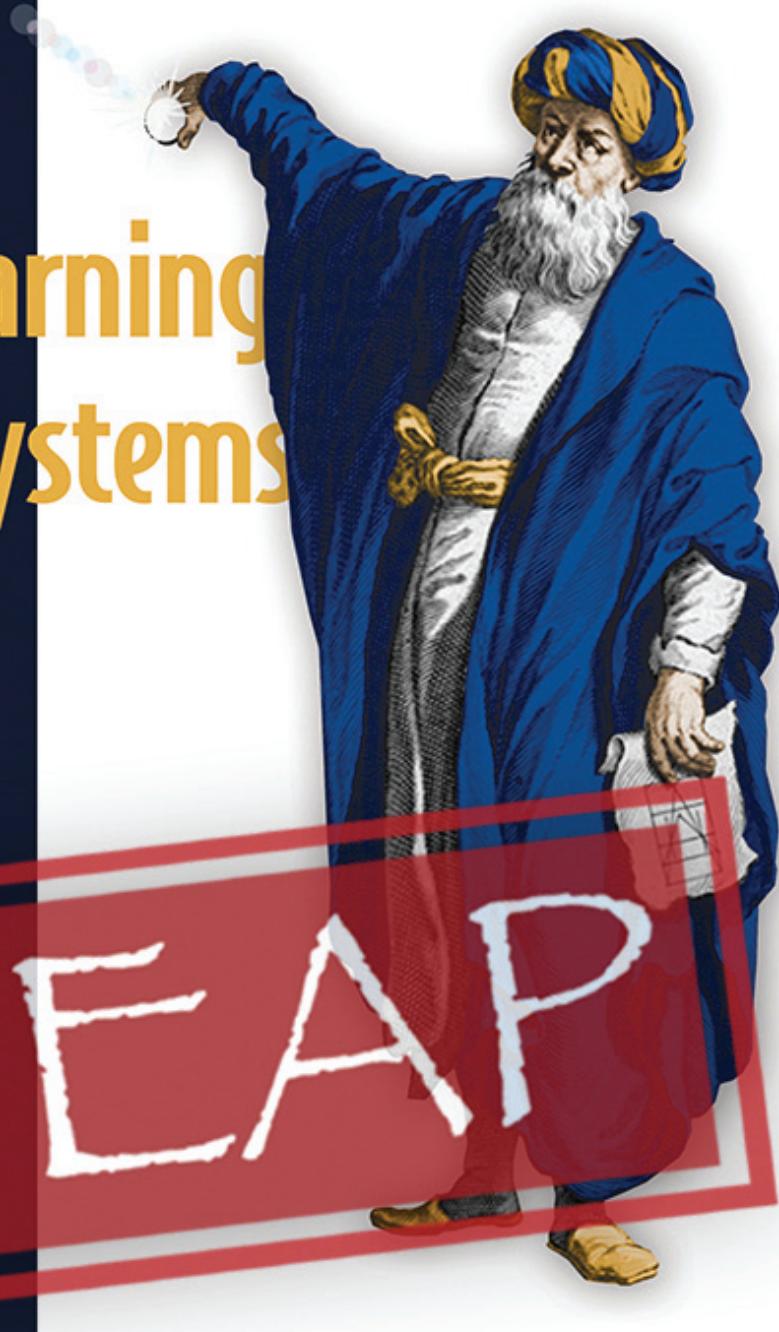


Deep Learning for Vision Systems

Mohamed Elgendi



MANNING



MEAP Edition
Manning Early Access Program
Grokking Deep Learning for Computer Vision
Version 6

Copyright 2019 Manning Publications

For more information on this and other Manning titles go to
manning.com/

welcome

Thank you for purchasing the MEAP for *Grokking Deep Learning for Computer Vision*. I'm very excited that you are here! You are making a great decision to learn deep learning and computer vision. The timing couldn't be more perfect. Computer vision is one of the areas that's been advancing rapidly thanks to the huge AI and deep learning advances that took place in the past few years. For example, neural networks are now allowing self-driving cars to navigate around other cars, pedestrians, and other obstacles. It is also greatly improving face recognition technologies, enabling smartphones to recognize faces to unlock a phone or smarlocks to unlock doors. Computer vision applications like these and others have become a staple in our daily lives. I wouldn't be surprised if some time in the near future your couch or television is able to recognize people in your house and react according to their personal preferences. By more than just recognizing objects, deep learning has given computers the power to imagine and create new things like art that wasn't there before, or create new faces of real human beings (or other objects).

The main reason I'm personally excited about deep learning for computer vision and what drew me to this field in the first place is the way rapid advances in AI research are enabling new applications to be built every day and across different industries that were not possible just a few years ago. The unlimited possibilities of computer vision research is what got me excited to write this book. By learning these tools, you will be empowered to invent some of these new products and applications yourself. Even if you end up not working on computer vision per se, you will find many concepts of this book are useful for your deep learning algorithms and architectures. While our main focus is computer vision applications, we cover the most important deep learning architectures like artificial neural networks (ANNs), convolutional networks (CNNs), recurrent networks (RNNs), and many more that are transferable to other domains like natural language processing (NLP) and voice user interface (VUI).

Finally, as this is the MEAP, if there is anything in these first few chapters that you feel does not make sense, please tweet your comments or questions to me @moelgendi. I would be happy to help -- and more importantly, I want to know if any section of the book is not fulfilling my personal commitment to write the most comprehensive computer vision book with the lowest barriers to entry possible, so that I can adjust it for the final published work.

It is an honor to have your time and attention, and please don't hesitate to reach out.

— Mohamed Elgendi

brief contents

PART 1: DEEP LEARNING FOUNDATION

- 1 Welcome to Computer Vision*
- 2 Deep learning and neural networks*
- 3 Convolutional Neural Networks (CNNs)*
- 4 Structuring Deep Learning Projects and Hyperparameters tuning*

PART 2: IMAGE CLASSIFICATION AND OBJECT DETECTION

- 5 Advanced CNN Architectures*
- 6 Transfer Learning*
- 7 Object detection with YOLO, SSD and R-CNN*

PART 3: GENERATIVE MODELS

- 8 Generative Adversarial Networks (GANs)*
- 9 Style transfer*
- 10 Face Recognition*

PART 4: CLOSING AND LAST WORDS

- 11 The Ethics of Artificial Intelligence*

Table of Contents

1	Welcome to Computer Vision	1
1.1	Computer vision intuition	2
1.1.1	What is visual perception?.....	2
1.1.2	Vision systems.....	3
1.1.3	Human vision systems.....	3
1.1.4	AI vision systems	4
1.1.5	Sensing device	4
1.1.6	Interpreting device.....	5
1.1.7	Can machine learning achieve better performance than human brain?.....	7
1.2	Applications of computer vision.....	8
1.2.1	Image classification	8
1.2.2	Object detection and localization.....	10
1.2.3	Automatic image captioning	10
1.2.4	Generate art (Style Transfer)	11
1.2.5	Create images.....	11
1.2.6	Conclusion	13
1.3	Computer Vision Pipeline - The big picture.....	14
1.4	Input image	17
1.4.1	Image as functions.....	17
1.4.2	How computers see images?	18
1.4.3	Color images.....	19
1.5	Image preprocessing	22
1.5.1	What is image processing?	22
1.5.2	Why image preprocessing?	22
1.6	Feature extraction	26
1.6.1	What is a feature in computer vision?	26
1.6.2	What makes a good (useful) feature?.....	27
1.6.3	Extracting features (hand-craft vs automatic extracting).....	30
1.6.4	Traditional machine learning uses hand-crafted features	30
1.6.5	Deep learning automatically extract features	30
1.6.6	Why use features?	31
1.7	Classifier learning algorithm.....	33
1.8	Chapter summary and takeaways.....	35
2	Deep learning and neural networks	39
2.1	The Perceptron intuition	40
2.1.1	What is a perceptron?.....	41
2.1.2	How does the perceptron learn?	45
2.1.3	Is one neuron enough to solve complex problems?	46
2.2	Multi-Layer Perceptron (MLP)	48
2.2.1	Multi-Layer Perceptron Architecture	49
2.2.2	What are the Hidden Layers?	50
2.2.3	How many layers and how many nodes in each layer?.....	51
2.2.4	MLP Takeaways	52
2.3	Activation functions.....	54
2.3.1	Linear Transfer Function.....	55
2.3.2	Heaviside Step Function (Binary classifier)	56
2.3.3	Sigmoid/Logistic function	57
2.3.4	Softmax Function.....	59
2.3.5	Hyperbolic Tangent Function (tanh).....	59
2.3.6	Rectified Linear Unit (ReLU).....	60
2.3.7	Leaky ReLU.....	61
2.4	Feedforward	64
2.4.1	Feedforward calculations	65
2.4.2	Feature learning.....	66
2.5	Error functions.....	71

2.5.1	What is the error function?	71
2.5.2	Why do we need an error function?.....	71
2.5.3	Error is always positive	71
2.5.4	Mean Square Error (MSE).....	72
2.5.5	Cross Entropy.....	73
2.5.6	A final note on errors and weights.....	74
2.6	Optimization algorithms.....	77
2.6.1	What is Optimization?.....	77
2.6.2	Batch Gradient Descent (BGD)	81
2.6.3	Stochastic Gradient Descent (SGD).....	87
2.6.4	Mini-batch Gradient Descent (MN-GD)	88
2.6.5	Gradient descent takeaways	89
2.7	Backpropagation	91
2.7.1	What is backpropagation?	91
2.7.2	Backpropagation takeaways	95
2.8	Chapter summary and takeaways.....	96
2.9	Project: Build Your first Neural Network	99
3	Convolutional Neural Networks (CNNs)	100
3.1	Image classification using MLP	102
3.1.1	Input layer	102
3.1.2	Hidden Layers	104
3.1.3	Output Layer	105
3.1.4	Putting it all together	105
3.1.5	Drawbacks of MLPs in processing images	108
3.1.6	Conclusion	111
3.2	CNNs Architecture	112
3.2.1	The big picture	112
3.2.2	A closer look on feature extraction	114
3.2.3	A closer look on classification	115
3.3	Basic components of the CNN.....	117
3.3.1	Convolutional layers (CONV)	118
3.3.2	Pooling layers or subsampling (POOL)	126
3.3.3	Why use a pooling layer?.....	128
3.3.4	Fully connected layers (FC)	131
3.4	Image classification using CNNs.....	134
3.4.1	Build the model architecture	134
3.4.2	Model summary	134
3.4.3	Number of parameters (weights)	136
3.5	Add Dropout layers to avoid overfitting	138
3.5.1	What is overfitting?	138
3.5.2	What is a dropout layer?	138
3.5.3	Why do we need dropout layers?	139
3.5.4	Where does dropout layer go in the CNN architecture?	140
3.6	Convolution over colored images (3D images).....	142
3.7	Chapter summary and takeaways.....	149
3.8	Project: Image classification for colored images (CIFAR-10 dataset)	151
3.8.1	Load the dataset	151
3.8.2	Image preprocessing	151
3.8.3	Define the model architecture	154
3.8.4	Compile the model	158
3.8.5	Train the model	159
3.8.6	Load the model with the best val_acc	161
3.8.7	Evaluate the model	161
4	Structuring Deep Learning Projects and Hyperparameters tuning	163
4.1	Define the performance metrics	164
4.1.1	Is accuracy the best metric to evaluate the model?	164
4.1.2	Confusion matrix	165
4.1.3	Precision and Recall	166
4.1.4	F-Score	166
4.2	Design a baseline model	168
4.3	Get your data ready for training	170

4.3.1	Split your data into train/validation/test datasets	170
4.3.2	Data preprocessing	172
4.4	Evaluate the model and interpret its performance (error analysis)	175
4.4.1	Diagnose for overfitting and underfitting.....	175
4.4.2	Plot the learning curves	177
4.4.3	Exercise: build, train, evaluate a simple network.....	178
4.5	Improve the network and tune hyperparameters	181
4.5.1	When to collect more data vs tuning hyperparameters?.....	181
4.5.2	Parameters vs. hyperparameters	182
4.5.3	Neural networks hyperparameters.....	182
4.5.4	Network architecture	183
4.5.5	Learning and optimization	185
4.5.6	Regularization techniques to avoid overfitting.....	196
4.6	Batch normalization (BN)	200
4.6.1	The covariance shift problem	200
4.6.2	Covariance shift in neural networks	201
4.6.3	How does batch normalization work?	202
4.6.4	Batch normalization implementation in Keras.....	203
4.6.5	Batch normalization recap	204
4.7	Chapter summary and takeaways.....	205
4.7.1	Final thoughts.....	205
4.7.2	Tips on hyperparameters tuning	205
4.7.3	Deep learning foundation takeaways	206
4.7.4	What should I do next?	206
4.7.5	An advice from the author	207
4.8	Project: Achieve >90% accuracy on the CIFAR-10 image classification project.....	208
4.8.1	Import dependencies	208
4.8.2	Get the data ready for training	209
4.8.3	Build the model architecture	210
4.8.4	Train the model	212
4.8.5	Evaluate the model	213
4.8.6	Plot learning curves.....	213
4.8.7	Further improvements	214
5	Advanced CNN Architectures.....	215
5.1	CNN design patterns	217
5.1.1	Pattern #1.....	217
5.1.2	Pattern #2.....	217
5.1.3	Pattern #3.....	218
5.2	LeNet-5.....	219
5.2.1	LeNet architecture	219
5.2.2	LeNet-5 implementation in Keras.....	220
5.2.3	Set up the learning hyperparameters	221
5.2.4	LeNet performance on MNIST dataset	222
5.3	AlexNet	223
5.3.1	AlexNet architecture.....	224
5.3.2	Novel features of AlexNet.....	224
5.3.3	AlexNet implementation in Keras.....	226
5.3.4	Set up the learning hyperparameters	229
5.3.5	AlexNet performance on CIFAR dataset	230
5.4	VGGNet.....	230
5.4.1	Novel features of VGGNet.....	231
5.4.2	4.2. VGGNet Configurations	232
5.4.3	VGG-16 in Keras	233
5.4.4	Learning hyperparameters	235
5.4.5	VGGNet performance on CIFAR dataset	235
5.5	Inception and GoogLeNet	235
5.5.1	Novel features of Inception	236
5.5.2	Inception module - naive version	237
5.5.3	Inception module with dimensionality reduction	239
5.5.4	Inception architecture	242
5.5.5	GoogleNet in Keras.....	245
5.5.6	Learning hyperparameters.....	250
5.5.7	Inception performance on CIFAR dataset	250

5.6	ResNet.....	250
5.6.1	Novel features of ResNet	251
5.6.2	Residual blocks.....	254
5.6.3	ResNet implementation in Keras.....	256
5.6.4	Learning hyperparameters.....	258
5.6.5	ResNet performance on CIFAR dataset.....	259
5.7	Summary and takeaways.....	260
6	Transfer Learning	262
6.1	What are the problems that transfer learning is solving?	264
6.2	What is transfer learning?.....	266
6.3	How transfer learning works.....	273
6.3.1	How do neural networks learn features?.....	274
6.3.2	What about the transferability of features extracted at later layers in the network?	277
6.4	Transfer learning approaches.....	277
6.4.1	Pretrained network as a classifier	277
6.4.2	Pretrained network as a feature extractor	279
6.4.3	Fine-tuning.....	281
6.5	Choose the appropriate level of transfer learning.....	283
6.5.1	Scenario #1: target dataset is small and similar to source dataset	284
6.5.2	Scenario #2: target dataset is large and similar to the source dataset	284
6.5.3	Scenario #3: target dataset is small and different from the source dataset	284
6.5.4	Scenario #4: target dataset is large and different from the source dataset	285
6.5.5	Recap of the transfer learning scenarios.....	285
6.6	Open-source datasets	286
6.6.1	MNIST	287
6.6.2	Fashion-MNIST	287
6.6.3	CIFAR-10	288
6.6.4	ImageNet	289
6.6.5	Microsoft's COCO	290
6.6.6	Google's Open Images.....	291
6.6.7	Kaggle.....	292
6.7	Chapter summary and takeaways.....	292
6.8	Project 1: A pretrained network as a feature extractor.....	293
6.9	Project 2: Fine tuning.....	298
7	Object detection with YOLO, SSD and R-CNN	307
7.1	General object detection framework.....	309
7.1.1	Region proposals	310
7.1.2	Network predictions	311
7.1.3	Non-maximum suppression (NMS).....	312
7.1.4	1.4. Object detector evaluation metrics.....	313
7.2	Region-Based Convolutional Neural Networks (R-CNNs)	317
7.2.1	R-CNN	317
7.2.2	Fast R-CNN.....	321
7.2.3	Faster R-CNN	324
7.2.4	Recap of the RCNN family.....	333
7.3	Single Shot Detection (SSD)	335
7.3.1	High level SSD architecture	336
7.3.2	Base network.....	338
7.3.3	Multi-scale feature layers	340
7.3.4	Non-maximum Suppression.....	345
7.4	You Only Look Once (YOLO)	346
7.4.1	How YOLOv3 works	347
7.4.2	YOLOv3 Architecture	351
7.5	Chapter summary and takeaways.....	354

8 Generative Adversarial Networks (GANs)	355
8.1 GANs Architecture	357
8.1.1 The Discriminator Model	359
8.1.2 The Generator Model.....	362
8.1.3 Training the GAN.....	365
8.1.4 GAN Minimax Function	369
8.2 Evaluate GAN models	371
8.2.1 Inception score.....	372
8.2.2 Fréchet Inception Distance (FID).....	373
8.2.3 Which evaluation scheme to use?.....	373
8.3 Popular GANs Applications	374
8.3.1 Text-to-Photo Synthesis	374
8.3.2 Image-to-image translation (Pix2Pix GAN).....	375
8.3.3 Image Super-Resolution GAN (SRGAN).....	377
8.3.4 Ready to get your hands dirty?.....	377
8.4 Building your own GAN project	377
8.4.1 Import libraries.....	379
8.4.2 Download and visualize the dataset.....	379
8.4.3 Build the generator.....	380
8.4.4 Build the discriminator	382
8.4.5 Build the combined model.....	383
8.4.6 Build the training function.....	384
8.4.7 Train and observe results	385
8.4.8 Closing.....	386
8.5 Summary and takeaways	387

1

Welcome to Computer Vision

“Just like to HEAR is not the same as to LISTEN, to TAKE PICTURES is not the same as to SEE.”

-- Fei-Fei Li

Hello! I'm very excited that you are here! You are making a great decision to learn deep learning and computer vision. The timing couldn't be more perfect. Computer vision is one of the areas that's been advancing rapidly thanks to the huge AI and deep learning advances that took place in the past few years. Neural networks are now allowing self-driving cars to figure out where other cars and pedestrians are to navigate around them. We are witnessing computer vision applications in our daily lives more and more with all the smart devices in our home -- from security cameras to door locks. It is also making face recognition work much better than ever before and we can see that smartphones are able to recognize faces to unlock a phone and smart locks can unlock doors. I wouldn't be surprised if some time in the near future your couch or television is able to recognize specific people in your house and react according to their personal preferences. It's not just about recognizing objects -- deep learning has given computers the power to imagine and create new things like art that wasn't there before, new objects, and even unique, realistic faces of human beings.

The main reason that I'm personally excited about deep learning for computer vision and what drew me to this field is how rapid advances in AI research are enabling new applications to be built every day and across different industries that were not possible just a few years ago. The unlimited possibilities of computer vision research is what inspired me to write this book. By learning these tools, perhaps you will be able to invent some of these new products and applications yourself. Even if you end up not working on computer vision per se, you will find many concepts of this book are useful for some of your deep learning algorithms and architectures. That is because, while our main focus is computer vision applications, we cover the most important deep learning architectures like artificial neural networks (ANNs),

convolutional networks (CNNs), recurrent networks (RNNs), and many more that are transferable to other domains like natural language processing (NLP) and voice user interface (VUI).

The high-level layout of this chapter will be as follows:

1. **Computer vision intuition:** First, we will start with the visual perception intuition and learn the similarities in humans and machine vision systems. We will see how vision systems have two main components: 1) sensing device and 2) interpreting device. Each is tailored to fulfill a specific task.
2. **Applications of computer vision:** Here we will take a bird's eye view of the deep learning algorithms used in different CV applications. We will then discuss vision in general for different creatures.
3. **Computer vision pipeline:** Lastly, we will zoom in to the second component of vision systems interpreting system. We will walk through the sequence of steps taken by vision systems to process and understand image data. These are referred to as a vision pipeline. The CV pipeline is composed of 4 main steps: 1) image input, 2) image preprocessing, 3) feature extraction, and 4) ML model to interpret the image. Here, we will dive deeper into each one of these 4 steps. We will talk about the image formation and how computers see images. Then, we will quickly review image processing techniques and extracting features.

Ready? Let's get started!

1.1 Computer vision intuition

The core concept of any AI system is that it can: 1) perceive its environment and 2) take actions based on this perceptions. Computer vision is concerned with the **visual perception** part. It is the science of **perceiving** and understanding the world through images and videos by constructing a physical model of the world so that the AI system can then take the appropriate actions. For humans, vision is only one aspect of perception. We perceive the world through our sight, but also through sound, smell and many other sensors that we have. It is similar with AI systems, vision is just one way to understand the world. Based on the application that you are building, you select the sensing device the captures the world as needed.

1.1.1 What is visual perception?

Visual perception, at its most basic definition, is the act of observing patterns and objects through sight or visual input. If we take autonomous vehicles for example, visual perception means understanding all the surrounding objects and their specific details like pedestrians around it, or if there is a specific lane than it needs to be centered in, detecting traffic signs and actually understand what they mean. That's why the word "perception" is added to the definition. We are not just looking to capture the surrounding environment. We are trying to build systems that can actually understand the environment through visual inputs.

1.1.2 Vision systems

In the past decades, traditional image processing techniques were used to be considered as computer vision systems but that is not totally accurate. Having machines process an image is completely different from *understanding what's happening within the image* which is not a trivial task. Image processing now became just a piece of a bigger, more complex system that aims to interpret the image content and doesn't just stop at the processing part.

1.1.3 Human vision systems

Vision systems are pretty much the same for humans, animals, insects, and most living organisms. They consist of 1) a sensor or an eye to capture the image and 2) a brain to process and interpret the image. The system then outputs a prediction of the image components based on the data we have about the information we extracted from the image.

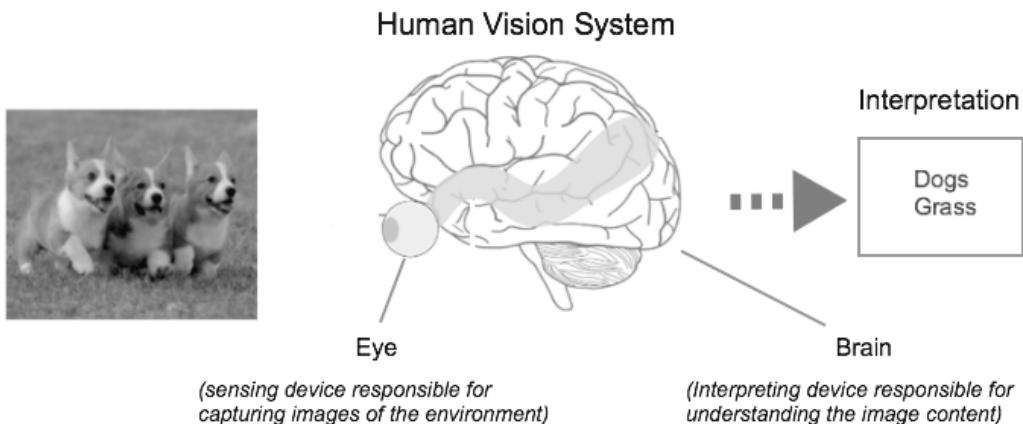


Figure 1.1

Let's see how the human vision system works. Suppose we want to interpret the image of dogs above. We look at it and directly understand that the image consists of a bunch of dogs (three to be specific). It comes pretty natural to us to classify and detect objects in this image because we have been trained over the years to identify dogs. Think of yourself when you saw a dog for the first time, you definitely didn't know what it is. Then somebody told you that this is a dog. A couple more times of this experiment and you will have been trained to identify dogs. Now we run a follow up exercise and show you a picture of a horse - when you look at the image your brain will start analyzing the object features as follows: Hmmm, it has 4 legs, long face, long ears. Could it be a dog? "Wrong, this is a horse" you were told. Then your brain will adjust some parameters in its algorithm to learn the differences between dogs and horses. Congratulations! You just trained your brain to classify dogs and horses. Can we add more animals to the equation like cats, tigers, cheetah, etc? Definitely. You can train your brain to identify almost anything. Same with computers. You can train machines to learn and identify objects but humans are much more intuitive than machines. It takes a few images for

you to learn to identify most objects. Whereas with machines, it takes thousands, and in more complex cases millions, of image samples to learn to identify objects.

Let's look at the above example from the machine learning eye:

- We learned to identify dogs by looking at examples of several dog-labeled images that is called **supervised learning**.
- **Labeled data:** data for which you already know the target answer. We showed you an image sample of a dog and told you this is a dog. Your brain learned to associate the features you have seen with this label, a dog.
- We then showed you different object, a horse, and asked you to predict it. At first your brain thought that it is a dog because it hasn't seen horses before and it got it confused with the dog features. When we told you that your prediction is wrong. Your brain went back to adjust its parameters to learn the horse features. As in: "yes, both have 4 legs, but the horse's legs are longer. Let's learn that longer legs means a horse". Then we run this experiment many times until the brain makes no mistakes. This is called **training by trial and error**.

1.1.4 AI vision systems

Scientists were inspired by the human vision system and in the past few years we have done an amazing job to extend this visual ability to machines. So, in order to mimic the human vision system, we need the same two main components: 1) a sensing device to mimic the function of the eye, and 2) a powerful algorithm to mimic the brain function in interpreting and classifying the image content.

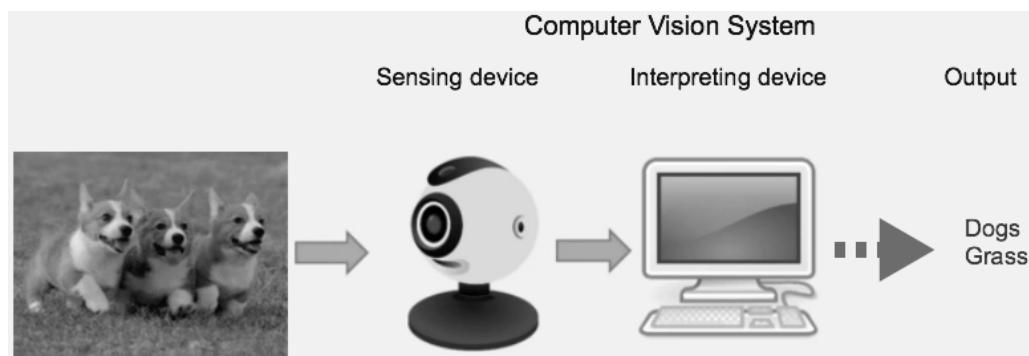


Figure 1.2

1.1.5 Sensing device

Vision systems are designed to fulfill a specific task. And an important aspect of designing your vision system is selecting the sensing device to capture the surroundings of that specific environment. Whether it is a camera, radar, x-ray, CT Scan, Lidar, or a combination of more than one device to provide a full scene of the environment to fulfil the task in hand.

Let's look at the autonomous vehicles (AV) example again. The main goal of the AV vision system is to allow the car to understand the environment around it to move from point A to point B safely and in a timely manner. To fulfil this goal, vehicles are equipped with a combination of cameras and sensors that can detect 360 degrees of movements of pedestrians, cyclists, vehicles, road work and other objects from up to three football fields away.

Here are some of the sensing devices usually used in self-driving cars to perceive its surrounding area:

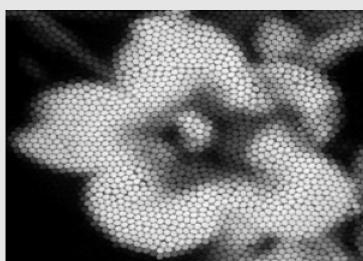
- Lidar: a radar-like technique that uses invisible pulses of light to create a high-resolution 3D map of the surrounding area.
- Cameras can see street signs and road markings, but cannot measure distance
- Radars can measure distance and velocity, but cannot see in fine detail

Other medical diagnosis applications will use x-rays or CT scans as sensing device. Or maybe you need to use some other radars to capture the ground landscape for agricultural vision systems. In essence, there are a variety of vision systems. Each is designed to perform their own task. The first step in designing vision systems is to identify the task they are built for. This is something to keep in mind when designing end-to-end vision systems.

Same thing among living creatures. Animals, humans, and insects all have eyes as sensing devices. But, not all eyes have the same structure and output image quality and resolution. They are all tailored for their specific needs. Bees, for instance, and many other insects have compound eyes that consist of multiple lenses (as many as 30,000 lenses in a single compound eye). Compound eyes have low resolution which make them not so good at recognizing objects from far distances. But they are very sensitive to motions which is very essential while flying with high speed. Which fulfills their need to survive. Bees don't need high resolution picture. Their vision systems are built to allow them to pick up the smallest movements while flying fast.



Compound eyes



How bees see a flower

Figure 1.3

1.1.6 Interpreting device

This is where most of the computer vision algorithm lies. The interpreter is the **brain** of the vision system. Its role is to take the output image from the sensing device and learn features and patterns to identify its objects. So, we need to build a brain. Simple! Scientists were

inspired by how our brains work and tried to reverse engineer the central nervous systems to get some insights on how to build our own artificial brain. Thus, *Artificial Neural Networks (ANNs)* were born.

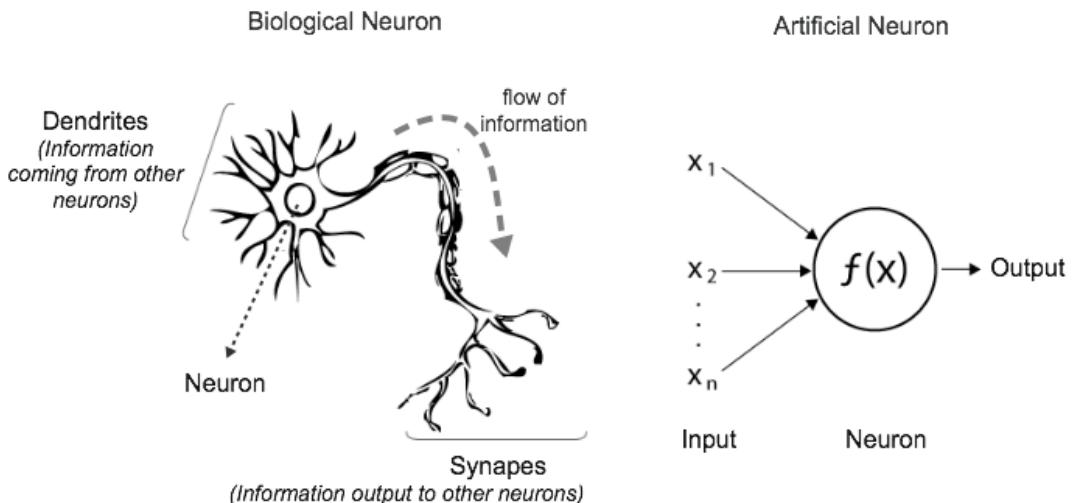


Figure 1.4

In the figure above, we can see that there is an analogy between the biological neurons and artificial systems. Both contain a main processing element called *neuron* with input signal X_1, X_2, \dots, X_n and an output.

The learning behavior of biological neurons inspired scientists to create a network of neurons that are connected to each other. Imitating how information is processed in the human brain, each individual artificial neuron will fire a signal to all neurons that it's connected to when enough of its input signals are activated. Thus, neurons have a very simple mechanism on the individual level (as you will see in the next chapter), but when you start to have many, many (millions) of these neurons stacked in layers and connected together, each of these neurons are connected to thousands of other neurons, this yields a learning behavior. Building a multi-layer neural network is called *Deep Learning*.

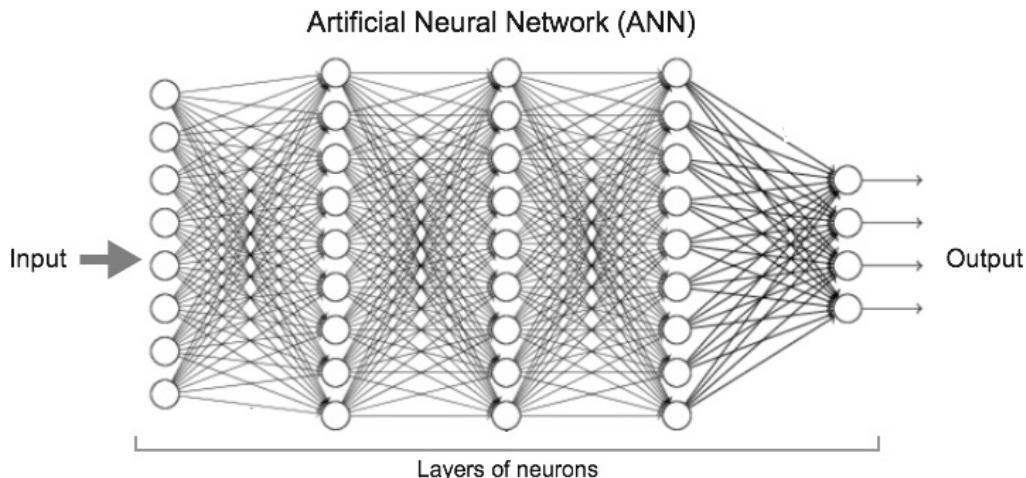


Figure 1.5

Deep learning methods learn representations through a sequence of transformations of the data through layers of neurons. In this book we will explain different deep learning architectures like ANNs, Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) and how they are used in computer vision applications.

1.1.7 Can machine learning achieve better performance than human brain?

Well, if you had asked me this question 10 years ago, I would've probably said no, machines cannot surpass the humans' accuracy. But let's take a look at the following two scenarios:

- Suppose you were given a book of 10,000 dog images classifying their breeds and you were asked to learn the properties of each breed. How long would it take you to study the different 130 breeds in 10,000 images. And if you were given a test of 100 dog images and asked to label them based on what you learned. Out of the hundred, how many will you get right? Well, a neural network that is trained in a couple of hours can achieve more than 95% accuracy.
- On the creation side, a neural network can study the patterns in the strokes, colors, and shading of a particular piece of art. From there, it can transfer the style from the original artwork into a new image based on the analysis and create a new piece of original art within a few seconds.

It turns out that the recent AI and deep learning advances have allowed machines to surpass humans visual abilities in many image classification and object detection applications and is rapidly expanding to many other applications. But don't take my word for it. Let's revisit this question after the next section where we review some of the most popular CV application using deep learning technology.

1.2 Applications of computer vision

Computers started to be able to recognize human faces in images decades ago, but now artificial intelligence systems are rivaling computers ability to classify objects in photos and videos. Thanks to the dramatical evolution in both computational power and the amount of data available, AI and deep learning have managed to achieve superhuman performance on many complex visual perception tasks like image search and captioning, image and video classification, and object detection, etc. Moreover, deep neural networks are not restricted to computer vision tasks: they are also successful at *Natural Language Processing (NLP)* and *Voice User Interface (VUI)* tasks. In this book, we will focus on visual applications that are applied in computer vision tasks.

Deep learning is used in many computer vision applications to recognize objects and their behavior. In this section, I'm not going to attempt to list all the CV applications that are out there. I would need an entire book for that. Instead, I'm going to give you a bird's eye view on some the most popular deep learning algorithms that we are going to cover in this book and their possible applications across different industries. Among these industries: autonomous cars, drones, robots, in-store cameras and medical diagnostic scanners that can detect lung cancer in early stages to save millions of lives.

1.2.1 Image classification

Image classification is the task of assigning a label to an image from a predefined set of categories. Convolutional Neural Networks (CNNs) is a neural networks type that truly shines in processing and classifying images in many different applications. To name a few:

4. Lung cancer diagnosis:

Lung cancer is such a growing problem. Several computer vision companies decided to tackle this challenge using deep learning technology. The main reason lung cancer is very dangerous is that when doctors diagnose patients with lung cancer it is usually in the mid or later stages. When diagnosing lung cancer, doctors typically use their eyes to examine CT scan images looking for small nodules in the lungs. At early stages, the nodules are usually very small and harder to spot.

Almost every single lung cancer started as a smaller nodule and they appear in a big variety of shapes which takes a doctor years to learn all these different shapes. Doctors are very good at identifying mid and large size of nodules such as 6-10 millimeters. But looking at smaller nodules like 4 mm or smaller sometimes they have difficulties identifying them. Deep learning networks, specifically CNNs, are now able to learn these features automatically from XRay and CT scan images and detect small nodules early before they become deadly.

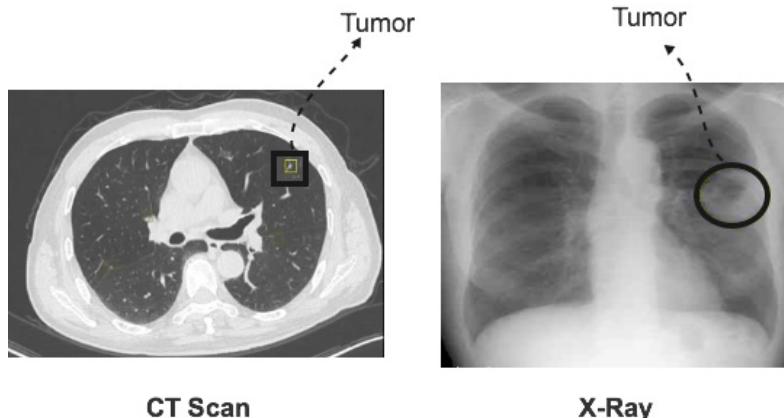


Figure 1.6

5. Traffic sign recognition:

Traditionally, standard computer vision methods were employed to detect and classify traffic signs, but these required considerable and time-consuming manual work to handcraft important features in images. Instead, by applying deep learning to this problem, we can create a model that reliably classifies traffic signs, learning to identify the most appropriate features for this problem by itself.



Figure 1.7

More and more image classification tasks are being solved with Convolutional Neural Networks (CNNs). Due to its high recognition rate and fast execution, CNNs have enhanced most of computer vision tasks, both existing and new ones. Just like the cancer diagnosis and traffic signs examples, you can feed tens or hundreds of thousands of images into a convolutional network to label them into as many classes as you want. Other image classification examples can include: identifying people and objects, classify different animals (like cats vs. dogs vs. horses, etc.), different breeds of animals, types of lands that are suitable for agriculture, and the list goes on. In short, if you have a set of *labeled* images, convolutional networks can classify them into a set of predefined classes.

1.2.2 Object detection and localization

Image classification problems are the most basic applications for CNNs. In which, each image contains only one object and our task is to label these images. But, if we are aiming to reach human levels of understanding, we have to add complexity to these networks so that they are able to recognize multiple objects and their locations in an image. To do that, we are going to build object detection systems like YOLO (You Only Look Once), SSD (Single-Shot Detection), and Faster R-CNN that not only classify images, but can locate and detect each object in images that contain multiple objects. These deep learning systems can look at an image, break it up into smaller regions, and label each region with a class so that a variable number of objects in a given image can be localized and labeled. You can imagine that such a task is a basic prerequisite for applications like autonomous systems.



Figure 1.8

1.2.3 Automatic image captioning

We have seen how CNNs are used in image classification and object detection. Another very popular neural network architecture that is used in CV applications is Recurrent Neural Networks (RNNs). RNNs are networks in loops in them allowing information to persist. It's like

adding memory to our networks. Both CNNs and RNNs are used in automatic captioning applications that allows computers to recognize behavior in an image and caption it correctly.



Figure 1.9

1.2.4 Generate art (Style Transfer)

Neural style transfer is one of the most interesting CV applications that is used to transfer the style from one image to another. The basic idea of style transfer is this: You take one image, say of a city, and then apply a style of art to that image, say the Starry Night painting (by Vincent Van Gogh) and outputs the same city in the original image but as though it was painted by Van Gogh.



Figure 1.10

This is actually a neat application. The astonishing thing is, if you know any painters, you know that it can take days or even weeks to finish a painting. And yet here is a machine that can paint a new image inspired by some style in a matter of seconds.

1.2.5 Create images

I kept this to the end because, although the earlier examples are truly impressive CV applications of AI, but this is where I see the real magic is happening. The magic of creation. In 2014, Ian Goodfellow, invented a new deep learning model that can imagine new things called GANs (Generative Adversarial Models). The name can make it sound a little

intimidating, but I promise you that it is not. It is an evolved CNN architecture which is considered a major advancement in deep learning. So, when you understand CNNs, GANs will make a lot more sense to you.

GANs, are sophisticated deep learning models that are able to generate stunningly accurate synthesized images of objects, people, and places among other things. If you give them a new set of images, they can make entirely new images that are realistic and have never been seen before. For example, StackGANs is one of the GAN architecture variations that can take a textual description of an object then generate a high resolution image of the object matching that description. This is not just running an image search on a database. These photos are never seen before and totally imaginary.

This small blue bird has a short pointy beak and brown on its wings



This bird is completely red with black wings and pointy beak



Figure 1.11

GANs is one of the most promising advancements in machine learning over the past few years. Research into GANs is very new and the results are already overwhelmingly promising. Most of the applications of GANs so far have been for images. But it makes you wonder, if machines are given the power of imagination to create pictures, what else can they create? Can your favorite movies, music, and maybe even books be created by computers in the future? The ability to synthesize one data type (text) to another (image) will eventually allow us to create all sorts of entertainment with just detailed text descriptions.

In October 2018, an AI-created painting called "The Portrait of Edmond Belamy" was sold for \$432,500. The artwork features a fictional person named Edmond de Belamy, possibly French and — to judge by his dark frock coat and plain white collar — a man of the church.



Figure 1.12

The artwork was created by a team of three 25-year-old French students, used GANs (generative adversarial network) to create the painting. The network was trained on a dataset of 15,000 portraits painted between the 14th and 20th centuries, and then it tried to create one of its own. The team printed the image, framed it, and signed it with part of the GAN's algorithm.

1.2.6 Conclusion

All these applications, and many more, are used across industries like healthcare, retail, autonomous systems (like cars, drones, robots), security, agriculture, etc. Lot's of very interesting computer vision applications that we can build using deep learning technology. In fact, in this book, you will get the chance to get your hands dirty with the code implementations and learn how to build and train neural networks that can do all of the above applications. By the end of this book, you will be equipped to build state-of-the-art computer vision applications to solve real-world problems.

1.3 Computer Vision Pipeline - The big picture

Okay, now that I got your attention, let's dig one level deeper into the computer vision systems. Remember earlier in this chapter how we discussed that vision systems composed of two main components: 1) sensing device and 2) interpreting device? In this section, we will zoom in to the interpreting device component to take a look at the pipeline it uses to process and understand images.

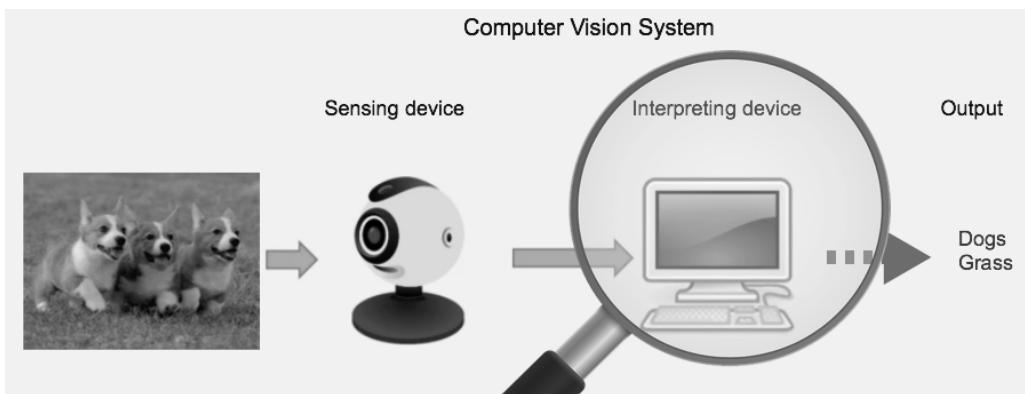


Figure 1.13

Applications of computer vision vary, but a typical vision system uses a similar sequence of distinct steps to process and analyze image data. These are referred to as a vision pipeline. Many vision applications start off by acquiring images and data , then processing that data, performing some analysis and recognition steps, then finally make a prediction based on the extracted information.

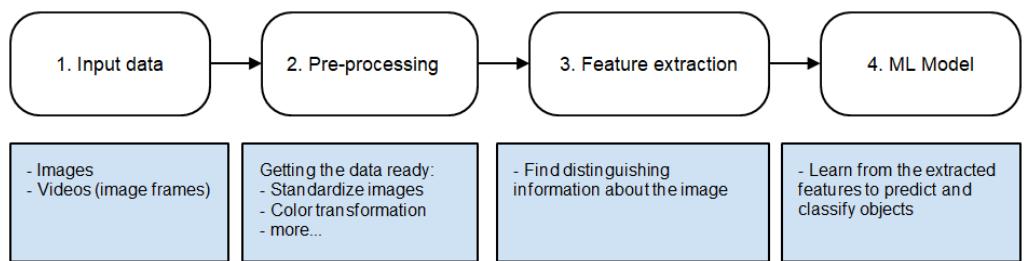


Figure 1.14

Let's apply the above pipeline on an image classifier example. Suppose we have an image of a motorcycle, and we want the model to predict the probability of the object from the following classes: car, motorcycle, and dog.

- Image classifier is an algorithm that takes in an image as input and outputs a label or “class” that identifies that image.
- A class in machine learning is the output category of your data. You can call these categories as well.

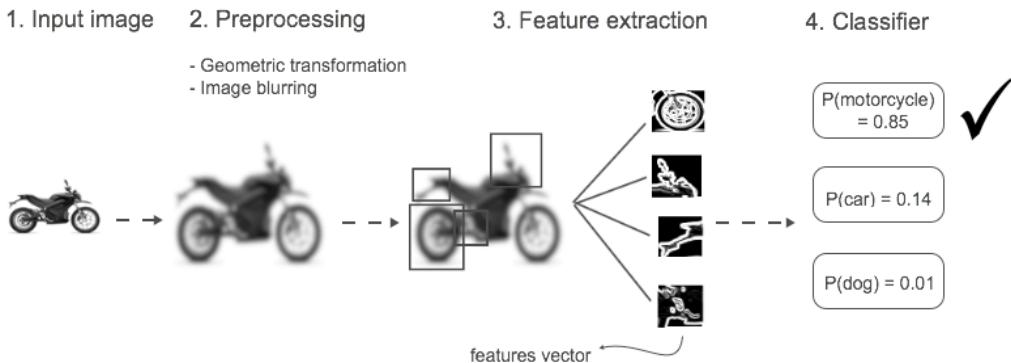


Figure 1.15

Here is how the image flows through the classification pipeline:

1. First, a computer receives visual **input** from an imaging device like a camera. This is typically captured as an image or a sequence of images forming a video.
2. Each image is then sent through some **pre-processing** steps whose purpose is to standardize each image. Common pre-processing steps include resizing an image, blurring, rotating, change its shape or transforming the image from one color to another - like from color to grayscale. Only by standardizing each image, for example: making them the same size, can you then compare them and further analyze them in the same way.
3. Next, we **extract features**. Features are what help us define certain objects, and they are usually information about object shape or color. For example, some features that distinguish a motorcycle is shape of the wheel, headlights, mudguards, and many more. The output of this process is a *features vector* that is a list of unique shapes that identify the object.
4. Finally, these features are fed into a **classification model!** This step looks at the features vector from the previous step and predicts the class of the image. Pretend that you are the classifier model for a few minutes and let's go through the classification process: You look at the list of features in the vector feature one-by-one and try to know what's in the image.
5. First you see a feature of a wheel - could this be a car, motorcycle or a dog? Clearly it is not a dog because dogs don't have wheels (at least normal dogs not robots!). Then, this is could be an image of a car or a motorcycle

6. Then you move on to the next feature "the headlights". There is a higher probability that this is a motorcycle than a usual car
7. The next feature is "rear mudguard" - again, there is a higher probability it is a motorcycle
8. The object has only two wheels, hmm, this is closer to a motorcycle
9. And you keep going through all the features like the body shape, pedal, etc. until you have created a better guess of the object in the image

The output of this process is the probabilities of each class. As you can see in the above example, the dog has the lowest probability of 1% whereas there is 85% probability that this is a motorcycle. You can see that, although the model was able to predict the right class with high probability, it is still a little confused in distinguishing between cars and motorcycles since it predicted that it there is a 14% chance this is an image of a car. Since we know that it is a motorcycle, then we can say that our ML classification algorithm is 85% accurate. Not so bad! To improve this accuracy, we may need to do more of step 1 (acquire more training images) or step 2 (more processing to remove noise) or step 3 (extract better features) or step 4 (change the classifier algorithm and tune some hyperparameters or even more training time). Many different approaches that we can take to improve the performance of our model. They all lie in one or more of the pipeline steps. And that is what we will be working on in this book.

That was the big picture of how images flow through the computer vision pipeline. Now, let's zoom in to another one level deeper into each one of the pipeline steps:

1. Input image
2. Image preprocessing
3. Feature extraction
4. Classification

1.4 Input image

In computer vision applications, we deal with images or videos data. Let's talk about grayscale and color images for now then in later chapters we will talk about videos since videos are just a sequential frames of images stacked together to form a video.

1.4.1 Image as functions

Images can be represented as a function of two variables X and Y which define a two dimensional area. Digital images are made of a grid of pixels. The *Pixel* is the raw building block of an image. Every image consists of a set of pixels in which their values represent the *intensity of light* that appears in a given place in our image. Let's take a look at the motorcycle example again after applying the pixels grid to it.

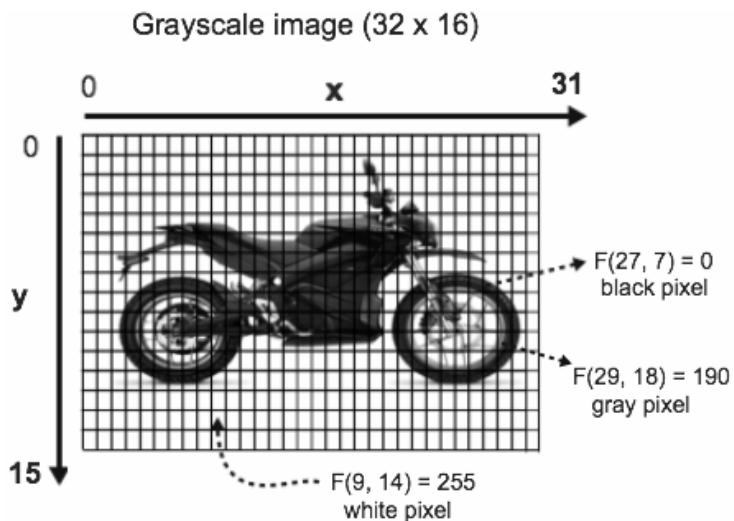


Figure 1.16

The image above has a size of 32×16 . This means that the dimensions of the image is 32 pixels wide and 16 pixels tall. The X axis starts from 0 to 31 and Y axis from 0 to 16. Overall, the image has $32 \times 16 = 512$ pixels. In this grayscale image, each pixel contains a value that represents the *intensity of light* on this specific pixel. The pixel values vary from 0 to 255. Since the pixel value represent the intensity of light, then the value 0 represent very dark pixels (black), 255 is very bright (white) and the values in between represent the intensity on the grayscale.

You can see that the image coordinate system is similar to the Cartesian coordinate system: images are two dimensional and lie on the x-y plane. The origin (0, 0) is at the top left of the image. To represent a specific pixel we use the following notations: "F" as a function, "x,y" the location of the pixel on the x, y coordinates. For example, the pixel located in $x=9$ and $y=14$

is white, this is represented by the following function: $F(9, 14) = 255$. Similarly, the pixel $(27,7)$ that lies on the front wheel of the motorcycle is black, represented as $F(27,7) = 0$.

Grayscale => $F(x, y)$ gives the intensity at position (x, y)

That was for grayscale images. How about color image?

In color images, instead of representing the value of the pixel by just one number, the value is represented by three numbers representing the intensity of each color in this pixel. In RGB system for example, the value of the pixel is represented by three numbers: the intensity of red, intensity of green, and intensity of blue. There are other color systems for images like HSV and LAB. All follow the same concept when representing the pixel value. More on colored images in the next page. Here is the function representing color images in the RGB system:

Color image in RGB => $F(x, y) = [red(x, y), green(x, y), blue(x, y)]$

Thinking of an image as a function is very useful in image processing. This way we can think of an image as a function of $F(x, y)$ and operate on it mathematically to transform it to a new image function $G(x, y)$. Let's take a look at the following image transformation examples:

Table 1.1

Application	Transformation
Darken the image	$G(x, y) = 0.5 * F(x, y)$
Brighten the image	$G(x, y) = 2 * F(x, y)$
Move an object 150 pixels down	$G(x, y) = F(x, y + 150)$
Remove the gray in an image to transform the image into only black and white	$G(x, y) = \{ 0 \text{ if } F(x, y) < 130, 255 \text{ otherwise } \}$

1.4.2 How computers see images?

When we look at an image, we see objects, landscape, colors, etc.. But that's not the case with computers. Consider the picture below - your human brain can look at it and immediately know that it is a picture of a motorcycle. To a computer the image looks like a 2D matrix of the pixels' values which represent intensities across the color spectrum. There is no context here, just a massive pile of data.



What we see

```

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 60 98 43 49 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 60 67 53 85 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 65 54 01 32 54 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 34 54 22 60 40 28 64 33 13 80
26 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 39 17 12 50
32 98 81 28 64 23 67 10 26 38 60 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 69 94 21
24 55 58 05 64 73 99 26 97 17 78 78 96 83 14 88 34 89 43 72
21 36 23 09 78 00 76 44 20 45 38 14 00 45 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 94 35 31 47 55 58 88 24 00 17 54 24 34 29 85 57
86 56 00 45 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 69 05 94 47 69 28 73 92 13 84 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 26 26 79 39 27 98 66
88 36 48 87 57 62 20 72 03 46 33 67 66 55 12 32 63 93 53 69
06 42 16 73 38 25 39 11 24 96 72 18 08 66 29 32 60 62 76 36
20 73 35 29 78 31 90 03 74 31 49 71 48 88 81 16 23 37 05 54
01 70 54 71 83 51 56 69 16 92 33 48 61 43 52 01 89 19 67 48
  
```

What computers see

Figure 1.17

You can see that the image above is of size 24*24. This size shows width and height of an image. Then there are 24 pixels horizontally and 24 vertically. That means there are total 576 (24*24) pixels. If the image is the size of 700 * 500 then the dimensionality of the matrix will be (700, 500). Where each pixel in the matrix represents the intensity of brightness in that pixel. 0 represents black and 255 represents white color.

1.4.3 Color images

In grayscale images, each pixel represents the intensity of only one color. Whereas in standard RGB system, color images have 3 channels (red, green blue). In other words, color images are represented by three matrices: one represents the intensity of red in the pixel, one for green and one for blue.

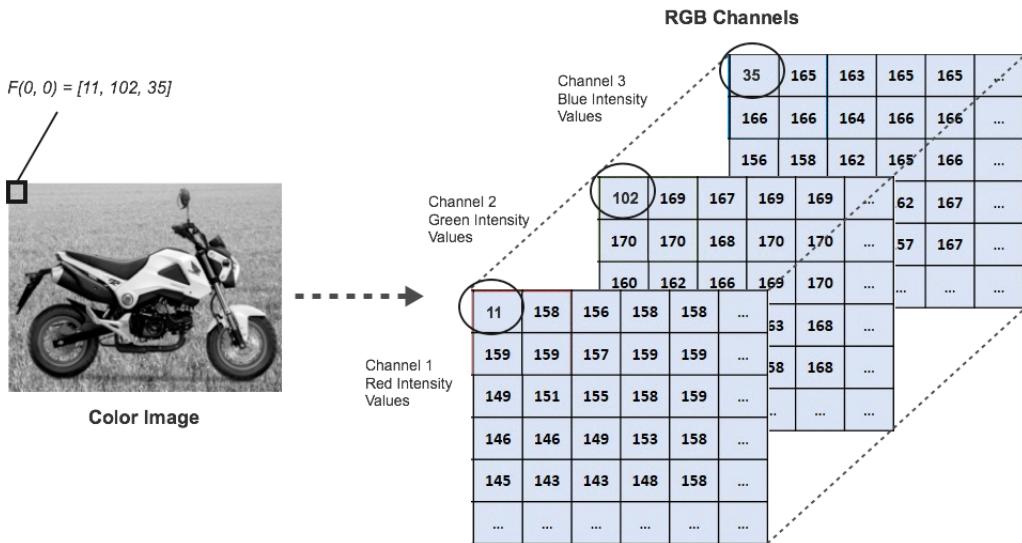


Figure 1.18

As you can see in above image, the color image is composed of three channels red, green and blue. Now the question is, how do computers see this image? Again, the answer is they see the matrix. Unlike grayscale images where we had only one channel. In this case, we will have 3 matrices stacked on top of each other, that's why it's 3D matrix. Dimensionality of 700 * 700 color image will be (700, 700, 3). Let's say, the first matrix represents red channel, then each element of that matrix represents an intensity of red color in that pixel, likewise in green and blue. Each pixel in color image has three numbers (0 to 255) associated with it. These numbers represent intensity of red, green and blue color in that particular pixel.

If we take the pixel (0,0) as an example, we will see that it represents the top-left pixel of the image of green grass. When we view this pixel in the color images, it looks like this:

$$\text{Red} \quad \text{Green} \quad \text{Blue} \\ 11 + 102 + 35 = \text{Forest Green } (11, 102, 35)$$

Figure 1.19

Take a look at the example below of some shades of the color green and their RGB values:

Forest	Forest Green Codes: Hex #0B6623 RGB 11 102 35	Mint Codes: HEX #98FB98 RGB 152 251 152	Mint Green Codes: Hex #98FB98 RGB 152 251 152
Olive	Olive Green Codes: Hex #708238 RGB 112 130 56	Lime Codes: HEX #C7EA46 RGB 199 234 70	Lime Green Codes: Hex #c7ea46 RGB 199 234 70
Jungle	Jungle Green Codes: Hex #29AB87 RGB 41 171 135	Jade Codes: HEX #00A86B RGB 0 168 107	Jade Green Codes: Hex #00A86B RGB 0 168 107

Figure 1.20

To recap - computers see an image as matrices. Grayscale images have one channel (gray), thus we can represent grayscale images in the 2D matrix, where each element represents the intensity of brightness in that particular pixel. Remember, 0 means black and 255 means white. Grayscale images have one channel, whereas color images have three channels RGB (red, green, blue). We can represent color images in the 3D matrix where the depth will be 3.

We've also seen how images can be treated as functions of space. This concept allows us to operate on images mathematically and change or extract information from them. Treating images as functions is the basis of many image processing techniques like converting color to grayscale or scaling an image. Each of these steps are just operating mathematical equations to transform an image pixel-by-pixel.

Grayscale: $f(x, y)$ gives the intensity at position (x, y)

Color image: $f(x, y) = [\text{red}(x, y), \text{green}(x, y), \text{blue}(x, y)]$

1.5 Image preprocessing

1.5.1 What is image processing?

In machine learning projects in general, you usually go through a data preprocessing or cleaning step. As a machine learning engineer, you will spend a good amount of your time in cleaning up and preparing the data before you build your learning model. The goal of this step is to make your data ready for the ML model to make it easier to analyze and process computationally. Same thing with images. Based on the problem you are solving and the dataset in hand, there is some data massaging required before you feed your images to the ML model.

Image processing could be simple tasks like image resizing. Later on, you will learn that in order to feed a dataset of images to a convolutional network, they all have to have be the same size. Other processing tasks can take place like geometric and color transformation or converting color to grayscale and many more. We will be covering different image processing techniques throughout the chapters of this book and in the projects.

1.5.2 Why image preprocessing?

The acquired data are usually messy and coming from different sources. To feed them to the ML model (or neural network), they need to be standardized and cleaned up. More usual than not, preprocessing is used to conduct steps that will reduce the complexity and increase the accuracy of the applied algorithm. We cannot write a unique algorithm for each of the condition in which an image is taken, thus, when we acquire an image, we tend to convert it into a form that would allow a general algorithm to solve it.

Data preprocessing techniques might include:

1. **Convert color images to grayscale to reduce computation complexity:** in certain problems you will find it useful to lose unnecessary information from your images to reduce space or computational complexity.

For example, converting your colored images to grayscale images. This is because in many objects, color is not necessary to recognize and interpret an image. Grayscale can be good enough for recognizing certain objects. Since color images contain more information than black and white images, they can add unnecessary complexity and take up more space in memory (Remember how color images are represented in 3 channels which means that converting it to grayscale will reduce the number of pixels that need to be processed).

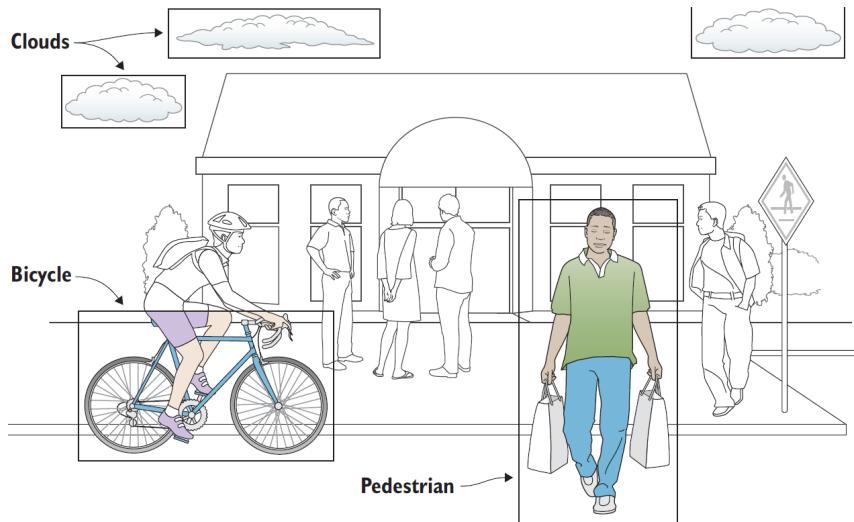


Figure 1.21

In the example above, you can see how patterns in brightness and darkness of an object (intensity) can be used to define the shape and characteristics of many objects. However, in other applications, color is important to define certain objects. Like skin cancer detection which relies heavily on the skin colors (red rashes).

When is color important?

Converting an image to grayscale might not be a good decision for some problems. There are a number of applications for which color is very important. For example, building a diagnostic system to identify red skin rashes in medical images. This system relies heavily on the intensity of the red color in the skin. Removing colors from the image will make it harder to solve this problem. In general, color images provide very helpful information in many medical applications.

Another example of the importance of color in images is lane detection applications in self-driving cars. Where the car has to identify the difference between yellow and white lanes because they are treated differently. Grayscale images do not provide enough information to distinguish between the yellow and white lanes.

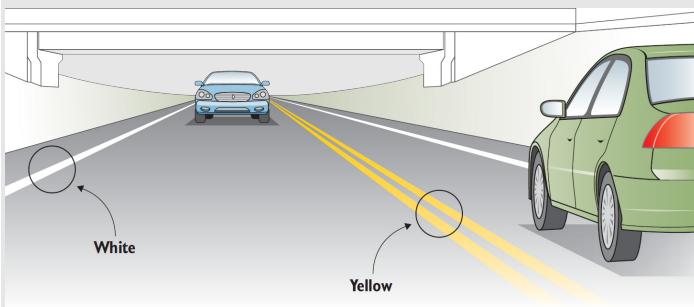


Figure 1.22

The rule of thumb to identify the importance of colors in your problem is to look at the image with the human eye, if you are able to identify the object that you are looking for in a gray image then you probably have enough information to feed to your model. If not, then you definitely need more information (colors) in your images. The same rule can be applied for most other preprocessing techniques that will be discussed next.

2. **Standardize images:** As you will see in chapter 3, one important constraint that exists in some machine learning algorithms, such as CNN, is the need to resize the images in your dataset to a unified dimension. This implies that our images must be preprocessed and scaled to have identical widths and heights before fed to the learning algorithm.
3. **Data augmentation:** Another common pre-processing technique involves augmenting the existing dataset with perturbed versions of the existing images. Scaling, rotations and other affine transformations are typical. This is done to enlarge your dataset and expose the neural network to a wide variety of variations of your images. This makes it more likely that your model recognizes objects when they appear in any form and shape. Here is an example of image augmentation applied to a butterfly image:

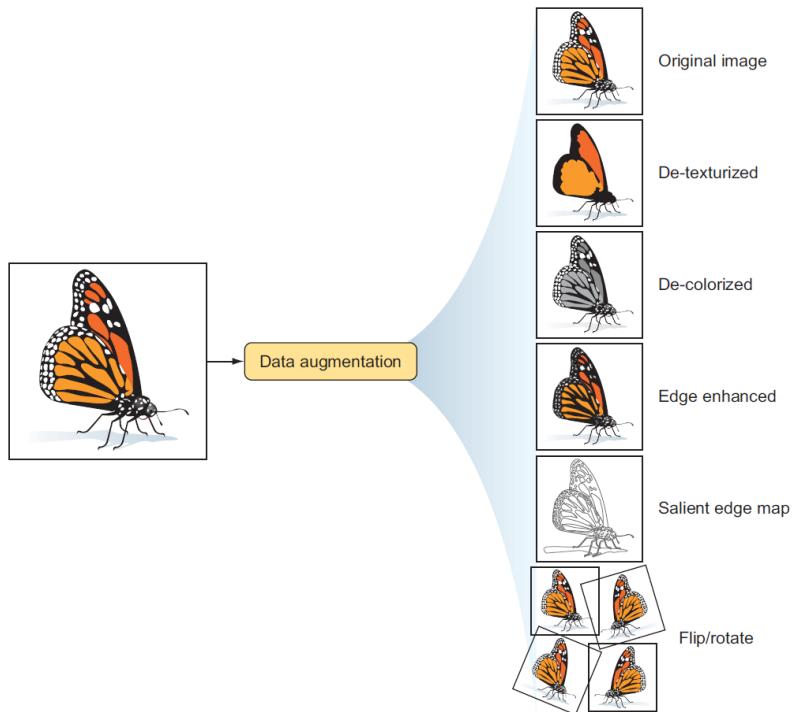


Figure 1.23

4. **Other techniques:** There are many more preprocessing techniques that you can use to get your images ready to train the machine learning model. In some projects, you might need to remove the background color from your images to reduce the noise.

Other projects might require that you brighten or darken your images. In short, any adjustments that you need to apply to your dataset is considered a sort of preprocessing. And you will select the appropriate processing techniques based on the dataset at hand and the problem you are solving. You will see many image processing techniques throughout the projects in this book. That will build your intuition of which ones you would need when working on your own projects.

No free lunch theorem

This is a term that was introduced by David Wolpert in his paper “No Free Lunch Theorems for Optimizations”. You will often hear this term when working on ML projects. It basically means that there is no one prescribed recipe that fits all models. When working on ML projects, you will need to make many choices like building your neural network architecture, tuning hyperparameters, as well as applying the appropriate data preprocessing techniques. No free lunch is used to say that, while there are some rule of thumb approaches to tackle certain problems, but there is really no one recipe that is priori guaranteed to work well in all situations. We must make certain assumptions about the dataset and the problem that we are trying to solve. For some datasets it is best to convert the colored images to grayscale, while for other datasets you might need to keep or adjust the color images.

The good news is, unlike traditional machine learning, deep learning algorithms require minimum data preprocessing because, as you will see in the next few pages, neural networks do most of the heavy lifting in processing an image and extracting features.

1.6 Feature extraction

Feature extraction is a core component of the computer vision pipeline. In fact, the entire deep learning model works around the idea of extracting **useful features** that clearly define the objects in the image. So we are going to spend a little more time here because it is important that you understand what is a feature, what is a vector of features and why we extract features.

A feature in machine learning is an individual measurable property or characteristic of a phenomenon being observed. Features are the input that you feed to your machine learning model to output a prediction or classification. Suppose you want to predict the price of a house, your input features (properties) might include: square_foot, number_of_rooms, bathrooms, etc. and the model will output the predicted price based on the values of your features. Selecting good features that clearly distinguish your objects increases the predictive power of machine learning algorithms.

1.6.1 What is a feature in computer vision?

In computer vision, a feature is a measurable piece of data in your image that is unique to this specific object. It maybe a distinct color in an image or a specific shape such as a line, edge, or an image segment. A good feature is used to distinguish objects from one another. For example, if I give you a feature like a wheel, and ask you to guess whether the object is a motorcycle or a dog. What would your guess be? A motorcycle. Correct! In this case, the wheel is a strong feature that clearly distinguishes between motorcycles and dogs. However, if I give you the same feature (a wheel) and ask you to guess whether the object is a bicycle or a motorcycle. In this case, this feature is not strong enough to distinguish between both objects. Then we need to look for more features like a mirror, license plate, maybe a pedal that collectively describe an object.

In machine learning projects, we want to transform the raw data (image) into a *features* vector to show to our learning algorithm to learn the characteristics of the object.

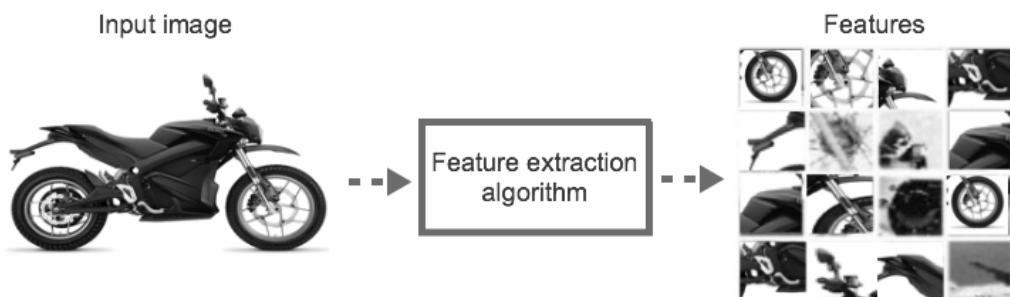


Figure 1.24

In the image above, we feed the raw input image of a motorcycle to a feature extraction algorithm. Let's treat the feature extraction algorithm as a black box for now and we will come back to it very soon. For now, we need to know that the extraction algorithm produces a

vector that contains a list of features. This is called **features vector** that is a 1D array that makes a robust representation of the object.

It is important to call out that the image above reflects features extracted from just one motorcycle. A very important characteristic of a feature is **repeatability**. As in the feature should be able to detect the motorcycles in general not just this specific one. So, in real world problems, the feature will not be an exact copy of the piece in the input image.

Feature after looking
at one image



Feature after looking
at 1000s of images



Figure 1.25

If we take the wheel feature for example, the feature will not look exactly like the wheel on just one motorcycle. Instead, it looks like a circular shape with some patterns that identify wheels in all images in the training dataset. When the feature extractor sees thousands of images of motorcycles, it recognizes patterns that define wheels in general regardless of where they appear in the image and what type of motorcycle it is.

1.6.2 What makes a good (useful) feature?

Machine learning models are only as good as the features you provide. That means coming up with good features is an important job in building ML models. But what makes a good feature? And how can you tell?

Let's discuss this by an example: Suppose we want to build a classifier to tell the difference between two types of dogs, Greyhound and Labrador. Let's take two features and evaluate them: 1) the dogs' height and 2) their eye color.

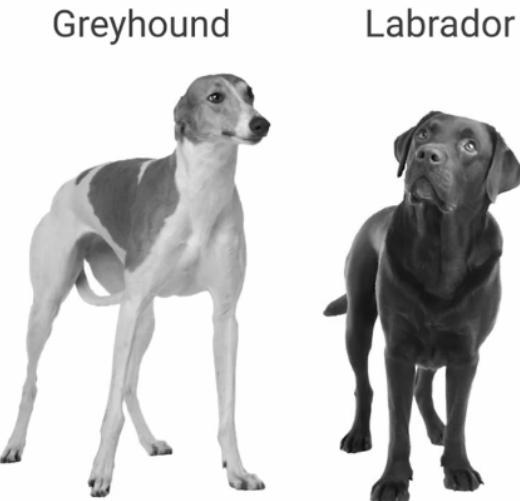


Figure 1.26

Let's begin with height. How useful do you think this feature is? Well, on average, greyhounds tend to be a couple of inches taller than labradors, but not always. There is a lot of variation in the world. So let's evaluate this feature across different values in both breeds population. Let's visualize the height distribution on a toy example in the histogram below:

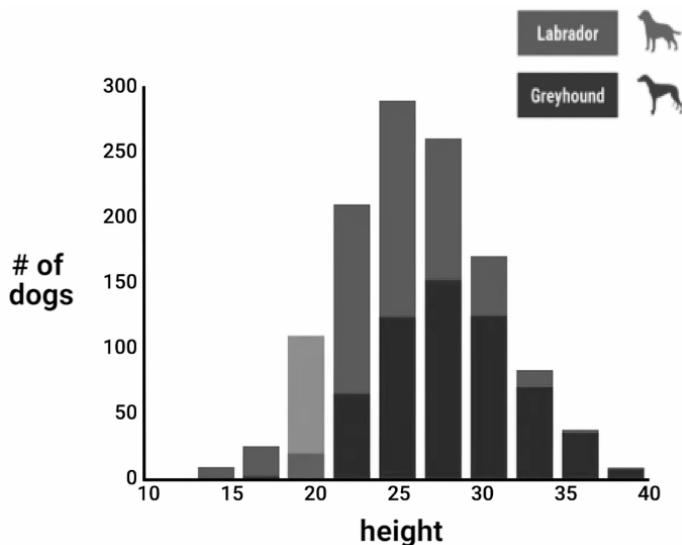


Figure 1.27

From the histogram above, we can see that if the dog's height is 20 inches or less, there is more than 80% probability that this dog is a labrador. On the other side of the histogram, if we look at dogs who are taller than 30 inches, we can be pretty confident that the dog is a greyhound. Now, what about the data in the middle of the histogram (heights from 20 to 30 inches)? We can see that the probability of each type of dog is pretty close. The thought process in this case will be as follows:

if height <= 20:

return higher probability to labrador

if height >= 30:

return higher probability to greyhound

if 20 < height > 30:

look for other features to classify the object

So the "height" of the dog in this case is a useful feature because it helps (adds information) in distinguishing between both dog types. We can keep it. But it doesn't distinguish between greyhounds and labradors in all cases. Which is fine. In ML projects, there is usually no one feature that can classify all objects on its own. That's why in machine learning we almost always need multiple features where each feature captures a different type of information. If only one feature would do the job, we can just write if-else statements instead of bothering with training a classifier.

Similar to what we did earlier with color conversion (color vs grayscale), to figure out which features you should use for a specific problem, do a thought experiment. Pretend you are the classifier. If you want to differentiate between greyhounds and labradors, what information you would need to know? You might ask about the hair length, or the body size, to color, and so on.

Another quick example of a non-useful feature to drive this idea home. Let's look at eye color. Just for this toy example, imagine that we have only two eye colors, blue and brown. Here is what a histogram might look like for this example:

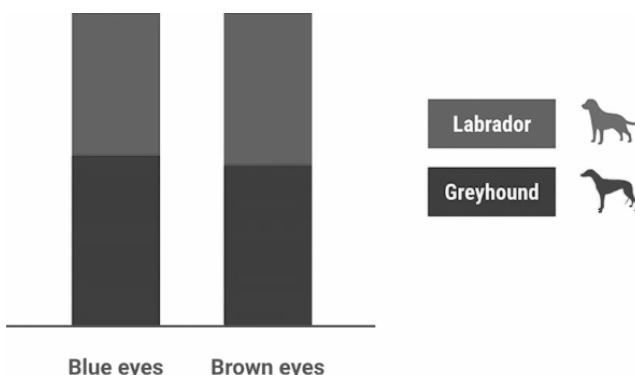


Figure 1.28

It is clear that for most values, the distribution is about 50/50 for both types. So practically this feature tells us nothing because it doesn't correlate with the type of dog. Hence, it doesn't distinguish between greyhounds and labradors.

A good feature will help us recognize an object in all the ways it may appear. Characteristics of a good feature:

- Identifiable
- Easily tracked and compared
- Consistent across different scales, lighting conditions, and viewing angles
- Still visible in noisy images or when only part of an object is visible

1.6.3 Extracting features (hand-craft vs automatic extracting)

Okay, this can be a large topic in machine learning that needs an entire book to discuss. Typically described in the context of a topic called *feature engineering*. In this book we are only concerned with extracting features in images. So, I'm going to touch on the idea very quickly in this chapter to build on it in the next chapters.

1.6.4 Traditional machine learning uses hand-crafted features

In traditional machine learning problems, we spend a good amount of time in **manual** features selection and engineering. In this process we rely on our domain knowledge (or partnering with domain experts) to create features that make machine learning algorithms work better. We then feed the produced features to a classifier like Support Vector Machines (SVM) or Adaboost to predict the output. Some of the hand-crafted feature sets are:

- Histogram of Oriented Gradients (HOG)
- Haar Cascades
- Scale-Invariant Feature Transform (SIFT)
- Speeded Up Robust Feature (SURF)

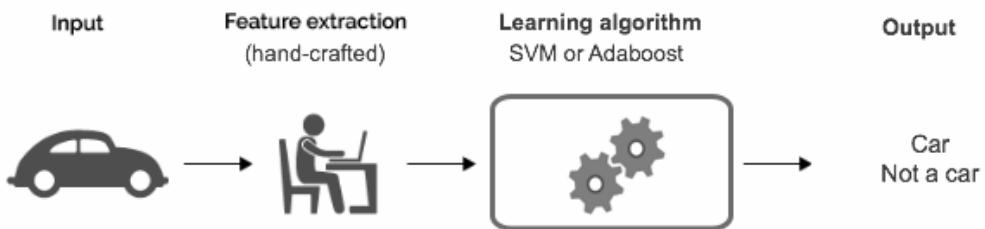


Figure 1.29

1.6.5 Deep learning automatically extract features

In deep learning however, we do not need to manually extract features from the image. The network extracts features **automatically** while and learns their importance on the output by applying weights to its connections. You just feed the raw image to the network and while it passes through the network layers it identifies patterns within the image to create features.

Neural networks can be thought of as feature extractors + classifiers that are end-to-end trainable as opposed to traditional ML models that use hand-crafted features.

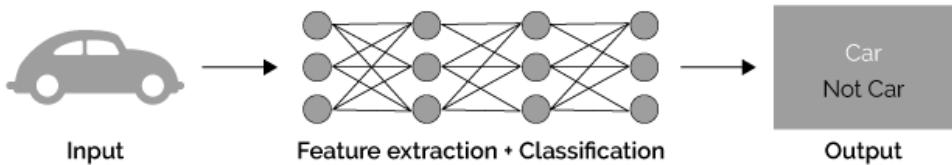


Figure 1.30

How does neural networks distinguish useful features from the non-useful features?

You might get the impression that neural networks only understand the useful features but that's not entirely true. Neural Networks scoop all the features available and give them random weights. During the training process it adjusts these weights to reflect their importance and how they should impact the output prediction. The patterns with the highest appearance frequency will have higher weights and in turn are considered more useful features. Whereas, features with lowest weights will have very low impact on the output. This learning process is going to be discussed in deep details in the next chapter.

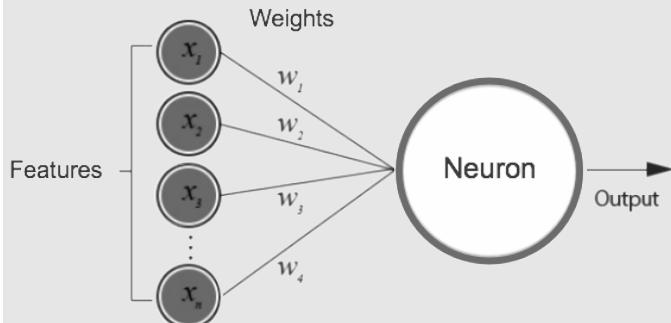


Figure 1.31

1.6.6 Why use features?

The input image has too much extra information that is not necessary for classification. Therefore, the first step after preprocessing the image is to simplify the image by extracting the important information contained in the image and throwing away non-essential information. By extracting important colors or image segments, we can transform complex and large image data into smaller sets of features. This makes the task of classifying images based on their features done simpler and faster.

Consider the example below. Suppose we are given a dataset of 10,000 images of motorcycles each of 1,000 width x 1,000 height. Some images have solid backgrounds and others have busy backgrounds of unnecessary data. When these thousands of images are fed to the feature extraction algorithms, we lose all the unnecessary data that are not important to

identify motorcycles and we only keep a consolidated list of useful features that can then be fed directly to the classifier. This process is a lot simpler than having the classifier look at a dataset of 10,000 images to learn the properties of motorcycles.

Images dataset of 10,000 images

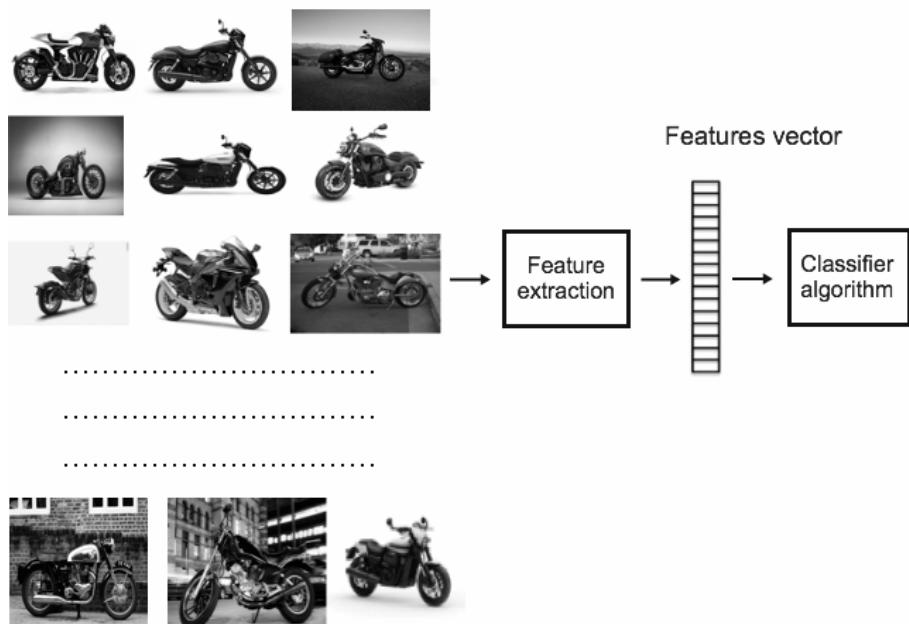


Figure 1.32

1.7 Classifier learning algorithm

Okay, here is what we discussed so far in the classifier pipeline?

1. **Input image:** We've seen how images are represented as functions and learned that computers see images as 2D matrix for grayscale images and 3D matrix (3 channels) for colored images.
2. **Image preprocessing:** Then we discussed some image preprocessing techniques to clean up our dataset and make it ready to be fed to the ML algorithm.
3. **Feature extraction:** we converted our large dataset of images into a vector of useful features that uniquely describe the objects in the image

Now it is time to feed the extracted features vector to the classifier to output a class label for the images (e.g. motorcycle or not).

As we discussed in the previous section, the classification task is done by either one of these types: 1) traditional machine learning algorithms like SVMs and Random Forest, or 2) deep neural network algorithms like CNNs. While traditional ML algorithms might get some decent results for some problems, convolutional neural networks (CNNs) truly shine in processing and classifying images in the most complex problems. This is the topic of this book. We will discuss neural networks in details and learn how they work. For now, I want you to know that neural networks **automatically** extract useful features from your dataset + act as a classifier to output class labels for the images. Input images passes through the layers of the neural network to learn their features layer-by-layer. The deeper your network is (more layers), the more it will learn its features. Hence, the name deep learning. More layers come with some tradeoffs that we will discuss in the next two chapters. The last layer of the neural network usually acts as the classifier that outputs the class label.

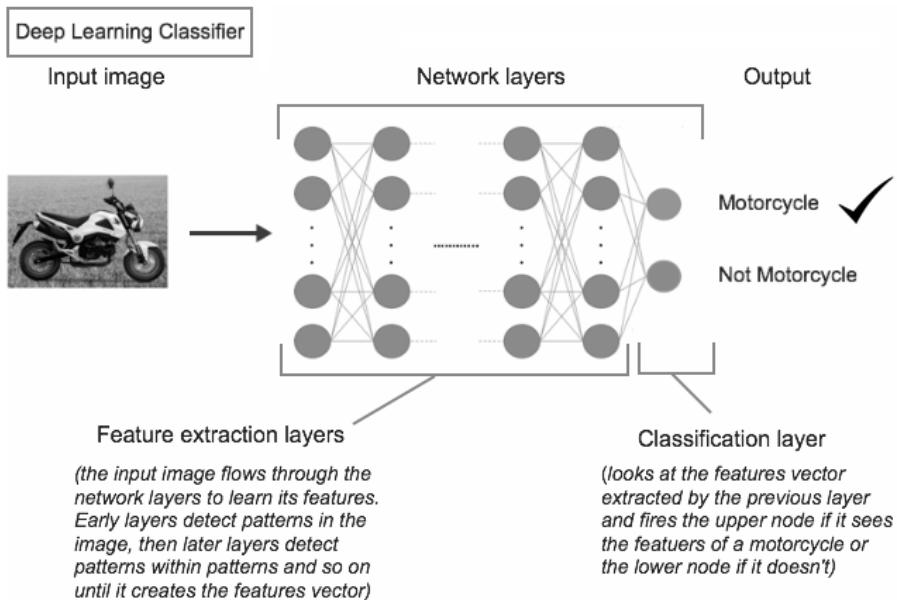


Figure 1.33

1.8 Chapter summary and takeaways

This is an introduction chapter to give you a 30,000 feet overview on computer vision systems and their applications. I don't expect that you would have a deep understanding on the pipeline components yet. We are going to dive deeper into these aspects throughout the chapters of this book. What I want you to take away from this chapter is the following:

- Human vs machine vision system: both contain two basic components: 1) sensing device and 2) interpreting device.

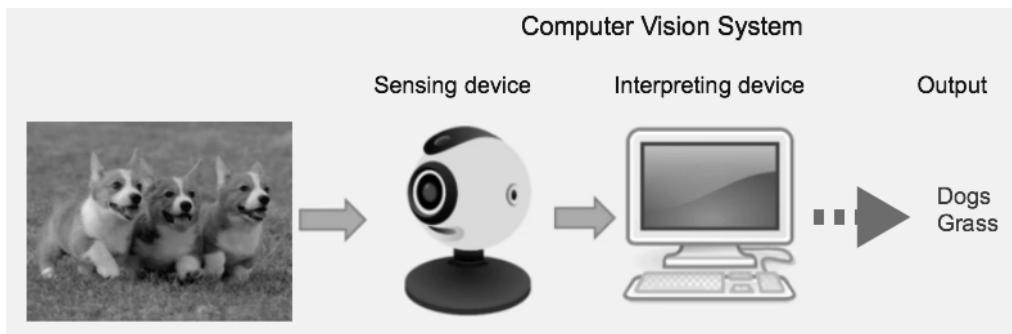


Figure 1.34

- Zooming in to the second component, interpreting device, we will see the pipeline used to process images coming from the sensing device and produce an output prediction of the image content.

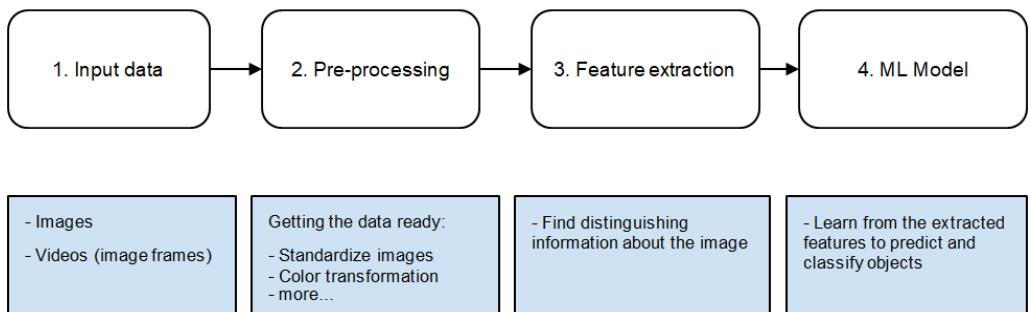


Figure 1.35

- **Input data:** we learned that image can be represented as a function of X and Y. We also say how computers see images as a matrix of pixel values. One channel for grayscale images and 3 channels for color images.

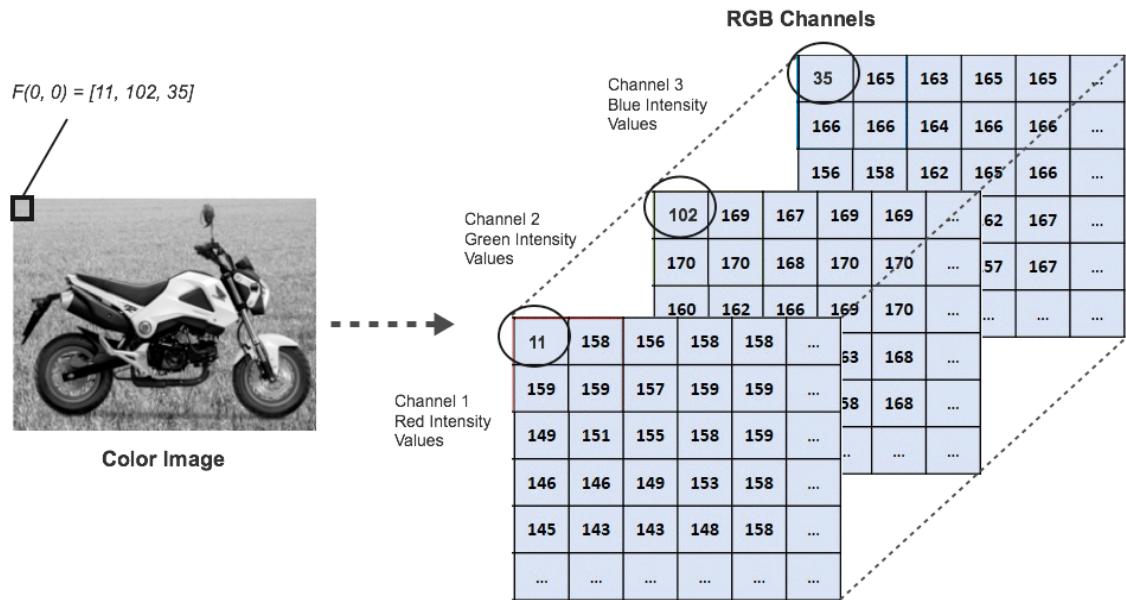


Figure 1.36

- **Preprocessing:** image processing techniques vary for each problem and dataset. Some of these techniques are: convert images to grayscale to reduce complexity, resizing images to a uniform size to fit your neural network, data augmentation, etc. As a ML engineer, you will spend a good amount of your time in cleaning up and preparing the data before you build your learning model. We will cover different preprocessing techniques throughout the projects in this book.
- **Feature extraction:** features are unique properties in the image that are used to classify its objects. Traditional machine learning algorithms use several feature extraction methods. On the other hand, neural networks does not require an extraction algorithm. It does most of the heavy lifting in processing an image and extracting useful features.

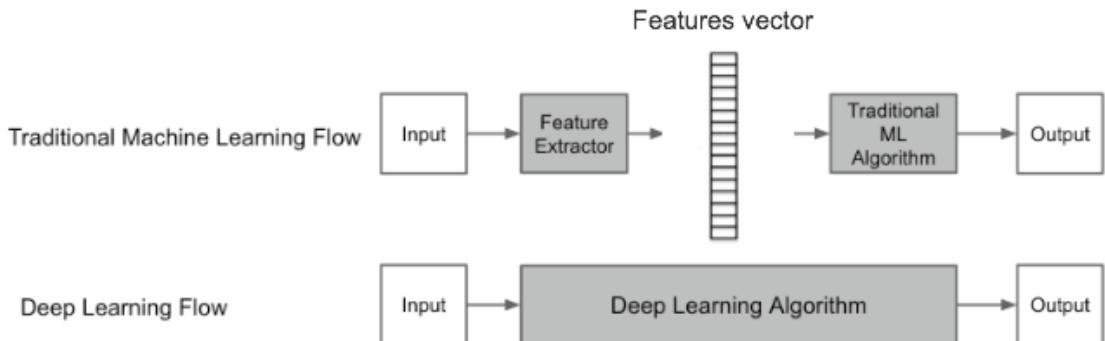


Figure 1.37

- **Classifier algorithm:** While traditional ML algorithms might get some decent results for some problems, convolutional neural networks (CNNs) truly shine in processing and classifying images in the most complex problems. Neural networks **automatically** extract useful features from your dataset + act as a classifier to output class labels for the images. Input images pass through the layers of the neural network to learn their features layer-by-layer. The deeper your network is (more layers), the more it will learn its features. Hence, the name deep learning. More layers come with some tradeoffs that we will discuss in the next two chapters. The last layer of the neural network usually acts as the classifier that outputs the class label.

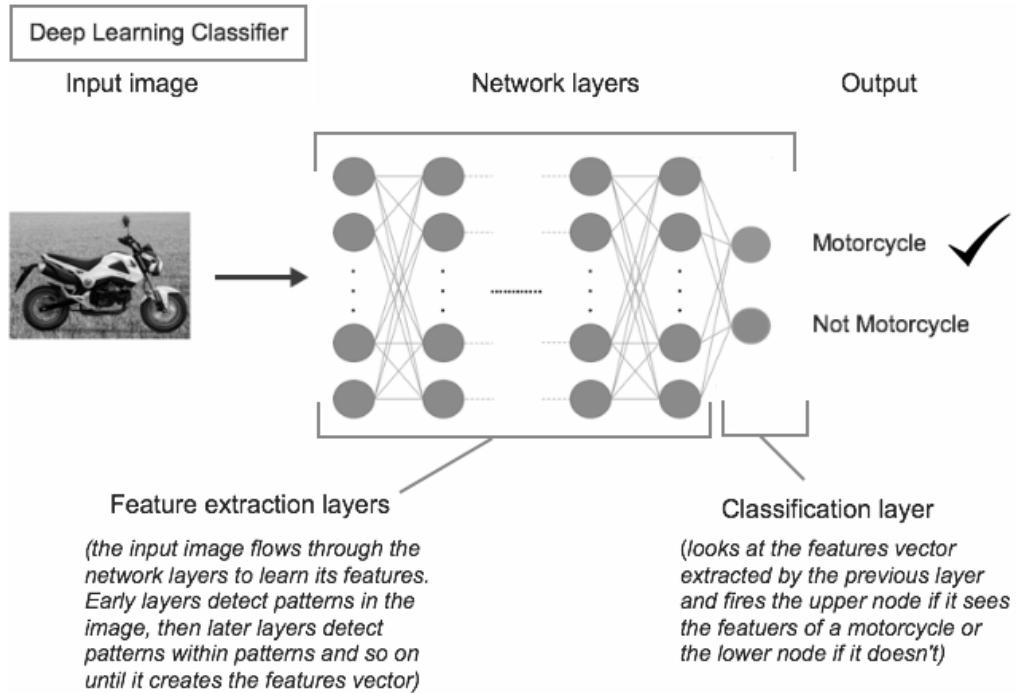


Figure 1.38

If you feel comfortable with the above takeaways, then you are 100% ready to move on the next chapter and learn all about Artificial Neural Networks (ANNs) and build more complex classifiers!

2

Deep learning and neural networks

"If the human brain were so simple that we could understand it, we would be so simple that we couldn't."

-- George Edgin Pugh

In the last chapter we discussed the computer vision pipeline components: 1) input image, 2) preprocessing, 3) extracting features, and 4) learning algorithm (classifier). We also discussed that in traditional ML algorithms, we **manually extract features** that produces a vector of features to be classified by the learning algorithm. Whereas, in deep learning, neural networks acts as the feature extractor + classifier. It **automatically** recognizes patterns and extracts features from the image and classifies them into labels.

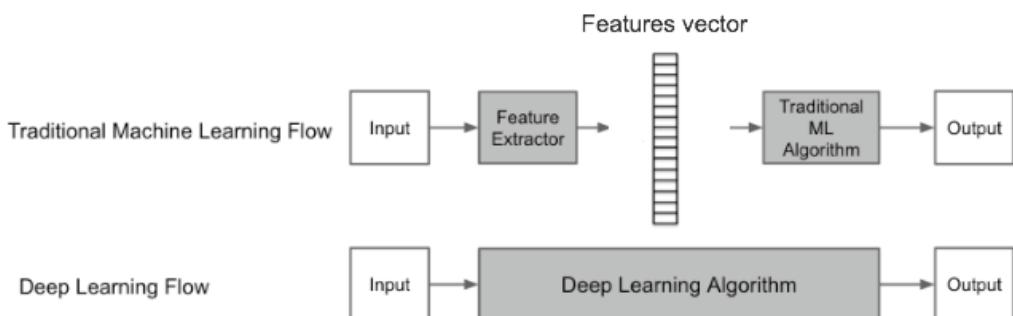


Figure 2.1

In this chapter, we will take a short pause from the computer vision context to open the “deep learning algorithm” box from the figure above. We will dive deeper into how neural networks learn features and make predictions. Then, in the next chapter, we will come back to computer vision applications with one of the most popular deep learning architectures, Convolutional Neural Networks (CNNs).

The high-level layout of this chapter will be as follows:

- We begin this chapter with the most basic component of the neural network, the **Perceptron**. Which is a neural network that contains only one neuron.
- Then, we will move on to a more complex neural network architecture that contain hundreds of neurons to solve more complex problems. This network is called **Multi-Layer Perceptrons (MLPs)**. Where neurons are stacked in layers called hidden layers. Here, you will learn the main components of the neural network architecture: 1) input layer, 2) hidden layers, 3) weight connections, and 4) output layer.
- You will learn that the network training process consists of 3 mains steps:
 1. **Feedforward** operation
 2. Calculate the error
 3. Error optimization: use **backpropagation** and **gradient descent** to select the most optimum parameters that minimize the error function
- We will dive deep into each of these steps. You will see that building a neural network requires making necessary design decisions: choosing an optimizer, the cost function, activation functions, as well as designing the architecture of the network including how many layers should be connected to each other and how many neuron should be in each layer.

Ready? Let's get started!

2.1 The Perceptron intuition

Let's take a look at the artificial neural networks (ANNs) diagram from chapter 1. You can see that ANNs consist of many neurons that are structured in layers to perform “some kind of calculations” and predict an output. This architecture can be also called Multi-Layer Perceptrons (MLPs) which is kind of more intuitive because it self-explains that the network consists of perceptrons structured in multiple layers. Both MLPs and ANNs notations are used interchangeably to describe this neural network architecture.

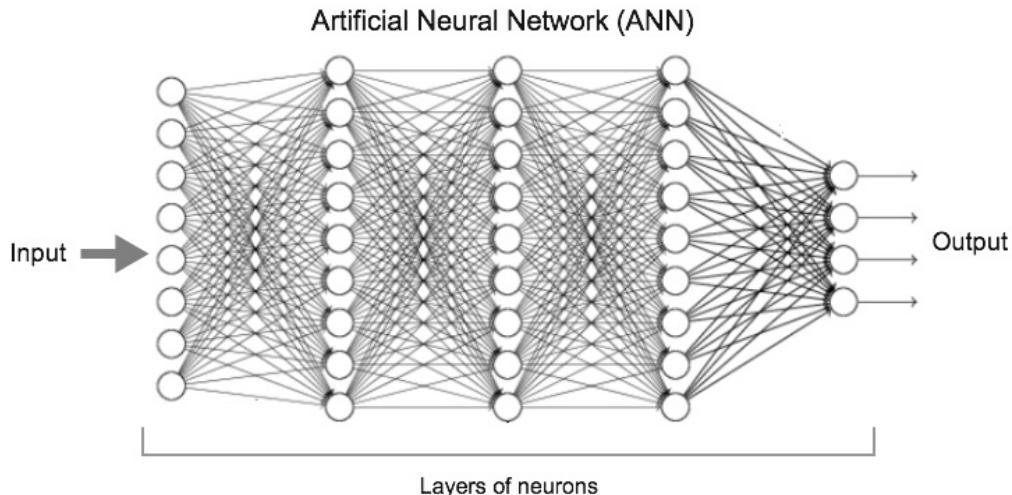


Figure 2.2

In the MLP diagram above, each node is called a neuron. We will explain how MLP networks work soon, but first let's zoom in to its most basic component, the perceptron. Once we understand how a single perceptron works, it will become more intuitive to understand how multiple perceptrons work together to learn the data features.

2.1.1 What is a perceptron?

The most simple neural network is the “perceptron”, which consists of a single neuron. Conceptually, the perceptron functions in a similar manner as the biological neurons. In biological neurons, the neuron receives electrical signals from its *Dendrites*, modulate the electrical signals in various amounts, then fires an output signal through its *Synapses* only when the total strength of the input signals exceed a certain threshold. The output is then fed to another neuron and so forth.

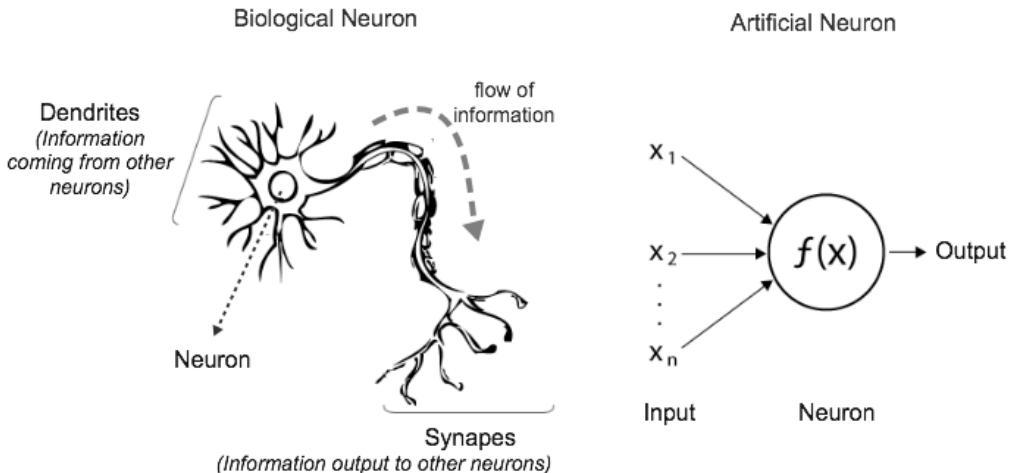


Figure 2.3

To model the biological neuron phenomenon, the artificial neuron performs two consecutive functions: 1) it calculates the *weighted sum* of the inputs to represent the total strength of the input signals and 2) applies a *step function* to the result to determine whether to fire an output 1 if the signal exceeds a certain threshold or 0 if the signal doesn't exceed the threshold.

As we discussed in chapter 1, not all input features are equally useful or important. To represent that, each input node is assigned a weight value to reflect its importance. They are called *connection weights*.

Connection weights:

Not all input features are equally important (or useful features). Each input feature (x_i) is assigned its own weight (w_i) that reflects its importance in the decision making process. In relatively easier terms, some inputs are made more important than others by giving them more weight so that they have a greater effect on the output. If the weight is high it amplifies the input signal and if the weight is low it diminishes the input. In common representations of neural networks, the weights are represented by lines or edges from the input node to the perceptron.

For example, if you are predicting a house price based on a set of features like size, neighborhood, and number_of_rooms, these are three inputs (x_1, x_2 , and x_3). Each of these inputs will have a different weight value that represents its effect on the final decision. So, if the size of the house has double the effect on the price than the neighborhood and the neighborhood is double the number of rooms, you will see the weights something like this 8, 4, and 2 respectively.

Now, how are these values assigned? That's what we are going to discuss very soon.

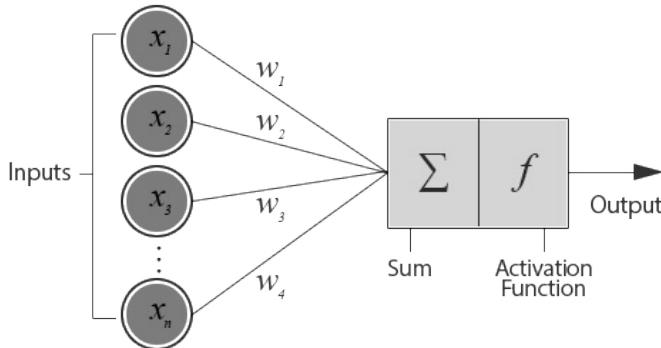


Figure 2.4

In the perceptron diagram above, you can see the following:

1. Input vector: The the features vector that is fed to the neuron. It is usually denoted with an uppercase X to represent a vector of inputs (x_1, x_2 , and x_3)
2. Weights vector: each x_i , is assigned a weight value w_i , that represents its importance.
3. Neuron functions: the calculations performed within the neuron to modulate the input signals - weighted sum and step activation function
4. Output: The output is controlled by the type of activation function you choose for your network. There are different activation functions that we will discuss in details in this chapter. For step function, the output is either 0 or 1. Other activation functions produce probability output or float numbers. The output node represents the perceptron prediction.

