Convolutional Layers: Space and Time

In our introductory examples

- The spatial dimension of output $\mathbf{y}_{(l)}$
- Is identical to the spatial dimension of input $\mathbf{y}_{(l-1)}$

There are different choices we can make when "sliding" the kernel over the input.

These choices impact

- The spatial dimension of the output
- And, in turn, the time requirements of subsequent layers (because of the size)

Let's do some quick calculations and then show choices for controlling the space consumed by $y_{(l)}$.

CNN Math: Time versus number of parameters

Consider input layer $(l-1)$ with

- N spatial dimensions
- $n_{(l-1)}$ feature maps/channels
  $||\mathbf{y}_{(l-1)}|| = (d_{(l-1),1} \times d_{(l-1),2} \times \ldots d_{(l-1),N} \times n_{(l-1)})$

Layer l will apply a Convolution that preserves the spatial dimensions

$$||\mathbf{y}_{(l)}|| = (d_{(l-1),1} \times d_{(l-1),2} \times \ldots d_{(l-1),N} \times n_{(l)})$$

For simplicity of presentation: consider the case when $N = 2$.

How many weights/parameters does layer l consume (i.e, what is size of $\mathbf{W}_{(l)}$) ?

- Each kernel $\mathbf{k}_{(l),j}$
    - Has spatial dimension $(f_{(l)} \times f_{(l)})$
    - And "depth" $n_{(l-1)}$ (to match the number of input feature maps/channels)
- There are $n_{(l)}$ kernels in layer l

So the size of $W_{(l)}$ (ignoring the optional bias term per output feature map)

$$||\mathbf{W}_{(l)}|| = n_{(l)} * (n_{(l-1)} * f_{(l)} * f_{(l)})$$

The part of the product that most concerns us is ($n_{(l)} * n_{(l-1)}$)

- Values for $n_{(l)}, n_{(l-1)}$ in $\{32, 64, 256\}$ are not uncommon !
- Hence $||\mathbf{W}_{(l)}||$ is often easily several thousand
- State of the art image recognition models use *several hundred million* weights !

How many multiplications (in the dot product) are required for layer l ?

- We will ignore additions (the part of the dot product that reduces pair-wise products to a scalar, and for the bias)
- Each kernel $\mathbf{k}_{(l),j}$ of dimension
$$(f_{(l)} \times f_{(l)} \times n_{(l-1)})$$
- Applied over each location in the $(d_{(l-1),1} \times d_{(l-1),2})$ spatial dimension of the input layer $(l-1)$
- There are $n_{(l)}$ kernels in layer l

So the number of multiplications

$$n_{(l)} * (d_{(l-1),1} * d_{(l-1),2}) * (n_{(l-1)} * f_{(l)} * f_{(l)})$$

Consider a grey-scale image of size $(d_{(l-1),1} * d_{(l-1),2}) = (1024 \times 1024)$

- Lower than your cell-phones camera !
- Easily several *million* multiplications

Expect the time to train a Neural Network with Convolutional layers to be long !

- That's why GPU's are important in training
- But GPU's have limited memory so space is important too
    - Can control with batch size

All of this ignores the final layer L

- Often a Fully Connected layer implementing Regression or Classification
- With n_L output features
    - e.g., For Classification over classes in set C, $y_{(L)}$ is a One Hot Vector of length $n_L = ||C||$

Suppose layer $(L-1)$ has dimension

$$||\mathbf{y}_{(L-1)}|| = (d_{(L-1),1} \times d_{(L-1),2} \times n_{(L-1)})$$

Before we can use it as input to the Fully Connected Layer L we flatten it to a vector of length

$$(d_{(L-1),1} * d_{(L-1),2} * n_{(L-1)})$$

The number of weights (ignoring biases) and multiplications is

$$||W_L|| = n_{(L)} * (d_{(L-1),1} * d_{(L-1),2} * n_{(L-1)})$$

- $n_{(L)} * n_{(L-1)}$ on the order of several thousand
- $\left(d_{(L-1),1} * d_{(L-1),2}\right)$ on the order of several million, for images

This may not even be feasible !

