

Rajalingappaa Shanmugamani

Foreword by:

Dr. Stephen Moore

Chief Technology Officer, EmotionReader, Singapore

Deep Learning for Computer Vision

Expert techniques to train advanced neural networks
using TensorFlow and Keras



Packt

Deep Learning for Computer Vision

Expert techniques to train advanced neural networks using TensorFlow and Keras

Rajalingappaa Shanmugamani

Packt>

BIRMINGHAM - MUMBAI

Deep Learning for Computer Vision

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Amey Varangaonkar

Acquisition Editor: Aman Singh

Content Development Editor: Varun Sony

Technical Editor: Dharmendra Yadav

Copy Editors: Safis Editing

Project Coordinator: Manthan Patel

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Graphics: Tania Dutta

Production Coordinator: Shantanu Zagade

First published: January 2018

Production reference: 1220118

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78829-562-8

www.packtpub.com



`mapt.io`

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Table of Contents

Preface	1
Chapter 1: Getting Started	7
Understanding deep learning	7
Perceptron	8
Activation functions	8
Sigmoid	9
The hyperbolic tangent function	10
The Rectified Linear Unit (ReLU)	11
Artificial neural network (ANN)	12
One-hot encoding	13
Softmax	13
Cross-entropy	13
Dropout	14
Batch normalization	14
L1 and L2 regularization	15
Training neural networks	15
Backpropagation	15
Gradient descent	16
Stochastic gradient descent	16
Playing with TensorFlow playground	17
Convolutional neural network	18
Kernel	18
Max pooling	19
Recurrent neural networks (RNN)	20
Long short-term memory (LSTM)	21
Deep learning for computer vision	22
Classification	22
Detection or localization and segmentation	23
Similarity learning	24
Image captioning	24
Generative models	25
Video analysis	26
Development environment setup	26
Hardware and Operating Systems - OS	27
General Purpose - Graphics Processing Unit (GP-GPU)	27
Computer Unified Device Architecture - CUDA	27
CUDA Deep Neural Network - CUDNN	29

Installing software packages	29
Python	29
Open Computer Vision - OpenCV	30
The TensorFlow library	31
Installing TensorFlow	31
TensorFlow example to print Hello, TensorFlow	31
TensorFlow example for adding two numbers	32
TensorBoard	33
The TensorFlow Serving tool	35
The Keras library	36
Summary	37
Chapter 2: Image Classification	39
Training the MNIST model in TensorFlow	39
The MNIST datasets	40
Loading the MNIST data	41
Building a perceptron	41
Defining placeholders for input data and targets	41
Defining the variables for a fully connected layer	42
Training the model with data	43
Building a multilayer convolutional network	44
Utilizing TensorBoard in deep learning	49
Training the MNIST model in Keras	53
Preparing the dataset	53
Building the model	54
Other popular image testing datasets	56
The CIFAR dataset	56
The Fashion-MNIST dataset	57
The ImageNet dataset and competition	59
The bigger deep learning models	59
The AlexNet model	60
The VGG-16 model	61
The Google Inception-V3 model	62
The Microsoft ResNet-50 model	65
The SqueezeNet model	65
Spatial transformer networks	67
The DenseNet model	67
Training a model for cats versus dogs	69
Preparing the data	69
Benchmarking with simple CNN	70
Augmenting the dataset	71
Augmentation techniques	72

Table of Contents

Transfer learning or fine-tuning of a model	72
Training on bottleneck features	73
Fine-tuning several layers in deep learning	74
Developing real-world applications	76
Choosing the right model	77
Tackling the underfitting and overfitting scenarios	77
Gender and age detection from face	78
Fine-tuning apparel models	78
Brand safety	78
Summary	78
Chapter 3: Image Retrieval	79
Understanding visual features	80
Visualizing activation of deep learning models	81
Embedding visualization	82
Guided backpropagation	84
The DeepDream	87
Adversarial examples	91
Model inference	92
Exporting a model	92
Serving the trained model	94
Content-based image retrieval	96
Building the retrieval pipeline	97
Extracting bottleneck features for an image	98
Computing similarity between query image and target database	100
Efficient retrieval	100
Matching faster using approximate nearest neighbour	101
Advantages of ANNOY	102
Autoencoders of raw images	102
Denoising using autoencoders	104
Summary	108
Chapter 4: Object Detection	109
Detecting objects in an image	109
Exploring the datasets	110
ImageNet dataset	111
PASCAL VOC challenge	111
COCO object detection challenge	111
Evaluating datasets using metrics	112
Intersection over Union	112
The mean average precision	113

Localizing algorithms	114
Localizing objects using sliding windows	114
The scale-space concept	115
Training a fully connected layer as a convolution layer	115
Convolution implementation of sliding window	116
Thinking about localization as a regression problem	117
Applying regression to other problems	117
Combining regression with the sliding window	118
Detecting objects	120
Regions of the convolutional neural network (R-CNN)	120
Fast R-CNN	121
Faster R-CNN	122
Single shot multi-box detector	122
Object detection API	123
Installation and setup	123
Pre-trained models	125
Re-training object detection models	126
Data preparation for the Pet dataset	126
Object detection training pipeline	126
Training the model	127
Monitoring loss and accuracy using TensorBoard	128
Training a pedestrian detection for a self-driving car	128
The YOLO object detection algorithm	128
Summary	130
Chapter 5: Semantic Segmentation	131
Predicting pixels	131
Diagnosing medical images	133
Understanding the earth from satellite imagery	134
Enabling robots to see	134
Datasets	135
Algorithms for semantic segmentation	136
The Fully Convolutional Network	136
The SegNet architecture	137
Upsampling the layers by pooling	137
Sampling the layers by convolution	138
Skipping connections for better training	141
Dilated convolutions	142
DeepLab	143
RefiNet	144
PSPnet	146

Large kernel matters	146
DeepLab v3	147
Ultra-nerve segmentation	148
Segmenting satellite images	153
Modeling FCN for segmentation	154
Segmenting instances	156
Summary	158
Chapter 6: Similarity Learning	159
Algorithms for similarity learning	159
Siamese networks	160
Contrastive loss	160
FaceNet	163
Triplet loss	164
The DeepNet model	165
DeepRank	166
Visual recommendation systems	168
Human face analysis	168
Face detection	169
Face landmarks and attributes	170
The Multi-Task Facial Landmark (MTFL) dataset	170
The Kaggle keypoint dataset	171
The Multi-Attribute Facial Landmark (MAFL) dataset	172
Learning the facial key points	173
Face recognition	175
The labeled faces in the wild (LFW) dataset	175
The YouTube faces dataset	176
The CelebFaces Attributes dataset (CelebA)	176
CASIA web face database	176
The VGGFace2 dataset	176
Computing the similarity between faces	177
Finding the optimum threshold	179
Face clustering	181
Summary	182
Chapter 7: Image Captioning	183
Understanding the problem and datasets	183
Understanding natural language processing for image captioning	184
Expressing words in vector form	184
Converting words to vectors	185
Training an embedding	186
Approaches for image captioning and related problems	188

Using a condition random field for linking image and text	188
Using RNN on CNN features to generate captions	189
Creating captions using image ranking	192
Retrieving captions from images and images from captions	193
Dense captioning	194
Using RNN for captioning	196
Using multimodal metric space	197
Using attention network for captioning	198
Knowing when to look	199
Implementing attention-based image captioning	201
Summary	203
Chapter 8: Generative Models	205
Applications of generative models	205
Artistic style transfer	206
Predicting the next frame in a video	207
Super-resolution of images	208
Interactive image generation	209
Image to image translation	210
Text to image generation	211
Inpainting	212
Blending	212
Transforming attributes	214
Creating training data	214
Creating new animation characters	215
3D models from photos	216
Neural artistic style transfer	216
Content loss	217
Style loss using the Gram matrix	222
Style transfer	225
Generative Adversarial Networks	228
Vanilla GAN	229
Conditional GAN	232
Adversarial loss	232
Image translation	233
InfoGAN	234
Drawbacks of GAN	234
Visual dialogue model	234
Algorithm for VDM	235

Table of Contents

Generator	236
Discriminator	237
Summary	237
Chapter 9: Video Classification	239
Understanding and classifying videos	239
Exploring video classification datasets	240
UCF101	240
YouTube-8M	242
Other datasets	243
Splitting videos into frames	243
Approaches for classifying videos	244
Fusing parallel CNN for video classification	245
Classifying videos over long periods	247
Streaming two CNN's for action recognition	249
Using 3D convolution for temporal learning	251
Using trajectory for classification	252
Multi-modal fusion	253
Attending regions for classification	256
Extending image-based approaches to videos	258
Regressing the human pose	258
Tracking facial landmarks	259
Segmenting videos	259
Captioning videos	260
Generating videos	263
Summary	264
Chapter 10: Deployment	265
Performance of models	265
Quantizing the models	266
MobileNets	266
Deployment in the cloud	269
AWS	270
Google Cloud Platform	276
Deployment of models in devices	279
Jetson TX2	280
Android	281
iPhone	281
Summary	282
Other Books You May Enjoy	283
Index	287

1

Getting Started

Computer vision is the science of understanding or manipulating images and videos. Computer vision has a lot of applications, including autonomous driving, industrial inspection, and augmented reality. The use of deep learning for computer vision can be categorized into multiple categories: classification, detection, segmentation, and generation, both in images and videos. In this book, you will learn how to train deep learning models for computer vision applications and deploy them on multiple platforms. We will use **TensorFlow**, a popular python library for deep learning throughout this book for the examples. In this chapter, we will cover the following topics:

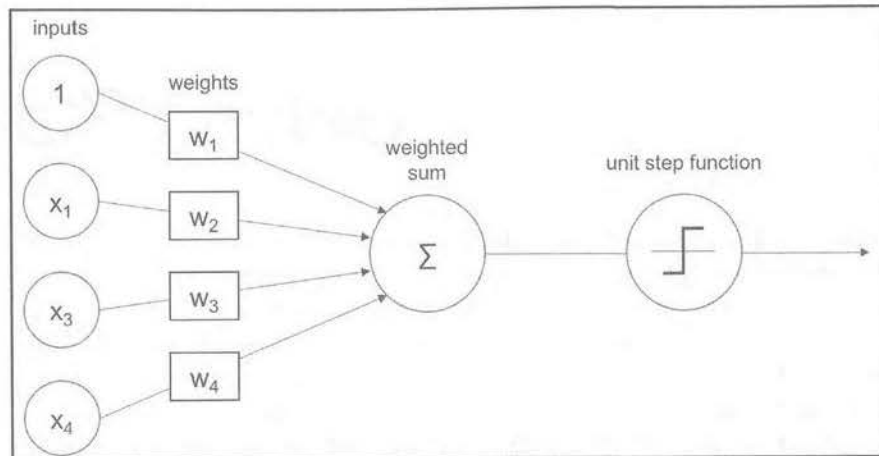
- The basics and vocabulary of deep learning
- How deep learning meets computer vision?
- Setting up the development environment that will be used for the examples covered in this book
- Getting a feel for TensorFlow, along with its powerful tools, such as TensorBoard and TensorFlow Serving

Understanding deep learning

Computer vision as a field has a long history. With the emergence of deep learning, computer vision has proven to be useful for various applications. Deep learning is a collection of techniques from **artificial neural network (ANN)**, which is a branch of machine learning. ANNs are modelled on the human brain; there are nodes linked to each other that pass information to each other. In the following sections, we will discuss in detail how deep learning works by understanding the commonly used basic terms.

Perceptron

An artificial neuron or perceptron takes several inputs and performs a weighted summation to produce an output. The weight of the perceptron is determined during the training process and is based on the training data. The following is a diagram of the perceptron:



The inputs are weighted and summed as shown in the preceding image. The sum is then passed through a unit step function, in this case, for a binary classification problem. A perceptron can only learn simple functions by learning the weights from examples. The process of learning the weights is called training. The training on a perceptron can be done through gradient-based methods which are explained in a later section. The output of the perceptron can be passed through an activation function or transfer function, which will be explained in the next section.

Activation functions

The activation functions make **neural nets** nonlinear. An activation function decides whether a perceptron should fire or not. During training activation, functions play an important role in adjusting the gradients. An activation function such as sigmoid, shown in the next section, attenuates the values with higher magnitudes. This nonlinear behaviour of the activation function gives the deep nets to learn complex functions. Most of the activation functions are continuous and differential functions, except rectified unit at 0. A continuous function has small changes in output for every small change in input. A differential function has a derivative existing at every point in the domain.

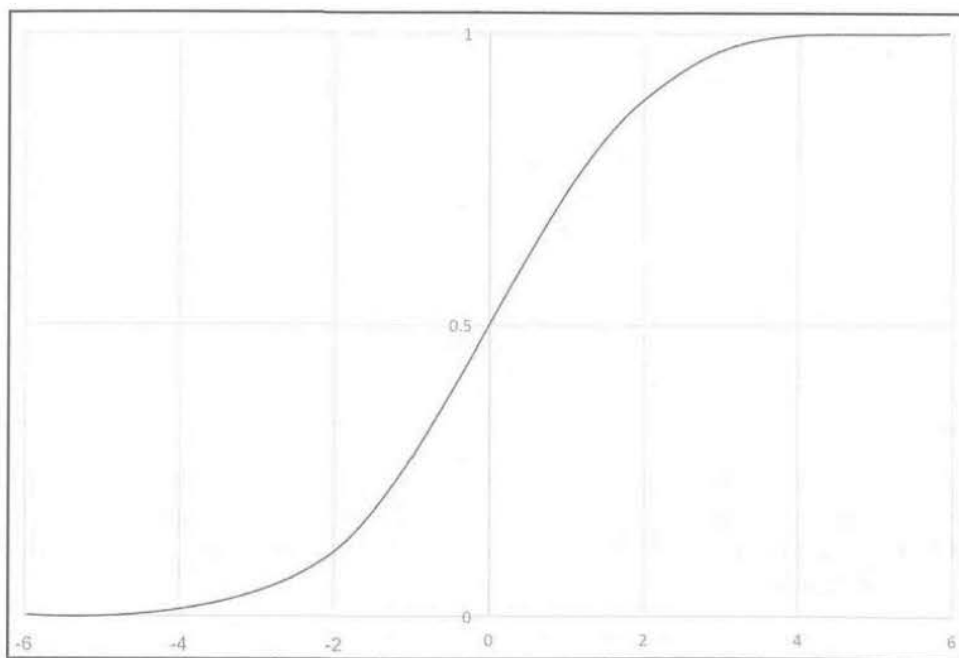
In order to train a neural network, the function has to be differentiable. Following are a few activation functions.



Don't worry if you don't understand the terms like continuous and differentiable in detail. It will become clearer over the chapters.

Sigmoid

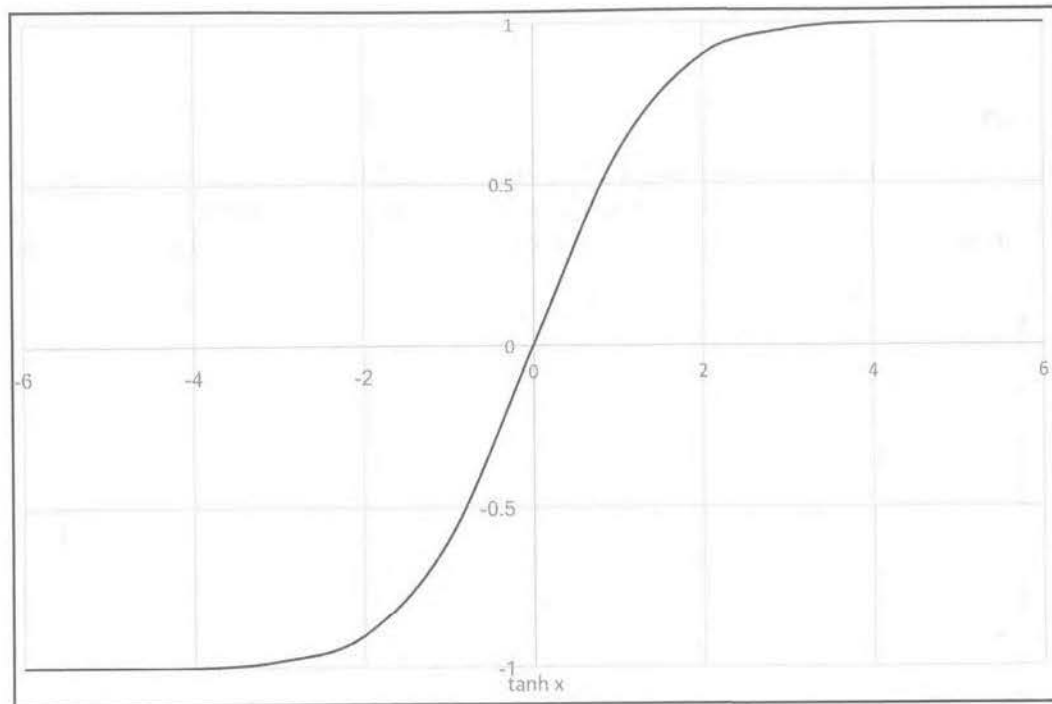
Sigmoid can be considered a smoothened step function and hence differentiable. Sigmoid is useful for converting any value to probabilities and can be used for binary classification. The sigmoid maps input to a value in the range of 0 to 1, as shown in the following graph:



The change in Y values with respect to X is going to be small, and hence, there will be vanishing gradients. After some learning, the change may be small. Another activation function called $\tanh$, explained in next section, is a scaled version of sigmoid and avoids the problem of a vanishing gradient.

The hyperbolic tangent function

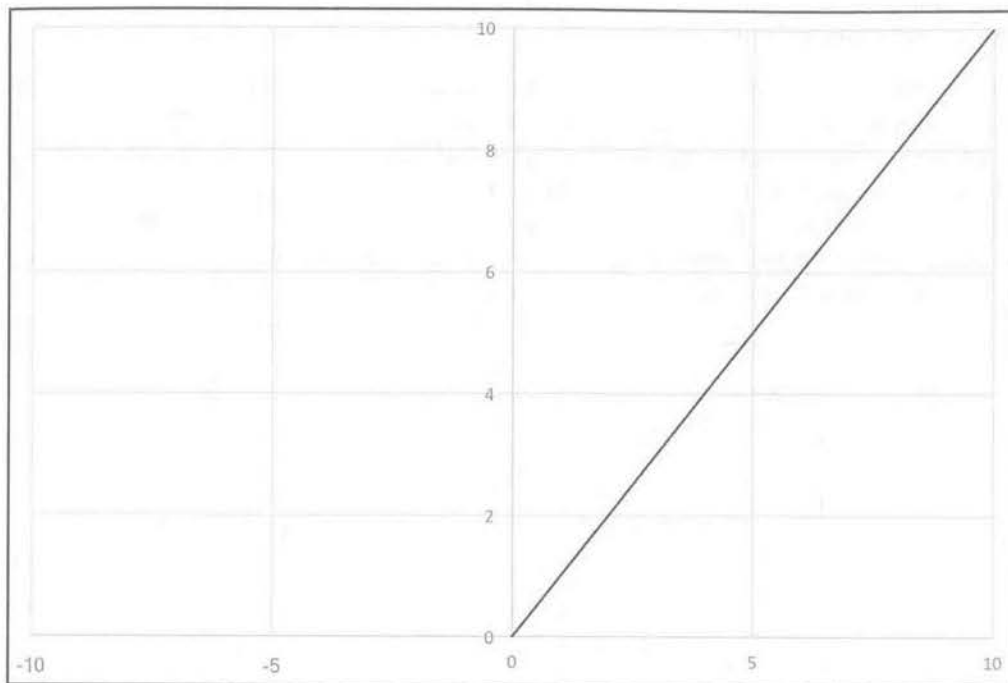
The hyperbolic tangent function, or $\tanh$, is the scaled version of sigmoid. Like sigmoid, it is smooth and differentiable. The $\tanh$ maps input to a value in the range of -1 to 1, as shown in the following graph:



The gradients are more stable than sigmoid and hence have fewer vanishing gradient problems. Both sigmoid and $\tanh$ fire all the time, making the ANN really heavy. The **Rectified Linear Unit (ReLU)** activation function, explained in the next section, avoids this pitfall by not firing at times.

The Rectified Linear Unit (ReLU)

ReLU can let big numbers pass through. This makes a few neurons stale and they don't fire. This increases the sparsity, and hence, it is good. The ReLU maps input x to $\max(0, x)$, that is, they map negative inputs to 0, and positive inputs are output without any change as shown in the following graph:

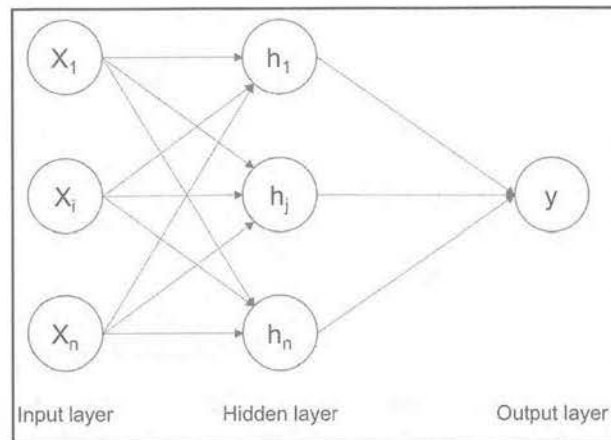


Because ReLU doesn't fire all the time, it can be trained faster. Since the function is simple, it is computationally the least expensive. Choosing the activation function is very dependent on the application. Nevertheless, ReLU works well for a large range of problems. In the next section, you will learn how to stack several perceptrons together that can learn more complex functions than perceptron.

Artificial neural network (ANN)

ANN is a collection of perceptrons and activation functions. The perceptrons are connected to form hidden layers or units. The hidden units form the nonlinear basis that maps the input layers to output layers in a lower-dimensional space, which is also called artificial neural networks. ANN is a map from input to output. The map is computed by weighted addition of the inputs with biases. The values of weight and bias values along with the architecture are called model.

The training process determines the values of these weights and biases. The model values are initialized with random values during the beginning of the training. The error is computed using a loss function by contrasting it with the ground truth. Based on the loss computed, the weights are tuned at every step. The training is stopped when the error cannot be further reduced. The training process learns the features during the training. The features are a better representation than the raw images. The following is a diagram of an artificial neural network, or multi-layer perceptron:



Several inputs of x are passed through a hidden layer of perceptrons and summed to the output. The universal approximation theorem suggests that such a neural network can approximate any function. The hidden layer can also be called a dense layer. Every layer can have one of the activation functions described in the previous section. The number of hidden layers and perceptrons can be chosen based on the problem. There are a few more things that make this multilayer perceptron work for multi-class classification problems. A multi-class classification problem tries to discriminate more than ten categories. We will explore those terms in the following sections.

One-hot encoding

One-hot encoding is a way to represent the target variables or classes in case of a classification problem. The target variables can be converted from the string labels to one-hot encoded vectors. A one-hot vector is filled with 1 at the index of the target class but with 0 everywhere else. For example, if the target classes are cat and dog, they can be represented by $[1, 0]$ and $[0, 1]$, respectively. For 1,000 classes, one-hot vectors will be of size 1,000 integers with all zeros but 1. It makes no assumptions about the similarity of target variables. With the combination of one-hot encoding with softmax explained in the following section, multi-class classification becomes possible in ANN.

Softmax

Softmax is a way of forcing the neural networks to output the sum of 1. Thereby, the output values of the softmax function can be considered as part of a probability distribution. This is useful in multi-class classification problems. Softmax is a kind of activation function with the speciality of output summing to 1. It converts the outputs to probabilities by dividing the output by summation of all the other values. The Euclidean distance can be computed between softmax probabilities and one-hot encoding for optimization. But the cross-entropy explained in the next section is a better cost function to optimize.

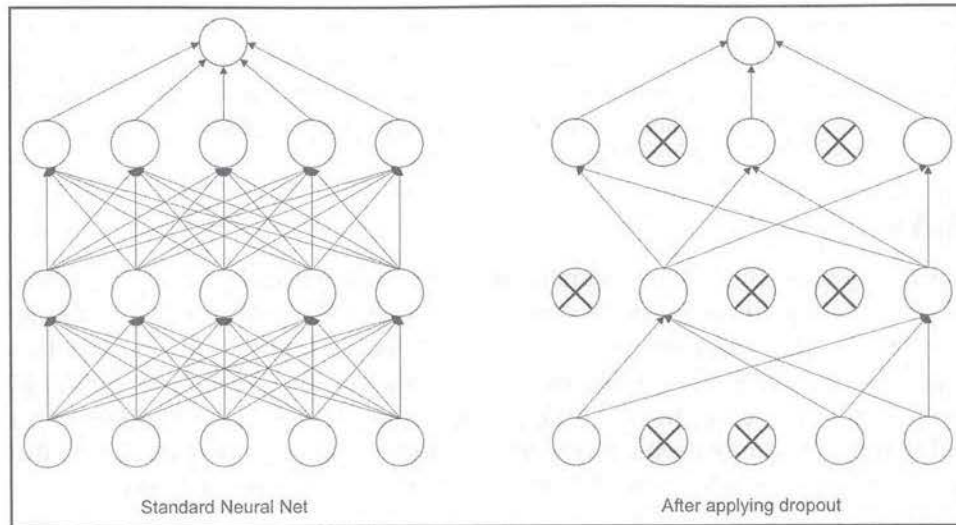
Cross-entropy

Cross-entropy compares the distance between the outputs of softmax and one-hot encoding. Cross-entropy is a loss function for which error has to be minimized. Neural networks estimate the probability of the given data to every class. The probability has to be maximized to the correct target label. Cross-entropy is the summation of negative logarithmic probabilities. Logarithmic value is used for numerical stability. Maximizing a function is equivalent to minimizing the negative of the same function. In the next section, we will see the following regularization methods to avoid the overfitting of ANN:

- Dropout
- Batch normalization
- L1 and L2 normalization

Dropout

Dropout is an effective way of regularizing neural networks to avoid the overfitting of ANN. During training, the dropout layer cripples the neural network by removing hidden units stochastically as shown in the following image:



Note how the neurons are randomly trained. Dropout is also an efficient way of combining several neural networks. For each training case, we randomly select a few hidden units so that we end up with different architectures for each case. This is an extreme case of bagging and model averaging. Dropout layer should not be used during the inference as it is not necessary.

Batch normalization

Batch normalization, or batch-norm, increase the stability and performance of neural network training. It normalizes the output from a layer with zero mean and a standard deviation of 1. This reduces overfitting and makes the network train faster. It is very useful in training complex neural networks.

L1 and L2 regularization

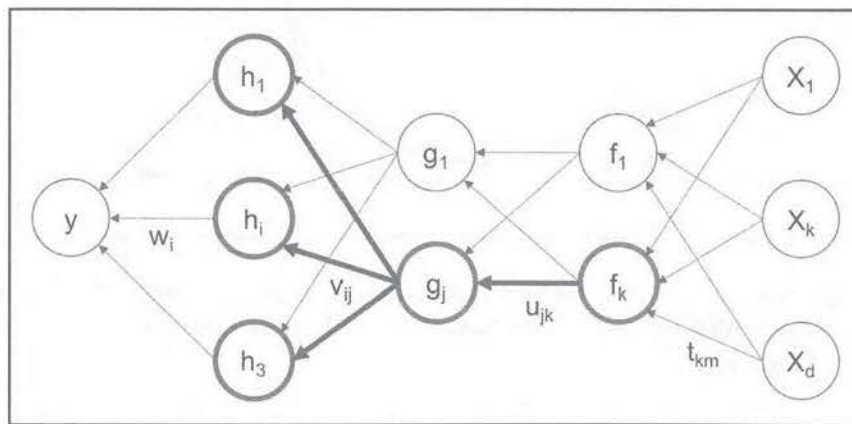
L1 penalizes the absolute value of the weight and tends to make the weights zero. L2 penalizes the squared value of the weight and tends to make the weight smaller during the training. Both the regularizes assume that models with smaller weights are better.

Training neural networks

Training ANN is tricky as it contains several parameters to optimize. The procedure of updating the weights is called backpropagation. The procedure to minimize the error is called optimization. We will cover both of them in detail in the next sections.

Backpropagation

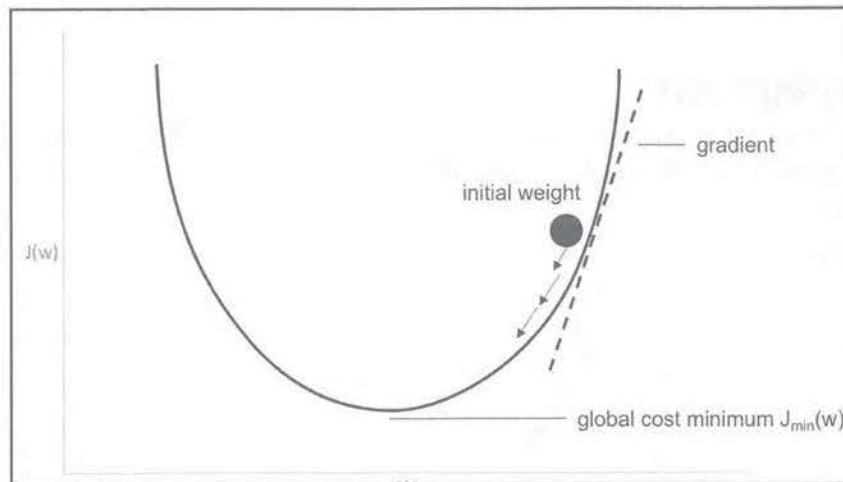
A backpropagation algorithm is commonly used for training artificial neural networks. The weights are updated from backward based on the error calculated as shown in the following image:



After calculating the error, gradient descent can be used to calculate the weight updating, as explained in the next section.