Let's take a deeper look at the calculations that happen inside the neuron: 1) weighted sum and 2) step function.

1) WEIGHTED SUM FUNCTION:

Also known as *linear combination*. It is the sum of all inputs multiplied by their weights then added to a bias term. This function produces a straight line represented in the following equation:

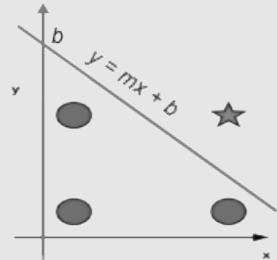
$$\begin{aligned} z &= \sum x_i \cdot w_i + b \text{ (bias)} \\ z &= x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \dots + x_n \cdot w_n + b \end{aligned}$$

Here is how we implement the weighted sum in python:

```
# X is the input vector (denoted with an uppercase X)
# w is the weights vector, b is y-intercept
z = np.dot(w.T,X) + b
```

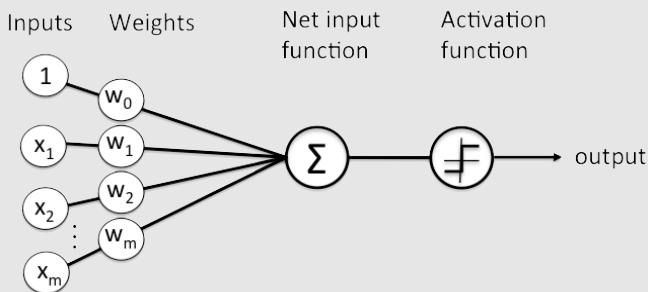
What is the bias?

Let's brush up our memory on linear algebra. The function of a straight line is represented by this equation: $y = mx + b$, where b is the y-intercept. To be able to define a linear line, you need two things: 1) the slope of the line, and 2) a point on that line. The bias is that point on the y-axis. It allows you to move the line up and down on the y-axis to better fit the prediction with the data. Without the bias (b), the line always has to go through the origin point $(0,0)$ and you will get a poorer fit. To visualize the importance of the bias, look at the graph in figure 2.x and try to separate the circles from the star using a line that passes through the origin $(0,0)$. It is not possible.



Let's brush up our memory on linear algebra. The function of a straight line is represented by this equation: $y = mx + b$, where b is the y-intercept. To be able to define a linear line, you need two things: 1) the slope of the line, and 2) a point on that line. The bias is that point on the y-axis. It allows you to move the line up and down on the y-axis to better fit the prediction with the data. Without the bias (b), the line always has to go through the origin point $(0,0)$ and you will get a poorer fit. To visualize the importance of the bias, look at the graph in figure 2.x and try to separate the circles from the star using a line that passes through the origin $(0,0)$. It is not possible.

The input layer can be given biases by introducing an extra input which always has a value of 1 as you can see in the figure below. In neural networks, the value of the bias (b) is treated as an extra weight and is learned and adjusted by the neuron to minimize the cost function.



2) STEP ACTIVATION FUNCTION:

In both artificial and biological neural networks, a neuron does not just output the bare input it receives. Instead, there is one more step, called an activation function, they are the decision making units of the brain. The activation function takes the same weighted sum input from before, $z = \sum x_i \cdot w_i + b$, and activates (fires) the neuron if the weighted sum is higher than a certain threshold. Later in this chapter we'll review the different types of activation functions and their general purpose in the broader context of neural networks. The simplest activation function used by the perceptron algorithm is the "step function" that produces a binary output

(0 or 1). It basically says that, if the summed *input* ≥ 0 , then it "fires" (*output* = 1). Else (*summed input* < 0) it doesn't fire (*output* = 0).

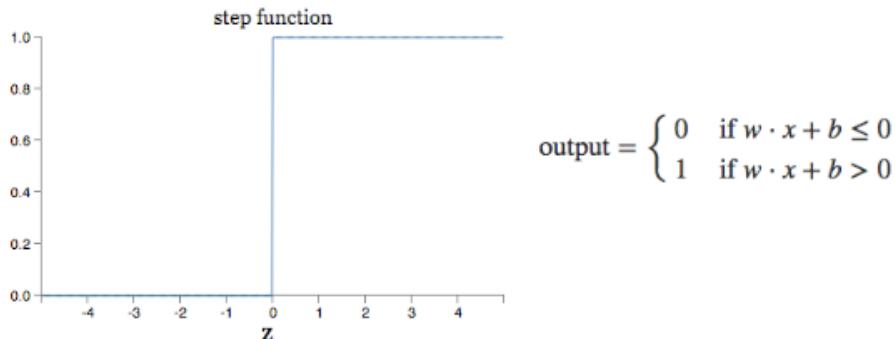


Figure 2.6

$\hat{y} = g(z)$, where g is activation function and z is the weighted sum $= \sum x_i \cdot w_i + b$

This is how the step function looks in python:

```
# z is the weighted sum = sum =  $\sum x_i \cdot w_i + b$ 
def step_function(z):
    if z <= 0:
        return 0
    else:
        return 1
```

2.1.2 How does the perceptron learn?

The neuron uses **trial and error** to learn from its mistakes. It uses the weights as knobs by tuning their values up and down until the network is trained. The perceptron's learning logic goes like this:

1. The neuron calculates the weighted sum and apply the activation function to make a prediction \hat{y} . This is called **feedforward process**.

$$\hat{y} = \text{activation} (\sum x_i \cdot w_i + b)$$

2. It then compares the prediction with the correct label to **calculate the error**

$$\text{error} = y - \hat{y}$$

3. **Update the weight:** if the prediction is too high, it will adjust the weights to make a lower prediction next time and vice versa.

4. Repeat!

This process is repeated many times, and the neuron continues to **update the weights** to improve its predictions until step 2 produces a very small error close to zero. Which means that the neuron's prediction is very close to the correct value. At this point, we can stop the training and save the weight values that yielded the best results to apply to future cases where the outcome is unknown and make real predictions.

2.1.3 Is one neuron enough to solve complex problems?

Well, the short answer is No. But let's see why. The perceptron is a linear function. This means that the trained neuron will produce a straight line that separates our data:

Suppose we want to train a perceptron to predict whether a player will be accepted into the college squad. We collect all the data from the previous years and train the perceptron to predict whether players will be accepted or not based on only two features (height and weight). The trained perceptron will find the best weights and bias values to produce the *straight line* that best separates the accepted from non-accepted (best fit). This line has this equation: $z = \text{height} \cdot w_1 + \text{age} \cdot w_2 + b$. After the training is complete on the training data, we can start using this perceptron to predict with new players. When we get a player who is 150 cm height and 12 years old, we compute the above equation with the values (150, 12). When plotted in the graph, you can see that it falls below the line, then the neuron is predicting that this player will not be accepted. If it falls above the line, then the player will be accepted.

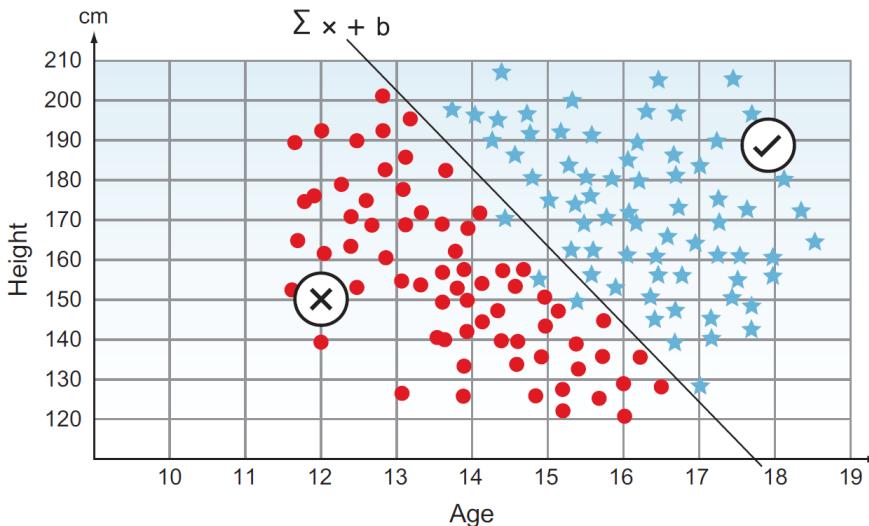


Figure 2.7

In the above example, the single perceptron works fine because our data was **linearly separable**. Which means that the training data can be separated by a straight line. But life

isn't always that simple. What happens when we have a more complex dataset that cannot be separated by a straight line (*non-linear dataset*)?

As you can see in the figure below, a single straight line will not separate our training data. We say that: "it does not fit our data". We need a more complex network for more complex data like this. What if we built a network with two perceptrons? This will produce two lines. Would that help us separate the data better?

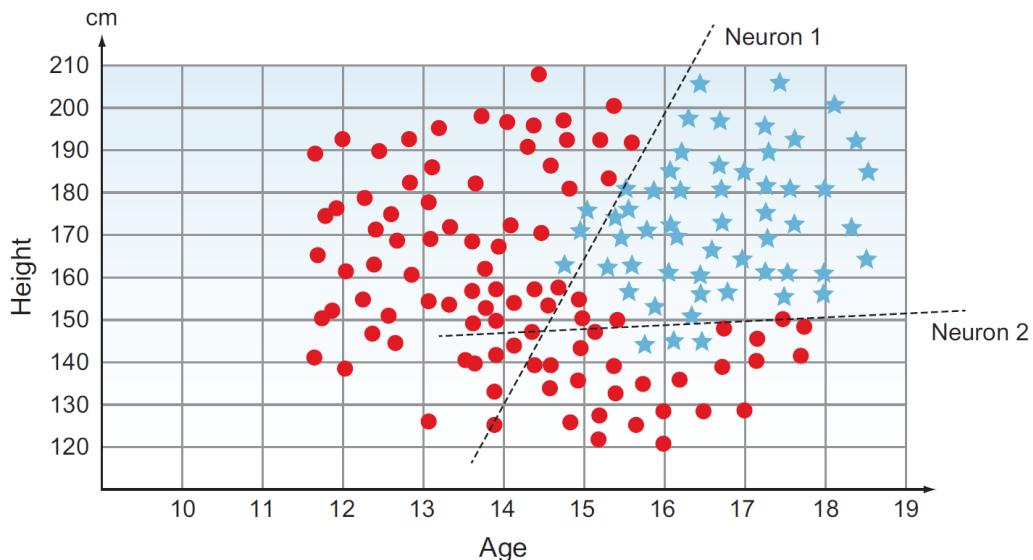


Figure 2.8

Okay, this is definitely better than the straight line. But, I still see some blue and red mispredicted. Can we add more neurons to make the function fit better? Now, you are getting it. Conceptually, the more neurons we add, the better the network will fit our training data. In fact, if we add too many neurons, this will make the network overfit the training data (not good). But we will talk about this later. The general rule here is that the more complex our network is, the better it learns the features of our data.

2.2 Multi-Layer Perceptron (MLP)

We saw that a single perceptron works great with simple datasets that can be separated by a linear line. But, as you can imagine, the real world is much more complex than that. This is where neural networks can show their full potential.

Linear vs nonlinear problems:

- A. Linear datasets: The data can be split with one straight line.
- B. Nonlinear datasets: The data cannot be split with a single straight line. We need more than one line to form a shape that splits the data.

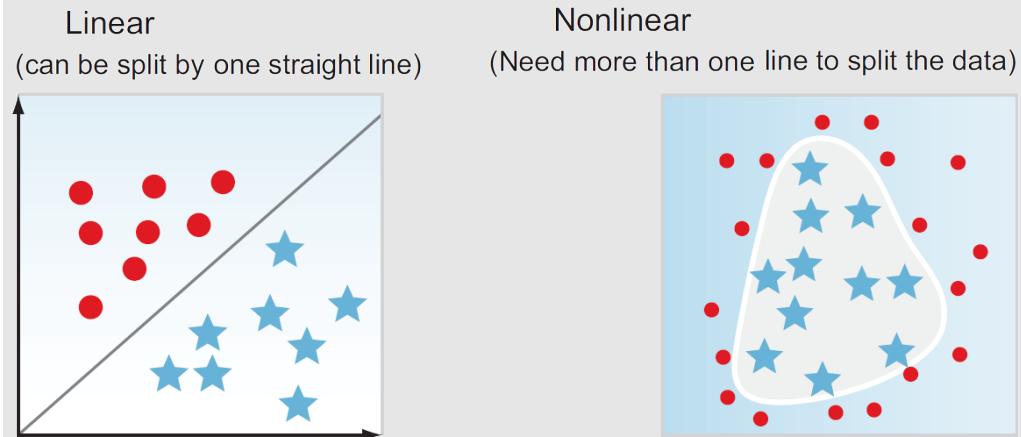


Figure 2.9

Look at the 2D data above. In the linear problem, the stars and dots shapes can be easily classified by drawing a single straight line. In nonlinear data, a single line will not separate both shapes.

To split a nonlinear dataset, we will need more than one line. This means that we will need to come up with an architecture to use tens and hundreds of neurons in our neural network. Let's look at the example below: We learned that the perceptron is a linear function that produces a straight line. Then in order to fit the data below, we try to create a triangle-ish shape that splits the dark dots. It looks like three lines would do the job.

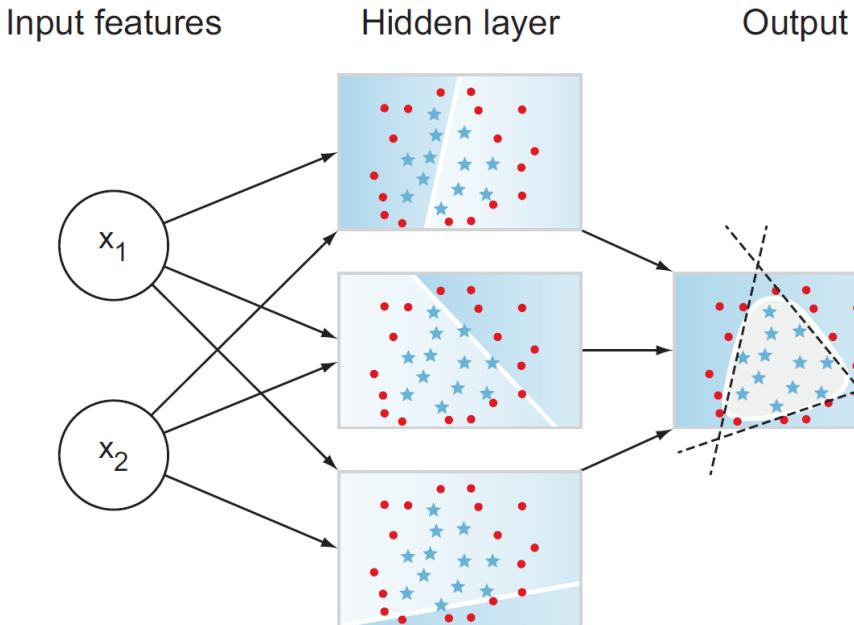


Figure 2.10

The above diagram is an example of a small neural network that is used to model nonlinear data. In this network, we used three neurons stacked together in one layer called **hidden layer**. They were given this name because we don't see the output of these layers during the training process.

2.2.1 Multi-Layer Perceptron Architecture

We've seen how a neural network can be designed to have more than one neuron. Let's expand on this idea with a more complex dataset. The below diagram is from the Tensorflow Playground website. We try to model a spiral data to classify between two classes. In order to fit this dataset, we need to build a neural network that contains tens of neurons. A very common neural network architecture is to stack the neurons in layers on top of each other called hidden layers. Each layer has n number of neurons. Layers are connected to each other by weights connections. This leads to the Multi-Layer Perceptron (MLP) architecture below.

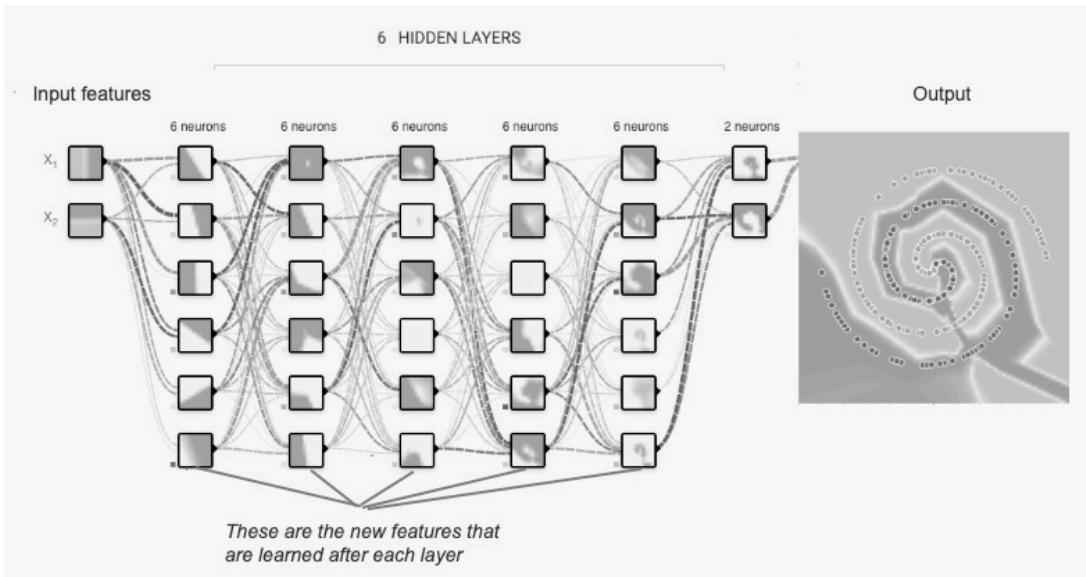


Figure 2.11

The main components of the neural network architecture are:

1. **Input layer:** contains the features vector
2. **Hidden layers:** the neurons are stacked on top of each other in layers called Hidden layers. The reason they are called “hidden” layers is because we don’t see or control the input going into these layers nor the output. All we do is feed the features vector to the input layer and see the output coming out of the last layer (output).
3. **Weight connections (edges):** weights are assigned to each connection between the nodes to reflect the importance of this node (feature) on the final output prediction. In graph network terms, these are called edges connecting the nodes.
4. **Output layer:** We get the answer or prediction from our model from the output layer. Depending on the setup of the neural network, the final output may be a real-valued output (regression problem) or a set of probabilities (classification problem). This is controlled by the type of activation function we use in the neurons in the output layer. We’ll discuss the different types of activation functions in the next section.

We already discussed the input layer, weights, and the output layer. The new thing in this architecture is the hidden layers.

2.2.2 What are the Hidden Layers?

This is where the core of the feature learning process takes place. When you look at the hidden layers nodes in the diagram above, you will see that the early layers detect simple patterns in the training to learn low level features (straight lines). Later layers detect patterns within patterns to learn more complex features and shapes. Then patterns within pattern

within patterns and so on. This concept will come in handy when we discuss Convolutional Networks (CNNs) in later chapters. For now, know that, in neural networks, we stack hidden layers to learn complex features from each other until we fit our data. So when designing your neural network, if your network is not fitting the data, the answer could be adding more hidden layers.

2.2.3 How many layers and how many nodes in each layer?



Hyperparameter Alert!

As a machine learning engineer, most of your work will be designing your network and tuning its hyperparameters. While there is no one prescribed recipe that fits all models, we will try throughout this book to build an intuition on the different hyperparameters and recommend some starting points. Setting the number of layers and the numbers of neurons in each layer is one of the important hyperparameters that you will be designing when working in neural networks.

The network can have one or more hidden layers (technically, as many as you want). Each layer has one or more neurons (also technically as many as you want). Your main job, as a machine learning engineer, is to design these layers. Usually when we have 2 or more hidden layers we call this a deep neural network. The general rule is: the deeper your network is, the more it will fit the training data. Which is not always a good thing, because the network can fit the training data too much that it fails to generalize when you show it new data (overfitting) also it gets more computationally expensive. So your job is to build a network that is not too simple (one neuron) and not too complex for your data. It is recommended that you read about different neural network architectures that are successfully implemented by others to build an intuition of what is too low for your problem. Then start from a point, maybe 2 or 3 layers and observe the network performance. If it is performing poorly (underfitting), add more layers. If you see signs of overfitting (will be discussed later), then decrease the number of layers. More on that later. To build an intuition on how neural networks perform when you add more layers, I advice that you play around with [Tensorflow Playground](#).

Fully connected layers:

It is important to call out that the layers in classical MLP network architectures are fully connected to the next hidden layer. In the diagram above, notice that each node in a layer is connected to all nodes in the previous layer. This is called a *fully connected network*. These edges are the weights that represent the importance of this node to the output value.

In later chapters we will discuss other variations of the neural networks architecture (like Convolutional and Recurrent networks). For now, know that this is the most basic neural network architecture and it can be referred to as any of these names: ANN, MLP, fully connected network, or feedforward network.

Let's do this fun exercise to find out how many edges we have in our example. Suppose that we designed an MLP network with 2 hidden layers, each has 5 neurons:

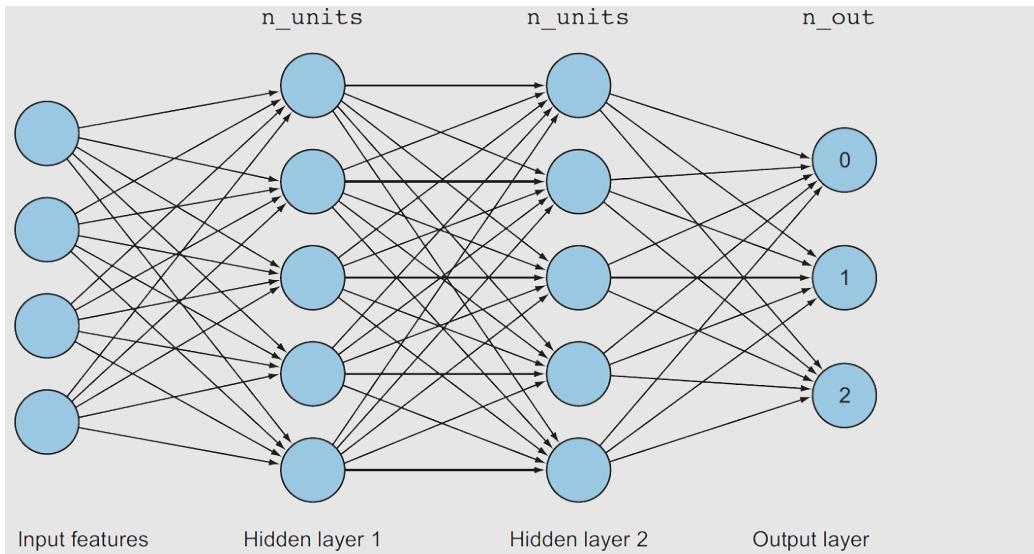


Figure 2.12

- **Weights_0_1:** 4 nodes in the input layer * 5 nodes in layer 1 + 1 for the bias = 21 edges
- **Weights_1_2:** 5 nodes in layer 1 * 5 nodes in layer 2 + 1 for the bias = 26 edges
- **Weights_2_output:** 5 nodes in layer 2 * 3 nodes in the output layer + 1 bias = 16 edges
- Total weights in this network = 63

Then, we have a total of 63 weights in this very simple network. The values of these weights are randomly initialized then the network performs feedforward and backpropagation to learn the best values of weights that most fit our model to the training data. You will see that very soon.

2.2.4 MLP Takeaways

This could feel like a soup of ideas. Let's recap what we discussed so far:

- First we talked about the analogy between the biological and artificial neurons: both have inputs + a neuron that does some calculations to modulate the input signals + output.
- Then we zoomed in to the artificial neurons' calculations to explore its two main functions: 1) weighted sum and 2) activation function.
- **Weights:** We learned that the network assigns random weights to all the edges. These weight parameters reflect the usefulness (or importance) of these features on the output prediction.
- Lastly, we saw that perceptrons contain a single neuron. They are linear functions that produce a straight line to split linear data. In order to split more complex data (nonlinear), we need to apply more than one neuron in our network to form a multilayer perceptron MLP.

- The MLP architecture contains: 1) input features, 2) connection weights, 3) hidden layers, and 4) output layer.

We discussed the high level process of how the perceptron learns. The learning process is a repetition of three main steps: 1) feedforward calculations to produce a prediction (weighted sum and activation), 2) calculate the error, and 3) backpropagate the error and update the weights to minimize the error.

Next, we will dive deeper into each one of these steps:

1. What is an activation function and what are the different types of activations?
2. Explain the feedforward process
3. Calculating the error and the different types of error functions
4. Optimization algorithms and Backpropagation

2.3 Activation functions

When you are building your neural network, one of the design decisions that you will need to make is what activation function to use for our neurons calculations. Activation function is also referred to as *transfer function* or *nonlinearities* because they transform the linear combination of the weighted sum into non-linear models. It is placed at the end of each perceptron to decide whether to activate this neuron or not.

Why use activation functions at all? Why not just calculate the weighted sum of our network and propagate that through the hidden layers to produce an output?

The purpose of the activation function is to introduce non-linearity into the network. Without it, a multi-layer perceptron will perform similar to a single perceptron no matter how many layers we add. Activation functions are needed to restrict the output value to a certain finite value. Let's revisit the example of predicting whether a player gets accepted or not:

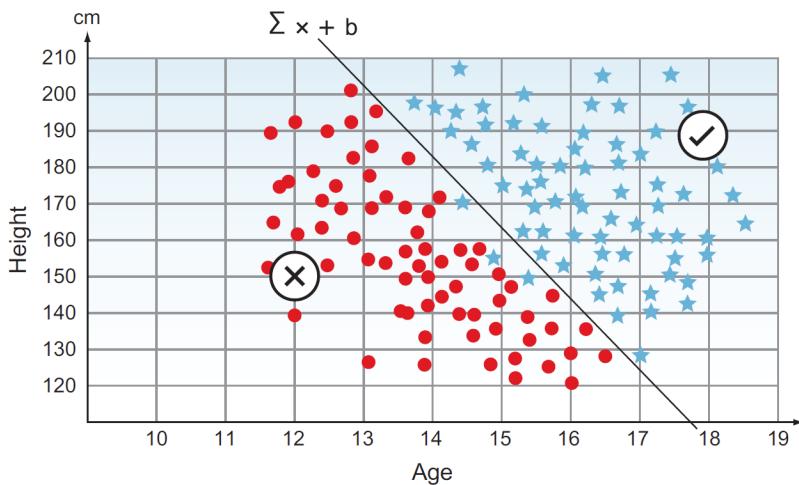


Figure 2.13

First, the model will calculate the weighted sum and produce the linear function: $z = \text{height} \cdot w_1 + \text{age} \cdot w_2 + b$. . The output of this function has no bound. (z) could literally be any number. We use an activation function to wrap the prediction values to a finite value. In this example, we used the step function. Where if $z > 0$ then above the line (accepted) and if $z < 0$ then below the line (rejected). So, without the activation function, we just have a linear function that produces a number but no decision is made in this perceptron. Because the activation function is the decider whether to fire this perceptron or not.

There are an infinite number of activation functions. In fact, the last few years have seen a lot of progress in "state-of-the-art" activations. However, there are still a relatively small list of

activations that account for the vast majority of activation needs. Let's dive deeper into some of the most common types of activation functions:

$$\text{activation}(z) = z = Wx + b$$

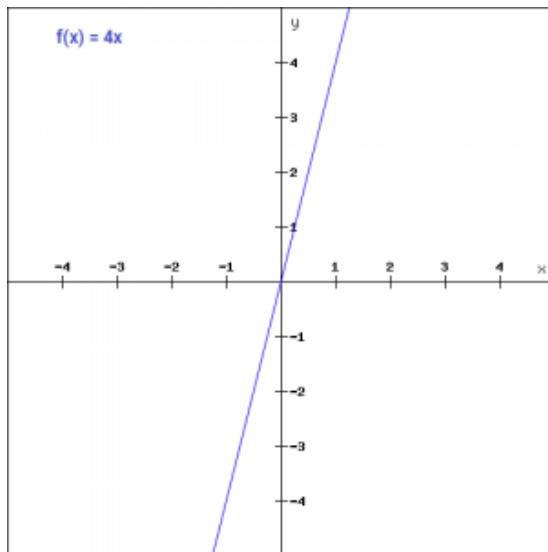
2.3.1 Linear Transfer Function

Also called *identity function*. Which means that the function passes the signal through unchanged. In practical terms, the output will be equal to the input which means that we don't actually have an activation function so no matter how many layers our neural network has, all it is doing is just computing a linear activation function or at most scaling the weighted average coming in but it doesn't transform it into a nonlinear function.

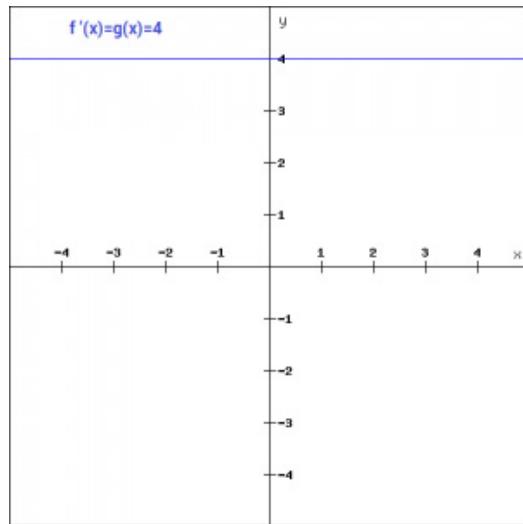
$$\text{activation}(z) = z = wx + b$$

The composition of two linear functions is a linear function so unless you throw a non-linear activation function in your neural network then you are not computing any interesting functions no matter how deep you make your network. No learning here!

To understand why, Let's calculate the derivative of the activation $z(x) = w.x + b$. Where w here = 4 and b = 0. When we plot this function it looks like this:



Then the derivative of $z(x) = 4x$ will be $z'(x) = 4$.



The derivative of a linear function is constant i.e. it does not depend upon the input value x . This means that every time we do a back propagation, the gradient would be the same. And this is a big problem, we are not really improving the error since the gradient is pretty much the same. This will be clearer when we explain backpropagation at the end of this chapter.

2.3.2 Heaviside Step Function (Binary classifier)

The *Step Function* produces a binary output. It basically says that, if the input $x \geq 0$, then it "fires" (output $y = 1$). Else (input < 0) it doesn't fire (output $y = 0$). It is mainly used in binary classification problems like true or false, spam or not spam, pass or fail.

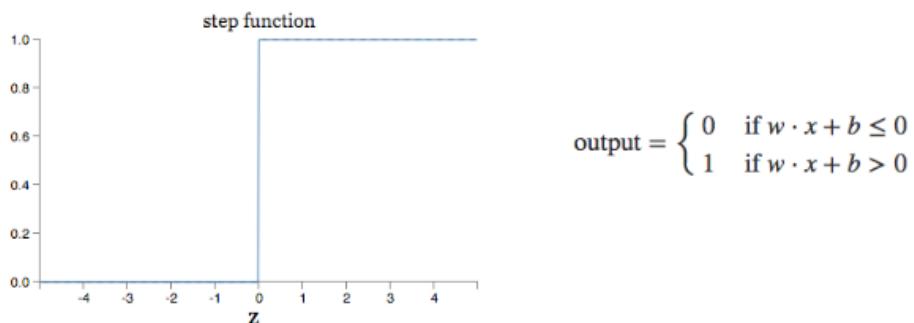


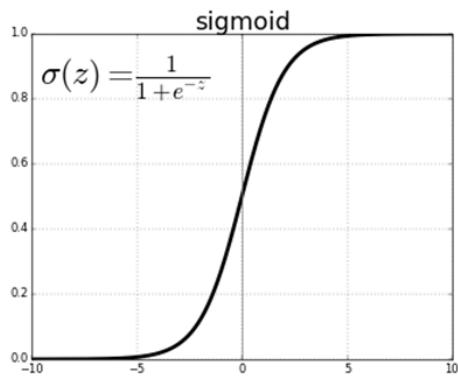
Figure 2.14

2.3.3 Sigmoid/Logistic function

This is one of the most common activation functions. It is commonly used in binary classifiers to predict the *probability* of a class when you have 2 classes. The sigmoid squishes all the values to a probability between 0 and 1 which reduces extreme values or outliers in the data without removing them. Sigmoid or logistic functions converts infinite continuous variables (range between $-\infty$ to $+\infty$) into simple probabilities between 0 and 1. It is also called the "S" shape curve because when plotted in a graph it produces an S shaped curve. While the step function is used to produce discrete answer (pass or fail), Sigmoid is used to produce the probability of pass and probability of fail.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Figure 2.15



Here is how sigmoid is implemented in python:

```
# import numpy
import numpy as np

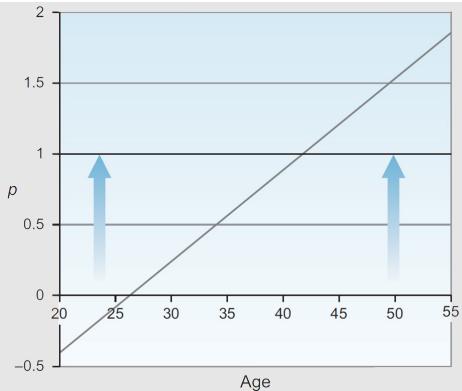
# sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Just in Time Linear Algebra (Optional)

Let's take a deeper dive into the math side of the sigmoid function to understand the problem it is trying to solve and how the sigmoid function equation is driven. Suppose that we are trying to predict the medical condition of the patient whether they have diabetes or not based on only one feature, the patient's age. When we plot the data we have about our patients, we get the linear model in the figure 2.x.

$$z = \beta_0 + \beta_1 \text{age}$$

In this plot you can observe the balance of probabilities that should go from 0 to 1. Note that when patients are below age of 25, the predicted probabilities are negative meanwhile it is higher than 1 (100%) when they are older than 43 years old. This is a clear example of why linear functions do not work in most cases. Now, how do we fix this to give us probabilities within the range of 0 < probability < 1?



First, we need to do something to eliminate all the negative probability values. The exponential function is a great solution for this problem because the exponent of anything (and I mean anything) is always going to be positive. So let's apply that to our linear equation to calculate the probability (p):

$$p = \exp(z) = \exp(\beta_0 + \beta_1 \text{age}) \dots \text{equation 2.3.1}$$

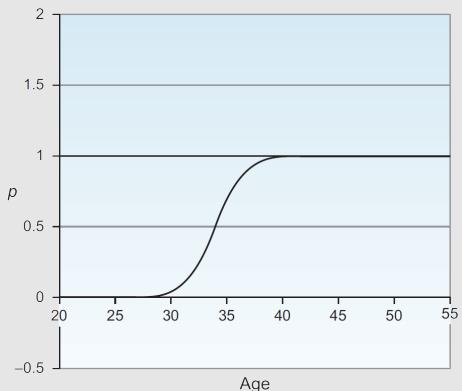
This equation ensures that we always get probabilities greater than 0. Now, what about the values that are higher than 1?

Second, we need to do something about the values that are higher than 1. If you think about proportions, any given number divided by a number that is greater than it, will give us a number smaller than 1. Let's do exactly that to equation 2.3.1 above. We divide equation 2.3.1 by its value + a small value 1 or a in some cases a very small value let's call it epsilon (ϵ):

$$p = \exp(z) / (\exp(z) + 1)$$

If you divide equation 2.3.1 by $\exp(z)$ you get:

$$p = 1 / (1 + \exp(-z))$$



When we plot the probability of this equation we get the "S" shape of the sigmoid function where probability is no longer below 0 or above 1. In fact, as patients' ages grow, the probability asymptotically get closer to 1 and as the weights move down, the function asymptotically get closer to 0 but never outside the $0 < p < 1$ range. This is the plot of the sigmoid function and logistic regression.

2.3.4 Softmax Function

Softmax function is a generalization of the sigmoid function. It is used to obtain classification probabilities when we have more than 2 classes. It is used to force the outputs of Neural Network to sum to 1 (i.e. $0 < \text{output} < 1$). The most common use case in deep learning (especially in computer vision) is to predict a single class out of many options (more than 2). For example, if you want to build a digit classifier to predict which number is in the image, we are trying to find the probability of a number in the image out of 10 classes (numbers from 0 to 9 since there can be only one digit in the image).

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

$\begin{bmatrix} 1.2 \\ 0.9 \\ 0.4 \end{bmatrix}$	\rightarrow	$\boxed{\text{Softmax}}$	\rightarrow	$\begin{bmatrix} 0.46 \\ 0.34 \\ 0.20 \end{bmatrix}$
---	---------------	--------------------------	---------------	--

TIP: Softmax function is the go to function that you will often use at the output layer of a classifier when you are working on a problem that you need to predict a class between more than 2 classes. Softmax can work fine if you are classifying two classes as well. It will basically work as a sigmoid function. By the end of this section, I'll tell you my recommendations on when to use each activation function.

2.3.5 Hyperbolic Tangent Function (tanh)

It is a shifted version of the sigmoid version. Instead of squeezing the signals values between 0 and 1, tanh squishes all values to the range -1 to 1. Tanh almost always works better than the sigmoid function in the hidden layers because it has the effect of centring your data so that the mean of the data is close to zero rather than 0.5 which makes learning for the next layer a little bit easier.

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

One of the downsides for both sigmoid and tanh functions is that, if (z) is very large or very small, then the gradient (or derivative or slope) of this function becomes very small (close to zero) which will slow down gradient descent. This is when ReLU activation function (explained next) comes in to solve this problem.

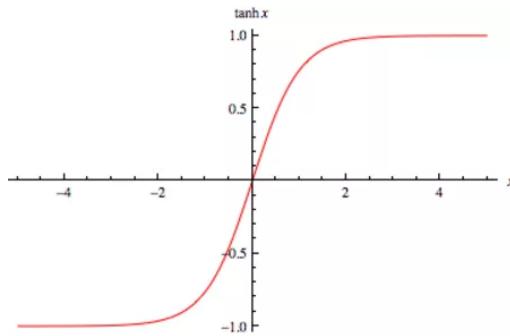


Figure 2.16

2.3.6 Rectified Linear Unit (ReLU)

ReLU activation function activates a node only if the input is above zero. If the input is below zero, the output is always zero. But when the input is higher than zero, it has a linear relationship with the output variable $f(x) = \max(0, x)$. At the time this book is being written, ReLU is considered the current state of the art of activation functions because it works well in many different situations and it tends to train better than sigmoid and tanh in the hidden layers.

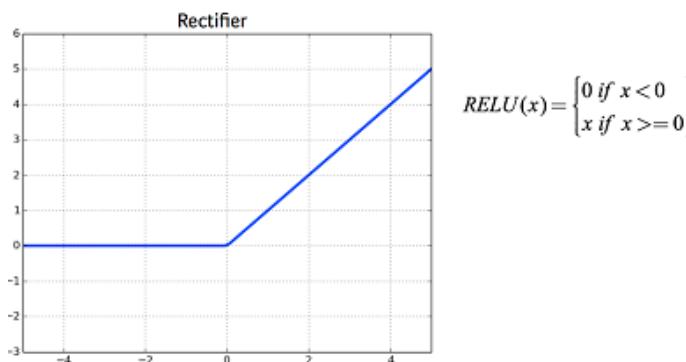


Figure 2.17

Here is how ReLU is implemented in python:

```
# relu activation function
def relu(x):
    if x < 0:
        return 0
    else:
        return x
```

2.3.7 Leaky ReLU

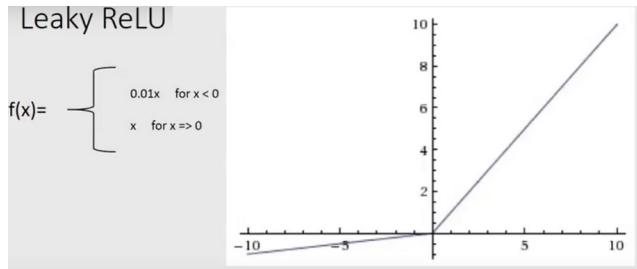
One disadvantage of ReLU activation is that the derivative is equal to zero when (x) is negative. Leaky ReLU is a ReLU variation that is trying to mitigate this issue. Instead of having the function being zero when $x < 0$, Leaky ReLU introduces a small negative slope (around 0.01) when (x) is negative. It usually works better than the ReLU function although it's just not used as much in practice. Take a look at the Leaky ReLU graph in figure 2.x, can you see the leak?

$$f(x) = \max(0.01x, x)$$

Why 0.01? Some people like to use this as another hyperparameter to tune, but that would be an overkill since you already have other bigger problems to worry about. Feel free to try different values (0.1, 0.01, 0.002) in your model and see how they work.

Here is how Leaky ReLU is implemented in python:

```
# Leaky relu activation function with
# a 0.01 Leak
def leaky_relu(x):
    if x < 0:
        return x * 0.01
    else:
        return x
```



Hyperparameter Alert!

Due to the number of activation functions, it may appear to be an overwhelming task to select the appropriate activation function for your network. While it is important to select a good activation function, I promise this is not going to be a challenging task when you design your network. There are some rules of thumb that you can start with then you can tune the model as needed. If you are not sure what to use, here are my two cents when choosing an activation function:

For hidden layers: in most cases you can use the ReLU activation function (or LeakyReLU) in the hidden layers as you will see in the projects that we are going to build throughout this book. It is increasingly becoming the default choice for activation functions because it is a bit faster to compute than other activation functions. More importantly, it reduces the likelihood of the gradient to vanish because it does not saturate for large input value as opposed to sigmoid and tanh activation functions who saturate at ~ 1. Remember, gradient is the slope. When the function plateaus, this will lead to no slope, hence the gradient starts to vanish. Which makes it harder to descent to the minimum error (we will talk more about this phenomenon that is called vanishing/exploding gradients in later chapters).

For the output layer: the softmax activation function is generally a good choice for most classification problems when the classes are mutually exclusive. The sigmoid function serves the same purpose when you are doing binary classification. For regression problems, you can simply use no activation function at all since the weighted sum node

produces the continuous output that you need. For example, if you want to predict the house pricing based on the prices of other houses in the same neighbourhood.

Here is a cheat sheet of the most common activation functions:

Table 2.1

Activation function	Description	Plot	Equation
Linear Transfer Function (identity function)	The signal passes through it is unchanged. It remains linear function. Almost never used.		$f(x) = x$
Heaviside Step Function (Binary classifier)	Produces a binary output 0 or 1. Mainly used in binary classification to give a discrete value.		$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$
Sigmoid/Logistic function	It squishes all the values to a probability between 0 and 1 which reduces extreme values or outliers in the data. Usually used to classify 2 classes.		$\sigma(z) = \frac{1}{1 + e^{-z}}$

Hyperbolic Tangent Function (tanh)	<p>It squishes all values to the range -1 to 1. Tanh almost always works better than the sigmoid function in the hidden layers.</p>		$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Rectified Linear Unit (ReLU)	<p>It activates a node only if the input is above zero. Always recommended to use Relu for hidden layers. Better than tanh.</p>		$f(x) = \max(0, x)$
Leaky ReLU	<p>Instead of having the function being zero when $x < 0$, Leaky ReLU introduces a small negative slope (around 0.01) when (x) is negative.</p>		$f(x) = \max(0.01x, x)$

2.4 Feedforward

Now that we understand how to stack perceptrons in layers, connect them with the weights edges, perform weighted sum function, and apply activation functions, let's implement the complete forward pass calculations to produce a prediction output. **The process of computing the linear combination and applying activation function is called Feedforward.** We briefly saw how the feedforward is calculated several times in the previous sections, let's take a deeper look at what happens in this process.

The term feedforward is used to imply the forward direction in which the information flows from the input layer, through the hidden layers, all the way to the output layer. This process happens through the implementation of two consecutive functions: 1) the weighted sum, and 2) the activation function. In short, the forward pass is the calculations through the layers to make a prediction.

Let's take a look at this simple three-layer neural network and explore each of its components:

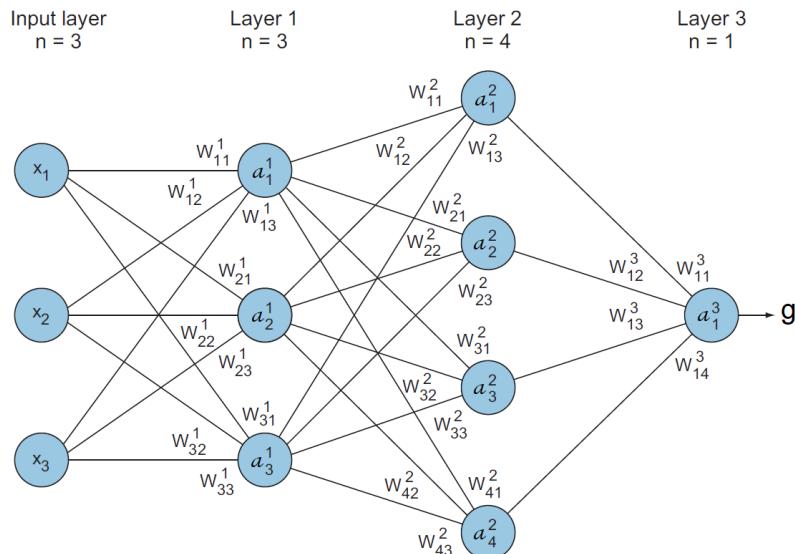


Figure 2.19

- **Layers:** this network consists of an input layer with 3 input features and three hidden layers with 3, 4, 1 neurons in each layer.
- **Weights and biases (w, b):** the edges between nodes are assigned random weights denoted as $W_{ab}^{(n)}$. Where (n) indicates the layer number, (ab) indicates the weighted edge connecting the a^{th} neuron in layer (n) to the b^{th} neuron in the previous layer $(n-1)$. For example, $W_{23}^{(2)}$ is the weight that connects the second node in layer 2 to

the third node in layer 1 (a_2^2 to a_3^1).

(Note that you can see different denotation of $W_{ab}^{(n)}$ in other deep learning literature which is fine as long as you follow one convention for your entire network.)

The **biases** are treated similar to weights because they are randomly initialized and their values are learned during the training process. So we are going to give them the weights notation (w) for convenience.

- **Activation functions** $\sigma(x)$. In this example we are using sigmoid function as an activation function $\sigma(x)$.
- **Node values (a):** We are going to calculate the weighted sum and apply the activation function and assign this value to the node a_m^n . Where the n is the layer number and m is the node index in the layer. For example, a_2^3 means node number 2 in layer 3.

2.4.1 Feedforward calculations

We have all we need to start the feedforward calculations:

$$a_1^{(1)} = \sigma(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3)$$

$$a_2^{(1)} = \sigma(w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{32}^{(1)} x_3)$$

$$a_3^{(1)} = \sigma(w_{13}^{(1)} x_1 + w_{23}^{(1)} x_2 + w_{33}^{(1)} x_3)$$

Then we do the same calculations for layer 2 ($a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$, and $a_4^{(2)}$) all the way to the output prediction in layer 3:

$$\hat{y} = a_1^{(3)} = \sigma(w_{11}^{(3)} a_1^{(2)} + w_{12}^{(3)} a_2^{(2)} + w_{13}^{(3)} a_3^{(2)} + w_{14}^{(3)} a_4^{(2)})$$

And there you have it! You just calculated the feedforward of a two-layer neural network. Let's take a moment to reflect on what we just did. Take look at how many equations we need to solve for such a small network. What happens when we have a more complex problem with 100s of nodes in the input layer + hundreds more in the hidden layers. It is more efficient to use matrices to pass through multiple inputs at once. Doing this allows for big computational speed ups, especially when using tools like numpy where we can implement this by one line of code.

Let's see how the matrices computation will look:

$$\hat{y} = \sigma \begin{bmatrix} W_{11}^3 & W_{12}^3 & W_{13}^3 & W_{14}^3 \end{bmatrix} \cdot \sigma \begin{bmatrix} W_{11}^2 & W_{12}^2 & W_{13}^2 \\ W_{21}^2 & W_{22}^2 & W_{23}^2 \\ W_{31}^2 & W_{32}^2 & W_{33}^2 \\ W_{41}^2 & W_{42}^2 & W_{43}^2 \end{bmatrix} \cdot \sigma \begin{bmatrix} W_{11}^1 & W_{12}^1 & W_{13}^1 \\ W_{21}^1 & W_{22}^1 & W_{23}^1 \\ W_{31}^1 & W_{32}^1 & W_{33}^1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Layer 3 Layer 2 Layer 1 Input vector

Figure 2.20

All we did here is simply stacked the inputs and weights in matrices and multiplied them together. The intuitive way to read this equation is from the **right to left**. Start with the far right and follow with me:

1. first we stacked all the inputs together in one vector (row, column), in this case (3, 1),
2. multiply the input vector by the weights matrix from layer 1 $W^{(1)}$ then apply the sigmoid function
3. Then multiply the result for layer 2 $\Rightarrow \sigma \circ W^{(2)}$ and layer 3 $\Rightarrow \sigma \circ W^{(3)}$
4. If you have a fourth layer, you will multiply the result from the above step by $\sigma \circ W^{(4)}$. And so on until we get the final prediction output \hat{y} !

Here is a simplified representation of this matrices formula:

$$\hat{y} = \sigma \circ W^{(3)} \circ \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)} \circ (x)$$

2.4.2 Feature learning

The nodes in the hidden layers (a_i) are the new features that are learned after each layer. For example, if you look at the diagram from the previous page, you see that we have three feature inputs (x_1 , x_2 , and x_3). After computing the forward pass in the first layer, the network learns patterns and these features are transformed to three new features with different values ($a_1^{(1)}, a_2^{(1)}, a_3^{(1)}$). Then in the next layer, the network learns patterns within the patterns and produces new features ($a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$, and $a_4^{(2)}$) and so forth. The produced features after each layer are not totally understood and we don't see them nor do we have much control over them. It is part of the neural networks magic. That's why they are given the name hidden layers. What we do is: we look at the final output prediction and keep tuning some parameters until we are satisfied by the network's performance.

To reiterate, let's see this in a small example. Below is a small neural network to estimate the price of a house based on three features: 1) how many bedrooms it has, 2) how big it is, and

3) which neighborhood it is in. You can see that the original input features values 3, 2000, and 1 were transformed into new feature values after performing the feedforward process in the first layer ($a_1^{(1)}, a_2^{(1)}, a_3^{(1)}, a_4^{(1)}$) . Then transformed again to a prediction output value (\hat{y}). Then training a neural network, we see the prediction output and compare it with the true price to calculate the error and repeat until we get the minimum error.

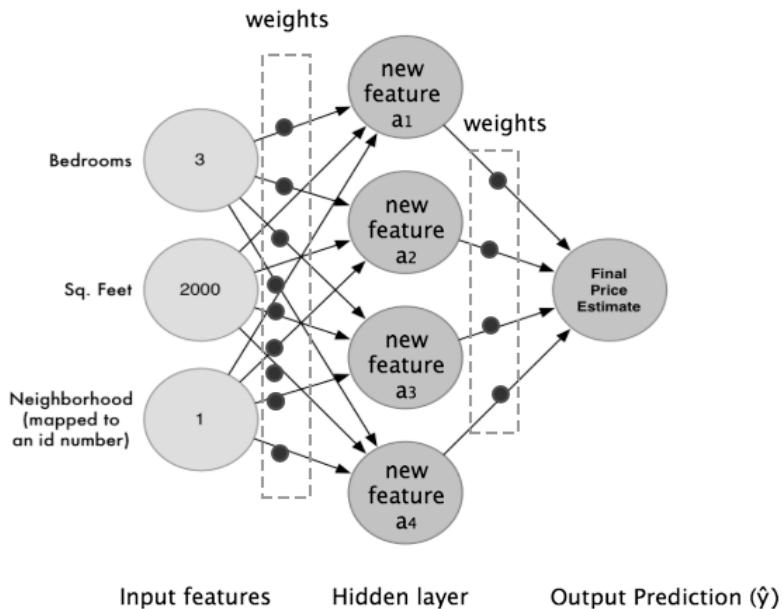


Figure 2.21

To help visualize the feature learning process, let's take another look at the image we showed earlier (below) from the Tensorflow playground. You can see that the first layer learns basic features like lines and edges. The second layer start to learn more complex features like corners. And so on until the last layers of the network learn even more complex feature shapes like circles and spiral shapes that fit the dataset.

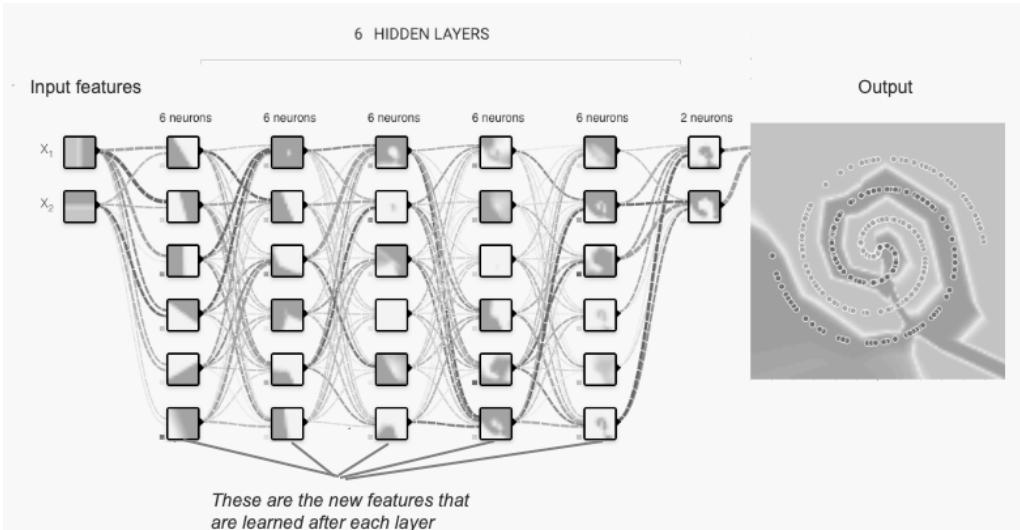


Figure 2.22

That is how the neural network learn new features via their hidden layers. First, they recognize patterns in the data. Then, they recognize patterns within patterns. Then patterns within patterns within patterns. And so on. The deeper the network is, the more it learns about the training data.

Vectors and Matrices refresher

If you understood the matrix calculations we just did in the feedforward, feel free to skip this section. If you are still not convinced. Hang tight, this section is for you.

The feedforward calculations are a set matrix multiplications. While you are not going to do these calculations by hand, because there are a lot of great deep learning libraries that do that for you with just one line of code, it is valuable to understand the mathematics that happens under the hood to be able to debug your network for errors. Especially that it is very trivial and interesting. Let's quickly review matrices calculations.

Let's start with some basic definitions - matrices dimensions:

- A scalar is a single number
- A vector is an array of numbers
- A matrix is a 2-D array
- A tensor is a n-dimensional array with $n > 2$

Scalar Vector Matrix Tensor

1	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 7 & 5 \\ 2 & 4 & 6 \end{bmatrix}$
---	--	--	---

We will follow the conventions used in many mathematical literature:

- scalars are written in lowercase and italics. For instance: n
- vectors are written in lowercase, italics and bold type. For instance: \mathbf{x}
- matrices are written in uppercase, italics and bold. For instance: \mathbf{X}
- The matrix dimensions are written as follows \Rightarrow (row x column)

Multiplication:

- Scalar multiplication: Simply multiply the scalar number by all the numbers in the matrix. Note that that scalar multiplications don't change the matrix dimensions.

$$2 \cdot \begin{bmatrix} 10 & 6 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 2 \cdot 10 & 2 \cdot 6 \\ 2 \cdot 4 & 2 \cdot 3 \end{bmatrix}$$

- Matrix multiplication: When multiplying two matrices, such as in the case of (row 1 x column 1) \times (row 2 x column 2), column 1 and row 2 must be equal to each other and the product will have the dimensions (row 1 x column 2)

$$\begin{bmatrix} 3 & 4 & 2 \end{bmatrix} \cdot \begin{bmatrix} 13 & 9 & 7 \\ 8 & 7 & 4 \\ 6 & 4 & 0 \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix}$$

Same

 1 x 3 3 x 3 1 x 3
 Product

Where $x = 3*13 + 4 * 8 + 2 * 6 = 83$ same for $y = 63, z = 37$.

Now that you know the matrices multiplication rules, pull a piece of paper and work through the dimension of matrices in the neural network example above. Here is the matrix equation again for your convenience:

$$\hat{y} = \sigma \left[\begin{array}{cccc} W_{11}^3 & W_{12}^3 & W_{13}^3 & W_{14}^3 \end{array} \right] \cdot \sigma \left[\begin{array}{ccc} W_{11}^2 & W_{12}^2 & W_{13}^2 \\ W_{21}^2 & W_{22}^2 & W_{23}^2 \\ W_{31}^2 & W_{32}^2 & W_{33}^2 \\ W_{41}^2 & W_{42}^2 & W_{43}^2 \end{array} \right] \cdot \sigma \left[\begin{array}{ccc} W_{11}^1 & W_{12}^1 & W_{13}^1 \\ W_{21}^1 & W_{22}^1 & W_{23}^1 \\ W_{31}^1 & W_{32}^1 & W_{33}^1 \end{array} \right] \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array} \right]$$

Layer 3 Layer 2 Layer 1 Input vector

Figure 2.26

Last thing I want you to know about matrices is **Transposition**:

With transposition you can convert a row vector to a column vector and vice versa. Where the shape ($m \times n$) is inverted and becomes ($n \times m$). The superscript (A^T) is used for transposed matrices:

$$A = \begin{bmatrix} 2 \\ 8 \end{bmatrix} \Rightarrow A^T = [2 \ 8]$$
$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$
$$A = \begin{bmatrix} 0 & 1 \\ 2 & 4 \\ 1 & -1 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 0 & 2 & 1 \\ 1 & 4 & -1 \end{bmatrix}$$

Figure 2.27

2.5 Error functions

Up until this point, we learned how to implement the forward pass in neural networks to produce a prediction which consists of the weighted sum + activation operations. Now, how do we evaluate this prediction that the network just produced? More importantly, how do we know how far this prediction is from the correct answer (the label)? The answer is: Measure the error. Selection of the error function is another important aspect of the design of a neural network. Error functions can also be referred to as *cost functions* or *loss functions* and they are always used interchangeably in different deep learning literature.

2.5.1 What is the error function?

It is a measure of "how wrong" the neural network prediction is with respect to the expected output (the label). It quantifies how far we are from the correct solution. For example, if we have a high loss function then our model is not doing a good job. The smaller the loss, the better a job the model is doing. The larger the loss, the more our model needs to be trained to increase its accuracy.

2.5.2 Why do we need an error function?

Calculating an error turns this problem into an optimization problem. Which is something all machine learning engineers love (mathematicians too). Optimization problems focus on defining an error function and trying to optimize its parameters to get the minimum error. More on optimization in the next section. But for now, it is good to know that, in general, when we are working on a problem, if we are able to find the error function, we have a very good shot at solving it by optimizing the error.

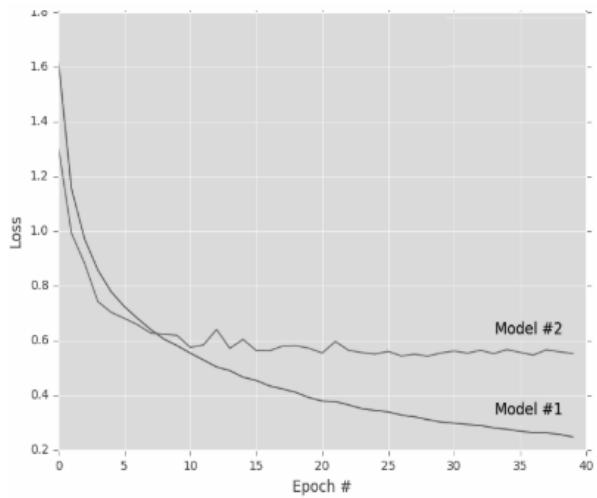
In optimization problems, our ultimate goal is to find the optimum variables (weights) that would minimize the error function as much as we can. If we don't know how far we are off from the target, how would we know what to change in the next iteration? The process of minimizing this error is called *error optimization*. There are several optimization methods that we are going to review in the next section. But for now, all we need to know from the error function is how far we are from the correct prediction or "how much we missed".

2.5.3 Error is always positive

Consider this scenario: Suppose we have two data points (two input - goal_predictions pairs) that we are trying to get our network to correctly predict. If the first gives an error of 10 and the second gives an error of -10, then our average error would be ZERO! Which is misleading because error = 0 means that our network is producing perfect predictions where in fact it missed by 10 each time. We don't want that. Thus, we want to make the error of each prediction to always be positive so that they don't cancel each other when we take the average error. Think of an archer hitting a target and he missed by 1 inch. We are not really concerned in which direction they missed. All we need to know is how far they are from the target after each shot.

A visualization of loss functions plotted over time of two separate models is shown in figure 2.x. You can see that model 1 is doing a better job minimizing the error whereas model 2 started better until epoch 6 then plateaus.

Different loss functions will give different errors for the same prediction, and thus have a considerable effect on the performance of the model. A thorough discussion of loss functions is outside the scope of this book. Instead, we will focus on the two most commonly used loss functions: 1) Mean Squared Error (and its variations) usually used for regression problems, and 2) Cross Entropy used for classification problems.



2.5.4 Mean Square Error (MSE)

MSE is commonly used in regression problems that requires the output to be a real value (like house pricing). Instead of just comparing the prediction output with the label ($\hat{y}_i - y_i$), the error is squared and is averaged over the number of data points as you see in the below equation:

$$E(W, b) = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2$$

The MSE is a good choice for a few reasons. The square ensures the error is always positive and larger errors are penalized more than smaller errors. Also, it makes the math nice, always a plus.

The notations in this formula are listed in the table below:

Table 2.1

Notation	Meaning
$E(W, b)$	The loss function. Can be also annotated as $J(W, b)$ in other literature
W	Weights matrix. In some literature, the weights are denoted by the theta sign θ
b	Biases vector
N	Number of training examples
\hat{y}_i	Prediction output. Also notated as $h_{w, b}(X)$ in some deep learning literature

y_i	The correct output (the label)
$(\hat{y}_i - y_i)$	Usually called the residual

MSE sensitivity to outliers

MSE is quite sensitive to outliers, since it squares the error value. This might not be a problem to the specific problem that you are solving. In fact, this sensitivity to outliers might be beneficial in some cases. For example, if you are predicting a stock price, you would want to take outliers into account, then the sensitivity to outliers is a good thing in this case. In other scenarios, you wouldn't want to build a model that is skewed by outliers like predicting a house price in a city. In this case, we are more interested in the median and less in the mean. A variation error function of MSE called *Mean Absolute Error* (MAE) is developed just for this purpose. It averages the absolute error over the entire dataset without taking the square of the error.

$$E(W, b) = \frac{1}{N} \sum_{i=1}^N |\hat{y}_i - y_i|$$

2.5.5 Cross Entropy

Cross Entropy is commonly used in classification problems because it quantifies the difference between two probability distributions. For example, suppose for a specific training instance, you are trying to classify a dog image out of three possible classes (dogs, cats, fish). The true distribution for this training instance is therefore:

Probability(cat)	P(dog)	P(fish)
0.0	1.0	0.0 .

You can interpret the above "true" distribution to mean that the training instance has 0% probability of being class A, 100% probability of being class B, and 0% probability of being class C. Now, suppose your machine learning algorithm predicts the following probability distribution:

Probability(cat)	P(dog)	P(fish)
0.2	0.3	0.5

How close is the predicted distribution to the true distribution? That is what the cross-entropy loss determines. Use this formula:

$$E(W, b) = - \sum_{i=1}^m y_i \log(p_i)$$

Where (y) is the target probability, (p) is the predicted probability, and (m) is the number of classes. The sum is over the three classes cat, dog, and fish. In this case the loss is 1.2 :

$$E = - (0.0 * \log(0.2) + 1.0 * \log(0.3) + 0.0 * \log(0.5)) = 1.2$$

So that is how "wrong" or "far away" your prediction is from the true distribution.

Let's do this one more time just to give some intuition of how the loss changes when the network makes better predictions. In the above example, we showed the network an image of a dog and it predicted that it is 30% a dog which is very far from the target prediction. In later iterations, the network learns some patterns and gets the predictions a little better up to 50%:

Probability(cat)	P(dog)	P(fish)
0.3	0.5	0.2

Then, we calculate the loss again:

$$E = - (0.0 * \log(0.3) + 1.0 * \log(0.5) + 0.0 * \log(0.2)) = 0.69$$

You see how when the network made a better prediction (dog up to 50% from 30%), the loss decreased from 1.2 to 0.69. In the ideal case, when the network predicts that the image is 100% a dog, the cross entropy loss will be zero (feel free to try the math).

To calculate the cross entropy error across all the training examples (n), we use this general formula:

$$E(W, b) = - \sum_{i=1}^n \sum_{j=1}^m y_{ij} \log(p_{ij})$$

It is important to call out that you will not be doing these calculations by hand. Understanding how things work under the hood, gives you a better intuition when you are designing your neural network. In deep learning projects, we usually use libraries like Tensorflow, PyTorch or Keras where the error function is usually a parameter choice.

2.5.6 A final note on errors and weights

As we mentioned before, in order for the neural network to learn, the network needs to minimize the error function as much as it can (0 is ideal). The lower the error gets, the higher the accuracy this model will be in predicting values. Now, how do we minimize this error?

Let's look at a the perceptron example below with a single input to understand the relationship between the weight and the error:

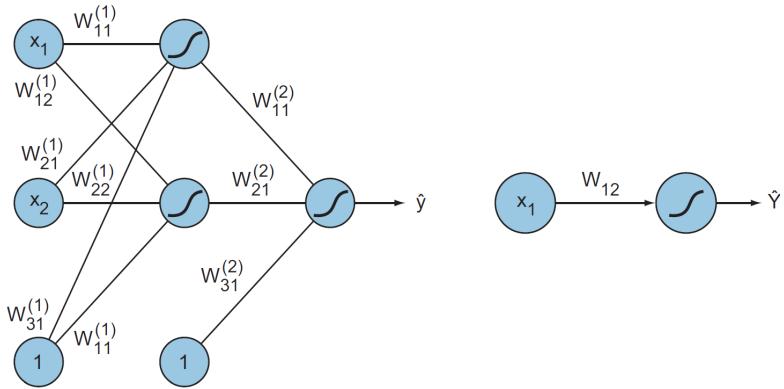


Figure 2.29

Suppose the input $x = 0.3$ and its label (goal prediction) $y = 0.8$. Then, the prediction output (\hat{y}) of this perceptron is calculated as follows:

$$\hat{y} = w \cdot x = w \cdot 0.3$$

And the error, in its simplest form, is calculated by comparing the prediction \hat{y} and the label y :

$$\begin{aligned} \text{error} &= |\hat{y} - y| \\ &= |(w \cdot x) - y| \\ &= |w \cdot 0.3 - 0.8| \end{aligned}$$

If you look at the error function above, you will notice that the input value x and the goal prediction y are fixed values. They will never change for this specific data points. The only two variables that we can change in this equation are the error and the weight. Now, if we want to get to the minimum error, which variable can we play with? Correct, the weight! The weight acts as a knob that the network needs to adjust up and down until it gets the minimum error. This is how the network learns. By adjusting weights. When we plot the error function with respect to the weight, we get the following graph:

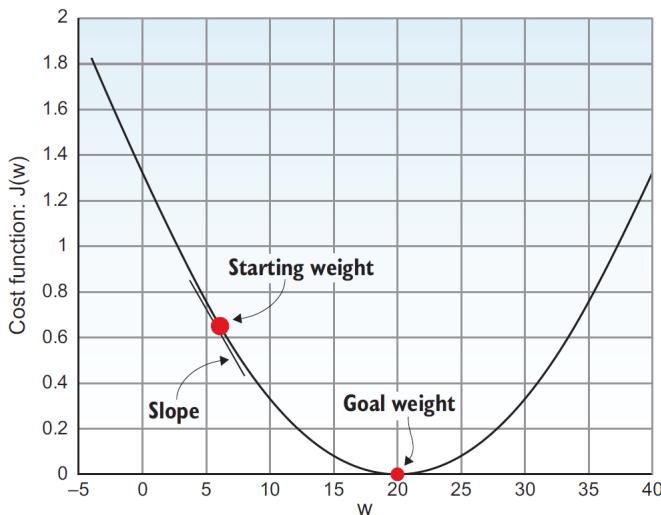


Figure 2.30

As we mentioned before, we initialize the network with *random* weights. This weight lies somewhere on this curve and our mission is to make it *descent* this curve to its optimal value with the minimum error. The process of finding the goal weights of the neural network happens by adjusting the weight values in an iterative process using an *optimization algorithm*.

2.6 Optimization algorithms

Training a neural network means showing the network many examples (training dataset), the network makes predictions through **feedforward** calculations and compare them with the correct labels to **calculate the error**. Finally, the neural network needs to **adjust the weights** (on all edges) until it gets the minimum error value. Which means maximum accuracy. Then all we need now is to build algorithms that can find these **optimum weights** for us.

2.6.1 What is Optimization?

Ahh, Optimization! A topic that is dear to my heart, and every machine learning engineer (mathematicians too). Optimization is a way of framing a problem in order to maximize or minimize some value. The best thing about computing an error function, is that we turned the neural network into an optimization problem where our goal is to *minimize the error*.

Optimization example: Suppose you want to optimize your commute from home to work. First, you need to define the metric that you are optimizing or the “error function”. Maybe you want to optimize the price of the commute, or the time, or distance. Then based on that specific loss function, you work on minimizing its value by changing some parameters. Changing the parameters to minimize (or maximize) a value is called optimization. If you choose the loss function to be the price, maybe you will take a longer commute that will take 2 hours or maybe (hypothetically) walk for 5 hours to minimize the price. On the other hand, if you want to optimize the time spent in the commute, maybe you will spend \$50 to take a cab that will make the commute time decrease to 20 minutes. So, based on the loss function that you defined, you can start changing your parameters to get the results you want.

In neural networks, optimizing the error means updating the weights and biases until we find the ***optimal weights*** or the best values for weights to produce the minimum error.

Let's look at the space that we are trying to optimize

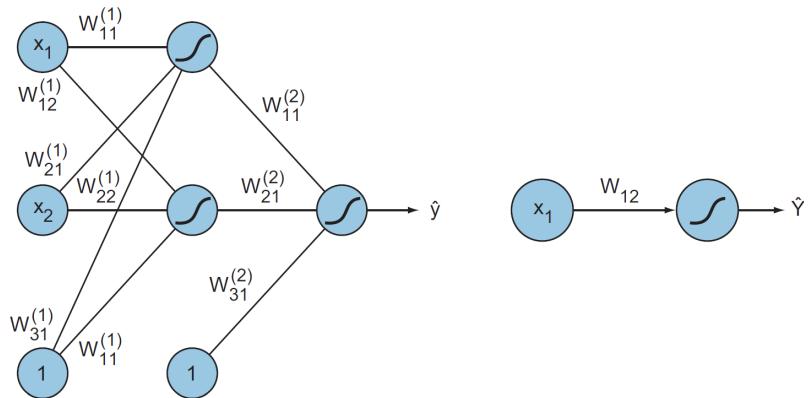


Figure 2.31

In the simplest form, a perceptron with one input, we have only one weight in our network. Then we can easily plot the error (that we are trying to minimize) with respect to this weight. Represented by this 2D curve:

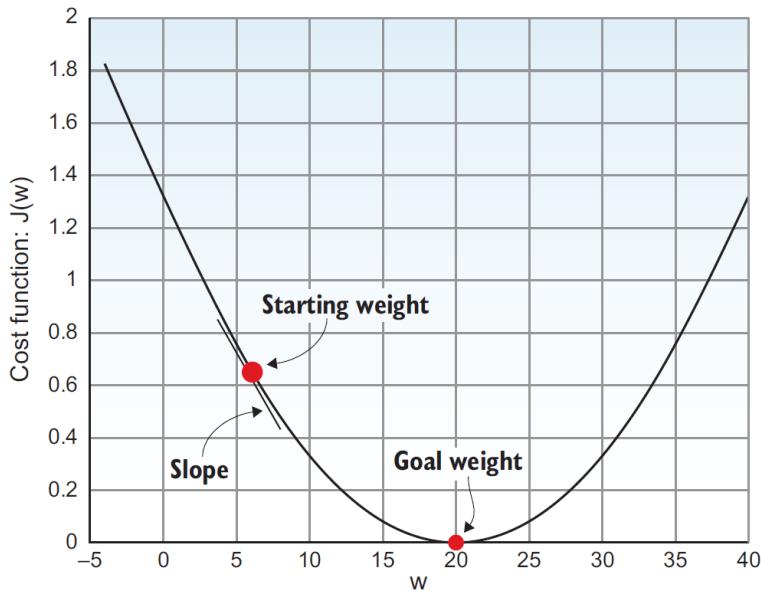


Figure 2.32

Okay, what if we have 2 weights? If we were able to graph out all the possible values of these 2 weights, we get a 3D plane of the error.

More than 2 weights? Your network will most probably have hundreds or thousands of weights (because each edge in your network has its own weight value). Since we, humans, are only equipped to understand a maximum of 3 dimensions, it is impossible for us to visualize error graphs when we have 10 weights. Not to mention 100s or 1000s weights. So, from this point we will study the error function using the 2D or 3D planes of the error. In order to optimize the model, our goal is to search this space to find the best weights that will achieve the lowest possible error.

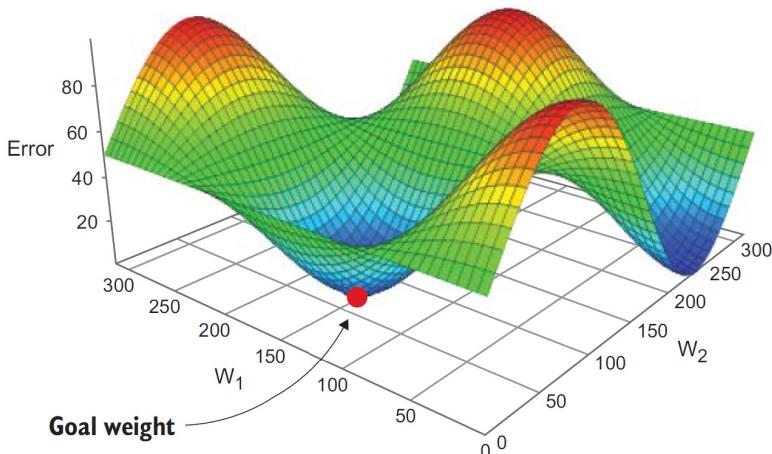


Figure 2.33

Why do we need an optimization algorithm? Can we just brute force through a lot of weight values until we get the minimum error?

One possible approach to do (only theoretically) is a brute force approach where we just try out a lot of different possible weights (say 1,000 values) and find the weight that produces the minimum error. Would that work? Well, theoretically yes. This approach might work when we have very few inputs and a one or two neurons in our network. Let me try to convince you that this approach wouldn't scale. Let's take a look at a scenario when we have a very simple neural network - suppose we want to predict house price based on only four features (inputs) and just one hidden layer of 5 neurons:

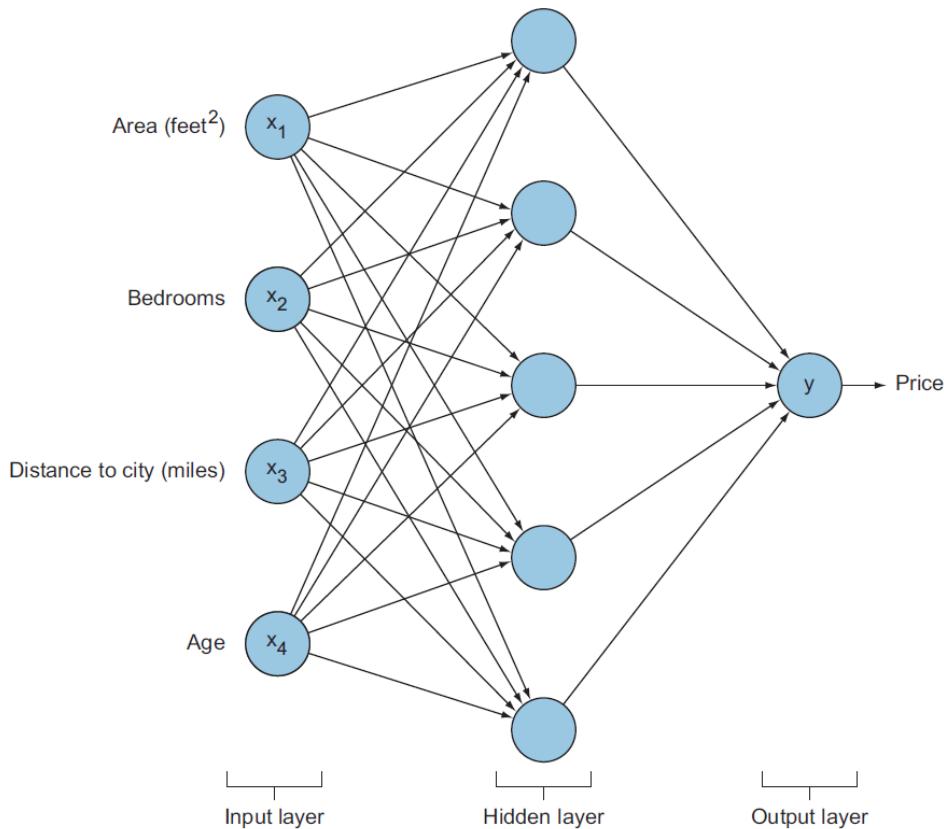


Figure 2.34

As you can see, we have 20 edges (weights) from the input to the hidden layer + 5 weights from the hidden layer to the output prediction = 25 total weight variables that need to be adjusted for optimum values. To brute force our way through a simple neural network of this size, we need the following:

If we are trying 1,000 different values for each weight then we will have a total of 10^{75} combinations

$$1,000 * 1,000 * \dots * 1,000 = 1,000^{25} = 10^{75} \text{ combinations}$$

Let's say we were able to get our hands on the fastest supercomputer in the world, Sunway Taihulight, that operates at a speed of 93 petaFLOPS (Floating Operation Per Second) => 93×10^{15} FLOPS. In the best case scenario, this supercomputer would need:

$$\frac{10^{75}}{(93 * 10^{15})} = 1.08 \times 10^{58} \text{ seconds} = 3.42 \times 10^{50} \text{ years}$$

That is a huge number that is longer than the universe has existed. Who has that kind of time to wait for the network to train? Remember that this is a very simple neural network that usually takes a few minutes to train using a smarter optimization algorithms. In real world, you will be building more complex networks that have thousands of inputs and tens of hidden layers and you will be required to train them in a matter of hours (or days and sometime weeks). So we have to come up with a different approach to find the optimal weights.

Hopefully, I have convinced you that brute forcing through the optimization process is not the answer. Now, let's study the most popular optimization algorithm for neural networks, the Gradient Descent. Gradient Descent has several variations, Batch gradient descent (BGD), Stochastic GD (SGD), and Mini-batch GD (MN-GD).

2.6.2 Batch Gradient Descent (BGD)

What is a gradient?

The general definition of the gradient is that it is the function that tells you the slope or rate of change of the line tangent to the curve at any given point. Also known as the derivative. It is just a fancy term for slope or steepness of the curve.

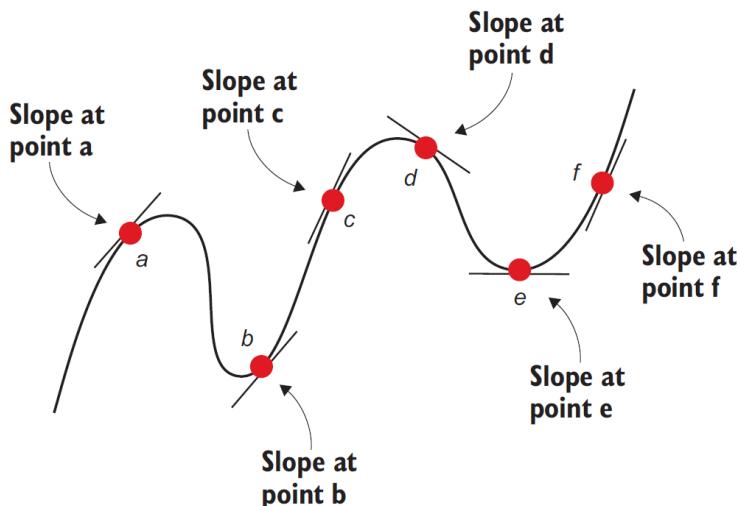


Figure 2.35

What is gradient descent?

Gradient descent simply means, updating the weights iteratively to descent the slope of the error curve until we get the point with minimum error. Let's that a look at the error function

that we introduced earlier with respect to the weights. At the initial weight point, we calculate the derivative of the error function to get the slope (direction) of the next step. We keep taking steps down the curve until we reach to the minimum error.

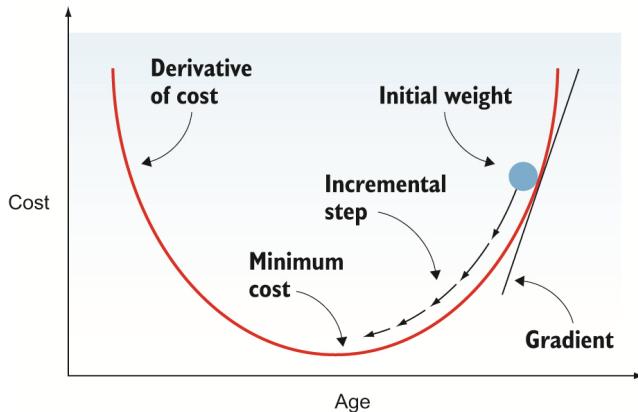


Figure 2.36

How does gradient descent work?

To visualize how the gradient descent works, let's plot the error function in 3D graph and go through the process step-by-step. You can see the random initial weight (starting weight) is at point A and our goal is to descent this error mountain to the goal w_1 and w_1 weight values which produce the minimum value of the error. The way we do that is by taking a series of steps *down* the curve until we get the minimum error. In order to descent the error mountain, we need to determine two things for each step:

1. The step direction (gradient)
2. The step size (learning rate)

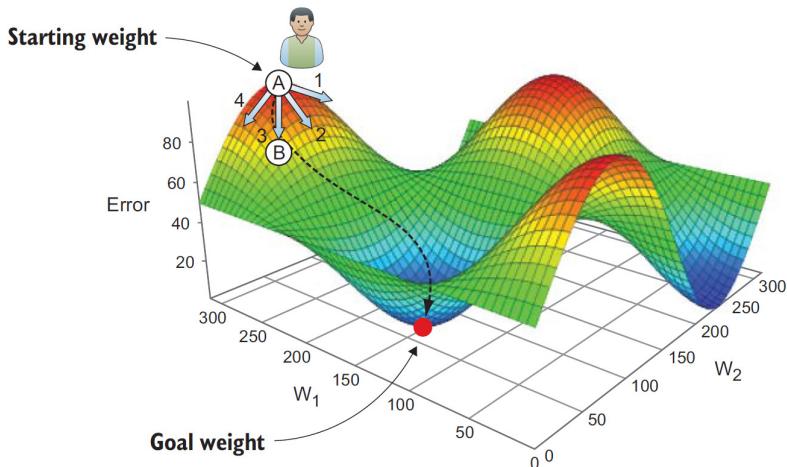


Figure 2.37

1) The direction (gradient)

Suppose you are standing on the top of the error mountain at point A. To get to the bottom, you need to determine the step direction that will make you descend the most (i.e. has the steepest slope). And what is the slope again? It is the derivative of the curve. So, if you are standing on the top of that mountain, you need to look at all the directions around you and find out which direction will get you to descent the most (1, 2, 3, or 4 for example). Let's say it is direction 3. Then, we get to point B and we restart the process again (calculate feedforward and error) and find the direction that will descent the most and so forth until we get to the bottom of the mountain.

This process is called gradient descent. By taking the derivative of the error with respect to the weight (dE/dw), we get the direction that we should take. Now, one thing left. The gradient only determines the direction. How big should the size of the step be? It could be 1 foot step or a 100 feet jump. This is what we need to determine next.

2) The step size (learning rate α)

Learning rate is the size of each step that the network will be taking when it is descending the error mountain and it is usually denoted by the greek letter alpha (α). It is one of the most important hyperparameters that you will be tuning when you are training your neural network (more on that later). Larger learning rate means that the network will learn faster (since it is descending the mountain with larger steps) and smaller steps means slower training. Well, this sounds simple enough. Let's use large learning rates and complete the neural network training in minutes instead of waiting for hours. Right? Not quite. Let's take a look at what could happen when we set a very large learning rate value.

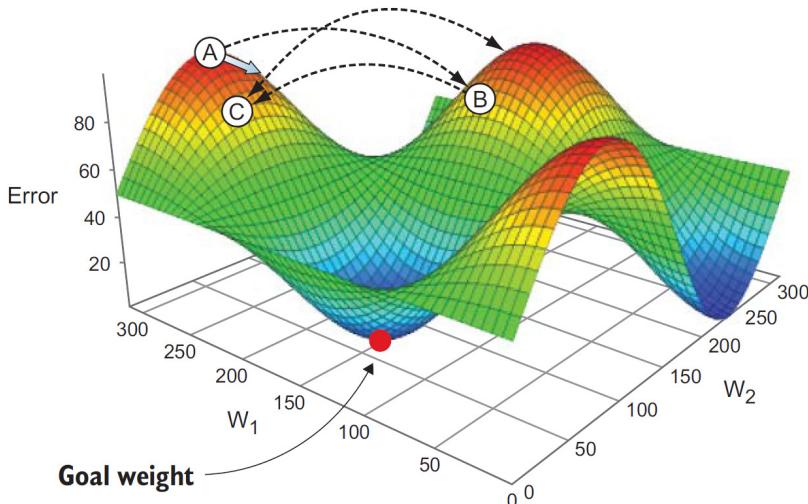


Figure 2.38

In figure 2.x, you are starting at point A. When you take a large step in the arrow direction, instead of descending the error mountain, you will end up in point B on the other side. Then another large step to C and so forth. The error will keep *oscillating* and will never descent. We will talk more about the learning rate tuning and how to determine that the error is oscillating. But for now, you need to know this: If you use very small learning rate, the network will eventually descent the mountain and will get to the minimum error. But this will take longer time to train (maybe weeks or months). On the other hand, if you use very large learning rate, the network might keep oscillating and never train. So, we usually initialize the learning rate value to 0.1 or 0.01 then see how the network performs then tune it further.

Putting direction and step together

By multiplying the direction (derivative) by the step size (learning rate), we get the change of the weight for each step:

$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

We add the minus sign because the derivative always calculates the slope in the upward direction. Since we need to descent the mountain, we go in the opposite direction of the slope.

$$w_{next-step} = w_{current} + \Delta w$$

Calculus refresher

Calculate the Partial Derivative

The derivative is the study of change. It measures the steepness of a curve at some particular point on that graph.

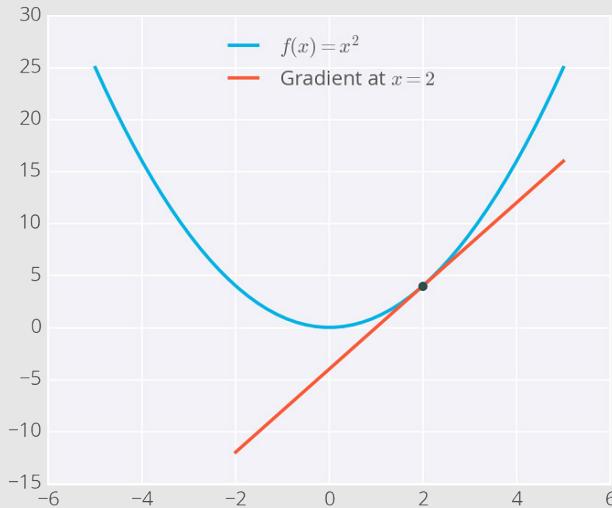


Figure 2.39

It looks like Mathematics has given us just what we are looking for. On the error graph, we want to find the steepness of the curve at the exact weight point. Thank you, Math!

Other words for the derivative is the slope or rate of change. If the error function is denoted as $E(x)$, then the derivative of the error function *with respect to the weight* is denoted as: $d/dw \cdot E(x)$ or just $dE(x)/dw$. This formula basically says, how much will the total error change when we change the weight.

Luckily, mathematicians created some rules for us to calculate the derivative. Since this is not a mathematics book, we will not discuss the proof of the rules. Instead, we will just take it from this point and start applying these rules to calculate our gradient. Here are the basic derivative rules:

Basic derivatives rules:

Constant Rule: $\frac{d}{dx}(c) = 0$	Difference Rule: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$
Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = cf'(x)$	Product Rule: $\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$
Power Rule: $\frac{d}{dx}(x^n) = nx^{n-1}$	Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Sum Rule: $\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$	Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

Figure 2.40

Let's take a look at a simple function to apply the above derivative rules:

$$f(x) = 10x^5 + 4x^7 + 12x$$

We can apply the power, constant, and sum rules to get df/dx also denoted as $f'(x)$:

then, $f'(x) = 50x^4 + 28x^6 + 12$

To get an intuition of what this means, let's plot $f(x)$:

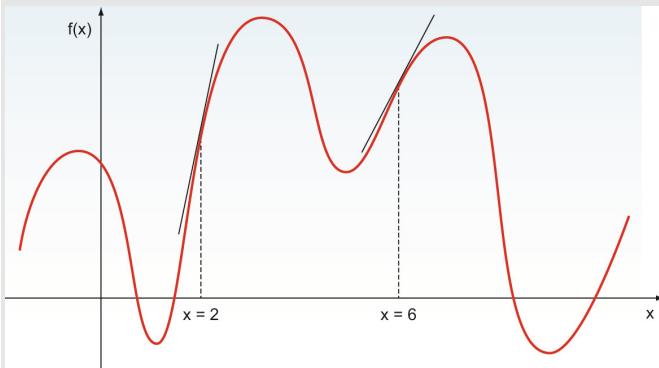


Figure 2.41

If we want to get the slope at any point, we can just compute the $f'(x)$ at that point. So, $f'(2)$ gives us the slope of the red line on the left and $f'(6)$ is the slope of the second line. Get it?

Last example on derivatives, let's apply the power rule to calculate the derivative of the sigmoid function:

$$\begin{aligned}\frac{d}{dx} \sigma(x) &= \frac{d}{dx} \left[\frac{1}{1+e^{-x}} \right] \\ &= \frac{d}{dx} (1+e^{-x})^{-1} \quad \leftarrow \text{power rule} \\ &= -(1+e^{-x})^{-2}(-e^{-x}) \\ &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\ &= \sigma(x) \cdot (1 - \sigma(x))\end{aligned}$$

If you want to write out the derivative of the sigmoid activation function in code, it will look like this:

```
# Sigmoid activation function
def sigmoid(x):
    return 1/(1+np.exp(-x))

# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

Note that you don't need to memorize the derivative rules nor would you need to calculate the derivatives of the functions yourself. Thanks to the awesome deep learning community, we have great libraries that will compute these functions for you in just one line of code. It is just valuable to understand how things are happening under the hood.

PITFALLS OF BATCH GRADIENT DESCENT

Gradient descent is a very powerful algorithm to get to the minimum error. But, it has two major pitfalls:

- 1) Not all cost functions look like simple bowls as we saw above. There maybe holes, ridges and all sorts of irregular terrains that make reaching the minimum error very hard. Consider figure 2.x, where the error function is a little more complex with ups and downs.

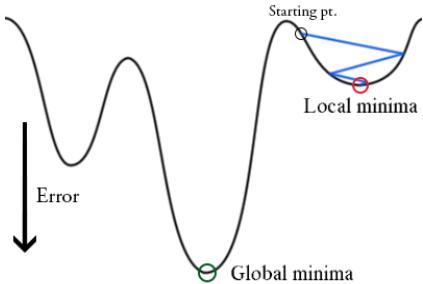


Figure 2.42

Remember what we said about weights initialization? The starting point is randomly selected. What if the starting point is as shown in this figure when we start the gradient descent algorithm? The error will start descending the small mountain on the right and will indeed reach to a minimum value. But, this minimum value is not the lowest possible error value for this error function. This is called the *local minima*. Where this specific point is the minimum value for the local mountain where the weight randomly started. Instead, we want to get to the lowest possible error called *global minima*.

- 2) Batch gradient descent uses the entire training set to compute the gradients at every step. Remember the loss function below?

$$L(W, b) = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2$$

This means that if your training set (N) has 100,000,000 (100 million) records, then the algorithm will need to sum over 100 million records just to take *one step*. And that is computationally very expensive and slow to train. That is why this algorithm is also called *Batch Gradient Descent* because it uses the entire training data in one batch.

One possible approach to solve these two problems is Stochastic Gradient Descent (SGD). By which the algorithm randomly selects data points and go through the gradient descent one data point at a time. This will provide many different weight starting points and descend all the mountains to calculate their local minimas. Then the minimum value of all these local minimas is the global minima. Sounds very intuitive. That is concept behind *Stochastic Gradient Descent* algorithm.

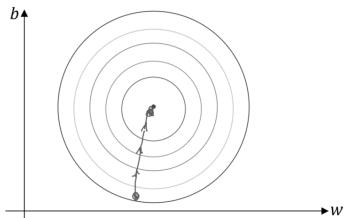
2.6.3 Stochastic Gradient Descent (SGD)

Stochastic is just a fancy word for random. Stochastic gradient descent (SGD) is probably the most used optimization algorithms for machine learning in general and for deep learning in particular. While gradient descent measures the loss and gradient over the full training set to take one step towards the minimum, stochastic gradient descent *randomly* picks *one instance* in the training set for each one step and calculates the gradient based only on that single

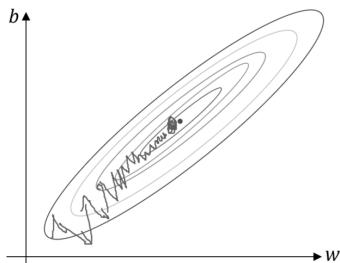
instance. Now, let's take a look at the pseudocode of both GD and SGD to get a better understanding of the differences between both algorithms:

Table 2.1

GD	Stochastic GD
1) Take ALL the data	1) randomly shuffle samples in the training set
2) Compute the gradient	2) Pick one data instance
3) Update the weights and take a step down	3) Compute the gradient
4) Repeat for n number of epochs (iterations)	4) Update the weights and take a step down
	5) Pick another one data instance
	6) Repeat for n number of epochs (training iterations)



Top View of the error mountain



Top View of the error mountain

Because we take a step after we compute the gradient for the entire training data in batch GD, you can see that the path down the error is smooth and almost a straight line. Whereas, due to the stochastic (random) nature of SGD, you will see the path towards the global cost minimum is not direct as in BGD, but may go "zig-zag" if we are visualizing the cost surface in a 2D space (figure 2.x). And that is because in SGD every iteration is trying to fit just a single training example better which makes it a lot faster but it does not guarantee that every step is taking us a step down the curve. Which is fine because it will end up very close to the global minimum and once it get there it will continue to bounce around never settling down. In practice this isn't a problem because it will end up very close to the *global minimum* which is good enough for most practical purposes. Generally SGD almost always performs better and faster than batch GD.

2.6.4 Mini-batch Gradient Descent (MN-GD)

Mini-Batch Gradient Descent (MB-GD) a compromise between BGD and SGD. Instead of computing the gradient from 1 sample (SGD) or all training samples (BGD), we divide the training sample into *mini-batches* to compute the gradient from (a common mini-batch size is $k=256$). MB-GD converges in fewer iterations than GD because we update the weights more

frequently; however, MB-GD let us utilize vectorized operation, which typically results in a computational performance gain over SGD.

2.6.5 Gradient descent takeaways

There is kind of a lot going on here, so let's just sum it up for ourselves, shall we? Here is how gradient descent is summarized in my head:

- Three types: batch, stochastic, mini-batch gradient descent
- All follow the same concept:
 - They find the direction of the steepest slope: derivative of the error with respect to the weight dE/dw_i
 - Set the learning rate (or step size): the algorithm will compute the slope, but you will set the learning rate as a hyperparameter that you will tune by trial and error
 - Start the learning rate with 0.01 then go down 0.001, 0.0001, 0.00001. The lower you set your learning rate the more guaranteed to descent to the minimum error (if you train for infinity of time). Since we don't have infinity time, 0.01 is a reasonable start then go down from there
- Batch GD: updates the weights after computing the gradient of *ALL* the training data. This can be computationally very expensive when the data is huge. Doesn't scale well.
- Stochastic: updates the weights after computing the gradient of a single instance of the training data. SGD is faster and usually reaches very close to the global minimum
- Mini-batch GD: a compromise between batch and stochastic. Not all data nor single instance. Instead, takes a group of training instances (called mini-batch) to compute the gradient on them and update the weights, then repeats until it does that on all the training data. In most cases, mini-batch gradient descent is a good starting point to start experimenting.
 - `batch_size`: is a hyperparameter that you will tune. This will come up again in the hyperparameter tuning section in chapter 4. But typically you can start experimenting with `batch_size = 32, 64, 128, 256`
 - Don't get the `batch_size` confused with the `epochs`. An epoch is the full cycle over all the training data. The batch is how many training sample in the group that we are computing the gradient for. For example, if we have 1000 sample in our training data and set the `batch_size = 256`, then Epoch 1 = batch 1 of 256 samples + batch 2 (256 samples) + batch 3 (256 samples) + batch 4 (232 samples)

Finally, you need to know that there have been a lot of variations to gradient descent that have been used over the years. And this is a very active research area. Some of the most popular enhancements are:

- Nesterov accelerated gradient
- RMSprop
- Adam
- Adagrad

But, don't worry about these optimizers now. In chapter 4, we will discuss tuning techniques to choose and improve your optimizers learning in more details.

I know that was a lot, but stay with me. These are the main things I want to you remember from this section:

- How gradient descent works (slope + step size),
- the difference between batch, stochastic, and mini-batch, and
- these are GD hyperparameters that you will tune: learning rate and batch_size.

If you got this covered, you are good to move to the next section. And don't worry a lot about the hyperparameter tuning. I'll be covering the network tuning in more details in the next chapters and in almost all projects in this book.

2.7 Backpropagation

Backpropagation is the core of how neural networks learn. Up until this point, we learned that training a neural network typically happens by the repetition of the following 3 steps:

- 1) Feedforward: get the linear combination (weighted sum) and apply the activation function to get the output prediction (\hat{y}).

$$\hat{y} = W^{(3)} \circ \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)} \circ \sigma \circ (x)$$

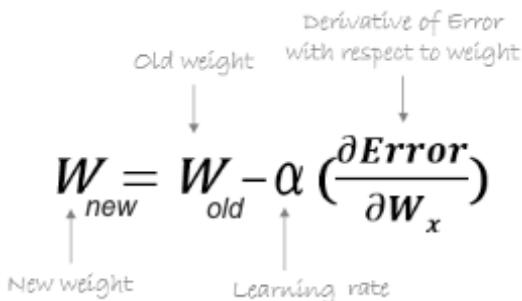
- 2) Compare the prediction with the label to calculate the error or loss function

$$E(W, b) = \frac{1}{N} \sum_{i=1}^N |\hat{y}_i - y_i|$$

- 3) a - Use gradient descent optimization algorithm to compute the Δw that optimizes the error function

$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

b - Backpropagate the delta_weight through the network to update the weights



In this section, we will dive deeper into step 3-b, backpropagation.

2.7.1 What is backpropagation?

Backpropagation or backward pass is propagating derivatives of the error, with respect to each specific weight dE/dw_i from the last layer (output) back to the first layer (inputs) to adjust weights.

By propagating the delta_weight backwards from the prediction node ($y_{\hat{y}}$) all the way through the hidden layers back to the input layer, the weights get updated ($w_{next-step} =$

$w_{current} + \Delta w$) which will take the error one step down the error mountain. Then the cycle starts again (steps 1 to 3) to update the weights to take the error another step down. Until we get to the minimum error. This backward pass process is called backpropagation.

Backpropagation might sound clear when we have only one weight. We simply adjust the weight by adding the Δw ($w_{new} = w - \alpha \cdot dE/dw_i$). But it gets complicated when have a multi-layer perceptron (MLP) network with many weights variables. To make this clearer, consider this scenario:

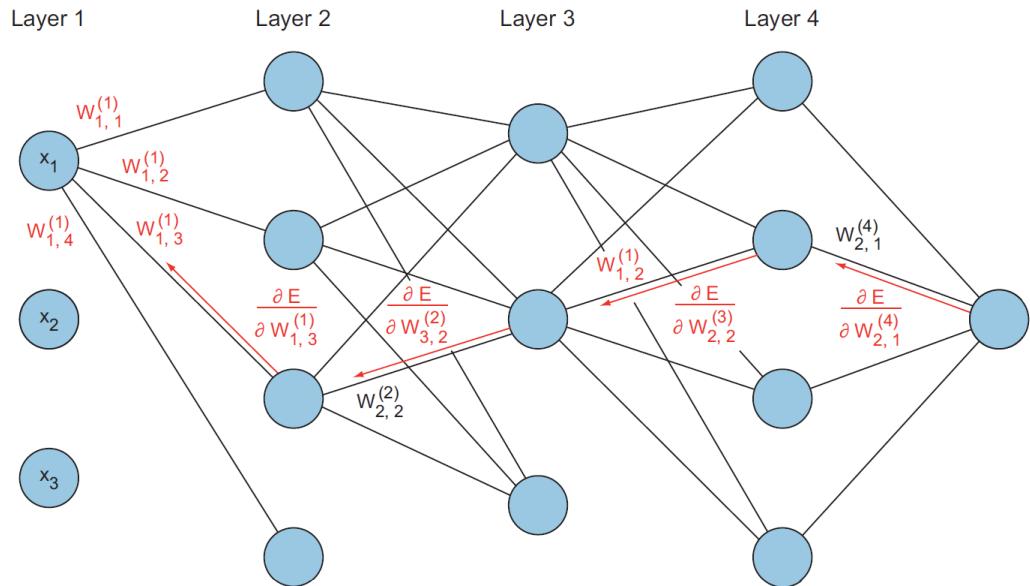


Figure 2.45

How do we compute the change of the total error with respect to w_{13} (dE/dw_{13})? Remember that dE/dw_{13} basically says: how much would the total error change when we change the parameter dw_{13} ?

We learned how to compute dE/dw_{21} by applying the derivative rules on the error function. That is straightforward because w_{21} is directly connected to the error function. But to compute the derivatives of the total error with respect to the weights all the way back to the input, we need a calculus rule called *The Chain Rule*.

Calculus refresher

Chain Rule In Derivatives

Back again to Calculus. Remember in the derivative rules that we listed earlier? One of the most important rules is the Chain rule. Let's dive deep into it to see how it is implemented in backpropagation.

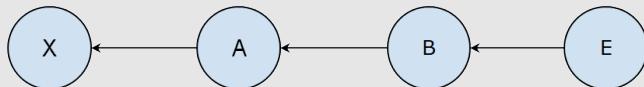
$$\text{Chain Rule: } \frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$$

The chain rule is a formula for calculating the derivatives of functions that are composed of functions inside other functions. It is also called the outside-inside rule. Look at this:

$$\begin{aligned}\frac{d}{dx} f(g(x)) &= \frac{d}{dx} \text{ outside function} * \frac{d}{dx} \text{ inside function} \\ &= \frac{d}{dx} f(g(x)) * \frac{d}{dx} g(x)\end{aligned}$$

The chain rule just says: "when composing functions, the derivatives just multiply". That is going to be very useful for us when implementing backpropagation because feedforwarding is just composing a bunch of functions and backpropagation is taking the derivative at each piece of this function.

To implement the chain rule in backpropagation, all we are going to do is multiply a bunch of partial derivatives to get the effect of errors all the way back to the input. Here is how it works, but first, remember that our goal is to propagate the error backwards all the way to the input layer. So in the figure below, we want to calculate dE/dx which is the effect of total error on input (x):



$$\frac{dE}{dx} = \frac{dE}{dB} * \frac{dB}{dA} * \frac{dA}{dx}$$

All we did here is multiply the upstream gradient * the local gradient all the way until we get to the target value.

