Convolutional Layers: Space and Time

In our introductory examples

- The non-feature dimension of output $\color{red}{\backslash \mathbf{y}_{\backslash \mathrm{llp}}}$
- Is identical to the non-feature dimension of input $\color{red}{\backslash \mathbf{y}}_{(\ll -1)}$

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

These choices impact

- The size of the non-feature 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 $y_{\backslash\mathrm{llp}}$.

CNN Math: Time versus number of parameters

In designing a Neural Network we are confronted with choices

- how many layers
- width (number of features) at each layer

When Convolutional layers are included, there are additional choices

- size f of filter
- increment with which we slide the kernel over the non-feature dimensions locations

In the absence of a science defined optimal values for the choices

- we resort to empirical studies
- treat the choices as hyper-parameters
- establish a Performance Metric and a set of Benchmark examples
- examine the trade-off between Performance Metric and hyper-parameter choice.

One element in the trade-off involves external costs

- amount of space (memory)
- amount of time

We explore these costs in this section.

Consider input layer $(\ll -1)$ with

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

Layer $\ll$ will apply a Convolution that preserves the non-feature dimensions

$$||\backslash\mathbf{y}_{\backslash\mathrm{llp}}|| = (\dim_{(\ll-1),1} \times \dim_{(\ll-1),2} \times \ldots \dim_{(\ll-1),N} \times n_{\backslash\mathrm{llp}})$$

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

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

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

So the size of $W_{\backslash\text{llp}}$ (ignoring the optional bias term per output feature map)

$$||\backslash\mathbf{W}_{\backslash\text{llp}}|| = n_{\backslash\text{llp}} * (n_{(\ll-1)} * f_{\backslash\text{llp}} * f_{\backslash\text{llp}})$$

The part of the product that most concerns us is $(n_{\backslash\text{llp}} * n_{(\ll -1)})$

- Values for $n_{\backslash\text{llp}}, n_{(\ll -1)}$ in $\{32, 64, 256\}$ are not uncommon !
- Hence $||\backslash\mathbf{W}_{\backslash\text{llp}}||$ 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 $\ll$?

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

So the number of multiplications

$$n_{\backslash\mathrm{llp}} * (\mathrm{dim}_{(\ll-1),1} * \mathrm{dim}_{(\ll-1),2}) * (n_{(\ll-1)} * f_{\backslash\mathrm{llp}} * f_{\backslash\mathrm{llp}})$$

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

- 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, $\backslash\mathbf{y}_{(L)}$ is a One Hot Vector of length $n_L = ||C||$

Suppose layer $(L-1)$ has dimension

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

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

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

The number of weights (ignoring biases) and multiplications is

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

- $n_{(L)} * n_{(L-1)}$ on the order of several thousand
- $(\text{dim}_{(L-1),1} * \text{dim}_{(L-1),2})$ on the order of several million, for images

This may not even be feasible !

Thus, controlling the size of each layer $\backslash\mathbf{y}_{\backslash\mathrm{llp}}$ is of great *practical* importance.

Controlling the size of the non-feature dimensions

Padding

In our examples thus far

- When a location in a non-feature 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 size of the non-feature dimension of output $\y_{\llp}$ is the same as the input $\y_{(\llp-1)}$
- One can simply *not* produce an output for such locations
- It just means the output non-feature dimension shrinks in each dimension by $f_{\llp} - 1$
    - Assuming $f_{\llp}$ is odd
    - The number of locations in which the kernel extends over the border
    - Is Half of the filter size $(f_{\llp} - 1)/2$ times two (for each edge)