Gradient descent

The gradient descent algorithm performs multidimensional optimization. The objective is to reach the global maximum. Gradient descent is a popular optimization technique used in many machine-learning models. It is used to improve or optimize the model prediction. One implementation of gradient descent is called the **stochastic gradient descent (SGD)** and is becoming more popular (explained in the next section) in neural networks. Optimization involves calculating the error value and changing the weights to achieve that minimal error. The direction of finding the minimum is the negative of the gradient of the loss function. The gradient descent procedure is qualitatively shown in the following figure:



The learning rate determines how big each step should be. Note that the ANN with nonlinear activations will have local minima. SGD works better in practice for optimizing non-convex cost functions.

Stochastic gradient descent

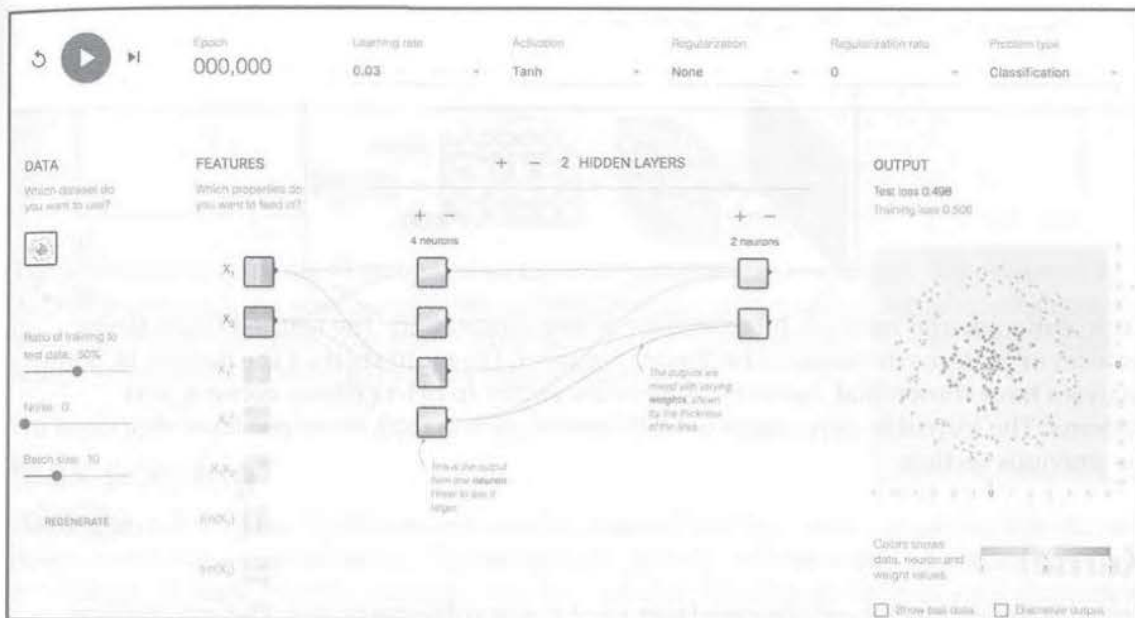
SGD is the same as gradient descent, except that it is used for only partial data to train every time. The parameter is called mini-batch size. Theoretically, even one example can be used for training. In practice, it is better to experiment with various numbers. In the next section, we will discuss convolutional neural networks that work better on image data than the standard ANN.



Visit <https://yihui.name/animation/example/grad-desc/> to see a great visualization of gradient descent on convex and non-convex surfaces.

Playing with TensorFlow playground

TensorFlow playground is an interactive visualization of neural networks. Visit <http://playground.tensorflow.org/>, play by changing the parameters to see how the previously mentioned terms work together. Here is a screenshot of the playground:

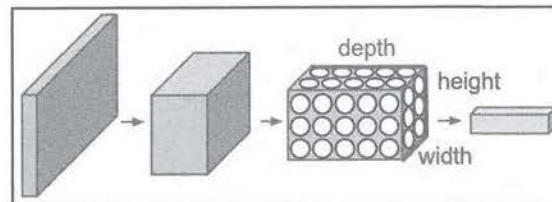


Dashboard in the TensorFlow playground

As shown previously, the reader can change learning rate, activation, regularization, hidden units, and layers to see how it affects the training process. You can spend some time adjusting the parameters to get the intuition of how neural networks for various kinds of data.

Convolutional neural network

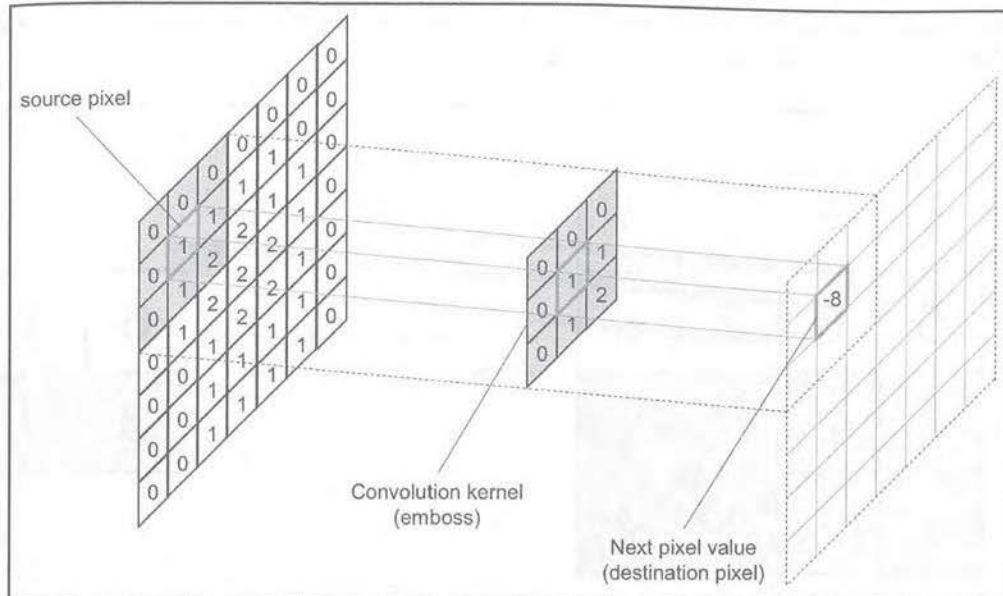
Convolutional neural networks (CNN) are similar to the neural networks described in the previous sections. CNNs have weights, biases, and outputs through a nonlinear activation. Regular neural networks take inputs and the neurons fully connected to the next layers. Neurons within the same layer don't share any connections. If we use regular neural networks for images, they will be very large in size due to a huge number of neurons, resulting in overfitting. We cannot use this for images, as images are large in size. Increase the model size as it requires a huge number of neurons. An image can be considered a volume with dimensions of height, width, and depth. Depth is the channel of an image, which is red, blue, and green. The neurons of a CNN are arranged in a volumetric fashion to take advantage of the volume. Each of the layers transforms the input volume to an output volume as shown in the following image:



Convolution neural network filters encode by transformation. The learned filters detect features or patterns in images. The deeper the layer, the more abstract the pattern is. Some analyses have shown that these layers have the ability to detect edges, corners, and patterns. The learnable parameters in CNN layers are less than the dense layer described in the previous section.

Kernel

Kernel is the parameter convolution layer used to convolve the image. The convolution operation is shown in the following figure:

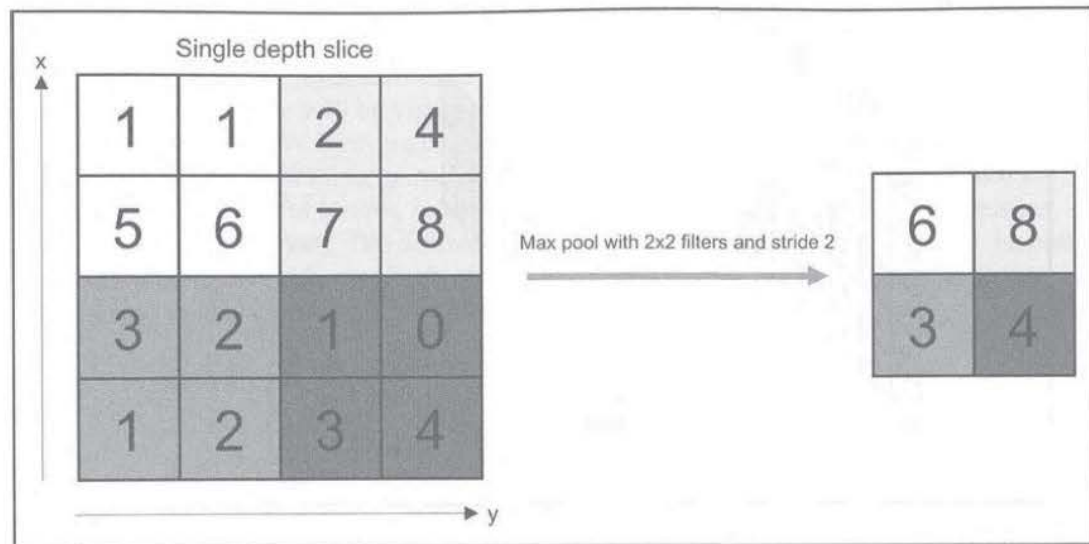


The kernel has two parameters, called stride and size. The size can be any dimension of a rectangle. Stride is the number of pixels moved every time. A stride of length 1 produces an image of almost the same size, and a stride of length 2 produces half the size. Padding the image will help in achieving the same size of the input.

Max pooling

Pooling layers are placed between convolution layers. Pooling layers reduce the size of the image across layers by sampling. The sampling is done by selecting the maximum value in a window. Average pooling averages over the window. Pooling also acts as a regularization technique to avoid overfitting. Pooling is carried out on all the channels of features. Pooling can also be performed with various strides.

The size of the window is a measure of the receptive field of CNN. The following figure shows an example of max pooling:



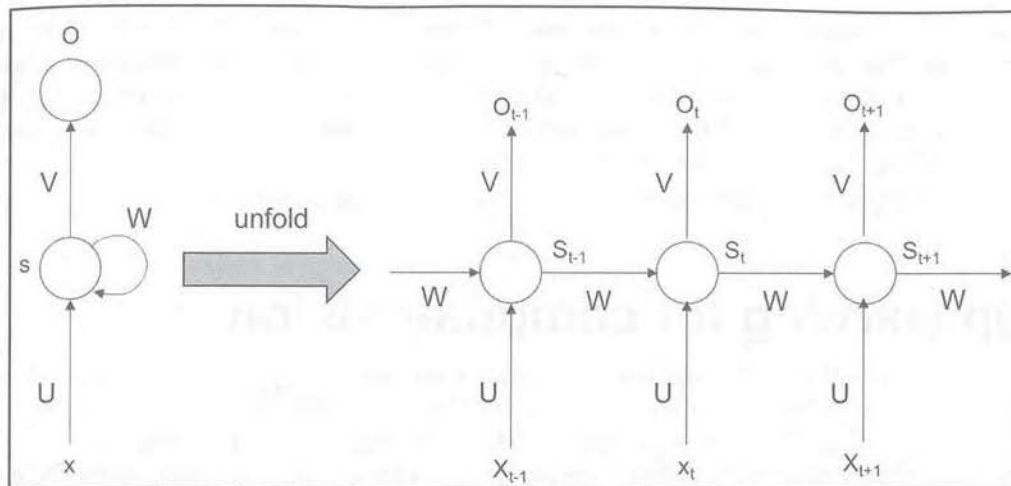
CNN is the single most important component of any deep learning model for computer vision. It won't be an exaggeration to state that it will be impossible for any computer to have vision without a CNN. In the next sections, we will discuss a couple of advanced layers that can be used for a few applications.



Visit <https://www.youtube.com/watch?v=jaajksuQW4mc> for a great visualization of a CNN and max-pooling operation.

Recurrent neural networks (RNN)

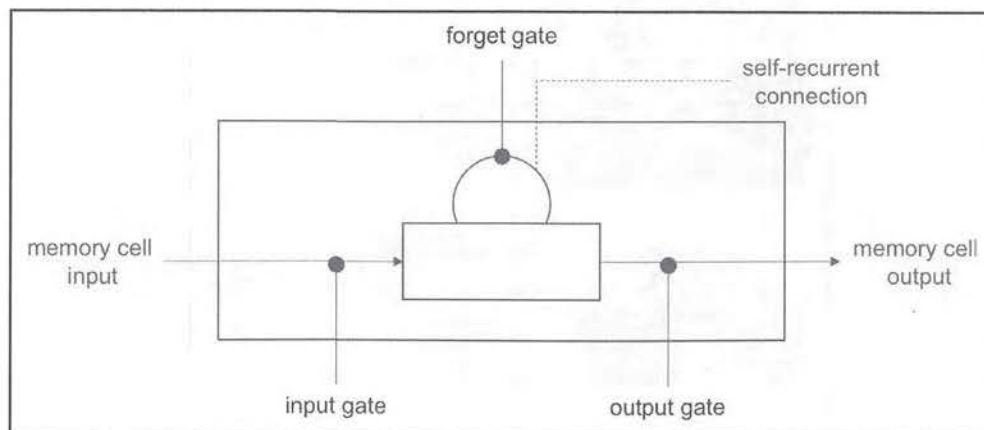
Recurrent neural networks (RNN) can model sequential information. They do not assume that the data points are intensive. They perform the same task from the output of the previous data of a series of sequence data. This can also be thought of as memory. RNN cannot remember from longer sequences or time. It is unfolded during the training process, as shown in the following image:



As shown in the preceding figure, the step is unfolded and trained each time. During backpropagation, the gradients can vanish over time. To overcome this problem, Long short-term memory can be used to remember over a longer time period.

Long short-term memory (LSTM)

Long short-term memory (LSTM) can store information for longer periods of time, and hence, it is efficient in capturing long-term efficiencies. The following figure illustrates how an LSTM cell is designed:



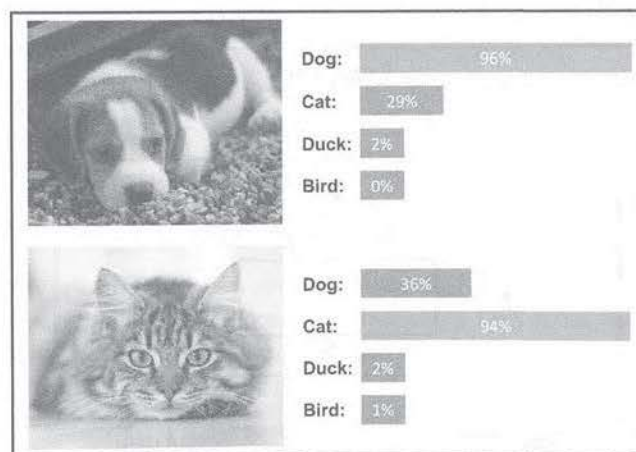
LSTM has several gates: forget, input, and output. Forget gate maintains the information previous state. The input gate updates the current state using the input. The output gate decides the information be passed to the next state. The ability to forget and retain only the important things enables LSTM to remember over a longer time period. You have learned the deep learning vocabulary that will be used throughout the book. In the next section, we will see how deep learning can be used in the context of computer vision.

Deep learning for computer vision

Computer vision enables the properties of human vision on a computer. A computer could be in the form of a smartphone, drones, CCTV, MRI scanner, and so on, with various sensors for perception. The sensor produces images in a digital form that has to be interpreted by the computer. The basic building block of such interpretation or intelligence is explained in the next section. The different problems that arise in computer vision can be effectively solved using deep learning techniques.

Classification

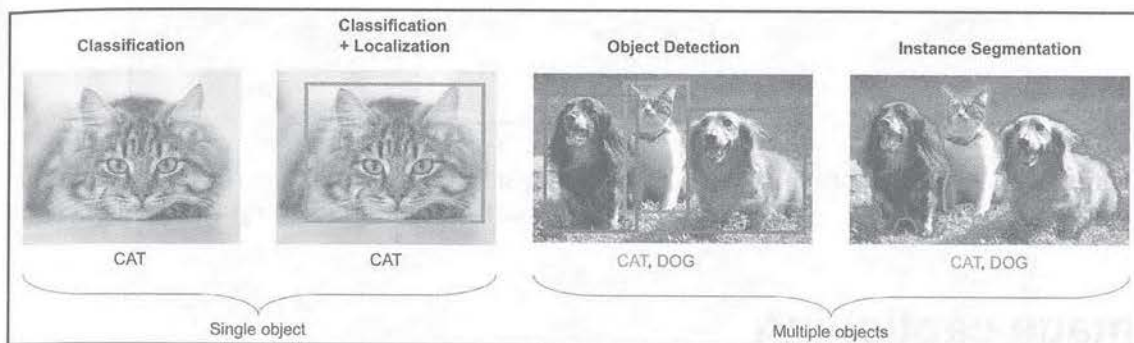
Image classification is the task of labelling the whole image with an object or concept with confidence. The applications include gender classification given an image of a person's face, identifying the type of pet, tagging photos, and so on. The following is an output of such a classification task:



The Chapter 2, *Image Classification*, covers in detail the methods that can be used for classification tasks and in Chapter 3, *Image Retrieval*, we use the classification models for visualization of deep learning models and retrieve similar images.

Detection or localization and segmentation

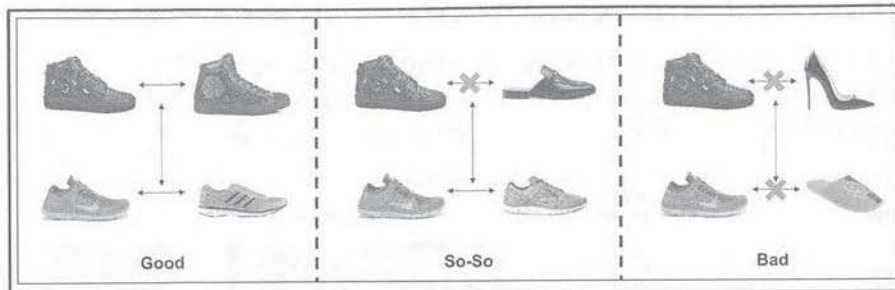
Detection or localization is a task that finds an object in an image and localizes the object with a bounding box. This task has many applications, such as finding pedestrians and signboards for self-driving vehicles. The following image is an illustration of detection:



Segmentation is the task of doing pixel-wise classification. This gives a fine separation of objects. It is useful for processing medical images and satellite imagery. More examples and explanations can be found in Chapter 4, *Object Detection* and Chapter 5, *Image Segmentation*.

Similarity learning

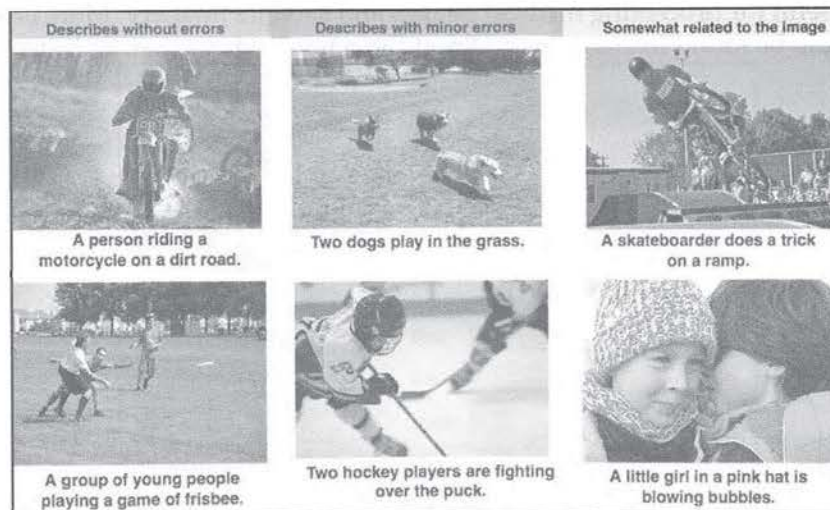
Similarity learning is the process of learning how two images are similar. A score can be computed between two images based on the semantic meaning as shown in the following image:



There are several applications of this, from finding similar products to performing the facial identification. Chapter 6, *Similarity learning*, deals with similarity learning techniques.

Image captioning

Image captioning is the task of describing the image with text as shown [below] here:

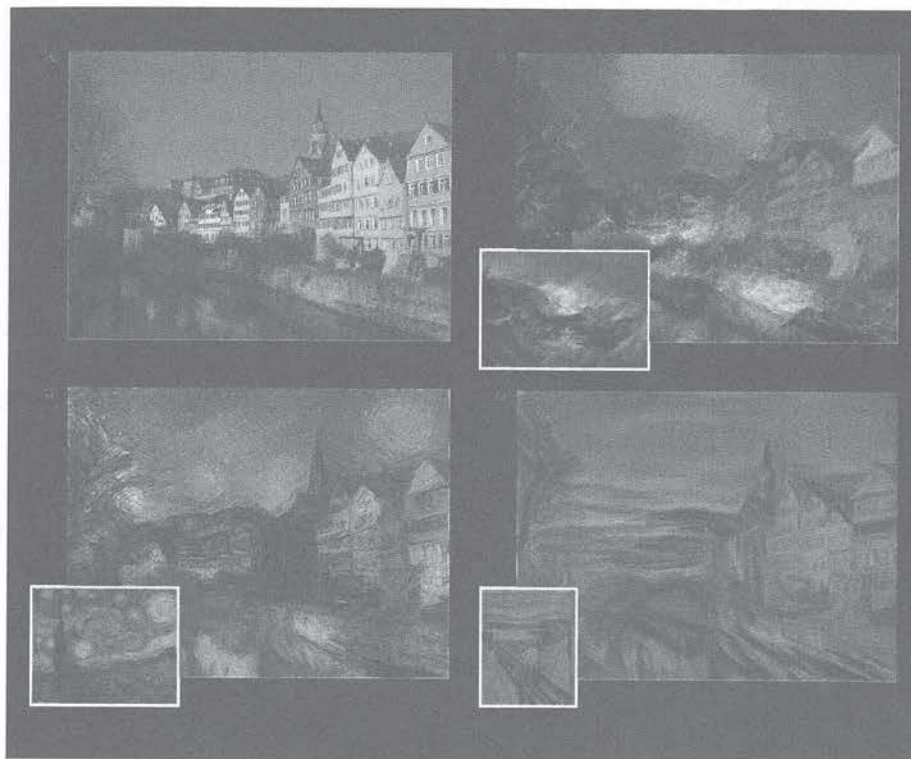


Reproduced with permission from Vinyals et al.

The Chapter 8, *Image Captioning*, goes into detail about image captioning. This is a unique case where techniques of **natural language processing (NLP)** and computer vision have to be combined.

Generative models

Generative models are very interesting as they generate images. The following is an example of style transfer application where an image is generated with the content of that image and style of other images:



Reproduced with permission from Gatys et al.

Images can be generated for other purposes such as new training examples, super-resolution images, and so on. The [Chapter 7, Generative Models](#), goes into detail of generative models.

Video analysis

Video analysis processes a video as a whole, as opposed to images as in previous cases. It has several applications, such as sports tracking, intrusion detection, and surveillance cameras. [Chapter 9, Video Classification](#), deals with video-specific applications. The new dimension of temporal data gives rise to lots of interesting applications. In the next section, we will see how to set up the development environment.

Development environment setup

In this section, we will set up the programming environment that will be useful for following the examples in the rest of the book. Readers may have the following choices of Operating Systems:

- **Development Operating Systems(OS)** such as Mac, Ubuntu, or Windows
- **Deployment Operating Systems** such as Mac, Windows, Android, iOS, or Ubuntu installed in Cloud platform such as **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)**, Azure, Tegra, Raspberry Pi

Irrespective of the platforms, all the code developed in this book should run without any issues. In this chapter, we will cover the installation procedures for the development environment. In [Chapter 10, Deployment](#), we will cover installation for deployment in various other environments, such as AWS, GCP, Azure, Tegra, and Raspberry Pi.

Hardware and Operating Systems - OS

For the development environment, you need to have a lot of computing power as training is significantly computationally expensive. Mac users are rather limited to computing power. Windows and Ubuntu users can beef up their development environment with more processors and **General Purpose - Graphics Processing Unit (GP-GPU)**, which will be explained in the next section.

General Purpose - Graphics Processing Unit (GP-GPU)

GP-GPUs are special hardware that speeds up the training process of training deep learning models. The GP-GPUs supplied by NVIDIA company are very popular for deep learning training and deployment as it has well-matured software and community support. Readers can set up a machine with such a GP-GPU for faster training. There are plenty of choices available, and the reader can choose one based on budget. It is also important to choose the RAM, CPU, and hard disk corresponding to the power of the GP-GPU. After the installation of the hardware, the following drivers and libraries have to be installed. Readers who are using Mac, or using Windows/Ubuntu without a GP-GPU, can skip the installation.

The following are the libraries that are required for setting up the environment:

- **Computer Unified Device Architecture (CUDA)**
- **CUDA Deep Neural Network (CUDNN)**

Computer Unified Device Architecture - CUDA

CUDA is the API layer provided by NVIDIA, using the parallel nature of the GPU. When this is installed, drivers for the hardware are also installed. First, download the CUDA library from the NVIDIA-portal: <https://developer.nvidia.com/cuda-downloads>.

Getting Started

Go through the instructions on the page, download the driver, and follow the installation instructions. Here is the screenshot of Ubuntu CUDA and the installation instructions:

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System	Windows	Linux	Mac OS X			
Architecture	x86_64	ppc64le				
Distribution	Fedora	OpenSUSE	RHEL	CentOS	SLES	Ubuntu
Version	17.04	16.04				
Installer Type	runfile (local)	deb (local)	deb (network)	cluster (local)		

Download Installer for Linux Ubuntu 16.04 x86_64

The base installer is available for download below.

> Base Installer

Download (2.8 KB)

Installation Instructions:

1. `sudo dpkg -i cuda-repo-ubuntu1604_9.0.176-1_amd64.deb`
2. `sudo apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub`
3. `sudo apt-get update`
4. `sudo apt-get install cuda`

Other installation options are available in the form of meta-packages. For example, to install all the library packages, replace "cuda" with the "cuda-libraries-9-0" meta package. For more information on all the available meta packages click [here](#).

The CUDA Toolkit contains Open-Source Software. The source code can be found [here](#).

The checksums for the installer and patches can be found in [Installer Checksums](#).

For further information, see the [Installation Guide for Linux](#) and the [CUDA Quick Start Guide](#).

These commands would have installed the `cuda-drivers` and the other CUDA APIs required.



You can check whether the drivers are properly installed by typing `nvidia-smi` in the command prompt.

CUDA Deep Neural Network - CUDNN

The CUDNN library provides primitives for deep learning algorithms. Since this package is provided by NVIDIA, it is highly optimized for their hardware and runs faster. Several standard routines for deep learning are provided in this package. These packages are used by famous deep learning libraries such as `tensorflow`, `caffe`, and so on. In the next section, instructions are provided for installing CUDNN. You can download CUDNN from the NVIDIA portal at <https://developer.nvidia.com/rdp/cudnn-download>.



User account is required (free signup).

Copy the relevant files to the CUDA folders, making them faster to run on GPUs. We will not use CUDA and CUDNN libraries directly. Tensorflow uses these to work on GP-GPU with optimized routines.

Installing software packages

There are several libraries required for trained deep learning models. We will install the following libraries and see the reason for selecting the following packages over the competing packages:

- Python and other dependencies
- OpenCV
- TensorFlow
- Keras

Python

Python is the de-facto choice for any data science application. It has the largest community and support ecosystem of libraries. TensorFlow API for Python is the most complete, and hence, Python is the natural language of choice. Python has two versions—Python2.x and Python3.x. In this book, we will discuss Python3.x. There are several reasons for this choice:

- Python 2.x development will be stopped by 2020, and hence, Python3.x is the future of Python

- Python 3.x avoids many design flaws in the original implementation
- Contrary to popular belief, Python3.x has as many supporting libraries for data science as Python 2.x.

We will use Python version 3 throughout this book. Go to <https://www.python.org/downloads/> and download version 3 according to the OS. Install Python by following the steps given in the download link. After installing Python, **pip3** has to be installed for easy installation of Python packages. Then install the several Python packages by entering the following command, so that you can install **OpenCV** and **tensorflow** later:

```
sudo pip3 install numpy scipy scikit-learn pillow h5py
```

The description of the preceding installed packages is given as follows:

- **numpy** is a highly-optimized numerical computation package. It has a powerful N-dimensional package array object, and the matrix operations of **numpy** library are highly optimized for speed. An image can be stored as a 3-dimensional **numpy** object.
- **scipy** has several routines for scientific and engineering calculations. We will use some optimization packages later in the book.
- **scikit-learn** is a machine-learning library from which we will use many helper functions.
- **Pillow** is useful for image loading and basic operations.
- **H5py** package is a Pythonic interface to the HDF5 binary data format. This is the format to store models trained using Keras.

Open Computer Vision - OpenCV

The **OpenCV** is a famous computer vision library. There are several image processing routines available in this library that can be of great use. Following is the step of installing **OpenCV** in Ubuntu.

```
sudo apt-get install python-opencv
```

Similar steps can be found for other OSes at <https://opencv.org/>. It is cross-platform and optimized for CPU-intensive applications. It has interfaces for several programming languages and is supported by Windows, Ubuntu, and Mac.

The TensorFlow library

The `tensorflow` is an open source library for the development and deployment of deep learning models. TensorFlow uses computational graphs for data flow and numerical computations. In other words, data, or tensor, flows through the graph, thus the name `tensorflow`. The graph has nodes that enable any numerical computation and, hence, are suitable for deep learning operations. It provides a single API for all kinds of platforms and hardware. TensorFlow handles all the complexity of scaling and optimization at the backend. It was originally developed for research at Google. It is the most famous deep learning library, with a large community and comes with tools for visualization and deployment in production.

