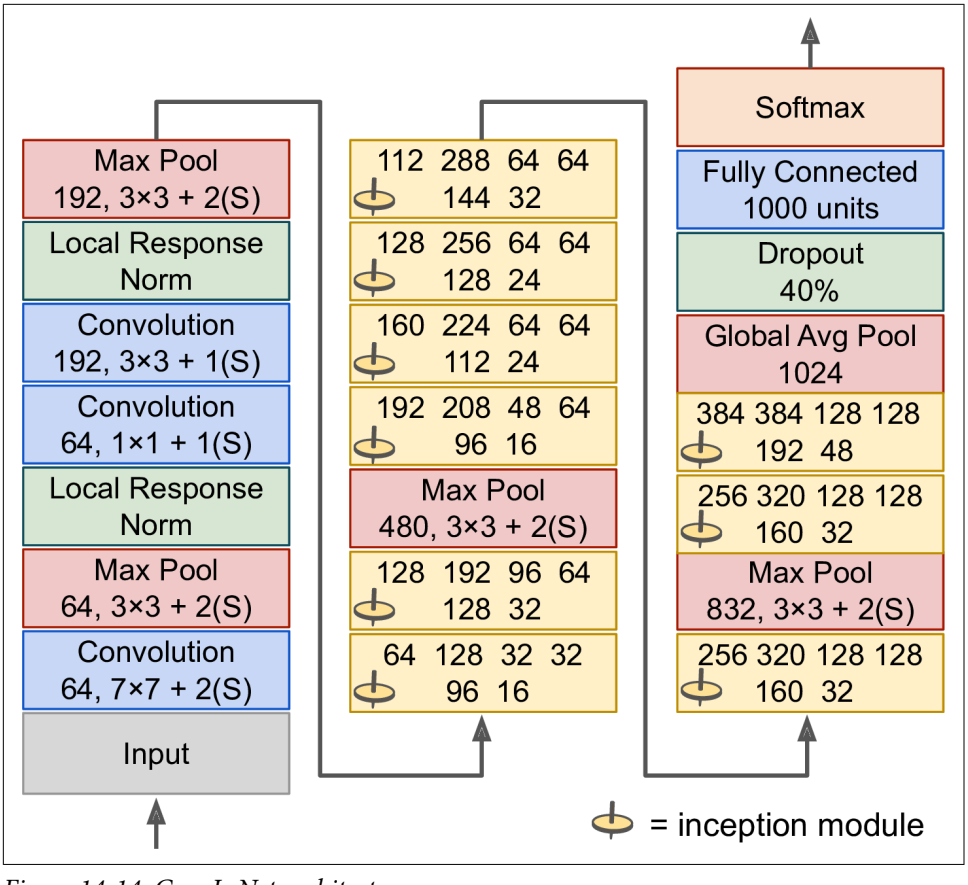

Figure 14-14. GoogLeNet architecture

Let's go through this network:

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load. The first layer uses a large kernel size, so that much of the information is still preserved.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features (as discussed earlier).
- Two convolutional layers follow, where the first acts like a *bottleneck layer*. As explained earlier, you can think of this pair as a single smarter convolutional layer.
- Again, a local response normalization layer ensures that the previous layers capture a wide variety of patterns.

- Next a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the tall stack of nine inception modules, interleaved with a couple max pooling layers to reduce dimensionality and speed up the net.
- Next, the global average pooling layer simply outputs the mean of each feature map: this drops any remaining spatial information, which is fine since there was not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be 224 × 224 pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to 7 × 7. Moreover, it is a classification task, not localization, so it does not matter where the object is. Thanks to the dimensionality reduction brought by this layer, there is no need to have several fully connected layers at the top of the CNN (like in AlexNet), and this considerably reduces the number of parameters in the network and limits the risk of overfitting.
- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with 1,000 units, since there are a 1,000 classes, and a softmax activation function to output estimated class probabilities.

This diagram is slightly simplified: the original GoogLeNet architecture also included two auxiliary classifiers plugged on top of the third and sixth inception modules. They were both composed of one average pooling layer, one convolutional layer, two fully connected layers, and a softmax activation layer. During training, their loss (scaled down by 70%) was added to the overall loss. The goal was to fight the vanishing gradients problem and regularize the network. However, it was later shown that their effect was relatively minor.

Several variants of the GoogLeNet architecture were later proposed by Google researchers, including Inception-v3 and Inception-v4, using slightly different inception modules, and reaching even better performance.

