Hands-on Machine Learning

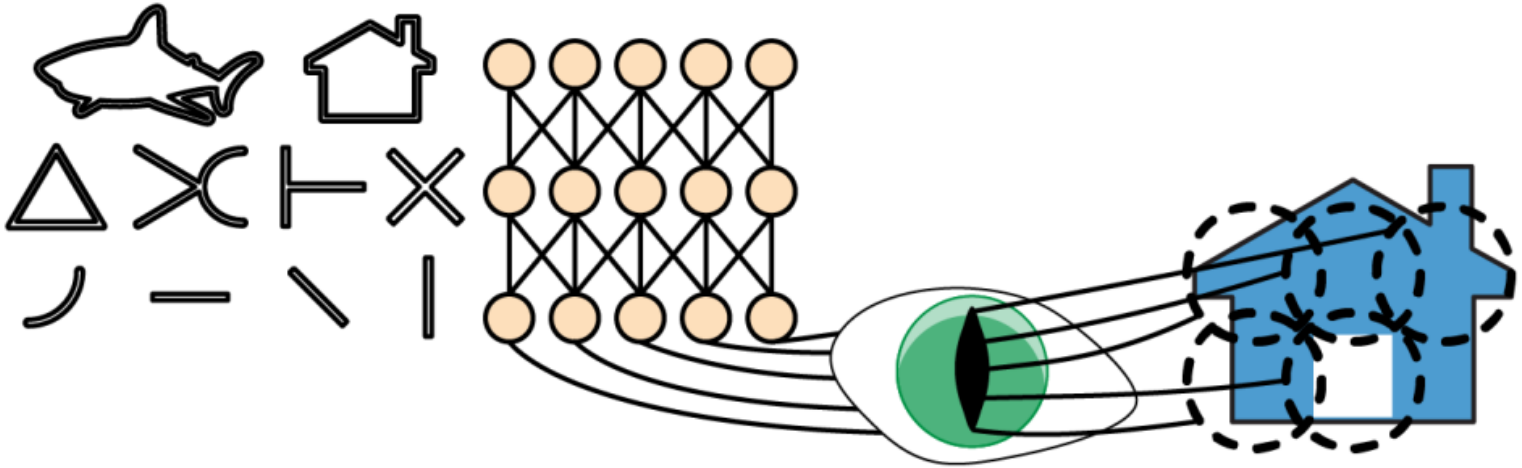
14. Convolutional Neural Networks

Convolutional Neural Networks

- *Convolutional neural networks* (CNNs) arose from the study of the brain's visual cortex, and they have been used in computer image recognition since the 1980s.
- They power image search services, self-driving cars, automatic video classification systems, and more.
- CNNs are not restricted to visual perception:
  - voice recognition
  - natural language processing

The Architecture of the Visual Cortex

- Hubel and Wiesel performed a series of experiments on cats, and showed that many neurons in the visual cortex have a small *local receptive field.*



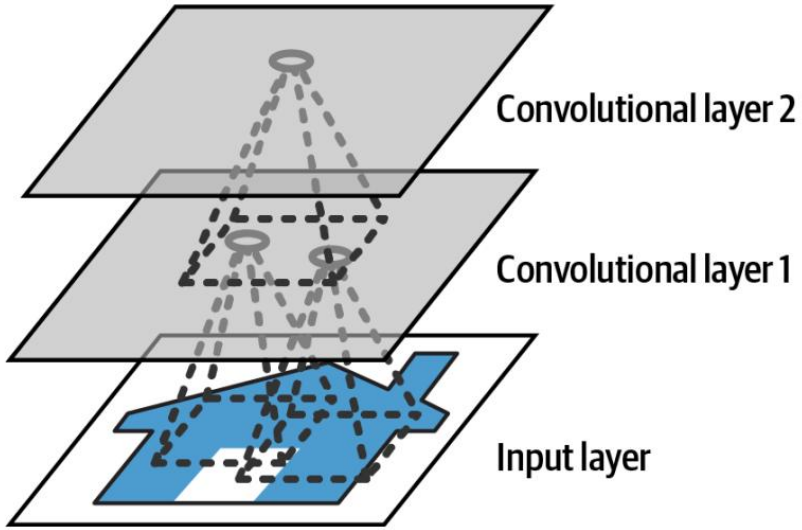
History of CNNs

- The studies of the visual cortex inspired the neocognitron, which is a neural network proposed by Fukushima in 1979 used to recognize Japanese handwritten characters.
  - This gradually evolved into CNNs.
- A 1998 paper by Yann LeCun et al. introduced the famous *LeNet-5* architecture, which became widely used by banks to recognize hand-written digits on checks.
- It introduces two new building blocks:
  - *convolutional layers*
  - *pooling layers*

1.
Convolutional Layer

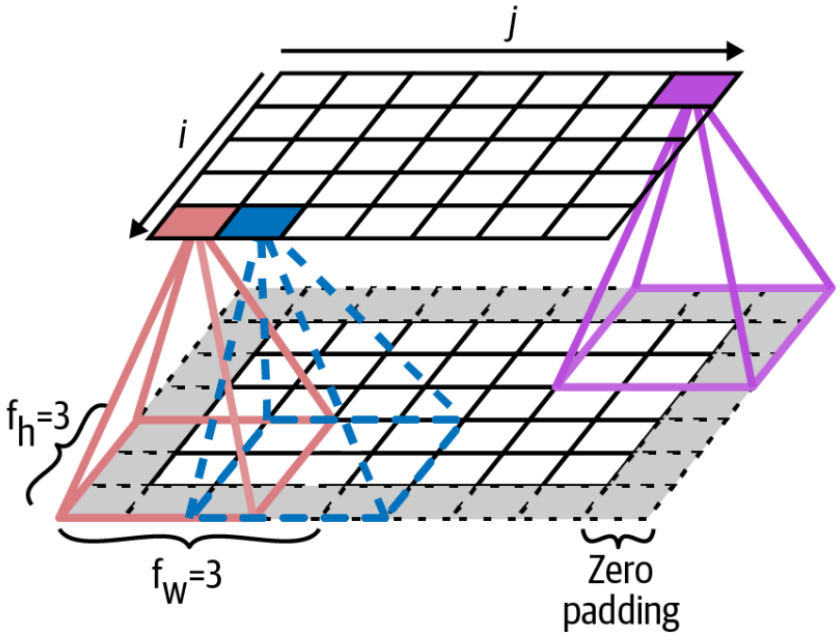
Convolutional Layers

- Neurons in the first convolutional layer are not connected to every single pixel in the input image but only to pixels in their receptive fields.
- This architecture helps the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer.

