CNN using Keras

Convolutional Layers

From Geron,
The most important building block of a CNN is the *convolutional layer*: neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in the layers discussed in previous chapters), but only to pixels in their receptive fields. In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows 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, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

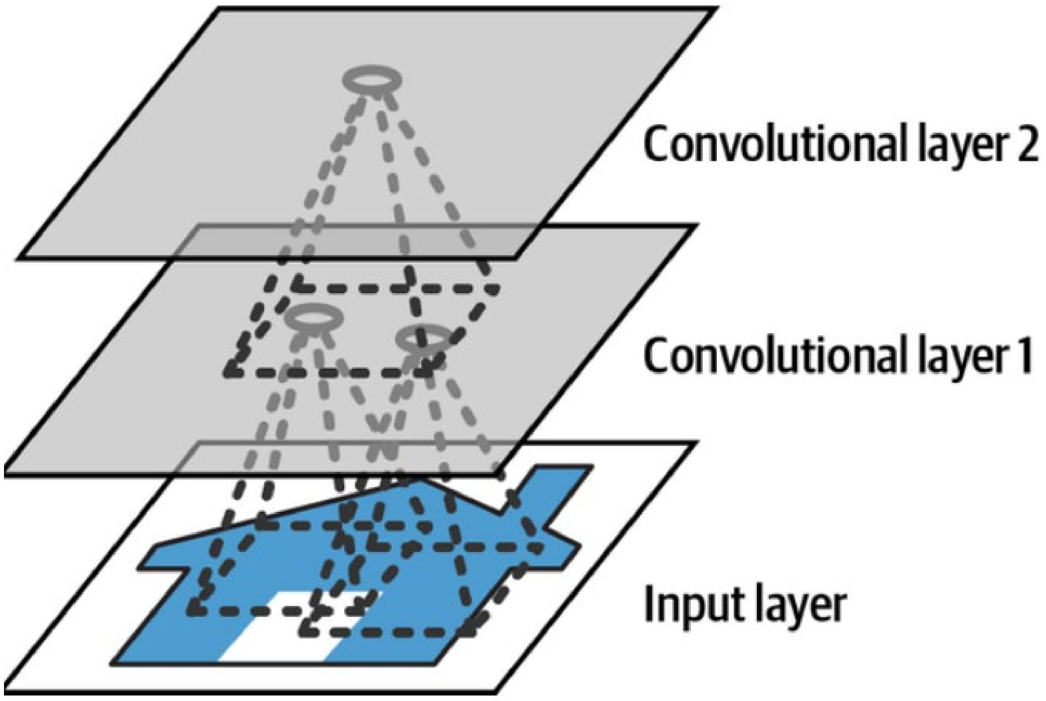

Figure 14-2. CNN layers with rectangular local receptive fields

A neuron located 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, where f_h and f_w are the height and width of the receptive field (see Figure 14-3). 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, as shown in the diagram. This is called *zero padding*.