Let's see how backpropagation uses the chain rule to flow the gradients in the backward direction through the network:

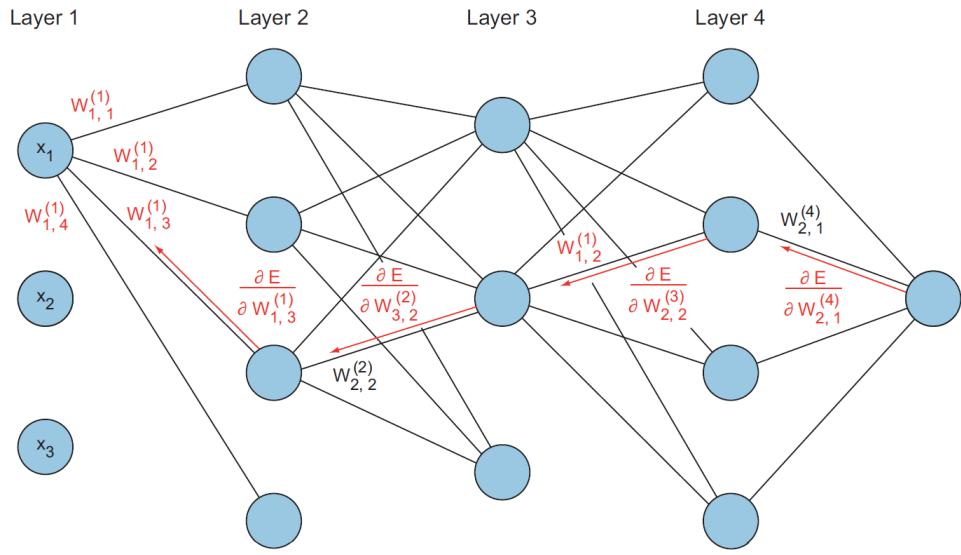


Figure 2.45b

Okay, let's apply the chain rule to calculate the derivative of the error with respect to the third weight on the first input $w_{1,3}^{(1)}$, where the (1) means layer 1 and $w_{1,3}$ means node number 1 and weight number 3.

$$\frac{dE}{dw_{1,3}^{(1)}} = \frac{dE}{dw_{2,1}^{(4)}} * \frac{dw_{2,1}^{(4)}}{dw_{2,2}^{(3)}} * \frac{dw_{2,2}^{(3)}}{dw_{3,2}^{(2)}} * \frac{dw_{3,2}^{(2)}}{dw_{1,3}^{(1)}}$$

The equation might look complex at the beginning but all what we are doing really is multiplying the partial derivative of the edges starting from the output node all the way backward to the input node. All the notations are what is making this look complex but once you understand how to read $w_{1,3}^{(1)}$, the backward pass equation will really look like this:

The error backpropagated to the edge $w_{1,3}^{(1)} = \text{effect of error on edge 4} * \text{effect on edge 3} * \text{effect on edge 2} * \text{effect on target edge}$

There you have it. That is the backpropagation technique used by neural networks to update the weights to best fit our problem. Let's take a quick look on how this is implemented in code.

2.7.2 Backpropagation takeaways

- Backpropagation is a learning procedure for neurons
- Backpropagation repeatedly adjusts weights of the connections (weights) in the network to minimize the cost function (the difference between the actual output vector and the desired output vector)
- As a result of the weight adjustments, hidden layers come to represent important features other than the features represented in the input layer.
- For each layer, the goal is to find a set of weights that ensure that for each input vector the output vector produced is the same (or close to) the desired output vector. The difference in values between the produced and desired outputs is called the error function.
- *Backward pass (or backpropagation)*: starts at the end of the network, backpropagate or feed the errors back, recursively apply chain rule to compute gradients all the way to the inputs of the network and then update the weights. This method of backpropagating the errors and computing the gradients is called backpropagation.
- To reiterate, the goal of a typical neural network problem is to discover a model that fits our data 'best'. Ultimately, we want to minimize the cost or loss function by choosing the best set of weights parameters.

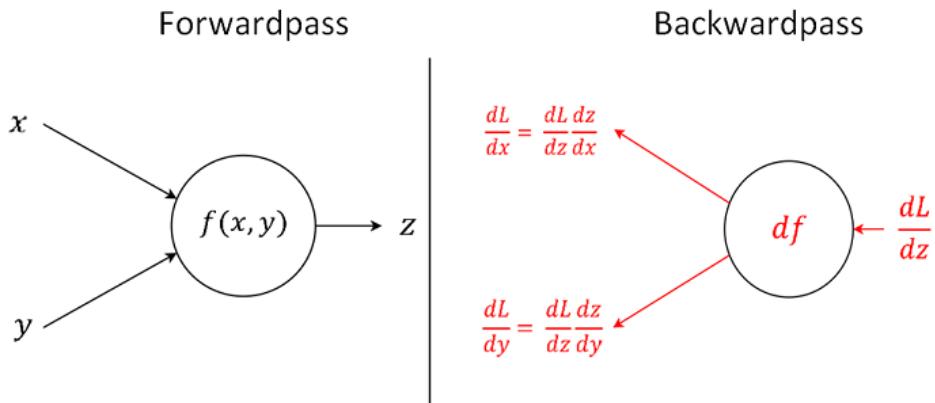


Figure 2.47

2.8 Chapter summary and takeaways

I know this could be a lot to take. But, here is what I want you to take away from this chapter:

- Perceptrons work fine for datasets that can be separated by one straight line (linear operation)
- Nonlinear datasets that cannot be modeled by a straight line need more complex neural network that contains many neurons. Stacking neurons in layers is called Multi-layer Perceptron (MLP).
- The network learns by the repetition of 3 main steps:
 - 1) **Feedforward:** contains two main operations: calculate the weighted sum then apply the activation function.
 $\hat{y} = \text{activation}(\sum x_i \cdot w_i + b)$
 - 2) **Calculate error:** compare the predicted output from the feedforward process with the true label
 $\text{error} = y - \hat{y}$
 - 3) **Optimize weights:** backpropagate the error using gradient descent to minimize weights

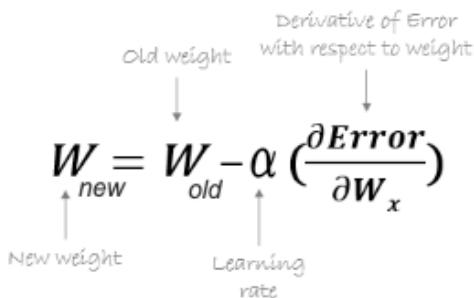


Figure 2.48

- Pay attention to the difference between the parameters and hyperparameters:
 - **Parameters** are variables that are updated by the network during the training process like weights and biases. These are tuned automatically by the model during training.
 - **Hyperparameters** are variables that you tune like number of layers, activation functions, loss functions, optimizers, early stopping, learning rate. These are tuned by the us before training the model.

Neural network hyperparameters:

- Number of hidden layers: You can have as many layers as you want, each with as many neurons as you want. The general idea is, the more neurons you have the more your network will learn the training data. But if you have too many neurons, this might

lead to a phenomenon that is called **overfitting**. Which means that they network learned the training set too much that it memorized it instead of learning its features. Thus, will fail to generalize. To get the appropriate number of layers, start with a small network and observe the network performance. Then start adding layers until you get satisfying results.

- **Activation functions:** There are many types of activation functions that you can use. The two most popular ones are ReLU and Softmax. It is recommended that you use ReLU activation in the hidden layers and Softmax for the output layer (you will see how this is implemented in this chapter's project coming next).
- **Error functions:** measures how far the network's prediction from the true label. Mean Square Error (MSE) is common for regression problems and Cross Entropy is common for classification problems.
- **Optimizer:** optimization algorithm is used to find the optimum weight values that minimize the error. There are several optimizer types to choose from. In this chapter, we discussed Batch Gradient Descent BGD, Stochastic GD, and Mini-batch GD. Other popular optimizers that we didn't discuss here are: Adam and RMSProp.
- **Batch size:** Mini batch size is the number of sub samples given to the network after which parameter update happens. Bigger batch sizes, learn faster but requires bigger space in memory. A good default for batch size might be 32. Also try 32, 64, 128, 256, and so on.
- **Number of epochs:** Number of epochs is the number of times the whole training data is shown to the network while training. Increase the number of epochs until the validation accuracy starts decreasing even when training accuracy is increasing (overfitting).
- **Learning rate:** one of the optimizer's input parameter that we will tune is the learning rate. Theoretically, too small learning rate is guaranteed to reach to the minimum error (if you train for infinity of time). Too big learning rate speeds up the learning but not guaranteed to find the minimum error. The default "lr" value of the optimizer in most deep learning library is a reasonable start to get you descent results. Then go down or up by one order of magnitude from there.

Gradient Descent

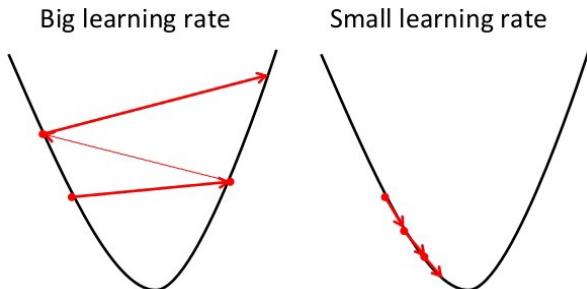


Figure 2.49

There are other hyperparameters that we did not discuss yet like dropout, regularization, etc. We will discuss the hyperparameter tuning in details in chapter 4 after we cover convolutional neural networks in the next chapter. In chapter 4, we will also cover some techniques to tune your hyperparameters.

In general, the best way to tune hyperparameters is by trial and error. By getting your hands dirty with your own projects as well as learning from people's other neural network architectures, you start to build an intuition on good starting points for hyperparameters.

The second thing is to learn to analyze your network's performance and understand which hyperparameter you need to tune for each symptom. And this is what we are going to do in this book. By understanding the intuition behind these hyperparameters and observing the network performance in the projects at the end of the chapters, we will build a clear intuition about which hyperparameter to tune to achieve one effect. For example, if we see that our error value is not decreasing and keeps oscillating, then we might fix that by reducing the learning rate. OR if we see that the network is performing poorly in learning the training data, this might mean that the network is underfitting and we need to build a more complex model by adding more neurons and hidden layers.

2.9 Project: Build Your first Neural Network

[***** **Work in Progress** *****]

3

Convolutional Neural Networks (CNNs)

Previously, we talked about Artificial Neural Networks (ANN) also known as Multi-Layer Perceptron (MLP) which are basically layers of neurons stacked on top of each other that have learnable weights and biases. Each neuron receives some inputs, multiplied by their weights and apply non-linearity via activation functions. In this chapter, we will talk about Convolutional Neural Networks (CNNs) which are considered an evolution of the MLP architecture that perform a lot better with images. The high-level layout of this chapter will be as follows:

- A) **Image classification with MLP:** First, we will start off by a mini-project to classify images using MLP topology and understand how the regular neural network architecture process images. Here you will learn the MLP architecture drawbacks in processing images and why we need a new creative neural network architecture to process images.
- B) **Understand CNNs:** Explain convolutional networks to see how they extract features from images and classify objects. Here, you will learn the three main components of CNNs: convolutional layer, pooling layer, and fully connected layer. Then we will apply that in another mini-project to classify images with CNNs.
- C) **Color images:** We will explain how computers see colored images vs grayscale images and how the convolution is implemented over colored images.
- D) **Image classification project:** Lastly, we will apply all what we learned in this chapter in an end-to-end image classification project to classify colored images with CNNs.

One thing worth noting before jumping into CNNs, is that the basic concepts of how the network learns and optimizes parameters are the same with both MLPs and CNNs:

- **Architecture:** MLPs and CNNs are composed of layers of neurons that are stacked on top of each other. With CNNs they have different structures (convolutional vs fully-connected FC layers) as we are going to see in the coming sections.
- **Weights and biases:** In Convolutional and FC layers, inference works the same way. Both have weights and biases that are initially randomly generated and their values are learned by the network. The main difference between both is that the weights in MLPs are in a vector form, whereas in convolutional layers they take the form of convolutional filters or kernels.
- **Hyperparameters:** As with MLP, when we design CNNs we will always specify the error function, activation function and optimizer. All hyperparameters explained in the previous chapters remain the same. We will add some new ones that are specific to CNNs.
- **Training:** both networks learn the same way. 1) Perform a forward pass to get predictions, 2) compare the prediction with the true label to get the loss function ($y - \hat{y}$), 3) optimize parameters using gradient descent and backpropagate the error to all the weights and update their values to minimize the loss function.

By the end of this chapter you will:

- Build an image classifier using MLP
- Understand the drawbacks of MLPs in treating images and how CNNs fix them
- Understand the convolutional neural networks CNNs architecture
- Build a Convolutional Neural Network to classify images
- Tune hyperparameters to improve the CNN performance
- Add dropout layers to avoid overfitting

Ready? let's get started!

3.1 Image classification using MLP

Let's recall the MLP architecture from chapter 2. Neurons are stacked in layers on top of each other with weights connections. The MLP architecture consists of 1) input layer, 2) one or more hidden layers, and 3) output layer.

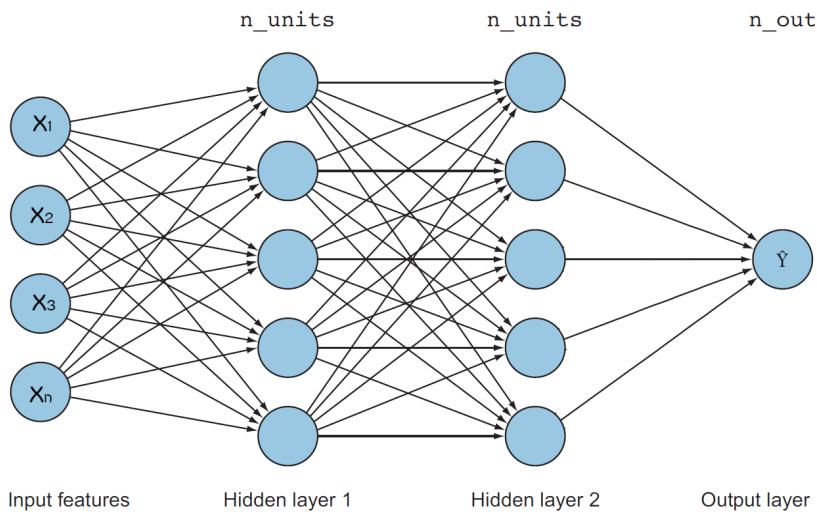


Figure 3.1

In chapter 2, you saw how we implemented MLP in the house price prediction model. Let's take what we learned about MLPs to solve an image classification problem using the MNIST dataset. The goal of this classifier is to classify images of digits from 0 to 9 (i.e. 10 classes).

Let's look at the three main components of our MLP architecture (input layer, hidden layers, and output layer):

3.1.1 Input layer

In the last chapter things were pretty straight forward. We had a 1D vector of features ($x_1, x_2, x_3, \dots, x_n$) that are fed to the input layer of the network. But to allow the network to understand 2D images, we need to preprocess the images first before feeding them to the network.

First, let's see how computers see images: In the figure below, we have an image of 28 pixels wide x 28 pixels height. This image is seen by the computer as a 28x28 matrix with pixel values range from 0 to 255 (0 for black and 255 for white and the range in between for gray scale).



$$28 \times 28 \\ = 784 \text{ pixels}$$

Figure 3.2

Since MLPs only take the input as a 1D vector with dimensions $(1, n)$, they cannot take the raw 2D image matrix with dimensions (x, y) . To fit the image in the input layer, we first need to transform our image into one large vector with the dimensions $(1, n)$ that contains all the pixels' values of the image. This process is called *image flattening*. In this example, the total number (n) of pixels in this image is $28 \times 28 = 784$. Then, in order to feed this image to our network, we need to flatten the (28×28) matrix to one long vector with dimensions $(1, 784)$. The input vector will look like this:

$x = [row1, row2, row3, \dots, row28]$

That said, the input layer in this example will have a total of 784 nodes. $x_1, x_2, x_3, \dots, x_{784}$.

To help visualize the flattened input vector, let's look at a much smaller matrix (4, 4):

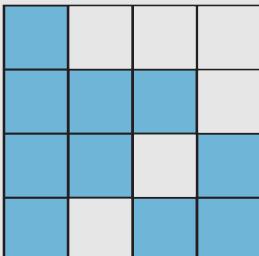


Figure 3.3

The input (x) is a flattened vector with the dimensions (1, 16):

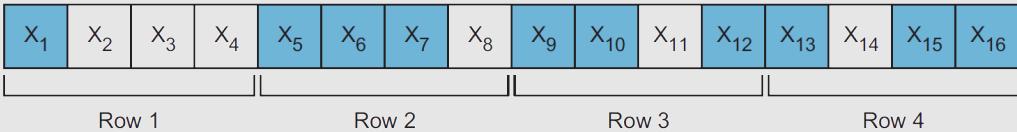


Figure 3.4

So, if we have pixel values 0 for black and 255 for white, the above input vector will be as follows:

```
Input = [0, 255, 255, 255, 0, 0, 0, 255, 0, 0, 255, 0, 0, 255, 0, 0]
```

Here is how we flatten the input image in Keras:

```
# same as before, we start with importing keras library
from keras.models import Sequential
# here we import a Layer called Flatten to convert the image matrix into a vector
from keras.layers import Flatten

# define the model
model = Sequential()
# add the flatten Layer - also known as the input layer
model.add(Flatten(input_shape = (28,28)))
```

Alright, the *Flatten* layer in Keras takes care of this process for us. It takes the 2D image matrix input and converts it to a 1D vector. Note that the *Flatten* layer must be supplied a parameter value of the shape of the input image. Now the image is ready to be fed to the neural network. What's next? Hidden layers.

3.1.2 Hidden Layers

As discussed in the previous chapter, the neural network can have one or more hidden layers (technically, as many as you want). Each layer has one or more neurons (also technically as many as you want). Your main job, as a neural network engineer, is to design these layers. For the sake of this example, let's say we decided to arbitrarily design the network to have 2

hidden layers, each have 512 nodes. And don't forget to add the ReLU activation function for each hidden layer.

Note on activation functions

In chapter 2, we discussed the different types of activation functions in details. As a deep learning engineer, you will often have a lot of different choices when you are building your network. Choosing the activation function that is the most suitable to the problem you are solving is one of these choices. While there is no one best answer that fits all problems, in most cases, the Relu function perform best in the hidden layers and Softmax is generally a good choice in the output layer for most classification problems when classes are mutually exclusive. The Softmax function gives us the probability that the input image depicts one of the (n) classes.

Same as the previous chapter, let's add two Dense layers using Keras. Dense layer is another name for Fully Connected layer (FC).

```
# import Dense Layer
from keras.layers import Dense

# add two Dense Layers with 512 nodes each
model.add(Dense(512, activation = 'relu'))
model.add(Dense(512, activation = 'relu'))
```

3.1.3 Output Layer

The output layer is pretty straight forward. In classification problems, the number of nodes in the output layer should be equal to the number of classes that you are trying to classify. In this problem, we are classifying 10 digits (0, 1, 2, 3, 4, 5, 6, 7 ,8 , 9). Then we will need to add one last Dense layer that contains 10 nodes.

```
# add one output Dense Layer with 10 nodes
model.add(Dense(10, activation = 'softmax'))
```

3.1.4 Putting it all together

When we put all these layers together we get the following neural network:

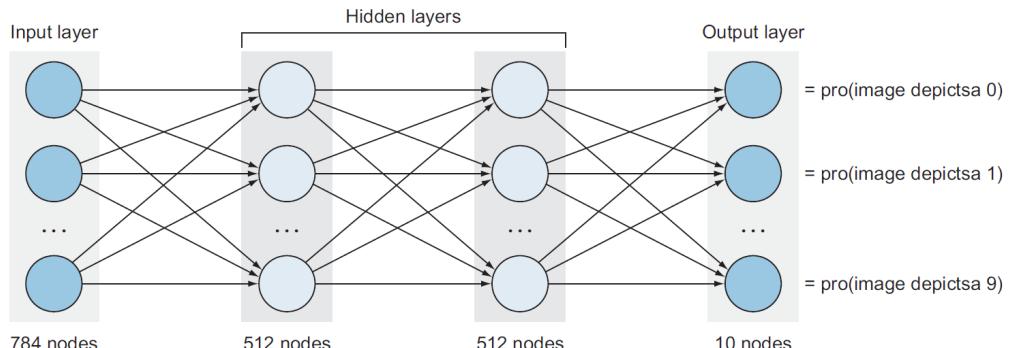


Figure 3.5

In code:

```
# same as before, we start with importing keras library
from keras.models import Sequential

# here we import a layer called Flatten to convert the image matrix
into a vector
from keras.layers import Flatten, Dense

# define the neural network architecture
model = Sequential()

# add the flatten layer
model.add( Flatten(input_shape = (28,28) ) )

# add two hidden layers with 512 nodes each
model.add(Dense(512, activation = 'relu')) ←—————
model.add(Dense(512, activation = 'relu'))

# add one output Dense layer with 10 nodes
model.add(Dense(10, activation = 'softmax')) ←—————
# print a summary of the model architecture
model.summary()
```

A it is recommended to use
relu activation func in the
hidden layers

B it is recommended to use
softmax act func in the
output layer for multi-
class classification
problems

When you run this code, you will see the model summary printed as follows:

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_1 (Dense)	(None, 512)	401920
dense_2 (Dense)	(None, 512)	262656
dense_3 (Dense)	(None, 10)	5130
<hr/>		
Total params:	669,706	
Trainable params:	669,706	
Non-trainable params:	0	

Figure 3.6

You can see the output of the flatten layer is a vector with 784 nodes as discussed before since we have 784 pixels in the 28x28 images. And as designed, the hidden layers produce 512 nodes each and finally the output layer (dense_3) produces a layer with 10 nodes.

The *Param #* field represents the number of parameters (weights) that are produced at each layer. These are the weights that will be adjusted and learned during the training process. They are calculated as follows:

- Params after the flatten layer is 0 because this layer only flattens the image to a vector to be fed to the input layer. The weights haven't been added yet.
- params after layer1 = 784 nodes in input layer X 512 in hidden layer 1 +512 connections to biases = 401,920*
- params after layer2 = 512 nodes in hidden layer 1 X 512 in hidden layer 2 +512 connections to biases =262,656*
- params after layer3= 512 nodes in hidden layer 2 X 10 in output layer +10 connections to biases =5,130*
- total params in the network = 401,920 + 262,656 + 5130 =669,706*

That said, this means that in this tiny network, we have a total of 669,706 parameters (weights and biases) that the network needs to learn and tune their values to optimize the error function. This is a huge number for such a small network. You can see how this number will grow out of control when we add more nodes and layers or when we use bigger images. This is a major drawback of MLPs that we will discuss next.

If you train the above MLP on the MNIST dataset you will get pretty good results (close to 96% accuracy compared to 99% with CNNs). But MLPs and CNNs do not usually yield comparable results. The MNIST dataset is very special because it is very clean and perfectly preprocessed. For example, all images have the same size and are centered in a

28x28 pixel grid. Also, the MNIST dataset contains only grayscale images. It would be a much harder task if the images are colored or digits were skewed a little or not centered.

If you try the above MLP architecture with a little more complex dataset like CIFAR-10 as we will be doing in the project at the end of this chapter. The network will perform very poorly (around 30 - 40% accuracy). Even worse with more complex datasets. In real-world messy image data, CNNs will truly SHINE over MLPs. This is because MLPs have two major drawbacks when processing images. We will discuss that next.

3.1.5 Drawbacks of MLPs in processing images

Alright, we are almost about to start talking about the topic of this chapter, CNNs. But, first let's discuss the two major problems in MLPs that the Convolutional Networks are designed to fix:

- 1) ***Flattening the image to a 1D vector input leads to losing the spatial features of 2D images***

As we seen in the mini-project above, before feeding the image to the hidden layers, we must flatten the image matrix to a 1D vector because MLPs take a flatten vector as an input. Which means throwing away all the 2D information contained in an image. Treating an input as a simple vector of number with no special structure might work well for 1D signals, like the housing price project in chapter 2. But in 2D images, it will lead to information loss because the network doesn't relate the pixel values to each other when trying to find patterns. MLPs have no knowledge of the fact that these pixel numbers were originally spatially arranged in a grid and that they are connected to each other. CNNs on the other hand, do not require flattening the image. We can feed the raw image matrix of pixels to our network. This will allow the CNN to understand that pixels that are close to each other are heavily related than the pixels that are far apart.

Let's oversimplify things to learn more about the importance of spatial features in an image. Suppose we are trying to teach our neural network to identify the shape of a square. In the image below, consider the pixel values 1 is white and 0 is black. Then when we draw a white square on a black background, the matrix will look like this:

1	1	0	0
1	1	0	0
0	0	0	0
0	0	0	0

Figure 3.7

Since MLPs take 1D vector as an input, then in order to feed the image to the MLP we have to flatten the 2D image to a 1D vector. The input vector of the above image will look like this:

Input vector = [1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

When the training is complete, the network will learn to identify a square *only* when the input nodes x_1, x_2, x_5 , and x_6 are fired. But what happens when we have new images with square shapes but located in different areas in the image?

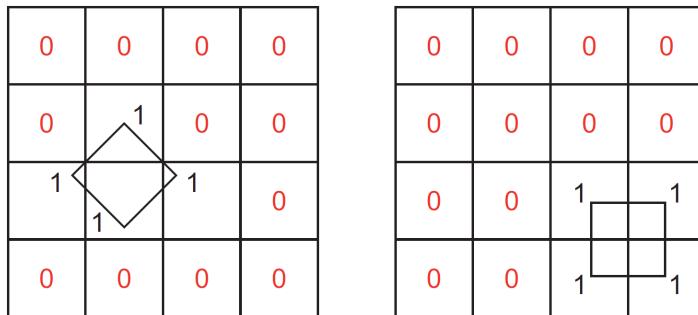


Figure 3.8

The MLP will have no idea that these are the shapes of squares because the network didn't learn the square shape as a feature. Instead, it learned the input nodes that when fired might lead to a square picture. With that said, in order to teach squares to our network we need a lot of square images located everywhere in the image. You can see how this solution won't scale for complex problems because in order to teach squares to our network we need to feed it images of squares located everywhere in the picture.

Another example of feature learning: if we want to teach the network to recognize cats, ideally we want the neural network to learn the full shape of the cat features regardless to where they appear on the image (ears, nose, eyes, etc.). This only happens when the network looks at the image as a set of pixels that when close to each other they are heavily related.

The mechanism of how CNNs learn will be explained in details in this chapter. But the diagram below shows how the network learns features throughout its layers.

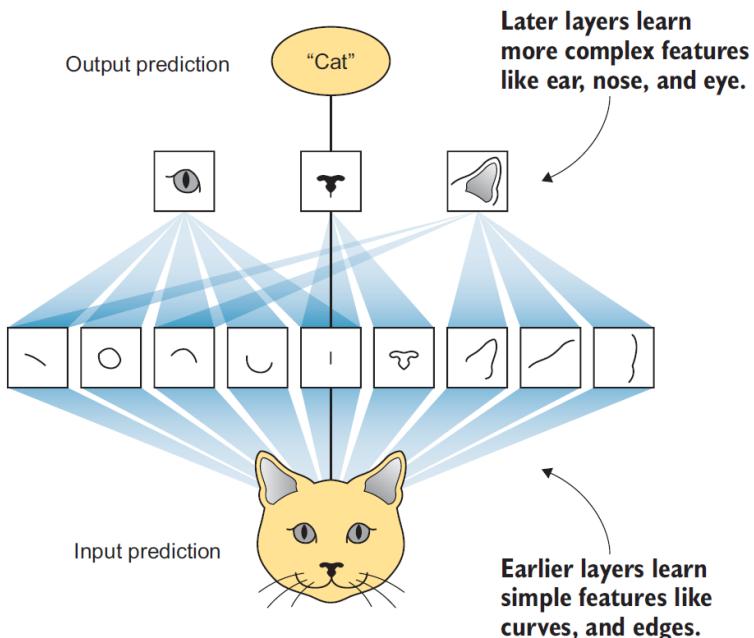


Figure 3.9

2) **Fully connected (dense) layers**

MLPs are composed of Dense layers that are fully connected to each other. Fully connected means that every node in one layer is connected to ALL nodes from the previous layer and all nodes in the next layer. In this scenario, each neuron will have parameters (weights) to train per each neuron from the previous layer. While this was not a big problem for the MNIST dataset because the images are really small in size (28x28), but what happens when we are trying to process bigger images? For example, if we have an image with dimensions = (1,000 x 1,000), it will yield a million parameters for each single node in the first hidden layer. So if the first hidden layer has 1,000 neurons, this will yield a 1 billion parameters in such a small network. You can imagine the computational complexity to optimize 1 billion parameters after only the first layer. This number will drastically increase when we have tens or hundreds of layers. This could get out of control pretty fast and will not scale.

CNNs on the other hand are *locally connected* layers. Where there nodes are connected to only a small subset of the previous layers' nodes. Locally connected layers use far fewer parameters than a densely connected layer as we will see when we discuss CNNs.

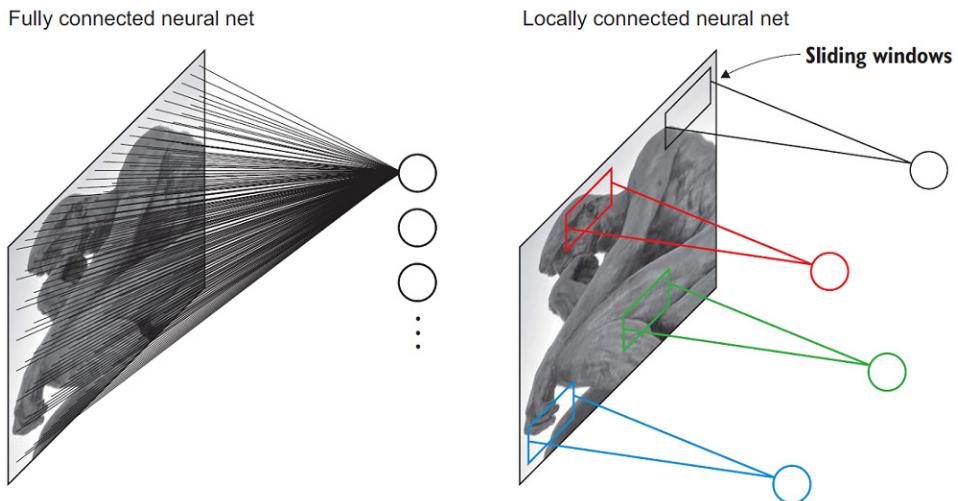


Figure 3.10

3.1.6 Conclusion

The loss of information caused by flattening the 2D image matrix to a 1D vector and the computational complexity of fully connected layers with larger images suggest that we need an entirely new way of processing the image input where the 2D information is not entirely lost. This is where Convolutional Networks come in. CNNs accept the full image matrix as an input which helps the network significantly understand the pattern contained in the pixel values.

3.2 CNNs Architecture

3.2.1 The big picture

Just like how regular neural networks contain multiple layers which allow each layer to find successively complex features, CNNs work the same way. The first layer of convolutions learns some basic features (edges and lines), then the next layer will learn features that are a little more complex (circles, squares, etc), and then the following layer will find even more complex features (like parts of the face, car wheel, dog whiskers, etc.) and so on. You will see this demonstrated real soon. For now, know that the CNN architecture will follow the same pattern we started with neural networks: we will stack neurons in hidden layers on top of each other, we will have weights that will be randomly initiated and learned during the network training, we will apply activation functions, calculate error ($y - \hat{y}$) then backpropagate the error to update the weights. All the same. The difference is that we will use convolutional layers instead of regular fully connected layers for the feature learning piece.

Before we look into the CNN architecture, let's back up for a moment to see the big picture. Remember the image classification pipeline we discussed in chapter 1?

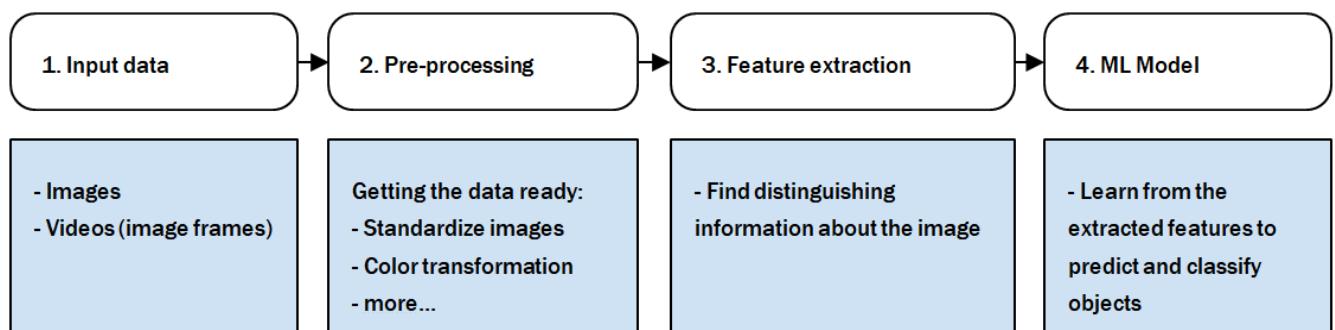


Figure 3.11

Before deep learning, we used to manually extract features from the images then feed the features vector to a classifier (regular machine learning algorithm like SVM) to classify them. With the magic that neural networks are providing us, we are going to replace the manual work of step 3 with a neural network (MLPs or CNNs) that is going to do both the feature learning and classification for us (steps 3 and 4).

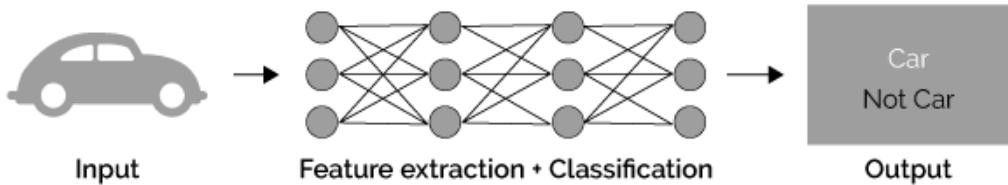


Figure 3.12

We saw earlier in the digits classification project how to use MLP to learn features and classify the image (steps 3 and 4 together). It turns out that our issue with the fully connected layers is not the classification part. In fact, FC layers do a great job in that part. Our issue was in the way FC layers process the image to learn features. So we are going to get a little creative. So, let's keep what's working and make modifications to what is not working. The FC layers are not doing a great job in feature extraction (step 3). So, let's replace that with locally connected layers (convolutional layers). On the other hand, FC layers do great job classifying the extracted features (step 4). So, let's keep them for the classification part.

That said, the high level architecture of CNNs will look like this:

- Input layer
- Convolutional layers for feature extraction
- Fully connected layers for classification
- Output prediction

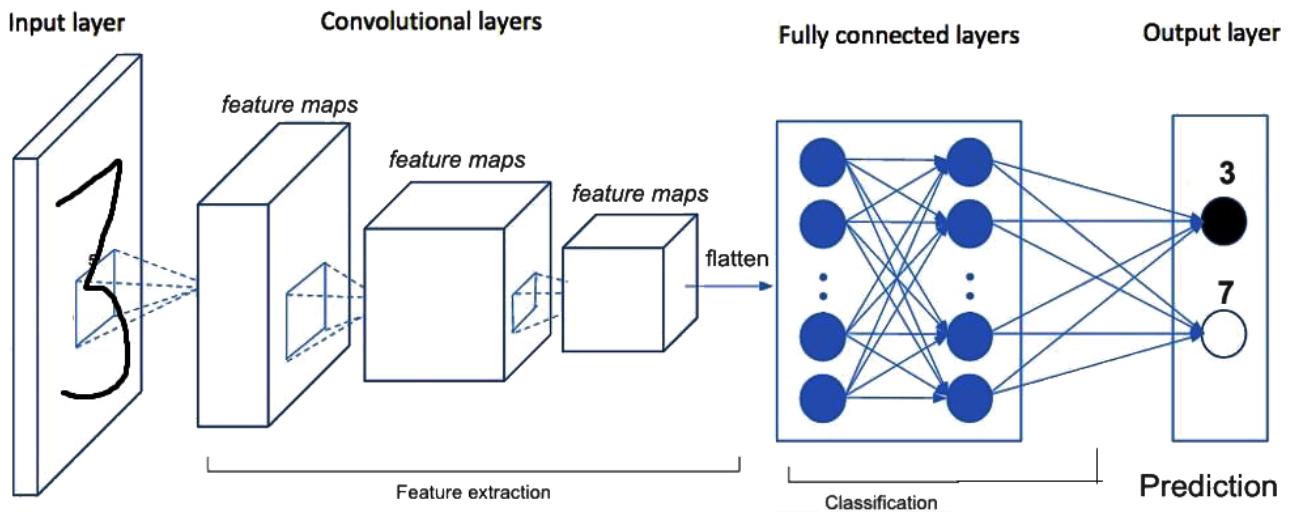


Figure 3.13

Remember, we are still talking about the big picture here. We will dive into each of these components soon. For the diagram above, suppose we are building a CNN to classify between

images of two classes: the number 3 and 7. Look at the diagram above and follow along with these steps:

- 1) First we feed the raw image to the convolutional layers
- 2) The image passes through the CNN layers to detect patterns and extract features called *feature maps*. The output of this step is then flattened to a vector of the learned features of the image

(**Notice** that the image dimensions shrink after each layer and the number of feature maps (layer depth) increases until we have a long array of small features in the last layer of the feature extraction part. Conceptually, you can think of this step as the neural network learning to represent more abstract features of the original image.)

- 3) The flattened features vector is then fed to the Fully Connected (FC) layers to classify the extracted features of the image
- 4) The neural network then fires the node that represents the correct prediction of the image. Note that in this example we are classifying two classes (3 and 7), then the output layer will have two nodes. One to represent the digit 3 and one for 7.

What is feature maps?

The basic idea of neural networks is that neurons learn features from the input. In CNNs, the feature map is the output of one filter applied to the previous layer. It is called a feature map because it is a mapping of where a certain kind of feature is found in the image. Convolutional Neural Networks look for "features" such as straight lines, edges, or even objects. Whenever they spot these features they report them to the feature map. Each feature map is looking for something else. One feature map could be looking for straight lines, the other for curves.

3.2.2 A closer look on feature extraction

You can think of the feature extraction step as breaking large images into smaller pieces of features and stacking them into a vector. For example, an image of the digit "3" is one image (so depth = 1) will be broken into smaller images that contain specific features of the digit 3. If it is broken into 4 features, then the depth = 4. So, as the image passes through the CNN layers, it shrinks in dimensions and the layer gets deeper because it contains more "images" of small features.

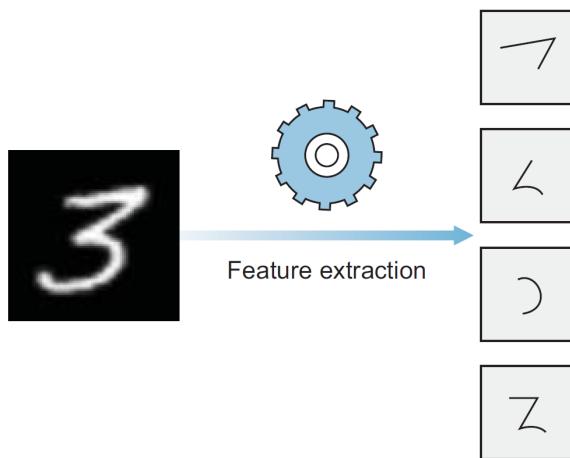


Figure 3.14

Note that this is just a metaphor to help visualize the feature extraction process. In reality, CNNs don't literally break the image into pieces. Instead, they *extract meaningful features* that separate this object from other images in the training set and stack them in an array of features.

3.2.3 A closer look on classification

After feature extraction is complete, we add fully connected layers (regular MLP) to look at the features vector and say: "the first feature (top) has what looks like an edge, this could be 3, or 7, or maybe an ugly two? I'm not sure, let's look at the second feature. Hmm, this is definitely not a 7 because it has a curve" and so on until it is confident that this is an image of 3.

It is important to callout that the CNN doesn't go from the image input to the features vector directly in one layer. This usually happens in 10s and 100s of layers as we will see later in this chapter. The feature learning process happens step-by-step after each hidden layer. So, the first layer usually learns very basic features like lines and edges, then the second will assemble those lines together to be able to recognize shapes, corners and circles. Then in the deeper layers of the network, it learns more complex shapes such as human hands, eyes, ears, etc. For example, here is a simplified version of how CNNs learn faces:

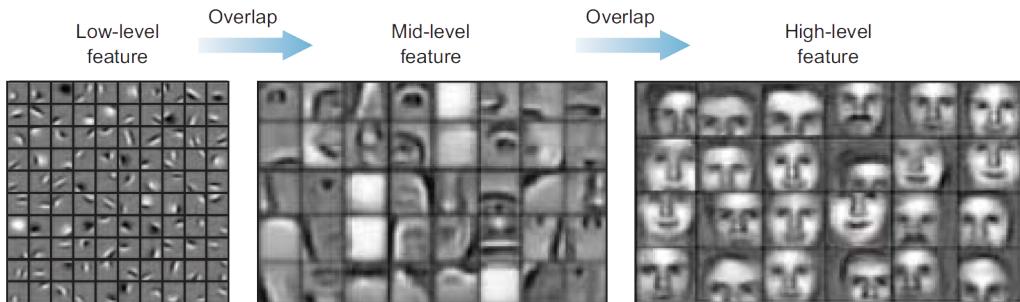


Figure 3.15

You can see that the early layers detect patterns in the image to learn low level features like edges and the later layers detect *patterns within patterns* to learn more complex features like parts of the face and then *patterns within pattern within patterns* and so on. This concept will come in handy when we discuss more advanced CNN architectures in later chapters. For now, know that, in neural networks, we stack hidden layers to learn patterns from each other until we have an array of meaningful features to identify the image.

Input Image

- + Layer 1 => *patterns*
- + Layer 2 => *patterns with patterns*
- + Layer 3 => *patterns within patterns within patterns*
- ... and so on.

3.3 Basic components of the CNN

Without and further ado, let's discuss the main components of the CNNs architecture. There are three main types of layers that you will see in almost every convolutional network:

1. Convolutional layer [CONV]
2. Pooling layer [POOL]
3. Fully connected layer [FC]

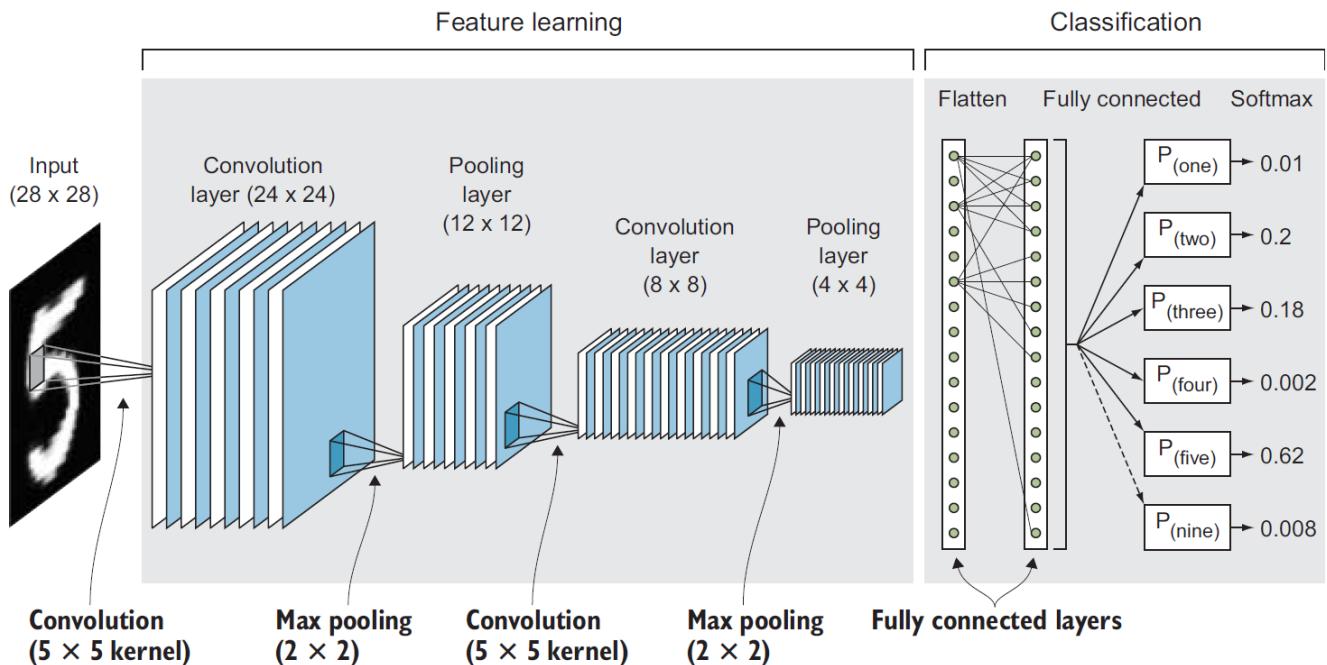


Figure 3.16

The text representation of the above architecture goes like this:

CNN architecture: INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => SOFTMAX

Note that RELU and SOFTMAX activation functions are not really standalone layers, they are the activation functions used in the previous layer. The reason they are represented this way in the text representation is to call out that the CNN designer is using ReLU activation function in the CONV layers and softmax activation in the FC layer. So the above representation represents a CNN architecture that contains 2 CONV Layers + 1 FC layer. You can add as many CONV and FC layers as you see fit. The CONV layers are for the feature learning or extraction and the FC layers are for the classification.

Now that we saw the full architecture of a convolutional network, let's dive deeper into each of the layer types separately to get a deeper intuition on how they work. Then at the end of this section, we will put them all back together.

3.3.1 Convolutional layers (CONV)

Convolutional layer is the core building block of a Convolutional Neural Network. Convolution layers act like a feature finder window that slides over the image pixel-by-pixel to extract meaningful features that identify the objects in the image.

What is convolution?

In mathematics, convolution is the operation of two functions to produce a third modified function. To explain this definition in the CNNs context, the first function is the input image and the second function is the CONV filter. We will perform some mathematical operations to produce a modified image with new pixel values.

Let's zoom in to the first convolutional layer to see how it processes an image. By sliding the convolutional filter over the input image, the network is breaking the image into little chunks and it processes those chunks individually to assemble the modified image that is called *feature map*.

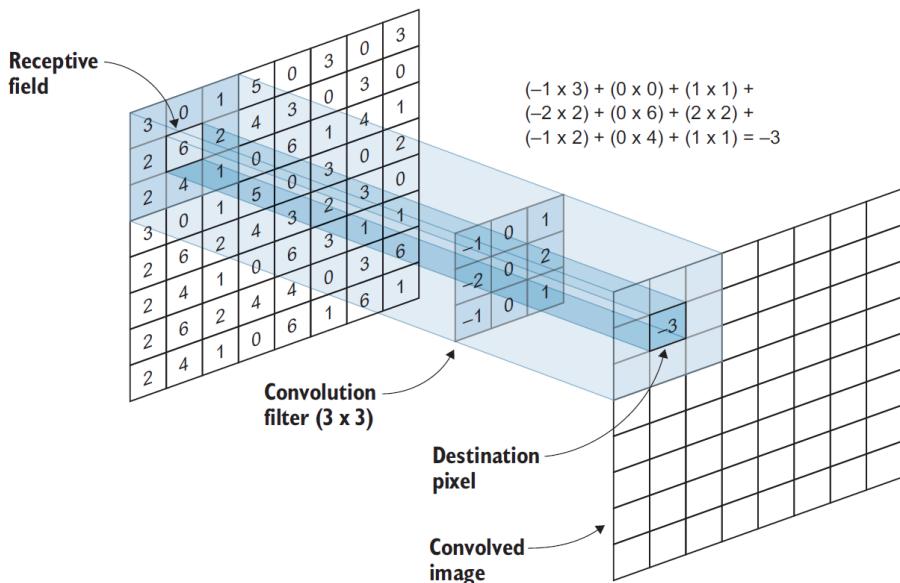


Figure 3.17

Now that we have this diagram, let me list some facts about convolution filters:

- The small 3×3 matrix in the middle *is* the convolution filter. They are also called *Kernels*.
- The kernel slides over the original image pixel-by-pixel and does some math calculations to get the values of the new “convolved” image at the next layer. The area of the image that the filter convolves is called the *receptive field*.

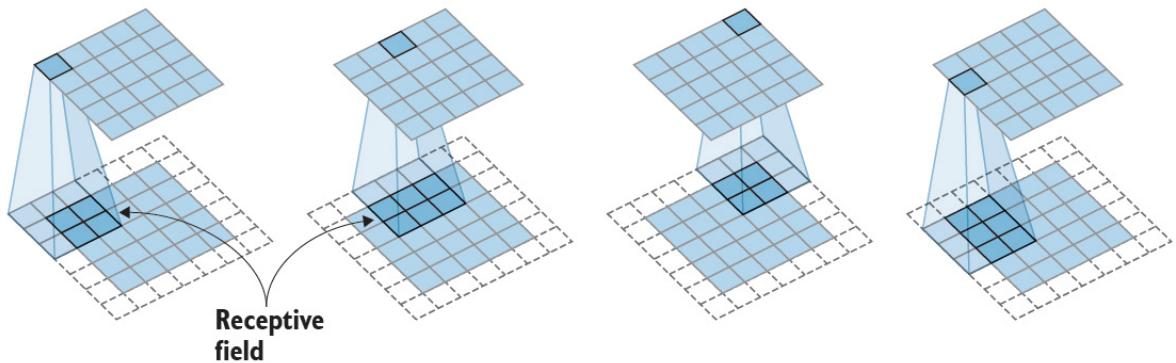


Figure 3.18

What are the Kernel values?

Remember the connection weights in the MLP from chapter 2? In CNNs, this convolution matrix *is* the weights. This means that they are also *randomly initialized* and the values are *learned* by the network (so you will not have to worry about assigning its values).

CONVOLUTIONAL OPERATION

The math goes like this: same as in MLP, remember we used to multiply the input by the weights and sum them all together to get the weighted sum?

$$\text{weighted sum} = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \dots + x_n \cdot w_n + b$$

We do the same thing, except that in CNNs the neurons and the weights are structured in a matrix shape. So, we multiply each pixel in the receptive field by the corresponding pixel in the convolution filter and sum them all together to get the value of the center pixel in the new image. This is the same matrix dot product we did in the previous chapter.

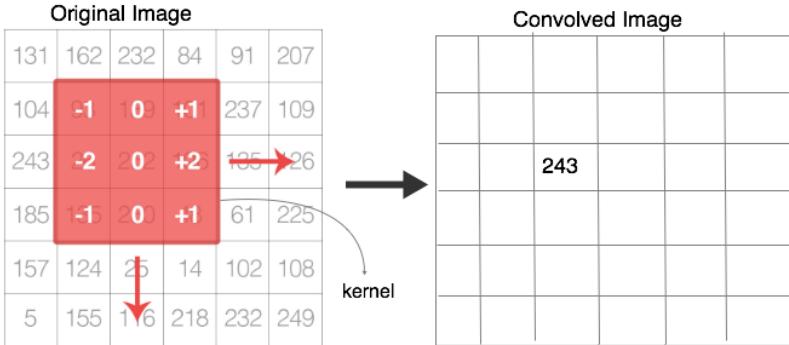


Figure 3.19

$$(93 \times -1) + (139 \times 0) + (101 \times 1) + (26 \times -2) + (252 \times 0) + (196 \times 2) + (135 \times -1) + (240 \times 0) + (48 \times 1) = 243$$

The filter (or kernel) slides over the whole image and everytime we multiply every corresponding pixel element-wise then add them all together to create a new image with new pixel values. This convolved image is called a feature map or activation map.

Applying filters to learn features

0	-1	0
-1	4	-1
0	-1	0

Figure 3.20

Let's not lose focus of the initial goal. We are doing all this so that the network extracts features from the image. How does applying filters fall into this goal? In image processing, filters are used to filter out unwanted information or amplify features in an image. These filters are matrices of numbers that convolve with the input image to modify it. Let's look at this edge detection filter.

When this Kernel (K) is convolved with the input image $F(x,y)$, it creates a new convolved image amplifying the edges. Also known as *feature map*:

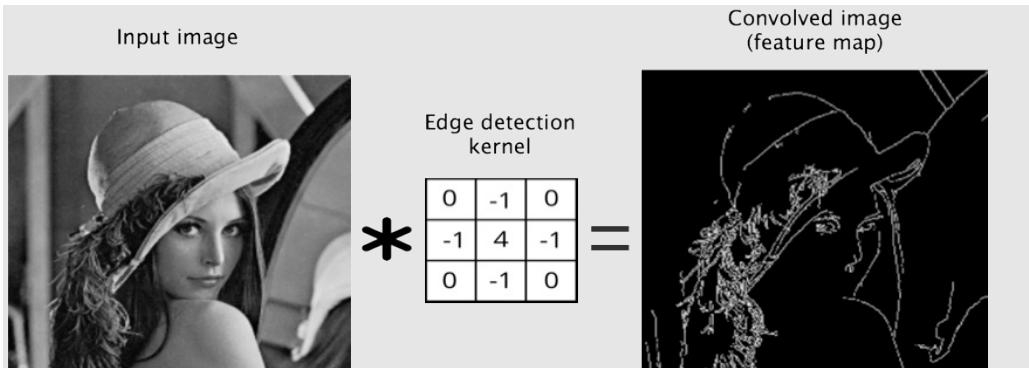


Figure 3.21

To understand how the convolution happens, let's zoom into a small piece of the image:

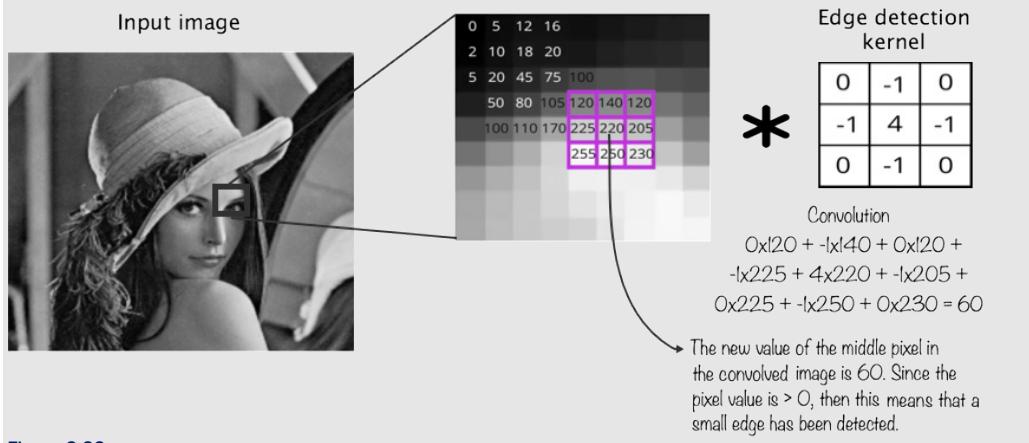


Figure 3.22

The image above shows the convolution calculations in one area of the image to compute the value of one pixel. We compute the values of all the pixels by sliding the kernel over the input image pixel-by-pixel and apply the same convolution process.

These kernels are often called weights because they determine how important a pixel is in forming a new output image. Similar to what we discussed before in MLP about weights, the weights represent the importance of the feature on the output. In images, the input features are the pixel values.

Other filters can be applied to detect different types of features. For example, some filters detect horizontal edges, others detect vertical edges, some other filters detect more complex shapes like corners and so on. The point is, these filters when applied in the convolutional layers, yield the feature learning behavior that we discussed earlier: first they learn simple features like edges and straight lines then later layers learn more complex features.

We are basically done with the filters concept now. This is all what it is!

Now, let's take a look at the CONV Layer as a whole: Each convolutional layer contains 1 or more convolutional filters. The number of filters in each CONV layers determines the depth of

the next layer because each filter produces its own feature map (convolved image). Let's look at the CONV layers in Keras to see how they work:

```
model.add(Conv2D(filters=16, kernel_size=2, stride='1', padding='same', activation='relu'))
```

And there you have it. One line of code creates the CONV layer. We will see where this line fits in the full code later in this chapter. Let's stay focused on the CONV layer. As you can see from the code above, the CONV layer takes 5 main arguments. As we mentioned in the deep learning chapter, it is recommended that we use the ReLU activation function in the neural networks hidden layers. That's one argument out of the way. Now, let's explain the remaining four hyperparameters that control the size and depth of the output volume:

- 1) Filters = the number of convolutional filters in each layer. This represents the depth of its output
- 2) Kernel size = the size of the convolutional filter matrix. Sizes vary from 2x2, 3x3, 5x5
- 3) Stride and 4) Padding = we will explain these two together in the next section

Let's look at each one of these hyperparameters:

As we learned in the deep learning chapter, hyperparameters are the knobs that you will tune (increase and decrease) when configuring your neural network to improve performance.

1) ***Number of filters in the CONV Layer***

Each CONV layer has one or more filters. To understand this, let's back up to MLPs from chapter 2. Remember how we stacked neurons in hidden layers, and each hidden layer has n number of neurons called hidden units? Here is the MLP diagram from chapter 2:

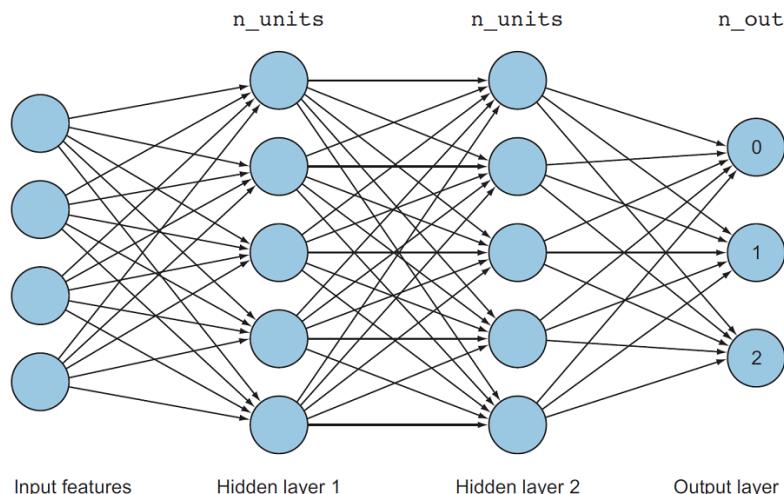


Figure 3.23

Same thing for CNNs, the CONV layers are the hidden layers. And to increase the number of neurons in hidden layers, we increase the number of kernels in CONV layers. Each Kernel unit is considered a neuron. For example, if we have a 3×3 kernel in the CONV layer, this means we have 9 hidden units in this layer. When we add another 3×3 kernel we will have 18 hidden units. And another one to be 27 and so on. So, by increasing the number of kernels in a CONV layer, we are increasing the number of hidden units which make our network more complex to detect more complex patterns. Similarly when we added more neurons (hidden units) to the hidden layers in the MLP. In the figure below, you will see a representation of the CNN layers that shows the number of kernels idea more clearly.

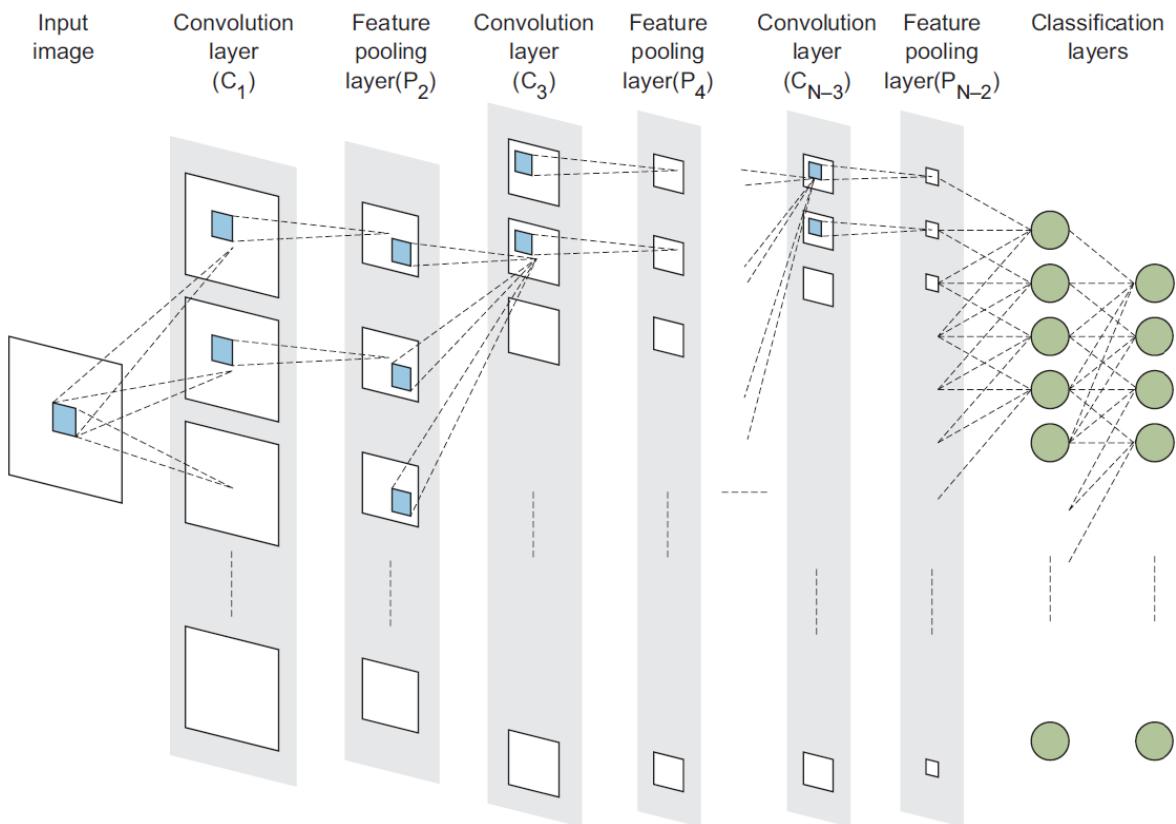


Figure 3.24

Kernel Size

As we said before, the convolution filter is also known as *kernel*. It is a matrix of weights that slides over the image to extract features. The Kernel size here refers to the dimensions of the CONV filter (width x height).

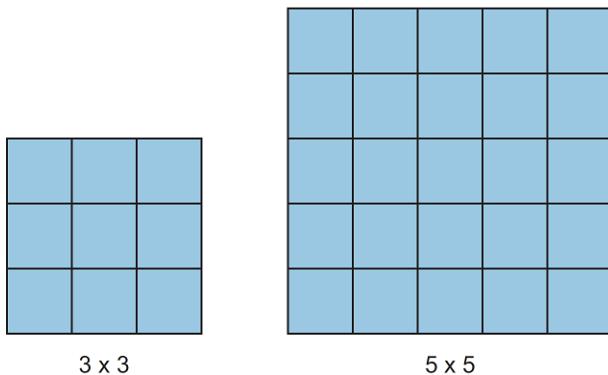


Figure 3.25

Kernel_size is one of the hyperparameters that you will be setting when building a convolutional layer. Like most neural networks hyperparameters, there is no one good answer that fits all problems. The intuition is: smaller filters will capture very fine details of the image and bigger filters will leave out minute details in the image.

Remember that these filters contain the weights that will be learned by the network. So, theoretically, the bigger the *kernel_size* is → the deeper the network → the better it learns. However, this comes with higher computational complexity and might lead to overfitting.

The kernel filters are almost always square and will range from the size of 2x2 at the smallest and 5x5 at the largest. Theoretically you can use bigger filters but not preferred because this will lead to losing important details of the image.

I don't want you to get overwhelmed with all the hyperparameters tuning. Deep learning is really an art as well as it is science. I can't emphasize this enough, most of your work as a deep learning engineer will not be spent on building the actual algorithms as it is building your network architecture and setting, experimenting, and tuning your hyperparameters. There is a lot of research goes into trying to find the optimal topologies and parameters for a CNN given a type of problem. Fortunately, the problem of tuning the hyperparameters doesn't have to be as hard as it might seem to be. Throughout the book, I will point out the good starting points to these hyperparameters and teach you the intuition to evaluate your model and analyze its results to know which knob (hyperparameter) you need to tune (increase or decrease).

STRIDE AND PADDING

These two hyperparameters you will usually think of them both together because they both control the shape of the output of this CONV layer. Let's see how:

- 1) Stride: is the amount by which the filter slides over the image. For example, to slide the CONV filter one pixel at a time, then the stride = 1. If we want to jump two pixels at a time, then stride = 2. Strides of 3 or more are uncommon and rare in practice. Jumping pixels will produce smaller output volumes spatially.

A Stride of 1 will make the output image roughly the same height and width of the input image. While a stride of 2 will make the output image roughly about half of the input image size. I say roughly because it depends on what you set the padding parameter to do with the edge of the image.

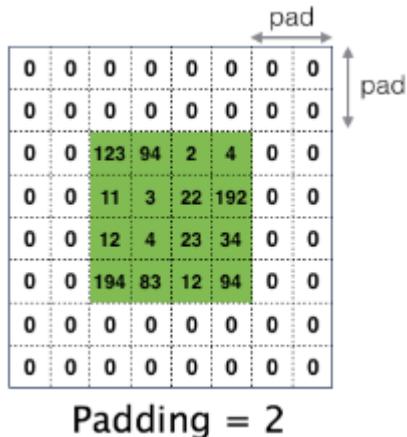


Figure 3.26

- 2) Padding: often called zero-padding because we add zeros around the border of an image. Padding is most commonly used to allow us to preserve the spatial size of the input volume so the input and output width and height are the same. This way we can use CONV layers without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as you go to deeper layers.

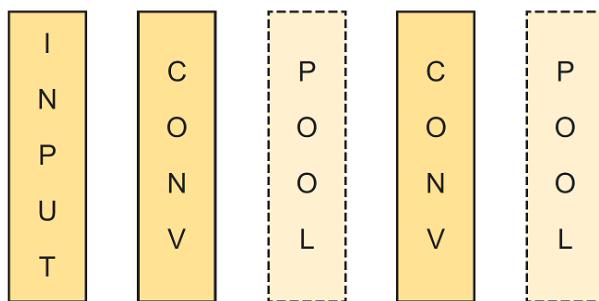
The idea of stride and padding hyperparameters is we should be able to do one of two things: 1) keep all the important details of the image and transfer them to the next layer (when stride = 1 and padding = same), or 2) ignore some of the spatial information of the image to make the processing computationally more affordable. Note that we will be adding the POOL layer (discussed next) to reduce the size of the image anyways to focus on the extracted features. Let's revisit this again after we understand the pooling layers. But for now, know that stride and padding hyperparameters are

meant to control the behavior of the CONV layer and the size of its output on whether to pass on all of the image details or ignore some of them.

3.3.2 Pooling layers or subsampling (POOL)

Adding more CONV layers increases the depth of the output layer which leads to the increase of the number of parameters that the network needs to optimize (learn). You can see that after adding several CONV layers (usually tens or even hundreds) this will produce a huge number of parameters (weights). This increase in the network dimensionality will increase the time and space complexity of the mathematical operations that take place in the learning process. This is when the pooling layers come in. Subsampling or pooling helps in reducing the size of the network by reducing the number of parameters passed to the next layer. The pooling operation resizes its input by applying a summary statistic function, such as a maximum or average, to reduce the overall number of parameters to be passed on to the next layer.

The goal of the pooling layer is to downsample the feature maps produced by the CONV layer into smaller number of parameters to reduce the computational complexity. It is a common practice to add POOL layers after each one or two CONV layers in the CNN architecture.



INPUT => CONV => POOL => CONV => POOL

Figure 3.27

MAX POOLING VS AVERAGE POOLING

There are two main types of pooling layers. The max-pooling and average-pool.

1) **Max pooling:**

Similar to convolutional kernels, pooling kernels are windows with a certain size that slide over the image with a stride value. The difference is that they don't have weights or any values. All they do is slide over the feature map created by the previous CONV layer and select the max pixel value to pass along to the next layer and ignore the remaining values. Look at the example below, we created a pooling filter with a size of

2×2 and a stride = 2 (as in jumps 2 pixels when sliding over the image). This pooling layer reduced the feature map size from 4×4 to 2×2 .

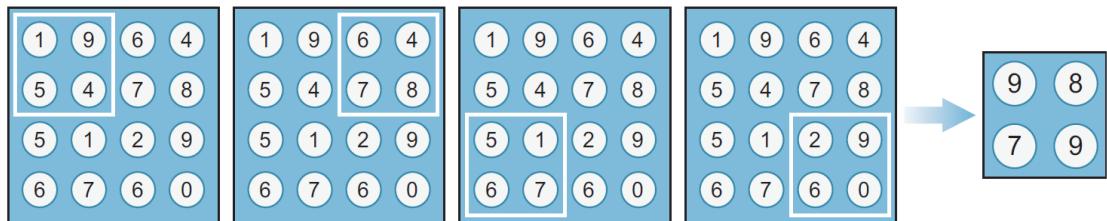


Figure 3.28

When we do that to all feature maps in the CONV layer, we get a smaller maps in dimensions (width x height) but the depth of the layer is kept the same because we apply the pooling filter to each one of the feature maps from the previous filter. So if the CONV layer had 3 feature maps, the output of the pooling layer will also have 3 feature maps but of shrunk sizes.

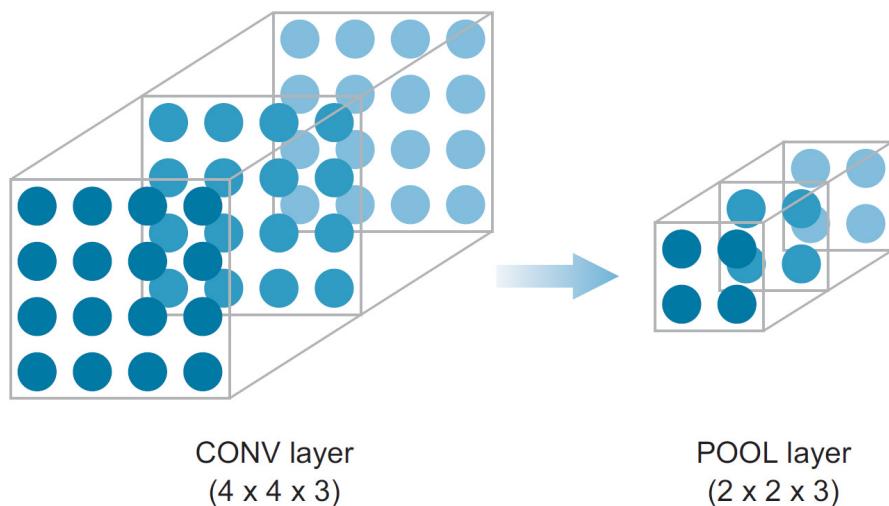


Figure 3.29

- 2) **Global average pooling:** This is a more extreme type of dimensionality reduction. Instead of setting a window size and stride, average pooling calculates the average values of all pixels in the feature map.

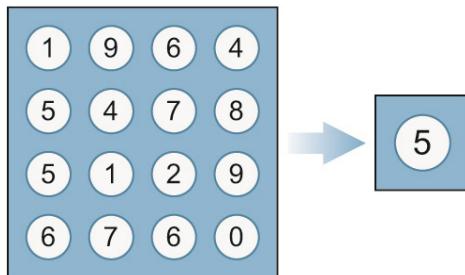


Figure 3.30

You can see that the global average pooling layer takes a 3D array and turns it into a vector

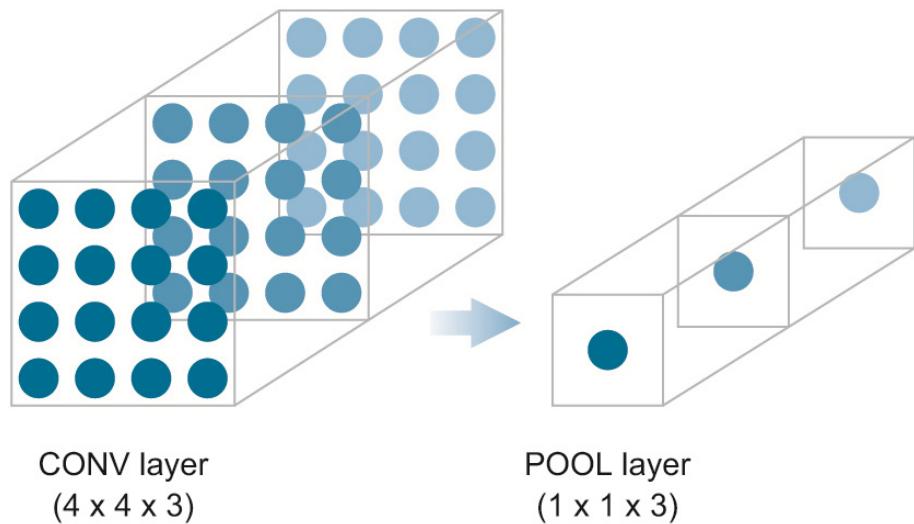


Figure 3.31

3.3.3 Why use a pooling layer?

As you can see from the examples above, the pooling layers reduce the dimensionality of our convolutional layers. The reason it is important to reduce the dimensionality is that in complex projects, CNNs will contain many CONV layers and each have tens or hundreds of convolutional filters (kernels). Since the kernel contain the parameters (weights) that the network learns, this can get out of control very quickly and the dimensionality of our CONV layers can get very large. So adding POOL layers will help us keep the important features and pass them along to the next layer while shrinking the image dimensionality. Think of pooling layers as the image compressing programs. They reduce the image resolution while keeping its important features.



Figure 3.32

Pooling vs strides and padding

The main purpose of pooling and strides is to reduce the number of parameters in the neural network. The more parameters we have, the more computationally expensive the training process will be. Many people dislike the pooling operation and think that we can get away without it in the favor of tuning the strides and padding of the CONV layer. For example, [Striving for Simplicity: The All Convolutional Net](#) paper proposes to discard the pooling layer in favor of architecture that only consists of repeated CONV layers. To reduce the size of the representation they suggest using larger stride in CONV layer once in a while. Discarding pooling layers has also been found to be important in training good generative models, such as generative adversarial networks (GANs) that we will discuss later in chapter 10. It seems likely that future architectures will feature very few to no pooling layers. But for now, pooling layers are still widely used to downsample the images from one layer to another.

CONV + POOL LAYERS RECAP

Let's review what we did so far. Up until this point, we used a series of CONV and POOL layers to process the image and extract meaningful features that are specific to the images in the training dataset. To summarize how we get here:

- 1) The raw image is fed to the CONV layer: which is a set of kernel filters that slide over the image to extract features
- 2) The CONV layer has the following attributes that we need to configure:

```
model.add(Conv2D(filters=16, kernel_size=2, strides='1', padding='same',
activation='relu'))
```

- a. filters = number of kernel filters in each layer (the depth of the hidden layer)
- b. kernel_size = the size of the filter (aka kernel). Usually 2, or 3, or 5
- c. stride = is the amount by which the filter slides over the image. A stride of 1 or 2 are usually recommended as a good start
- d. padding = add columns and rows of zero values around the border of the image to reserve the image size in the next layer
- e. activation = it is strongly recommended to use "relu" activation function in the hidden layers

- 3) POOL layer has the following attributes that we need to configure:

```
model.add(MaxPooling2D(pool_size=(2, 2), strides = 2))
```

And we keep adding pairs of CONV + POOL layers to achieve the depth needed for our “deep” neural network.

An important closing note on CNNs architecture: I strongly recommend looking at existing architectures, since many people have already done the work of throwing things together and seeing what works. Practically speaking, unless you are working on research problems, you will find yourself starting from a CNN architecture that has already been built by other people to solve similar problems to yours. Then tune it further to fit your data. In chapter 4, we will explain how to diagnose your network performance and discuss tuning strategies to improve its performance. In chapter 5, we will discuss the most popular CNN architectures to give you an intuition of how other researchers built this network. What I want you to take from this section is to understand, conceptually, how the CNN is built and that more layers = more neurons = more learning behavior. But this comes with computational cost. So you should always consider the size and complexity of your training data (many layers may not be necessary for a simple task).

Visualize what happens after each layer

After the CONV layers: the image keeps its width and height dimensions (usually) but it gets deeper and deeper after each layer. Why? Remember the cutting-the-image-into-pieces-of-features analogy we mentioned earlier? This is what's happening after the CONV layer.

For example, suppose the input image is 28x28 (like in the MNIST dataset). When we add a CONV_1 layer (with filters = 4, stride = 1, padding = same), the output will be the same width and height dimensions but with depth = 4 (28x28x4).

Now we add CONV_2 layer with the same hyperparameters but more filters = 12, we get a deeper output = 28x28x12.

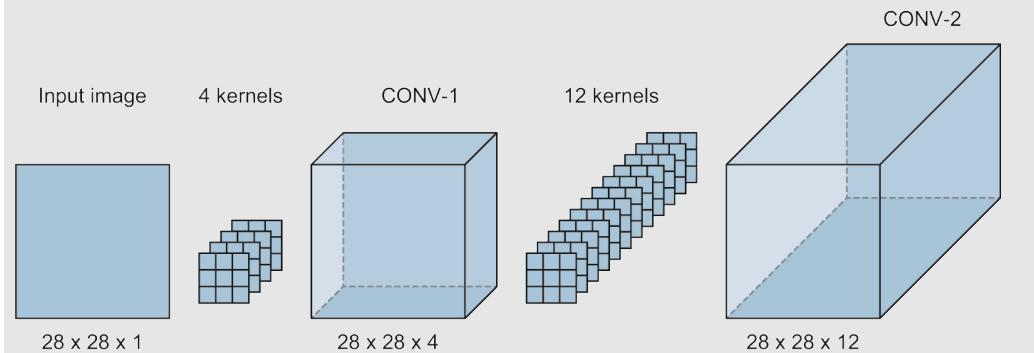


Figure 3.33

After the POOL layers: the image keeps its depths but shrinks in the width and height dimensions.

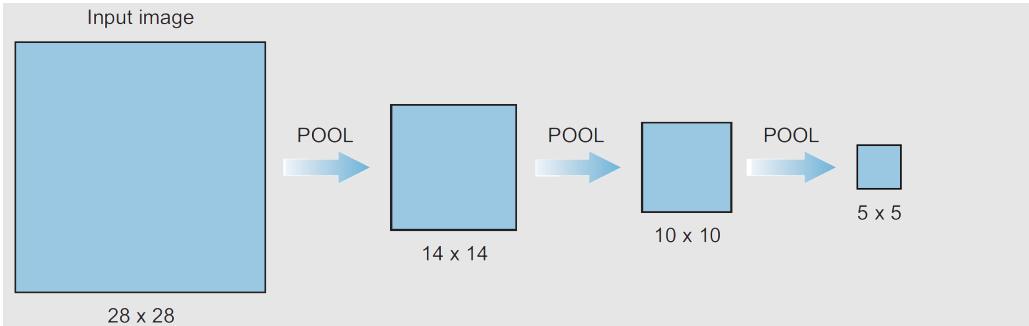


Figure 3.34

Putting the CONV and POOL together we get something like this:

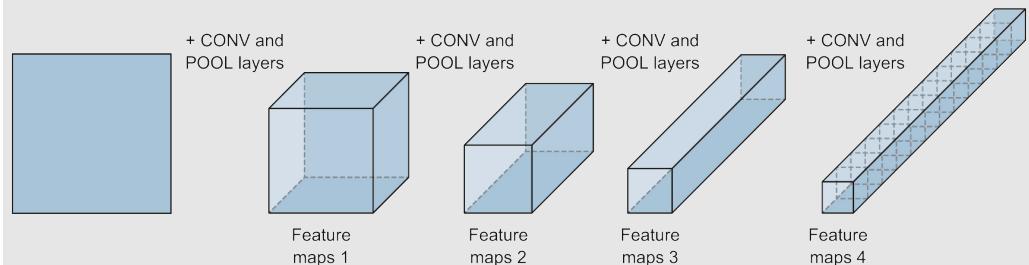


Figure 3.35

This keeps happening until we have at the end a long tube of small shaped images that contain all the features in the image.

The output of the CONV + POOL layers produces the features tube ($5 \times 5 \times 40$) that is almost ready to be classified. 40 here is just an example for the depth of the features tube. As in 40 feature maps. The last step is to flatten this tube to be ready to be fed to the fully connected layer for classification. As discussed earlier, the flattened layer will have the dimensions of $(1, m)$ where $m = 5 \times 5 \times 40 = 1,000$ neurons.

3.3.4 Fully connected layers (FC)

After passing the image through the feature learning process using the CONV+POOL layers, we now have extracted all the features from this image and put them all in the long tube of features. Now, it is time to look at the extracted features and classify the images based on the presence of these features in them. We are going to do that using the regular neural network architecture MLPs that we explained in chapter 2.

WHY USE FC LAYERS?

Well, MLPs work great in classification problems. The reason we used CONV layers earlier is that MLP lose a lot of valuable information when trying to extract features from the image because we have to flatten the image before feeding it to the network, whereas CONV layers can process raw images. Now that we have the features extracted, we can use regular MLPs to classify them.

We have already discussed the MLP architecture thoroughly in chapter 2. Nothing new here. To reiterate, here are the FC layers:

Input flattened vector: Look at the diagram below, to feed the features tube to the MLP for classification, we flatten it to a vector with the dimensions $(1, n)$. For example, if the features tube has the dimensions of $5 \times 5 \times 40$, the flattened vector will be $= (1, 1000)$.

Hidden layer (FC): we add one or more FC layers and each layer has 1 or more neurons (similarly to what we did when we built regular MLPs)

Output layer: As explained in the activation functions section in chapter 2, it is recommended to use “softmax activation function” for classification problems when we are classifying more than 2 classes. In this example, we are trying to classify digits from 0 to 9 \Rightarrow 10 classes. And the number of neurons in the output layers are equal to the number of classes. Then the output layer will have 10 nodes.

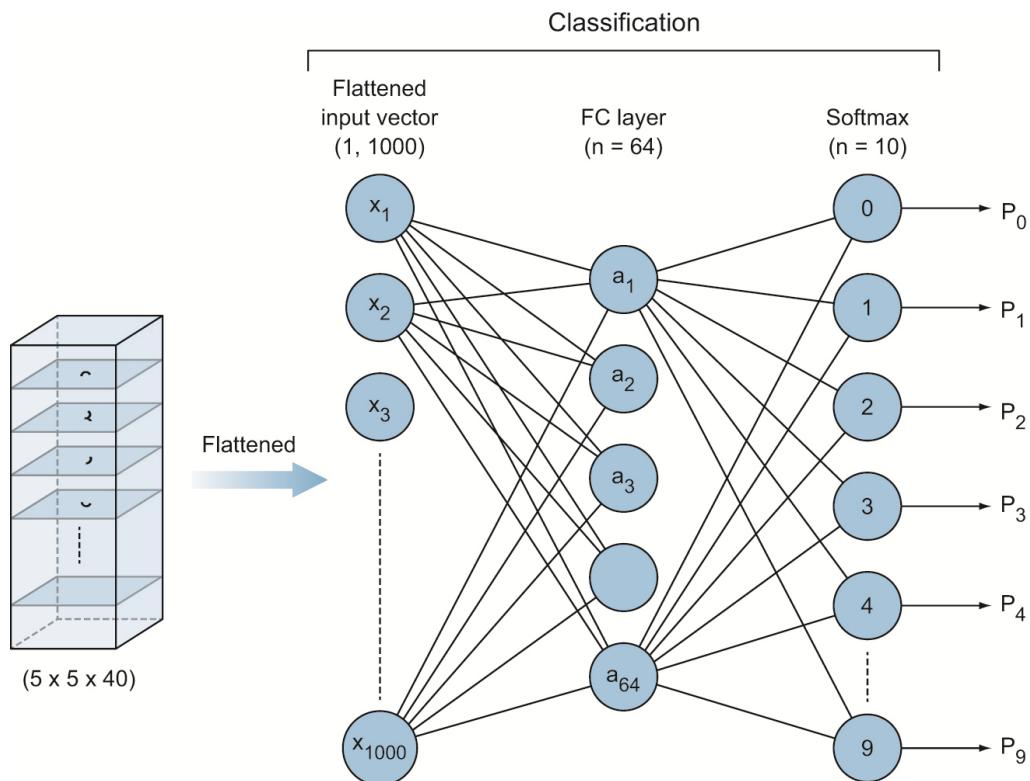


Figure 3.36

Remember that the Multi-Layer Perceptron MLP from chapter 2 are also called fully connected FC layers because all the nodes from one layer are connected to all the nodes in the previous and next layers. They are also called Dense layers. MLP, FC, Dense, and sometimes feedforward neural network are used interchangeably to refer the regular neural network architecture.

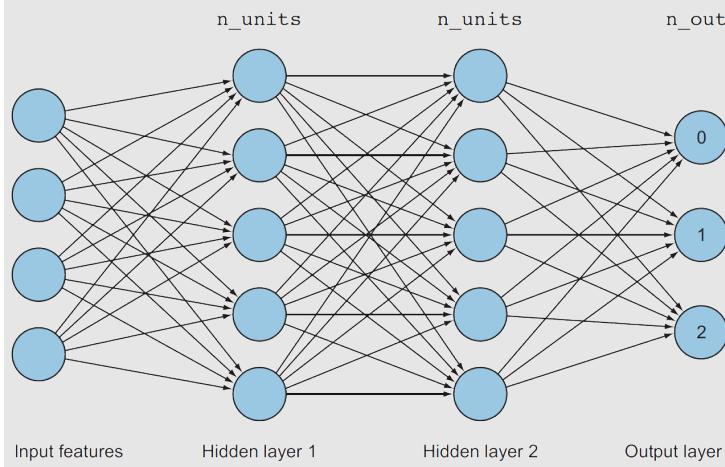


Figure 3.37

3.4 Image classification using CNNs

Okay, you are now fully equipped to build your own CNN model to classify images. For this mini-project, we will use the same MNIST dataset which is a simple problem but will help build the foundation to more complex problems in the following chapters. (MNIST dataset is like the hello world! for deep learning)

Note that regardless of which deep learning library you decide to use, the concept is pretty much the same. You basically start with designing the CNN architecture in your mind or on a piece of paper then start stacking layers on top of each other and setting their parameters. Both Keras and MXNet (along with Tensorflow, Pytorch, and other deep learning libraries) have their pros and cons that we will discuss later in this book. But the idea is pretty much the same. So for the rest of this book we will be mostly working with Keras with a little overview here and there on other libraries.

3.4.1 Build the model architecture

This is the part in your project where you define and build the CNN model architecture. To look at the full code of the project that includes the image preprocessing, training, and evaluating the model, go to the book's Github repo and open the `mnist_cnn` notebook or go to the book's website: www.computerVisionBook.com. At this point, we are more concerned with the model architecture piece of the code. At the end of this chapter, we will build an end-to-end image classifier and dive deeper into the other pieces.

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# build the model object
model = Sequential()

# CONV_1: add CONV layer with RELU activation and depth = 32 kernels
model.add(Conv2D(32, kernel_size=(3, 3), stride=1, padding='same', activation='relu',
                 input_shape=(28,28,1)))
# POOL_1: downsample the image to choose the best features
model.add(MaxPooling2D(pool_size=(2, 2)))

# CONV_2: here we increase the depth to 64
model.add(Conv2D(64, (3, 3), stride=1, padding='same', activation='relu'))
# POOL_2: more downsampling
model.add(MaxPooling2D(pool_size=(2, 2)))

# flatten since too many dimensions, we only want a classification output
model.add(Flatten())

# FC_1: fully connected to get all relevant data
model.add(Dense(64, activation='relu'))

# FC_2: output a softmax to squash the matrix into output probabilities for the 10 classes
model.add(Dense(10, activation='softmax'))

# print model architecture summary
model.summary()
```

3.4.2 Model summary

When you run this code, you will see the model summary printed as follows:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 64)	0
flatten_1 (Flatten)	(None, 3136)	0
dense_1 (Dense)	(None, 64)	200768
dense_2 (Dense)	(None, 10)	650
<hr/>		
Total params: 220,234		
Trainable params: 220,234		
Non-trainable params: 0		

Figure 3.38

General observations before looking at the model summary:

- We need to pass the `input_shape` argument to the first CONV layer only. Then we don't need to declare the input shape to the model since the output of the previous layer is the input of the current layer. So it is already known to the model.
- You can see the output of every CONV and POOL layers is a 3D tensor of shape (None, height, width, channels). The height and width are pretty straight forward. They are the dimensions of the image at this layer. The channels value represents the depth of the layer. This is the number of feature maps in each layer. The first value in this tuple which is set to `None` is the number of images that are processed in this layer. Keras sets this to `None` which means that this dimension is variable and accepts any number of `batch_size`.
- As you can see in the output shape columns, as you go deeper through the network, the image dimensions shrink and the depth increases as we discussed earlier in this chapter.
- Notice the number of total params (weights) that the network needs to optimize = 220,234 compared to the number of params from the MLP network we created earlier in this chapter = 669,706. We were able to cut it down to almost the third.

Let's take a look at the model summary line by line:

- CONV_1: we know that the input shape is (28x28x1). Look at the output shape of `conv2d_1`. It is (28x28x32). Since we set the stride to 1 and padding to 'same', the dimensions of the input image did not change. But the depth increased to 32. Why? Because we added 32 filters in this layer. Each filter produces one feature map
- POOL_1: the input of this layer is the output of its previous layer = (28x28x32). After the POOL, the image dimensions shrink and depth stays the same. Since we used a 2x2 pool, the output shape is (14x14x32)
- CONV_2: same as before, CONV layers increase depth and keeps dimensions. The input

from the previous layer is $(14 \times 14 \times 32)$. Since the filters in this layer is set to 64, the output = $(14 \times 14 \times 64)$

- POOL_2: same 2x2 pool, keeps the depth and shrinks the dimensions. Output = $(7 \times 7 \times 64)$
- Flatten: when you flatten the features tube of dimensions = $(7 \times 7 \times 64)$. This converts it into a flat vector of dimensions $(1, 3136)$
- Dense_1: we set this FC layer to have 64 neurons, then the output is 64
- Dense_2: this is the output layer that we set to 10 neurons since we have 10 classes

3.4.3 Number of parameters (weights)

Okay, now we know how to build the model and read the summary line by line to see how the image shape changes as it passes through the network layers. One important thing left, the 'param #' column on the right in the model summary.

What are the parameters?

Parameters are just another name for weights. These are the things that your network learns. As we discussed in chapter 2, the network's goal is to update the weights values during the gradient descent and backpropagation process until it finds the optimal values for these parameters that minimize the error function.

How are these parameters calculated?

In MLP, we know that the layers are fully connected to each other so the weight connections or edges are simply calculated by multiplying the number of neurons in each layer. In CNNs, the weights calculations are not as straightforward. Fortunately, there is an equation for this:

number of params = filters x kernel size x depth of the previous layer + no. of filters (for biases)

Let's apply this equation in an example. Suppose we want to calculate the parameters at the second layer of the previous mini-project:

Here is the code for CONV_2 again: `model.add(Conv2D(64, (3, 3), stride=1, padding='same', activation='relu'))`

Since we know that the depth of the previous layer = 32, then => *Params = $64 \times 3 \times 3 \times 32 + 64 = 18,496$*

Note that the POOL layers *do not* add any parameters. Hence, you will see the param # = 0 after the POOL layers in the model summary. Same thing for the flatten layer. No extra weights added.

Layer (type)	Output Shape	Param #
max_pooling2d_1 (MaxPooling2D (None, 14, 14, 32)		0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_2 (MaxPooling2D (None, 7, 7, 64)		0
flatten_1 (Flatten)	(None, 3136)	0

Figure 3.39

When we add all these parameters in the Param # columns together, we get the total number of parameters that this network needs to optimize = 220,234.

Trainable and non-trainable params

In the model summary, you will see the number of total params then below it the summary lists the number of trainable and non-trainable params. The trainable params are the weights that this neural network needs to optimize during the training process. In this example, all our params are trainable.

