Part 2 – Look Ma, No Hands! Deep Networks in the Real World

Part 2 shifts the focus from theoretical foundations of deep learning to their **practical application in real-world scenarios**. A proficient machine learning practitioner must not only understand deep learning concepts and frameworks such as TensorFlow but also be capable of navigating complex APIs to build scalable solutions for real problems in domains like **computer vision** and **natural language processing**.

This part explores applied deep learning through two major domains. First, it addresses **computer vision**, focusing on image classification and image segmentation tasks. State-of-the-art deep learning models that perform well in these tasks are analyzed in depth, including the motivations behind their architectural design choices. These models are implemented from scratch to provide both practical and conceptual understanding. The discussion then transitions to **natural language processing**, covering sentiment analysis and language modeling. Key preprocessing techniques and the use of word vectors are examined, along with pretraining strategies that underpin modern NLP systems.

Chapter 6 – Teaching Machines to See: Image Classification with CNNs

This chapter focuses on applying **convolutional neural networks (CNNs)** to real-world image classification problems. CNNs are well suited for image data because they operate on two-dimensional inputs and use convolutional kernels to extract spatial features. As CNNs grow deeper, they learn increasingly abstract feature representations. Pooling layers are used between convolutional layers to reduce spatial dimensionality while preserving important information. Ultimately, the extracted features are passed to fully connected layers to produce class predictions.

Earlier chapters introduced CNNs using the **Keras Sequential API** and standard layers such as Conv2D, MaxPool2D, and Dense. In this chapter, the discussion advances toward **real-world data**, emphasizing that practical machine learning extends beyond clean, curated datasets. Real-world image data often contains noise, inconsistencies, and imbalance, which must be addressed before model training.

The chapter introduces **exploratory data analysis (EDA)** as a critical step in the machine learning lifecycle. Additionally, it examines a state-of-the-art computer vision architecture—the **Inception model**—highlighting its design principles and advantages. The chapter concludes with training and evaluating the model using performance metrics such as classification accuracy.

6.1 Putting the Data Under the Microscope: Exploratory Data Analysis

This section introduces **exploratory data analysis (EDA)** in the context of building an image classification system for an intelligent shopping assistant. The objective of the system is to analyze photos uploaded by users and identify household objects to infer stylistic preferences. To support this goal, a diverse and representative dataset is required, and EDA is used to understand the dataset's characteristics and limitations.

EDA is described as the **cornerstone of data science development**, aiming to produce a clean, high-quality dataset before modeling begins. During this process, the data is examined to detect common issues such as class imbalance, corrupted samples, missing features, outliers, and features that require transformation (e.g., normalization or encoding). Thorough exploration improves data quality and increases the likelihood of successful model training.

For this project, the **tiny-imagenet-200** dataset is selected. It contains real-world images spanning **200 different object classes**, making it suitable for training a versatile image classification model capable of recognizing a wide range of household items.

*Figure 6.1 Some sample images from the tiny-imagenet-200. You can see that these images belong to a wide variety of categories***.**

Dataset Acquisition

The first practical step in EDA is obtaining the dataset. The provided Python script automates this process by creating a local data directory, downloading the compressed dataset file, and extracting its contents. Once completed, the dataset is available in a structured directory (tiny-imagenet-200) for further analysis and preprocessing.

Code insertion:
The **dataset download and extraction code** should be placed **directly after the explanation of the data acquisition step**, as it operationalizes the described process.

Section Purpose

By the end of this section, the dataset is locally available and ready for deeper inspection. This prepares the groundwork for subsequent steps, including dataset inspection, preprocessing pipelines, and model implementation, which are necessary to train high-performing CNN-based image classifiers.

6.1.1 The Folder/File Structure

With the dataset extracted, the next step is to manually explore its folder and file structure. The tiny-imagenet-200 directory contains **three main folders and two text files**, as illustrated in the dataset overview.

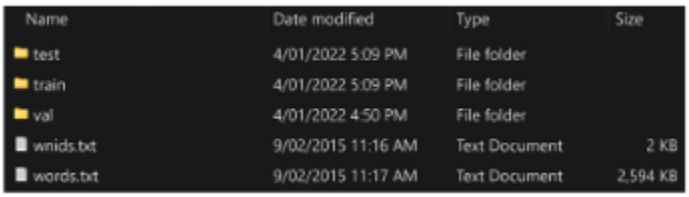


Figure 6.2 The folders and files found in the tiny-imagenet-200 data setThe file wnids.txt contains 200 WordNet IDs (wnids), where each ID corresponds to a single image class. These IDs are derived from the WordNet lexical database and uniquely identify object categories, such as "goldfish" or "basketball."

```
n02124075
n04067472
n04540053
n04099969
n07749582
n01641577
n02802426
n09246464
n07920052
n03970156
n03891332
```

Figure 6.3 Sample content from wnids.txt. It contains wnids (WordNetIDs), one per line.

To make these identifiers human-readable, the dataset includes the file words.txt. This file maps wnids to textual class descriptions in a tab-separated format. Although words.txt contains over 82,000 entries originating from a much larger dataset, only a subset of these entries corresponds to the 200 classes used in Tiny ImageNet.

Table 6.1 Sample content from words.txt. It contains the wnids and their

descriptions for the data found in the data set.

n00001740	entity
n00001930	physical entity
n00002137	abstraction, abstract entity
n00002452	thing
n00002684	object, physical object
n00003553	whole, unit
n00003993	congener
n00004258	living thing, animate thing
n00004475	organism, being
n00005787	benthos
n00005930	dwarf
n00006024	heterotroph
n00006150	parent
n00006269	life
n00006400	biont

The train folder contains the labeled training data. Inside it, there are **200 subfolders**, each named after a wnid. Each of these subfolders includes an images directory containing **500 images per class**, resulting in a total of **100,000 training images**.
The val folder contains validation images stored in a single directory, with their corresponding labels specified in the val_annotations.txt file.
The test folder, which lacks labels, is reserved for competition scoring and is not used in this chapter.

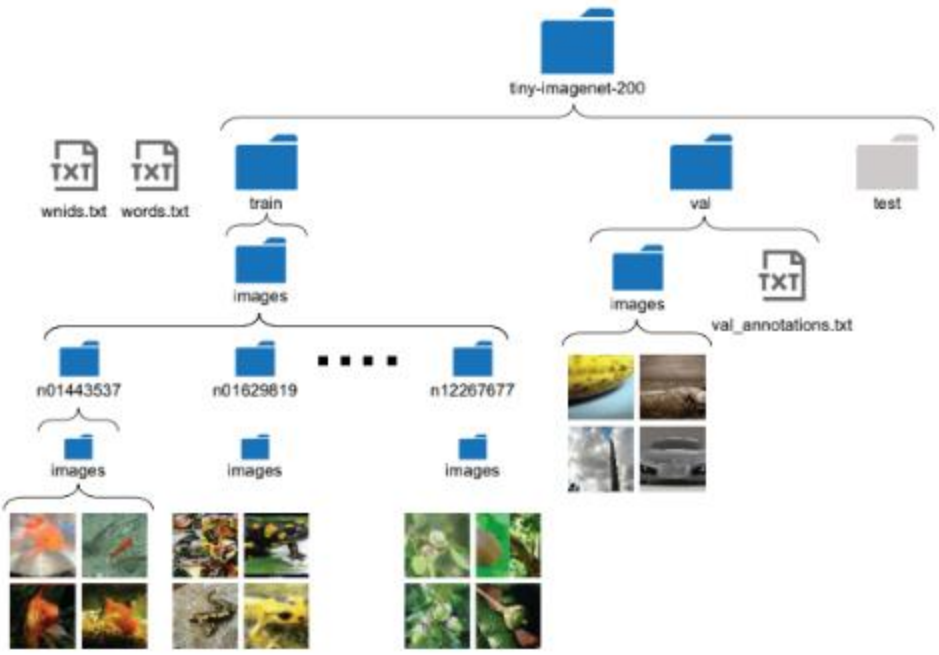


Figure 6.4 The overall structure of the tiny-imagenet-200 data set. It has three text files (wnids.txt, words.txt, and val/val_annotations.txt) and three folders (train, val, and test). We will only use the train and val folders.