Installing TensorFlow

Install `tensorflow` using `pip3` for the CPU using the following command:

```
sudo pip3 install tensorflow
```

If you are using GPU hardware and have installed `CUDA` and `CUDNN`, install the GPU version of the `tensorflow` with the following command:

```
sudo pip3 install tensorflow-gpu
```

Now the `tensorflow` is installed and ready for use. We will try out a couple of examples to understand how TensorFlow works.

TensorFlow example to print Hello, TensorFlow

We will do an example using TensorFlow directly in the Python shell. In this example, we will print **Hello, TensorFlow** using TensorFlow.

1. Invoke Python from your shell by typing the following in the command prompt:

```
python3
```

2. Import the `tensorflow` library by entering the following command:

```
>>> import tensorflow as tf
```

3. Next, define a constant with the string `Hello, TensorFlow`. This is different from the usual Python assignment operations as the value is not yet initialized:

```
>>> hello = tf.constant('Hello, TensorFlow!')
```

4. Create a session to initialize the computational graph, and give a name to the session:

```
>>> session = tf.Session()
```

The session can be run with the variable `hello` as the parameter.

5. Now the graph executes and returns that particular variable that is printed:

```
>>> print(session.run(hello))
```

It should print the following:

```
Hello, TensorFlow!
```

Let us look at one more example to understand how the session and graph work.



Visit <https://github.com/rajacheers/DeepLearningForComputerVision> to get the code for all the examples presented in the book. The code will be organised according to chapters. You can raise issues and get help in the repository.

TensorFlow example for adding two numbers

Here is another simple example of how TensorFlow is used to add two numbers.

1. Create a Python file and import `tensorflow` using the following code:

```
import tensorflow as tf
```

The preceding import will be necessary for all the latter examples. It is assumed that the reader has imported the library for all the examples. A `placeholder` can be defined in the following manner. The placeholders are not loaded when assigned. Here, a variable is defined as a `placeholder` with a type of `float32`. A `placeholder` is an empty declaration and can take values when a session is run.

2. Now we define a `placeholder` as shown in the following code:

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
```

3. Now the sum operation of the placeholders can be defined as a usual addition. Here, the operation is not executed but just defined using the following code:

```
z = x + y
```

4. The session can be created as shown in the previous example. The graph is ready for executing the computations when defined as shown below:

```
session = tf.Session()
```

5. Define the value of the placeholder in a dictionary format:

```
values = {x: 5.0, y: 4.0}
```

6. Run the session with variable `z` and the values. The graph feeds the values to appropriate placeholders and gets the value back for variable `z`:

```
result = session.run([z], values)
print(result)
```

This program should print `[9.0]` as the result of the addition.

It's understandable that this is not the best way to add two numbers. This example is to understand how tensors and operations are defined in TensorFlow. Imagine how difficult it will be to use a trillion numbers and add them. TensorFlow enables that scale with ease with the same APIs. In the next section, we will see how to install and use TensorBoard and TensorFlow serving.

TensorBoard

TensorBoard is a suite of visualization tools for training deep learning-based models with TensorFlow. The following data can be visualized in TensorBoard:

- **Graphs:** Computation graphs, device placements, and tensor details
- **Scalars:** Metrics such as loss, accuracy over iterations
- **Images:** Used to see the images with corresponding labels

- **Audio:** Used to listen to audio from training or a generated one
- **Distribution:** Used to see the distribution of some scalar
- **Histograms:** Includes histogram of weights and biases
- **Projector:** Helps visualize the data in 3-dimensional space
- **Text:** Prints the training text data
- **Profile:** Sees the hardware resources utilized for training

Tensorboard is installed along with TensorFlow. Go to the python3 prompt and type the following command, similar to the previous example, to start using Tensorboard:

```
x = tf.placeholder(tf.float32, name='x')
y = tf.placeholder(tf.float32, name='y')
z = tf.add(x, y, name='sum')
```

Note that an argument name has been provided as an extra parameter to placeholders and operations. These are names that can be seen when we visualize the graph. Now we can write the graph to a specific folder with the following command in TensorBoard:

```
session = tf.Session()
summary_writer = tf.summary.FileWriter('/tmp/1', session.graph)
```

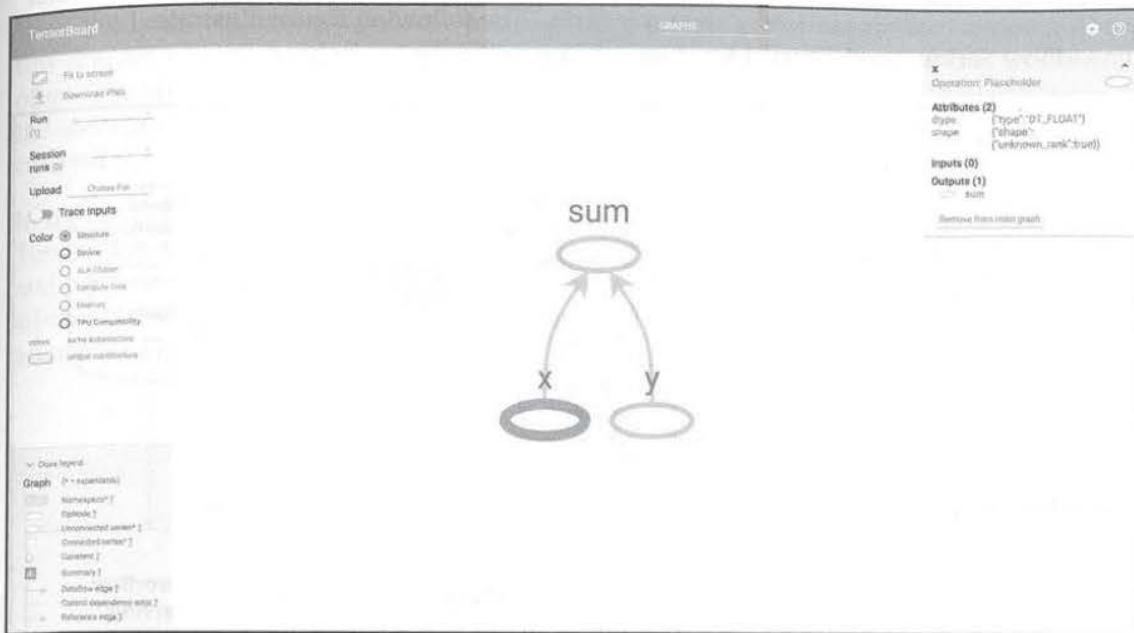
This command writes the graph to disk to a particular folder given in the argument. Now Tensorboard can be invoked with the following command:

```
tensorboard --logdir=/tmp/1
```

Any directory can be passed as an argument for the `logdir` option where the files are stored. Go to a browser and paste the following URL to start the visualization to access the TensorBoard:

```
http://localhost:6006/
```

The browser should display something like this:



The TensorBoard visualization in the browser window

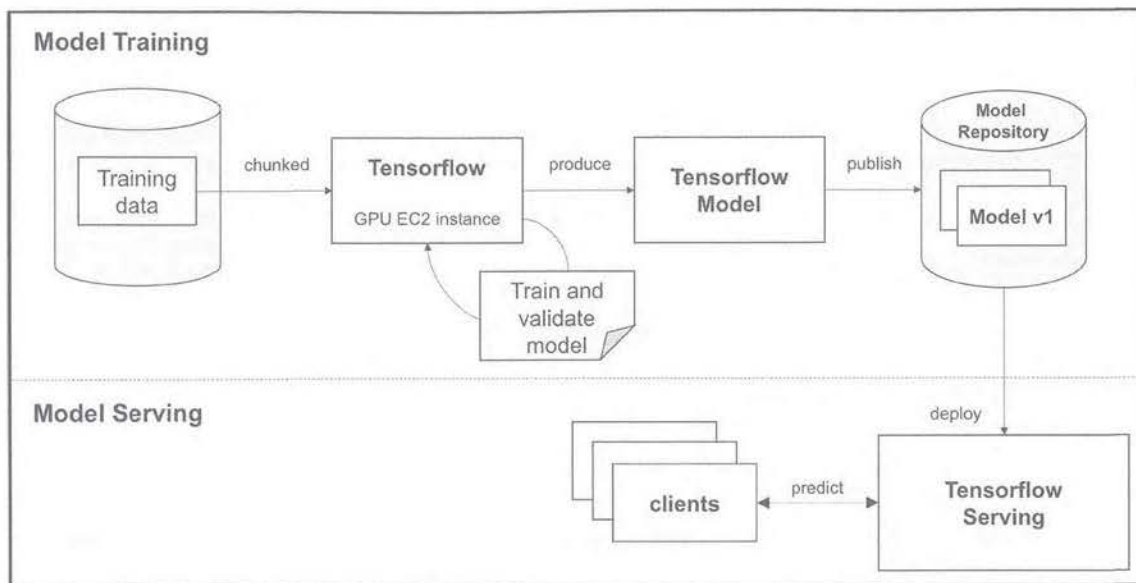
The graph of addition is displayed with the names given for the placeholders. When we click on them, we can see all the particulars of the tensor for that operation on the right side. Make yourself familiar with the tabs and options. There are several parts in this window. We will learn about them in different chapters. TensorBoard is one the best distinguishing tools in TensorFlow, which makes it better than any other deep learning framework.

The TensorFlow Serving tool

TensorFlow Serving is a tool in TensorFlow developed for deployment environments that are flexible, providing high latency and throughput environments. Any deep learning model trained with TensorFlow can be deployed with serving. Install the Serving by running the following command:

```
sudo apt-get install tensorflow-model-server
```

Step-by-step instructions on how to use serving will be described in Chapter 3, *Image Retrieval*. Note that the Serving is easy to install only in Ubuntu; for other OSes, please refer to <https://www.tensorflow.org/serving/setup>. The following figure illustrates how TensorFlow Serving and TensorFlow interact in production environments:



Many models can be produced by the training process, and Serving takes care of switching them seamlessly without any downtime. TensorFlow Serving is not required for all the following chapters, except for Chapter 3, *Image Retrieval* and Chapter 10, *Deployment*.

The Keras library

Keras is an open source library for deep learning written in Python. It provides an easy interface to use TensorFlow as a backend. Keras can also be used with Theano, deep learning 4j, or CNTK as its backend. Keras is designed for easy and fast experimentation by focusing on friendliness, modularity, and extensibility. It is a self-contained framework and runs seamlessly between CPU and GPU. Keras can be installed separately or used within TensorFlow itself using the `tf.keras` API. In this book, we will use the `tf.keras` API. We have seen the steps to install the required libraries for the development environment. Having CUDA, CUDNN, OpenCV, TensorFlow, and Keras installed and running smoothly is vital for the following chapters.

Summary

In this chapter, we have covered the basics of deep learning. The vocabulary introduced in this chapter will be used throughout this book, hence, you can refer back to this chapter often. The applications of computer vision are also shown with examples. Installations of all the software packages for various platforms for the development environment were also covered.

In the next chapter, we will discuss how to train classification models using both Keras and TensorFlow on a dataset. We will look at how to improve the accuracy using a bigger model and other techniques such as augmentation, and fine-tuning. Then, we will see several advanced models proposed by several people around the world, achieving the best accuracy in competitions.

2

Image Classification

Image classification is the task of classifying a whole image as a single label. For example, an image classification task could label an image as a dog or a cat, given an image is either a dog or a cat. In this chapter, we will see how to use TensorFlow to build such an image classification model and also learn the techniques to improve the accuracy.

We will cover the following topics in this chapter:

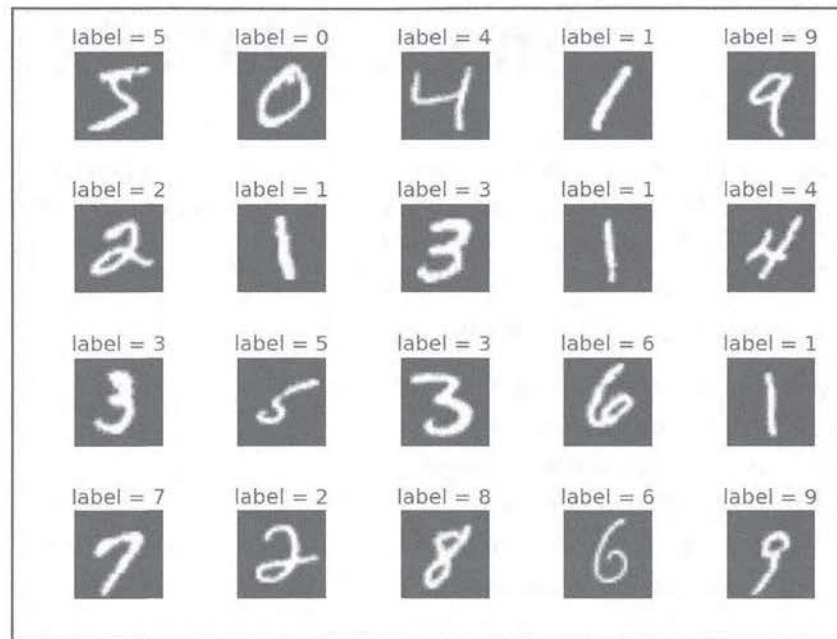
- Training the MNIST model in TensorFlow
- Training the MNIST model in Keras
- Other popular image testing datasets
- The bigger deep learning models
- Training a model for cats versus dogs
- Developing real-world applications

Training the MNIST model in TensorFlow

In this section, we will learn about the **Modified National Institute of Standards and Technology (MNIST)** database data and build a simple classification model. The objective of this section is to learn the general framework for deep learning and use TensorFlow for the same. First, we will build a perceptron or logistic regression model. Then, we will train a CNN to achieve better accuracy. We will also see how TensorBoard helps visualize the training process and understand the parameters.

The MNIST datasets

The MNIST data has handwritten digits from 0–9 with 60,000 images for training and 10,000 images for testing. This database is widely used to try algorithms with minimum preprocessing. It's a good and compact database to learn machine learning algorithms. This is the most famous database for image classification problems. A few examples are shown here:



As can be seen in the preceding figure, there are 10 labels for these handwritten characters. The images are normalized to the size of 28 image pixels by 28 image pixels, converted to grey size, and centered to a fixed size. This is a small dataset on which an algorithm can be quickly tested. In the next section, we will see how to load this dataset to be used in TensorFlow.

Loading the MNIST data

Load the MNIST data directly from TensorFlow. Note that we specify one hot encoding as an argument while loading the data. The label is stored as integers but should be loaded as one-hot encoding in-order to train. It is assumed that the reader is running the code from an editor with TensorFlow imported `tf` from this point onward. The following is the code snippet to load `MNIST_data`:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist_data = input_data.read_data_sets('MNIST_data', one_hot=True)
```

For the first run, the data will be downloaded and may take some time. From the second run, the cached data will be utilized. In the next section, we will build a perceptron to classify the digits.

Building a perceptron

A perceptron is a single-layer neural network. The concepts covered in this chapter, such as a fully connected layer, activation function, Stochastic Gradient Descent, logits, one hot encoding, softmax, and cross-entropy will be useful here. You will learn how to define these components of neural networks in TensorFlow and use the network to train the MNIST data.

Defining placeholders for input data and targets

A placeholder is a tensor where the data is passed. Placeholders aren't specific values but will receive input during computation. The input size of the perceptron, number of classes, batch size, and the total number of iterations or batches are first declared. `x_input` is the input where the images will be fed later. `y_input` is the placeholder where the one-shot labels or targets will be supplied as shown here:

```
input_size = 784
no_classes = 10
batch_size = 100
total_batches = 200

x_input = tf.placeholder(tf.float32, shape=[None, input_size])
y_input = tf.placeholder(tf.float32, shape=[None, no_classes])
```

The `None` in the shape argument indicates that it can be of any size as we have not yet defined the batch size. The second argument is the size of the tensor for `x_input` and the number of classes for `y_input`. Based on the type of placeholder, we have sent the data as floats. Next, we can define the perceptron.

Defining the variables for a fully connected layer

Let's define a simple linear classifier or perceptron by explaining the variables such as `weights` and `bias`. The values of these variables will be learned during computation. These are also referred to as parameters of the model. The weight variables are initialized with normal random distribution with the shape of input size and number of classes. The input size is 784 here as the image is reshaped into a single vector. The number of classes is 10 which is equal to the number of digits in the dataset. The bias variable is also initialized with random normal distribution with the size equal to the number of classes. The `weights` and `bias` are defined as follows:

```
weights = tf.Variable(tf.random_normal([input_size, no_classes]))
bias = tf.Variable(tf.random_normal([no_classes]))
```

The initialization of the variables can be zeroes but a random normal distribution gives a steady training. The inputs are then weighted and added with the bias to produce logits as shown next:

```
logits = tf.matmul(x_input, weights) + bias
```

The logits produced by the perceptron has to be compared against one-hot labels `y_input`. As learned in Chapter 1, *Getting Started*, it is better to use softmax coupled with cross-entropy for comparing logits and one-hot labels.

The `tf.nn.softmax_cross_entropy_with_logits` API from TensorFlow does this for us. The loss can be computed by averaging the cross-entropies. Then the cross-entropy is fed through gradient descent optimization done by `tf.train.GradientDescentOptimizer`. The optimizer takes the loss and minimizes it with a learning rate of 0.5. The computation of softmax, cross-entropy, loss, optimization is shown next:

```
softmax_cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
    labels=y_input, logits=logits)
loss_operation = tf.reduce_mean(softmax_cross_entropy)
optimiser = tf.train.GradientDescentOptimizer(
    learning_rate=0.5).minimize(loss_operation)
```

The softmax and cross-entropies are computed together from the `tf.nn` package, which has several other useful methods. The `tf.train` has several optimizers, and here, we are using the vanilla gradient descent. You can visit TensorFlow API documentation to understand other optional parameters. Until now, the placeholders, variables, and operations are defined and yet to be populated with tensors.



Read the list of optimizers available in TensorFlow at https://www.tensorflow.org/api_guides/python/train. The Adam optimizer is particularly useful for computer vision applications. It generally converges faster and we need not define a learning rate to start with. For a theoretical summary of optimizers, visit <http://ruder.io/optimizing-gradient-descent>.

Training the model with data

Now you have defined the model and training operation. The next step is to start training the model with the data. During training, the gradients are calculated and the weights are updated. The variables have not yet been initialized. Next, start the session and initialize the variables using a global variable initializer:

```
session = tf.Session()
session.run(tf.global_variables_initializer())
```

The preceding two lines are required for most of the examples in this book. It is assumed that the reader will use these two lines wherever required. Now the graph is ready to be fed with data and start training. Over a loop, read the data in batches and train the model. Training the model is carried out by running the session with the required tensors. The optimizer has to be called in order for the graph to update the weights:

```
for batch_no in range(total_batches):
    mnist_batch = mnist_data.train.next_batch(batch_size)
    _, loss_value = session.run([optimiser, loss_operation], feed_dict={
        x_input: mnist_batch[0],
        y_input: mnist_batch[1]
    })
    print(loss_value)
```

The first argument of the `run` method can have an array for which the outputs of the values are requested. We pass `loss` because printing `loss` tells us whether the model is getting trained or not. The `loss` is expected to decrease as we are minimizing the loss. The `feed_dict` is a Python dictionary used to directly feed the input and target labels to the placeholders. Once this loop ends, the `loss` should be generally lower than 0.46. Next, we can evaluate how well the model worked by computing the accuracy, as shown here:

```
predictions = tf.argmax(logits, 1)
correct_predictions = tf.equal(predictions, tf.argmax(y_input, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_predictions,
                                          tf.float32))

test_images, test_labels = mnist_data.test.images, mnist_data.test.labels
accuracy_value = session.run(accuracy_operation, feed_dict={
    x_input: test_images,
    y_input: test_labels
})
print('Accuracy : ', accuracy_value)
session.close()
```

The prediction should be the index of the maximum activation. It should be compared with the ground truth on MNIST labels for correct predictions. The accuracy is calculated using the average of correct predictions. The accuracy of the data can be evaluated by running the session with test data as the feed dictionary. When the whole program is run, it should finally produce an accuracy of around **90%**. The definition of the model may seem too explicit without simpler APIs for training and testing. This level of basic definition gives the power of expressiveness for TensorFlow. In the next sections, we will see higher level APIs. The accuracy obtained by the perceptron is not great, and in the next section, we will use a deeper network with convolution layers to improve the accuracy.

Building a multilayer convolutional network

In this section, we will see how to create a multilayer convolutional network in TensorFlow and watch how a deeper network improves classification accuracy. We will define the layers with TensorFlow layers' APIs rather than defining them from scratch. The best practice methods are engrained in those methods. The import of libraries, datasets, and placeholders can be followed from the previous section. This time, we will use TensorBoard for visualizing the training process. In order to visualize the statistics of the variables, the values of variable statistics have to be added to `tf.summary`.

The summaries will be written to a folder that is interpretable to TensorBoard. Let's define a function to write the summaries so that TensorBoard can be used to visualize them:

```
def add_variable_summary(tf_variable, summary_name):
    with tf.name_scope(summary_name + '_summary'):
        mean = tf.reduce_mean(tf_variable)
        tf.summary.scalar('Mean', mean)
        with tf.name_scope('standard_deviation'):
            standard_deviation = tf.sqrt(tf.reduce_mean(
                tf.square(tf_variable - mean)))
        tf.summary.scalar('StandardDeviation', standard_deviation)
        tf.summary.scalar('Maximum', tf.reduce_max(tf_variable))
        tf.summary.scalar('Minimum', tf.reduce_min(tf_variable))
        tf.summary.histogram('Histogram', tf_variable)
```

The variable summary function writes the summaries of a variable. There are five statistics added to the summaries: mean, standard deviation, maximum, minimum and histogram. Summaries can be either a scalar or a histogram. We will see how these values can be visualized in TensorBoard when logged for several variables. Unlike the previous model, we will resize the MNIST data into a square and use it like a two-dimensional image. The following is the command to reshape the image into 28 image pixels by 28 image pixels:

```
x_input_reshape = tf.reshape(x_input, [-1, 28, 28, 1],
    name='input_reshape')
```

The dimension -1 denotes that the batch size can be any number. Note that there is an argument called `name` that will be reflected in the TensorBoard graph for ease of understanding. We will define a 2D convolution layer where the input, filters, kernels, and activations are defined. This method can be called anywhere for further examples and is useful when the activation function has to have **Rectified Linear Unit (ReLU)** activation. The convolution function layer is defined as follows:

```
def convolution_layer(input_layer, filters, kernel_size=[3, 3],
    activation=tf.nn.relu):
    layer = tf.layers.conv2d(
        inputs=input_layer,
        filters=filters,
        kernel_size=kernel_size,
        activation=activation,
    )
    add_variable_summary(layer, 'convolution')
    return layer
```

There are default parameters for `kernel_size` and `activation`. The summaries are added to the layer within the function and the layer is returned. Whenever the function is called, `input_layer` has to be passed as a parameter. This definition will make our other code simple and small. In a very similar way, we will define a function for the `pooling_layer` as follows:

```
def pooling_layer(input_layer, pool_size=[2, 2], strides=2):
    layer = tf.layers.max_pooling2d(
        inputs=input_layer,
        pool_size=pool_size,
        strides=strides
    )
    add_variable_summary(layer, 'pooling')
    return layer
```

This layer has default parameters for `pool_size` and `strides` to be `[2, 2]` and `2` respectively. These parameters generally work well but can be changed when necessary. The summaries are added for this layer too. We will next define a dense layer as follows:

```
def dense_layer(input_layer, units, activation=tf.nn.relu):
    layer = tf.layers.dense(
        inputs=input_layer,
        units=units,
        activation=activation
    )
    add_variable_summary(layer, 'dense')
    return layer
```

The dense layer defined has default parameters for `activation` and variable summaries are added as well. The `pooling_layer` takes the feature map from the convolution layer and reduces it to half its size by skipping, using the pool size and strides. All these layers are connected as a graph and are just defined. None of the values is initialized. Another convolution layer can be added to transform the sampled features from the first convolution layer to better features. After pooling, we may reshape the activations to a linear fashion in order to be fed through dense layers:

```
convolution_layer_1 = convolution_layer(x_input_reshape, 64)
pooling_layer_1 = pooling_layer(convolution_layer_1)
convolution_layer_2 = convolution_layer(pooling_layer_1, 128)
pooling_layer_2 = pooling_layer(convolution_layer_2)
flattened_pool = tf.reshape(pooling_layer_2, [-1, 5 * 5 * 128],
                             name='flattened_pool')
dense_layer_bottleneck = dense_layer(flattened_pool, 1024)
```

The only difference between the convolution layers is the filter size. It's important that the dimensions change appropriately from layer to layer. Choosing the parameters for kernel and stride are arbitrary and these numbers are chosen by experience. Two convolution layers are defined, and this can be followed by a fully connected layer. A dense-layer API can take any vector of a single dimension and map it to any number of hidden units, as in this case is 1024. The hidden layer is followed by ReLU activation to make this a non-linear computation. Variable summaries are added for this layer as well. This is followed by a dropout layer with a rate of dropping out. Keeping this high will stop the network from learning. The training mode can be set to `True` and `False` based on when we use this. We will set this as `True` (default is `False`) for the training. We will have to change this while the accuracy is calculated. Hence, a bool is kept for this, that will be fed during training:

```
dropout_bool = tf.placeholder(tf.bool)
dropout_layer = tf.layers.dropout(
    inputs=dense_layer_bottleneck,
    rate=0.4,
    training=dropout_bool
)
```

The dropout layer is fed again to a dense layer, which is called logits. Logits is the final layer with activations leading to the number of classes. The activations will be spiked for a particular class, which is the target class, and can be obtained for a maximum of those 10 activations:

```
logits = dense_layer(dropout_layer, no_classes)
```

The logits output is very similar to the model created in the previous section. Now the logits can be passed through the softmax layer followed by the cross-entropy calculation as before. Here, we have added a scope name to get a better visualization in TensorBoard as follows:

```
with tf.name_scope('loss'):
    softmax_cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
        labels=y_input, logits=logits)
    loss_operation = tf.reduce_mean(softmax_cross_entropy, name='loss')
    tf.summary.scalar('loss', loss_operation)
```

This loss function can be optimized with `tf.train` APIs' methods. Here, we will use the Adam optimiser. The learning rate need not be defined and works well for most cases:

```
with tf.name_scope('optimiser'):
    optimiser = tf.train.AdamOptimizer().minimize(loss_operation)
```

The accuracy is calculated as before but name scopes are added for correct predictions and accuracy calculation:

```
with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        predictions = tf.argmax(logits, 1)
        correct_predictions = tf.equal(predictions, tf.argmax(y_input, 1))
    with tf.name_scope('accuracy'):
        accuracy_operation = tf.reduce_mean(
            tf.cast(correct_predictions, tf.float32))
    tf.summary.scalar('accuracy', accuracy_operation)
```

A scalar summary for accuracy is also added. The next step is to start the session and initialize the variables as in the previous section. The lines are not repeated here. The summaries have to be merged, and the files for writing the training and testing summaries have to be defined:

```
merged_summary_operation = tf.summary.merge_all()
train_summary_writer = tf.summary.FileWriter('/tmp/train', session.graph)
test_summary_writer = tf.summary.FileWriter('/tmp/test')
```

Note that the graph is written once with the `summary_writer`. The training is very similar to before except that the accuracy calculations while training and the values are added to the summaries. Next, the data can be loaded in batches and training can be started:

```
test_images, test_labels = mnist_data.test.images, mnist_data.test.labels

for batch_no in range(total_batches):
    mnist_batch = mnist_data.train.next_batch(batch_size)
    train_images, train_labels = mnist_batch[0], mnist_batch[1]
    _, merged_summary = session.run([optimiser, merged_summary_operation],
                                    feed_dict={
                                        x_input: train_images,
                                        y_input: train_labels,
                                        dropout_bool: True
                                    })
    train_summary_writer.add_summary(merged_summary, batch_no)
    if batch_no % 10 == 0:
        merged_summary, _ = session.run([merged_summary_operation,
                                         accuracy_operation], feed_dict={
                                            x_input: test_images,
                                            y_input: test_labels,
                                            dropout_bool: False
                                        })
        test_summary_writer.add_summary(merged_summary, batch_no)
```

Summaries are returned in every iteration for training data and are added to the writer. For every tenth iteration, the test summaries are added. Note that the dropout is enabled only during training and not during testing. We have completed the definition, and summaries of the network and this can be run. To see the training process, we can go to TensorBoard as described in Chapter 1, *Getting Started*.