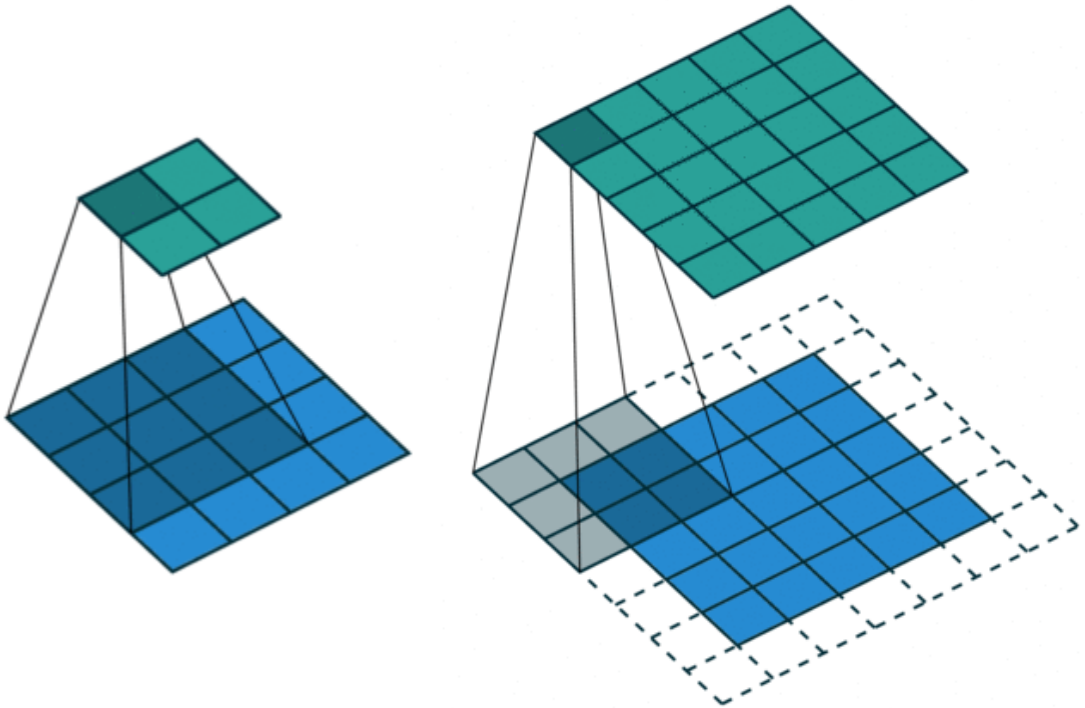


Zero Padding

- A neuron in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$.
- *Zero padding*: in order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs.

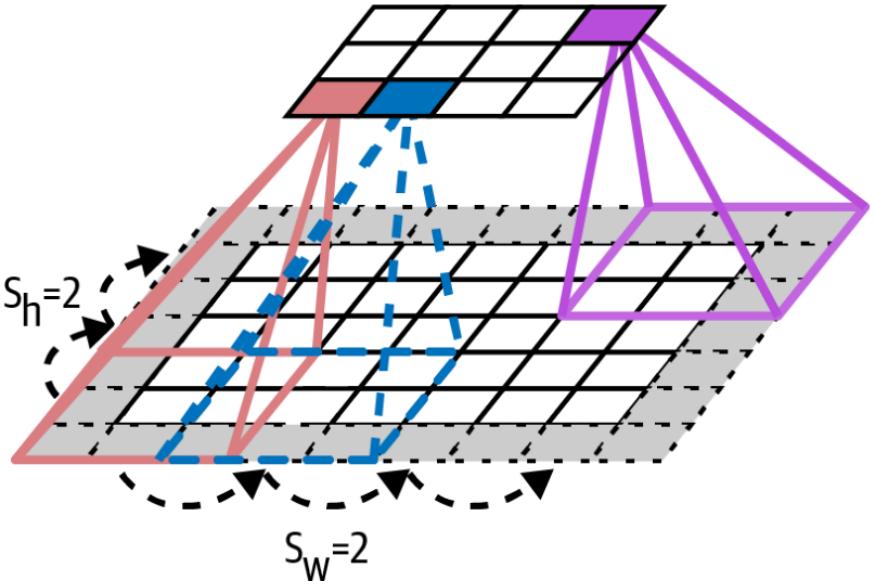


Zero Padding



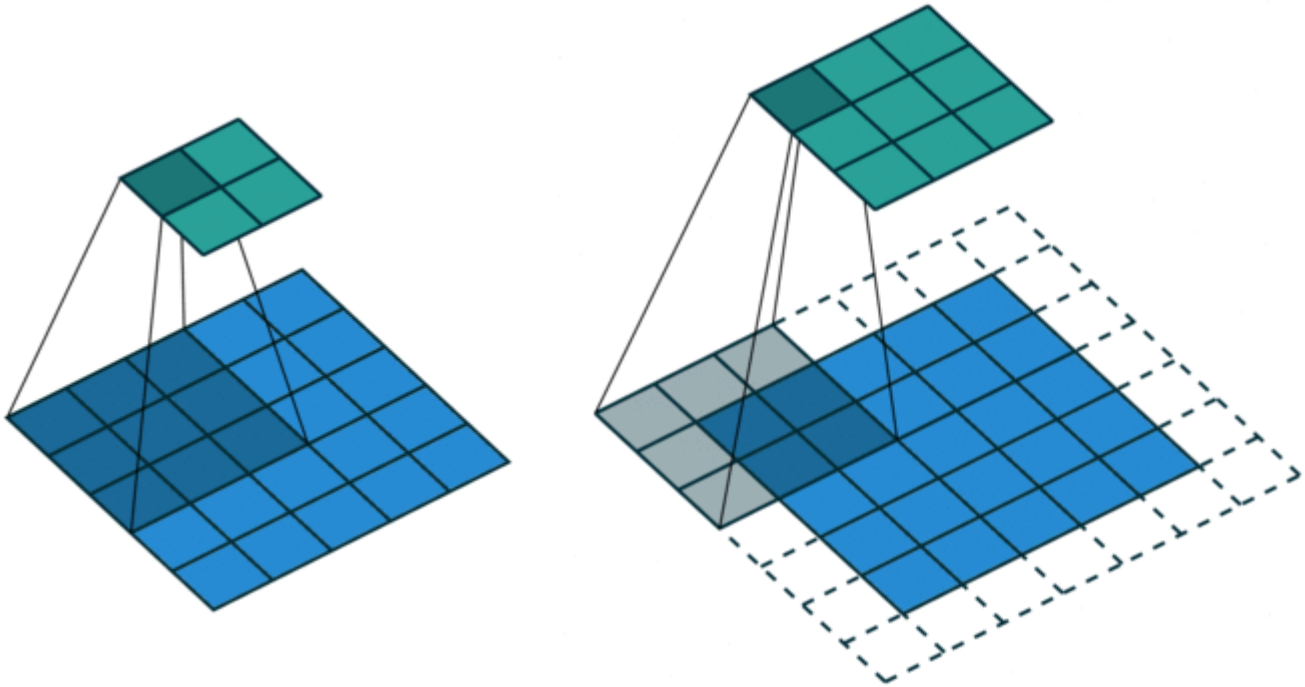
Reducing dimensionality using strides

- We can connect a large input layer to a much smaller layer by spacing out the receptive fields.
  - This dramatically reduces the computational complexity.
- The horizontal or vertical step size from one receptive field to the next is called the *stride*.