VGGNet

The runner up in the ILSVRC 2014 challenge was VGGNet[14], developed by K. Simonyan and A. Zisserman. It had a very simple and classical architecture, with 2 or 3 convolutional layers, a pooling layer, then again 2 or 3 convolutional layers, a pooling layer, and so on (with a total of just 16 convolutional layers), plus a final dense network with 2 hidden layers and the output layer. It used only 3 × 3 filters, but many filters.

14 "Very Deep Convolutional Networks for Large-Scale Image Recognition," K. Simonyan and A. Zisserman (2015).

ResNet

The ILSVRC 2015 challenge was won using a *Residual Network* (or *ResNet*), developed by Kaiming He et al.,[15] which delivered an astounding top-5 error rate under 3.6%, using an extremely deep CNN composed of 152 layers. It confirmed the general trend: models are getting deeper and deeper, with fewer and fewer parameters. The key to being able to train such a deep network is to use *skip connections* (also called *shortcut connections*): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack. Let's see why this is useful.

When training a neural network, the goal is to make it model a target function $h(\mathbf{x})$. If you add the input $\mathbf{x}$ to the output of the network (i.e., you add a skip connection), then the network will be forced to model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ rather than $h(\mathbf{x})$. This is called *residual learning* (see Figure 14-15).

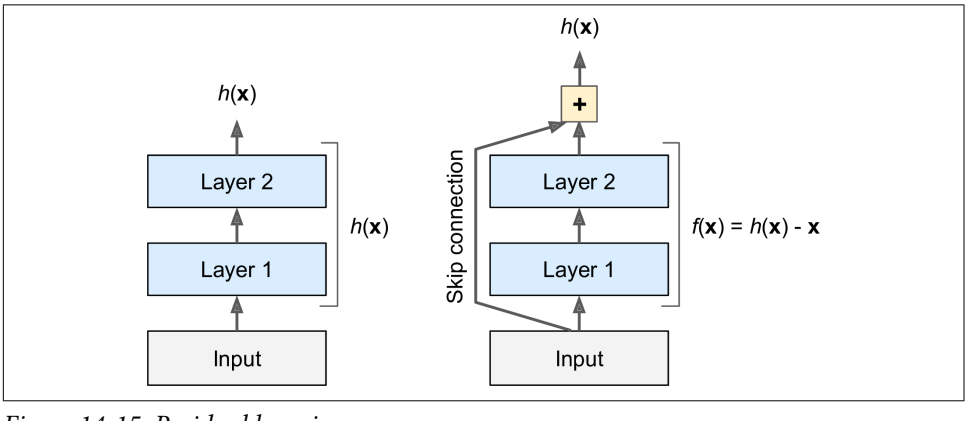

Figure 14-15. Residual learning

When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero. If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function. If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

Moreover, if you add many skip connections, the network can start making progress even if several layers have not started learning yet (see Figure 14-16). Thanks to skip connections, the signal can easily make its way across the whole network. The deep residual network can be seen as a stack of *residual units*, where each residual unit is a small neural network with a skip connection.

15 "Deep Residual Learning for Image Recognition," K. He (2015).

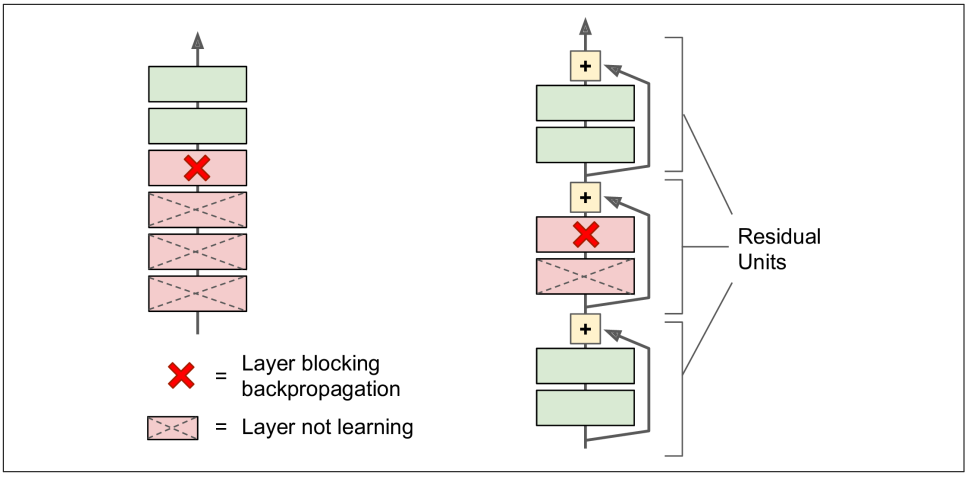

Figure 14-16. Regular deep neural network (left) and deep residual network (right)

Now let's look at ResNet's architecture (see Figure 14-17). It is actually surprisingly simple. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of simple residual units. Each residual unit is composed of two convolutional layers (and no pooling layer!), with Batch Normalization (BN) and ReLU activation, using 3×3 kernels and preserving spatial dimensions (stride 1, SAME padding).