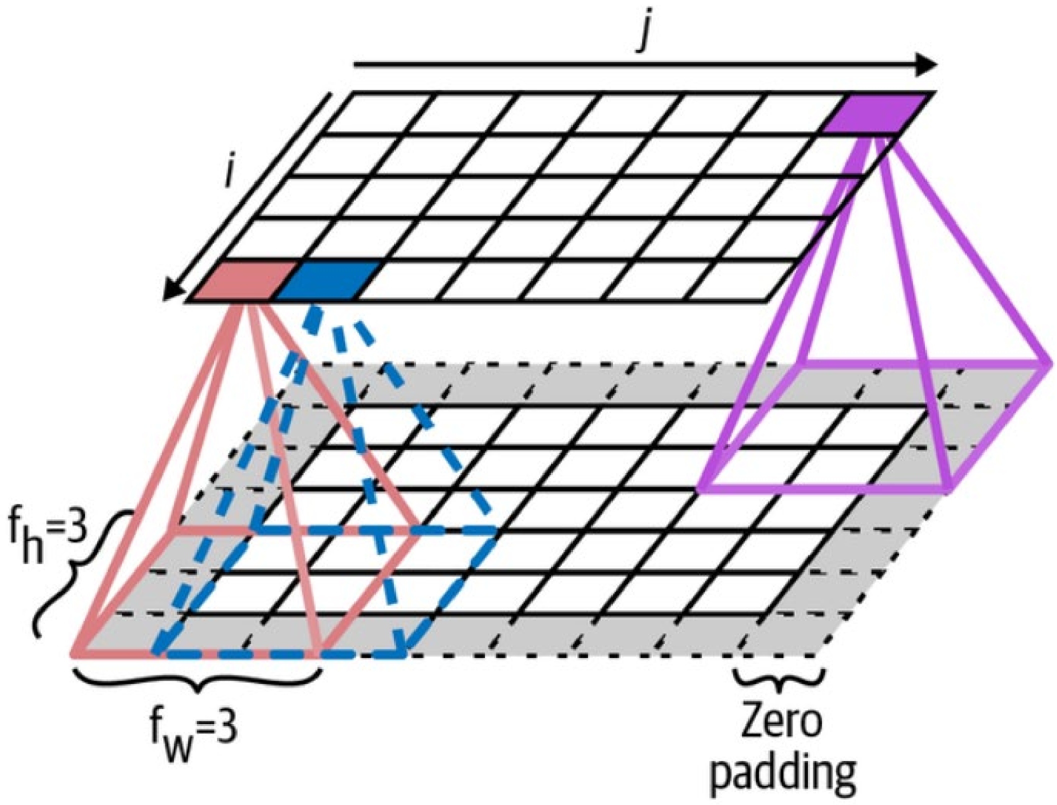

Figure 14-3. Connections between layers and zero padding

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown. This dramatically reduces the model's computational complexity. The horizontal or vertical step size from one receptive field to the next is called the *stride*. In the diagram below, a 5 × 7 input layer (plus zero padding) is connected to a 3 × 4 layer, using 3 × 3 receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h -$ 1, columns $j \times s_w$ to $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

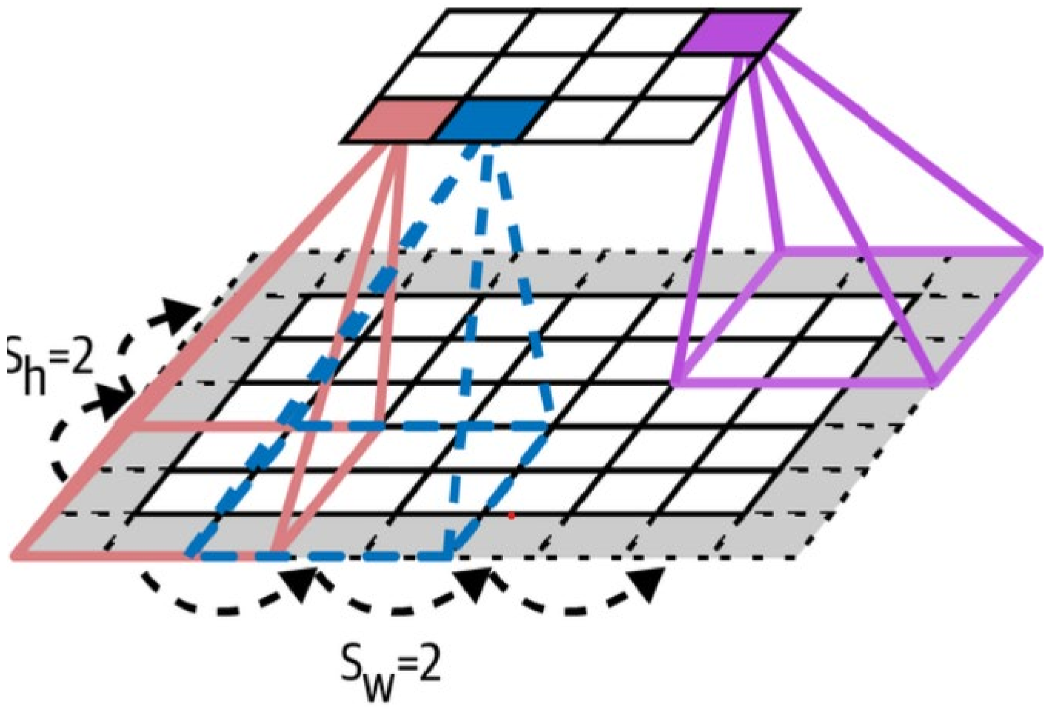

Figure 14-4. Reducing dimensionality using a stride of 2

Filters

A neuron's weights can be represented as a small image of the size of the receptive field. For e.g. Vertical and Horizontal filters. if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown here below (the bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what you get if all neurons use the same horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter outputs a *feature map*, which highlights the areas in an image that activates the filter the most.