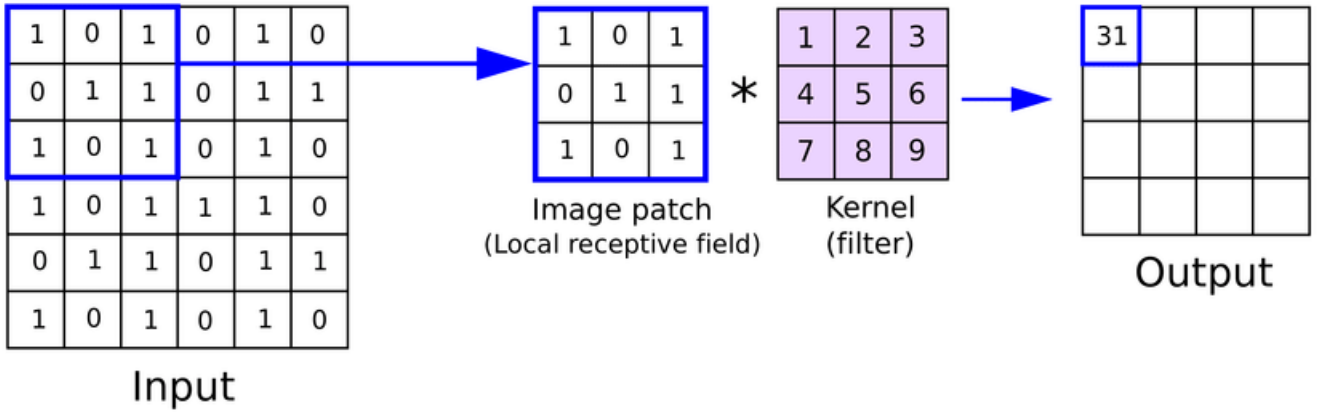
- (i, j) is connected to $(i \times s_h$ to $i \times s_h + f_h - 1$, j $\times s_w$ to $j \times s_w + f_w - 1)$ in previous layer.

Using Strides



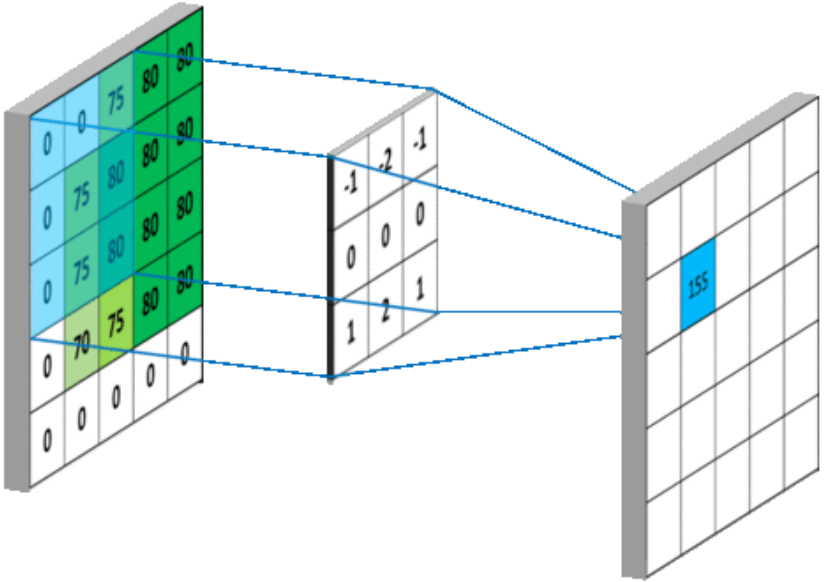
Filters

- A neuron's weights can be represented as a small matrix the size of the receptive field.
- These sets of weights are called *filters* (or *convolution kernels*, or just *kernels*).

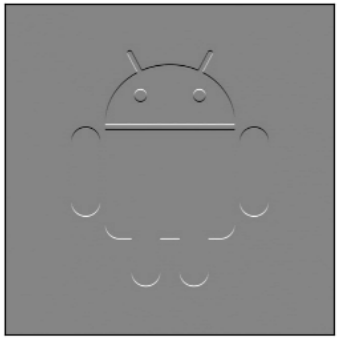


Feature Map

- A layer full of neurons using the same filter outputs a *feature map*, which highlights the areas in an image that activate the filter the most.