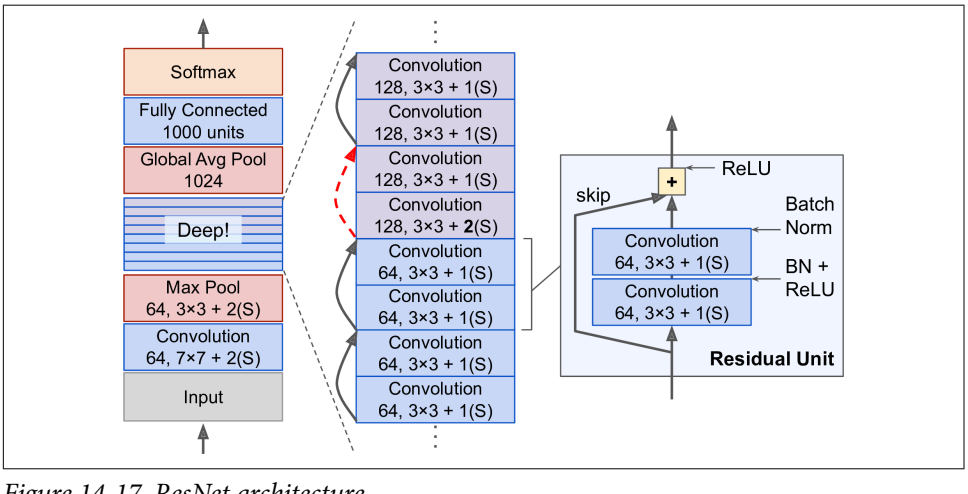

Figure 14-17. ResNet architecture

Note that the number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2). When this happens the inputs cannot be added directly to the outputs of the residual unit since they don't have the same shape (for example, this problem affects the skip

connection represented by the dashed arrow in Figure 14-17). To solve this problem, the inputs are passed through a 1 × 1 convolutional layer with stride 2 and the right number of output feature maps (see Figure 14-18).

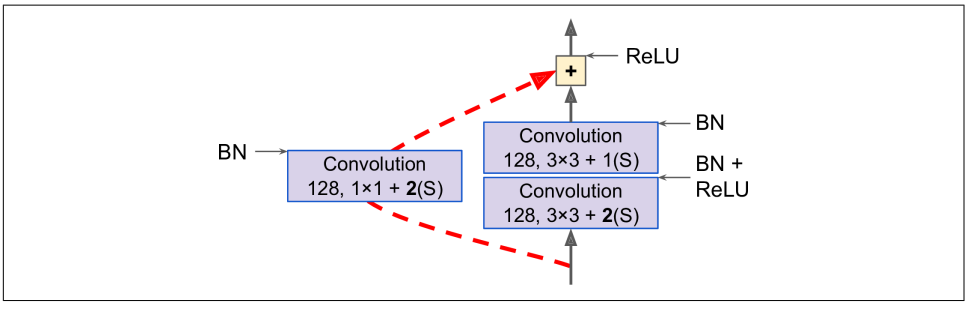

Figure 14-18. Skip connection when changing feature map size and depth

ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer) containing three residual units that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps. We will implement this architecture later in this chapter.

ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two 3 × 3 convolutional layers with (say) 256 feature maps, they use three convolutional layers: first a 1 × 1 convolutional layer with just 64 feature maps (4 times less), which acts as a bottleneck layer (as discussed already), then a 3 × 3 layer with 64 feature maps, and finally another 1 × 1 convolutional layer with 256 feature maps (4 times 64) that restores the original depth. ResNet-152 contains three such RUs that output 256 maps, then 8 RUs with 512 maps, a whopping 36 RUs with 1,024 maps, and finally 3 RUs with 2,048 maps.

Google's Inception-v4[16] architecture merged the ideas of GoogLeNet and ResNet and achieved close to 3% top-5 error rate on ImageNet classification.

Xception

Another variant of the GoogLeNet architecture is also worth noting: Xception[17] (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the

16 "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," C. Szegedy et al. (2016).

17 "Xception: Deep Learning with Depthwise Separable Convolutions," François Chollet (2016)

author of Keras), and it significantly outperformed Inception-v3 on a huge vision task (350 million images and 17,000 classes). Just like Inception-v4, it also merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a *depthwise separable convolution* (or *separable convolution* for short[18]). These layers had been used before in some CNN architectures, but they were not as central as in the Xception architecture. While a regular convolutional layer uses filters that try to simultaneously capture spatial patterns (e.g., an oval) and cross-channel patterns (e.g., mouth + nose + eyes = face), a separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately (see Figure 14-19). Thus, it is composed of two parts: the first part applies a single spatial filter for each input feature map, then the second part looks exclusively for cross-channel patterns—it is just a regular convolutional layer with 1 × 1 filters.

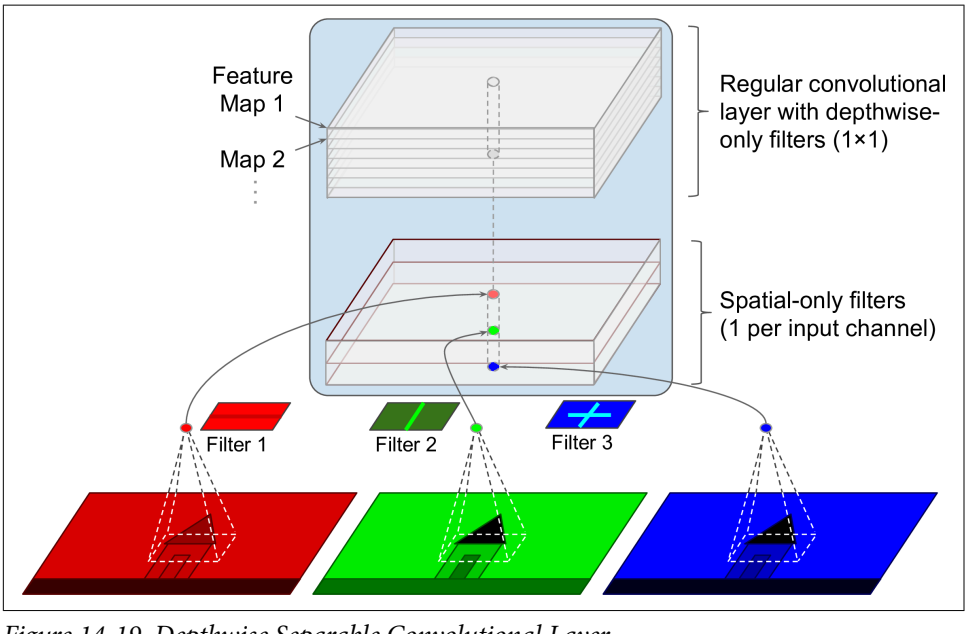

Figure 14-19. Depthwise Separable Convolutional Layer

Since separable convolutional layers only have one spatial filter per input channel, you should avoid using them after layers that have too few channels, such as the input layer (granted, that's what Figure 14-19 represents, but it is just for illustration purposes). For this reason, the Xception architecture starts with 2 regular convolutional layers, but then the rest of the architecture uses only separable convolutions (34 in

18 This name can sometimes be ambiguous, since spatially separable convolutions are often called "separable convolutions" as well.

all), plus a few max pooling layers and the usual final layers (a global average pooling layer, and a dense output layer).

You might wonder why Xception is considered a variant of GoogLeNet, since it contains no inception module at all? Well, as we discussed earlier, an Inception module contains convolutional layers with 1×1 filters: these look exclusively for cross-channel patterns. However, the convolution layers that sit on top of them are regular convolutional layers that look both for spatial and cross-channel patterns. So you can think of an Inception module as an intermediate between a regular convolutional layer (which considers spatial patterns and cross-channel patterns jointly) and a separable convolutional layer (which considers them separately). In practice, it seems that separable convolutions generally perform better.

Separable convolutions use less parameters, less memory and less computations than regular convolutional layers, and in general they even perform better, so you should consider using them by default (except after layers with few channels).

The ILSVRC 2016 challenge was won by the CUImage team from the Chinese University of Hong Kong. They used an ensemble of many different techniques, including a sophisticated object-detection system called GBD-Net[19], to achieve a top-5 error rate below 3%. Although this result is unquestionably impressive, the complexity of the solution contrasted with the simplicity of ResNets. Moreover, one year later another fairly simple architecture performed even better, as we will see now.

SENet

The winning architecture in the ILSVRC 2017 challenge was the Squeeze-and-Excitation Network (SENet)[20]. This architecture extends existing architectures such as inception networks or ResNets, and boosts their performance. This allowed SENet to win the competition with an astonishing 2.25% top-5 error rate! The extended versions of inception networks and ResNet are called *SE-Inception* and *SE-ResNet* respectively. The boost comes from the fact that a SENet adds a small neural network, called a *SE Block*, to every unit in the original architecture (i.e., every inception module or every residual unit), as shown in Figure 14-20.

19 "Crafting GBD-Net for Object Detection," X. Zeng et al. (2016).

20 "Squeeze-and-Excitation Networks," Jie Hu et al. (2017)

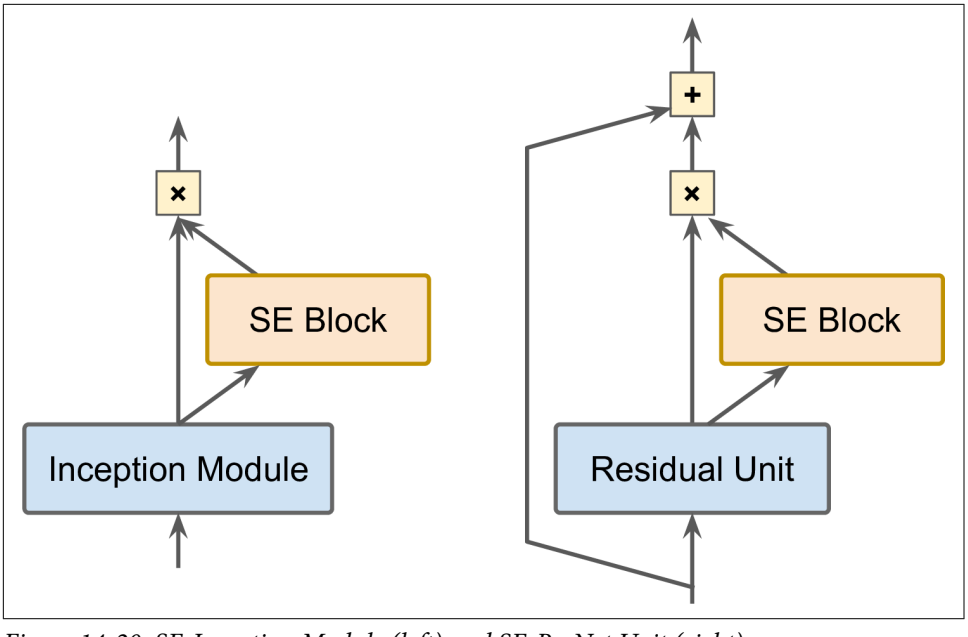

Figure 14-20. SE-Inception Module (left) and SE-ResNet Unit (right)

A SE Block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together. It then uses this information to recalibrate the feature maps, as shown in Figure 14-21. For example, a SE Block may learn that mouths, noses and eyes usually appear together in pictures: if you see a mouth and a nose, you should expect to see eyes as well. So if a SE Block sees a strong activation in the mouth and nose feature maps, but only mild activation in the eye feature map, it will boost the eye feature map (more accurately, it will reduce irrelevant feature maps). If the eyes were somewhat confused with something else, this feature map recalibration will help resolve the ambiguity.

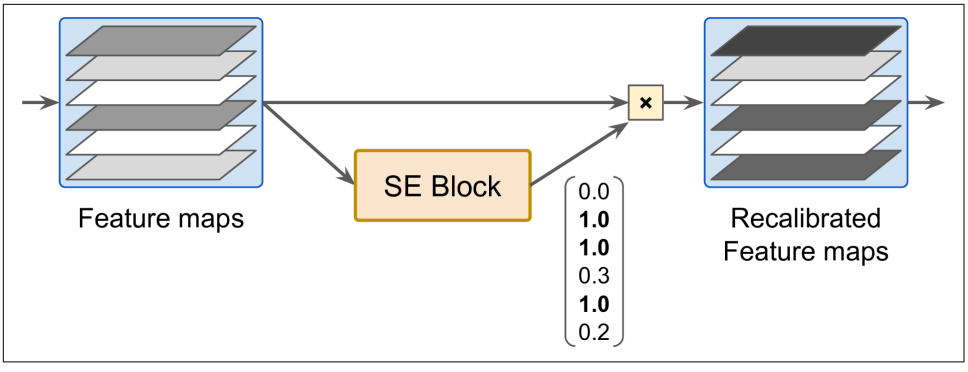

Figure 14-21. An SE Block Performs Feature Map Recalibration

A SE Block is composed of just 3 layers: a global average pooling layer, a hidden dense layer using the ReLU activation function, and a dense output layer using the sigmoid activation function (see Figure 14-22):

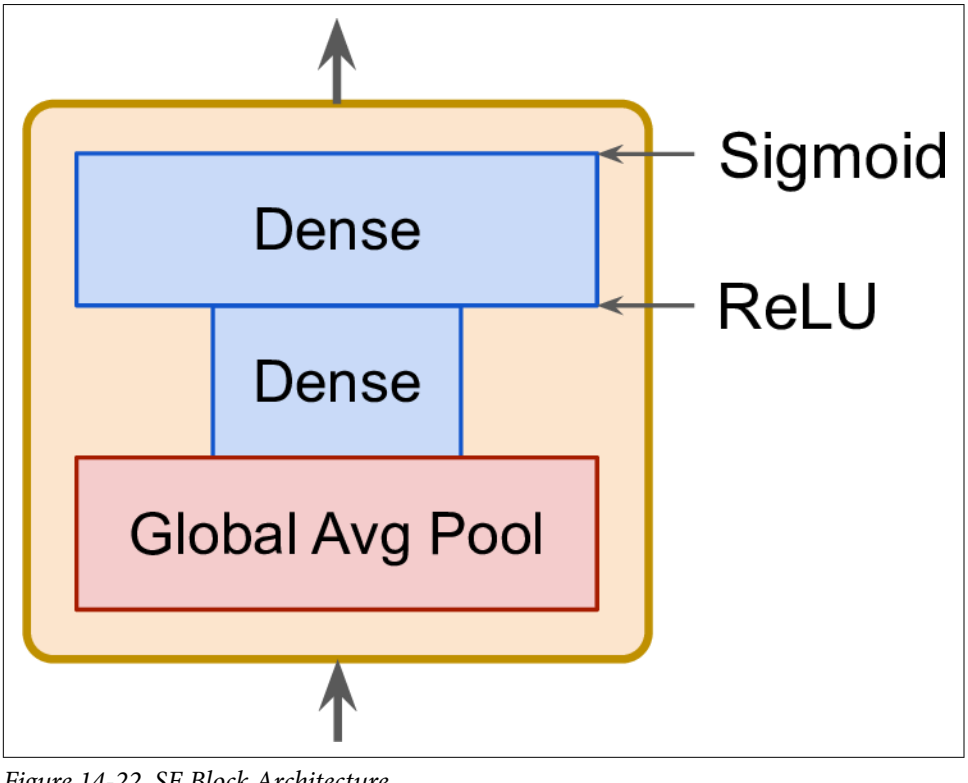

Figure 14-22. SE Block Architecture

As earlier, the global average pooling layer computes the mean activation for each feature map: for example, if its input contains 256 feature maps, it will output 256 numbers representing the overall level of response for each filter. The next layer is where the "squeeze" happens: this layer has much less than 256 neurons, typically 16 times less than the number of feature maps (e.g., 16 neurons), so the 256 numbers get compressed into a small vector (e.g., 16 dimensional). This is a low-dimensional vector representation (i.e., an embedding) of the distribution of feature responses. This bottleneck step forces the SE Block to learn a general representation of the feature combinations (we will see this principle in action again when we discuss autoencoders in ???). Finally, the output layer takes the embedding and outputs a recalibration vector containing one number per feature map (e.g., 256), each between 0 and 1. The feature maps are then multiplied by this recalibration vector, so irrelevant features (with a low recalibration score) get scaled down while relevant features (with a recalibration score close to 1) are left alone.

Implementing a ResNet-34 CNN Using Keras

Most CNN architectures described so far are fairly straightforward to implement (although generally you would load a pretrained network instead, as we will see). To illustrate the process, let's implement a ResNet-34 from scratch using Keras. First, let's create a `ResidualUnit` layer:

```
DefaultConv2D = partial(keras.layers.Conv2D, kernel_size=3, strides=1,
                        padding="SAME", use_bias=False)

class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size=1, strides=strides),
                keras.layers.BatchNormalization()]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
```