Utilizing TensorBoard in deep learning

Once the TensorBoard is opened in the browser, go to the **Graphs** tab. The graph that we have defined and been getting trained should be displayed. Right-clicking on the nodes, we can choose the operations to be removed from the main graph. After some alignment, the graph should look as follows:

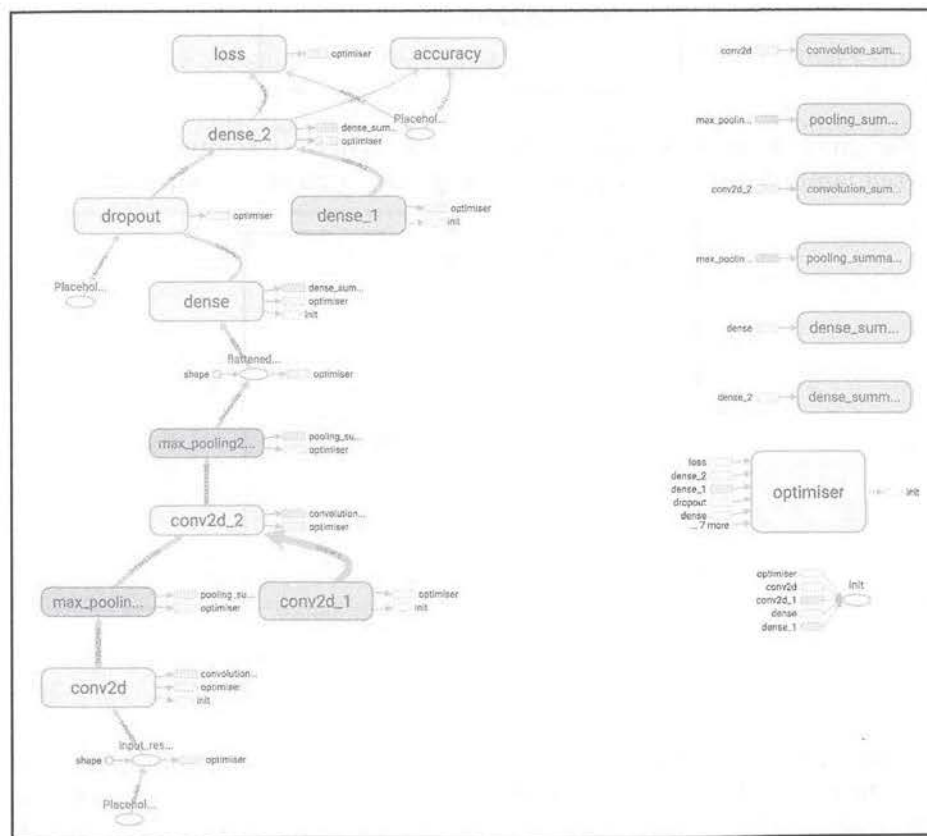
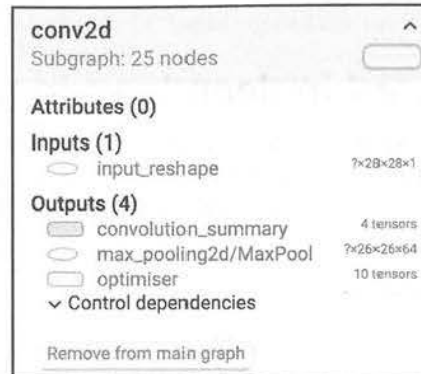


Figure illustrating the graphs that were trained and defined in the previous instance

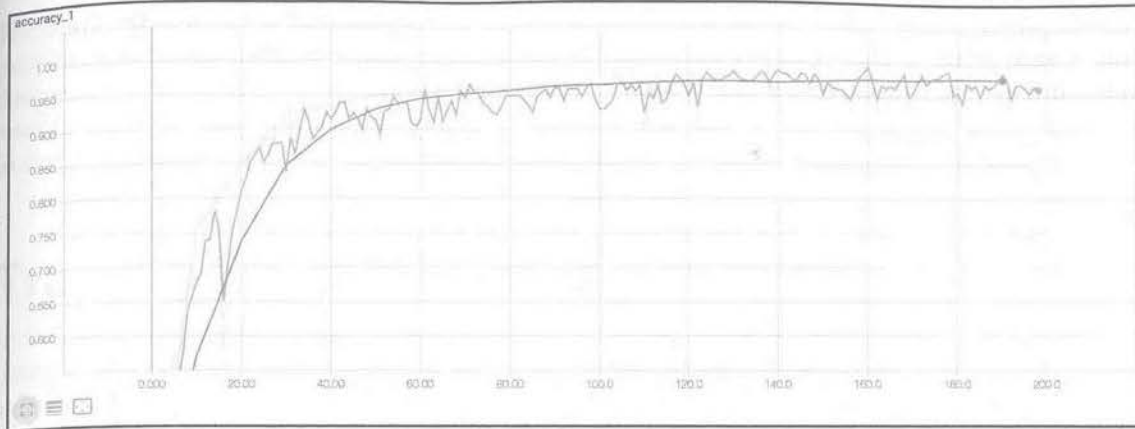
Notice how nicely all the layers that we have defined are displayed. This is very useful to check the definition of the architecture. The direction of the graph is nicely visualized with all the particulars. By clicking on each node, you can see the particulars of the node, such as input and output tensor shapes, as shown here:



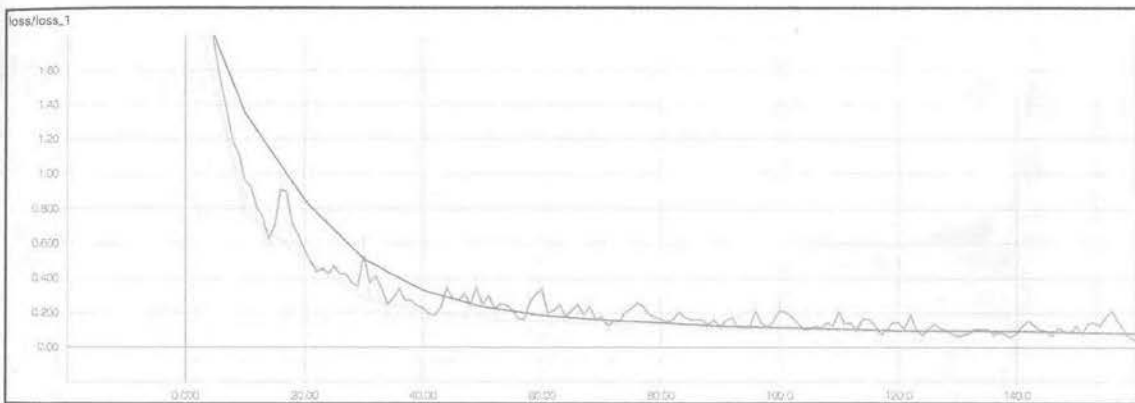
These values can be used to cross-check the definitions of the parameters of the layers. Make yourself familiar with this page by noticing the legend in the bottom left, as follows:



The name scopes are grouped and individual components can be seen by clicking on the plus sign on the nodes. The nodes are arranged by colors. Now we can move to the scalars page. By moving around the page, you can spot the accuracy graphs, as shown in the following figure:

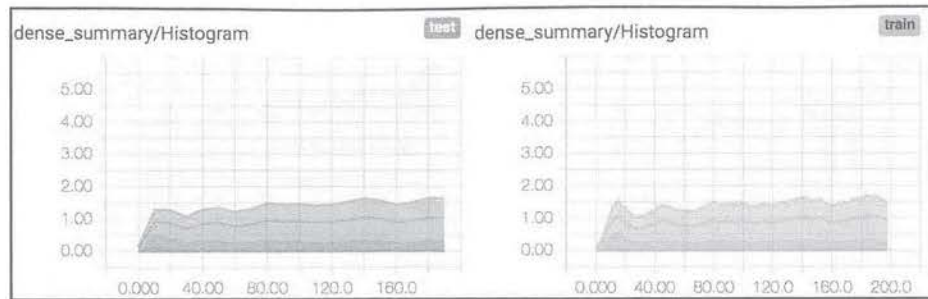


The orange line is for training data and the blue line for testing data. They roughly follow the same pattern. There are slightly lesser bright lines indicating the original values, whereas the brighter ones are smoothed curves. The smoothing coefficient can be selected in the UI. The accuracy of the test data has reached above 97%. The following is the figure from the loss summary:

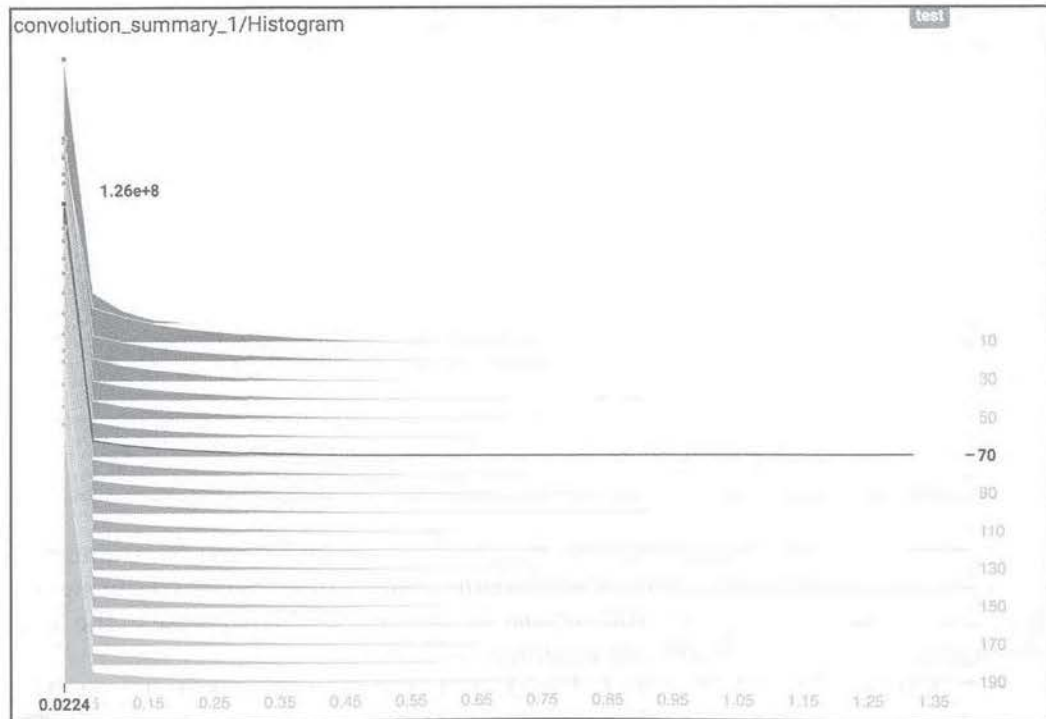


The loss was steadily decreasing for both training and testing data during the training process, which is a good sign. The data of all the summaries will be refreshed during the training process, and we can witness the increase in accuracy with a decrease in the loss to achieve an excellent result of 97.38% test accuracy.

This helps you see whether the model is learning and moving toward better results. The other summaries such as min, max, mean, and standard deviation are also useful. The following are the graphs of a dense layer:



These summaries are useful to notice the change in weights. These distributions can also be visualized as histograms, as follows:



These are the spreads of the weights of the logits. These are the beautiful visualizations possible with TensorBoard and are extremely helpful in training. By making the model deeper, we were able to witness a huge increase in accuracy. In the next section, we will see how to train the same model with Keras APIs. Now you can see the power of TensorBoard in inspecting the deep learning model and the training process.

Training the MNIST model in Keras

In this section, we will use the same model as defined in the previous section using `tf.keras` APIs. It is better to learn both Keras and layers packages from TensorFlow as they could be seen at several open source codes. The objective of the book is to make you understand various offerings of TensorFlow so that you can build products on top of it.

"Code is read more often than it is written."

Bearing in mind the preceding quote, you are shown how to implement the same model using various APIs. Open source code of any implementation of the latest algorithms will be a mix of these APIs. Next, we will start with the Keras implementation.

Preparing the dataset

The MNIST data is available with Keras. First, import `tensorflow`. Then define a few constants such as batch size, the classes, and the number of epochs. The batch size can be selected based on the RAM available on your machine. The higher the batch size, the more RAM required. The impact of the batch size on the accuracy is minimal. The number of classes is equal to 10 here and will be different for different problems. The number of epochs determines how many times the training has to go through the full dataset. If the loss is reduced at the end of all epochs, it can be set to a high number. In a few cases, training longer could give better accuracy. Let us now look at the steps to create the dataset:

1. Set the dimensions of the input images as well as follows:

```
batch_size = 128
no_classes = 10
epochs = 2
image_height, image_width = 28, 28
```

2. Load the data from disk to memory using the Keras utilities:

```
(x_train, y_train), (x_test, y_test) =  
tf.keras.datasets.mnist.load_data()
```

3. Reshape the vector into an image format, and define the input dimension for the convolution using the code given:

```
x_train = x_train.reshape(x_train.shape[0], image_height,  
image_width, 1)  
x_test = x_test.reshape(x_test.shape[0], image_height, image_width,  
1)  
input_shape = (image_height, image_width, 1)
```

4. Convert the data type to float as follows:

```
x_train = x_train.astype('float32')  
x_test = x_test.astype('float32')
```

5. Normalize the data by subtracting the mean of the data:

```
x_train /= 255  
x_test /= 255
```

6. Convert the categorical labels to one-shot encoding:

```
y_train = tf.keras.utils.to_categorical(y_train, no_classes)  
y_test = tf.keras.utils.to_categorical(y_test, no_classes)
```

This is very different from the TensorFlow way of writing the code. The data is loaded already in memory and none of the concepts of `Placeholders` is present here.

Building the model

In this section, we will use a few convolution layers followed by fully connected layers for training the preceding dataset. Construct a simple sequential model with two convolution layers followed by pooling, dropout, and dense layers. A sequential model has the `add` method to stack layers one above another. The first layer has 64 filters, and the second layers have 128 filters. The kernel size is 3 for all the filters. Apply the max pooling after the convolution layers. The output of the convolution layers is flattened connecting to a couple of fully connected layers with dropout connections.

The last layer is connected to softmax as this is a multiclass classification problem. The following code shows how to define the model:

```
def simple_cnn(input_shape):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Conv2D(
        filters=64,
        kernel_size=(3, 3),
        activation='relu',
        input_shape=input_shape
    ))
    model.add(tf.keras.layers.Conv2D(
        filters=128,
        kernel_size=(3, 3),
        activation='relu'
    ))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(rate=0.3))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(units=1024, activation='relu'))
    model.add(tf.keras.layers.Dropout(rate=0.3))
    model.add(tf.keras.layers.Dense(units=no_classes,
        activation='softmax'))
    model.compile(loss=tf.keras.losses.categorical_crossentropy,
        optimizer=tf.keras.optimizers.Adam(),
        metrics=['accuracy'])
    return model
simple_cnn_model = simple_cnn(input_shape)
```

The model is just defined and has to be compiled. During compilation loss, optimizer and metrics have to be defined. The loss will be cross-entropy, optimized by the Adam algorithm, and we will report the accuracy as the metric. Using the loaded data, train and evaluate the data. Load the training data with the training parameters and fit the model:

```
simple_cnn_model.fit(x_train, y_train, batch_size, epochs, (x_test,
    y_test))
train_loss, train_accuracy = simple_cnn_model.evaluate(
    x_train, y_train, verbose=0)
print('Train data loss:', train_loss)
print('Train data accuracy:', train_accuracy)
```

A session is not created when Keras APIs are used. Then evaluate the test data as follows:

```
test_loss, test_accuracy = simple_cnn_model.evaluate(
    x_test, y_test, verbose=0)
print('Test data loss:', test_loss)
print('Test data accuracy:', test_accuracy)
```

The evaluation is also created without any explicit creation of the session. After finishing the run, the result should look similar to the following:

```
Loss for train data: 0.0171295607952
Accuracy of train data: 0.9950166666667
Loss for test data: 0.0282736890309
Accuracy of test data: 0.9902
```

This should give a better accuracy of 99% on the test data. Note that the training accuracy is higher than the test data, and it's always a good practice to print both of them. The difference in accuracy is due to the number of iterations. The accuracy is a bit more than the previous model created in TensorFlow because of the difference in the dataset.

Other popular image testing datasets

The MNIST dataset is the most commonly used dataset for testing the algorithms. But there are other datasets that are used to test image classification algorithms.

The CIFAR dataset

The Canadian Institute for Advanced Research (CIFAR)-10 dataset has 60,000 images with 50,000 images for training and 10,000 images for testing. The number of classes is 10. The image dimension is 32 pixels by 32 pixels. The following are randomly selected images from each of the class:

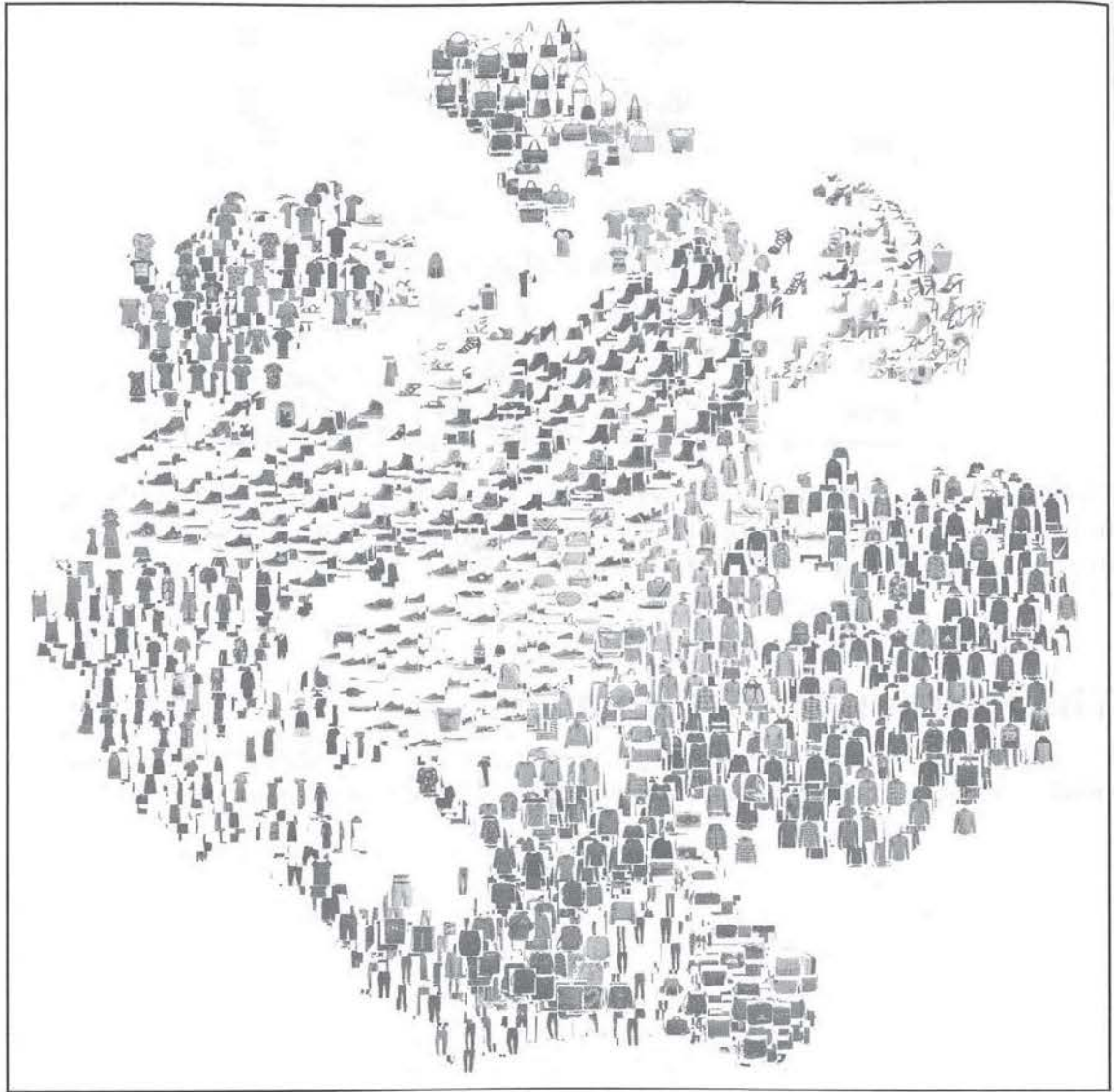


The images are tiny and just contain one object. The CIFAR-100 dataset contains the same number of images but with 100 classes. Hence, there are only 600 images per class. Each image comes with a super label and a fine label. This dataset is available at `tf.keras.datasets` if you wish to experiment.

The Fashion-MNIST dataset

Fashion-MNIST is a dataset created as an alternative to the MNIST dataset. This dataset created as MNIST is considered as too easy and this can be directly replaced with MNIST.

The following is randomly selected examples from the dataset after **principal component analysis (PCA)** is performed:

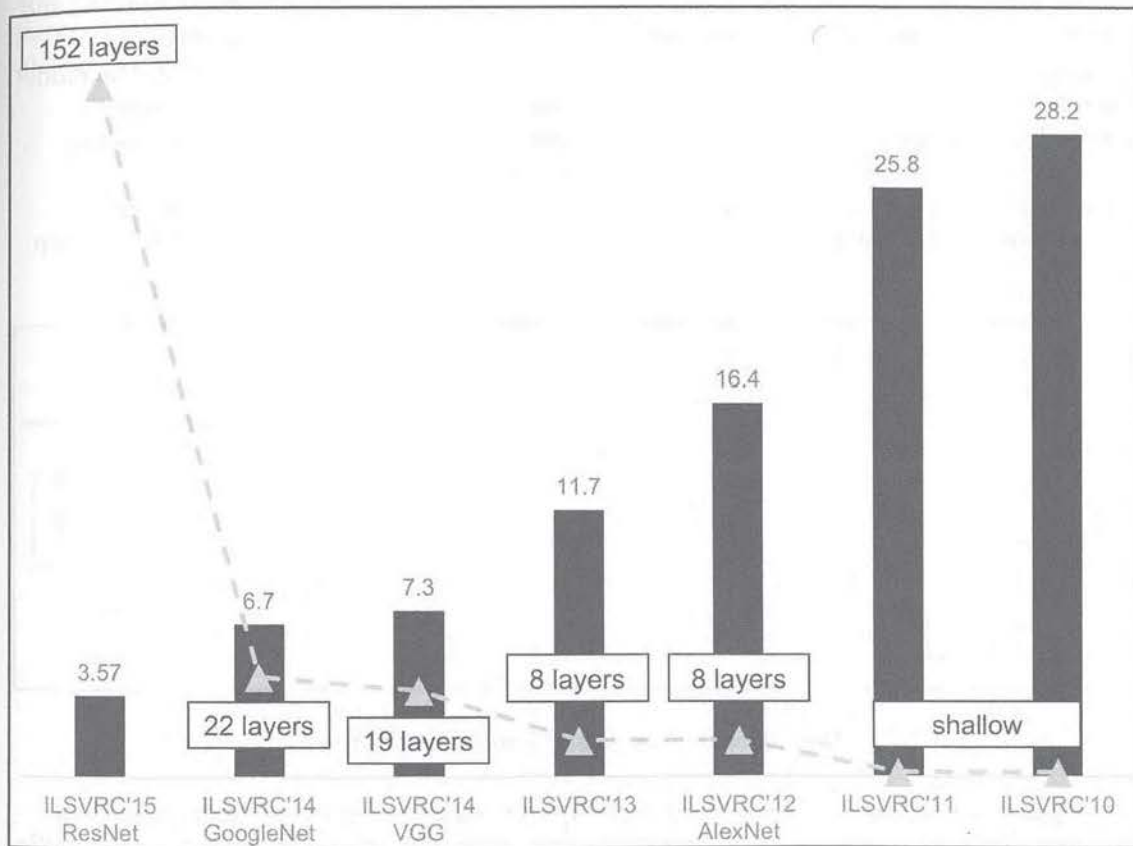


The dataset size, number of labels and image size are similar to MNIST. More details can be found at <https://github.com/zalandoresearch/fashion-mnist>. You can run the models learned previously and check the accuracy.

The ImageNet dataset and competition

ImageNet is a computer vision dataset with 14,197,122 images, 21,841 Synsets indexed. Synset is a node in WordNet hierarchy that in turn is a set of synonyms. There is a competition held every year with 1,000 classes from this dataset. It has been the standard benchmark for assessing the performance of image classification algorithms.

In 2013, a deep learning based computer vision model got the number one spot. From then, only deep learning models have won the competition. The following is the top five error rate over the years in the competition:



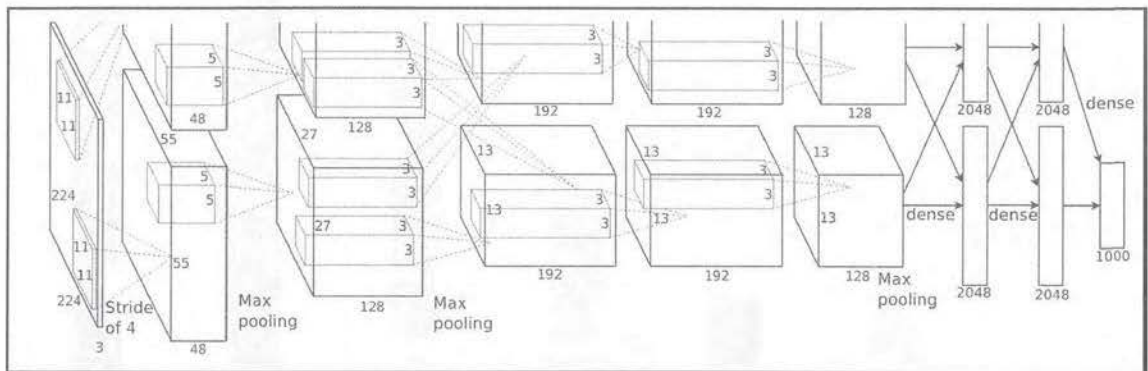
You can notice that the accuracy has been increasing over the years, as well as the depth of the layers. Next, we will understand the models that are present in this graph.

The bigger deep learning models

We will go through several model definitions that have achieved state-of-the-art results in the ImageNet competitions. We will look at them individually on the following topics.

The AlexNet model

AlexNet is the first publication that started a wide interest in deep learning for computer vision. Krizhevsky et al. (<https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>) proposed AlexNet and it has been a pioneer and influential in this field. This model won the ImageNet 2013 challenge. The error rate was 15.4%, which was significantly better than the next. The model was relatively a simple architecture with five convolution layers. The challenge was to classify 1,000 categories of objects. The image and data had 15 million annotated images with over 22,000 categories. Out of them, only a 1,000 categories are used for the competition. AlexNet used ReLU as the activation function and found it was training several times faster than other activation functions. The architecture of the model is shown here:



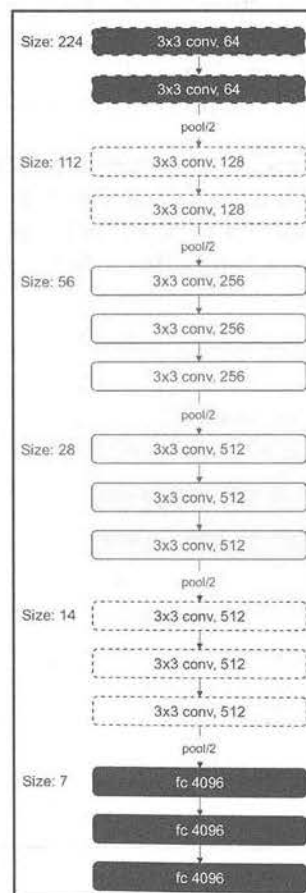
Reproduced with permission from Krizhevsky et al.

The paper also used data augmentation techniques such as image translations, horizontal flips, and random cropping. The dropout layer prevents overfitting. The model used vanilla **Stochastic Gradient Descent (SGD)** for training. The parameters of SGD are chosen carefully for training. The learning rate changes over a fixed set of training iterations. The momentum and weight decay take fixed values for training. There is a concept called **Local Response Normalization (LRN)** introduced in this paper. The LRN layers normalize every pixel across the filters to avoid huge activation in a particular filter.

This layer is not used anymore as recent research suggests that there is not much improvement because of LRN. AlexNet has 60 million parameters in total.

The VGG-16 model

The VGG model stands for the **Visual Geometry Group** from Oxford. The model was very simple and had a greater depth than AlexNet. The paper had two models with 16 and 19 layers depth. All the CNN layers were using 3 by 3 filters with stride and a pad of size 1 and a max pooling size of 2 with stride 2. This resulted in a decrease in the number of parameters. Though the size is decreasing because of max pooling, the number of filters is increasing with layers. The architecture of the 16-layer deep model is as follows:

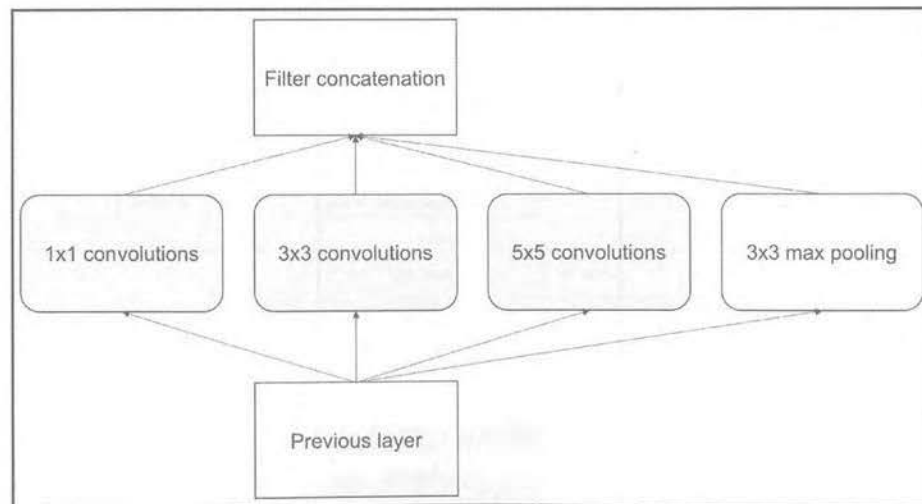


This model has 138 million parameters and is the largest of all the models described here. But the uniformity of parameters is quite good. The characteristic is such that, as deep as the network gets, the smaller the image is with an increased number of filters. One of the data augmentation techniques used was scale jittering. Scale jittering is an augmentation technique where a side with random size is considered to vary the scales.