```
=====
Total params: 220,234
Trainable params: 220,234
Non-trainable params: 0
```

Figure 3.40

In later chapters, we will talk about using a pre-trained network and combining it to your own network for faster and more accurate results. In this case, you will make a decision to freeze some layers because they were already pre-trained. So you will see that not all the total params of the network will be trained. This is useful for you to understand the memory and space complexity of your model before starting the training process. But more on that later. As far as we know now, all our params are trainable until we get to this part.

3.5 Add Dropout layers to avoid overfitting

So far, you have been introduced to the main three layers of CNNs: 1) CONV, 2) POOL, and 3) FC layers. You will find these three layer types in almost every CNN architecture. But that's not all of them. There are additional layers that you can add to avoid overfitting. The one we will discuss now is Dropout layers.

3.5.1 What is overfitting?

The main cause of poor performance in machine learning is either overfitting or underfitting the data.

Underfitting is as the name implies, the model fails to fit the training data. This happens when the model is too simple to fit the data. For example, using one perceptron to classify non-linear dataset.

Overfitting on the other hand means fitting the data too much. As in memorizing the training data and not really learning the features to identify them. This happens when we build a super network that fits the training dataset perfectly (very low error while training) but fails to generalize to other data samples that it hasn't seen before. So you will see that, in overfitting, the network performs very well in the training dataset but performs poorly in the test dataset.



Figure 3.41

In machine learning, we don't want to build too simple models that underfit the data or too complex models that overfit. We want other techniques to build a neural network that is just right for our problem.

3.5.2 What is a dropout layer?

It is one of the most commonly used layers to prevent overfitting. Dropout turns off a percentage of neurons (nodes) that make up the layer of our network. This percentage is identified as a hyperparameter that you will be tuning when building your network. By "turning off", I mean these neurons are not considered during a particular forward or

backward pass. It may seem counterintuitive to throw away a connection in our network, but as a network trains, some nodes can dominate others or end up making large mistakes, and dropout gives us a way to balance our network so that every node works equally towards the same goal, and if one makes a mistake, it won't dominate the behavior of our model. You can think of dropout as a technique that makes a network resilient; it makes all the nodes work well as a team by making sure no node is too weak or too strong.

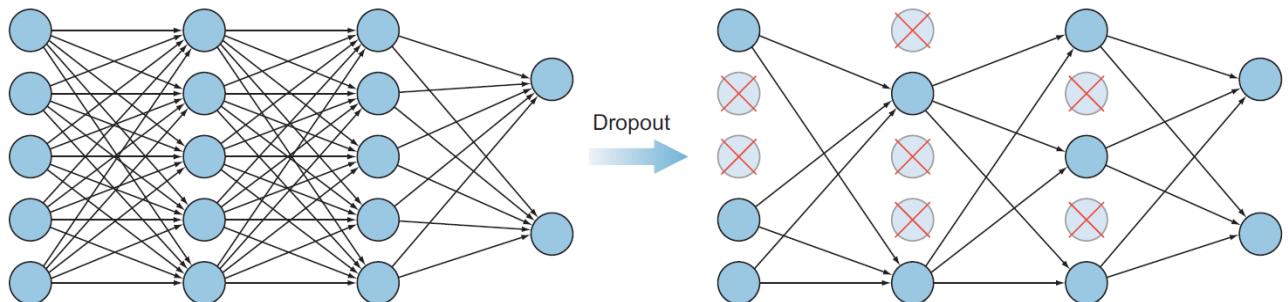


Figure 3.42

3.5.3 Why do we need dropout layers?

Neurons develop co-dependency amongst each other during training which controls the individual power of each neuron leading to overfitting of training data. To really understand why dropouts are effective, let's take a closer look at the above MLP and think about what the nodes in each layer really represent. The very first layer is the input layer that contains the input features. Correct? Okay, the second layer contains the features learned from the patterns of the previous layer when multiplied by the weights. Then the following layer is patterns learned within patterns. And so on. So each neuron represent a certain feature that when multiplied by a weight it is transformed into another feature. Now, when we randomly turn off some of these nodes, we are forcing the other nodes to learn patterns without relying on only one or two features because any feature can be randomly dropped out at any point. Which means spreading out the weights amongst all the features. This will lead to more trained neurons.

Dropout helps reducing interdependent learning amongst the neurons. And in that sense it helps to view dropout as a form of ensemble learning. In ensemble learning, we take a number of 'weaker' classifiers, train them separately and then at test time we use them by averaging the responses of all ensemble members. Since each classifier has been trained separately, it has learned different 'aspects' of the data and their mistakes (errors) are different. Combining them helps to produce a stronger classifier, which is less prone to overfitting.

The best analogy that helps me understand dropout is training your biceps with the bar. When lifting a bar with both arms, we usually tend to rely more on our stronger arm to lift a little extra weight than the other arm. Which will end up getting more training than the other one and develop a larger muscle on that arm.

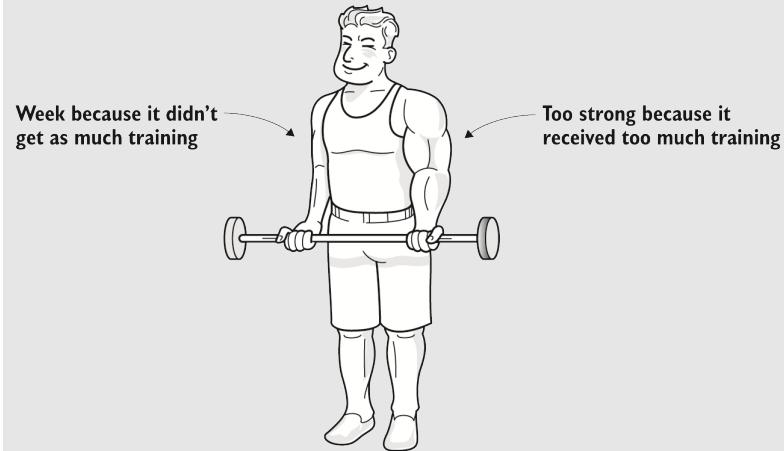


Figure 3.43

Dropout means mixing up our workout (training) a little. We tie our right arm and train our left arm only. Then tie the left arm and train the right arm only. Then we mix it up and go back to the bar with both arms. And so on. After some time, you will see that you have developed both of your biceps.



Figure 3.44

This is exactly what happens when we train neural networks. Sometimes we have very large weights and ends up dominating all the training while another part of the network doesn't really get much training. What dropout does is that it turns some neurons off and lets the rest of the neurons train. Then in the next epoch, it turns other neurons off. And so on.

3.5.4 Where does dropout layer go in the CNN architecture?

As we have learned in this chapter, a standard CNN consists of alternating CONV and POOL layers, with fully-connected FC layers at the end. To prevent overfitting, it's become a standard practice to inject a few dropout layers between the FC layers at the end of the architecture after you flatten the image. Why? Because dropout is known to work well in fully-

connected layers of convolutional neural nets and its effect in convolutional and pooling layers is, however, not well studied yet.

CNN architecture: ... CONV => POOL => Flatten => DO => FC => DO => FC

Let's see how we use Keras to add a dropout layer to our previous model:

```
# CNN and POOL layers
# .....
# .....
# flatten layer
model.add(Flatten())
model.add(Dropout(rate=0.3)) ← Dropout layer with 30% probability
# FC_1: fully connected to get all relevant data
model.add(Dense(64, activation='relu'))
model.add(Dropout(rate=0.5)) ← Dropout layer with 50% probability
# FC_2: output a softmax to squash the matrix into output probabilities
# for the 10 classes
model.add(Dense(10, activation='softmax'))

# print model architecture summary
model.summary()
```

As you can see, the dropout layer takes the *rate* as an argument. It represents the fraction of the input units to drop. For example, if we set the rate to 0.3, it means that 30% of the neurons in this layer will be randomly dropped in each epoch. So if we have 10 nodes in a layer, 3 of these neurons will be turned off and 7 will be trained. The 3 neurons are randomly selected and in the next epoch another randomly-selected neurons are turned off, and so on. Since we do this randomly, some neurons may get turned off more than other. And some others may never get turned off. This is ok because we do this many times that in average each neuron will get almost the same treatment. Note that this rate is another hyperparameter that we will tune when building our CNN.

3.6 Convolution over colored images (3D images)

Gray images representation

We have seen in chapter 1 how computers see grayscale images as a 2D matrix of pixels. To a computer the image looks like a 2D matrix of the pixels' values which represent intensities across the color spectrum. There is no context here, just a massive pile of data.



What we see

```

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 60 17 83 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 60 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 54 01 32 54 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 60 40 28 66 33 13 80
26 47 32 60 99 03 45 02 44 75 33 53 78 34 84 20 35 17 12 50
32 98 83 28 64 23 47 10 26 38 40 67 59 54 70 66 18 38 64 70
47 26 20 68 02 62 12 20 95 63 94 39 63 01 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 49 35 14 00 42 39 97 24 91 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 46 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 92 08 83 97 35 99 16 07 97 57 32 16 22 26 79 33 27 98 44
88 36 68 87 57 62 20 72 09 66 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 66 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 79 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 49 16 92 33 69 41 43 52 01 89 19 47 48
    
```

What computers see

Figure 3.45

What about color images?

Color images on the other hand are interpreted by the computer as a 3D matrix with height, width and **depth**. In the case of RGB images (red, green, and blue) the depth is three. One channel for each color. For example, a colored 28x28 image will be seen by the computer as a 28x28x3 matrix. Think of this as a stack of three 2D matrices. One for Red, one for Green and one for Blue channel of the image. Each one of the three matrices represent the value of intensity of this color. And when they are stacked together, they create a complete color image.

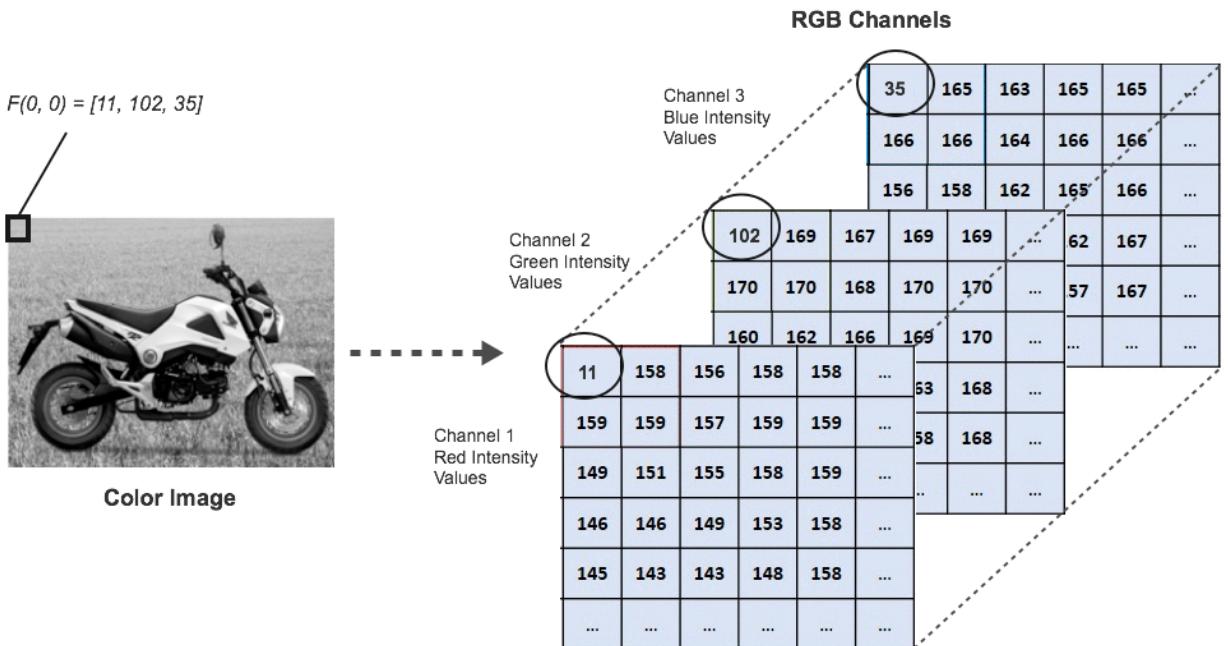


Figure 3.46

For generalization, we represent images as a 3D array = (height x width x depth). For grayscale images the depth = 1 and for color images the depth = 3.

How do we perform a convolution on a color image?

Similarly to what we did with grayscale images, we slide the convolutional kernel over the image and compute the feature maps. Only now the kernel is itself three-dimensional. One for each color channel.

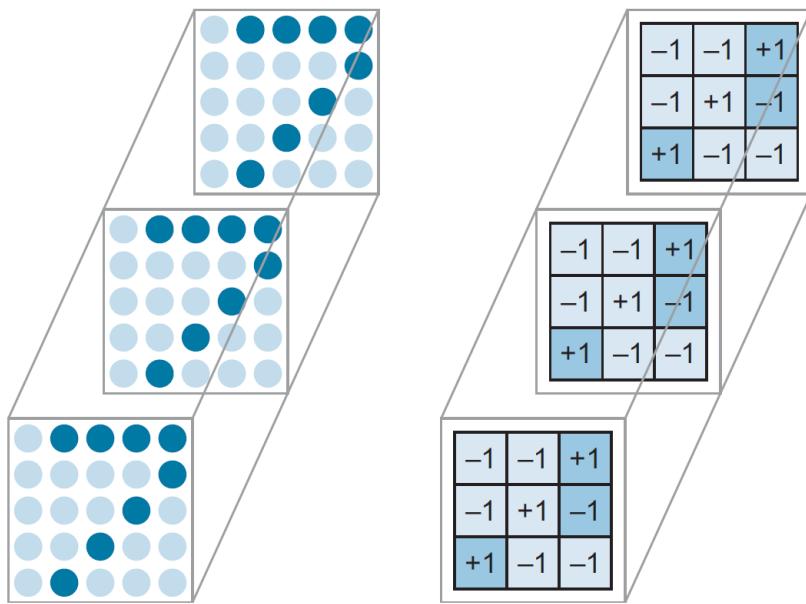


Figure 3.47

To perform convolution, we are going to do the same thing we did before. Only now, our sum is three times as many terms. Let's see how:

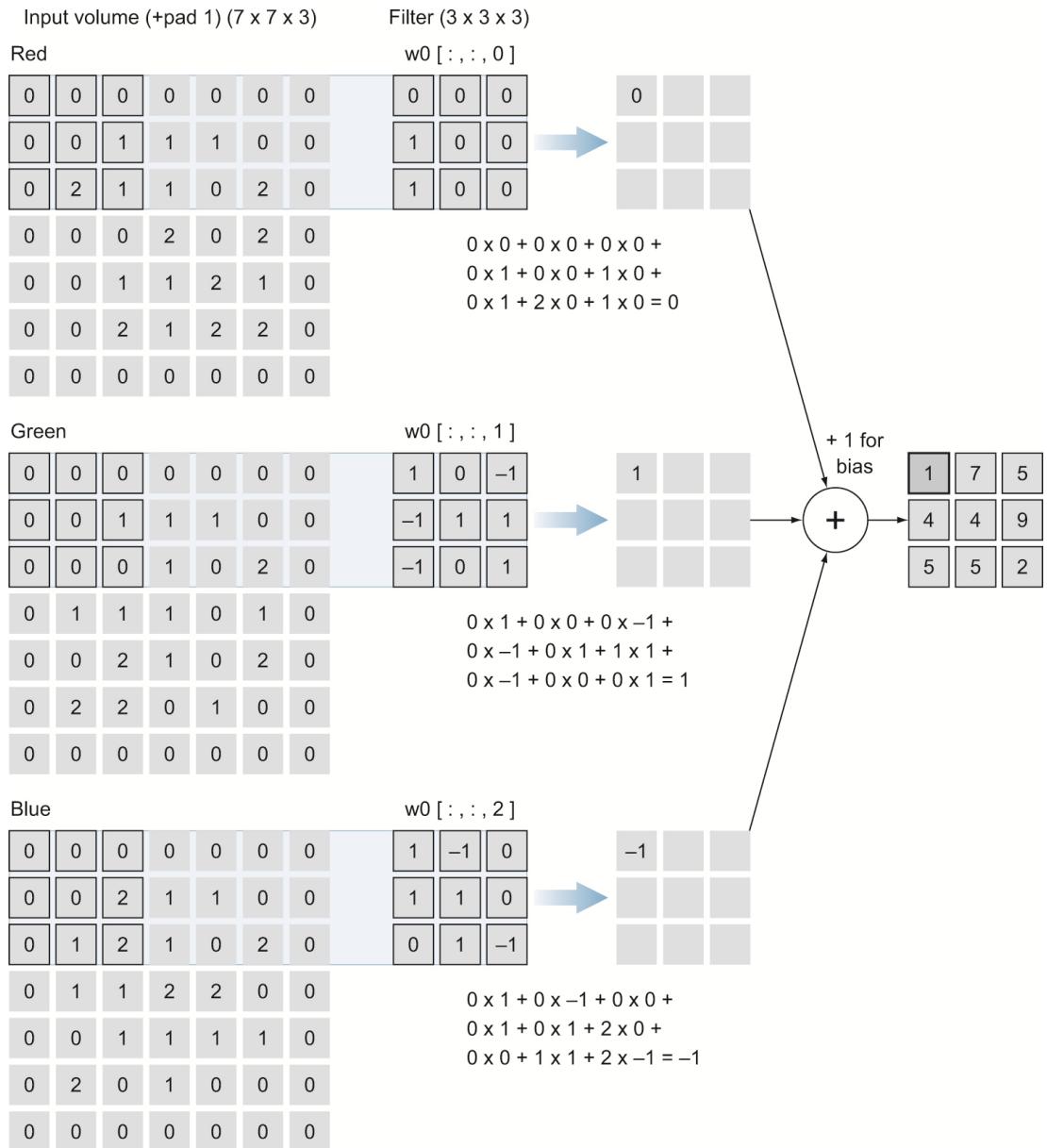


Figure 3.48

- Each of the color channels have their own corresponding filter.
- Each filter is going to slide over its image and multiply every corresponding pixel element-wise then add them all together to compute the convolved pixel value of each

filter. Similar to what we did before.

- We then add the three values together to get the value of a single node in the convolved image or feature map. And don't forget to add the bias value =1. Then we slide the filters over with one or more pixels (based on the stride value) and do the same thing. And so on until we compute the pixel values of all nodes in the feature map.

What happens to the computational complexity?

Note that if we pass a 3x3 filter over a grayscale image, we will have a total of 9 parameters (weights) for each filter (as we have already learned). In color images, every filter is itself a 3D filter. Which means that every filter has a number of parameters = height x width x depth = $3 \times 3 \times 3 = 27$. You can see how the network complexity increased when processing color images because it has to optimize more parameters as well as taking up more space in memory.

Color images contain more information than grayscale images. This can add unnecessary computation complexity and take up memory in space. However, color images are also really useful for certain classification tasks. That's why in some use cases you, as a computer vision engineer, will use your judgement to whether to convert your color images to grayscale in some use cases where color doesn't really matter. This is because in many objects, color is not needed to recognize and interpret an image. Grayscale could be enough for recognizing objects, and because color images contain more information than black and white images, they can add unnecessary complexity and take up more space in memory for all the extra parameters.

In the example below, you can see how patterns in lightness and darkness of an object (intensity) can be used to define the shape and characteristics of many objects. However, in other applications, color is important to define certain objects. Like skin cancer detection which relies heavily on the skin colors (red rashes). In general, when you think of computer vision applications like identifying cars, people, or skin cancer, you can decide whether color information is important or not by thinking about your own vision. If the identification problem is easier in color for us humans, it's likely easier for an algorithm to see color images too.

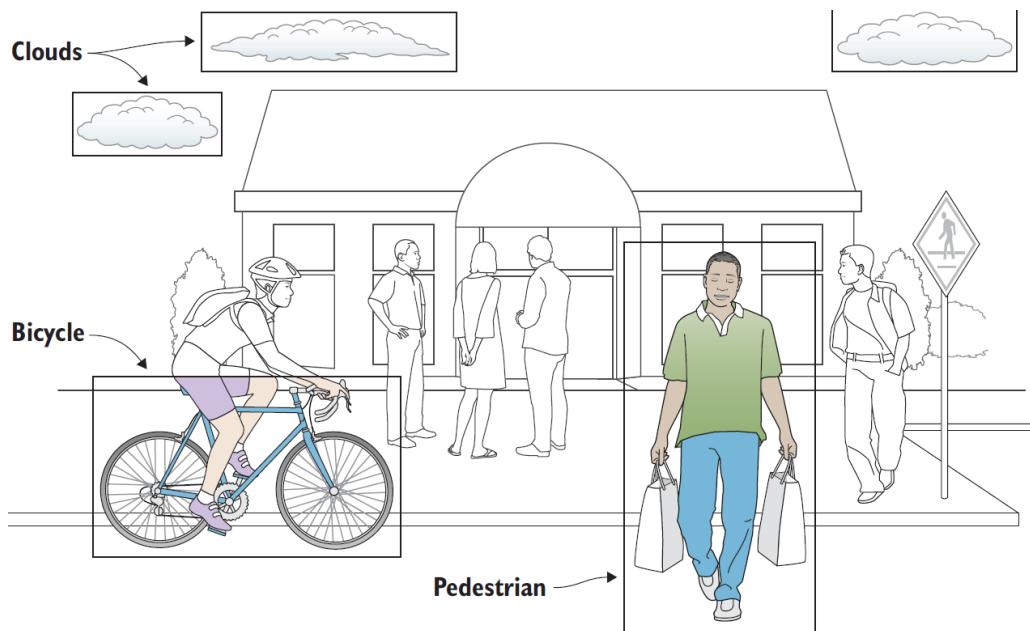


Figure 3.49

Note that in the above diagram we added only one filter (that contains 3 channels) which produced one feature map. Similar to what we did in the grayscale images, each filter we add will produce its feature map. In the CNN below, we have an input image of dimensions = $7 \times 7 \times 3$. We added two CONV filters of dimensions = 3×3 . The output feature map has a depth of 2 since we added two filters. Similar to what we did with grayscale images.

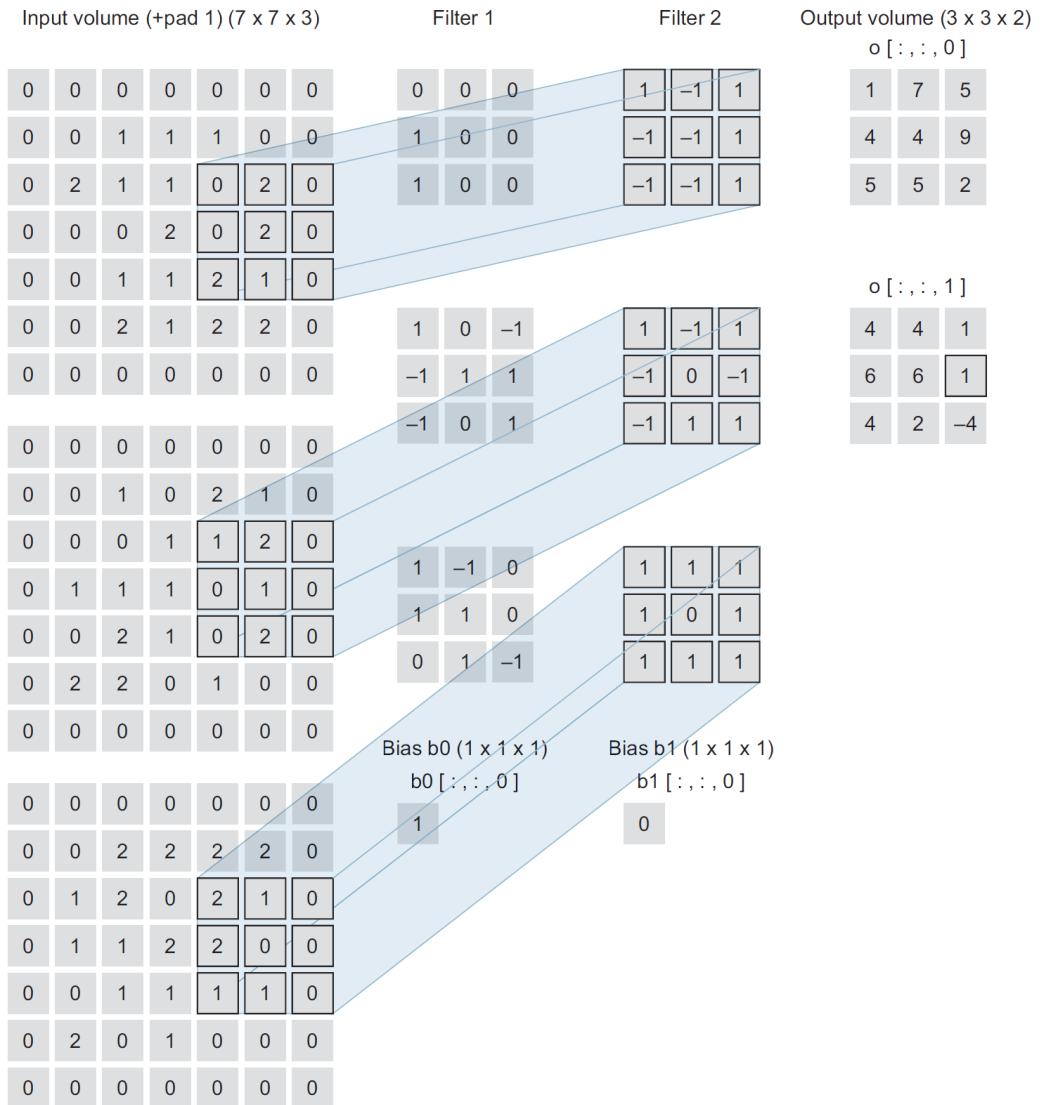


Figure 3.50

3.7 Chapter summary and takeaways

Here is what I want you to take away from this chapter:

Processing images with MLPs

- MLPs, ANNs, Dense, Feedforward network all refer to the regular fully connected neural network architecture that we discussed in chapter 2.
- MLPs usually work well for 1D inputs like the housing project from chapter 2. But they perform poorly with images because of two main reasons. 1) They only accept feature inputs in a vector form with dimensions $1 \times n$. This requires flattening the image which will lead to losing the spatial information from the image. 2) MLPs composed of fully connected (FC) layers which will yield millions and billions of parameters when processing bigger images. This in turn will raise the computational complexity and will not scale for many image problems.
- We need a new neural network architecture that solves these two drawbacks.

Convolutional Neural Networks

- CNNs really shine in image processing because they take the raw image matrix as an input without having to flatten the image. And they are composed of locally connected layers called CONV filters as opposed to the MLPs' dense layers.
- CNNs are composed of three main layers: 1) CONV layer for feature extraction, 2) POOL layer to reduce the network dimensionality, and 3) Fully connected layer for classification.
- The main cause of poor prediction performance in machine learning is either overfitting or underfitting the data. Underfitting means that the model is too simple and fails to fit (learn) the training data. Overfitting means that the model is too complex that it memorizes the training data and fails to generalize for test data that it hasn't seen before.
- Dropout layer is added to prevent overfitting. Dropout turns off a percentage of neurons (nodes) that make up the layer of our network.

CNNs hyperparameters

In addition to the regular neural network hyperparameters discussed in chapter 2, convolutional networks have their own hyperparameters:

- The CONV layer has the following attributes that we need to configure:

```
model.add(Conv2D(filters=16, kernel_size=2, strides='1', padding='same',
activation='relu'))
```

- filters = number of kernel filters in each layer (the depth of the hidden layer)
- kernel_size = the size of the filter (aka kernel). Usually 2, or 3, or 5.
- stride = is the amount by which the filter slides over the image. A stride of 1 or 2 are usually recommended as a good start
- padding = add columns and rows of zero values around the border of the image to reserve the image size in the next layer

- activation = it is strongly recommended to use "relu" activation function in the hidden layers
- POOL layer has the following attributes that we need to configure:

```
model.add(MaxPooling2D(pool_size=(2, 2), strides = 2))
```

- Dropout layer:

```
model.add(Dropout(0.4))
```

- rate: percentage of neurons that will be randomly turned off in each epoch. A rate of 0.3 to 0.5 is a good start. If you see signs of overfitting, increase this parameter.

The next chapter is solely dedicated for hyperparameter tuning. We will dive deeper into these hyperparameters and you will learn how to observe the network's performance during and after training to build an intuition on which knob to tune for each symptoms.

3.8 Project: Image classification for colored images (CIFAR-10 dataset)

Let's take a look at a complete end-to-end image classification project. In this project, we train a CNN to classify images from the CIFAR-10 dataset. [CIFAR-10](#) is an established computer-vision dataset used for object recognition. It is a subset of the [80 million tiny images dataset](#) and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. Now, fire your notebook and let's get started.

3.8.1 Load the dataset

First step is to load the dataset into our train and test objects. Luckily, Keras provides the CIFAR dataset for us to load using `load_data()` method. All we have to do is to import `keras.datasets` then load the data.

```
import keras
from keras.datasets import cifar10

# Load the pre-shuffled train and test data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks[])
    ax.imshow(np.squeeze(x_train[i]))
```

3.8.2 Image preprocessing

Based on your dataset and the problem you are solving, you will need to do some data cleanup and processing to get it ready for your learning model.

In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure below shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).

TIP: When using gradient descent, you should ensure that all features have a similar scale or else it will take much longer to converge.

Gradient descent with and without feature scaling

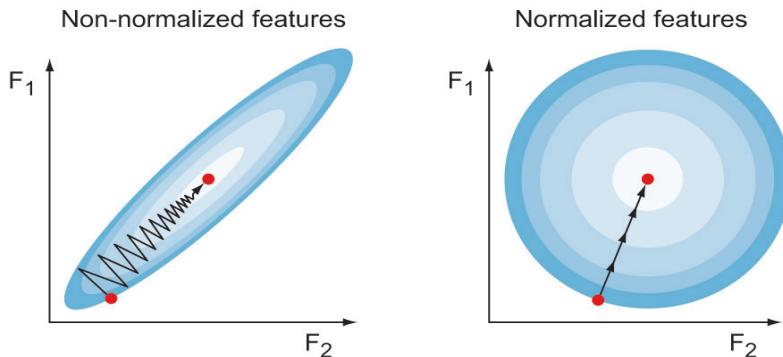


Figure 3.51

```
# rescale images by dividing the pixel values by 255 [0,255] --> [0,1]
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

Split dataset to train and validation

In addition to splitting our data into train and test datasets (step 1), it is a standard practice to further split the training data into train and validation datasets. Why? Because each split is used for a different purpose as follows:

- Training Dataset: The sample of data used to train the model.
- Validation Dataset: The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration.
- Test Dataset: The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.



Figure 3.52

```
# break training set into training and validation sets
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]

# print shape of training set
print('x_train shape:', x_train.shape)
```

```
# print number of training, validation, and test images
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')
```

Preparing the labels (one-hot encoding)

In this entire chapter and probably for the entire book, we discuss how computers process the input data (images) by converting them into numeric values in the form of matrix of pixel intensity. But what about the labels? How are the labels understood by computers? Every image in our dataset has a specific label that explains (in text) how this image is categorized. In this particular dataset for example, the labels are categorized by following 10 classes: ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']. We need to convert these text labels into a form that could be processed by computers. Computers are good with numbers so we will do something called *One-hot encoding*. One hot encoding is a process by which categorical variables are converted into a numeric form.

Say suppose the dataset is as follows:

Table 3.1

Image	Label
image_1	dog
image_2	automobile
image_3	airplane
image_4	truck
image_5	bird

After one-hot encoding we end up with the following:

Table 3.2

	airplane	bird	cat	deer	dog	frog	horse	ship	truck	automobile
image_1	0	0	0	0	1	0	0	0	0	0
image_2	0	0	0	0	0	0	0	0	0	1
image_3	1	0	0	0	0	0	0	0	0	0
image_4	0	0	0	0	0	0	0	0	1	0
image_5	0	1	0	0	0	0	0	0	0	0

Luckily, keras has a method that does just that for us:

```
from keras.utils import np_utils
```

```
# one-hot encode the labels
num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

One-hot encoding converted the $(1 \times n)$ labels vector to a labels matrix of dimensions $10 \times n$, where n is the number of sample images. So, if we have a 1,000 images in our dataset, the labels vector will have the dimensions of 1×1000 . After one-hot encoding, the labels matrix dimensions will be 1000×10 . That's why when we are defining our network architecture (in the next step) we will make the output softmax layer contains 10 nodes. Where each node represent the probability of each class we have.

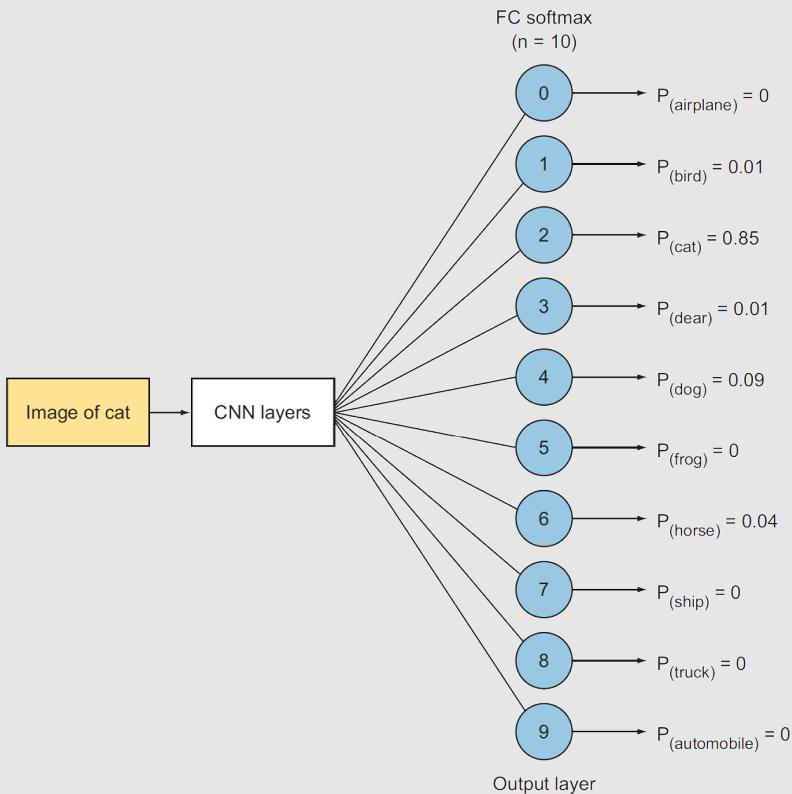


Figure 3.53

3.8.3 Define the model architecture

You learned that the core building block of CNNs (and neural networks in general) is the *layer*. Most of deep learning project consist of stacking together simple layers that will implement a form of *data distillation*. As we learned in this chapter, the main CNN layers are CONV, POOL, and FC layers. In addition to the activation functions.

How do you decide on the network architecture?

How many convolutional layers to create? How many pooling layers?

In my opinion is very helpful to read about some of the most popular architectures (Alexnet, Resnet, Inception), and extract the key ideas leading to the design decisions. Looking at how these state-of-the-art architectures are built and playing with your own projects will help you build an intuition on the CNN architecture the most suits the problem you are solving. We will explain the most popular CNN architectures in the next chapter. Until then here is what we need to know:

- The more layers you add the better (at least theoretically) your network will learn but this will come with the cost of increasing the computational and space complexity because it increases the number of parameters to optimize
 - As pointed out earlier in this chapter, the height and width of the image will decrease and the depth (or number of channels) will decrease
 - In general, two or three layers of 3x3 CONV layers followed by 2x2 POOL can be a good start. Add more CONV + POOL layers until your image is a reasonable size (say 4x4 or 5x5), then add a couple fully connected layers for the classification
 - There are several hyperparameters that you need to set up here (like filter, kernel_size, and padding). One common guideline is to not try to reinvent the wheel here setting the hyperparameters. But instead look in the literature to see what hyperparameters usually work for others. And choose an architecture that worked well for someone else as a starting point then tune these hyperparameters to improve on it.
- Like I said, the next chapter is all dedicated to look at what worked well for others.

Important callout on hyperparameter tuning. I don't want you to get hung up on hyperparameter setting when building your first CNNs. One of the best ways for you to gain an intuition on how to put these layers and hyperparameters together is to actually see a number of concrete examples of how other have done it. As I pointed out earlier, most of your work as a deep learning engineer will be building your architecture and tuning the parameters. The main takeaway that I want you to get out of this chapter is:

- understand how the main CNN layers work (CONV, POOL, FC, Dropout) and why they are here
- Understand what each hyperparameter does (number of filters in the CONV layer, kernel_size, stride and padding)
- At the end, being able to implement any given architecture in keras. If you are able to replicate this project on your own dataset, you are good to go.

In the chapter 4, we will review several state-of-the-art architectures and see what worked for them. For now, let's move on with the architecture below which is inspired by AlexNet, a popular CNN architecture that won the ImageNet challenges in 2011. More details on AlexNet in the next chapter.

AlexNet CNN architecture is composed of 5 CONV and POOL layers + 3 FC layers.

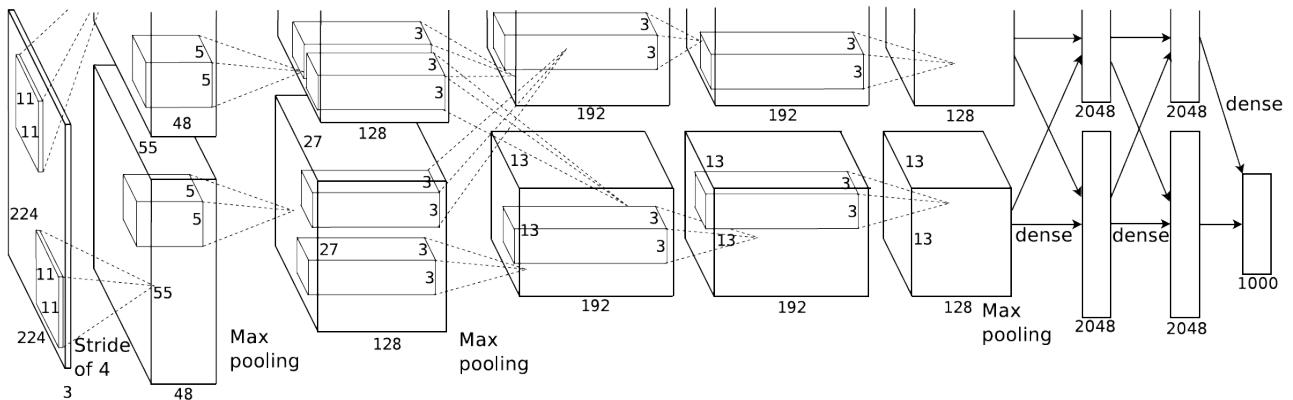


Figure 3.54

Let's try a smaller version of AlexNet and see how it performs with our dataset. And based on the results, we can add more layers. Our architecture will be stacking 3 CONV layers and 2 FC layers as follows:

CNN: INPUT => CONV_1 => POOL_1 => CONV_2 => POOL_2 => CONV_3 => POOL_3

=> DO => FC => DO => FC (Softmax)

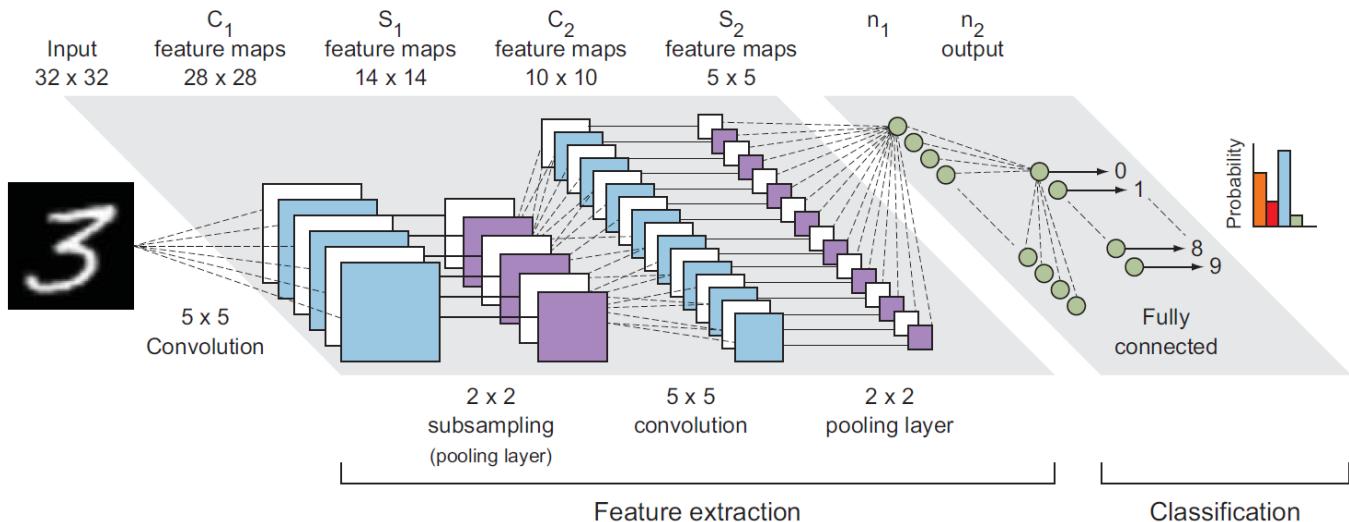
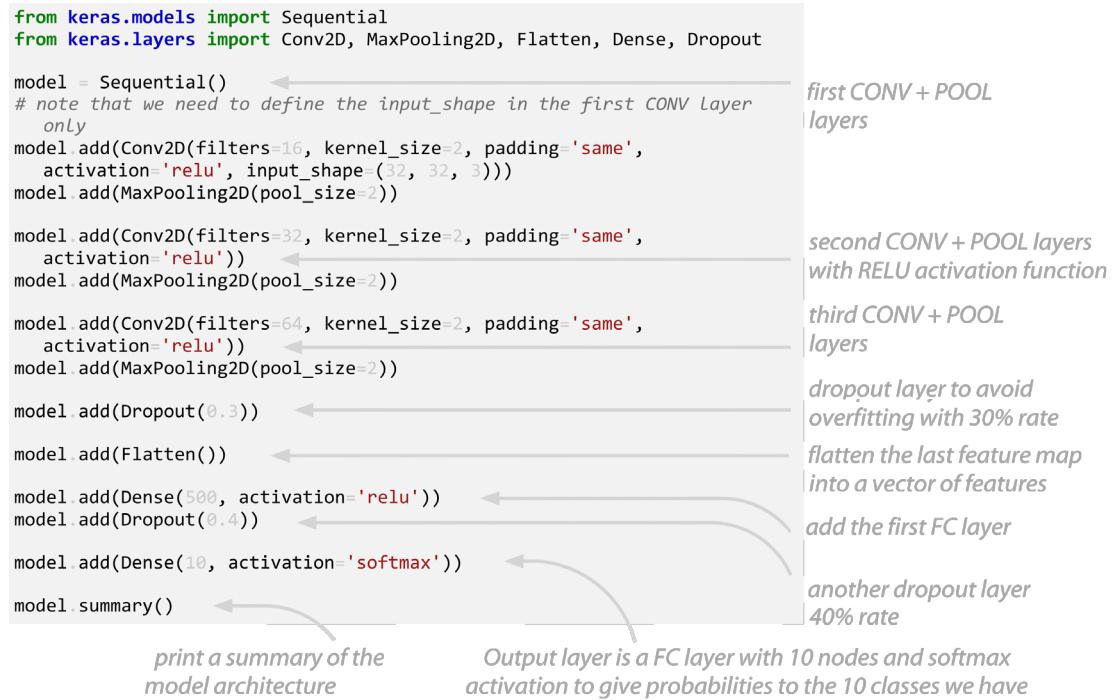


Figure 3.55

As you can see, our network consists of three CONV and POOL layers + 2 FC layers (also called dense). Note that we will use the RELU activation function for all the hidden layers. In

the last dense layer we will use a Softmax activation function with 10 nodes to return an array of 10 probability scores (summing to 1). Each score will be the probability that the current image belong to our 10 image classes.



When you run the above cell, you will see the model architecture and how the dimensions of the feature maps change with every successive layer:4

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 32, 16)	208
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 16)	0
conv2d_2 (Conv2D)	(None, 16, 16, 32)	2080
max_pooling2d_2 (MaxPooling2D)	(None, 8, 8, 32)	0
conv2d_3 (Conv2D)	(None, 8, 8, 64)	8256
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 64)	0
dropout_1 (Dropout)	(None, 4, 4, 64)	0
flatten_1 (Flatten)	(None, 1024)	0
dense_1 (Dense)	(None, 500)	512500
dropout_2 (Dropout)	(None, 500)	0
dense_2 (Dense)	(None, 10)	5010

Total params: 528,054
Trainable params: 528,054
Non-trainable params: 0

Figure 3.56

We learned how to understand this summary before. As you can see, our model has 528,054 parameters (weights and biases) to train. We also learned how this number was calculated.

3.8.4 Compile the model

The last step before training our model is to define three more hyperparameters: loss function, optimizer, and metrics to monitor during training and testing.

- Loss function: how the network will be able to measure its performance on the training data
- Optimizer: the mechanism that the network will use to optimize its parameters (weights and biases) to yield the minimum loss value. It is usually one of the variants of stochastic gradient descent (SGD) explained in chapter 2.
- Metrics to monitor during training and test:

Feel free to revisit chapter 2 for more details on the exact purpose and different types of loss functions and optimizers.

```
# compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

3.8.5 Train the model

We are now ready to train the network. In Keras, this is done via a call to the network's `.fit()` method (as in fit the model to the training data).

```
from keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1,
                               save_best_only=True)

hist = model.fit(x_train, y_train, batch_size=32, epochs=100,
                  validation_data=(x_valid, y_valid), callbacks=[checkpointer], verbose=2,
                  shuffle=True)
```

When you run the above cell, the training will start and the below verbose will display one epoch at a time. Since we cannot fit 100 epochs of verbose in one page, we just added a screenshot of the first 13 epochs. But when you run this on your notebook, this will keep going for 100 epochs.

```

Train on 45000 samples, validate on 5000 samples
Epoch 1/100
Epoch 00000: val_loss improved from inf to 1.35820, saving model to model.weights.best.hdf5
46s - loss: 1.6192 - acc: 0.4140 - val_loss: 1.3582 - val_acc: 0.5166
Epoch 2/100
Epoch 00001: val_loss improved from 1.35820 to 1.22245, saving model to model.weights.best.hdf5
53s - loss: 1.2881 - acc: 0.5402 - val_loss: 1.2224 - val_acc: 0.5644
Epoch 3/100
Epoch 00002: val_loss improved from 1.22245 to 1.12096, saving model to model.weights.best.hdf5
49s - loss: 1.1630 - acc: 0.5879 - val_loss: 1.1210 - val_acc: 0.6046
Epoch 4/100
Epoch 00003: val_loss improved from 1.12096 to 1.10724, saving model to model.weights.best.hdf5
56s - loss: 1.0928 - acc: 0.6160 - val_loss: 1.1072 - val_acc: 0.6134
Epoch 5/100
Epoch 00004: val_loss improved from 1.10724 to 0.97377, saving model to model.weights.best.hdf5
52s - loss: 1.0413 - acc: 0.6382 - val_loss: 0.9738 - val_acc: 0.6596
Epoch 6/100
Epoch 00005: val_loss improved from 0.97377 to 0.95501, saving model to model.weights.best.hdf5
50s - loss: 1.0090 - acc: 0.6484 - val_loss: 0.9550 - val_acc: 0.6768
Epoch 7/100
Epoch 00006: val_loss improved from 0.95501 to 0.94448, saving model to model.weights.best.hdf5
49s - loss: 0.9967 - acc: 0.6561 - val_loss: 0.9445 - val_acc: 0.6828
Epoch 8/100
Epoch 00007: val_loss did not improve
61s - loss: 0.9934 - acc: 0.6604 - val_loss: 1.1300 - val_acc: 0.6376
Epoch 9/100
Epoch 00008: val_loss improved from 0.94448 to 0.91779, saving model to model.weights.best.hdf5
49s - loss: 0.9858 - acc: 0.6672 - val_loss: 0.9178 - val_acc: 0.6882
Epoch 10/100
Epoch 00009: val_loss did not improve
50s - loss: 0.9839 - acc: 0.6658 - val_loss: 0.9669 - val_acc: 0.6748
Epoch 11/100
Epoch 00010: val_loss improved from 0.91779 to 0.91570, saving model to model.weights.best.hdf5
49s - loss: 1.0002 - acc: 0.6624 - val_loss: 0.9157 - val_acc: 0.6936
Epoch 12/100
Epoch 00011: val_loss did not improve
54s - loss: 1.0001 - acc: 0.6659 - val_loss: 1.1442 - val_acc: 0.6646
Epoch 13/100
Epoch 00012: val_loss did not improve
56s - loss: 1.0161 - acc: 0.6633 - val_loss: 0.9702 - val_acc: 0.6788

```

Figure 3.57

Observe the network behavior

Looking at the verbose above will help you analyze how your network is performing and will give you pointers on which knob (hyperparameter) to tune. We will discuss this in details in chapter 4. For now, let's look at the most important giveaways from the above verbose:

- “loss” and “acc” are the error and accuracy values for the training data. “val_loss” and “val_acc” are the error and accuracy values for the validation data.
- Look at the `val_loss` and `val_acc` values after each epoch. Ideally we want the `val_loss` value to be decreasing and the `val_acc` to be increasing. This means that the network is actually learning after each epoch.
- From epochs 1 to 6, you can see that the model is saving the weights after each epoch because the validation loss value is improving. So by the end of each epoch, we save the weights which are considered the best weights so far.
- At epoch #7 the `val_loss` went up to 1.1300 from 0.9445 which means that it did not improve. So, the network did not save the weights at this epoch. If you stop the

training now and load the weights from epoch #6, you will get the best results that you achieved during the training.

- Same for epoch #8, the val_loss decrease, so the network saves the weights as best values. And at epoch #9 no improvement and so forth.
- If you stop your training after 12 epochs and load the best weights. The network will load the weights saved after epoch #10 at val_loss = 0.9157 and validation accuracy = 0.6936. This means that you can expect to get accuracy on the test data close to 69%.

One tip is to keep your eyes on the following popular phenomenons:

val_loss is oscillating: if the val_loss is oscillating up and down then you might want to decrease the learning rate hyperparameter. For example, if you see the val_loss going from 0.8 to 0.9 to 0.7 to 1.0 and so on, this might mean that your learning rate is too high to descent the error mountain. Try decreasing the learning rate and let the network train for longer time.

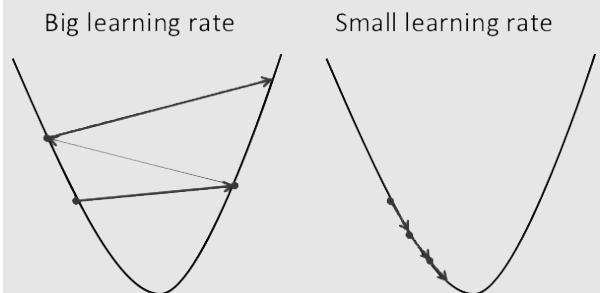


Figure 3.58

val_loss is not Improving (underfitting): if the val_loss is not decreasing, this might mean that your model is too simple to fit the data (underfitting). Then you might want to build a more complex model by adding more hidden layers to help the network fit the data.

Overfitting: if the “loss” value is decreasing and the “val_loss” stopped improving, this means that you network started to overfit the training data and failing to decrease the error for the validation data. In this case, consider adding using one of the techniques to prevent overfitting like dropout layers. There are other techniques to avoid overfitting that we will discuss in the next chapter.

3.8.6 Load the model with the best val_acc

```
# Load the weights that yielded the best validation accuracy
model.load_weights('model.weights.best.hdf5')
```

3.8.7 Evaluate the model

The last step is to evaluate our model and calculate the accuracy value in percentage of how many times our model is correct in predicting the image classification.

```
# evaluate and print test accuracy
score = model.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', score[1])
```

When you run the above cell you will get an accuracy of about 70%. That is not bad. But we can do a lot better. Try to play with the CNN architecture by adding more CONV + POOL layers and see if you can improve your model.

In the next chapter we will discuss strategies to setup your deep learning project and hyperparameters tuning to improve the model's performance. At the end of chapter 4, we will revisit this project to apply these strategies and improve the accuracy to above 90%.

4

Structuring Deep Learning Projects and Hyperparameters tuning

“With four parameters I can fit an elephant and with five I can make him wiggle his trunk”

-- **John von Neumann**

This chapter concludes the first part of this book, the deep learning foundation. In chapter 2, you learned how to build a multilayer perceptron (MLP). In chapter 3, you learned a new neural network architecture topology that are very commonly used in computer vision problems called convolutional neural networks (CNNs). In this chapter, we will wrap up this topic by discussing how to structure your machine learning project from the start to finish. You will learn strategies to quickly and efficiently get your deep learning systems working, analyze the results and improve the network performance.

As you might have already noticed from the previous projects, deep learning is a very empirical process. It mostly relies on running experiments and observing the model performance more than having one magic formula for success that fits all problems. We often have an idea of the starting solution, code it up, run the experiment to see how it did, and then use the outcome of this experiment to refine our ideas. When building and tuning a neural network, you will find yourself making many seemingly arbitrary decisions:

- What is a good architecture to start with?
- How many hidden layers should you stack?
- How many hidden units or filters should go in each layer?
- What is the learning rate?
- Which activation function to use?
- Which is more effective to improve results, getting more data or tuning

- hyperparameters?
- And many more decisions to build your experiment and improve further.

In this chapter you will learn the following:

- **Define the performance metrics for your system:** in addition to model accuracy, you will learn other metrics like precision, recall, and f-score that can be used when evaluating your network.
- **Define a baseline model:** you will learn how to choose an appropriate neural network architecture to run your first experiment.
- **Get your data ready for training:** in real-world problems, data comes in messy and not ready to be fed to the neural network. In this section, you will learn how to massage your data to get it ready for learning.
- **Evaluate your model and interpret its performance:** when training is complete, you need analyze your model performance to identify bottlenecks in the model performance and narrow down your improvement options. This means diagnosing which of the network components are performing worse than expected and identify whether poor performance is due to overfitting, underfitting, or a defect in the data.
- **Improve the network and tune hyperparameters:** Finally, we will dive deep into the most important hyperparameters to help you build an intuition to determine which hyperparameters you need to tune. We will explain tuning strategies to make incremental changes based on your diagnosis from the previous step.

With more practice and trying many different things, deep learning engineers and researchers build an intuition over time as to what are the most effective improvements they can make. My advice is to get your hands dirty and try different architectures and approaches to develop your hyperparameter-tuning skills.

Ready? Let's get started!

4.1 Define the performance metrics

Performance metrics provide us a way to *evaluate our system*. When we develop a model, we want to find out how good it is doing. The simplest way to measure the “goodness” of our model is by measuring its **accuracy**. The accuracy metric measures how many times our model made the correct prediction. So, if we test the model with 100 input samples and it made the correct prediction 90 times, this means that the model is 90% accurate.

Here is the equation used to calculate the model accuracy:

$$\text{accuracy} = \frac{\text{correct predictions}}{\text{total number of examples}}$$

4.1.1 Is accuracy the best metric to evaluate the model?

We have been using accuracy as a metric to evaluate our model in the past projects and it works fine for many cases. But let's consider the following problem: You are designing a medical diagnosis test for a rare disease. Suppose that only one in every million people have this disease. Without any training or even building any system at all, if you hardcode the output to be always negative (no disease found), your system is set to achieve 999.999%

accuracy all the time. Is that good? The system is 99.999% accurate, which might sound fantastic, but it will never capture the patients with the disease. This means that the accuracy metric is not suitable to measure the "goodness" of this model. We need other evaluation metrics that measure different aspects of the model's prediction ability.

4.1.2 Confusion matrix

To set the stage for other metrics, we are going to use a **confusion matrix**. It is a table that is often used to describe the performance of a classification model. The confusion matrix itself is relatively simple to understand, but the related terminology can be a little confusing at first. Once you understand it, you'll find that the concept is really intuitive and make a lot of sense. Let's go through it step-by-step.

The goal is to describe the model performance from different angles other than the prediction accuracy. For example, suppose we are building a classifier to predict whether a patient is sick or healthy. The expected classifications are either *positive* as in the patient is sick, or *negative*, the patient is healthy. We run our model on 1,000 patients and enter the model predictions in the following table:

Table 4.1

	Predicted sick (positive)	Predicted healthy (negative)
Sick Patients (positive)	100 True Positives (TP)	30 False Negative (FN)
Healthy Patients (negative)	70 False Positives (FP)	800 True Negatives (TN)

Let's now define the most basic terms, which are whole numbers (not rates):

- **true positives (TP):** These are cases in which the model has correctly predicted yes (they have the disease).
- **true negatives (TN):** the model correctly predicted no disease.
- **false positives (FP):** the model falsely predicted yes, but they don't actually have the disease (In some literature known as a "Type I error" or "error of the first kind").
- **false negatives (FN):** the model falsely predicted no, but they actually do have the disease (In some literature known as a "Type II error" or "error of the second kind")

Now, the patients that the model predict are negative (no disease) are the ones that the model believes that they are healthy and we can send them home without further care and the ones that we predict positive are the ones that we are going to send for further investigation. Now, which mistake would we rather make? Mistakenly diagnosing someone as positive (has disease) and sending them for more investigation is not as bad as mistakenly diagnosing someone as negative (healthy) and sending them home which could risk his life. The obvious choice of evaluation metric in this case is that we care more about the number of false negatives (FN). We want to find all the sick people, even if the model accidentally classify more healthy people as sick. This metric is called **Recall**.

4.1.3 Precision and Recall

Recall (also known as **Sensitivity**): out of the sick patients, how many did we correctly diagnose as sick? In other words, how many times did the model mistakenly diagnose a positive patient as negative (i.e. false negatives FN)? Recall is calculated by the following equation:

$$\text{Recall} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}}$$

Precision (also known as **Specificity**): it is the opposite of recall. It basically says: out of all the patients predicted to be positive, how many of them were actually positive (how many were actually sick)? In other words, what is the false positive rate of this model? Precision is calculated by the following equation:

$$\text{Precision} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$

It is important to call out that, while in the health diagnostic example we decided that recall is a better metric, other use cases would require different metrics like Precision. To be able to identify the most appropriate metric for your problem, ask yourself which of the *two falses* would you care about more, the false positive or the false negative? If FP, then you are looking for precision. If you care more about FN, then recall is your answer.

Consider the spam email classifier for example, which of the two falses would you care about more? False classifying a non-spam email as spam and it gets lost or falsely classifying a spam email as non-spam and it makes its way to the inbox folder? I believe you would care more about the former. You don't want the receiver to lose an email because your model misclassified it as spam. We want to catch all spam BUT it is very bad to lose a non-spam email. In this example, precision is a suitable metric to use.

In some applications, you might care about both precision and recall at the same time. That's called F-Score explained next.

4.1.4 F-Score

In many cases, you would want to summarize the performance of the classifier with a single metric that represents both recall and precision. To do so, we can convert precision p and recall r into a single F-score metric. In mathematics, this is called the *Harmonic Mean* of p and r .

$$F\text{-score} = \frac{2pr}{p + r}$$

The good thing about f-score is that it gives a good overall representation to how your model is doing. Let's take a look at the health diagnostic example again. We agreed that this is a *high recall* model. But what if the model is doing really good on the FN and giving us a high recall score but it is performing really poor on the FP and giving us a low precision score. Doing poor on FP means, that in order to not miss any sick patient, it's gone so far on the other side to mistakenly diagnosing a lot of patients as sick to be on the safe side. So, while recall might be more important for this problem, it is good to look at the model from both scores precision and recall together.

Table 4.1

	Precision	Recall	F-Score
Classifier A	95%	90%	92.4%
Classifier B	98%	85%	91%

Defining the model evaluation metric is a necessary step because it will guide your actions to improve the system. Without clearly defined metrics, it can be difficult to tell whether changes to a machine learning system make progress or not.

4.2 Design a baseline model

Now that you have selected the metrics that you are going to use to evaluate your system, it is time to establish a reasonable end-to-end system for your problem to start your training. Depending on the problem that you are solving, you need to design the baseline to your network type and architecture. In this step you will want to answer questions like:

- Should I use an MLP or CNN network (or RNN explained later in the book)?
- Should I use other object detection techniques like YOLO or SSD? (will be explained in later chapters)
- How deep should my network be?
- Which activation type?
- What kind of optimizer to you use?
- Do I need to add any other regularization layers like dropout or batch normalization to avoid overfitting?

The best way to get started with your computer vision model is that if your problem is similar to another problem that has been studied extensively, you will probably do well by first copying the model and algorithm that is already known to perform best on the previously studied task. You can even take a pre-trained model that was trained on a different dataset to your own problem without having to train it from scratch. This is called Transfer Learning that will be discussed in details in the next chapter.

For example, in the last chapter's project we used the architecture of the popular **AlexNet** as a baseline model. In the figure below you will see the architecture of AlexNet deep convolutional neural network, showing the dimensions of each layer. The input layer is followed by 5 convolutional layers (CONV1 to CONV5), the output of the fifth convolutional layer is fed into two fully-connected layers (FC6-7), then the output layer is a fully-connected layer (FC8) with a softmax function.

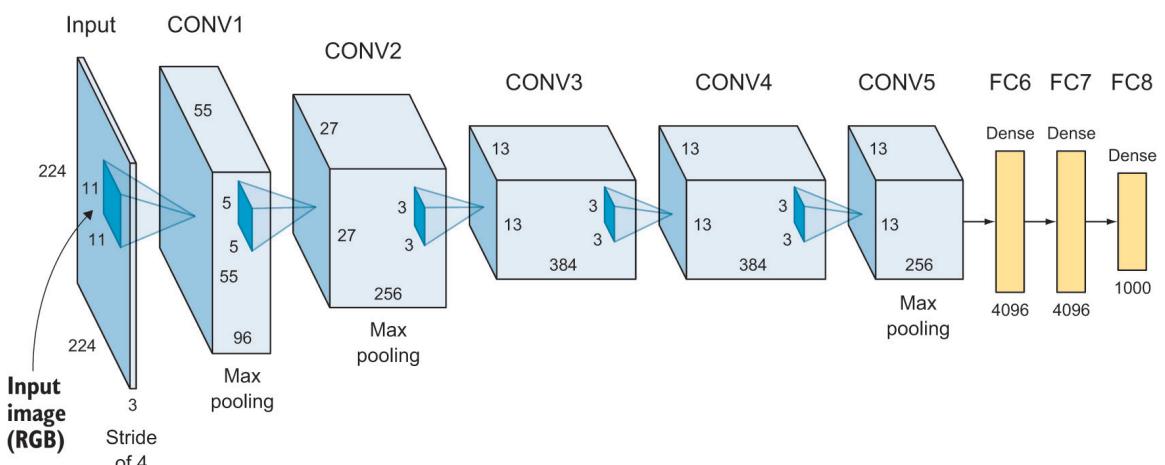


Figure 4.1

Network architecture

```
INPUT => CONV1 => POOL1 => CONV2 => POOL2 => CONV3 => CONV4 => CONV5 => POOL3 => FC6 => FC7  
=> SOFTMAX_8
```

Looking at the AlexNet architecture, you will find all the network hyperparameters that you need to get started with your own model:

- Network depth (no. of layers): 5 CONV layers + 3 FC layers
- Layers' depth (no. of filters): CONV1 = 96, CONV2 = 256, CONV3 = 384, CONV4 = 385, CONV5 = 256
- Filter size: 11x11, 5x5, 3x3, 3x3, 3x3
- ReLU as activation function in the hidden layers (CONV1 all the way to FC7)
- Max POOL layers after CONV1, CONV2, and CONV5
- FC6 and FC7 have 4096 neurons
- FC8 has 1000 neurons and uses Softmax activation function

In the next chapter we will discuss some of the most popular CNN architectures with their code implementation in Keras. We will discuss networks like LeNet, AlexNet, VGG, ResNet and Inception that will build more intuition on what architecture works best for different problems and perhaps inspire you to invent your own CNN architecture.

4.3 Get your data ready for training

OK! Now, we defined the performance metrics that we will use to evaluate our model and built the architecture of our baseline model. Let's get our data ready for training. It is important to call out that this process varies a lot based on the problem and the data you have. In here, I'll explain the basic data massaging techniques that you need to do before training the model. I'll also try to build an intuition on what "ready data" look like to help you determine which preprocessing techniques you need to apply.

4.3.1 Split your data into train/validation/test datasets

The basic idea behind training any machine learning model is that we split the data into train and test datasets. We use the training dataset to train the model and update the weights then evaluate the model against the test dataset that it hasn't seen before. The golden rule here is: *never use the testing data for training*. The reason you should never show the test samples to the model while training is to make sure the model is not cheating. You show it the training samples to learn their features then test how it generalizes on a dataset that it never sees before to give you an unbiased evaluation on the model performance.



Figure 4.2

WHAT IS THE VALIDATION DATASET?

Well, we found out that during the training process, we need to evaluate the model accuracy and error after each epoch to understand how it is performing to tune its parameters. If we use the test dataset to evaluate the model during training we will break our golden rule of never using the testing data during training. The test data is only used to evaluate the final performance of the model **after** training is complete. So we will make an additional split called **validation dataset** to evaluate the model during training and tune its parameters. And once the model has completed the training we test the final performance over the test dataset.

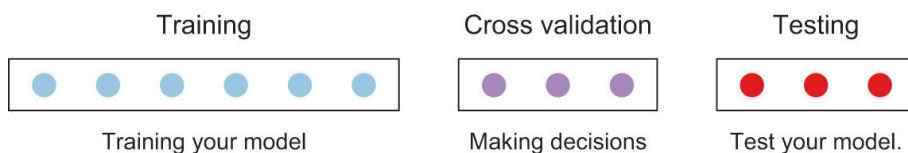


Figure 4.3

Take a look at this pseudo code for model training:

for each epoch for each training data instance

```

propagate error through the network
adjust the weights
calculate the accuracy and error over training data

for each validation data instance
    calculate the accuracy and error over the validation data

```

As we've seen in the project in chapter 3, when we train the model, we get the train_loss, train_acc, val_loss, and val_acc after each epoch. We use this data to analyze the network performance and diagnose for overfitting and underfitting as you will see in section 4 in this chapter.

```

Epoch 1/100
Epoch 00000: val_loss improved from inf to 1.35820, saving model to model.weights.best.hdf5
46s - loss: 1.6192 - acc: 0.4140 - val_loss: 1.3582 - val_acc: 0.5166
Epoch 2/100
Epoch 00001: val_loss improved from 1.35820 to 1.22245, saving model to model.weights.best.hdf5
53s - loss: 1.2881 - acc: 0.5402 - val_loss: 1.2224 - val_acc: 0.5644

```

Figure 4.4

WHAT IS A GOOD TRAIN/VALIDATION/TEST DATA SPLITS?

Traditionally, a 80/20 or 70/30 split between train and test datasets has been commonly used in ML projects. When we add the validation data set, we can go with 60/20/20 or 70/15/15. But that was back then we the entire dataset was just tens of thousands of samples. But with the huge amount of data that we have now, sometimes 1% for validation and test set is enough. For example, if our dataset contains 1 million samples, 10,000 samples is very reasonable for each of test and validation sets because it doesn't make sense to set a 100 or 200 thousand samples of your dataset away from your training process. It is better to use this data for the model training.

So to recap, if you have a relatively small dataset, the traditional ratios might be okay and if you are dealing with a large dataset, then it is fine to set your train and validation sets to much smaller than 10% or 20% of your data.

An important thing to be aware of when splitting your data, is to make sure that your train/val/test datasets are coming from the same distribution. Let's take a look at this example: suppose you are building a car classifier that will be deployed on cell phones to detect car models. Keeping in mind that deep learning networks are data hungry and the common rule of thumb is the more data you have, the better your model is set up to succeed. So, to source your data, you decided to go ahead and crawl the internet for car images that are all high quality, professionally framed images. Now you train your model, tune it, achieve satisfying results on your test dataset and ready to put it out to the world only to discover that it is performing poorly on real life images taken by phone cameras. This happens because your model has been trained and tuned to achieve good results on high quality images and failed to generalize on real life images that are maybe blurry or lower on resolution or any other different characteristics.

In more technical words: your training and validation datasets are composed of high quality images, whereas the production images (real life) will be lower quality images. So, it is very important that you add lower quality images into

your train and validate datasets as well. Hence, train/validate/test datasets should be coming from the same distribution.

4.3.2 Data preprocessing

Before you feed your data to the neural network you will need to do some data cleanup and processing to get it ready for your learning model. There are several preprocessing techniques that vary based on the state of your dataset and the problem that you are solving. The good news about neural networks is that they require minimum data preprocessing. Because when given a large amount of training data, they are able to extract and learn features from raw data unlike the other traditional machine learning techniques.

With that said, there are still some preprocessing techniques that might be required to improve performance or suffice specific limitations to the neural networks. Some of the most common preprocessing techniques are converting images to grayscale, image resizing, normalization, and data augmentation. In this section, I'll go through these preprocessing concepts and we will see their code implementation in the projects at the end of the chapters.

Image grayscaling

We talked in chapter 3 about how the color images are represented in 3 matrices vs only 1 matrix for grayscale images which adds computational complexity due to the larger number of parameters. So, you can make a judgment call to convert all your images to grayscale if your problem doesn't require colors to save on the computational complexity. A good rule of thumb here is to use the human-level performance rule: if you are able to identify the object with your eyes on grayscale images, then the neural network will probably be able do to it.

Image resizing

A main limitation for neural networks is that they require all images to be the same shape. If you are using MLPs for example, you need to set up the number of nodes in the input layer to be equal to the number of pixels in the image (remember, in chapter 3, how we flattened the image to feed it to the MLP). Same thing for CNNs. You will need to set the input shape of the first CONV layer. To demonstrate this, let's take a look at the Keras code to add the first CNN layer:

```
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',
                 input_shape=(32, 32, 3)))
```

As you can see above, we have to define the shape of the image at the first CONV layer. For example if we have three images with dimensions of: 32x32, 28x28, and 64x64, we have to resize all the images to one size before feeding them to the model.

IMAGE NORMALIZATION

Data normalization is the process of rescaling your data to ensure that each input feature (pixel, in the image case) has a similar data distribution. Often, raw images are comprised of pixels with varying scales (i.e. ranges of values). For example, one image may have pixels value range from 0 to 255 and another may be in the range of 20 to 200. Although not

required, it is preferred to normalize the pixel values from its raw state to the range of 0 to 1 to boost the learning performance and make the network converge faster.

How to normalize images

To make learning faster for your neural network, your data should have the following characteristics:

- Small values — Typically, most values should be in the [0, 1] range.
- Be homogenous — meaning, all pixels should have values in the same range.

Data normalization is done by subtracting the mean from each pixel and then dividing the result by the standard deviation. The distribution of such data would resemble a Gaussian curve centered at zero.

To add more intuition into the normalization process, let's visualize the operation in the scatterplot below:

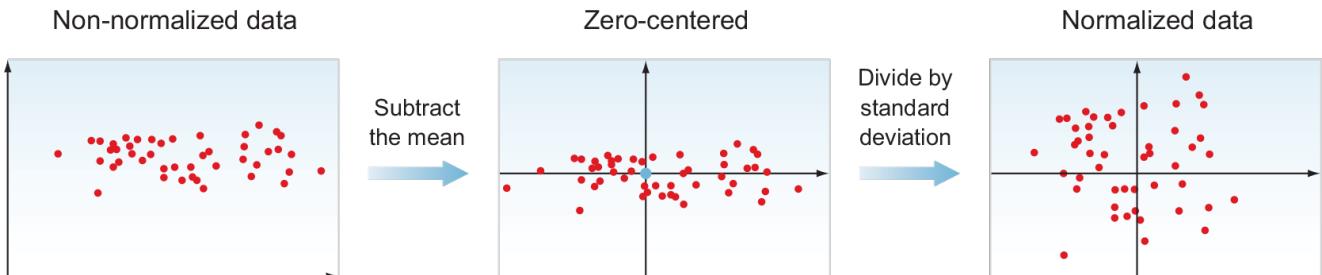


Figure 4.4b

When normalizing your data, make sure you normalize your training and test data by using the same mean and standard deviation. Because you want your data to go through the same transformation and rescaled exactly the same way. You will see how this is implemented in the project at the end of this chapter.

Why does normalization speed up training?

In non-normalized data the cost function will likely look like squished, elongated bowl. Whereas, when you normalize your features, your cost function will look more symmetric. Look at the figure below of the cost function of two features w_1 and w_2 .

Gradient descent with and without feature scaling

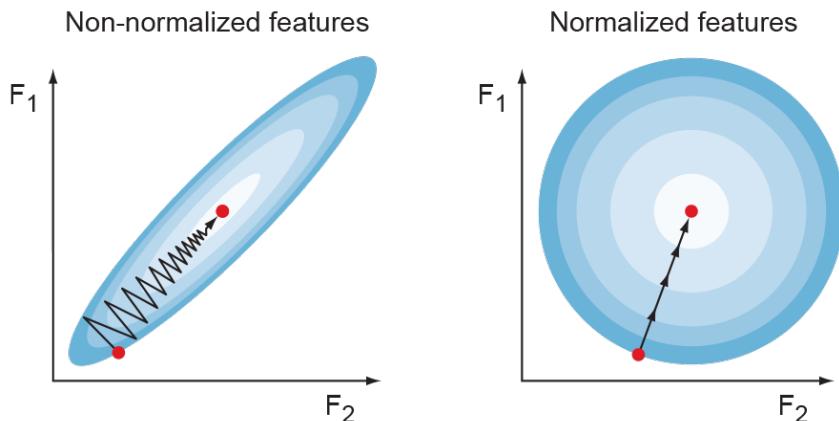


Figure 4.4c

As you can see, for normalized features (on the left), the gradient descent algorithm goes straight forward toward the minimum error, thereby reaching it quickly, whereas for non-normalized features (on the right), it oscillates towards the direction of the minimum error and it ends with a long march down the error mountain. It will eventually reach the minimum, but it will take longer time to converge.

Why does gradient descent oscillate for non-normalized features

If we don't normalize our data, the ranges of our distributions of feature values would likely be different for each feature, and thus the learning rate would cause corrections in each dimension that would differ proportionally from one another. This forces gradient descent to oscillate to the direction of the minimum error and ends up with a longer path down the error.

IMAGE AUGMENTATION

Data augmentation will be discussed in more details later in this chapter when we cover the regularization techniques. But it is important for you to know that this is another pre-processing technique that you have under your toolbelt to use when needed.

4.4 Evaluate the model and interpret its performance (error analysis)

After the first baseline model is established and the data is ready for training, it is time to train the model and measure its performance. After the training is complete, you need to determine bottlenecks in the model performance, diagnose which components are performing worse than expected, and determine whether the poor performance is due to overfitting, underfitting, or a defect in the training data.

One of the main criticisms of neural networks is that they are “black boxes”. Even when they work very well, it is hard to understand why they work so well. Many efforts are being made to improve the interpretability of neural networks, and this field is likely to evolve rapidly in the next few years. In this section, I’ll show you how to diagnose neural networks and analyze their behavior.

4.4.1 Diagnose for overfitting and underfitting

After running your experiment, you want to observe its performance and determine the bottlenecks that is impacting its performance to give you indicators on the areas you need to improve. The main cause of poor performance in machine learning is either overfitting or underfitting the training dataset. This is a very important concept in machine learning. We talked about overfitting and underfitting in chapter 3, I’m going to dive a little deeper here to understand how to detect when the system is fitting the training data too much (overfitting) and when it is too simple to fit the data (underfitting).

- Underfitting means that the model is too simple that it fails to learn the training data. So, it performs poorly on the training datasets. One example of underfitting is using a single perceptron to classify the O and X shapes in the figure below. As you can see, a straight line wouldn’t split our data accurately.
- Overfitting is when the model is too complex for the problem that we are solving. Instead of learning features to fit the training data, it actually memorizes the training data very well. So, it performs very well on the training data but it fails to generalize when tested with new data that it hasn’t seen before. In the figure on the right side, you will see that the model is fitting the data too much so it splits the training data very well but this kind of fitting will fail to generalize.

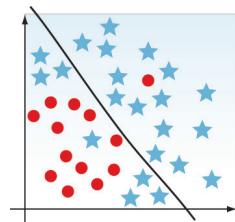


Figure 4.5

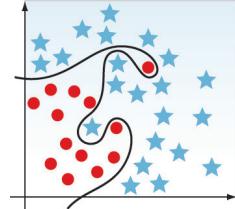


Figure 4.6

- What we want is to build a model that is *just right* for the data. Not too complex to overfit nor too simple to underfit. In the figure below, you will see that the model missed on data sample of the shape O but it looks much more reasonable to generalize on new data.

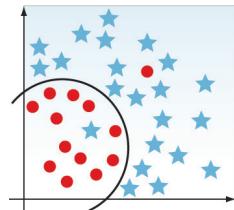


Figure 4.7

The analogy I like to use to explain overfitting and underfitting is similar to a student studying for the exam. Underfitting is when the student doesn't study very well. So it fails the test. Overfitting is when the student memorizes the book that she will answer very well when you ask her questions from the book, but answers poorly when asked questions from outside the book. She failed to generalize. What we want is a student to learn from the book (training data) very well to be able to generalize when asked new questions.

To diagnose for underfitting and overfitting, the two main values that you want to look for while training is the training error and the validation error:

- If the model is doing very well on the training set but it is doing relatively poorly on the validation set, then the model is overfitting. For example, if the train_error = 1% and the val_error = 10%. Then it looks like the model has memorized the training dataset but it is failing to generalize on the validation set. In this case, you might consider tuning your hyperparameters to avoid overfitting and iteratively train, test, and evaluate until you achieve an acceptable performance.
- If the model is performing poorly on the training set, then you are underfitting. For example, if the train_error = 14% and the val_error = 15%. It is clear here that your model might be too simple that it is failing to learn the training set. Maybe you want to consider adding more hidden layers or training longer (more epochs) or try different neural network architectures.

In the next section, we will discuss several hyperparameter tuning techniques to avoid overfitting and underfitting.

Use human-level performance to identify Bayes Error

We talked about achieving a satisfying performance, but how can we know whether this performance is good or not? We need a realistic baseline to compare the training and validation errors to in order to know whether we are improving or not. Of course, ideally, a zero percent error is great. But it is not realistic to target a zero percent error for all problems. In some problems it is impossible to achieve a zero percent error. That is why we need to define a **Bayes Error**.

Bayes error is the best possible error our model can achieve (theoretically). Since humans are usually very good with visual tasks, we can use human-level error as proxy to measure bayes error. For example, if you are working on a relatively simple task like classifying dogs and cats, humans can be very accurate with this. The human error here would be very low, say 0.5%. Then we want to compare our train_error with this value. Which means that if our model accuracy is 95%, that's not satisfying performance and the model might be underfitting. On the other hand, suppose we

are working on a more complex task for humans, like building a medical image classification model for radiologist. The human-error could be a little higher here. Let's say 5%. Then a model that is 95% accurate is actually doing a good job.

Of course this is not to say that deep learning models can never surpass human performance. On the contrary. But it is a good way draw a baseline to gauge whether a model is doing a good job or not. Note that the error percentages above are just arbitrary numbers for the sake of the example.

4.4.2 Plot the learning curves

Instead of looking at the training verbose and comparing the error numbers, one way to diagnose for overfitting and underfitting is to plot your train and val errors throughout the training as you see in the figures below to help you analyze the training performance.

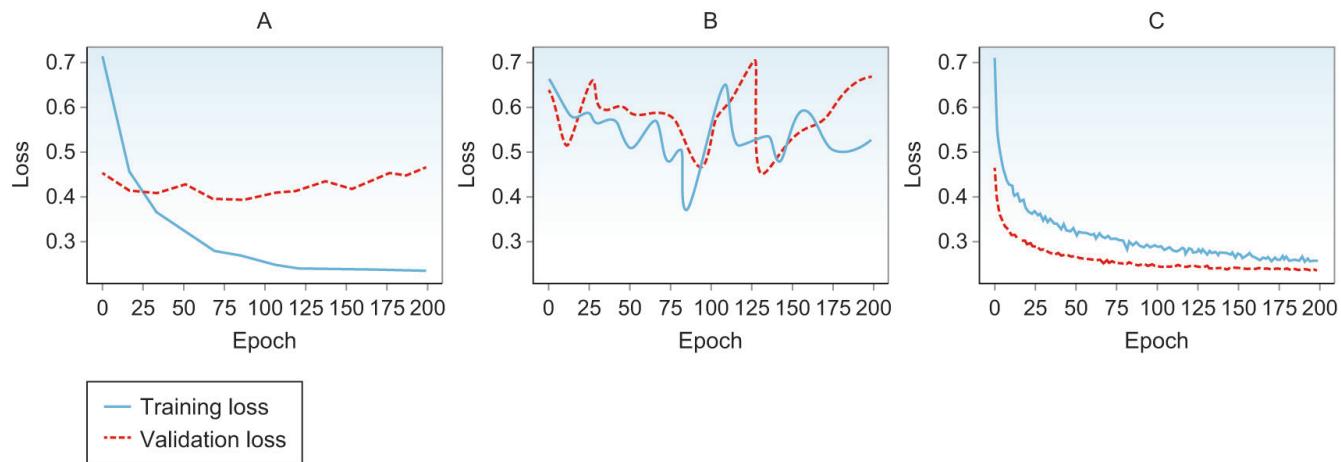


Figure 4.8

Figure A: shows that network is able to improve the loss value (aka learn) on the training data but failed to generalize on the validation data. You can see that learning on the validation data is progressing in the first couple of epochs and then flattens out and maybe decreasing. This is a form of overfitting. Note that this graph shows that the network is actually learning on the training data. This is a good sign that the training is happening. So you don't need to add more hidden units nor you need to build a more complex model. If anything, your network is too complex for the your data because it is learning the training data *too much* that it is actually memorizing it and failing to generalize to new data. In this case, you know that your next step is maybe to collect more data or apply techniques to avoid overfitting.

Figure B: shows that the network is performing poorly on both the training and validation data. In this case, your network is not learning. So you don't need more data because the network is too simple to learn from the data that you already have. Your next step is to build a more complex model.

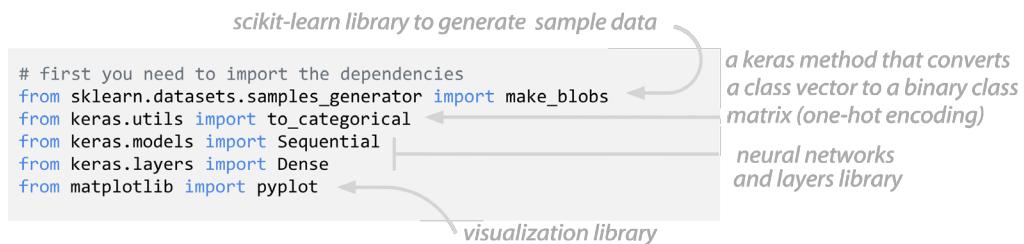
Figure C: shows that the network is doing a good job learning the training data and generalizing to the validation data. This means that there is a good chance that the network will have a good performance out in the wild on the test data.

4.4.3 Exercise: build, train, evaluate a simple network

Before we move on to the hyperparameter tuning section, let's run a quick experiment to see how we split the data, build, train and visualize the model results. You can also see this exercise notebook on www.computervisionbook.com.

In this exercise we will do the following:

- Create toy data for our experiment
- Split the data into 80% training and 20% testing datasets
- Build the MLP neural network
- Train the model
- Evaluate the model
- Visualize the results



```
# use "make_blobs" from scikit-learn to generate a toy dataset of
# only two features and three label classes
X, y = make_blobs(n_samples=1000, centers=3, n_features=2,
cluster_std=2, random_state=2)

# use "to_categorical" from keras to one-hot encode the label
y = to_categorical(y)

# split into 80% training data and 20% test data
# note that we did not create a validation dataset in this example
# for simplicity
n_train = 200
train_X, test_X = X[:n_train, :], X[n_train:, :]
train_y, test_y = y[:n_train], y[n_train:]
print(train_X.shape, test_X.shape)

>> (200, 2) (800, 2)

# develop the model architecture
# here we are building a very simple, two-layer MLP network
model = Sequential()

model.add(Dense(25, input_dim=2 , activation='relu' ))
model.add(Dense(3, activation='softmax' ))
model.compile(loss='categorical_crossentropy' , optimizer='adam'
metrics=['accuracy'])
model.summary()
```

*two input dimensions
because we have two features*

*relu activation function
for hidden layers*

*softmax activation for
output layer with 3 nodes
because we have 3 classes*

*cross entropy loss function
(explained in chapter 2)*

*adam optimizer will be
explained in the next section*

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 25)	75
dense_2 (Dense)	(None, 3)	78
<hr/>		
Total params: 153		
Trainable params: 153		
Non-trainable params: 0		

```
# train the model for 1000 epochs
history = model.fit(train_X, train_y, validation_data=(test_X, test_y), epochs=1000,
verbose=1)

# evaluate the model
_, train_acc = model.evaluate(train_X, train_y)
_, test_acc = model.evaluate(test_X, test_y)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))

>> Train: 0.825, Test: 0.819

# plot Learning curves of model accuracy
```

```
pyplot.plot(history.history['acc'], label='train')
pyplot.plot(history.history['val_acc'], label='test')
pyplot.legend()
pyplot.show()
```

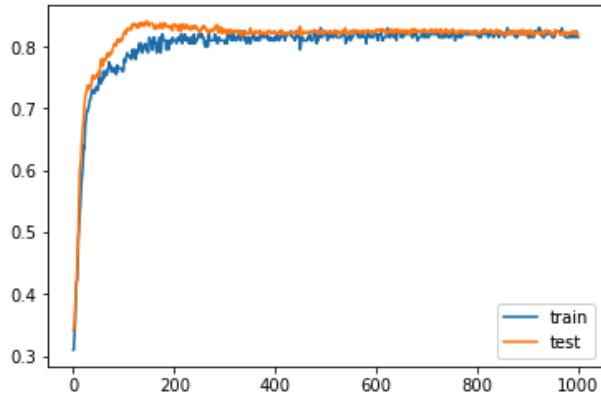


Figure 4.8.b

Network analysis:

Looking at the above learning curve, you can see that both train and test curves are fitting the data with a similar behavior. This means that the network is not overfitting. The network would have been overfitting if the train curve is doing well but the test curve is not. But in this case the network is not overfitting. Could it be underfitting? Maybe. 82% on a very simple dataset like this is considered a poor performance. To improve the performance of this neural network, I would try to build a more complex network and experiment with other underfitting techniques.

4.5 Improve the network and tune hyperparameters

On a high-level, when you run your training experiment and diagnose for overfitting and underfitting, you need to decide whether it is more effective to spend time in tuning the network, clean up and process your data, or go out and collect more data. The last thing you want to do is to spend a few months working on one direction only to find out the it barely improved the network performance. So, before discussing the different hyperparameters to tune, let's answer this question first: Should I collect more data or not?

4.5.1 When to collect more data vs tuning hyperparameters?

Let me say this right now to set the stage for further discussions: **deep neural networks are data hungry.** The overall consensus is that the more data you have, the more you are setting your network up for success. With that in mind, a common behavior by machine learning novices is throwing more data on the learning algorithm as their first solution to improve its performance. But collecting and labeling more data is not always a feasible option. Also, depending on your problem, it could be very costly to collect and label data. Plus, it might not even be that effective.

Image labeling process

While there are some trials to automate some of the data labeling process, at the time of writing, most of labeling is done manually. Especially in computer vision problems. By manually, I mean actual human beings are looking at each image and labeling them one by one (this is called human in the loop). To add another layer of complexity: if you are labeling lung x-ray images to detect a certain tumor for example, you will need qualified physicians to diagnose the images which will cost a lot more than hiring people to classify dogs and cats. What I'm trying to say here is: while collecting more data might be a good solution for some accuracy issues and increase the model's robustness, it is not always a feasible option.

In other scenarios, it is much better to collect more data than to improve the learning algorithm. So, it would be nice if you had quick and effective ways to figure out whether it is better to collect more data or tune the model hyperparameters.

The way I would go about answering this question is as follows:

- Evaluate the model to determine if the performance on the training set is acceptable
- Visualize and observe the performance of these two metrics: training accuracy (train_acc) and validation accuracy (val_acc)
- If the network is yielding poor performance on the training dataset, this is a sign of underfitting. There is no reason to gather more data because the learning algorithm is not using the training data that is already available. Instead, try tuning the hyperparameters or cleaning up the training data
- On the other hand, if the performance on the training set is acceptable and is much worse on the test dataset, then the network is overfitting your training data and failed to generalize to the validation set. In this case, collecting more data could be effective.

The goal, when evaluating the model performance, is to be able to categorize the high-level problem:

- Data problem: you would be better off spending more time on data preprocessing or collecting more data
- Learning algorithm problem: try to tune the network

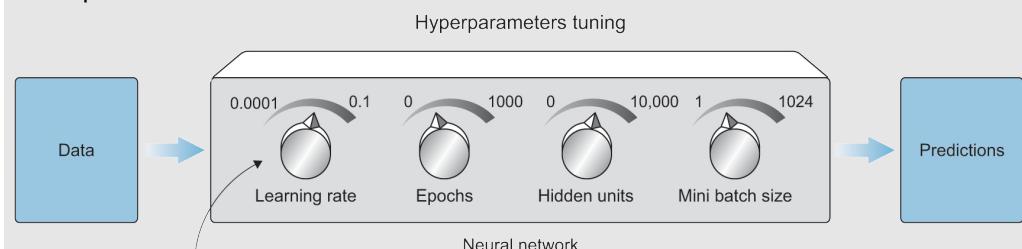
4.5.2 Parameters vs. hyperparameters

First, let's not get **parameters** confused with **hyperparameters**. Hyperparameters are the variables that we set and tune. Whereas, parameters are the variables that the network updates with no direct manipulation from us.

Let's look at the differences:

Parameters	Hyperparameters
These are variables that are <u>learned</u> and updated by the network during the training and <u>we do not touch</u> .	Variables that are <u>not learned</u> by the network. They are set by the machine learning engineer before training the model and tuned to get best results.
In neural networks, the parameters are the weights and biases that are optimized automatically during the backpropagation process to produce the minimum error.	They are variables that define the network structure and determine how the network is trained. Hyperparameter examples are: learning rate, batch size, number of epochs, number of hidden layers, and others that we will explain in the next section.

Think of hyperparameters as knobs of a closed box (the neural network). Our job is to set and tune these knobs to yield the best performance.



The hyperparameters are knobs that act as the network-human interface.
Figure 4.9

4.5.3 Neural networks hyperparameters

As we've seen in chapter 2 and 3, deep learning algorithms come with several hyperparameters that control many aspects of the model's behavior. Some of these

hyperparameters affect the **time and memory cost** of running the algorithm and others affect the **prediction ability** of the model.

The challenge with hyperparameters tuning is that there are no magic numbers that work for every problem. This is what we referred to in chapter 2 as the free-lunch rule. Good hyperparameters values depend on the dataset and the task at hand. Choosing the correct hyperparameters and knowing how to tune requires an understanding of what each hyperparameter does. In this section, we are going to build an intuition on why we would want to nudge a hyperparameter one way or another and I'll propose good starting values for some of the most effective hyperparameters.

Generally speaking, we can categorize the neural network hyperparameters into three main categories:

1. Network architecture:

- o Number of hidden layers (network depth)
- o Number of neurons in each layer (layer width)
- o Activation type

2. Learning and optimization:

- o Learning rate and decay schedule
- o Minibatch size
- o Optimization algorithms
- o Number of training iterations or epochs (and early stopping criteria)

3. Regularization techniques to avoid overfitting:

- o L2 regularization
- o Dropout layers
- o Data augmentation

We have discussed all of the above hyperparameters in chapters 2 and 3 except the regularization techniques. Next, we are going to go through them quickly with the focus on understanding what happens when we tune each knob up or down and how to know which hyperparameter to tune.

4.5.4 Network architecture

First, let's talk about the hyperparameters that define the neural network architecture:

- the number of hidden layers (representing the network depth),
- the number of neurons in each layer - also known as the hidden units (representing the network width),
- and the activation functions.

Depth and width of the neural network

Whether you are designing an MLP, CNN or other neural networks, you need to decide the number of hidden layers in your network (depth) and the number of neurons in each layer (width). The number of hidden layers and units describe the learning capacity of the network. The main goal here is to set the number of units to be “large enough” for the network to learn the data features. A smaller network might underfit and a larger network might overfit. To know what is “large enough” network, you would want to pick a starting point and observe the performance then tune up or down.

The more complex the dataset, the more learning capacity the model will need to learn its features. Take a look at the three datasets below:

Very simple dataset

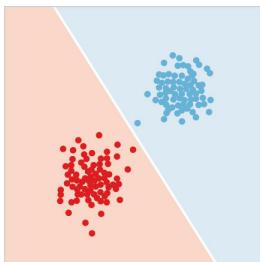


Figure 4.10 Can be separated by a single perceptron.

Medium complexity dataset

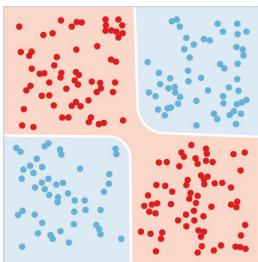


Figure 4.11 Can be separated by adding a few more neurons

Complex dataset

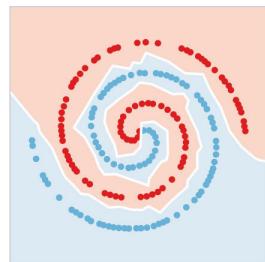


Figure 4.12 Needs a lot of neurons to separate the data

If you provide the model with too much learning capacity (i.e. too many hidden units), it might tend to overfit the data and tend to memorize the training set. If you find that your model is overfitting the training data, you might want to decrease the number of hidden units.

As a general rule, it is usually better to add more hidden neurons until the validation error is no longer improving. The trade off is that it is computationally expensive to train deeper networks. Having a small amount of units may lead to underfitting while having more units are usually not harmful with appropriate regularization (like dropout and others discussed later in this chapter).

Try playing with around with [Tensorflow's Playground](#) to get a better intuition. Try to experiment with different architectures and gradually add more layers and more units in hidden layers while observing the network learning behavior.

Activation type

Activation functions introduce nonlinearity to our neurons. Without activations, our neurons would be passing linear combinations (weighted sum) to each other and not solve any nonlinear problems. Activation functions are discussed extensively in chapter 2. This is a very active area of research. Every few weeks we are introduced to new types of activations and there are many of them that you can use. But as the time of writing these pages, it has

proven that ReLU functions and its variations (like Leaky ReLU) perform the best in the hidden layers. And in the output layer, it is very common to use Softmax function for classification problems with a number of neurons = the number of classes in your problem.

Pro Tip:

When considering the number of hidden layers and units in your neural network architecture, it is useful to think of this in terms of the number of parameters in the network and their effect on the computational complexity. The more neurons in your network, the more parameters that the network has to optimize (In chapter 3, we learned how to print the model summary to see the total number of parameters that will be trained).

Based on your hardware setup for the training process (computational power and memory) you can determine whether you need to reduce the number of parameters. To reduce the number training parameters of the network you can do one of the following:

- Reduce the depth and width of the network (hidden layers and units) this will reduce the number of training parameters, hence, reduce the neural network complexity
- Or, add POOL layers or tweaking the strides and padding of the CONV layer to reduce the feature maps dimensions = lower number of parameters

These are just examples to help you see how you will look at the number of training parameters in real projects and the tradeoffs that you will need to make. Complex networks = large number of training params = high computational power and memory.

The best way to build your baseline architecture is to look at literature of the popular architectures that are built to solve specific problems and start from there, evaluate its performance, tune its hyperparameters, and repeat. We saw how we were inspired by AlexNet to design our CNN in the image classification project in chapter 3. In the next chapter, we will explain some of the most popular CNN architectures like LeNet, AlexNet, VGG, ResNet and Inception.

4.5.5 Learning and optimization

Now, that we have built our network architecture, it is time to discuss the hyperparameters that determine how the network learns and optimizes its parameter to achieve the minimum error.

LEARNING RATE AND DECAY SCHEDULE

"The learning rate is the single most important hyperparameter and one should always make sure that is has been tuned. If there is only time to optimize one hyperparameter, then this is the hyperparameter that is worth tuning."

- Yoshua Bengio.

Learning rate "lr" intuition

The learning rate has been discussed extensively in chapter 2 in the optimizers section. As a refresher, let's remember how gradient descent (GD) works. The GD optimizer searches for the optimal values of weights that yield the lowest error possible. When setting up our optimizer, we need to define the step size that it takes when it is descending the error mountain. This step size is the learning rate. It represents how fast or slow the optimizer is

descending the error curve. When we plot the cost function with only one weight, we get the oversimplified u-curve below where the weight is randomly initialized at a point on the curve.

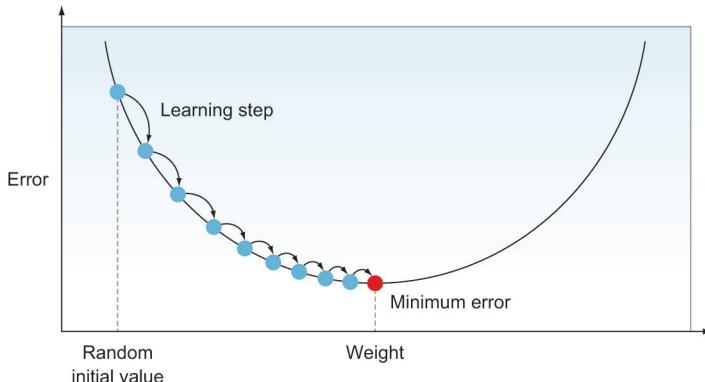


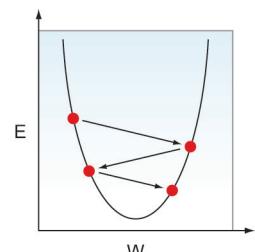
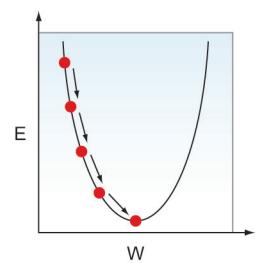
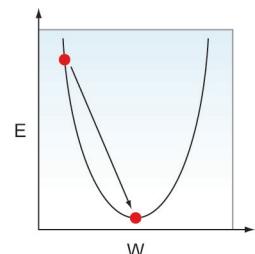
Figure 4.13

The GD calculates the gradient to find the direction that reduces the error (derivative). In the above example, the descending direction is to the right. The GD starts taking steps down after each iteration (epoch). Now, as you can see in the figure on the right, if we make a *miraculously* correct choice of the learning rate value, then we would land on the best weight value that minimizes the error in only one step. This is an impossible case that I'm just using for elaboration purpose. Let's call this the **ideal "lr" value**.

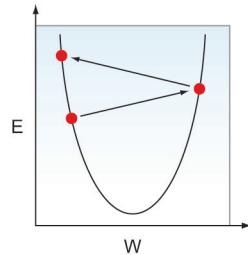
If the learning rate is **smaller** than the ideal "lr" value, then the model can continue to learn by taking smaller steps down the error curve until it finds the most optimal value for the weight.

Much smaller means that it will eventually converge but will take a longer time.

If the learning rate is **larger** than the ideal "lr" value, the optimizer will overshoot the optimal weight value in the first step, then overshoot again on the other side in the next step. This could possibly yield a lower error than what we started with and converges to a reasonable value, but not the lowest error that we are trying to reach.



If the lr is **much larger** than the ideal “lr” value (more than twice as much), the optimizer will not only overshoots the ideal weight, but it gets farther and farther from the lowest error. This phenomenon is called *divergence*.



Too high vs too low learning rate

Setting the learning rate high or low is a tradeoff between the optimizer speed vs performance. Too low “lr” requires many epochs to converge. Often times too many. Theoretically, if the lr is too small, the algorithm is guaranteed to eventually converge if kept running for infinity of time. On the other hand, too high lr might get us to a lower error value faster because we are taking bigger steps down the error curve, but there is a better chance that the algorithm will oscillate and diverge away from the minimum value. So, ideally, we want to pick the lr that is **Just right** (optimal) that swiftly reaches the minimum point. Not too slow and without being too big that it might diverge.

When plotting the loss value against the number of training iterations (epochs), you will notice the following:

- **Much smaller lr:** the loss keeps decreasing but needs a lot more time to converge
- **Larger lr:** the loss achieves a better value than what we started with, but still far from optimal
- **Much larger lr:** the loss might decrease in the beginning but starts to increase as the weight values get farther and farther away from the optimal values
- **Good lr:** the loss decreases consistently until it reaches the minimum possible value

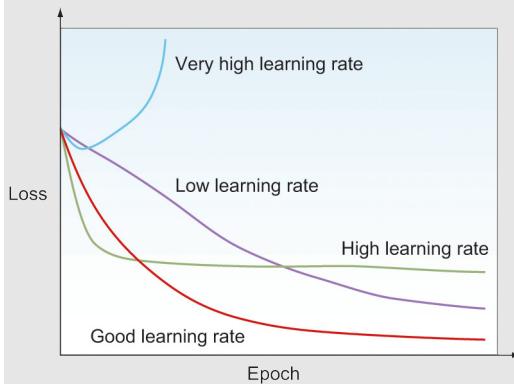


Figure 4.16

A systematic approach to find the optimal learning rate

The optimal learning rate will be dependent on the topology of your loss landscape, which is in turn dependent on both your model architecture and your dataset. Whether you are using Keras, Tensorflow, PyTorch or any other deep learning (DL) library, using the default learning rate value of the optimizer is a good start to provide decent results. Each optimizer type has its own default value. Read the documentation of the DL library that you are using to find out

the default value of your optimizer. If your model doesn't train well, you can play around the "lr" variable with these usual suspects: 0.1, 0.01, 0.001, 0.0001, 0.00001, and 0.000001 to improve the performance or speed up training by searching for an optimal learning rate.

The way to debug this is to look at the validation loss values in the training verbose:

- If the val_loss is decreasing after each step, that's good. Keep training until it stops improving.
- If the training is complete and the val_loss is still decreasing, then maybe the lr was too small that it didn't converge yet. In this case, you can do one of two things:
 1. Train again with the same lr but with more training iterations (epochs) to give the optimizer more time to converge
 2. Increase the lr value a little and train again
- If val_loss started to increase or oscillate up and down, then the lr is too high and you need to decrease its value.

Learning rate decay and adaptive learning

Finding the learning rate value that is just right for your problem is an iterative process. You start with a static "lr" value, wait until training is complete, evaluate, then tune. Another way to go about tuning your learning rate, is to set a learning decay. Learning rate decay is a method by which the learning rate changes **during** training. It often performs better than a static value, and drastically reduces the time required to get optimal results.

By now, it's been made clear that when we try lower learning values, we have a better chance to get to a lower error point. But it will take longer time to train. In some cases, too long that it becomes not feasible. A good trick is to implement a decay in our learning rate. The decay simply means that we are telling our network to automatically decrease the "lr" throughout the training process. For example, we can decrease the "lr" by a constant value of (x) for each (n) number of steps. This way, it can start with the higher value to take bigger steps towards the minimum, then gradually decreases the learning rate every (n) epochs to avoid overshooting the ideal "lr".

Linear decay - one way to do this is by reducing the learning rate linearly. For example, you can decrease it by half every 5 epochs. Like the example below:

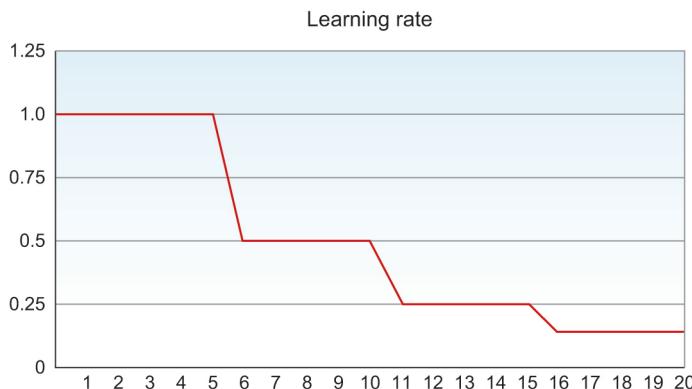


Figure 4.17

Exponential decay - another way is to decrease the “lr” exponentially. For example, you can multiply the “lr” by 0.1 every 8 epochs. Clearly this way the network is going to converge a lot slower than the linear decay but it is eventually going to converge.

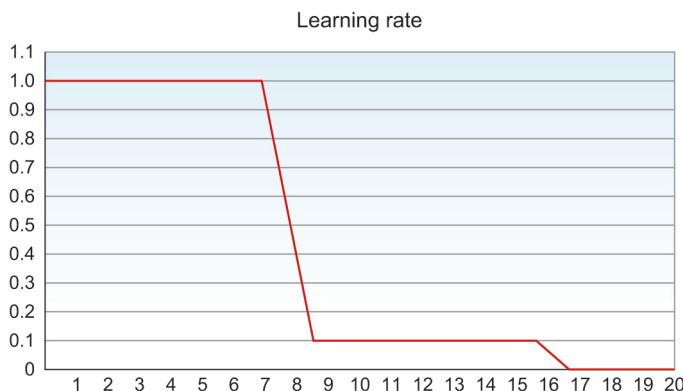


Figure 4.18

Adaptive learning - there are more clever learning algorithms that have an adaptive learning rate. These algorithms use a heuristic way that automatically updates the “lr” when the training stops. This means, not only decreasing the “lr” when needed, but also increasing it when it notices that the improvements are too slow (i.e. too small lr). Adaptive learning usually works better than other learning rate setting strategies. An example of adaptive learning optimizers are Adam and Adagrad. More on adaptive optimizers later in this chapter.

MINIBATCH SIZE

Minibatch size is another hyperparameter that you need to set and tune in the optimizer algorithm. The batch_size hyperparameter has a big effect on the resources requirements of the training process and the training speed.

In order to understand the minibatch, let's back up to the three gradient descent types that we explained in chapter 2. Batch, stochastic, and mini-batch gradient descent.

- 1) **Batch gradient descent (BGD):** we feed the entire data to the network all at once + apply the feedforward process + calculate the error + calculate the gradient and backpropagate to update the weights. In here the optimizer calculates the gradient by looking at the error generated after it sees at all the training data and the weights are updated only once after each epoch. So, in this case, the minibatch size = the entire training data. The main advantage of BGD, is that it has relatively low noise and larger steps towards the minimum (see figure below). The main disadvantage is that it can take too long to process the entire training data at each step, especially when training on large data. BGD will also require a huge memory for training large datasets that might not be available. BGD might be a good option if you are training on a small dataset.

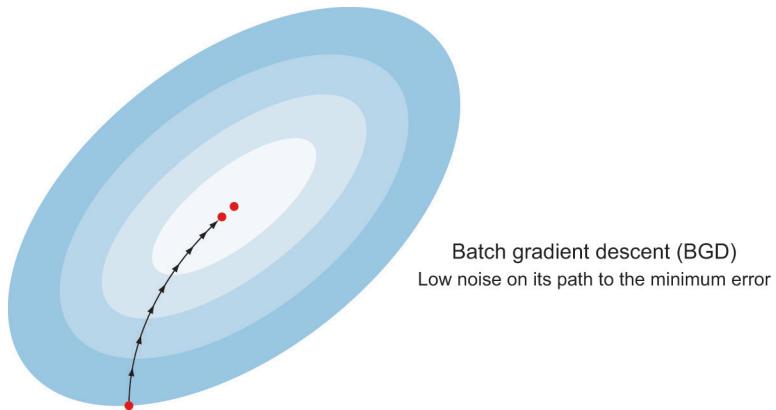


Figure 4.19

- 2) **Stochastic gradient descent (SGD):** It is also called online learning. We feed the network a single instance of the training data at a time and use this one instance to do the forward pass + calculate error + calculate the gradient and backpropagate to update the weights. In SGD, the weights are updated after it sees each single instance (as opposed to processing the entire dataset before each step for BGD). SGD can be extremely noisy as it oscillates on its way to the global minimum because it takes a step down after each single instance which could sometimes hit the wrong direction. This noise can be reduced by just using a smaller learning rate so on average it will

take you in a good direction and almost always performs better than BGD. With SGD you get to make progress quickly and usually reaches very close to the global minimum. The main disadvantage is that by calculating the GD for one instance at a time, you lose the speed up from not utilizing matrix multiplication in the training calculations.

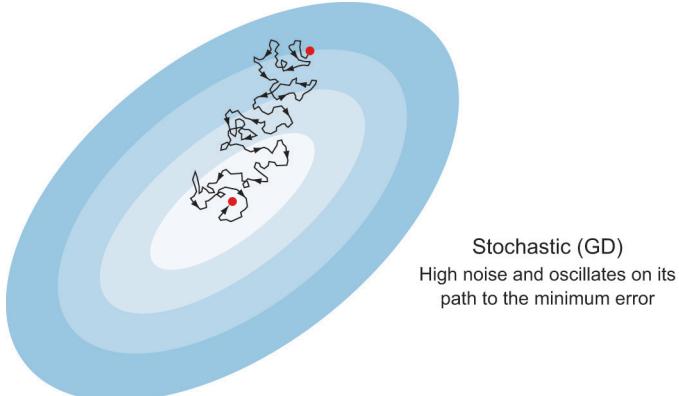


Figure 4.20

Batch and Stochastic GD recap:

Okay, so far we learned that, on one extreme, if you set your minibatch size to 1 (i.e. stochastic training), the optimizer will take a step down the error curve after computing the gradient for each single instance of the training data which is good but loses the speed up from not utilizing matrix multiplication. On the other extreme, if your minibatch = the number of your entire training data, then you are using BGD. It takes too long to make a step towards the minimum error when processing large datasets. Between the two extremes, there is **Minibatch gradient descent**.

- 3) **Minibatch gradient descent (MN-GD):** it is a compromise between both batch and stochastic GD. Instead of computing the gradient from 1 sample (SGD) or all training samples (BGD), we divide the training sample into *minibatches* to compute the gradient from. This way, we can take advantage of the matrix multiplication to speed up training and also start making progress faster instead of having to wait to training the entire training set.

Guidelines to choose minibatch size

First, if you have a small dataset (i.e. < 2,000), you might be better off just using BGD. You will be able to train the whole dataset quite fast.

For large datasets, you can use a whole scale of minibatch size values. A typical starting value for the minibatch is 64 or 128. You can then tune it up and down on this scale: 32, 64, 128, 256, 512, 1024, and keep doubling it as needed to speed up training. But make sure that your minibatch size fits in your CPU/GPU memory. Minibatch sizes of 1024 and larger are possible but it's quite rare. A larger minibatch size allows computational boost that utilizes matrix

multiplication in the training calculations. But that comes at the expense of needing more memory for the training process and generally more computational resources. The diagram below shows the relationship between the batch size, computational resources, and number of epochs needed for the neural network training.

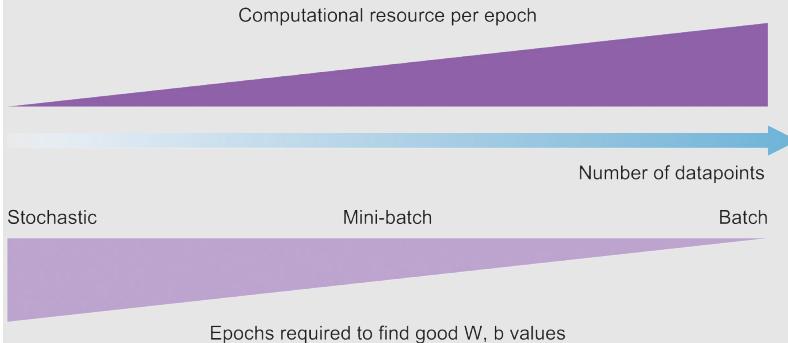


Figure 4.21

OPTIMIZATION ALGORITHMS

During the history of deep learning, many researchers proposed optimization algorithms and showed that they work well in some problems. But most of those optimization algorithms subsequently shown not to really generalize well to the wide range of neural networks that we might want to train. So over time, the deep learning community felt that the gradient descent algorithm and some of its variants work really well. So far, we discussed batch, stochastic, and mini-batch gradient descent.

We also learned that choosing a proper learning rate can be challenging because a too small learning rate leads to painfully slow convergence, while too large learning rate can hinder convergence and cause the loss function to fluctuate around the minimum or even diverge. We need more creative solutions to further optimize gradient descents.

Optimizers types are explained very well in the documentation of most deep learning frameworks. In this section, I'll explain the intuition of two of the most popular **gradient-descent-based** optimizers: Momentum and Adam that have really stood up and have been shown to work well across a wide range of deep learning architectures. This will help you build a good foundation to dive deeper on other optimization algorithms. If you are interested, you can read this paper for more optimization algorithms: <https://arxiv.org/pdf/1609.04747.pdf>

Gradient descent with Momentum

We recall from before that SGD ends up with some oscillations in the vertical direction towards the minimum error. These oscillations slow down the convergence process and make it harder to use larger learning rates. Because if you use larger learning rates, your algorithm might end up overshooting and diverging.

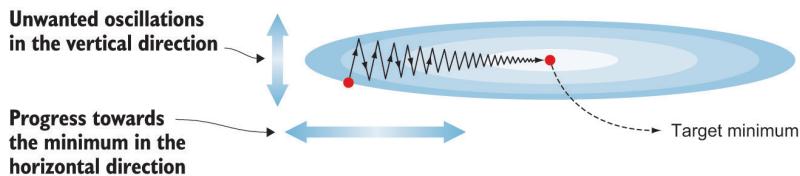


Figure 4.22

To reduce these oscillations, a technique called momentum was invented that lets the gradient descent navigate along the relevant directions and softens the oscillation in the irrelevant directions. In other words, it makes the learning slower in the vertical direction oscillations and faster in the horizontal direction progress which will help the optimizer reach the target minimum much faster.

Similar to the idea of momentum from classical physics, when a snowball falls down the hill, it accumulates momentum, becoming faster and faster. In the same way, our momentum term increases for dimensions whose gradient point in the same direction and reduces updates for dimensions whose gradients change direction. This leads to: 1) a faster convergence, and 2) a reduced oscillations.

How the math works in Momentum

The math here is really simple and straightforward. The momentum is built by adding a **velocity** term to the equation that is updating the weight:

$$w_{new} = w_{old} - \alpha \frac{dE}{dw_i} \quad \xleftarrow{\hspace{10em}} \quad \text{Original update rule}$$

$$W_{new} = W_{old} - \text{learning rate} * \text{gradient} + \text{velocity term} \quad \xleftarrow{\hspace{10em}} \quad \text{new rule after adding velocity}$$

The velocity term here equals the weighted average of the past gradients.

Adam

Stands for *adaptive moment estimation*. Adam also keeps an exponentially decaying average of past gradients, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with **friction** to slow down the momentum and control it.

Adam optimizer usually outperforms the other optimizers. Because it helps train a neural network model much more quickly than the techniques we have seen earlier.

Adam hyperparameters:

Yes, once again new hyperparameters to tune. But the good news is, the default values of major deep learning frameworks often work well, so you may not need to tune at all, except for the learning rate which is not an Adam-specific hyperparameter.