Stride

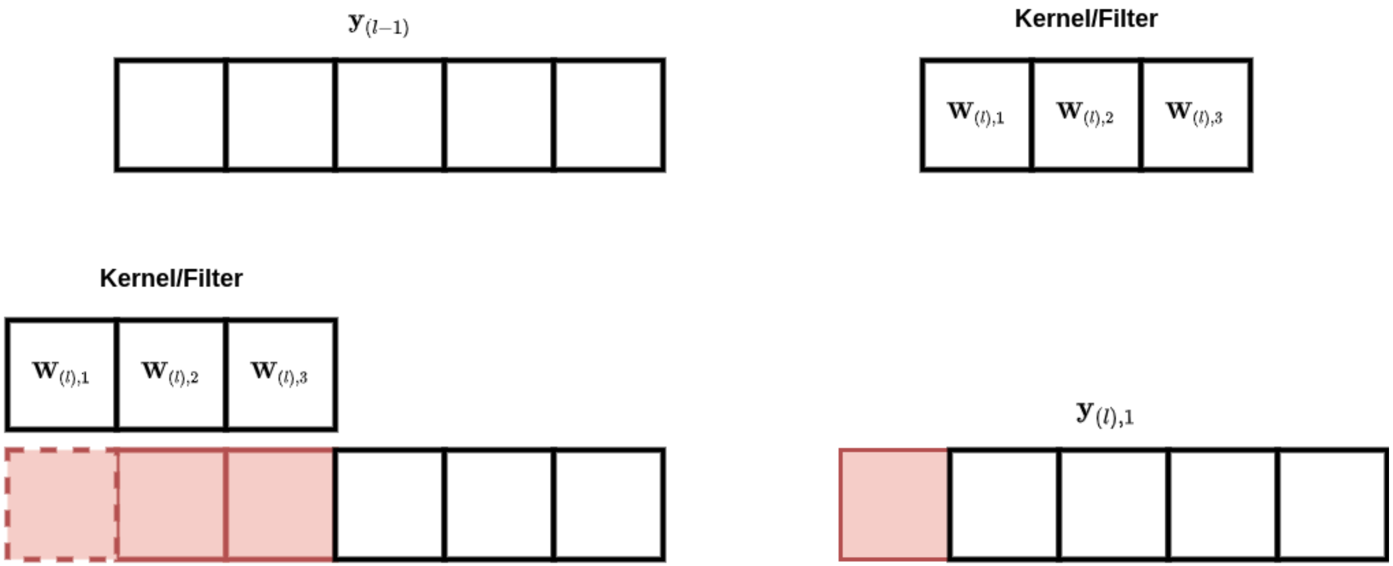
Thus far, we have placed the kernel over *each* location in the non-feature dimensions of the input layer.

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

In the diagram below

- $N = 1$ non-feature dimensions; length $\text{dim}_1 = 5$
- $n = 1$ feature
- $f = 3$ kernel size
- we slide the kernel over just the first two locations (for brevity)

Sliding the kernel over each location



$\mathbf{y}_{(l-1)}$
Kernel/Filter
$\mathbf{w}_{(l),1}$
$\mathbf{w}_{(l),2}$
$\mathbf{w}_{(l),3}$
Kernel/Filter
$\mathbf{w}_{(l),1}$
$\mathbf{w}_{(l),2}$
$\mathbf{w}_{(l),3}$
$\mathbf{y}_{(l),1}$

Consider two adjacent locations in the non-feature 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 non-feature dimension of the input layer

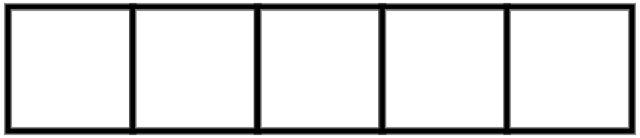
- We may still be able to recognize features
- And reduce the size of the non-feature dimension of the output layer by a factor of 2 for each dimension.

In the diagram below

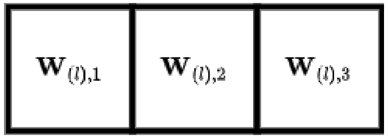
- we use stride $S = 2$
- center the kernel over *every other* location
- reducing the size of the output non-feature dimension $\text{dim}'_1 = \frac{\text{dim}_1}{2}$

Sliding the kernel with stride $S = 2$

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



Kernel/Filter



Kernel/Filter

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

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

When the number N of non-feature dimensions is greater than 1

- we apply the stride S to each dimension

Size of output

We can combine choices of Padding and Stride to control the size of the non-feature dimension of the output layer $\ll$:

Let

- $\mathrm{dim}_{(\ll-1),j}$ denote the number of elements in non-feature dimension j of layer $(\ll -1)$
- P denote the number of elements added as padding on each border
- S denote the stride
- $f_{\backslash\mathrm{llp}}$ be the size of the filter (for each non-feature dimension)

Then the number of elements in non-feature dimension j of output layer $\backslash\text{llp}$ is

$$\text{dim}_{\backslash\text{llp},j} = \frac{\text{dim}_{(\ll -1),j} + 2P - f_{\backslash\text{llp}}}{S} + 1$$

You can see that increasing the stride has the biggest impact on reducing the size of the non-feature dimension of the output.

Pooling layer

There is a layer type with the specific purpose of changing the size of the non-feature dimension of the output.

This is called a Pooling Layer.

A Pooling Layer combines the information from adjacent locations in the non-feature dimension of the input layer.

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

Pooling:

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

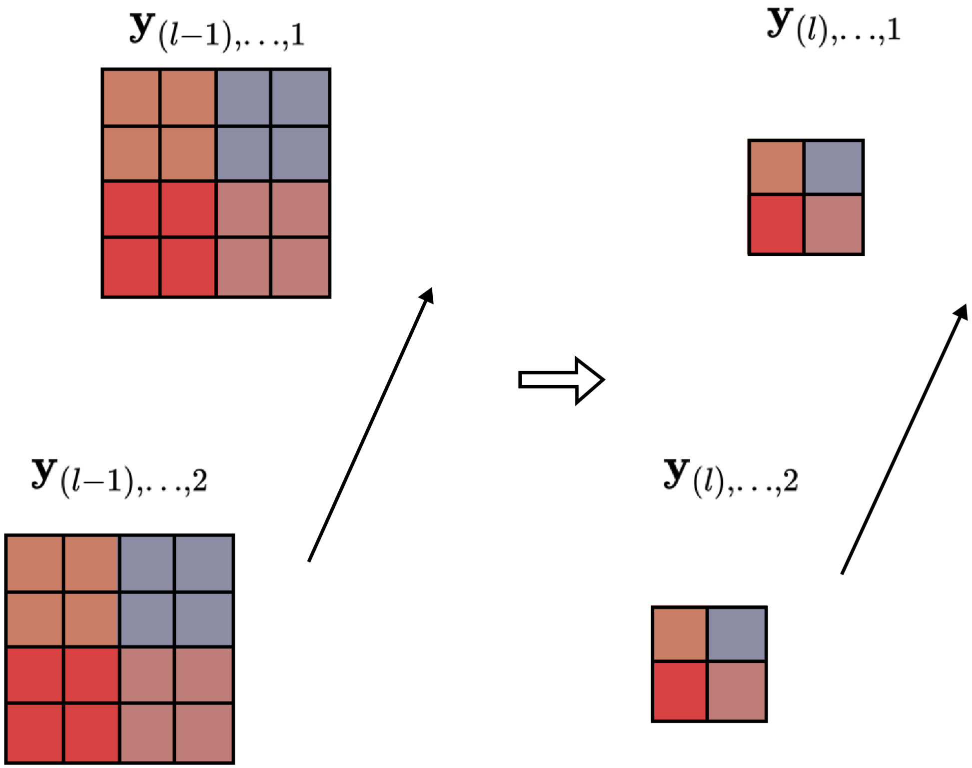
and produces a value in the corresponding location of output layer $\ll$

- That *summarizes* the selected region by applying
    - *pooling operation* $p_{\backslash \mathrm{llp}}$ to the selected region
    - typical pooling operations: maximum, average

Here is an illustration of Pooling

- $N = 2$ non-feature dimensions; $\text{dim}_1 = \text{dim}_2 = 4$
- $n = 2$ features
- $f_{\color{red}{\backslash\text{llp}}} = 2$
- with **stride** $S = 2$

Conv 2D: Pooling (Max/Average)



/

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:

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

The analogous One Dimensional Pooling Layer (Pooling1D) computes

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

where $N'(\backslash\mathbf{y}_{(\ll-1)}, \boldsymbol{f}_{\backslash\mathbf{llp}}, \boldsymbol{j})$

- selects a subsequence of $\backslash\mathbf{y}_{(\ll-1)}$ centered at $\backslash\mathbf{y}_{(\ll-1),\ldots,j}$
- of length $\boldsymbol{f}_{\backslash\mathbf{llp}}$

and $\boldsymbol{p}_{\backslash\mathbf{llp}}$ is a *pooling operation*

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

- Selects a region of length $f_{\backslash\mathbf{llp}}$
- Centered at each location in the non-feature dimension of the input layer $(\ll -1)$

and produces a value in the corresponding location of output layer $\ll$

- 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 non-feature dimension ($N > 1$) and higher number of input channels $n_{(\ll-1)} > 1$.

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

$$||\backslash\mathbf{y}_{(\ll-1)}|| = (\mathbf{dim}_{(\ll-1),1} \times \mathbf{dim}_{(\ll-1),2} \times \ldots \mathbf{dim}_{(\ll-1),N} \times n_{(\ll-1)})$$

Pooling with a stride $S > 1$

- "Down samples" the non-feature dimension
- Sacrificing some information about locality

It effectively asks the question

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

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 non-feature 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 non-feature dimension when it is unimportant or highly variable

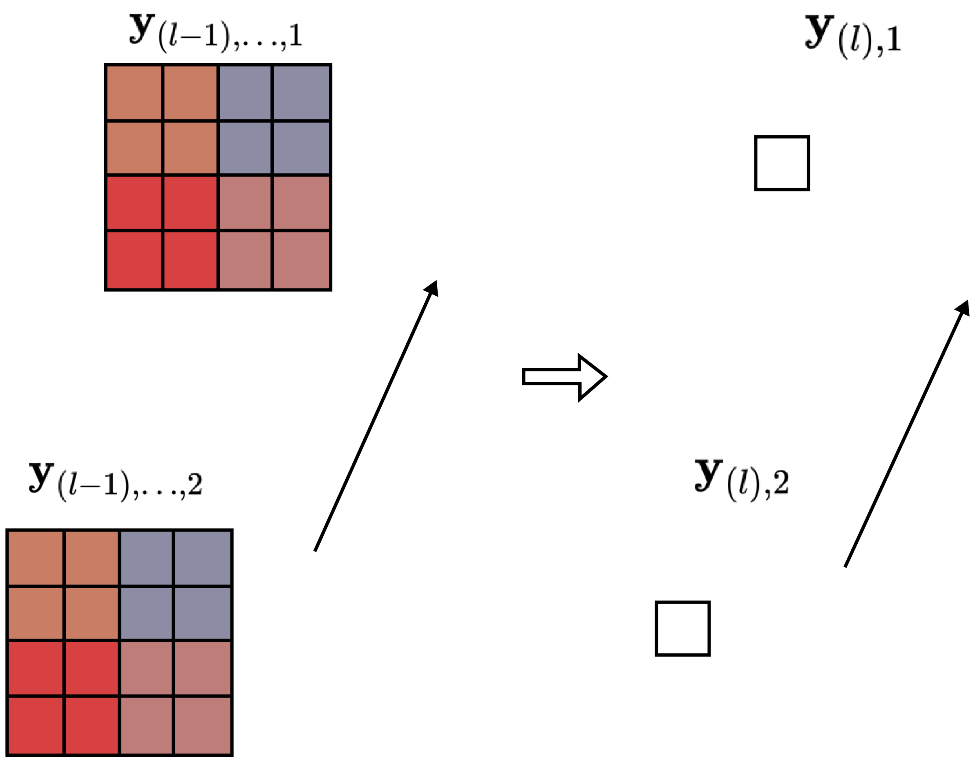
Global Pooling

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

- Is summarized by a single value produced by Max Pooling operation $p'_{\backslash\mathbf{llp}}$
- *eliminating* the non-feature dimensions
- preserving the number of features

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

Conv 2D: Global Pooling (Max/Average)



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

- **No non-feature dimension in $\backslash\mathbf{y}_{\backslash\mathbf{llp}}$ (hence no "$\ldots$")**

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 non-feature 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

Kernel size 1

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

Why might that be ?