6.1.2 Understanding the Classes in the Dataset

After understanding the dataset structure, the next step is to identify and describe the image classes programmatically. This is achieved by defining a function called get_tiny_imagenet_classes(), which reads the wnids.txt and words.txt files and produces a pandas DataFrame containing two columns: the wnid and its corresponding class description.

The function first loads the wnids into a pandas Series and then loads the word-to-description mappings into a DataFrame. Since words.txt contains far more entries than required, the function filters this DataFrame to retain only the wnids present in the dataset, resulting in exactly **200 labeled classes**.

Listing 6.1 Getting class descriptions of the classes in the data set

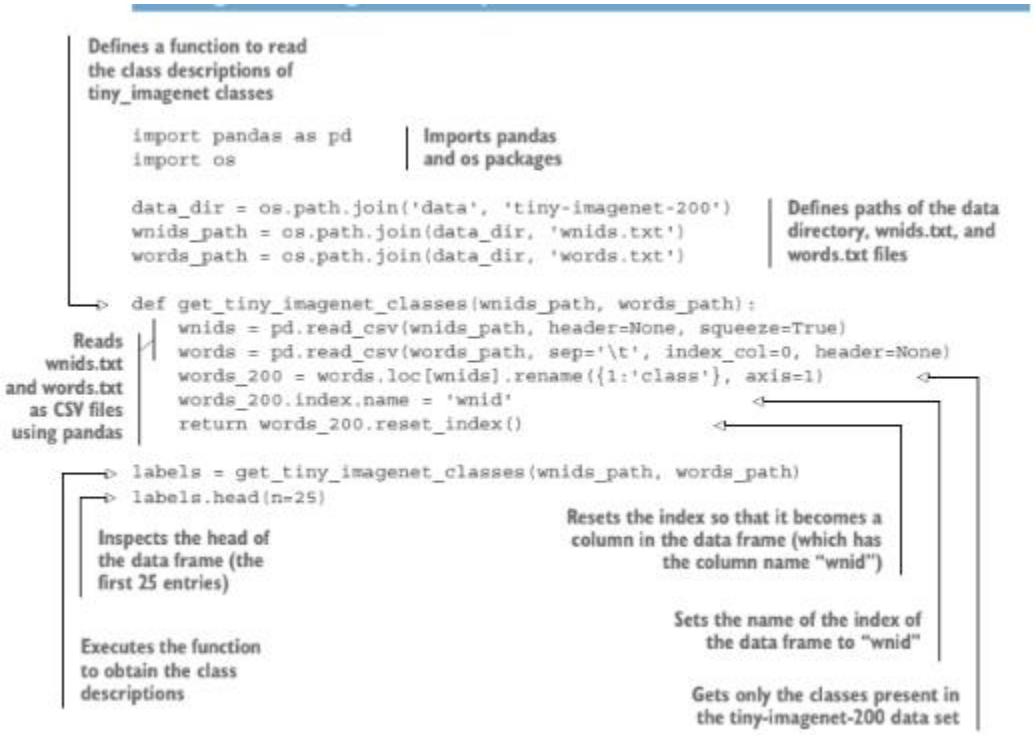


Table 6.2 Sample of the labels' IDs and their descriptions that we generate using the get_tiny_imagenet_classes() function

	wind	class
0	n02124075	Egyptian cat
1	n04067472	reel
2	n04540053	volleyball
3	n04099969	rocking chair, rocker
4	n07749582	lemon
5	n01641577	bullfrog, Rana catesbeiana
6	n02802426	basketball
7	n09246464	cliff, drop, drop-off
8	n07920052	espresso
9	n03970156	plunger, plumber's helper
10	n03891332	parking meter
11	n02106662	German shepherd, German shepherd dog, German p...
12	n03201208	dining table, board
13	n02279972	monarch, monarch butterfly, milkweed butterfly
14	n02132136	brown bear, bruin, Ursus arctos
15	n041146614	school bus

To further analyze the dataset, the number of training images per class is computed. A helper function, get_image_count(), counts the number of JPEG files in each class directory. Using pandas' apply() function, this count is computed for every wnid and stored in a new column called n_train.

Table 6.3 Sample of the data where n_train (number of training samples) has been calculated

	wnid	class	n_train
0	n02124075	Egyptian cat	500
1	n04067472	reel	500
2	n04540053	volleyball	500
3	n04099969	rocking chair, rocker	500
4	n07749582	lemon	500
5	n01641577	bullfrog, Rana catesbeiana	500
6	n02802426	basketball	500
7	n09246464	cliff, drop, drop-off	500
8	n07920052	espresso	500
9	n03970156	plunger, plumber's helper	500

Figure 6.5 Sample images for the wnid category n02802426 (i.e., basketball)

These images often contain occlusions, unusual colors, or irrelevant context, highlighting why image classification is a difficult task and why deep CNNs are required to capture semantic meaning beyond simple pixel patterns.

Statistical analysis of the n_train column using describe() confirms that every class contains exactly **500 images**, indicating a **perfectly balanced dataset**. This balance is advantageous for training classification models, as it reduces bias toward specific classes.

6.1.3 Computing Simple Statistics on the Dataset

Beyond class distribution, analyzing **image-level statistics** is essential for making informed preprocessing decisions. In this section, image width and height statistics are computed to understand the typical image dimensions in the dataset.

To make the computation efficient, only the first 25 classes are sampled. For each image, the width and height are extracted using the PIL library, and these values are stored as records in a list. This list of records is then converted into a pandas DataFrame, where each row corresponds to one image and columns represent width and height.

Descriptive statistics such as mean, standard deviation, and quartiles are computed using the describe() function. These statistics help determine appropriate resizing, cropping, or padding strategies, which are necessary because CNNs require fixed-size inputs.

Listing 6.2 Computing image width and height statistics

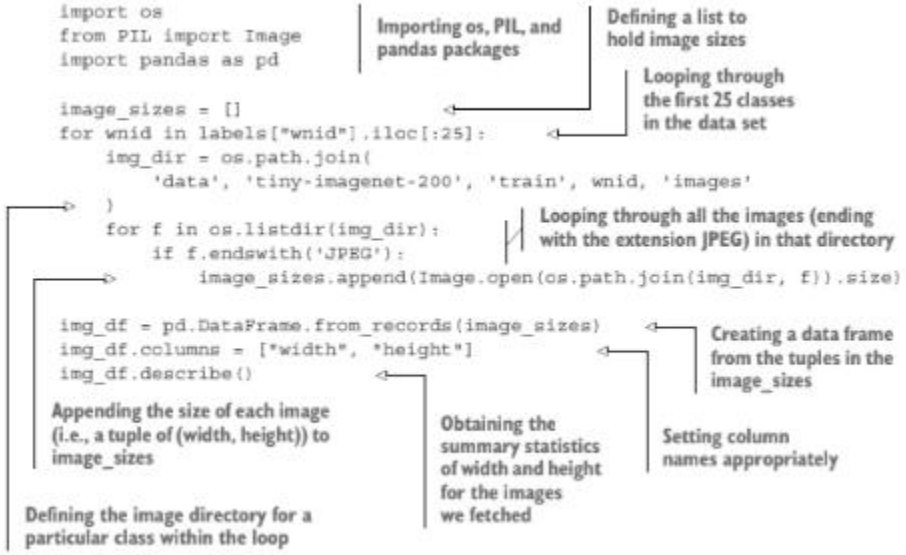


Table 6.4 Width and height statistics of the images

	width	height
count	12500.0	12500.0
mean	64.0	64.0
std	0.0	0.0
min	64.0	64.0
25%	64.0	64.0
50%	64.0	64.0
75%	64.0	64.0
max	64.0	64.0

This analysis concludes the exploratory phase and prepares the groundwork for building an efficient data pipeline to feed images into CNN models.

6.2 Creating Data Pipelines Using the Keras ImageDataGenerator

After thoroughly exploring the dataset and understanding key properties such as the number of classes, object categories, and image dimensions, the next step is to construct data pipelines that can efficiently supply data to a model. In this section, three separate data generators are created for training, validation, and testing. These generators are responsible for loading images from disk in batches and performing any necessary preprocessing so that the data can be directly consumed by a deep learning model. To achieve this, the tensorflow.keras.preprocessing.image.ImageDataGenerator class is used.