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0)
```

The authors of Adam propose the default values :

- Learning rate α : needs to be tuned
- Momentum term β_1 : common choice is 0.9
- RMSprop term β_2 : common choice is 0.999
- $\epsilon: 10^{-8}$

NUMBER OF TRAINING ITERATIONS (EPOCHS) AND EARLY STOPPING CRITERIA

A training iteration or an epoch is when the model goes a full cycle and sees the entire training dataset once. In other words, the epoch hyperparameter is set to define how many iterations we want to keep our network training. The more training iterations the model takes during training, the more it learns the features of your training data. In fact, too many iterations might lead to overfitting. To diagnose whether your network needs more or less training epochs, keep your eyes on the training and validation error values.

The intuitive way to think about this is that we want to keep the network training as long as the error value is decreasing. Correct? Let's take a look at this sample verbose from a network training:

```
Epoch 1, Training Error: 5.4353, Validation Error: 5.6394
Epoch 2, Training Error: 5.1364, Validation Error: 5.2216
Epoch 3, Training Error: 4.7343, Validation Error: 4.8337
Epoch 4, Training Error: 4.1362, Validation Error: 4.5394
Epoch 5, Training Error: 3.9243, Validation Error: 4.0394
```

You can see that both the train and val errors are decreasing. This means that the network is still learning. It doesn't make sense to stop the training at this point. The network is clearly still making progress towards the minimum error. Let's let it train for 6 more epochs and observe the results:

```
Epoch 6, Training Error: 3.7312, Validation Error: 3.8324
Epoch 7, Training Error: 3.5324, Validation Error: 3.7215
Epoch 8, Training Error: 3.7343, Validation Error: 3.8337
Epoch 9, Training Error: 3.2152, Validation Error: 3.6252
Epoch 10, Training Error: 2.9243, Validation Error: 4.0231
Epoch 11, Training Error: 2.8923, Validation Error: 4.1764
```

It looks like the training error is doing good and still improving. That's good. This means that the network is improving on the training set. However, if you look at epoch #8, you will see that the val_error stopped improving and started to increase after that. Improving on the train_error while not improving on the validation_error means that the network is starting to overfit the training data and failing to generalize to the validation data.

Let's plot the training and validation errors:

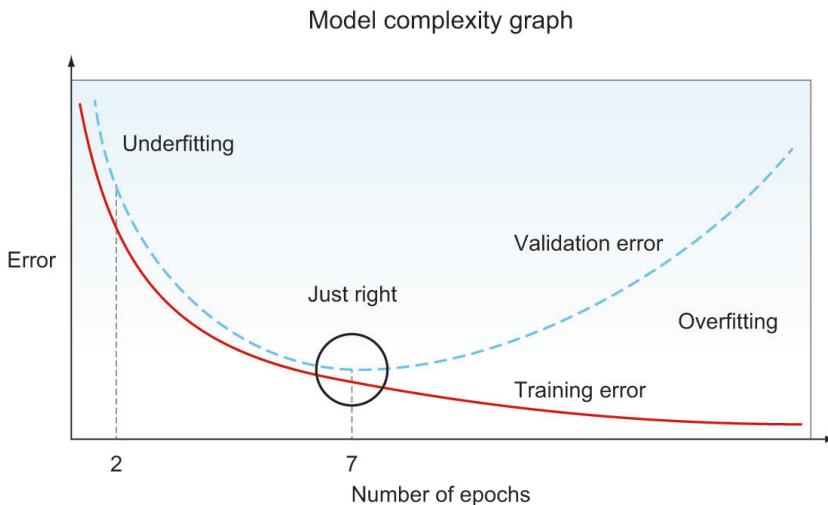


Figure 4.23

You can see that both training and validation errors were improving at first, but then the val error started to increase leading to overfitting. We need to find a way to stop the training right before it starts to overfit. This technique is called **Early Stopping**.

Early stopping

It is an algorithm that is widely used when training neural networks to determine the right time to stop the training process before it starts to overfit the training data. The way it works is really simple, it monitors the validation error value and it stops the training when the value starts to increase.

Here is the early stopping function in Keras:

```
EarlyStopping(monitor='val_loss', min_delta=0, patience=20)
```

The early stopping function takes the following arguments:

- monitor: this is the metric that you want to monitor during the training. Usually we want to keep an eye on val_loss because it represents our internal testing on the model performance. If the network is doing well on the validation data, it will probably do well on test data and production.

- min_delta: the minimum change that qualify as an improvement. There is no standard value for this variable. To decide the min_delta value, run a few epochs and see the change in error & validation accuracy. Depending on the rate of change, it should be defined. The default value 0 works pretty well in many cases.
- Patience: this variable tells the algorithm how many epochs it should to wait before stopping the training if the error does not improve. For example, if we set patience = 1, the training will stop at the epoch where the error increases. We must be a little flexible though when setting the patience value because it is very common for the error to oscillate a little and continue improving. We can stop the training if it didn't improve in the last 10 or 20 epochs.

The good thing about early stopping is that it allows you to worry less about the epochs hyperparameter. You can set a high number of epochs and let the stopping algorithm take care of stopping the training when no improvements to the error are happening.

4.5.6 Regularization techniques to avoid overfitting

If you observe that your neural network is overfitting the training data, this means that your network might be too complex for the training data and needs to be simplified. One of the first things you should try to reduce your network complexity is regularization. In this section, we will discuss three of the most common regularization techniques: L2 regularization, dropout, and data augmentation.

5.3.1 L2 REGULARIZATION

The basic idea of L2 regularization is that it penalizes the error function by adding a "regularization term" to it. Which in turn reduces the weight values at the hidden units and make them too small, very close to zero, to help simplify the model.

Let's see how regularization works:

First, we update the error function by adding another term known as the regularization term.

$$\text{error function}_{\text{new}}) = \text{error function}_{\text{old}} + \text{regularization term}$$

Note that you can use any of the error functions explained in chapter 2 like MSE or cross entropy.

Now, let's take a look at the regularization term:

$$\text{L2 regularization term} = \lambda/2m * \sum ||w||^2$$

Where lambda (λ) is the regularization parameter, m is the number of instances, and w is the weight.

The updated error function:

$$\text{error function new} = \text{error function old} + \lambda/2m * \sum ||w||^2$$

Why does L2 regularization reduce overfitting?

Well, let's look at how the weights are updated during the backpropagation process. We learned from chapter 2 that the optimizer calculates the derivative of the error, multiplies it by the learning rate, and subtract this value from the old weight. Here is the backpropagation equation that updates the weights:

$$W_{\text{new}} = W_{\text{old}} - \alpha \left(\frac{\partial \text{Error}}{\partial W_x} \right)$$

↑ New weight ↑ Learning rate ↑ Old weight ↑ Derivative of Error with respect to weight

Now, since we add the regularization term to the error function, then the new error becomes bigger than the old error. Which also means that its derivative ($d\text{Error}/dW_x$) is bigger. This leads to a smaller W_{old} . L2 regularization is also known as **weight decay** as it forces the weights to decay towards zero (but not exactly zero).

Reducing weights leads to a simpler neural networks

To see the intuition of this let's look at this example, if the regularization term is very large that when multiplied to the learning rate will be equal to W_{old} , then this will make the new weight equals to zero. This cancels the effect of this neuron, leading to a simpler neural network with less neurons.

In practice, L2 regularization does not make the weights equal to zero. It just makes them smaller to reduce their effect. Large regularization parameter (λ), leads to negligible weights. When the weights are negligible, the model will not learn much from these units. This will end up making the network simpler and thus reduce overfitting:

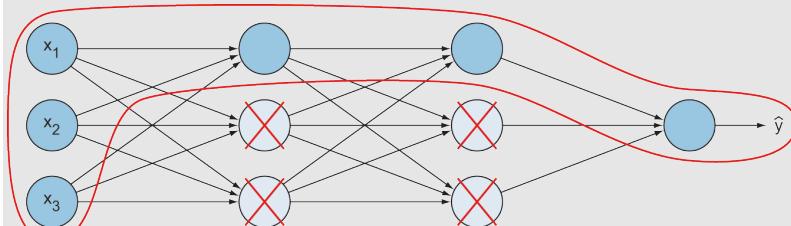


Figure 4.27

L2 regularization in Keras

```
# When adding a hidden layer to your network, add the kernel_regularization argument
# with L2 regularizer as follows:
model.add(Dense(units=16, kernel_regularizer=regularizers.l2(λ),
                activation='relu'))
```

The lambda value is a hyperparameter that you can tune. The default value of your deep learning library usually works well. If you still see signs of overfitting, increase the lambda hyperparameter to reduce the model complexity.

DROPOUT LAYERS

Dropout is another regularization technique that is very effective to simplify the complexity of the neural network to avoid overfitting the training set. We discussed dropouts extensively in chapter 3. As a refresher, the dropout algorithm is fairly simple:: at every training iteration, every neuron has a probability p of being temporarily ignored (dropped out) during this training iteration. This means that it may be active during the next iterations. While it is counterintuitive to intentionally pause the learning on some of the network neurons but it is quite surprising how well this technique works. The probability p is a hyperparameter that is called *dropout rate* and is typically set to 0.3 to 0.5 range. Start with 0.3 and if you see signs of overfitting, increase the rate.

I like to think of dropout as tossing a coin every morning with your team to decide which one will do a specific critical task. After a few iterations, all your team members will learn how to do this task and not rely on a single member to get it done. The team would become much more resilient to changes.

Both L2 regularization and dropout aim to reduce the network complexity by reducing its neurons' effectiveness. The difference is that dropout completely cancels the effect of some neurons with every iteration. While, L2 regularization just reduces the weight values to reduce the neurons effectiveness. Both lead to a more robust, resilient neural network and reduce overfitting. It is recommended that you use both types of regularization techniques together in your network.

DATA AUGMENTATION

One way to avoid overfitting is to get more data. Since getting more data is not always a feasible option, we can augment our training data by generating new instances of these images with some transformations. Data augmentation can be an inexpensive way to give your learning algorithm more training data and therefore reduce overfitting.

There are a lot of image augmentation techniques that you can use. To mention a few: flipping, rotation, scaling, zooming, lighting condition, and many other transformations that you can apply to your dataset to provide the learning algorithm a variety of images to train on. In the figure below, you can see some of the transformation techniques applied to an image of the digit 6.

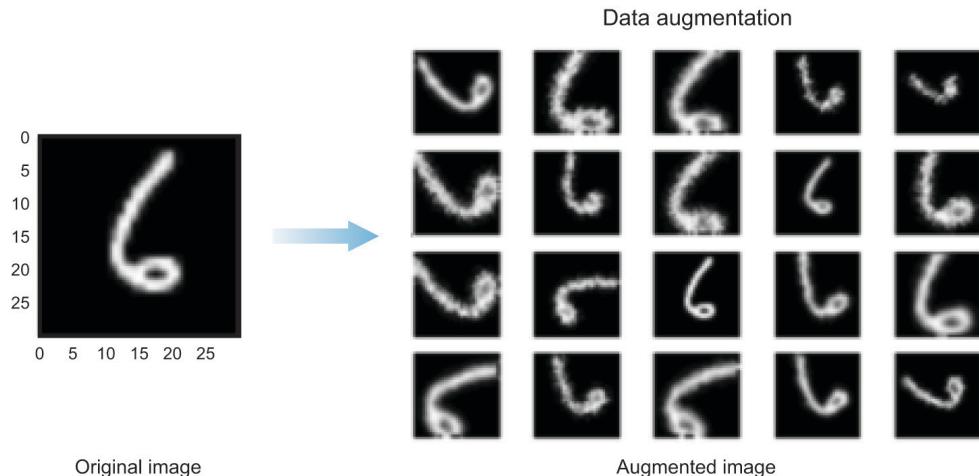


Figure 4.28

In the above figure, we created 20 new images that the network can learn from. The main advantages of synthesizing images like the example above is: A) now you have more data (20x), B) tells your algorithm that if this image is a digit 6, then flipping it vertically, horizontally, and rotating it is still a digit 6. This makes the model more robust to detect the number 6 in any form and shape.

Data augmentation is considered a regularization technique, because allowing the network to see a lot more variants of the object reduces its dependence on the original form of the object when its learning its features. Which makes the network more resilient when tested on new data.

Data augmentation in Keras:

```
# import ImageDataGenerator from Keras
from keras.preprocessing.image import ImageDataGenerator

# generate batches of new image data
# ImageDataGenerator takes transformation types as arguments
# in this example, we set horizontal and vertical flip to True
# read Keras documentation (or your deep Learning library) for more
# transformation arguments
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)

# compute the data augmentation on the training set
datagen.fit(training_set)
```

4.6 Batch normalization (BN)

Earlier in this chapter, we explained the concept of data normalization to speed up learning. The normalization techniques that we discussed earlier were focused on preprocessing the training set before feeding it to the input layer. If the input layer is benefiting from normalization, why not do the same thing for the **extracted features** in the hidden units, that are changing all the time, and get much more improvement in the training speed and the network resilience.

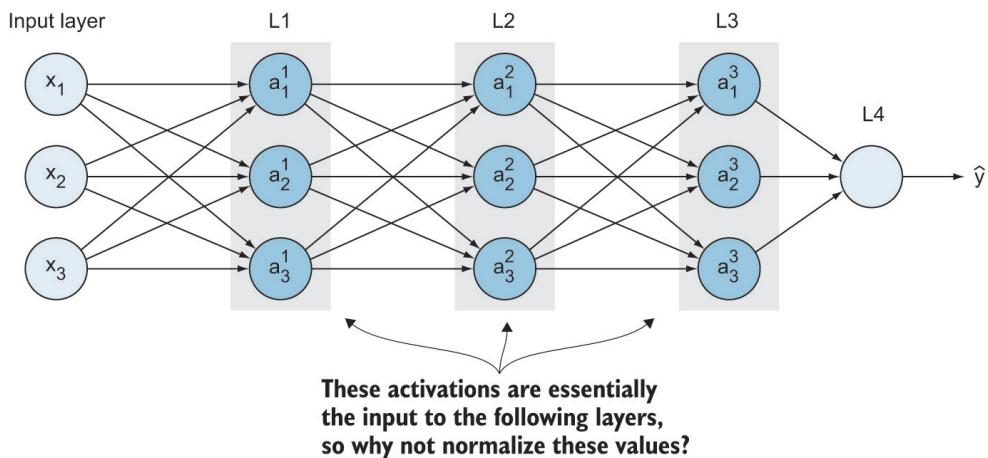


Figure 4.29

4.6.1 The covariance shift problem

Before we define covariance shift, let's take a look at this example to gain a better intuition of the problem that the batch normalization is trying to solve: Suppose you are building a cat classifier and you train your algorithm on images of white cats only. When you test this classifier on other cat images with different colors, it will not perform well. Why? Because the model has been trained on a training set with a specific distribution (white cats). When the distribution changed in the test set, it confused the model. See graphs A and B below.

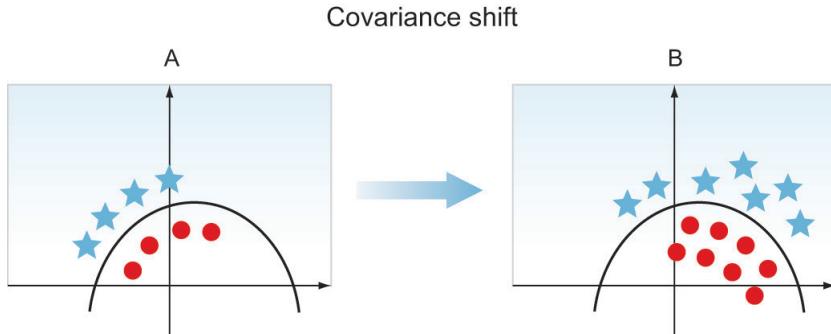


Figure 4.30

Graph A is the training set of only white cats and graph B is the testing set with cats with different colors. Where the circles are the cat images and the x is the non-cat images.

Now, you should not expect that the model that was trained on the data in graph A will do very well in the new distribution in B. The idea of the change of data distribution goes by the fancy name: **covariance shift**.

Covariance shift: if the model is learning to map X dataset to its label y. When the distribution of X changes then you might need to retrain your learning algorithm.

4.6.2 Covariance shift in neural networks

To understand how the covariance shift happens in neural networks, consider the follow simple 4-layer MLP:

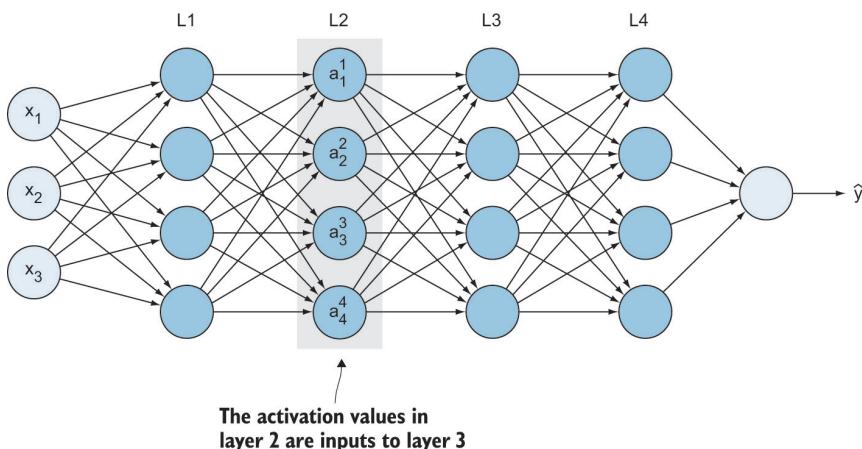


Figure 4.31

Let's look at the network from the third layer (L3) perspective. Its input are the activation values in L2 ($a_1^2, a_2^2, a_3^2, \text{and } a_4^2$) which are the features that are extracted from the previous layers. The main goal that L3 is trying to achieve is to map these inputs to $y_{\hat{}}$ to make it as close as possible to the label y . Now, the third layer is doing that while the network is adapting the values of the parameters from the previous layers as well. Because as the parameters (w, b) are changing in layer 1, the activation values in the second layer are changing too. So from the perspective of the third hidden layer, the values of the second hidden layers are changing all the time suffering from the problem of the covariance shift. So, what batch norm does is: it reduces the amount of change in the distribution of these hidden unit value. Causing these values to become more stable so that the later layers of the neural network has more firm ground to stand on.

It is important to call out that BN does not cancel or reduce the change in the hidden units values. What it does is, it ensures that the distribution of that change remains the same. So even if the exact values of the units change, the mean and variance do not change.

4.6.3 How does batch normalization work?

In their 2015 paper ([Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#)), Sergey Ioffe and Christian Szegedy proposed the batch normalization technique to reduce the covariance shift. The way batch normalization works is that it adds an operation in the neural network just before the activation function of each layer to do the following:

- 1) Zero-center the inputs,
- 2) normalize the results,
- 3) then scale and shift the results

This operation lets the model learn the optimal scale and mean of the inputs for each layer.

How the math works in batch normalization

1. To zero-center the inputs, the algorithm needs to calculate the input mean and standard deviation (the input here means means the current minibatch, hence the name batch normalization):

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad \xleftarrow{\hspace{1cm}} \text{minibatch mean}$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad \xleftarrow{\hspace{1cm}} \text{minibatch variance}$$

Where m is the number of instances in the minibatch, μ_B is mean, and σ_B is the standard deviation over the current minibatch.

2. Normalize the input

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \longleftarrow \text{normalize}$$

Where \hat{x}_i is the zero-centered and normalized inputs. Note that there is a variable here that we added (ϵ). This is a tiny number (typically 10^{-5}) to avoid the division by zero if σ is zero in some estimates.

3. Scale and shift the results

Now, we multiply the normalized output by a variable γ to scale it and add (β) to shift it.

$$y_i = \gamma \hat{x}_i + \beta \quad \longleftarrow \text{scale and shift}$$

Where y_i is the output scaled and shifted of the BN operation.

Notice that BN introduces two new learnable parameters to the network γ and β . So, our optimization algorithm will update the parameters of γ and β just like it updates weights and biases. In practice, this means that you may find that training is rather slow at first while gradient descent is searching for the optimal scales and offsets for each layer, but it accelerates once it's found reasonably good values.

4.6.4 Batch normalization implementation in Keras

It is important to know how BN works so you can get a better understand of what your code is doing. But when implementing BN to your network, you don't have to implement all these details yourself. Implementing BN is often done by adding one line of code using any deep learning frameworks. In Keras, the way you add batch norm to your neural network is by adding a BN layer after the hidden layer to normalize its results before they are fed to the next layer.

The code snippet below shows you how to add a batch normalization layer when building your neural network:

```
# import BatchNormalization layer from Keras Library
from keras.layers.normalization import BatchNormalization

# initiate the model
model = Sequential()

# add the first hidden layer
model.add(Dense(hidden_nits, activation='relu'))

# add the batch norm Layer to normalize the results of Layer 1
model.add(BatchNormalization())

# if you are adding dropout to your network, it is preferred to add
# it after the batch norm layer. Because you don't want the nodes that
# are randomly turned off to miss the normalization step
model.add(Dropout(0.5))

# add the second hidden layer
```

```
model.add(Dense(units, activation='relu'))  
# add the batch norm Layer to normalize the results of layer 1  
model.add(BatchNormalization())  
  
# output layer  
model.add(Dense(2, activation='softmax'))
```

4.6.5 Batch normalization recap

The intuition that I hope you'll take away from this is that batch norm applies the normalization process not just to the input layer, but to the values in hidden layer in the neural network. This weakens the coupling of the learning process between the earlier and later layers. Allowing each layer of the network to learn by itself more independently.

The idea is, from the perspective of the later layers in the network, the earlier layers don't get to shift around as much because they are constrained to have the same mean and variance. Which makes the job of the learning in the later layers easier. The way this happens is by ensuring that the hidden units have standardized distribution (mean and variance) controlled by two explicit parameters γ and β which the learning algorithm sets during the training.

4.7 Chapter summary and takeaways

4.7.1 Final thoughts

In this chapter, we have covered a wide range of topics to structure your deep learning project and improve its performance. We talked about different performance metrics like accuracy, precision, and recall. We then discussed different techniques to preprocess your data to get it ready for training and different ways to evaluate your model and diagnose for overfitting and underfitting.

While we tried to cover the most important scenarios that you might go through when you are building, training, evaluating, and tuning your network, it is almost impossible to list all possible scenarios that you might come across in your project. What I really want you to take away from this chapter is the following:

- I. Understand the overfitting and underfitting phenomena very well. Almost all neural networks predictions problems can be boiled down in to these two.
- II. When analyzing your network's results, know that the intuitive goal is for the network to: A) learn the "right" features in your images, and B) learn these features "well enough" to generalize on new data that it hasn't seen before. Let me elaborate on these two:
 - A. **Learn the "right" features:** the best way to understand whether the network is learning the right features or not, is to look at the samples that the model is missing (i.e. bad prediction). And try to intuitively understand why it is performing poorly on them. It could be many things: maybe these images are noisy, or the objects that you are trying to classify are hidden, or many other reasons. The answer is almost always in data cleaning. When analyzing the poorly-predicted images, try to think of ways to preprocess your data so that the network is actually learning the right features of the target label not the noise.
 - B. **Learn the features "well enough":** learning "well" means not underfitting and "enough" means not overfitting and is able to generalize on the test data. Too much learning is not necessarily good. To diagnose for underfitting, look at the error performance on the training set. If it is not doing well, then you might be underfitting. And to diagnose for overfitting, look at the validation error. If it is not doing well, then you might be overfitting.

4.7.2 Tips on hyperparameters tuning

I understand that you might be a little overwhelmed by all the hyperparameters that we introduced in the last few chapters and probably concerned about the tuning process. Let me make this easier for you by providing some pointers:

- Network architecture (number of hidden layers and units): I recommend you start with an architecture that have been tried by other researchers on similar problems. Chapter

5 will help you a lot with that. We will discuss the architecture of many popular networks and teach you how to implement them on Keras. The general rule of thumb is that the deeper your network is, the better it learns

- Activation functions: at the time of writing, relu performs best in the hidden layers and softmax in the output layer
- Optimizer: You can use stochastic gradient descent (SGD). It usually succeeds to find a minimum. But if you care about fast convergence and training a complex neural network, it's safe to go with Adam. It usually outperform the other optimizers
- Learning rate - start with the default learning rate of your optimizer
- Minibatch size - start with 64 or 128. You can then tune it up and down on this scale: 32, 64, 128, 256, 512, 1024 to boost up the training speed
- Epochs - the more you train usually the better. Start with a number (say 20) and observe the training performance. If the error is still improving, increase your epochs. You might use a large number and use Early Stopping
- Regularization: use L2 regularization and dropout. They work well together to reduce the network complexity and reduce overfitting
- Batch normalization works well to speed up training
- Data preprocessing: you will need to resize your images to a uniform size

4.7.3 Deep learning foundation takeaways

The first four chapters of this book are meant to be a standalone subset of the book to teach you the end-to-end process of deep learning projects. This includes: prepping your data, build, train, evaluate and tune a neural network. After the first four chapters of this book, I'm expecting that you are empowered to do the following:

- Able to explain how neural networks work (feedforward, calculate the error, backpropagate the error)
- Understand the MLPs and CNNs topology and the differences between them
- Be able to choose the appropriate performance metric to evaluate your CNN
- Prepare your dataset to train your model
- Build a CNN architecture and run your first experiment
- Analyze the model results
- Tune the neural networks' hyperparameters to improve its performance

4.7.4 What should I do next?

Let's review what you've learned so far: In chapter 2, you learned how neural networks work. In chapter 3, you were introduced to convolutional network which are commonly to process images. In this chapter, you learned how to structure your deep learning projects and tune your hyperparameters to improve the network performance. In my opinion, that's pretty much all you need to get started with your own neural networks projects. Before moving on the the next chapter, I recommend that you go back to the projects in chapters 2 and 3 and try to apply the techniques that you learned in this chapter to improve the model's performance.

4.7.5 An advice from the author

In most sections in this chapter (and the rest of the book), I added the research papers where the concepts where originated and discussed. I recommend that you spend some time reading these papers along with my explanation. Getting into this habit will empower you with the skills needed to make you comfortable reading and implementing research papers to stay up-to-date with the latest industry advancements beyond this book.

4.8 Project: Achieve >90% accuracy on the CIFAR-10 image classification project

In this project, we are going to revisit the CIFAR classification project from chapter 3 and apply some of the improvement techniques that we learned in this chapter to improve the accuracy from ~65% to ~90%.

You can follow along with this example by visiting the book's website: www.computervisionbook.com to see the code notebook.

Step-by-step overview

We are going to implement this project by following the below steps:

- 1) Import dependencies
- 2) Get the data ready for training
 - o Download the data from the Keras library
 - o Split into train, validate, and test datasets
 - o Normalize the data
 - o One-hot encoding the labels
- 3) Build the model architecture - in addition to our regular CONV and POOL layers that we learned in chapter 3, I added the following layers to our architecture from what we learned in this chapter:
 - o Deeper neural network to increase the learning capacity
 - o dropout layers
 - o L2 regularization to our CONV layers
 - o batch normalization layers
- 4) Train the model
- 5) Evaluate the model
- 6) Plot the learning curve

Let's see how this is implemented.

4.8.1 Import dependencies

```
# Keras Library to download the datasets, preprocess images and network components
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.utils import np_utils
from keras.layers import Dense, Activation, Flatten, Dropout, BatchNormalization, Conv2D,
                      MaxPooling2D
from keras.callbacks import ModelCheckpoint
```

```
from keras import regularizers, optimizers

# import numpy for math operations
import numpy as np

# import matplotlib library to visualize results
from matplotlib import pyplot
```

4.8.2 Get the data ready for training

Keras has some datasets available for us to download and experiment with. These datasets are usually preprocessed and almost ready to be fed to the neural network. In this project, we are going to use CIFAR10 dataset. Which is a dataset of 50,000 32x32 color training images, labeled over 10 categories, and 10,000 test images. Check the keras documentation for more datasets like CIFAR100, MNIST, Fashion_MNIST, and more.

Keras provides us the CIFAR10 dataset already split into training and testing sets. We are going to load them then split the training dataset into 45,000 images for training and 5,000 images for validation as explained in this chapter.

DOWNLOAD AND SPLIT THE DATA

```
# download and split the data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# break training set into training and validation sets
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]
```

Let's print the shape of x_train, x_valid, and x_test:

```
print('x_train =', x_train.shape)
print('x_train =', x_train.shape)
print('x_train =', x_train.shape)

>> x_train = (45000, 32, 32, 3)
>> x_valid = (5000, 32, 32, 3)
>> x_test = (1000, 32, 32, 3)
```

The format of the shape tuple is as follows (number of instances, width, height, channels)

NORMALIZE THE DATA

As explained in this chapter, normalizing the pixel values of our images is done by subtracting the mean from each pixel and then dividing the result by the standard deviation.

```
mean = np.mean(x_train, axis=(0,1,2,3))
std = np.std(x_train, axis=(0,1,2,3))
x_train = (x_train - mean)/(std+1e-7)
x_valid = (x_valid - mean)/(std+1e-7)
x_test = (x_test - mean)/(std+1e-7)
```

ONE-HOT ENCODE THE LABELS

```
# one-hot encode the labels in train, valid, and test datasets
# we use "to_categorical" function in keras
num_classes = 10
y_train = np_utils.to_categorical(y_train,num_classes)
y_valid = np_utils.to_categorical(y_valid,num_classes)
y_test = np_utils.to_categorical(y_test,num_classes)
```

DATA AUGMENTATION

For augmentation techniques, we arbitrarily decided to go with the following transformations: rotation, width and height shift, and horizontal flip. When you are working on your problems, view the images that the network missed or provided poor detections on and try to understand why it is not performing well on them. Then create your hypothesis and experiment it. For example, if the missed images are of shapes that are rotated, you might want to try the rotation augmentation. Then you apply that, experiment, evaluate and repeat. This decision purely comes from analyzing your data and understanding the network performance.

```
# data augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    vertical_flip=False
)
# compute the data augmentation on the training set
datagen.fit(x_train)
```

4.8.3 Build the model architecture

In chapter 3, we built an architecture inspired from AlexNet network ($3 * \text{CONV} + 2 * \text{FC}$). In this project, we are going to build a deeper network to increase its learning capacity ($6 * \text{CONV} + 1 \text{ FC}$).

Network configuration:

- Instead of adding the POOL layer after each CONV layer, we are going to add it every 2 CONV layers. This idea was inspired by VGGNet. A popular neural network architecture that was developed by the Visual Geometry Group, hence the name VGG. VGGNet will be explained in chapter 5.
- Inspired by VGGNet, we will set the kernel_size of our CONV layers to 3x3 and the pool_size of the POOL layer to 2x2
- Dropout layers every other CONV layer with (p) ranges from 0.2 and 0.4
- Add a batch normalization layer after each CONV layer to normalize the input for the following layer
- Add L2 regularization: in keras, regularization is added to the CONV layer code as you will see in the code implementation below

Let's see the code implementation.

```

# number of hidden units variable
# we are declaring this variable here and use it in our CONV Layers to
make it easier to update from one place
base_hidden_units = 32

# L2 regularization hyperparameter ( $\lambda$ )
weight_decay = 1e-4

# create a sequential model (i.e. a Linear stack of layers)
model = Sequential()

# CONV1
# notice that we defined the input_shape here because this is the first
CONV layer. We don't need to do that for the remaining layers
model.add(Conv2D(base_hidden_units, kernel_size= 3, padding='same',
kernel_regularizer=regularizers.l2(weight_decay) , ←
input_shape x_train shape[1:])) ←
model.add(Activation('relu')) ←
model.add(BatchNormalization()) ←

# CONV2
model.add(Conv2D(base_hidden_units, kernel_size= 3, padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2)) ←
# CONV3
model.add(Conv2D(base_hidden_units * 2 , kernel_size= 3,
padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

```

add l2 regularization to
the CONV layer

relu activation function for
all hidden layers

add batch normalization layer

dropout layer with 20%
probability

number of hidden units = 64

```

# CONV4
model.add(Conv2D(base_hidden_units * 2, kernel_size= 3, padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))

model.add(Activation('relu'))
model.add(BatchNormalization())

# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.3))

# CONV5
model.add(Conv2D(base_hidden_units * 4, kernel_size= 3, padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# CONV6
model.add(Conv2D(base_hidden_units * 4, kernel_size= 3, padding='same',
kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.4))

# FC7
model.add(Flatten())
model.add(Dense(10, activation='softmax'))

# print model summary
model.summary()

```

10 hidden units because we have 10 class labels in this dataset

softmax activation function for the output layer (explained in chapter 2)

flatten the feature map into a 1D features vector (explained in chapter 3)

The model summary is printed as below:

4.8.4 Train the model

Before we jump into the training code, let's discuss the strategy of setting some hyperparameters.

- `batch_size`: this is the minibatch hyperparameter that we explained in this chapter. As I mentioned earlier, the higher the `batch_size` the faster your algorithm learns. You can start by a `minibatch= 64` and go up by doubling this value to speed up training. I tried `256` on my machine and got the error below which means that my machine is running out of memory. I then lowered it back to `128`.

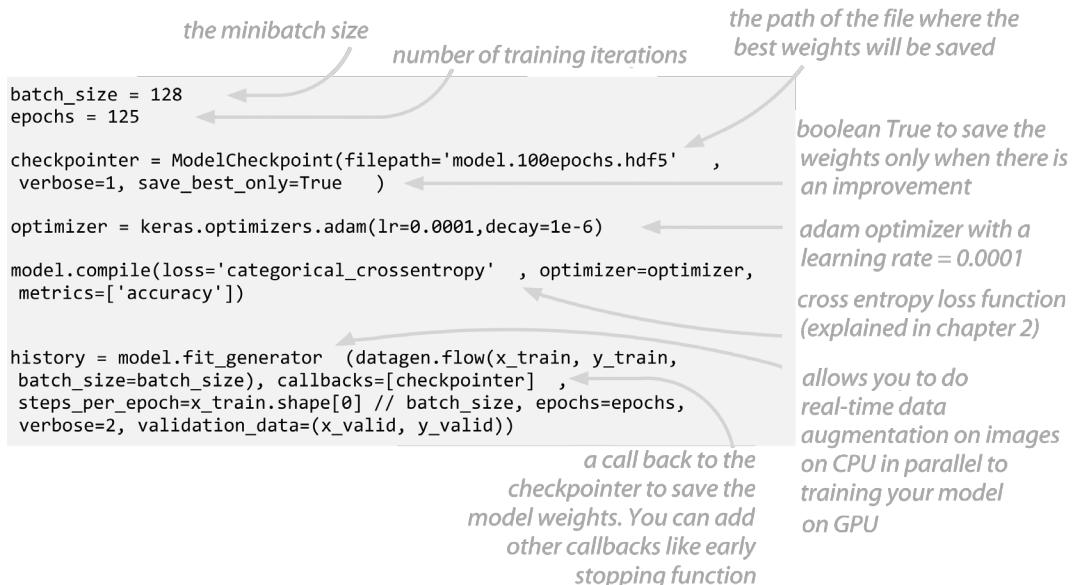
```
Resource exhausted: OOM when allocating tensor with shape[256,128,4,4]
```

- `epochs`: I started with `50` training iterations and found out that the network was still improving. So, I kept adding more epochs and observing the training results. In this project, I was able to achieve `>90%` accuracy after `125` epochs. As you will see soon, there is still room for improvements if you let it train for more time.

- Optimizer: I used Adam optimizer. See the optimization section in this chapter to learn more about optimization algorithms.

It is important to call out that I'm using a GPU for this experiment. The training took around 3 hours. It is recommended that you use your own GPU or any cloud computing service to get best results. If you don't have access to a GPU, I recommend that you try a smaller number of epochs or plan to leave your machine training overnight or even for a couple of days depending on your CPU specifications.

Let's see the training code.



When you run the above code, you will see the verbose of the network training for each epoch. Keep your eyes on the "loss" and "val_loss" values to analyze the network and diagnose bottlenecks. Below is an example verbose of epochs 121 to 125:

4.8.5 Evaluate the model

To evaluate the model, we use a keras function called "evaluate" and print the results:

```
scores = model.evaluate(x_test, y_test, batch_size=128, verbose=1)
print('\nTest result: %.3f loss: %.3f' % (scores[1]*100,scores[0]))

>> Test result: 90.260 loss: 0.398
```

4.8.6 Plot learning curves

Plot the learning curves to analyze the training performance and diagnose for overfitting and underfitting.

```
# plot Learning curves of model accuracy
pyplot.plot(history.history['acc'], label='train')
```

```
pyplot.plot(history.history['val_acc'], label='test')
pyplot.legend()
pyplot.show()
```

4.8.7 Further improvements

90% accuracy is pretty good but you can still improve further. Here are some ideas that you can experiment with:

- **More training epochs:** notice that the network was still improving until epoch #123. You can increase the number of epochs to 150 or 200 and let the network train for more time.
- **A deeper network:** try adding more layers to increase the model complexity which increases the learning capacity
- **Lower learning rate:** decrease the "lr" (you would want to train longer if you lower the "lr")
- **A different CNN architecture** like Inception or ResNet (will be explained in details in the next chapter). You can get up to 95% accuracy with the ResNet neural network after 200 epochs of training. More on that in the next chapter.
- **Transfer learning:** in the next chapter we will explain the technique of utilizing a pre-trained networks on your dataset to get higher results with fraction of the learning time

5

Advanced CNN Architectures

“Architecture begins when you place two bricks carefully together. There it begins.”

-- Ludwig Mies van der Rohe

Welcome to part two of this book, Image Classification and Object Detection. Part one was a foundation on neural networks architectures where we covered Multilayer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) or Convnets for short. We wrapped up part one with strategies to structure your deep neural network projects and tune their hyperparameters to improve your network performance. In part two, we are going to build on this foundation to develop computer vision systems that solve complex image classification and object detection problems.

In chapters 3 and 4, we talked about the main components of convnets and the different hyperparameters setup like the number of hidden layers, learning rate, optimizer, etc. In addition to other techniques to improve the network performance like regularization, augmentation, dropouts, and many more. In this chapter, you will see how all these come together to build an end-to-end convolutional network. I will walk you through five of the most popular CNNs that were state-of-the-art at their times and you will see how the authors of these networks thought about building, training, and improving their networks. We will start with LeNet that was developed in 1998 by and performed fairly well in handwritten character recognition problems. You will then see how CNN architectures have evolved from LeNet to deeper convnets like AlexNet and VGGNet, all the way to more advanced and super deep networks like Inception and ResNet developed in 2014 and 2015 respectively.

For each CNN architecture you will learn the following:

1. Novel features: in here we will explain the novel features that distinguishes these networks from the previous ones and what specific problems that the authors were trying to solve

2. Network architecture: we will cover the architecture and the components of each network and see how they all come together to form the end-to-end network
3. Network code implementation: we will walk step-by-step through the network implementation using Keras deep learning library. The goal of this section is for you to learn how to read research papers and implement new architectures as they come up
4. Set up the learning hyperparameters: after you implement the network architecture, you need to set up the hyperparameters of the learning algorithms that you learned in chapter 4 like the optimizer, learning rate, weight decay, etc. In this chapter, we will implement the learning hyperparameters as presented in the original research paper of each network. In this section, you will see how the performance evolved from one network to another over the years
5. Network performance: finally, you will see how each network performed on benchmark datasets like MNIST and ImageNet as represented in their research papers

Three main takeaways from reading this chapter:

1. Understand the architecture and learning hyperparameters of state-of-the-art CNNs. You will be implementing simpler CNNs like AlexNet and VGGNet for simple to medium-complexity problems. For very complex problems you might want to use deeper networks like Inception and ResNet.
2. Understand the novel features of each network and the motives behind them. Each CNN architecture solves a specific limitation in the previous one. After reading the five networks in this chapter (and their research paper), you will build a strong foundation to read and understand new state-of-the-art networks as they come up.
3. Learning how convnets have evolved and the authors thought process helps you build an intuition of what works well and what are the problems that may arise when building your own network.

In chapter three, we learned about the basic building blocks of convolutional layers, pooling layers and fully-connected layers of convnets. As you will see in the different CNN architectures in this chapter, a lot of the past few years of computer vision research has been on how to put together these basic building blocks to form effective convolutional neural networks. And one of the best ways for you to get intuition yourself is to see some of these state-of-the-art architectures yourself and learn from them. Similar to how most of us may have learned to write code by reading other people's code. This exercise will help you build a good intuition on how to build your own convnets.

To get the most out of this chapter, I encourage you to read the research papers that are linked in each section before you read my explanation. What you have learned in part one of this book fully equips you to start reading research papers written by pioneers in the AI field. Reading and implementing research papers is by far one of the most valuable skill that you will build from reading this book.

Why read research papers?

Personally, I find the task of going through a research paper, interpreting the crux behind it, and implementing the code as a very important skill every deep learning enthusiast and practitioner should possess. Practically implementing research ideas brings out the thought process of the author and also helps transform those ideas into real-world

industry applications. By reading this chapter, I hope you get comfortable reading research papers and implement their findings to your own problem. The fast paced evolution in this field requires us to always stay up-to-date with the latest research. What you will learn in this book (or other publications) now will not be the latest and greatest in the three or four years. Maybe even sooner. The most valuable skill that I want you to take away from this book is to build a strong deep learning foundation that empowers you to get out to the real world and be able to read the latest research and implement them yourself.

Are you ready? Let's get started!

5.1 CNN design patterns

Before we jump into the details of the common CNN architectures, we are going to look at some common design choices when it comes to convolutional neural networks. It might seem at first that there are way too many choices to make. Everytime we learn about something new in deep learning, it gives us more hyperparameters to design. So it is good to be able to narrow down our choices by looking at some common patterns that were created by pioneer researchers in the field, understand their motivation and start from where they ended rather than doing things completely randomly.

5.1.1 Pattern #1

Convolutional nets are typically composed of two parts; 1) the feature extraction part which consists of a series of convolutional layers, and 2) the classification part which consists of a series of fully-connected layers. This is pretty much always the case with convnets, starting from LeNet and AlexNet to the very recent CNNs that have came out in the past few years like Inception and ResNet.

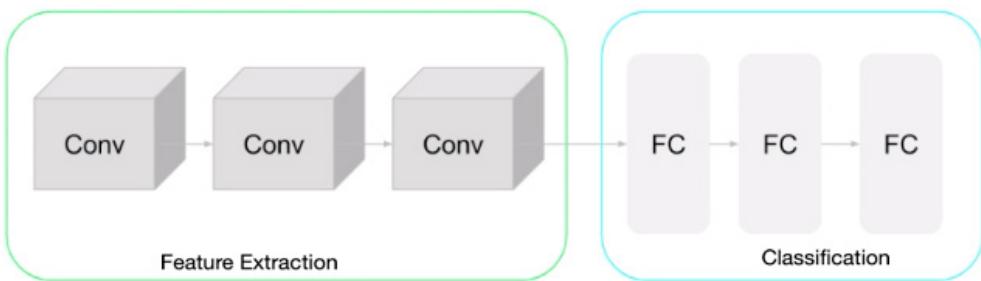


Figure 5.1

5.1.2 Pattern #2

The input data at each layer is an image. With each layer we are applying a new convolution over a new image. This pushes us to think of an image in a more generic way. First, you see that each image is a 3D object that has a height, width, and depth. Depth is referred to as the color channel where depth = 1 for grayscale images and 3 for color images. In the later layers, the images still have depth but they are not colors per se. They are feature maps that

represent the features extracted from the previous layers. That's why you see the depth increasing as we go deeper through the network layers. So, you can see the depth of an image is equal 96. This represents the number of feature maps in this layer. So that's one pattern that you will always see: the image depth increases and the dimensions decrease.

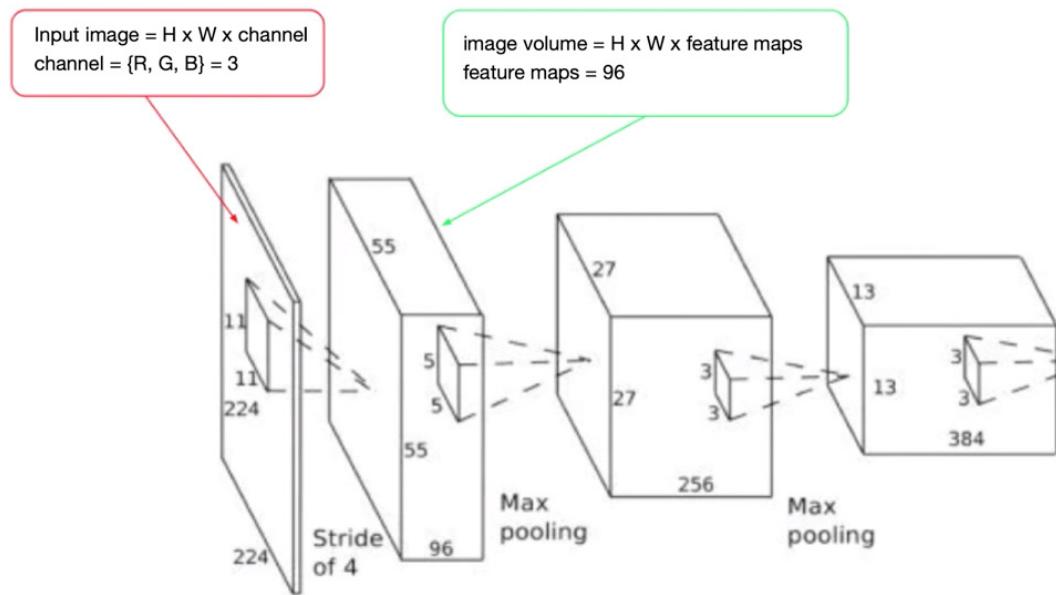


Figure 5.2

5.1.3 Pattern #3

The final pattern that we will discuss is the fully-connected layers. This generally doesn't follow as strict of a pattern as the previous patterns but very helpful to know. Typically, either all fully-connected layers in a network have the same number of hidden units or decrease at each layer. It is very rare to find a network where the number of units in FC layer increase at each layer. Research has found that keeping the number of units constant doesn't hurt the neural network so it may be a good approach if you want to limit the number of choices you have to make when designing your network. This way, all you have to do is to pick a number of units per layer and apply that to all your FC layers.

Now that you understand the basic CNN patterns, let's look at some architectures that have implemented these patterns. Most of these architectures are famous because they performed well on the ImageNet competition which is a famous benchmark that contains millions of images. Deep learning and computer vision researchers use ImageNet dataset to compare the algorithms. More on that later. Now, let's get started with the first network that we are going to discuss in this chapter, LeNet.

5.2 LeNet-5

In 1998, LeCun et al. introduced in their paper “[Gradient-Based Learning Applied to Document Recognition](#)” a pioneering convolutional neural network called **LeNet-5**. The LeNet-5 architecture is straightforward and you have seen all of its components in the previous chapters of this book. It is composed of 5 weight layers, hence the name LeNet-5: 3 convolutional layers + 2 fully connected layers.

What are weight layers?

We refer to the convolutional and fully-connected layers as weight layers because they contain trainable weights as opposed to pooling layers that don't contain any weights. The common convention is to use the number of weight layers to describe the depth of the network. For example, AlexNet (explained next) is said to be 8 layers deep because it contains 5 CONV + 3 FC layers. The reason we care more about weight layers is mainly because they reflect the model computational complexity.

5.2.1 LeNet architecture

The architecture of LeNet-5 looks like this:

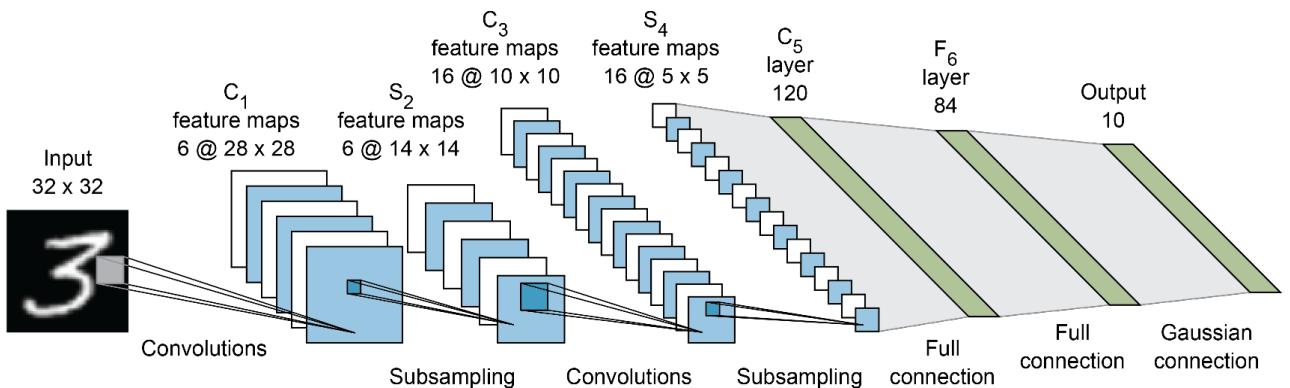


Figure 5.3

LeNet architecture in text:

INPUT IMAGE => C1 => TANH => S2 => C3 => TANH => S4 => C5 => TANH => FC6 => SOFTMAX7

Where C is the CONV layer, S is the subsampling or POOL layer, and FC is the fully connected layer.

The building components of the LeNet architecture is not new to you (it was new back in 1998). You have already learned the CONV, POOL, and FC layers in chapter 3. Notice that Yann LeCun and his team used **tanh** as an activation function instead of the nowadays state-of-the-art ReLU. This is because back in 1998, ReLU had not been used in the context of deep

learning yet and it was more common to use tanh or sigmoid as an activation function in the hidden layers.

Without further ado, let's implement LeNet-5 in Keras.

5.2.2 LeNet-5 implementation in Keras

To implement LeNet-5 in Keras, read the [original paper](#) and follow the architecture information from pages 6, 7 and 8. Here are the main takeaways to build the LeNet-5 network:

- Number of filters in each CONV layer: you can see from the diagram (and defined in the paper) that the depth (number of filters) of each convolutional layer is as follows: C1 = 6, C3 = 16, C5 = 120 layers.
- Kernel size of each CONV layer: from the paper, the kernel_size is = 5×5
- A subsampling layer (POOL) is added after each convolutional layer. The receptive field of each unit is a 2×2 area (i.e. pool_size = 2). Note that the LeNet-5 creators used *average pooling* which computes the average value of its inputs instead of the *max pooling* layer that we used in our earlier projects which passes the maximum value of its inputs. You can try both if you are interested to see the difference. For this experiment, we are going to follow the paper architecture.
- Activation function: as we mentioned before, the creators of LeNet-5 used **tanh** activation function for the hidden layers because symmetric functions are believed to yield faster convergence compared to sigmoid functions.

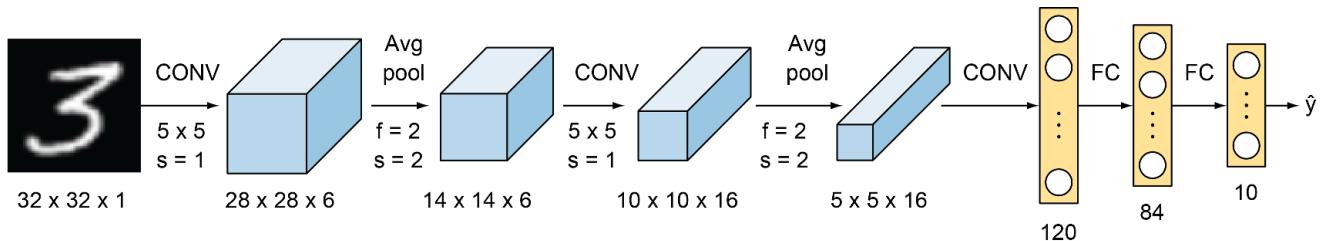


Figure 5.4

Now let's put that in code to build the LeNet-5 architecture:

```
# Instantiate an empty sequential model
model = Sequential()
# C1 Convolutional Layer
model.add(Conv2D(filters = 6, kernel_size = 5, strides = 1, activation = 'tanh',
input_shape = (32,32,1), padding = 'same'))

# S2 Pooling Layer
model.add(AveragePooling2D(pool_size = 2, strides = 2, padding = 'valid'))

# C3 Convolutional Layer
model.add(Conv2D(filters = 16, kernel_size = 5, strides = 1,activation = 'tanh',
padding = 'valid'))
# S4 Pooling Layer
model.add(AveragePooling2D(pool_size = 2, strides = 2, padding = 'valid'))
```

```

# C5 Convolutional Layer
model.add(Conv2D(filters = 120, kernel_size = 5, strides = 1, activation = 'tanh',
                 padding = 'valid'))

# Flatten the CNN output to feed it with fully connected layers
model.add(Flatten())

# FC6 Fully Connected Layer
model.add(Dense(units = 84, activation = 'tanh'))

# FC7 Output layer with softmax activation
model.add(Dense(units = 10, activation = 'softmax'))

# print the model summary
model.summary()

```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_1 (Conv2D)	(None, 28, 28, 6)	156
average_pooling2d_1 (Average)	(None, 14, 14, 6)	0
conv2d_2 (Conv2D)	(None, 10, 10, 16)	2416
average_pooling2d_2 (Average)	(None, 5, 5, 16)	0
conv2d_3 (Conv2D)	(None, 1, 1, 120)	48120
flatten_1 (Flatten)	(None, 120)	0
dense_1 (Dense)	(None, 84)	10164
dense_2 (Dense)	(None, 10)	850
<hr/>		
Total params:	61,706	
Trainable params:	61,706	
Non-trainable params:	0	

Figure 5.5

LeNet-5 is a small neural network with today's standards. It has 61,706 parameters compared to millions of parameters in more modern networks as you will see later in this chapter in more modern architectures.

5.2.3 Set up the learning hyperparameters

The authors used a scheduled decay learning where the value of the learning rate was decreasing using the following schedule: 0.0005 for the first two epochs, 0.0002 for the next three epochs, 0.00005 for the next four, then 0.00001 thereafter. In their paper, the authors trained their network for 20 epochs.

Let's build a lr_schedule function with the above schedule. The method will take an integer epoch number as an argument and returns the learning rate (lr).

```
def lr_schedule(epoch):
    # initiate the learning rate with value = 0.0005
    lr = 5e-4

    # lr = 0.0005 for the first two epochs, 0.0002 for the next three epochs,
    # 0.00005 for the next four, then 0.00001 thereafter.
    if epoch > 2:
        lr = 2e-4
    elif epoch > 5:
        lr = 5e-5
    elif epoch > 9:
        lr = 1e-5
    return lr
```

We will then use the lr_schedule function in the code snippet below to compile the model:

```
model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=lr_schedule(0)),
              metrics=['accuracy'])
```

Now start the network training for 20 epochs as mentioned in the paper:

```
hist = model.fit(X_train, y_train, batch_size=32, epochs=20,
                  validation_data=(X_test, y_test), callbacks=[checkpointer],
                  verbose=2, shuffle=True)
```

5.2.4 LeNet performance on MNIST dataset

When you train LeNet-5 on the MNIST dataset you will get above 99% accuracy (see the code notebook attached to this chapter at www.computervisionbook.com). Try to re-run this experiment with 'relu' activation function in the hidden layers and observe the difference in the network performance.

When you read the LeNet-5 paper, just know that this paper is one of the hardest ones to read than the ones we will cover in this chapter. Most of the ideas that I mentioned in the previous pages are in sections two and three of the paper. The later sections of the paper talk about something called the graph transformer network, which isn't widely used today. So if you do try to read this paper, I recommend focusing on section two which talks about the LeNet architecture and the learning details, and maybe take a quick look at section three which has a bunch of experiments and results, that is pretty interesting.

I recommend starting with the [AlexNet](#) paper followed by the [VGGNet](#) paper explained in the following sections and then the LeNet paper because it is a bit harder to read but it is a good classic one once you go over the other ones.

5.3 AlexNet

We saw how LeNet performed very well on the MNIST dataset. But it turns out that the MNIST dataset is very simple because it contains gray scale images (1 channel) and classified into only 10 classes which makes it a simpler challenge. The main motivation behind AlexNet was to build a deeper network that can learn more complex functions.

AlexNet was the winner of the ILSVRC image classification competition in 2012. Alex Krizhevsky, Geoffrey Hinton and Ilya Sutskever created a neural network architecture called 'AlexNet' in their paper "[ImageNet Classification with Deep Convolutional Neural Networks](#)". They trained their network on 1.2 million high-resolution images into 1,000 different classes of the ImageNet dataset. AlexNet was state-of-the-art at its time because it was the first real "deep" network (back then) that opened the door for the computer vision community to seriously consider convolutional networks in their applications. We will explain deeper networks later in this chapter like VGGNet and ResNet, but it is good to see how convnets evolved and the main drawbacks of AlexNet that were the main motivation for the later networks.

The AlexNet architecture is shown in the figure below:

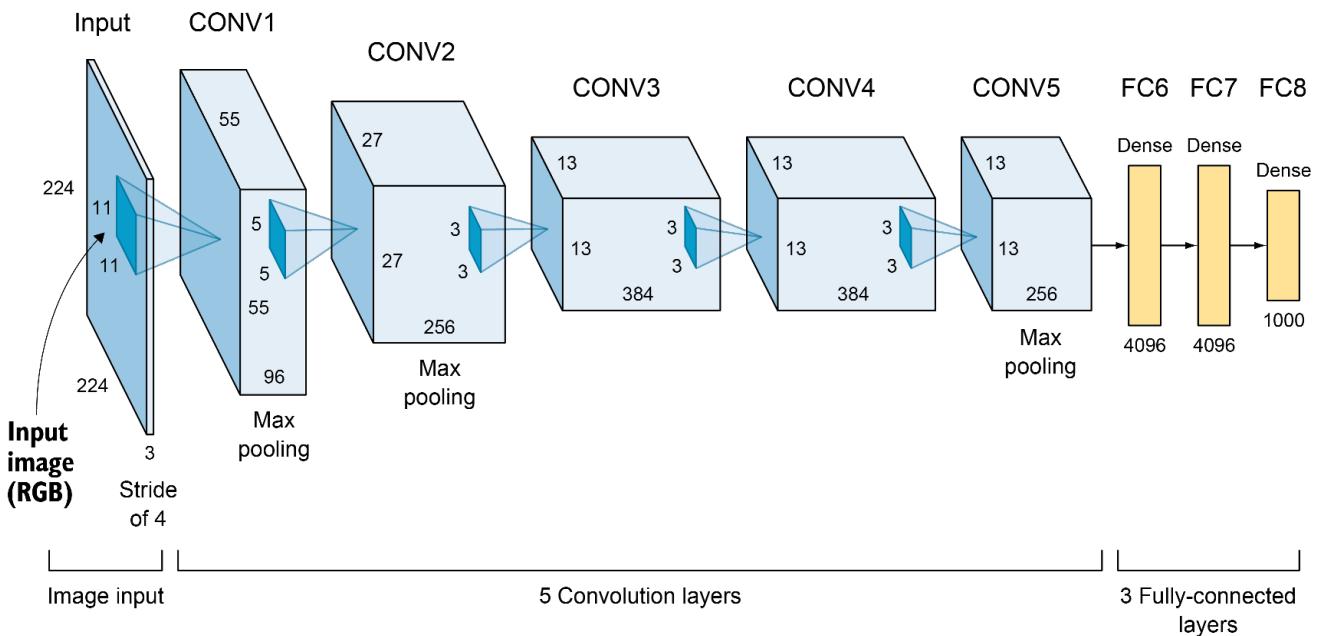


Figure 5.6

As you see in the diagram above, AlexNet has a lot of similarities to LeNet but it is much deeper (more hidden layers) and bigger (more filters per layer). They both have similar building blocks of a series of CONV + POOL layers stacked on top of each other followed by FC layers and a Softmax. We've seen that LeNet has around 61 thousand parameters whereas AlexNet has about 60 million parameters and 650,000 neurons which gives it a larger learning

capacity to understand more complex features. This allowed AlexNet to have a remarkable performance in the ILSVRC image classification competition in 2012.

What is ImageNet and ILSVRC?

[ImageNet](#) is a large visual database designed for use in visual object recognition software research. It is aimed at labeling and categorizing images into almost 22,000 categories based on a defined set of words and phrases. The images were collected from the web and labeled by human labelers using Amazon's Mechanical Turk crowdsourcing tool. At the time of this writing, there are over 14 million images in the ImageNet project. To organize such a massive amount of data, the creators of ImageNet followed the WordNet hierarchy where each meaningful word/phrase inside WordNet is called a "synonym set" or "synset" for short. Within the ImageNet project, images are organized according to these synsets, with the goal being to have 1,000+ images per synset.

[ILSVRC](#): The ImageNet project runs an annual software contest called the "ImageNet Large Scale Visual Recognition Challenge (ILSVRC)", where software programs compete to correctly classify and detect objects and scenes. We will be using the ILSVRC challenge as a benchmark to compare the different networks' performance.

5.3.1 AlexNet architecture

You've seen a version of the AlexNet architecture in the project at the end of chapter 3. The architecture is pretty straightforward. It consists of:

- Convolutional layers with the following kernel sizes: 11x11, 5x5, and 3x3
- Max pooling layers for images downsampling
- Dropout layers to avoid overfitting
- Unlike LeNet, AlexNet authors used ReLU activation functions in the hidden layers and a softmax activation in the output layer

AlexNet consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. You can represent the AlexNet architecture in text as follows:

INPUT IMAGE => CONV1 => POOL2 => CONV3 => POOL4 => CONV5 => CONV6 => CONV7 => POOL8 => FC9 => FC10 => SOFTMAX7

5.3.2 Novel features of AlexNet

Before AlexNet, deep learning was starting to gain traction in speech recognition and a few other areas. But AlexNet was the milestone that convinced a lot of the computer vision community to take a serious look at deep learning and demonstrate that deep learning really works in computer vision. Compared to previous CNNs (like LeNet), AlexNet presented some novel features that were not used in previous architectures. You are already familiar of all of them from the previous chapters in this book so it should be quick for us to go through them here.

3.2.1. ReLU activation function:

AlexNet, proposed by Alex Krizhevsky, uses ReLU(Rectified Linear Unit) for the non-linear part, instead of a Tanh or Sigmoid functions that were the earlier standard for traditional neural

networks (like LeNet), ReLU was used in the hidden layers of AlexNet architecture because it trains much faster. This is because the derivative of the sigmoid function becomes very small in the saturating region and therefore the updates applied to the weights almost vanish. This phenomenon is called the **vanishing gradient problem**. ReLU is represented by this equation $f(x) = \max(0, x)$ and is discussed in details in chapter 2.

Certain activation functions, like the sigmoid function, squishes a large input space into a small input space between 0 and 1 (-1 to 1 range for Tanh activations). Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Which makes the derivative become very small.

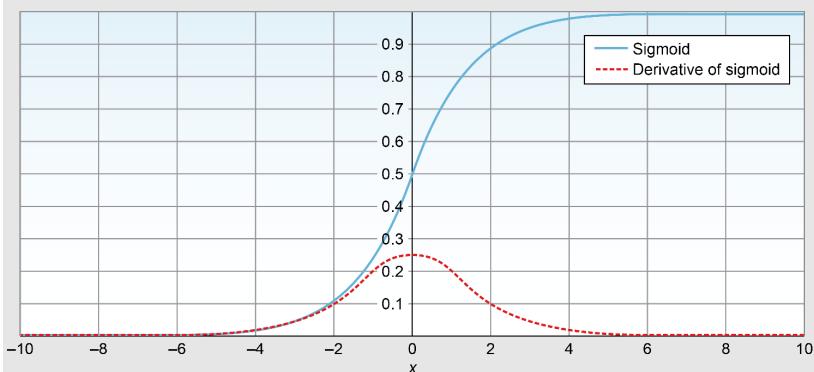


Figure 5.7

We will talk more about the vanishing gradient phenomenon later in this chapter when we talk about the ResNet architecture.

5.3.2.2 Dropout layer:

as explained in chapter 3, dropout layers are used to avoid the neural network overfitting. The neurons which are “dropped out” do not contribute to the forward pass and do not participate in backpropagation. This means that every time an input is presented, the neural network samples a different architecture, but all these architectures share the same weights. This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. The authors used dropout with a probability = 0.5 in the two fully-connected layers.

5.3.2.3. Data augmentation:

one popular and very effective approach to avoid overfitting is to artificially enlarge the dataset using label-preserving transformations. This happens by generating new instances of the training images with some transformations like image rotation, flipping, scaling, and many more. Data augmentation is explained in details in chapter 4.

5.3.2.4. LOCAL RESPONSE NORMALIZATION:

in AlexNet, local response normalization is used. It is different from the batch normalization technique (explained in chapter 4). Normalization helps to speed up the convergence. Nowadays, batch normalization (BN) is used instead of using local response normalization and we will be using BN in our implementation in this chapter.

5.3.2.5. Weight regularization:

the authors used a weight decay of 0.0005. Weight decay is another term for the L2 regularization technique explained in chapter 4. It is an approach to reduce the overfitting of a deep learning neural network models on the training data to allow it to generalize better on new data.

```
model.add(Conv2D(32, (3,3), kernel_regularizer=l2( $\lambda$ )))
```

The lambda value is the weight decay hyperparameter that you can tune. If you still see overfitting size, increase the lambda value to reduce overfitting. In this case, the authors found that a small decay value of 0.0005 was good enough for the model to learn.

5.3.2.6. Training on multiple GPUs:

the authors used a GTX 580 GPU that has only 3GB of memory. It was state-of-the-art at the time but not large enough to train the 1.2 million training examples in their dataset. Therefore they developed a complicated way to spread their network across two GPUs. The basic idea was that, a lot of these layers were split across two different GPUs and there was a thoughtful way for when the two GPUs would communicate with each other. You don't need to worry about these details nowadays because there are far more advanced ways to train your deep networks on distributed GPUs that we will discuss later in this book.

5.3.3 AlexNet implementation in Keras

Okay, now that you've learned the basic components of AlexNet and the novel features, let's apply all these together to build the AlexNet neural network. I suggest that you read the architecture description in page 4 in the [original paper](#) and follow along with the next section.

As depicted in the figure below, the network contains eight weight layers: the first five are convolutional and the remaining three are fully-connected. The output of the last fully-connected layer is fed to a 1000-way softmax which produces a distribution over the 1000 class labels.

AlexNet input starts with 227x227x3 images. If you read the paper, you will notice that it refers to the dimensions volume of 224x224x3 for the input images. But the numbers make sense only for 227x227x3 images. I suggest that this could be a typing mistake in the paper.

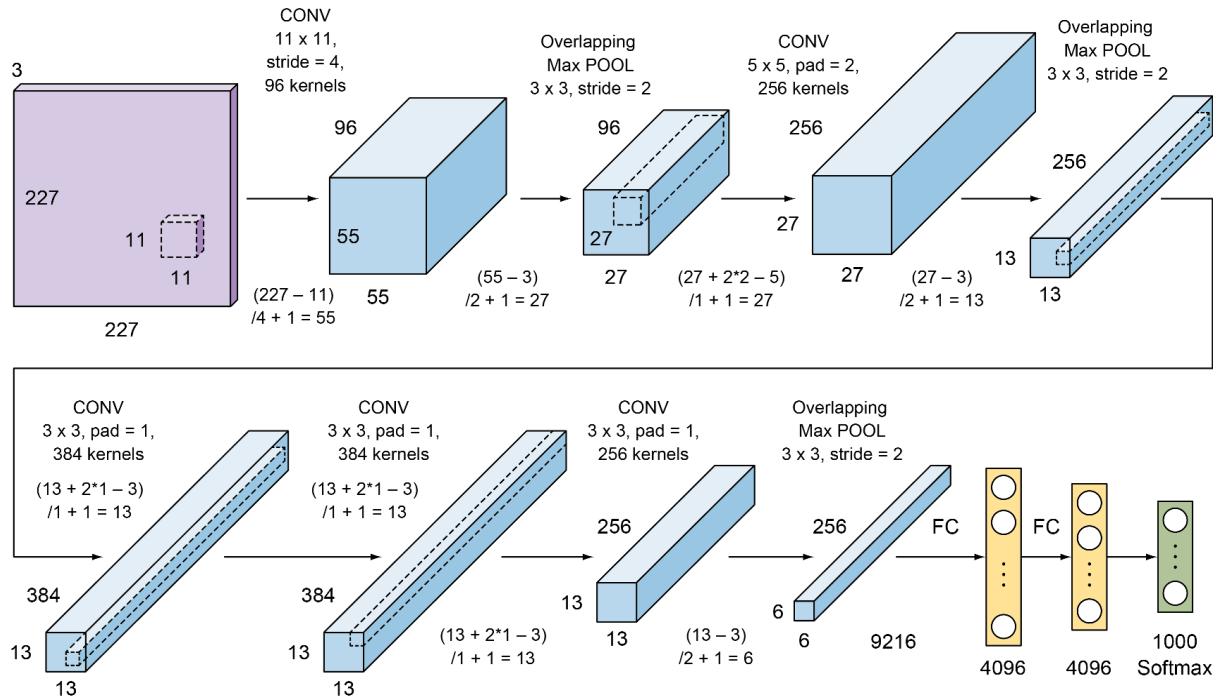


Figure 5.8

The layers are stacked together as follows:

- CONV1 - the authors used a large kernel size = 11. They also used a large stride = 4 which makes the input dimensions roughly shrink by a factor 4 (from 227x227 to 55x55).

Calculate the dimensions of the output as follows:

$$(227 - 11)/4 + 1 = 55 \text{ and the depth is the number of filters in the conv layer} = 96$$

The output dimension = 55x55x96

- POOL layer with a filter size of 3x3 which reduces the dimensions from 55x55 to 27x27. $(55 - 3)/2 + 1 = 27$. The pooling layer doesn't change the depth of the volume.

The output dimension = 27x27x96

Similarly, you can calculate the output dimensions of the remaining layers.

- CONV2 - kernel size = 5, depth = 256, and stride = 1
- POOL layer with size = 3x3 which downsamples its input dimensions from 27x27 to 13x13.
- CONV3 - kernel size = 3, depth = 384, and stride = 1
- CONV4 - kernel size = 3, depth = 384, and stride = 1

- CONV5 - kernel size = 3, depth = 256, and stride = 1
- POOL layer with size = 3x3 which downsamples its input from 13x13 to 6x6
- Flatten layer to flatten the dimension volume 6x6x256 to 1x9216
- FC layer with 4096 neurons
- FC layer with 4096 neurons
- Softmax layer with 1000 neurons

You might be wondering, how did the authors decide to implement this configuration? Setting up the right values of the network hyperparameters like kernel size, depths, stride, pooling size, etc. is a tedious work of a lot of trial and error. The idea remains the same, we want to apply many weight layers to increase the model's learning capacity to learn more complex functions. We also need to add pooling layers in between to downsample the input dimensions as discussed in chapter 2. With that said, setting up the exact hyperparameters comes with a lot of trial and error and this is one of the challenges of CNNs. VGGNet (explained next) solves this problem by implementing uniform layers configuration to reduce the amount of trial and error work when designing your network.

Note that all CONV layers are followed by a batch normalization layer and all hidden layers are followed by ReLU activations. Now, let's put that in code to build the AlexNet architecture:

```
# Instantiate an empty sequential model
model = Sequential()
# 1st layer (conv + pool + batchnorm)
model.add(Conv2D(filters= 96, kernel_size= (11,11), strides=(4,4), padding='valid',
input_shape = (224,224,3)))
model.add(Activation('relu')) <---- activation function can be added on its own layer or
                           within the Conv2D function as we did in previous implementations
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2)))
model.add(BatchNormalization())

# 2nd layer (conv + pool + batchnorm)
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same',
                 kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(MaxPool2D(pool_size=(3,3), strides=(1,1)))
model.add(BatchNormalization())

# layer 3 (conv + batchnorm)      <--- note that the authors did not add a POOL layer here
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same',
                 kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# layer 4 (conv + batchnorm)      <--- similar to layer 4
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same',
                 kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# layer 5 (conv + batchnorm)
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same',
```

```

        kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2)))

# Flatten the CNN output to feed it with fully connected layers
model.add(Flatten())

# layer 6 (Dense layer + dropout)
model.add(Dense(units = 4096, activation = 'relu'))
model.add(Dropout(0.5))

# layer 7 (Dense layers)
model.add(Dense(units = 4096, activation = 'relu'))
model.add(Dropout(0.5))

# layer 8 (softmax output layer)
model.add(Dense(units = 1000, activation = 'softmax'))

# print the model summary
model.summary()

```

Model summary

When you print the model summary you will see the number of total parameters = 62 million as follows:

```

Total params: 62,383,848
Trainable params: 62,381,096
Non-trainable params: 2,752

```

Figure 5.9

A note on LeNet and AlexNet architectures

Both LeNet and AlexNet have so many hyperparameters to tune. The authors had to go through so many experiments to set the kernel_size, strides, and padding for each layer which makes it harder to understand and manage. VGGNet (explained next) solves this problem with a very simple and uniform architecture.

5.3.4 Set up the learning hyperparameters

AlexNet was trained for 90 epochs which took 6 days simultaneously on two Nvidia Geforce GTX 580 GPUs. This is the reason for why you will see that their network is split into two pipelines in the original paper. The authors started with an initial learning rate 0.01 with a momentum of 0.9. The "lr" is then divided by 10 when the validation error stops improving.

```

# reduce learning rate by 0.1 when the validation error plateaus
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=np.sqrt(0.1))

# set the SGD optimizer with lr of 0.01 and momentum of 0.9
optimizer = keras.optimizers.sgd(lr = 0.01, momentum = 0.9)

```

```
# compile the model
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])

# train the model
# call the reduce_lr value using callbacks in the training method
model.fit(X_train, y_train, batch_size=128, epochs=90, validation_data=(X_test, y_test),
           verbose=2, callbacks=[reduce_lr])
```

5.3.5 AlexNet performance on CIFAR dataset

AlexNet significantly outperformed all the prior competitors in the ILSVRC challenges and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry of that year that uses other traditional classifiers. This huge improvement in performance attracted the computer vision community's attention to the potential that convolutional networks have to solve complex vision problems and lead to more advanced CNN architectures as you will see in the following sections of this chapter.

What are Top-1 and Top-5 error rates?

Top-1 and Top-5 are terms used mostly in research papers to describe the accuracy of an algorithm on a given classification task. Where the Top-1 error rate is the percentage of the time that the classifier did not give the correct class the highest score and the Top-5 error is the percentage of the time that the classifier did not include the correct class among its top 5 guesses.

Let's apply this in an example. Suppose that there are 100 classes, and we show the network an image of a cat. The classifier outputs a score or confidence value for each class as follows:

1. Cat - 70%
2. Dog - 20%
3. Horse - 5%
4. Motorcycle - 4%
5. Car - 0.6%
6. Plane - 0.4%

This means that the classifier was able to correctly predict the true class of the image in the Top-1. Try the same experiment for 100 images and observe how many times the classifier missed the true label and that's your Top-1 error rate.

Same idea for the Top-5 error rate. In the example above, if the true label is the Horse. Then, the classifier missed the true label in the Top-1 but it caught it in the first 5 predicted classes (i.e. Top-5). Calculate how many times the classifier missed the true label in the top 5 predictions and that's your Top-5.

Ideally, we want the model to always predict the correct class in the Top-1. But Top-5 gives a more holistic evaluation of the model performance by defining how close the model is to the correct prediction for the missed ones.

5.4 VGGNet

VGGNet was developed by the Visual Geometry Group at Oxford University, hence the name VGG, in 2014 by Karen Simonyan and Andrew Zisserman. It was introduced in their paper "[Very Deep Convolutional Networks for Large-Scale Image Recognition](#)".

The building components of VGGNet are exactly the same as LeNet and AlexNet except that it is an even deeper network with more convolutional, pooling, and dense layers. Other than that, there are no new components that are introduced here.

VGGNet, also known as VGG-16, consists of 16 weight layers; 13 convolutional layers + 3 fully-connected layers. Its uniform architecture makes it very appealing in the deep learning community because it is very easy to understand.

5.4.1 Novel features of VGGNet

We've seen how challenging it can be to set up the CNN hyperparameters like kernel size, padding, strides, etc. VGGNet novel concept is that it designed a simple architecture that contains uniform components (CONV and POOL layers). It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layers, respectively) with multiple 3x3 pool-size filters one after another.

The architecture is composed of a series of uniform CONV building blocks followed by a unified POOL layer where:

- all convolutional layers are 3x3 kernel-sized filters with a stride = 1 and padding = same
- all pooling layers are 2x2 pool-size and a stride = 2

Why use smaller 3x3 convolutions?

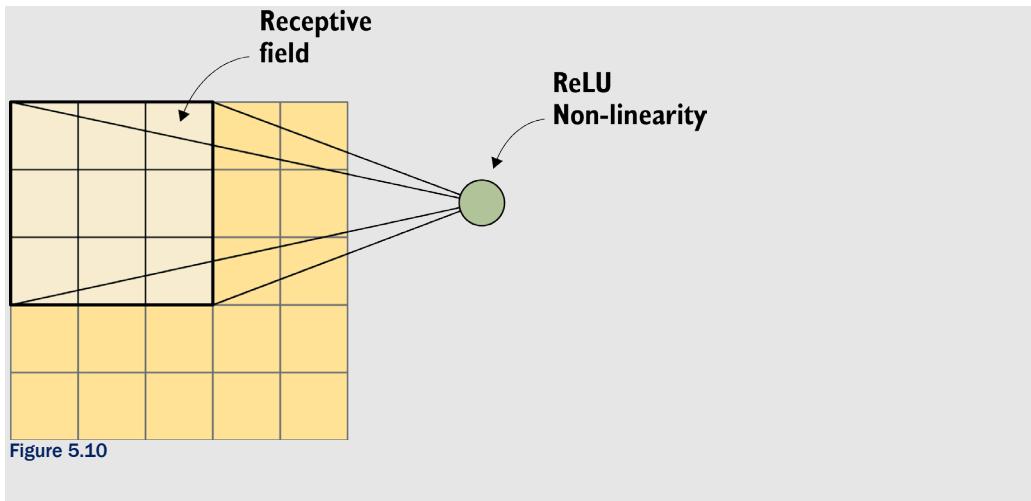
The authors of VGGNet decided to use a smaller 3X3 kernels to allow the network to extract finer level features of the image compared to AlexNet's large kernels of 11x11 and 5x5 kernels. The idea is that with a given convolutional receptive field, multiple stacked smaller size kernel is better than the one with a larger size kernel because multiple non-linear layers increases the depth of the network which enables it to learn more complex features at a lower cost because it has a lower number of learning parameters.

For example: In their experimentations, the authors noticed that a stack of two 3×3 conv. layers (without spatial pooling in between) has an effective receptive field of 5×5 and three 3×3 conv. layers have the effect of a 7×7 receptive field. So by using 3×3 convolutions with higher depth you get the benefits using more nonlinear rectification layers (relu) which makes the decision function more discriminative. Secondly, this decreases the number of training parameters because when you use three-layer 3×3 conv. with C channels, the stack is

parameterised by $3(3^2C^2) = 27C^2$ weights compared to a single 7×7 conv. layer which

requires $7^2C^2 = 49C^2$ weights which is 81% more parameters.

Receptive field: explained in chapter 3, it is the effective area of input image on which output depends on.



This unified configuration of the CONV and POOL components, simplifies the neural network architecture which makes it very easy to understand and implement.

VGGNet architecture is developed by stacking 3×3 convolutional layers with 2×2 pooling layers inserted after several CONV layers. This is followed by the traditional classifier that is composed of some fully-connected layers and a softmax as depicted in the figure below:

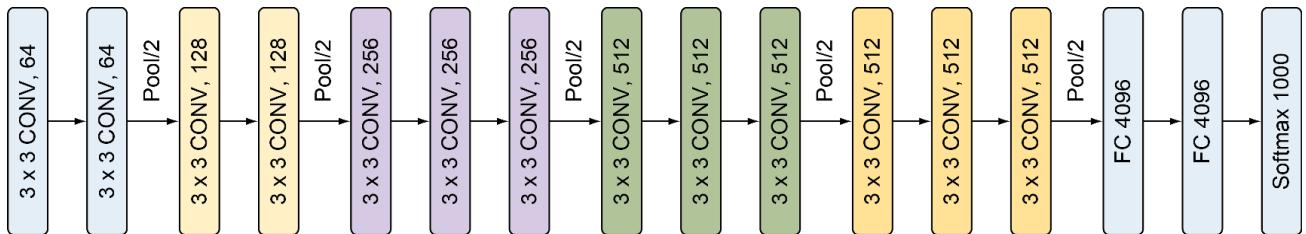


Figure 5.11

5.4.2 4.2. VGGNet Configurations

The authors created several configurations for the VGGNet architecture as you see in the table below. All configurations follow the same generic design. Configurations D and E are the most commonly used and referred to as VGG-16 and VGG-19 referring to the number of weight layers. Each block contains a series of 3×3 convolutional layers with similar hyperparameters configuration and followed by a 2×2 pooling layer.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256	conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512	conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512	conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					

Figure 5.12

In the table below you will see the number of learning parameters in millions for each configuration. VGG-16 yields ~ 138 million parameters and VGG-19 is a deeper version of VGGNet that has more than 144 million parameters. VGG-16 is more commonly used because it performs almost as well as VGG-19 but with fewer parameters.

Table 2: Number of parameters (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Figure 5.13

5.4.3 VGG-16 in Keras

Configurations D (VGG-16) and E (VGG-19) are the most commonly used configurations because they are deeper networks that can learn more complex functions. So, in this chapter we will implement configuration D of the VGGNet that has 16 weight layers. VGG-19

(Configuration E) can be similarly implemented by just adding a fourth CONV layer to the third, fourth, and fifth blocks as you can see in the above table. You can see the notebooks attached to this chapter for a full implementation of both VGG-16 and VGG-19 at www.computervisionbook.com.

Note that the authors used the following regularization techniques to avoid overfitting:

- L2 regularization with weight decay of 5×1^{-4} . This is not added to the implementation below for simplicity
- Dropout regularization for the first two fully-connected layers with a dropout ratio set to 0.5

```
# Instantiate an empty sequential model
model = Sequential()

# block #1
model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same', input_shape=(224,224, 3)))
model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #2
model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #3
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #4
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #5
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu',
                 padding='same'))
model.add(MaxPool2D((2,2), strides=(2,2)))

# block #6 (classifier)
model.add(Flatten())
```

```

model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1000, activation='softmax'))

# print the model summary
model.summary()

```

Model summary

When you print the model summary you will see the number of total parameters ~ 138 million as follows:

```

Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0

```

Figure 5.14

5.4.4 Learning hyperparameters

The authors followed a similar training procedure to AlexNet. Namely, the training is carried out using mini-batch gradient descent SGD with momentum = 0.9. The learning rate was initially set to 0.01 , and then decreased by a factor of 10 when the validation set accuracy stopped improving.

5.4.5 VGGNet performance on CIFAR dataset

VGG-16 achieved the top-5 error rate of 8.1% on ImageNet compared to 15.3% achieved by AlexNet. Even better accuracy with the VGG-19 where it was able to achieve a top-5 error rate of $\sim 7.4\%$. It is worth noting that, in spite of the larger number of parameters and the greater depth of VGGNet compared to AlexNet, VGGNet required less epochs to converge due to implicit regularisation imposed by greater depth and smaller convolutional filter sizes

5.5 Inception and GoogLeNet

The Inception network came to the world in 2014 when a group of researchers at Google published their paper "[Going Deeper with Convolutions](#)". The main hallmark of this architecture is building a deeper neural network while improving the utilization of the computing resources inside the network. One particular incarnation of the Inception network is called GoogleNet and was used in their submission for ILSVRC14. It uses a 22 layers deep network which is deeper than VGGNet while reducing the number of parameters 12 times fewer (from ~ 138 million to ~ 13 million) while achieving significantly more accurate results. The network used a CNN inspired by the classical networks (AlexNet and VGGNet) but implemented a novel element which is dubbed as the **Inception Module**.

5.5.1 Novel features of Inception

The authors of the Inception network took a different approach when designing their network architecture. As we've seen in the previous networks, there are some architectural decisions that you need to make for each layer when you are designing your network. Decisions like:

- **What should the kernel size of the convolutional layer be?** we've seen in previous architectures the kernel size varies between 1x1, 3x3, 5x5 and in some cases 11x11 like in AlexNet. When designing the CONV layer, we find ourselves trying to pick and tune the kernel size of each layer that fits our dataset. As you recall from chapter 3, smaller kernels capture finer details of the image whereas bigger filters will leave out minute details.
- **When to use the pooling layer?** AlexNet uses the pooling layers every 1 or 2 convolutional layer to downsize the spatial features. VGGNet applies pooling after every 2, 3, or 4 CONV layers as network gets deeper.

Configuring the kernel size and positioning the pool layers are decisions that you need to make mostly by trial and error and experiment with to get the optimal results. Inception says: instead of choosing a desired filter size in a CONV layer and where to place the pooling layers, let's apply all of them all together in one block and call it the "**Inception Module**".

Instead of stacking layers on top of each other like in classical architectures, the authors suggest that we create an "inception module" that consists of several convolutional layers with different kernel size. The architecture is then developed by stacking the inception modules on top of each other.

Let's take a look at how classical convolutional networks are architected vs. the Inception network:

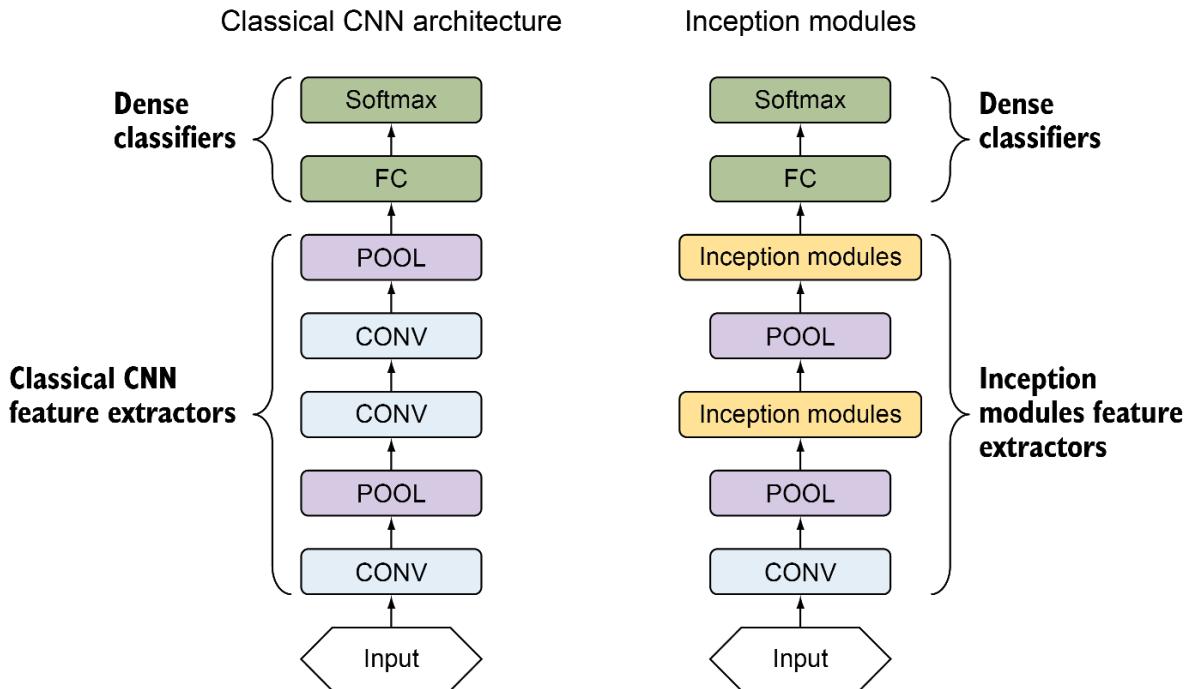


Figure 5.15

From the above diagram, you can observe the following:

- In classical architectures like LeNet, AlexNet, and VGGNet, we stack convolutional and pooling layers on top of each other to build the feature extractors. At the end we add the dense FC layers to build the classifier.
- In the Inception architecture, we start with a conv + pool layers then we stack the inception modules + pooling layers to build the feature extractors then add the regular dense classifier layers.

We've been treating the inception modules as black boxes to understand the bigger picture of the inception architecture. Now, we will unpack the inception module to understand how it works.

5.5.2 Inception module - naive version

The Inception module is a combination of four layers:

- 1) 1×1 Convolutional layer,
- 2) 3×3 Convolutional layer,
- 3) 5×5 Convolutional layer, and
- 4) 3×3 max-pooling layer

The outputs of these layers are then concatenated into a single output volume forming the input of the next stage. The naive representation of the inception module is represented in the figure below:

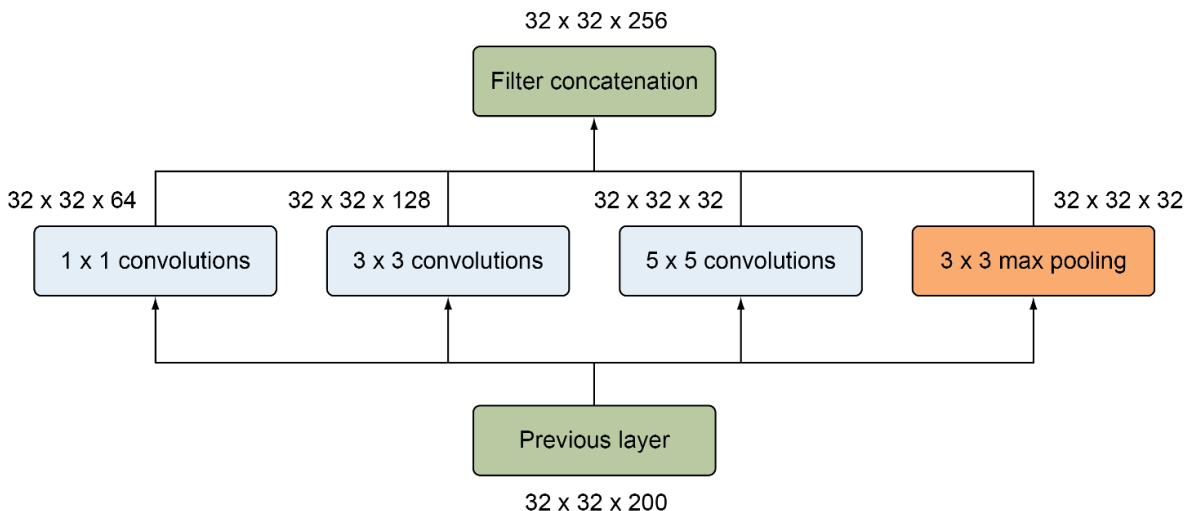


Figure 5.16

The diagram might look a little overwhelming but the idea is simple to understand. Let's follow along with this example:

1. Suppose we have an input dimensional volume from the previous layer of size = 32x32x200
2. We then feed this input to 4 convolutions simultaneously:
 - a. 1x1 conv with depth = 64 and same padding. The output of this kernel = 32x32x64
 - b. 3x3 conv with depth = 128 and same padding. Output = 32x32x128
 - c. 5x5 conv with depth = 32 and same padding. Output = 32x32x32
 - d. 3x3 max-pooling layer with same padding and stride = 1. Output = 32x32x32
3. Then, we concatenate the depth of the 4 outputs to create one output volume of dimensions = 32x32x256

Now we have an inception module that takes an input volume of 32x32x200 and outputs a volume of 32x32x256.

Why do we use padding = 'same' ?

Note that in the above example we used the "same" padding. In keras, padding can be set as 'same' or 'valid' as we seen in chapter 3. Where 'same' results in padding the input such that the output has the same length as the original input. We do that because we want the output to have similar width and height dimensions to the input. And the reason we want to output similar dimensions in the inception module is to simplify the depth concatenation process. Now we

can just add up all the outputs' depth to concatenate them into one output volume to be fed to the next layer in our network.

5.5.3 Inception module with dimensionality reduction

The naive representation of the inception module that we just saw has a big computational cost problem that comes with processing larger filters like the 5x5 convolutional layer. To get a better sense of the compute problem with the naive representation, let's calculate the number of operations that will be performed for the 5x5 CONV layer in the previous example.

The input volume with dimensions of 32x32x200 will be fed to the 5x5 conv of 32 filters with dimensions = 5x5x32. This means that total number of multiplies that the computer needs to compute is = $32 \times 32 \times 200$ multiplied by $5 \times 5 \times 32$ which is more than 163 million operations. While we can do this much of operations in modern computers, but this is still a pretty expensive one. This is when the dimensionality reduction layers can be very useful.

Dimensionality reduction layer (1x1 convolutional layers):

The 1x1 convolutional layer can reduce the operational cost of 163 million operations to about a tenth of that. That is why it is given the name "**reduce layer**". The idea here is to add a 1x1 CONV layer before the bigger kernels like the 3x3 and 5x5 CONV to reduce their depth which in turn will reduce the number of operations.

Let's look at the example below:

Suppose we have an input dimension volume of $32 \times 32 \times 200$. We then add a 1x1 CONV with depth = 16. This will reduce the dimension volume from 200 to 16 channels. We can then apply the 5x5 CONV on the output that has much less depth.

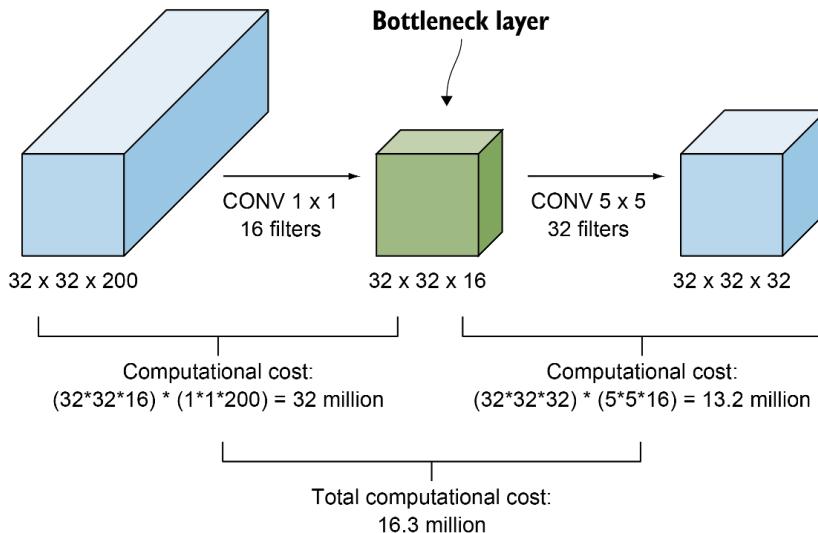


Figure 5.17

Notice that the input of $32 \times 32 \times 200$ is processed through the two conv layers and outputs a volume of dimensions $32 \times 32 \times 32$ which is the same dimension that we produced before without applying the dimensionality reduction layer. But what we've done here is, instead of processing the 5×5 conv layer on the entire 200 channels of the input volume, we're taking this huge volume and shrunk its representation to a much smaller intermediate volume which only has 16 channels.

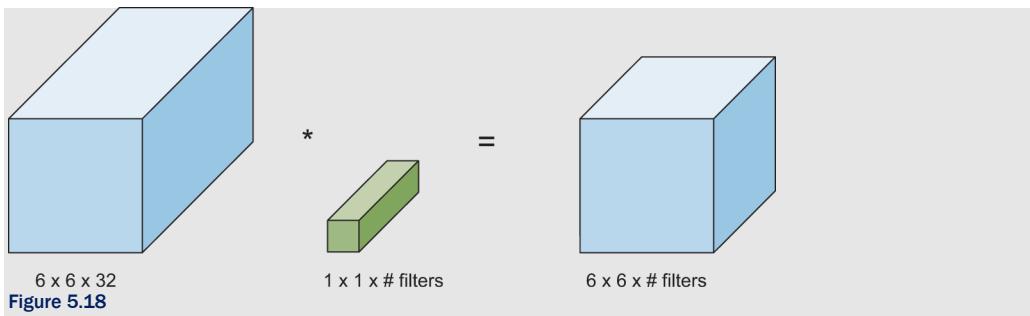
Now, let's look at the computational cost involved in this operation and compare it to the 163 million multiplications that we got before applying the reduce layer.

$$\begin{aligned}
 \text{Computation} &= \text{operations in the } 1 \times 1 \text{ convolution} + \text{operations in the } 5 \times 5 \text{ convolution} \\
 &= 32 \times 32 \times 200 \text{ multiplied by } 1 \times 1 \times 16 + 32 \times 32 \times 16 \text{ multiplied by } 5 \times 5 \times 32 \\
 &= 3.2 \text{ million} + 13.1 \text{ million}
 \end{aligned}$$

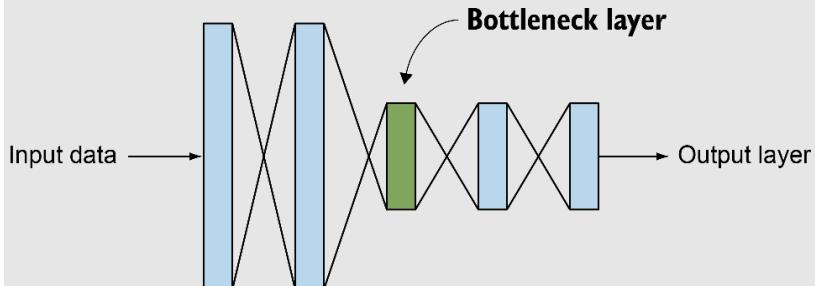
Total number of multiplications in this operation = 16.3 millions which is tenth of the 163 million multiplications that we calculated earlier without the reduce layers.

What is the 1×1 convolutional layer?

The idea of the 1×1 CONV layer is that it preserves the spatial dimensions (H & W) of the input volume but changes the number of channels of the volume (depth).



The 1×1 convolutional layers are also known as “**bottleneck layers**”. The analogy is made because the bottleneck is the smallest part of the bottle and the reduce layers reduce the dimensionality of the network making it look like a bottleneck.



What is the impact of dimensionality reduction on the network performance?

Now you might be wondering, does shrinking down the representation size so dramatically hurt the performance of the neural network? The authors ran experimentations and found out that as long as you implement this reduce layer with moderation, you can shrink down the representation size significantly without hurting the performance and saves a lot of computation.

Now let's put the reduce layers in action and build the new *inception module with dimensionality reduction*. To do that, we will keep the same concept of concatenating all the 4 layers that we had from the naive representation. We will add a 1×1 convolutional reduce layer before the 3×3 and 5×5 convolutions to reduce their computational cost. We will also add a 1×1 conv after the 3×3 max-pooling layer because pooling layers don't reduce the depth for their inputs. So, we will need to apply the reduce layer to their output before we do the concatenation. See the diagram below:

Inception module with dimensionality reduction

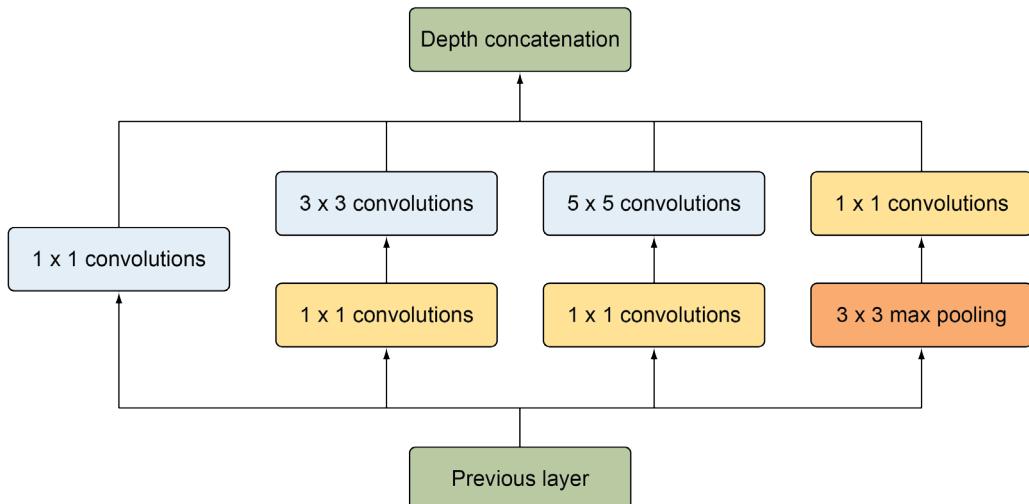


Figure 5.20

We add dimensionality reduction prior to bigger convolutional layers to allow for increasing the number of units at each stage significantly without an uncontrolled blow-up in computational complexity at later stages. Furthermore, the design follows the practical intuition that visual information should be processed at various scales and then aggregated so that the next stage can abstract features from the different scales simultaneously.

Recap of the inception modules

To summarize, if you are building a layer of a neural network and you don't want to have to decide what filter size to use in the convolutional layers or when to add pooling layer, the inception module let's you use them all and concatenate the depth of all the outputs. This is called the naive representation of the inception module.

We then run to the problem of computational cost that comes with using large size filters. In here we use a 1×1 convolutional layer called the reduce layer that reduces the computational cost significantly. We add the reduce layers before the 3×3 and 5×5 convolutional layers and after the max-pooling layer to create the inception module with dimensionality reduction.

5.5.4 Inception architecture

Okay, now that we understand the components of the inception module, we are ready to build the inception network architecture. In here, we are going to use the dimension reduction representation of the inception module and simply stack them on top of each other and add a 3×3 pooling layer in between for downsampling as you can see in the figure below.

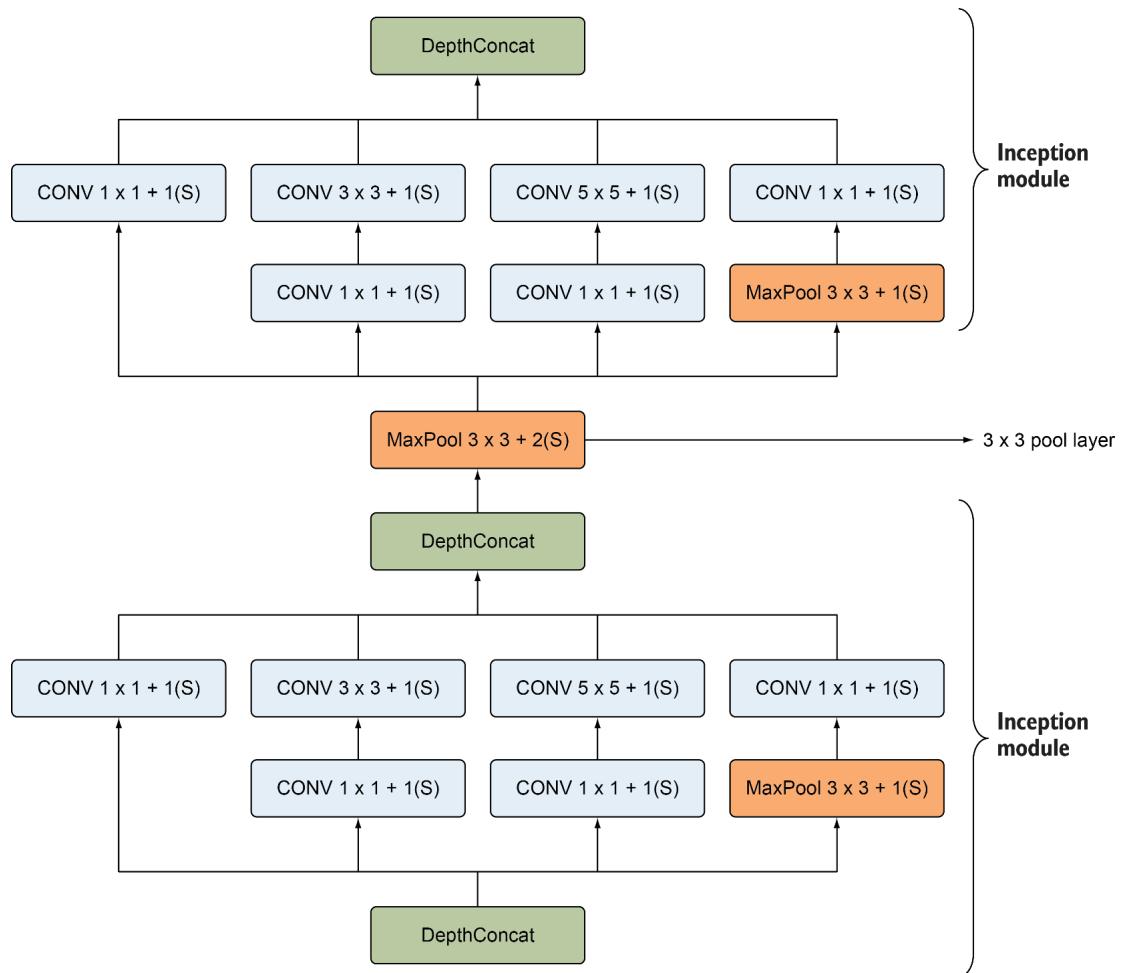


Figure 5.21

In the figure above, we stacked two inception modules with a pooling layer in between. We can stack as many inception modules as we want to build a very deep convolutional network. In the original paper, the authors built a specific incarnation of the inception module and called it **GoogLeNet**. They used this network in their submission for the ILSVRC 2014 competition.

The GoogLeNet architecture is depicted in the diagram below:

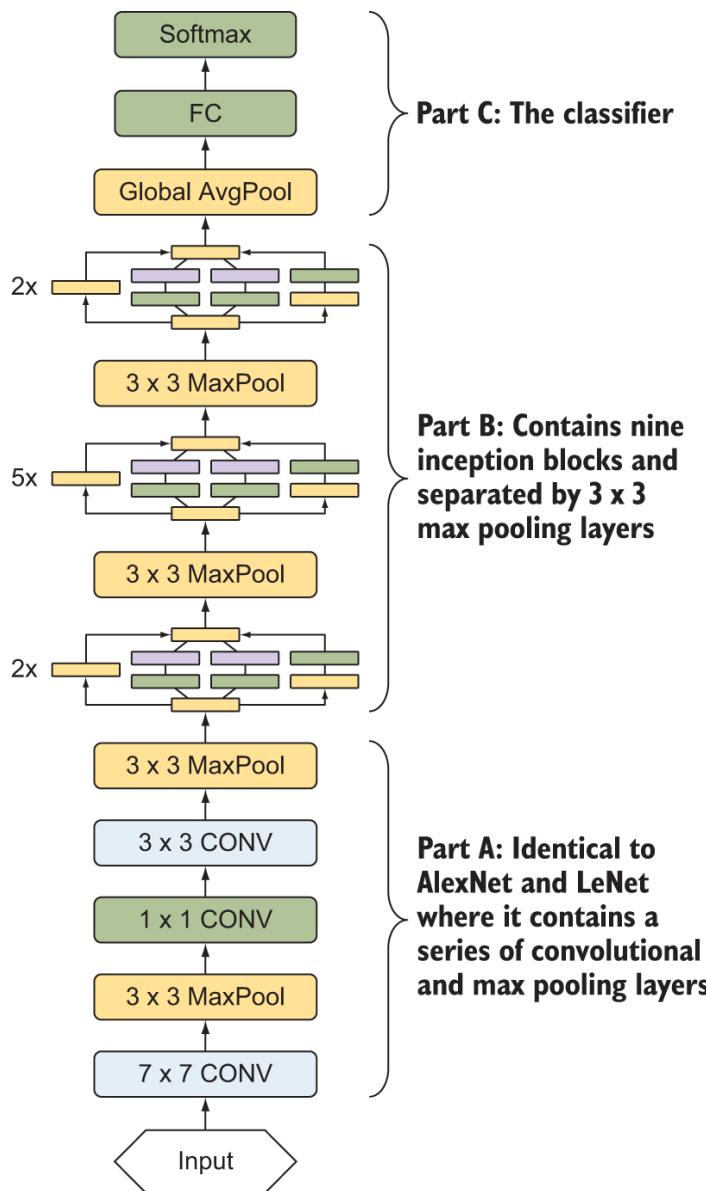


Figure 5.22 : Full GoogLeNet Model

As you can see in the diagram above, GoogLeNet uses a stack of a total of 9 inception blocks and max pooling layer after every several blocks to reduce the dimensionality. To simplify this

implementation, we are going to breakdown the GoogLeNet architecture into three parts A, B, and C:

- Part A: identical to AlexNet and LeNet architectures where it contains a series of CONV and POOL layers
- Part B: contains nine inception modules stacked as follows - 2 inception modules + pooling layer + 5 inception modules + pooling layer + 2 inception modules
- Part C: is the classifier part of the network where it consists of the fully connected and softmax layers

5.5.5 GoogleNet in Keras

Now, let's implement GoogleNet architecture in Keras. First we build the inception module function to use it in our architecture:

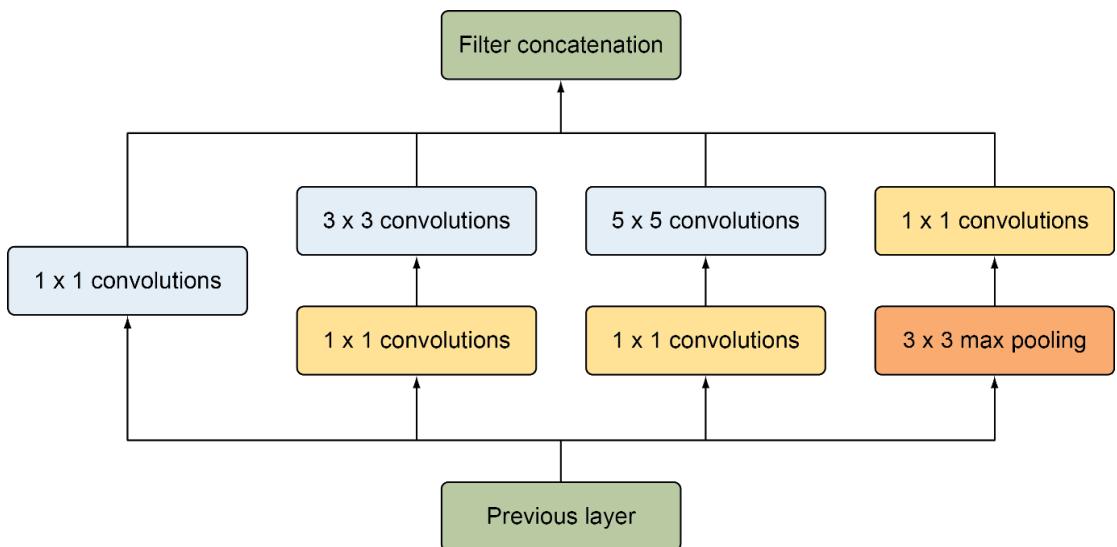


Figure 5.23

Notice that the inception module takes the features from the previous module as an input, passes it through 4 routes, then concatenate the depth of the output of all 4 routes, then pass the concatenated output to the next module. The four routes are as follows:

1. 1x1 conv
2. 1x1 conv + 3x3 conv
3. 1x1 conv + 5x5 conv
4. 3x3 pool + 1x1 conv

Now, let's build the *inception_module* function. The function takes the number of filters of each convolutional layer as an argument and returns the concatenated output.