Thus, controlling the size of each layer $y_{(l)}$ is of great *practical* importance.

Controlling the output spatial dimensions

Padding

In our examples thus far

- When a location in the spatial dimensions of the input
- Is such that, when the kernel is placed there, it extends beyond the input
- We have added "padding"

This is not strictly necessary

- But has advantage that the spatial dimension of output $\mathbf{y}_{(l)}$ is the same as the input $\mathbf{y}_{(l-1)}$
- One can simply *not* produce an output for such locations
- It just means the output spatial dimension shrinks in each dimension by $f_{(l)} - 1$
    - Assuming $f_{(l)}$ is odd
    - The number of locations in which the kernel extends over the border
    - Is Half of the filter size $(f_{(l)} - 1)/2$ times two (for each edge)

Stride

Thus far, we have placed the kernel over *each* location in the spatial dimension of the input layer.

This, along with padding, ensures that the spatial dimension of the input and output layers are identical.

Consider two adjacent locations in the spatial dimension of the input layer

- The values of the input layer that appear in each dot product overlap

By placing the kernel over *every other* location of the spatial dimension of the input layer

- We may still be able to recognize features
- And reduce the spatial dimension of the output layer by a factor of 2 for each dimension.

In general, we can choose to choose to pass over $(S-1)$ locations in the spatial dimension of the input layer

- S is called the *stride*
- Up until now: $S = 1$
- But you are free to choose

Size of output

We can combine choices of Padding and Stride to control the spatial dimension of the output layer l:

Let

- $d_{(l-1),j}$ denote the number of elements in spatial dimension j of layer $(l-1)$
- P denote the number of elements added as padding on each border
- S denote the stride
- $f_{(l)}$ be the size of the filter (for each spatial dimension)

Then the number of elements in spatial dimension j of output layer (l) is

$$d_{(l),j} = \frac{d_{(l-1),j} + 2P - f_{(l)}}{S} + 1$$

You can see that increasing the stride has the biggest impact on reducing the spatial dimension of the output.

Pooling layer

There is a layer type with the specific purpose of changing the spatial dimension of the output.

This is called a Pooling Layer.

A Pooling Layer combines the information from adjacent locations in the spatial dimension of the input layer.

- The "combining" operation may be average or maximum
- Sacrificing the exact location in the spatial dimension
- Often in exchange for reduced space

A Pooling Layer is similar in *some* respects to a Convolution.

Recall that the One Dimensional Convolutional Layer (Conv1d) with a single input feature computes the following for output feature/channel j:

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

The analogous One Dimensional Pooling Layer (Pooling1D) computes

$$\mathbf{y}_{(l),j} = \begin{pmatrix} p_{(l)} \left(N'(\mathbf{y}_{(l-1)}, f_{(l)}, 1) \right) \\ p_{(l)} \left(N'(\mathbf{y}_{(l-1)}, f_{(l)}, 2) \right) \\ \vdots \\ p_{(l)} \left(N'(\mathbf{y}_{(l-1)}, f_{(l)}, n_{(l-1)}) \right) \end{pmatrix}$$

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

- selects a subsequence of $\mathbf{y}_{(l-1)}$ centered at $\mathbf{y}_{(l-1),j}$
- of length $f_{(l)}$

and $p_{(l)}$ is a *pooling operation*

That is, similar to a Convolutional Layer, the Pooling Layer

- Selects a region of length $f_{(l)}$
- Centered at each location in the spatial dimension of the input layer $(l-1)$

and produces a value in the corresponding spatial location of output layer l

- That *summarizes* the selected region

Observe that

- There are *no* weights
- No dot product
- Just a pooling operation

Similar to Convolution, we can extend pooling to higher spatial dimension ($N > 1$) and higher number of input channels $n_{(l-1)} > 1$.

Suppose the input $\mathbf{y}_{(l-1)}$ is $(N+1)$ dimensional of shape

$$||\mathbf{y}_{(l-1)}|| = (d_{(l-1),1} \times d_{(l-1),2} \times \dots d_{(l-1),N} \times n_{(l-1)})$$

Pooling:

- Selects an N-dimensional region, where each dimension is of length $f_{(l)}$
- Centered at each location in the spatial dimension
    - Of a **single feature map j** of the input layer $(l-1)$: $\mathbf{y}_{(l-1),\ldots,j}$

and produces a value in the corresponding spatial location of output layer l

- That *summarizes* the selected region by applying $p_{(l)}$ to the selected region

Pooling with a stride $S > 1$

- "Down samples" the spatial dimension
- Sacrificing some information about locality

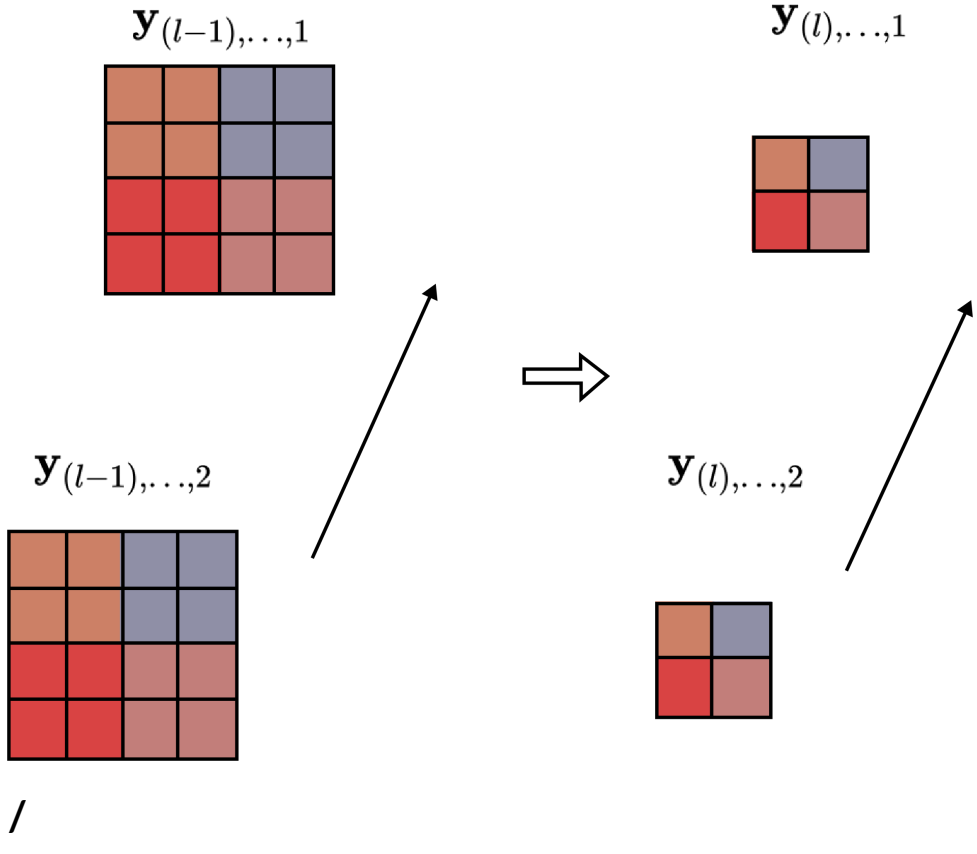
It effectively asks the question

- Does the feature exist in a broader neighborhood of the spatial dimension

Here is a two dimensional example with a filter size and stride of 2:

- $N = 2$
- $f_{(l)} = 2$
- $S = 2$

Conv 2D: Pooling (Max/Average)



The key difference between Pooling and Convolution (other than the absence of the dot product and kernel weights)

- **The pooling operation is applied to each input feature map *separately***
- **Versus *all the input feature maps* at a given location in the spatial dimension of the input**

Pooling operations

- **Max pooling**
    - **Maximum over the selected region**
    - **Good for answering the question: "Does the feature exist" in the neighborhood**
- **Average pooling**
    - **average over the selected region**
    - **"blurs" the location in the spatial dimension when it is unimportant or highly variable**

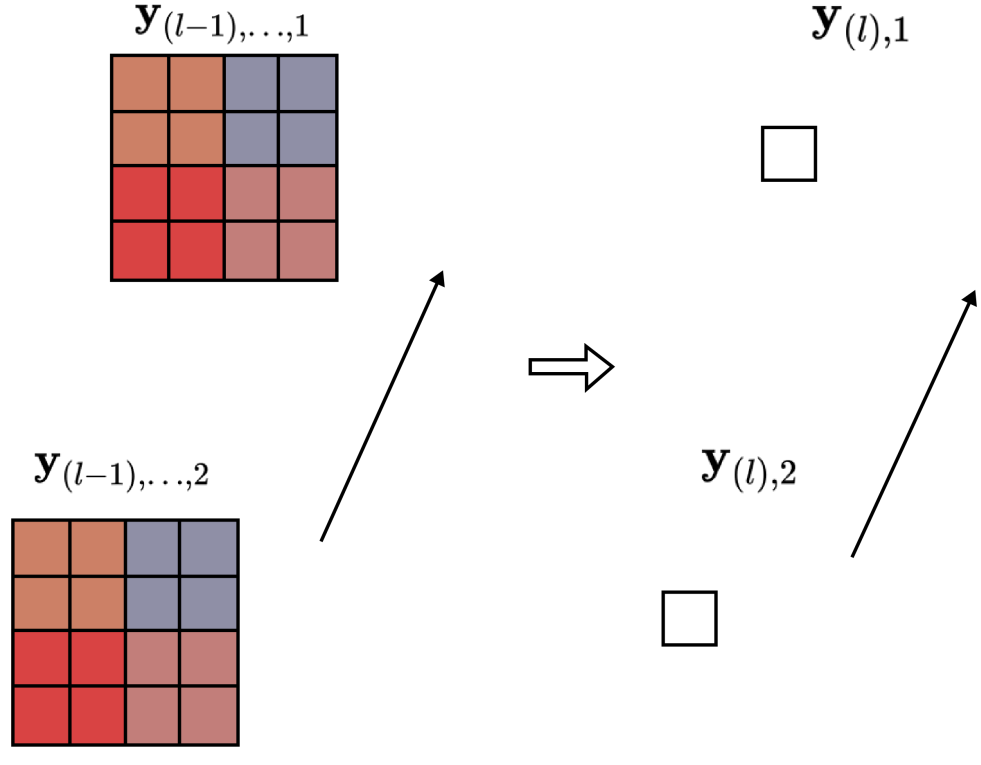
Global Pooling

Each **feature map j of the input layer $(\mathbf{y}_{(l-1),\ldots,j})$**

- **Is summarized by a single value produced by Max Pooling operation $p'_{(l)}$**

$$\mathbf{y}_{(l),j} = p'_{(l)}(\mathbf{y}_{(l-1),\ldots,j})$$

Conv 2D: Global Pooling (Max/Average)



Notice that each input feature map has been reduced to a single value in the output.

- **No spatial dimension in $\mathbf{y}_{(l)}$ (hence no "...")**

The Global Pooling operation effectively asks the question

- **Does the feature occur *anywhere* in the feature map ?**
- **Losing information about the exact location in the spatial dimensions**

Global pooling operations

- Global average pooling
    - Maximum over the feature map
- K-Max pooling
    - replace one dimension of the volume with the K largest elements of the dimension

Review

Let's summarize our knowledge of controlling the size of $\mathbf{y}_{(l-1)}$:

- Controlling spatial dimensions
    - Increase stride
    - Pooling
        - Global average pooling often used in final Convolutional Layer
- Control number of feature maps per layer
    - Choice of $n_{(l),1}$
    - Kernel size $f_{(l)} = 1$
        - preserve spatial dimension
        - change number of feature maps from $n_{(l-1),1}$ to $n_{(l),1}$

Striding and Pooling

- **increase receptive field**
- **typically small values (e.g., $S = 2$)**
    - **limited reduction**

Kernel size $f_{(l)} = 1$

- **reduction depends on the ratio of $n_{(l),1}$ to $n_{(l-1),1}$**
    - **unlimited reduction possible**

Kernel size 1

A less obvious way to control the size of $\mathbf{y}_{(l)}$ is to use a kernel with $f_{(l)} = 1$

Why might that be ?

Recall that a Convolutional Layer

- **Preserves the spatial dimension**
- **Replaces the channel/feature dimension (number of feature maps)**

That is

$$
\begin{aligned}
||\mathbf{y}_{(l-1)}|| &= (\boldsymbol{n}_{(l-1),1} \times \boldsymbol{n}_{(l-1),2} \times \ldots \boldsymbol{n}_{(l-1),N}, \quad \mathbf{n}_{(\mathrm{l}-1)}) \\
||\mathbf{y}_{(l)}|| &= (\boldsymbol{n}_{(l-1),1} \times \boldsymbol{n}_{(l-1),2} \times \ldots \boldsymbol{n}_{(l-1),N}, \quad \mathbf{n}_{(\mathrm{l})})
\end{aligned}
$$

So a kernel of size $\boldsymbol{f}_{(l)} = 1$ in all N spatial dimensions

- With "depth" $\boldsymbol{n}_{(l-1)}$
- Is just a way to resize $\mathbf{y}_{(l-1)}$ from $\boldsymbol{n}_{(l-1)}$ feature maps to a *single* feature map
    - That sums, across feature maps, the elements in each feature map at the same spatial location

In other words:

- Yet another way to reduce the size of $\mathbf{y}_{(l)}$.

Receptive field

The filter size $f_{(l)}$ also plays a role in the space and time requirements of a Convolutional Layer.

It turns out that

- **We can achieve the effect of a large $f_{(l)}$**
- **With a smaller $f_{(l)}$ in conjunction with *more* Convolutional Layers**

The *receptive field* of a layer l feature at a single spatial location in feature map k

- **are the spatial locations of Layer 0 (input) features that affect this single feature**

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 ($f_{(l)}$)**

We can determine spatial locations of the layer 0 features influencing a single layer l location by working backwards from layer l

- **As we will demonstrate shortly**

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 $(f \times f \times n_{(l-1)})$**

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.

Increase the size of the kernel

Although this is the most *obvious* way of increasing the receptive field, we tend to avoid it !

We will see that adding layers is a more efficient way of achieving a bigger receptive field.

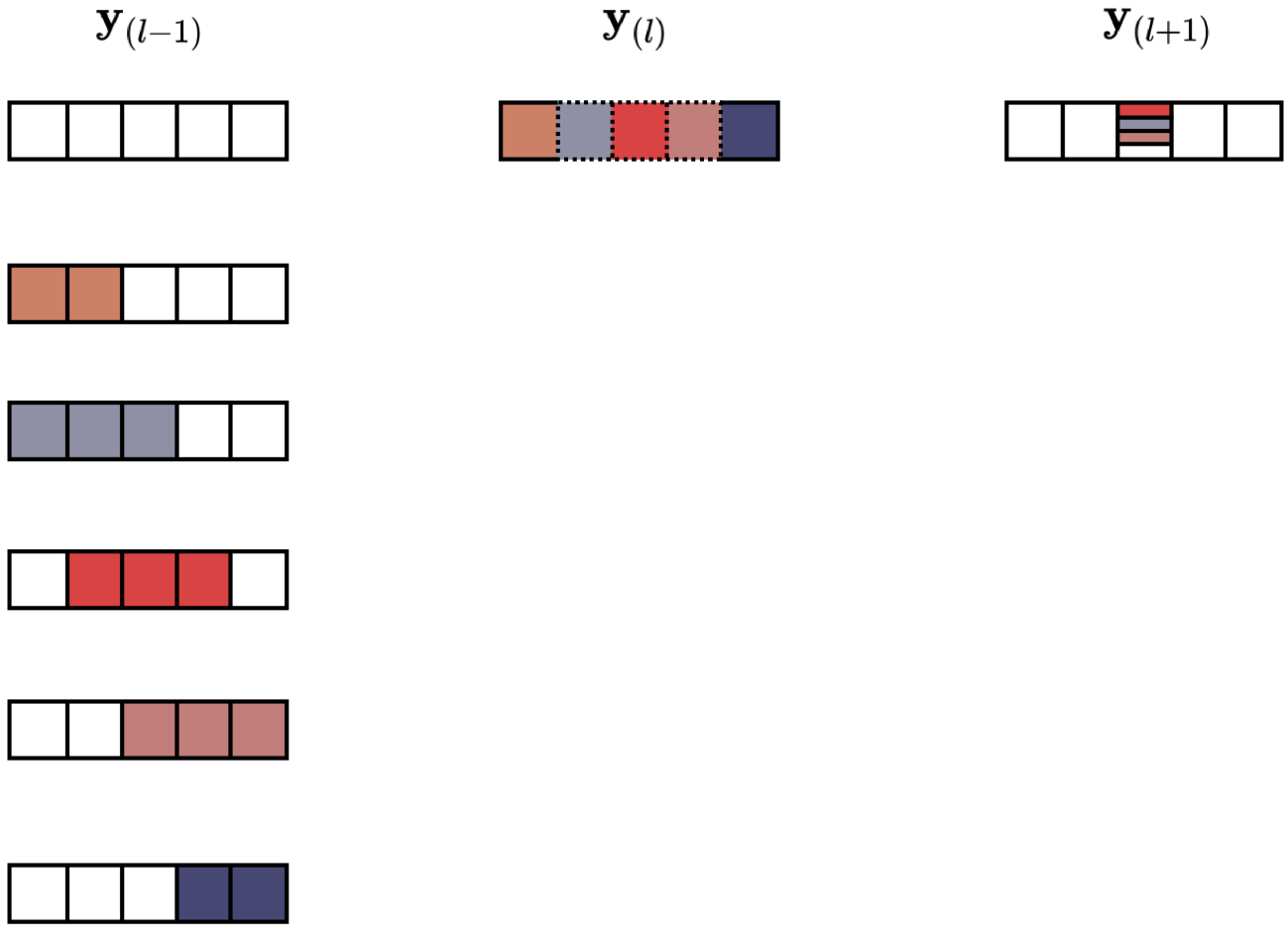
Stacking Convolutional Layers

Let's introduce the idea of stacking multiple Convolutional Layers using $N = 1$.

Consider

- $N = 1$
- $f_{(l)} = f_{(l-1)} = 3$
- $(L - 1) = 2$: **Two Convolutional Layers**

Conv 1D Receptive field: 2 layers



The elements in $\mathbf{y}_{(l)}$

- Are colored
- The same color as the elements of $\mathbf{y}_{(l-1)}$ that they depend on

Each element of layer l depends on $f_{(l)} = 3$ elements of layer $(l-1)$.

Consider the element in the center of the second layer: $\boldsymbol{y}_{(l+1)}$, i.e., $\boldsymbol{y}_{(l+1),3}$

- **It depends on the Orange, Green, and Blue elements of $\mathbf{y}_{(l)}$**
- **Which in turn depend on the Orange, Green and Blue elements of $\mathbf{y}_{(l-1)}$**
- **This includes $\mathbf{5}$ elements of $\mathbf{y}_{(l-1)}$**

So the Two layer network with $f_{(l)} = f_{(l-1)} = 3$

- Is exposed to the *same* layer $(l-1)$ elements
- As a Single Convolutional Layer with $f_{(l)} = f_{(l-1)} = 5$

One can trace an element in layer $l+1$

- **Backwards through layers**
- **To input layer 0**

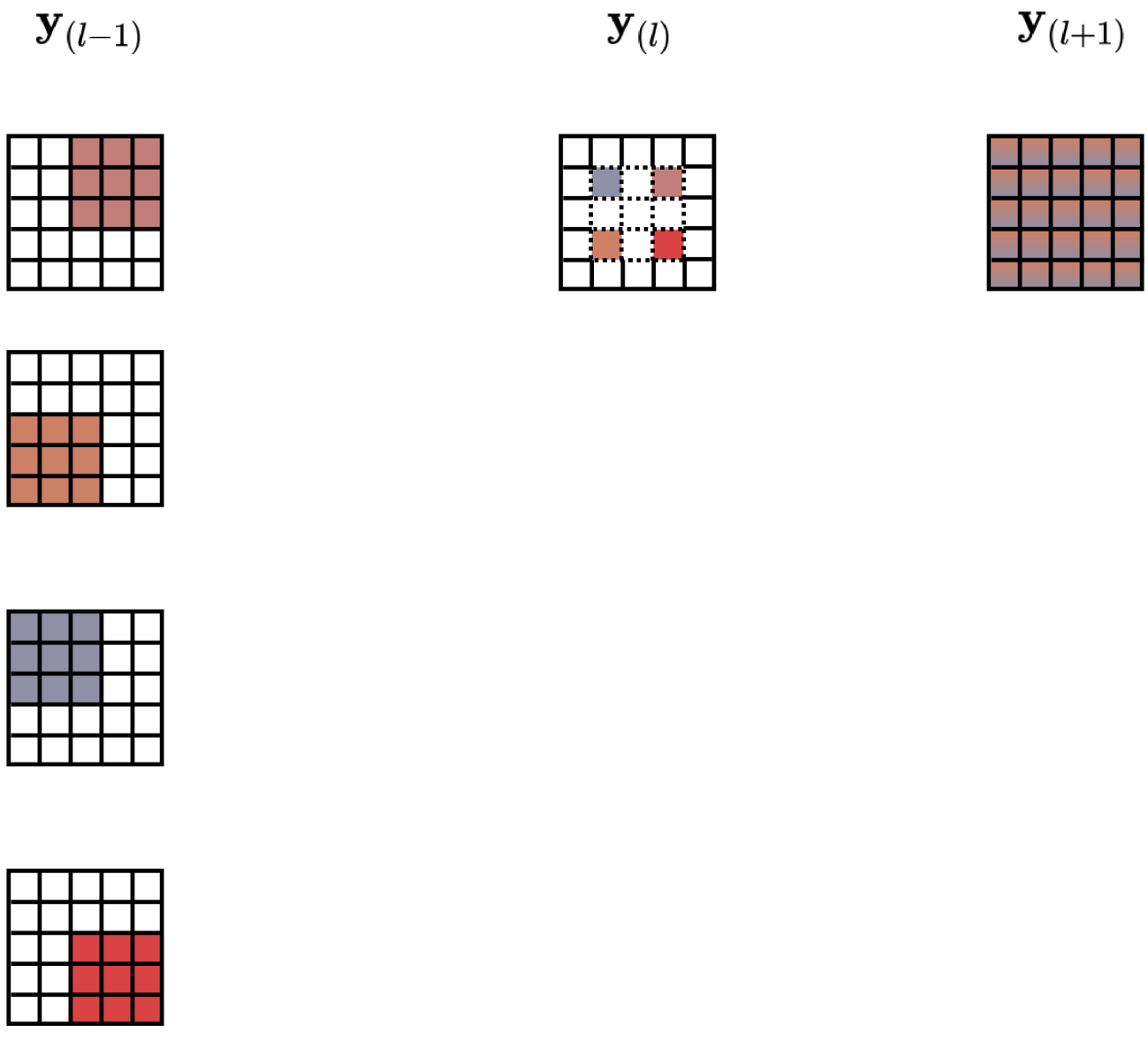
in order to determine the receptive field of layer $l+1$/

Bottom line: The size of the receptive field increases with the depth of the layer.

One can apply similar logic to $N = 2$ spatial dimensions.

As you go one layer deeper in the NN, the receptive field width and height increase by (2 stride*)

Conv 2D Receptive field: 2 layers


$\mathbf{y}_{(l-1)}$
$\mathbf{y}_{(l)}$
$\mathbf{y}_{(l+1)}$

Receptive field

- **The spatial locations in layer l**
- **Are color coded to match the spatial locations in layer $(l-1)$**
- **That affect it**

So the yellow location in layer l is a function of the yellow locations in layer $(l-1)$

The central location in layer $(l+1)$

- **Is a function of the spatial locations in layer l that are encircled by the dashed square**
- **The layer l locations are a function of all the layer $(l-1)$ locations**

So the receptive field for the central location in layer $(l+1)$

- **Includes all the locations of layer $(l-1)$**

In other words: the size of the receptive field grows with layer depth.,

Layer	Receptive field
1	(3×3)
2	(5×5)
3	(7×7)
$\vdots$	$\vdots$

Let's compare

- **The math of 2 layers with $f_{(l)} = f_{(l-1)} = 3$**
- **The math of 1 layer with $f_{(l)} = 5$**

In terms of number of weights:

- The one layer network uses

$$
\begin{aligned}
||\mathbf{W}|| &= n_{(l)} * (n_{(l-1)} * f'(l) * f'(l)) \\
&= 25 * n_{(l)} * n_{(l-1)} \qquad \text{when } f'(l) = 5
\end{aligned}
$$

- The two layer network uses $$ \begin{array}{lll} || \W\llp || & = & n\llp (n_{(\ll-1)} f\llp * f\llp) \\ || \W{(\ll+1)} || & = & n{(\ll+1)} (n_ \llp f{(\ll+1)} * f{(\ll+1)}) \\ || \W || & = & || \W\llp || + || \W{(\ll+1)} || \\

```
   & = & (9 * n_\llp * n_{(\ll-1)}) + 9 *( n_\llp * n_{(\ll+1)}) ) & \te
xt{ when } f_\llp = f_{(\ll+1)} = 3 \\
```

\end{array} $$

The two layer network uses *fewer* weights when

$$9 * (n_{(l)} * n_{(l+1)})) < (25 - 9) * n_{(l)} * n_{(l-1)}$$

This will be the case when the number of feature maps in all layers is roughly the same.

- The advantage of the smaller network increases as $f'_{(l)} - f_l$ increases
  - For example: $f'_{(l)} = 7$
  - Versus 3 Convolutional Layers
  - With $f_{(l)} = f_{(l-1)} = f_{(l+1)} = 3$

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**

How many feature maps to use (What value to choose for $n_{(l)}$)

Bag of Tricks for Image Classification with CNNs (https://arxiv.org/abs/1812.01187)

Remember that a larger value for $n_{(l)}$ will increase space and time requirements.

One rule of thumb:

- For $N = 2$
- With filter size $f_{(l)}$
- The number of elements in the spatial dimension of input $\mathrm{y}_{(l-1)}$ involved in the dot product is

$$e = (n_{(l-1)} * f_{(l)} * f_{(l)})$$

- It may not make sense to create *more* than e output features $n_{(l)} > e$
    - We would generate more features than input elements

Inverting convolution

The typical flow for multiple layers of Convolutions

- **Is for the spatial dimension of successive layers to get smaller**
- **By using stride $S > 1$**
- **By using Pooling Layers**

This brings up the question: Can we invert the process ?

- **That is, go from a smaller spatial dimension back to the spatial dimension of input layer 0**

The answer is yes.

This process is sometimes called *Deconvolution* or *Transposed Convolution*.

- **In a Deeper Dive, we relate Convolution to Matrix Multiplication**
- **So the inverting matrix's *dimensions* are the transpose of the matrix implementing the convolution**

We will revisit this in the lecture addressing "What is a CNN looking for ?"

Technical points

Convolution versus Cross Correlation

- math definition of convolution
    - dot product of input and *reversed* filter
    - we are doing [cross correlation (https://en.wikipedia.org/wiki/Convolution)]

In [5]:
```
print("Done")
```

```
Done
```