The process begins by defining a Keras ImageDataGenerator object, along with a fixed random seed and batch size. The generator is initialized with samplewise_center=True, which ensures that each image is normalized individually by subtracting its mean pixel value, and validation_split=0.1, which reserves 10% of the training data for validation.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

import os

random_seed = 4321

batch_size = 128

image_gen = ImageDataGenerator(samplewise_center=True, validation_split=0.1)
```

The dataset images have uniform dimensions, which is reflected in the descriptive statistics shown below.

	width	height
count	12500.0	12500.0
mean	64.0	64.0
std	0.0	0.0
min	64.0	64.0
25%	64.0	64.0
50%	64.0	64.0
75%	64.0	64.0
max	64.0	64.0

Table 6.4
Width and height statistics of the images.

This table confirms that all images are 64 × 64 pixels, with no variation across the dataset. The validation_split parameter plays a critical role by allowing the training data to be split into training and validation subsets while keeping the validation set fixed throughout training. In a typical machine learning workflow, the training dataset is used to learn model parameters, the validation dataset is used to monitor model performance during training, and the test dataset is used only after training to evaluate how well the model generalizes to unseen data.

Once the ImageDataGenerator has been defined, several flow methods become available for reading data from different sources. These include reading from NumPy arrays or DataFrames, reading from DataFrames containing filenames and labels, and reading directly from directory structures. In this case, flow_from_directory() is used because the training data is organized in the expected directory format.

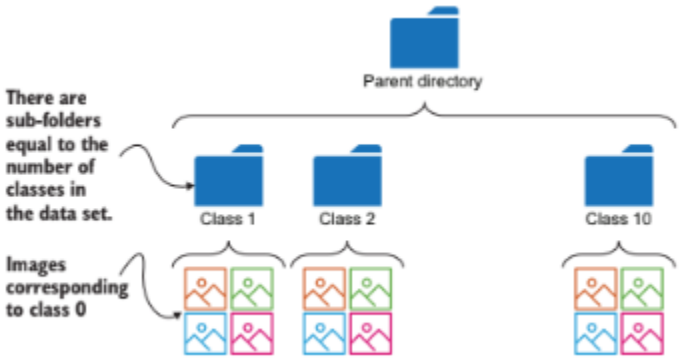


Figure 6.6
Folder structure expected by the flow_from_directory() method.

The flow methods return Python generators. A generator is a function that yields values one at a time using the yield keyword, rather than returning all values at once. This approach is significantly more memory efficient than storing all data in lists. In the context of deep learning, each iteration of the generator yields a single batch of images and labels, which can be fed directly into tf.keras.models.Model.fit().

```
target_size = (56,56)

train_gen = image_gen.flow_from_directory(

directory=os.path.join('data','tiny-imagenet-200', 'train'),

target_size=target_size, classes=None,

class_mode='categorical', batch_size=batch_size,

shuffle=True, seed=random_seed, subset='training'

)
```

```
valid_gen = image_gen.flow_from_directory (
directory=os.path.join('data','tiny-imagenet-200', 'train'),
target_size=target_size, classes=None,
class_mode='categorical', batch_size=batch_size,
shuffle=True, seed=random_seed, subset='validation'
```

The training and validation generators are created using flow_from_directory() with nearly identical arguments. The key difference is the subset parameter, which is set to "training" for the training generator and "validation" for the validation generator. Although the original images are 64 × 64 pixels, they are resized to 56 × 56 to make them easier to adapt to a model architecture designed for 224 × 224 inputs.

```
from functools import partial
target_size = (56,56)
partial_flow_func = partial(
image_gen.flow_from_directory,
directory=os.path.join('data','tiny-imagenet-200', 'train'),
target_size=target_size, classes=None,
class_mode='categorical', batch_size=batch_size,
shuffle=True, seed=random_seed)
train_gen = partial_flow_func(subset='training')
valid_gen = partial_flow_func(subset='validation')
```

Because most arguments passed to the training and validation generators are identical, the partial function from Python is used to reduce redundancy. This approach defines a partially configured version of flow_from_directory() and then supplies only the differing subset argument when creating each generator, resulting in cleaner and less error-prone code.

Even though the framework provides built-in mechanisms for splitting data, it is emphasized that the consistency of the validation data should not be taken for granted. The validation set must remain consistent across epochs and runs, and it is good practice to explicitly verify this behavior.

The data generators require further modification because the model used later has three output layers: one final prediction layer and two auxiliary output layers. As a result, the generator output

must be transformed from (x, y) into (x, (y, y, y)) by replicating the labels. This is accomplished by defining a wrapper generator that modifies the output format.

```
def data_gen_aux(gen):

for x,y in gen:

yield x,(y,y,y)

train_gen_aux = data_gen_aux(train_gen)

valid_gen_aux = data_gen_aux(valid_gen)
```

The test data requires special handling because its directory structure differs from that of the training data. All test images are stored in a single folder, and their labels are provided in a separate file named val_annotations.txt. This file is read into a pandas DataFrame containing image filenames and corresponding class labels, which is then used with flow_from_dataframe() to create the test data generator. The test generator is configured with shuffle=False to ensure consistent evaluation results.

```
def get_test_labels_df(test_labels_path):

test_df = pd.read_csv(test_labels_path, sep='\t', index_col=None,

header=None)

test_df = test_df.iloc[:,[0,1]].rename({0:"filename", 1:"class"}, axis=1)

return test_df

test_df = get_test_labels_df(os.path.join('data','tiny-imagenet-200', 'val',

'val_annotations.txt'))
```

An exercise concludes this section, asking how to modify the data generator to corrupt labels with a 50% probability in order to test the robustness of the model against noisy training data.

6.3 Inception Net: Implementing a State-of-the-Art Image Classifier

After preparing the data pipelines, attention turns to building a powerful image classification model. For image-based tasks, convolutional neural networks are the dominant approach, and in this section, a state-of-the-art CNN known as Inception Net is introduced and implemented using the Keras Functional API.

Inception Net is a highly complex model that achieved state-of-the-art performance in computer vision tasks. Multiple versions of the model were released within a short period, highlighting its impact and popularity in the research community. In this work, the first version of the model,

Inception Net v1, is implemented. Due to its complexity, understanding its architectural design and underlying motivations is essential.

Unlike a typical CNN, Inception Net is characterized by its depth and architectural sophistication. Although increasing model complexity often improves accuracy, it also introduces challenges such as overfitting when data is limited and increased computational and memory requirements. The Inception architecture addresses these challenges by introducing sparsity, reducing the number of parameters without sacrificing representational power.

6.3.1 Recap on CNNs

Convolutional neural networks are primarily used for computer vision problems such as image classification and object detection. A typical CNN consists of convolution layers, pooling layers, and fully connected layers, as illustrated below.

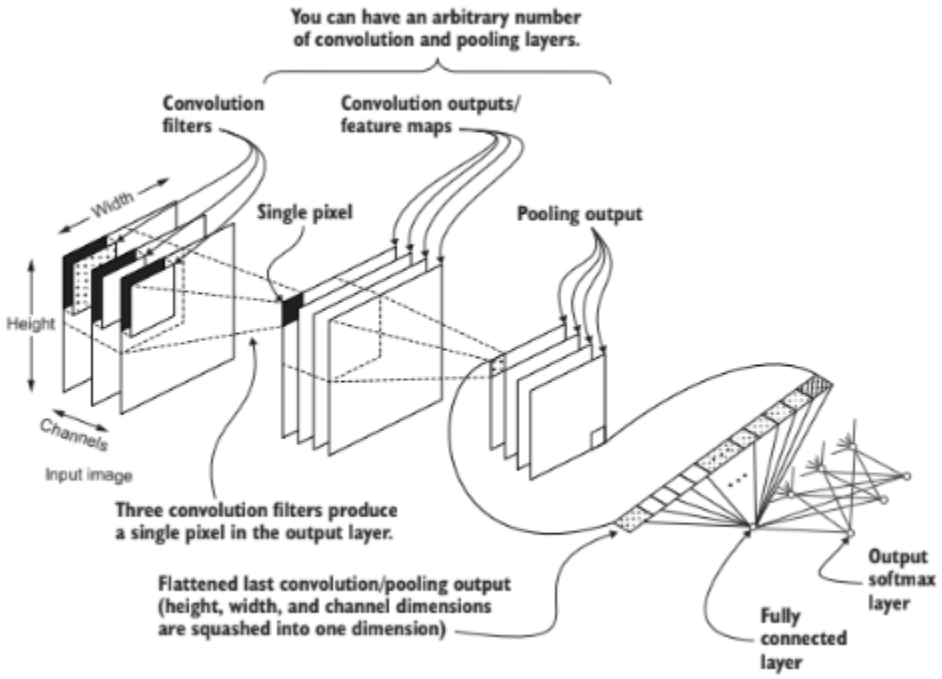


Figure 6.7 A simple convolutional neural network. First, we have an image with height, width, and channel dimensions, followed by a convolution and a pooling layer. Finally, the last convolution/pooling layer output is flattened and fed to a set of fully connected layers.\

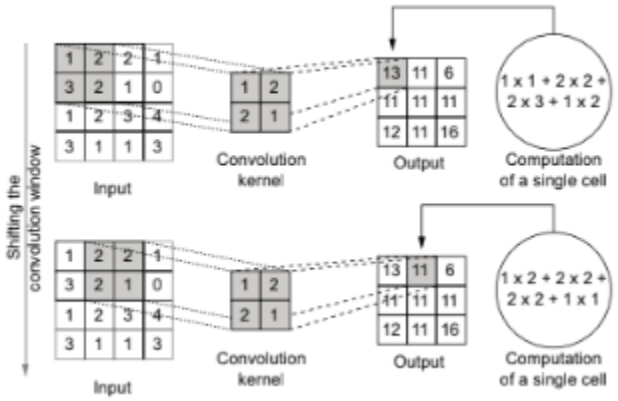


Figure 6.8 The computations that happen in the convolution operation while shifting the window

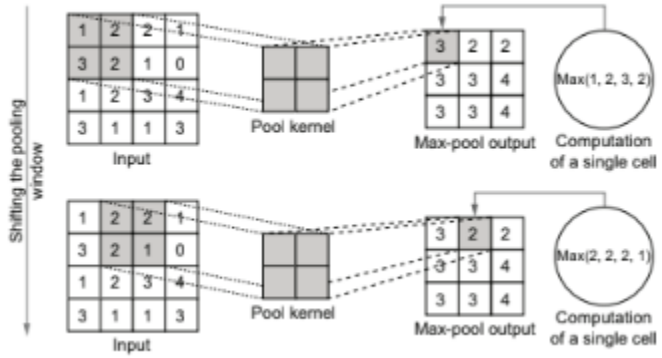


Figure 6.9 How the pooling operation computes the output. It looks at a small window and takes the maximum of the input in that window as the output for the corresponding cell.

6.3.2 Inception Net v1

Inception Net v1, also known as GoogLeNet, extends standard CNNs by introducing the Inception block, which consists of parallel convolution and pooling operations with different kernel sizes. At a high level, the model begins with a stem composed of standard convolution and pooling layers, followed by multiple Inception blocks interleaved with pooling layers, and ends with fully connected layers and auxiliary outputs.

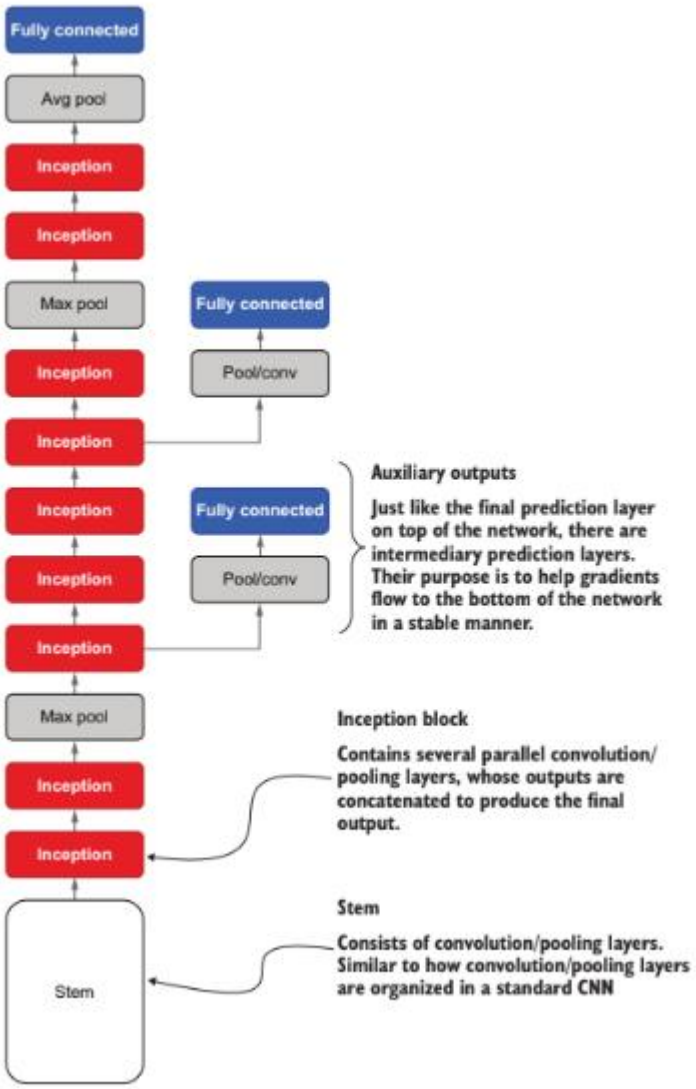
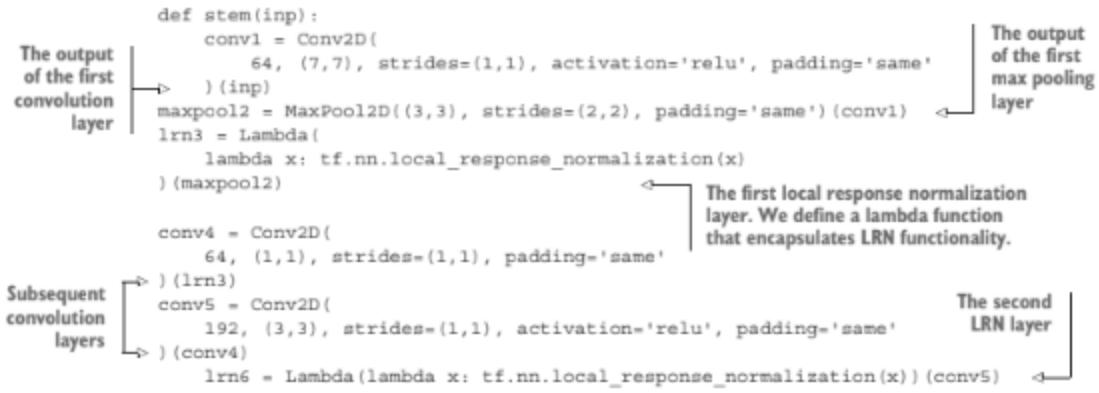
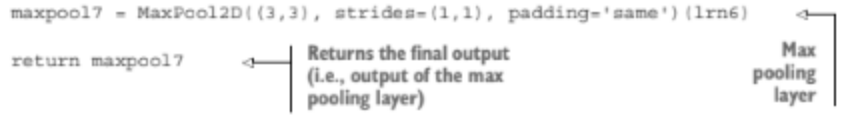


Figure 6.10 Abstract architecture of Inception net v1. Inception net starts with a stem, which is an ordinary sequence of convolution/pooling layers that is found in a typical CNN. Then Inception net introduces a new component known as an Inception block. Finally, Inception net also makes use of auxiliary output layers.

Listing 6.3 Defining the stem of Inception net


def stem(inp):
conv1 = Conv2D(
64, (7,7), strides=(1,1), activation='relu', padding='same'
)(inp)
maxpool2 = MaxPool2D((3,3), strides=(2,2), padding='same')(conv1)
lrn3 = Lambda(
lambda x: tf.nn.local_response_normalization(x)
)(maxpool2)
conv4 = Conv2D(
64, (1,1), strides=(1,1), padding='same'
)(lrn3)
conv5 = Conv2D(
192, (3,3), strides=(1,1), activation='relu', padding='same'
)(conv4)
lrn6 = Lambda(lambda x: tf.nn.local_response_normalization(x))(conv5)
The output of the first convolution layer
The output of the first max pooling layer
The first local response normalization layer. We define a lambda function that encapsulates LRN functionality.
Subsequent convolution layers
The second LRN layer


maxpool7 = MaxPool2D((3,3), strides=(1,1), padding='same')(lrn6)
return maxpool7
Returns the final output (i.e., output of the max pooling layer)
Max pooling layer

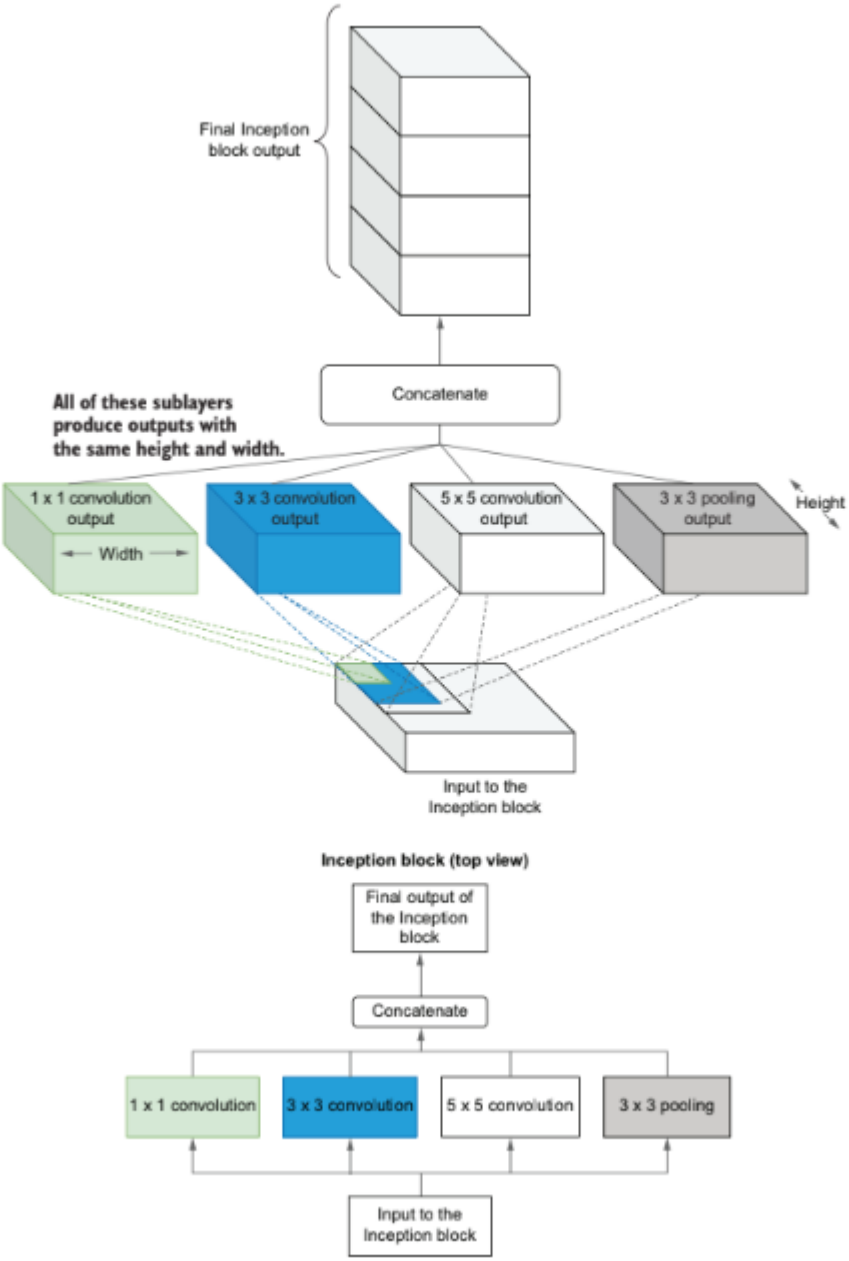


Figure 6.11 The computations in the Inception block, which is essentially a set of parallel convolution/pooling layers with different kernel sizes

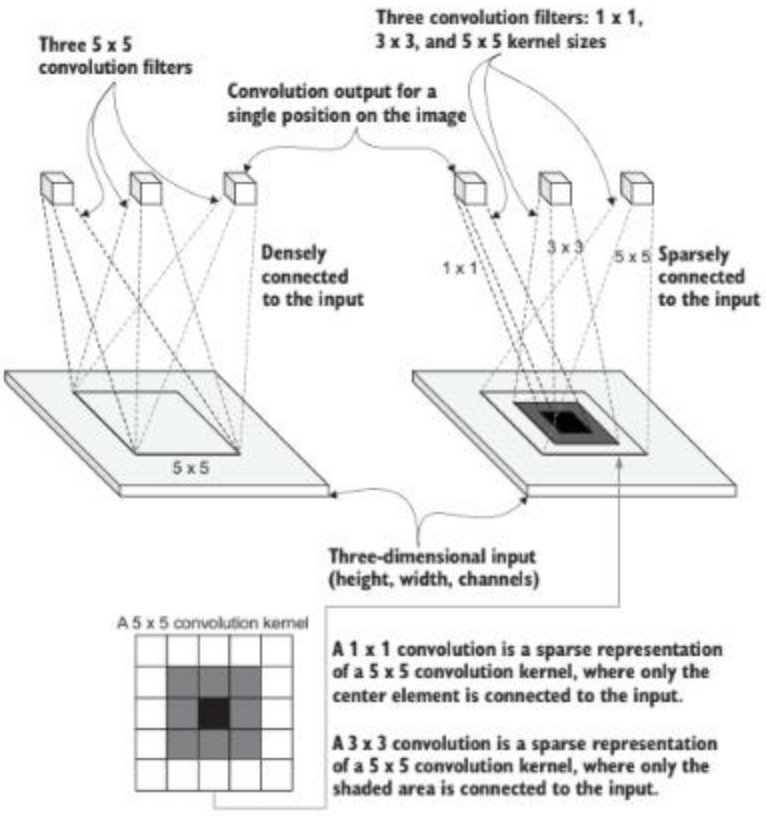


Figure 6.12 How the Inception block encourages sparsity in the model. You can view a 1 × 1 convolution as a highly sparse 5 × 5 convolution.

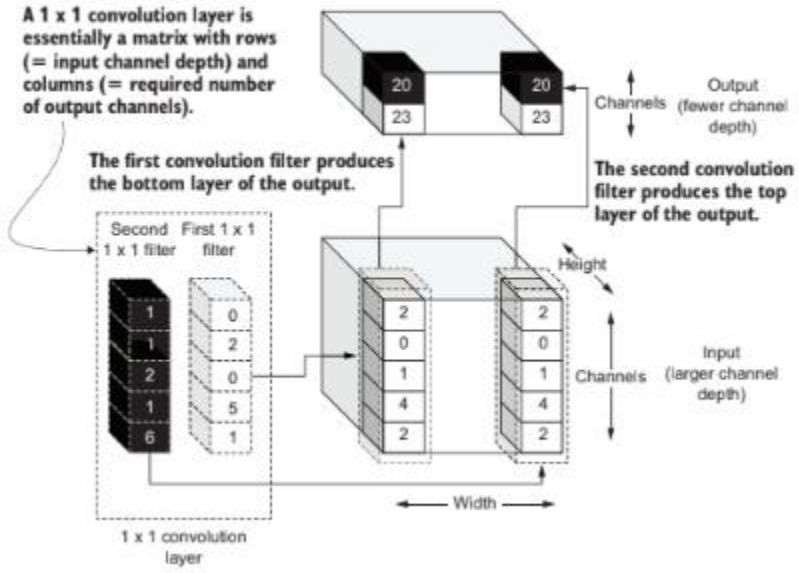


Figure 6.13 The computations of a 1 × 1 convolution and how it enables reduction of a channel dimension of an input

Listing 6.4 Defining the Inception block of the Inception net

```
def inception(inp, n_filters):

    # 1x1 layer
    out1 = Conv2D(
        n_filters[0][0], (1,1), strides=(1,1), activation='relu',
➥ padding='same'
    )(inp)

    # 1x1 followed by 3x3
    out2_1 = Conv2D(
        n_filters[1][0], (1,1), strides=(1,1), activation='relu',
➥ padding='same')
(inp)
    out2_2 = Conv2D(
        n_filters[1][1], (3,3), strides=(1,1), activation='relu',
➥ padding='same'
)(out2_1)

# 1x1 followed by 5x5
out3_1 = Conv2D(
    n_filters[2][0], (1,1), strides=(1,1), activation='relu',
➥ padding='same'
)(inp)
out3_2 = Conv2D(
    n_filters[2][1], (5,5), strides=(1,1), activation='relu',
➥ padding='same'
)(out3_1)

# 3x3 (pool) followed by 1x1
out4_1 = MaxPool2D(
    (3,3), strides=(1,1), padding='same'
)(inp)
out4_2 = Conv2D(
    n_filters[3][0], (1,1), strides=(1,1), activation='relu',
➥ padding='same'
)(out4_1)

out = Concatenate(axis=-1)([out1, out2_2, out3_2, out4_2])
return out
```

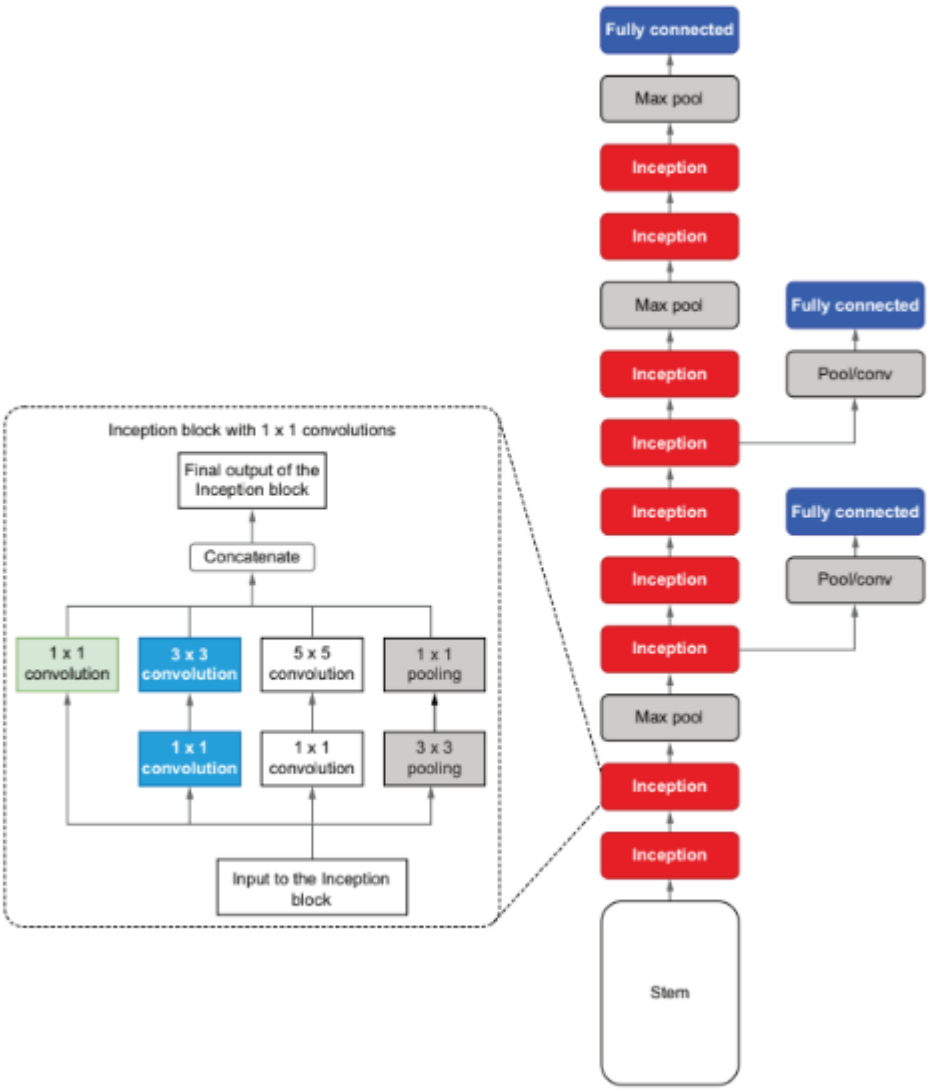


Figure 6.14 The Inception block alongside the full architecture of the Inception net model

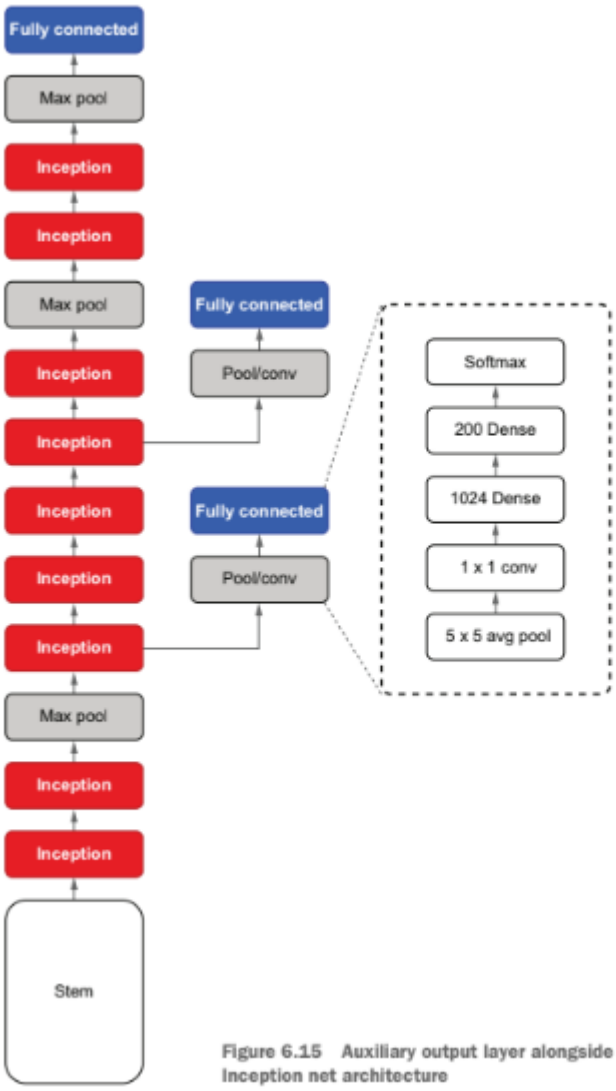
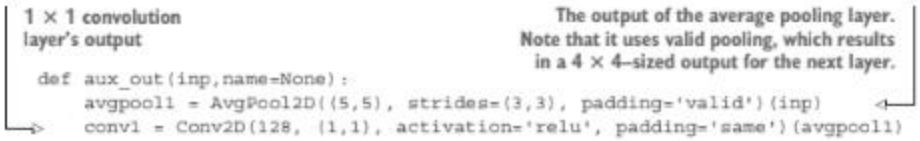


Figure 6.15 Auxiliary output layer alongside the full Inception net architecture

Figure 6.15 Auxiliary output layer alongside the full Inception net architecture

Listing 6.5 Defining the auxiliary output as a Python function



6.3.3 Putting Everything Together

At this stage, all the conceptual and implementation components required to build the Inception Net model have been completed. The architecture has been decomposed into three major elements: the stem, the Inception blocks, and the auxiliary output layers. The stem closely resembles the convolutional portion of a standard CNN, excluding the fully connected layers. The Inception blocks introduce parallel convolutional paths with different kernel sizes, enabling the network to capture features at multiple receptive fields while encouraging sparsity and reducing overfitting. Auxiliary outputs are incorporated to stabilize training by mitigating numerical instability and improving gradient flow in deep networks.

To facilitate reuse and clarity, each of these components was encapsulated into separate functions. With these building blocks in place, the complete Inception Net v1 architecture can now be defined.

Listing 6.6 Defining the full Inception net model

```
def inception_v1():

    K.clear_session()

    inp = Input(shape=(56,56,3))
    stem_out = stem(inp)
```

Defines an input layer. It takes a batch of 64 × 64 × 3–sized inputs.

To define the stem, we use the previously defined stem() function.

Defines Inception blocks. Note that each Inception block has different numbers of filters.

```
    inc_3a = inception(stem_out, [(64,),(96,128),(16,32),(32,)])
    inc_3b = inception(inc_3a, [(128,),(128,192),(32,96),(64,)])

    maxpool = MaxPool2D((3,3), strides=(2,2), padding='same')(inc_3b)

    inc_4a = inception(maxpool, [(192,),(96,208),(16,48),(64,)])
    inc_4b = inception(inc_4a, [(160,),(112,224),(24,64),(64,)])
```

Defines auxiliary outputs

```
    aux_out1 = aux_out(inc_4a, name='aux1')

    inc_4c = inception(inc_4b, [(128,),(128,256),(24,64),(64,)])
    inc_4d = inception(inc_4c, [(112,),(144,288),(32,64),(64,)])
    inc_4e = inception(inc_4d, [(256,),(160,320),(32,128),(128,)])

    maxpool = MaxPool2D((3,3), strides=(2,2), padding='same')(inc_4e)

    aux_out2 = aux_out(inc_4d, name='aux2')

    inc_5a = inception(maxpool, [(256,),(160,320),(32,128),(128,)])
    inc_5b = inception(inc_5a, [(384,),(192,384),(48,128),(128,)])
    avgpool1 = AvgPool2D((7,7), strides=(1,1), padding='valid')(inc_5b)

    flat_out = Flatten()(avgpool1)
    out_main = Dense(200, activation='softmax', name='final')(flat_out)

    model = Model(inputs=inp, outputs=[out_main, aux_out1, aux_out2])
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam', metrics=['accuracy'])
    return model
```

The Flatten layer flattens the average pooling layer and prepares it for the fully connected layers.

The final pooling layer is defined as an Average pooling layer.

When compiling the model, we use categorical cross-entropy loss for all the output layers and the optimizer adam.

The final prediction layer that has 200 output nodes (one for each class)

Table 6.5
Summary of the filter counts of the Inception modules in the Inception Net v1 model.
C(n × n) represents an n × n convolution layer, and MaxP(m × m) represents an m × m max-pooling layer.

Inception layer	C(1 × 1)	C(1 × 1); before C(3 × 3)	C(3 × 3)	C(1 × 1); before C(5 × 5)	C(5 × 5)	C(1 × 1); after MaxP(3 × 3)
Inc_3a	64	96	128	16	32	32
Inc_3b	128	128	192	32	96	64
Inc_4a	192	96	208	16	48	64
Inc_4b	160	112	224	24	64	64
Inc_4c	128	128	256	24	64	64

Table 6.5 *Summary of the filter counts of the Inception modules in the Inception net v1 model. C(nxn) represents a nxn convolution layer, whereas MaxP(mxm) represents a mxm max-pooling layer. (continued)*

Inception layer	C(1 × 1)	C(1 × 1); before C(3 × 3)	C(3 × 3)	C(1 × 1); before C(5 × 5)	C(5 × 5)	C(1 × 1); after MaxP(3 × 3)
Inc_4d	112	144	288	32	64	64
Inc_4e	256	160	320	32	128	128
Inc_5a	256	160	320	32	128	128
Inc_5b	384	192	384	48	128	128

Continuation of the Inception module specifications.

Although the layer definitions themselves are similar to previously encountered Keras layers, the way the model is defined and compiled introduces new concepts. Inception Net is a multi-output model, meaning it produces one final output and two auxiliary outputs. In Keras, such models are defined by passing a list of outputs when instantiating the Model object.

When compiling the model, a single loss function can be specified and automatically applied to all outputs. In this case, categorical cross-entropy is used for both the final and auxiliary outputs, along with the Adam optimizer, which is a widely used adaptive optimization algorithm. Model accuracy is also tracked during training.

Once the inception_v1() function is defined, the model can be instantiated directly by calling it.

At this point, the full pipeline has been completed. The dataset has been downloaded, explored, and analyzed. Image data pipelines were created using ImageDataGenerator, and the dataset was split into training, validation, and test sets. Finally, a state-of-the-art image classification model—Inception Net v1—was implemented. The next step is to explore how the Inception architecture evolved over time.

6.3.4 Other Inception Models

Having implemented Inception Net v1, it becomes easier to understand subsequent versions of the architecture. Since the introduction of v1, five additional Inception models have been proposed. This section provides a high-level overview of their evolution.

Inception v1

Inception Net v1 introduced several groundbreaking ideas. The most notable contribution is the Inception block, which allows multiple convolutional kernel sizes to operate at the same depth within the network. This design increases sparsity, reduces the number of parameters, and lowers the risk of overfitting.

Because Inception Net v1 is a deep model with approximately 20 layers, GPU memory usage can become a serious concern. This issue is addressed by using 1×1 convolutions to reduce the channel depth of feature maps when necessary. Additionally, auxiliary output layers are introduced in intermediate stages of the network to combat unstable gradients by providing additional supervision during training.

Inception v2

Inception Net v2 was introduced to address representational bottlenecks—situations where a layer lacks sufficient capacity to learn meaningful representations. This version restructures layer sizes to ensure sufficient representational power throughout the network while maintaining similar architectural principles.

Another major improvement in v2 is the further reduction of parameters through convolution factorization. Larger convolutional kernels, such as 5×5 and 7×7, are replaced with sequences of smaller 3×3 convolutions.

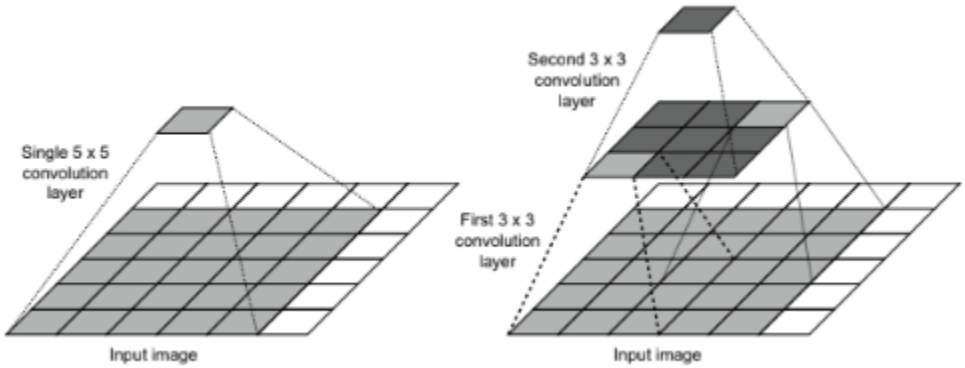


Figure 6.16
A 5 × 5 convolution layer compared with two stacked 3 × 3 convolution layers.

Replacing a single 5 × 5 convolution with two 3 × 3 convolutions reduces the number of parameters by approximately 28%. The architectural differences between Inception v1 and v2 blocks are illustrated below.

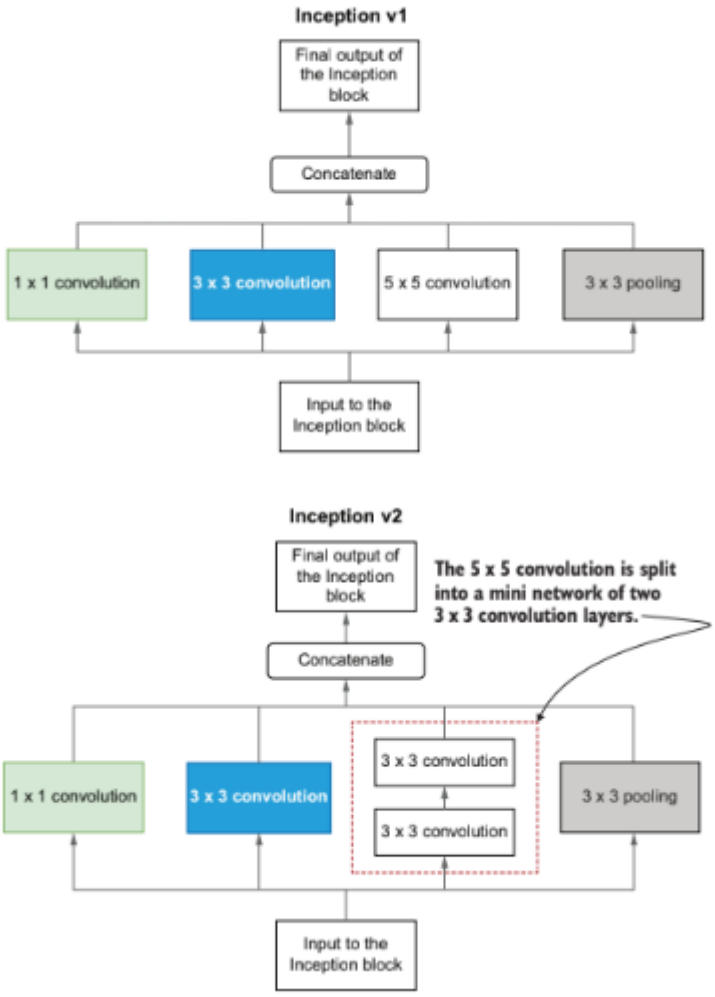


Figure 6.17
Inception block in Inception Net v1 versus Inception Net v2.

In practice, this factorization is implemented by stacking convolution layers, as shown in the provided TensorFlow code example. Additionally, convolution factorization can be extended further by decomposing an n × n convolution into a sequence of 1 × n and n × 1 convolutions.

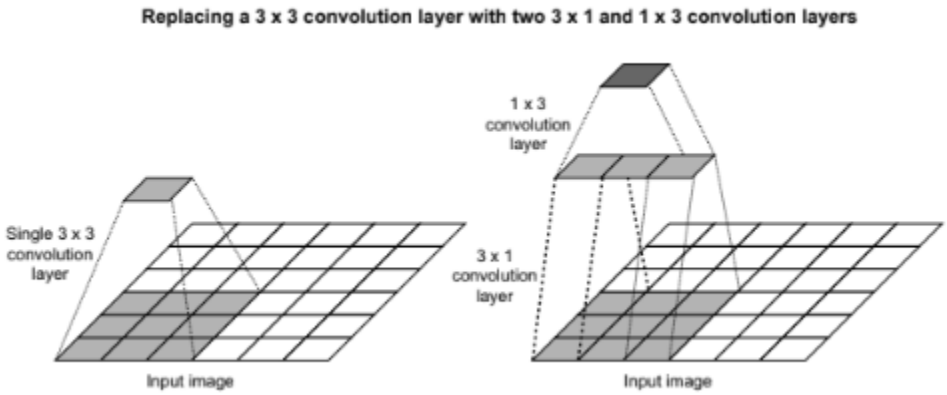


Figure 6.18
A 3 × 3 convolution layer factorized into 3 × 1 and 1 × 3 convolutions.

This technique yields a parameter reduction of approximately 33% for a 3 × 3 convolution and is most effective in deeper layers of the network.

Inception v3

Inception v3 was introduced alongside v2 and primarily differs through the systematic use of batch normalization. Batch normalization normalizes layer outputs by subtracting the mean and dividing by the standard deviation, then applying trainable scaling and shifting parameters. This stabilizes training by preventing activations from becoming too large or too small and allows the network to learn optimal normalization behavior.

Inception v4

Inception v4 focuses on architectural simplification rather than introducing new concepts. The stem and several internal components are streamlined to improve performance without increasing complexity. Because these changes primarily involve hyperparameter tuning, the model is not explored in detail.

Inception-ResNet v1 and v2

Inception-ResNet models combine Inception blocks with residual connections. Residual connections, introduced in deep residual networks, add the output of an earlier layer directly to a deeper layer. This creates shortcut paths that allow gradients to flow more easily during backpropagation.

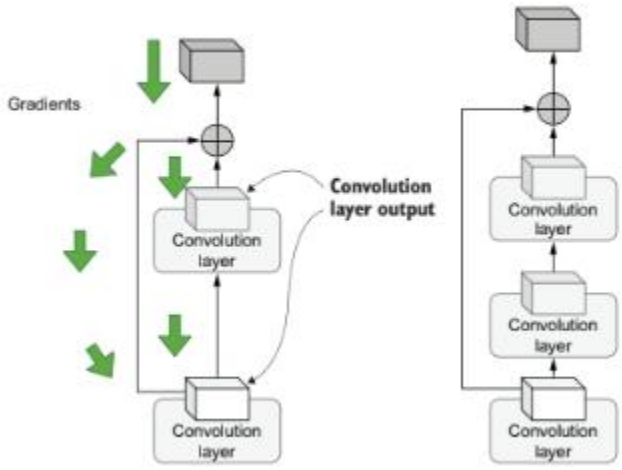


Figure 6.19
Illustration of residual (skip) connections and their effect on gradient flow.

Residual connections reduce training difficulty in very deep networks and are responsible for performance improvements across many domains. Inception-ResNet models will be discussed in more detail in the next chapter.

6.4 Training the Model and Evaluating Performance

With the architecture defined, the next step is to train the model and analyze its performance. Model training involves optimizing the network parameters so that it can produce correct predictions for unseen data. Training is performed over multiple epochs, where each epoch consists of several iterations. An iteration corresponds to processing a single batch of data.

To ensure consistent training behavior, a helper function is defined to compute the number of steps per epoch based on dataset size and batch size.

Listing 6.7
Training the Inception Net model.

During training, the model is trained for a fixed number of epochs. At the end of each epoch, performance is evaluated on the validation set. After training is complete, the model is evaluated on the test dataset, which has not been used during training.

```
from tensorflow.keras.callbacks import CSVLogger
import time
import os

if not os.path.exists('eval'):
    os.mkdir('eval')

csv_logger = CSVLogger(os.path.join('eval','1_eval_base.log'))

history = model.fit(
    x=train_gen_aux,
    validation_data=valid_gen_aux,
    steps_per_epoch=get_steps_per_epoch(0.9*500*200,batch_size),
    validation_steps=get_steps_per_epoch(0.1*500*200,batch_size),
    epochs=50,
    callbacks=[csv_logger]
)

if not os.path.exists('models'):
    os.mkdir("models")
model.save(os.path.join('models', 'inception_v1_base.h5'))
```

Creates a directory called eval to store the performance results

This is a Keras callback that you pass to the fit() function. It writes the metrics data to a CSV file.

By fitting the model, you can see that we are passing the train and validation data generators to the function.

Saves the model to disk so it can be brought up again if needed

The model.fit() function accepts the training data generator, validation generator, number of steps per epoch, validation steps, number of epochs, and optional callbacks. Sample training logs demonstrate that training accuracy reaches approximately 94%, while validation accuracy plateaus around 30%.

After training, model performance on the test set is evaluated using model.evaluate().

The results show that the model achieves roughly 27–30% accuracy on both validation and test data. While this indicates that the model correctly classified approximately 3,000 out of 10,000 images, it also reveals a significant gap between training and validation performance—an indicator of overfitting.

Overfitting can arise due to suboptimal architecture choices, insufficient regularization, or the absence of pretrained weights. These issues will be addressed in the following chapter.

Summary

Exploratory data analysis is a crucial prerequisite for effective modeling. Keras data generators provide an efficient mechanism for loading image data from disk. Inception Net v1 is a state-of-the-art CNN architecture designed to reduce overfitting and memory usage through architectural sparsity and auxiliary outputs. The training process involves careful separation of training, validation, and test data. Overfitting occurs when a model performs well on training data but fails to generalize, highlighting the need for regularization and architectural improvements.