```
def inception_module(x, filters_1x1, filters_3x3_reduce, filters_3x3, filters_5x5_reduce,
```

```

        Filters_5x5, filters_pool_proj, name=None):
# create the 1x1 convolution layer that takes its input directly from the previous
layer
conv_1x1 = Conv2D(filters_1x1, kernel_size=(1, 1), padding='same', activation='relu',
kernel_initializer=kernel_init, bias_initializer=bias_init)(x)

# 3x3 route = 1x1 conv + 3x3 conv
pre_conv_3x3 = Conv2D(filters_3x3_reduce, kernel_size=(1, 1), padding='same',
activation='relu', kernel_initializer=kernel_init, bias_initializer=bias_init)(x)
conv_3x3 = Conv2D(filters_3x3, kernel_size=(3, 3), padding='same', activation='relu',
kernel_initializer=kernel_init, bias_initializer=bias_init)(pre_conv_3x3)

# 5x5 route = 1x1 conv + 5x5 conv
pre_conv_5x5 = Conv2D(filters_5x5_reduce, kernel_size=(1, 1), padding='same',
activation='relu', kernel_initializer=kernel_init, bias_initializer=bias_init)(x)
conv_5x5 = Conv2D(filters_5x5, kernel_size=(5, 5), padding='same', activation='relu',
kernel_initializer=kernel_init, bias_initializer=bias_init)(pre_conv_5x5)

# pool route = pool layer + 1x1 conv
pool_proj = MaxPool2D((3, 3), strides=(1, 1), padding='same')(x)
pool_proj = Conv2D(filters_pool_proj, (1, 1), padding='same', activation='relu',
kernel_initializer=kernel_init, bias_initializer=bias_init)(pool_proj)

# concatenate the depth of the 3 filters together
output = concatenate([conv_1x1, conv_3x3, conv_5x5, pool_proj], axis=3, name=name)

return output

```

GoogLeNet architecture:

Now that we have the *inception_module* function ready, let's build the GoogLeNet architecture that we explained in the previous diagram. To get the values of the *inception_module* arguments, we are going to go through the table below that represents the hyperparameters setup as implemented by the authors in the original paper "Going Deeper with Convolutions paper".

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figure 5.24

Note that “#3×3 reduce” and “#5×5 reduce” in the table below represent the 1×1 filters in the reduction layers that are used before the 3×3 and 5×5 convolutions.

Now, let's go through the implementations of parts A, B, and C.

Part A: build the bottom part of the network

Let's build the bottom part of the network. This part consists of: 7x7 CONV > 3x3 POOL > 1x1 CONV > 3x3 CONV > 3x3 POOL as you can see in the diagram below.

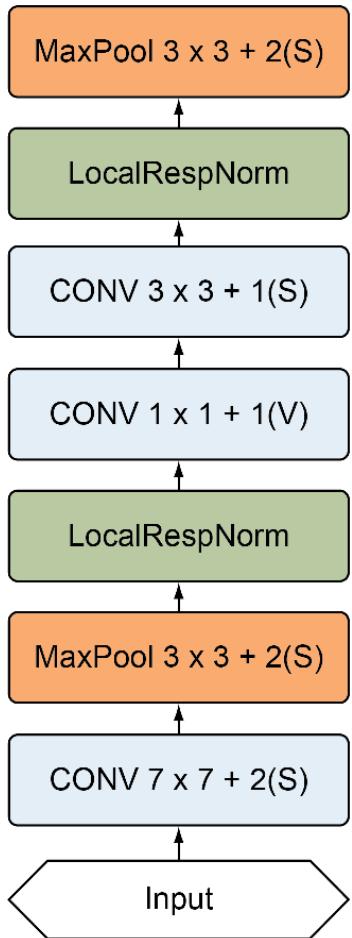


Figure 5.25

LocalResponseNorm layer: similar to AlexNet, a local response normalization is used. It is a normalization technique that helps speed up the convergence. Nowadays, batch normalization (BN) is used instead of using local response normalization and we will be using BN in our implementation in this chapter.

```

# input layer with size = 24x24x3
input_layer = Input(shape=(224, 224, 3))

x = Conv2D(64, (7, 7), padding='same', strides=(2, 2), activation='relu',
           name='conv_1_7x7/2', kernel_initializer=kernel_init,
           bias_initializer=bias_init)(input_layer)

x = MaxPool2D((3, 3), padding='same', strides=(2, 2), name='max_pool_1_3x3/2')(x)

```

```

x = BatchNormalization()(x)

x = Conv2D(64, (1, 1), padding='same', strides=(1, 1), activation='relu')(x)
x = Conv2D(192, (3, 3), padding='same', strides=(1, 1), activation='relu')(x)

x = BatchNormalization()(x)

x = MaxPool2D((3, 3), padding='same', strides=(2, 2))(x)

```

Part B:

Build inception modules 3a and 3b + the max pooling layer

type	#1x1	#3x3 reduce	#3x3	#5x5 reduce	#5x5	Pool proj
inception (3a)	64	96	128	16	32	32
inception (3b)	128	128	192	32	96	64

```

x = inception_module(x, filters_1x1=64, filters_3x3_reduce=96, filters_3x3=128,
                     filters_5x5_reduce=16, filters_5x5=32, filters_pool_proj=32, name='inception_3a')

x = inception_module(x, filters_1x1=128, filters_3x3_reduce=128, filters_3x3=192,
                     filters_5x5_reduce=32, filters_5x5=96, filters_pool_proj=64, name='inception_3b')

x = inception_module(x, filters_1x1=64, filters_3x3_reduce=96, filters_3x3=128,
                     filters_5x5_reduce=16, filters_5x5=32, filters_pool_proj=32, name='inception_3a')

x = inception_module(x, filters_1x1=128, filters_3x3_reduce=128, filters_3x3=192,
                     filters_5x5_reduce=32, filters_5x5=96, filters_pool_proj=64, name='inception_3b')

x = MaxPool2D((3, 3), padding='same', strides=(2, 2))(x)

```

Similarly, let's create modules 4a, 4b, 4c, 4d, 4e, and the max pooling layer:

```

x = inception_module(x, filters_1x1=192, filters_3x3_reduce=96, filters_3x3=208,
                     filters_5x5_reduce=16, filters_5x5=48, filters_pool_proj=64, name='inception_4a')

x = inception_module(x, filters_1x1=160, filters_3x3_reduce=112, filters_3x3=224,
                     filters_5x5_reduce=24, filters_5x5=64, filters_pool_proj=64, name='inception_4b')

x = inception_module(x, filters_1x1=128, filters_3x3_reduce=128, filters_3x3=256,
                     filters_5x5_reduce=24, filters_5x5=64, filters_pool_proj=64, name='inception_4c')

x = inception_module(x, filters_1x1=112, filters_3x3_reduce=144, filters_3x3=288,
                     filters_5x5_reduce=32, filters_5x5=64, filters_pool_proj=64, name='inception_4d')

x = inception_module(x, filters_1x1=256, filters_3x3_reduce=160, filters_3x3=320,
                     filters_5x5_reduce=32, filters_5x5=128, filters_pool_proj=128, name='inception_4e')

x = MaxPool2D((3, 3), padding='same', strides=(2, 2), name='max_pool_4_3x3/2')(x)

```

Now, let's create modules 5a and 5b:

```

x = inception_module(x, filters_1x1=256, filters_3x3_reduce=160, filters_3x3=320,
                     filters_5x5_reduce=32, filters_5x5=128, filters_pool_proj=128, name='inception_5a')

x = inception_module(x, filters_1x1=384, filters_3x3_reduce=192, filters_3x3=384,
                     filters_5x5_reduce=48, filters_5x5=128, filters_pool_proj=128, name='inception_5b')

```

Part D: the classifier part

In their experimentations, the authors found that adding an 7×7 average pooling improved the top-1 accuracy by about 0.6%. Then added a dropout layer with 40% probability to reduce overfitting.

```
x = AveragePooling2D(pool_size=(7,7), strides=1, padding='valid')(x)
x = Dropout(0.4)(x)
x = Dense(10, activation='softmax', name='output')(x)
```

5.5.6 Learning hyperparameters

The authors used stochastic gradient descent optimizer with 0.9 momentum. They also implemented a fixed learning rate decay schedule of 4% every 8 epochs.

```
epochs = 25
initial_lrate = 0.01

# implement the learning rate decay function
def decay(epoch, steps=100):
    initial_lrate = 0.01
    drop = 0.96
    epochs_drop = 8
    lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
    return lrate

lr_schedule = LearningRateScheduler(decay, verbose=1)

sgd = SGD(lr=initial_lrate, momentum=0.9, nesterov=False)

model.compile(loss=['categorical_crossentropy', 'categorical_crossentropy',
                    'categorical_crossentropy'], loss_weights=[1, 0.3, 0.3], optimizer=sgd,
               metrics=['accuracy'])

model.fit(X_train, [y_train, y_train, y_train], validation_data=(X_test, [y_test, y_test,
                                                                     y_test]), epochs=epochs, batch_size=256, callbacks=[lr_schedule])
```

5.5.7 Inception performance on CIFAR dataset

GoogleNet is the winner of the ILSVRC 2014 competition. It achieved a top-5 error rate of 6.67% which was very close to human level performance and much better than the previous CNNs like AlexNet and VGGNet.

5.6 ResNet

Residual Neural Network (ResNet) was developed in 2015 by Kaiming He et al, from the Microsoft Research team, in their paper "[Deep Residual Learning for Image Recognition](#)". They introduced a novel architecture with "skip connections" called **residual module**. The network also features heavy batch normalization for the hidden layers. This technique allowed the authors to train a very deep neural networks with 50, 101, and 152 weight layers while still having lower complexity than smaller networks like VGGNet (19 layers). ResNet was able to achieve a top-5 error rate of 3.57% in the ILSVRC15 which beats the performance of all prior convnets.

5.6.1 Novel features of ResNet

Looking at how neural network architectures evolved from LeNet, AlexNet, VGGNet, and Inception you might have noticed that the deeper the network, the larger learning capacity it has, and the better it extracts features from images. This mainly happens because very deep networks are able to represent very complex functions that allows the network to learn features at many different levels of abstraction, from edges (at the lower layers) to very complex features (at the deeper layers).

Earlier in this chapter, we saw deep neural networks like VGGNet-19 that contains 19 layers and GoogLeNet that contains 22 layers. Both have performed very well in the ImageNet challenge. But can we build even deeper networks? We learned from chapter 4 that one downside from adding too many layers is that it makes the network more prone to overfit the training data. This is not a big problem because there are many regularization techniques that we learned in chapter 4 that we can use to avoid overfitting like dropout, L2 regularization, and batch normalization. So, if we take care of the overfitting problem, wouldn't we want to build very deep networks that are 50, 100, or even 150 layers deep? The answer is, Yes. We definitely should try to build very deep neural networks. Only one other problem that we need to fix to unblock the capability of building super deep networks for us. It is a phenomenon that is called the **vanishing gradients**.

Vanishing and exploding gradients problem

The problem with very deep networks is that the signal required to change the weights becomes very small at the earlier layers. To understand why, let's look at the gradient descent process explained in chapter 2. As the network backpropagates the gradient of the error from the final layer back to the first layer, it is multiplying by the weight matrix on each step, and thus the gradient can decrease exponentially quickly to zero leading to a vanishing gradient phenomenon that prevents the earlier layers from learning. As a result, the network's performance gets saturated or even starts degrading rapidly.

In other cases the gradient might grow exponentially quickly and "explode" to take very large values. This phenomenon is called the **exploding gradients**.

To solve the vanishing gradient problem, the authors created a shortcut that allows the gradient to directly backpropagate to earlier layers. These shortcuts are called "**skip connections**". The skip connections are used to flow information from earlier layers in the network to later layers creating an alternate shortcut path for the gradient to flow through. Another important benefit of the skip connections is that they allow the model to learn an identity function which ensures that the layer will perform at least as good as the previous layer.

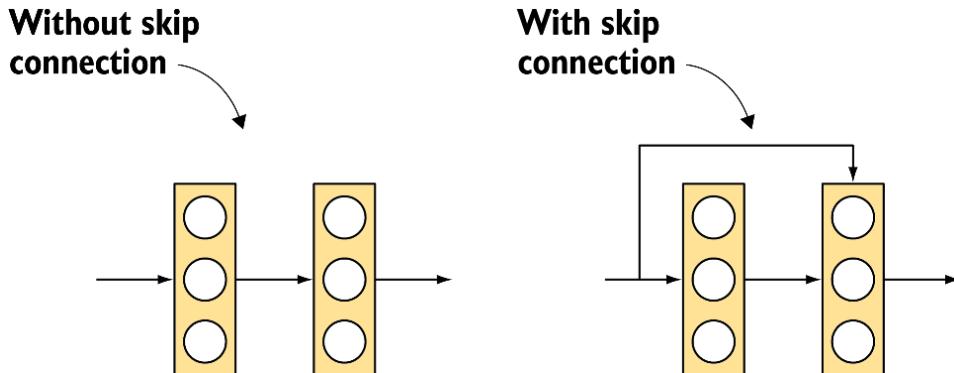


Figure 5.26

The figure on the left is the traditional stacking of convolution layers together one after the other. On the right we still stack convolution layers as before but we now also add the original input to the output of the convolution block. This is called skip connection. We then add both signals; the skip connection + the main path.

Note that the shortcut arrow is pointing to the end of the second convolutional layer. **Not after.** The reason is that you will do the addition of both paths before you apply the activation function 'relu' of this layer. It goes like this:

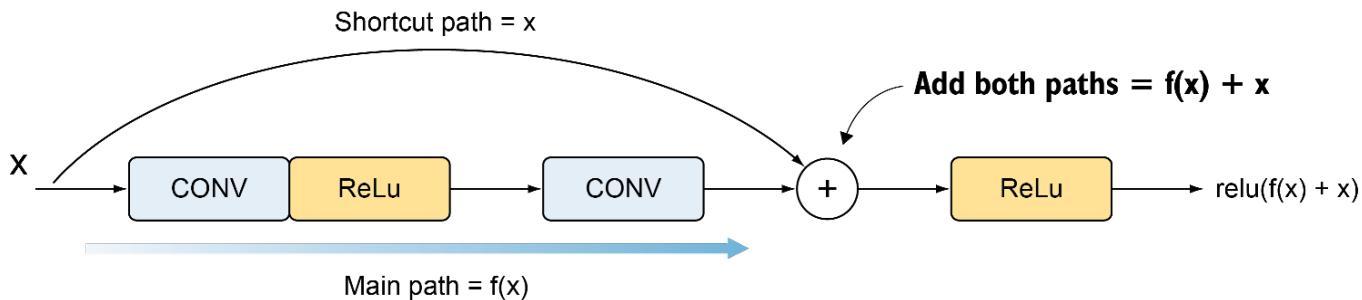


Figure 5.27

As you can see in the figure above, the X signal is passed along the shortcut path and then added to the main path $f(x)$. Then, we apply the relu activation to $f(x) + x$ to produce the output signal $= \text{relu}(f(x) + x)$

The code implementation of the skip connection is straight forward. Look at the code snippet below:

```
# You first store the value of the shortcut to be equal to the input x
X_shortcut = X

# Then perform the main path operations: CONV+ReLU + CONV
X = Conv2D(filters = F1, kernel_size = (3, 3), strides = (1,1))(X)
X = Activation('relu')(X)
```

```

X = Conv2D(filters = F1, kernel_size = (3, 3), strides = (1,1))(X)

# Then add the both paths together
X = Add()([X, X_shortcut])

# and finally, apply the relu activation function
X = Activation('relu')(X)

```

This combination of the skip connection and convolutional layers is called the **residual block**. Similar to the Inception network, ResNet is composed of a series of building blocks that are stacked on top of each other. These building blocks are called residual blocks.

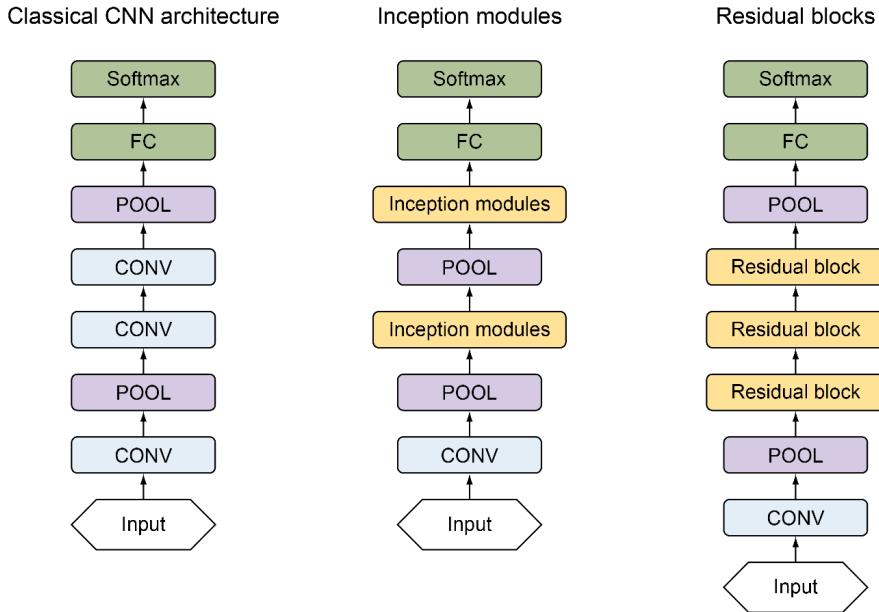


Figure 5.28

From the above diagram, you can observe the following:

- **Feature extractors:** to build the feature extractors part of the ResNet, we start with a **CONV + POOL** layer, then stack residual blocks on top of each other to build the network. When we are designing our ResNet network, we can add as many residual blocks as we want to build even deeper networks.
- **Classifiers:** the classification part is still the same as we learned in other networks. Fully-connected layers followed by a softmax.

Now that you know what a skip connection is and you are familiar with the high-level architecture of ResNets, let's unpack the residual blocks to understand how they work.

5.6.2 Residual blocks

The residual module consists of two branches:

1. Shortcut path: which connects the input to an addition of the second branch
2. Main path: a series of convolutions and activations. The main path consists of 3 convolutional layers with relu activations. We will also add batch normalization to each CONV layer to reduce overfitting and speed up training. The main path architecture looks like this: [CONV > BN > ReLU] x 3.

Residual blocks

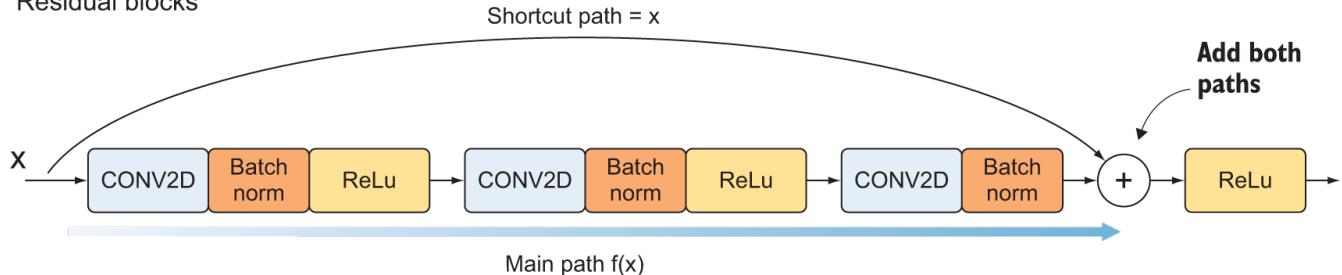


Figure 5.29

Similar to what we explained in the previous page, the shortcut path is added to the main path right before the activation function of the last CONV layer. Then we apply the 'relu' function after adding the two paths.

Notice that there are no pooling layers in the residual block. Instead, the authors of ResNet decided to do dimensions downsampling using bottleneck 1x1 convolutional layers similar to the Inception network. So, each residual block will start with a 1x1 CONV to downsample the input dimension volume + 3x3 CONV + another 1x1 CONV to downsample the output. This is a good technique to keep control of the volume dimensions across many layers. This configuration of the residual block is called the **bottleneck residual block**.

Now, you are ready to start building your ResNet in code. One thing left. When you are stacking residual blocks on top of each other, the volume dimensions change from one block to another. And as you might recall from the matrices introduction in chapter 2, to be able to perform the matrix addition operations, the matrices should have similar dimensions. To fix this problem, we need to downsample the shortcut path as well before merging both paths. We do that by adding a bottleneck layer (1x1 CONV + BN) to the shortcut path as you see in the diagram below. This is called the **reduce shortcut**.

Bottleneck residual block with reduce shortcut

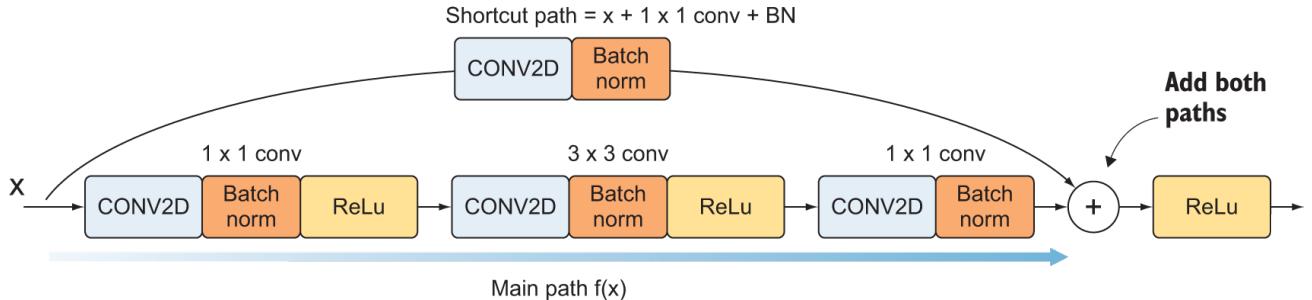


Figure 5.30

Before we jump in to the code implementation, let's recap what we discussed in residual blocks:

- Residual blocks contain two paths: the shortcut path and the main path
- The main path consists of three CONV layers and we add a batch norm layer to them
 - 1x1 conv
 - 3x3 conv
 - 1x1 conv
- There are two ways we can implement the shortcut path:
 - Regular shortcut: in which we just add the input dimensions to the main path
 - Reduce shortcut: in which we add a CONV layer in the shortcut path before merging with the main path

When we are implementing the ResNet network, we will use both regular and reduce shortcuts. This will get clearer when you see the full implementation soon. But for now, we will implement *bottleneck_residual_block* function that takes a boolean argument *reduce*. When *reduce* = True, this means we want to use the reduce shortcut, else it will implement the regular short cut. The function takes the following arguments:

- X -- input tensor of shape (number of samples, height, width, channel)
- f -- integer, specifying the shape of the middle CONV's window for the main path
- filters -- python list of integers, defining the number of filters in the CONV layers of the main path
- reduce -- boolean, True = identifies the reduction layer
- s -- integer, strides

And returns:

- X -- output of the residual block, tensor of shape (height, width, channel)

```
def bottleneck_residual_block(X, kernel_size, filters, reduce=False, s=1):
    # unpack the tuple to retrieve Filters of each CONV Layer
```

```

F1, F2, F3 = filters

# Save the input value to use it later to add back to the main path.
X_shortcut = X

# if condition if reduce is True
if reduce:
    # if we are to reduce the spatial size, apply a 1x1 CONV layer to the shortcut
    # path
    # to do that, we need both CONV layers to have similar strides
    X_shortcut = Conv2D(filters = F3, kernel_size = (1, 1), strides =
(s,s))(X_shortcut)
    X_shortcut = BatchNormalization(axis = 3)(X_shortcut)

# if reduce, we will need to set the strides of the first conv to be similar to the
# shortcut strides
    X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (s,s), padding =
'valid')(X)
    X = BatchNormalization(axis = 3)(X)
    X = Activation('relu')(X)

else:
    # First component of main path
    X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (1,1), padding =
'valid')(X)
    X = BatchNormalization(axis = 3)(X)
    X = Activation('relu')(X)

    # Second component of main path
    X = Conv2D(filters = F2, kernel_size = kernel_size, strides = (1,1), padding =
'same')(X)
    X = BatchNormalization(axis = 3)(X)
    X = Activation('relu')(X)

    # Third component of main path
    X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding =
'valid')(X)
    X = BatchNormalization(axis = 3)(X)

    # Final step: Add shortcut value to main path, and pass it through a RELU
    # activation
    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)

return X

```

5.6.3 ResNet implementation in Keras

Alright, we've learned a lot about residual blocks so far. Let's add these blocks on top of each other to build the full ResNet architecture. In this chapter, we will implement ResNet50. It is a version of the ResNet architecture that contains 50 weight layers, hence the name ResNet50. You can use the same approach to develop ResNet with 18, 34, 101, and 152 layers by following the architecture in the table below from the [Deep Residual Learning for Image Recognition](#) paper.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
conv2_x	56×56			3×3 max pool, stride 2		
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1			average pool, 1000-d fc, softmax		
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 5.31

We know from the previous section that each residual module contains 3 × CONV layers, we now can compute the total number of the weight layers inside the ResNet50 network as follows:

- Stage 1: 7x7 CONV layer
- Stage 2: 3 residual blocks, each contain 1x1 CONV + 3x3 CONV + 1x1 CONV] = total of 9 CONV layers
- Stage 3: 4 residual blocks = total of 12 CONV layers
- Stage 4: 6 residual blocks = total of 18 convolutional layers
- Stage 5: 3 residual blocks = total of 9 convolutional layers
- FC Softmax layer

When you sum all these layers together, you will get a total of 50 weight layers that describe the architecture of ResNet50. Similarly, you can compute the number of weight layers in the other ResNet versions.

Regular shortcut vs. reduce shortcut

In the implementation below, we will use the residual block with reduce shortcut at the beginning of each stage to reduce the spatial size of the output from the previous layer. Then we use the regular shortcut for the remaining layers of that stage. As you recall from our implementation of the `bottleneck_residual_block`, we will set the argument `reduce` = True to apply the reduce shortcut.

Now let's follow the 50-layer architecture in the table above to build the ResNet50 network. We will build a `ResNet50` function that takes the `input_shape` and `classes` as arguments and outputs the `model`.

```
def ResNet50(input_shape, classes):
    # Define the input as a tensor with shape input_shape
    X_input = Input(input_shape)

    # Stage 1
    X = Conv2D(64, (7, 7), strides=(2, 2), name='conv1')(X_input)
    X = BatchNormalization(axis=3, name='bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3, 3), strides=(2, 2))(X)

    # Stage 2
    X = bottleneck_residual_block(X, 3, [64, 64, 256], reduce=True, s=1)
    X = bottleneck_residual_block(X, 3, [64, 64, 256])
    X = bottleneck_residual_block(X, 3, [64, 64, 256])

    # Stage 3
    X = bottleneck_residual_block(X, 3, [128, 128, 512], reduce=True, s=2)
    X = bottleneck_residual_block(X, 3, [128, 128, 512])
    X = bottleneck_residual_block(X, 3, [128, 128, 512])
    X = bottleneck_residual_block(X, 3, [128, 128, 512])

    # Stage 4
    X = bottleneck_residual_block(X, 3, [256, 256, 1024], reduce=True, s=2)
    X = bottleneck_residual_block(X, 3, [256, 256, 1024])
    X = bottleneck_residual_block(X, 3, [256, 256, 1024])

    # Stage 5
    X = bottleneck_residual_block(X, 3, [512, 512, 2048], reduce=True, s=2)
    X = bottleneck_residual_block(X, 3, [512, 512, 2048])
    X = bottleneck_residual_block(X, 3, [512, 512, 2048])

    # AVGPOOL
    X = AveragePooling2D((1,1))(X)

    # output layer
    X = Flatten()(X)
    X = Dense(classes, activation='softmax', name='fc' + str(classes))(X)

    # Create the model
    model = Model(inputs = X_input, outputs = X, name='ResNet50')

    return model
```

5.6.4 Learning hyperparameters

The authors followed a similar training procedure to AlexNet. Namely, the training is carried out using mini-batch gradient descent SGD with momentum = 0.9. They set the learning rate to start with value = 0.1 , and then decreased by a factor of 10 when the validation error stopped improving. They also used L2 regularization with weight decay of 0.0001 that is not implemented in this chapter for simplicity. As you saw in the implementation above, they used

batch normalization (BN) right after each convolution and before activation to speed up training.

```
from keras.callbacks import ReduceLROnPlateau

# set the training parameters
epochs = 200
batch_size = 256

# min_lr: lower bound on the learning rate
# factor: factor by which the learning rate will be reduced
reduce_lr= ReduceLROnPlateau(monitor='val_loss',factor=np.sqrt(0.1),patience=5, min_lr=0.5e-6)

# compile the model
model.compile(loss='categorical_crossentropy', optimizer=SGD, metrics=['accuracy'])

# train the model
# call the reduce_lr value using callbacks in the training method
model.fit(X_train, Y_train, batch_size=batch_size, validation_data=(X_test, Y_test),
          epochs=epochs, callbacks=[reduce_lr])
```

5.6.5 ResNet performance on CIFAR dataset

Similar to the other networks explained in this chapter, the performance of ResNet models are benchmarked their results in the ImageNet challenge (ILSVRC). ResNet-152 won the first place in the 2015 classification competition with top-5 error rate of 4.49% with a single model and lowered to 3.57% using an ensemble of models. Which is much better than all the other networks like GoogLeNet (Inception) that achieved a top-5 error rate of 6.67%. ResNet also won the first place in many object detection and image localization challenges as we will see the in Object Detection chapter in this book. More importantly, the residual blocks concept in ResNet opened the door to a whole new possibilities to efficiently train super deep neural networks with hundreds of layers.

Using open-source implementation

Now that you have learned some of the most popular CNN architectures, I want to share with you some practical advice on how to use them. It turns out that a lot of these neural networks are difficult or finicky to replicate because a lot of details about tuning of the hyperparameters such as learning decay and other things that make some difference to the performance. Deep learning researchers can even have a hard time replicating someone else's polished work just from reading their paper. Fortunately, a lot of deep learning researchers routinely open source their work on the Internet, such as on GitHub. A simple search on Github on the network implementation would point you towards implementations in several deep learning libraries that you can clone and train. Because if you can get the author's implementation, you can usually get going much faster than trying to reimplement it from scratch. Although sometimes reimplementing from scratch could be a good exercise to do as well like what we did earlier.

5.7 Summary and takeaways

- In this chapter, we've explained the network architectures of five popular CNN networks. The classical CNN architectures: LeNet, AlexNet, and VGGNet, and the advanced architectures: ResNet, and Inception (GoogleNet).
- **Classical CNN architectures** have the same classical architecture of stacking convolution and pooling layers on top of each other with different configurations for their layers.
- **LeNet:** consists of 5 weight layers; 3 convolutional and 2 fully-connected layers with a pooling layer after the first and second convolutional layers.
- **AlexNet:** it is deeper than LeNet where it contains 8 weight layers; 5 convolutional and 3 fully-connected layers. AlexNet used larger filter sizes 11x11, 5x5, and 3x3 and pooling layers after the first, second and the fifth convolutional layers.
- **VGGNet:** it solved the problem of setting up the hyperparameters of the conv. and pooling layers by creating a one uniform configuration for them to be used across the entire network. All convolutional layers in the VGG network have 3x3 filter size with stride = 1 and padding = 'same'. Also all pooling layers have a pool size of 2x2 and a stride = 2. The idea being, using 3x3 CONV layers with larger depth performs better than less larger filters like 5x5 and 11x11.
- We also discussed the different VGGNet configurations like VGG-16 that contains 16 weight layers and VGG-19 that is 19 layers deep.
- **Advanced CNN architectures:** We then went beyond the classic networks and discussed two more advanced, even more powerful neural network architectures: Inception and ResNet.
- **Inception:** tried to solve the same problem that VGGNet is solving. It was trying to save all the trial and error work to configure the convolutional and pooling layers. Instead of having to decide which filter size to use and where to add the pooling layer, Inception network says: "Let's use them all". The authors created an inception module that contains three convolutional layers with filter sizes of 1x1, 3x3, and 5x5 in addition to a 3x3 pooling layer. The outputs are then concatenated creating a single volume that will be fed to the next inception module.
- **ResNet:** it followed the same approach as Inception and created residual blocks that when stacked on top of each other, they form the network architecture. ResNet attempted to solve the vanishing gradient problem that made the learning plateaus or degrades when training very deep neural networks. The authors introduced the "skip connections" idea that allows information to flow from earlier layers in the network to later layers creating an alternate shortcut path for the gradient to flow through. The fundamental breakthrough with ResNet was that it allowed us to train extremely deep neural networks with hundreds of layers. Prior to ResNet training very deep neural networks was difficult due to the problem of vanishing gradients.

Below is a summary of the classical and advanced networks:

Table 5.2

Year	CNN	Number of layers	Top-5 error rate	Number of parameters
1998	LeNet	5 layers	NA	60 thousand
2012	AlexNet	8 layers	15.3%	60 million
2014	VGGNet	16 layers	7.3%	138 million
2014	Inception (GoogLeNet)	22	6.67%	12 million
2015	ResNet-152	152	4.49%	

6

Transfer Learning

"Share your knowledge. It's a way to achieve immortality"

-- Dalai Lama

Learning goals from this chapter

- Understand transfer learning technique
- Understand how transfer learning works
- Learn the different transfer learning approaches
- Learn how to choose the learning approach for your problem
- Explore some of the most popular open-source image datasets that you can download and train your model for your own task
- Build two end-to-end transfer learning projects using different transfer learning approaches

When you're building a computer vision application, you can build your convnets as we learned in chapter three and start the training from scratch. And that is an acceptable approach. Another much faster approach is to download a neural network that someone else has already built and trained on a large dataset in a certain domain and use this pretrained network as a starting point to train the network on your new task. This approach is called **transfer learning**.

Transfer learning is one of the most important techniques of deep learning. When building a vision system to solve a specific problem, you usually need to collect and label a huge amount of data to train your network. But what if we could use an existing neural network, that someone else has tuned and trained, and use it as a starting point for our new task? Transfer learning allows us to do just that. We can download an open-source model that someone else

has already trained and tuned for weeks and use their optimized parameters (weights) as a starting point to train our model just a little bit more on a smaller dataset that we have for a given task. This way we can train our network a lot faster and achieve very high results.

Deep learning researchers and practitioners have posted a lot of research papers and open-source projects of their trained algorithms that they have worked on for weeks and months and trained on many GPUs to get state-of-the-art results on many problems. The fact that someone else has done this work and gone through the painful high-performance research process, means that you can often download open source architecture and weights that took someone else many weeks or months to build and tune and use that as a very good start for your own neural network. This is **transfer learning**. It is referring to the knowledge transfer from pretrained network in one domain to your own problem in a different domain.

In this chapter, I will explain transfer learning and I will outline reasons why using it is important. I will also detail different transfer learning scenarios and how to use them. Finally, we will see examples of using transfer learning to solve real-world problems.

Ready? Let's get started!

6.1 What are the problems that transfer learning is solving?

As the name implies, transfer learning means transferring what a neural network has learned from being trained on a specific dataset to another related problem. Transfer learning is currently very popular in the field of deep learning because it enables you to train deep neural networks with comparatively little data and a short amount of training time. The importance of transfer learning comes from the fact that in most real-world problems, we typically do not have millions of labeled images to train such complex models.

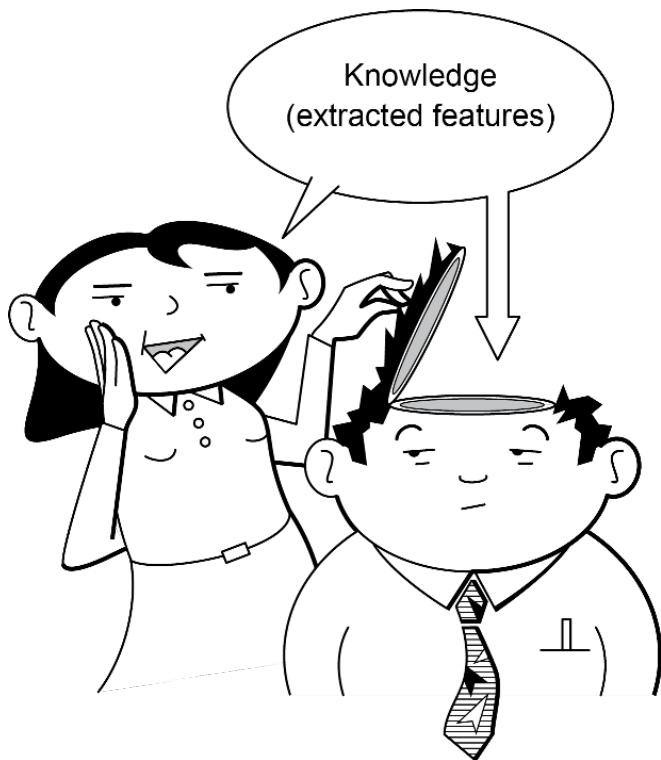


Figure 6.1: Transfer learning is the transfer of the knowledge that the network has acquired from one task to another new task.

The idea is pretty straightforward. First we train a deep neural network on a very large amount of data. During the training process, the network will extract a large amount of useful features that can be used to detect objects in this dataset. We then transfer these extracted features (feature maps) to a new network. We then train this new network on our new dataset to solve a different problem. Transfer learning is a great way to shortcut the process of collecting and training huge amount of data simply by reusing the model weights from pretrained models that were developed for standard computer vision benchmark datasets,

such as the ImageNet image recognition tasks. Top performing models can be downloaded and used directly, or integrated into a new model for your own.

The question here is; why would we want to use transfer learning? Why don't we just train a neural network directly on our new dataset to solve our problem? To answer this question, we first need to know what are the main problems that transfer learning is solving. This is what we are going to discuss now. Then I'll go into the details of how transfer learning works and the different approaches to apply transfer learning.

Deep neural networks are immensely data-hungry and rely on huge amounts of labeled data to achieve high performance. In practice, very few people train an entire convolutional network from scratch. This is because of two main problems:

- **Data problem:** it requires a lot of data to be able to get decent results which is not very feasible in most cases. It is relatively rare to have a dataset of sufficient size to solve your problem. It is also very expensive to acquire and label data which is mostly a manual process that has to be done by humans capturing images and labeling them one-by-one which makes it a non-trivial, very expensive task.
- **Computation problem:** even if you are able to acquire hundreds of thousands of images for your problem, it is computationally very expensive to train a deep neural network on millions of images. The training process of a deep neural network from scratch is very expensive because it usually requires weeks of training on multiple GPUs. Also Keep in mind that the neural network training process is an iterative process. So, even if you happen to have the computing power that is needed to train complex neural networks, having to spend a few weeks experimenting different hyperparameters in each training iteration will make the project very expensive until you finally reach satisfactory results.

Additionally, one very important benefit of using transfer learning is that it helps the model generalize its learnings and avoid overfitting. When applying a deep learning model in the wild, it is faced with countless conditions which the model might have never seen before and does not know how to deal with; each client has their own preferences and generates data that is different than the data that is used for training. The model is asked to perform well on many tasks that are related to but not exactly similar to the task it was trained for.

For example, when you deploy a car classifier model to production, people usually have different camera types, each with its own image quality and resolution. Also images can be taken during different weather conditions. All these different image nuances vary from one user to another. To train the model on all these different cases you either have to account for each one of these cases and acquire a lot of images to train the network on them or try to build a more robust model that does better in generalizing to new use cases. This is what transfer learning does, since it is not realistic to account for all the cases that the model can face in the wild, transfer learning can help us deal with these novel scenarios and is necessary for production-scale use of deep learning that goes beyond tasks and domains where labeled data is plentiful. This is because we are transferring the features extracted from another network that has seen millions of images which will make our model less prone to overfit and generalize better when faced with novel scenarios. You will be able to fully grasp this concept when we explain how transfer learning works in the following sections.

6.2 What is transfer learning?

Armed with the understanding of the problems that transfer learning is solving, let's look at its formal definition. Transfer learning is the transfer of the knowledge (feature maps) that the network has acquired from one task where we have a large amount of data to a new task where data is not abundantly available. It is usually used where a neural network model is first trained on a problem similar to the problem that is being solved. One or more layers from the trained model are then used in a new model trained on the problem of interest.

As we discussed earlier, to train an image classifier that will achieve near or above human level accuracy on image classification, we'll need massive amounts of data, large compute power, and lots of time on our hands. This I'm sure most of us don't have. Knowing this would be a problem for people with little or no resources, researchers built state-of-the-art models that were trained on large image datasets like ImageNet, MS COCO, Open Images, etc. and decided to share their models to the general public for reuse. This means you should never have to train an image classifier from scratch again, unless you have an exceptionally large data set and a very large computation budget to train everything from scratch by yourself. Even if that is the case, you might be better off using transfer learning to fine-tune the pretrained network on your large dataset. Later in this chapter we will discuss the different transfer learning approaches and you will understand what fine tuning means and why it is better to use transfer learning even when you have large datasets. We will also talk briefly on some of the popular datasets mentioned above.

When we say train the model from scratch, we mean that the model starts with zero knowledge of the world and the structure and the parameters of the model begin as random guesses. Practically speaking, this means that the weights of the model are randomly initialized and they need to go through a training process to be optimized.

The intuition behind transfer learning is that if a model is trained on a large and general enough dataset, this model will effectively serve as a generic representation of the visual world. We can then leverage the feature maps that it has learned without having to train on a large dataset by transferring what they learned to our model and use that as a base starting model for our own task.

"In transfer learning, we first train a base network on a base dataset and task, and then we repurpose the learned features, or transfer them to a second target network to be trained on a target dataset and task. This process will tend to work if the features are general, meaning suitable to both base and target tasks, instead of specific to the base task."

— ["How transferable are features in deep neural networks?" paper](#)

Let's jump directly to an example to get a better intuition on how to use transfer learning. Suppose you want to train a model that classifies dog and cat images and you have only two

classes in your problem: 1) dog and 2) cat. You need to collect hundreds of thousands of images for each class, label them, and train your network from scratch. Another option is to use transfer knowledge from another pretrained network.

First, we need to find a dataset that has similar features to our problem at hand. This involves spending sometime exploring different open-source datasets to find the closest one to our problem. For the sake of this example, let's use ImageNet since we are already familiar with it from the previous chapter and it has a lot of dog and cat images. So the pretrained network will be already familiar with dogs and cats features and will require minimum training. Later in this chapter we will explore other datasets. Next, we need to choose a network that has been trained on ImageNet and achieved good results. In the last chapter, we learned different state-of-the-art architectures like VGGNet, GoogLeNet, and ResNet. All would work fine. For this example, we will go with a VGG16 network that has been trained on ImageNet dataset.

To adapt the VGG16 network to our problem, we are going to download the VGG16 network with the pretrained weights and remove the classifier part then add our own classifier. Then retrain the new network. This is called **using a pretrained network as a feature extractor**. We will discuss the different types of transfer learning later in this chapter.

A **pretrained model** is a network that has been previously trained on a large dataset, typically on a large-scale image-classification task. We can either 1) directly use the pretrained model as it to run our predictions, or 2) use the pretrained feature extraction part of the network then add our own classifier. The classifier here could be one or more dense layers or even traditional machine learning algorithms like Support Vector Machines (SVM).

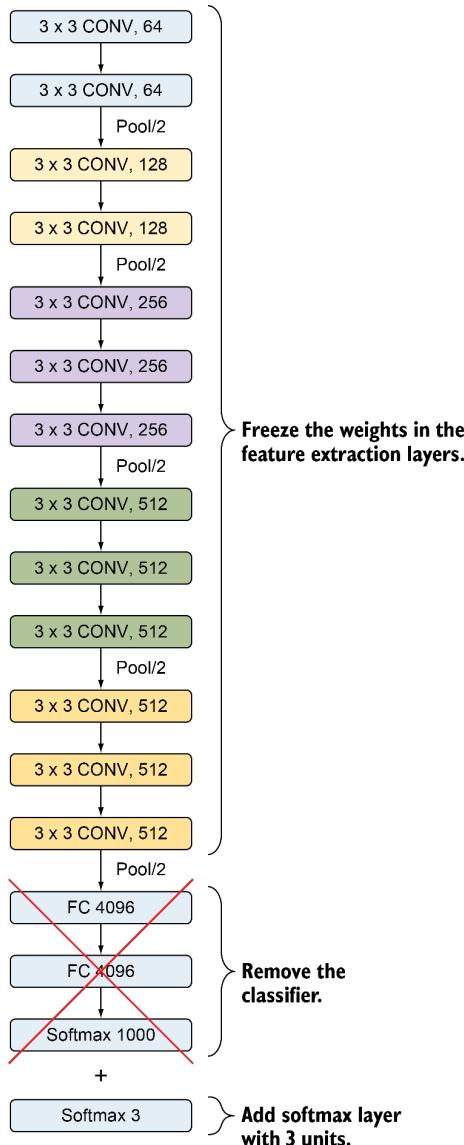


Figure 6.2: Example of applying transfer learning to VGG16 network. We freeze the feature extraction part of the network and remove the classifier part. Then add our new classifier softmax layer with 2 hidden units.

To get the full intuition of how to use transfer learning, let's implement this example step-by-step in keras:

1. Download the open-source code of the VGG16 network and its weights to create our base model and remove the classification layers from the VGG network (FC_4096 > FC_4096 > Softmax_1000). Luckily, Keras has a set of pretrained networks that are ready for us to download and use. The complete list of models can be [seen here](#).

```
# import the VGG16 model from Keras
from keras.applications.vgg16 import VGG16

# download the model's pretrained weights and save it in the variable base_model
# note that here we specified to keras to download the imagenet weights
# include_top is False to ignore the FC classifier part on top of the model
base_model = VGG16(weights = "imagenet", include_top=False, input_shape = (224,224, 3))
base_model.summary()
```

2. When you print a summary of the base model, you will notice that we downloaded the exact VGG16 architecture that we implemented in chapter 5. This is a fast approach to download popular networks that are supported by the deep learning library that you are using. Alternatively, you can build the network yourself like we did in chapter 5 and download the weights separately. I'll show you how in the project at the end of this chapter. But for now, let's look at the base_model summary that we just downloaded:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
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_pool (MaxPooling2D)	(None, 7, 7, 512)	0
<hr/>		
Total params: 14,714,688		
Trainable params: 14,714,688		
Non-trainable params: 0		

Notice that the downloaded architecture does not contain the classifier part (3 FC layers) at the top of the network because we set the `include_top` argument to False. More importantly, notice the number of trainable and non-trainable parameters in the summary. The downloaded network as it is makes all the network parameters trainable. As you can see above, our `base_model` has more than 14 million trainable parameters. Now, we want to freeze all the downloaded layers and add our own classifier. Let's do that next.

3. Freeze the feature extraction layers that have been trained on the ImageNet dataset. Freezing layers means freezing their trained weights to prevent them from being re-trained when we run our training.

```
# iterate through its layers and lock them to make them not trainable with this code
for layer in base_model.layers:
    layer.trainable = False

base_model.summary()
```

The model summary is omitted in this case for brevity, as it is similar to the model summary in the previous page. The difference is that all the weights have been frozen and the trainable parameters are now equal to zero and all the parameters of the frozen layers are non-trainable.

```
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

4. Add our own classification dense layer. In here, we will just add a softmax layer with 3 units because we have only 3 classes in our problem. See figure 6.3.

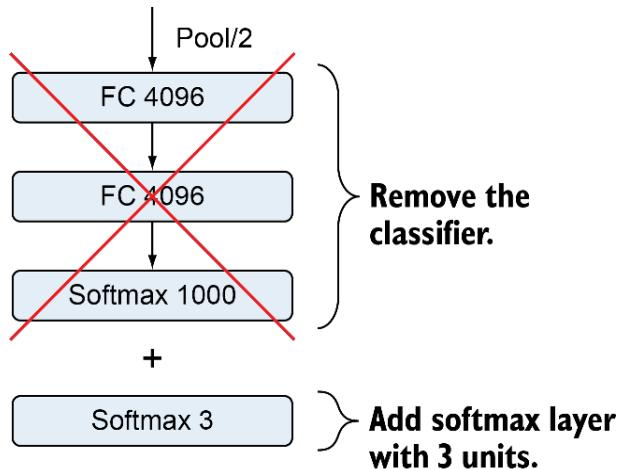


Figure 6.3: remove the classifier part of the network and add a softmax layer with 2 hidden nodes

```
# import keras modules
from keras.layers import Dense, Flatten
from keras.models import Model

# use "get_layer" method to save the last layer of the network
last_layer = base_model.get_layer('block5_pool')
# save the output of the last layer to be the input of the next layer
last_output = last_layer.output

# flatten the classifier input which is output of the last layer of VGG16 model
x = Flatten()(last_output)

# add our new softmax layer with 3 hidden units
x = Dense(2, activation='softmax', name='softmax')(x)
```

5. Build your new_model which takes the input of the base model as its input and the output of your last softmax layer as an output. The new model will be composed of all the feature extraction layers in VGGNet with the pretrained weights + our new, *untrained*, softmax layer. In other words, when we train the model, we are only going to train the softmax layer in this example to detect the specific features of our new problem; german shepherd, beagle, neither.

```
# instantiate a new_model using keras's Model class
new_model = Model(inputs=base_model.input, outputs=x)

# print the new_model summary
new_model.summary()

Layer (type)          Output Shape         Param #
=====
input_1 (InputLayer)   (None, 224, 224, 3)    0
block1_conv1 (Conv2D)  (None, 224, 224, 64)   1792
```

block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
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_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_layer (Flatten)	(None, 25088)	0
softmax (Dense)	(None, 2)	50178
<hr/>		
Total params: 14,789,955		
Trainable params: 50,178		
Non-trainable params: 14,714,688		

Training the new model will be a lot faster than training the network from scratch. To verify that, look at the number of trainable params in this model (~50k) compared to the number of non-trainable params in the network (~14M). These “non-trainable” parameters are already trained on a large dataset and we froze them to use the extracted features in our problem. With this new model, we don’t have to train the entire VGGNet from scratch because we only have to deal with the newly added softmax layer.

Additionally, we get a lot better performance with transfer learning because the new model has been trained on millions of images (ImageNet dataset + your small dataset) which allows the network to understand the finer details of the objects nuances which in turn makes it generalize better on new unseen images.

Note that in this example, we only explored the part where we build the model to show you how transfer learning is used. At the end of this chapter, I’ll walk you through two end-to-

end projects to see how to train the new network on your small data. But now, let's see how transfer learning works.

6.3 How transfer learning works

So far, we learned what transfer learning technique is and the main problems that it is solving. We also saw an example of how to use a pretrained network that was trained on ImageNet and transfer its learnings to our specific task. Now, let's see **why transfer learning works**, what is really being transferred from one problem to another, and how would a network that is trained on one dataset be able to perform well in a different dataset that might not even be related to it. This is what we are going to discuss in this section.

Let's answer a series of quick questions as a reminder from the previous chapters to get us to the core of what is happening in transfer learning:

1. What is really being learned by the network during training? The short answer is "Feature maps".
2. How are these features learned? During the backpropagation process, the weights are updated until we get to the "optimized weights" that minimize the error function.
3. What is the relationship between features and weights? A feature map is the result of passing the weights filter on the input image during the convolution process. See figure 6.4.

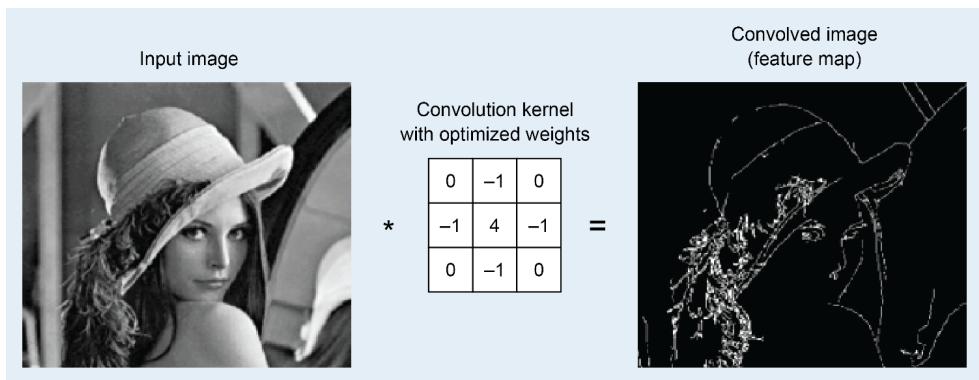


Figure 6.4

4. So, what is really being transferred from one network to another? To transfer features, we download the **optimized weights** of the pretrained network. These weights are then re-used as the starting point for the training process and retrained to adapt to the new problem.

Okay, let's dive in to the details to understand what we mean when we say "pretrained" network.

When we're training a convolutional neural network, the network extracts features from an image in the form of "feature maps". As we explained in chapters 2 and 3, feature maps are the outputs of each layer in a neural network after applying the weights filter. They are

representations of the features that exist in the training set. They are called feature maps because they map where a certain kind of feature is found in the image. Convolutional Neural Networks look for "features" such as straight lines, edges, or even objects. Whenever they spot these features they report them to the feature map. Each feature map is looking for something else. One feature map could be looking for straight lines, the other for curves. See figure 6.5

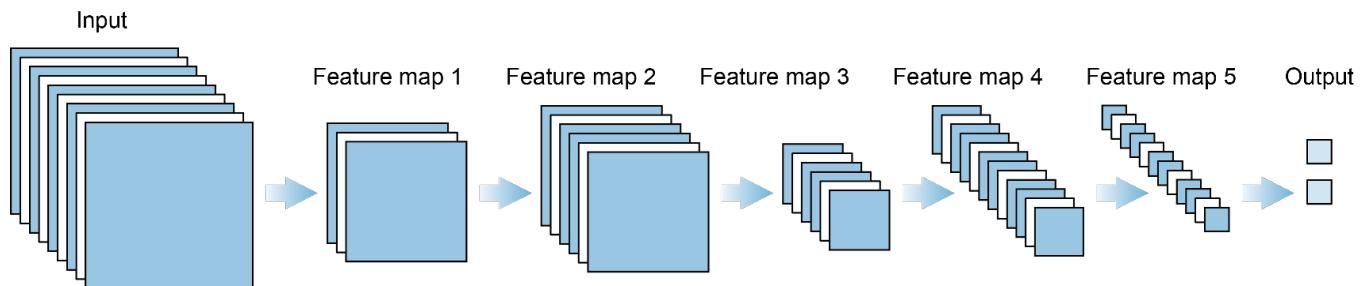


Figure 6.5

Now, recall that neural networks iteratively update their weights during the training cycle of feedforward and backpropagation. We call the network to have been trained when we go through a series of training iterations and hyperparameter tuning until the network yields satisfactory results. When training is complete, we output two main items: 1) the network architecture, and 2) the trained weights. So, when we say that we are going to use a pretrained network, we mean that we will download the network architecture + the weights.

During training, the model learns only the features that exist in this training dataset. But when we download large models (like Inception) that have been trained on huge amount of datasets (like ImageNet), all of these features that have already been extracted off these large datasets are now available for us to use. I find that really exciting because these **pretrained** models have spotted other features that weren't in our dataset and will help us build better convolutional networks.

In vision problems, there's so much stuff for neural networks to learn about the training dataset. There are low level features like edges, corners, round shapes, curvy shapes, blobs and then there are mid and higher level features like eyes, circles, squares, wheels. There are so many details in the images that convolutional neural networks can pick up on but if we have only a 1,000 images or even 25,000 images in our training dataset, this may not be enough data for the model to learn all those things. So by taking a pretrained network that someone else has trained we can basically download all this knowledge into our neural network to give it a huge and much faster start with even higher performance levels.

6.3.1 How do neural networks learn features?

The neural network learns the features in your dataset step-by-step in an increasing level of complexity layer after the other. These are called feature maps. The deeper you go through the network layers, the more image specific features are learned. See figure 6.6, where the first layer detects low level features such as edges and curves. The output of the first layer

becomes input to the second layer which produces higher level features, like semi-circles and squares. The next layer assembles the output of the previous layer to parts of familiar objects, and a subsequent layer detects the objects. As we go through more layers, the network yields an activation map that represents more and more complex features. The deeper you go into the network, the filters begin to be more responsive to a larger region of the pixel space. Higher level layers amplify aspects of the received inputs that are important for discrimination and suppress irrelevant variations.

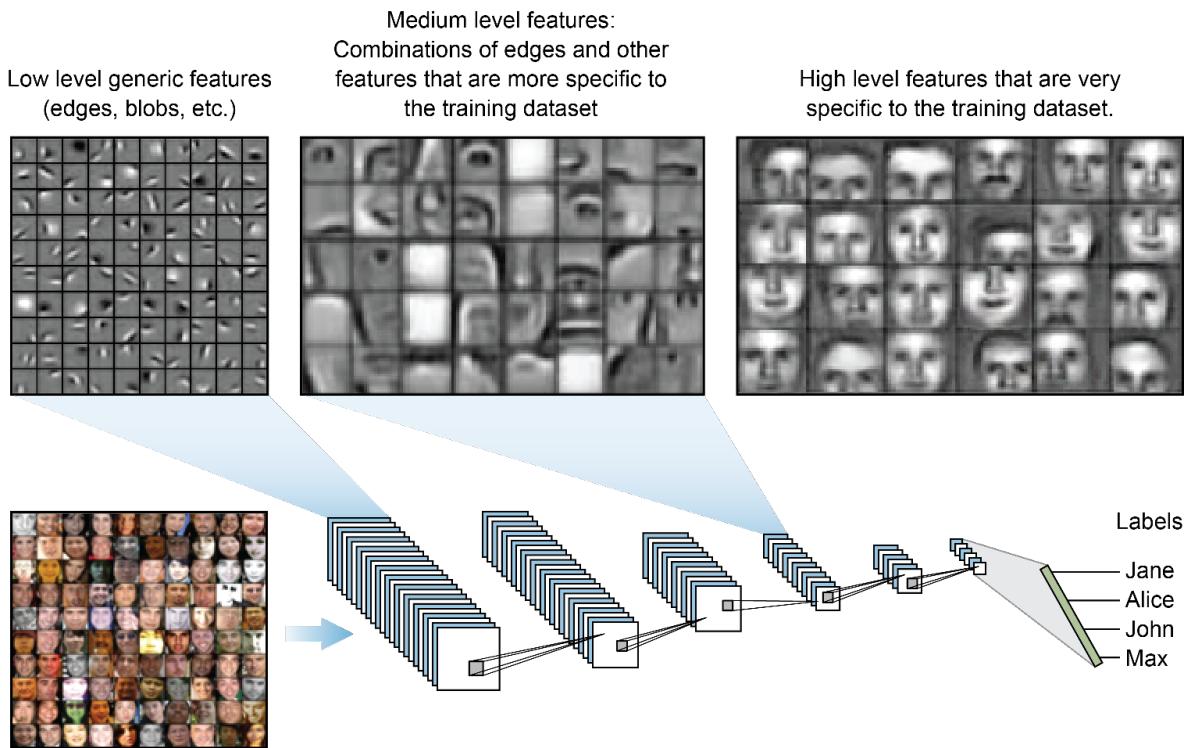


Figure 6.6: an example of how CNNs detects low-level generic features at the early layers of the network. The deeper you go through the network layers, the more image specific features are learned.

Consider the example in the figure above. Suppose that you are building a model that detects human faces. We notice that the networks learns low-level features like lines, edges, and blobs in the first layer. These low level features appear not to be specific to a particular dataset or task, they are general features that are applicable to many datasets and tasks. Then the mid-level layers will assemble those lines together to be able to recognize shapes, corners and circles. Notice that the extracted features started to get a little more specific to our task (human faces). So you will see that the mid-level features contain combinations of shapes that form objects in the human face like eyes and nose. The deeper we go through the network we notice that features will eventually transition from general to specific by the last

layer of the network forming high level features that are very specific to our tasks. So you will start seeing parts of human faces that distinguishes one person from another.

Now, let's take this example and compare the feature maps extracted from four models that are trained to classify faces, cars, elephants and chairs depicted in figure 6.7.

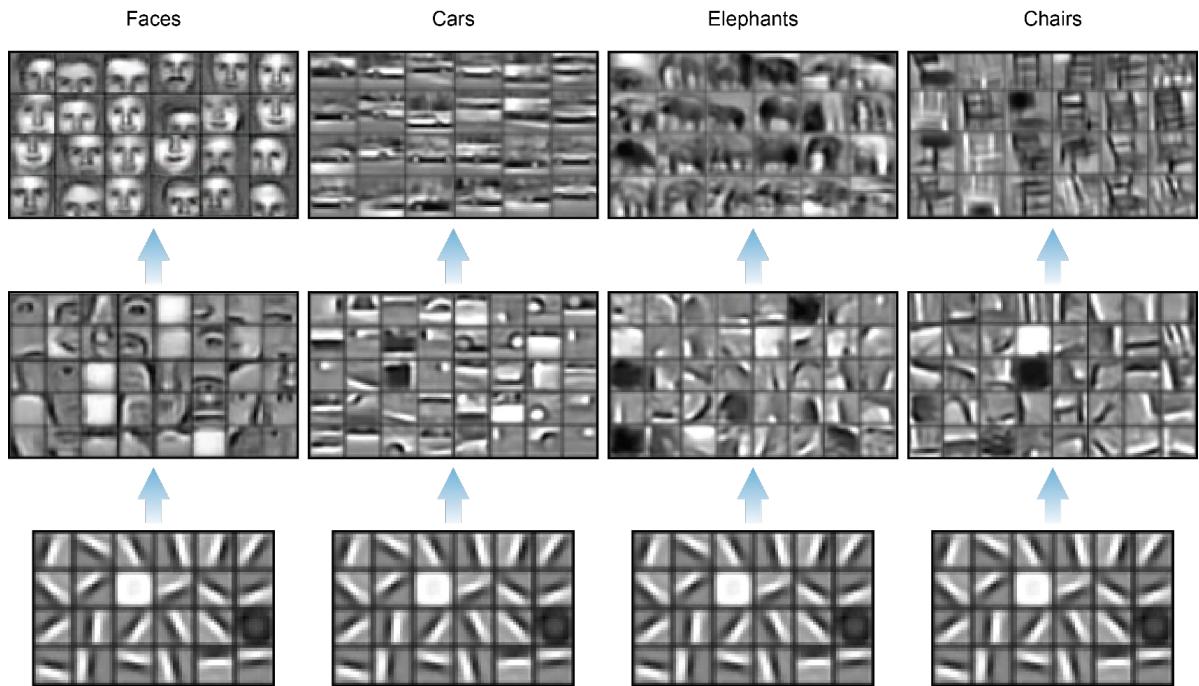


Figure 6.7: feature maps extracted from four models that are trained to classify faces, cars, elephants and chairs.

Notice that the earlier layer's features are very similar for all models. They represent the low level features like edges, lines, and blobs. This means that models that are trained on one task capture similar relations in the data type in the earlier layers of the network and can easily be reused for different problems in other domains. The deeper we go in to the network, we notice that the features start to be more specific until the network overfits its training data and it becomes harder to generalize to different tasks. The lower level features are almost always transferable from one task to another because they contain generic information like the structure and the nature of how images look. Transferring information like lines, dots, curves, and small parts of the objects is very valuable for the network to learn faster and with less data on the new task.

6.3.2 What about the transferability of features extracted at later layers in the network?

The transferability of features that are extracted at later layers depends on the similarity of the original and new datasets. The idea here is that all images must have shapes and edges so the early layers are usually transferable between different domains. We can only identify differences between objects when we start extracting higher level features like-say nose in a face or tires in a car. Only then can we say, okay; this is a person, because it has a nose and this is a car because it has tires. Based on the similarity of the source and target domains, we can decide whether to transfer only the low level features from the source domain or all the high level features or somewhere in between. This is motivated by the observation that the later layers of the network becomes progressively more specific to the details of the classes contained in the original dataset. This is what we are going to discuss in the next section.

Source Domain: the original dataset that the pretrained network is trained on.

Target Domain: the new dataset that we want to train the network on.

6.4 Transfer learning approaches

There are three major transfer learning approaches as follows:

1. Pretrained network as a classifier
2. Pretrained network as feature extractor
3. Fine tuning

Each approach can be effective and save significant time in developing and training a deep convolutional neural network model. It may not be clear as to which usage of the pre-trained model may yield the best results on your new computer vision task, therefore some experimentation may be required. In this section we will explain these three scenarios and give examples on how to implement each of them.

6.4.1 Pretrained network as a classifier

The pre-trained model is used directly to classify new images with no changes applied to it and no extra training. All we do here is download the network architecture and its pretrained weights. Then run the predictions directly on our new data. In this case, we are saying that the domain of our new problem is very similar to the one that the pretrained network was trained on and it is ready to just be "deployed". So no training is done here.

In the dog breed example, we could have used VGG16 network that was trained on ImageNet dataset directly to run predictions. Because ImageNet already contains a lot dog breeds images so the significant portion of the representational power of the pretrained network may be devoted to features that are specific to differentiating between dog breeds.

Using a pretrained network as a classifier doesn't really involve any layers freezing or extra model training. Instead, it is just taking a network that was trained on your similar problem and deploying it directly to your task.

Let's see how to use a pretrained network as a classifier. In this example, we will use a VGG16 network that was pretrained on the ImageNet dataset to classify the image of the German Shepherd dog in figure 6.8.



Figure 6.8: a sample image of a German Shepherd dog that we will be using to run predictions on.

The steps are as follows:

1. First, we import the necessary libraries

```
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions
from keras.applications.vgg16 import VGG16
```

2. Download the pretrained model of VGG16 and its ImageNet weights. We will set include_top to True because we want to use the entire network as a classifier.

```
# load the VGG16 model
model = VGG16(weights = "imagenet", include_top=True, input_shape = (224,224, 3))
```

3. Load and preprocess the input image

```
# load an image from file
image = load_img(path/to/image.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
```

```

image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)

4. Now our input image is ready for us to run the predictions

# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result with the highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%)' % (label[1], label[2]*100))

```

When you run the code above you will get this output:

```
>> German_shepherd (99.72%)
```

You can see that the model was already trained to predict the correct dog breed with a high confidence score (99.72%). This is because the ImageNet dataset has more than 20,000 labeled dog images and classified into 120 classes. Go to the book website to play with the code yourself with your own images: www.computervisionbook.com. Feel free to explore the classes available in ImageNet and run this experiment on your own images.

6.4.2 Pretrained network as a feature extractor

Similar to the dog and cat example that we implemented earlier in this chapter. We take a pretrained CNN on ImageNet, freeze its feature extraction part, remove the classifier part, and add our own new dense classifier layers.

In figure 6.9, we use a pretrained VGG16 network, freeze the weights in all the 13 convolutional layers, replace the old classifier with a new one to be trained from scratch:

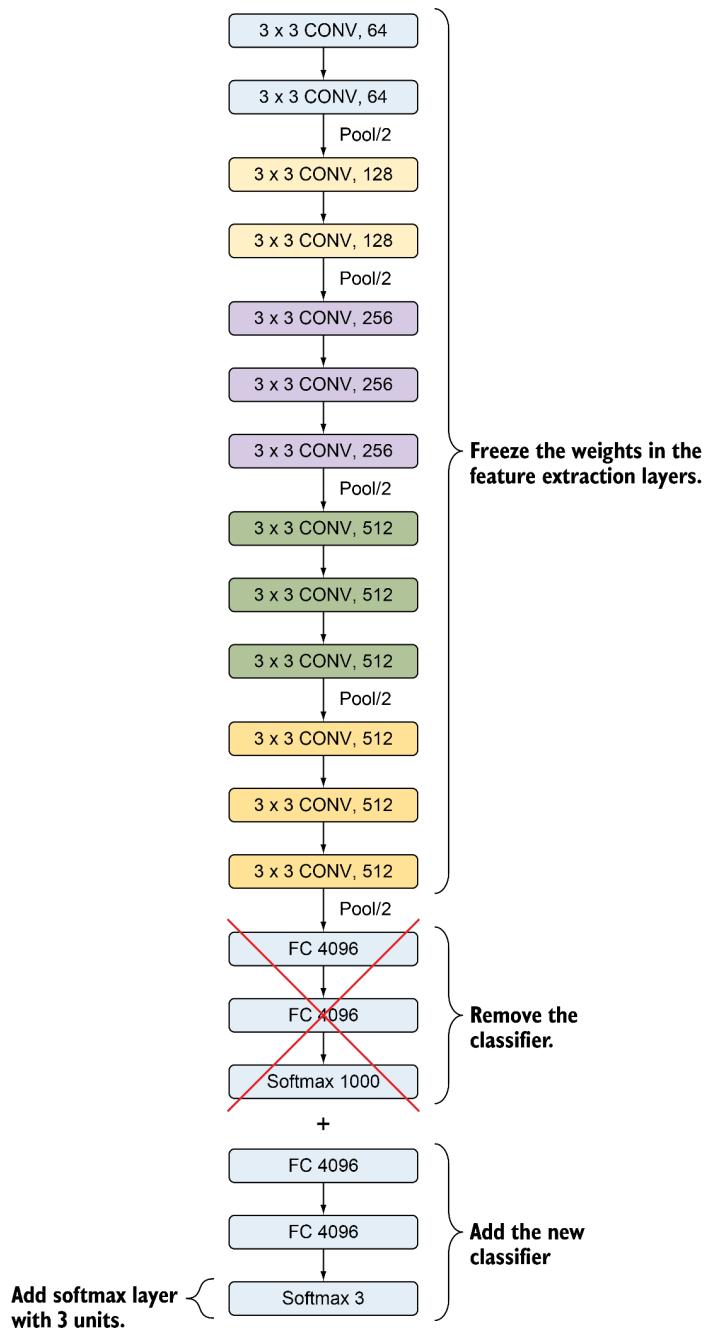


Figure 6.9: Load a pretrained VGG16 network. Remove the classifier. Add your own classifier.

We usually go with this scenario when our new task is similar to the original dataset that the pretrained network was trained on. Since the ImageNet dataset has a lot of dog and cat examples, the feature maps that the network has learned contain a lot of dog and cat features that are very applicable to our new task. This means that we can utilize the high level features that were extracted from the ImageNet dataset in to this new task.

To do that, we will freeze all the layers from the pretrained network and only train the classifier part that we just added on the new dataset. This approach is called “using a pretrained network as a feature extractor” because we froze the feature extractor part to transfer all the learned feature maps to our new problem. We only added a new classifier, which will be trained from scratch, on top of the pretrained model so that we can repurpose the feature maps learned previously for our dataset.

The reason we remove the classification part of the pretrained network is that it is often very specific to the original classification task, and subsequently specific to the set of classes on which the model was trained. For example, ImageNet has 1,000 classes. The classifier part has been trained to overfit the training data to classify them into 1,000 classes. But in our new problem, let’s say cats vs dogs, we only have 2 classes. So it is a lot more effective to train a new classifier from scratch to overfit these 2 classes.

6.4.3 Fine-tuning

So far, we’ve seen two basic approaches of using a pretrained network in transfer learning: 1) pretrained network as a classifier, and 2) pretrained network as a feature extractor. We usually use these two approaches when the target domain is somewhat similar to the source domain. Like using a network that was trained on ImageNet dataset to classify cats and dogs or trained on MS COCO dataset to classify traffic signs. But what if the target domain is different from the source domain? What if it is very different? Can we still use transfer learning? Yes. Transfer learning works great even when the domains are very different. We just need to extract the **correct** feature maps from the source domain and “fine tune” it to fit the target domain.

In figure 6.10, we show the different approaches of transferring knowledge from a pretrained network. If you are downloading the entire network with no changes and just running predictions, they you are using the network as a classifier. If you are freezing the convolutional layers only, then you are using the pretrained network as a feature extractor and transferring all its high level feature maps to your domain. Fine tuning is when you decide to freeze part of the feature extraction part of the network not all of it. But how do we decide where to fine tune? Should we freeze the layers until feature maps 1, or 2, or 3, or 4? This is what we are going to discuss in the next section. But first let’s get more intuition on how fine tuning works.

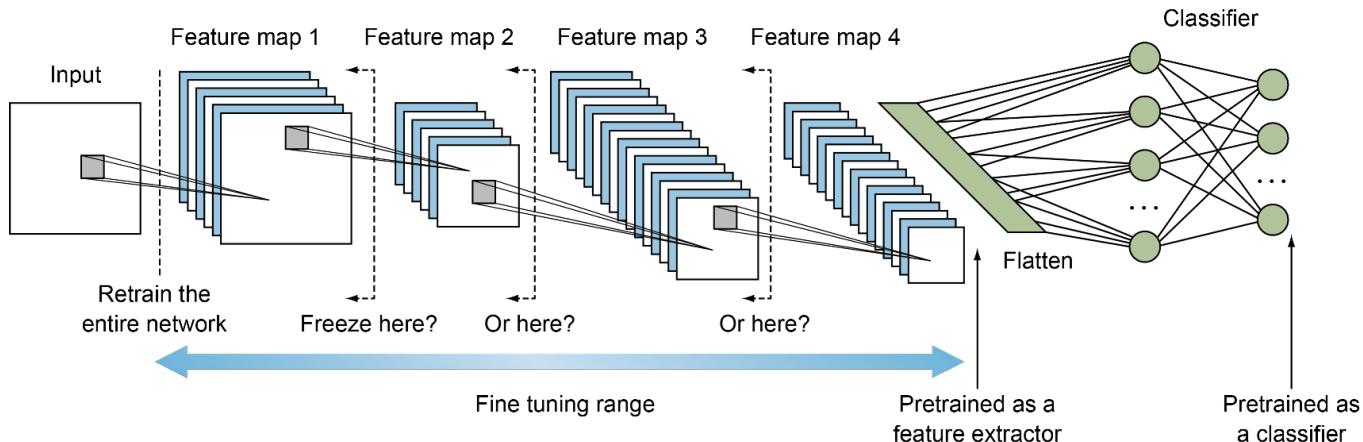


Figure 6.10: different approaches of transferring knowledge from a pretrained network.

As we discussed before, the feature maps that are extracted early in the network are generic and they progressively get more specific as we go deeper in the network. This means that feature maps 4 in the figure above are very specific to the source domain. Based on the similarity of the two domains, we can decide to freeze the network at the appropriate level of feature maps:

1. If the domains are similar, we might want to freeze all the network up to the last feature map level (i.e feature maps 4 in the above example)
2. If the domains are very different, we might decide to freeze the pretrained network after feature maps 1 and retrain all the remaining layers

Between these two options a range of fine tuning options that we can apply. We can decide to retrain the entire network or freeze the pretrained network at any level of feature maps 1, 2, 3, and 4 and retrain the remaining of the network. We typically decide the appropriate level of fine tuning by trial and error. But there are guidelines that we can follow to intuitively decide on the fine tuning level of the pretrained network. The decision is a function of two factors: 1) the amount of data that we have, and 2) the level of similarity between the source and target domains. We will explain these two factors in details and the four possible scenarios to choose the appropriate level of fine tuning in the next section.

WHAT IS FINE TUNING?

The formal definition of fine-tuning is freezing a few of the network layers that are used for feature extraction, and jointly training both the non-frozen layers and the newly added classifier layers of the pretrained model. It is called fine tuning because when we retrain the feature extraction layers, we "fine tune" the higher order feature representations to make them more relevant for the new task dataset.

In more practical terms, if we freeze features maps 1 and 2. This means that the new network will take feature maps 2 as its input and will start learning from this point to adapt the

features of the later layers to the new dataset. This saves the network the time that it would have spent to learn feature maps 1 and 2.

WHY FINE TUNING IS BETTER THAN TRAINING FROM SCRATCH?

When we train a network from scratch, we usually randomly initialize the weights and apply gradient descent optimizer to find the best set of weights that optimizes our error function (chapter 2). Since these weights start with random values, there is no guarantee that they will start with values that are close to the desired optimal values. And if the initialized value is far away from the optimal value, the optimizer will take a long time to converge. This is when fine tuning can be very useful. The pretrained network weights have been already optimized to learn from its dataset. This means that when we use this network in our problem, we start with the weights values that it ended with. This makes the network much faster to converge than having to randomly initialize the weights. This is what the term "fine-tuning" refers to. We are basically fine-tuning the already-optimized weights to fit our new problem instead of training the entire network from scratch with random weights. So, even if we decide to retrain the entire pretrained network, starting with the trained weights will converge faster than having to train the network from scratch with randomly initialized weights.

USE A SMALLER LEARNING RATE WHEN FINE TUNING

it's common to use a smaller learning rate for ConvNet weights that are being fine-tuned, in comparison to the (randomly-initialized) weights for the new linear classifier that computes the class scores of your new dataset. This is because we expect that the ConvNet weights are relatively good, so we don't wish to distort them too quickly and too much (especially while the new classifier above them is being trained from random initialization).

6.5 Choose the appropriate level of transfer learning

Recall that early convolutional layers extract generic features then they get more specific to the training data the deeper we go through the network. With that said, a level of detail for feature extraction from an existing pre-trained model can be chosen. For example, if a new task is quite different from the source domain of the pretrained network (e.g. different to ImageNet), then perhaps the output of the pretrained model after the first few layers would be appropriate. If a new task is quite similar to the source domain, then perhaps the output from layers much deeper in the model can be used, or even the output of the fully connected layer prior to the softmax layer can be used.

Choosing the appropriate level of using transfer learning is a function of two important factors:

1. **The size of the target dataset (small or large)** - When we have a small dataset, there is probably not much information that the network would learn from training more layers, so it will tend to overfit the new data. In this case we probably want to do less fine tuning and rely more on the source dataset.
2. **Domain similarity of the source and target datasets** - how similar is your new problem to the domain of the original dataset. For example, if your problem is to classify cars and boats, ImageNet could be a good option because it contains a lot of images of similar features. On the other hand, if your problem is to classify chest

cancer on x-ray images, this is a completely different domain that will likely require a lot of fine-tuning.

These two factors develop the four major scenarios below:

1. Target dataset is small and similar to the source dataset
2. Target dataset is large and similar to the source dataset
3. Target dataset is small and very different from the source dataset
4. Target dataset is large and very different from the source dataset

Let's discuss these scenarios one by one to learn the common rules of thumb for navigating our options.

6.5.1 Scenario #1: target dataset is small and similar to source dataset

Since the original dataset is similar to our new dataset, we can expect that the higher-level features in the pretrained ConvNet to be relevant to our dataset as well. Then it might be best to freeze the feature extraction part of the network and only retrain the classifier.

Another reason it might not be a good idea to fine tune the network is that our new dataset is small. If we fine tune the feature extraction layers on a small dataset, that will force the network to overfit to our data. This is not good because by definition, a small dataset doesn't have enough information to cover all possible features of its objects which makes it fail to generalize to new unseen data. So in this case, the more fine tuning we do, the more the network is prone to overfit the new data.

For example, if all of the images in our new dataset contains dogs in a specific weather environment, snow for example. If we fine tune on this dataset, this would force the new network to pick up features like snow and white background as dog-specific features and makes it fail to classify dogs in other weather conditions. So the general rule of thumb is: if you have a small amount of data, be careful of overfitting when you fine tune your pretrained network.

6.5.2 Scenario #2: target dataset is large and similar to the source dataset

Since both domains are similar, we can freeze the feature extraction part and retrain the classifier similar to what we did in scenario #1. But, since we have more data in the new domain, we can get a performance boost from fine tuning through all or part of the pretrained network with more confidence that we won't overfit. Fine tuning through the entire network is not really needed because the higher-level features are related (since the datasets are similar). So a good start is to freeze approximately 60% - 80% of the pretrained network and retrain the rest on the new data.

6.5.3 Scenario #3: target dataset is small and different from the source dataset

Since the dataset is different, it might not be best to freeze the higher-level features of the pretrained network because they contain more dataset-specific features. Instead, it would work better to retrain layers from somewhere earlier in the network. Or even don't freeze any layers fine tune the entire network. However, since you have a small dataset, fine tuning the entire network on your small dataset might not be a good idea because it makes it prone to

overfitting. A mid-way solution would work better in this case. So a good start is to freeze approximately the first third or half of the pretrained network. Afterall, the early layers contain very generic feature maps that would be useful for your dataset even if it is very different.

6.5.4 Scenario #4: target dataset is large and different from the source dataset

Since the new dataset is large, you might be tempted just train the entire network from scratch and not use transfer learning at all. However, in practice it is often still very beneficial to initialize with weights from a pretrained model as we discussed earlier. It makes the model converges faster. In this case, we have a large dataset that provides us confidence to fine-tune through the entire network without having to worry about overfitting.

6.5.5 Recap of the transfer learning scenarios

We explored the two main factors that help us define which transfer learning approach to use: 1) size of our data, and 2) similarity between the source and target datasets. These two factors develop the four major scenarios defined in the table below:

Table 6.1

Scenario	Size of the target data	Domain difference between the original and new datasets	Approach
1.	Small	Similar	Pretrained network as a feature extractor
2.	Large	Similar	Fine tune through the full network
3.	Small	Very different	Fine tune from activations earlier in the network
4.	Large	Very different	Fine-tune through the entire network

Figure 6.11 summarizes the guidelines for the appropriate fine tuning level to use in each of the four scenarios:

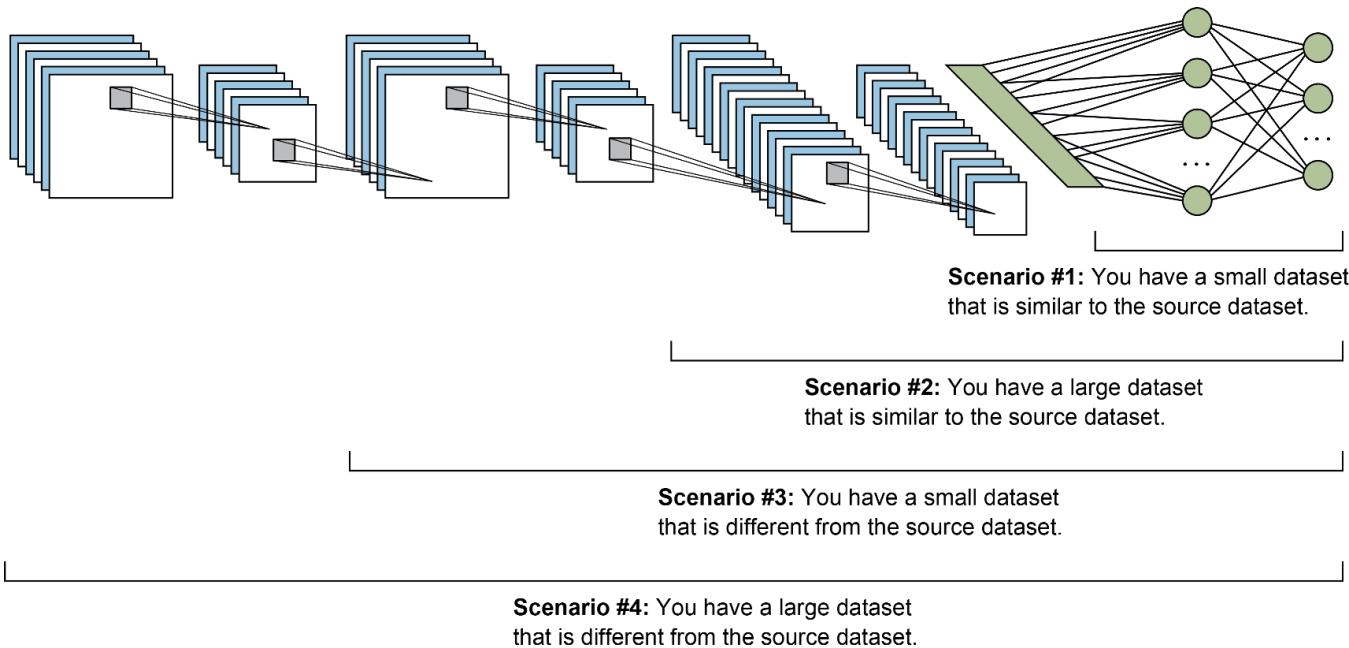


Figure 6.11: guidelines for the appropriate fine tuning level to use in each of the four scenarios.

6.6 Open-source datasets

In the previous chapters of this book, we worked with some open source datasets like MNIST, CIFAR, and ImageNet. These are not the only image datasets that we can download and use for experimentation and train networks in our projects. In this section, we will review some of the popular open source datasets to help guide you in your search to find the most suitable dataset for your problem. Keep in mind that the ones listed in this chapter are the most popular datasets at the time of writing of this chapter that are used in the computer vision research communities and we do not intend to provide a comprehensive list of all the open-source datasets out there. There is a very large number of image datasets that are growing everyday. Before starting your project, I encourage you to do your own research to explore the available datasets.

The computer vision research community has been pretty good at posting lots of datasets on the internet. So, if you hear of names like ImageNet, MS COCO, or Open Images, and many others, these are the names of datasets that people have posted online and a lot of computer researchers have used as benchmarks to train their algorithms and get state-of-the-art results.

Last chapter we explained the [ImageNet](#) dataset that is used in the "ImageNet Large Scale Visual Recognition Challenge ([ILSVRC](#))". There are other popular datasets like MS COCO, Open Images, and many more that are ready for you to explore, download and experiment with. When applying transfer learning, you might need to do a little research to use a dataset that is close enough to your domain.

6.6.1 MNIST

[MNIST](#) stands for "Modified National Institute of Standards and Technology". It contains labeled handwritten images of digits from 0 to 9. The goal of this dataset is to classify handwritten digits. MNIST was popularly used by the research community to benchmark their classification algorithms. In fact, it is considered as the "hello, world!" of image datasets. Nowadays, MNIST dataset is pretty simple and a simple CNN can achieve more than 99% accuracy so it is not considered a benchmark for CNN performance anymore. We implemented a CNN classification project using MNIST dataset in chapter 3. Feel free to go back and review the project.

MNIST consists of 60,000 training images and 10,000 test images. All are grayscale images (i.e. 1 channel). Each image has 28 pixel height and 28 pixel width. Figure 6.12 shows a sample image of the MNIST dataset.



Figure 6.12: a sample image of the MNIST dataset.

6.6.2 Fashion-MNIST

The data is stored in the same format as the original MNIST dataset. It was made with the intention to replace the MNIST dataset because it became too simple for modern convolutional networks. Instead of handwritten digits, it contains 60,000 training images and 10,000 test images of 10 fashion clothing classes. The 10 supported classes are: t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. Visit the [Github repo](#) for the dataset to explore and download. Below is a sample of the represented classes in this dataset. See figure 6.13.



Figure 6.13: a sample image of the Fashion-MNIST dataset.

6.6.3 CIFAR-10

CIFAR is considered as another benchmark dataset for image classification in the computer vision and machine learning literature. CIFAR images are more complex than MNIST in the sense that MNIST are all grayscale images and their objects are perfectly centered in the middle of the image while CIFAR's images are all colored (3 channels) with dramatic variance of how the objects appear in the image. The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images.

Here are the classes in the dataset taken from the dataset [website](#) in figure 6.14.:

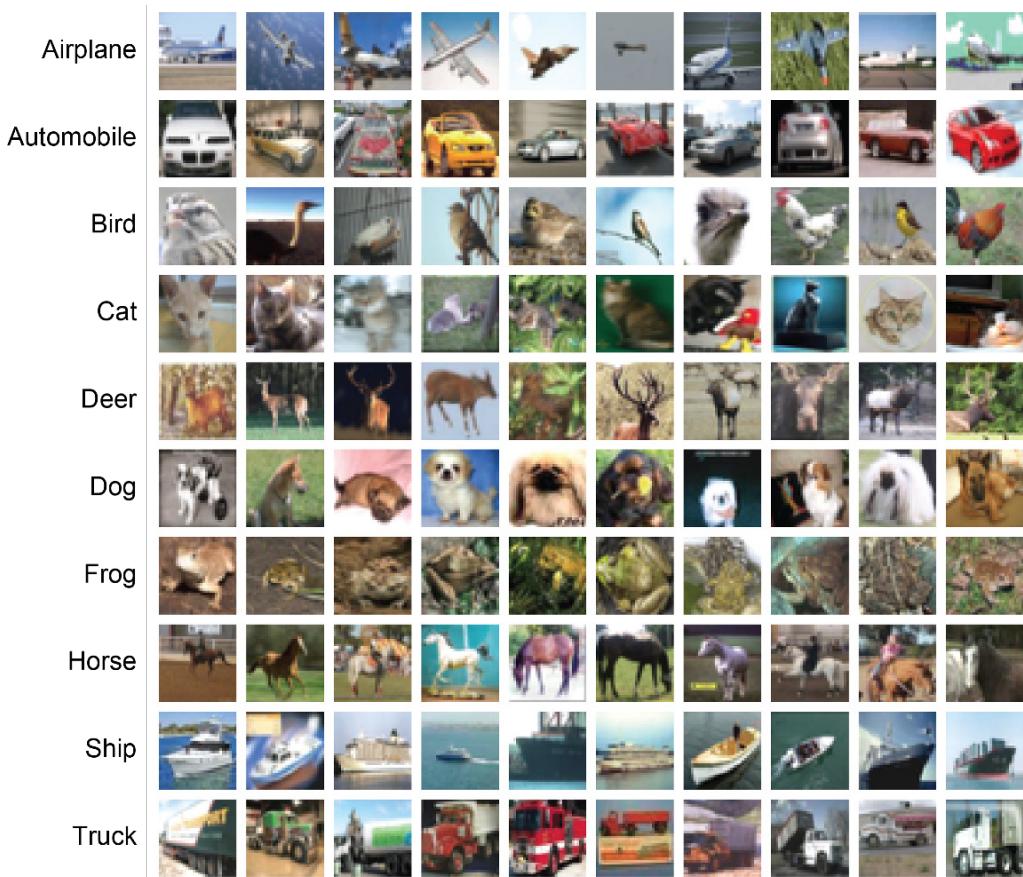


Figure 6.14: a sample image of the CIFAR-10 dataset.

CIFAR-100 is the bigger brother of CIFAR-10. It contains 100 classes with 600 images each. These 100 classes are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

6.6.4 ImageNet

We've discussed the ImageNet dataset several times throughout the previous chapters and used it extensively in chapter 5 and this chapter. But for completeness of this list, we are listing it here as well. At the time of writing, ImageNet is considered the current benchmark that is widely used by computer vision researchers to evaluate their classification algorithms.

ImageNet is a large visual database designed for use in visual object recognition software research. It is aimed at labeling and categorizing images into almost 22,000 categories based

on a defined set of words and phrases. The images were collected from the web and labeled by human labelers using Amazon's Mechanical Turk crowdsourcing tool. At the time of this writing, there are over 14 million images in the ImageNet project. To organize such a massive amount of data, the creators of ImageNet followed the WordNet hierarchy where each meaningful word/phrase inside WordNet is called a "synonym set" or "synset" for short. Within the ImageNet project, images are organized according to these synsets, with the goal being to have 1,000+ images per synset.

Figure 6.15 shows a collage of ImageNet examples put together by Stanford University.



Figure 6.15: a collage of ImageNet examples put together by Stanford University.

The computer vision community usually refer to the ILSVRC challenge when talking about ImageNet. ILSVRC is an ImageNet project that runs an annual software contest called the "ImageNet Large Scale Visual Recognition Challenge (ILSVRC)", where software programs compete to correctly classify and detect objects and scenes. We will be using the ILSVRC challenge as a benchmark to compare the different networks' performance.

6.6.5 Microsoft's COCO

[MS COCO](#) is short for "Microsoft Common Objects in Context" that was created by Microsoft. It is an open-source database that aims to enable future research for object detection, instance segmentation, image captioning, and person keypoints localization. It contains 328K images. More than 200K of them are labeled, 2.5 million object instances, and 80 object categories

that would be easily recognizable by a 4 year old. Read the [original research paper](#) that was provided by the creators of the dataset. It describes the motivation and content of this dataset very well.

Figure 6.16 shows a sample of the dataset provided on the [cocodataset.org](#) website:

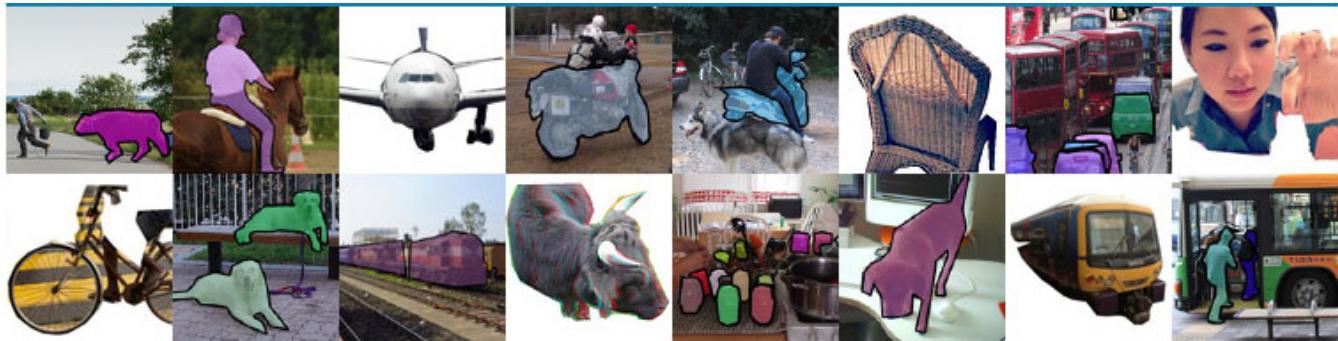


Figure 6.16: a sample of the MS COCO dataset.

6.6.6 Google's Open Images

[Open Images](#) is an open-source image database that was created by Google. It contains more than 9 million images as of the writing of this chapter. What makes it stand out is that these images are mostly of complex scenes that span thousands of classes of objects. Additionally, more than 2 million of these images are hand-annotated with bounding boxes making Open Images by far the largest existing dataset with object location annotations. In this subset of images, there are ~15.4 million bounding boxes of 600 classes of object. Similar to ImageNet and ILSVRC, Open Images has a challenge called "[Open Images Challenge](#)".

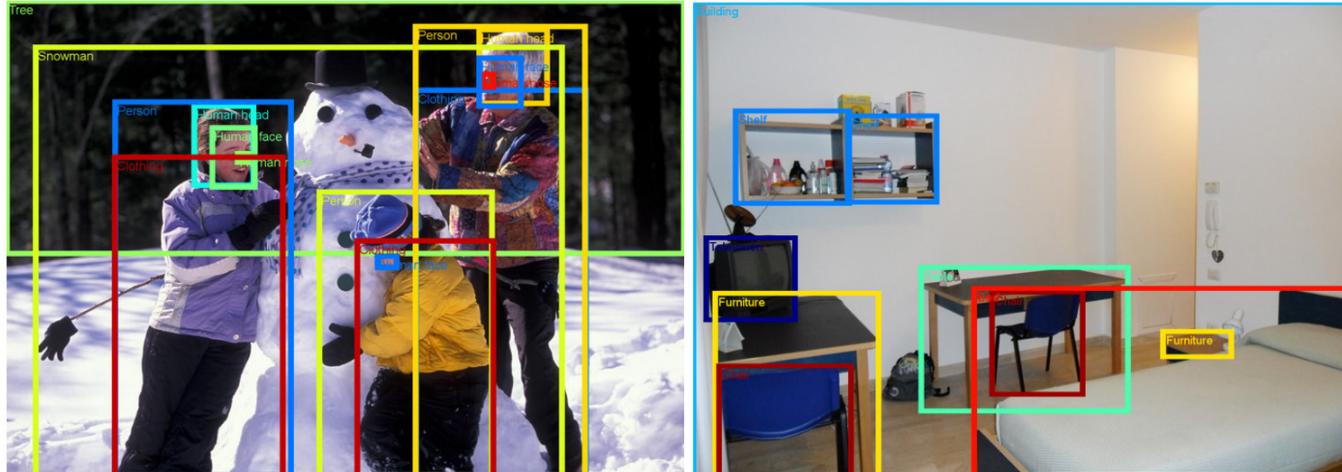


Figure 6.17: Annotated images from the Open Images dataset taken from [Google AI Blog](#).

6.6.7 Kaggle

In addition to the datasets listed in this section, Kaggle is another great source for datasets. Kaggle is a website that hosts machine learning and deep learning challenges where people from all around the world can participate and submit their algorithms for evaluations.

Finally, you are strongly encouraged to explore these datasets and search for many other open-source datasets that come up every day to have a better understanding of the classes and use cases that they support. We will mostly use ImageNet in this chapter and throughout the book we will be using MS COCO, especially in the object detection chapter in this book.

6.7 Chapter summary and takeaways

- Transfer learning usually is the go-to approach when starting your classification and object detection project especially when you don't have a lot of training data.
- For example, if you have a dataset that contains a million images, that's a lot of data for the network to learn a lot of useful features. Then you have another vision task and you only have a few hundred training images. That is very little data for your network to learn from in your new task. So, a lot of features that the network learned from the large dataset can be transferred to your new task where you don't have enough data.
- Transfer learning migrates the knowledge learned from the source data set to the target data set to save training time and computational cost.
- The neural network learns the features in your dataset step-by-step in an increasing level of complexity layer after the other. The deeper you go through the network layers, the more image-specific features are learned.
- Early layers in the network learn low-level features like lines, blobs, and edges. The output of the first layer becomes input to the second layer which produces higher-level features, like semi-circles and squares. The next layer assembles the output of the

- previous layer to parts of familiar objects, and a subsequent layer detects the objects.
- The three main transfer learning approaches are: 1) using a pretrained network as a classifier, 2) using a pretrained network as a feature extractor, and 3) fine-tuning
 - Using a pretrained network as a classifier means using the network directly to classify new images without freezing layers or apply model training.
 - Using a pretrained network as a feature extractor means freezing the classifier part of the network and retrain the new classifier.
 - Fine tuning means freezing a few of the network layers that are used for feature extraction, and jointly training both the non-frozen layers and the newly added classifier layers of the pretrained model.
 - The transferability of features from one network to another is a function of two factors: 1) the size of the target data, and 2) the domain similarity between the source and target data.
 - Generally, fine tuning parameters use a smaller learning rate, while training the output layer from scratch can use a larger learning rate.

6.8 Project 1: A pretrained network as a feature extractor

In this project, we are going to use a very small amount of data to train a classifier that detects dogs and cats. This is a pretty simple project but the goal of this exercise is to see how to implement transfer learning when you have a very small amount of data where the target domain is similar to the source domain (Scenario #1). As explained in this chapter, in this case we are going to use the pretrained convolutional network as a feature extractor. This means that we are going to freeze the feature extractor part of the network and add our own classifier then retrain the network on our new small dataset.

One other important takeaway from this project is to learn how to preprocess custom data and make it ready to train your neural network. In previous projects, we used CIFAR and MNIST datasets which are already preprocessed for us by Keras and all we had to do is to download it from Keras library and directly use them to train the network. In this project, I'm going to show a tutorial on how to structure your data repository and use Keras library to get your data ready.

Visit the book website at www.computervisionbook.com to download the code notebook and the dataset used for this project. Since we are using transfer learning, the training does not require high computation power so you can run this notebook on your personal computer and you don't need a GPU.

For this implementation, we'll be using the VGG16. Although it didn't record the lowest error in the ILSVRC, I found it worked well for the task and was quicker to train than other models. I got an accuracy of about 96% but you can feel free to use GoogLeNet or ResNet for experimentation and compare results.

The process to use a pretrained model as a feature extractor is well-established:

1. Import the necessary libraries.
2. Preprocess the data to make it ready for the neural network.
3. Load in pretrained weights from the VGG16 network trained on a large dataset.

4. Freeze all the weights in the convolutional layers (feature extraction part). Remember - the layers to freeze are adjusted depending on the similarity of new task to original dataset. In our case, we observed that ImageNet has a lot of dogs and cats images so the network has already been trained to extract the detailed features of our target object.
5. Replace the fully-connected layers of the network with a custom classifier. You can add as many FC layers as you see fit and each have as many hidden units as you want. For simple problem like this, we are just going to add one hidden layer with 64 units. You can observe the results and tune up if the model is underfitting or down if the model is overfitting. For the softmax layer, the number of units must be set equal to the number of classes (2 units in our case).
6. Compile the network and run the training process on the new data of cats and dogs to optimize the model for the smaller dataset.
7. Evaluate the model.

Now let's go through these steps one-by-one and implement this project.

1. IMPORT THE NECESSARY LIBRARIES

```
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing import image
from keras.applications import imagenet_utils
from keras.applications import vgg16
from keras.applications import mobilenet
from keras.optimizers import Adam, SGD
from keras.metrics import categorical_crossentropy
from keras.layers import Dense, Flatten, Dropout, BatchNormalization
from keras.models import Model
from sklearn.metrics import confusion_matrix
import itertools
import matplotlib.pyplot as plt
%matplotlib inline
```

2. PREPROCESS THE DATA TO MAKE IT READY FOR THE NEURAL NETWORK

Keras has this ImageDataGenerator class which allows us to perform image augmentation on the fly in a very easy way. You can read about that in Keras's official [documentation](#). In this example, we are going to use the ImageDataGenerator class to generate our image tensors but we are not going to implement image augmentation for simplicity.

The ImageDataGenerator class has a method called `flow_from_directory()` that is used to read the images from folders containing images. This method expects your data directory to be structured as follows:

- Train
 - class_a
 - class_a_1.jpg
 - class_a_2.jpg
 - class_b

- Valid
 - class_a
 - class_b
- Test
 - test_folder
- test_1.jpg
- test_2.jpg

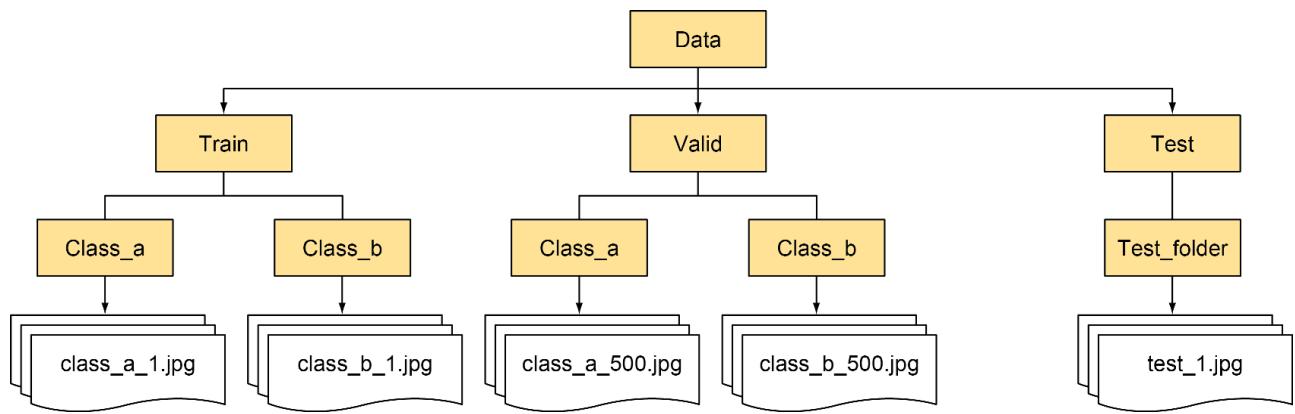


Figure 6.18: a recommended directory structure to your dataset to use `.flow_from_directory()` method from Keras.

I have the data structured for you in the Github repo to be ready for you to use `flow_from_directory()` method. Now, let's load the data into `train_path`, `valid_path`, and `test_path` variables then generate the train, valid, and test batches:

```

train_path = 'data/train'
valid_path = 'data/valid'
test_path = 'data/test'

# ImageDataGenerator generates batches of tensor image data with real-time data augmentation.
# The data will be looped over (in batches).
# in this example, we won't be doing any image augmentation
train_batches = ImageDataGenerator().flow_from_directory(train_path,
                                                       target_size=(224,224),
                                                       batch_size=10)

valid_batches = ImageDataGenerator().flow_from_directory(valid_path,
                                                       target_size=(224,224),
                                                       batch_size=30)

test_batches = ImageDataGenerator().flow_from_directory(test_path,
                                                       target_size=(224,224),
                                                       batch_size=50,
                                                       shuffle=False)

```

3. LOAD IN PRETRAINED WEIGHTS FROM THE VGG16 NETWORK TRAINED ON A LARGE DATASET

Similar to the examples in this chapter, we are going to download the VGG16 network from Keras and download it's weights after being pretrained on ImageNet dataset. Remember that we want to remove the classifier part from this network so we will set the parameter include_top=False.

```
base_model = vgg16.VGG16(weights = "imagenet", include_top=False, input_shape = (224,224, 3))
```

4. FREEZE ALL THE WEIGHTS IN THE CONVOLUTIONAL LAYERS (FEATURE EXTRACTION PART)

We will freeze the convolutional layers from the base_model created from the previous step and use that as a feature extractor, then add a classifier on top of it in the next step.

```
# iterate through its layers and lock them to make them not trainable with this code
for layer in base_model.layers:
    layer.trainable = False
```

5. ADD THE NEW CLASSIFIER AND BUILD THE NEW MODEL

Now let's add a few layers on top of the base model. In this example, we will add one FC layer with 64 hidden units and a softmax with 2 hidden units. We will also add batch norm and dropout layers to avoid overfitting.

```
# use "get_layer" method to save the last layer of the network
# then save the output of the last layer to be the input of the next layer
last_layer = base_model.get_layer('block5_pool')
last_output = last_layer.output

# flatten the classifier input which is output of the last layer of VGG16 model
x = Flatten()(last_output)

# add 1 FC layers that has 64 units, batchnorm, dropout, and softmax layers
x = Dense(64, activation='relu', name='FC_2')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(2, activation='softmax', name='softmax')(x)

# instantiate a new_model using keras's Model class
new_model = Model(inputs=base_model.input, outputs=x)
new_model.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584

block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
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_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_1 (Flatten)	(None, 25088)	0
FC_2 (Dense)	(None, 64)	1605696
batch_normalization_1 (Batch)	(None, 64)	256
dropout_1 (Dropout)	(None, 64)	0
softmax (Dense)	(None, 2)	130
<hr/>		
Total params: 16,320,770		
Trainable params: 1,605,954		
Non-trainable params: 14,714,816		

6. COMPILE THE MODEL AND RUN THE TRAINING PROCESS

```
new_model.compile(Adam(lr=0.0001), loss='categorical_crossentropy', metrics=['accuracy'])

new_model.fit_generator(train_batches, steps_per_epoch=4,
                        validation_data=valid_batches, validation_steps=2, epochs=20, verbose=2)
```

When you run the code snippet above, the training verbose is printed after each epoch as follows:

```
Epoch 1/20
- 28s - loss: 1.0070 - acc: 0.6083 - val_loss: 0.5944 - val_acc: 0.6833
Epoch 2/20
- 25s - loss: 0.4728 - acc: 0.7754 - val_loss: 0.3313 - val_acc: 0.8605
Epoch 3/20
- 30s - loss: 0.1177 - acc: 0.9750 - val_loss: 0.2449 - val_acc: 0.8167
Epoch 4/20
```

```

- 25s - loss: 0.1640 - acc: 0.9444 - val_loss: 0.3354 - val_acc: 0.8372
Epoch 5/20
- 29s - loss: 0.0545 - acc: 1.0000 - val_loss: 0.2392 - val_acc: 0.8333
Epoch 6/20
- 25s - loss: 0.0941 - acc: 0.9505 - val_loss: 0.2019 - val_acc: 0.9070
Epoch 7/20
- 28s - loss: 0.0269 - acc: 1.0000 - val_loss: 0.1707 - val_acc: 0.9000
Epoch 8/20
- 26s - loss: 0.0349 - acc: 0.9917 - val_loss: 0.2489 - val_acc: 0.8140
Epoch 9/20
- 28s - loss: 0.0435 - acc: 0.9891 - val_loss: 0.1634 - val_acc: 0.9000
Epoch 10/20
- 26s - loss: 0.0349 - acc: 0.9833 - val_loss: 0.2375 - val_acc: 0.8140
Epoch 11/20
- 28s - loss: 0.0288 - acc: 1.0000 - val_loss: 0.1859 - val_acc: 0.9000
Epoch 12/20
- 29s - loss: 0.0234 - acc: 0.9917 - val_loss: 0.1879 - val_acc: 0.8372
Epoch 13/20
- 32s - loss: 0.0241 - acc: 1.0000 - val_loss: 0.2513 - val_acc: 0.8500
Epoch 14/20
- 29s - loss: 0.0120 - acc: 1.0000 - val_loss: 0.0900 - val_acc: 0.9302
Epoch 15/20
- 36s - loss: 0.0189 - acc: 1.0000 - val_loss: 0.1888 - val_acc: 0.9000
Epoch 16/20
- 30s - loss: 0.0142 - acc: 1.0000 - val_loss: 0.1672 - val_acc: 0.8605
Epoch 17/20
- 29s - loss: 0.0160 - acc: 0.9917 - val_loss: 0.1752 - val_acc: 0.8667
Epoch 18/20
- 25s - loss: 0.0126 - acc: 1.0000 - val_loss: 0.1823 - val_acc: 0.9070
Epoch 19/20
- 29s - loss: 0.0165 - acc: 1.0000 - val_loss: 0.1789 - val_acc: 0.8833
Epoch 20/20
- 25s - loss: 0.0112 - acc: 1.0000 - val_loss: 0.1743 - val_acc: 0.8837

```

Notice that the model was trained very quickly using a regular CPU computing power. Each epoch took approximately 25 to 29 seconds which means that it took the model less than 10 minutes to train for 20 epochs.

7. EVALUATE THE MODEL

Now, let's use Keras's `evaluate()` method to calculate the model accuracy.

```

print('\nTesting loss: {:.4f}\nTesting accuracy:
 {:.4f}'.format(*new_model.evaluate(test_tensors, test_targets)))

Testing loss: 0.1042
Testing accuracy: 0.9579

```

The model has achieved an accuracy of 95.79% in less than 10 minutes of training. This is very good given the very small dataset we have.

6.9 Project 2: Fine tuning

In this project, we are going to explore scenario #3 that was discussed earlier in this chapter. Where the target dataset is small and it very different from the source dataset. The goal of this project is to build a **Sign Language Classifier** that classifies between 10 classes. The

classes are the digits from 0 to 9. Figure 6.19 below shows a sample of the sign language dataset that we have.

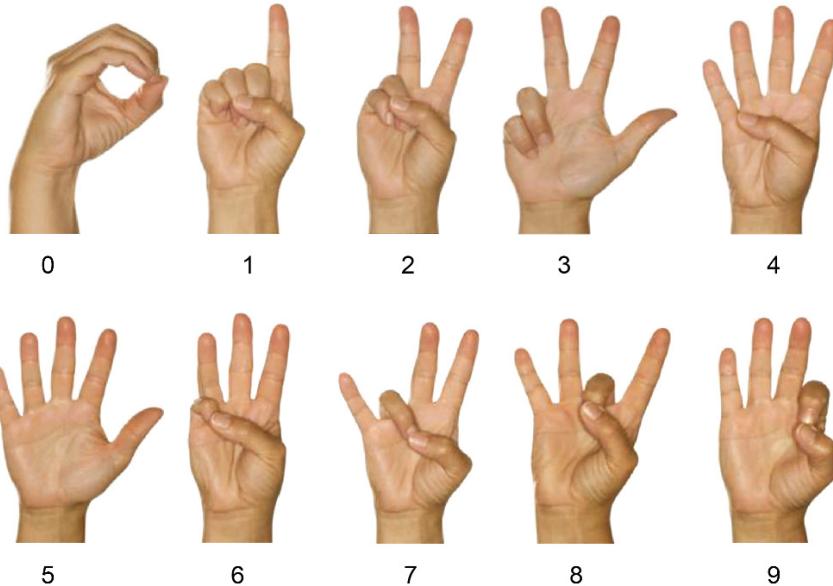


Figure 6.19: a sample from the sign language dataset

Dataset details:

Our dataset contains the following:

- Number of classes = 10 (digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9)
- Total number of images
- Image size = 100x100
- Color space = RGB
- 1712 images in the training set
- 300 images in the validation set
- 50 images in the test set

It is very noticeable how small our dataset is. If you try to train your network from scratch on this very small dataset, you will not achieve any good results. On the other hand, we were able to achieve an accuracy higher than 98% using transfer learning even though the source and target domains were so different.

** please take this evaluation with a grain of salt because it hasn't been thoroughly tested with a lot of data since we only have 50 test image in this dataset. Transfer learning is expected to achieve good results anyway but I just wanted to highlight this fact.

Visit the book website at www.computervisionbook.com to download the source code notebook and the dataset used for this project. Similar to project 1, the training does not require high computation power so you can run this notebook on your personal computer and you don't need a GPU.

For ease of comparison with the previous project, we are going to use VGG16 network that was trained on ImageNet dataset as well.

The process that we are going to use to fine tune a pretrained network is as follows:

1. Import the necessary libraries
2. Preprocess the data to make it ready for the neural network
3. Load in pretrained weights from the VGG16 network trained on a large dataset (ImageNet)
4. Freeze **part** of the feature extractor part
5. Add the new classifier layers
6. Compile the network and run the training process to optimize the model for the smaller dataset
7. Evaluate the model

Now let's go through these steps one-by-one and implement this project.

1. IMPORT THE NECESSARY LIBRARIES

```
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing import image
from keras.applications import imagenet_utils
from keras.applications import vgg16
from keras.optimizers import Adam, SGD
from keras.metrics import categorical_crossentropy
from keras.layers import Dense, Flatten, Dropout, BatchNormalization
from keras.models import Model
from sklearn.metrics import confusion_matrix
import itertools
import matplotlib.pyplot as plt
%matplotlib inline
```

2. PREPROCESS THE DATA TO MAKE IT READY FOR THE NEURAL NETWORK

Similar to project one, we will use the ImageDataGenerator class from Keras and the flow_from_directory() method to preprocess our data. The data is already structured for you to directly create your tensors.

```
train_path = 'dataset/train'
valid_path = 'dataset/valid'
test_path = 'dataset/test'

# ImageDataGenerator generates batches of tensor image data with real-time data augmentation.
# The data will be looped over (in batches).
# in this example, we won't be doing any image augmentation
train_batches = ImageDataGenerator().flow_from_directory(train_path,
                                                       target_size=(224, 224),
                                                       batch_size=10)

valid_batches = ImageDataGenerator().flow_from_directory(valid_path,
```

```

        target_size=(224,224),
        batch_size=30)

test_batches = ImageDataGenerator().flow_from_directory(test_path,
                                                       target_size=(224,224),
                                                       batch_size=50,
                                                       shuffle=False)

Found 1712 images belonging to 10 classes.
Found 300 images belonging to 10 classes.
Found 50 images belonging to 10 classes.

```

3. LOAD IN PRETRAINED WEIGHTS FROM THE VGG16 NETWORK TRAINED ON A LARGE DATASET (IMAGENET)

We download the VGG16 architecture from Keras library with ImageNet weights. Note that we used the parameter pooling='avg' here. This is basically means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. We use this as an alternative to the Flatten layer before adding the FC layers.

```
base_model = vgg16.VGG16(weights = "imagenet", include_top=False, input_shape = (224,224, 3),
                           pooling='avg')
```

4. FREEZE PART OF THE FEATURE EXTRACTOR PART

Here we want to freeze only a part of the feature extraction part and fine tune the rest on our new training data. Remember that in fine tuning we want to freeze part of the network and tune the rest. How we decide the level of fine tuning is usually determined by trial and error. VGG16 has 13 convolutional layers, you can freeze them all or freeze few of them depends on how similar your data is to the source data. In the sign language case, the new domain is very different from our domain, so we will start with fine tuning only the last 5 layers then if we don't get satisfying results we can fine tune more. It turns out that after we trained the new model, we got 98% accuracy so this is a good level of fine tuning. But in other cases, if you find that your network didn't converge, try to fine tune more layers.

```
# iterate through its layers and lock them except for the last 5 layers
for layer in base_model.layers[:-5]:
    layer.trainable = False

base_model.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584

block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
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_pool (MaxPooling2D)	(None, 7, 7, 512)	0
global_average_pooling2d_1 ((None, 512)	0
<hr/>		
Total params:	14,714,688	
Trainable params:	7,079,424	
Non-trainable params:	7,635,264	

5. ADD THE NEW CLASSIFIER LAYERS AND BUILD THE NEW MODEL

```
# save the output of the base_model to be the input of the next layer
last_output = base_model.output

# add our new softmax layer with 10 hidden units
x = Dense(10, activation='softmax', name='softmax')(last_output)

# instantiate a new_model using keras's Model class
new_model = Model(inputs=base_model.input, outputs=x)

# print the new_model summary
new_model.summary()

Layer (type)          Output Shape         Param #
=====
input_1 (InputLayer)  (None, 224, 224, 3)   0
=====
block1_conv1 (Conv2D)  (None, 224, 224, 64)  1792
block1_conv2 (Conv2D)  (None, 224, 224, 64)  36928
block1_pool (MaxPooling2D) (None, 112, 112, 64)  0
block2_conv1 (Conv2D)  (None, 112, 112, 128)  73856
```

block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
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_pool (MaxPooling2D)	(None, 7, 7, 512)	0
global_average_pooling2d_1 ((None, 512)	0
softmax (Dense)	(None, 10)	5130
<hr/>		
Total params: 14,719,818		
Trainable params: 7,084,554		
Non-trainable params: 7,635,264		

6. COMPILE THE NETWORK AND RUN THE TRAINING PROCESS TO OPTIMIZE THE MODEL FOR THE SMALLER DATASET

```
new_model.compile(Adam(lr=0.0001), loss='categorical_crossentropy', metrics=['accuracy'])

from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='signlanguage.model.hdf5', save_best_only=True)

history = new_model.fit_generator(train_batches, steps_per_epoch=18,
                                   validation_data=valid_batches, validation_steps=3, epochs=20, verbose=1,
                                   callbacks=[checkpointer])

Epoch 1/150
18/18 [=====] - 40s 2s/step - loss: 3.2263 - acc: 0.1833 - val_loss: 2.0674 - val_acc: 0.1667
Epoch 2/150
18/18 [=====] - 41s 2s/step - loss: 2.0311 - acc: 0.1833 - val_loss: 1.7330 - val_acc: 0.3000
Epoch 3/150
18/18 [=====] - 42s 2s/step - loss: 1.5741 - acc: 0.4500 - val_loss:
```

```

    1.5577 - val_acc: 0.4000
Epoch 4/150
18/18 [=====] - 42s 2s/step - loss: 1.3068 - acc: 0.5111 - val_loss:
0.9856 - val_acc: 0.7333
Epoch 5/150
18/18 [=====] - 43s 2s/step - loss: 1.1563 - acc: 0.6389 - val_loss:
0.7637 - val_acc: 0.7333
Epoch 6/150
18/18 [=====] - 41s 2s/step - loss: 0.8414 - acc: 0.6722 - val_loss:
0.7550 - val_acc: 0.8000
Epoch 7/150
18/18 [=====] - 41s 2s/step - loss: 0.5982 - acc: 0.8444 - val_loss:
0.7910 - val_acc: 0.6667
Epoch 8/150
18/18 [=====] - 41s 2s/step - loss: 0.3804 - acc: 0.8722 - val_loss:
0.7376 - val_acc: 0.8667
Epoch 9/150
18/18 [=====] - 41s 2s/step - loss: 0.5048 - acc: 0.8222 - val_loss:
0.2677 - val_acc: 0.9000
Epoch 10/150
18/18 [=====] - 39s 2s/step - loss: 0.2383 - acc: 0.9276 - val_loss:
0.2844 - val_acc: 0.9000
Epoch 11/150
18/18 [=====] - 41s 2s/step - loss: 0.1163 - acc: 0.9778 - val_loss:
0.0775 - val_acc: 1.0000
Epoch 12/150
18/18 [=====] - 41s 2s/step - loss: 0.1377 - acc: 0.9667 - val_loss:
0.5140 - val_acc: 0.9333
Epoch 13/150
18/18 [=====] - 41s 2s/step - loss: 0.0955 - acc: 0.9556 - val_loss:
0.1783 - val_acc: 0.9333
Epoch 14/150
18/18 [=====] - 41s 2s/step - loss: 0.1785 - acc: 0.9611 - val_loss:
0.0704 - val_acc: 0.9333
Epoch 15/150
18/18 [=====] - 41s 2s/step - loss: 0.0533 - acc: 0.9778 - val_loss:
0.4692 - val_acc: 0.8667
Epoch 16/150
18/18 [=====] - 41s 2s/step - loss: 0.0809 - acc: 0.9778 - val_loss:
0.0447 - val_acc: 1.0000
Epoch 17/150
18/18 [=====] - 41s 2s/step - loss: 0.0834 - acc: 0.9722 - val_loss:
0.0284 - val_acc: 1.0000
Epoch 18/150
18/18 [=====] - 41s 2s/step - loss: 0.1022 - acc: 0.9611 - val_loss:
0.0177 - val_acc: 1.0000
Epoch 19/150
18/18 [=====] - 41s 2s/step - loss: 0.1134 - acc: 0.9667 - val_loss:
0.0595 - val_acc: 1.0000
Epoch 20/150
18/18 [=====] - 39s 2s/step - loss: 0.0676 - acc: 0.9777 - val_loss:
0.0862 - val_acc: 0.9667

```

Notice the training time of each epoch from the verbose above. It took the model approximately the model was trained very quickly using a regular CPU computing power. Each epoch took approximately 25 to 29 seconds which means that it took the model less than 10 minutes to train for 20 epochs.

7. EVALUATE THE MODEL

Now, it's time to evaluate the accuracy of our model. Similar to the previous example, we are going to use the `evaluate()` method from Keras.

```
print('\nTesting loss: {:.4f}\nTesting accuracy:\n    {:.4f}'.format(new_model.evaluate(test_tensors, test_targets)))
Testing loss: 0.0574
Testing accuracy: 0.9800
```

A deeper level of evaluating your model is to create the confusion matrix. We explained the confusion matrix in chapter 4. It is a table that is often used to describe the performance of a classification model to give us a deeper understanding of how the model performed on the test dataset. Please go back to chapter 4 for more details on the different model evaluation metrics. Now, let's build the confusion matrix for our model.

```
from sklearn.metrics import confusion_matrix

cm_labels = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

cm = confusion_matrix(np.argmax(test_targets, axis=1),
                      np.argmax(new_model.predict(test_tensors), axis=1))
plt.imshow(cm, cmap=plt.cm.Blues)
plt.colorbar()
indexes = np.arange(len(cm_labels))
for i in indexes:
    for j in indexes:
        plt.text(j, i, cm[i, j])
plt.xticks(indexes, cm_labels, rotation=90)
plt.xlabel('Predicted label')
plt.yticks(indexes, cm_labels)
plt.ylabel('True label')
plt.title('Confusion matrix')
plt.show()
```

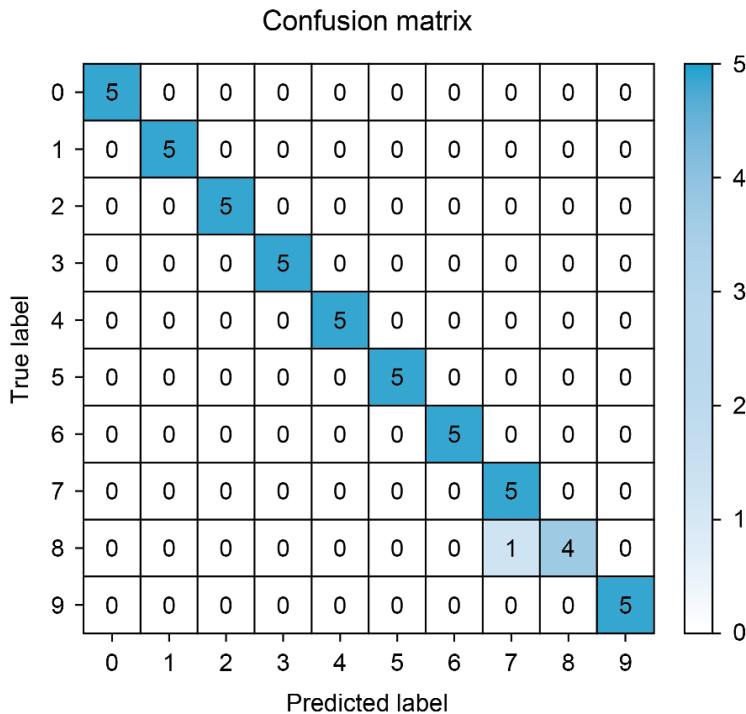


Figure 6.20: confusion matrix for the sign language classifier.

To read this confusion matrix, look at the number in the Predicted Label axis and check if it was correctly classified in the True Label axis. For example, look at number 0 in predicted label axis, you will see that all 5 images are classified as 0 and zero number of images were mistakenly classified as any other number. Similarly, go through the rest of the numbers in the predicted label axis. You will notice that the model successfully made the correct predictions for all the test images except for the image with true label = 8. Where the model has mistakenly classified an image of number 8 and thought it was number = 7.

7

Object Detection with R-CNN, SSD, and YOLO

“The eye sees only what the mind is prepared to comprehend.”

-- Robertson Davies

Learning goals from this chapter:

- Understand the difference between image classification and object detection tasks
- Understand the general framework of object detection projects
- Learn how to use different object detection algorithms like R-CNN, SSD, and YOLO
- By the end of this chapter, we will have gained an understanding of how deep learning is applied to object detection, and how the different object detection models inspire and diverge from one another.

In the previous chapters, we explained how we can use deep neural networks for image classification tasks. In image classification, we assume that there is only one main target object in the image and the model’s sole focus is to identify the target category. However, in many situations, there are multiple targets in the image that we are interested in. We not only want to classify them, but also want to obtain their specific positions in the image. In computer vision, we refer to such tasks as object detection. See Figure 7.1 that explains the difference between image classification and object detection tasks.

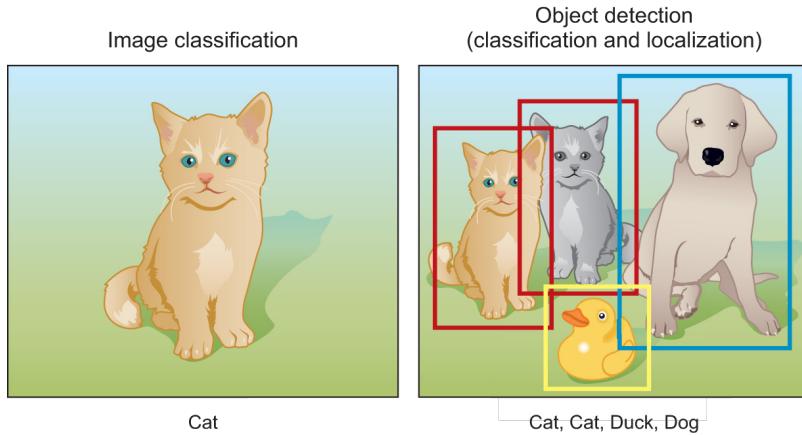


Figure 7.1: Image Classification vs. Object Detection tasks. In classification tasks, the classifier outputs the class probability (cat) whereas, in object detection tasks, the detector outputs the bounding box coordinates (4 boxes in this example) and the predicted classes (2 cats + duck + dog).

Object detection is a computer vision task that involves both main tasks: 1) localizing one or more objects within an image, and 2) classifying each object in the image. This is done by drawing a bounding box around the identified object with its predicted class. This means that the system doesn't just predict the class of the image like in image classification tasks. It also predicts the coordinates of the bounding box that fits the detected object.

It is a challenging computer vision task because it requires both successful object localization in order to locate and draw a bounding box around each object in an image, and object classification to predict the correct class of object that was localized.

Table 7.1

Image Classification	Object Detection
<p>The goal is to predict the type or class of an object in an image.</p> <ul style="list-style-type: none"> • Input: an image with a single object. • Output: a class label (e.g. cat, dog, etc.). • Example output => class probability (e.g. 84% cat). 	<p>The goal is to predict the location of objects in an image via bounding boxes and the classes of the located objects.</p> <ul style="list-style-type: none"> • Input: an image with one or more objects. • Output: one or more bounding boxes (e.g. defined by its coordinates), and a class label for each bounding box. • Example output for an image with 2 objects => <ul style="list-style-type: none"> ◦ box1 coordinates (x, y, w, h) + class probability ◦ box2 coordinates + class probability <p>Note that the image coordinates (x, y, w, h) are as follows: (x and y) are the coordinates of the bounding box center point. (w and h) are the width and height of the box.</p>

Object detection is widely used in many fields. For example, in self-driving technology, we need to plan routes by identifying the locations of vehicles, pedestrians, roads, and obstacles

in the captured video image. Robots often perform this type of task to detect targets of interest. Systems in the security field need to detect abnormal targets, such as intruders or bombs.

Now that you understand what object detection is and what differentiates it from image classification tasks, let's take a look at the general framework of object detection projects. The chapter layout is as follows:

1. First, we will explore the general framework of the object detection algorithms.
2. then, we will dive deep into three of the most popular detection algorithms:
 - a. R-CNN family of networks
 - b. SSD
 - c. YOLO family of networks
3. At the end of the chapter, we will implement what we've learned in a real-world project to train an end-to-end object detector.

Ready? Let's get started!

7.1 General object detection framework

Before we jump into the object detection systems like R-CNN, SSD, and Yolo, let's discuss the general framework of these systems to understand the high level workflow that deep learning based systems follow to detect objects and the metrics they use to evaluate their detection performance.

Don't worry about the code implementation detail of object detectors yet. The goal of this section is to give you a high-level overview of how different object detection systems approach this task and introduce you to a new way of thinking about this problem and a set of new concepts to set you up to understand the different deep learning architectures that we will explain in sections 2, 3, and 4.

Typically, there are four components of an object detection framework.

1. **Region proposal** - an algorithm or a deep learning model is used to generate regions of interest ROI to be further processed by the system. These region proposals are regions that the network believes might contain an object and output a large number of bounding boxes with an *objectness score*. Boxes with large objectness score are then passed along the network layers for further processing.
2. **Feature extraction and network predictions** - visual features are extracted for each of the bounding boxes, they are evaluated and it is determined whether and which objects are present in the proposals based on visual features (i.e. an object classification component).
3. **Non-maximum suppression (NMS)** - at this step, the model has likely found multiple bounding boxes for the same object. Non-max suppression helps avoid repeated detection of the same instance by combining overlapping into a single bounding box for each object.
4. **Evaluation metrics** - similar to accuracy, precision, and recall metrics in image classification tasks (see chapter 4), object detection systems have their own metrics to evaluate their detection performance. In this section we will explain the most popular metrics like mean average precision (mAP), precision-recall curve (PR curve), and intersection over union (IoU).

Now, let's dive one level deeper into each one of these components to build an intuition on what their goals are.

7.1.1 Region proposals

In this step, the system looks at the image and proposes regions of interest for further analysis. The regions of interest (ROI) are regions that the system believes that they have a high likelihood that they contain an object, called *objectness score*. Regions with high objectness score are passed to the next steps whereas, regions with low score are abandoned.

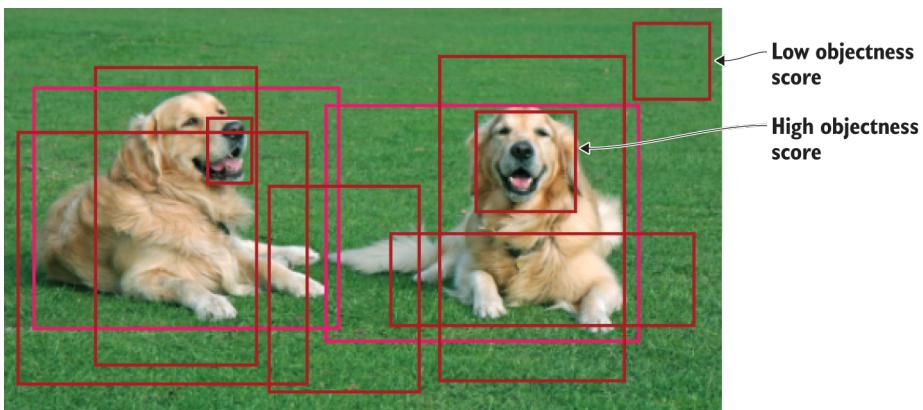


Figure 7.2: Regions of interest (ROI).

There are several approaches to generate region proposals. Originally, the '*selective search*' algorithm was used to generate object proposals. We will talk more about the selective search algorithm when we discuss R-CNN network. Other approaches use more complex visual features extracted by a deep neural network from the image to generate regions (for example, based on the features from a deep learning model).

We will talk in more detail about how different object detection systems approach this task. The important thing to note is that this step produces a lot (thousands) of bounding boxes to be further analyzed and classified by the network. During this step, the network analyzes these regions in the image and classifies each region into *foreground* (object) or a *background* (no object) based on their objectness score. If the objectness score is above a certain threshold, then this region is considered a foreground and pushed forward in the network. Note that this threshold is configurable based on your problem. If the threshold is too low, your network will exhaustively generate all possible proposals and you will have better chances to detect all objects in the image. On the flip side, this will be very computationally expensive and will slow down your detections. So there is a trade-off that is made with region proposal generation is the number of regions vs. the computational complexity and the right approach is to use problem-specific information to reduce the number of ROI's.

7.1.2 Network predictions

This component includes the pretrained CNN network that is used for feature extraction to extract features from the input image that are representative for the task at hand and use these features to determine the class of the image. In object detection frameworks, people typically use pretrained image classification models to extract visual features, as these tend to generalise fairly well. For example, a model trained on the MS COCO or ImageNet dataset is able to extract fairly generic features.

In this step, the network analyzes all the regions that have been identified with high likelihood of containing an object and makes two predictions for each region:

5. Bounding box prediction: the coordinates that locate the box surrounding the object. The bounding box coordinates are represented as the following tuple (x, y, w, h) . Where the x and y are the coordinates of the center point of the bounding box and w and h are the width and height of the box.
6. Class prediction: this is the classic softmax function that predicts the class probability for each object

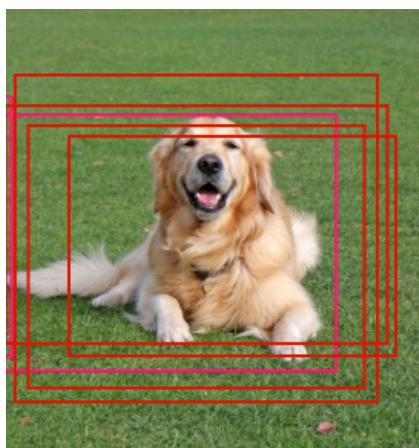


Figure 7.3: Object detector predicting 5 bounding boxes for the dog in the image.

Since there are thousands of regions proposed, each object will always have multiple bounding boxes surrounding it with the correct classification. For example, take a look at the image of the dog in figure 7.3. The network was clearly able to find the object (dog) and successfully classify it. But the detection fired a total of five times because the dog was present in the 5 regions of interest produced in the previous step, hence the five bounding boxes around the dog in the figure. Now, while the detector was successfully able to locate the dog in the image and correctly classify it, but this is not exactly what we need. We just need one bounding box for each object for most problems. Because what if we are building a system to count dogs in an image? Our current system will count 5 dogs. We don't want that. This is when the non-maximum suppression technique comes in handy.

7.1.3 Non-maximum suppression (NMS)

As you can see in figure 7.3, one of the problems of object detection algorithms is that it may find multiple detections of the same object. So, instead of creating only one bounding box around the object, it draws multiple boxes for the same object. Non-maximum suppression (NMS) is a technique that is used to make sure that the detection algorithm detects each object only once. As the name implies, NMS technique looks at all the boxes surrounding an object to find the box that has the *maximum* prediction probabilities and *suppress* or eliminate the other boxes, hence the name, non-maximum suppression.

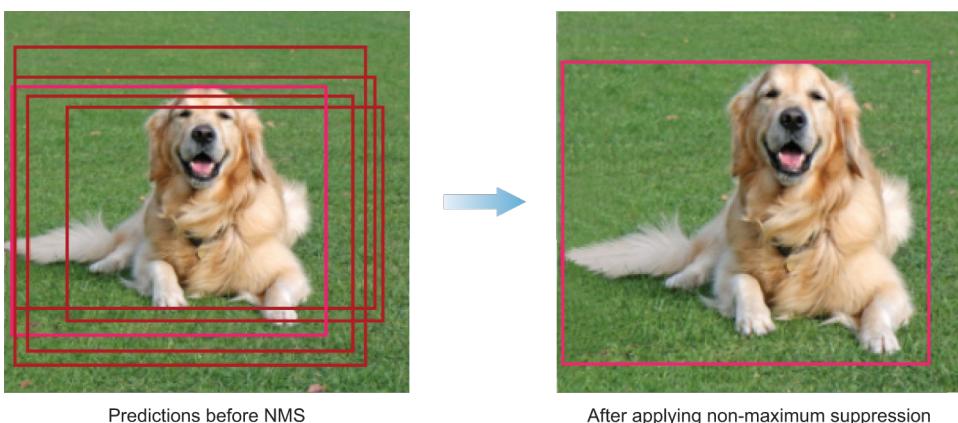


Figure 7.4: multiple bounding boxes detect the dog in the image. After non-maximum suppression, only the box that fits the object the best remains and the rest are ignored as they have large overlaps with the selected one.

The general idea of non-maximum suppression is to reduce the number of candidate boxes to only one bounding box for each object. For example, if the object in the frame is fairly large and more than 2,000 object proposals have been generated, it is quite likely that some of these will have significant overlap with each other and the object.

Let's see the steps of how the NMS algorithm works:

1. Discard all bounding boxes that have predictions that are less than a certain threshold, called **confidence threshold**. This threshold is tunable. This means that the box will be suppressed if the prediction probability is less than the set threshold.
2. Look at all the remaining boxes and select the bounding box with the highest probability.
3. Then calculate the overlap of the remaining boxes that have the same class prediction. Bounding boxes that have high overlap with each other and are predicting the same class are averaged together. This overlap metric is called **Intersection Over Union (IOU)**. The IOU is explained in detail in the next section.
4. The algorithm then suppresses any box that has an IOU value that is smaller than a certain threshold (called **NMS threshold**). Usually the NMS threshold is equal to 0.5 but it is tunable as well if you want to output less or more bounding boxes.

NMS techniques are typically standard across the different detection frameworks, but it is an important step that might require tweaking hyperparameters like the confidence threshold and the NMS threshold based on the scenario.

7.1.4 1.4. Object detector evaluation metrics

When evaluating the performance of an object detector, we use two main evaluation metrics: 1) FPS (frame-per-second) to measure the network detection speed, and 2) mAP (mean Average Precision) to measure the network precision.

FPS TO MEASURE THE DETECTION SPEED

The most common metric that is used to measure the detection speed is the number of frames per second (FPS). For example, Faster R-CNN operates at only 7 frames per second (FPS) whereas SSD operates at 59 FPS. In benchmarking experiments, you will see the authors of a paper stating their network results as: "Network X achieves mAP of Y% at Z FPS". Where X is the network name, Y is the mAP percentage, and Z is the FPS.

MEAN AVERAGE PRECISION (MAP)

The most common evaluation metric that is used in object recognition tasks is 'mAP', which stands for *mean average precision*. It is a percentage from 0 to 100 and higher values are typically better, but its value is different from the accuracy metric in classification.

To understand how mean Average Precision (mAP) is calculated, you first need to understand the Intersection Over Union (IOU) and the Precision-Recall Curve (PR Curve). Let's explain IOU and PR curve then come back to mAP.

Intersection Over Union (IOU)

It is a measure that evaluates the overlap between two bounding boxes: the ground truth bounding box $B_{ground\ truth}$ and the predicted bounding box $B_{predicted}$. By applying the IOU we can tell if a detection is valid (True Positive) or not (False Positive). Figure 7.5 illustrates the IOU between a ground truth bounding box and a predicted bounding box.



Figure 7.5: IoU score is the overlap between the ground truth bounding box and the predicted bounding box.

The intersection over the union value ranges from 0, meaning no overlap at all, to 1 which means that the two bounding boxes 100% overlap on each other. The higher the overlap between the two bounding boxes (IoU value), the better.



Figure 7.6: the higher overlap (IoU) between the two bounding boxes, the better.

To calculate the IoU of a prediction, we need:

- The ground-truth bounding box ($B_{\text{ground truth}}$) the hand-labeled bounding box that is created during the labeling process.
- The predicted bounding box ($B_{\text{predicted}}$) from our model

IoU is calculated by dividing the area of overlap by the area of the union as the following equation:

$$IoU = \frac{B_{\text{ground truth}} \cap B_{\text{predicted}}}{B_{\text{ground truth}} \cup B_{\text{predicted}}}$$

IoU is used to define a “correct prediction”. Meaning, a “correct” prediction (True Positive) is one that has IoU greater than some threshold. This threshold is a tunable value depending on the challenge but 0.5 is a standard value. For example, some challenges like MS COCO, uses mAP@0.5 meaning IoU threshold = 0.5 or mAP@0.75 meaning IoU threshold = 0.75. This means that if the IoU is above this threshold is considered a True Positive (TP) and if it is below it is considered as a False Positive (FP).

Precision-Recall Curve (PR Curve)

With the TP, FP, and FN defined, we can now calculate the precision and recall of our detection for a given class across the testing dataset. As explained in chapter 4, we calculate the precision and recall as follows:

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

After calculating the precision and recall for all classes, the PR Curve is then plotted as follows:

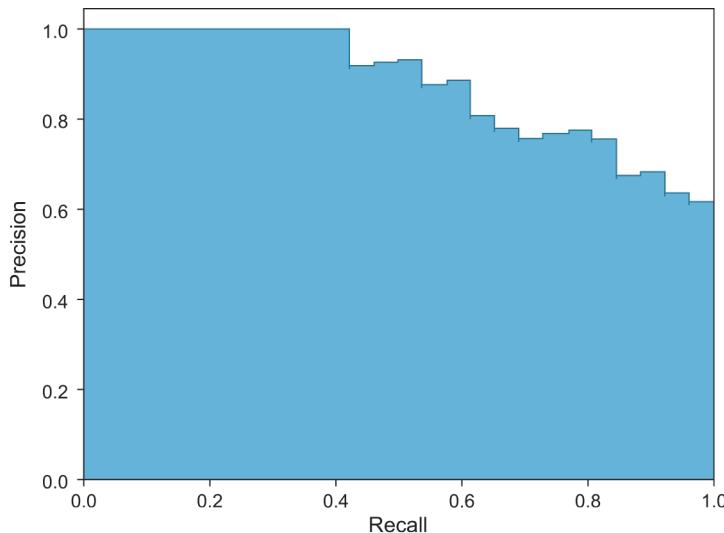


Figure 7.7: precision-recall curve

The PR curve is a good way to evaluate the performance of an object detector as the confidence is changed by plotting a curve for each object class. A detector is considered good if its precision stays high as recall increases, which means that if you vary the confidence threshold, the precision and recall will still be high. On the other hand, a poor detector needs to increase the number of FPs (lower precision) in order to achieve a high recall. That's why the PR curve usually starts with high precision values, decreasing as recall increases.

Now, that we have the PR curve, we calculate the AP (Average Precision) by calculating the Area Under the Curve (AUC). Then finally, mAP for object detection is the average of the AP calculated for all the classes. It is also important to note that for some papers, they use AP and mAP interchangeably.

RECAP

To recap, the mAP is calculated as follows:

1. Each bounding box will have an objectness score associated (probability of the box containing an object).
2. Precision and recall are calculated.
3. Precision-recall curve (PR curve) is computed for each class by varying the score threshold.
4. Calculate the average precision (AP): it is the area under the PR curve. In this step, the AP is computed for each class.
5. Calculate mAP: the average AP over all the different classes.

Last thing to note on mAP is that it is more complicated to calculate than other traditional metrics like accuracy. The good news is that you don't need to compute the mAP values

yourself. Most deep learning object detection implementations handle computing mAP for you as you will see at the end of this chapter.

Now, that we understand the general framework of object detection algorithms, let's dive deeper into three of the most popular detection algorithms. In this chapter, will discuss the R-CNN family of networks, SSD, and YOLO networks in detail to see how object detectors evolved through time and learn the pros and cons of each network to be able to choose the most appropriate algorithm for your problem.

7.2 Region-Based Convolutional Neural Networks (R-CNNs)

The R-CNN family of object detection techniques usually referred to as R-CNNs, which is short for Region-based Convolutional Neural Networks, developed by Ross Girshick et al. in 2014 in their paper "[Rich feature hierarchies for accurate object detection and semantic segmentation](#)". The R-CNN family has then expanded to include [Fast-RCNN](#) and [Faster-RCNN](#) that came out in 2015 and 2016. In this section, I'll quickly walk you through the evolution of R-CNN family from R-CNNs to Fast R-CNN to Faster R-CNN and then we will dive deeper into the Faster R-CNN architecture and code implementation.

7.2.1 R-CNN

The R-CNN is the least sophisticated region-based architecture in its family, but it is the basis for understanding how multiple object recognition algorithms work for all of them. It may have been one of the first large and successful applications of convolutional neural networks to the problem of object detection and localization that paved the way for the other advanced detection algorithms. The approach was demonstrated on benchmark datasets, achieving then state-of-the-art results on the PASCAL VOC-2012 dataset and the ILSVRC-2013 object detection challenge.

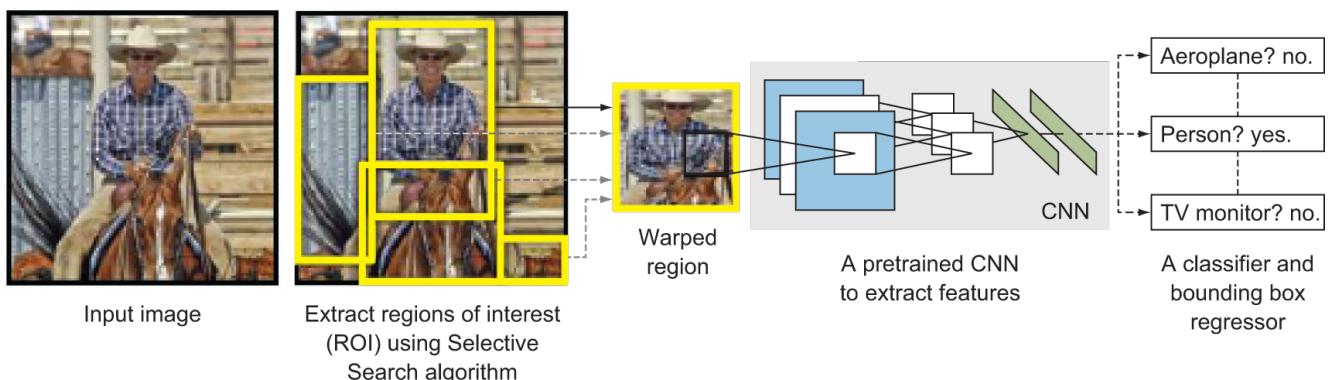


Figure 7.8: Summary of the R-CNN Model Architecture. Taken from the 2014 R-CNN paper and modified.

The R-CNN model is comprised of four components:

1. **Extract regions of interest (RoI)** - also known as extracting region proposals. These are regions that have a high probability of containing an object. The way this is done is

by using an algorithm, called *Selective Search*, to scan the input image to find regions that contain blobs and propose them as regions of interest to be processed by the next modules in the pipeline. The proposed regions of interest are then warped to have a fixed size because they usually vary in size and as we learned in the previous chapters, CNNs require fixed input image size.

Selective Search

Selective search is a greedy search algorithm that is used to provide region proposals that potentially contain objects. It tries to find the areas that might contain an object by combining similar pixels and textures into several rectangular boxes. Selective Search combines the strength of both *exhaustive search algorithm* (which examines all possible locations in the image) and *bottom-up segmentation algorithm* (that hierarchically groups similar regions) to capture all possible object locations.



Figure 7.9: Selective search algorithm looks for blob-like areas in the image to extract regions. In the right image, the segmentation algorithm defines blobs that could be objects. Then the selective search algorithm, select these areas to be passed along for further investigation.

The way the selective search algorithm works is that it applies a segmentation algorithm to find blobs in an image to figure out what could be an object (see right image in figure 7.9). Selective search recursively combines these groups of regions together into larger ones to create 2,000 areas to be investigated as follows:

- First the similarities between all neighbouring regions are calculated.
- The two most similar regions are grouped together, and new similarities are calculated between the resulting region and its neighbours.
- This process is then repeated until the whole object is covered in a single region.

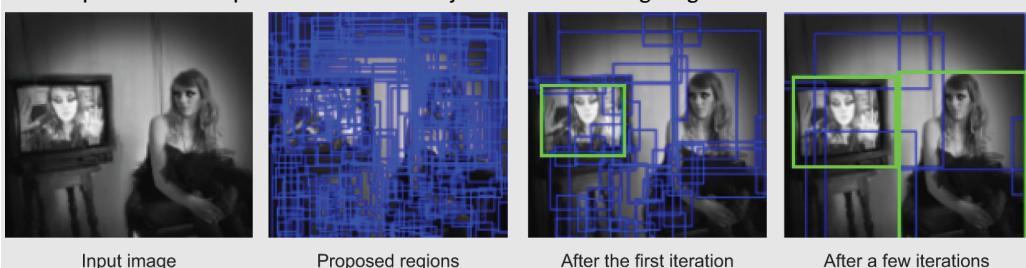


Figure 7.10: An example of bottom-up segmentation using selective search algorithm. It combines similar regions in every iteration until the whole object is covered in a single region.

Note that a review of the Selective Search algorithm and how it calculates the regions similarity is outside the scope of this book. If you are interested to learn more about this technique, you can refer to the original paper “[Selective Search for Object Recognition](#)”. For the purpose of understanding R-CNNs, you can treat the Selective Search algorithm as a black box that intelligently scans the image and proposes ROI locations for us to use in the following steps.

2. **Feature Extraction module** - we run a pretrained convolutional network on top of the region proposals to extract features from each candidate region. This is the typical CNN feature extractor that we learned in the previous chapters of this book.
3. **Classification module** - train a classifier like Support Vector Machine (SVM), a traditional machine learning algorithm, to classify candidate detections based on the extracted features from the previous step.
4. **Localization module** - also known as, bounding box regressor. Let's take a step back to understand regression. Machine learning problems are categorized as classification and regression problems. Classification algorithms output a discrete, predefined classes (dog, cat, elephant) whereas regression algorithms output continuous value predictions. In this module, we want to predict the location and size of the bounding box that surrounds the object. The bounding box is represented by identifying four values: the x and y coordinates of the box's origin (x, y), the width, and the height of the box (w, h). Putting this together, the regressors predicts the four real-valued numbers that define the bounding box as the following tuple (x, y, w, h) .

Figure 7.11 illustrates the R-CNN architectures in an intuitive way.

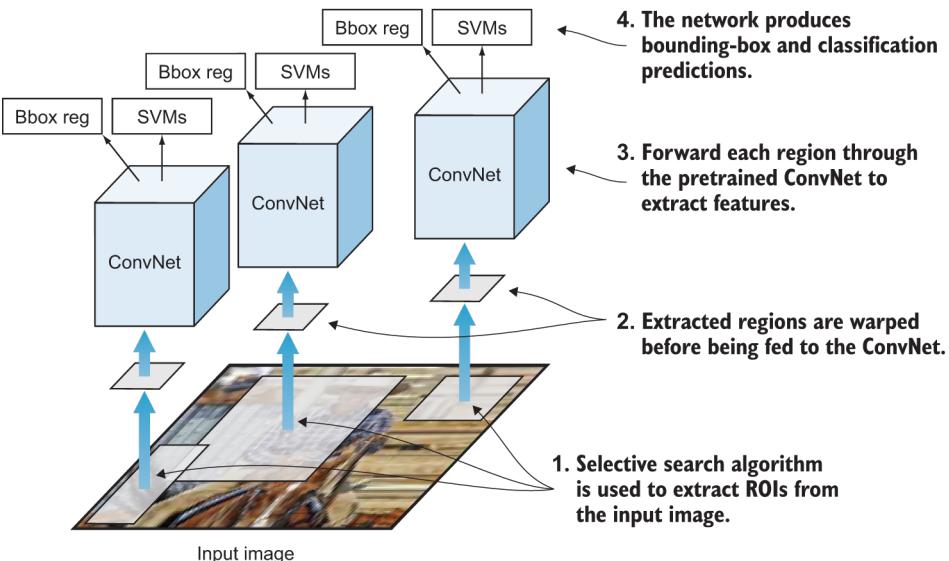


Figure 7.11: R-CNN architecture. Each proposed ROI is passed through the CNN to extract features then an SVM classifier.

As you can see in figure 7.11, the network first proposes regions of interest, then extract features, and then classify those regions based on their features. In essence, we have turned object detection into an image classification problem.

HOW DO WE TRAIN R-CNNs?

We learned that R-CNNs are composed of four modules: selective search region proposal, feature extractor, classifier, and bounding box regressor. All the R-CNN modules need to be trained except for the selective search algorithm. So, in order to train R-CNNs, we need to train the following modules:

1. Feature extractor CNN: this is a typical CNN training process. In here, we either train a network from scratch which rarely happens or fine-tune a pretrained network as we learned in chapter 6.
2. Train the SVM classifier: the Support Vector Machine algorithm is not covered in this book, but it is a traditional machine learning classifier that is no different than deep learning classifiers in the sense that it needs to be trained on labeled data.
3. Train the bounding box regressors: another model that outputs four real-valued numbers for each of the K object classes for tightening region bounding boxes.

Looking through the R-CNN learning steps, you could easily find out that training an R-CNN model is expensive and slow. The training process involves training three separate modules without much shared computation. This multi-stage pipeline training is one of the disadvantages of R-CNNs as we will see next.

WHAT ARE THE DISADVANTAGES OF R-CNN?

R-CNN is very simple to understand and it achieved state-of-the-art results when it first came out. Especially when using deep ConvNet to extract features. However, it has the following notable drawbacks:

1. Very slow object detection: selective search algorithm proposes about 2,000 regions of interest per a single image to be examined by the entire pipeline (CNN feature extractor + classifier). It is very computationally expensive because it performs a ConvNet forward pass for each object proposal, without sharing computation which will make it incredibly slow since the Selective Search algorithm extracts thousands of regions that need to be investigated for our objects. This high computation need makes R-CNN not a good fit for many applications, especially real-time applications that requires very fast inferences like self-driving cars and many others.
2. Training is a multi-stage pipeline: as we discussed earlier, R-CNNs requires the training of three modules: CNN feature extractor, SVM classifier, and the bounding-box regressors. Which makes the training process very complex and not an end-to-end training.
3. Training is expensive in space and time: when training the SVM classifier and the bounding-box regressor, features are extracted from each object proposal in each image and written to disk. With very deep networks, such as VGG16, the training process of a few thousand images takes days using GPUs. The training process is expensive in space as well because the extracted features require hundreds of gigabytes of storage.

R-CNN is not actually a one end-to-end system that learns to localize via deep neural network. Instead, it is a combination of standalone algorithms, added together to perform object detection. First, it uses a greedy search algorithm to propose regions of interest (ROIs) that the algorithm believes are “worth examining” and then pass these regions on to a feature extractor then to a classifier to label them. Which makes the process very complex in training and very slow in detections. What we need is to build an end-to-end deep learning system that fixes the disadvantages of R-CNNs while improving their speed and accuracy.

7.2.2 Fast R-CNN

Fast R-CNN was an immediate descendant for R-CNN. It came out in 2015 by Ross Girshick in his paper "[Fast R-CNN](#)". Fast R-CNN resembled the R-CNN technique in many ways, but improved on its detection speed while also increasing detection accuracy through two main changes:

1. Instead of starting with the regions proposal module then the feature extraction module like R-CNN, Fast-RCNN proposes that we apply the CNN feature extractor first to the entire input image then propose regions. This way we only run one ConvNet over the entire image instead of 2000 ConvNets over 2000 overlapping regions.
2. Extend the ConvNet to do the classification part as well by replacing the traditional machine learning algorithm, SVM, with a softmax layer. This way we have only one model to perform both tasks: 1) feature extraction and 2) object classification.

FAST R-CNN ARCHITECTURE

As we can see from the image, Fast R-CNN is now generating region proposals based on the last feature map of the network, not from the original image itself like in R-CNN. As a result, we can train just one ConvNet for the entire image. In addition, instead of training many different SVM algorithms to classify each object class, there is a single softmax layer that outputs the class probabilities directly. Now we only have one neural net to train, as opposed to one neural net and many SVM's.

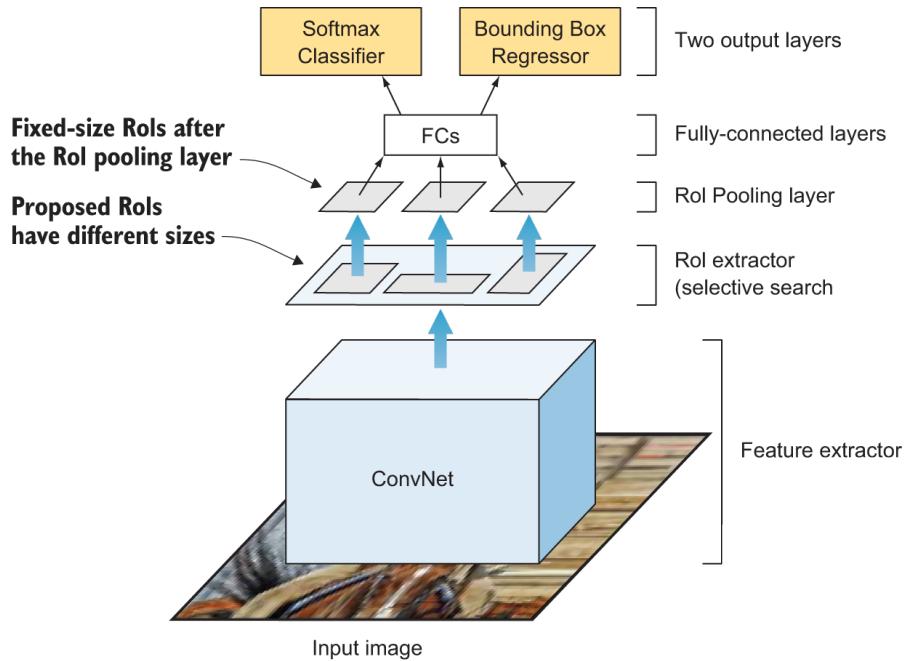


Figure 7.12: Fast R-CNN architecture. An input image is the input of a fully convolutional network. Each ROI

The architecture of Fast R-CNN consists of the following modules:

1. **Feature extractor module** - the network starts with a ConvNet to extract features from the full image.
2. **Regions of Interest (RoI) extractor** - selective search algorithm to propose 2,000 regions candidates per image.
3. **RoI Pooling layer** - this is a new component that was introduced in Fast R-CNN architecture to extract a fixed-size window from the feature map before feeding the RoIs to the fully-connected layers. It uses max pooling to convert the features inside any valid region of interest into a small feature map with a fixed spatial extent of Height \times Width (HxW). The RoI pooling layer will be explained in more detail in the Faster R-CNN section, but for now, understand that the RoI pooling layer is applied on the last feature map layer extracted from the CNN and its goal is to extract fixed-size regions of interest to feed then in to the FC layers then the output layers.
4. **Two-head output layer** - the model branches into two heads:
 - a. A softmax classifier layer that outputs a discrete probability distribution per ROI
 - b. A bounding-box regressor layer to predict offsets relative to the original ROI

MULTI-TASK LOSS FUNCTION IN FAST R-CNNs

Since Fast R-CNN is an end-to-end learning architecture to learn the class of object as well as the associated bounding box position and size, the loss is multi-task loss. With multi-task loss,

at the output, we have softmax and bounding box regressor as shown at the top right of the figure.

In any optimization problem, we need to define a loss function that our optimizer algorithm is trying to minimize. Read chapter 2 for more details on optimization and loss functions. In object detection problems, our goal is to optimize for two goals: object classification and object localization. Therefore, we will have two loss functions in this problem: L_{cls} for the classification loss and L_{loc} for the bounding box prediction defining the object location.

Fast R-CNN network has two sibling output layers with two loss functions:

1. **Classification:** the first outputs a discrete probability distribution (per RoI), over $K + 1$ categories (we add 1 class for the background). The probability P is computed by a softmax over the $K+1$ outputs of a fully connected layer. The classification loss function is a log loss for the true class u .

$$L_{cls}(p, u) = -\log p_u$$

u = is the true label, $u \in 0, 1, 2, \dots K+1$. where $u = 0$ is the background

p = discrete probability distribution per RoI over $k+1$ classes

2. **Regression:** the second sibling layer outputs bounding-box regression offsets, $v = (x, y, w, h)$, for each of the K object classes. The loss function is the loss for bounding box for class u .

$$L_{loc}(t^u, u) = \sum L_{1smooth}(t_i^u - v_i)$$

v = true bounding box, $v = (x, y, w, h)$

t^u = prediction bounding box correction, $t^u = (t_x^u, t_y^u, t_w^u, t_h^u)$

$L_{1smooth}$ = the bounding box loss measures the difference between t_i^u and v_i using the Smooth L1 Loss function. It is a robust function and is claimed to be less sensitive to outliers than other regression loss like L2.

The overall loss function is:

$$L = L_{cls} + L_{loc}$$

$$L(p, u, t^u, v) = L_{cls}(p, u) + [u \geq 1] l_{box}(t^u, v)$$

Note that $[u \geq 1]$ is added before the regression loss to indicate 0 when the region inspected doesn't contain any object and contains a background. It is a way of ignoring the bounding box regression when the classifier labels the region as a background. The indicator function $[u \geq 1]$ is defined as:

$$[u \geq 1] = \begin{cases} 1 & \text{if } u \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

DISADVANTAGES OF FAST R-CNN

Fast R-CNN is much faster in both testing time, because you don't have to feed 2000 region proposals to the convolutional neural network every time. Instead, a convolution operation is done only once per image and a feature map is generated from it. Training is also faster because all of its components are in one CNN network: feature extractor, object classifier, and bounding-box regressor. However, there was just one big bottleneck remaining: the selective search algorithm for generating region proposals is very slow and it is generated separately by another model. The last step to achieve a complete end-to-end object detection system using deep learning is to find a way to combine the region proposal algorithm into our end-to-end deep learning network. This is what Faster R-CNN did as we will see next.

7.2.3 Faster R-CNN

Faster R-CNN is the third iteration of the R-CNN family. It came out in 2016 by Ross Girshick et. al in their paper "[Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks](#)". Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map. Instead of using a selective search algorithm on the feature map to identify the region proposals, a network is used to predict the region proposals as part of the training process, called **Region Proposal Network (RPN)**. The predicted region proposals are then reshaped using an ROI pooling layer then used to classify the image within the proposed region and predict the offset values for the bounding boxes. These improvements both reduce the number of region proposals and accelerate the test-time operation of the model to near real-time with then state-of-the-art performance.

FASTER R-CNN ARCHITECTURE

The architecture of Faster R-CNN can be described by two main networks:

1. **Region Proposal Network (RPN)** - selective search is replaced by a ConvNet that to propose regions of interest (RoI) from the last feature maps of the feature extractor to be considered for investigations. RPN has two outputs; the "objectness score" (object or no object) and the box location
2. **Fast R-CNN** - consists of the typical components of Fast R-CNN:
 - a. Base network for Feature extractor: a typical pre-trained CNN model to extract features from the input image.
 - b. ROI pooling layer: to extract fixed-size regions of interest.
 - c. Output layer: contains 2 fully-connected layers: 1) a softmax classifier to output the class probability, and 2) a bounding-box regression CNN to the bounding box predictions.

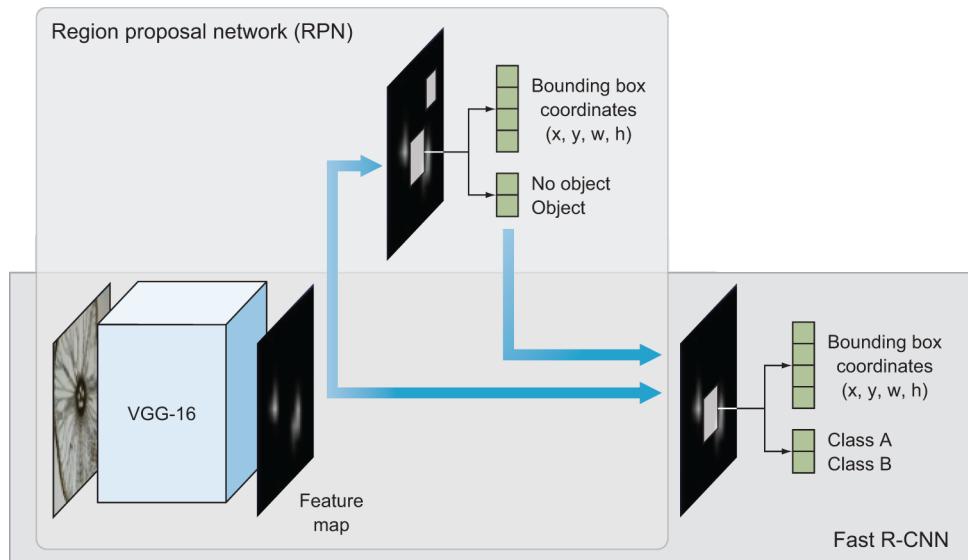


Figure 7.13: Faster R-CNN architecture has two main components: 1) a region proposal network (RPN), which identifies regions that may contain objects of interest and their approximate location; and 2) a Fast R-CNN network, which classifies objects, and refines their location, defined using bounding boxes. The two components share the convolutional layers of the pre-trained VGG-16

As you can see in the Faster R-CNN architecture diagram (Figure 7.13), the input image is presented to the network and its features are extracted via a pre-trained CNN. These features, in parallel, are sent to two different components of the Faster R-CNN architecture:

1. The RPN to determine where in an image a potential object could be. At this point we do not know what the object is, just that there is potentially an object at a certain location in the image.
2. ROI pooling to extract fixed-size windows of features

The output is then passed into two fully-connected layers: one for the object classifier and one for the bounding box coordinates predictions to obtain our final localizations.

This architecture achieves an end-to-end trainable and the complete object detection pipeline where all the required components take place inside the network, including:

1. Base network feature extractor
2. Regions proposal
3. ROI pooling
4. Object classification
5. Bounding box regressor

BASE NETWORK TO EXTRACT FEATURES

Similar to Fast R-CNN, the first step is using a pretrained CNN and slice off its classification part. The base network is used to extract features from the input image. We covered how this works in detail in chapter 6 explaining Transfer Learning. In this component, you can use any of the popular CNN architectures based on the problem that you are trying to solve. The original Faster R-CNN paper used [ZF](#) and [VGG](#) pretrained networks on ImageNet but since then there have been lots of different networks with a varying number of weights. For example, [MobileNet](#), a smaller and efficient network architecture optimized for speed, has approximately 3.3M parameters, while ResNet-152 (152 layers), once the state of the art in the ImageNet classification competition, has around 60M. Most recently, new architectures like [DenseNet](#) are both improving results while lowering the number of parameters.

VGGNet vs. ResNet

Nowadays, ResNet architectures have mostly replaced VGG as a base network for extracting features. The obvious advantage of ResNet over VGG is that it has much more layers (deeper) giving it more capacity to learn very complex features. This is true for the classification task and should be equally true in the case of object detection. Also, ResNet makes it easy to train deep models with the use of residual connections and batch normalization, which was not invented when VGG was first released. Please revisit chapter 5 for more detail review on the different CNN architectures.

As we learned in the previous chapters, each convolutional layer creates abstractions based on the previous information. The first layers usually learn edges, the second finds patterns in edges in order to activate for more complex shapes and so forth. Eventually we end up with a convolutional feature map that can be fed to the RPN to extract regions that contain objects.

REGION PROPOSAL NETWORK (RPN)

The region proposal network identifies regions that could potentially contain objects of interest, based on the last feature map of the pre-trained convolutional neural network. RPN is also known as the ‘attention network’ because it guides the networks attention to interesting regions in the image. Faster R-CNN uses Region Proposal Network (RPN) to bake the region proposal directly into the R-CNN architecture instead of running a Selective Search algorithm to extract regions of interests.

The architecture of the RPN is composed of two layers:

- A 3x3 fully-convolutional layer with 512 channels
- Two parallel 1x1 convolutional layers: one is a classification layer that is used to predict the class of whether the region contains an object or not (the score of it being background or foreground). And the other layer is for regression or bounding box prediction.

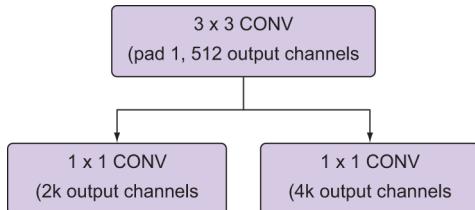


Figure 7.14: Convolutional implementation of an RPN architecture, where k is the number of anchors.

Fully-convolutional networks (FCN)

One important aspect of object detection networks is that they should be fully convolutional. A fully convolutional neural network means that the network does not contain any fully-connected (FC) layers typically found at the end of a network prior to making output predictions.

In the context of image classification, removing the fully-connected layers is normally accomplished by applying average pooling across the entire volume prior to a single dense softmax classifier used to output the final predictions. A fully-convolutional network (FCN) has two main benefits:

1. Faster because it contains only convolution operations and no FC layers.
2. Able to accept images of any spatial resolution (width and height), provided that the image and network can fit into memory, of course.

Being an FCN makes the network invariant to the size of the input image. However, in practice, we might want to stick to a constant input size due to various problems that only show their heads when we are implementing the algorithm. A big one amongst these problems is that if we want to process our images in batches (images in batches can be processed in parallel by the GPU, leading to speed boosts), we need to have all images of fixed height and width.

The 3x3 CONV layer is applied on the last feature map of the base network where a sliding window of size 3x3 is passed over the feature map. The output is then passed to two 1x1 CONV layers, a classifier and a bounding-box regressor. Note that the classifier and the regressor of the RPN are not trying to predict the class of the object and its bounding box. This will come later after the RPN. Remember, the goal of the RPN is to determine whether the region has an object or not to be investigated by the fully-connected layers afterwards. In RPN, we use a binary classifier to predict the *objectness score* of the region to determine the probability of this region being a foreground (containing an object) or a background (doesn't contain an object). It basically looks at the region and asks: "Does this region contain an object?". If the answer is yes, then the region is passed along for further investigation by ROI pooling and the final output layers. See figure 7.15.

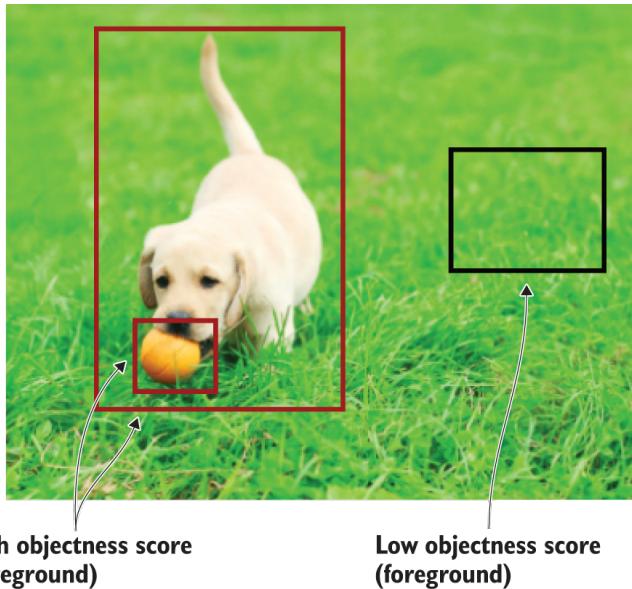


Figure 7.15: The RPN classifier predicts the objectness score which is the probability of an image containing an object (foreground) or a background.

HOW DOES THE REGRESSOR PREDICT THE BOUNDING BOX?

To answer this question, let's first define the bounding box. It is the box that surrounds the object and is identified by the following tuple (x, y, w, h) . Where the x and y are the coordinates in the image that describes the center of the bounding box and the h and w are the height and width of the bounding box. Researchers found that defining the (x, y) coordinates of the center point could be challenging because we have to enforce some rules to make sure that the network predicts values *inside* the boundaries of the image. Instead, we can create reference boxes called **anchor boxes** in the image and make the regression layer predict the offsets from these boxes called deltas $(\Delta_x, \Delta_y, \Delta_w, \Delta_h)$ to adjust the anchor boxes to better fit the object to get the final proposals.

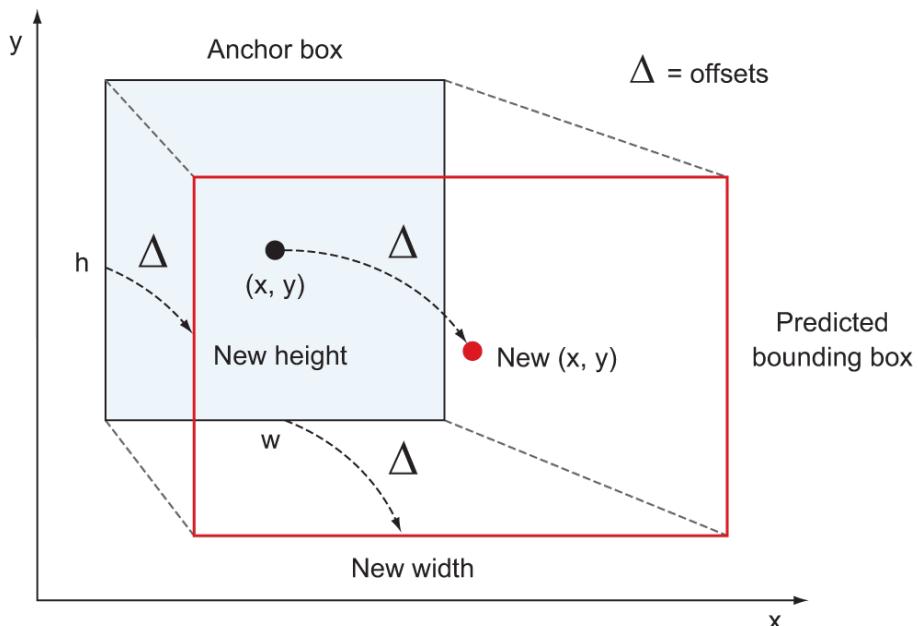


Figure 7.16: illustration of predicting the delta shift from the anchor boxes and the bounding-box coordinates.

Anchor boxes - using a sliding window approach, RPN generates k regions for each location in the feature map. These regions, are represented as boxes called anchor boxes. The anchors are all centered in the middle of their corresponding sliding window, and differ in terms of scale and aspect ratio to cover a wide variety of objects. These are fixed bounding boxes that are placed throughout the image to be used for reference when first predicting object locations. In their paper, Ross Girshick et. al generated 9 anchor boxes all have the same center but with 3 different aspect ratios and 3 different scales as shown below.

Let's see how the anchor boxes are applied in an example:

Anchors are at the center of the sliding windows. Each window has k anchor boxes with the anchor at their center.

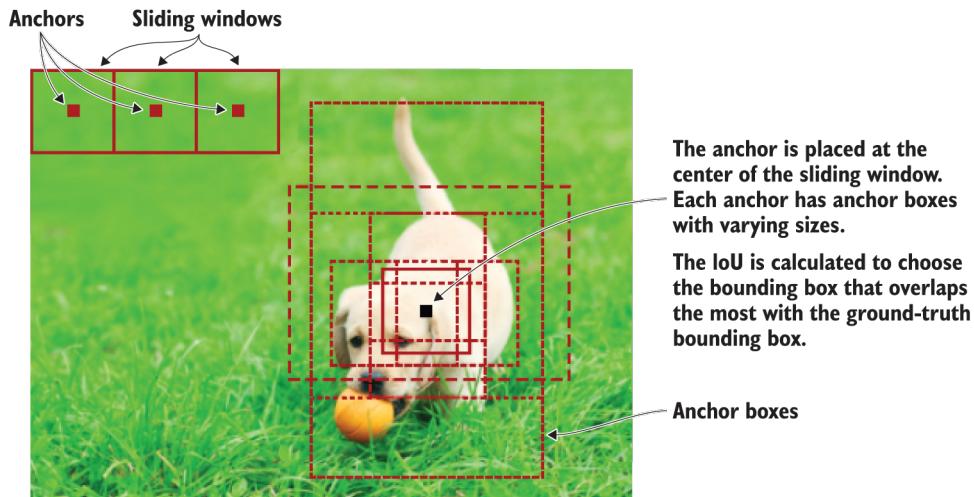


Figure 7.17: Anchor boxes at the center of each sliding window. IoU is calculated to select the bounding box the overlaps the most with the ground-truth.

HOW DO WE TRAIN THE RPN?

The region proposal network is trained to classify an anchor box to output an “objectness” score (i.e., as containing an object of interest or not), and also to approximate the four coordinates of the object (location parameters). RPN is trained using human annotators to label the bounding boxes, this labeled box is called the **ground-truth**.

For each anchor box, the overlap probability value (p) is computed which indicates how much these anchors overlap with the ground-truth bounding boxes.

$$p = \begin{cases} 1 & \text{if } IoU > 0.7 \\ -1 & \text{if } IoU < 0.3 \\ 0 & \text{otherwise} \end{cases}$$

If an anchor has high overlap with a ground-truth bounding box, then it is likely that the anchor box includes an object of interest, and it is labeled as positive with respect to the object versus no object classification task. Similarly, if an anchor has small overlap with a ground truth bounding box, it is labeled as negative. During the training process, the positive and negative anchors are passed as input to two fully connected layers corresponding to the classification of anchors as containing object or no object, and to the regression of location parameters (four coordinates), respectively. Corresponding to the k number of anchors from a location, the RPN network outputs $2k$ scores and $4k$ coordinates. This means that if the number of anchors per sliding window (k) = 9, then the RPN outputs 18 objectness scores and 36 location coordinates.

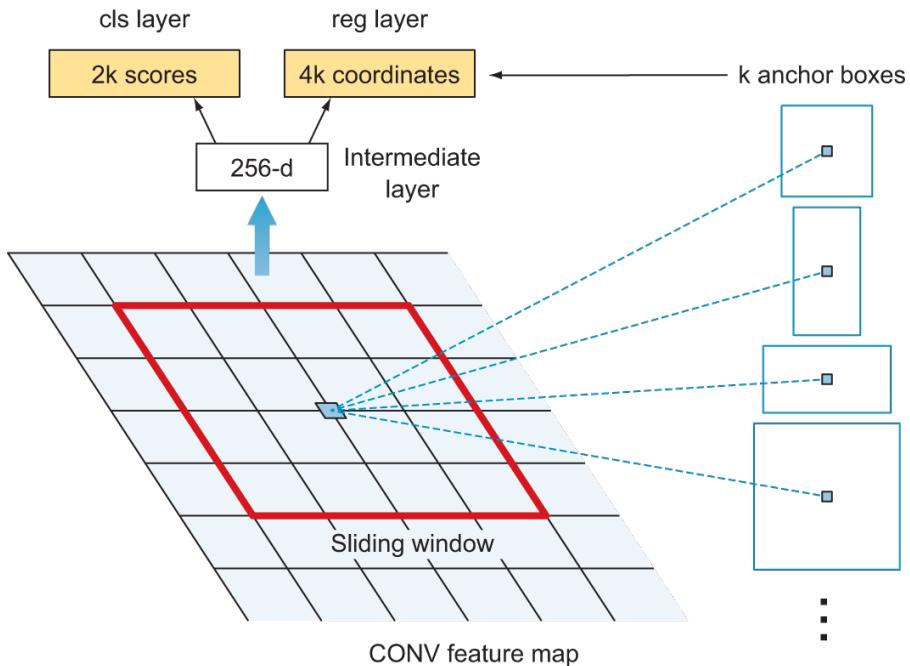


Figure 7.18: Region Proposal Network

RPN as a standalone application

The RPN can be used by itself as a standalone application. For example, in problems where there is only a single class of objects, the objectness probability can be used as the final class probability. This is because for this case, “foreground” = “single class” and “background” = “not single class”. The reason you would want to use RPN for cases like single class detection is the gain in speed both in training and prediction. Since the RPN is a very simple network which only uses convolutional layers, the prediction time can be faster than using the classification base network.

FULLY CONNECTED LAYERS

The output FC layers take two inputs: one is the feature maps coming from the base convnet and the other is the regions of interest coming from the RPN. It then classifies the selected regions and outputs their prediction class and the bounding box parameters. The object classification layer in Fast R-CNN uses the softmax activation, while the location regression layer uses linear regression over the coordinates defining the location as a bounding box. All parameters of the network are trained together using a multi-task loss.

MULTI-TASK LOSS FUNCTION

Similar to Fast R-CNN, Faster R-CNN is optimized for a multi-task loss function where the loss function combines the losses of classification and bounding box regression.

$$L = L_{cls} + L_{loc}$$

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum L_{cls}(p_i, p_i^*) + \frac{\lambda}{N_{loc}} \sum p_i^* \cdot L1_{smooth}(t_i - t_i^*)$$

The loss equation might look a little overwhelming at first. But it is actually simpler than it looks. Understanding the loss function is not necessary to be able to run and train Faster R-CNNs, so feel free to skip this section. But I encourage you to power through this explanation because it will add a lot of depth to your understanding on how the optimization process works under the hood. Let's go through the symbols first and explain it after.

Symbol	Explanation
p_i and p_i^*	p_i is the predicted probability of the anchor (i) being an object and the ground truth (0 or 1) of the anchor is an object.
t_i and t_i^*	t_i is the predicted four parameters that define the bounding box and t_i^* is the ground-truth parameters
N_{cls}	Normalization term for the classification loss. In the paper, it is set to be minibatch size ~256
N_{loc}	Normalization term for the bounding box regression. In the paper, it is set to the number of anchor locations ~2400
$L_{cls}(p_i, p_i^*)$	The log loss function over two classes, as we can easily translate a multi-class classification into a binary classification by predicting a sample being a target object versus not. $L_{cls}(p_i, p_i^*) = -p_i^* \log p_i - (1-p_i^*) \log (1-p_i)$
$L1_{smooth}$	As described in the Fast RCNN section, the bounding box loss measures the difference between the predicted and true location parameters (t_i, t_i^*) using the Smooth $L1$ Loss function. It is a robust function and is claimed to be less sensitive to outliers than other regression loss like $L2$.
λ	A balancing parameter, set to be ~10 in the paper (so that both L_{cls} and L_{loc} terms are roughly equally weighted).

Now that you know the definitions of the symbols, let's try to read the multi-task loss function again. To understand this equation let's go through some ideas to simplify it:

- Just for a moment, ignore the normalization terms and the (i) terms and let's look at the simplified loss function for each one instance (i):

$$L_{cls}(p_i, p_i^*) = -p_i^* \log p_i - (1-p_i^*) \log (1-p_i)$$

- The Location loss is the difference between the predicted and true location parameters (t_i, t_i^*) using the Smooth $L1$ Loss function. The difference is then multiplied by the ground truth probability of the region containing an object p^* . If it is not an object, $p^* = 0$ to eliminate the entire location loss for non-object regions.
- Finally we add the values of both losses to create the multi-loss function.

$$L = L_{cls} + L_{loc}$$

There you have it. That was the explanation of the multi-loss function for each instance (i). Put back the (i) and Σ symbols to calculate the summation of losses for each instance.

7.2.4 Recap of the RCNN family

Let's recap the evolution of the R-CNN architecture that we learned:

- **R-CNN:** bounding boxes are proposed by the “selective search” algorithm, each of which is warped and features are extracted via a deep convolutional neural network, such as AlexNet, before a final set of object classifications and bounding box predictions are made with linear SVMs and linear regressors.
- **Fast R-CNN:** a simplified design with a single model, a region-of-interest pooling layer is used **after** the CNN to consolidate regions and the model predicts both class labels and regions of interest directly.
- **Faster R-CNN:** a fully end-to-end deep learning object detector. Faster R-CNN replaced the Selective Search algorithm to propose regions of interest with a Region Proposal Network (RPN) that interprets features extracted from the deep CNN and learns to propose regions-of-interest directly.

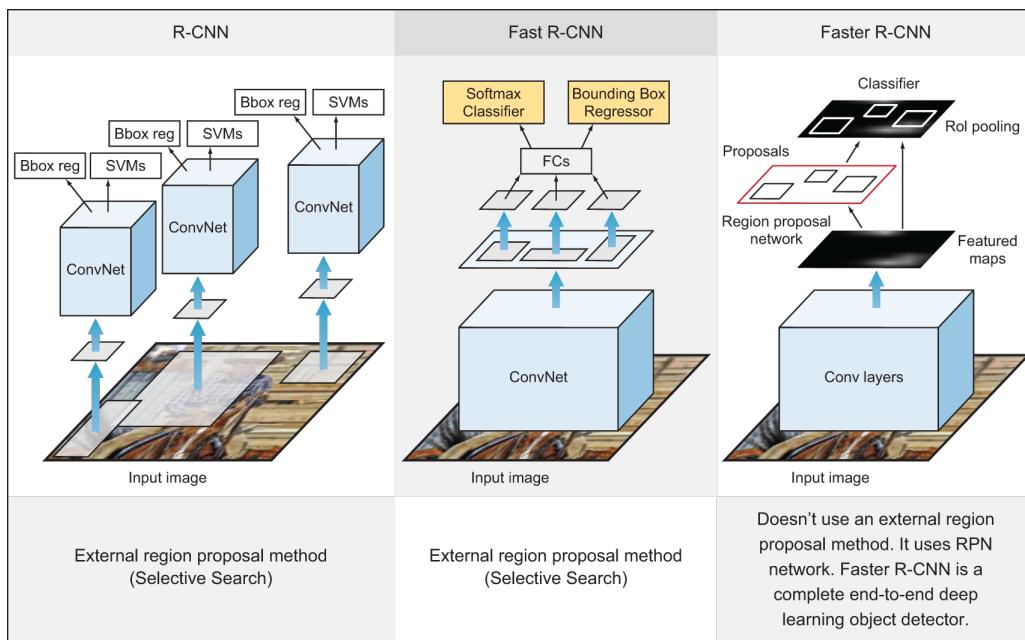


Figure 7.20: The evolution of the CNN family of networks from R-CNN to Fast R-CNN to Faster R-CNN.

Each image is passed only once to the CNN and feature maps are extracted.

1. A ConvNet is used to extract features maps from the input image.
2. Selective search is used on these maps to generate predictions.

This way we only run one ConvNet over the entire image instead of 2000 ConvNets over 2000 overlapping regions.

Table 7.2

	R-CNN	Fast R-CNN	Faster R-CNN
mAP on PASCAL VOC2007	66.0%	66.9%	66.9%
Features	<ol style="list-style-type: none"> 1. Applies selective search to extract regions of interest (~2,000) from each image. 2. A ConvNet is used to extract features from each of the 2,000 regions extracted 3. Classification and bounding box predictions. 	<p>Each image is passed only once to the CNN and feature maps are extracted.</p> <ol style="list-style-type: none"> 1. A ConvNet is used to extract features maps from the input image. 2. Selective search is used on these maps to generate predictions. <p>This way we only run one ConvNet over the entire image instead of 2000 ConvNets over 2000 overlapping regions.</p>	<p>Replaces the selective search method with region proposal network (RPN) which makes the algorithm much faster.</p> <p>An end-to-end deep learning network.</p>
Limitations	High computation time as each region is passed to the CNN separately. Also, it uses three different models for making predictions.	Selective search is slow and hence computation time is still high.	Object proposal takes time and as there are different systems working one after the other, the performance of systems depends on how the previous system has performed.
Test time per image	50 seconds	2 seconds	0.2 seconds
Speed-up from R-CNN	1x	25x	250x

R-CNN LIMITATIONS

As you might have noticed, each next paper proposes improvements to the seminal work done in R-CNN to develop a faster network, with the goal of achieving real-time object detection. The achievements displayed through this set of work is truly amazing, yet none of these

architectures manage to create a real-time object detector. Without going too much into details, the following problems with the above networks were identified:

- Training the data is unwieldy and too long
- Training happens in multiple phases (e.g. training region proposal vs classifier)
- Network is too slow at inference time

Fortunately, in the last few years, new architectures were created to address the bottlenecks of R-CNN and its successors, enabling real-time object detection. The most famous ones are [SSD](#) (Single Shot Detector) and [YOLO](#) (You Only Look Once) which we will explain next.

MULTI-STAGE VS. SINGLE-STAGE DETECTOR

Models in the R-CNN family are all region-based. The detection happens in two stages, thus called two-stage detectors:

1. First, the model proposes a set of regions of interest by Selective Search or Regional Proposal Network (RPN). The proposed regions are sparse as the potential bounding box candidates can be infinite.
2. Then a classifier only processes the region candidates.

One-stage detectors take a different approach. They skip the region proposal stage and run detection directly over a dense sampling of possible locations. This is faster and simpler, but might potentially drag down the performance a bit. In the next two sections, we will explain one-stage object detectors like Single-Shot Detection (SSD) and You Only Look Once (YOLO). In general, single-stage detectors tend to be less accurate than two-stage detectors but are significantly faster.

In the next section, we will explain SSD and should also find it easier to grasp YOLO once you understand SSD.

7.3 Single Shot Detection (SSD)

The paper about [SSD: Single Shot MultiBox Detector](#) was released in November 2016 by C. Szegedy et al. Single Shot Detection network reached new records in terms of performance and precision for object detection tasks, scoring over 74% mAP (mean Average Precision) at 59 frames per second (FPS) on standard datasets such as PascalVOC and MS COCO.

FPS to measure the detector speed

As discussed in the beginning of this chapter, the most common metric that is used to measure the detection speed is the number of frames per second (FPS). For example, Faster R-CNN operates at only 7 frames per second (FPS). There have been many attempts to build faster detectors by attacking each stage of the detection pipeline but so far, significantly increased speed comes only at the cost of significantly decreased detection accuracy. In this section you will see why single-stage networks like SSD can achieve faster detections that are more suitable for real-time detections.

For benchmarking, SSD300 achieves 74.3% mAP at 59 FPS while SSD512 achieves 76.8% mAP at 22 FPS, which outperforms Faster R-CNN (73.2% mAP at 7 FPS). SSD300 refers to the size of the input image of 300x300 and SSD512 refers to an input image of size = 512x512.

SSD IS A SINGLE-STAGE DETECTOR

We learned before that the R-CNN family is a multi-stage detector. Where the network first predict the objectness score of the bounding box then pass this box through a classifier to predict the class probability. In single-stage detectors like SSD and YOLO (explained later in this chapter), the convolutional layers make both predictions directly in one shot, hence the name Single Shot Detector. Where the image is passed once through the network and the objectness score for each bounding box is predicted using logistic regression to indicate the level of overlap with the ground truth. If the bounding box overlaps 100% with the ground truth, the objectness score is = 1 and if there is no overlap, the objectness score = 0. We then set a threshold value (0.5) that says: "if the objectness score is above 50%, then this bounding box likely has an object of interest and we get predictions, if it is less than 50%, we ignore the prediction."

7.3.1 High level SSD architecture

The single shot name was developed because SSD is a single-stage detector. It doesn't follow the RCNN approach of having two separate stages for each of regions proposal and detections. The SSD approach is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes, followed by a non-maximum suppression step to produce the final detections.

The architecture of the SSD model is composed of three main parts:

1. **Base network to extract feature maps** - a standard pretrained network used for high quality image classification and truncated before any classification layers. In their paper, C. Szegedy et al. used VGG16 network. Other networks like VGG19 and ResNet can be used and should produce good results.
2. **Multi-scale feature layers** - a series of convolution filters are added after the base network. These layers decrease in size progressively to allow predictions of detections at multiple scales.
3. **Non-maximum suppression** - to eliminate overlapping boxes and keep only one box for each object detected.

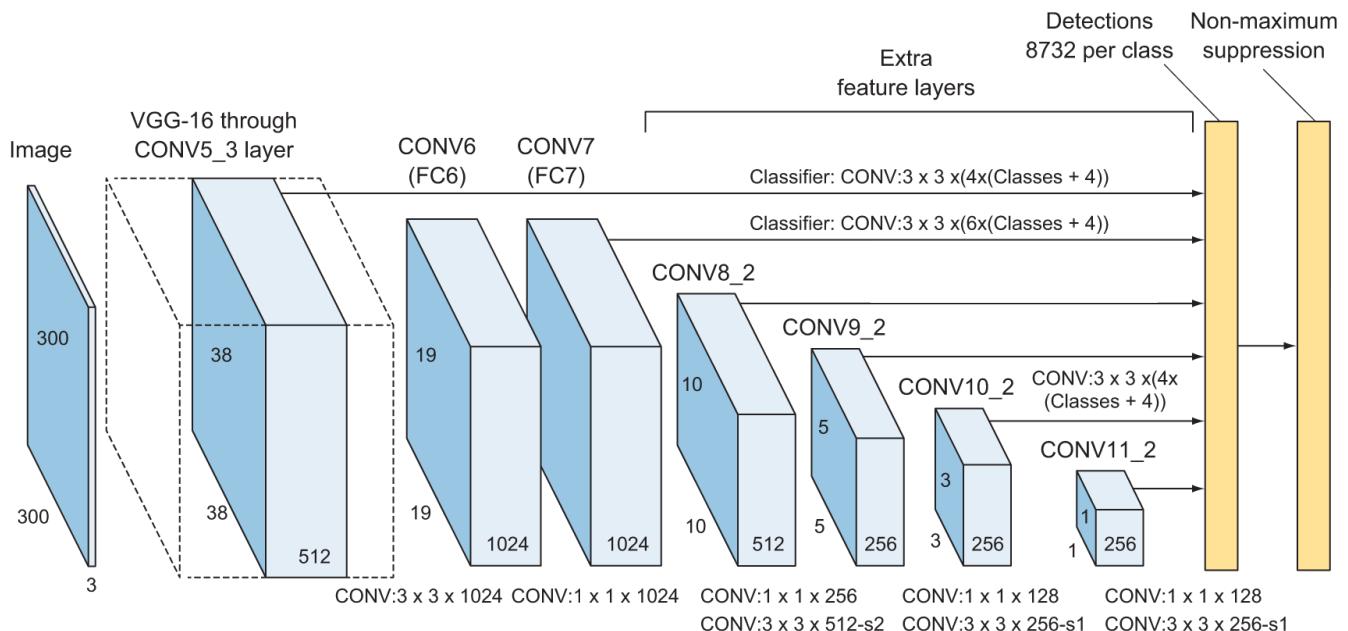


Figure 7.21: SSD architecture is composed of a base network (VGG16) + extra convolutional layers for object detection + a non-maximum suppression (NMS) layer for final detections (from the original paper). Note that the convolution layers 7, 8, 9, 10, and 11 make predictions that are directly fed to the NMS layer.

As you can see in the SSD architecture diagram (Figure 7.21), layers 4_3, 7, 8_2, 9_2, 10_2, and 11_2 make predictions directly to the Non-Maximum Suppression layer. We will talk about why these layers progressively decrease in size later in the “Multi-scales layer” section. For now, let’s follow along to understand the end-to-end flow of data in SSD.

You can see in the SSD architecture figure, that the network is making a total of 8,732 detections per class that are then fed to a non-max suppression layer to reduce down to one detection per object. Where did the number 8,732 come from?

To have more accurate detection, different layers of feature maps are also going through a small 3×3 convolution for object detection. For example, at Conv4_3, it is of size $38 \times 38 \times 512$. 3×3 CONV is applied. And there are 4 bounding boxes and each bounding box will have (number of classes + 4 box values) outputs. Suppose there are 20 object classes plus one background class, the output number of bounding boxes is $38 \times 38 \times 4 = 5,776$ bounding boxes. Similarly, we calculate the number of bounding boxes for the other conv layers:

- Conv7: $19 \times 19 \times 6 = 2,166$ boxes (6 boxes for each location)
- Conv8_2: $10 \times 10 \times 6 = 600$ boxes (6 boxes for each location)
- Conv9_2: $5 \times 5 \times 6 = 150$ boxes (6 boxes for each location)
- Conv10_2: $3 \times 3 \times 4 = 36$ boxes (4 boxes for each location)
- Conv11_2: $1 \times 1 \times 4 = 4$ boxes (4 boxes for each location)

If we sum them up, we got $5,776 + 2,166 + 600 + 150 + 36 + 4$. That is a total of 8,732 boxes are produced. This is a huge number of boxes to show for our detector. That's why we apply non-max suppression to reduce the number of the output boxes. As you will see when we discuss YOLO in the next section, in YOLO there are 7×7 locations at the end with 2 bounding boxes for each location. YOLO only got $7 \times 7 \times 2 = 98$ boxes.

What does the output prediction look like?

For each feature, the network predicts the following:

- 4 values that describe the bounding-box (x, y, w, h)
- + 1 value for the objectness score
- + C values that represent the probability of each class.

That's a total number of prediction values = $5 + C$. Suppose there are 4 object classes in our problem. Then each prediction will be a vector that looks like this: $[x, y, w, h, \text{Objectness Score}, C_1, C_2, C_3, C_4]$.

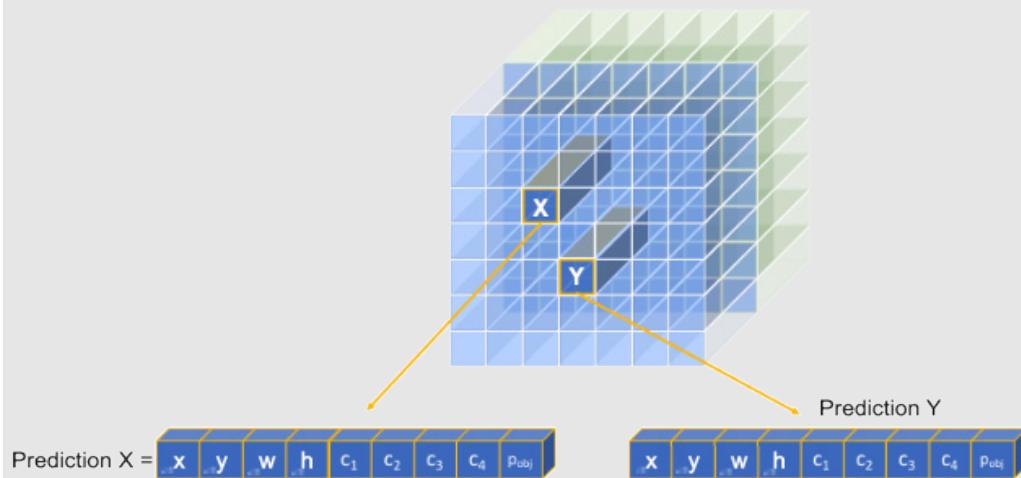


Figure 7.22: a conv layer makes predictions for each feature map. It predicts the bounding box coordinates + objectness score + four class probabilities: C_1, C_2, C_3 , and C_4 .

Now, let's dive a little deeper into each one of the 3 main components of the SSD architecture.

7.3.2 Base network

As you can see from the SSD diagram in figure 7.21, the SSD architecture builds on the VGG16 architecture after slicing off the fully connected classification layers (VGG16 is explained in detail in chapter 5). The reason VGG16 was used as the base network is because of its strong performance in high quality image classification tasks and its popularity for

problems where transfer learning helps in improving results. Instead of the original VGG fully connected layers, a set of supporting convolutional layers (from conv6 onwards) were added to enable to extract features at multiple scales and progressively decrease the size of the input to each subsequent layer.

Below is a simplified code implementation of the VGG16 network used in SSD using Keras. You will not need to implement this from scratch, my goal from putting this code snippet here is to show you that this is a typical VGG16 network that we implemented in chapter 5.

```
conv1_1 = Conv2D(64, (3, 3), activation='relu', padding='same')(x1)
conv1_2 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv1_1)
pool1 = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same')(conv1_2)

conv2_1 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool1)
conv2_2 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv2_1)
pool2 = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same')(conv2_2)

conv3_1 = Conv2D(256, (3, 3), activation='relu', padding='same')(pool2)
conv3_2 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv3_1)
conv3_3 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv3_2)
pool3 = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same')(conv3_3)

conv4_1 = Conv2D(512, (3, 3), activation='relu', padding='same')(pool3)
conv4_2 = Conv2D(512, (3, 3), activation='relu', padding='same')(conv4_1)
conv4_3 = Conv2D(512, (3, 3), activation='relu', padding='same')(conv4_2)
pool4 = MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same')(conv4_3)

conv5_1 = Conv2D(512, (3, 3), activation='relu', padding='same')(pool4)
conv5_2 = Conv2D(512, (3, 3), activation='relu', padding='same')(conv5_1)
conv5_3 = Conv2D(512, (3, 3), activation='relu', padding='same')(conv5_2)
pool5 = MaxPooling2D(pool_size=(3, 3), strides=(1, 1), padding='same')(conv5_3)
```

You've seen the VGG16 implemented in Keras in chapter 5. The two main takeaways from adding this here are:

1. Layer "4_3" will be used again to make direct predictions.
2. Layer "pool5" will be fed to the next layer (conv6) which is the first layer of the "multiscale features" layers.

HOW DOES THE BASE NETWORK MAKE PREDICTIONS?

Consider the following example. Suppose you have the image below (Figure 7.23) and the network's job is to draw bounding boxes around all the boats in the image. The process goes as follows:

1. Similar to the Anchors concept in R-CNN, SSD overlays a grid of Anchors around the image and for each anchor, the network will create bounding boxes at its center. In SSD, anchors are called **Priors**.
2. The base network looks at each bounding box as a separate image. Within each bounding box, the network will ask the question: is there a boat in this box? In other words, it will ask: did I extract any features of a boat in this box?
3. When the network finds a bounding box that contains boat features, it sends its coordinates prediction and object classification to the non-maximum suppression layer.

- Non-maximum suppression will then eliminate all the boxes except the one that overlaps the most with the ground truth bounding box.

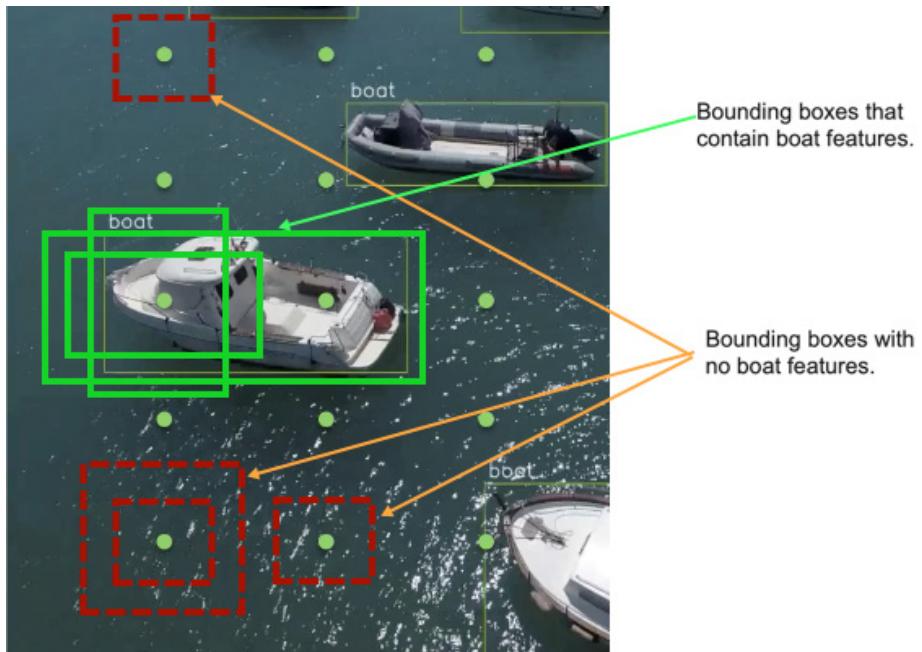


Figure 7.23: SSD Base Network looks at the anchor boxes to find features of a boat. Green (solid) boxes indicate that the network has found boat features. Red (dotted) boxes indicates no boat features.

A final note on the base network is that the authors here used VGG16 because of its strong performance in complex image classification tasks. You can use other networks like the deeper VGG19 or ResNet for the base network and it should perform as good if not better in accuracy but it could be slower if you chose to implement a deeper network. MobileNet is a good choice if you want to balance between a complex, high-performing deep network as well as being fast.

Now, on to the next component of the SSD architecture. The Multi-scale Feature Layers.

7.3.3 Multi-scale feature layers

They are convolutional feature layers that are added to the end of the truncated base network. These layers decrease in size progressively to allow predictions of detections at multiple scales.

MULTI-SCALE DETECTIONS

To understand the goal of the multi-scale features layers and why vary in size, let's look at the horses image in Figure 7.24 below:

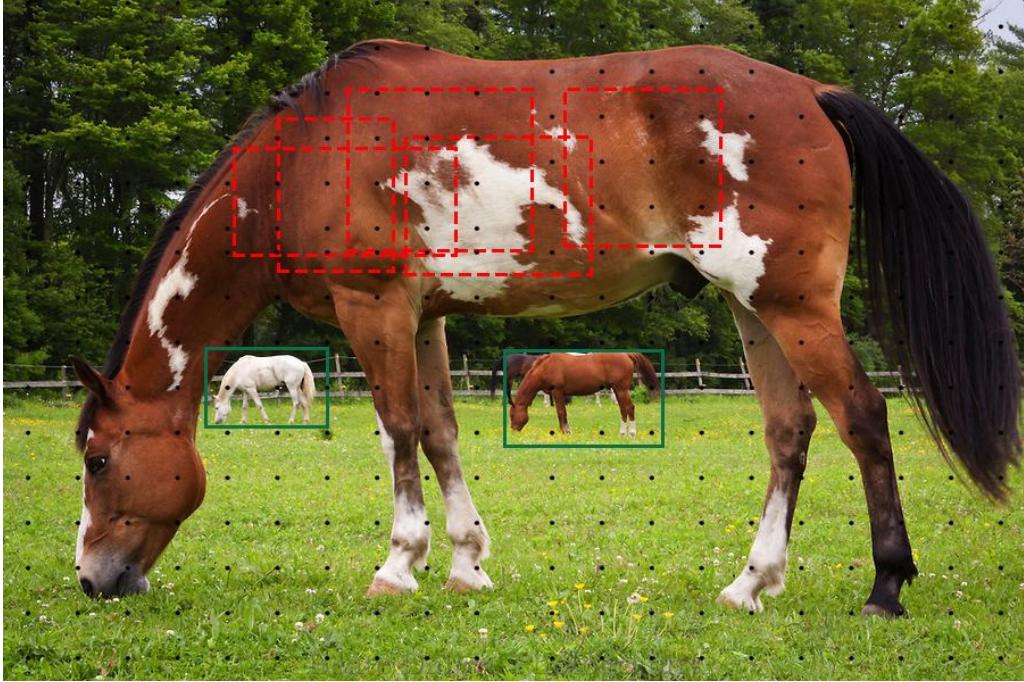


Figure 7.24: horses at different scales in an image.

