```
validation_labels = train_labels[:VALIDATION_SIZE]
train_data = train_data[VALIDATION_SIZE:, ...]
train_labels = train_labels[VALIDATION_SIZE:]
```

Choosing the Correct Validation Set

In Example 6-5, we use the final fragment of training data as a validation set to gauge the progress of our learning methods. In this case, this method is relatively harmless. The distribution of data in the test set is well represented by the distribution of data in the validation set.

However, in other situations, this type of simple validation set selection can be disastrous. In molecular machine learning (the use of machine learning to predict properties of molecules), it is almost always the case that the test distribution is dramatically different from the training distribution. Scientists are most interested in *prospective* prediction. That is, scientists would like to predict the properties of molecules that have never been tested for the property at hand. In this case, using the last fragment of training data for validation, or even a random subsample of the training data, will lead to misleadingly high accuracies. It's quite common for a molecular machine learning model to have 90% accuracy on validation and, say, 60% on test.

To correct for this error, it becomes necessary to design validation set selection methods that take pains to make the validation dissimilar from the training set. A variety of such algorithms exist for molecular machine learning, most of which use various mathematical estimates of graph dissimilarity (treating a molecule as a mathematical graph with atoms as nodes and chemical bonds as edges).

This issue crops up in many other areas of machine learning as well. In medical machine learning or in financial machine learning, relying on historical data to make forecasts can be disastrous. For each application, it's important to critically reason about whether performance on the selected validation set is actually a good proxy for true performance.

TensorFlow Convolutional Primitives

We start by introducing the TensorFlow primitives that are used to construct our convolutional networks (Example 6-6).

Example 6-6. Defining a 2D convolution in TensorFlow

```
tf.nn.conv2d(
    input,
    filter,
    strides,
    padding,
    use_cudnn_on_gpu=None,
    data_format=None,
    name=None
)
```

The function `tf.nn.conv2d` is the built-in TensorFlow function that defines convolutional layers. Here, `input` is assumed to be a tensor of shape `(batch, height, width, channels)` where `batch` is the number of images in a minibatch.

Note that the conversion functions defined previously read the MNIST data into this format. The argument `filter` is a tensor of shape `(filter_height, filter_width, channels, out_channels)` that specifies the learnable weights for the nonlinear transformation learned in the convolutional kernel. `strides` contains the filter strides and is a list of length 4 (one for each input dimension).

`padding` controls whether the input tensors are padded (with extra zeros as in Figure 6-18) to guarantee that output from the convolutional layer has the same shape as the input. If `padding="SAME"`, then `input` is padded to ensure that the convolutional layer outputs an image tensor of the same shape as the original input image tensor. If `padding="VALID"` then extra padding is not added.

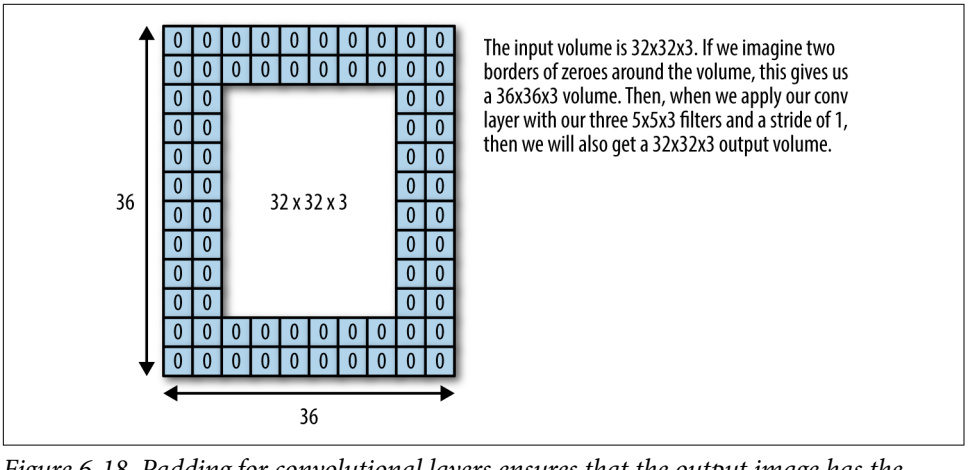

Figure 6-18. Padding for convolutional layers ensures that the output image has the same shape as the input image.

The code in Example 6-7 defines max pooling in TensorFlow.

Example 6-7. Defining max pooling in TensorFlow

```
tf.nn.max_pool(
    value,
    ksize,
    strides,
    padding,
    data_format='NHWC',
    name=None
)
```

The `tf.nn.max_pool` function performs max pooling. Here `value` has the same shape as `input` for `tf.nn.conv2d`, `(batch, height, width, channels)`. `ksize` is the size of the pooling window and is a list of length 4. `strides` and `padding` behave as for `tf.nn.conv2d`.

The Convolutional Architecture

The architecture defined in this section will closely resemble LeNet-5, the original architecture used to train convolutional neural networks on the MNIST dataset. At the time the LeNet-5 architecture was invented, it was exorbitantly expensive computationally, requiring multiple weeks of compute to complete training. Today's laptops thankfully are more than sufficient to train LeNet-5 models. Figure 6-19 illustrates the structure of the LeNet-5 architecture.

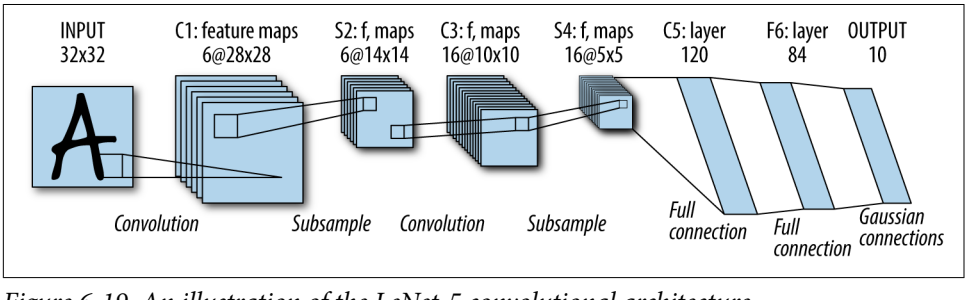

Figure 6-19. An illustration of the LeNet-5 convolutional architecture.