Feature Map



-1	0	1
-2	0	2
-1	0	1

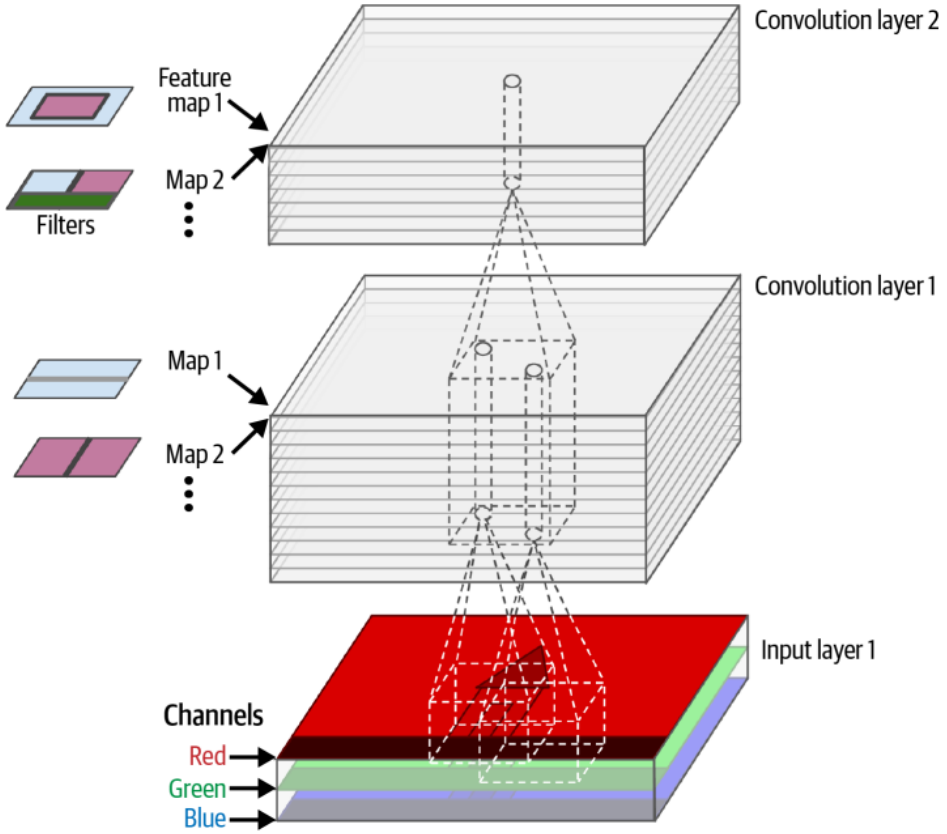
Vertical Filter

-1	-2	-1
0	0	0
1	2	1

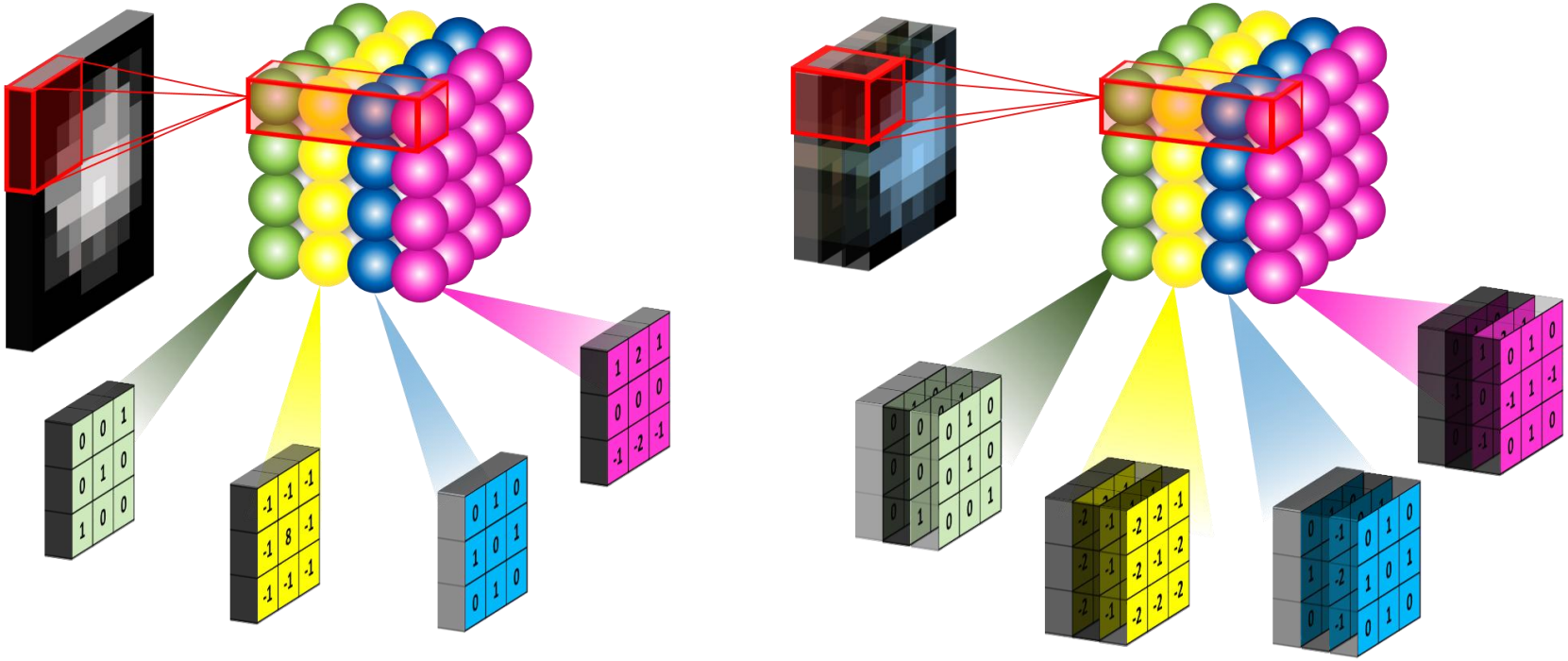
Horizontal Filter

Stacking Multiple Feature Maps

- Convolutional layer has multiple filters and outputs one feature map per filter.
- It has one neuron per pixel in each feature map, and all neurons within a given feature map share same parameters (same kernel and bias term).
- A convolutional layer applies multiple trainable filters to its inputs simultaneously, making it capable of detecting multiple features anywhere in its inputs.



Stacking Multiple Feature Maps



Computing the output of a neuron

- A neuron in row i, column j of feature map k in convolutional layer l is connected to the outputs of the neurons in the previous layer $l-1$, in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$: output of neuron in row i, column j in feature map k
- $x_{i',j',k'}$: output of neuron in layer $l-1$, row i', column j', feature map k'
- $w_{u,v,k',k}$: connection weight between any neuron in feature map k of the layer l and its input at row u, column v (relative to the neuron's receptive field), and feature map k'.