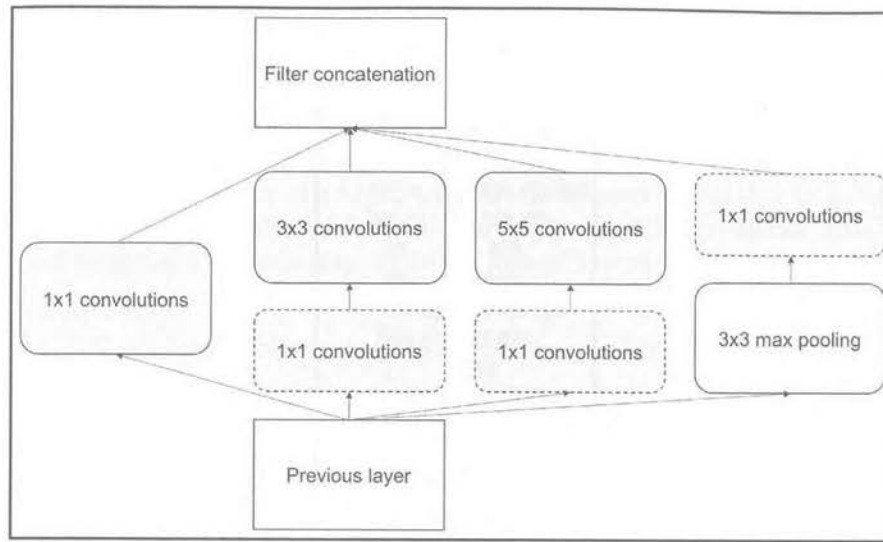
The Google Inception-V3 model

Inception-V3 was proposed by Szegedy et al. (<https://arxiv.org/pdf/1409.4842.pdf>) and introduced the concept of inception that has a better way of generalization. This was the architecture that won the ImageNet competition in 2014. It is geared towards efficiency for speed and size. It has 12 times lesser parameters than AlexNet. Inception is the micro-architecture on which a macro-architecture is built. Each hidden layer has a higher-level representation of the image. At each layer, we have an option of using pooling or other layers. Instead of using one type of kernel, inception uses several kernels. An average pooling is followed by various size convolutions and then they are concatenated.

The kernel parameters can be learned based on the data. Using several kernels, the model can detect small features as well as higher abstractions. The 1×1 convolution will reduce the feature and, hence, computations. This takes less RAM during inference. The following is the inception module in its simplest form where there are options of convolutions with various kernel sizes and pooling:



Notice that operations are happening in parallel, as opposed to AlexNet or VGG. The output volume is huge, and hence, 1×1 filters are introduced for dimensionality reduction. When the reduced dimensions are added to the architecture it becomes as follows:



The whole architecture of the model is as follows with all the bells and whistles:

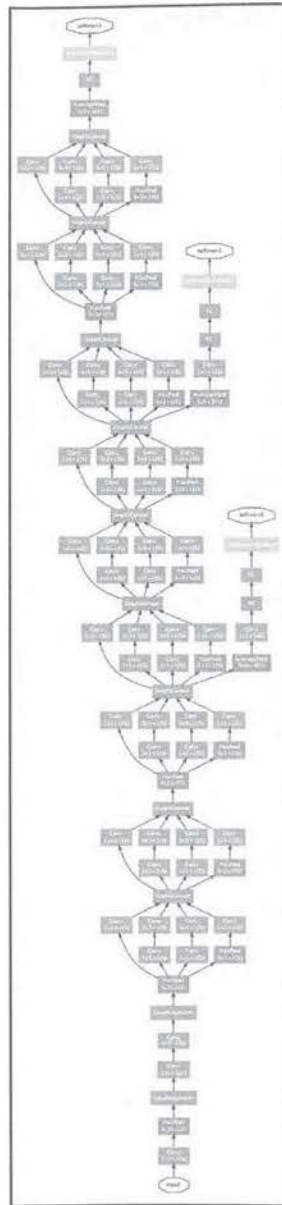
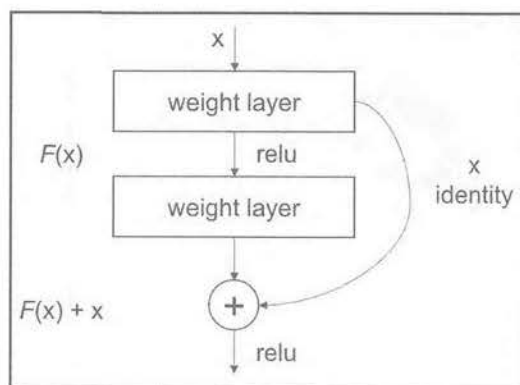


Figure illustrating the Google Inception V3 model architecture [Reproduced with permission from Szegedy et al.]

There are nine inception modules with a total of 100 layers and they achieve good performance.

The Microsoft ResNet-50 model

ResNet was proposed by He et al. (<https://arxiv.org/pdf/1512.03385.pdf>) and won the ImageNet competition in 2015. This method showed that deeper networks can be trained. The deeper the network, the more saturated the accuracy becomes. It's not even due to overfitting or due to the presence of a high number of parameters, but due to a reduction in the training error. This is due to the inability to backpropagate the gradients. This can be overcome by sending the gradients directly to the deeper layers with a residual block as follows:



Every two layers are connected forming a residual block. You can see that the training is passed between the layers. By this technique, the backpropagation can carry the error to earlier layers.

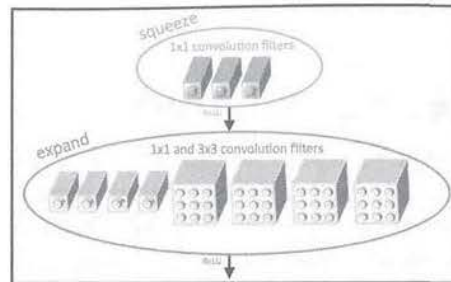


The model definitions can be used from https://github.com/tensorflow/tensorflow/tree/r1.4/tensorflow/python/keras/_impl/keras/applications. Every layer in the model is defined and pre-trained weights on the ImageNet dataset are available.

The SqueezeNet model

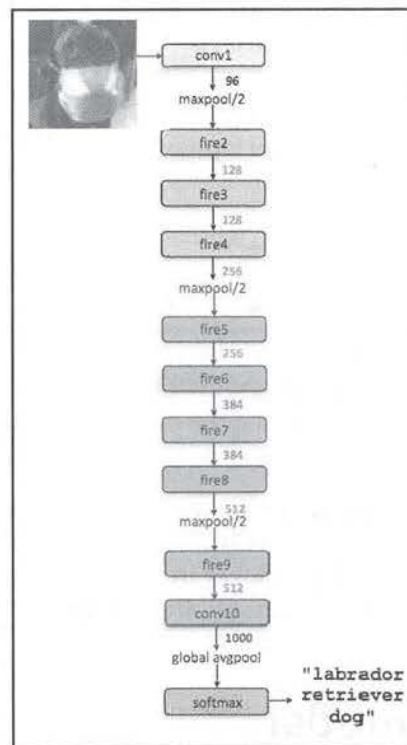
The **SqueezeNet** model was introduced by Iandola et al. (<https://arxiv.org/pdf/1602.07360.pdf>), to reduce the model size and the number of parameters.

The network was made smaller by replacing 3×3 filters with 1×1 filters as shown here:



Reproduced with permission from Iandola et al.

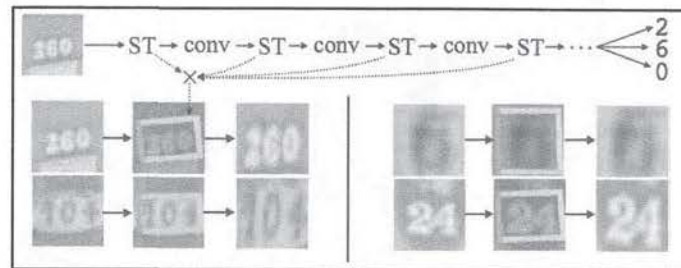
The number of inputs of the 3×3 filters has also reduced downsampling of the layers when happening at the higher level, providing large activation maps:



Reproduced with permission from Iandola et al.

Spatial transformer networks

The **spatial transformer networks** proposed by Jaderberg et al. (<https://arxiv.org/pdf/1506.02025.pdf>) try to transform the image before passing to the CNN. This is different from other networks because it tries to modify the image before convolution. This network learns the parameters to transform the image. The parameters are learned for an **affine transformation**. By applying an affine transformation, **spatial invariance** is achieved. In the previous networks, spatial invariance was achieved by max-pooling layers. The placement of spatial transformer networks is shown as follows:

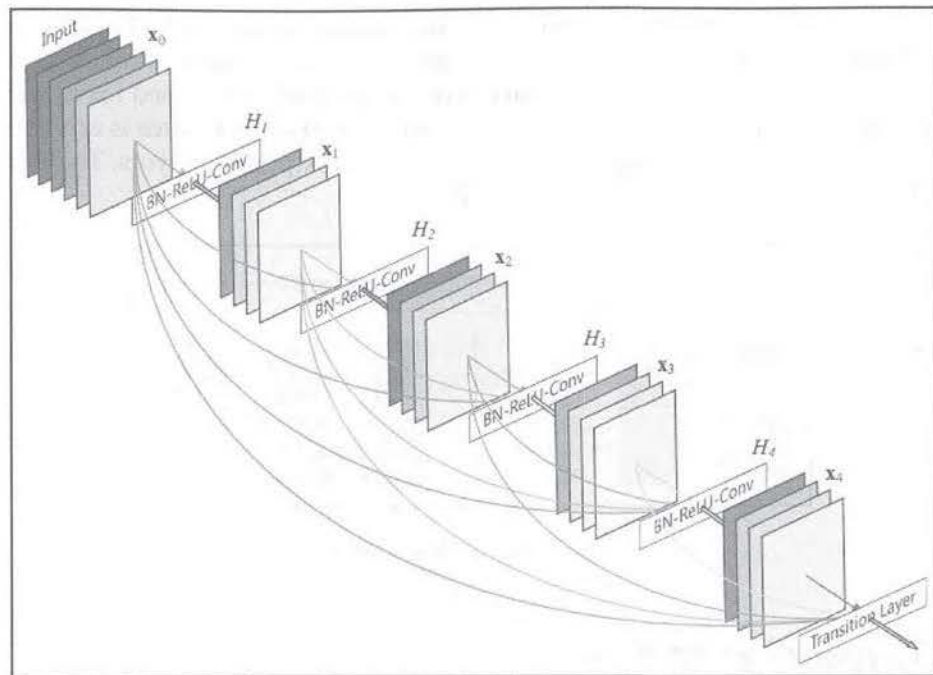


Reproduced with permission from Jaderberg et al.

The DenseNet model

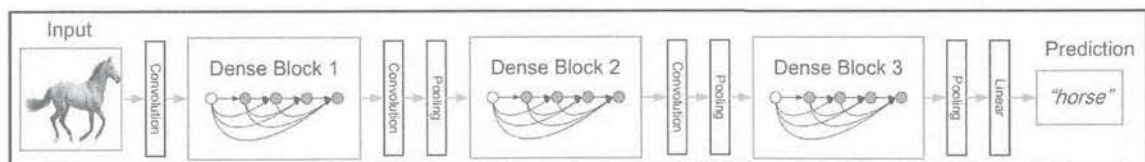
DenseNet is an extension of ResNet proposed by Huang et al. (<https://arxiv.org/pdf/1608.06993.pdf>). In ResNet blocks, the previous layer is merged into the future layer by summation. In DenseNet, the previous layer is merged into the future layer by concatenation. DenseNet connects all the layers to the previous layers and the current layer to the following layers.

In the following diagram, it can be seen how the feature maps are supplied as input to the other layers:



Reproduced with permission from Huang et al.

This way, it provides several advantages such as smoother gradients, feature transformation and so on. This also reduces the number of parameters:



Reproduced with permission from Huang et al.

We have covered all the latest algorithms for the image classification task. Any of the architectures can be used for an image classification task. In the next section, we will see how to train a model to predict pets, using these advanced architectures and improve the accuracy.

Training a model for cats versus dogs

In this section, we will prepare and train a model for predicting cats versus dogs and understand some techniques which increase the accuracy. Most of the image classification problems come into this paradigm. Techniques covered in this section, such as augmentation and transfer learning, are useful for several problems.

Preparing the data

For the purpose of classification, we will download the data from **kaggle** and store in an appropriate format. Sign up and log in to www.kaggle.com and go to <https://www.kaggle.com/c/dogs-vs-cats/data>. Download the `train.zip` and `test1.zip` files from that page. The `train.zip` file contains 25,000 images of pet data. We will use only a portion of the data to train a model. Readers with more computing power, such as a **Graphics Processing Unit (GPU)**, can use more data than suggested. Run the following script to rearrange the images and create the necessary folders:

```
import os
import shutil

work_dir = '' # give your correct directory
image_names = sorted(os.listdir(os.path.join(work_dir, 'train'))))

def copy_files(prefix_str, range_start, range_end, target_dir):
    image_paths = [os.path.join(work_dir, 'train', prefix_str + '.' +
                                str(i) + '.jpg')
                    for i in range(range_start, range_end)]
    dest_dir = os.path.join(work_dir, 'data', target_dir, prefix_str)
    os.makedirs(dest_dir)
    for image_path in image_paths:
        shutil.copy(image_path, dest_dir)

copy_files('dog', 0, 1000, 'train')
copy_files('cat', 0, 1000, 'train')
copy_files('dog', 1000, 1400, 'test')
copy_files('cat', 1000, 1400, 'test')
```

For our experiments, we will use only 1,000 images of cats and dogs. So, copy images 0–999 from the downloaded folder to the newly created `train` folder under `cats`. Similarly, copy 1,000–1,400 to `data/test/cat`, 10–999 in `train/dogs` and 1,000–1,400 in `data/test/dog` so that we have 1,000 training examples for each class and 400 validation examples for each class.

Benchmarking with simple CNN

Let's run the previous `simple_cnn` model on this dataset and see how it performs. This model's performance will be the basic benchmark against which we judge other techniques. We will define a few variables for data loading and training, as shown here:

```
image_height, image_width = 150, 150
train_dir = os.path.join(work_dir, 'train')
test_dir = os.path.join(work_dir, 'test')
no_classes = 2
no_validation = 800
epochs = 2
batch_size = 200
no_train = 2000
no_test = 800
input_shape = (image_height, image_width, 3)
epoch_steps = no_train // batch_size
test_steps = no_test // batch_size
```

This constant is used for the techniques discussed in this section of training a model for predicting cats and dogs. Here, we are using 2,800 images to train and test which is reasonable for a personal computer's RAM. But this is not sustainable for bigger datasets. It's better if we load only a batch of images at a time for training and testing. For this purpose, a `tf.keras` has a class called `ImageDataGenerator` that reads images whenever necessary. It is assumed that a `simple_cnn` model is imported from the previous section. The following is an example of using a generator for loading the images:

```
generator_train =
tf.keras.preprocessing.image.ImageDataGenerator(rescale=1. / 255)
generator_test = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1.
/ 255)
```

This definition also rescales the images when it is loaded. Next, we can read the images from the directory using the `flow_from_directory` method as follows:

```
train_images = generator_train.flow_from_directory(
    train_dir,
    batch_size=batch_size,
    target_size=(image_width, image_height))

test_images = generator_test.flow_from_directory(
    test_dir,
    batch_size=batch_size,
    target_size=(image_width, image_height))
```

The directory to load the images, size of batches and target size for the images are passed as an argument. This method performs the rescaling and passes the data in batches for fitting the model. This generator can be directly used for fitting the model. The method `fit_generator` of the model can be used as follows:

```
simple_cnn_model.fit_generator(
    train_images,
    steps_per_epoch=epoch_steps,
    epochs=epochs,
    validation_data=test_images,
    validation_steps=test_steps)
```

This model fits the data from the generator of training images. The number of epochs is defined from training, and validation data is passed for getting the performance of the model overtraining. This `fit_generator` enables parallel processing of data and model training. The CPU performs the rescaling while the GPU can perform the model training. This gives the high efficiency of computing resources. After 50 epochs, this model should give an accuracy of 60%. Next, we will see how to augment the dataset to get an improved performance.

Augmenting the dataset

Data augmentation gives ways to increase the size of the dataset. Data augmentation introduces noise during training, producing robustness in the model to various inputs. This technique is useful in scenarios when the dataset is small and can be combined and used with other techniques. Next, we will see the different types of augmentation.

Augmentation techniques

There are various ways to augment the images as described as follows:

- **Flipping:** The image is mirrored or flipped in a horizontal or vertical direction
- **Random Cropping:** Random portions are cropped, hence the model can deal with occlusions
- **Shearing:** The images are deformed to affect the shape of the objects
- **Zooming:** Zoomed portions of images are trained to deal with varying scales of images
- **Rotation:** The objects are rotated to deal with various degrees of change in objects
- **Whitening:** The whitening is done by a Principal Component Analysis that preserves only the important data
- **Normalization:** Normalizes the pixels by standardizing the mean and variance
- **Channel shifting:** The color channels are shifted to make the model robust to color changes caused by various artifacts

All these techniques are implemented in `ImageDataGenerator` to increase the dataset size. The following is a modified version of `generator_train` with some augmentation techniques discussed previously:

```
generator_train = tf.keras.preprocessing.image.ImageDataGenerator(  
    rescale=1. / 255,  
    horizontal_flip=True,  
    zoom_range=0.3,  
    shear_range=0.3,)
```

Replacing the `generator_train` in the preceding code will increase the accuracy to 90%. Change the parameters of augmentation and notice the changes. We will discuss a technique called transfer learning in the following section, which helps in training bigger models with fewer data.

Transfer learning or fine-tuning of a model

Transfer learning is the process of learning from a pre-trained model that was trained on a larger dataset. Training a model with random initialization often takes time and energy to get the result. Initializing the model with a pre-trained model gives faster convergence, saving time and energy. These models that are pre-trained are often trained with carefully chosen hyperparameters.

Either the several layers of the pre-trained model can be used without any modification, or can be bit trained to adapt to the changes. In this section, we will learn how to fine-tune or transfer learning for a model that was trained on the ImageNet dataset with millions of classes.

Training on bottleneck features

The models that are covered in the previous sections are simple and hence, may yield less accuracy. Complex models should be built from them. They cannot be built from scratch. Hence, bottleneck features are extracted and the classifier is trained on them. Bottleneck features are the features that are produced by complex architectures training several million images. The images are done with a forward pass and the pre-final layer features are stored. From these, a simple logistic classifier is trained for classification. Extract the bottleneck layers as follows:

```
generator = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1. /
255)

model = tf.keras.applications.VGG16(include_top=False)

train_images = generator.flow_from_directory(
    train_dir,
    batch_size=batch_size,
    target_size=(image_width, image_height),
    class_mode=None,
    shuffle=False
)

train_bottleneck_features = model.predict_generator(train_images,
epoch_steps)

test_images = generator.flow_from_directory(
    test_dir,
    batch_size=batch_size,
    target_size=(image_width, image_height),
    class_mode=None,
    shuffle=False
)

test_bottleneck_features = model.predict_generator(test_images, test_steps)
```

The VGG model is taken and used to predict the images. The labels are assigned as follows:

```
train_labels = np.array([0] * int(no_train / 2) + [1] * int(no_train / 2))
test_labels = np.array([0] * int(no_test / 2) + [1] * int(no_test / 2))
```

A sequential model with a couple of layers is built, compiled, and trained with the bottleneck features and can be implemented using the code given as follows:

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=train_bottleneck_features.shape[1:]))
model.add(tf.keras.layers.Dense(1024, activation='relu'))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(1, activation='softmax'))
model.compile(loss=tf.keras.losses.categorical_crossentropy,
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

These bottleneck features are trained with the model using the code shown as follows:

```
model.fit(
    train_bottleneck_features,
    train_labels,
    batch_size=batch_size,
    epochs=epochs,
    validation_data=(test_bottleneck_features, test_labels))
```

This gives a different approach to training the model and is useful when the training data is low. This is often a faster method to train a model. Only the final activations of the pre-trained model are used to adapt to the new task. This idea can be extended to fine-tune several layers as shown next:

Fine-tuning several layers in deep learning

A pre-trained model can be loaded and only a few layers can be trained. This approach works better when the given problem is very different from the images that the model is trained upon. **Fine-tuning** is a common practice in deep learning. This gives advantages when the dataset is smaller. The optimization also can be obtained faster.

Training a deep network on a small dataset results in overfitting. This kind of overfitting can also be avoided using the fine-tuning procedure. The model trained on a bigger dataset should be also similar, as we are hoping that the activations and features are similar to the smaller dataset. You can start with the stored weights path as show below:

```
top_model_weights_path = 'fc_model.h5'
```

Load the **Visual Geometry Group (VGG)** model and set the initial layers to be non-trainable. The VGG model will be covered in detail in the following section. For now, consider VGG as a big deep learning model that works well on image data. Replace the fully connected layers with new trainable layers using the code given as follows:

```
model = tf.keras.applications.VGG16(include_top=False)
```

A small two-layer feedforward network can be built on top of the VGG model with usually hidden units, activations, and dropout as follows:

```
model_fine_tune = tf.keras.models.Sequential()
model_fine_tune.add(tf.keras.layers.Flatten(input_shape=model.output_shape))
model_fine_tune.add(tf.keras.layers.Dense(256, activation='relu'))
model_fine_tune.add(tf.keras.layers.Dropout(0.5))
model_fine_tune.add(tf.keras.layers.Dense(no_classes,
activation='softmax'))
```

The top model has also to be loaded with weights that are already fully trained. The top model can then be added to the convolutional base:

```
model_fine_tune.load_weights(top_model_weights_path)
model.add(model_fine_tune)
```

We can set the top 25 layers to be non-trainable up to the last convolution block so that their weights will be not be updated. Only the rest of the layers will be updated:

```
for vgg_layer in model.layers[:25]:
    vgg_layer.trainable = False
```

Compile the model with the gradient descent optimizer at a slow learning rate with a magnitude of order of 4:

```
model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

We can combine the augmentation techniques that were covered earlier with shear, zoom, and flip. The generator can be added with flow from the directory with both the train and validation datasets. Now the model can be fine-tuned combined with data augmentation. This way of training gives a better accuracy than all the previous methods. The following is a guide for transfer learning:

Data Size	Similar Dataset	Different Dataset
Smaller data	Fine-tune the output layers	Fine-tune the deeper layer
Bigger data	Fine-tune the whole model	Train from scratch

Depending on the data size, the number of layers to fine-tune can be determined. The less data there is, the lesser the number of layers to fine-tune. We have seen how to improve the accuracy of the model using transfer learning techniques.

Developing real-world applications

Recognizing cats and dogs is a cool problem but less likely a problem of importance. Real-world applications of image classification used in products may be different. You may have different data, targets, and so on. In this section, you will learn the tips and tricks to tackle such different settings. The factors that should be considered when approaching a new problem are as follows:

- The number of targets. Is it a 10 class problem or 10,000 class problem?
- How vast is the intra-class variance? For example, does the different type of cats have to be identified under one class label?
- How vast is the inter-class variance? For example, do the different cats have to be identified?
- How big is the data?
- How balanced is the data?
- Is there already a model that is trained with a lot of images?
- What is the requisite for deployment inference time and model size? Is it 50 milliseconds on an iPhone or 10 milliseconds on Google Cloud Platform? How much RAM can be consumed to store the model?

Try to answer these questions when approaching an image classification problem. Based on the answers, you can design the training architecture and improve the accuracy as described in the next section.

Choosing the right model

There are a lot of options for architectures. Based on the flexibility of deployment, you can choose the model. Remember that convolution is smaller and slower, but dense layers are bigger and faster. There is a trade-off between size, runtime, and accuracy. It is advisable to test out all the architectures before the final decision. Some models may work better than others, based on the application. You can reduce the input size to make the inference faster. Architectures can be selected based on the metrics as described in the following section.

Tackling the underfitting and overfitting scenarios

The model may be sometimes too big or too small for the problem. This could be classified as underfitting or overfitting, respectively. Underfitting happens when the model is too small and can be measured when training accuracy is less. Overfitting happens when the model is too big and there is a large gap between training and testing accuracies. Underfitting can be solved by the following methods:

- Getting more data
- Trying out a bigger model
- If the data is small, try transfer learning techniques or do data augmentation

Overfitting can be solved by the following methods:

- Regularizing using techniques such as dropout and batch normalization
- Augmenting the dataset

Always watch out for loss. The loss should be decreasing over iterations. If the loss is not decreasing, it signals that training has stopped. One solution is to try out a different optimizer. Class imbalance can be dealt with by weighting the loss function. Always use **TensorBoard** to watch the summaries. It is difficult to estimate how much data is needed. This section is the best lesson on training any deep learning models. Next, we will cover some application-specific guidance.

Gender and age detection from face

Applications may require gender and age detection from a face. The face image can be obtained by face detectors. The cropped images of faces can be supplied as training data, and the similar cropped face should be given for inference. Based on the required inference time, OpenCV, or CNN face detectors can be selected. For training, Inception or ResNet can be used. If the required inference time is much less because it is a video, it's better to use three convolutions followed by two fully connected layers. Note that there is usually a huge class imbalance in age datasets, hence using a different metric like precision and recall will be helpful.

Fine-tuning apparel models

Fine-tuning of apparel models is a good choice. Having multiple softmax layers that classify attributes will be useful here. The attributes could be a pattern, color, and so on.

Brand safety

Training bottleneck layers with **Support Vector Machine (SVM)** is a good option as the images can be quite different among classes. This is typically used for content moderation to help avoid images that are explicit. You have learned how to approach new problems in image classification.

Summary

We have covered basic, yet useful models for training classification tasks. We saw a simple model for an MNIST dataset with both Keras and TensorFlow APIs. We also saw how to utilize TensorBoard for watching the training process. Then, we discussed state-of-the-art architectures with some specific applications. Several ways to increase the accuracy such as data augmentation, training on bottleneck layers, and fine-tuning a pre-trained model were also covered. Tips and tricks to train models for new models were also presented.

In the next chapter, we will see how to visualize the deep learning models. We will also deploy the trained models in this chapter for inference. We will also see how to use the trained layers for the application of an image search through an application. Then, we will understand the concept of autoencoders and use it for the dimensionality of features.

4

Object Detection

Object detection is the act of finding the location of an object in an image. In this chapter, we will learn the techniques of object detection and implement pedestrian detection by understanding the following topics:

- Basics and the difference between localization and detection
- Various datasets and their descriptions
- Algorithms used for object localization and detection
- TensorFlow API for object detection
- Training new object detection models
- Pedestrian detection on a moving car with YOLO algorithm

Detecting objects in an image

Object detection had an explosion concerning both applications and research in recent years. Object detection is a problem of importance in computer vision. Similar to image classification tasks, deeper networks have shown better performance in detection. At present, the accuracy of these techniques is excellent. Hence it is used in many applications.

Image classification labels the image as a whole. Finding the position of the object in addition to labeling the object is called **object localization**. Typically, the position of the object is defined by rectangular coordinates. Finding multiple objects in the image with rectangular coordinates is called detection. Here is an example of object detection:



The image shows four objects with bounding boxes. We will learn algorithms that can perform the task of finding the boxes. The applications are enormous in robot vision, such as self-driving cars and industrial objects. We can summarize localization and detection tasks to the following points:

- Localization detects one object in an image within a label
- Detection finds all the objects within the image along with the labels

The difference is the number of objects. In detection, there are a variable number of objects. This small difference makes a big difference when designing the architectures for the deep learning model concerning localization or detection. Next, we will see various datasets available for the tasks.

Exploring the datasets

The datasets available for object localization and detection are many. In this section, we will explore the datasets that are used by the research community to evaluate the algorithms. There are datasets with a varying number of objects, ranging from 20 to 200 annotated in these datasets, which makes object detection hard. Some datasets have too many objects in one image compared to other datasets with just one object per image. Next, we will see the datasets in detail.

The average number of objects is 7.2 per image. These are the famous datasets for the object detection challenge. Next, we will learn how to evaluate the algorithms against these datasets.

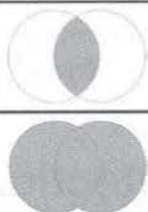
Evaluating datasets using metrics

Metrics are essential for understanding in the context of a deep learning task. The metrics of object detection and localization are peculiar because of human annotation. The human may have annotated a box that is called **ground-truth**. The ground-truth need not be the absolute truth. Moreover, the boxes can be a few pixels different from human to human. Hence it becomes harder for the algorithm to detect the exact bounding box drawn by humans.

Intersection over Union (IoU) is used to evaluate the localization task. **Mean Precision Average (mAP)** is used to evaluate the detection task. We will see the descriptions of the metrics in the next sections.

Intersection over Union

The IoU is the ratio of the overlapping area of **ground truth** and predicted area to the total area. Here is a visual explanation of the metric:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


The two squares represent the bounding boxes of ground truth and predictions. The IoU is calculated as a ratio of the area of overlap to the area of the union. Here is the script to compute the IoU is given ground truth and prediction bounding boxes:

```
def calculate_iou(gt_bb, pred_bb):
    """
    :param gt_bb: ground truth bounding box
    :param pred_bb: predicted bounding box
    """
    gt_bb = tf.stack([
        gt_bb[:, :, :, :, 0] - gt_bb[:, :, :, :, 2] / 2.0,
        gt_bb[:, :, :, :, 1] - gt_bb[:, :, :, :, 3] / 2.0,
        gt_bb[:, :, :, :, 0] + gt_bb[:, :, :, :, 2] / 2.0,
```

