dot product
- A dark element in the output indicates a low (or highly negative) dot product

In our example

- $N = 2$: Two spatial dimensions
- One input feature: $n_{(l-1)} = 1$
- One output feature $n_{(l)} = 1$
- $f_{(l)} = 3$
    - Kernel is $(3 \times 3 \times 1)$.

The template match will be maximized when

- high values in the input correspond to high values in the matching location of the template
- low values in the input correspond to low values in the matching locations of the template

In [5]:

```
print("Done")
```

Done