2.
Implementing Convolutional Layers

Load and Preprocess Images

- Keras's `CenterCrop` layers: crops the center of each image to the specified height and width.
- Keras's `Rescaling` layers: multiplies each pixel value to `scale`.

```
from sklearn.datasets import load_sample_images
import tensorflow as tf

images = load_sample_images()["images"]
images = tf.keras.layers.CenterCrop(height=70, width=120)(images)
images = tf.keras.layers.Rescaling(scale=1 / 255)(images)
```

```
images.shape
```

```
TensorShape([2, 70, 120, 3])
```

2D Convolutional Layer

- Create a 2D convolutional layer and feed it these images:

```
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7)
fmaps = conv_layer(images)
```

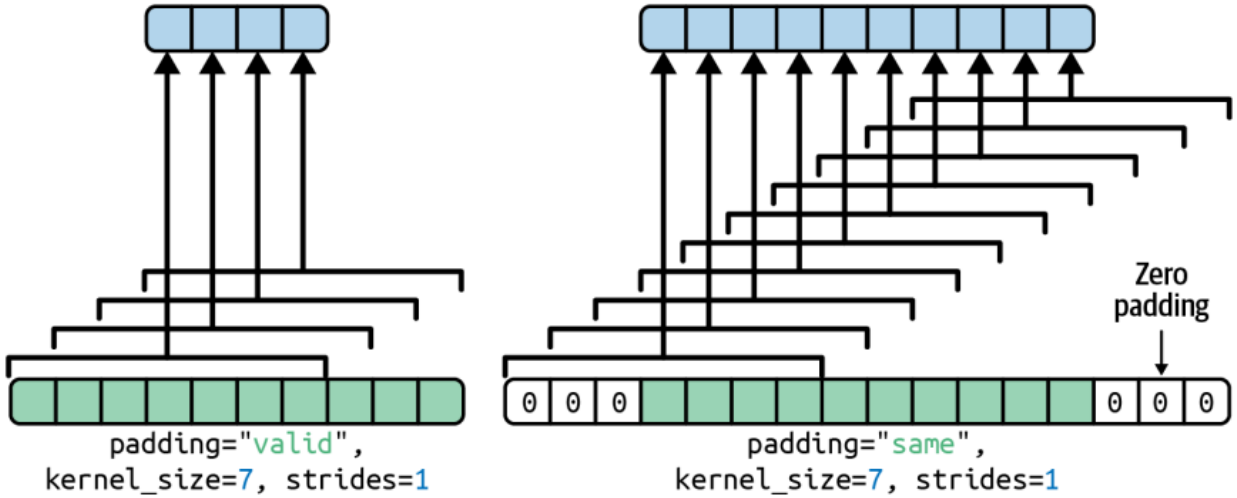
- "2D" refers to the number of *spatial* dimensions (height and width).
  - the layer takes 4D inputs: the two additional dimensions are the batch size (first dimension) and the channels (last dimension).

```
fmaps.shape
```

```
TensorShape([2, 64, 114, 32])
```

- 32 channels (feature maps) because we set `filters=32`.
- Height and width have shrunk by 6 pixels because of no zero-padding.

Padding Options

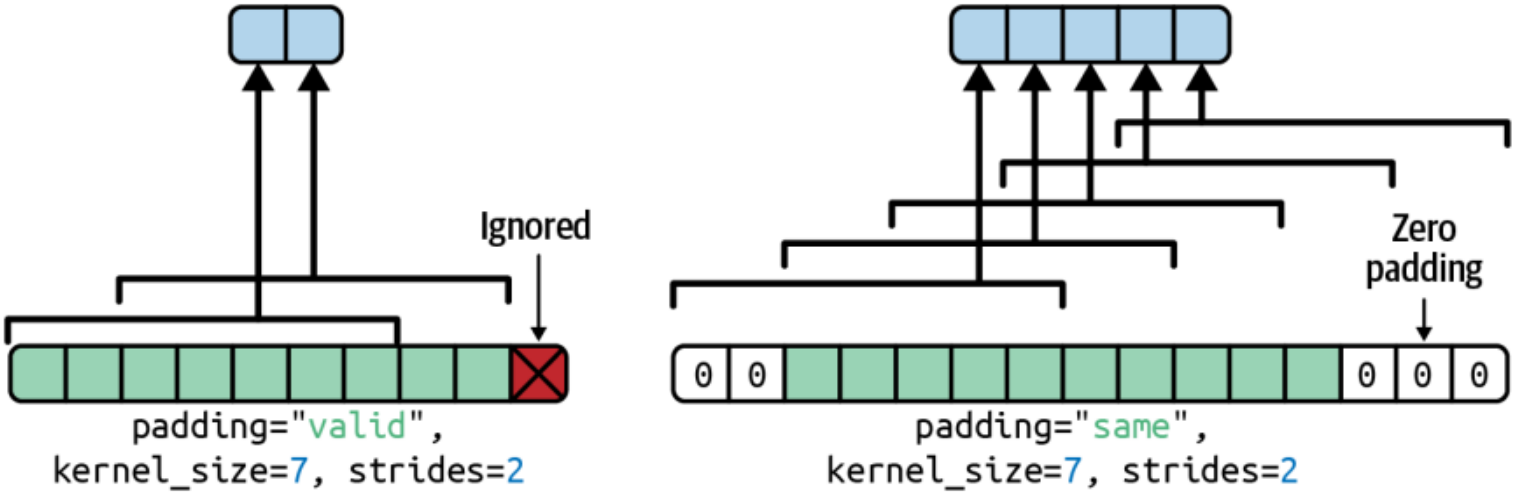



```
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7, padding="same")
fmaps = conv_layer(images)
```

```
fmaps.shape
```

```
TensorShape([2, 70, 120, 32])
```

Strides




```
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7, padding="same", strides=2)
fmaps = conv_layer(images)
fmaps.shape
```

```
TensorShape([2, 35, 60, 32])
```

Layer Weights

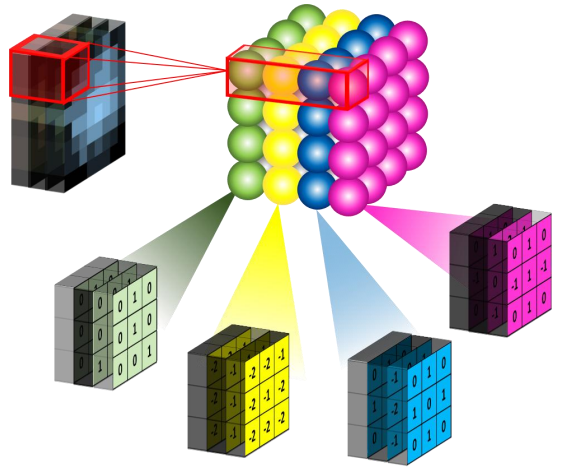
- Just like a `Dense` layer, a `Conv2D` layer holds all the layer's weights, including the kernels ($w_{u,v,k',k}$) and biases (b_k).
- Kernels are initialized randomly, while the biases are initialized to zero.
- The `kernels` array is 4D, and its shape is [*kernel_height*, *kernel_width*, *input_channels*, *output_channels*].

```
kernels, biases = conv_layer.get_weights()
kernels.shape
```

```
(7, 7, 3, 32)
```

```
biases.shape
```

```
(32,)
```



Activation Function

- Specify an activation function (such as ReLU) for a Conv2D layer.
  - and specify the corresponding kernel initializer (such as He initialization).

```
tf.keras.layers.Conv2D(32, kernel_size=3, padding="same",
                       activation="relu", kernel_initializer="he_normal"),
```

- A convolutional layer performs a linear operation, so if you stacked multiple convolutional layers without any activation functions they would all be equivalent to a single convolutional layer.

