Table of Contents

Convolutional Neural Networks
1

In this chapter, we introduce the first of various specialized deep learning architectures that we will cover in part four. Deep **Convolutional Neural Networks** (**CNNs**), also known as **ConvNets**, have enabled superhuman performance in classifying images, video, speech, and audio. Recurrent nets, the subject of the following chapter, have performed exceptionally well on sequential data such as text and speech.

CNNs are named after the linear algebra operation called **convolution**, which replaces the general matrix multiplication typical of feedforward networks (see `Chapter 16`, *Deep Learning*) in at least one of their layers. We will discuss how convolutions work, and why they are particularly useful to data with a certain regular structure, such as images and time series.

Research into CNN architectures has proceeded very rapidly, and new architectures that improve benchmark performance continue to emerge frequently. We will describe a set of building blocks that consistently appear in successful applications, and illustrate their application to image data and financial time series. We will also present how transfer learning can speed up learning by using pre-trained weights for some of the CNN layers.

As we will discuss in the first section, CNNs are designed to learn hierarchical feature representations from grid-like data. One of their shortcomings is that they do not learn spatial relationships; that is, the relative positions of these features. In the last section, we will outline how capsule networks that have emerged to overcome these limitations work.

More specifically, in this chapter, we will cover the following topics:

- How CNNs use key building blocks to efficiently model grid-like data
- How to design CNN architectures using Keras and PyTorch
- How to train, tune, and regularize CNN for various data types
- How to use transfer learning to streamline CNN, even with less data
- How capsule networks improve on CNN, and may enable a new wave of innovation

References and code examples for this chapter are in the corresponding directory of this book's GitHub repository: `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`.

How ConvNets work

CNNs are conceptually similar to the feedforward neural networks we covered in the previous chapter. They consist of units that contain parameters, called **weights** and **biases**, and the training process adjusts these parameters to optimize the network's output for a given input. Each unit applies its parameters to a linear operation on the input data or activations received from other units, possibly followed by a non-linear transformation.

The overall network models a differentiable function that maps raw data, for example, image pixels to class probabilities, using an output activation such as the softmax function. CNNs also use a loss function such as cross-entropy to compute a single quality metric from the output layer, and use the gradients of the loss with respect to the network parameter to learn.

Feedforward neural networks with fully-connected layers do not scale well to high-dimensional image data that has a large number of pixel values. Low-resolution images, such as those included in the CIFAR-10 dataset that we'll use in the next section, are 32 8-bit pixels wide and high so that, with three channels, a single unit in the input layer needs 32 x 32 x 3 = 3,072 weights. A standard resolution of 640 x 480 already yields closer to 1 million weights for a single input unit. Deep architectures with several layers of meaningful width, each quickly lead to an exploding number of parameters, which make overfitting during training all but certain.

CNNs differ, because they encode the assumption that the input has a structure most commonly found in image data where pixels form a two-dimensional grid, typically with several channels to represent the components of the color signal, such as the red, green, and blue channels of the RGB color model.

While initial CNN applications focused on image data, researchers realized over time that a wider range of data sources has a similar, grid-like topology, broadening the scope for the productive use of CNN. For example, time-series data consists of measurements at regular intervals, resulting in a one-dimensional grid along the time axis, and the notion of multiple channels carries to the case of multivariate time series. A common use case includes audio data, either as one-dimensional waveform data in the time domain or, after a Fourier transform, as a two-dimensional spectrum in the frequency domain. CNNs also play a key role in AlphaGo, the first algorithm to win the game of Go against humans, where they evaluated different positions on the grid-like board.

The most important element to encode the assumption of a grid-like topology is the convolution operation that gives CNNs their name, combined with pooling. We will see that the specific assumptions about the functional relationship between input and output data implies that CNNs need far fewer parameters, and compute more efficiently.

In this section, we will explain how convolution and pooling work, and why these operations are particularly suitable for data with only the structure described. Deep CNNs that yield state-of-the-art performance combine many of these basic building blocks in order to achieve the layered representation learning described in the last chapter. We conclude this section by describing key innovations since the AlexNet architecture became the first CNN to win the **ImageNet Large Scale Visual Recognition Challenge** (**ILSVRC**) in 2012, an event often credited with putting deep learning on the global technology map.

How a convolutional layer works

Fully-connected feedforward neural networks make no assumptions about the topology or local structure of the input data, so that arbitrarily reordering the features has no impact on the training result.

For many data sources, however, local structure is quite significant. Examples include autocorrelation in time series, or the spatial correlation among pixel values due to common patterns such as edges or corners. For image data, this local structure has traditionally motivated the development of hand-coded filter methods that extract local patterns for use as features in **machine learning** (**ML**) models.

The following diagram illustrates the effect of simple filters that detect basic edges. The `filter_example` notebook illustrates how to use hand-coded filters in a convolutional network, and visualize the resulting transformation of the image. The filters are shown in the upper right beneath and are simple [-1, 1] patterns arranged in a 2 x 2 matrix. The effect is somewhat subtle, and may be easier to spot in the accompanying notebook:
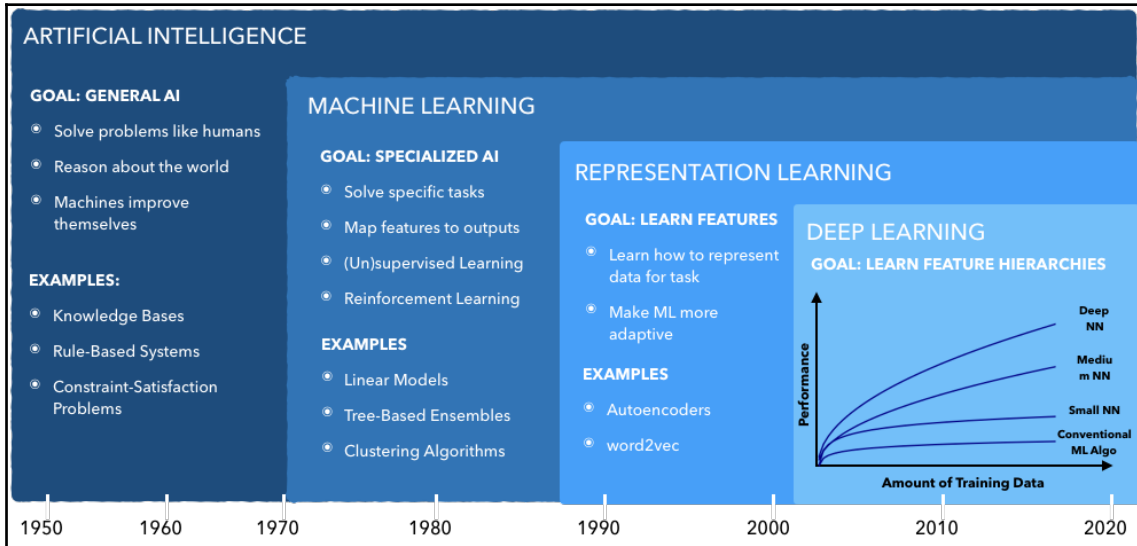
Convolutional layers, in contrast, are designed to learn such local feature representations from the data. A key insight is to restrict their input, called **receptive field**, to a small area of the input, so that it captures basic pixel constellations that reflect common patterns, such as edges or corners. Such patterns may occur anywhere in an image, though, so that CNNs also need to recognize similar patterns in different locations, and possibly with small variations. Subsequent layers then learn to synthesize these local features to detect higher-order features. The Python script `conv_filter_viz` creates a visualization of the filters learned by a deep CNN using the VGG16 architecture that we present in the section on *Reference ConvNet architectures*.

Conceptually, convolutional layers integrate three architectural ideas that enable a CNN to learn feature representations, which are, to some degree, invariant to shifts, changes in scale, and distortion: sparse rather than dense connectivity, shared weights, and spatial or temporal downsampling. Moreover, convolutional layers allow for inputs of variable size. We will walk through a typical convolutional layer, and describe each of these ideas in turn:
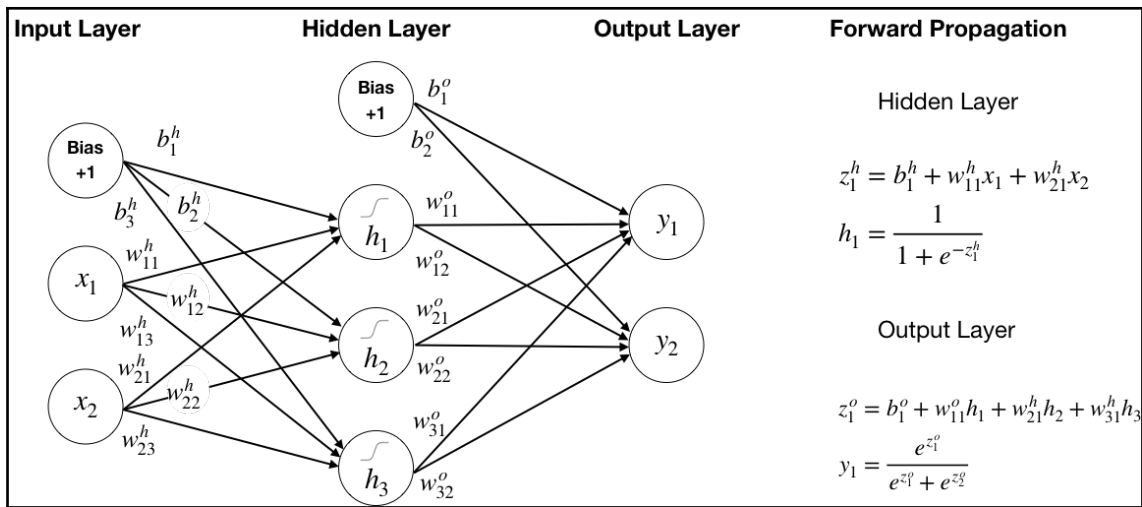
The preceding diagram outlines the set of operations that typically take place in a convolutional layer, assuming image data inputs with the three dimensions height, width, and depth, or number of channels. The range of pixel values depends on the bit representation, for example [0, 255] for 8 bits.

Successive computations process the input through the convolutional, detector, and pooling stages. In the case of image data, a convolutional layer receives three-dimensional input, and produces an output of the same dimensionality.

We will see how state-of-the-art CNNs are composed of several of these layers, either stacked on top of each other, as for a feedforward neural network, or including different branches. With each layer, the network can detect higher-level, more abstract features.

The convolution stage – detecting local features

The first stage successively applies a filter, also called **kernel**, to overlapping patches of the input image. The filter is a matrix of much smaller size than the input, so its receptive field is limited to a few contiguous pixels of the input. Hence, it focuses on local patterns, and reduces the number of parameters and computation relative to a fully-connected layer.

A complete convolutional layer has several feature maps organized in depth slices, so that multiple features can be extracted at each location.

The convolution operation

While scanning the input, the kernel is convolved with each input segment covered by its receptive field. The convolution operation used in CNN corresponds to the dot product between the filter and the target input area after both have been reshaped to vectors.

Each convolution results in a single number, and the result of the entire scan is a feature map that indicates activated input regions.

The following diagram illustrates the result of a scan of a 5 x 5 input, using a 3 x 3 filter, and how the activation in the upper-right corner of the feature map results from the dot product of the flattened input region and the kernel:
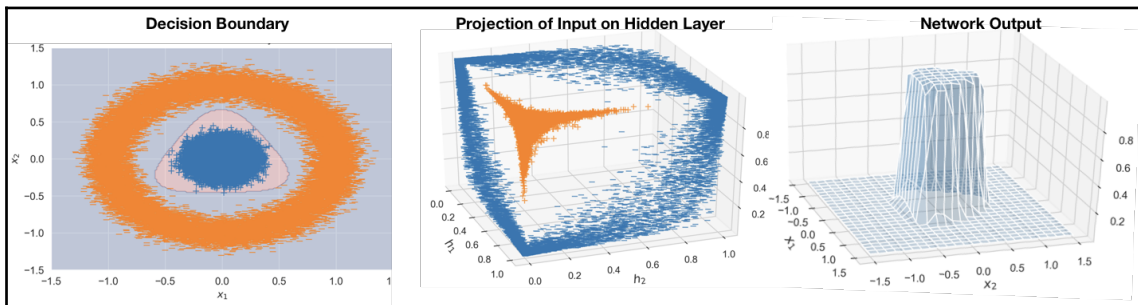
The most important aspect is that the filter values are the parameter of the convolutional layers, learned from the data during training to minimize the chosen loss function. In other words, CNNs learn useful feature representations by finding kernel values that activate input patterns most useful for the task at hand.

How to scan the input – strides and padding

The stride defines the step size used for scanning the input, that is, the number of pixels to shift horizontally and vertically. Smaller strides scan more (overlapping) areas, but are computationally more expensive.

The following three options are commonly used when the filter does not fit the input perfectly, and partially crosses the image boundary during the scan:

- **Valid convolution**: Discard scans where image and filter do not perfectly match
- **Same convolution**: Zero-pad the input to produce a feature map of equal size
- **Full convolution**: Zero-pad the input so that each pixel is scanned an equal number of times, including pixels at the border (to avoid oversampling pixels closer to the center)

The choices depend on the nature of the data, and where useful features are most likely located. In combination with the number of depth slices, they determine the output size of the convolution stage. The Stanford Lecture notes by Andrew Karpathy (see GitHub: `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`) contain helpful examples using NumPy.

Parameter sharing

As stated earlier, the location of salient features may vary due to distortions or shifts. Furthermore, elementary feature detectors are likely to be useful across the entire image.

CNNs encode these assumptions by sharing or tying the weights for the filter in a given depth slice.

As a result, each depth slice specializes in a certain pattern and the number of parameters is further reduced. Weight sharing does not work as well, however, when images are spatially centered, and key patterns are less likely to be uniformly distributed across the input area.

The detector stage – adding non-linearity

The feature maps are usually passed through a non-linear transformation. The **rectified linear unit** (**ReLU**) that we encountered in the last chapter is a common function for this purpose. ReLUs replace negative feature map activations element-wise by zero.

A popular alternative is the Softplus function:

$$f(x) = \ln(1 + e^x)$$

In contrast to ReLU, it has a derivative everywhere, which is the sigmoid function that we used for logistic regression (see `Chapter 7`, *Linear Models*).

The pooling stage – downsampling the feature maps

The last stage of the convolutional layer downsamples the input representation learned by the feature map, to reduce its dimensionality and prevent overfitting, lower the computational cost, and enable basic translation invariance. The precise location of the learned features is not only less important for identifying a pattern or object, it can even be harmful, because the locations will likely vary for different instances of the target. Pooling lowers the spatial resolution of the feature map as a simple way to render the location information less precise.

It is optional, however, and many architectures do not use pooling, or only use it for some convolutional layers.

Max pooling

A common pooling operation is max pooling that downsamples a feature map by using only the maximum activation value from (typically) non-overlapping sub-regions. For a small 4 x 4 feature map, 2 x 2 max pooling would compute the max for each of the four non-overlapping 2 x 2 areas, and output the result.

Alternative pooling operators use the average or the median instead of the maximum, but the latter is most common. Pooling does not add or learn new parameters, but the size of the input window, and possibly the stride, are additional hyperparameters.

Inspiration from neuroscience

CNNs are a prominent example of biologically inspired **Artificial Intelligence** (**AI**). While many other fields influenced the development of CNN, some of their key design principles were drawn from neuroscience. In particular, CNNs reflect key insights from research on the primary visual cortex.

The visual cortex it is organized as a spatial map, which mirrors the structure of the image on the retina. It contains a complex arrangement of cells that are sensitive to small sub-regions of the visual field (called **receptive field**), and are tiled to cover the entire visual field. These cells act as local filters over the input space, and are well-suited to exploit the strong spatially local correlation present in natural images.

Additionally, two basic cell types have been identified. *Simple cells* respond maximally to specific edge-like patterns within their receptive field. *Complex cells* have larger receptive fields, and are locally invariant to the exact position of the pattern.

However, there are also many differences between CNN and the human vision system, much of which remains poorly understood. Nonetheless, the parallels provide at least some heuristic explanation for the impressive performance of CNN on vision tasks.

Reference ConvNet architectures

Several CNN architectures have pushed performance boundaries over the past two decades by introducing important innovations. Predictive performance growth accelerated dramatically with the arrival of big data in the form of ImageNet, with 14 million images assigned to 20,000 classes by humans using Amazon MTurk. The ILSVRC became the focal point of CNN progress around a slightly smaller set of 1.2 million images from 1,000 classes.

It is useful to be familiar with the reference architectures resulting from these competitions for practical reasons. As we will see in the next section, *Transfer learning – faster training with less data*, many computer vision tasks can be addressed by building on a successful architecture with pre-trained weights. Transfer learning not only speeds up architecture selection and training, but also enables successful applications on much smaller datasets.

In addition, many publications refer to these architectures, and they often serve as a basis for networks tailored to segmentation or localization tasks.

LeNet5 – the first modern CNN (1998)

Yann LeCun, now the Director of AI Research at Facebook, was a leading pioneer in CNN development. In 1998, after several iterations starting in the 1980s, LeNet5 became the first modern CNN used in real-world applications, which introduced several architectural elements still relevant today.

LeNet5 was published in a very instructive paper, *Gradient-Based Learning Applied to Document Recognition* (co-authored by Yoshua Bengio and others: `http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf`), that laid out many of the central concepts. Most importantly, it promoted the insight that convolutions with learnable filters are effective at extracting related features at multiple locations with few parameters. Given limited computational resources at the time, efficiency was of paramount importance.

LeNet5 was designed to recognize the handwriting on checks, and was used by several banks. It was state-of-the-art, with classification accuracy of 99.2% on the MNIST digit dataset:
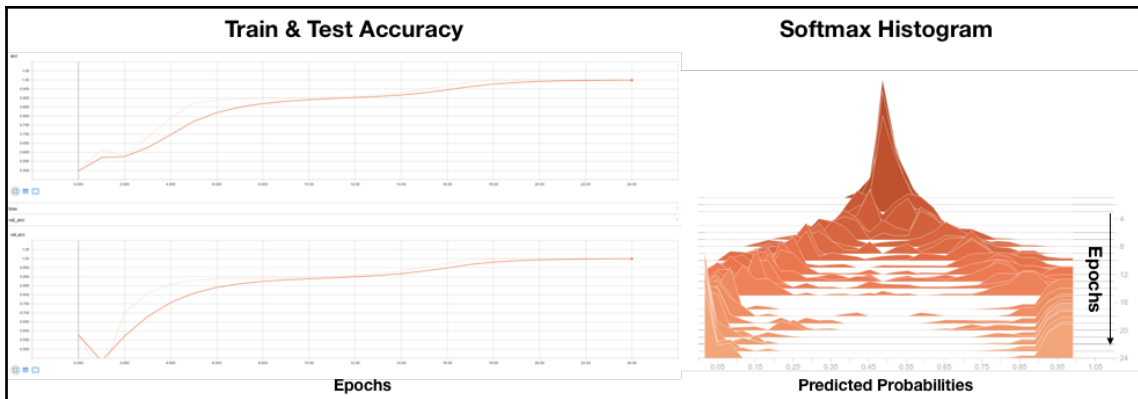
Source: LeCun, Bengio et al, 1998

It consists of three convolutional layers, each containing a non-linear tanh transformation, a pooling operation, and a fully connected output layer. Throughout the convolutional layers, the number of feature maps increases, while their dimensions decrease. It has a total of 60,850 trainable parameters.

The `mnist_with_ffnn_and_lenet5` notebook contains a slightly simplified Keras implementation that, after 10 epochs, achieves 99.0% test accuracy on a 28 x 28 MNIST digit dataset. See the section on CNN with Keras and PyTorch for more details.

AlexNet – putting CNN on the map (2012)

AlexNet, developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton at the University of Toronto, significantly outperformed the runner-up at the 2012 ILSVRC (top 5 error of 16% versus 26%). The performance breakthrough achieved by AlexNet triggered a renaissance of ML research, and put deep learning for computer vision firmly on the global technology map.

The AlexNet architecture is similar to LeNet, but much deeper and wider, and included convolutions stacked on top of each other rather than combining each convolution with a pooling stage.

It discovered the importance of depth, and successfully used dropout for regularization and ReLU as efficient non-linear transformations. It also employed data augmentation to increase the number of training samples, added weight decay, and used a more efficient implementation of convolutions. It also accelerated training by distributing the network over two GPUs.

It has around 60 million parameters, exceeding LeNet5 by a factor of 1,000; a testament to increased computing power, not least due to the use of GPU, and much larger datasets.

The ILSVRC 2013 winner was ZF Net, named after their creators Rob Fergus and Matthew Zeiler, who subsequently started Clarifai. It improved on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers, and reducing the stride and filter size on the first layer.

The `cifar10_image_classification` notebook has a slightly simplified version of AlexNet tailored to the CIFAR10 dataset, which contains 60,000 images with 32 x 32 pixel resolution from 10 classes. It achieves 76.84% top-1 accuracy after only 20 epochs that take less than 10 minutes on a single GPU. The original network was trained for several days on two GPUs using over 1 million images for around 85% top-1 accuracy.

VGGNet – going for smaller filters

The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman from Oxford University. It demonstrated the effectiveness of much smaller 3 x 3 convolutional filters combined in sequence, and reinforced the importance of depth for strong performance. It contains 16 convolutional and fully-connected layers that only perform 3 x 3 convolutions and 2 x 2 pooling:
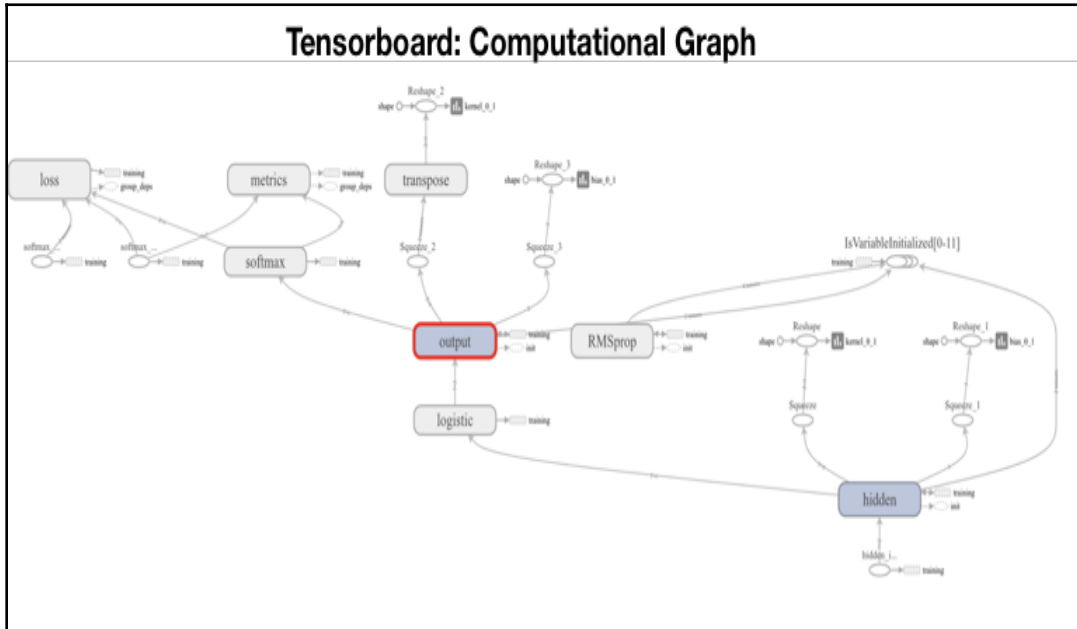

VGG16 has 140 million parameters that increase the computational costs of training and inference, as well as memory requirements. However, most parameters are in the fully connected layers that were since discovered to be not essential, so that removing them greatly reduces the number of parameters without negatively impacting performance.

GoogLeNet – fewer parameters through Inception

Christian Szegedy and his team at Google began working on more efficient CNN implementations that reduce the computational costs to facilitate practical applications at scale. The resulting GoogLeNet won the ILSVRC 2014 with only 4 million parameters, due to the *Inception module*, compared to AlexNet's 60 million and VGG's 140 million.

The Inception module builds on the network-in-network concept that uses 1 x 1 convolutions to compress a deep stack of convolutional filters, and reduce the cost of computation. The module uses parallel 1 x 1, 3 x 3, and 5 x 5 filters, but combines the latter two with 1 x 1 convolutions to reduce the dimensionality of the filters passed in by the previous layer. The left panel in the following diagram illustrates the module's architecture as published by the authors, highlighting the 1 x 1 convolutions used for dimensionality reduction:
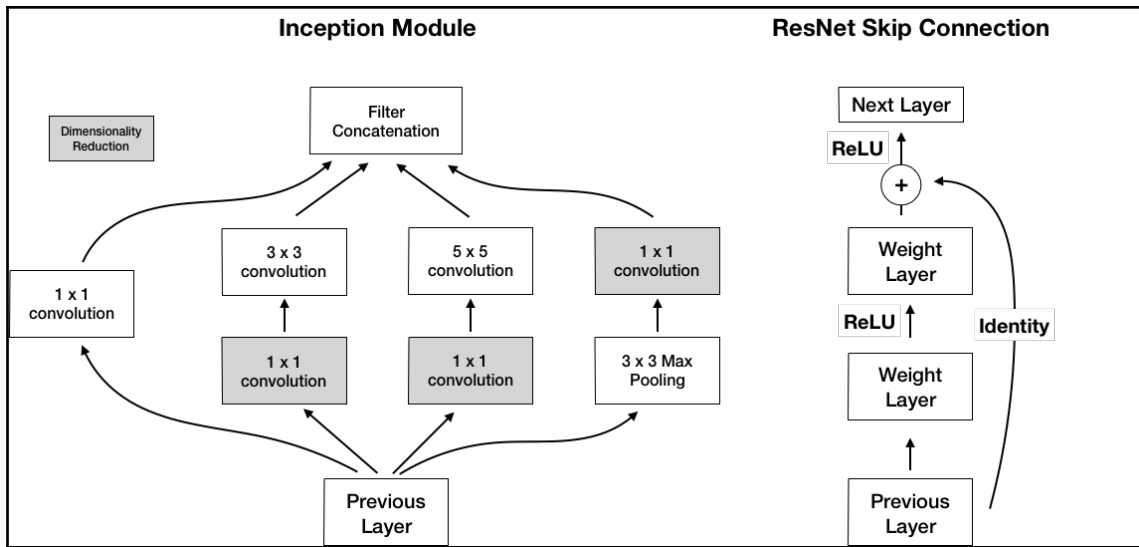
In addition, it uses average pooling instead of fully connected layers on top of the convolutional layers to eliminate many of the less impactful parameters. There have been several enhanced versions, most recently Inception-v4.

ResNet – current state-of-the-art

The residual network architecture was developed by Kaiming He and others at Microsoft Research, and won the ILSVRC 2015. It pushed the top-5 error to 3.7%, below the human level performance of around 5%.

It introduces identity shortcut connections that skip several layers, and overcome some of the challenges of training deep networks, enabling the use of hundreds or even over a thousand layers. It also heavily uses batch normalization that had been shown to allow for higher learning rates, and be more forgiving about weight initialization. The architecture also omits fully connected final layers.

The right panel in the preceding diagram shows the use of skip connections that bypass two layers, before adding the original output to the values produced by the intermediate, skipped layers.

As discussed in the last chapter, the training of deep networks faces the notorious vanishing gradient challenge: as the gradient propagates to earlier layers, repeated multiplication of small weights risks shrinking the gradient towards zero. Hence, increasing depth may limit learning.

The shortcut connection that skips two or more layers has become one of the most popular developments in CNN architectures, and has triggered numerous research efforts to refine and explain its performance. See references on GitHub for additional detail.

Benchmarks

The following diagram on the left plots the top-1 accuracy against the computational cost of a variety of network architectures. It suggests a positive relationship between the number of parameters and performance, but also shows that the marginal benefit of more parameter declines, and that architectural design and innovation also matter.

The diagram on the right plots the top-1 accuracy per parameter for all networks. Several new architectures target use cases on less powerful devices such as mobile phones. While they do not achieve state-of-the-art performance, they have found much more efficient implementations. See references on GitHub for more details on these architectures, and the analysis behind these charts:
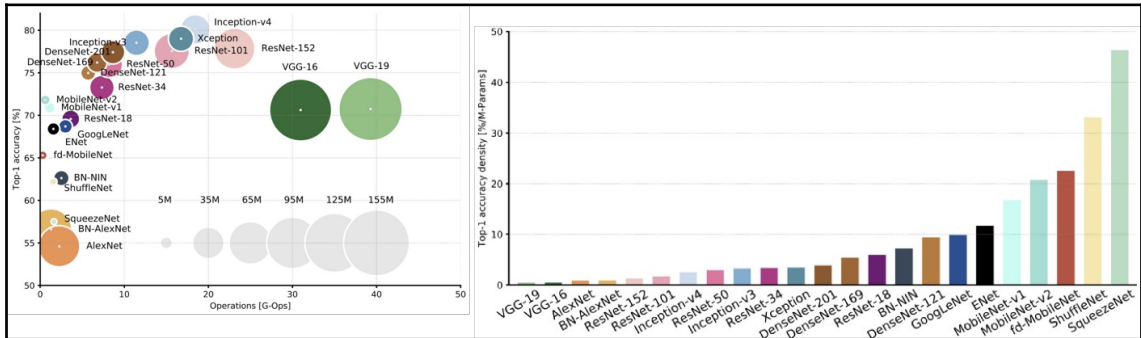
Lessons learned

Some of the lessons learned from twenty years of CNN architecture developments, and especially since 2012, include the following:

- Smaller convolutional filters perform better (possibly except at the first layer) because several small filters can substitute for a larger filter at a lower computational cost
- 1 x 1 convolutions reduce the dimensionality of feature maps so that the network can learn a larger number overall
- Skip connections are able to create multiple paths through the network, and enable the training of much higher-capacity CNNs

Computer vision beyond classification – detection and segmentation

Image classification is a fundamental computer vision task that requires labeling an image based on certain objects it contains. Many practical applications, including investment and trading strategies, require additional information.

The object detection tasks require not only the identification, but also the spatial location of all objects of interest, typically using bounding boxes. Several algorithms have been developed to overcome the inefficiency of brute-force, sliding-window approaches, including region proposal methods (R-CNN) and the **You Only Look Once** (**YOLO**) real-time object detection algorithm (see references on GitHub).

The object segmentation task goes a step further, and requires a class label and an outline of every object in the input image. This may be useful to count objects in an image, and evaluate a level of activity.

Semantic segmentation, also called **scene parsing**, makes dense predictions to assign a class label to each pixel in the image. As a result, the image is divided into semantic regions, and each pixel is assigned to its enclosing object or region.

In the next section, we will move to implement some of these ideas using the deep learning libraries introduced in the last chapter.

How to design and train a CNN using Python

All libraries we introduced in the last chapter provide support for convolutional layers. We are going to illustrate the LeNet5 architecture using the most basic MNIST handwritten digit dataset, and then use AlexNet on CIFAR10, a simplified version of the original ImageNet, to demonstrate the use of data augmentation.

## LeNet5 and MNIST using Keras

The original MNIST dataset contains 60,000 images in 28 x 28 pixel resolution, with a single grayscale containing handwritten digits from 0 to 9. A good alternative is the more challenging, but structurally similar, Fashion MNIST dataset, which we encountered in `Chapter 12`, *Unsupervised Learning*. See the `mnist_with_ffnn_and_lenet5` notebook for implementation details.

How to prepare the data

We can load it in Keras out of the box:

```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train.shape, X_test.shape
((60000, 28, 28), (10000, 28, 28))
```

The following screenshot shows the first ten images in the dataset, and highlights significant variation among instances of the same digit. On the right, it shows how the pixel values for an individual image range from 0 to 255:
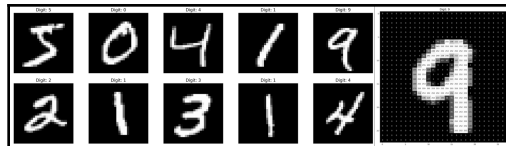
We rescale the pixel values to the range [0, 1] in order to normalize the training data, facilitate the backpropagation process, and convert the data to 32 bit floats that reduce memory requirements and computational cost, while providing sufficient precision for our use case:

```
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
```

We also need to convert the one-dimensional label to 10-dimensional, one-hot encoding to make it compatible with the cross-entropy loss, which receives a 10-class softmax output from the network:

```
y_train = np_utils.to_categorical(y_train, 10)
y_test = np_utils.to_categorical(y_test, 10)
```

How to define the architecture

We can define a simplified version of LeNet5 that omits the original final layer containing radial basis functions as follows, using the default `'valid'` padding and single step strides, unless defined otherwise:

```
lenet5 = Sequential([
        Conv2D(filters=6, kernel_size=5, activation='relu',
            input_shape=(28, 28, 1), name='CONV1'),
            AveragePooling2D(pool_size=(2, 2), strides=(1, 1),
            padding='valid', name='POOL1'),
```

```
            Conv2D(filters=16, kernel_size=(5, 5),
            activation='tanh', name='CONV2'),
            AveragePooling2D(pool_size=(2, 2), strides=(2, 2),
            name='POOL2'),
            Conv2D(filters=120, kernel_size=(5, 5),
            activation='tanh', name='CONV3'),
            Flatten(name='FLAT'),
            Dense(units=84, activation='tanh', name='FC6'),
            Dense(units=10, activation='softmax', name='FC7')
        ])
```

The summary indicates that the model, thus defined, has over 300,000 parameters:

```
_____
Layer (type)                 Output Shape              Param #
=================================================================
CONV1 (Conv2D)               (None, 24, 24, 6)         156
_____
POOL1 (AveragePooling2D)     (None, 23, 23, 6)         0
_____
CONV2 (Conv2D)               (None, 19, 19, 16)        2416
_____
POOL2 (AveragePooling2D)     (None, 9, 9, 16)          0
_____
CONV3 (Conv2D)               (None, 5, 5, 120)         48120
_____
FLAT (Flatten)               (None, 3000)              0
_____
FC6 (Dense)                  (None, 84)                252084
_____
FC7 (Dense)                  (None, 10)                850
=================================================================
Total params: 303,626
Trainable params: 303,626
Non-trainable params: 0
_____
```

We compile using cross-entropy loss, and the original stochastic gradient optimizer:

```
lenet5.compile(loss=categorical_crossentropy,
               optimizer='SGD',
               metrics=['accuracy'])
```

Now we are ready to train the model. The model expects 4D input, so we reshape accordingly. We use the standard batch size of 32, 80-20 train-validation split, use checkpointing to store the model weights if the validation error improves, and make sure the dataset is randomly shuffled:

```
training = lenet5.fit(X_train.reshape(-1, 28, 28, 1),
                      y_train,
                      batch_size=32,
                      epochs=50,
                      validation_split=0.2, #use 0 to train on all data
                      callbacks=[checkpointer],
                      verbose=1,
                      shuffle=True)
```

On a single GPU, 50 epochs take around 2.5 minutes, resulting in a test accuracy of 99.19%, almost exactly the same result as for the original LeNet5:

```
accuracy = lenet5.evaluate(X_test.reshape(-1, 28, 28, 1), y_test,
                           verbose=0)[1]
print('Test accuracy: {:.2%}'.format(accuracy))
Test accuracy: 99.19%
```

For comparison, a simple two-layer feedforward network achieves only 37.36% test accuracy (see notebook). The LeNet5 improvement on MNIST is, in fact, modest. Non-neural methods have also achieved classification accuracies greater than, or equal to, 99%, including **k-Nearest Neighbours** (**k-NNs**) or **Support Vector Machines** (**SVMs**). CNNs really shine with more challenging datasets, as we will see next.

AlexNet and CIFAR10 with Keras

Fast-forward to 2012, and we move on to the deeper, and more modern, AlexNet architecture. We will use the CIFAR10 dataset that uses 60,000 ImageNet samples, compressed to 32 x 32 pixel resolution (from the original 224 x 224), but still with three color channels. There are only 10 of the original 1,000 classes. See the `cifar10_image_classification` notebook for implementation details; we will skip over some repetitive steps.

How to prepare the data using image augmentation

CIFAR10 can also be downloaded from Keras, and we similarly rescale the pixel values and one-hot encode the ten class labels.

We first train a two-layer feedforward network on 50,000 training samples for training 20 epochs to achieve a test accuracy of 44.22%. We also experiment with a three-layer convolutional net, with 500 K parameters for 67.07% test accuracy (see notebook).

A common trick to enhance performance is to artificially increase the size of the training set by creating synthetic data. This involves randomly shifting or horizontally flipping the image, or introducing noise into the image.

Keras includes an `ImageDataGenerator` for this purpose, which we can configure and fit to the training data as follows:

```
datagen = ImageDataGenerator(
    width_shift_range=0.1,      # randomly horizontal shift
    height_shift_range=0.1,     # randomly vertial shift
    horizontal_flip=True)       # randomly horizontalflip
datagen.fit(X_train)
```

The result shows how the augmented images have been altered in various ways, as expected:
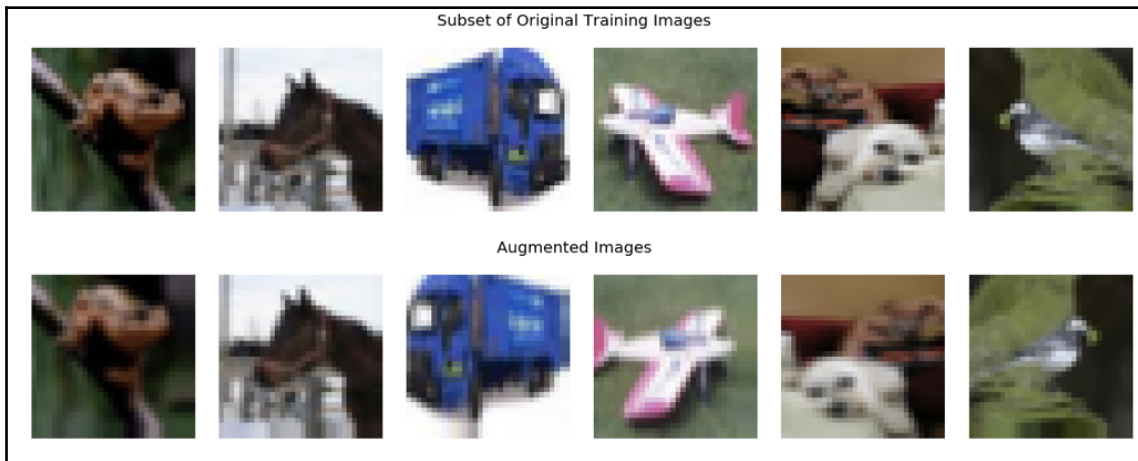
The test accuracy for the three-layer CNN improves markedly to 74.79% after training on the larger, augmented data.

How to define the model architecture

We also need to simplify the AlexNet architecture in response to the lower dimensionality of CIFAR10 images, relative to the ImageNet samples used in the competition. We use the original number of filters, but make them smaller (see notebook for implementation). The summary shows the five convolutional layers, followed by two fully-connected layers with frequent use of batch normalization, for a total of 21.5 million parameters:

```
Layer (type)                   Output Shape              Param #
=================================================================
CONV_1 (Conv2D)                (None, 16, 16, 96)        2688

POOL_1 (MaxPooling2D)          (None, 8, 8, 96)          0

NORM_1 (BatchNormalization)    (None, 8, 8, 96)          384

CONV2 (Conv2D)                 (None, 8, 8, 256)         614656

POOL2 (MaxPooling2D)           (None, 3, 3, 256)         0

NORM_2 (BatchNormalization)    (None, 3, 3, 256)         1024

CONV3 (Conv2D)                 (None, 3, 3, 384)         885120

CONV4 (Conv2D)                 (None, 3, 3, 384)         1327488

CONV5 (Conv2D)                 (None, 3, 3, 256)         884992

POOL5 (MaxPooling2D)           (None, 1, 1, 256)         0

NORM_5 (BatchNormalization)    (None, 1, 1, 256)         1024

FLAT (Flatten)                 (None, 256)               0

FC1 (Dense)                    (None, 4096)              1052672

DROP1 (Dropout)                (None, 4096)              0

FC2 (Dense)                    (None, 4096)              16781312

DROP2 (Dropout)                (None, 4096)              0

dense_1 (Dense)                (None, 10)                40970
=================================================================
Total params: 21,592,330
Trainable params: 21,591,114
```

```
Non-trainable params: 1,216
```

After training for 20 episodes, each of which takes a little under 30 seconds on a single GPU, we obtain 76.84% test accuracy.

How to use CNN with time-series data

The regular measurements of time-series result in a similar grid-like data structure, as with the image data we have focused on so far. As a result, we can use CNN architectures for univariate and multivariate time-series. In the latter case, we consider different time series as channels, similar to the different color signals.

We will illustrate the time series use case with the univariate asset price forecast example we introduced in the last chapter. Recall that we create rolling monthly stock returns, and use the 24 lagged returns alongside one-hot-encoded month information to predict whether the subsequent monthly return is positive or negative.

We will design a simple one-layer CNN, which uses one-dimensional convolutions, combined with max pooling to learn time series patterns:

```
model = Sequential([
Conv1D(filters=32, kernel_size=3, activation='relu', input_shape=
        (X_train.shape[1], 1)),
        MaxPooling1D(pool_size=2),
        Flatten(),
        Dense(1, activation='relu'),
        Activation('sigmoid')])
```

The model has 673 trainable parameters:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_1 (Conv1D)            (None, 34, 32)            128

max_pooling1d_1 (MaxPooling1 (None, 17, 32)            0

flatten_1 (Flatten)          (None, 544)               0

dense_1 (Dense)              (None, 1)                 545

activation_1 (Activation)    (None, 1)                 0
=================================================================
Total params: 673
```

```
    Trainable params: 673
    Non-trainable params: 0
```
_____

We will now compile, using our custom `auc_roc` metric developed in the last chapter:

```
model.compile(loss='binary_crossentropy',
              optimizer='Adam',
              metrics=['binary_accuracy', auc_roc])
```

We train on returns for the years 2010-16 for 20 epochs, using the default batch size of 32. Each epoch takes around 13 seconds on a single NVIDIA GTX 1080 GPU.

For 2017 returns, we find a test accuracy of 58.28% and test AUC of 0.5701. The network is still underfitting at this point, because both training and validation AUC are still improving after 20 epochs, suggesting that longer training, and potentially a higher-capacity network, would improve results. You should try!

Transfer learning – faster training with less data

In practice, we often do not have enough data to train a CNN from scratch with random initialization. Transfer learning is a ML technique that re-purposes a model trained on one set of data for another task. Naturally, it works if the learning from the first task carries over to the task of interest. If successful, it can lead to better performance and faster training, which requires less labeled data than training a neural network from scratch on the target task.

How to build on a pre-trained CNN

The transfer learning approach to CNN relies on pre-training on a very large dataset such as **ImageNet**. The goal is that the convolutional filters extract a feature representation that generalizes to new images. In a second step, it leverages the result to either initialize and retrain a new CNN or as inputs to in a new network that tackles the task of interest.

As discussed, CNN architectures typically use a sequence of convolutional layers to detect hierarchical patterns, adding one or more fully-connected layers to map the convolutional activations to the outcome classes or values. The output of the last convolutional layer that feeds into the fully-connected part is called bottleneck features. We can use the bottleneck features of a pre-trained network as inputs into a new fully-connected network, usually after applying a ReLU activation function.

In other words, we freeze the convolutional layers, and replace the dense part of the network. An additional benefit is that we can then use inputs of different sizes, because it is the dense layers that constrain the input size.

Alternatively, we can use the bottleneck features as inputs into a different machine learning algorithm. In the AlexNet architecture, for example, the bottleneck layer computes a vector with 4096 entries for each 224 x 224 input image. We then use this vector as features for a new model.

Alternatively, we can go a step further, and not only replace and retrain the classifier on top of the CNN using new data, but also fine-tune the weights of the pre-trained CNN. To achieve this, we continue training later layers while freezing the weights of some earlier layers. The motivation is presumably to preserve more generic patterns learned by lower layers, such as edge or color blob detectors, while allowing later layers of the CNN to adapt to the details of a new task. ImageNet, for example, contains a wide variety of dog breeds, which may lead to feature representations specifically useful for differentiating between these classes.

How to extract bottleneck features

Modern CNNs can take weeks to train on multiple GPUs on ImageNet. Fortunately, many researchers share their final weights. Keras, for example, contains pre-trained models for several of the reference architectures discussed previously, namely VGG16 and 19, ResNet50, InceptionV3 and InceptionResNetV2, MobileNet, DenseNet, NASNet, and MobileNetV2.

The `bottleneck_features` notebook illustrates how to download pre-trained VGG16 model, either with the final layers to generate predictions, or without the final layers, as illustrated in the following diagram, to extract the outputs produced by the bottleneck features:
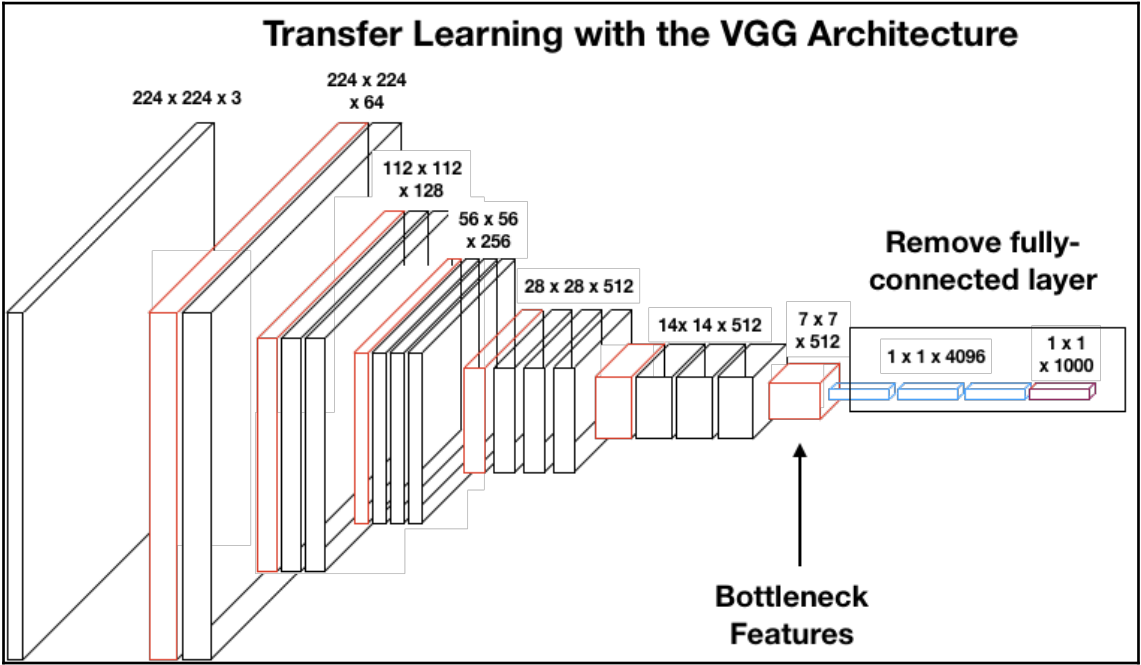
Keras makes it very straightforward to download and use pre-trained models:

```
vgg19 = VGG19()
vgg19.summary()
```

which yields the following output:

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_____
... several layers omitted ...
_____
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_____
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_____
```

```
block5_conv3 (Conv2D)        (None, 14, 14, 512)        2359808
─────────────────────────────────────────────────────────────────
block5_conv4 (Conv2D)        (None, 14, 14, 512)        2359808
─────────────────────────────────────────────────────────────────
block5_pool (MaxPooling2D)   (None, 7, 7, 512)          0
─────────────────────────────────────────────────────────────────
flatten (Flatten)            (None, 25088)              0
─────────────────────────────────────────────────────────────────
fc1 (Dense)                  (None, 4096)               102764544
─────────────────────────────────────────────────────────────────
fc2 (Dense)                  (None, 4096)               16781312
─────────────────────────────────────────────────────────────────
predictions (Dense)          (None, 1000)               4097000
=================================================================
Total params: 143,667,240
Trainable params: 143,667,240
Non-trainable params: 0
```

You can use this model like any other Keras model for predictions. To exclude the fully-connected layers, just add the `include_top = False` keyword to obtain the output of the final convolutional layer when passing an image to the CNN:

```
vgg19 = VGG19(include_top=False)
vgg16.predict(img_input).shape
(8, 7, 7, 512)
```

By omitting the fully-connected layers, we are no longer forced to use a fixed input size for the model (224 x 224, the original ImageNet format). By only keeping the convolutional modules, our model can be adapted to arbitrary input sizes.

How to further train a pre-trained model

We will demonstrate how to freeze some, or all, of the layers of a pre-trained model, and continue training using a new fully-connected set of layers and data with a different format.

We use the VGG16 weights, pre-trained on ImageNet with the much smaller 32 x 32 CIFAR10 data. Note that we indicate the new input size upon import, and set all layers to not trainable:

```
vgg16 = VGG16(include_top=False, input_shape =X_train.shape[1:])
for layer in vgg16.layers:
layer.trainable = False
vgg16.summary()
```

```
Layer (type)                   Output Shape             Param #
=================================================================
input_1 (InputLayer)           (None, 32, 32, 3)         0
_____
.. multiple layers omitted
_____
block5_conv2 (Conv2D)          (None, 2, 2, 512)         2359808
_____
block5_conv3 (Conv2D)          (None, 2, 2, 512)         2359808
_____
block5_pool (MaxPooling2D)     (None, 1, 1, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_____
```

We use Keras' functional API to define the `vgg16` output as input into a new set of fully-connected layers like so:

```
x = vgg16.output
x = Flatten()(x)
x = Dense(512, activation="relu")(x)
x = Dropout(0.5)(x)
x = Dense(256, activation="relu")(x)
predictions = Dense(10, activation="softmax")(x)
```

We define a new model in terms of inputs and output, and proceed from there as follows:

```
transfer_model = Model(inputs = vgg16.input,
                       outputs = predictions)
transfer_model.compile(loss = 'categorical_crossentropy',
                       optimizer = 'Adam',
                       metrics=["accuracy"])
```

We use a more elaborate `ImageDataGenerator` that also defines `validation_split`:

```
datagen = ImageDataGenerator(
          rescale=1. / 255,
          horizontal_flip=True,
          fill_mode='nearest',
          zoom_range=0.1,
          width_shift_range=0.1,
          height_shift_range=0.1,
          rotation_range=30,
          validation_split=validation_split)
```

We define both train and validation generators for the `fit` method:

```
train_generator = datagen.flow(X_train, y_train, subset='training')
val_generator = datagen.flow(X_train, y_train, subset='validation')
```

And now we proceed to train the model:

```
n = X_train.shape[0]
    transfer_model.fit_generator(train_generator,
    steps_per_epoch=n // batch_size,
    epochs=epochs,
    validation_data=val_generator,
    validation_steps=(n * .2) // batch_size)
```

10 epochs lead to a mediocre test accuracy of 35.87%, because the assumption that image features translate to so much smaller images is somewhat questionable, but it serves to illustrate the workflow.

How to detect objects

Object detection requires the ability to distinguish between several classes of objects, and to decide how many, and which, of these objects are present in an image.

Google Street View house number dataset

A prominent example is Ian Goodfellow's identification of house numbers from Google's Street View dataset. It is required to identify the following:

- How many of up to five digits make up the house number
- The correct digit for each component
- The proper order of the constituent digits

The `svhn_preprocessing` notebook contains code to produce a simplified, cropped dataset that uses bounding box information to create regularly shaped 32 x 32 images containing the digits; the original images are of arbitrary shape.

The `svhn_object_detection` notebook illustrates how to build a deep CNN using Keras' functional API to generate multiple outputs: one to predict how many digits are present, and five for the value of each in the order they appear.

How to define a CNN with multiple outputs

The best-performing architecture on the original dataset has eight convolutional layers, and two final fully-connected layers. The convolutional layers are similar, so we can define a function to simplify their creation:

```
def svhn_layer(model, filters, strides, n, input_shape=None):
    if input_shape is not None:
        model.add(Conv2D(filters, kernel_size=5, padding='same',
name='CONV{}'.format(n), input_shape=input_shape))
    else:
        model.add(Conv2D(filters, kernel_size=5, padding='same',
activation='relu', name='CONV{}'.format(n)))
    model.add(BatchNormalization(name='NORM{}'.format(n)))
    model.add(MaxPooling2D(pool_size=2, strides=strides,
name='POOL{}'.format(n)))
    model.add(Dropout(0.2, name='DROP{}'.format(n)))
    return model
```

The entire model combines the `Sequential` and functional API as follows:

```
model = Sequential()

svhn_layer(model, 48, 1, n=1, input_shape=(32,32,1))
for i, kernel in enumerate([48, 64, 128, 160] + 3 * [192], 2):
    svhn_layer(model, kernel, strides=2 if i % 2 == 0 else 1, n=i)

model.add(Flatten())
model.add(Dense(3072, name='FC1'))
model.add(Dense(3072, name='FC2'))
y = model.output
```

The output of the `Sequential` model becomes the input to the output branches defined using the functional API:

```
n_digits = (Dense(units=6, activation='softmax'))(y)
digit1 = (Dense(units=10, activation='softmax'))(y)
digit2 = (Dense(units=11, activation='softmax'))(y)
digit3 = (Dense(units=11, activation='softmax'))(y)
digit4 = (Dense(units=11, activation='softmax'))(y)
digit5 = (Dense(units=11, activation='softmax'))(y)
svhn_model = Model(inputs=model.input, outputs=[n_digits, digit1,
                   digit2, digit3, digit4, digit5])
```

As a result, the model produces six distinct outputs that we can evaluate.

See the notebook for additional implementation details, and evaluation of the results.

Recent developments

Research on CNN architectures has progressed extremely dynamically. Several developments are of potential interests in the context of trading strategies that use alternative data, especially image data.

Fast detection of objects on satellite images

Kaggle recently conducted a competition to detect container ships on satellite images, emphasizing fast inference. Prominent architectures included UNET, which has been used in bioinformatics (see GitHub: `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading` for references).

How capsule networks capture pose

CNNs have a key weakness due to their invariance to spatial translations of features. They do not learn relative positions of the various features they detect to recognize or locate an object. As a result, the image of a face with key features such as eyes, ears, and nose swapped would still be recognized as a human face, or a particular person.

Geoff Hinton, one of the leading researchers in the field, recently proposed a new architecture called **capsule nets**, which may yield significant improvements relative to CNN. The paper is called *Dynamic Routing Between Capsules*, and the new model stands out through its ability to encode the pose of an object. As a result, it is better in detecting objects that are partially occluded, or distinguish objects that overlap. See references on GitHub for more background and implementations using Keras and TensorFlow.

Summary

In this chapter, we introduced convolutional networks, a specialized neural network architecture that has taken cues from our (limited) understanding of human vision, and performs particularly well on grid-like data. We covered the central operation of convolution or cross-correlation that drives the discovery of filters, which, in turn, detects features that are useful for solving the task at hand.

We reviewed several state-of-the-art architectures that are good starting points, especially because transfer learning enables us to reuse pre-trained weights, and reduce the otherwise rather computationally and data-intensive training effort. We also saw that Keras makes it relatively straightforward to implement and train a diverse set of deep CNN architectures.

In the next chapter, we will introduce unsupervised deep learning techniques, such as autoencoders and **Generative Adversarial Networks** (**GANs**), and their applications in investment and trading strategies.

Index