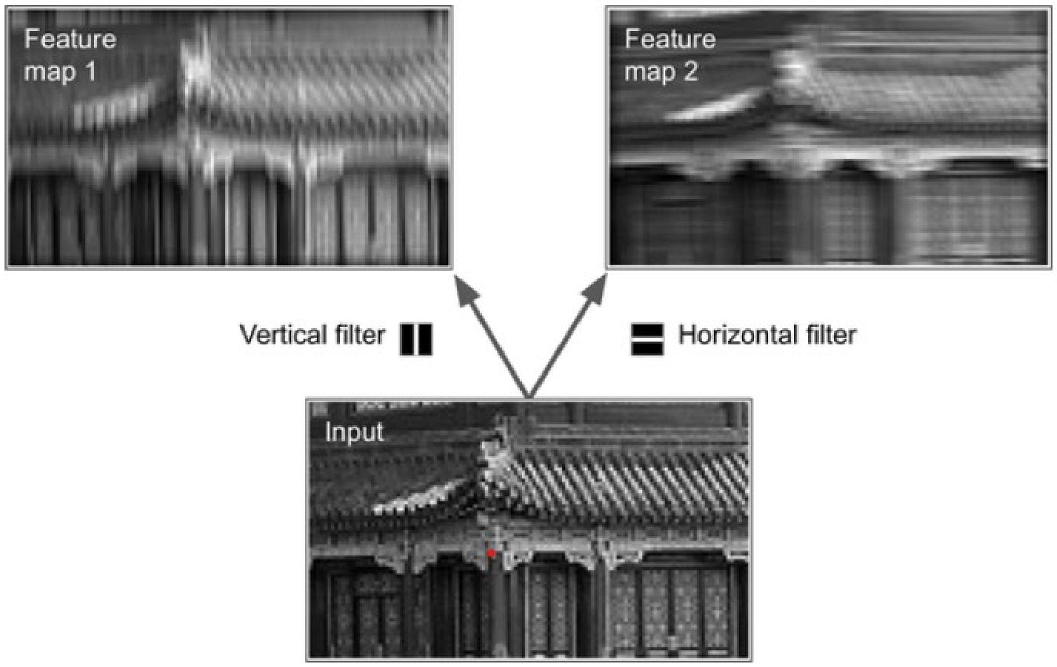

Figure 14-5. Applying two different filters to get two feature maps

Stacking Multiple Feature Maps

A convolutional layer has multiple filters (you decide how many) and outputs one feature map per filter, so it is more accurately represented in 3D

Avoiding Overfitting Through Regularization

1. Early stopping
2. **Batch Normalization:** Adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs. Adds 4 parameters per input: `[gamma weights, beta weights, moving_mean(non-trainable), moving_variance(non-trainable)]`
3. ℓ_1 and ℓ_2 regularization, and max-norm regularization

4. **Dropout:** At every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability *p* of being temporarily “dropped out”, meaning it will be entirely ignored during this training step, but it may be active during the next step. Many state-of-the-art neural networks use dropout, as it gives them a 1%–2% accuracy boost. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

Assignment 2:

1. Model with conv2d

Miscellaneous Notes

`Sequential` model. This is the simplest kind of Keras model for neural networks that are just composed of a single stack of layers connected sequentially. This is called the sequential API.

tf.layers.flatten. Flatten into 1 D vector

Output function for regression:

1. None or
2. ReLU(or softplus for positive outputs)
3. TanH or sigmoid for bounded outputs

Hidden Layers activation functions: ReLU (o/p is +ve)

Loss Function for regression:

MSE or huber (if outliers)

Minimize MSE with Regularization: L2

Output function for classification:

1. Sigmoid activation function (probability between 0-1) for binary classification
2. Softmax activation function for multiclass classification: ensures all probabilities add upto 1

Loss Function for Classification:

1. Cross-entropy loss (or x-entropy or log-loss) is suited for probability distributions.