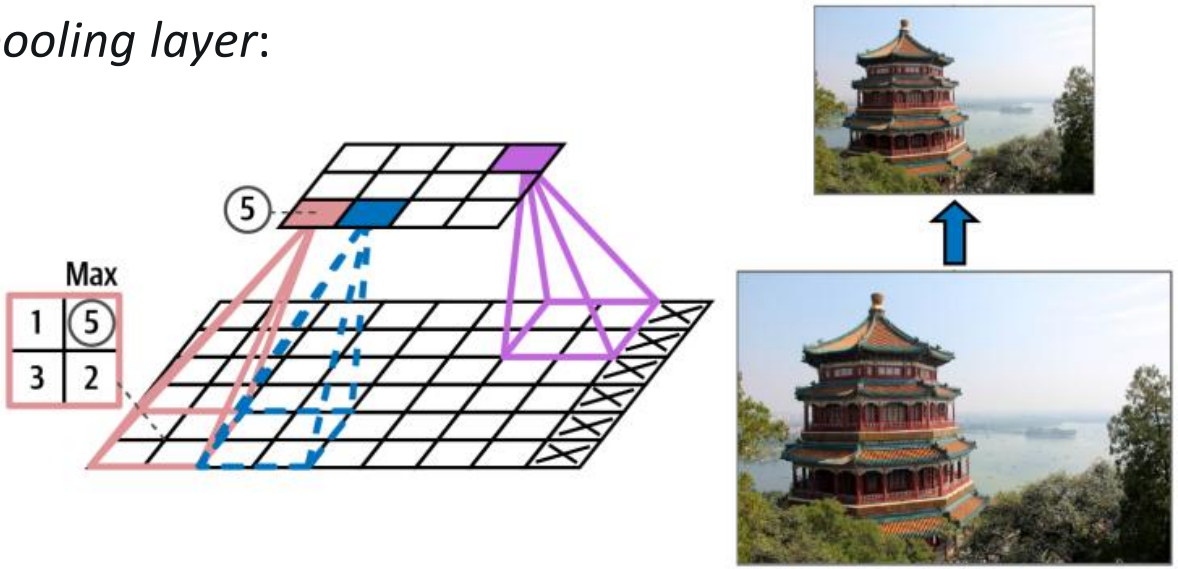
Memory Requirements

- The convolutional layers require a huge amount of RAM.
  - especially during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass.
- A convolutional layer with 200 5×5 filters, with stride 1 and "same" padding, that takes 150×100 RGB images as input requires 12 MB of RAM for each instance, i.e. 1.2 GB of RAM for a training batch of 100 instances.
- During inference (i.e., when making a prediction for a new instance), you only need as much RAM as required by two consecutive layers.
  - But during training, the amount of RAM needed is (at least) the total amount of RAM required by all layers.

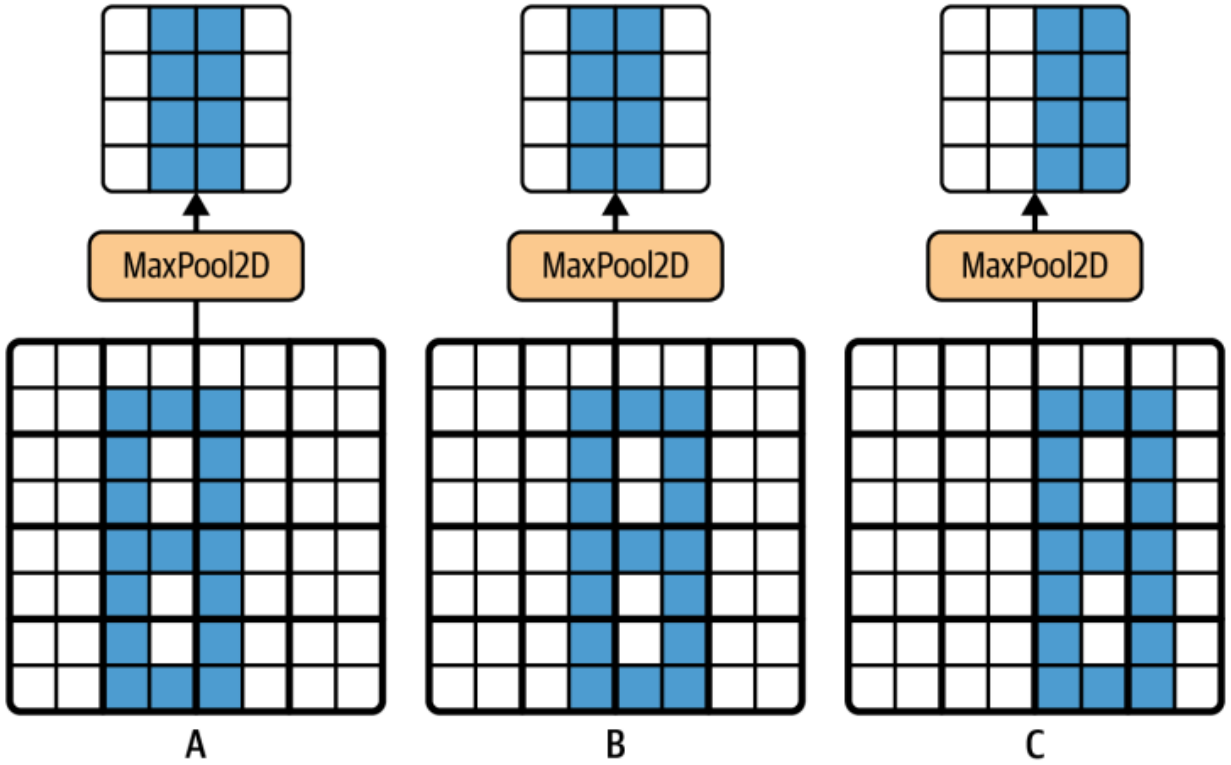
3.
Pooling Layers

Pooling Layer

- Pooling layers' goal is to *subsample* (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters.
- A *max pooling layer*:



Invariance to Small Translations



Pros and Cons of Pooling Layer

- Advantages of using a pooling layer:
  - Small translation (shift) invariance
  - Small rotational invariance
  - Slight scale invariance
- Disadvantages:
  - The layer drops a large portion of the input values.
  - In applications such as classifying each pixel in an image according to the object that pixel belongs to (semantic segmentation), invariance is not desirable.
    - If the input image is shifted by one pixel to the right, the output should also be shifted by one pixel to the right.
    - Here we need *equivariance*, not invariance: a small change to the inputs should lead to a corresponding small change in the output.

