

nected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits (about 11.4 MB) of RAM.⁸ And that's just for one instance! If a training batch contains 100 instances, then this layer will use up over 1 GB of RAM!

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.



If training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, or removing a few layers. Or you can try using 16-bit floats instead of 32-bit floats. Or you could distribute the CNN across multiple devices.

Now let's look at the second common building block of CNNs: the *pooling layer*.

Pooling Layer

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. Their 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 (thereby limiting the risk of overfitting). Reducing the input image size also makes the neural network tolerate a little bit of image shift (*location invariance*).

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean. **Figure 13-8** shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2×2 *pooling kernel*, a stride of 2, and no padding. Note that only the max input value in each kernel makes it to the next layer. The other inputs are dropped.

⁸ 1 MB = 1,024 kB = 1,024 × 1,024 bytes = 1,024 × 1,024 × 8 bits.

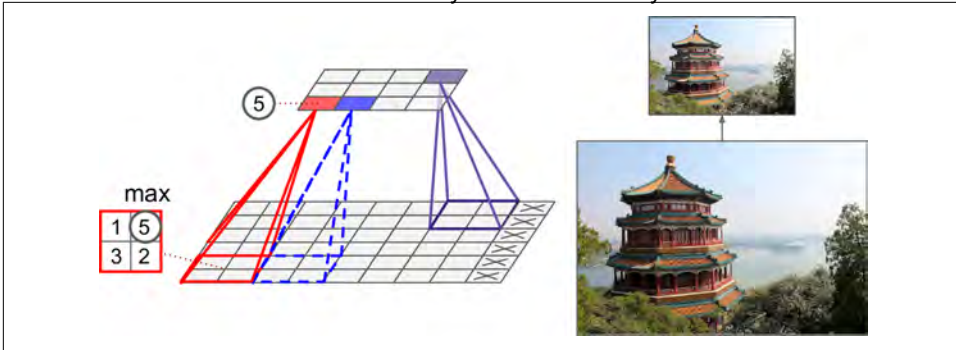


Figure 13-8. Max pooling layer (2×2 pooling kernel, stride 2, no padding)

This is obviously a very destructive kind of layer: even with a tiny 2×2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values.

A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth. You may alternatively pool over the depth dimension, as we will see next, in which case the image's spatial dimensions (height and width) remain unchanged, but the number of channels is reduced.

Implementing a max pooling layer in TensorFlow is quite easy. The following code creates a max pooling layer using a 2×2 kernel, stride 2, and no padding, then applies it to all the images in the dataset:

```
[...] # load the image dataset, just like above

# Create a graph with input X plus a max pooling layer
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8)) # plot the output for the 1st image
plt.show()
```

The `ksize` argument contains the kernel shape along all four dimensions of the input tensor: [batch size, height, width, channels]. TensorFlow currently does not support pooling over multiple instances, so the first element of `ksize` must be equal to 1. Moreover, it does not support pooling over both the spatial dimensions (height and width) and the depth dimension, so either `ksize[1]` and `ksize[2]` must both be equal to 1, or `ksize[3]` must be equal to 1.

To create an *average pooling layer*, just use the `avg_pool()` function instead of `max_pool()`.

Now you know all the building blocks to create a convolutional neural network. Let's see how to assemble them.

CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps) thanks to the convolutional layers (see [Figure 13-9](#)). At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLU), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

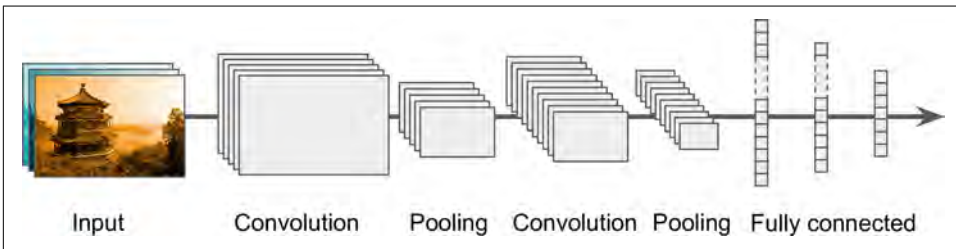


Figure 13-9. Typical CNN architecture



A common mistake is to use convolution kernels that are too large. You can often get the same effect as a 9×9 kernel by stacking two 3×3 kernels on top of each other, for a lot less compute.

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC [ImageNet challenge](#). In this competition the top-5 error rate for image classification fell from over 26% to barely over 3% in just five years. The top-five error rate is the number of test images for which the system's top 5 predictions did not include the correct answer. The images are large (256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds). Looking at the evolution of the winning entries is a good way to understand how CNNs work.

We will first look at the classical LeNet-5 architecture (1998), then three of the winners of the ILSVRC challenge: AlexNet (2012), GoogLeNet (2014), and ResNet (2015).