Recall that a Convolutional Layer

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

That is\

$$
\begin{aligned}
||\backslash\mathbf{y}_{(\ll-1)}|| &= (\mathbf{dim}_{(\ll-1),1} \times \mathbf{dim}_{(\ll-1),2} \times \ldots \mathbf{dim}_{(\ll-1),N}, \quad \mathbf{n}_{(\ll-1)}) \\
||\backslash\mathbf{y}_{\backslash\mathbf{llp}}|| &= (\mathbf{dim}_{(\ll-1),1} \times \mathbf{dim}_{(\ll-1),2} \times \ldots \mathbf{dim}_{(\ll-1),N}, \quad \mathbf{n}_{\backslash\mathbf{llp}})
\end{aligned}
$$

A single kernel of size $f_{\backslash\text{llp}} = 1$ in all N non-feature dimensions

- replaces the $n_{(\ll-1)}$ features at each location
- with a sum of the features (weighted by the kernel value corresponding to that feature)

With $n_{\backslash\text{llp}}$ such kernels at layer $\ll$, each with $f_{\backslash\text{llp}} = 1$

- the convolution changes the feature dimension from $n_{(\ll-1)}$ to $n_{\backslash\text{llp}}$
- without performing any substantial pattern match

Setting $n_{\backslash\text{llp}}$ much less than $n_{(\ll-1)}$ is thus a convenient way to reduce the feature dimension.

Receptive field

The filter size $f_{\llp}$ 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 $f_{\llp}$
- With a smaller $f_{\llp}$ in conjunction with *more* Convolutional Layers

Let's demonstrate this by examining the concept of [Receptive field (CNN_Receptive_Field.ipynb)](CNN_Receptive_Field.ipynb)

Review: Controlling the size

Let's summarize our knowledge of controlling the size of $\y_{(\ll-1)}$:

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

Striding and Pooling

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

Kernel size $f_{\text{\llp}} = 1$

- reduction depends on the ratio of $n_{\text{\llp}}$ to $n_{(\ll-1)}$

Interfacing with other layer types

The CNN layer type is the only one (so far) that accepts inputs with non-feature dimensions.

Before we can append

- the output of a CNN layer
    - has non-feature dimensions
- to a layer type that does not process inputs with non-feature dimensions

we must eliminate the non-feature dimension of the CNN layer output.

Two common layer types that eliminate the non-feature dimensions are

- `Flatten`
- Global Pooling variants (Average, Max)

Note that

- `Flatten` does not reduce the size
    - each element of the non-feature dimension becomes a feature in the flattened representation
- Global pooling does reduce the size
    - the collection of elements are replaced by a scalar summary (average, max)

If the size if not reduced

- the number of parameters in subsequent layers may be very large.

Consider

- **appending a Classifier (e.g., `Dense` layer with $n_{(L)}$ output classes)**
- **to the output (after removing non-feature dimensions) of the CNN layers**
  - **call the size n_{CNN}**

The number of parameters in the Classifier layer is

$$n_{\text{CNN}} * n_{(L)}$$

Thus not reducing n_{CNN} affects the parameter count of succeeding layers.

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 $n_{\llp}$)

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

Remember that a larger value for $n_{\llp}$ will increase space and time requirements.

One rule of thumb:

- For $N = 2$
- With filter size $f_{\llp}$
- The number of elements in the non-feature dimension of input $\y_{(\ll-1)}$ involved in the dot product is
$$e = (n_{(\ll-1)} * f_{\llp} * f_{\llp})$$
- It may not make sense to create *more* than e output features $n_{\llp} > e$
    - We would generate more features than input elements

Inverting convolution

The typical flow for multiple layers of Convolutions

- Is for the non-feature 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 non-featue dimension back to the non-feature 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 [4]:

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

```
Done
```