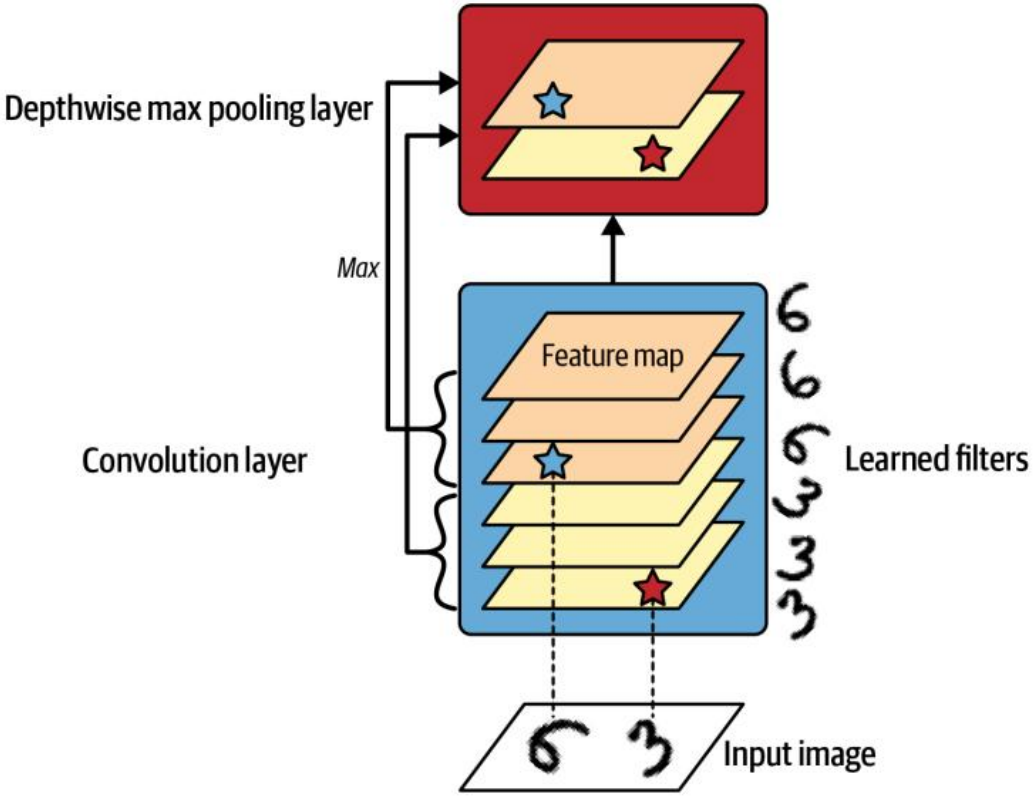
Implementing Pooling Layers

- To create a 2D max pooling layer, using a 2×2 kernel:

```
max_pool = tf.keras.layers.MaxPool2D(pool_size=2)
```

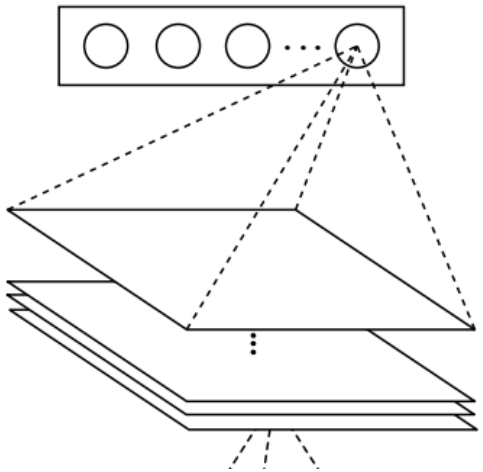
- The strides default to the kernel size
- It uses no padding by default.
- To create an *average pooling layer*, use `AvgPool2D`.
- Max pooling layers generally perform better than average pooling.
  - Max pooling preserves only the strongest features, getting rid of all the meaningless ones, so the next layers get a cleaner signal to work with.
  - Max pooling offers stronger translation invariance than average pooling, and it requires slightly less compute.

Depthwise Max Pooling



Global Average Pooling Layer

- *Global average pooling layer* computes the mean of each entire feature map.
  - Like an average pooling layer using a pooling kernel with the same spatial dimensions as the inputs.
- The layer outputs a s ap and per instance.
- This is extremely de rmation in the feature map is lost), it can be ut layer.
- To create such a laye

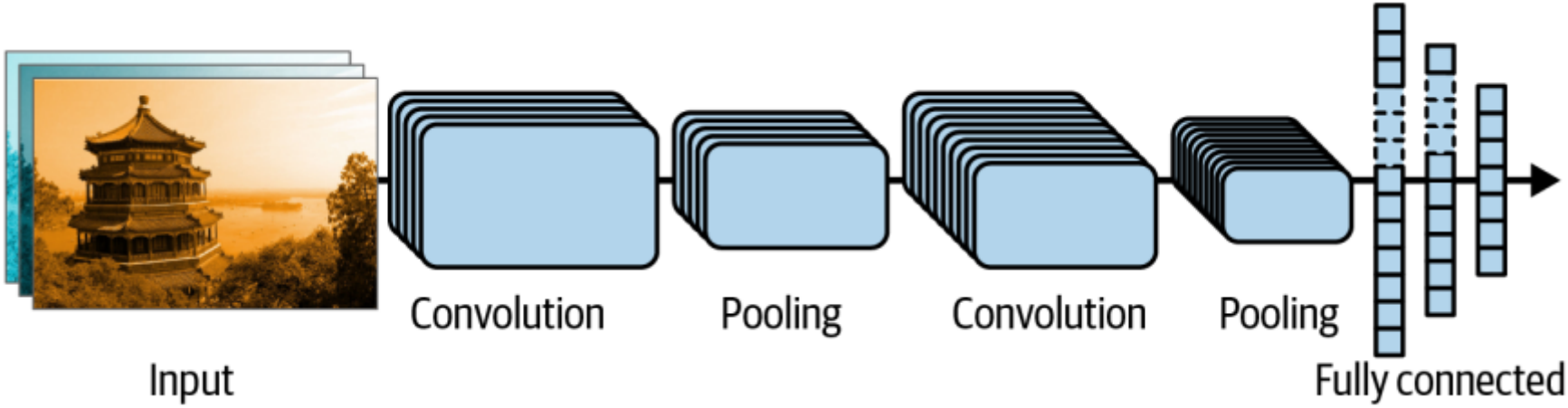


```
global_avg                    vgPool2D()
```

4.
CNN Architectures

Typical CNN Architecture

- Typical CNN stacks a few convolutional layers (each one followed by a ReLU layer), then a pooling layer, repeatedly.
- At the top, a regular feedforward neural network is added.
- The final layer (e.g. a softmax layer) outputs the prediction.