As you can see, the base network might be able to detect the horses features in the background but it might fail to detect the horse that is closest to the camera. To understand why, take a close look at the dotted bounding box and try to imagine this box alone outside the context of the full image (see figure below).



Figure 7.25

Can you see horse features in this bounding box in Figure 7.25? No. To deal with objects of different scales in an image, some methods suggest preprocessing the image at different sizes and combining the results afterwards. However, by using different convolution layers that vary in size, we can utilize feature maps from several different layers in a single network for prediction we can mimic the same effect, while also sharing parameters across all object

scales. As CNN reduces the spatial dimension gradually, the resolution of the feature maps also decrease. SSD uses lower resolution layers to detect larger scale objects. For example, the 4×4 feature maps are used for larger scale object.

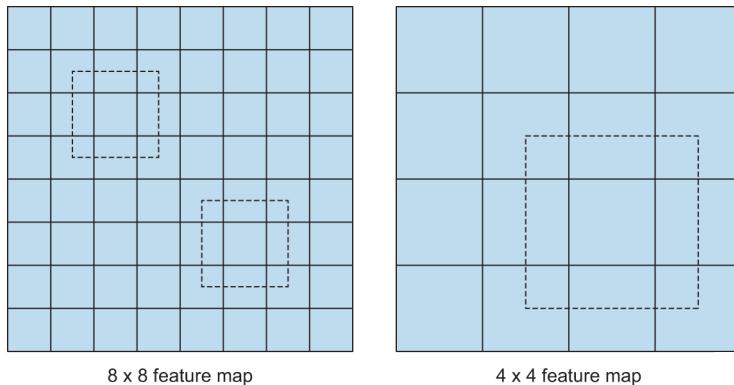


Figure 7.26: Right image - lower resolution feature maps detect larger scale objects. Left image - higher resolution feature maps detect smaller scale objects.

To visualize this, imagine that the network reduces the image dimensions to be able to fit the horses inside its bounding boxes. See Figure 7.27.

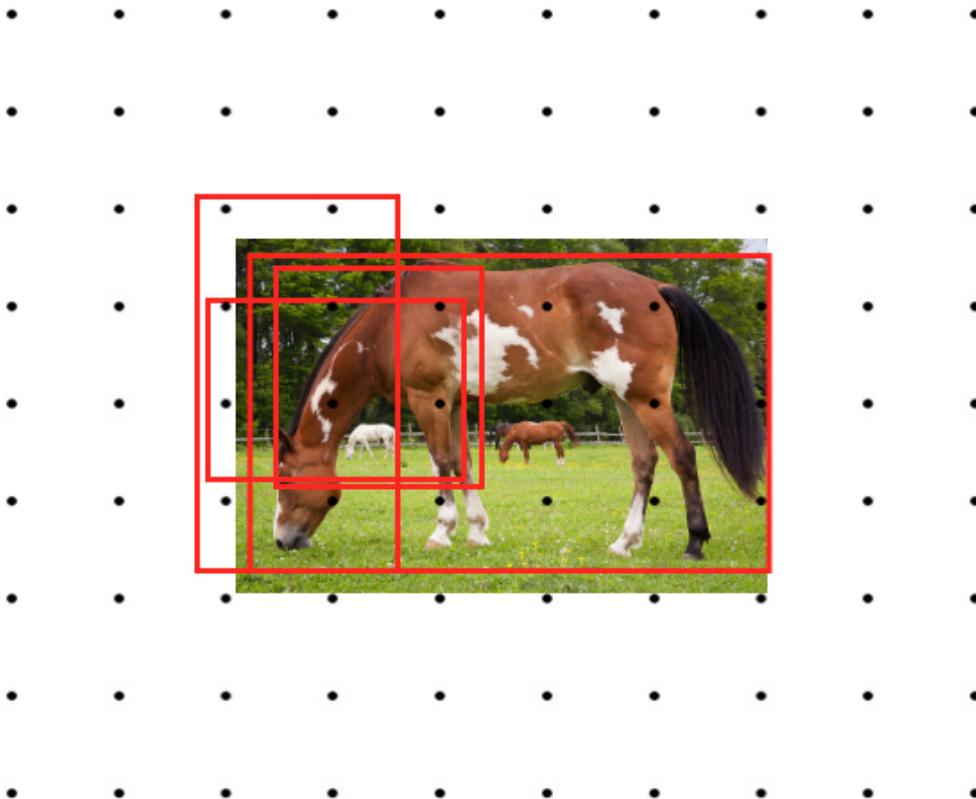


Figure 7.27: Multi-scale feature layers reduce the spatial dimensions of the input image to detect objects with different scales.

The multi-scale feature layers resize the image dimensions and keep the bounding boxes sizes so that they can fit the larger horse. In reality, convolutional layers do not literally reduce the size of the image. This is just for illustration to help us to intuitively understand the concept. In reality it is not just resized, it actually goes through the convolutional process so the image won't look anything like itself anymore. It will be a completely random looking image but it will preserve its features. The convolutional process is explained in detail in chapter 3.

Using a multi-scale feature maps improves the network accuracy significantly. The authors of the original paper ran an experiment to measure the advantage gained by adding the multi-scale feature layers. The table below shows a decrease in accuracy with less layers. Here is the accuracy with different number of feature map layers used for object detection.

Prediction source layers from:	mAP		# Boxes
	use boundary boxes?		
	Yes	No	
conv4_3 conv7 conv8_2 conv9_2 conv10_2 conv11_2			
✓ ✓ ✓ ✓ ✓ ✓	74.3	63.4	8732
✓ ✓ ✓ ✓ ✓	74.6	63.1	8764
✓ ✓ ✓ ✓ ✓	73.8	68.4	8942
✓ ✓ ✓ ✓	70.7	69.2	9864
✓ ✓ ✓	64.2	64.4	9025
✓ ✓	62.4	64.0	8664

Figure 7.28: effects of using multiple output layers from the original paper.

Notice that the network accuracy has dropped from 74.3% when having the prediction source from all the 6 layers to 62.4% for one source layer. When using only the conv7 layer for prediction, the performance is the worst, reinforcing the message that it is critical to spread boxes of different scales over different layers.

ARCHITECTURE OF THE MULTI-SCALE LAYERS

The authors decided to add 6 convolutional layers that are decreasing in size. This has been done by a lot of tuning and trial and error until they produced the best results. As you saw in the SSD architecture diagram (Figure 7.21), the convolutional layers 6 and 7 are pretty straight forward. Conv6 has a kernel size of 3x3 and conv7 has a kernel size of 1x1. Layers 8 through 11 on the other hand are treated more like blocks. Where each block consists of two convolutional layers of kernel sizes 1x1 and 3x3.

Look at the code implementation in Keras for layers 6 through 11:

```
# conv6 and conv7
conv6 = Conv2D(1024, (3, 3), dilation_rate=(6, 6), activation='relu', padding='same')(pool5)
conv7 = Conv2D(1024, (1, 1), activation='relu', padding='same')(conv6)
# conv8 block
conv8_1 = Conv2D(256, (1, 1), activation='relu', padding='same')(conv7)
conv8_2 = Conv2D(512, (3, 3), strides=(2, 2), activation='relu', padding='valid')(conv8_1)

# conv9 block
conv9_1 = Conv2D(128, (1, 1), activation='relu', padding='same')(conv8_2)
conv9_2 = Conv2D(256, (3, 3), strides=(2, 2), activation='relu', padding='valid')(conv9_1)

# conv10 block
conv10_1 = Conv2D(128, (1, 1), activation='relu', padding='same')(conv9_2)
conv10_2 = Conv2D(256, (3, 3), strides=(1, 1), activation='relu', padding='valid')(conv10_1)

# conv11 block
conv11_1 = Conv2D(128, (1, 1), activation='relu', padding='same')(conv10_2)
conv11_2 = Conv2D(256, (3, 3), strides=(1, 1), activation='relu', padding='valid')(conv11_1)
```

As mentioned before, if you are not working in research or academia, you will most probably won't need to implement object detection architectures yourself. In most cases, you will download an open source implementation and build on it to work on your problem. I just

added these code snippets to help you internalize the information discussed about different layers architectures.

Atrous (or Dilated) convolutions

Dilated convolutions introduce another parameter to convolutional layers called the dilation rate. This defines the spacing between the values in a kernel. A 3×3 kernel with a dilation rate of 2 will have the same field of view as a 5×5 kernel, while only using 9 parameters. Imagine taking a 5×5 kernel and deleting every second column and row.

This delivers a wider field of view at the same computational cost. Dilated convolutions are particularly popular in the field of real-time segmentation. Use them if you need a wide field of view and cannot afford multiple convolutions or larger kernels.

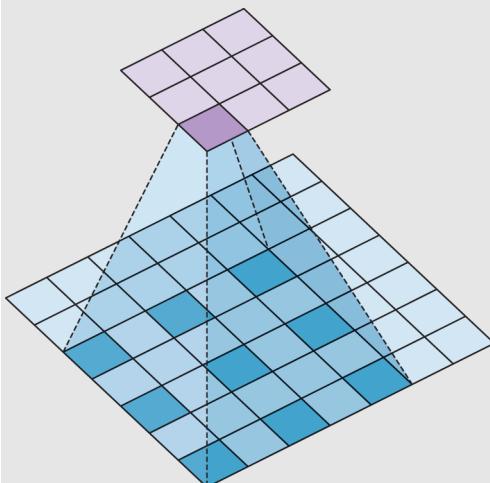


Figure 7.29: Dilated convolutions.

To build a dilated 3×3 convolution layer dilation rate = 2, using Keras:

```
Conv2D(1024, (3, 3), dilation_rate=(2,2), activation='relu', padding='same')
```

Next, we discuss the third and last component of the SSD architecture. The non-maximum suppression.

7.3.4 Non-maximum Suppression

Given the large number of boxes generated by the detection layer per class during a forward pass of SSD at inference time , it is essential to prune most of the bounding box by applying the non-maximum suppression technique (explained earlier in this chapter). Where boxes with a confidence loss and IoU less than a certain threshold are discarded, and only the top N predictions are kept. This ensures only the most likely predictions are retained by the network, while the more noisier ones are removed.

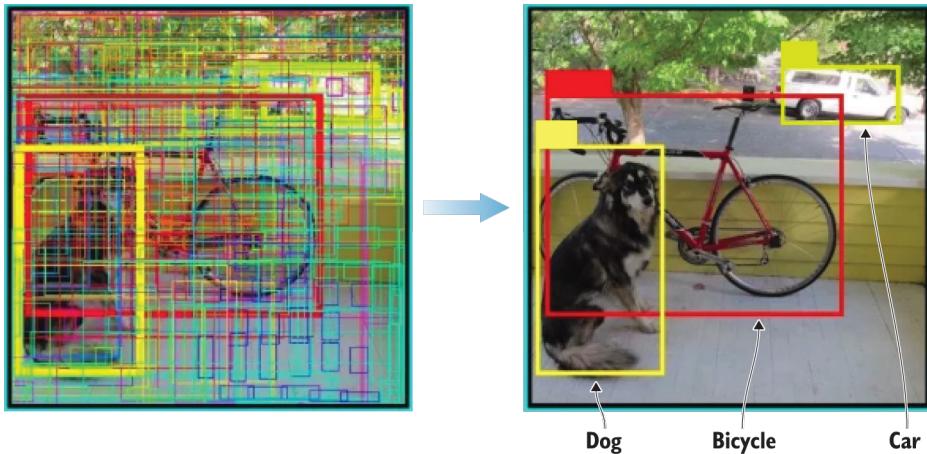


Figure 7.30: Non-maximum suppression reduces the number of bounding boxes to only one box for each object.

SSD sorts the predictions by the confidence scores. Start from the top confidence prediction, SSD evaluates whether any previously predicted boundary boxes have an IoU higher than 0.45 with the current prediction for the same class (the threshold value of 0.45 is set by the authors of the original paper). If found, the current prediction will be ignored. At most, we keep the top 200 predictions per image.

7.4 You Only Look Once (YOLO)

Similar to the R-CNN family, YOLO ("You Only Look Once") is a family of object detection networks that improved over the years through the following versions; [YOLOv1](#) published in 2016, [YOLOv2](#) (also known as YOLO9000) published later in 2016, and [YOLOv3](#) published in 2018. The YOLO family of models is a series of end-to-end deep learning models designed for fast object detection, developed by Joseph Redmon, et al. and is considered one of the first attempts to build a fast real-time object detector. It is one of the faster object detection algorithms out there. Though it is no longer the most accurate object detection algorithm, it is a very good choice when you need real-time detection, without loss of too much accuracy.

The creators of YOLO took a different approach than the previous networks. YOLO does not undergo the region proposal step like R-CNNs. Instead, it only predicts over a limited number of bounding boxes by splitting the input into a grid of cells and each cell directly predicts a bounding box and object classification. The result is a large number of candidate bounding boxes that are consolidated into a final prediction using non-maximum suppression.

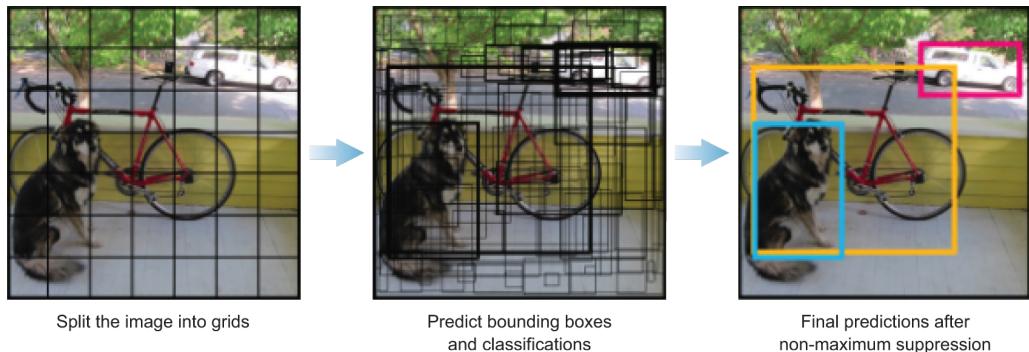


Figure 7.31: YOLO splits the image into grids, predicts objects for each grid, then use NMS to finalize predictions.

Although the accuracy of the models is close but not as good as Region-Based Convolutional Neural Networks (R-CNNs), they are popular for object detection because of their detection speed, often demonstrated in real-time on video or camera feed input.

At the time of writing, the YOLO family has three main variations that came out consequently to improve over the last version.

- YOLOv1: [YOLO Unified, Real-Time Object Detection](#). It is called unified because it is a single detection network that unifies the two components of a detector; object detector and class predictor.
- YOLOv2: [YOLO9000 - Better, Faster, Stronger](#) that is capable of detecting over 9,000 objects, hence the name YOLO9000. It has been trained on ImageNet and MS COCO datasets and has achieved 16% mean Average Precision (mAP) which is not quite good but it was very fast during test time.
- YOLOv3: [An Incremental Improvement](#). YOLOv3 is significantly larger than previous models and has achieved mAP of 57.9% which is the best results yet out of the YOLO family of object detectors.

The first version proposed the general architecture, whereas the second version refined the design and made use of predefined anchor boxes to improve bounding box proposal, and version three further refined the model architecture and training process. In this section we are going to focus on YOLOv3 because it is currently the state-of-the-art architecture in the YOLO family.

7.4.1 How YOLOv3 works

The YOLO network splits the input image into a grid of $S \times S$ cells. If the center of the ground truth box falls into a cell, that cell is responsible for detecting the existence of that object. Each grid cell predicts B number of bounding boxes and their objectness score along with their class predictions as follows:

1. **Coordinates of B bounding boxes** - similar to previous detectors, YOLO predicts 4 coordinates for each bounding box (b_x, b_y, b_w, b_h). Where x and y are set to be offset of a cell location.
2. **Objectness score** (P_0) - indicates the probability that the cell contains an object. The objectness score is passed through a sigmoid function to be treated as a probability with a value range between 0 and 1. The objectness score is calculated as follows:

$$P_0 = \text{Pr}(\text{containing an object}) \times \text{IoU}(\text{pred, truth})$$

3. **Class prediction** - if the bounding box contains an object, the network predicts the probability of K number of classes. Where K is the total number of classes in your problem.

It is important to note that before v3, YOLO used softmax function for the class scores. In v3 the authors have decided to use sigmoid instead. The reason is that Softmax imposes the assumption that each box has exactly one class which is often not the case. In other words, if an object belongs to one class, then it's guaranteed it cannot belong to another class. While this assumption is true for some datasets, it may not work when we have classes like Women and Person. A multilabel approach models the data more accurately. This is the reason that authors have steered clear of using a Softmax activation.

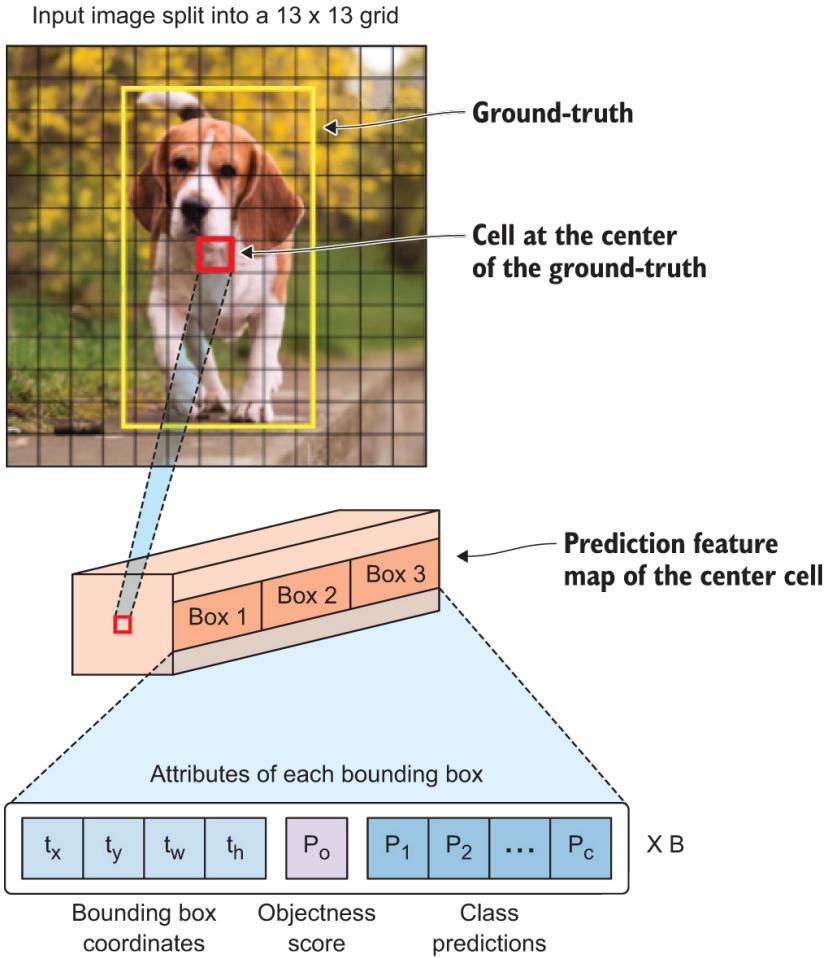


Figure 7.32: YOLOv3 workflow

As you can see in Figure 7.32, for each bounding box (B) the prediction looks like this: [(bounding box coordinates), (objectness score), (class predictions)]. We've learned that the bounding box coordinates = 4 values + 1 value for the objectness score, and K values for class predictions. Then the total number of values predicted for all bounding boxes (B) = $5B + K$ multiplied by the number of cells in the grid $S \times S$.

$$\text{Total predicted values} = S * S * (5B + K)$$

PREDICTIONS ACROSS DIFFERENT SCALES

Look closely at Figure 7.32. Notice that the prediction feature map has three boxes. You might have asked yourself, why are there three boxes? Similar to the "Anchors" concept in SSD,

YOLOv3 has 9 anchors to allow for prediction at 3 different scales per cell. The detection layer makes detection at feature maps of three different sizes, having strides 32, 16, and 8 respectively. This means, with an input image of size 416×416 , we make detections on scales 13×13 , 26×26 and 52×52 . The 13×13 layer is responsible for detecting large objects, the 26×26 layer is for detecting medium objects, and the 52×52 layer detects the smaller objects.

This results in the prediction of 3 bounding boxes for each cell ($B=3$). That's why in Figure 7.32 you see that the prediction feature map is predicting box1, box2 , and box3. The bounding box responsible for detecting the dog will be the one whose anchor has the highest IoU with the ground truth box.

Detections at different layers helps address the issue of detecting small objects, which was a frequent complaint with YOLOv2. The upsampling layers can help the network preserve and learn fine-grained features which are instrumental for detecting small objects.

The way the network do this is by downsampling the input image until the first detection layer, where a detection is made using feature maps of a layer with stride 32. Further, layers are upsampled by a factor of 2 and concatenated with feature maps of a previous layers having identical feature map sizes. Another detection is now made at layer with stride 16. The same upsampling procedure is repeated, and a final detection is made at the layer of stride 8.

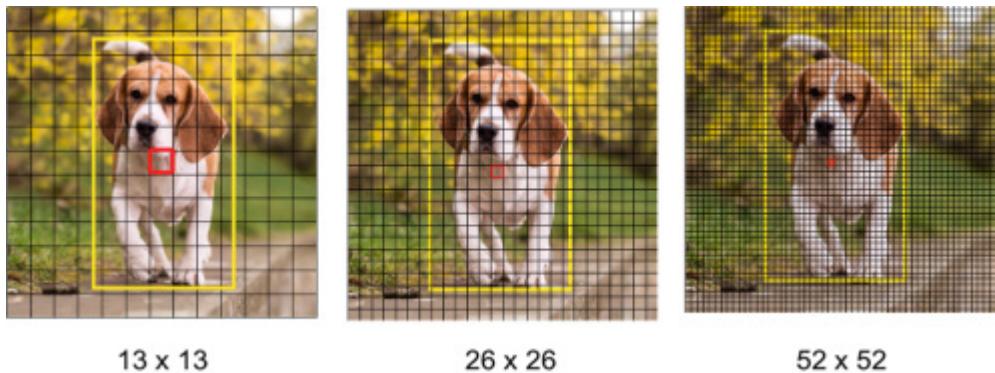


Figure 7.33: prediction feature maps at different scales

YOLOV3 OUTPUT BOUNDING BOXES

For an input image of size 416×416 , YOLO predicts $((52 \times 52) + (26 \times 26) + 13 \times 13) \times 3 = 10,647$ bounding boxes. That is a huge number of boxes for an output. In our dog example, we have only one object. We want only one bounding box around this object. How do we reduce the boxes down from 10,647 to 1?

First, we filter boxes based on their objectness score. Generally, boxes having scores below a threshold are ignored. Second, we use Non-maximum Suppression (NMS). NMS intends to cure the problem of multiple detections of the same image. For example, all the 3 bounding boxes of the red grid cell at the center of the image may detect a box or the adjacent cells may detect the same object.

7.4.2 YOLOv3 Architecture

Now that you understand how YOLO works, going through the architecture will be very simple and straightforward. YOLO is a single neural network that unifies object detection and classifications into one end-to-end network. The neural network architecture was inspired by the GoogLeNet model (Inception) for feature extraction. Instead of the inception modules used by GoogLeNet, YOLO uses 1×1 reduction layers followed by 3×3 convolutional layers. The authors called this **Darknet**.

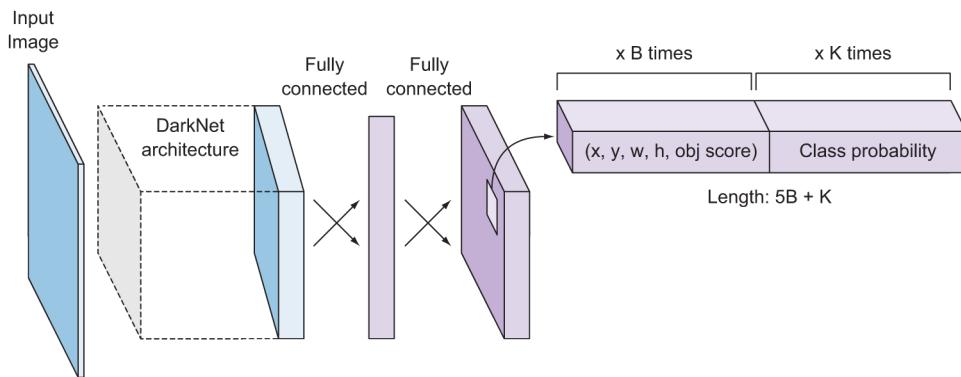


Figure 7.34: High level architecture of YOLO

YOLOv2 used a custom deep architecture darknet-19, an originally 19-layer network supplemented with 11 more layers for object detection. With a 30-layer architecture, YOLOv2 often struggled with small object detections. This was attributed to loss of fine-grained features as the layers downsampled the input. However, YOLO v2's architecture was still lacking some of the most important elements that are now stable in most state-of-the art algorithms. No residual blocks, no skip connections and no upsampling. YOLOv3 incorporates all of these updates.

YOLOv3 uses a variant of Darknet called Darknet-53. It has 53 layer network trained on Imagenet. For the task of detection, 53 more layers are stacked onto it, giving us a 106 layer fully convolutional underlying architecture for YOLOv3. This is the reason behind the slowness of YOLOv3 compared to YOLOv2. But this comes with a great boost in the detection accuracy.

Type	Filters	Size	Output
Convolutional	32	3×3	256×256
Convolutional	64	$3 \times 3 / 2$	128×128
1x	Convolutional	32	1×1
Convolutional	64	3×3	128×128
	Residual		
Convolutional	128	$3 \times 3 / 2$	64×64
2x	Convolutional	64	1×1
Convolutional	128	3×3	64×64
	Residual		
Convolutional	256	$3 \times 3 / 2$	32×32
8x	Convolutional	128	1×1
Convolutional	256	3×3	32×32
	Residual		
Convolutional	512	$3 \times 3 / 2$	16×16
8x	Convolutional	256	1×1
Convolutional	512	3×3	16×16
	Residual		
Convolutional	1024	$3 \times 3 / 2$	8×8
4x	Convolutional	512	1×1
Convolutional	1024	3×3	8×8
	Residual		
Avgpool		Global	
Connected		1000	
Softmax			

Figure 7.35: Darknet-53 feature extractor architecture from the YOLOv3 paper.

FULL ARCHITECTURE OF YOLOV3

We just learned that YOLOv3 makes predictions across three different scales. This gets a lot clearer now when you see the full architecture of YOLOv3 in Figure 7.35.

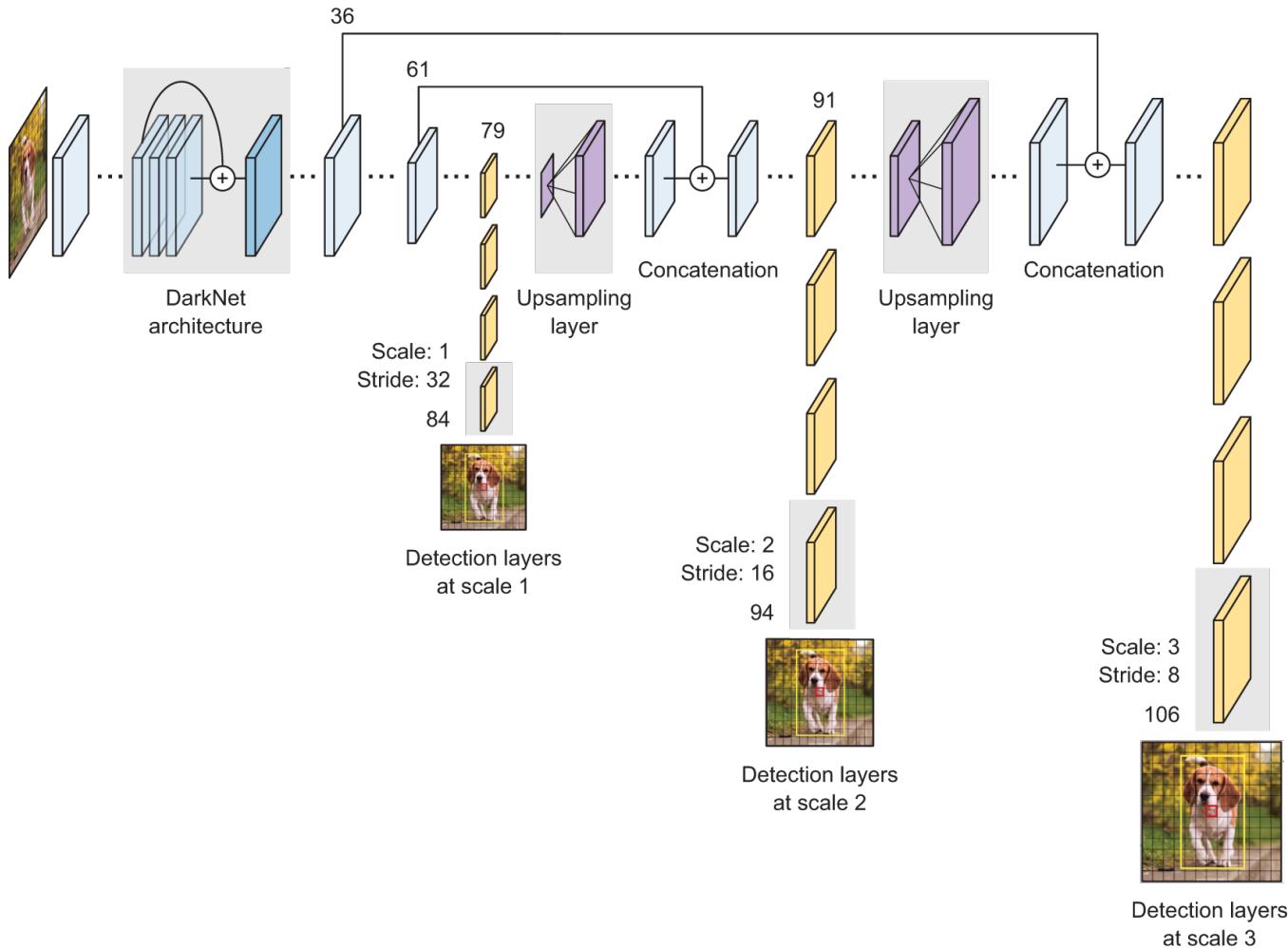


Figure 7.36: YOLOv3 network architecture. Inspired by the diagram in [this post](#).

The input image goes through Darknet-53 feature extractor then the image is downsampled by the network until layer 79. Then the network branches out and continues to downsample the image until it makes its first prediction at layer 82. This detection is made on a grid scale of 13x13 which is responsible for detecting large objects as we explained before.

Then the feature map from layer 79 is then **upsampled** by 2x to the dimensions of 26x26 and then **concatenated** with the feature map from layer 61. Then, the second detection is made by layer 94 on a grid scale of 26x26 which is responsible for detecting medium objects.

Finally, a similar procedure is followed again, where the feature map from layer 91 is subjected to few upsampling convolutional layers before being depth concatenated with a

feature map from layer 36. Then, a third prediction is made by layer 106 on a grid scale of 52x52 which is responsible for detecting small objects.

7.5 Chapter summary and takeaways

- Image classification is the task of predicting the type or class of an object in an image.
- Object detection is the task of predicting the location of objects in an image via bounding boxes and the classes of the located objects.
- The general framework of object detection systems consists of four main components: 1) region proposals, 2) feature extraction and predictions, 3) non-maximum suppression, and 4) evaluation metrics.
- Object detection algorithms are evaluated using two main metrics: 1) FPS (frame per second) to measure the network's speed, and 2) mAP (mean Average Precision) to measure the network's precision.
- The three most popular object detection systems are: 1) R-CNN family of networks, 2) SSD, and 3) YOLO family of networks.
- The R-CNN family of networks has three main variations: R-CNN, Fast R-CNN, and Faster R-CNN.
- R-CNN and Fast R-CNN used Selective Search algorithm to propose regions of interest (ROI). Whereas, Faster R-CNN is an end-to-end deep learning system that uses a Region Proposal Network (RPN) to propose ROI.
- R-CNN is a multi-stage detector where it separates the process to predict the objectness score of the bounding box and the object class into two different stages.
- SSD and YOLO are single-stage detectors where the image is passed once through the network to predict the objectness score and the object class.
- In general, single-stage detectors tend to be less accurate than two-stage detectors but are significantly faster.
- YOLO family of networks have three main networks: YOLOv1, YOLOv2 (or YOLO9000), and YOLOv3.

8

Generative Adversarial Networks (GANs)

"The desire to create is one of the deepest yearnings of the human soul."

– Dieter F. Uchtdorf

Learning goals from this chapter:

- What Are Generative Adversarial Networks?
- Understand the basic components of GANs: Generative and Discriminative models.
- Learn different techniques to evaluate generative models
- Learn how to build a GAN model

Generative Adversarial Networks (GANs) are a new type of neural architectures introduced by Ian Goodfellow and other researchers at the University of Montreal, including Yoshua Bengio, in June 2014, in their paper "[Generative Adversarial Nets](#)" in 2014. GANs have been called "the most interesting idea in the last 10 years in ML" by Yann LeCun, Facebook's AI research director. The excitement is well justified. The most notable features of GANs are their capacity to create hyperrealistic images, videos, music, and text. For example, none of the faces in Figure 8.1 (right) belong to a real human; they are all fake. Same thing for the handwritten digits in the left image in the same figure. This shows GAN's ability to learn the features from the training images and imagine its own new images using these patterns learned.

We've learned in the past chapters how deep neural networks can be used to understand image features and perform deterministic tasks on them like object classification and detection. In this section of the book we will talk about a different type of applications for deep learning in the computer vision world, called *Generative Models*. They are neural network

models that are able to imagine and produce new content that hasn't been created before. They can imagine new worlds, new people, and new reality in a seemingly magical way. We train generative models by providing a training dataset in a specific domain and their job is to create images that have new objects from the same domain that look so close to the real data.

To illustrate this notion of "generative models", we can take a look at some well known examples of results obtained with GANs in Figure 8.1.

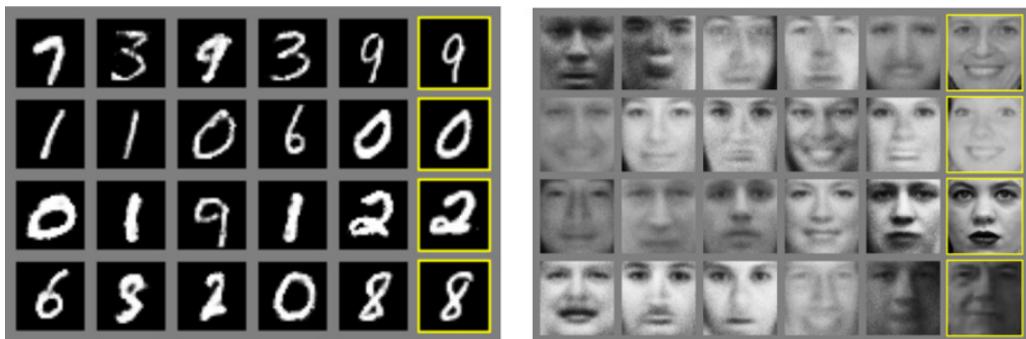


Figure 8.1: Illustration of GANs abilities by Ian Goodfellow and co-authors. These are samples generated by Generative Adversarial Networks after training on two datasets: MNIST and TFD. For both, the rightmost column contains true data. This shows us that the produced data are really generated and not only memorised by the network. (source: "Generative Adversarial Nets" paper).

For long humans have had the advantage ability over computers to imagine and create. Computers have excelled in solving problems like Regression, Classification, and Clustering but with the introduction of generative networks, researchers were able to make computers generate content of the same or higher quality compared to their human counterparts. By allowing computers to learn to mimic any distribution of data, they can be taught to create worlds that are similar to our own in any domain: images, music, speech, prose. They are robot artists in a sense, and their output is impressive. GANs are also seen as an important stepping stone toward achieving artificial general intelligence (AGI), an artificial system capable of matching human cognitive capacity to acquire expertise in virtually any domain - from images, to language, to creative skills needed to compose sonnets.

Naturally, this ability to generate new content makes GANs look a little bit like "magic", at least at first sight. In this chapter, we will only attempt to scratch the surface of what is possible with GANs. We will overcome the apparent magic of GANs in order to dive into the architecture ideas, maths and modelling behind these models in hope that this chapter will provide you with the necessary theoretical knowledge and practical skills to continue exploring any facet of this field that you find most interesting. Not only we will discuss the fundamental notions Generative Adversarial Networks rely on but, more, we will implement and train an end-to-end GAN and go through it step by step.

Ready? Let's get started!

8.1 GANs Architecture

GANs are based on the idea of adversarial training. They basically consist of two neural networks which compete against each other. This competitiveness helps them to mimic any distribution of data. I like to think of the GAN architecture as two boxers fighting. In their conquest to win the game, both are learning each others' moves and techniques. They both start the game with less knowledge about their opponent. As the game goes, they *learn* and become better and better.

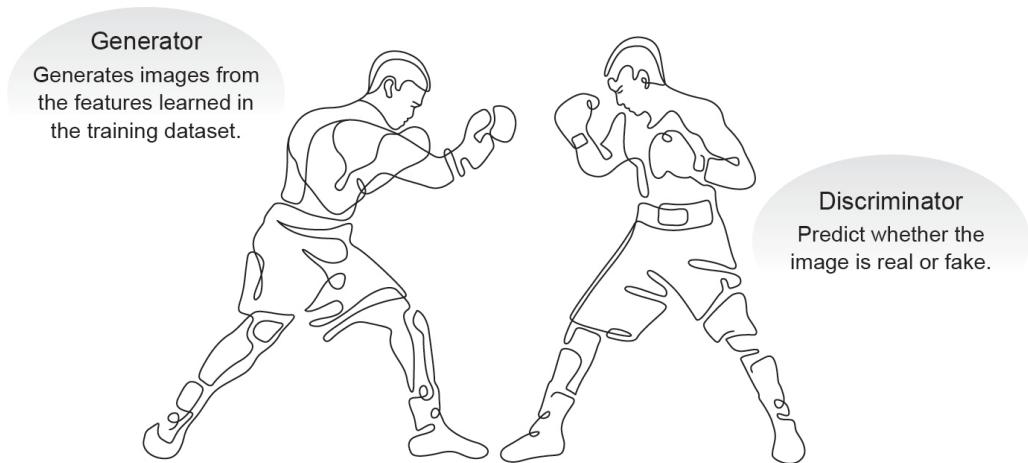


Figure 8.2: A fight between two adversary networks, generative and discriminative networks.

Another analogy to help drive the idea home: think of GANs as the opposition of a counterfeiter and a cop in a game of cat and mouse, where the counterfeiter is learning to pass false notes, and the cop is learning to detect them. Both are dynamic. Meaning, as the counterfeiter is learning to perfect creating false notes, the cop is in training and getting better too and each side comes to learn the other's methods in a constant escalation.



Figure 8.3: The GAN's generator and discriminator models are like a counterfeiter and a police officer.

The GAN architecture is composed of two main networks:

4. **The Generator** tries to convert random noise into observations that look as if they have been sampled from the original dataset.
5. **The Discriminator** tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries.

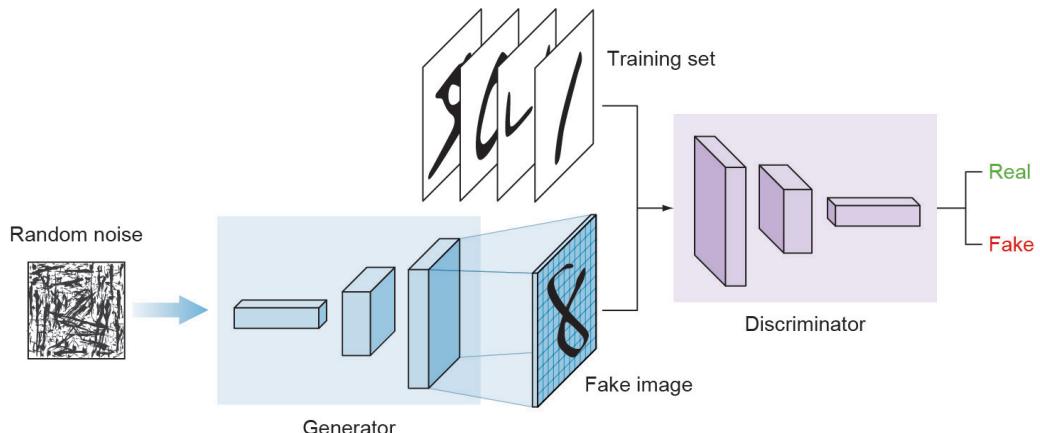


Figure 8.4: The GAN architecture is composed of Generator and Discriminator networks. Note that the Discriminator network is a typical CNN where the convolutional layers reduce in size until they get to the flattened layer. The Generator network on the other hand is an inverted CNN that starts with the flattened vector and the conv layers increase in size until they form the dimension of the input images.

As you can see in the architecture diagram (Figure 8.4), the steps a GAN takes are as follows:

6. The generator takes in random numbers and returns an image.
7. This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.
8. The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.

If you take a close look at the generator and discriminator networks, you will notice that the generator network is an inverted ConvNet that starts with the flattened vector and then the images are upscaled until they have a similar size to the images in the training dataset. We will dive deeper into the generator architecture later in this chapter but I just wanted you to notice this phenomenon now.

DEEP CONVOLUTIONAL GANS (DCGANs)

In the original GAN paper in 2014, multi-layer perceptron (MLP) networks were used to build the generator and discriminator networks. However, since then, it has been proven that convolutional layers give greater predictive power to the discriminator which in turn enhances the accuracy of the generator and the overall model. This type of GAN is called a DCGAN (deep convolutional GAN). DCGANs were developed by Alec Radford, et al. in 2016 in their

paper “[Unsupervised representation learning with deep convolutional generative adversarial networks](#)”. Now, all GAN architectures contain convolutional layers, so the “DC” is implied when we talk about GANs and we will be referring to DCGANs for the rest of the chapter as both GAN and DCGAN. You can also go back to chapters 2 and 3 in this book to learn more about the differences between MLP and CNN networks and why CNN is preferred for image problems.

Now, let’s dive deeper into the architecture of the discriminator and generator networks.

8.1.1 The Discriminator Model

As explained earlier, the goal of the discriminator is to predict if an image is real or fake. This is a typical supervised classification problem, so we can use the traditional classifier network that we learned in the previous chapters of this book. The network consists of stacked convolutional layers, followed by a dense output layer with sigmoid activation function. We use sigmoid activation function because this is a binary classification problem and the goal of the network is to output prediction probabilities values that range between 0 and 1. Where, 0 means that the image generated by the generator is fake and 1 means that it is 100% real.

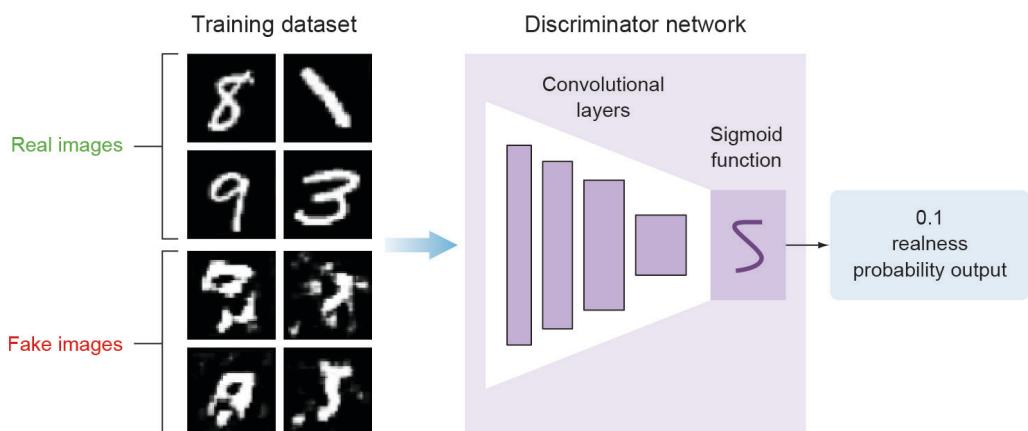


Figure 8.5: The discriminator for the GAN

The discriminator is a normal and well understood classification model. As you can see in Figure 8.5, training the discriminator is pretty straight forward. As a supervised classification problem, we feed the discriminator with labeled images: fake (or generated) and real images. The real images come from the training dataset and the fake images are the output of the generator model.

Now, let’s implement the discriminator network in Keras. At the end of this chapter, we will compile all the code snippets together to build an end-to-end GAN. So, let’s implement a `discriminator_model` function. In this code snippet, we will use the shape of an image input = 28×28 . You should change that to your own problem.

```
def discriminator_model():
```

```

# instantiate a sequential model and name it discriminator
discriminator = Sequential()
# add a convolutional layer to the discriminator model
discriminator.add(Conv2D(32, kernel_size=3, strides=2, input_shape=(28,28,1),
                       padding="same"))
# add a LeakyReLU activation function
discriminator.add(LeakyReLU(alpha=0.2))
# add a dropout layer with a 25% dropout probability
discriminator.add(Dropout(0.25))
# add a second convolutional layer with zero padding
discriminator.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
discriminator.add(ZeroPadding2D(padding=((0,1),(0,1))))
# add a BatchNormalization layer for faster Learning and higher accuracy
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))

# add a third convolutional layer with batch norm, LeakyReLU, and a dropout
discriminator.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))

# add the fourth convolutional layer with batch norm, LeakyReLU, and a dropout
discriminator.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))

# flatten the network and add the output Dense layer with sigmoid activation function
discriminator.add(Flatten())
discriminator.add(Dense(1, activation='sigmoid'))

# print the model summary
discriminator.summary()

# set the input image shape
img = Input(shape=img_shape)
# run the discriminator model to get the output probability
probability = discriminator(img)

# return a Model that takes the image as an input and produces the probability output
return Model(img, probability)

```

As you might have noticed, there is nothing new in the discriminator model. It follows the regular pattern of the traditional CNN networks that we learned in chapters 3, 4 and 5. We stack convolutional, batch norm, activation, and dropout layers to create our model. All these layers have their hyperparameters that we tune when we are training the network. For your own implementation, you can tune these hyperparameters and add or remove layers as you see fit. Tuning CNN hyperparameters is explained in great detail in chapters 3 and 4.

The output summary of the discriminator model is shown in Figure 8.6.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 14, 14, 32)	320
leaky_re_lu_1 (LeakyReLU)	(None, 14, 14, 32)	0
dropout_1 (Dropout)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 7, 7, 64)	18496
zero_padding2d_1 (ZeroPadding2D)	(None, 8, 8, 64)	0
batch_normalization_1 (BatchNormalization)	(None, 8, 8, 64)	256
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 64)	0
dropout_2 (Dropout)	(None, 8, 8, 64)	0
conv2d_3 (Conv2D)	(None, 4, 4, 128)	73856
batch_normalization_2 (BatchNormalization)	(None, 4, 4, 128)	512
leaky_re_lu_3 (LeakyReLU)	(None, 4, 4, 128)	0
dropout_3 (Dropout)	(None, 4, 4, 128)	0
conv2d_4 (Conv2D)	(None, 4, 4, 256)	295168
batch_normalization_3 (BatchNormalization)	(None, 4, 4, 256)	1024
leaky_re_lu_4 (LeakyReLU)	(None, 4, 4, 256)	0
dropout_4 (Dropout)	(None, 4, 4, 256)	0
flatten_1 (Flatten)	(None, 4096)	0
dense_1 (Dense)	(None, 1)	4097
<hr/>		
Total params: 393,729		
Trainable params: 392,833		
Non-trainable params: 896		

Figure 8.6: The output summary of the discriminator model

From the output summary in Figure 8.6, note that the width and height of the output feature maps are decreasing in size whereas the depth is increasing in size. This is the expected behavior for traditional CNN networks as we've seen in the previous chapters. Let's see what happens to the feature maps size in the generator network in the next section.

8.1.2 The Generator Model

The generator takes in some random data and try to mimic the training dataset to generate fake images. Its goal is to trick the discriminator by trying to generate images that are perfect replicas from the training dataset. As it gets trained it gets better and better after each iteration. On the other hand, the discriminator is being trained at the same time so the generator has to keep improving as the discriminator learns its tricks.

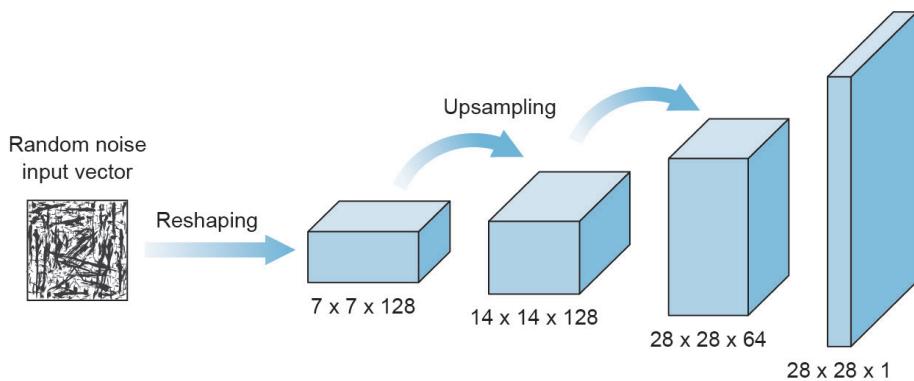


Figure 8.7: The generator model of the GAN.

As you might have noticed from Figure 8.7, the generator model looks like an inverted ConvNet. The generator takes a vector input with some random noise data, reshapes it to cube volume that has a width, height and depth. This volume is meant to be treated as a feature map that will be fed to several convolutional layers that will create the final image.

UPSAMPLING TO SCALE FEATURE MAPS

Just like traditional convolutional neural networks use pooling layers to downsample input images. In order to scale the feature maps, we use upsampling layers that scales the image dimensions by repeating each row and column of its input pixels.

Keras has an upsampling layer (`UpSampling2D`) that scales the image dimensions by taking a scaling factor (size) as an argument.

```
keras.layers.UpSampling2D(size=(2, 2))
```

What this line of code does is that it repeats every row and column of the image matrix two times because the size of the scaling factor is set to $(2, 2)$. See Figure 8.8 for an example.

```

Input = 1, 2
        3, 4
        1, 1, 2, 2
Output = 1, 1, 2, 2
        3, 3, 4, 4
        3, 3, 4, 4

```

Figure 8.8: upsampling example when scaling size = (2, 2)

If the scaling factor = (3, 3), the upsampling layer will repeat each row and column of the input matrix three times.

```

[[1. 1. 1. 2. 2. 2.]
 [1. 1. 1. 2. 2. 2.]
 [1. 1. 1. 2. 2. 2.]
 [3. 3. 3. 4. 4. 4.]
 [3. 3. 3. 4. 4. 4.]
 [3. 3. 3. 4. 4. 4.]]

```

Figure 8.9: upsampling example when scaling size = (3, 3)

When we build the generator model, we keep adding upsampling layers until the size of the feature maps is similar to the training dataset. You will see how this is implemented in Keras in the next section.

Now, let's build the `generator_model` function that builds the generator network.

```

def generator_model():
    # instantiate a sequential model and name it generator
    generator = Sequential()
    # add the Dense Layer that has a number of neurons = 128x7x7
    generator.add(Dense(128 * 7 * 7, activation="relu", input_dim=100))
    # reshape the image dimensions to 7 x 7 x 128
    generator.add(Reshape((7, 7, 128)))
    # upsampling layer to double the size of the image dimensions to 14 x 14
    generator.add(UpSampling2D(size=(2,2)))

    # add a convolutional Layer to run the convolutional process + batchNorm
    generator.add(Conv2D(128, kernel_size=3, padding="same", activation="relu"))
    generator.add(BatchNormalization(momentum=0.8))
    # upsample the image dimensions to 28 x 28
    generator.add(UpSampling2D(size=(2,2)))

    # conv + batchNorm layers
    # notice that we didn't add upsampling here because we have the image size of 28 x 28 which
    # is equal to the image size in the MNIST dataset. You can adjust this for your own
    # problem.
    generator.add(Conv2D(64, kernel_size=3, padding="same", activation="relu"))
    generator.add(BatchNormalization(momentum=0.8))

    # convolutional Layer with filters = 1
    generator.add(Conv2D(1, kernel_size=3, padding="same", activation="relu"))

```

```

# print the model summary
generator.summary()

# generate the input noise vector of Length = 100
# we chose 100 here to create a simple network
noise = Input(shape=(100,))
# run the generator model to create the fake image
fake_image = generator(noise)
# return a model that takes the noise vector as an input and outputs the fake image
return Model(noise, fake_image)

```

In the generator code snippet, we don't use any components that you are not already familiar with. The only new component is the `Upsampling` layer to double its input dimensions by repeating pixels. Similar to the discriminator, we stack convolutional layers on top of each other and add other optimization layers like `BatchNormalization`. The key difference in the generator model is that it starts with the flattened vector then images are upsampled until they have similar dimensions to the training dataset. All these layers have their hyperparameters that we tune when we are training the network. For your own implementation, you can tune these hyperparameters and add or remove layers as you see fit. Tuning CNN hyperparameters is explained in great detail in chapters 3 and 4.

The output summary of the generator model is shown in Figure 8.10.

Layer (type)	Output Shape	Param #
<hr/>		
dense_2 (Dense)	(None, 6272)	633472
reshape_1 (Reshape)	(None, 7, 7, 128)	0
up_sampling2d_1 (UpSampling2	(None, 14, 14, 128)	0
conv2d_5 (Conv2D)	(None, 14, 14, 128)	147584
batch_normalization_4 (Batch	(None, 14, 14, 128)	512
activation_1 (Activation)	(None, 14, 14, 128)	0
up_sampling2d_2 (UpSampling2	(None, 28, 28, 128)	0
conv2d_6 (Conv2D)	(None, 28, 28, 64)	73792
batch_normalization_5 (Batch	(None, 28, 28, 64)	256
activation_2 (Activation)	(None, 28, 28, 64)	0
conv2d_7 (Conv2D)	(None, 28, 28, 1)	577
activation_3 (Activation)	(None, 28, 28, 1)	0
<hr/>		
Total params:	856,193	
Trainable params:	855,809	
Non-trainable params:	384	

Figure 8.10: The output summary of the generator model

Notice that change in the output shape after each layer. It starts from a 1D vector of 6,272 neurons then we reshaped it to 7x7x128 volume then the width and height have been upsampled twice to 14x14 then 28x28. The depth decreases from 128 to 64 to 1 because this network is built to deal with the gray scale mnist dataset project that we will implement in this chapter. If you are building a generator model to generate color images, then you would want to make the filters in the last conv layer = 3.

8.1.3 Training the GAN

Now that we've learned the discriminator (D) and generator (G) models separately, let's put them together to train an end-to-end generative adversarial network. The discriminator is being trained to become a better classifier to maximize the probability of assigning the correct label to both training examples (real) and images generated (fake) by the generator (i.e the policeman becomes better at differentiating between fakes and real dollar signs). The generator on the other hand is being trained to become a better forger to maximize its chances to fool the discriminator. Both networks are getting better at what they do.

The process of training GAN models involves two training processes:

9. Train the discriminator: this is a straightforward supervised training process. Where the network is given labeled images coming from the generator (fake) and the training data (real) and it learns to classify between real and fake images with a sigmoid prediction output. Nothing new here.
10. Train the generator: this process is a little tricky. The generator model cannot be trained alone like the discriminator. It needs the discriminator model to tell it whether it did a good job faking images or not. So we create a **combined network** to train the generator that is composed of both G and D models.

Think of the two training processes as two parallel lanes. One lane training the discriminator alone, and the other lane is the combined model that trains the generator. The GAN training process is illustrated in Figure 8.11.

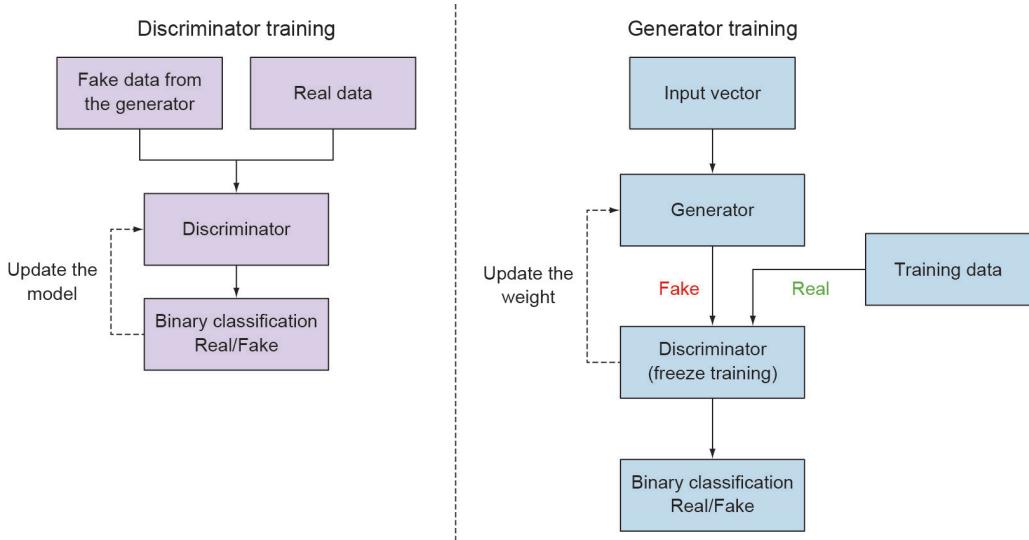


Figure 8.11: The process flow to train GANs.

As you can see in Figure 8.11, when training the combined model, we freeze the weights of the discriminator because this model focuses only on training the generator. We will discuss the intuition of this idea when we explain the generator training process in the next section. For now, just know that we need to build and train two models, one for the discriminator alone and the other one is for both G and D models.

Both processes follow the traditional neural network training process that was explained in chapter 2. It starts with the feedforward process, then make predictions, calculate the error and backpropagate the error. When training the discriminator, the error is backpropagated back to the discriminator model to update its weights whereas in the combined model, the error is backpropagated back to the generator to update its weights.

During the training iterations, we follow the same neural network training procedure to observe its performance and tune its hyperparameters. Until we notice that the generator is achieving satisfying results to our problem. This is when we can stop the training and deploy the generator model.

Now, let's see how we compile the discriminator and the combined networks to train the GAN model.

1.3.1. TRAIN THE DISCRIMINATOR

As we said before, this is a straightforward process. First, we build the model from the `discriminator_model` method that we created earlier in this chapter. Then compile the model and use `binary_crossentropy` loss function and an optimizer of your choice (we will use `Adam` in this example).

Let's see the Keras implementation. Please note that this code snippet is not meant to be compilable on its own. It is here for illustration. At the end of this chapter, you can find the full code of this project.

```
# Build and compile the discriminator
discriminator = discriminator_model()
discriminator.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

We can train the model by creating random training batches using Keras' `train_on_batch` method to run a single gradient update on a single batch of data.

```
# Sample noise and generate a batch of new images
noise = np.random.normal(0, 1, (batch_size, 100))
gen_imgs = generator.predict(noise)

# Train the discriminator (real classified as ones and generated as zeros)
d_loss_real = discriminator.train_on_batch(imgs, valid)
d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
```

1.3.2. TRAIN THE GENERATOR (COMBINED MODEL)

Here is the one tricky part in training GANs, training the generator. While the discriminator can be trained in isolation from the generator model, the generator needs the discriminator to be trained. For this, we build a **combined model** that contains both the generator and the discriminator as you can see in Figure 8.12.

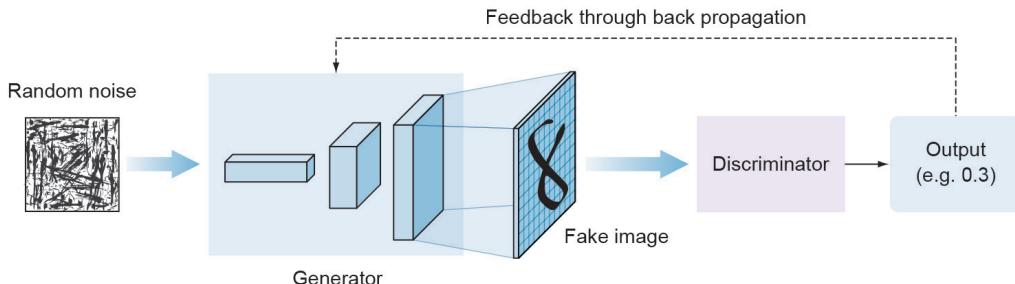


Figure 8.12: illustration of the combined model that contains both the generator and discriminator models.

When we want to train the generator, we freeze the weights of the discriminator model because the generator and discriminator have different loss functions pulling in different directions. If we don't freeze the discriminator weights, it will be pulled in the same direction the generator is learning so it will be more likely to predict generated images as real, which is not the desired outcome. Freezing the weights of the discriminator model doesn't affect the existing discriminator model that we compiled earlier when we were training the discriminator. Think of it as that we have two discriminator models which is not the case, but just easier to imagine.

In figure 8.11., we see that we compile the discriminator model twice:

11. Once when training the discriminator. In which, the error is backpropagated to the discriminator model to update its weights, and

12.the second time when we build the combined model. In which, we freeze the weights of the discriminator model and the error is backpropagated to the generator model to update its weights.

Now let's build the combined model:

```
# Build the generator
generator = generator_model()

# The generator takes noise as input and generates image
z = Input(shape=(100,))
image = generator(z)

# freeze the weights of the discriminator model
discriminator.trainable = False

# The discriminator takes generated images as input and determines validity
valid = discriminator(image)

# The combined model (stacked generator and discriminator)
# Trains the generator to fool the discriminator
combined = Model(z, valid)
```

Now that we have built the combined model, we can proceed with the training process as our normal process. We compile the combined model with a `binary_crossentropy` loss function and an optimizer `Adam`.

```
combined.compile(loss='binary_crossentropy', optimizer=optimizer)
# Train the generator (wants discriminator to mistake images as real)
g_loss = self.combined.train_on_batch(noise, valid)
```

TRAINING EPOCHS

In the project at the end of the chapter you will see that the above training code snippet is put inside a loop function to perform the training for a certain number of epochs. For each epoch, the two compiled models (discriminator and combined) are being trained simultaneously. During the training process, both the generator and discriminator are improving and getting better. You can observe the performance of your GAN by printing out the results after each epoch (or a set of epochs) to see how the generator is doing at generating the synthetic images. In Figure 8.13. you will see an example of the evolution of the generator's performance throughout its training process on the MNIST dataset.

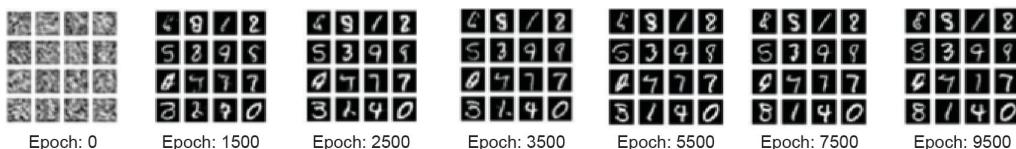


Figure 8.13: The generator is getting better at mimicking the handwritten digits of the MNIST dataset throughout its training from epoch 0 to epoch 9,500.

As you can see, epoch 0 starts with the random noise data which doesn't represent the features in the training dataset yet. As the GAN model goes through the training, its generator gets better and better at creating high quality imitation of the training dataset that can fool the discriminator. Manually observing the generator's performance is a good way to evaluate the system performance to decide on the number of epochs and when to stop training. More on the GAN evaluation techniques in section 2 of this chapter.

8.1.4 GAN Minimax Function

GAN training is more of a zero-sum game than it is an optimization problem. In zero-sum games, the total utility score is divided among the players. An increase in one player's score results into the decrease in another player's score. In AI, this is called a minimax game theory. Minimax is a decision-making algorithm, typically used in a turn-based, two player games. The goal of the algorithm is to find the optimal next move. In which, one player is called the maximizer that works to get the maximum possible score, and the other player is a minimizer that tries to get the lowest score by trying to counter move the maximizer.

GANs plays a minmax game, where the entire network attempts to optimize the function $V(D, G)$ in Figure 8.14a. The goal of the discriminator (D) is to **maximize** the probability of getting the correct label of the image. Whereas, the generator's (G) goal on the other hand is to **minimize** the chances of getting caught, hence the name minmax objective function.

Minimax game theory

In a two-person, zero-sum game, a person can win only if the other player loses. No cooperation is possible. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc. In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

We train D to maximize the probability of assigning the correct label to both training examples and samples from G . We simultaneously train G to minimize $\log(1 - D(G(z)))$. In other words, D and G play the following two-player minimax game with value function $V(D, G)$:

$$\text{Min}_{G} \text{ Max}_{D} V(D, G) = E_{x \sim p_{\text{data}}} [\log D(x)] + E_{z \sim P_Z(z)} [\log(1 - D(G(z)))]$$

Figure 8.14a: Minimax objective function.

Like any other mathematical equation, to anyone who isn't well versed in the math behind it, it looks terrifying, but the idea it represents is simple, yet powerful. It's just a mathematical

representation of the two competing objectives of the discriminator and the generator models. Let's go through the symbols first and explain it after.

Table 8.1

Symbol	Explanation
G	Generator
D	Discriminator
z	the random noise that is fed to the generator (G).
G(z)	the generator G takes the random noise data (z) and tries to reconstruct the real images.
D(G(z))	The discriminator output from the generator
D(x)	The discriminator's probability output for real data is defined by D(x)

The discriminator takes its input from two sources:

13. Data from the generator $G(z)$ - this is the fake data (z). The discriminator output from the generator is denoted as $D(G(z))$
14. The real input from the real training data (x) - The discriminator output from the real data is denoted as $\log D(x)$.

To simplify the minmax equation in Figure 8.14a, the best way to look at it is to break it down into two components: 1) the discriminator training function, and 2) the generator training (combined model) function. During the training process, we created two training flows, each have their own error function:

15. One for the discriminator alone represented by the function below which aims to maximize the minimax function by making the predictions as close as possible to 1:

$$E_{x \sim p_{\text{data}}} [\log D(x)]$$

Figure 8.14b: error of the discriminator

16. And another one for the combined model to train the generator represented by the function below which aims to minimize the minimax function by making the predictions as close as possible to 0:

$$E_{z \sim P_Z(z)} [\log(1 - D(G(z)))]$$

Figure 8.14c: error of the generator

Now that we have understood the equation symbols and have a better intuition on how the minimax function works, let's look at the minmax function again.

$$\text{Min}_{G} \text{ Max}_{D} V(D, G) = E_{x \sim p_{data}} [\log D(x)] + E_{z \sim P_Z(z)} [\log(1 - D(G(z)))]$$

↓ ↓ ↓
 Error from the discriminator
model training Error from the combined
model training

Figure 8.14d: Minimax objective function.

The goal of the minmax objective function $V(D, G)$ is to maximize $D(x)$ from the true data distribution and minimize $D(G(z))$ that is from the fake data distribution. To achieve this, we use the log-likelihood of $D(x)$ and $1-D(z)$ in the objective function. The log just makes sure that the closer it is to an incorrect value, the more it is penalized.

INTUITION OF MINIMAX IN GANS

Early in the training process, the Discriminator will reject generates fake data from Generator with high confidence because the fake images will be very different from the real training data since the Generator hasn't learned yet. As we train Discriminator to maximize the probability of assigning the correct labels to both real examples and fake images from Generator. We simultaneously train Generator to minimize the Discriminator classification error for the generated fake data. The discriminator wants to maximize objective such that $D(x)$ is close to 1 for real data and $D(G(z))$ is close to 0 for fake data. Whereas, the generator wants to minimize objective such that $D(G(z))$ is close to 1 so that the discriminator is fooled into thinking generated $G(z)$ is real. We stop the training when the fake data generated by the Generator is recognized as the real data.

8.2 Evaluate GAN models

Deep learning neural network models that are used for classification and detection problems are trained with a loss function until convergence. A GAN generator model on the other hand is trained using a second model called a discriminator that learns to classify images as real or generated. As we just learned in the previous section, both the generator and discriminator model are trained together to maintain an equilibrium. As such, there is no objective loss function used to train the GAN generator models and no way to objectively assess the progress of the training and the relative or absolute quality of the model from loss alone. This means that models must be evaluated using the quality of the generated synthetic images and manually inspecting the generated images.

A good way to identify evaluation techniques is to review research papers and find out the techniques that the authors used to evaluate their GAN. One technique that Ian Goodfellow et al. used in their paper "[Improved Techniques for Training GANs](#)" to evaluate their GAN performance is by having human annotators manually judge the visual quality of the synthesized samples. They created a web interface (Figure 8.15) and hired annotators on Amazon Mechanical Turk (MTurk) to distinguish between generated data and real data.

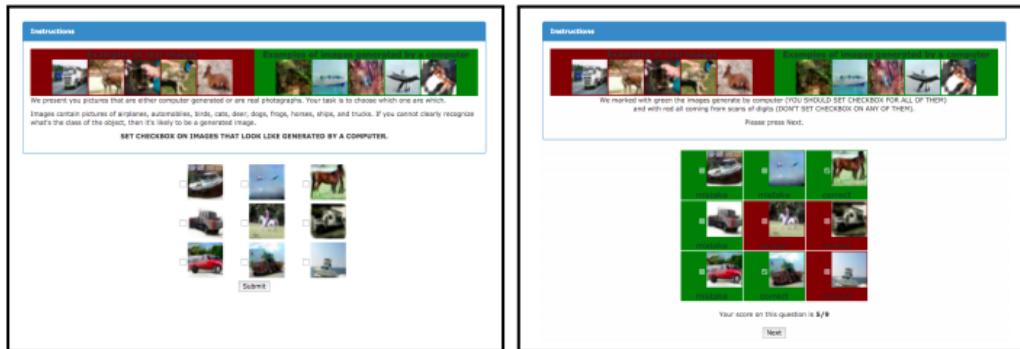


Figure 8.15: Web interface given to annotators. Annotators are asked to distinguish computer generated images from real ones. Screenshot taken from the [original paper](#).

One downside of using human annotators is that the metric varies depending on the setup of the task and the motivation of the annotators. They also found that results change drastically when they give annotators feedback about their mistakes: By learning from such feedback, annotators are better able to point out the flaws in generated images, giving a more pessimistic quality assessment. The left image in Figure 8.15 presents a screen from the annotation process, while the right column shows feedback to annotators about their mistakes.

Other non-manual ways were used in their paper and by other researchers that we will go through in this section. In general, there is no consensus on a one correct way of evaluating a given GAN generator model. This makes it challenging for researchers and practitioners to:

- Select the best GAN generator model during a training run. In other words, when to stop training.
- Choose generated images to demonstrate the capability of a GAN generator model.
- Compare and benchmark GAN model architectures.
- Tune the model hyperparameters and configuration and compare results.

Finding quantifiable ways to understand GAN progress and output quality is still an active area of research. A suite of qualitative and quantitative techniques have been developed to assess the performance of a GAN model based on the quality and diversity of the generated synthetic images. Two commonly used evaluation metrics for image quality and diversity are: the *Inception Score* and the *Fréchet Inception Distance (FID)*.

In this section, you will discover techniques for evaluating generative adversarial network models based on generated synthetic images.

8.2.1 Inception score

The Inception Score is based on a heuristic that realistic samples should be able to be classified when passed through a pre-trained network, such as Inception on ImageNet, hence the name Inception Score. The idea is really simple. The heuristic relies on two values:

17. High predictability of the generated image: we apply a pre-trained inception classifier model to every generated image and get its softmax prediction. If the generated image is so good, then it should give us high predictability score.
18. Diverse the generated samples: meaning that there should not be any dominating classes over the distribution of the generated images.

A large number of generated images are classified using the model. Specifically, the probability of the image belonging to each class is predicted. The probabilities are then summarized in the score to both capture how much each image looks like a known class and how diverse the set of images are across the known classes. If both these traits are satisfied, there should be a large Inception Score. A higher inception score indicates better-quality generated images.

8.2.2 Fréchet Inception Distance (FID)

The Fréchet Inception Distance (FID) score was proposed and used by Martin Heusel et al. in their paper "[GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium](#)" in 2017. The score was proposed as an improvement over the existing Inception Score.

Like the inception score, the FID score uses the inception model to capture specific features of an input image. These activations are calculated for a collection of real and generated images. The activations for each real and generated image are summarized as a multivariate Gaussian and the distance between these two distributions is then calculated using the Frechet distance, also called the Wasserstein-2 distance.

An important note is that FID needs a decent sample size to give good results (suggested size = 50k samples). If you use too few samples, you will end up over-estimating your actual FID and the estimates will have a large variance. A lower FID score indicates more realistic images that match the statistical properties of real images.

8.2.3 Which evaluation scheme to use?

Both measures (inception score and FID) are easy to implement and calculate on batches of generated images. As such, the practice of systematically generating images and saving models during training can and should continue to be used to allow post-hoc model selection. Diving deep into the inception score and FID is out of scope of this book. As mentioned earlier, this is an active area of research and there is no consensus in the industry as of the time of writing on the one best approach to evaluate GANs performance. Different scores assess various aspects of the image generation process, and it is unlikely that a single score can cover all aspects. The goal of this section is to expose you to some techniques that have been developed in recent years to automate the GAN evaluation process. Manual evaluation is still widely used to evaluate the GANs performance.

When getting started, it is a good idea to start with the manual inspection of generated images in order to evaluate and select generator models. Developing GAN models is complex enough for beginners and experts. Manual inspection can get you a long way while refining your model implementation and testing model configurations.

Other researchers are taking different approaches by using domain-specific evaluation metrics. In '[How good is my GAN?](#)', Konstantin Shmelkov and his team use two measures

based on image classification, GAN-train and GAN-test, which approximate the recall (diversity) and precision (quality of the image) of GANs respectively.

8.3 Popular GANs Applications

Generative modeling has come a long way in the last five years. The field has developed to the point where it is expected that the next generation of generative models will be more comfortable creating art even more than humans. GANs now have the power to solve the problems of many industries like healthcare, Automobile fine arts, and many others. In this section, we will learn about some of the use cases of adversarial networks and which GAN architecture is used for that application. The goal of this section is not to implement the variations of GAN network, but to provide some exposure of potential applications GAN models and resources for further readings.

8.3.1 Text-to-Photo Synthesis

Synthesis of high-quality images from text descriptions is a challenging problem in computer vision. Samples generated by an existing text to image approaches can roughly reflect the meaning of the given descriptions, but they fail to contain necessary details and vivid object parts.

The GAN network that was built for this application is the StackGAN or Stacked Generative Adversarial Network which was introduced in the paper titled: "[StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks](#)". In the paper, StackGAN was able to generate 256x256 photo-realistic images conditioned on text descriptions.

StackGANs work in two stages:

- **Stage-I:** GAN sketches the primitive shape and colors of the object based on the given text description, yielding low-resolution images
- **Stage-II:** GAN takes output of stage-I and text description as input and generates high-resolution images with photo-realistic details. It is able to rectify defects in the images created in Stage-I and add compelling details with the refinement process.

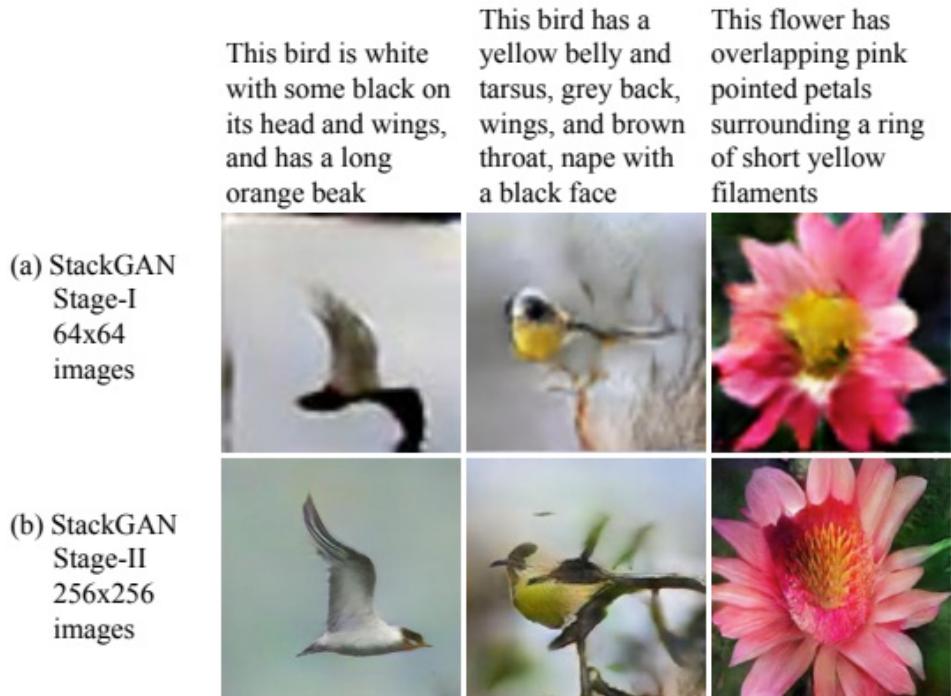


Figure 8.16: (a) Stage-I: Given text descriptions, StackGAN sketches rough shapes and basic colors of objects, yielding low-resolution images. (b) Stage-II of StackGAN takes Stage-I results and text descriptions as inputs, and generates high-resolution images with photo-realistic details. Image taken from the [original paper](#).

8.3.2 Image-to-image translation (Pix2Pix GAN)

Image-to-image translation is inspired by the language translation analogy. Just like an idea can be expressed by many different languages, a scene may be rendered by grayscale image, RGB image, semantic label map, edge sketches, etc. Image-to-image translation is defined as translating one representation of a scene into another given sufficient training data. See examples in Figure 8.17.

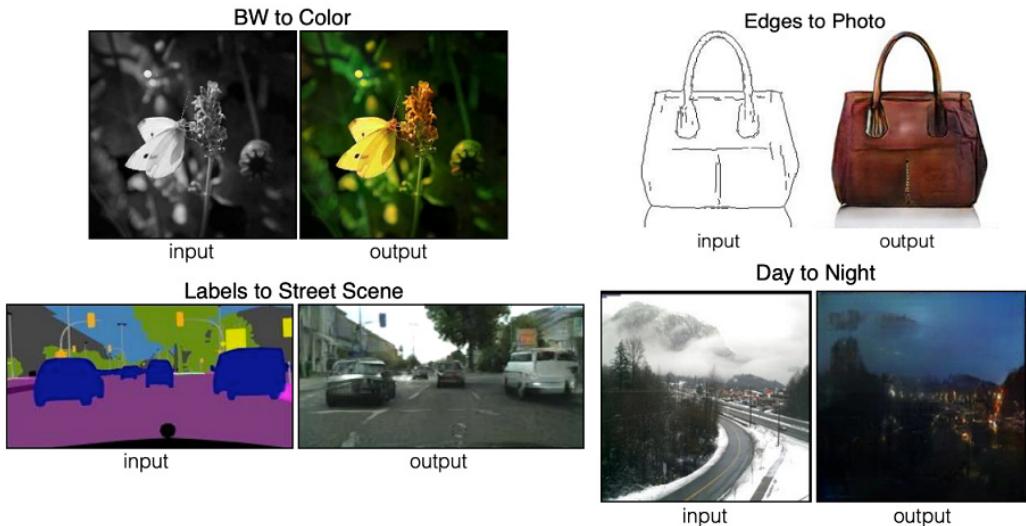


Figure 8.17: Examples for Pix2Pix applications taken from the [original paper](#).

As you can see in Figure 8.17, the image-to-image translation tasks have been demonstrated on a wide range of applications such as converting street scene segmentation labels to real images, grayscale to color images, sketches of products to product photographs, and day photographs to night ones.

Pix2Pix is a type of GAN models that is designed for general purpose image-to-image translation. The network was presented by Phillip Isola, et al. in their 2016 paper titled "[Image-to-Image Translation with Conditional Adversarial Networks](#)". The Pix2Pix network architecture is similar to the GAN concept. It is comprised of a generator model for outputting new synthetic images that look realistic, and a discriminator model that classifies images as real (from the dataset) or fake (generated). The training process is also similar to GANs: the discriminator model is updated directly, whereas the generator model is updated via the discriminator model. As such, the two models are trained simultaneously in an adversarial process where the generator seeks to better fool the discriminator and the discriminator seeks to better identify the counterfeit images.

The novel idea of Pix2Pix network is that they learn a loss function adapted to the task and data at hand, which makes them applicable in a wide variety of settings. They are a type of conditional GAN, or cGAN, where the generation of the output image is **conditional** on an input source image. The discriminator is provided both with a source image and the target image and must determine whether the target is a plausible transformation of the source image.

The results of the Pix2Pix network are really promising for many image-to-image translation tasks. Visit <https://affinelayer.com/pixsrv/> to play more with the Pix2Pix network. This is an interactive demo that was created by the authors of the Pix2Pix paper. In which you can convert sketch edges of cats or products to photos and facades to real images.

8.3.3 Image Super-Resolution GAN (SRGAN)

A certain type of GANs models can be used to convert low-resolution images into high-resolution images. This type is called Super-Resolution Generative Adversarial Networks (SRGAN) which was introduced in the paper titled: "[Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network](#)". We can see how SRGAN were able to create the highest resolution image as seen in Figure 8.18 taken from the original paper.

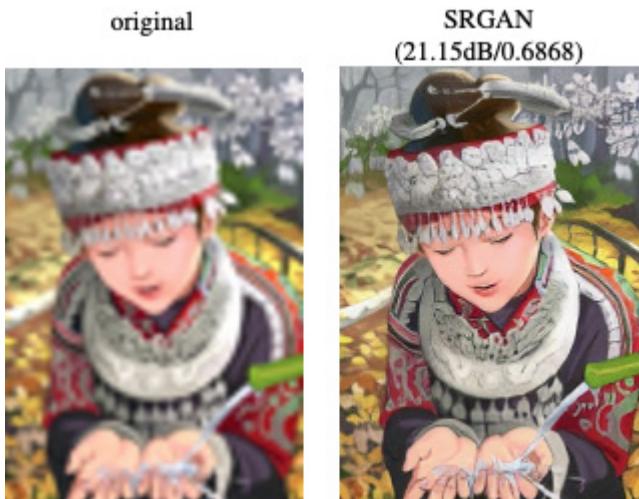


Figure 8.18: SRGAN converting low-resolution image to a high-resolution image. Taken from the [original paper](#).

8.3.4 Ready to get your hands dirty?

GAN models have a huge potential in creating and imagining new realities that have never existed before. The applications that are mentioned in this chapter are just a few selected ones are just examples to give you an idea on what GANs can do now. These applications and many more come out every few weeks that are worth trying out. My goal from this section is to give you a taste of the kind of magical applications that you can build using the GAN model and its variations.

If you are interested in getting your hands dirty with more GAN applications, visit this amazing Keras-GAN repository: <https://github.com/eriklindernoren/Keras-GAN> maintained by [Erik Linder-Norén](#). It has many GAN models created using Keras and is an excellent resource for many Keras examples. Much of the code in this chapter has been inspired by and adapted from this repository.

8.4 Building your own GAN project

In this project, you'll build a GAN using convolutional layers in the generator and discriminator. This is called a Deep Convolutional GAN, or DCGAN for short. The DCGAN architecture was first explored in 2016 by Alec Radford et al. and has seen impressive results

in generating new images, you can read the [original paper here](#). You can follow along with the implementation in this chapter or run code in the project notebook at [www.computervisionbook.com](#).

In this project, you'll be training DCGAN on the [Fashion MNIST](#) dataset. Fashion MNIST dataset consists of 60,000 grayscale images for training and a test set of 10,000 images. Each image is a 28x28 grayscale image, associated with a label from 10 classes. Fashion-MNIST is intended to serve as a direct replacement for the original MNIST dataset for benchmarking machine learning algorithms. I chose grayscale images for this project because it requires less computational power to train convolutional networks on 1 channel grayscale images compared to 3 channels colored images which makes it easier for you to train on a personal computer without a GPU.



Figure 8.19: Fashion_MNIST dataset examples.

The dataset is broken in to 10 fashion categories. The class labels are:

Table 8.2

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt

7	Sneaker
8	Bag
9	Ankle boot

8.4.1 Import libraries

As always, the first thing we are going to do is to import all the libraries that we are going to use in this project.

```
from __future__ import print_function, division

# import the fashion_mnist dataset from Keras
from keras.datasets import fashion_mnist

# import Keras layers and models
from keras.layers import Input, Dense, Reshape, Flatten, Dropout
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D, Conv2D
from keras.models import Sequential, Model
from keras.optimizers import Adam

# import numpy and matplotlib
import numpy as np
import matplotlib.pyplot as plt
```

8.4.2 Download and visualize the dataset

Keras has the Fashion MNIST dataset available for us to download with only one command: `fashion_mnist.load_data()`. In this section, we are going to download the dataset and rescale the training set to the range -1 to 1 to allow the model to converge faster. See the “feature scaling” section in the project at the end of chapter 3 to more details on image scaling.

```
# Load the dataset
(training_data, _), (_, _) = fashion_mnist.load_data()

# Rescale the training data to scale -1 to 1.
X_train = training_data / 127.5 - 1.
X_train = np.expand_dims(X_train, axis=3)
```

Just for the fun of it, let’s visualize the image matrix:

```
def visualize_input(img, ax):
    ax.imshow(img, cmap='gray')
    width, height = img.shape
    thresh = img.max()/2.5
    for x in range(width):
        for y in range(height):
            ax.annotate(str(round(img[x][y],2)), xy=(y,x),
                        horizontalalignment='center',
                        verticalalignment='center',
                        color='white' if img[x][y]<thresh else 'black')
```

```
fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
visualize_input(training_data[3343], ax)
```

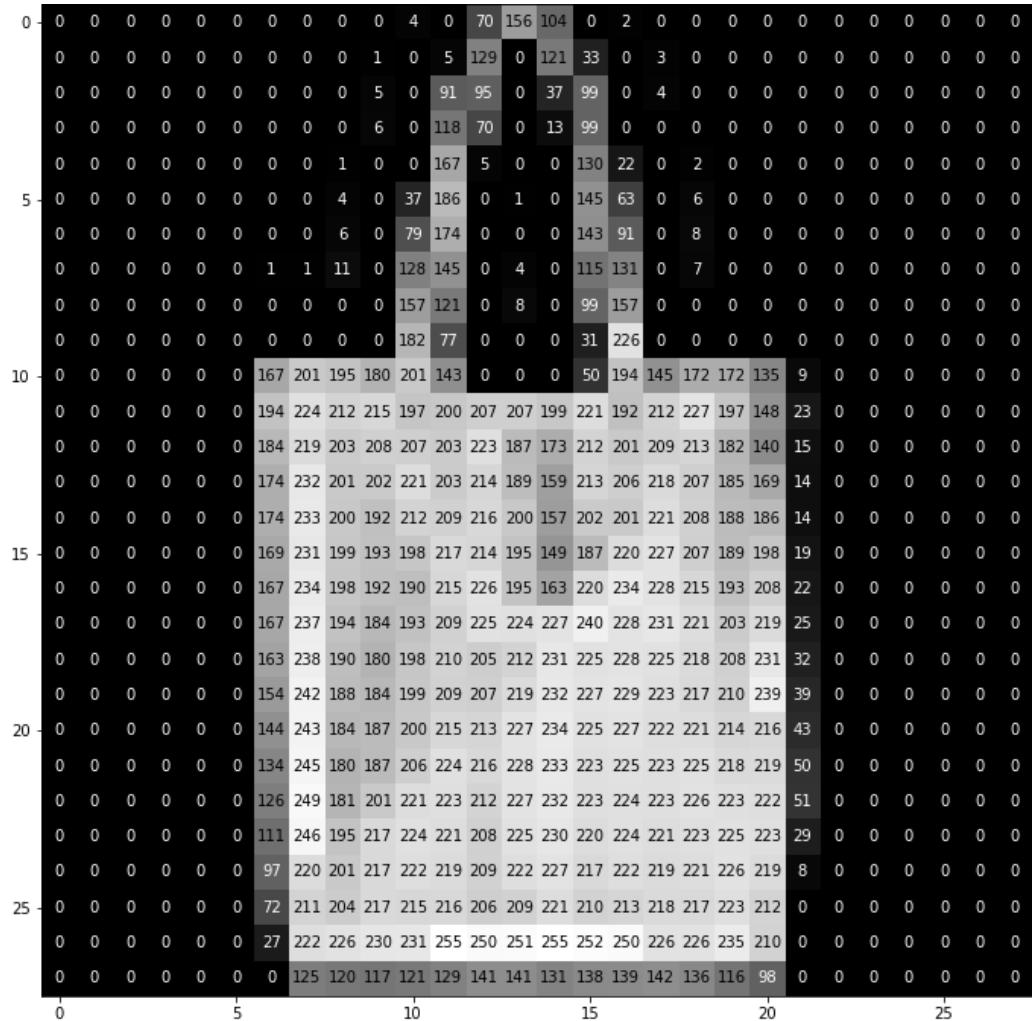


Figure 8.20: A visualized example of the Fashion-MNIST dataset.

8.4.3 Build the generator

Now, let's build the generator model. The input will be our noise vector (z) as explained earlier in this chapter. The generator architecture looks like the following:

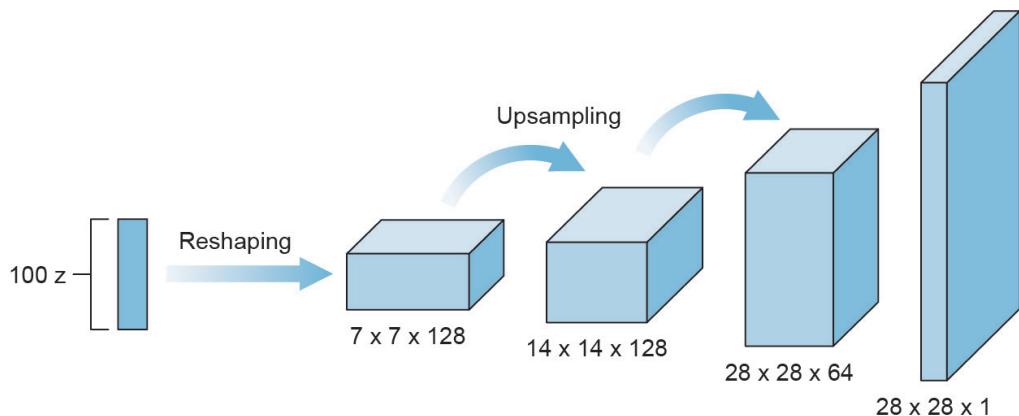


Figure 8.21: Architecture of the generator model.

The first layer is a fully connected layer which is then reshaped into a deep and narrow layer, something like 7x7x128 (in the original DCGAN paper, the authors reshaped the input to 4x4x1024). Then we use the upsampling layer to double the feature maps dimensions from 7x7 to 14x14 then again to 28x28. In this network, we use 3 convolutional layers. We also use batch normalization and a ReLU activation. For each of these layers, the general scheme is convolution > batch norm > ReLU. You keep stacking layers up like this until you get the final transposed convolution layer with shape 28x28x1.

```

def build_generator():
    # instantiate a sequential model and name it generator
    generator = Sequential()
    # add the Dense Layer that has a number of neurons = 128x7x7
    generator.add(Dense(128 * 7 * 7, activation="relu", input_dim=100))
    # reshape the image dimensions to 7 x 7 x 128
    generator.add(Reshape((7, 7, 128)))
    # upsampling Layer to double the size of the image dimensions to 14 x 14
    generator.add(UpSampling2D())

    # add a convolutional Layer to run the convolutional process + batchNorm
    generator.add(Conv2D(128, kernel_size=3, padding="same", activation="relu"))
    generator.add(BatchNormalization(momentum=0.8))
    # upsample the image dimensions to 28 x 28
    generator.add(UpSampling2D())

    # conv + batchNorm Layers
    # notice that we didn't add upsampling here because we have the image size of 28 x 28 which
    # is equal to the image size in the MNIST dataset. You can adjust this for your own
    # problem.
    generator.add(Conv2D(64, kernel_size=3, padding="same", activation="relu"))
    generator.add(BatchNormalization(momentum=0.8))

    # convolutional Layer with filters = 1
    generator.add(Conv2D(1, kernel_size=3, padding="same", activation="relu"))

# print the model summary

```

```

generator.summary()

# generate the input noise vector of length = 100
# we chose 100 here to create a simple network
noise = Input(shape=(100,))
# run the generator model to create the fake image
fake_image = generator(noise)
# return a model that takes the noise vector as an input and outputs the fake image
return Model(inputs=noise, outputs=fake_image)

```

8.4.4 Build the discriminator

The discriminator is just a convolutional classifier like what we have built before. The input to the discriminator are 28x28x1 images. You'll want a few convolutional layers, then a fully connected layer for the output. As before, we want a sigmoid output, and you'll need to return the logits as well. For the depths of the convolutional layers I suggest starting with 32 or 64 filters in the first layer, then double the depth as you add layers. In this implementation, we will start with 64 layers depth, then 128, then 256 layers. For downsampling, we will not be using pooling layers. Instead, we will be using only strided convolutional layers for downsampling similar to the DCGAN paper implementation.

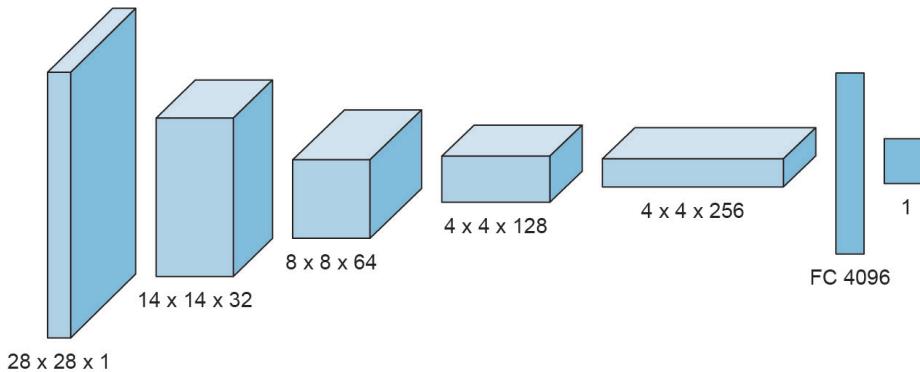


Figure 8.22: Architecture of the discriminator model

We will also use batch norm and dropout to optimize training as we learned in chapter 4. For each of the four convolutional layers, the general scheme is convolution > batch norm > Leaky ReLU. Now, let's build the `build_discriminator` function:

```

def build_discriminator():
    # instantiate a sequential model and name it discriminator
discriminator = Sequential()
# add a convolutional layer to the discriminator model
discriminator.add(Conv2D(32, kernel_size=3, strides=2, input_shape=(28,28,1),
    padding="same"))
# add a LeakyReLU activation function
    discriminator.add(LeakyReLU(alpha=0.2))
# add a dropout layer with a 25% dropout probability

```

```

discriminator.add(Dropout(0.25))
# add a second convolutional layer with zero padding
discriminator.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
# add a zero padding layer to change the dimension from 7x7 to 8x8
discriminator.add(ZeroPadding2D(padding=((0,1),(0,1))))
# add a BatchNormalization Layer for faster Learning and higher accuracy
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))

# add a third convolutional layer with batch norm, LeakyRelu, and a dropout
discriminator.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))

# add the fourth convolutional layer with batch norm, LeakyRelu, and a dropout
discriminator.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
discriminator.add(BatchNormalization(momentum=0.8))
discriminator.add(LeakyReLU(alpha=0.2))
discriminator.add(Dropout(0.25))

# flatten the network and add the output Dense Layer with sigmoid activation function
discriminator.add(Flatten())
discriminator.add(Dense(1, activation='sigmoid'))

# set the input image shape
img = Input(shape=(28,28,1))
# run the discriminator model to get the output probability
probability = discriminator(img)

# return a Model that takes the image as an input and produces the probability output
return Model(inputs=img, outputs=probability)

```

8.4.5 Build the combined model

As explained in section 1.3. in this chapter, to train the Generator, we need to build a combined network that contains both the generator and the discriminator. The combined model takes the noise signal as input (z) and output the discriminator's prediction output as fake or real.

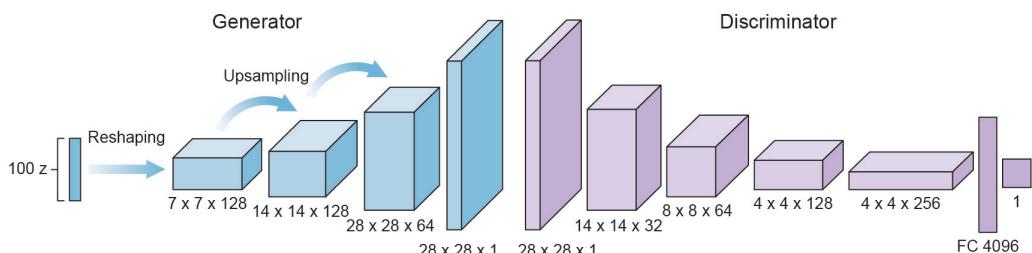


Figure 8.23: Architecture of the combined model.

Remember that we want to disable the discriminator training for the combined model. This was explained in detail in section 1.3 of this chapter. When training the generator, we don't want the discriminator to be updating weights as well but we still want to include the discriminator model in the generator training. So, we create a combined network that include both models but freeze the weights of the discriminator model in the combined network.

```
# define the optimizer
optimizer = Adam(learning_rate=0.0002, beta_1=0.5)

# Build and compile the discriminator. Freeze the discriminator weights because we don't want
# to train it during the generator training
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])

discriminator.trainable = False

# Build the generator
generator = build_generator()

# The generator takes noise as input with latent_dim = 100 and generates images
z = Input(shape=(100,))
img = generator(z)

# The discriminator takes generated images as input and determines validity
valid = discriminator(img)

# The combined model (stacked generator and discriminator)
# Trains the generator to fool the discriminator
combined = Model(inputs=z, outputs=valid)
combined.compile(loss='binary_crossentropy', optimizer=optimizer)
```

8.4.6 Build the training function

When training the GAN model, we train two networks: 1) the discriminator, and 2) the combined network that we just created in the previous section. Let's build the `train` function that takes the following arguments:

- The number of epochs
- The batch size
- `save_interval` to state how often we want to save the results

```
def train(epochs, batch_size=128, save_interval=50):

    # Adversarial ground truths
    valid = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))

    for epoch in range(epochs):

        ## Train Discriminator network

        # Select a random half of images
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]

        # Sample noise and generate a batch of new images
        noise = np.random.normal(0, 1, (batch_size, 100))
```

```

gen_imgs = generator.predict(noise)

# Train the discriminator (real classified as ones and generated as zeros)
d_loss_real = discriminator.train_on_batch(imgs, valid)
d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

## Train the combined network (Generator)

# Train the generator (wants discriminator to mistake images as real)
g_loss = combined.train_on_batch(noise, valid)

# Print progress
print("%d [D loss: %f, acc.: %.2f%] [G loss: %f]" % (epoch, d_loss[0], 100*d_loss[1],
g_loss))

# save generated image samples if at save_interval
if epoch % save_interval == 0:
    plot_generated_images(epoch, generator)

```

8.4.7 Train and observe results

Now that all the code implementation is complete, we are ready to start the DCGAN training. To train the model, run the code snippet below:

```
train(epochs=1000, batch_size=32, save_interval=50)
```

This will run the training for 1,000 epochs and saves images every 50 epochs. When you run the "train()" function, the training progress prints as follows:

```

0 [D loss: 0.963556, acc.: 42.19%] [G loss: 0.726341]
1 [D loss: 0.707453, acc.: 65.62%] [G loss: 1.239887]
2 [D loss: 0.478705, acc.: 76.56%] [G loss: 1.666347]
3 [D loss: 0.721997, acc.: 60.94%] [G loss: 2.243804]
4 [D loss: 0.937356, acc.: 45.31%] [G loss: 1.459240]
5 [D loss: 0.881121, acc.: 50.00%] [G loss: 1.417385]
6 [D loss: 0.558153, acc.: 73.44%] [G loss: 1.393961]
7 [D loss: 0.404117, acc.: 78.12%] [G loss: 1.141378]
8 [D loss: 0.452483, acc.: 82.81%] [G loss: 0.802813]
9 [D loss: 0.591792, acc.: 76.56%] [G loss: 0.690274]
10 [D loss: 0.753802, acc.: 67.19%] [G loss: 0.934047]
11 [D loss: 0.957626, acc.: 50.00%] [G loss: 1.140045]
12 [D loss: 0.919308, acc.: 51.56%] [G loss: 1.311618]
13 [D loss: 0.776363, acc.: 56.25%] [G loss: 1.041264]
14 [D loss: 0.763993, acc.: 56.25%] [G loss: 1.090716]
15 [D loss: 0.754735, acc.: 56.25%] [G loss: 1.530865]
16 [D loss: 0.739731, acc.: 68.75%] [G loss: 1.887644]

```

I ran this training myself for 10,000 epochs and here are my results after 0, 50, 1,000, and 10,000 epochs.



Figure 8.24: Outputs of the GAN Generator after 0, 50, 1000, and 10000 epochs.

As you can see in Figure 8.24, at epoch 0, the images are just random noise. It doesn't contain any patterns or meaningful data. At epoch 50, you can see that patterns started to form. One very apparent pattern is the bright pixels started to form at the center of the image and the surroundings are darker pixels. This is because in the training data all the shapes are located at the center of the image. Later in the training process, at epoch 1,000, things started to take clear shapes. If you look at the images generated at epoch 1000, you can probably guess the type of training data that is fed to the GAN model. Fast forward to epoch 10,000, you can see that the generator has become very good at recreating new images that is not present in the training dataset. For example, pick any of the objects that is created at this epoch, let's say the top left image (dress). This is totally new dress design that is not present in the training dataset. So the GAN model just created a completely new dress design after learning the dress patterns from the training set. You can keep the training running even more or make the generator network even deeper to get more refined results.

8.4.8 Closing

In this project, I picked Fashion-MNIST dataset because the images are very small and are in grayscale (1 channel) which makes it computationally not expensive for you to train on your local computer with no GPU. Fashion-MNIST is also very clean data because all images are

centered exactly in the middle of the image and have less noise which doesn't require much preprocessing before you kick off your GAN training. This makes it a good toy dataset to jumpstart your first GAN project. If you are excited to get your hands dirty with more advanced datasets, you can try CIFAR as your next step or Google's "Quick, Draw!" dataset which is considered the world's largest doodle dataset at the time of writing. Other more serious datasets like Stanford's [Car Dataset](#) that contains more than 16k images of 196 classes of cars. You can try to train your GAN model to design a completely new design for your dream car!

8.5 Summary and takeaways

- GANs learn patterns from the training dataset and create new images that have similar distribution of the training set.
- The GAN architecture consists of two deep neural networks that compete with each other.
- **The Generator** tries to convert random noise into observations that look as if they have been sampled from the original dataset.
- **The Discriminator** tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries.
- The discriminator's model is a typical classification neural network that aims to classify images generated by the generator as real or fake.
- The generator's architecture looks like an inverted CNN that starts with a narrow input and it is upsampled a few times until it gets to the desired size.
- The upsampling layer scales the image dimensions by repeating each row and column of its input pixels.
- To train the GAN, we train the network in batches through two parallel networks: 1) the discriminator, and 2) a combined network where we freeze the weights of the discriminator and update only the generator's weights.
- To evaluate the GAN, we mostly rely on our observation of the quality of images created by the generator. Other evaluation metrics are Inception Score and Fréchet Inception Distance (FID).
- In addition to generating new images, GANs can be used in other applications like: text-to-photo synthesis, image-to-image translation, image super-resolution, and many other applications.