```

gt_bb[:, :, :, :, 1] + gt_bb[:, :, :, :, 3] / 2.0))
gt_bb = tf.transpose(gt_bb, [1, 2, 3, 4, 0])
pred_bb = tf.stack([
    pred_bb[:, :, :, :, 0] - pred_bb[:, :, :, :, 2] / 2.0,
    pred_bb[:, :, :, :, 1] - pred_bb[:, :, :, :, 3] / 2.0,
    pred_bb[:, :, :, :, 0] + pred_bb[:, :, :, :, 2] / 2.0,
    pred_bb[:, :, :, :, 1] + pred_bb[:, :, :, :, 3] / 2.0])
pred_bb = tf.transpose(pred_bb, [1, 2, 3, 4, 0])
area = tf.maximum(
    0.0,
    tf.minimum(gt_bb[:, :, :, :, 2:], pred_bb[:, :, :, :, 2:]) -
    tf.maximum(gt_bb[:, :, :, :, 2:], pred_bb[:, :, :, :, 2:]))
intersection_area = area[:, :, :, :, 0] * area[:, :, :, :, 1]
gt_bb_area = (gt_bb[:, :, :, :, 2] - gt_bb[:, :, :, :, 0]) * \
    (gt_bb[:, :, :, :, 3] - gt_bb[:, :, :, :, 1])
pred_bb_area = (pred_bb[:, :, :, :, 2] - pred_bb[:, :, :, :, 0]) * \
    (pred_bb[:, :, :, :, 3] - pred_bb[:, :, :, :, 1])
union_area = tf.maximum(gt_bb_area + pred_bb_area - intersection_area,
1e-10)
iou = tf.clip_by_value(intersection_area / union_area, 0.0, 1.0)
return iou

```

The ground truth and predicted bounding boxes are stacked together. Then the area is calculated while handling the case of negative area. The negative area could occur when bounding box coordinates are incorrect. The right side coordinates of the box may occur left to the left coordinates. Since the structure of the bounding box is not preserved, the negative area is bound to occur. The union and intersection areas are computed followed by a final IoU calculation which is the ratio of the overlapping area of **ground truth** and predicted area to the total area. The IoU calculation can be coupled with algorithms to train localization problems.

The mean average precision

The mAP is used for evaluating detection algorithms. The mAP metric is the product of precision and recall of the detected bounding boxes. The mAP value ranges from 0 to 100. The higher the number, the better it is. The mAP can be computed by calculating **average precision (AP)** separately for each class, then the average over the class. A detection is considered a true positive only if the mAP is above 0.5. All detections from the test images can be combined by drawing a draw precision/recall curve for each class. The final area under the curve can be used for the comparison of algorithms. The mAP is a good measure of the sensitivity of the network while not raising many false alarms. We have learned the evaluating algorithms for the datasets. Next, we will look at algorithms for a localization task.

Localizing algorithms

Localization algorithms are an extension of the materials learned in Chapter 2, *Image Classification* and Chapter 3, *Image Retrieval*. In image classification, an image is passed through several layers of a CNN (convolutional neural network). The final layer of CNN outputs the probabilistic value, belonging to each of the labels. This can be extended to localize the objects. We will see these ideas in the following sections.

Localizing objects using sliding windows

An intuitive way of localization is to predict several cropped portions of an image with an object. The cropping of the images can be done by moving a window across the image and predicting for every window. The method of moving a smaller window than the image and cropping the image according to window size is called a **sliding window**. A prediction can be made for every cropped window of the image which is called sliding window object detection.

The prediction can be done by the deep learning model trained for image classification problems with closely-cropped images. Close cropping means that only one object will be found in the whole image. The movement of the window has to be uniform across the image. Each portion of the image is passed through the model to find the classification. There are two problems with this approach.

- It can only find objects that are the same size as the window. The sliding window will miss an object if the object size is bigger than the window size. To overcome this, we will use the concept of **scale space**.
- Another problem is that moving the window over pixels may lead to missing a few objects. Moving the window over every pixel will result in a lot of extra computation hence it will slow down the system. To avoid this, we will incorporate a trick in the convolutional layers.

We will cover both these techniques in the next section.

The scale-space concept

The scale-space is the concept of using images that are of various sizes. An image is reduced to smaller size, hence bigger objects can be detected with the same-sized window. An image can be resized to some sizes with decreasing sizes. The resizing of images by removing alternative pixels or interpolation may leave some artefacts. Hence the image is smoothened and resized iteratively. The images that are obtained by smoothening and resizing are scale space.

The window is slide on every single scale for the localization of objects. Running multiple scales is equivalent to running the image with a bigger window. The computational complexity of running on multiple scales is high. Localization can be sped up by moving faster with a trade-off for accuracy. The complexity makes the solution not usable in production. The idea of the sliding window could be made efficient with a fully convolutional implementation of sliding windows.

Training a fully connected layer as a convolution layer

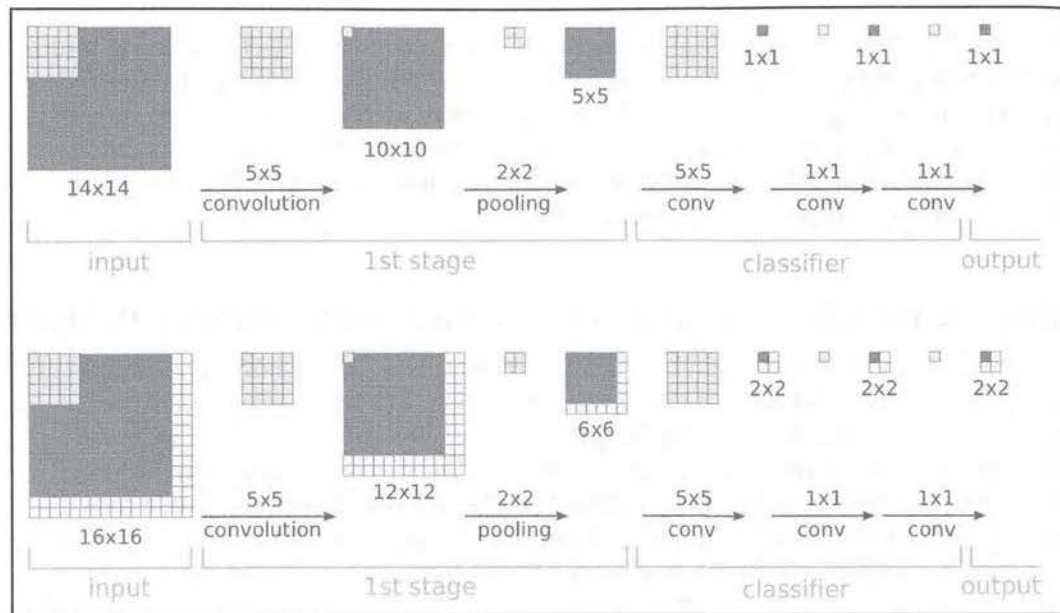
The problem with the sliding window is the computational complexity. The complexity is because predictions are made for every window. Deep learning features have been computed for every window for overlapping regions. This computation of features for overlapping regions in cropped windows can be reduced. The solution is to use a fully convolutional net which computes the feature only once. For understanding a fully convolutional net, let's first see how to convert a fully connected layer to a `convolution_layer`. The kernel is changed to the same size, with the same number of filters as the number of neurons. It can be repeated for other layers too. Changing the kernel size is an easier way to convert a fully connected layer to a `convolution_layer`:

```
convolution_layer_1 = convolution_layer(x_input_reshape, 64)
pooling_layer_1 = pooling_layer(convolution_layer_1)
convolution_layer_2 = convolution_layer(pooling_layer_1, 128)
pooling_layer_2 = pooling_layer(convolution_layer_2)
dense_layer_bottleneck = convolution_layer(pooling_layer_2, 1024, [5, 5])
logits = convolution_layer(dense_layer_bottleneck, no_classes, [1, 1])
logits = tf.reshape(logits, [-1, 10])
```

The dense layers are expressed as convolution layers. This idea is powerful and useful in various scenarios. We will extend this idea to express sliding window as a full convolution network.

Convolution implementation of sliding window

In this technique, instead of sliding, the final target is made into some targets required as depth and a number of boxes as the window. Sermanet et al. (<https://arxiv.org/pdf/1312.6229.pdf>) used fully convolution implementation to overcome this problem of the sliding window. Here is an illustration of such convolution implementation, of the sliding window:



Reproduced with permission from Sermanet et al.

In the upper part of the example, normal classification is represented as a fully convolutional layer. In the lower part of the illustration, the same kernel is applied to a bigger image producing 2x2 at the end instead of 1. The final layer denotes four of the output of those bounding boxes. Having a volume for prediction improves efficiency, but the boxes still have a problem with accurate positioning. So the sliding window is not necessary, hence it solves the complexity. The aspect ratio is always changing and has to be seen at multiple scales. The bounding boxes produced by the fully convolutional method are not very accurate. The extra computations are done only for the extra region. As you can imagine, the boxes are rather restricted to the number of boxes that are trained with. Next, we will see a method to detect the bounding box positions more accurately.

Thinking about localization as a regression problem

One fundamental way to think about localization is modeling the problem as a regression problem. The bounding box is four numbers and hence can be predicted in a direct manner with a setting for regression. We will also need to predict the label, which is a classification problem.

There are different parameterizations available to define the bounding boxes. There are four numbers usually for the bounding box. One of the representations is the center of the coordinates with the height and width of the bounding box. A pre-trained model can be used by removing the fully connected layer and replacing it with a regression encoder. The regression has to be regularized with the L2 loss which performs poorly with an outlier. The L1 loss is better than L2. Swapping regression with a smoothed version of regularization is better. Fine-tuning the model gives a good accuracy, whereas training the whole network gives only a marginal performance improvement. It's a trade-off between training time and accuracy. Next, we will see different applications of regression using convolutional networks.

Applying regression to other problems

Regressing image coordinates is applicable to several other applications, such as **pose detection** and **fiducial point detection**. Pose detection is the act of finding joint locations in a human, as shown here:



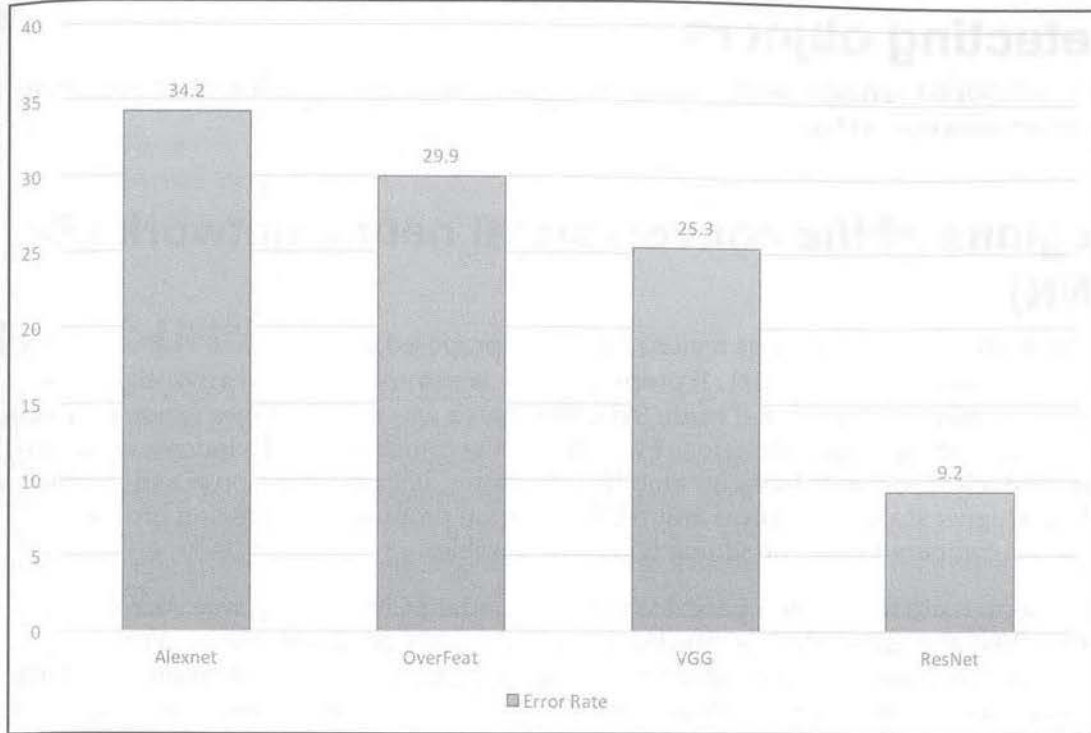
In the preceding image, multiple locations such as head, neck, shoulders, ankles, and hands were detected. This can be extended to all human parts. The regression we learned could be used for this application. Here is an example of fiducial point detection:



Fiducial points are landmarks on the face with respect to the location of the eyes, nose, and lips. Finding these landmarks are vital for face-based augmented reality applications. There are some more landmarks available in the face and will be covered in detail in Chapter 6, *Similarity Learning*, in the context of face recognition.

Combining regression with the sliding window

The classification score is computed for every window in the sliding window approach or the fully convolutional approach to know what object is present in that window. Instead of predicting the classification score for every window to detect an object, each window itself can be predicted with a classification score. Combining all the ideas such as sliding window, scale-space, full convolution, and regression give superior results than any individual approach. The following are the top five localization error rates on the ImageNet dataset achieved by various networks using the regression approach:



The preceding graph shows that the deeper the network, the better the results. For AlexNet, localization methods were not described in the paper. The OverFeat used multi-scale convolutional regression with box merging. VGG used localization but with fewer scales and location. These gains are attributed to deep features. The ResNet uses a different localization method and much deeper features.

The regression encoder and classification encoder function independently. Hence there is a possibility of predicting an incorrect label for a bounding box. This problem can be overcome by attaching the regression encoder at different layers. This method could also be used for multiple objects hence solving the object detection problem. Given an image, find all instances in that. It's hard to treat detection as regression because the number of outputs are variable. One image may have two objects and another may have three or more. In the next section, we will see the algorithms dealing with detection problems more effectively.

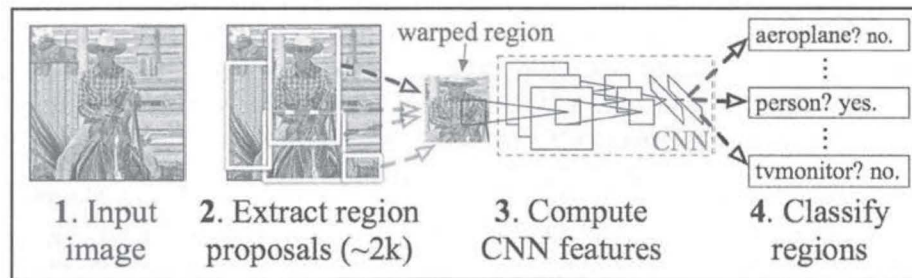
Detecting objects

There are several variants of object detection algorithms. A few algorithms that come with the object detection API are discussed here.

Regions of the convolutional neural network (R-CNN)

The first work in this series was regions for CNNs proposed by Girshick et al. (<https://arxiv.org/pdf/1311.2524.pdf>). It proposes a few boxes and checks whether any of the boxes correspond to the ground truth. **Selective search** was used for these region proposals. Selective search proposes the regions by grouping the color/texture of windows of various sizes. The selective search looks for blob-like structures. It starts with a pixel and produces a blob at a higher scale. It produces around 2,000 region proposals. This region proposal is less when compared to all the sliding windows possible.

The proposals are resized and passed through a standard CNN architecture such as Alexnet/VGG/Inception/ResNet. The last layer of the CNN is trained with an SVM identifying the object with a no-object class. The boxes are further improved by tightening the boxes around the images. A linear regression model to predict a closer bounding box is trained with object region proposals. The architecture of R-CNN is shown here:



Reproduced with permission from Girshick et al.

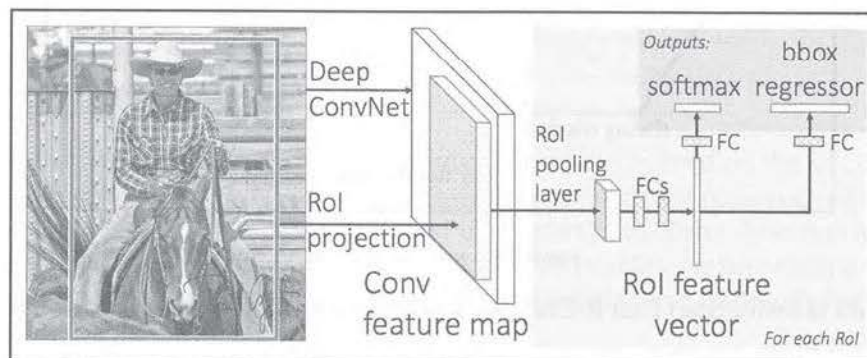
The encoder can be a pre-trained model of a standard deep learning model. The features are computed for all the regions from the training data. The features are stored and then the SVM is trained. Next, the bounding boxes are trained with the normalized coordinates. There may be some proposals outside the image coordinates and hence it is normalized for training and inference.

The disadvantages of this method are:

- Several proposals are formed by selective search and hence many inferences have to be computed, usually around 2,000
- There are three classifiers that have to be trained, which increases the number of parameters
- There is no end-to-end training

Fast R-CNN

The Fast R-CNN proposed by Girshick et al. (<https://arxiv.org/pdf/1504.08083.pdf>) method runs CNN inference only once and hence reduces computations. The output of the CNN is used to propose the networks and select the bounding box. It introduced a technique called **Region of Interest pooling**. The Region of Interest pooling takes the CNN features and pools them together according to the regions. The features obtained after the inference using CNN is pooled and regions are selected, as shown in the following image:

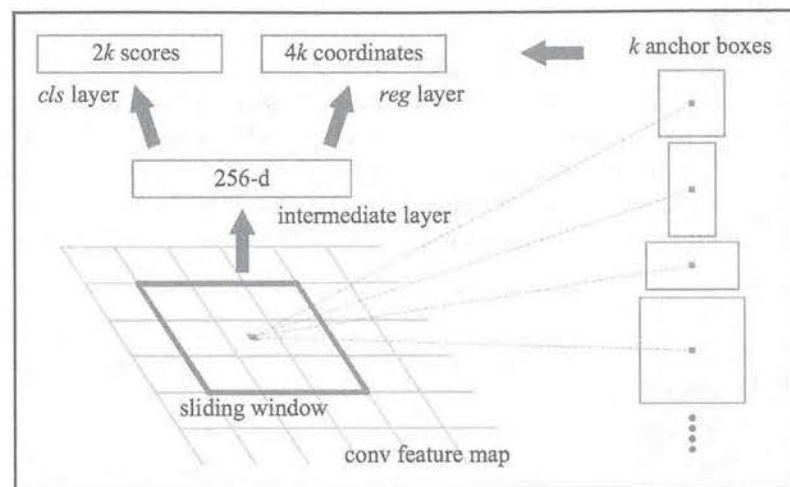


Reproduced with permission from Girshick et al.

This way, an end-to-end training is performed, avoiding multiple classifiers. Note that the SVM is replaced by the softmax layer and the box regressor is replaced by bounding box regressors. The disadvantage that still remains is the selective search, which takes some time.

Faster R-CNN

Faster R-CNN is proposed by Ren et al. (<https://arxiv.org/pdf/1506.01497.pdf>). The difference between Faster R-CNN and the Fast R-CNN method is that the Faster R-CNN uses CNN features of architecture such as VGG and Inception for proposals instead of selective search. The CNN features are further passed through the region proposal network. A sliding window is passed through features with potential bounding boxes and scores as the output, as well as a few aspect ratios that are intuitive, the model outputs bounding box and score:



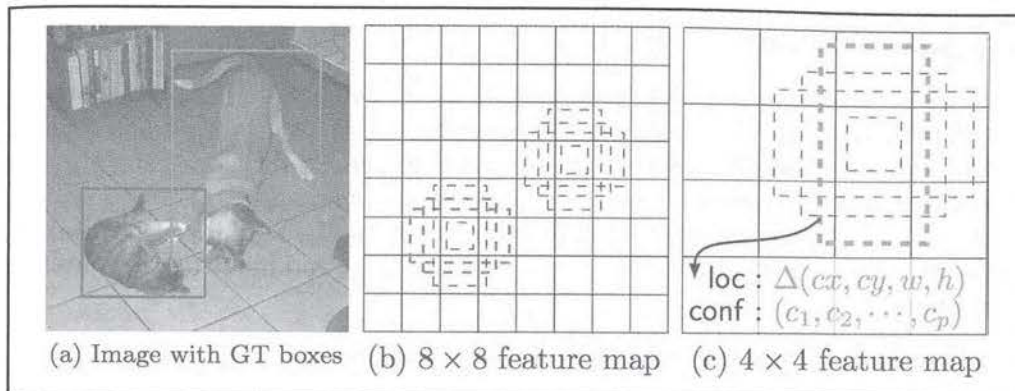
Reproduced with permission from Ren et al.

Faster R-CNN is faster than Fast R-CNN as it saves computation by computing the feature only once.

Single shot multi-box detector

SSD (Single shot multi-box) is proposed by is the fastest of all the methods. This method simultaneously predicts the object and finds the bounding box. During training, there might be a lot of negatives and hence hard-negative mining the class imbalance. The output from CNN has various sizes of features. These are passed to a 3x3 convolutional filter to predict bounding box.

This step predicts the object and bounding box:



Reproduced with permission from Liu et al.

These are the algorithms available for object detection and we will learn how to implement them in the following section.

Object detection API

Google released pre-trained models with various algorithms trained on the COCO dataset for public use. The API is built on top of TensorFlow and intended for constructing, training, and deploying object detection models. The APIs support both object detection and localization tasks. The availability of pre-trained models enables the fine-tuning of new data and hence making the training faster. These different models have trade-offs between speed and accuracy.

Installation and setup

Install the Protocol Buffers (**protobuf**) compiler with the following commands. Create a directory for protobuf and download the library directly:

```
mkdir protoc_3.3
cd protoc_3.3
wget https://github.com/google/protobuf/releases/download/v3.3.0/protoc-3.3.0-linux-x86_64.zip
```

Change the permission of the folder and extract the contents, as shown here:

```
chmod 775 protoc-3.3.0-linux-x86_64.zip
unzip protoc-3.3.0-linux-x86_64.zip
```

Protocol Buffers (protobuf) is Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. It serves the use of XML but is much simpler and faster. The models are usually exported to this format in TensorFlow. One can define the data structure once but can be read or written in a variety of languages. Then run the following command to compile the protobufs. Move back to the working folder and clone the repo from <https://github.com/tensorflow/models.git> and move them to the following folder:

```
git clone https://github.com/tensorflow/models.git
```

Now, move the model to the research folder, using the following code:

```
cd models/research/
~/protoc_3.3/bin/protoc object_detection/protos/*.proto --python_out=.
```

The TensorFlow object detection API uses protobufs for exporting model weights and the training parameters. The TensorFlow, models, research, and slim directories should be appended to PYTHONPATH by the following command:

```
export PYTHONPATH=../slim/
```

Adding to the python path with the preceding command works only one time. For the next, this command has to be run again. The installation can be tested by running the following code:

```
python object_detection/builders/model_builder_test.py
```

The output of this code is given here:

```
Ran 7 tests in 0.022s
```

```
OK
```



More information about the installation can be obtained from https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/installation.md. Now the installation is complete and tested.

Pre-trained models

There are several models that are pre-trained and made available. All these models are trained on the COCO dataset and can be used for detecting the objects that are available in the COCO dataset such as humans and cars. These models are also useful for transfer learning for a new task such as traffic sign detection. A table of pre-trained models is shown here with relative speed and mAP on the COCO dataset. Various algorithms are trained with different CNN and are depicted in the names:

Model name	Speed	COCO mAP
ssd_mobilenet_v1_coco	fast	21
ssd_inception_v2_coco	fast	24
rfcn_resnet101_coco	medium	30
faster_rcnn_resnet101_coco	medium	32
faster_rcnn_inception_resnet_v2_atrous_coco	slow	37

Based on the requirement, you can choose from the model. Download the SSD model trained on Mobilenet and extract it as shown here by going to the working directory:

```
mkdir Chapter04 && cd Chapter04
wget
http://download.tensorflow.org/models/object_detection/ssd_mobilenet_v1_coco_11_06_2017.tar.gz
tar -xvzf ssd_mobilenet_v1_coco_11_06_2017.tar.gz
```

There will be various files in the Chapter04 folder, which are listed here:

- The is the proto-definition of the graph—graph.pbtxt
- The weights of the graph frozen and can be used for inference—frozen_inference_graph.pb
- Checkpoint files
 - model.ckpt.data-00000-of-00001
 - model.ckpt.meta
 - model.ckpt.index

This model will be used in the next section for detection tasks.

Re-training object detection models

The same API lets us retrain a model for our custom dataset. Training of custom data involves the preparation of a dataset, selecting the algorithm, and performing fine-tuning. The whole pipeline can be passed as a parameter to the training script. The training data has to be converted to TensorFlow records. TensorFlow records is a file format provided by Google to make the reading of data faster than regular files. Now, we will go through the steps of training.

Data preparation for the Pet dataset

The Oxford-IIIT Pet dataset is used for this example. Download the image and annotations with these commands from the Chapter04 directory.

```
wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
```

Extract the image and annotations as shown here:

```
tar -xvf images.tar.gz
tar -xvf annotations.tar.gz
```

Create the `pet_tf` record file to create the dataset in the `tf` records, as they are the required input for the object detection trainer. The `label_map` for the Pet dataset can be found at `object_detection/data/pet_label_map.pbtxt`. Move to the research folder and run the following command:

```
python object_detection/create_pet_tf_record.py \
  --label_map_path=object_detection/data/pet_label_map.pbtxt \
  --data_dir=~/.chapter4/. \
  --output_dir=~/.chapter4/.
```

You can see two `.record` files in the research directory named `pet_train.record` and `pet_val.record`.

Object detection training pipeline

The training protobuf has to be configured for training. The following five things are important in this process:

- The model configuration with the type of model
- The `train_config` for standard training parameters

- The `eval_config` for the metrics that have to be reported
- The `train_input_config` for the dataset
- The `eval_input_config` for the evaluation dataset

We will use the config file from https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssd_mobilenet_v1_pets.config. Download it to the `Chapter04` folder by running the following command. Open the config file and edit the following lines:

```
fine_tune_checkpoint:
  "~/Chapter04/ssd_mobilenet_v1_coco_11_06_2017/model.ckpt"

train_input_reader: {
  tf_record_input_reader {
    input_path: "~/Chapter04/pet_train.record"
  }
  label_map_path:
  "~/model/research/object_detection/data/pet_label_map.pbtxt"
}

eval_input_reader: {
  tf_record_input_reader {
    input_path: "~/Chapter04/pet_val.record"
  }
  label_map_path:
  "~/model/research/object_detection/data/pet_label_map.pbtxt"
}
```

Save the config file. There are various parameters in the file that affect the accuracy of the model.

Training the model

Now the API, data and config files are ready for re-training. The training can be triggered by the following command:

```
PYTHONPATH=../../slim/. python object_detection/train.py \
  --logtostderr \
  --pipeline_config_path=~/.chapter4/ssd_mobilenet_v1_pets.config \
  --train_dir=~/.Chapter04
```

The training will start with a loss of around 140 and will keep decreasing. The training will run forever and has to be killed manually by using the `Ctrl + C` command. The checkpoints created during the training can be used for inference later.

Monitoring loss and accuracy using TensorBoard

The training loss and accuracy can be monitored using TensorBoard. Run the TensorBoard using the following command:

```
tensorboard --logdir=/home/ubuntu/Chapter04
```

Both training and evaluation can be visualized in the TensorBoard.

Training a pedestrian detection for a self-driving car

The dataset for training a pedestrian object detection can be found at <http://pascal.inrialpes.fr/data/human/>. The steps to detecting pedestrians can be found at https://github.com/diegocavalca/machine-learning/blob/master/supervised/object_detection_tensorflow/simple.detection.ipynb. The dataset for training a Sign Detector can be downloaded from http://www.vision.ee.ethz.ch/~timofte/traffic_signs/ and <http://btsd.ethz.ch/shareddata/>. In the case of a self-driving car, there would be four classes in an image for labeling: pedestrian, car, motorcycle, and background. The background class has to be detected when none of the classes is present. An assumption in training a deep learning classification model is that at least one of the objects will be present in the image. By adding the background class, we are overcoming the problem. The neural network can also produce a bounding box of the object from the label.

The YOLO object detection algorithm

A recent algorithm for object detection is **You look only once (YOLO)**. The image is divided into multiple grids. Each grid cell of the image runs the same algorithm. Let's start the implementation by defining layers with initializers:

```
def pooling_layer(input_layer, pool_size=[2, 2], strides=2,
padding='valid'):
    layer = tf.layers.max_pooling2d(
        inputs=input_layer,
        pool_size=pool_size,
        strides=strides,
        padding=padding
    )
    add_variable_summary(layer, 'pooling')
    return layer
```

```

def convolution_layer(input_layer, filters, kernel_size=[3, 3],
padding='valid',
                    activation=tf.nn.leaky_relu):
    layer = tf.layers.conv2d(
        inputs=input_layer,
        filters=filters,
        kernel_size=kernel_size,
        activation=activation,
        padding=padding,
        weights_initializer=tf.truncated_normal_initializer(0.0, 0.01),
        weights_regularizer=tf.l2_regularizer(0.0005)
    )
    add_variable_summary(layer, 'convolution')
    return layer

def dense_layer(input_layer, units, activation=tf.nn.leaky_relu):
    layer = tf.layers.dense(
        inputs=input_layer,
        units=units,
        activation=activation,
        weights_initializer=tf.truncated_normal_initializer(0.0, 0.01),
        weights_regularizer=tf.l2_regularizer(0.0005)
    )
    add_variable_summary(layer, 'dense')
    return layer

```

It can be noticed that the activation layer is `leaky_relu` and the weights are initialized with truncated normal distribution. These modified layers can be used for building the model. The model is created as follows:

```

yolo = tf.pad(images, np.array([[0, 0], [3, 3], [3, 3], [0, 0]]),
name='pad_1')
yolo = convolution_layer(yolo, 64, 7, 2)
yolo = pooling_layer(yolo, [2, 2], 2, 'same')
yolo = convolution_layer(yolo, 192, 3)
yolo = pooling_layer(yolo, 2, 'same')
yolo = convolution_layer(yolo, 128, 1)
yolo = convolution_layer(yolo, 256, 3)
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = pooling_layer(yolo, 2, 'same')
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)

```