Size of Convolution Kernels

- A common mistake is to use convolution kernels that are too large.
- Instead of using a convolutional layer with a 5×5 kernel, stack two layers with 3×3 kernels.
  - It will use fewer parameters and require fewer computations, and it will usually perform better.
- One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5×5), usually with a stride of 2 or more.
  - This will reduce the spatial dimension of the image without losing too much information, and since the input image only has three channels in general, it will not be too costly.

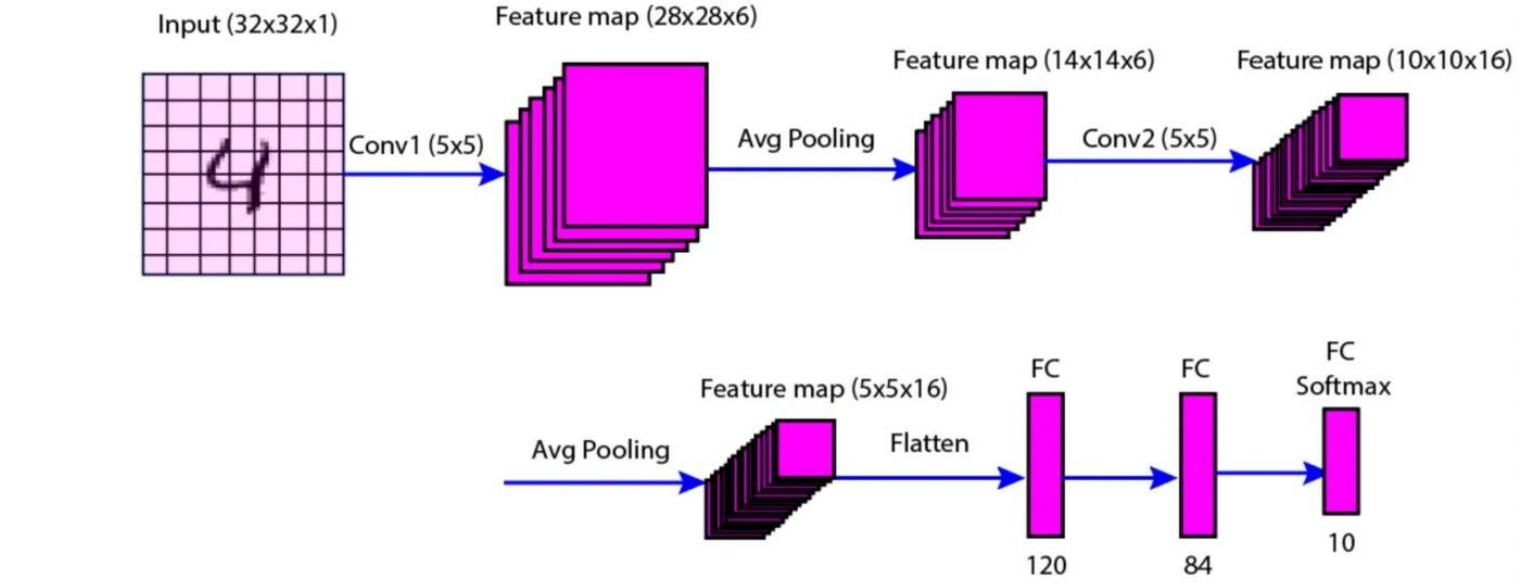
Implement a Basic CNN

```
from functools import partial

DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, padding="same",
                        activation="relu", kernel_initializer="he_normal")
model = tf.keras.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=10, activation="softmax")
])
```

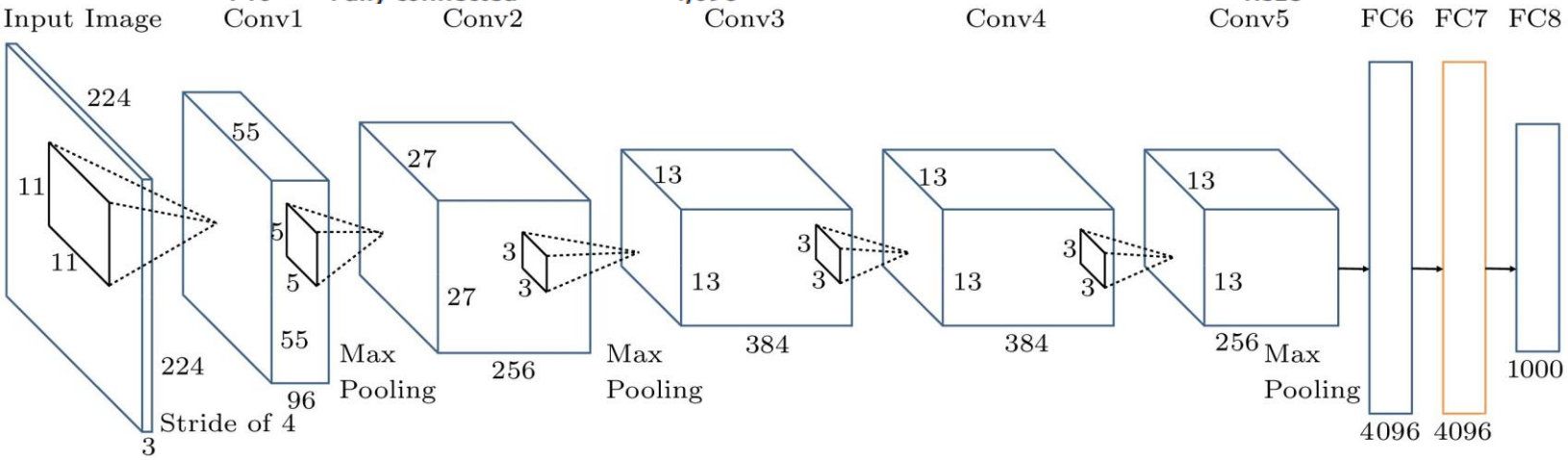
LeNet-5

- The LeNet-5 architecture was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition. It is composed of the following layers:



AlexNet

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU



S2	Max pooling	96	27×27	3×3	2	valid	–
C1	Convolution	96	55×55	11×11	4	valid	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

Data Augmentation

- Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance.