Convolutional Neural Networks: the spatial dimensions

Our treatment, thus far, of Neural Networks has been rather limited. An example has consisted of

- Multiple features
- At a single spatial location
- Represented as a vector of shape $(1 \times n_{(l)})$
    - But we often ignored the singleton dimension

But the natural world's spatial dimensions are much higher than 1 !

- $N > 1$ dimensions
- Our examples become $(N+1)$ dimensional
- Represented as a vector of shape $(d_{(l),1} \times d_{(l),2} \times \ldots d_{(l),N} \times n_{(l)})$

When $N = 1$ and $d_1 = 1$

- we have our case of $n_{(l)}$ features at a single location

We have shown that permuting the order of features has no effect on a Dense layer

- There is no ordering relationship among features

But when $d_1 > 1$, there is a *spatial ordering*. For example

- a 2D image
- time ordered data

We need some terminology to distinguish the final dimension from the non-final dimensions

Suppose $\mathbf{y}_{(l)}$ is $(N_{(l)} + 1)$ dimensional of shape

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

(Thus far: $N_{(l)} = 1$ and $n_{(l)} = 1$ but that will soon change)

The first $N_{(l)}$ dimensions $(d_{(l),1} \times d_{(l),2} \times \ldots d_{(l),N})$

- Are called the *spatial* dimensions of layer l

The last dimension (of size $n_{(l)}$)

- Indexes the features i.e., varies over the number of features
- Called the *feature* or *channel* dimension

Notation

- $N_{(l)}$ denotes the *number* of spatial dimensions of layer l
- $n_{(l)}$ denotes the *number of features* in layer l
- We elide the spatial dimensions as necessary, writing
  $$\mathbf{y}_{(l),\ldots,j}$$
  to denote *feature map* j of layer l
  - where the dots $(\ldots)$ indicate the $N_{(l)}$ spatial dimensions
  - e.g., the feature map detecting a "smile" in the image of a face

For example

- A grey-scale image
    - $N = 2, n_{(l)} = 1$
    - Each pixel in the image has one feature
        - the grey-scale intensity
    - There is an ordering relationship between 2 pixels
        - "left/right", "above/below"
- A color image
    - $N = 2, n_{(l)} = 3$
    - Each pixel in the image has 3 features/attributes
        - the intensity of each of the colors

One can imagine even higher dimensional data ($N > 2$)

- Equity data with "spatial location" identified by (Month, Day, Time)
    - With attributes: $\{$ Open, High, Low, Close $\}$
    - Month/Day/Time are ordered

Note the distinction between the cases

- When layer l has dimension $(d_{(l)} \times 1)$
    - a single feature
    - at $d_{(l)} = d_{(l-1)}$ *spatial* locations
- When layer l has dimension $(1 \times d_{(l)})$
    - (which is how we have implicitly been considering vectors when discussing the Dense layer type)
    - $d_{(l)} = d_{(l-1)}$ features
    - at a single spatial location

$n_{(l)}$ will always refer to the *number of features* of a layer l

Here is a [picture (CNN_pictorial.ipynb#Conv-1D:-single-feature)](CNN_pictorial.ipynb#Conv-1D:-single-feature) of a Convolutional layer l transforming

- a 1-dimensional input layer $(l-1)$ consisting of a single feature
  - $N_{(l-1)} = 1, n_{(l-1)} = 1$
- into a 1-dimensional output layer l consisting of a single feature
  - $N_{(l)} = 1, n_{(l)} = 1$

We will generalize Convolution to deal with

- $N_{(l)} > 1$ spatial dimensions
- $n_{(l)} > 1$ features

As a preview of concepts to be introduced, consider

- the input layer $(l-1)$ is a two-dimensional ($N_{(l-1)} = 2$) grid of pixels
- $n_{(l-1)} = 1$
- layer l is a Convolutional Layer identifying $n_{(l)} = 3$ features

Convolution: 1 input feature to 3 output features

Layer $(l-1)$ is three-dimensional tensor: $8 \times 8 \times 1$

- Spatial dimension 8×8
- 1 feature map (channel dimension $= 1$)

- Kernel $k_{(l),j}$ is applied to each spatial location of layer $(l-1)$
- Detecting the presence of the pattern (defined by the kernel) at that location
    - kernel $k_{(l),1}$ detects an eye
- Which results in feature map $\mathbf{y}_{(l)}, \ldots, j$ being created at layer l
    - $\mathbf{y}_{(l),\ldots,1}$ are indicators of the presence of an "eye" feature

Convolutional Layer description

With this terminology we can say that Convolutional Layer l:

- Transforms the $n_{(l-1)}$ feature maps of layer $(l-1)$
- Into $n_{(l)}$ feature maps of layer l
- Preserving the spatial dimensions: $d_{(l),p} = d_{(l-1),p} \; 1 \le p \le N_{(l-1)}$
- Uses a different kernel $\mathbf{k}_{(l),j}$ for each output feature/channel $1 \le j \le n_{(l)}$
- Applies this kernel to *each* element in the *spatial* dimensions
- Recognizing a single feature at each location within the spatial dimension

Channel Last/First

We have adopted the convention of using the final dimension as the feature dimension.

- This is called *channel last* notation.

Alternatively: one could adopt a convention of the first channel being the feature dimension.

- This is called *channel first* notation.

When using a programming API: make sure you know which notation is the default

- Channel last is the default for TensorFlow, but other toolkits may use channel first.

Conv1d transforming single feature to multiple features

Here is a [picture (CNN_pictorial.ipynb#Conv-1D:-single-feature-to-multiple-features)](CNN_pictorial.ipynb#Conv-1D:-single-feature-to-multiple-features) of a Convolutional layer l transforming

- a 1-dimensional input layer $(l-1)$ consisting of a single feature
  - $N_{(l-1)} = 1, n_{(l-1)} = 1$
- into a 1-dimensional output layer l consisting of a *multiple* features
  - $N_{(l)} = 1, n_{(l)} > 1$

Conv1d transforming multiple features to multiple features

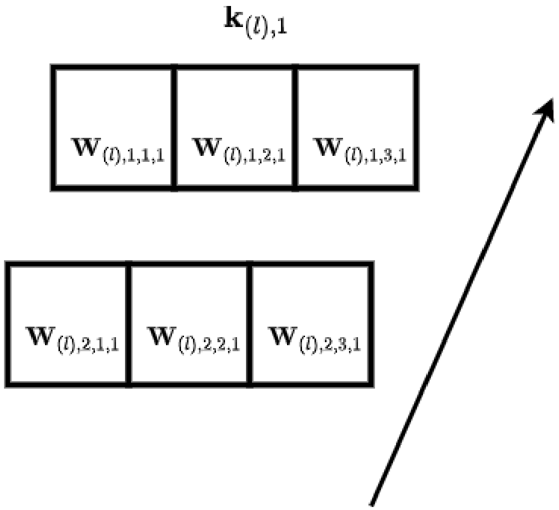
What happens when the input layer has multiple features ?

- e.g., applying Convolutional layer $(l+1)$ to the $n_{(l)}$ features created by Convolutional layer l

The answer is

- The kernels of layer l also have a *feature* dimension
  - Kernel dimensions are $(f_{(l)} \times f_{(l)} \times n_{(l-1)})$
- This kernel is applied
  - at each spatial location
  - to *all features* of layer $(l-1)$
  - Computing a generalized "dot product": sum of element-wise products

Conv 1D: 2 input features: kernel 1



- $\mathbf{W}_{(l),j',\ldots,j}$
  - layer l
  - output feature j
  - spatial location: $\ldots \in \{1, 2, 3\}$
  - input feature j'

Notice that (apart from combining spatial locations)

- multiple feature maps from layer $(l-1)$ are combined into one feature map at layer l.
- This is how the "left" half-smile and "right" half-smile features combine into the single "smile" feature

Here is a picture (CNN_pictorial.ipynb#Conv-1D:-Multiple-features-to-multiple-features) of a Convolutional layer l transforming

- a 1-dimensional input layer $(l-1)$ consisting of a 2 features
  - $N_{(l-1)} = 1, n_{(l-1)} = 2$
- into a 1-dimensional output layer l consisting of a *multiple* features
  - $N_{(l)} = 1, n_{(l)} = 3$

With an input layer having N spatial dimensions, a Convolutional Layer l producing $n_{(l)}$ features

- Preserves the "spatial" dimensions of the input
- Replaces the channel/feature dimensions

That is\

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

Conv2d: Two dimensional convolution ($N = 2$)

Thus far, the spatial dimension has been of length $N = 1$.

Generalizing to $N = 2$ is straightforward.

- The number of spatial dimensions (elements denoted by $\ldots$) expands from 1 to 2

Conv 2D: single input feature: kernel 1

$\mathbf{k}_{(l),1,1}$

$\mathbf{W}_{(l),1,1,1,1}$	$\mathbf{W}_{(l),1,1,2,1}$	$\mathbf{W}_{(l),1,1,3,1}$
$\mathbf{W}_{(l),1,2,1,1}$	$\mathbf{W}_{(l),1,2,2,1}$	$\mathbf{W}_{(l),1,2,3,1}$
$\mathbf{W}_{(l),1,3,1,1}$	$\mathbf{W}_{(l),1,3,2,1}$	$\mathbf{W}_{(l),1,3,3,1}$

- $\mathbf{W}_{(l),j',\ldots,j}$
  - layer l
  - output feature j
  - spatial location: $\ldots \in \{(\alpha, \alpha') \in (d_{(l-1),1} \times d_{(l-1),2}\}$
  - input feature j'

Here is a [picture (CNN_pictorial.ipynb#Conv-2D:-single-feature-to-single-feature)](CNN_pictorial.ipynb#Conv-2D:-single-feature-to-single-feature) of a Convolutional layer l transforming

- a 2-dimensional input layer $(l-1)$ consisting of a 1 feature
  - $N_{(l-1)} = 2, n_{(l-1)} = 1$
- into a 2-dimensional output layer l consisting of 1 feature
  - $N_{(l)} = 1, n_{(l)} = 1$

We can further generalize to producing multiple output features

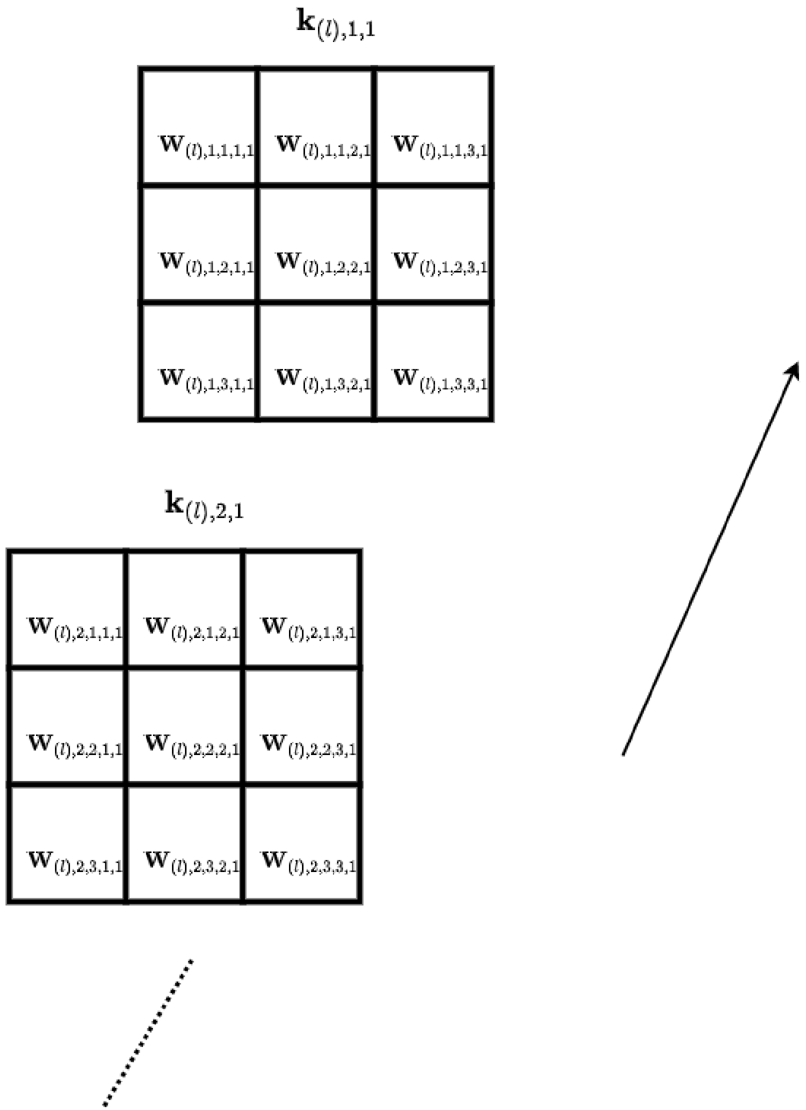
Here is a [picture (CNN_pictorial.ipynb#Conv-2D:-single-feature-to-multiple-features)](CNN_pictorial.ipynb#Conv-2D:-single-feature-to-multiple-features) of a Convolutional layer l transforming

- a 2-dimensional input layer $(l-1)$ consisting of a 1 feature
    - $N_{(l-1)} = 2, n_{(l-1)} = 1$
- into a 2-dimensional output layer l consisting of 2 feature
    - $N_{(l)} = 1, n_{(l)} = 2$

Dealing with multiple input features works similarly as for $N = 1$:

- The dot product
- Is over a spatial region that now has a "depth" $n_{(l-1)}$ equal to the number of input features
- Which means the kernel has a depth $n_{(l-1)}$

Conv 2D: multiple input features: kernel 1



Here is a [picture (CNN_pictorial.ipynb#Conv-2D:-multiple-features-to-single-feature)](CNN_pictorial.ipynb#Conv-2D:-multiple-features-to-single-feature) of a Convolutional layer l transforming

- a 2-dimensional input layer $(l-1)$ consisting of multiple features
    - $N_{(l-1)} = 2, n_{(l-1)} = 2$
- into a 2-dimensional output layer l consisting of 1 feature
    - $N_{(l)} = 1, n_{(l)} = 1$

And finally: the general case for a 2 spatial dimensions

Here is a [picture (CNN_pictorial.ipynb#Conv-2D:-multiple-features-to-multiple-features)](CNN_pictorial.ipynb#Conv-2D:-multiple-features-to-multiple-features) of a Convolutional layer l transforming

- a 2-dimensional input layer $(l-1)$ consisting of multiple features
  - $N_{(l-1)} = 2, n_{(l-1)} = 3$
- into a 2-dimensional output layer l consisting of multiple features
  - $N_{(l)} = 1, n_{(l)} = 2$

Training a CNN

Hopefully you understand how kernels are "feature recognizers".

But you may be wondering: how do we determine the weights in each kernel ?

Answer: a Convolutional Layer is "just another" layer in a multi-layer network

- The kernels are just weights (like the weights in Fully Connected layers)
- We solve for all the weights $\mathbf{W}$ in the multi-layer network in the same way

The answer is: exactly as we did in Classical Machine Learning

- Define a loss function that is parameterized by $\mathbf{W}$:
$$\mathcal{L} = L(\hat{\mathbf{y}}, \mathbf{y}; \mathbf{W})$$

- The kernel weights are just part of $\mathbf{W}$

- Our goal is to find $\mathbf{W}^*$ the "best" set of weights
$$\mathbf{W}^* = \operatorname*{argmin}_W L(\hat{\mathbf{y}}, \mathbf{y}; \mathbf{W})$$
- Using Gradient Descent !

In other words: their is nothing special about finding the "best" kernels.

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