```
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = convolution_layer(yolo, 512, 1)
yolo = convolution_layer(yolo, 1024, 3)
yolo = pooling_layer(yolo, 2)
yolo = convolution_layer(yolo, 512, 1)
yolo = convolution_layer(yolo, 1024, 3)
yolo = convolution_layer(yolo, 512, 1)
yolo = convolution_layer(yolo, 1024, 3)
yolo = convolution_layer(yolo, 1024, 3)
yolo = tf.pad(yolo, np.array([[0, 0], [1, 1], [1, 1], [0, 0]]))
yolo = convolution_layer(yolo, 1024, 3, 2)
yolo = convolution_layer(yolo, 1024, 3)
yolo = convolution_layer(yolo, 1024, 3)
yolo = tf.transpose(yolo, [0, 3, 1, 2])
yolo = tf.layers.flatten(yolo)
yolo = dense_layer(yolo, 512)
yolo = dense_layer(yolo, 4096)

dropout_bool = tf.placeholder(tf.bool)
yolo = tf.layers.dropout(
    inputs=yolo,
    rate=0.4,
    training=dropout_bool
)
yolo = dense_layer(yolo, output_size, None)
```

Several convolution layers are stacked, producing the YOLO network. This network is utilized for creating the object detection algorithm for real-time detection.

Summary

In this chapter, we have learned the difference between object localization and detection tasks. Several datasets and evaluation criteria were discussed. Various approaches to localization problems and algorithms, such as variants of R-CNN and SSD models for detection, were discussed. The implementation of detection in open-source repositories was covered. We trained a model for pedestrian detection using the techniques. We also learned about various trade-offs in training such models.

In the next chapter, we will learn about semantic segmentation algorithms. We will use the knowledge to implement the segmentation algorithms for medical imaging and satellite imagery problems.

5

Semantic Segmentation

In this chapter, we will learn about various semantic segmentation techniques and train models for the same. Segmentation is a pixel-wise classification task. The ideas to solve segmentation problem is an extension to object detection problems. Segmentation is highly useful in applications such medical and satellite image understanding.

The following topics will be covered in the chapter:

- Learning the difference between semantic segmentation and instance segmentation
- Segmentation datasets and metrics
- Algorithms for semantic segmentation
- Application of segmentation to medical and satellite images
- Algorithms for instance segmentation

Predicting pixels

Image classification is the task of predicting labels or categories. Object detection is the task of predicting a list of several deep learning-based algorithms with its corresponding bounding box. The bounding box may have objects other than the detected object inside it. In some applications, labeling every pixel to a label is important rather than bounding box which may have multiple objects. **Semantic segmentation** is the task of predicting pixel-wise labels.

9

Video Classification

In this chapter, we will see how to train deep learning models for video data. We will start classifying videos on a frame basis. Then, we will use the temporal information for better accuracy. Later, we will extend the applications of images to videos, including pose estimation, captioning, and generating videos.

We will cover the following topics in this chapter:

- The datasets and the algorithms of video classification
- Splitting a video into frames and classifying videos
- Training a model for visual features on an individual frame level 0
- Understanding 3D convolution and its use in videos
- Incorporating motion vectors on video
- Object tracking utilizing the temporal information
- Applications such as human pose estimation and video captioning

Understanding and classifying videos

A video is nothing but a series of images. Video brings a new dimension to the image along the temporal direction. The spatial features of the images and temporal features of the video can be put together, providing a better outcome than just the image. The extra dimension also results in a lot of space and hence increases the complexity of training and inference. The computational demands are extremely high for processing a video. Video also changes the architecture of deep learning models as we have to consider the temporal features.

Video classification is the task of labeling a video with a category. A category can be on the frame level or for the whole video. There could be actions or tasks performed in the video. Hence, a video classification may label the objects present in the video or label the actions happening in the video. In the next section, we will see the available datasets for video classification tasks.

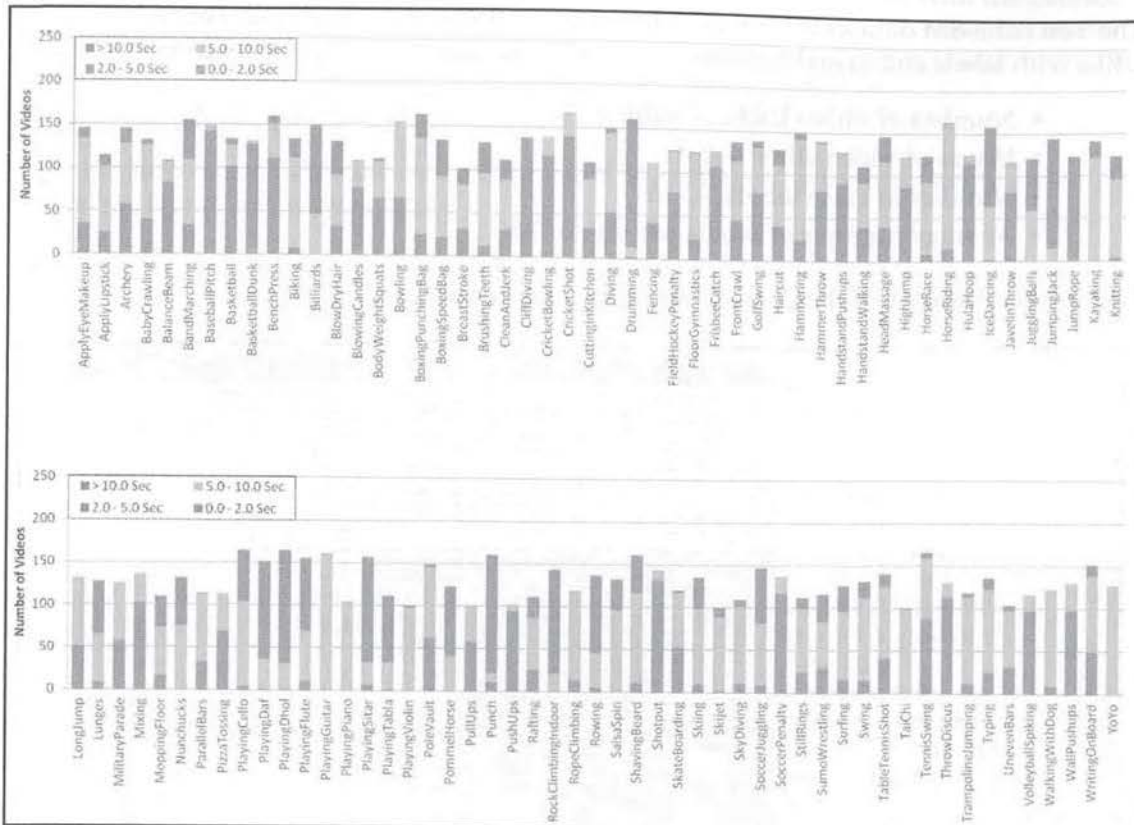
Exploring video classification datasets

Video classification is the major problem that is studied with video data. Several videos are taken and labelled with various objects or actions that are associated with the data. The datasets vary according to size, quality, and the type of labels. Some even include multiple labels for video. The videos are rather usually short in length. A long video may have various actions performed and hence can be segmented temporally, before classifying the cut video segments or snippets individually. Next, we will consider the details of some specific datasets.

UCF101

The **University of Central Florida (UCF101)** is a dataset for action recognition. The videos are collected on YouTube and consist of realistic actions. There are 101 action categories available in this dataset. There is another dataset called **UCF50** which has 50 categories. There are 13,320 videos in this dataset across the actions. The videos have a diversified variation of background, scale, pose, occlusion, and illumination conditions. The action categories are grouped into 25, which share similar variations such as the background, pose, scale, viewpoint, illumination and so on.

The actions and number of videos per action are shown here:



Source: <http://crev.ucf.edu/data/UCF101/Number%20of%20Videos%20.jpg>

All 101 actions are grouped into five types of actions as follows: Human-object interaction, Body motion, Human-human interaction, Playing musical instruments, and Sports. The dataset and annotation can be downloaded from <http://crev.ucf.edu/data/UCF101.php>.

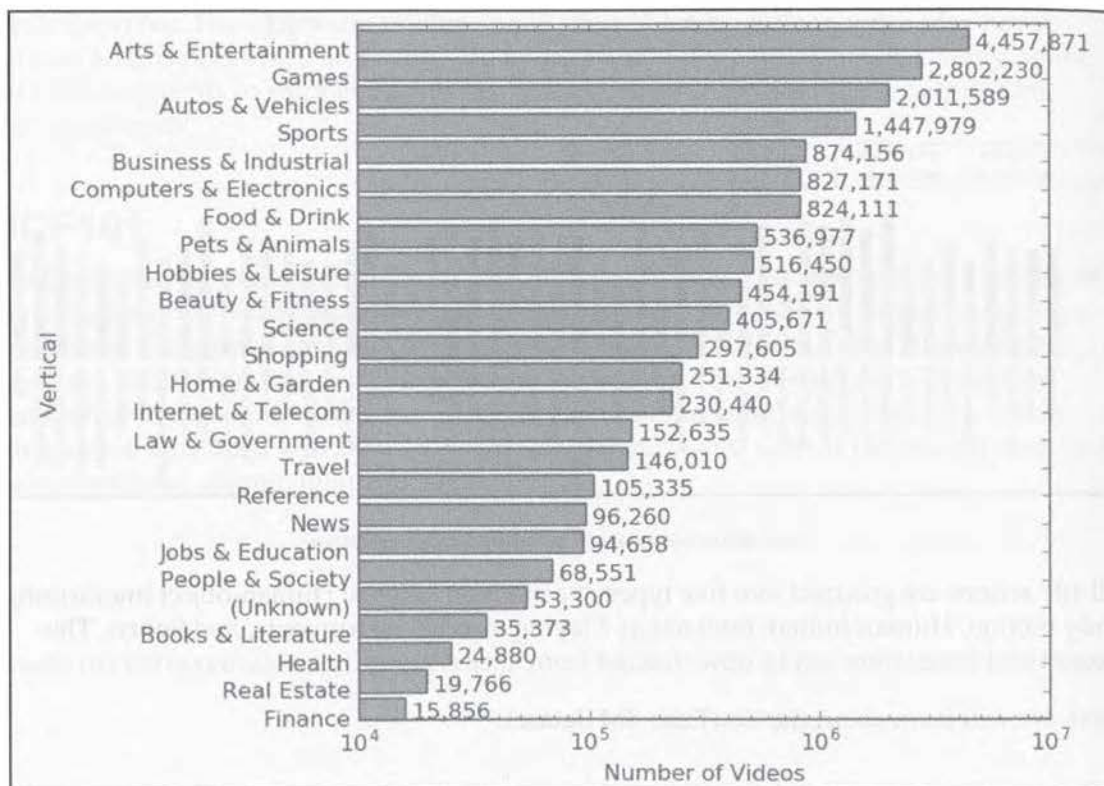
Next, we will learn about the YouTube-8M dataset.

YouTube-8M

The **YouTube-8M** dataset is for video classification problems. The dataset contains video URLs with labels and visual features. Here are a few statistics about the dataset:

- **Number of video URLs:** 7 million
- **Hours of video clips:** 450,000
- **Number of class labels:** 4,716
- **Average number of labels per video:** 3.4

Here is the summary of the dataset, across various genres:



Source: <https://research.google.com/youtube8m/vertical-videos.png>

The preceding image can give a glimpse of the type of labels available in the dataset. The video data is large and hence visual features are computed and provided with the dataset. The dataset can be accessed through this link: <https://research.google.com/youtube8m/>.

Other datasets

There are some more datasets available for video classification problems. Here are the details of a few more datasets:

- **Sports-1M (Sports - 1 Million):** Has 1,133,158 videos with 487 classes. The annotations are done automatically. The dataset can be downloaded from: <http://cs.stanford.edu/people/karpathy/deepvideo/>.
- **UCF-11 (University of Central Florida - 11 actions):** Has 1,600 videos with 11 actions. The videos have 29.97 fps (frames per second). The dataset can be downloaded along with UCF101.
- **HMDB-51 (Human Motion DataBase - 51 actions):** Has 5,100 videos with 51 actions. The dataset link is: <http://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database>.
- **Hollywood2:** Has 1,707 videos with 12 actions. The dataset link is: <http://www.di.ens.fr/~laptev/actions/hollywood2>.

We have seen the datasets available for video classification tasks, along with the description and access links. Next, we will see how to load a video and split it into frames for further processing.

Splitting videos into frames

A video can be converted to frames and saved in a directory for further usage. Splitting into frames helps us save time by decompressing the video before the training process. First, let's see a code snippet for converting video to frames:

```
import cv2
video_handle = cv2.VideoCapture(video_path)
frame_no = 0
while True:
    eof, frame = video_handle.read()
    if not eof:
        break
    cv2.imwrite("frame%d.jpg" % frame_no, frame)
    frame_no += 1
```



Using this snippet, all of the preceding datasets can be converted to frames. Note that this will require a lot of hard disk space.

Approaches for classifying videos

Videos have to be classified for several applications. Since the video is a lot of data, training and inference computations must also be accounted for. All video classification approaches are inspired by image classification algorithms. The standard architectures such as VGG, Inception, and so on are used for feature computation at a frame level and then processed further. Concepts such as **CNN**, **attention**, and **LSTM** learned in previous chapters will be useful here. Intuitively, the following approaches can be used for video classification:

- Extract the frames and use the models learned in Chapter 2, *Image Classification*, for classification on a frame basis.
- Extract the image features learned in Chapter 3, *Image Retrieval*, and the features can be used to train an RNN as described in Chapter 7, *Image Captioning*.
- Train a **3D convolution** network on the whole video. 3D convolution is an extension of 2D convolution; we will see the workings of 3D convolution in detail in the following sections.
- Use the **optical flow** of the video to further improve the accuracy. Optical flow is the pattern of movement of objects, which we will see in detail in coming sections.

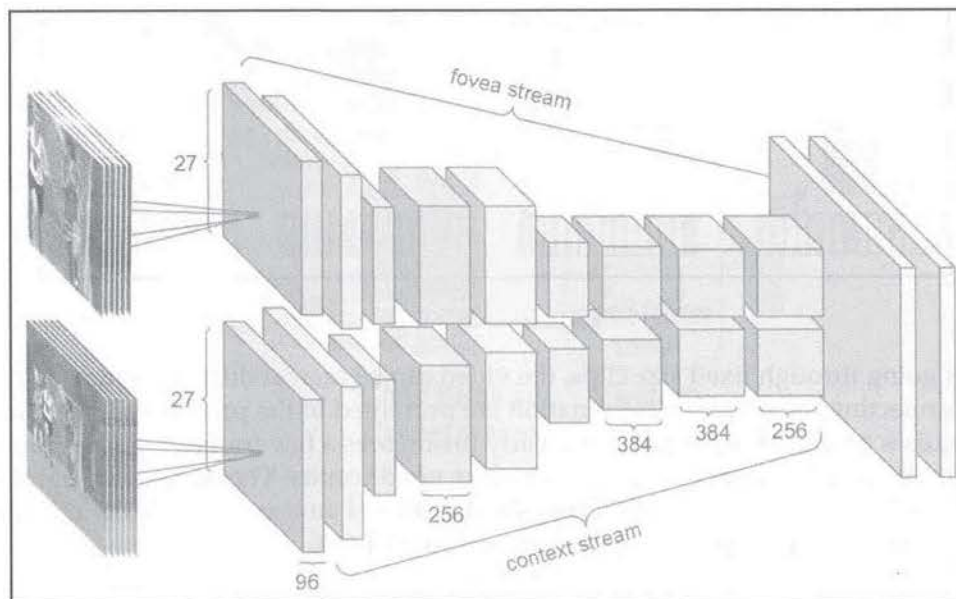
We will see several algorithms that give good accuracy with various computational complexities. The dataset can be prepared by converting it into frames and subsampling it to the same length. Some preprocessing can help, such as subtracting the mean of the Imagenet.

Fusing parallel CNN for video classification

Frame-wise, the prediction of a video may not yield good results due to the downsampling of images, which loses fine details. Using a high-resolution CNN will increase the inference time. Hence, Karpathy et al. (<https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42455.pdf>) propose fusing two streams that are run in parallel for video classification. There are two problems with doing frame-wise predictions, namely:

- Predictions may take a long time because of the larger CNN architecture
- Independent predictions lose the information along the temporal dimension

The architecture can be simplified with fewer parameters with two smaller encoders running in parallel. The video is passed simultaneously through two CNN encoders. One encoder takes a low resolution and processes high resolution. The encoder has alternating convolution, normalization, and pooling layers. The final layer of the two encoders is connected through the fully connected layer. The other encoder is of the same size, but takes only the central crop, as shown here:

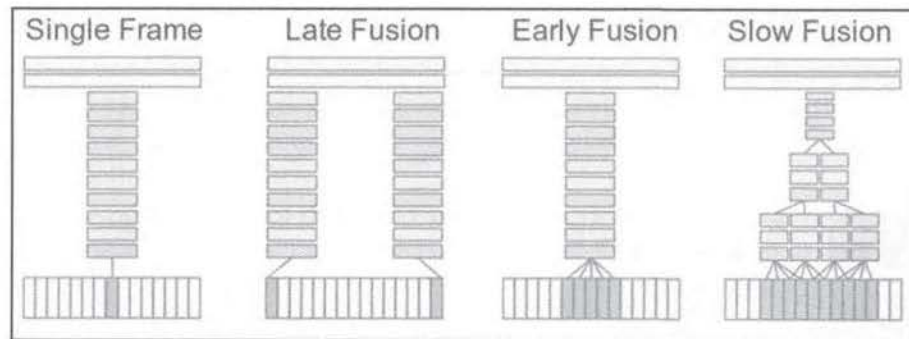


Reproduced from Karpathy et al.

Parallel processing of frames makes the runtime faster by downsampling the video. The CNN architecture is halved regarding the parameter while maintaining the same accuracy. The two streams are called the **fovea** and **context** streams. The streams are shown in the following code snippet:

```
high_resolution_input = tf.placeholder(tf.float32, shape=[None,
input_size])
low_resolution_input = tf.placeholder(tf.float32, shape=[None, input_size])
y_input = tf.placeholder(tf.float32, shape=[None, no_classes])
high_resolution_cnn = get_model(high_resolution_input)
low_resolution_cnn = get_model(low_resolution_input)
dense_layer_1 = tf.concat([high_resolution_cnn, low_resolution_cnn], 1)
dense_layer_bottleneck = dense_layer(dense_layer_1, 1024)
logits = dense_layer(dense_layer_bottleneck, no_classes)
```

The frames for processing across temporal dimensions are as shown in the following diagram:



Reproduced from Karpathy et al.

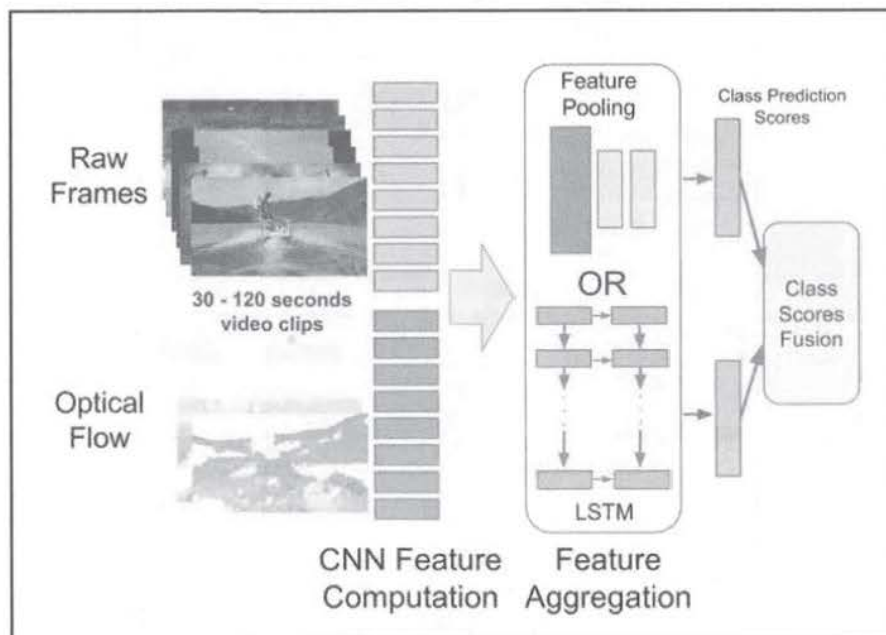
Instead of going through fixed size clips, the video can be seen at different times. Three ways of connecting the temporal information are presented in the preceding image. Late fusion requires a longer time frame while early fusion sees a few frames together. Slow fusion combines both late and early fusion to give good results. The model was trained on the Sports1M dataset, which has 487 classes and achieved an accuracy of 50%. The same model, when applied to UCF101, achieves an accuracy of 60%.

Classifying videos over long periods

The fusion method works well for short video snippets. Classifying longer videos is difficult as a lot of frames have to be computed and remembered. Ng et al. (https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Ng_Beyond_Short_Snippets_2015_CVPR_paper.pdf) proposed two methods for classifying longer videos:

- The first approach is to pool the convolutional features temporally. Max-pooling is used as a feature aggregation method.
- The second approach has an LSTM connecting the convolutional features that handle the variable length of the video.

Both approaches are shown in the following image:

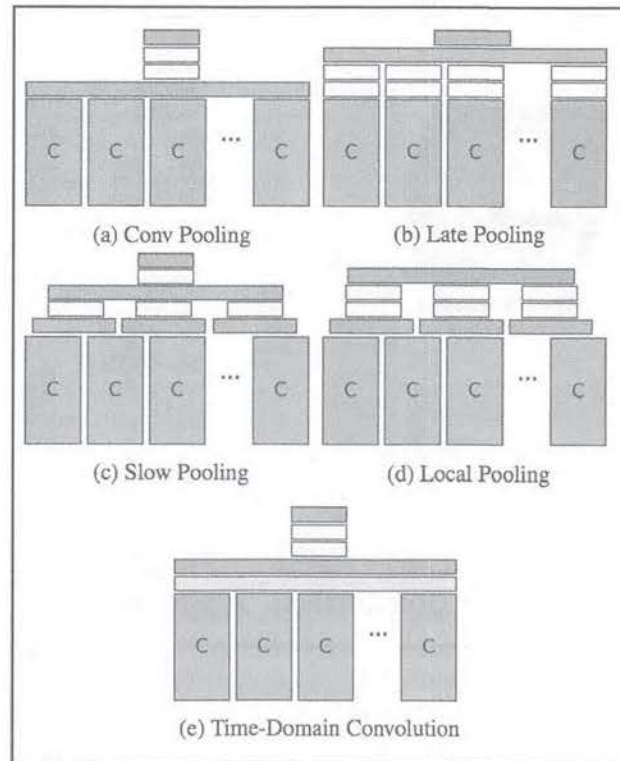


Reproduced from Ng et al.

The CNN features can be extracted and fed to a small LSTM network, as demonstrated in the following code:

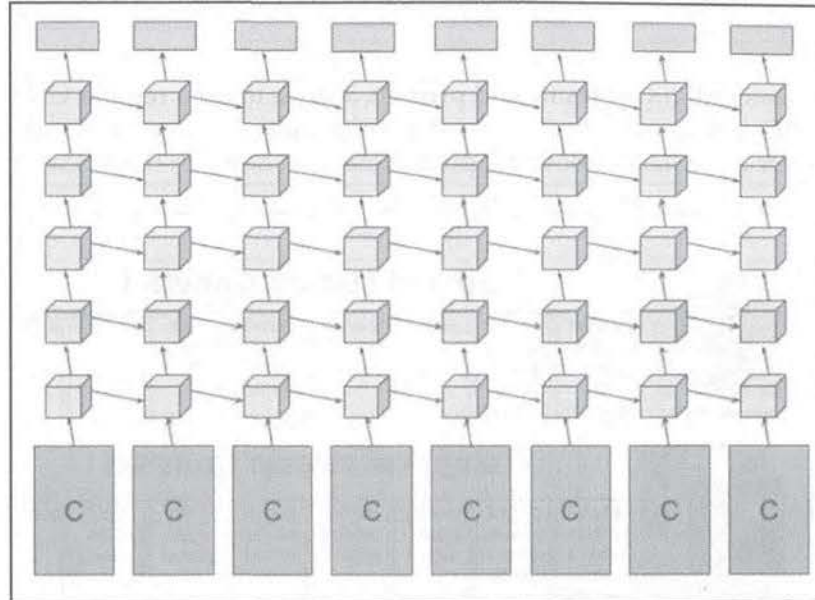
```
net = tf.keras.models.Sequential()
net.add(tf.keras.layers.LSTM(2048,
                             return_sequences=False,
                             input_shape=input_shape,
                             dropout=0.5))
net.add(tf.keras.layers.Dense(512, activation='relu'))
net.add(tf.keras.layers.Dropout(0.5))
net.add(tf.keras.layers.Dense(no_classes, activation='softmax'))
```

Adding LSTM for feature pooling instead provides better performance. The features are pooled in various ways, as shown in the following image:



Reproduced from Ng et al.

As shown in the diagram, the convolutional features can be aggregated in several different ways. The pooling is done after the fully connected layer before it. This method achieved an accuracy of 73.1% and 88.6% in the *Sports1M* dataset and *UCF101* datasets respectively. The LSTM approach is shown in the following image:



Reproduced from Ng et al.

The computations are high for this model because several LSTM's are used.

Streaming two CNN's for action recognition

The motion of objects in videos has very good information about the actions performed in the video. The motion of objects can be quantified by optical flow. Simonyan and Zisserman (<http://papers.nips.cc/paper/5353-two-stream-convolutional-networks-for-action-recognition-in-videos.pdf>) proposed a method for action recognition that uses two streams from images and optical flow.

Optical flow measures the motion by quantifying the relative movement between the observer and scene. A detailed lecture on optical flow can be found at <https://www.youtube.com/watch?v=5VyLAH8BhF8>. The optical flow can be obtained by running the following command:

```
p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None,
**lk_params)
```

One stream takes an individual frame and predicts actions using a regular CNN. The other stream takes multiple frames and computes the optical flow. The optical flow is passed through a CNN for a prediction. Both the predictions are shown in the following image:

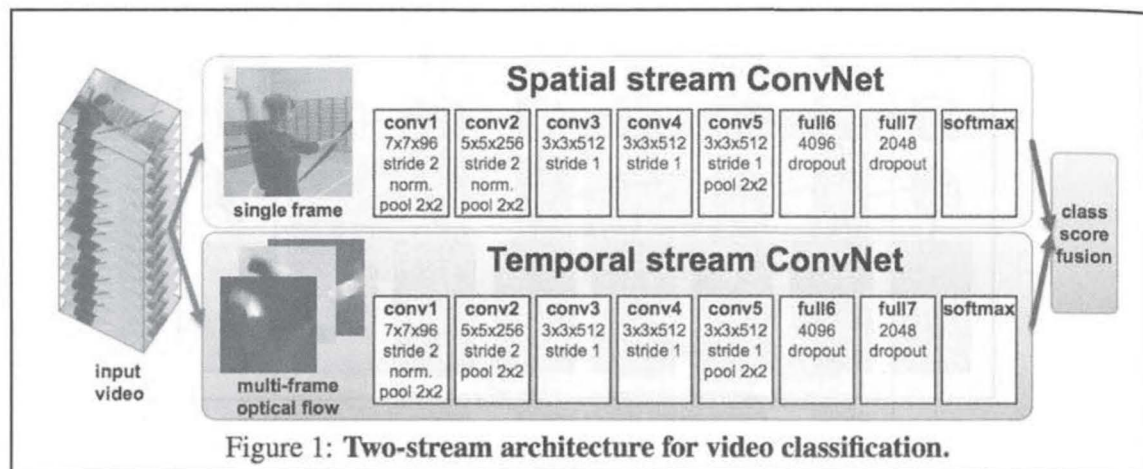


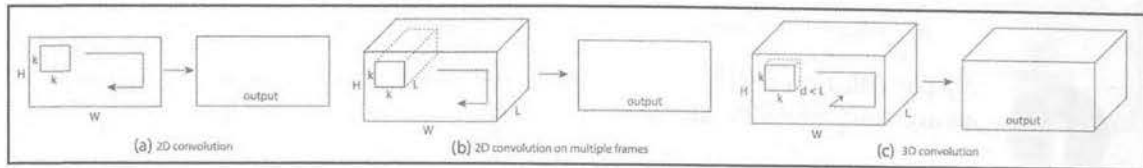
Figure 1: Two-stream architecture for video classification.

Reproduced from Simonyan and Zisserman

Both predictions can be combined with the final prediction.

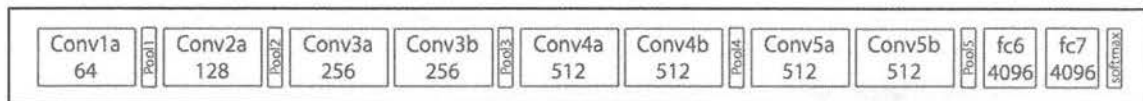
Using 3D convolution for temporal learning

A video can be classified with 3D convolution. 3D convolution operation takes a volume as input and outputs the same, whereas a 2D convolution can take a 2D or volume input and outputs a 2D image. The difference is shown as follows:



Reproduced from Tran et al.

The first two images belong to 2D convolution. The output is always an image. 3D convolution, meanwhile, outputs a volume. The difference is a convolution operation in 3 directions with the kernel. Tran et al. (https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Tran_Learning_Spatiotemporal_Features_ICCV_2015_paper.pdf) used 3D convolution for video classification. The 3D convolution model is shown as follows:



Reproduced from Tran et al.

The following is a code snippet of the model using 3D convolution:

```
net = tf.keras.models.Sequential()
net.add(tf.keras.layers.Conv3D(32,
                                kernel_size=(3, 3, 3),
                                input_shape=input_shape)))
net.add(tf.keras.layers.Activation('relu'))
net.add(tf.keras.layers.Conv3D(32, (3, 3, 3)))
net.add(tf.keras.layers.Activation('softmax'))
net.add(tf.keras.layers.MaxPooling3D())
net.add(tf.keras.layers.Dropout(0.25))

net.add(tf.keras.layers.Conv3D(64, (3, 3, 3)))
net.add(tf.keras.layers.Activation('relu'))
net.add(tf.keras.layers.Conv3D(64, (3, 3, 3)))
net.add(tf.keras.layers.Activation('softmax'))
net.add(tf.keras.layers.MaxPool3D())
```

```
net.add(tf.keras.layers.Dropout(0.25))

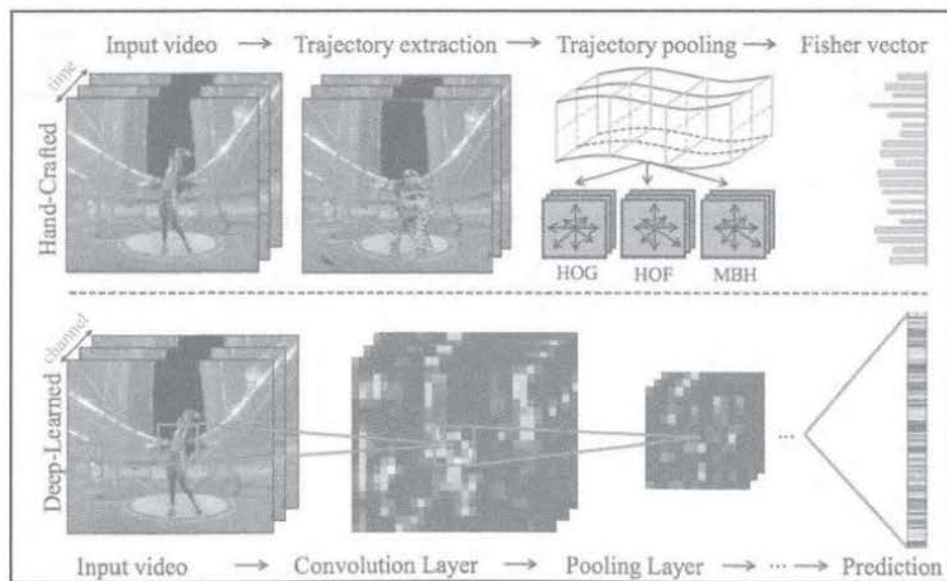
net.add(tf.keras.layers.Flatten())
net.add(tf.keras.layers.Dense(512, activation='sigmoid'))
net.add(tf.keras.layers.Dropout(0.5))
net.add(tf.keras.layers.Dense(no_classes, activation='softmax'))
net.compile(loss=tf.keras.losses.categorical_crossentropy,
            optimizer=tf.keras.optimizers.Adam(), metrics=['accuracy'])
```



3D convolution needs a lot of computing power. 3D convolution achieves an accuracy of 90.2% on the Sports1M dataset.

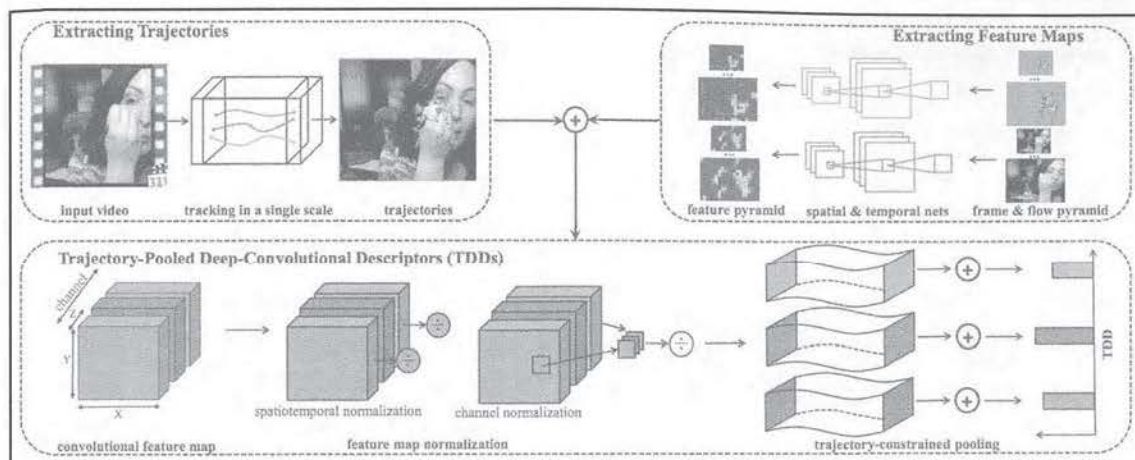
Using trajectory for classification

Wang et al. (https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Wang_Action_Recognition_With_2015_CVPR_paper.pdf) used the trajectory of parts of bodies to classify the actions performed. This work combines handcrafted and deep learned features for final predictions. The following is a representation of the classification:



Reproduced from Wang et al.

The handcrafted features are **Fisher vector** and the features are from CNN. The following image demonstrates the extraction of the trajectories and features maps:



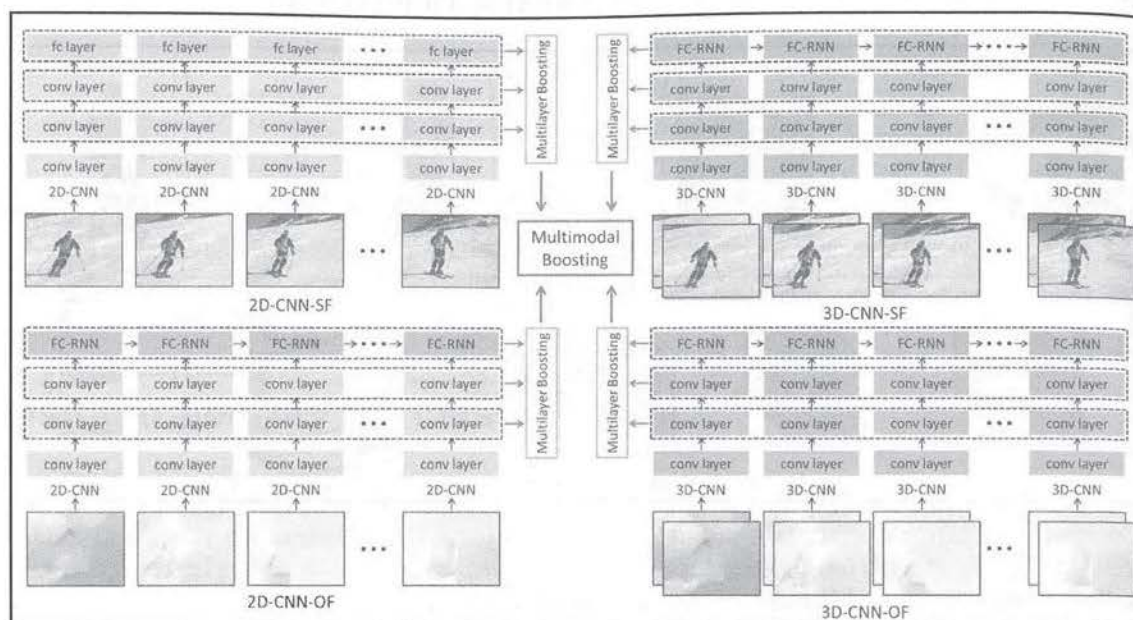
Reproduced from Wang et al.

Both the trajectories and features maps are combined temporally to form the final predictions over the temporal snippet.

Multi-modal fusion

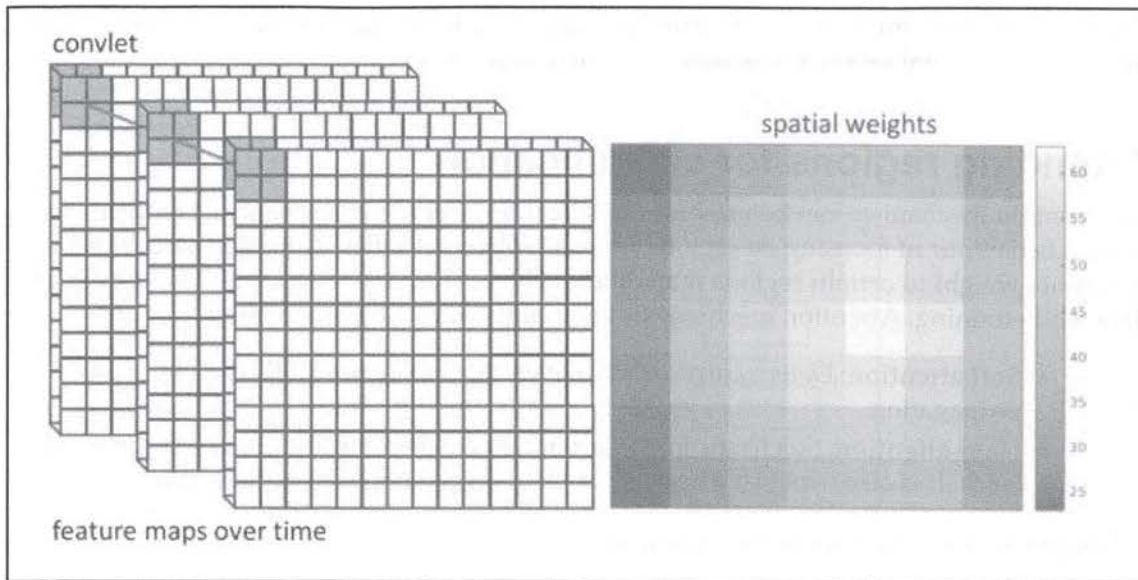
Yang et al. (http://research.nvidia.com/sites/default/files/pubs/2016-10_Multilayer-and-Multimodal/MM16.pdf) proposed a multi-modal fusion, with 4 models, for video classification. The four models are 3D convolution features, 2D optical flow, 3D optical flow, and 2D convolution features.

The flow of data in this method is shown as follows:



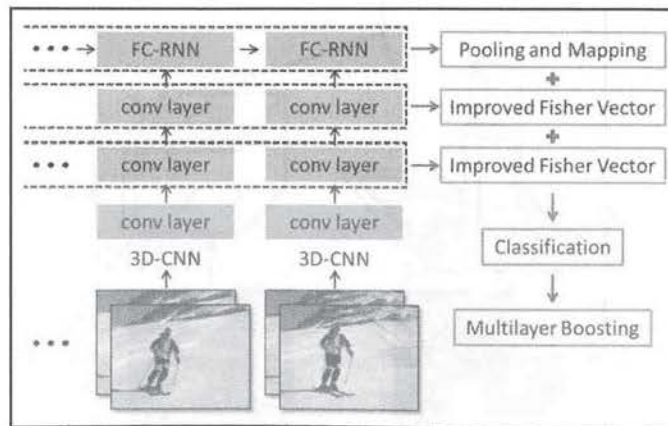
Reproduced from Yang et al.

Now, let's learn about **Convlet**. A **Convlet** is the small convolutional output from a single kernel. The learning of spatial weights in the convolution layer by **convlet** is shown in the following image:



Reproduced from Yang et al.

A spatial weight indicates how discriminative or important a local spatial region is in a convolutional layer. The following image is an illustration of fusing multi-layer representation, done at various layers of convolutional and fully connected layers:



Reproduced from Yang et al.

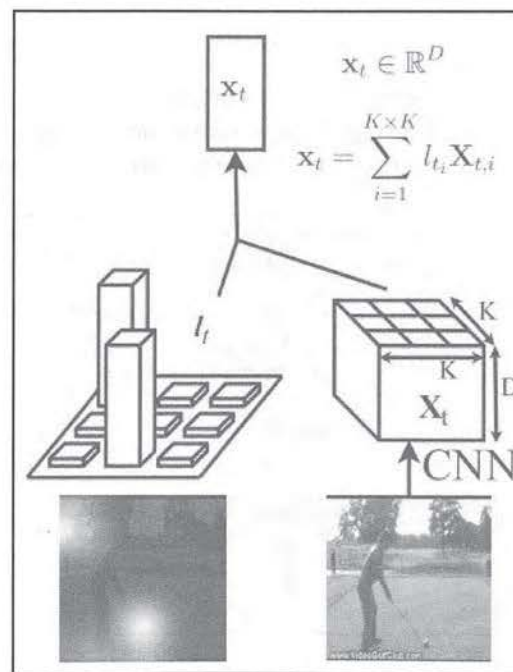
The boosting mechanism is used to combine the predictions. **Boosting** is a mechanism that can combine several model prediction into a final prediction.

Attending regions for classification

An attention mechanism can be used for classification. Attention mechanisms replicate the human behaviour of focusing on regions for recognition activities. Attention mechanisms give more weight to certain regions than others. The method of weight is learned from the data while training. Attention mechanisms are mainly of two types, namely:

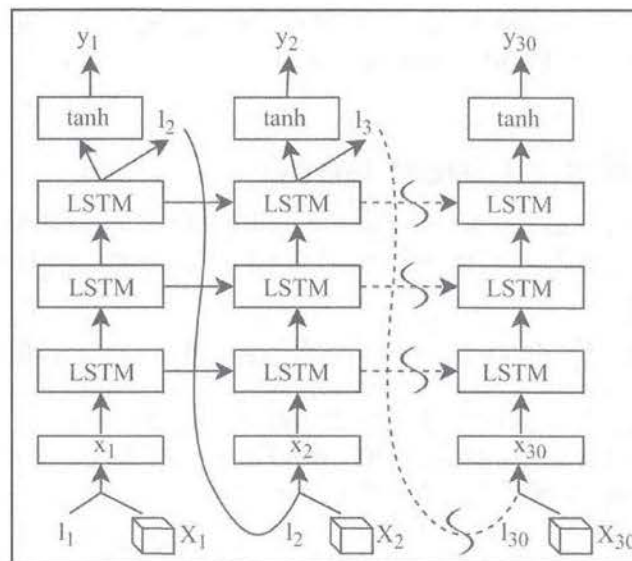
- **Soft attention:** Deterministic in character, this can hence be learned by back-propagation.
- **Hard attention:** Stochastic in character, this needs complicated mechanisms to learn. It is also expensive because of the requirement of sampling data.

Following is a visualization of soft attention:



Reproduced from Sharma et al.

The CNN features are computed and weighted according to the attention. The attention or weights given to certain areas can be used for visualization. Sharma et al. (<https://arxiv.org/pdf/1511.04119.pdf>) used this idea to classify the videos. LSTM were used to take the convolution features. The LSTM predicts the regions by using attention on following frames, as shown in the following image:



Reproduced from Sharma et al.

Each stack of LSTM predicts location and labels. Every stack has three LSTM. The input to the LSTM stack is a convolution feature cube and location. The location probabilities are the attention weights. The use of attention gives an improvement in accuracy as well as a method to visualize the predictions.

We have seen various approaches for video classification. Next, we will learn about other applications in videos.

Extending image-based approaches to videos

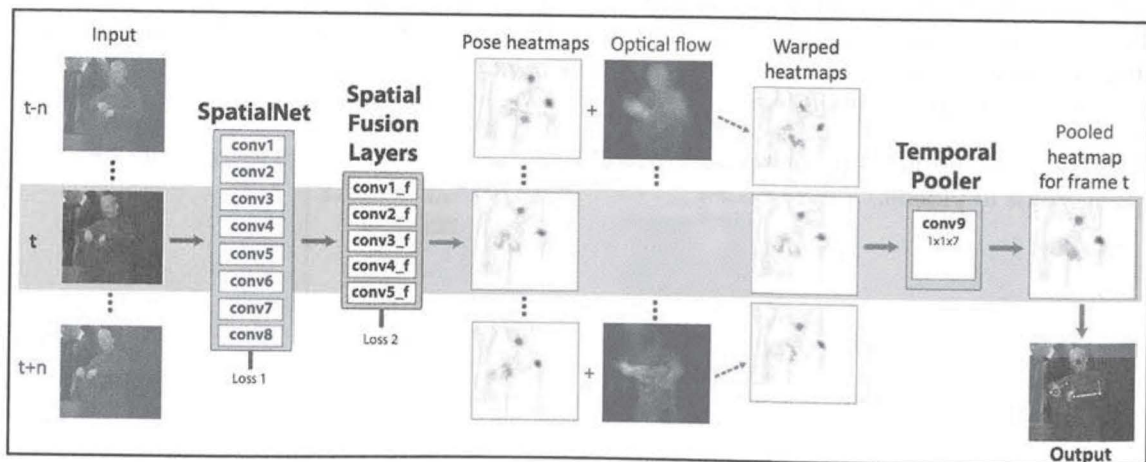
Images can be used for pose estimation, style transfer, image generation, segmentation, captioning, and so on. Similarly, these applications find a place in videos too. Using the temporal information may improve the predictions from images and vice versa. In this section, we will see how to extend these applications to videos.

Regressing the human pose

Human pose estimation is an important application of video data and can improve other tasks such as action recognition. First, let's see a description of the datasets available for pose estimation:

- **Poses in the wild dataset:** Contains 30 videos annotated with the human pose. The dataset link is: <https://lear.inrialpes.fr/research/posesinthewild/>. The dataset is annotated with human upper body joints.
- **Frames Labeled In Cinema (FLIC):** A human pose dataset obtained from 30 movies, available at: <https://bensapp.github.io/flic-dataset.html>.

Pfister et al. (https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Pfister_Flowing_ConvNets_for_ICCV_2015_paper.pdf) proposed a method to predict the human pose in videos. The following is the pipeline for regressing the human pose:



Reproduced from Pfister et al.

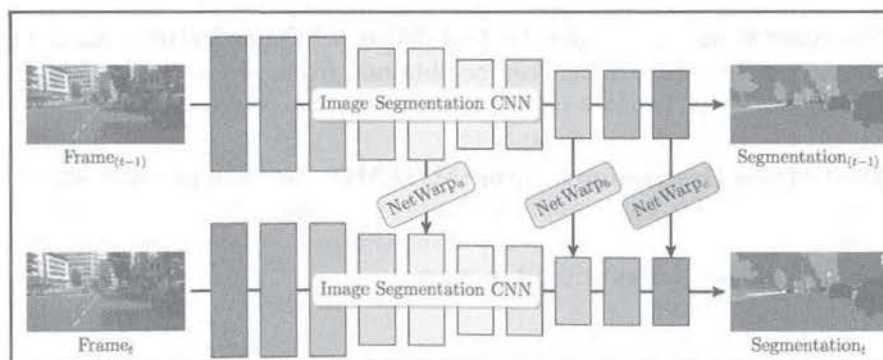
The frames from the video are taken and passed through a convolutional network. The layers are fused, and the pose heatmaps are obtained. The pose heatmaps are combined with optical flow to get the warped heatmaps. The warped heatmaps across a timeframe are pooled to produce the pooled heatmap, getting the final pose.

Tracking facial landmarks

Face analysis in videos requires face detection, landmark detection, pose estimation, verification, and so on. Computing landmarks are especially crucial for capturing facial animation, human-computer interaction, and human activity recognition. Instead of computing over frames, it can be computed over video. Gu et al. (http://research.nvidia.com/sites/default/files/pubs/2017-07_Dynamic-Facial-Analysis/rnnface.pdf) proposed a method to use a joint estimation of detection and tracking of facial landmarks in videos using RNN. The results outperform frame wise predictions and other previous models. The landmarks are computed by CNN, and the temporal aspect is encoded in an RNN. Synthetic data was used for training.

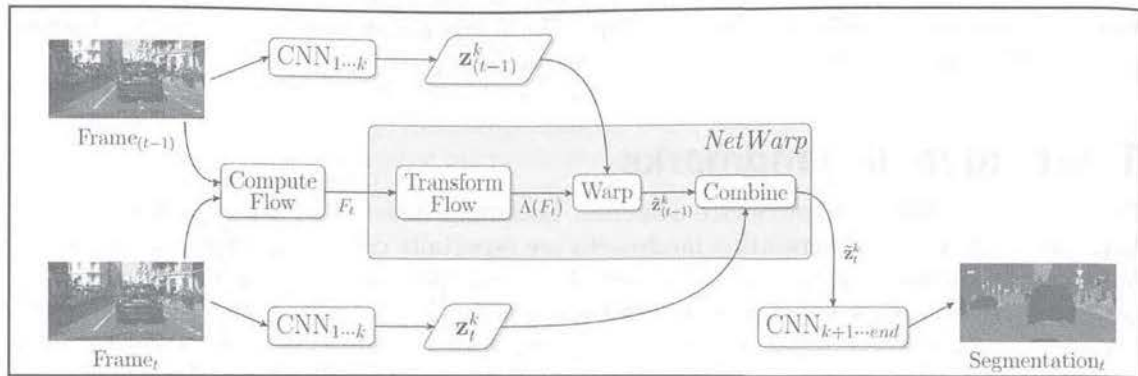
Segmenting videos

Videos can be segmented in a better way when temporal information is used. Gadde et al. (https://ps.is.tuebingen.mpg.de/uploads_file/attachment/attachment/386/gadde2017videocnns.pdf) proposed a method to combine temporal information by warping. The following image demonstrates the solution, which segments two frames and combines the warping:



Reproduced from Gadde et al.

The warping net is shown in the following image:



Reproduced from Gadde et al.

The optical flow is computed between two frames, which are combined with warping. The warping module takes the optical flow, transforms it, and combines it with the warped representations.

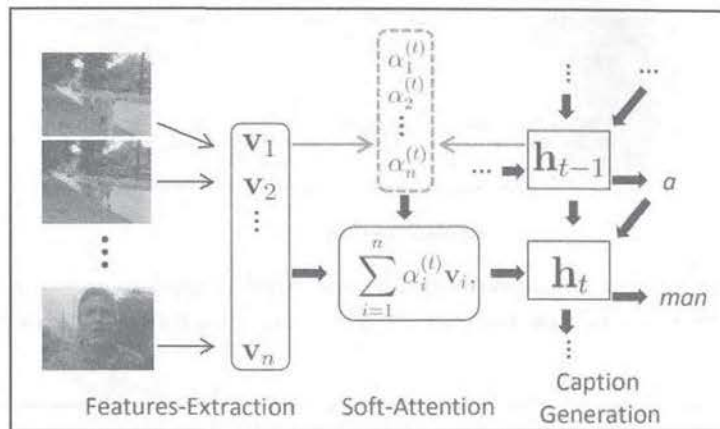
Captioning videos

The Chapter 7, *Image Captioning*, illustrated several ways to combine text and image. Similarly, captions can be generated for videos, describing the context. Let's see a list of the datasets available for captioning videos:

- **Microsoft Research - Video To Text (MSR-VTT)** has 200,000 video clip and sentence pairs. More details can be obtained from: <https://www.microsoft.com/en-us/research/publication/msr-vtt-a-large-video-description-dataset-for-bridging-video-and-language/>.
- **MPII Movie Description Corpus (MPII-MD)** can be obtained from: <https://www.mpi-inf.mpg.de/departments/computer-vision-and-multimodal-computing/research/vision-and-language/mpii-movie-description-dataset>. It has 68,000 sentences with 94 movies.

- **Montreal Video Annotation Dataset (M-VAD)** can be obtained from: <https://mila.quebec/en/publications/public-datasets/m-vad/> and has 49,000 clips.
- **YouTube2Text** has 1,970 videos with 80,000 descriptions.

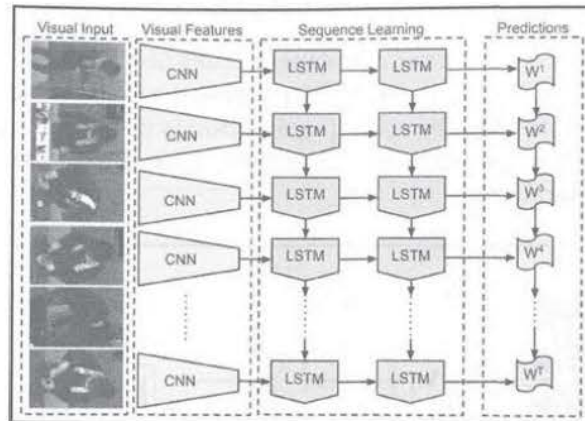
Yao et al. (https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Yao_Describing_Videos_by_ICCV_2015_paper.pdf) proposed a method for captioning videos. A 3D convolutional network trained for action recognition is used to extract the local temporal features. An attention mechanism is then used on the features to generate text using an RNN. The process is shown here:



Reproduced from Yao et al.

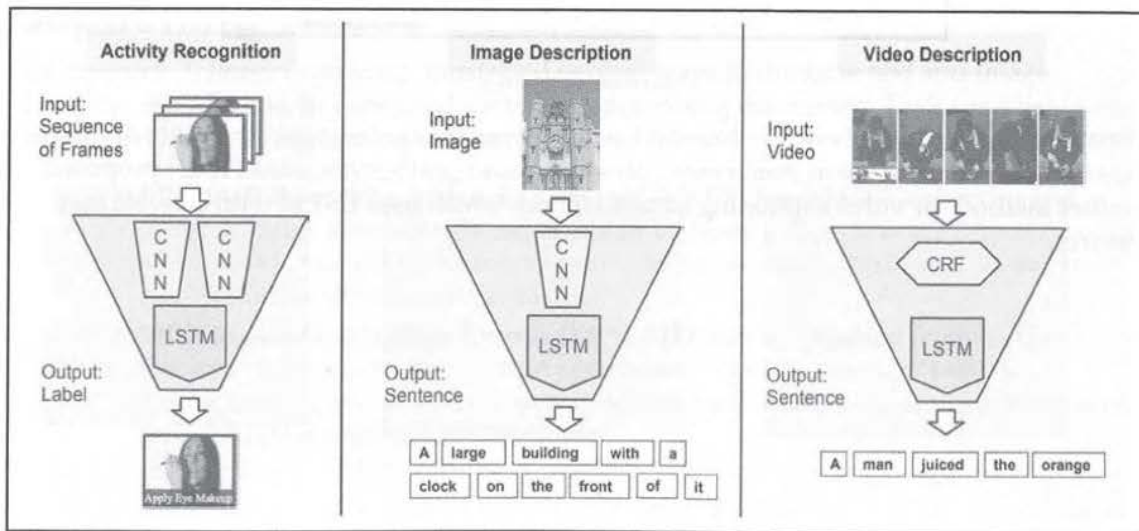
Donahue et al. (https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Donahue_Long-Term_Recurrent_Convolutional_2015_CVPR_paper.pdf) proposed another method for video captioning or description, which uses **LSTM** with convolution features.

This is similar to the preceding approach, except that we use 2D convolution features over here, as shown in the following image:



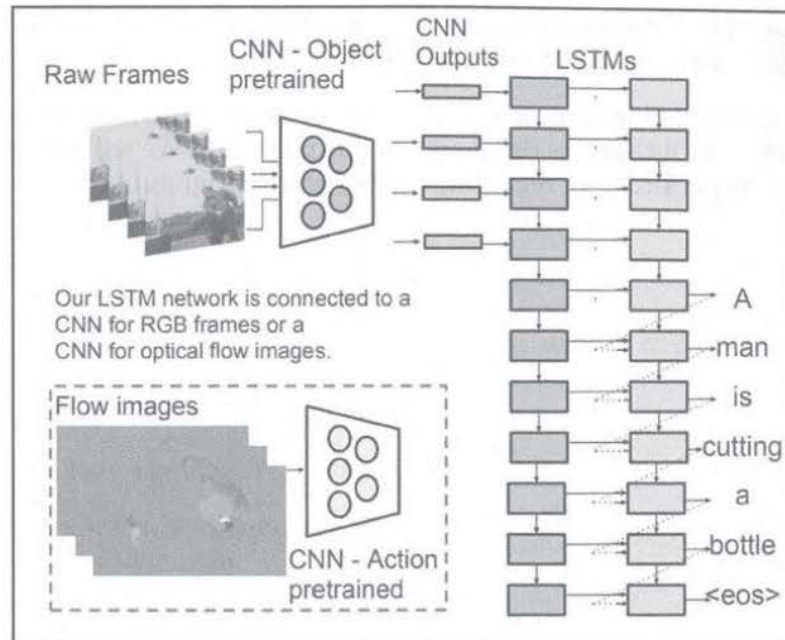
Reproduced from Donahue et al.

We have several ways to combine text with images, such as activity recognition, image description, and video description techniques. The following image illustrates these techniques:



Reproduced from Donahue et al.

Venugopalan et al. (https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Venugopalan_Sequence_to_Sequence_ICCV_2015_paper.pdf) proposed a method for video captioning using an encoder-decoder approach. The following is a visualization of the technique proposed by him:



Reproduced from Venugopalan et al.



The CNN can be computed on the frames or the optical flow of the images for this method.

Generating videos

Videos can be generated using generative models, in an unsupervised manner. The future frames can be predicted using the current frame. Ranzato et al. (<https://arxiv.org/pdf/1412.6604.pdf>) proposed a method for generating videos, inspired by language models. An RNN model is utilized to take a patch of the image and predict the next patch.

Summary

In this chapter, we covered various topics related to video classification. We saw how to split videos into frames and use the deep learning models that are in images for various tasks. We covered a few algorithms that are specific to video, such as tracking objects. We saw how to apply video-based solutions to various scenarios such as action recognition, gesture recognition, security applications, and intrusion detection.

In the next chapter, we will learn how to deploy the trained models from the previous chapter into production on various cloud and mobile platforms. We will see how different hardware affects the performance regarding latency and throughput.